Architecture Enhancement of Firefox

December 7, 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

Abstract

In this report, we examine the implementation details of the Core App Services layer of Mozilla Firefox. This layer contains all platform specific components implemented as XPCOM components. These components interact to allow for the retrieval, processing and displaying of information including primary content such as webpages.

The current state of the architecture of this layer contains very tightly coupled components with low cohesion. This shows poor design, especially since this is part of the Mozilla Application Framework (which is used in numerous applications). Since Firefox is an open source project, it is the collaborative effort of numerous developers, which tends to allow design decisions to be lost in implementation.

In this project, we will be comparing two different ways of cleaning up the Core App Services layer. Both methods include implementation of one or more design patterns in an attempt to allow each component to have its own pluggable interface as part of the larger framework. The goal is to abstract out the implementation of the components, while still allowing for the same functionality.

In the interest of achieving more interfaces that are pluggable, we will examine the bridge pattern. In particular, we will examine how it allows for hiding implementation details, allowing for a central point of access to and from the layer. We will also look at the façade pattern, specifically how it can be used to provide interfacing between components. This separation can be used along with the factory pattern to separate the implementation using abstraction for each component.

Introduction and Overview

The purpose of this project is to propose a particular feature or enhancement to Firefox that is not currently available or being worked on. Our proposed enhancement was to clean up the Core App Services layer of Firefox by eliminating interdependencies and removing communication between non-adjacent layers, in this case between the UI and Core App Services layers. This can be achieved with additional components that act as an adapter to the entire layer, allowing components both within the layer and outside the layer to communicate solely with this component.

This enhancement, if implemented correctly, will eliminate interdependencies and greatly reduce the barrier to entry for developers interested in contributing. The elimination of interdependencies will also allow for easier testing, more maintainable and extendible architecture while simultaneously reducing effort required to port the application to other platforms.

The enhancement would introduce a slight performance degradation, though this is a justifiable trade-off when considering the gains in ease of developing, porting and debugging with the new architecture.

Enhancement

The enhancement proposed in this project was the cleanup of the Core App Services layer to allow for easier maintenance and extendibility. This will primarily be achieved through the removal of interdependencies within Core App Services. A desirable outcome of this is the unification of access to Core App Services from the DOM layer to reduce direct accessing of individual components.

Motivation

To assess the motivation of the proposed enhancement, it is important to consider the impact of any changes upon the Mozilla App Framework in context. Firefox, as with all Mozilla Foundation's applications, is based off the Mozilla App Framework that powers numerous open-source applications including many industry standard applications such as Firefox and Thunderbird. Any changes to these will then affect an entire suite of applications managed by separate teams.

It is also worth noting that due to the open-source nature of the application, many contributions are made through developers who are not employed or part of the Mozilla Foundation. These developers contribute to parts that they see can be improved or fix, and dedicate their time as they see fit. This results in many features or fixes being worked on by different developers or teams at different points in time, and some being the collaborative effort of many different developers.

This in mind, certain priorities for the architecture of Core App Services, as well as the Application Framework as a whole, become clearer. Reasonable performance penalties become more justifiable in exchange for improved ability for developers to contribute without overall knowledge of the architecture, or the intricacies of components not related to their contribution.

This would allow for a much lower barrier-to-entry for those developers interested in contributing. This is very important to open-source applications considering these developers are the driving force behind the application. Allowing more developers to participate as well as allowing products to be more easily passed between teams result in a much higher product quality due to quicker development and streamlined debugging.

Implementation Possibilities

This may be accomplished by the creation of a class that acts as an interface and allows abstraction to hide particular implementations. This concept is very similar to the bridge pattern. The bridge design pattern allows implementation to be hidden from the client (in this case, the DOM or UI layer) as well as isolate components from each other.

This enhancement can also be implemented by utilizing the façade pattern to provide an interface, and the factory pattern to separate the implementation from the abstraction. Keeping in mind the motivation behind the factory pattern is to allow for abstraction of the types of runtime object during creation, this may not be the preferred

implementation. In this application, it is not likely that creation of the components would need to be abstracted, thus, the factory pattern is not as good a fit as the bridge pattern since it does much more than just separate interface from code.

This feature was selected in order to make repairs to the underlying architecture of the application. The interconnectedness of the Core App Services layer implies that dependencies were made in order to write code that simply leveraged existing code. Without periodic repair, this type of programming style can lead to a very complex dependency graph, as can be seen currently in the Core App Services, Figure 1.

Current State

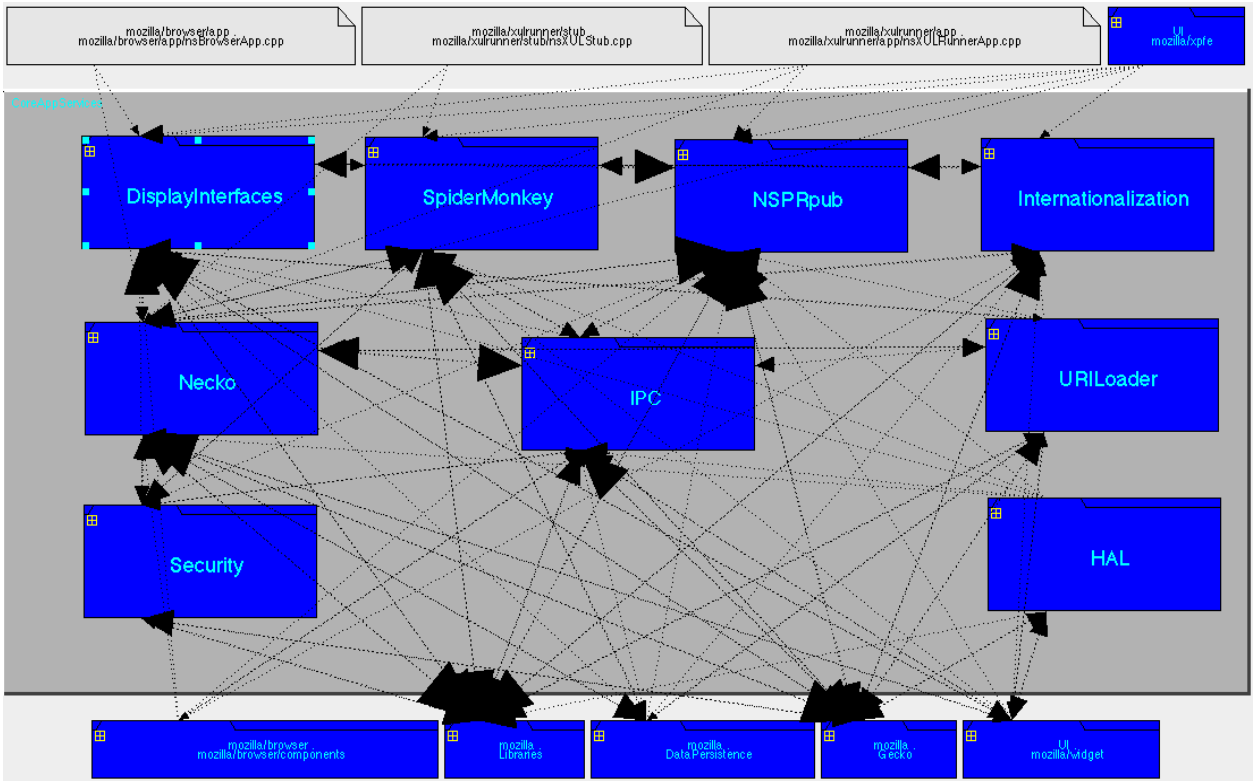

Figure 1 - Interdependencies of the Core App Services layer

Currently in Firefox, the Core App Services is a tightly coupled system with low cohesion. Components communicate amongst themselves, and some components are so tightly coupled there is not a clear boundary separating the two. An example of this is shown in Figure 1; many components communicate with Necko, as it is the sole provider of a networking interface. This creates a very strong dependency between Necko and the rest of the Core App Services layer. This increases the complexity of the layer and increases the likelihood of serious bugs or security issues (Perrin).

In some situations, the apparent inter-communication is due to the importation of library files to another component. In the example of Necko, the network data types are defined inside the Necko component. This works against the concept of the Core App Services being modular. However, it is unlikely that this dependency actually relies on

the implementation details of the Necko component; this data type definition (in Figure 2) could be defined elsewhere (such as in a Library) effectively removing this dependency.

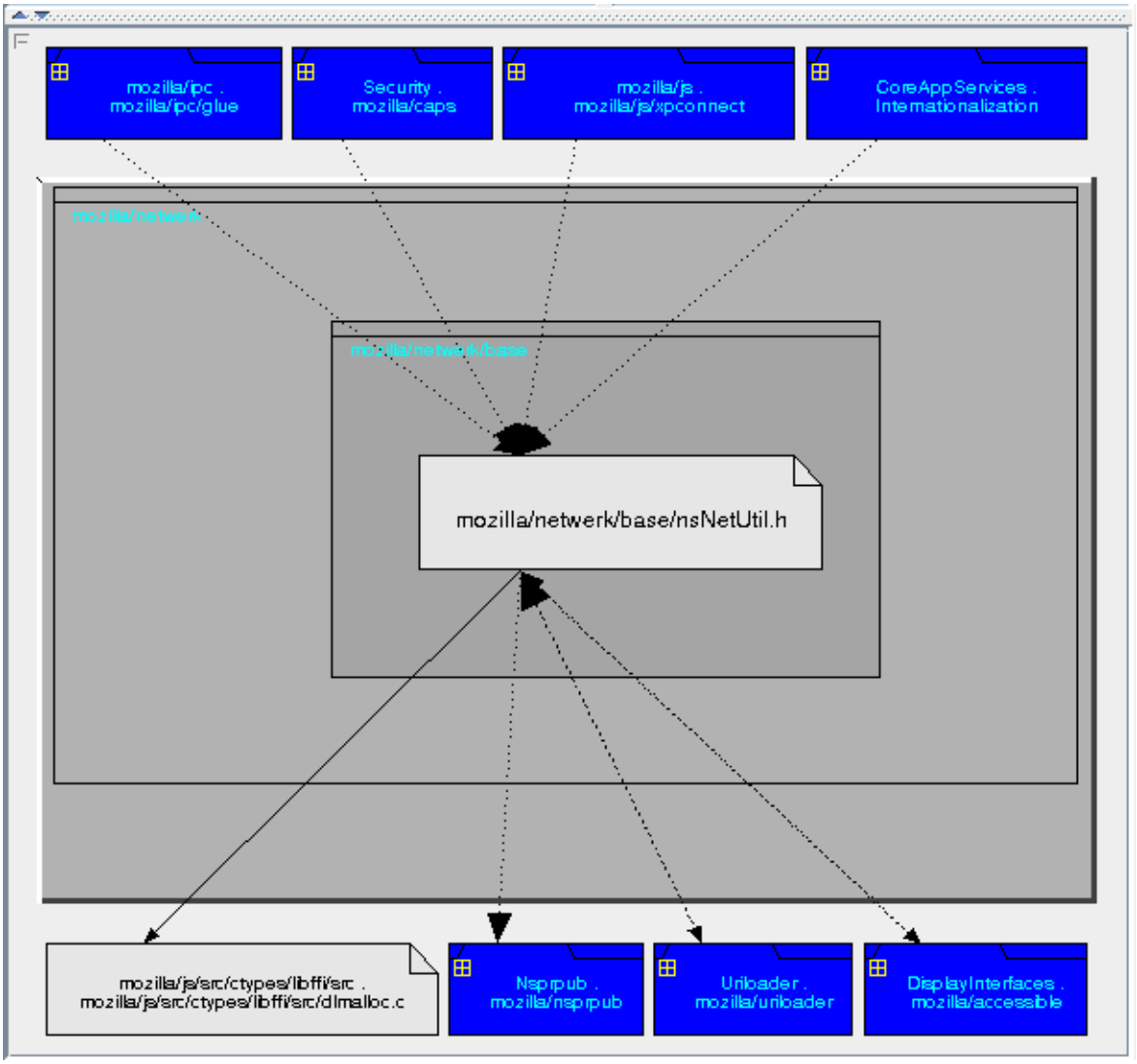

Figure 2 - Dependencies of the nsNetUtil.h class in Necko

In other situations, the communication seems more deliberate, such as calls made to the IPC component, specifically the `URIUtils.cpp` class, for its serializing service. The Necko component, for example, makes 42 calls to this class for serializing or de-serializing cookies (through its `CookieServiceChild` class), HTTP channels (through its `HttpChannelChild` class), FTP channels (through its `FTPChannelChild` class) and other protocols. This communication cannot easily be rectified, and would require additional planning to resolve.

The UI layer communicates directly with the Core App Services layer in order to perform memory management. For example, the `nsBidiKeyboard` class in the widgets component of the UI layer communicates with `prmemc` class in the NSPRpub component

for allocating a buffer to hold a list, and then makes another call to fill the buffer with keys from the keyboard.

Impacts of Change

This architectural repair would involve serious restructuring of the Core App Services layer. In most components, type definitions would have to be refactored such that they now implement an abstract class, which is defined to provide a unified access point for the DOM layer. One example of this change is in the DOM class, the header file for `nsIDocument` interacts with `nsOfflineCacheUpdate`. After this change, the DOM class will depend on a bridge class, rather than directly with the URILoader component.

Enhancement Process

To explain how our change will affect the system we will look at the different design patterns we expect to implement for our feature change. As mentioned above, we suggest two ways of implementing this. The first possibility includes the utilization of the bridge pattern whereas the second possibility makes use of the façade pattern and the factory method pattern. In this section, we will attempt to explain at an abstract level how each pattern will be used as well as the structure and participants of each. By the end of this section, a selection will be made about which scenario seems the best fit for this solution.

Utilization of the Bridge Pattern

The bridge pattern attempts to separate the implementation from the specification of both the concrete and abstract classes. See Figure 3 for the bridge pattern structure. In the bridge pattern, the participants are as follows:

- **Client** - any class which uses the implementation of the specific bridge pattern
- **Abstraction** - class that defines the interface which the client sees
- **RefinedAbstraction** – class that further refines the Abstraction interface
- **Implementor** – class that further refines the Abstraction class
- **ConcreteImplementor** – this class defines the operations in the **Implementor** class as well as its own.

In our implementation, there would be many Clients as these would include most, if not all, Core App Services components as well as a few from the DOM layer. The Abstraction class in our enhancement would be an interface to Core App Services. Implementor(s) would consist of components within the Core App Services Layer. Each Implementor would have one or more ConcreteImplementor(s), each a platform independent concrete implementation of the specific component.

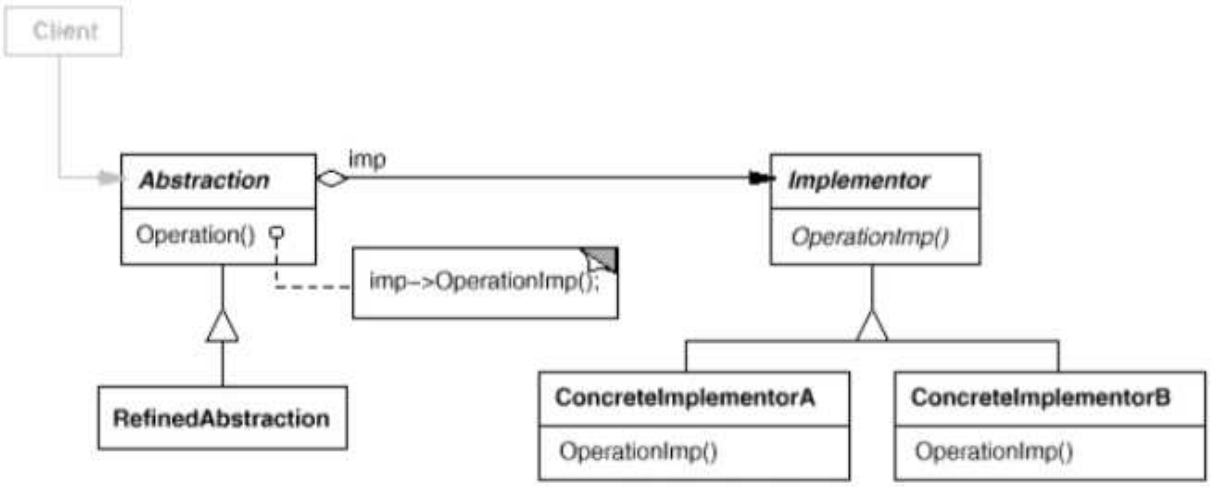

Figure 3 - Bridge Pattern

This solution provides a general access point for Core App Services components as well as other layers in the framework.

Alternative Façade and Factory Method Pattern

An alternative solution would employ the factory method pattern for handling the creation of platform specific components and the façade pattern for controlling the communication between these components.

The Façade pattern would be used to provide a unified interface to a set of interfaces to make the sub systems appear as a single unified unit, abstracting implementation details from the client. The participants are then the façade itself as well as the subcomponents. The façade receives external requests and forwards these requests to the appropriate subcomponent or subsystem without the need for subcomponents to have knowledge of the façade. See

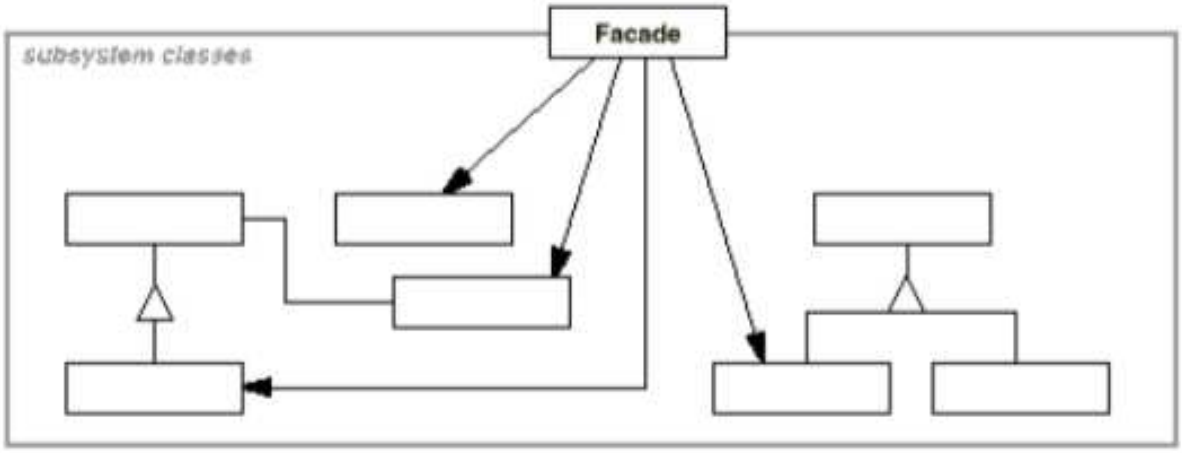

Figure 4 for the structure of the façade pattern.

In the context of Firefox, a new ‘Core App Services’ component acts as the façade, forwarding tasks which come from the client (DOM layer) to the corresponding subcomponents. These Subcomponents are the various components within the Core App Services layer.

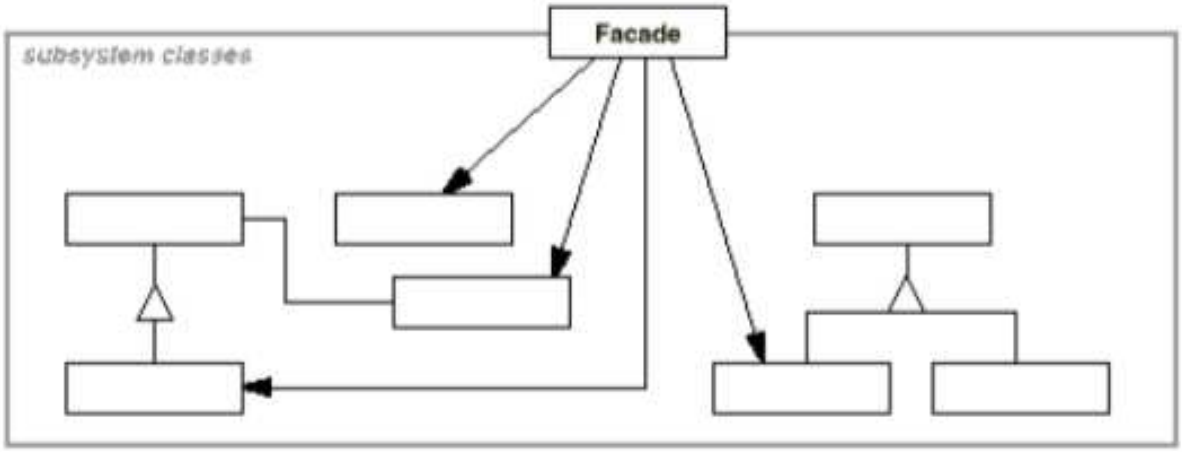

Figure 4 - Façade Pattern

The factory method pattern defines an interface for creating an object, but defers the instantiation of the class to subclasses. The participants in the factory method pattern are

- **Product** - defines the objects that are to be made.
- **ConcreteProduct** - implements the **Product** interface.
- **Creator** defines the **FactoryMethod**
- **ConcreteCreator** returns a new **ConcreteProduct**.

See Figure 5 for the structure of the factory method pattern. In this implementation, a factory method exists for each of the platform specific components. These components are not then fully realized until runtime when they are created for the platform on which it is run.

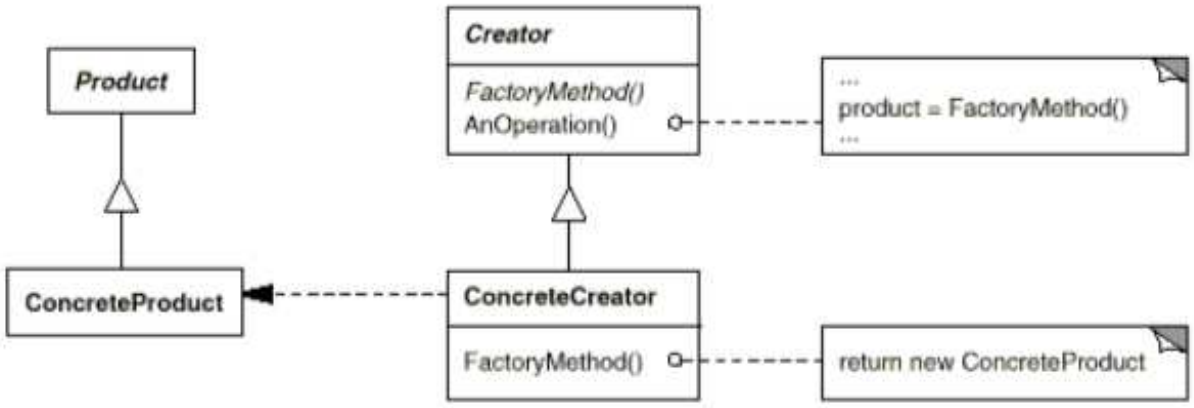

Figure 5 - Factory Method

Both these solutions create an interface for communication from other layers to the Core App Services layer, as well as for communication between the various Core App Services components. The second solution differentiates itself in its ability to defer the creation of platform specific implementations until runtime.

Implementation Decision

The choice of the Bridge Pattern stems from the easy point of access from both sides of the Abstraction class. Both the DOM layer and the components within the Core App Services layer can utilize the Abstraction class as clients, thus creating a two-way API. This contrasts to the alternative solution where subcomponents do not keep track of the façade.

Results of Enhancement

Architectural Change

Changes at the high and low levels remain unchanged with this modification. At the high level, the Mozilla Firefox's Architecture – or more accurately the Mozilla Application Framework's Architecture – is a three-tiered layered architecture. At the low level, the Core App Services retains its Object Oriented architecture.

This modification then does not modify any existing architectures, rather it makes use of low-level design patterns to bring the concrete implementation of the overall architecture more in line with the conceptual architecture.

Extendibility/Evolvability

The proposed enhancement goes quite a way in allowing greater extendibility. With proper implementation, new developers do not need to dive through the code or architecture to find the locations of certain functionality. Rather, all functionality offered by the Core App Services layer will be abstracted through the newly created component and leave implementation of the functionality to components. This means small ideas or fixes can easily be added, greatly reducing the barrier-to-entry for new developers who do not wish for a long-term commitment to contribution.

It is also improved in a sense that new components can be easily plugged in without having to resolve dependencies across a multitude of files; rather the application will merely have to be plugged directly into the abstraction component.

For example, a developer looking to add a new component that requires network functionality, as well as security related functionality would not have to go digging through Necko, Security or the platform-specific implementations. Rather, they would merely have to interface with the abstraction component that would act as an interface to the relevant functionality.

Maintainability and Testing

With the new architecture, the source of bugs will be easier to locate and greatly streamline the debugging process. This is due to the much lower surface area for failure

points as any functionality can be tested on both sides of the new component to ensure that the functionality is implemented correctly.

Previously testing would have consisted of searching through the data path as it crossed through multiple components to determine the cause of failure. An example of this can be considered through the standard process of loading a webpage. With the current implementation a webpage loading incorrectly could be a result of a failure from anywhere between the URILoader's resolution of the webpage address, Necko's parsing of the return data, or even the Security model's functions should this have happened over a secure connection.

Performance of Firefox

Relative to the current implementation of Core App Services, we expect the introduction of the Bridge Pattern to have a negative impact on performance. This is due to the layer remaining functionally the same with added function calls due to the abstraction. In testing, we found this to be a relatively small performance penalty of about 1-2% increase in execution time relative to direct access.

Note, however, that this is considering the case of a single abstracted interface, whereas many operations in Firefox will make use of numerous components, which may even scale up to a 4-5% increase in execution time.

Conclusion

As long as Firefox depends on volunteers, it needs to make sure it is easily read and understood by volunteers. An easy to maintain and test architecture will attract more volunteers than the opposite. Our proposed enhancement, terminating interdependencies and stopping non-adjacent layers from communicating to one another, allows for that.

Our feature can be implemented by making use of the factory method pattern and the façade pattern. The factory method pattern allows each component to be its own object inheriting from an interface while the façade pattern handles delegating tasks coming from the DOM layer and communication between each factory object.

A better way to implement our feature is by implementing the bridge pattern. This allows every component to have a concrete definition by making each component interface its abstraction class. The bridge pattern attempts to separate the implementation from the specification of both, the concrete and abstract classes. This method allows each component to have its own point of access while eliminating interdependencies. This solution provides a general access point for each component as well as other layers in the framework. While this solution does introduce a need for both an interface and concrete implementation for each pluggable component, it cleans up the design of the Core App Services Layer and places a single point of entry for the layer.

These solutions create an API for communication from other layers to the Core App Services layer and reduce the entry barrier for other developers. It greatly reduces the complexity, which, in turn, reduces the possibility of bugs and allows for easier testing. It makes Firefox a more maintainable project that can be easily improved.

Lessons Learned

The tools used for Assignment 2 as well as Assignment 3 leveraged a base relation that was exported from Understand. In at least one instance, it seems that the data from Understand was mistaken in the relation generated. The relation in question is between `transportlayer.cpp` inside media and `nsOfflineCacheUpdate.h` inside `URILoader`. Using the Linux commands `sed` and `grep`, the common code appears to be due to both files having a method titled "`InitInternal()`". Upon closer inspection, the two are unrelated. No other common words have been found.

```
#include "logging.h"
#include "transportflow.h"
#include "transportlayer.h"
#include "nsThreadUtils.h"

// Logging context
namespace mozilla {

MOZ_MTLOG_MODULE("mtransport")

nsresult TransportLayer::Init() {
  if (state_ != TS_NONE)
    return state_ == TS_ERROR ? NS_ERROR_FAILURE : NS_OK;

  nsresult rv = InitInternal();

  if (!NS_SUCCEEDED(rv)) {
    state_ = TS_ERROR;
    return rv;
  }
  state_ = TS_INIT;

  return NS_OK;
}
```

Figure 6 - View of Common Words between nsOfflineCacheUpdate.h and trasnportlayer.cpp

Appendix A: Affected Directories and Files

The list of directories and files modified by the proposed enhancement is as follows:

Components in Core App Services	External
• mozilla/ipc • mozilla/nsprpub • mozilla/uriloader • mozilla/hal • mozilla/netwerk • mozilla/security • mozilla/services • mozilla/accessible • mozilla/caps • mozilla/intl	• mozilla/browser/base/content • mozilla/browser/components/ • mozilla/browser/devtools • mozilla/browser/extensions/shumway/content • mozilla/browser/extensions/pdfjs/content/web • mozilla/browser/metro/base/content • mozilla/browser/metro/base/tests/mochitest • mozilla/browser/modules/test • mozilla/widget/windows • mozilla/widget/nslWidget.h • mozilla/widget/android • mozilla/widget/crashtests • mozilla/widget/gonk • mozilla/widget/cocoa • mozilla/widget/qt • mozilla/widget/windows • mozilla/widget/gtk/compat/gdk • mozilla/xpfe/appshell • mozilla/docshell/base • mozilla/docshell/shistory/src • mozilla/docshell/build • mozilla/dom • mozilla/chrome • mozilla/extensions/auth • mozilla/editor/libeditor • mozilla/embedding • mozilla/gfx • mozilla/image/decoders/icon/gtk • mozilla/image/src • mozilla/image/test/unit • mozilla/layout • mozilla/media/libvpx/vpx_mem • mozilla/media/libyuv/include/libyuv • mozilla/media/mtransport/third_party/nrappkit/src/util/libekr • mozilla/media/omx-plugin/include/gb • mozilla/media/libstagefright/system/core • mozilla/media/webrtc • mozilla/parser • mozilla/rdf/base • mozilla/rdf/tests/triplescat • mozilla/tools/jprof • mozilla/tools/profiler • mozilla/toolkit

Bibliography

Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1994.

Perrin, Chad. *The Danger of Complexity: More Code, More Bugs*. 1 February 2010. Tech Republic. 2 January 2015. <http://www.techrepublic.com/blog/it-security/the-danger-of-complexity-more-code-more-bugs/.>.