# Object Oriented Framework design in JavaScript: A Pattern Oriented Approach

Mayank Sinha

This dissertation was submitted in part fulfilment of requirements for the degree of MSc Advanced Computer Science

Department of Computer and Information Sciences

University of Strathclyde

August 2017

# Declaration

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself. Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive.

Yes [ ] No [ ]

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices is .

I confirm that I wish this to be assessed as a

Type 1 2 3 4 5 dissertation (please circle)

Signature:

Date:

## Abstract

Design Patterns are essentially a structured approach to Object Oriented Programming. They are the body of knowledge acquired from decades of collective experience of the programming community detailing a "general reusable solutions to commonly occurring problems" in object oriented application design. Patterns are not an exact solution, they must be adapted to the design problems they intend to solve.

Patterns are intended to be language agnostic, but are almost always described and discussed in terms of statically typed languages and languages that support object oriented programming constructs. This dissertation will explore the application of object oriented design patterns and practices to web technologies, by prototyping a “Single Page Application” framework in JavaScript, a loosely typed, event driven programming language.

# Contents

# List of figures



# Code Listings

# Introduction

Object oriented programming emerged as way to achieve code reuse at a coarser granularity than subroutines. Design patterns then emerged as a formalized discussion of best practices in object oriented design, shifting focus from class or object level to module level reuse. Patterns are an abstract concept, they do not by themselves prescribe an actual implementation that can be reused. Before the appearance of pattern based literature, the concepts had already been applied to successful object-oriented designs, that were reused in other projects, through object oriented application frameworks.

An application framework is an abstraction that provides some generic functionality and allows programmers to extend and override its components to specialize them towards the requirements of their own applications. Frameworks are distinct from class libraries. The control flow of an application that calls class libraries is dictated by the calling code, in contrast the control flow of an application built on a framework is driven by the framework itself. Application frameworks can be considered partially complete applications, which can be completed by programmers by instantiating and extending the framework components.

Patterns and frameworks have been described as "synergistic concepts", both as a "means to achieve large-scale (code)reuse by capturing successful software development strategies" (Fayad & Schmidt, 1997). Patterns are a shared understanding of the structure and collaborations of a software component in a context, and the common vocabulary they provide allows framework design and architecture to be described and documented in terms of patterns.

This dissertation explores the practical aspects of object oriented framework development and attempts to demonstrate a pattern-oriented approach to designing, constructing and documenting a framework for Single Page Web Applications (SPA) (Scott, 2015). Single page applications are web applications that are contained within a single HTML page. An SPA never requests a new page from a web server, only refreshing parts of its page depending upon user interaction.

An SPA framework was chosen as the subject of this dissertation because the considerations in their design and architecture are the same as would be needed to design a typical GUI application for desktop and mobile platforms. The technologies that SPAs are based on are largely responsible for the fluid user experience that modern web applications provide. The traditional approach to building dynamic web applications has been using two distinct sets of technologies. Server-side technologies like JSP, ASP and PHP have been used to generate HTML content on the server, and to perform other resource intensive processing. Client-side technologies are responsible for scripting the UI and user interactions for the application on the web browser. Until Ajax (Garrett, 2005) was supported most of the widely available web browsers, most user interactions required a client-server-client round trip for the generation of an entirely new web page. An Ajax application does not require a full page reload every time new data is requested from or posted to the server, although it is not prohibited.

Single Page Applications are an evolutionary progression of Ajax based web applications. An SPA exists entirely within a single HTML page. This reduces the responsibility of the server to be simply be a source of data, while all other application concerns are handled on the client side. With more recent innovations it is now it's even possible to eliminate the server altogether, allowing the development of standalone applications entirely using browser based technologies that can still have the same networking capabilities that a web application has. Thus, SPAs emulate the experience provided by desktop applications.

## About JavaScript

Unlike traditional web applications that required two different platforms and programming languages to develop a single application, single page applications can be programmed entirely in JavaScript. SPAs have played a significant role in bringing JavaScript applications to the desktop, with the help of enabling technologies such as node.js.

JavaScript today is one of the most popular programming languages, for developing web, mobile and desktop applications. Due to its accessibility on a wide range of platforms that support it, JavaScript has seen increased adoption in the programming community. The popularity of JavaScript has steadily risen during the last two decades in the programming community. It is popular despite some of its language features, or lack thereof, that could be considered deterrents to its use for development of large scale systems. For example, JavaScript has historically lacked language level support for modules and namespaces, and its mechanism for inheritance is uniquely different from other traditional object-oriented programming languages. To a programmer who is unfamiliar with JavaScript, at first glance it may look like a procedural language with severe limitations. Experienced JavaScript programmers however appreciate it for being lightweight and extremely expressive (Crockford, 2008). JavaScript supports the principles of object-oriented programming like dynamic dispatch, inheritance, encapsulation and polymorphism in its own unique way.

JavaScript started off as a rushed prototype for a scripting language that could be embedded into, and manipulate elements of a web page (Severance, 2012). Since then it has been parallelly developed by competing vendors and has gradually moved towards standardization (Champeon, 2001). With Ajax and Web 2.0, "Software as a Service" solutions moved to the web which created demand for skilled JavaScript developers. The development of the Node.js runtime environment in 2009 brought JavaScript out of the browser. Node.js for the first time brought added I/O capabilities to JavaScript, which has led to a new generation of cross platform applications that have the same capabilities as native applications on mobile devices and desktops. Nowadays, there are number of well-designed JavaScript frameworks and applications that support multiple platforms and it is starting to be taken more seriously. The exposure of the browser's drawing API has brought JavaScript more into the mainstream. It has made esoteric programming disciplines such as game programming accessible to JavaScript developers. With web based technologies being natively adopted by several platforms, JavaScript runtimes are set to be as widely available as Java.

JavaScript has some characteristics that sets it apart from other programming languages. It belongs to a small set of languages that are natively prototype-based or "classless" (Stefanov, 2017). This means

objects are instantiated and inherited from other objects rather than classes. While the 'class' keyword has been added to JavaScript as part of its latest standard specification, it is merely syntactic sugar that has no effect on its underlying prototypical nature (MDN, 2017). Most object-oriented programming languages provide a way to declare classes as templates that their object instances must adhere to. Once instantiated an object's behavior may not be modified or extended at run time. Objects in JavaScript however are completely mutable, every object in JavaScript, including those that are built-in, may be completely altered at run time (Recent version of the ECMAScript standard do provide a mechanism to protect objects from being mutable). It is a versatile language that fully supports multiple programming paradigms. At its core, it has more in common with functional languages than object oriented ones.

JavaScript is not a "restrictionist" language, and the barrier to entry for JavaScript programming is very low. Anyone with a web browser may write a JavaScript program, it does not require any packages to be installed or any other run-time requirements. Its syntax does not necessarily encourage readable code. Static analyzers and strict type checking have desirable effect of limiting bad programming practices to some extent. JavaScript doesn't have any such constraints on it. It is easy to fall into an imperative or procedural programming paradigm, which may be sufficient for relatively simple scripting tasks, but does not scale easily for development of non-trivial applications and systems. Even though it would seem to be relatively simple to write working programs in JavaScript, programmers that are used to class based languages find may find it difficult to implement simple object-oriented designs.

The difficulties of understanding JavaScript code are both lexical and conceptual, especially so for programmers not used to dynamic or functional languages. ES6, the latest version of the standard that JavaScript is based on, introduces the class keyword to the language. This and related enhancements have arguably made JavaScript more approachable to OO programmers, and subsequent improvements may address more of its readability concerns. It still does not support access modifiers, even though "private", "protected" and "public" are reserved words.

JavaScript is the subject of this dissertation to evaluate its proclivity towards object oriented design and development in a hands-on manner, and discuss the findings objectively. Proper application of widely accepted and applied object-oriented design principles in JavaScript would lead to an enhanced understanding of the object-oriented programming paradigm. This is one of the research objectives of this dissertation.

There have already been many applications developed entirely in JavaScript, but it is still a relative newcomer and adoption of JavaScript for native application development is low. Most successful JavaScript projects consist primarily of functional source code and substantial number of them are written in languages that transpile to JavaScript.

## Pattern-Oriented Approach

Framework design is a non-trivial software engineering endeavor. A framework is the backbone of the applications that are developed on it, therefore it must be designed to be robust and extensible. An ideal framework should also be easy to comprehend and extend. This is not easy to accomplish. Lack of framework comprehension leads to its improper use and implementation issues during application

development. This dissertation is an attempt to examine an incremental, pattern-oriented self-documenting approach to framework development and evaluate its impact on framework comprehension and extensibility.

The main text of the dissertation is in the form of a narrative that walks the reader through the process of prototyping a single page application framework. and how object-oriented design patterns are used to solve the problems encountered during developing its components. It is a thorough discussion of object oriented design and development in JavaScript. We apply some of the most widely used patterns of object orient design to JavaScript in the context of designing a framework that provides support for common features of an SPA. The discussion also investigates the language agnostic nature of patterns, whether they can be implemented in a dynamic language without trying to emulate features of other programming languages.

There is a lot of supporting literature that emphasizes the integral role of patterns in the design, development and documentation of a frameworks and application. Prior knowledge of design patterns makes the various phases of framework development more intuitive and the resulting framework easier to understand and use. It could be argued that the significance of design patterns is better explained through case studies of successful large-scale applications and systems. This dissertation however takes the opposite approach.

This dissertation augments existing literature on pattern-based approaches on documenting frameworks, by detailing the design and construction of a single page application framework in JavaScript. It is a practical exercise in validating the ideas presented in related papers and theses. This dissertation may assist programmers to internalize the concept of design patterns as being fundamental to design and development of applications and frameworks. It is as much a study in patterns and frameworks as it is an investigation of how the application of patterns may vary depending upon the programming language they are being implemented in. Thus, it also addresses, to some extent, the language agnosticism of design patterns. The intent of the dissertation is not to reverse engineer existing JavaScript frameworks that support Single page applications, or to equal their capabilities by the one that will be created. Instead we walk through the process of constructing an SPA framework that is extensible enough, through appropriate application of design patterns, that more capabilities may be added to it incrementally.

We will explore a self-documenting approach to object oriented framework development in JavaScript and evaluate whether object-oriented design patterns can appropriately describe the complexity of a framework written in a dynamic, functional programming language. This involves incrementally designing and implementing the primary components of the framework and documenting the structure and collaboration of framework components in terms of design patterns. Large JavaScript code bases consist primarily of functional code, usually difficult to understand for object oriented programmers. A pattern-oriented approach also attempts to address this problem by documenting the design of an object-oriented JavaScript framework in terms of the patterns that are used to build its core components. Such a design may be more easily absorbed by programmers that are already familiar with

the patterns and the context in which they are used in other, typical object-oriented programming languages.

It can be asserted that a well-designed application or framework depends on appropriate use of patterns during its high and low-level design and implementation. Supporting and validating this assertion is one of the primary goals of this dissertation.

## Related Work

The impact of pattern-oriented design and development on the architecture of frameworks and their documentation has extensively been studied and discussed in academic works. Some of them suggest Pattern Languages as a method to enhance the documentation of complex software (Züllighoven, 1994) (Kirk, 2005). Pattern languages are "design metaphors" that describe useful design practices in terms of multiple patterns and their collaborations. Patterns and pattern languages were initially proposed design concepts to be used in the architecture of cities and buildings (Alexander, 1977). His work was later adapted by programmers to object-oriented programming, by proposing 5 patterns to build GUI based application that comprises of "windows and panes" (Beck & Cunningham, 1987).

The first formal academic work to discuss "design patterns" introduced them as a "mechanism for expressing object-oriented design experience" (Gamma, Helm, Johnson, & Vlissides, 1993). The findings of the paper pointed out the existence of "idiomatic class and object structures" that were often found in reusable designs. The pattern examples presented in the paper also focused on the concerns of building an extensible GUI based application.

A seminal book by the same authors was responsible for growing the interest in object oriented design patterns and the associated body of literature (Vlissides, Johnson, Helm, & Gamma, Design Patterns: Elements of Reusable Object-Oriented Software, 1994). The 23 design patterns, popularly referred to as the Gang of Four patterns have since then been extensively studied, discussed and applied to software projects, to the extent that they have become an essential part of a programmer's vocabulary and skill set. The patterns described in the book are the central focus of this dissertation as they can be considered the "base" patterns that other higher level architectural patterns are based on. There have even been several attempts to formalize and codify these patterns with limited success (Leeuwen, 2013; Baron, 2003).

The series of books on "Pattern Oriented Software Architecture" (Buschmann, Stal, Sommerlad, Rohnert, & Meunier, 1996) (Schmidt, Rohnert, Stal, & Buschmann, 2000) (Jain & Kircher, 2004) (Douglas C. Schmidt, Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, 4th Volume, 2007) (Douglas C. Schmidt, Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages, 2007) along with "Patterns of Enterprise Application Architecture" (Fowler, 2002), further explored the relationship between design patterns and software architectures. The patterns described in these books are usually describe higher level abstractions than the ones documented by the Gang of Four. The six books together are perhaps some of the most important and most widely circulated works on patterns and architecture. The patterns they document have found their way into many frameworks and internalized by the programming community. Contemporary discourse on patterns has since shifted to discussions of patterns that are much more domain specific than the ones described in these books.

Dirk Riehle's dissertation (Riehle, 2000) on a role modeling approach for object oriented framework design discusses role-modeling as an extension to class based modeling of object oriented frameworks. While acknowledging that framework based development contributes to product success through higher productivity and code reuse, it also identifies certain problems associated with object oriented frameworks. The dissertation discusses the difficulties associated with understanding and using a framework due to its complexity. It asserts that as a system grows in size and scope, its architecture becomes increasingly more difficult to describe in terms of the classes and objects it comprises of. Role modelling is introduced as a way of reducing the complexity of designing and documenting an object-oriented framework. A role is a higher-level abstraction than a class or a pattern, and multiple classes and objects may participate in fulfilling a particular role. The concept of a role is closely related to design patterns, some of the roles discussed in the case studies are pattern instances. The correlation between patterns and roles is especially high in the JHotDraw case study (Riehle, Case Study: The JHotDraw Framework, 2000), with all of the design patterns mapping to one or more roles. The dissertation presents the role modelling approach as a way to enhance the design documentation of JHotDraw, where a pattern based approach may not be sufficient to describe all object collaborations. Another one of his more recent papers points out that the dominant use of patterns in the software industry is in communication between software developers (Riehle, 2011).

Patterns and frameworks and their interdependence are rarely discussed in the context of dynamic programming languages however. JavaScript has not been the subject of conversations about design patterns and object-oriented frameworks together. Bridging this gap is one of the primary motivations of this dissertation.

# The Patterns

The following sections are discussions about specific design patterns, how they are relevant in JavaScript along with examples of framework components or services designed and implemented for the “Blocks” single page application framework. The examples are illustrated through figures and code listings.

We will incrementally develop the framework, adding support for building features that a typical single page application would require. Each feature is discussed in the context of one or more design patterns that we might deem appropriate toward producing a reusable, extendible design for the components that support it. The framework has three high level modules. The first module is responsible for Ajax related functionality and any other HTTP related concerns. The second module is the view engine of the framework, responsible for generating and managing GUI related concerns of the application.

The discussions will focus primarily on the Gang of Four design patterns proposed in the seminal book (Vlissides, Johnson, Helm, & Gamma, Design Patterns: Elements of Reusable Object-Oriented Software, 1994). Since we are designing an object-oriented framework for single page applications with no domain specific concerns, the “recurring problems” and related patterns that we include in the discussion should be the ones related to the object-oriented design. Henceforth, a standard description of a pattern will simply be referred to as Gang of Four or abbreviated as GoF when comparing to the JavaScript framework specific implementation.

It has been suggested that higher level architectural patterns must be considered before low level patterns when designing a system (Buschmann, Stal, Sommerlad, Rohnert, & Meunier, 1996). The high-level architectural pattern of the SPA framework is Model-View-Controller. The core components of the framework will be prototyped in subsequent sections. Since we are applying a pattern-oriented approach to a specific technology and programming language, it makes sense to focus on low level patterns, because we also need to address the implementation details of the patterns in the that language.

## Foundational Patterns

As mentioned earlier, JavaScript has historically lacked language level support for modules (Cho & Ryu, 2014), which has been addressed to some extent with the introduction of classes in its latest standard. The problems associated with lack of support for namespaces remain however. Components of a JavaScript program can only interact in the global namespace. This permits entire JavaScript programs to be written with variables in the global namespace. This is clearly not a good practice as it might lead to variable name collisions and simple changes to global variables cause unexpected behavior in other parts of the application.

JavaScript programmers have developed techniques to address the problem of global scope using modules as a way of localizing variable and function names. ES6 classes also belong inside such modules

to prevent naming conflicts. Since these techniques are not part of the language itself, they are also considered “patterns”. Any discussion of design and architecture in JavaScript must be preceded by a discussion of these foundational patterns.

Objects in JavaScript are similar to associative arrays, a collection of key-value pairs, where values can be primitive types or other objects. Since functions are also objects in JavaScript, values can also be functions and closures. Thus, new objects can be created simply by declaring an associative array and assigning it to a variable. This is what is known as the Object-literal pattern in JavaScript (Murphey, 2009). An Object literal is the simplest way to encapsulate data and behavior in JavaScript. Every key is an object literal is effectively a namespace, which may contain other object literals to achieve nested namespaces.

```
objectLiteral = {
    counter: 0,
    increment: function () {
        this.counter++;
    }
}
```

*Code Listing 1 A JavaScript Object Literal*

The Module pattern is an extension of the object literal pattern emulates the concept of classes by allowing public and private members for an object. It is another way of creating namespaces in JavaScript as a module may encapsulate other modules, object literals and classes. A module pattern allows programmers to hide parts some parts of the module from the global scope. The module pattern is implemented using a programming technique unique to JavaScript, known as an Immediately Invoked Function Expression (IIFE). An IIFE is essentially an anonymous function that is called as soon as it is declared (Osmani, Namespacing Fundamentals, 2012). In JavaScript, an Object or function is completely aware of the environment that it’s declared in. Variables and functions declared inside other functions however are not accessible from outside, thus IIFEs are an effective way to ensure privacy of encapsulated code. The module pattern is implemented by assigning the result of an IIFE to a variable. Unlike the object literal example in listing 1, listing 2 shows an equivalent implementation of the module pattern using an IIFE, to create an object with a private counter variable. This allows programmers to include members in the module that can only be consumed within it and are invisible to the outer scope.

```
module = (function () {
    var counter = 0;
    return {
        increment: function () {
            return counter++;
        }
    }
})();
```

*Code Listing 2 A JavaScript module*

Finally, the revealing module pattern is a slight variation of the module pattern which makes designated private members public by "revealing" public pointers to them. The revealing module can be considered a more readable version of the module pattern and allows consistent naming scheme for members. In the example shown below, an ajax module exposes methods from an internal nested module.

```
revealingModule = (function () {
    var counter = 0;
    function _increment() {
        return counter++;
    }
    return {
        increment: _increment
    }
})();
```

*Code Listing 3 A revealing module with naming conventions*

## Pattern: Prototype

The prototype pattern allows a reduction of the total number of classes as well as the number of new objects that must be created in an application. The pattern is applicable when "instances of a class can have one of only a few different combinations of state" (Vlissides, Johnson, Helm, & Gamma, Creational Patterns, 1994). This is achieved by prototype objects supporting a Clone operation that returns a copy of the prototype. The prototype can allow certain properties to be set on it, to get it into a desired state before or after cloning. The key consequence of the prototype pattern is that allows creation of new "types" of objects without defining additional classes.

Some critical discussions on design patterns emphasize the point that design patterns are an indication of missing language features or that some language features may make certain design patterns "invisible or simpler" (Norvig, 1996). Prototype is an example of a pattern that is truly invisible in JavaScript. Any discussion of the prototype pattern in JavaScript point out the prototypal inheritance mechanism of JavaScript (Timms, 2016) (Osmani, Prototype Pattern, 2012).

The prototype pattern is invisible in JavaScript even if we ignore prototypical inheritance. Any new object creates in JavaScript inherits from the base JavaScript object – **Object.** This is similar to other Type based languages, such as *java.lang.Object* in Java and *System.Object* in C#. Any object created in JavaScript is mutable. The dynamic nature of JavaScript allows properties to be assigned to objects at run-time. These properties are owned by that particular instance and not shared by others even if they had the same base object at the time of instantiation. JavaScript allows complete modification of state and behavior on an object unless it is explicitly sealed. This kind of dynamic, run time modification of state and associated behavior is the crux of the prototype pattern.

### Prototyping the base module

As an example of using the prototype pattern in JavaScript, we may consider the top-level module in our SPA framework to be a prototype. All the framework code is encapsulated in a module named "Block". Each instance of the Block module is associated with an independently updateable section in the single

page application. The Object.create method in JavaScript corresponds to the clone method of the prototype pattern. Object.create takes an object as its argument, which becomes the prototype of the newly created returned by the function. Figures 1 and 2 compare the structure of the Prototype pattern as illustrated in GoF vs its application in the SPA framework.

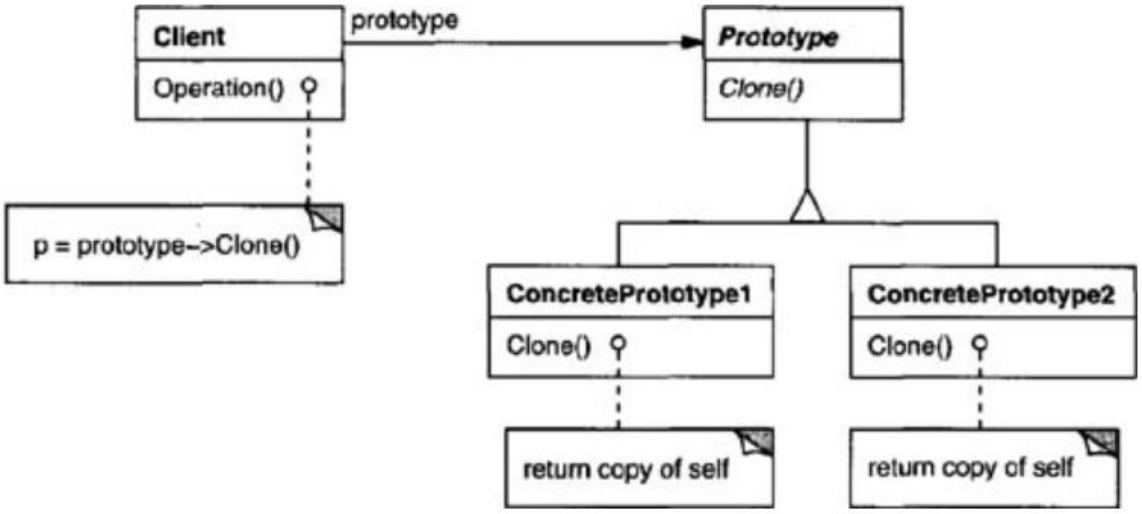



*Figure 1 GoF Prototype Pattern (Vlissides, Johnson, Helm, & Gamma, Creational Patterns, 1994)*

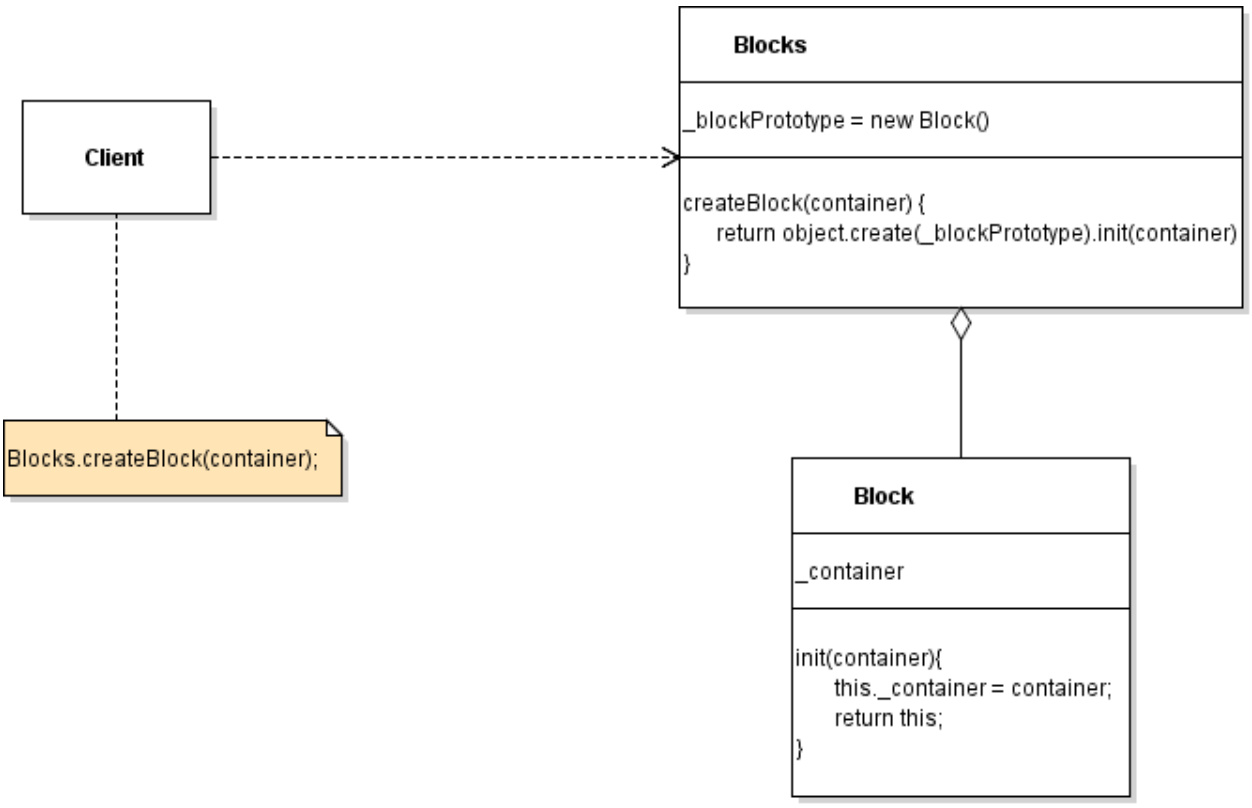



*Figure 2 Blocks framework prototype pattern*

At the time of prototype initialization all blocks are the same. At the very least we would want the different sections of the single page application to have distinct views that may possibly represent distinct data models. These properties are set on the prototypes after initialization, they are passed different container elements for rendering the view by combining a model and a template to generate a dynamic view.

This example gives us a first glance at what inheritance looks like in JavaScript. There are many syntactical variations to doing this, but the essence is the same. There is no real concept of classes in JavaScript. Properties and behaviors may be added to an object dynamically, and these properties and methods are owned by the that object alone. One way to make objects inherit from other objects is to utilize the "prototype chain" in JavaScript. Every object created in JavaScript has a "prototype" property which can be used to assign members to objects of that type. Any object that has its prototype property set to another object, also inherits all its members.

There is one caveat to using the prototype pattern for inheritance in this way. The JavaScript runtime needs to look up the prototype chain for any invoked property or method, which has a performance impact. The deeper and invoked prototype is, the more iterations it takes to invoke it. However in this case the pattern is appropriate as we are expecting the prototype chain to be only one level deep.

## Pattern: Factory Method

The factory method is the simplest GoF design pattern to implement and the most common. A factory method is used when a client cannot anticipate which one of the several sub-types of a given type needs to be instantiated. A factory method "encapsulates the knowledge" of which subclass to create and return.

The crux of the Factory method is that it must create a new instance of a class and return the newly created instance to the client, which only knows about the interface of the received object. Since JavaScript does not have any notion of interfaces, the created object must support the properties and operations that the client expects. The factory method is so called because a method name *createxyz* is usually part of its implementation, where *xyz* corresponds to the name of the object being created.

## Creating the xhr object

In the following example of the Factory method, the type of object that needs to be created is *XmlHttpRequest* (Gross, 2006). Any single page application must have the ability to make Ajax calls to request content from the server. This is done through the *XmlHttpRequest* Object. Because there are more than one JavaScript engines in distribution and the multiple (browser) vendors usually take time to completely converge on the standard, there is more than one way of instantiating the *XmlHttpRequest* object depending on the version of the browser. The latest versions of most browsers implement the standard *XMLHttpRequest* object, but on older versions of internet explorer the same functionality is available through an *ActiveXObject*

Regardless of the browser for which it is instantiated, a common set of methods and properties is associated with *XmlHttpRequest* for all platforms. These common properties and methods constitute an invisible interface. The client of the Factory method only knows about the interface and would attempt to invoke one or more of these methods on whatever object it receives from the factory method. The factory method is implemented as a single function, the sole member of a JavaScript module named *xhrFactory*, as shown in the listing 4.

```
xhrFactory = (function () {
    _createXHR = function () {
        if (window.XMLHttpRequest) {
            return new XMLHttpRequest();
        }
        else if (window.ActiveXObject) {
            var msxmls = new Array(
                'Msxml2.XMLHTTP.5.0',
                'Msxml2.XMLHTTP.4.0',
                'Msxml2.XMLHTTP.3.0',
                'Msxml2.XMLHTTP',
                'Microsoft.XMLHTTP');
            for (var i = 0; i < msxmls.length; i++) {
                try {
                    return new ActiveXObject(msxmls[i]);
                } catch (e) {
                }
            }
        }
        throw new Error("Could not instantiate XMLHttpRequest");
    }

    return {
        create: _createXHR
    }
})();
```

*Code Listing 4 An xhr Factory (Gross, 2006)*

Many popular JavaScript frameworks make use of the factory method pattern to create the xhr object. It is a common way to shield JavaScript framework users from browser specific details. The AngularJS framework even exposes an *xhrFactory* as a service to be replaced or decorated to create custom xhr objects.

The Gang of Four implementation emphasize that the factory method pattern “lets a class defer instantiation to sub classes” and that “Frameworks use abstract classes to define and maintain relationships between objects” (Vlissides, Johnson, Helm, & Gamma, Creational Patterns, 1994). Abstract classes are usually associated with inheritance hierarchies. In this framework specific example however, there is no evidence of classes or abstractions. This illustrates a key difference between JavaScript and other traditional object-oriented programming languages. Due to its prototypal, object-based inheritance mechanism, parallel class hierarchies like the ones described in the book will rarely be found in JavaScript. This is a significant departure from hierarchical, class based inheritance that are evident in many object-oriented designs.

Gang of Four discusses creational design patterns in terms of "Products" and "Creator" abstractions. Multiple subclasses called *ConcreteCreator* and *ConcreteProduct* participate in the structure of the various creational patterns. In the above example, *ConcreteProduct* is the *XmlHttpRequest* Object, and *ConcreteCreator* is the *xhrFactory*. *Product* and *Creator* are interfaces, a language feature that JavaScript does not support. Assuming a completely different object that supports making ajax requests is implemented in another browser engine. We will need to create another instance of the factory to create that object, but only if the ajax API provided by the new object is different from the xhr object. In that case the two factories will not share an invisible interface.

In a dynamic, loosely typed language like JavaScript, interfaces can only be implied, not enforced. This concept is also known as duck typing which is common in most dynamic programming languages.

## Pattern: Proxy

A proxy is essentially a gatekeeper for another object by encapsulating it through composition. Thus, the proxy object acts as a surrogate for the contained object (the subject). Proxy is a commonly used pattern and can be applied in a variety of scenarios. Remote proxies are usually frameworks for web services where a client may refer to a proxy object that ultimately makes requests to the remote web service. Web service proxies act as local objects to the client and shield it from all the networking concerns that are managed by the proxy itself. In this case the subject of the proxy is the remote web service.

The purpose of a proxy is to control access to the subject. A proxy may be used to defer initialization of a subject if its creation is expensive. This scenario is also common in web service proxies where initialization of network resources is usually time consuming. A protection proxy manages access rights to the subject and provides conditional access to the client. A proxy may also be used to perform additional tasks such as locking of the subject for thread synchronization, or loading the subject into memory from a persistent data store before granting access to it. These are some of the common use cases of the proxy pattern described in GoF (Vlissides, Johnson, Helm, & Gamma, Behavioral Patterns, 1994).

Later versions of the ECMAScript standard provide some language level support for the proxy pattern (Osmani, Introducing ES2015 Proxies, 2016). The Proxy object allows intercepting the operations on a target object by providing trap methods that may perform additional operations on the target. If a trap method is not provided to the Proxy, corresponding requests are forwarded directly to the target object.

### Controlling access to the xhr object

The xhr object provides two methods for making Ajax requests, “open” and “send”. Using the xhr object to invoke these two methods makes a synchronous ajax request. The program must wait for the request to complete before moving on. This causes the browser to freeze while the request is in progress. For this reason, synchronous requests have been deprecated and making a synchronous ajax request generates a console warning in most browsers. To make an asynchronous request, the script must handle the *onreadystatechange* event of the xhr object. The event handler must provide a callback function that is invoked with the result of the completed request, usually an HTML page or a JSON object. While it may be acceptable to allow client code to handle the onreadystatechange event by itself, it makes better design sense to encapsulate the concerns of making the ajax request.

We create a new module named “async” that controls access to the xhr object and implements an event handler that monitors the *readystate* of the ajax request with a switch statement. It exposes a simplified request API to the client that only needs to call the async module with the url of the request and a callback function. The callback is invoked inside the event handler in the async module when the ajax request is complete.

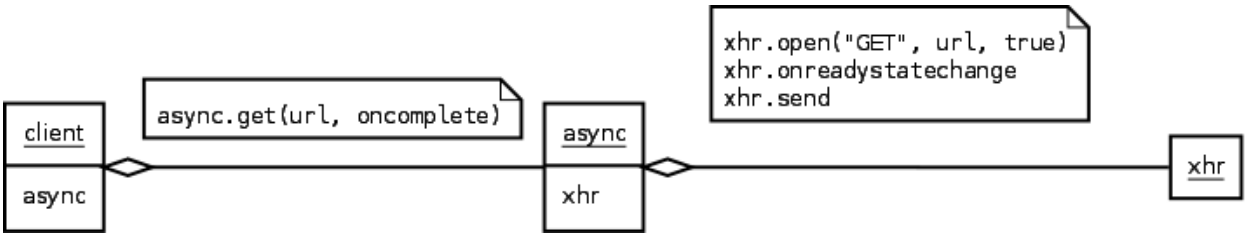



*Figure 3 asynchronous proxy*

This implementation of an asynchronous proxy does not simply mirror the requests on the subject, but combines them into a single request, shielding the client from the details of the xhr object altogether. This is a key point to consider when considering the appropriateness of applying a design pattern to solve a problem. The solution may not strictly adhere to the structure described through UML diagrams and code samples in pattern literature. In almost all cases a particular pattern implementation will deviate slightly from implementation other contexts. Design patterns must be adapted to problems without forcing their application through over design.

Let us consider another implementation of the proxy pattern for the SPA framework. This time it is a proxy that controls access to the asynchronous proxy itself. Web application frameworks often have support for caching the results of an HTTP request. There are many approaches to caching in web application. Server-side caching enables caching the http out in a process on the web application server. HTTP caching is a mechanism that is built into the protocol itself, controlled by properties included in HTTP headers. With some of the new HTML5 APIs for local storage on the client side, even more caching strategies are possible. Thus, we can implement a cache proxy that can do a cache lookup before routing the request to the asynchronous proxy.

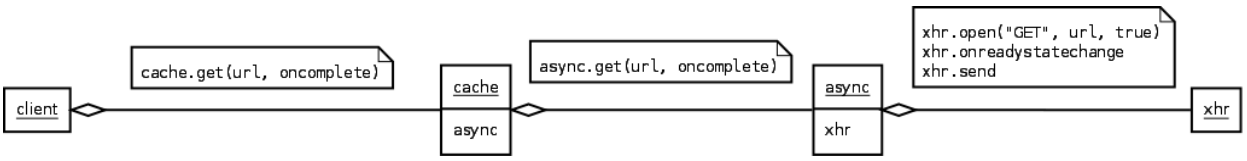



*Figure 4 cache proxy*

Caching is not a trivial concern and cache invalidation is considered one of the most difficult programming tasks. Therefore, the actual cache must be implemented in a completely different module, delegating caching responsibilities to external libraries that are built for that purpose. The cache proxy simply looks up the cache for a cached response for the current url that is not yet invalidated. If such a result exists, the cache proxy invokes the callback function with the stored result, otherwise the request is routed to the asynchronous proxy. This is another non- typical example of smart reference that performs some meaningful task before granting access to the subject.

## Pattern: Template Method

The template method pattern is a way to achieve runtime polymorphism, by allowing certain sections of an algorithm to be “farmed out” and fulfilled externally. A standard implementation of the template method pattern involves defining the structure of an algorithm in an abstract class with overridable

template methods acting as placeholders for the portions of the algorithm that must be implemented by subclasses. Thus, all the subclasses implement a variation of the same algorithm and each concrete sub class contains a complete algorithm implementation.

Template method pattern essentially plugs in function into an algorithm skeleton. In JavaScript, there are multiple ways to solve the problem of defining a partial algorithm. JavaScript supports first-class functions, which means functions are treated as regular objects and they may be passed to other functions as arguments. The template methods can thus be fulfilled by function parameters instead of subclasses. This approach is even applicable if the template methods need access to the private state of the module that declares the algorithm skeleton, as the injected functions will be invoked from within its scope. The appropriateness of either approach to solve a problem depends upon its context.

## Plugging in a progress indicator

HTTP requests take time to complete, even on high bandwidth networks HTTP responses will rarely be instant, especially if the server must perform expensive disk based I/O. To provide a fluid user experience, the SPA framework needs to support a progress indicator while a section of the page is waiting for an ajax response. To accomplish this, we need to modify the *onreadystatechange* event handler in the asynchronous class by inserting template methods to show and hide a progress indicator at appropriate points in the switch statement. We discard the functions as arguments approach for this case, because it will require changing the signature of the get method. The template methods must therefore be fulfilled by an external object.

```
this._xmlhttp.onreadystatechange = function () {
    switch (instance._xmlhttp.readyState) {
        case 0:
            instance.unintialized();
            break;
        case 1:
            instance.loading();
            break;
        case 2:
            instance.loaded();
            break;
        case 3:
            instance.interactive();
            break;
        case 4:
            instance.complete(instance._xmlhttp.status,
                instance._xmlhttp.statusText,
                instance._xmlhttp.responseText, instance._xmlhttp.responseXML);
            break;
    }
```

*Code Listing 5 Handling the onreadystatechange event with template methods (Gross, 2006)*

Now we could use prototypal inheritance to create a “sub-object” of the *async* module that implements the methods to show and hide a progress indicator. However, creating a subtype of an ajax related module simply for handling a GUI concern is isn’t appropriate or necessary. Another approach is to employ the mutability of JavaScript’s objects are reset the public pointers to the template methods directly on an instance of the *async* module. This however would require the *async* module to be

accessible from outside the scope of the ajax module, which is bad design and violates the modularity of the framework. Another rather inventive way to inject the behavior of template methods is to pass the implementation functions as parameters to the ajax module's IIFE itself.

This technique is often used to effectively incorporate external libraries into a module. For example, the following example shows the jQuery library being passed as a parameter to the *domUtil* module, to simplify DOM related operations for the framework.

## Pattern: Façade

The Façade pattern is an object that combines "sub-systems" and provides a simple, unified interface to access their functionality. The purpose of the Façade pattern is to simplify a set of granular APIs. Façade objects shield the client from the complexity of their subsystems, while also decoupling subsystems from the rest of the application, increasing their portability. Facades may compose subsystems hierarchically, thereby layering them and simplifying their interdependence. Facades also serve the purpose of preventing their subsystems to be used in an unexpected manner. The ability to independently build and deploy components is crucial to large scale software systems. For compiled languages, facades provide the added benefit of reducing compilation dependencies (Vlissides, Johnson, Helm, & Gamma, Behavioral Patterns, 1994). The Façade pattern has no specific structure, and a typical way to implement it is by object composition.

Facades are simpler and more intuitive to implement in JavaScript that in other class based languages. The façade pattern is central to almost all JavaScript libraries and modules. The jQuery library itself is a fitting example of a façade. The jQuery library was initially written to simplify working with the DOM API and handle ajax requests. These are two interdependent subsystems on their own. It provides a single method to replace many DOM API calls to select elements on an HTML page. jQuery's selector routes the request to the appropriate DOM API method depending upon the arguments that are passed to it. Since its initial release jQuery has added many other subsystems to add support for animations and HTML templating. Beyond the subsystems already part of it, jQuery also supports additional features to be plugged in, and all of this is exposed through a single façade, the jQuery object itself.

### An Ajax Façade

To more closely evaluate the application of the façade pattern in JavaScript, we may revisit the ajax related functionality that has been incrementally designed and implemented during previous pattern discussions. We briefly discussed the ajax module previously during the application of the template method pattern. The first ajax related module developed was the *xhrFactory*, followed by the *async* proxy that encapsulates it. The *async* proxy was further encapsulated with the *cacheproxy*. The *cacheproxy* needs to perform lookups against a local cache store, it makes sense to have a cache module dedicated to caching ajax responses. The cache module may be kept as a placeholder for future caching functionality, only exposing high-level APIs and exposing stub operations that the *cacheproxy* can use. Until a true cache is actually implemented, get requests will simply fall through to the *async* module to no adverse effect. These four modules (*xhrFactory*, *async*, *cacheproxy* and *cache*) are interdependent

and they work together to provide functionality related to one high level concern of the Blocks SPA framework, making ajax requests. The rest of the modules in the framework do not need to be directly coupled to the all the four ajax modules, only the *cacheproxy* and *async* modules provide APIs that would be required by other components, and framework users. Thus, they could be enclosed into another module that exposes a simplified API to the "external-world". Figure 5 and listing 6 illustrate the ajax façade and the API it exposes

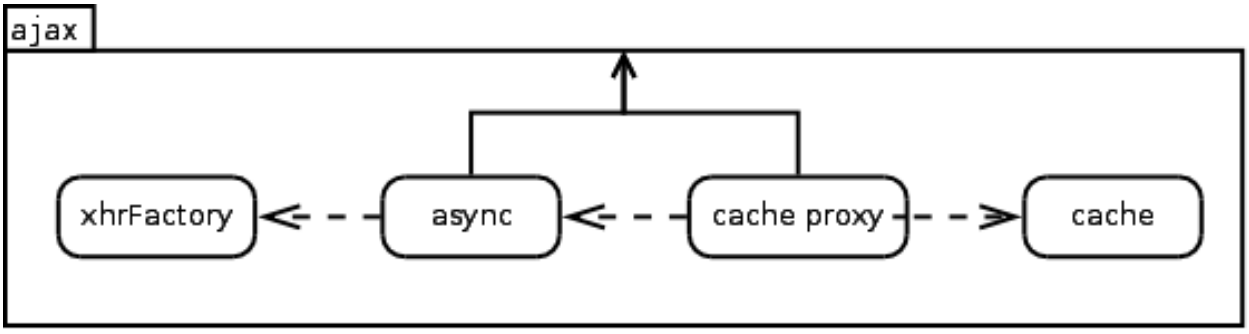



*Figure 5 Ajax Facade*

```
ajax = (function () {
    xhrFactory = (function ()...)();
    async = (function ()...)();
    cacheproxy = (function ()...)();
    cache = (function ()...)();
    return {
        get: cacheproxy.get,
        post: async.post
    }
})();
```

*Code Listing 6 Ajax Facade*

There is a one-way dependency between any two sub-modules and the complexity of the inner modules is hidden away from the client. The Façade this provides a smaller interface to the client, consisting of two methods, to perform get and post http requests. A post request gets routed directly to async proxy while a get request gets routed through the cache proxy to check for cached results first.

## Pattern: Flyweight

The Flyweight pattern encapsulates a set of lightweight, primitive, "fine-grained" objects called "flyweights", that are widely used in an application. It allows design flexibility in the application at a granular level if appropriately applied. Flyweights are mostly shared objects, which allows the same instance of a flyweight to be used in a variety of contexts. When used appropriately the pattern results in cost saving for the application in terms of memory usage. The Cost savings increase with the number flyweight objects, and the number of times a flyweight is used within an application. Access to the flyweights is managed through a flyweight factory, which is responsible for storing and creating

flyweight instances. This ensures that new instances of flyweights are not created directly by the client, they can only be accessed through the factory. The pattern enables run time addition and modification of “extrinsic state” on the flyweight depending upon the context in which it is being used.

## Creating a tag registry

A framework for single page applications will need the ability to generate a UI dynamically. A view in an SPA will comprise of HTML markup, which in turn comprises of various HTML tags and the set of styles & event handlers associated with them. The tags also need to be converted into a DOM element for additional (extrinsic) state (styles, events) to be attached to and removed from the respective HTML elements at run time.

As the number of views in an SPA grows, so does the number of dynamically generated HTML elements. Creating a new element for each tag from scratch every time it is needed will become cost prohibitive as the application scales. Composing many programmatically created nodes can also become cumbersome and difficult to understand and maintain. We need to come up with a design that allows a cost-efficient way to generate dynamic DOM elements and compose them in a view, while also allowing granular control over the markup it consists of.

JavaScript does not provide a way to create an HTML element from text markup in its DOM API. The only DOM method that creates an HTML node takes a single argument, the tag name, and returns a DOM Node. Creating a single node using this method is more efficient than creating one using markup text, which requires some additional steps. However, the advantage in efficiency dissipates as the element to be dynamically generated becomes more complex, with multiple nested and sibling nodes. Also, not all elements in the page need to be dynamically generated, only the ones that require extrinsic state to be manipulated.

Code listing 7 shows an implementation of Flyweight pattern for creating markup dynamically.

```
_elementFactory = (function () {
    _tagRegistry = {
        'p': '<p {{attributes}}>{{inner}}</p>',
        'a': '<a {{attributes}}>{{inner}}</a>',
        'div': '<div {{attributes}}>{{inner}}</div>'
    };

    _createElement = function (markup) {
        var div = document.createElement('div');
        div.innerHTML = markup;
        return div.childNodes;
    }

    _createTag = function (tag, inner, attributes) {
        if (_tagRegistry[tag]) {
            let nodeMarkup = '<' + tag + '{{attributes}}>{{inner}}</' + tag + '>';
            _tagRegistry[tag] = nodeMarkup;
        }
        return _tagRegistry[tag].replace("{{attributes}}", attributes).replace("{{inner}}", inner);
    }
    return {
        createElement: _createElement,
        createTag: _createTag
    }
})();
```

*Code Listing 7 Flyweight for HTML tags*

Complex markup can be generated using the above module by nesting calls to *createTag* method, which is another example of a factory method that creates new flyweight objects depending upon its arguments. The flyweight objects in this case a simply a string template for a tag. It could be argued that creating a new string every time it is needed is not very expensive. However, HTML has over 50 tags for sectioning, formatting, input and semantics. It could be rewarding from a design perspective to have a central module for generating markup dynamically.

## Creating a template registry

Another feature that an SPA framework must support is HTML templates. All JavaScript MV* frameworks define their views as declarative HTML templates, that have place holders for properties that will be replaced by corresponding properties in the model.

```
extends layout

block content
  h1= title
p Welcome to #{title}
```

*Code Listing 8 Express framework jade view template*

```
<div class="entry">
    <h1>{{title}}</h1>
    <div class="body">
        {{body}}
    </div>
</div>
```

*Code Listing 9 Handlebars framework template*

On traditional client server MVC based web applications, the process of replacing placeholders with actual data occurs on the server side, with the resulting HTML being sent to the browser as part of an HTTP response. This however does not preclude the need for using templates in the browser. Most modern web applications use HTML templates both on the client and the server. A Single page application may request all relevant code and resources from the server in a single request, or on demand. If the templates reside in separate files, they will need to be compiled into an in-memory representation before they can be used by the application.

Whether the templates reside on the client or the server, an application will need a way to easily access a view template through an identifier. The flyweight pattern example implemented for HTML tags can be adapted with a few modifications to act as a registry for HTML templates. Code listing 10 shows the implementation of a *templateFactory* object that manages access to a template registry. Its structure is very similar to the *elementFactory* implemented above, with a minor difference. The *templateFactory* does not have a way to automatically create a requested template if it does not already exist. zthe conditional expression to check for the existence of a tag is no longer required. Instead, an additional method is added to the factory to explicitly register new templates.

```
templateFactory = (function () {
    _templateRegistry = {
        indexTemplate: '<div class="indexRow" href="{{href}}"><div class="indexTitle">{{seq}}    {{linkText}}</div></div>',
        pagerTemplate: '<span class="pagerLink" href="{{href}}">{{linkText}}<span>'
    };
    _createElement = function (markup) {
        var div = document.createElement('div');
        div.innerHTML = markup;
        return div.childNodes;
    }
    _registerTemplate = function (name, template) {
        _templateRegistry[name] = template;
    }
    _createTemplate = function (name) {
        return _templateRegistry[name];
    }
    return {
        createElement: _createElement,
        registerTemplate: _registerTemplate,
        createTemplate: _createTemplate
    }
})();
```

*Code Listing 10 templateFactory*

The purpose of the flyweight pattern as explained in this section is simply to act as a central repository for all view templates that will be used in the application.  It enables the controllers in the application  to find view templates by name instead of maintaining a reference to the template string itself. Also, the *templatefactory* has the potential to provide more cost savings than the *elementFactory*. A template will

usually consist of markup containing multiple HTML tags, and as stated earlier, the cost penalty of creating an HTML element from markup dissipates as the size of the markup increases.

## Pattern: Builder

The motivation for the Builder pattern is to allow a complex object to be converted into different representations, where the number of possible representations an object might have is open ended. A complex representation of the object may be constructed one part at a time, while the client is unaware of the construction process.

The Builder pattern can be thought of to convert an Object from one representation into another. The participants in the pattern are 1) A Builder - a class encapsulating the creation of one representation of an object and 2) A Director class- a class responsible for encapsulating and parsing the original object (or data), and an instance of the builder class. The Builder is responsible for constructing one representation of the object one part at a time. The Director class traverses through the original object and sends parts of it to the builder for construction. The example for the Builder pattern discussed in the Gang of Four book converts an RTF document into other formats such as ASCII and *TeX*. In this case the Director is the *RTFReader* class, which parses the RTF document one token at a time. Each of the target conversion formats have their own builder class called *ASCIIConverter* and *TeXConverter*, which convert the RTF tokens into their respective formats, handing characters, fonts and paragraphs and storing the result.

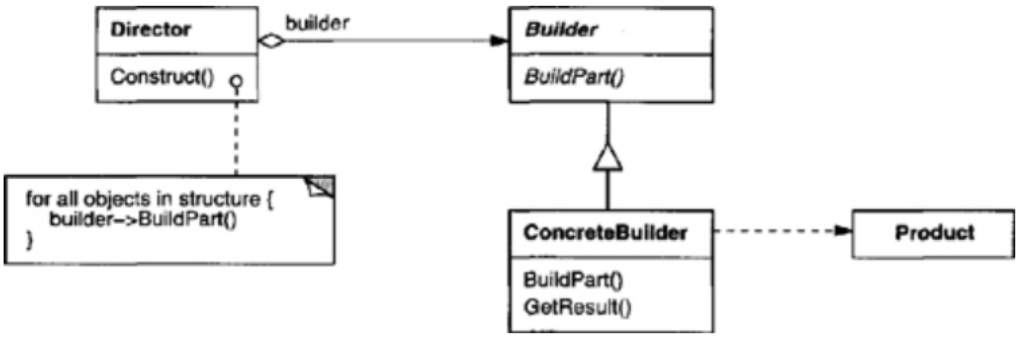



*Figure 6 The Builder pattern (Vlissides, Johnson, Helm, & Gamma, Creational Patterns, 1994)*

In the context of web applications there are some common representations of data that are frequently encountered, namely JSON, XML, HTML, Markdown etc. In web applications, data may be requested from servers in multiple formats. JSON (JavaScript Object Notation) is a popular data-interchange format for the web, consisting of name-value pairs and arrays. Converting JSON into an HTML representation is a very common use case in ajax based web applications. Data received in JSON needs to be converted into an HTML representation before it can be displayed on a web page. This is a good enough context to explore the application of the builder pattern to design a solution, especially if the JSON data needs to be iterated through to “hydrate” an HTML template incrementally or recursively.

## Creating an SPA view

As mentioned in the previous pattern discussion for the flyweight pattern, HTML templates are an essential part of any ajax based single page application. Before a template can be used in a web page however, the property placeholders contained in it must be replaced by properties from actual data, usually contained in a JSON object. This is known as hydrating a template, and the entire process of creating a view from JSON and templates is known as client-side templating. A variation of this is server-side templating, where server-side code may be intermingled with HTML markup before generating an HTML response that is sent to the client. In most SPA frameworks, templating concerns are usually handled on the client side. There many JavaScript frameworks that provide templating support and a few of them solely focus on it. We could always use an external library to handle the templating concerns for our SPA framework, but we do not currently need all the additional functionality and its associated weight. We would like to keep our design open enough so that it does not prevent us from using external templating libraries at a later point if we want to.

Figure 7 illustrates the of application a simple JSON object to a template to produce the resulting HTML.

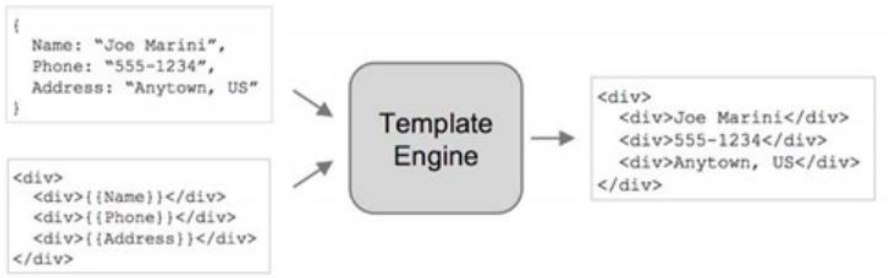



*Figure 7 Templating (Source: www)*

This example seems simple enough that it could be implemented in a single function. However, in real world scenarios

Let's consider an index of HTML documents, like the table of contents of this dissertation. An index with multiple nested headings and hypertext references to the corresponding articles on the server as shown in the code listing 11.

```
var indexModel = [
        { seq: "1", href: "/dissertation/abstract.html", linkText: "Abstract", children: [] },
        { seq: "2", href: "/dissertation/introduction", linkText: "Introduction", children: [] },
        { seq: "3", href: "/dissertation/frameworksAndPatterns", linkText: "Frameworks and Patterns", children: [] },
        {
            seq: "4", href: "/dissertation/aboutJavascript", linkText: "About Javascript", children:
            [
                { seq: "4.1", href: "/dissertation/aboutJavaScript/classesAndNamespace", linkText: "Classes & Namespace", children: [] },
                { seq: "4.2", href: "/dissertation/aboutJavaScript/inheritance", linkText: "Inheritance", children: [] },
                { seq: "4.3", href: "/dissertation/aboutJavaScript/inheritance", linkText: "Interfaces", children: [] }
            ]
        },
        {
            seq: "5", href: "/dissertation/foundationalPatterns", linkText: "Foundational Patterns", children:
            [
                { seq: "5.1", href: "/dissertation/foundationalPatterns/objectLiteral", linkText: "ObjectLiteralPattern", children: [] },
                { seq: "5.2", href: "/dissertation/foundationalPatterns/module", linkText: "Module & Revealing module", children: [] }
            ]
        },
        {
            seq: "6", href: "/dissertation/gof", linkText: "Gang Of Four Patterns", children:
            [
                { seq: "6.1", href: "/dissertation/gof/factory", linkText: "Factory Pattern", children: [] },
                { seq: "6.2", href: "/dissertation/gof/singleton", linkText: "Singleton Pattern", children: [] },
                { seq: "6.3", href: "/dissertation/gof/builder", linkText: "Builder Pattern", children: [] },
                { seq: "6.4", href: "/dissertation/gof/composite", linkText: "Composite Pattern", children: [] },
                { seq: "6.5", href: "/dissertation/gof/decorator", linkText: "Decorator Pattern", children: [] },
                { seq: "6.6", href: "/dissertation/gof/adapter", linkText: "Adapter Pattern", children: [] },
                { seq: "6.7", href: "/dissertation/gof/prototype", linkText: "Prototype Pattern", children: [] },
                { seq: "6.8", href: "/dissertation/gof/observer", linkText: "Observer Pattern", children: [] },
```

*Code Listing 11 JSON representation of a document index*

In the following example, we use the builder pattern to create an html index with anchor tags and appropriate indentation. We create the *ModelDirector* and *TemplateBuilder* classes. Note that we are referring to the two constructs as classes and not objects. This is because objects of these two types require explicit initialization using the “new” keyword, and maintain private fields. The Director class takes two arguments as it's constructor parameters, the JSON to iterate through, and an instance of the Builder class. The "construct" method of the director class iterates through the JSON array and passes index items to the builder instance to do the bulk of the work. Figure 8 illustrates the structure and collaboration of this builder pattern implementation.

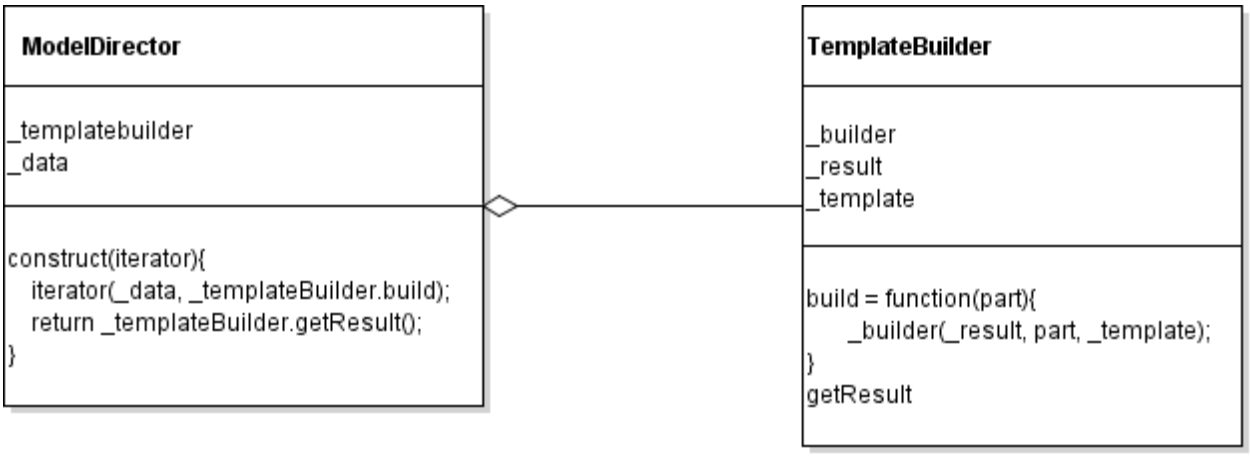
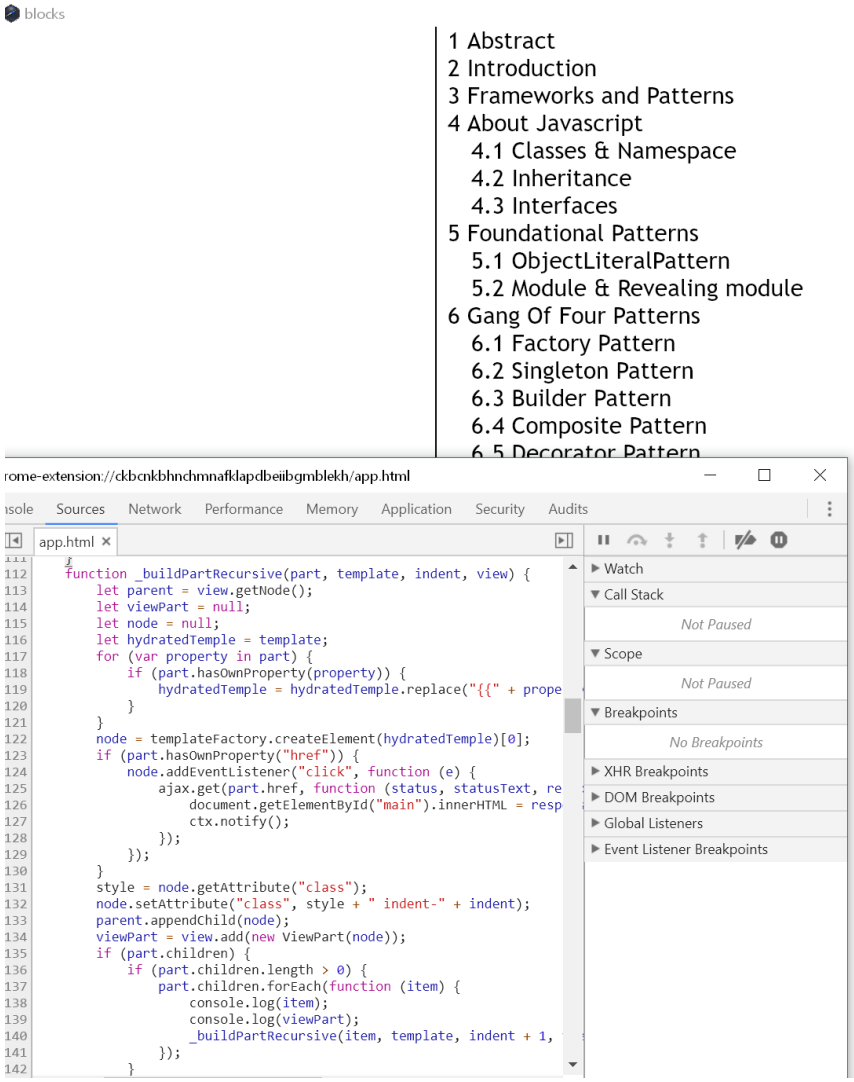



*Figure 8 Builder pattern for “client-side” templating*

The separation of the Director and the Builder into two different classes makes sense as their responsibilities are different. The two classes provide two separate points for customizing the template building process. The director controls how parts of the JSON model are passed to the builder, and the

Builder controls the combining a part of the model to a given template. Different scenarios with varying JSON and template structures and view requirements will need to be handled accordingly. We may note here that certain variables are being invoked as methods, which is possible in JavaScript because functions are treated as regular objects. This is another example of employing the functional strengths of JavaScript to produce an extendible design. Figure 12 demonstrates the application of the builder pattern to build a view recursively, by iterating through the *indexModel* JSON data shown in Code listing 11




*Code Listing 12 Builder Pattern with recursive templating*

New behaviors may be added to the *Director* and *Builder* classes simply by passing different iterator and builder functions to them. We may even implement recursively building a template by passing a recursive function. This it allows us to handle multi-dimensional arrays and nested objects using the same class without having to extend it. Listing 13 shows an API of Handlebars which allows registration of a function to handle a list in the data model. This function is invoked whenever the templating engine encounters a list property in the JSON its parsing.

```
Handlebars.registerHelper('list', function (items, options) {
    var out = "<ul>";

    for (var i = 0, l = items.length; i < l; i++) {
        out = out + "<li>" + options.fn(items[i]) + "</li>";
    }

    return out + "</ul>";
});
```

*Code Listing 13 http://handlebarsjs.com/*

Once again, The *Director* and *Builder* "interfaces" are only implied. For a different Builder implementation to be used with the same director it must conform to the same interface to avoid modifying the director. The above implementation of the builder pattern creates a view recursively and the two classes involved are central to the framework's view component. Templating is not the only responsibility of the builder pattern in the SPA framework, it may also convert the hydrated html templates into elements, attach events to them, and append the elements to the container of the SPA section, thus rendering the view. Another responsibility of the Director-Builder component is to construct an in-memory representation of the view, as discussed in the next section.

## Pattern: Composite

The key feature of the Composite pattern is that it allows clients to ignore the difference between individual components and compositions. In statically typed languages this is achieved by creating objects that implement both composite and component interfaces. Composites are tree-like structures, it is not clear from the discussion of the pattern in GoF if a composite may allow multiple root level nodes. Because a Composite and Component implement the same interface and operations, in a typical application of the composite pattern, invoking an operation on a non-leaf component invokes the same operation on all its children as well. Thus, Composite pattern allows invoking a command on a group of related objects recursively.

The Composite pattern is often described in the context of a GUI. In fact, graphical user interfaces are one of the primary reasons for the proliferation of object oriented frameworks (Fayad & Schmidt, 1997). The GoF discusses the case study of a text editor, the graphical primitives for which are represented in the form of a composite. Any GUI is primarily composed of graphical primitives such as lines, curves, text & images and they are composed into containers such as rows, columns, tabs and windows. Complex GUI applications such as a text editor or a drawing application must have an object structure to represent the graphical objects in memory while it is being edited. The GUI of a web page is declarative, comprising of HTML elements and CSS styles. Before the arrival of the HTML5, tags and css were the only GUI related primitives in a web page.

### Representing an SPA view in-memory

We need the GUI composite because before a view can be rendered on a browser, it needs to be in HTML form, which is text based. HTML by itself does not allow dynamic updates, which is the reason JavaScript exists in the first place. A dynamic single page application requires a view, the components of which can be manipulated in memory and converted to text before being rendered to the browser. To

provide a rich user interface, CSS classes and events need to be applied to html elements dynamically. It would be a bad design choice to consider doing this by editing the text of the HTML itself. It would lead to cluttered markup and poor performance.

HTML elements in JavaScript are represented in memory as Nodes, as part of its Document Object Model (DOM). JavaScript's DOM API allows dynamic addition, removal and manipulation of HTML elements, which is reflected on the GUI. Aside from the visual manipulation of GUI elements, Nodes are also used to attach and detach event handlers to them at run-time, apply and remove CSS classes. By setting the *contenteditable* property on an HTML element, we can make the content inside that element editable by the user. This is the basis for most browser based text editors, WYSIWYG editors and wikis.

Code listing 14 shows the implementation of the *ViewPart* class which is an application of the Composite pattern. There are no Composite or Component interfaces, the same object is both the composite and the component. There is no distinction between leaf and non-leaf properties. The *ViewPart* encapsulates an HTML element, and provides additional fields and methods to support child components and parent references. Finally, there's the semi-abstract render method that simply renders the elements to the view recursively. The render method can be customized and extended by subclasses and decorators to achieve more interesting behaviors.

```
ViewPart = (function () {
    function ViewPart(element) {
        this._parent = null;
        this._element = element;
        this._children = [];
    }
    function _getElement() {
        return this._element;
    }
    function _setParent(parent) {
        this._parent = parent;
    }
    function _add(part) {
        let instance = this;
        part.setParent(instance);
        this._children.push(part);
        return part;
    }
    function ViewPart_render()...
    ViewPart.prototype.setParent = _setParent;
    ViewPart.prototype.add = _add;
    ViewPart.prototype.getElement = _getElement;
    ViewPart.prototype.render = _render;
    return ViewPart;
})();
```

*Code Listing 14 The Composite Pattern - ViewPart class*

One could question the need for an in-memory representation of the view, as the builder pattern implemented in the last section is completely capable of attaching dynamic events and properties to

elements as well as rendering them. However not all updates to the view of an SPA application require fresh data and a new template. For example, a web form may need to be validated before its data is posted to the server. If the validation fails, the view needs to be updated with visual cues about errors, and tooltips to be shown with helpful information.

The Gang of Four suggest that the builder and composite patterns are related, and that builders are often used to build composites. That is precisely how the two patterns are used in this SPA framework specific implementation. Code listing 15 shows how the builder pattern is being used to construct the view for the application.

```
Block.prototype.buildView = function (data, template, iterator, builder) {
    return new ModelDirector(data, new TemplateBuilder(new ViewPart(this._container), template, builder)).construct(iterator);
}
```

*Code Listing 15 Builder Pattern for building a Composite view*

The *ViewPart* is a central component of the view engine of the SPA framework. It encapsulates the entire view of an independently updateable SPA section. All dynamic functionality and user interactions on will involve an instance of this object.

## Pattern: Iterator

Most object-oriented programming frameworks provide their own collection classes and iterators to navigate them. Such iterators are typically used to iterate through collections that are sequential. The design significance of iterators becomes apparent when non-sequential collections need to be iterated upon, such as tree and graph based data structures. Native implementations of such structures in languages and frameworks is rare. A system may need more than one way traversing an aggregate structure, for example pre-order and in-order traversal of trees. Even though a programmer may not anticipate an immediate need for multiple traversal methods, it makes design sense to separate the iteration/traversal mechanism from a complex collection object that has other responsibilities to ensure separating the two concerns.

The Gang of Four describe the concept "polymorphic iteration" to decouple an iterator from the object it traverses, which is based creating the *Iterator* and *Iterable* abstractions and their parallel class hierarchies. But as we have already seen, the existence of object hierarchies is not a prerequisite for the application of design patterns, though their often design significance becomes more apparent when they do.

### Iterating the composite view

The composite "View Model" developed in the previous section seems an useful design choice, but it doesn't do anything so far besides producing a string representation of itself. Subsequent discussions will demonstrate that the composite is a crucial part of the View component of this framework. It needs to support a lot more functionality and we should be sure we do not end up with a bloated structure as we continue adding features to it. The *toString* method of the composite iterates the composite recursively. As we add more methods to it, variations of that recursive logic will need to be repeated, thus violating the DRY programming principle. Also, this does not allow external classes to iterate over

the *ViewPart* components, the recursive Iteration is owned by the composite object itself. The purpose of the iterator pattern is external the traversal of an object, allowing its clients to iterate over it.

The composite view is not a sequential data structure, it is a tree and therefore, requires tree traversal methods to iterate through it. It is central to multiple aspects in the View module of the SPA framework, and will be accessed and manipulated by other components. The framework needs a way to sort and filter the parts that comprise the view. Recursive traversal may be appropriate for getting a text representation but may not be sufficient in other cases. There needs to be a way to address the competing concerns without making too many changes to the view composites. We need to separate the traversal logic for the composite into a separate module, so that they can vary independently.

A tree may be traversed in a couple of ways, breadth first or depth first. Breadth first traversal is easier to implement as it traverses a tree one level at a time, although it may require more memory. A breadth first iterator needs to maintain references to each child array found at each level of the composite. Code listing 16 shows a more involved implementation of an iterator that traverses the *ViewPart* composite depth-first, simulating recursion. It is implemented by using a JavaScript array as a stack, the API for which is built into the language itself.

```
Iterator = (function () {
    function Iterator(part) {
        this._stack = [];
        this._current = null;
        that = this;
        process(part, that);
        this._stack.reverse();
    }
    function process(part, instance) {
        part.getChildren().forEach(function (child) {
            process(child, instance);
        });
        instance._stack.push(part);
    }
    function _current() {
        return this._current;
    }
    function _next() {
        this._current = this._stack.pop()
        return this._current;
    }
    function _isComplete() {
        return this._stack.length == 0;
    }
    Iterator.prototype.current = _current;
    Iterator.prototype.next = _next;
    Iterator.prototype.isComplete = _isComplete;
    return Iterator;
})();
```

*Code Listing 16 Depth first iterator for ViewPart*

The latest standard for JavaScript provides some language support for creating iterators. For an object to be iterable, it must implement an iterator function that uses the yield keyword to return the next object in the sequence. The use of native JavaScript iterators is not appropriate to traverse a tree based structure however.

## Pattern: Decorator

The Decorator pattern allows adding functionality to an object dynamically. It is useful when we want to add responsibilities to an individual object but not to other objects of the same type. Another use case for the decorator pattern is to assign behavior to objects that may later be withdrawn. In class based languages, the decorator pattern provides an alternative to sub classing to extend the behavior or objects. Decorator is an example of dynamic inheritance, in contrast to static inheritance achieved by creating subclasses. The standard implementation of the decorator pattern involves a class to inherit from another base class as well as contain an instance of it.

The key aspect of the decorator pattern is that it enables inheritance through composition. Figure 9 shows the structure of an implementation of the decorator pattern. It illustrates how a visual component that may be adorned with a border and a scrolling functionality.

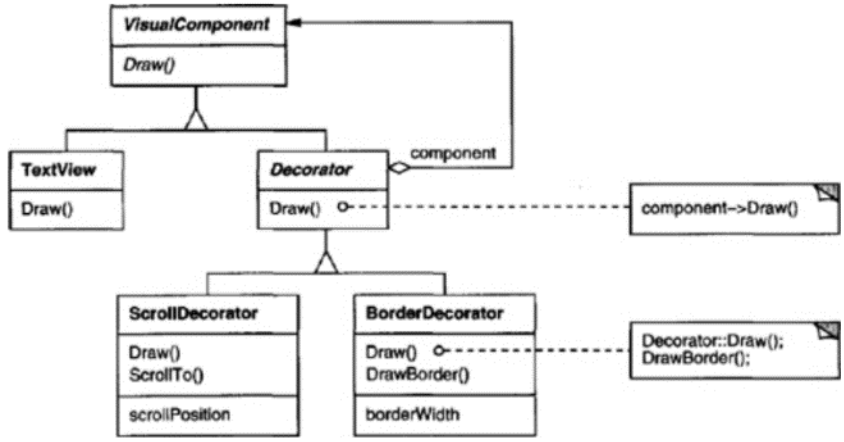



*Figure 9 Decorator pattern example (Helm, Johnson, Vlissides, & Gamma, Structural Patterns, 1994)*

The decorator has an instance of the component as one of its fields. The *BorderDecorator* works by first calling the *Draw* method on its component and then draws a border on it. The *ScrollDecorator* scrolls to a position on the visual component after drawing it first.

## Form Validation

HTML input elements are used to collect data from the application user. Validation of the data provided by the user is a usual concern of web applications. The HTML input tags provide support for validation without scripting, by setting the required attribute for a mandatory input field and by setting the pattern attribute to validate its text by matching it to a regular expression. The submit event of the form element checks the validity of form elements by invoking the *checkValidity* method on all its input fields. If an input field is invalid, the browser adds some default styles to the element to indicate an error,

usually a red border. Most framework users would seek to override default browser behaviour form validation to provide a more uniform experience across all browsers. As the framework constructs an in-memory representation of the view, it makes sense that the responsibility of validating a form based view be part of the composite itself. Code listing 17 shows the addition a validate function to a *ViewPart* object. The validation augmented object is referred to as *FormViewPart*. The validation works by first checking if the component is a leaf level part of the view, therefore an input field. If it is, the *checkValidity* method is invoked on the enclosed element, otherwise the validation methods all of the components children for validity through recursion.

```
FormViewPart = (function () {
    function FormViewPart(element) {
        let that = new ViewPart(element);
        let instance = that;
        that.validate = function () {
            if (that.getChildren().length > 0) {
                let result = true;
                return instance.getChildren().forEach(function (child) {
                    result = result && child.validate();
                });
            }
            else {
                return instance.getElement().checkValidity();
            }
        }
        return that;
    }
    return FormViewPart;
})();
```

*Code Listing 17 FormViewPart for encapsulating forms*

Now, different types of input fields will indicate invalid input with custom styles. Some input fields may be highlighted with a different background color, others may have colored borders, or by resizing the element. Any combination of the three styles may be used to indicate errors. Figure 10 and code listing 18 illustrate the decorator pattern as implemented for form validation. Two different types of input elements indicate errors in two different ways by composing the same type of FormViewPart object

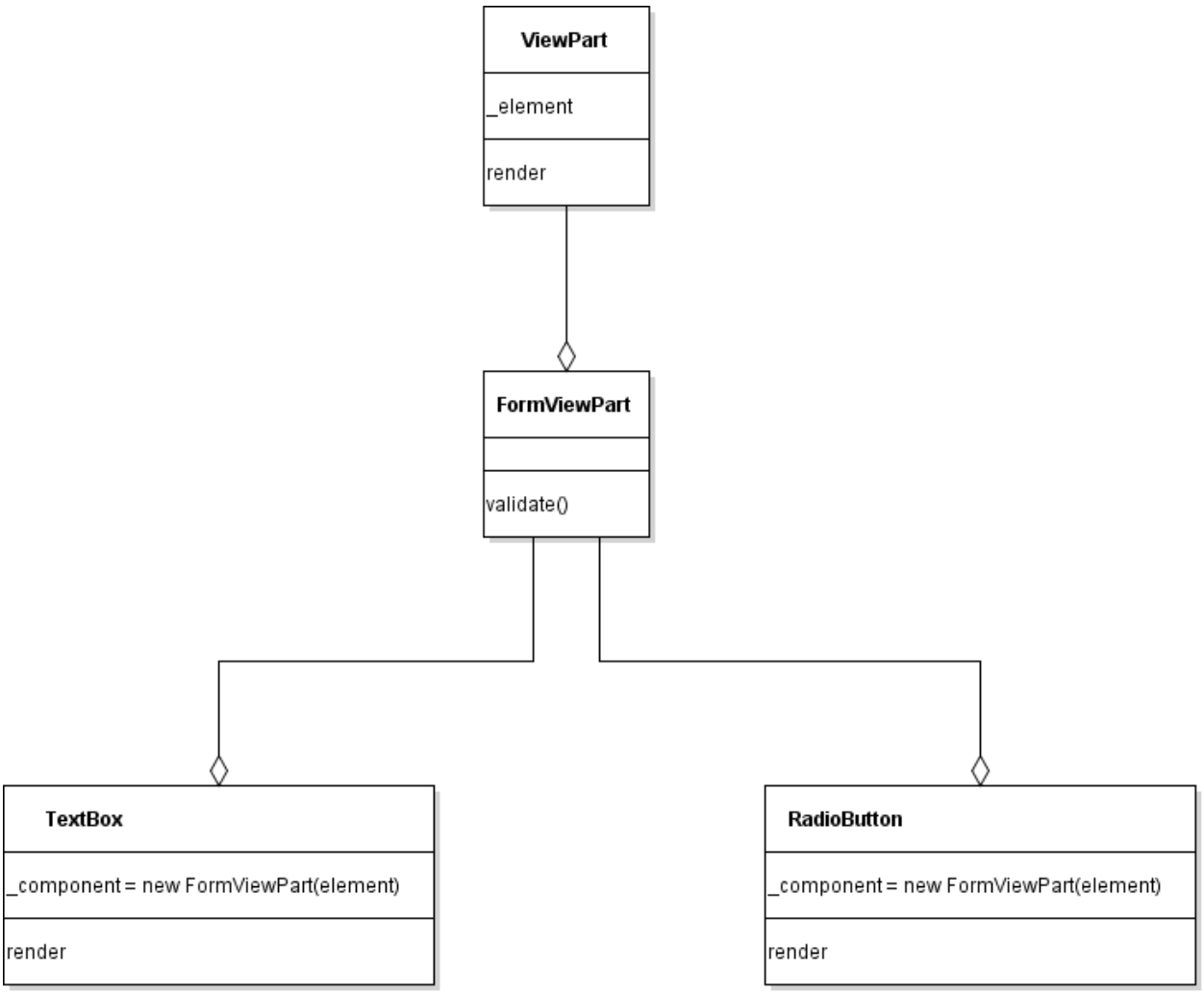



*Figure 10 Form Decorators*

```
TextBox = (function () {
    function TextBox(element) {
        this._component = new FormViewPart(element);
        this.render = function () {
            this._component.render();
            this._component.getNode().setAttribute("type", "text");
            if (!this._component.validate()) {
                this._component.getNode().setAttribute("class", "error");
            }
        }
    }
    return TextBox;
})();
```

*Code Listing 18 TextBox Form Decorator*

The component, a *FormViewPart* object is enclosed within a leaf level form view part. The leaf level form components call the render method of part they enclose, attributes are added to the corresponding

input element to indicate the type of input, and finally the validity is checked to conditionally add styles to indicate an error appropriately.

This implementation of the decorator pattern uses a technique called functional inheritance as opposed to prototypal inheritance described during the discussion of the prototype pattern. Functional inheritance is implemented through object composition, and does not involve a prototype chain. Therefore, functional inheritance is considered preferable to prototypal inheritance because it provides slightly better performance.

An implementation of the decorator pattern is possible even through prototype based inheritance. Code listing 19 shows how a component may be enclosed and extended by a Decorator object. The syntax for doing so is clearly more verbose and difficult to comprehend. This, and the associated performance penalty of looking up the prototype chain make the decorator pattern the preferred way of object inheritance in JavaScript. The rule of thumb is, whenever we need to extend a type of object, use prototype pattern and prototypal inheritance. And if we simply need to extend an object instance, use decorator pattern and functional inheritance.

```
function decorate(component) {
    const proto = Object.getPrototypeOf(component);

    function Decorator(component) {
        this.component = component;
    }

    Decorator.prototype = Object.create(proto);

    //new method
    Decorator.prototype.greetings = function () {
        return 'Hi!';
    };

    //delegated method
    Decorator.prototype.hello = function () {
        return this.component.hello.apply(this.component, arguments);
    };

    return new Decorator(component);
}
```

*Code Listing 19 Decorator with prototype inheritance (Mammino & Casciaro, 2016)*

## Pattern: Chain of Responsibility

The “Chain” in Chain of responsibility pattern refers to a chain of objects refers to a chain of objects. It might be a linear chain like a linked list, or a tree like structure, like a composite. The pattern allows a request to be propagated up or down a chain till one of the part handles it and stops further propagation. This requires the nodes in the chain references to their successor, the next node in the chain that may handle a request if the current one cannot. Chain of responsibility is most often used

along with the Composite pattern. The primary benefit of using the pattern is that it lets components in a Composite data structure to selectively handle events. Even though, the reference to the next item in the chain is termed successor, in case of a composite the successor would be the parent of a node.

Event bubbling is an example of the Chain of Responsibility pattern. An event invoked on a nested element is passed on to its parent if the element itself cannot handle it. In web browsers, event bubbling is implemented such that even if a nested element handles an event, it will be propagated up the chain unless it is explicitly prevented. The prototype chain in the JavaScript language itself is another example of Chain of Responsibility, if a property or method that does not exist in an object is invoked, the JavaScript runtime will continue to look for that member up the object's prototype chain.

GUIs in object oriented frameworks are almost always examples of the Composite pattern. One of the benefits of the Composite is that there is no need for a client to differentiate between a component and its container. A Composite view in the SPA framework, may need the ability to selectively handle an event, for example to provide contextual information about certain elements of the view. Not all components of the view may have contextual information associated with them. The event may be triggered by clicking on an icon rendered as part of a component, in which case the links in the chain that can handle an event can be visually identified. The chain of responsibility in this case will determine the specific handlers before rendering the view. In other cases, the events may be triggered simply by mouse/pointer movement, in which case the event needs to be captured by the element being pointed to, and either handled or propagated upwards, depending on one or more of its properties

## Tooltips for Forms

An HTML Form is a group of different input elements used to gather information from the user, which can then be posted to the server. A form can be created with our existing SPA framework, using a Builder and *ViewPart* composite, support for which has already been added. The *FormViewPart* composite adds support for validating form elements. We now need to embellish our forms with "Tooltip" functionality to provide users with helpful information about certain parts of the form. The Tooltip may appear on an input field or a section of the form containing multiple input fields. The Tooltip text on an input field may be used to show information like password rules or validation messages. Tooltips on sections of the form may display text about the purpose of information collected from the various fields in it. Parts of the Composite object representing a multi-section form may have a tooltip message property associated with them, we need a mechanism to generate tooltips selectively on components that have this property.

The Gang of Four version of the Chain of Responsibility works by setting successor nodes on an object if the current one cannot handle a request. A key aspect of the implementation is the following conditional block which checks the current node for an appropriate help item, and passing on the request to the successor if it doesn't. Listing 20 shows a simple if-else statement that accomplishes this in C++.

```
if (HasHelp()) {
    // offer help on the dialog
} else {
    HelpHandler::HandleHelp();
}
```

*Code Listing 20 Delegating help up the chain (Vlissides, Johnson, Helm, & Gamma, Behavioral Patterns, 1994)*

To implement the chain of responsibility pattern for tooltips, the pattern will make use of the parent references in the *ViewPart* composite. We need to implement a tooltip handler, that takes a *ViewPart* as parameter. The handler is responsible for looking up the tooltip message on the *viewPart* and attach relevant events to it if the property exists, or delegate the handling to its parent.

```
Tooltip = function (viewPart) {
    this._viewPart = viewPart;
    if (this._viewPart.getTooltip() != null) {
        addEvent(this._viewPart.getNode(), 'mouseover', function (e) { startdelay(e) });
        addEvent(this._viewPart.getNode(), 'onfocus', function (e) { startdelay(e) });
        addEvent(this._viewPart.getNode(), 'mouseout', function (e) { hide() });
        addEvent(this._viewPart.getNode(), 'onblur', function (e) { hide() });
    }
    else {
        new Tooltip(this._viewPart.getParent());
    }
};
```

*Code Listing 21 Tooltip Handler for ViewPart using parent references*

An implementation detail to note here is that the tooltip message property and its getter may be added to any component in the composite view, but the Tooltip class only needs to be associated only with leaf components. This makes sense for forms, as in case of validation errors, we would like to indicate the errors in leaf level elements first. This also allows us to selectively ignore branches of the composite that we know in advance will not have associated tooltips, by not associating the Tooltip class with the leaf components of that branch.

## Pattern: Bridge

The intent of the Bridge design pattern is to “decouple an abstraction from its implementation so the two can vary independently” (Vlissides, Johnson, Helm, & Gamma, Behavioral Patterns, 1994). The bridge pattern accomplishes this decoupling by containment and delegation. It is useful when there may be cases of different sets APIs that perform the same service to the application. Gang of Four discuss the motivation for the Bridge pattern through the example of a multi-platform application that needs to abstract away the details of the windowing systems of different platforms. Since the native APIs for managing windows and other GUI components for different operating systems are likely to be very different, it would be difficult to find commonalities between them to represent them with the same abstract class or interface.

The Bridge pattern decouples a component in an application or framework from the specific details of the platforms that it needs to support. This implies “parallel” objects, where framework specific

functionality in one of the objects is implemented in terms of platform specific functionality in another. This effectively decouples an interface (the framework) from its implementation (the platform).

## Bridging controller actions to browser events

The bridge pattern is especially relevant to applications the need to support GUI events. GUI based application frameworks will need to support GUI related objects and view components. Some of the previous pattern discussions in this text have already described the design and construction of some such objects. The *ViewPart* component represent the GUI of the Blocks application framework in memory. The frameworks GUI components will need to be coupled with browser specific APIs for triggering events by capturing mouse clicks or key strokes.

For creating a custom object structure that abstracts away the details of the browser, it makes sense to implement the framework's own API for event handling. Any event handling code for managing interactions with the *ViewPart* needs to be abstracted away from browser specific event handling. GUI objects are designed with framework specific needs in mind and as such need to be decoupled from platform specific details, even though an SPA framework will certainly need to run on a browser engine. It is still a good design decision that some components of the framework are kept independent from the browser. Even though we may not expect to port the application to other non-browser platforms, it still makes sense to shield the framework user from browser specific details.

Listing 22 is an example of the Bridge pattern being used to encapsulate the API to attach an event to an HTML node, with the *BrowserEventDispatcher* as the platform specific implementation. The framework users however only need to work with the *ViewPartEventDispatcher* which contains an instance of the platform specific implementation and invokes the relevant API when requested.

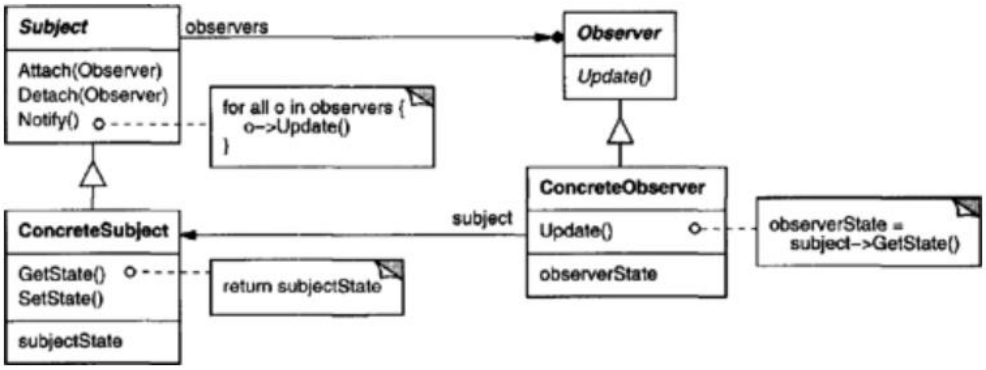
```
ViewPartEventDispatcher = (function () {
    function ViewPartEventDispatcher(context, viewPart) {
        this._context = context;
        this._browserEventDispatcher = new BrowserEventDispatcher(viewPart.getNode());
    }
    ViewPartEventDispatcher.prototype.attachEvent = function (eventType, handler, data, context) {
        this._browserEventDispatcher.addEvent(eventType, handler, data, this._context);
    }
    return ViewPartEventDispatcher;
})();

BrowserEventDispatcher = (function () {
    function BrowserEventDispatcher(node) {
        this._node = node;
    }
    BrowserEventDispatcher.prototype.addEvent = function(eventType, handler, data, context){
        _node.addEventListener(eventType, function (e) {
            e.preventDefault();
            handler(data, context);
        });
    }
    return BrowserEventDispatcher;
})();
```

*Code Listing 22 Bridging the event API*

A key detail of GUI applications based on the Model View Controller architectural framework is that user interactions on the view such as navigation and form submission are routed to the controller to be handled. This is on contrast to how user interactions are handled by the browser itself, by attaching event handlers to DOM elements. Thus, the framework user handles interactions with its GUI objects by routing them to controller actions, while unaware of the browser specific plumbing happening “under the hood”.

## Pattern: Observer

The observer pattern, also sometimes referred to as “publish-subscribe”, is a way to decouple modules when there may cases of one to many dependencies between them. Many application frameworks that include a GUI toolkit also provide an API for handling events. This is the most common application of the Observer pattern, allowing programmers to write event handlers to respond to events dispatched by GUI objects, thus allowing any application specific logic contained in the handler to be decoupled from the GUI. Figure 23 shows the structure of a typical application of the Observer pattern




*Code Listing 23 Observer Pattern (Helm, Johnson, Vlissides, & Gamma, Behavioral Patterns, 1994)*

## Observing an SPA module

Single page applications provide a richer interaction model than traditional web applications. A typical single page application allows a number of sections to display views that can be rendered and updated independently from the rest of the page. Each of these independent sections can be treated as separate modules, where each module can have its own data model and is responsible for rendering the view and handling user interactions on it. Each module is an object instance that responds to the events in its own view, dispatching the requests to its own controller which modifies its own data model before re-rendering the view. However, in some cases two module instances may share a data model, when updates to one section of the page requires an update to another. This is similar to the use case described in Gang of Four, where a bar chart, a pie graph must respond to changes in a spreadsheet displayed in the same window as the graphs.

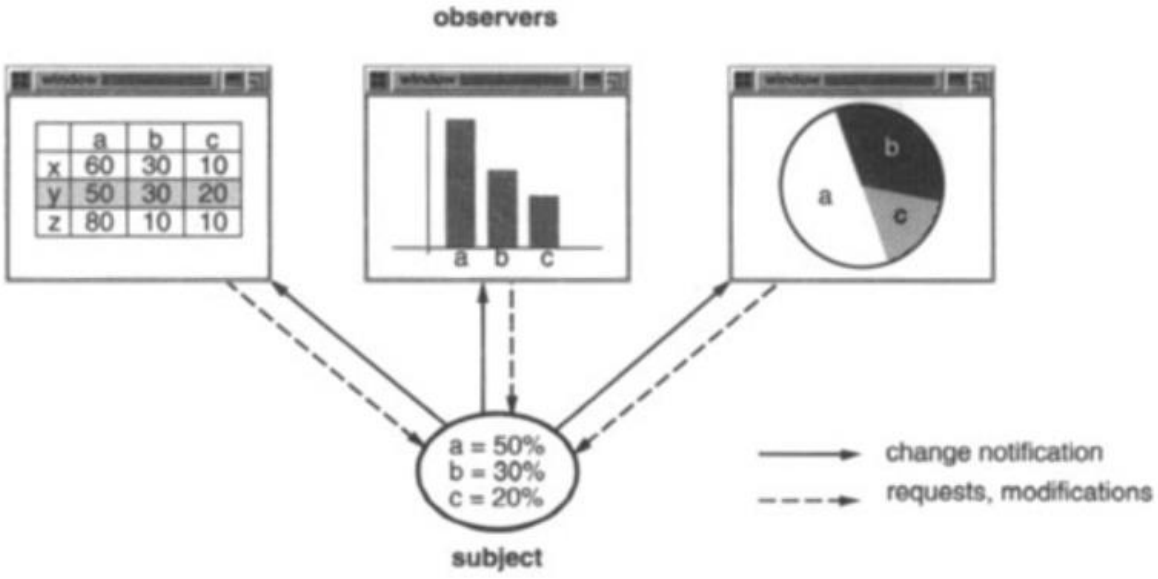



*Figure 11 Observer pattern for MVC (Vlissides, Johnson, Helm, & Gamma, Behavioral Patterns, 1994)*

Communicating such changes would be trivial if the same module handled all three view components, which is impossible to do with the architecture our SPA framework. Each Block component is associated with each independently updateable section of the view.

The first issue we need to address is that any Block instance may act as an Observer and an Observable. Code listing 12 illustrates a Block class implementing both Observer and Observable interfaces, to establish two-way communication between them.

```
Block = (function () {
    function Block()...
    Block.prototype.init = function (container)...
    Block.prototype.buildView = function (data, template, iterator, builder)...
    Block.prototype.update = function (data) { }
    Block.prototype.notify = function () { }
    Block.prototype.attach = function (observer) {
        this._observers.push(observer);
    }
    Block.prototype.detach = function (observer) {
        //find and detach observer, reset array
    }
    return Block;
})();
```

*Figure 12 Observer Pattern applied on the Block module*

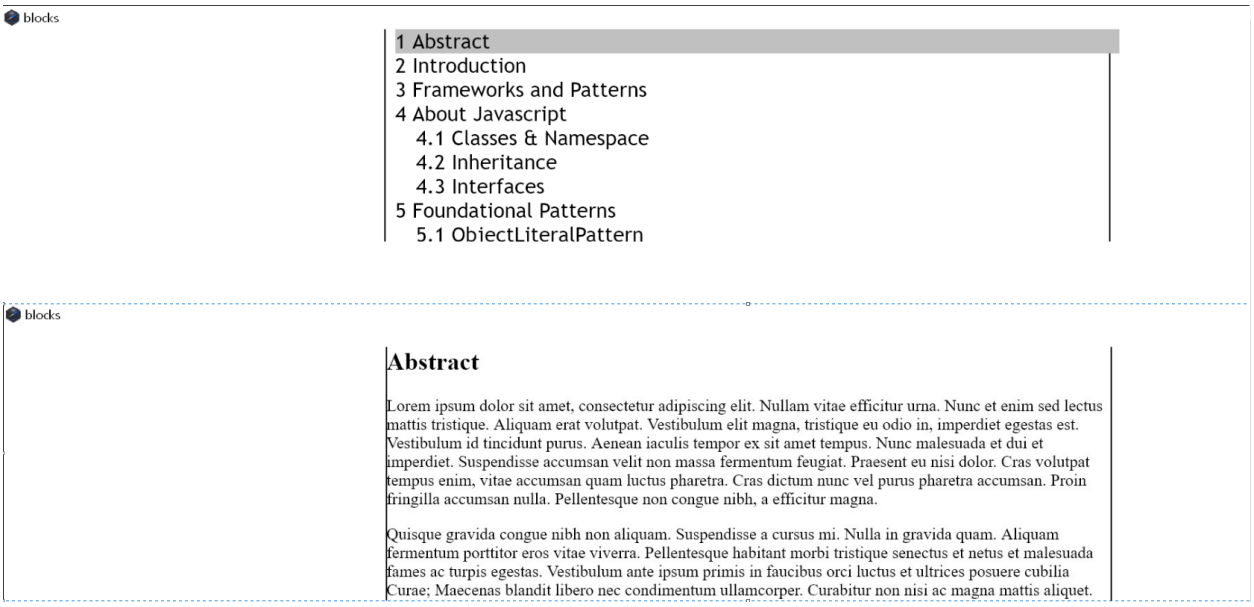
The key here is that the notify and update functions are not tied to the specifics of any Block instance. This is easy to accomplish due to the dynamic nature of JavaScript. Even if the Block class provides a default implementation for the notify and update functions, they can always be redefined overridden by any instance of it. Listing 13 shows how:

```
let block1 = new Blocks.createBlock("block1");
let block2 = new Blocks.createBlock("block2");
block1.notify = function () {
    let data = "Notification from " + this._privateData;
    this._observers.forEach(function (observer) {
        observer.update(data);
    });
}
block2.update = function (data) {
    console.log(data); //logs "Notification from block1"
}
block1.attach(block2);
block1.notify();
```

*Code Listing 24 Block Observer example*

This allows the publish subscribe mechanism to be completely independent of the specifics of an observable instance, or the type of data an instance may update its observers with. Each observer instance may also define completely different functions deal with the data received from the observable instance. To solve the problem of creating a pager module, the notify function on the main module must query its private data to find the currently loaded article, find the previous and next articles from its model and update the pager module with this data. The notify function must be triggered whenever an article loads in the main module, and the pager module would be updated accordingly.

Revisiting the example of the article “index” demonstrated in the discussion of the builder pattern. The index itself is contained within a single Block instance, and clicking on an index item renders the article within the same container as the index. But once an article is rendered, the GUI does not yet provide us with a way navigate any further, either by returning to the index or providing clickable links to navigate to the previous or next article. It is worth pointing out that the article itself is requested from the server in HTML form, not JSON, and therefore it does not require a template, or a custom view to be built for it.




*Code Listing 25 Article Browser without Paging*

We need another section in the page to contain navigation links to previous and next articles from the index, along with a link to navigate back to the index itself. The section must also allow updating the link targets for previous and next articles each time the user navigates to a new one. Also, this section must be hidden when the complete index is displayed in the main section. Thus, along with a main module, we need another "pager" module to provide navigational functionality for our application, and we need our framework to support event-driven communication between them. As mentioned earlier, each module in our SPA framework is encapsulated in a class named "Block". Each instance of Block is associated with an HTML element which acts as its container. The framework needs to support communication of private data between these instances on demand. Listing 26 and Figure 13 illustrate how we establish this communication while keeping the two Block instances completely independent using the Observer pattern. The notify method of the "main" Block is invoked whenever an item is clicked on the index, with the urls of the previous and next articles to the one requested. The pager module then starts itself with the paging functionality to navigate the articles.

```
main = new Block("main");
pager = new Block("pager");
main.buildView(indexModel, "indexTemplate", RecursiveIterator, RecursiveBuilder);
main.attach(pager);
main.notify = function (href) {
    let previous = this.model.previous(href);
    let next = this.model.next(href)
    this._observers.forEach(function (observer) {
        observer.notify({ "previous": previous, "next": next});
    });
}
pager.update = function (data) {
    pager.buildView(data, "pagerTemplate", SimpleIterator, SimpleBuilder);
}
```

*Code Listing 26 Observer Pattern to implement paging*

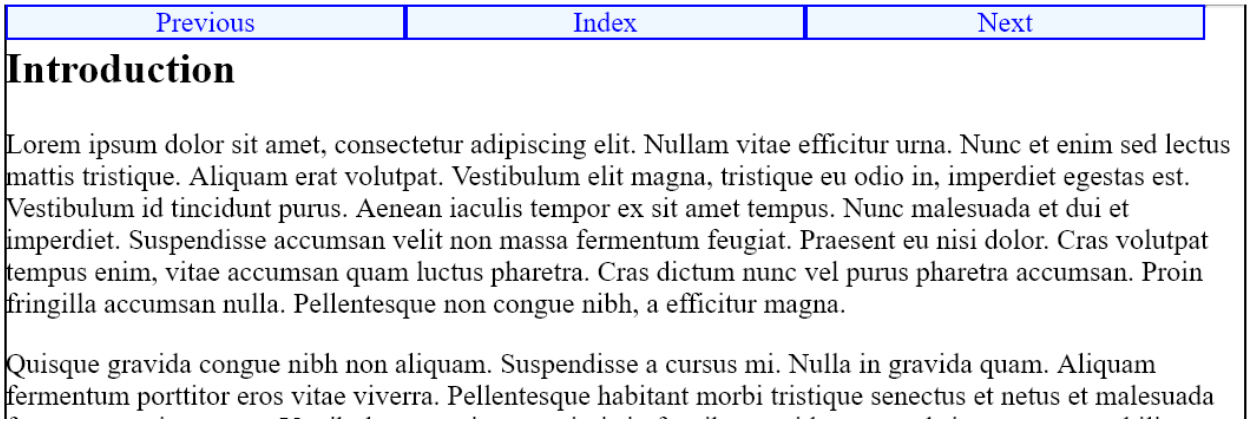



*Figure 13 Article Browser with paging*

It could be argued that the paging functionality belongs inside the same SPA module as the one that contains the article and the index. Even so, this implementation of the Observer pattern illustrates how

two separate views can share a data model, and how changes to the model through interaction in one view allows another to be updated without introducing any direct coupling between the two.

## Pattern: Adapter

The Adapter pattern, also known as wrapper, reconciles incompatible interfaces. It allows an object to be used by client that is expecting a different interface that the object already supports. The most common use case of applying the adapter pattern is incorporating "off the shelf" third party libraries into an application framework. It is very unlikely that the APIs provided by external libraries will be aligned with the API of an existing framework. To reconcile the APIs, it is usually preferable to adapt the external library to the framework's API rather than the other way around.

### Building a database module

A single page application may require access to a local data store, especially in standalone versions of the application that are expected to work even without networking. Modern browser engines provide APIs that can be used to store data into the browser's local storage. Although this data persists across browser restarts, it is a simple key value store not suitable to store large amounts of data.

Thus, we need to add another module to our SPA module that provides support of database I/O. This is a typical case for using an adapter, as its unlikely that we would implement a non-trivial feature like a database engine from scratch. We want the framework's data access API to be "RESTful", mirroring the API provided by HTTP, in order for the application to work the same in connected and disconnected states. Code listing 27 shows the *mongoose* API for the popular document database engine *mongodb* being adapted to the framework's API. The highlighted section is the adapted interface while the function that encloses it is the adapter.

```
db = (function () {
    function _get(model, id, callback) {
        mongo.repository(model).FindById(id, function (err, entity) {
            callback(entity);
        });
    }
    function _save(model, data)...
    function _remove(model, id)...
    mongo = (function ()...)();
    return {
        get: _get,
        save: _save,
        remove: _remove
    }
})();
```

*Code Listing 27 Adapting mongoose API*

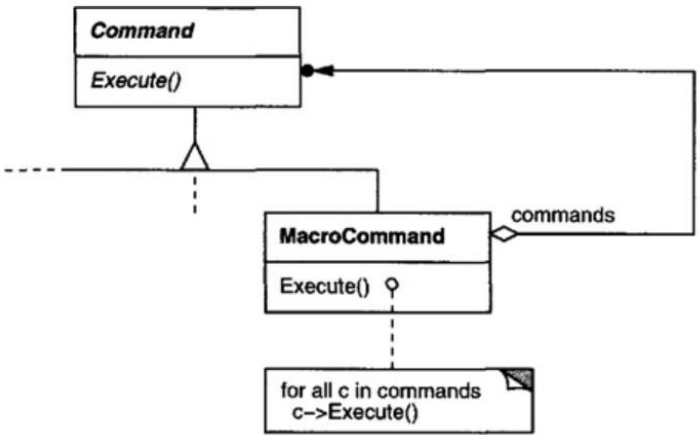
At first glance, this example may look similar to the use case in which the Bridge pattern was applied. However, as the Gang of Four have noted, adapters are significant during the implementation phase, while the bridge pattern is mostly applied during the early design of a framework. Choosing a new

database API is a late design decision. Choosing which platforms to support is an early design decision, while support for additional databases may be added after the framework is implemented.

The database module is also an example of the Repository Pattern (Fowler, 2002). The repository allows communication with the data access layer by querying business entities, instead of querying the persistent data store directly.

## Pattern: Command

The intent of the Command pattern is to support queueable requests and undoable operations. This is accomplished by treating requests and operations as objects that can be stored and enqueued in relevant data structures. Standard command pattern structure requires the queueable requests to encapsulated in an implementation of the Command interface.




*Code Listing 28 Command Pattern Structure (Vlissides, Johnson, Helm, & Gamma, Behavioral Patterns, 1994)*

Figure 28 illustrates the use of the Command pattern for queuing requests. The Command interface specifies a single method *Execute.* The *MacroCommand* class stores a sequence of commands in an iterable collection, and invokes the *Execute* operation of each of them. To support an undoable request, the interface would need to specify an additional method to reverse the effects of any processing the corresponding Execute method may perform.

### Creating a database transaction

Code listing 29 is a simplistic implementation of the Command pattern in JavaScript, to implement transactional database operations that could be rolled back if any of the commands within it fail. A client may run multiple save commands in the context of the *dbTransaction* class, they can be rolled back or finalized at any point by calling the *rollback* or *commit* methods respectively. The *rollback* method works by simply deleting the uncommitted saves from the database. This pattern also illustrates an inventive way to use first class functions in JavaScript

```
dbTransaction = (function () {
    function dbTransaction() {
        this._commands = [];
        this._params = [];
    }
    dbTransaction.prototype.save = function (model, data, callback) {
        let id = data.Id;
        this._commands.push(db.remove);
        this._params.push(id);
        db.save(model, data, callback);
    }
    dbTransaction.prototype.commit = function () {
        //clear the commands array
    }
    dbTransaction.prototype.rollback = function () {
        let instance = this;
        for (let i = 0; i < this._commands.length; i++) {
            this._commands[i](this._params[i]);
        }
        this.commit();
    }
    return dbTransaction
})();
```

*Code Listing 29 Command Pattern to implement a database transaction*

## Pattern: Singleton

A Singleton pattern is implemented when there needs to be a single instance of a particular class throughout the application, all references to a Singleton must point to the same object in memory. A Singleton class therefore creates (the first time it's called) and maintains an instance of itself and returns it to all clients.

Even though Singleton is one of the most commonly used patterns, its usefulness is frequently debated in the programming community. Some programmers consider the existence of Singletons in an application a design flaw, citing global state and testability as some of the arguments against their use.

Regardless of its drawbacks, the Singleton pattern a pragmatic solution in certain situations. Due to the classless nature of JavaScript, Singletons are far more common in practice than in other statically typed languages. Also, the distinction between static classes and Singletons that applies to a lot of statically typed languages does not hold in JavaScript.

We may revisit the Object Literal and the Module patterns in JavaScript to find that they do not require the 'new' keyword for object creation. In both cases, only a single object is ever instantiated, a reference to which is held in the variable used when creating the Object Literal or Module. This is easily demonstrated by the following code segment.

As a relevant example of a Singleton in JavaScript, we may revisit the XHRFactory created for the Factory method discussion. A single page application needs to support multiple asynchronous http requests concurrently, so that multiple parts of the page may request updates at the same time. Thus, it needs multiple distinct copies of the XmlHttpRequest object. There is no apparent drawback however, to

making the XHRFactory itself a Singleton. The following code shows the XHRFactory as a singleton module.

It is worth noting at this point most statically typed languages make a distinction between a static class and a singleton. The difference being that singletons can implement interfaces, can be sub-classed and can be passed around as parameters. These differences are not relevant in JavaScript as it has no classes or interfaces, and singleton Object Literals and modules can be passed as function parameters.

Finally, a true Singleton shouldn't allow cloning However there is no way to prevent cloning of objects in JavaScript preventing it in JavaScript. Therefore, truly limiting a class to a single instance isn't possible in JavaScript. An object may be sealed from extension but this does not prevent it from being used as a prototype for another object.

The Singleton design pattern, as described in the GoF book, is significant in JavaScript only when more control over access to the single instance is required or a class needs to permit a limited number of instances of itself instead of just one.

## A database singleton

One use case for a traditional singleton in JavaScript is when we would like to defer the initialization of a module that is expensive to create. As mentioned in the discussion of the Façade pattern, modules are usually nested with other modules that are its subsystems. Most modules written for the SPA framework are based on the JavaScript module pattern that uses an immediately invoked function expression for initialization. It is not required for a façade object to encapsulate its components, it simply needs to combine their APIs in a manner that makes sense for the application. If we do choose to encapsulate the components, as we have for this SPA framework, there may be an associated initialization cost. A façade that exposes all its sub components in their entirety is not really a façade. But this is precisely what we must do with our top-level module, “Block”.

The Blocks framework itself is a module, and it has a number of subsystems to manage views, databases, files, logging and ajax with and an open-ended number of new modules may need to be added in the future. Thus, initializing Blocks is potentially a very expensive operation. This is a valid use case for the application of the Singleton pattern. The actual initialization of an expensive to create module may hide behind a method that is called when the functionality of the module is actually needed. Listing 30 illustrates deferring the initialization of the “db” module by applying the Singleton pattern.

```
db = (function () {
    var instance;
    function init() {
        function _get(model, id, callback)...
        function _save(model, data, callback)...
        function _remove(model, id, callback)...
        mongo = (function ()...)();
        return {
            get: _get,
            save: _save,
            remove: _remove
        }
    }

    return {
        getInstance: function () {
            if (!instance) {
                instance = init();
            }
            return instance;
        }
    };
})();
```

*Code Listing 30 Singleton pattern applied to the database module*

Thus, the singleton pattern may be used to defer the initialization of a module that requires expensive initialization. The database module would involve establishing a connection to a database. Even though connection is an asynchronous operation, it would still require some I/O to be performed and take up CPU cycles. By deferring the initialization of the module, the connection is established the first time the db module is used. Clients that need to access the db functionality now must call the *getInstance* method on the db singleton to access its functionality.

# Discussion

## Summary

The dissertation narrative detailed the process of developing a single page application framework from the ground up. We incrementally designed and prototyped a framework that provides support for a range of concerns of typical rich internet applications, like ajax, templating and database I/O. Each feature or component added to the framework was discussed in terms of the design patterns that supported it. The synergy between design patterns and frameworks was explored as we designed the framework. Several components involved a collaborative application of multiple design patterns.

The pattern-oriented approach discussed design patterns in terms of the "real-world" problems that they solve, and provided some insights into what these solutions would look like in a dynamic, non-traditional object-oriented programming language like JavaScript. !6 out of the 23 Gang of Four patterns were discussed in detail with brief nods to the overall architecture of the framework, Model View Controller, and another architectural pattern specific to data access APIs, the Repository pattern.

The prototype pattern described how object types may be inherited and extended in JavaScript, the prototype property associated with each object, the prototype chain and prototypal inheritance. The pattern also indicated the demerits of prototypes and their performance impact on deep inheritance hierarchies. The factory method, another creational pattern was used to demonstrate the concept of implied interfaces and duck typing, and how distinctly unrelated objects may be used in the same context if they have the same semantics. Proper use of JavaScript modules and techniques to control access to them were discussed through the application of Proxy and Façade patterns. The Façade pattern also demonstrated a higher-level module simplifying access to the functionality of it's sub modules, an approach that is used to create a layered architecture in a language that does not support interfaces.

The template method pattern showed how the functional strengths of JavaScript can be used to implement an object-oriented pattern in multiple non-standard ways. The related discussion showed how a polymorphic algorithm is easier to implement in a dynamic, functional language than a typical object oriented one. Template method and builder patterns both used functional polymorphism to extend classes rather that inheritance or composition.

The substantial part of the pattern narrative was dedicated to building an appropriate GUI object model and handle other View related concerns of the Model View Controller architecture. The flyweight pattern encapsulated the GUI primitives of a web application, tags and templates. The builder pattern implementation was central to the templating needs of the framework, providing two distinct points of customizing templating behavior, one for processing the data model and the other for view generation.

The GUI object model and behavior was built along with a contextual discussion of Composite, Iterator, Decorator and Chain of Responsibility patterns. The *ViewPart* class is the central GUI construct in the framework, that represents and composes the entire GUI of a view in the single page application. The *VIewPart* and related classes effectively demonstrated the concepts of encapsulation and dynamic dispatch in JavaScript. The *ViewPart* class encapsulates GUI primitives as well as entire GUIs using the

Composite pattern. The iterator pattern externalized view traversal responsibilities. The Decorator pattern was used to create the first inheritance hierarchy in the SPA framework, through object composition. The Chain of Responsibility demonstrated an upwards traversal of the *ViewPart* composite. The Bridge pattern was then used to abstract the ViewParts from browser specific event handling concerns, bringing the framework closer to a fully realized MVC architecture.

The pattern narrative concluded with prototyping a database module for the framework. The adapter pattern was exemplified with a typical use case of adapting third party libraries. The Command Pattern employed the use of first class functions in JavaScript to create and database transaction that can contain multiple commands, and allows rolling them back in case any of the queued commands fail.

## Critique

It is a common scenario for programmers to work on extending software that has already been partially or wholly designed. There has been a lot of literature supporting the value of describing the core components and building blocks of the software design in terms of patterns that they are composed of. This familiarizes experienced programmers with the core architecture of the system or application that they must extend. Appropriate use of design patterns allows programmers to use their own knowledge of them to easily spot the points of extension where polymorphic behavior may be inserted. All frameworks have an associated learning curve, and it is the application programmer's burden to understand the overall design and architecture of the framework, above and beyond the API it exposes, to be able to properly use and customize it to fulfill the requirements of their own applications. A self-documenting, pattern based design helps programmers understand the internals of the framework without having to go through framework and API documentation in detail.

Partially due to the proliferation of frameworks, object-oriented or otherwise, the significance of documenting design in terms of patterns has diminished somewhat. All major programming languages are accompanied by their own software development kits and frameworks for application development. Whenever a language or technology gains widespread platform support, several application frameworks emerge into the market in a relatively brief time period. It could be said the frameworks have abstracted away the concept of design patterns allowing programmers to focus on domain specific issues instead.

Even though JavaScript is not a new programming language, its adoption for multi-platform application development has been recent. Still, there are several JavaScript frameworks already in circulation to assist programmers to build applications with it. JavaScript supports the object-oriented programming paradigm, but a survey of open source JavaScript frameworks will reveal that none of them are strictly object oriented. This is an indication that JavaScript is more suited to functional programming, than to produce purely object-oriented designs.

Thus, going against the current industry trends to some extent, this dissertation attempted to design and partially implement a purely object-oriented design by applying design patterns that have been widely accepted to be the foundations of object-oriented programming. The design and implementation of the framework was accompanied by a literature review of object oriented programming and design patterns in JavaScript. The findings of the literatures review and the framework development exercise

indicate that object-oriented programming in JavaScript requires a lot of additional work and improvisation due to lack of classes and other OO constructs that are found in other languages.

Arguably, any of the design problems discussed in the pattern narrative above can be solved entirely using functional JavaScript code without the need to create complex object based structures like the *ViewPart*. Also, the functional nature of JavaScript, allowed application of certain patterns in unique ways that would not be possible in programming languages like C++ and Java. The existence of first class functions gives the programmer the freedom to solve the design problems addressed by the template-method and command patterns without creating new objects or trying to coerce functions to look like objects. A family of builder objects for converting a source representation into different target representations can be accomplished entirely by appropriately using functors (Khot, 2015). The Singleton pattern was perhaps the weakest example on OO design practice applied to JavaScript. The problem that it solved was not very significant in the first place.

The reader may have noted that that we have often pointed out that JavaScript is classless, and Inheritance in JavaScript happens at object level. Yet during the pattern narrative we used the term "class" to refer to structures used in multiple pattern implementations. This contradiction can be resolved by considering the following statement. A function declaration in JavaScript can be considered a class, an invoked function like an IIFE, is an object. So why would we use classes at all in an object-based language? Because often we would like multiple objects to adhere to a certain "type", a common ancestor, and we want to be able to initialize that by passing it some parameters.

Many design patterns require the use of class and interface inheritance. Inheritance is uniquely different in JavaScript. In fact, JavaScript does not support the traditional concept of object oriented inheritance at all. "Prefer composition over inheritance" is a common object-oriented programming idiom, but JavaScript supports inheritance *through* composition. Large inheritance hierarchies are generally not a recommended design practice, but it is an especially bad idea in JavaScript due to its underlying prototypal nature. If hierarchy must exist, it is preferable to have it deep rather than wide (Johnson & Foote, 1988). Creating deep inheritance hierarchies by extending the prototype chain has a negative performance impact, as the JavaScript runtime must look up an invoked property or member up the prototype chain one level at a time. Attempting to access properties that do not exist will traverse the complete prototype chain every time.

The *ViewPart* Composite and related patterns were used in contexts very similar to how other GUI application frameworks may use them. The GUI is often represented as a Composite in other frameworks as well. Iterators are used to traverse the view structure, Builders are used to construct the composite and Decorators are used to add "one-off" responsibilities to various ViewParts. The Chain of Responsibility is also often used to handle event cascading and bubbling. A relatively small construct like the *ViewPart* class being used to encapsulate all the GUI elements can be considered a good example of a reusable design.

This leads to another relevant finding of the research project, that traditional object-oriented design patterns could be more effective in documenting and communicating the design of a framework, to programmers experienced in object oriented design, who would arguably find functional designs

difficult to comprehend. While the motivations for applying the patterns in object oriented JavaScript are largely the same as any other language, the pattern implementations do not strictly adhere to the class based structures described in pattern related literature.

Some design patterns are very typical in their contexts, for example the use of State pattern for handling the state of a database connection. It could be difficult to find such use cases in JavaScript due to it being completely asynchronous. In general, due to its object based, prototypal nature, design patterns that are based on object composition, such as Proxy and Decorator are more relevant to JavaScript than patterns that rely on inheritance, such as Bridge and Builder. Patterns that are intended to support parallel class hierarchies are almost completely irrelevant in JavaScript which allows objects to be created from prototypes or ex-nihilo.

The discussion of the pattern-based approach may conclude by making the following point. Using JavaScript for implementing object oriented application frameworks is neither impractical nor unrealistic. GUI related abstractions have been successfully applied to many existing and obsolete application frameworks in other languages, and the same concepts can be applied to JavaScript with equal effect. If an object-oriented approach is being used for architecting a JavaScript framework, it is preferable that it be pattern oriented as well, because the absence of design patterns is a Big Ball of Mud (Foote & Yoder, 2000), a design without structure, very difficult to manage and extend. As it sees increased adoption for development of native applications, domain relevant object-oriented application frameworks might well be the way forward for JavaScript in the future.

# Conclusions

Proper use of design patterns to develop applications and frameworks makes their design easier to communicate and understand. This is true for frameworks written in most languages and equally so in JavaScript. Functional programming is not as closely related to the concept of design patterns as object-oriented programming is. The foundation of object-oriented design is mutable state which contrasts with functional programming languages which tend to distance themselves with the ceremony associated with mutable data and consider it a bad “side-effect”. In functional programming, design considerations are limited to creating modules with fixed responsibilities.

Object-oriented programming is a way of thinking about computation. This statement applies to other programming paradigms as well. It takes years for programmers to develop expertise with a programming paradigm. As we have touched upon earlier, JavaScript at its core has more in common with functional programming languages than object-oriented ones. We have also stated that most open source JavaScript projects comprise primarily comprise of functional code.

As part of the research this dissertation covers, we applied object oriented design patterns and principles to a dynamic, functional programming language and prototyped an application framework. This resulted in a type of source code that would be difficult to find in other large-scale JavaScript projects. A majority of Gang of Four patterns found their way into the design into overall architecture of the framework. Even though some patterns have a larger impact on the framework architecture, all patterns form an integral part of the framework, the contexts they are used in are not contrived. This is at least one indication in favor of JavaScript's suitability as a language that can be used for development of complex systems. Some of the patterns may be more significant when working with legacy code, but all of them are important during the initial design of the system.

Lastly, any discussion of good design practices must be put into perspective by evaluating them in the terms of the needs of a software project, and the technology and language that it is based on. Most long-lived software projects involved rotating sets of teams, with team members being replaced and teams being resized depending on prevailing project requirements. A number of projects that have large code bases and several major components, will have entire software teams dedicated to working on a particular part of a system, with little knowledge about other sub-systems. Software teams today have come to accept testability as a prerequisite of good design. It is widely accepted that purely functional code is more testable through unit tests than an object-oriented design, which involves private members and contextual existence of objects. Design Patterns address some if the testability concerns of object oriented design, but unit testing an object-oriented application will always require more effort. The designs proposed in this dissertation do not prohibit this use of functional code, often supporting functional programming techniques where appropriate. If the high-level components of a system must be described in terms of objects, the strengths of one or more functional SPA frameworks could still be utilized by applying an object-oriented veneer over them.

# References


Alexander, C. (1977). *A Pattern Language.* Oxford University Press.

Baron, A. L. (2003). *Design Patterns Formalization.* Technical Report, École des Mines de Nantes.

Beck, K., & Cunningham, W. (1987). *Using Pattern Languages for Object-Oriented Programs.* Oxford University Press.

Buschmann, F., Stal, M., Sommerlad, P., Rohnert, H., & Meunier, R. (1996). *Pattern-Oriented Software Architecture, Volume 1, A System of Patterns.* John Wiley & Sons.

Champeon, S. (2001). *JavaScript: How Did We Get Here?* Retrieved from orielly.com: http://archive.oreilly.com/pub/a/javascript/2001/04/06/js_history.html

Cho, J., & Ryu, S. (2014). JavaScript module system: exploring the design space. *Proceedings of the 13th international conference on Modularity*, 229-240.

Crockford, D. (2008). Analyzing JavaScript. In D. Crockford, *JavaScript, The Good Parts.* O'Reilly Media, Inc.

Douglas C. Schmidt, K. H. (2007). *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages.* John Wiley & Sons.

Douglas C. Schmidt, K. H. (2007). *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, 4th Volume.* John Wiley & Sons.

Fayad, M., & Schmidt, D. C. (1997). Object-Oriented Application Frameworks. *Communications of the ACM, Special Issue on Object-Oriented Application Frameworks, Vol. 40, No. 10*.

Foote, B., & Yoder, J. (2000). "Big Ball of Mud.". In *In Pattern Languages of Program Design IV, eds. Neil Harrison, Brian Foote, and Hans Rohnert. Boston.* Addison-Wesley.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). *Design Patterns: Abstraction and Reuse of Object-Oriented Design.* Department of Computer Science, University of Illinois at Urbana Champaign.

Garrett, J. J. (2005, February 18). *Ajax: A New Approach to Web Applications.* Retrieved from Web archive: https://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php

Gross, C. (2006). The Nuts and Bolts of Ajax. In C. Gross, *Ajax Patterns and Best Practices.* Apress.

Helm, R., Johnson, R., Vlissides, J., & Gamma, E. (1994). Behavioral Patterns. In R. Helm, R. Johnson, J. Vlissides, & E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional.

Helm, R., Johnson, R., Vlissides, J., & Gamma, E. (1994). Structural Patterns. In R. Helm, R. Johnson, J. Vlissides, & E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional.

Jain, P., & Kircher, M. (2004). *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management.* John Wiley & Sons.

Johnson, R. E., & Foote, B. (1988). Designing Reusable Classes. *Journal of Object Oriented Programming, Department of Computer Science, University of Illinois, Urbana-Champaign*.

Khot, A. S. (2015). Functors. In A. S. Khot, *Scala Functional Programming Patterns.* Packt Publishing.

Kirk, D. S. (2005). Understanding Object-Oriented Frameworks, Doctoral Thesis, Unpublised.

Leeuwen, A. (2013). *Implementing a Reusable Design Pattern Library in C#.* MSc Dissertation, Unpublished.

Lott, S. (2015). *Functional Python Programming.* Packt Publishing.

Mammino, L., & Casciaro, M. (2016). Prototype Pattern. In L. Mammino, & M. Casciaro, *Node.js Design Patterns - Second Edition.* Pack Publishing.

MDN. (2017, July 9). *Class-JavaScript.* Retrieved from mozilla.org: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

Murphey, R. (2009, October 15). *Using objects to organize your code.* Retrieved from http://rmurphey.com: http://rmurphey.com/blog/2009/10/15/using-objects-to-organize-your-code

Norvig, P. (1996). *Design Patterns in Dynamic Languages*. Retrieved from http://www.norvig.com/design-patterns/design-patterns.pdf

Osmani, A. (2012). Namespacing Fundamentals. In A. Osmani, *Learning JavaScript Patterns.* O'Reilly Media, Inc.

Osmani, A. (2012). Prototype Pattern. In A. Osmani, *Learning JavaScript design patterns.* O'Reilly Media, Inc.

Osmani, A. (2016). *Introducing ES2015 Proxies.* Retrieved from Google Developers: https://developers.google.com/web/updates/2016/02/es2015-proxies

Riehle, D. (2000). Case Study: The JHotDraw Framework. In D. Riehle, *Framework Design: A Role Modeling Approach* (p. Chapter 8). Doctoral thesis, Unpublished.

Riehle, D. (2000). Framework Design: A Role Modeling Approach. Doctoral Thesis, Unpublished. Retrieved from http://dirkriehle.com/computer-science/research/dissertation/

Riehle, D. (2011). Lessons Learned from Using.

Schmidt, D. C., Rohnert, H., Stal, M., & Buschmann, F. (2000). *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects.* John Wiley & Sons.

Scott, E. A. (2015). What is a single-page application? In E. A. Scott, *SPA Design and Architecture: Understanding single-page web applications.* Manning Publications.

Severance, C. (2012). JavaScript: Designing a language in 10 days. *IEEE Computer Society*.

Stefanov, S. (2017). Object-Oriented JavaScript. In S. Stefanov, *Object-Oriented JavaScript.* Pack Publishing.

Timms, S. (2016). Prototype. In S. Timms, *Mastering JavaScript Design Patterns.* Packt Publishing.

Vlissides, J. J. (1994). Structural Patterns. In J. J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional.

Vlissides, J., Johnson, R., Helm, R., & Gamma, E. (1994). Behavioral Patterns. In J. Vlissides, R. Johnson, R. Helm, & E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional.

Vlissides, J., Johnson, R., Helm, R., & Gamma, E. (1994). Creational Patterns. In J. Vlissides, R. Johnson, R. Helm, & E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional.

Vlissides, J., Johnson, R., Helm, R., & Gamma, E. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Professional.

Züllighoven, D. R. (1994). In: Pattern Languages of Program Design. Edited by James O. Coplien and Douglas Schmidt.