**ASSIGNMENT OO**

**FIREFOX CODEBASE**

Mozilla is one of the largest open source software projects. The Mozilla codebase has millions of lines of code. Therefore, getting started in this huge project isn't easy. The intention of this document is to give an overview about what you should be aware of in order to hack Mozilla. It tries to build a bridge between the many different technologies used in the Mozilla project.

In order to have a proper understanding of the codebase one should be experienced with C/C++, as it is the main programming language used in the project.

As C++ and JavaScript are widely used in Mozilla, it makes sense to explain how JavaScript and C++ relate to each other in the Mozilla source code. C++ is a compiled language, while JavaScript is an interpreted language. JavaScript is most commonly known as a technology used to implement web sites. However, the developers of Mozilla decided that the Mozilla source code itself should consist of a mixture of both languages.

When you start the application, the C/C++ components start first. But in an early stage, a technology called XPConnect gets initialized that enables the use of interpreted JavaScript at runtime. In fact, a Mozilla browser distribution consists of both compiled C++ and uncompiled JS files.

Note that JavaScript cannot be compiled to be executed by the operating system directly, we still use C/C++ for the back-end of the program, JavaScript runs inside Mozilla. Also note, when surfing web pages which use JavaScript, that JavaScript code is executed within a sand box, and does not have access to Mozilla's internal objects. Only those objects that are exposed by DOM (Document Object Model) are accessible.

A primary requirement for Software developed in the Mozilla project is that it must be cross-platform, i.e. it must not be restricted to any particular operating system.

**C++ and JavaScript**

While C++ is intended to be a portable programming language, this portability aspect applies only to general program logic and data structures. If someone wants to write software for a particular operating system, they need to use functionality that is specific to that system. Often, one might want the same functionality on all systems, but in order to do it, one would need to write specific software for each platform.

**NSPR**

The intention of [NSPR](http://www.mozilla.org/projects/nspr/) (Netscape portable runtime) is to provide a layer between the OS and the Mozilla source code, to allow for simpler coding in other areas of the Mozilla source code.

Mozilla is a multithreaded application. NSPR provides an operating system independent facility to program with multiple threads. For example, all network data transfer happens on a separate thread, to make sure the user interface stays responsive while data is being transferred. One requirement for C++ code a person writes is that it is safe for multiple threads.

**Object oriented programming & Modularity**

Mozilla C++ source code is intended to follow the rules of OOP, that includes building modular components, where access to internal data (variables) is only allowed/possible using the public interfaces of your classes.

In most simple C++ projects, this only means, you carefully design your classes to use public/protected/private as appropriate, but all source code is still available everywhere. For instance, at any time, you can change any class' component from private to public, so it will be available at any other place in your project. This does not apply to Mozilla. It was decided that Mozilla should be even more modular.

The Mozilla source code is organized as separate components. While within one component, you have all the freedom as described in the previous paragraph for simple projects, you don't have the same level of flexibility between components.

When components talk to each other, they only do so using well defined interfaces using the component object model (COM).

**Interfaces**

The concept of interfaces is something that is used in the CORBA technology, for example. Both CORBA and Mozilla use a similar language to describe interfaces, which is XPIDL (IDL means Interface Definition Language).

In a CORBA environment, life is more restrictive and difficult, because you have inter-process and inter-network communication, something which Mozilla is not actively using. In a distributed CORBA environment, it is difficult to change the components of an interface, because you are usually unable to replace all running systems at the same time. If you want to change something, you have to define a new version of an interface, but you might still be required to support the old one.

As Mozilla is not a distributed application as of writing, it is currently possible to change most interfaces as the development process requires it. But because the Mozilla browser runs embedded in some environments, those environments must be able to rely on a fixed interface, therefore interfaces can be frozen. This state is usually indicated in the interface definition. As Mozilla stabilizes over time, or approaches the magical version number One-Dot-Zero, the ratio of frozen to not frozen interfaces is likely to increase.

One step of building Mozilla is automatically translating the interface definition files into C/C++ header files. That's the job of Mozilla's own IDL compiler, xpidl.

Besides of the methods and data members, interfaces have additional attributes. They have a UUID, a number to uniquely identify an interface. Interfaces can have the scriptable attribute, which means they will be accessible from the JavaScript code. A scriptable interface is restricted to only use data types for parameters that are valid within the JavaScript runtime.

## C++ and JavaScript

Code execution can fail at runtime. One programming mechanism to deal with failure is to use exceptions. While Mozilla uses Exceptions in its JavaScript code portions, it does not in C++. One out of several reasons for this is exceptions haven't always been portable, so what was done in the past has stuck. Mozilla C++ code uses return codes to simulate exceptions. That means, while in JavaScript you can use try-catch blocks, in C++ you should check the return code whenever you call an interface method. That return code is of type nsresult. For this reason, the logical return type, as defined in the IDL file, is mapped to an additional method parameter in the C++ code.

The nsresult type is intended to transport additional failure reasons. Instead of simply reporting failure or success, an integer type is used instead, to allow for defining a lot of different error codes.

There are some general result codes, like NS\_OK, which is used to indicate that everything went well and program flow can continue, or NS\_ERROR\_FAILURE, which is used if something went wrong, but no more details need to be provided as of yet.

In addition to that, each component can request its own range of integers to define error codes that will not overlap with those failure codes used in other areas of an application. Look at mozilla/xpcom/base/nsError.h for more information.

**Strings in C++**

While many application frameworks or class libraries decided to use just a single string class, the Mozilla developers have decided they need something more powerful. They have implemented a hierarchy of several string classes, which allows the dynamic runtime behaviour to be optimized for different situations. Sometimes you just need to fixed size string, sometimes you need a large string that grows over time. Therefore, for example, not only flat strings, but also segmented string types are available.

An additional requirement is that Mozilla has to be fully multi-language. All strings that deal with information shown to a user are therefore using multi-byte Unicode character strings.

The string types are template based, with the character type as the variable type, to allow the same logic to be used with regular and Unicode strings.

While that approach of having many string classes means a lot of flexibility, the drawback is that learning Mozilla's string classes is not trivial.

## Graphical User Interface / XUL

Most operating systems define their own way to develop graphical user interfaces, and they are mostly different.

For a cross-platform application like Mozilla it is crucial to have a set of technologies that hide the operating system dependent logic from the application logic.

In the past a lot of C/C++ libraries have been coded that were cross-platform. To my knowledge, none of them are used in Mozilla, yet we have created our own graphics system.

When defining the layout of a GUI (graphical user interface), you can choose to go with either of two possibilities. You could define the absolute positions of each UI (user interface) element that you want to appear. This approach actually has been chosen by a lot of GUI libraries. But it has some drawbacks - you are not very flexible when the layout changes when adding more elements, because you have to rearrange all elements to new positions. You also have to do that graphically, to get immediate feedback which elements overlap, etc. But still, the UI might not look as intended when a different font with different metrics has to be used - this can make a UI unusable.

Mozilla developers wanted to have something very flexible. As Mozilla is cross-platform, it has to be very flexible with regards to fonts.

Mozilla developers have chosen to use an approach where the contents of the UI are designed in a logical matter. We don't currently use an UI editor. We write a file with instructions how the UI should look. At runtime, the layout engine decides which fonts are available, and considers all the requests that have been defined in the UI description, and creates the actual UI dynamically. This is similar to what the web browser does when displaying web pages.

The web has gone from a mostly text-based system to a very graphical rich environment that has user interfaces akin to many programs. Therefore, it was only natural for a web browser to use web languages in order to define its user interface. It was decided to use an XML based syntax to write the UI description, which has been called XUL (extensible user-interface language). (A good reference for XUL is available at [XULPlanet](http://www.xulplanet.com/)).

A XUL file describes what elements the UI consists of, and where elements appear. The XUL language defines attributes that allow the programmer to define the actions controls react to. To define the dynamic behaviour of the application, one can define JavaScript functions that will be called when certain user interface events happen. Within those JavaScript functions, you can either do the required application behaviour directly, or call any other application logic available by COM objects, including logic defined in C++.

In addition to the logical representation of the UI, people also prefer to have a pretty looking UI. To define the detailed characteristics of the UI, one uses CSS files, that defines, for example, which images will be used to display certain UI elements. This makes it flexible to define additional "looks" for the application, which are referred to as "themes" or "skins". Mozilla defines currently two themes, classic and modern, which are actively maintained by the Mozilla developers. While there exists additional themes for Mozilla, they often exist only for certain versions of Mozilla. It is a lot of work for a theme designer to stay in sync with all the changes that happen to the UI each day.