What is a Pattern?

A pattern is a reusable solution that can be applied to commonly occurring problems in software design - in our case - in writing JavaScript web applications. Another way of looking at patterns are as templates for how we solve problems - ones which can be used in quite a few different situations.

So, why is it important to understand patterns and be familiar with them? Design patterns have three main benefits:

* + 1. **Patterns are proven solutions:** They provide solid approaches to solving issues in software development using proven techniques that reflect the experience and insights the developers that helped define them bring to the pattern.
    2. **Patterns can be easily reused:**A pattern usually reflects an out of the box solution that can be adapted to suit our own needs. This feature makes them quite robust.
    3. **Patterns can be expressive:**When we look at a pattern there’s generally a set structure and *vocabulary* to the solution presented that can help express rather large solutions quite elegantly.

Patterns are **not** an exact solution. It’s important that we remember the role of a pattern is merely to provide us with a solution scheme. Patterns don’t solve all design problems nor do they replace good software designers, however, they **do** support them. Next we’ll take a look at some of the other advantages patterns have to offer.

* **Reusing patterns assists in preventing minor issues that can cause major problems in the application development process.**What this means is when code is built on proven patterns, we can afford to spend less time worrying about the structure of our code and more time focusing on the quality of our overall solution. This is because patterns can encourage us to code in a more structured and organized fashion avoiding the need to refactor it for cleanliness purposes in the future.
* **Patterns can provide generalized solutions which are documented in a fashion that doesn't require them to be tied to a specific problem.** This generalized approach means that regardless of the application (and in many cases the programming language) we are working with, design patterns can be applied to improve the structure of our code.
* **Certain patterns can actually decrease the overall file-size footprint of our code by avoiding repetition.**By encouraging developers to look more closely at their solutions for areas where instant reductions in repetition can be made, e.g. reducing the number of functions performing similar processes in favor of a single generalized function, the overall size of our codebase can be decreased. This is also known as making code more *DRY*.
* **Patterns add to a developer's vocabulary, which makes communication faster.**
* **Patterns that are frequently used can be improved over time by harnessing the collective experiences other developers using those patterns contribute back to the design pattern community.** In some cases this leads to the creation of entirely new design patterns whilst in others it can lead to the provision of improved guidelines on how specific patterns can be best used. This can ensure that pattern-based solutions continue to become more robust than ad-hoc solutions may be.

We already use patterns everyday

To understand how useful patterns can be, let's review a very simple element selection problem that the jQuery library solves for us.

Imagine that we have a script where for each DOM element found on a page with class "foo" we wish to increment a counter. What's the most efficient way to query for this collection of elements? Well, there are a few different ways this problem could be tackled:

1. Select all of the elements in the page and then store references to them. Next, filter this collection and use regular expressions (or another means) to only store those with the class "foo".
2. Use a modern native browser feature such as querySelectorAll() to select all of the elements with the class "foo".
3. Use a native feature such as getElementsByClassName() to similarly get back the desired collection.

So, which of these options is the fastest? It's actually option 3. by a factor of 8-10 times the [alternatives](http://jsperf.com/getelementsbyclassname-vs-queryselectorall/5). In a real-world application however, 3. will not work in versions of Internet Explorer below 9 and thus it's necessary to use 1. where both 2. and 3. aren't supported.

Developers using jQuery don't have to worry about this problem however, as it's luckily abstracted away for us using the *Facade* pattern. As we'll review in more detail later, this pattern provides a simple set of abstracted interfaces (e.g $el.css(), $el.animate()) to several more complex underlying bodies of code. As we've seen, this means less time having to be concerned about implementation level details.

Behind the scenes, the library simply opts for the most optimal approach to selecting elements depending on what our current browser supports and we just consume the abstraction layer.

We're probably all also familiar with jQuery's $("selector"). This is significantly more easy to use for selecting HTML elements on a page versus having to manually opt for getElementById(),getElementsByClassName(), getElementByTagName() and so on.

Although we know that querySelectorAll() attempts to solve this problem, compare the effort involved in using jQuery's Facade interfaces vs. selecting the most optimal selection paths ourselves. There's no contest! Abstractions using patterns can offer real-world value.

We'll be looking at this and more design patterns later on in the book.

"Pattern"-ity Testing, Proto-Patterns & The Rule Of Three

Remember that not every algorithm, best practice or solution represents what might be considered a complete pattern. There may be a few key ingredients here that are missing and the pattern community is generally wary of something claiming to be one unless it has been heavily vetted. Even if something is presented to us which **appears** to meet the criteria for a pattern, it should not be considered one until it has undergone suitable periods of scrutiny and testing by others.

Looking back upon the work by Alexander once more, he claims that a pattern should both be a process and a "thing". This definition is obtuse on purpose as he follows by saying that it is the process which should create the "thing". This is a reason why patterns generally focus on addressing a visually identifiable structure i.e we should be able to visually depict (or draw) a picture representing the structure that placing the pattern into practice results in.

In studying design patterns, it's not irregular to come across the term "proto-pattern". What is this? Well, a pattern that has not yet been known to pass the "pattern"-ity tests is usually referred to as a proto-pattern. Proto-patterns may result from the work of someone that has established a particular solution that is worthy of sharing with the community, but may not have yet had the opportunity to have been vetted heavily due to its very young age.

Alternatively, the individual(s) sharing the pattern may not have the time or interest of going through the "pattern"-ity process and might release a short description of their proto-pattern instead. Brief descriptions or snippets of this type of pattern are known as patlets.

The work involved in fully documenting a qualified pattern can be quite daunting. Looking back at some of the earliest work in the field of design patterns, a pattern may be considered "good" if it does the following:

* **Solves a particular problem**: Patterns are not supposed to just capture principles or strategies. They need to capture solutions. This is one of the most essential ingredients for a good pattern.
* **The solution to this problem cannot be obvious**: We can find that problem-solving techniques often attempt to derive from well-known first principles. The best design patterns usually provide solutions to problems indirectly - this is considered a necessary approach for the most challenging problems related to design.
* **The concept described must have been proven**: Design patterns require proof that they function as described and without this proof the design cannot be seriously considered. If a pattern is highly speculative in nature, only the brave may attempt to use it.
* **It must describe a relationship**: In some cases it may appear that a pattern describes a type of module. Although an implementation may appear this way, the official description of the pattern must describe much deeper system structures and mechanisms that explain its relationship to code.

We would be forgiven for thinking that a proto-pattern which fails to meet guidelines isn't worth learning from, however, this is far from the truth. Many proto-patterns are actually quite good. I’m not saying that all proto-patterns are worth looking at, but there are quite a few useful ones in the wild that could assist us with future projects. Use best judgment with the above list in mind and you’ll be fine in your selection process.

One of the additional requirements for a pattern to be valid is that they display some recurring phenomenon. This is often something that can be qualified in at least three key areas, referred to as the*rule of three*. To show recurrence using this rule, one must demonstrate:

1. **Fitness of purpose** - how is the pattern considered successful?
2. **Usefulness** - why is the pattern considered successful?
3. **Applicability** - is the design worthy of being a pattern because it has wider applicability? If so, this needs to be explained. When reviewing or defining a pattern, it is important to keep the above in mind.

The Structure Of A Design Pattern

You may be curious about how a pattern author might approach outlining structure, implementation and purpose of a new pattern.  A pattern is initially presented in the form of a **rule** that establishes a relationship between:

* A **context**
* A system of **forces** that arises in that context and
* A **configuration** that allows these forces to resolve themselves in context

With this in mind, let’s now take a look at a summary of the component elements for a design pattern. A design pattern should have a:

* **Pattern name** and a **description**
* **Context outline** – the contexts in which the pattern is effective in responding to the users needs.
* **Problem statement** – a statement of the problem being addressed so we can understand the intent of the pattern.
* **Solution** – a description of how the user’s problem is being solved in an understandable list of steps and perceptions.
* **Design** – a description of the pattern’s design and in particular, the user’s behavior in interacting with it
* **Implementation**– a guide to how the pattern would be implemented
* **Illustrations** – a visual representation of classes in the pattern (e.g. a diagram)
* **Examples** – an implementation of the pattern in a minimal form
* **Co-requisites** – what other patterns may be needed to support use of the pattern being described?
* **Relations** – what patterns does this pattern resemble? does it closely mimic any others?
* **Known usage** – is the pattern being used in the *wild*? If so, where and how?
* **Discussions** – the team or author’s thoughts on the exciting benefits of the pattern

Design patterns are quite a powerful approach to getting all of the developers in an organization or team on the same page when creating or maintaining solutions. If considering working on a pattern of your own, remember that although they may have a heavy initial cost in the planning and write-up phases, the value returned from that investment can be quite worth it. Always research thoroughly before working on new patterns however, as you may find it more beneficial to use or build on top of existing proven patterns than starting afresh.

Writing Design Patterns

Although this book is aimed at those new to design patterns, a fundamental understanding of how a design pattern is written can offer a number of useful benefits. For starters, we can gain a deeper appreciation for the reasoning behind why a pattern is needed. We can also learn how to tell if a pattern (or proto-pattern) is up to scratch when reviewing it for our own needs.

Writing good patterns is a challenging task. Patterns not only need to (ideally) provide a substantial quantity of reference material for end-users, but they also need to be able to defend why they are necessary.

Having read the previous section on *what* a pattern is, we may think that this in itself is enough to help us identify patterns we see in the wild. This is actually not completely true. It's not always clear if a piece of code we're looking at is following a set pattern or just accidentally happens to appear like it does.

When we're looking at a body of code we think may be using a pattern, we should consider writing down some of the aspects of the code that we believe falls under a particular existing pattern or set of patterns.

In many cases of pattern-analysis we can find that we're just looking at code that follows good principles and design practices that could happen to overlap with the rules for a pattern by accident. Remember - solutions in which neither interactions nor defined rules appear are *not* patterns.

If interested in venturing down the path of writing your own design patterns I recommend learning from others who have already been through the process and done it well. Spend time absorbing the information from a number of different design pattern descriptions and take in what’s meaningful to you.

Explore structure and semantics - this can be done by examining the interactions and context of the patterns you are interested in so you can identify the principles that assist in organizing those patterns together in useful configurations.

Once we've exposed ourselves to a wealth of information on pattern literature, we may wish to begin writing our pattern using an *existing* format and see if we can brainstorm new ideas for improving it or integrating our ideas in there.

An example of a developer that did this is in recent years is Christian Heilmann, who took the existing*Module* pattern and made some fundamentally useful changes to it to create the *Revealing Module* pattern (this is one of the patterns covered later in this book).

The following are tips I would suggest if interested in creating a new design pattern:

* **How practical is the pattern?**: Ensure the pattern describes proven solutions to recurring problems rather than just speculative solutions which haven’t been qualified.
* **Keep best practices in mind:** The design decisions we make should be based on principles we derive from an understanding of best practices.
* **Our design patterns should be transparent to the user**: Design patterns should be entirely transparent to any type of user-experience. They are primarily there to serve the developers using them and should not force changes to behavior in the user-experience that would not be incurred without the use of a pattern.
* **Remember that originality is *not* key in pattern design**: When writing a pattern, we do not need to be the original discoverer of the solutions being documented nor do you have to worry about our design overlapping with minor pieces of other patterns. If the approach is strong enough to have broad useful applicability, it has a chance of being recognized as a valid pattern.
* **Patterns need a strong set of examples:** A good pattern description needs to be followed by an equally strong set of examples demonstrating the successful application of our pattern. To show broad usage, examples that exhibit good design principles are ideal.

Pattern writing is a careful balance between creating a design that is general, specific and above all, useful. Try to ensure that if writing a pattern you cover the widest possible areas of application and you should be fine.  I hope that this brief introduction to writing patterns has given you some insights that will assist your learning process for the next sections of this book.

Anti-Patterns

If we consider that a pattern represents a best practice, an anti-pattern represents a lesson that has been learned. The term anti-patterns was coined in 1995 by Andrew Koenig in the November C++ Report that year, inspired by the GoF's book *Design Patterns*. In Koenig’s report, there are two notions of anti-patterns that are presented. Anti-Patterns:

* Describe a*bad* solution to a particular problem which resulted in a bad situation occurring
* Describe *how* to get out of said situation and how to go from there to a good solution

On this topic, Alexander writes about the difficulties in achieving a good balance between good design structure and good context:

*“These notes are about the process of design; the process of inventing physical things which display a new physical order, organization, form, in response to function.…every design problem begins with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem”.*

While it’s quite important to be aware of design patterns, it can be equally important to understand anti-patterns. Let us qualify the reason behind this. When creating an application, a project’s life-cycle begins with construction however once you’ve got the initial release done, it needs to be maintained. The quality of a final solution will either be *good* or *bad*, depending on the level of skill and time the team have invested in it. Here *good* and *bad* are considered in context - a ‘perfect’ design may qualify as an anti-pattern if applied in the wrong context.

The bigger challenges happen after an application has hit production and is ready to go into maintenance mode. A developer working on such a system who hasn’t worked on the application before may introduce a *bad* design into the project by accident. If said *bad* practices are created as anti-patterns, they allow developers a means to recognize these in advance so that they can avoid common mistakes that can occur - this is parallel to the way in which design patterns provide us with a way to recognize common techniques that are *useful.*

To summarize, an anti-pattern is a bad design that is worthy of documenting. Examples of anti-patterns in JavaScript are the following:

* Polluting the global namespace by defining a large number of variables in the global context
* Passing strings rather than functions to either setTimeout or setInterval as this triggers the use of eval() internally.
* Modifying the Object class prototype (this is a particularly bad anti-pattern)
* Using JavaScript in an inline form as this is inflexible
* The use of document.write where native DOM alternatives such as document.createElement are more appropriate. document.write has been grossly misused over the years and has quite a few disadvantages including that if it's executed after the page has been loaded it can actually overwrite the page we're on, whilst document.createElement does not. We can see[here](http://jsfiddle.net/addyosmani/6T9vX/) for a live example of this in action. It also doesn't work with XHTML which is another reason opting for more DOM-friendly methods such as document.createElement is favorable.

Knowledge of anti-patterns is critical for success. Once we are able to recognize such anti-patterns, we're able to refactor our code to negate them so that the overall quality of our solutions improves instantly.

Categories Of Design Pattern

A glossary from the well-known design book, *Domain-Driven Terms*, rightly states that:

*“A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and their instances, their roles and collaborations, and the distribution of responsibilities.*

*Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether or not it can be applied in view of other design constraints, and the consequences and trade-offs of its use. Since we must eventually implement our designs, a design pattern also provides sample ... code to illustrate an implementation.*

*Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages ....”*

Design patterns can be broken down into a number of different categories. In this section we’ll review three of these categories and briefly mention a few examples of the patterns that fall into these categories before exploring specific ones in more detail.

Creational Design Patterns

Creational design patterns focus on handling object creation mechanisms where objects are created in a manner suitable for the situation we're working in. The basic approach to object creation might otherwise lead to added complexity in a project whilst these patterns aim to solve this problem by*controlling* the creation process.

Some of the patterns that fall under this category are: Constructor, Factory, Abstract, Prototype, Singleton and Builder.

Structural Design Patterns

Structural patterns are concerned with object composition and typically identify simple ways to realize relationships between different objects. They help ensure that when one part of a system changes, the entire structure of the system doesn't need to do the same. They also assist in recasting parts of the system which don't fit a particular purpose into those that do.

Patterns that fall under this category include: Decorator, Facade, Flyweight, Adapter and Proxy.

Behavioral Design Patterns

Behavioral patterns focus on improving or streamlining the communication between disparate objects in a system.

Some behavioral patterns include: Iterator, Mediator, Observer and Visitor.

Design Pattern Categorization

In my early experiences of learning about design patterns, I personally found the following table a very useful reminder of what a number of patterns has to offer - it covers the 23 Design Patterns mentioned by the GoF. The original table was summarized by Elyse Nielsen back in 2004 and I've modified it where necessary to suit our discussion in this section of the book.

I recommend using this table as reference, but do remember that there are a number of additional patterns that are not mentioned here but will be discussed later in the book.

A brief note on classes

Keep in mind that there will be patterns in this table that reference the concept of "classes". JavaScript is a class-less language, however classes can be simulated using functions.

The most common approach to achieving this is by defining a JavaScript function where we then create an object using the new keyword. this can be used to help define new properties and methods for the object as follows:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | // A car "class"  function Car( model ) {      this.model = model;    this.color = "silver";    this.year = "2012";      this.getInfo = function () {      return this.model + " " + this.year;    };    } |

We can then instantiate the object using the Car constructor we defined above like this:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5 | var myCar = new Car("ford");    myCar.year = "2010";    console.log( myCar.getInfo() ); |

For more ways to define "classes" using JavaScript, see Stoyan Stefanov's useful [post](http://www.phpied.com/3-ways-to-define-a-javascript-class/) on them.

Let us now proceed to review the table.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Creational** | | | | Based on the concept of creating an object. | | | |
| ***Class*** | | | | | | | |
| *Factory Method* | | | | | This makes an instance of several derived classes based on interfaced data or events. | | |
| ***Object*** | | | | | | | |
| *Abstract Factory* | | | | | Creates an instance of several families of classes without detailing concrete classes. | | |
| *Builder* | | | | | Separates object construction from its representation, always creates the same type of object. | | |
| *Prototype* | | | | | A fully initialized instance used for copying or cloning. | | |
| *Singleton* | | | | | A class with only a single instance with global access points. | | |
|  |  |  |  |  |  |  |  |
| **Structural** | | | | Based on the idea of building blocks of objects. | | | |
| ***Class*** | | | | | | | |
| *Adapter* | | | | | Match interfaces of different classes therefore classes can work together despite incompatible interfaces. | | |
| ***Object*** | | | | | | | |
| *Adapter* | | | | | Match interfaces of different classes therefore classes can work together despite incompatible interfaces. | | |
| *Bridge* | | | | | Separates an object's interface from its implementation so the two can vary independently. | | |
| *Composite* | | | | | A structure of simple and composite objects which makes the total object more than just the sum of its parts. | | |
| *Decorator* | | | | | Dynamically add alternate processing to objects. | | |
| *Facade* | | | | | A single class that hides the complexity of an entire subsystem. | | |
| *Flyweight* | | | | | A fine-grained instance used for efficient sharing of information that is contained elsewhere. | | |
| *Proxy* | | | | | A place holder object representing the true object. | | |
|  | | | | | | | |
| **Behavioral** | | | | Based on the way objects play and work together. | | | |
| ***Class*** | | | | | | | |
| *Interpreter* | | | | | A way to include language elements in an application to match the grammar of the intended language. | | |
| *Template        Method* | | | | | Creates the shell of an algorithm in a method, then defer the exact steps to a subclass. | | |
| ***Object*** | | | | | | | |
| *Chain of       Responsibility* | | | | | A way of passing a request between a chain of objects to find the object that can handle the request. | | |
| *Command* | | | | | Encapsulate a command request as an object to enable, logging and/or queuing of requests, and provides error-handling for unhandled requests. | | |
| *Iterator* | | | | | Sequentially access the elements of a collection without knowing the inner workings of the collection. | | |
| *Mediator* | | | | | Defines simplified communication between classes to prevent a group of classes from referring explicitly to each other. | | |
| *Memento* | | | | | Capture an object's internal state to be able to restore it later. | | |
| *Observer* | | | | | A way of notifying change to a number of classes to ensure consistency between the classes. | | |
| *State* | | | | | Alter an object's behavior when its state changes. | | |
| *Strategy* | | | | | Encapsulates an algorithm inside a class separating the selection from the implementation. | | |
| *Visitor* | | | | | Adds a new operation to a class without changing the class. | | |

JavaScript Design Patterns

In this section, we will explore JavaScript implementations of a number of both classic and modern design patterns.

Developers commonly wonder whether there is an *ideal* pattern or set of patterns they should be using in their workflow. There isn't a true single answer to this question; each script and web application we work on is likely to have its own individual needs and we need to think about where we feel a pattern can offer real value to an implementation.

For example, some projects may benefit from the decoupling benefits offered by the Observer pattern (which reduces how dependent parts of an application are on one another) whilst others may simply be too small for decoupling to be a concern at all.

That said, once we have a firm grasp of design patterns and the specific problems they are best suited to, it becomes much easier to integrate them into our application architectures.

**The patterns we will be exploring in this section are the:**

* [Constructor Pattern](https://addyosmani.com/resources/essentialjsdesignpatterns/book/#constructorpatternjavascript)
* [Module Pattern](https://addyosmani.com/resources/essentialjsdesignpatterns/book/#modulepatternjavascript)
* [Revealing Module Pattern](https://addyosmani.com/resources/essentialjsdesignpatterns/book/#revealingmodulepatternjavascript)
* [Singleton Pattern](https://addyosmani.com/resources/essentialjsdesignpatterns/book/#singletonpatternjavascript)
* [Observer Pattern](https://addyosmani.com/resources/essentialjsdesignpatterns/book/#observerpatternjavascript)
* [Mediator Pattern](https://addyosmani.com/resources/essentialjsdesignpatterns/book/#mediatorpatternjavascript)
* [Prototype Pattern](https://addyosmani.com/resources/essentialjsdesignpatterns/book/#prototypepatternjavascript)
* [Command Pattern](https://addyosmani.com/resources/essentialjsdesignpatterns/book/#commandpatternjavascript)
* [Facade Pattern](https://addyosmani.com/resources/essentialjsdesignpatterns/book/#facadepatternjavascript)
* [Factory Pattern](https://addyosmani.com/resources/essentialjsdesignpatterns/book/#factorypatternjavascript)
* [Mixin Pattern](https://addyosmani.com/resources/essentialjsdesignpatterns/book/#mixinpatternjavascript)
* [Decorator Pattern](https://addyosmani.com/resources/essentialjsdesignpatterns/book/#decoratorpatternjavascript)
* [Flyweight Pattern](https://addyosmani.com/resources/essentialjsdesignpatterns/book/#detailflyweight)

The Constructor Pattern

In classical object-oriented programming languages, a constructor is a special method used to initialize a newly created object once memory has been allocated for it. In JavaScript, as almost everything is an object, we're most often interested in *object* constructors.

Object constructors are used to create specific types of objects - both preparing the object for use and accepting arguments which a constructor can use to set the values of member properties and methods when the object is first created.

Object Creation

The three common ways to create new objects in JavaScript are as follows:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // Each of the following options will create a new empty object:    var newObject = {};    // or  var newObject = Object.create( Object.prototype );    // or  var newObject = new Object(); |

Where the "Object" constructor in the final example creates an object wrapper for a specific value, or where no value is passed, it will create an empty object and return it.

There are then four ways in which keys and values can then be assigned to an object:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79 | // ECMAScript 3 compatible approaches    // 1. Dot syntax    // Set properties  newObject.someKey = "Hello World";    // Get properties  var value = newObject.someKey;        // 2. Square bracket syntax    // Set properties  newObject["someKey"] = "Hello World";    // Get properties  var value = newObject["someKey"];        // ECMAScript 5 only compatible approaches  // For more information see: <http://kangax.github.com/es5-compat-table/>    // 3. Object.defineProperty    // Set properties  Object.defineProperty( newObject, "someKey", {      value: "for more control of the property's behavior",      writable: true,      enumerable: true,      configurable: true  });    // If the above feels a little difficult to read, a short-hand could  // be written as follows:    var defineProp = function ( obj, key, value ){    var config = {      value: value,      writable: true,      enumerable: true,      configurable: true    };    Object.defineProperty( obj, key, config );  };    // To use, we then create a new empty "person" object  var person = Object.create( Object.prototype );    // Populate the object with properties  defineProp( person, "car", "Delorean" );  defineProp( person, "dateOfBirth", "1981" );  defineProp( person, "hasBeard", false );    console.log(person);  // Outputs: Object {car: "Delorean", dateOfBirth: "1981", hasBeard: false}      // 4. Object.defineProperties    // Set properties  Object.defineProperties( newObject, {      "someKey": {      value: "Hello World",      writable: true    },      "anotherKey": {      value: "Foo bar",      writable: false    }    });    // Getting properties for 3. and 4. can be done using any of the  // options in 1. and 2. |

As we will see a little later in the book, these methods can even be used for inheritance, as follows:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // Usage:    // Create a race car driver that inherits from the person object  var driver = Object.create( person );    // Set some properties for the driver  defineProp(driver, "topSpeed", "100mph");    // Get an inherited property (1981)  console.log( driver.dateOfBirth );    // Get the property we set (100mph)  console.log( driver.topSpeed ); |

Basic Constructors

As we saw earlier, JavaScript doesn't support the concept of classes but it does support special constructor functions that work with objects. By simply prefixing a call to a constructor function with the keyword "new", we can tell JavaScript we would like the function to behave like a constructor and instantiate a new object with the members defined by that function.

Inside a constructor, the keyword *this* references the new object that's being created. Revisiting object creation, a basic constructor may look as follows:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | function Car( model, year, miles ) {      this.model = model;    this.year = year;    this.miles = miles;      this.toString = function () {      return this.model + " has done " + this.miles + " miles";    };  }    // Usage:    // We can create new instances of the car  var civic = new Car( "Honda Civic", 2009, 20000 );  var mondeo = new Car( "Ford Mondeo", 2010, 5000 );    // and then open our browser console to view the  // output of the toString() method being called on  // these objects  console.log( civic.toString() );  console.log( mondeo.toString() ); |

The above is a simple version of the constructor pattern but it does suffer from some problems. One is that it makes inheritance difficult and the other is that functions such as toString() are redefined for each of the new objects created using the Car constructor. This isn't very optimal as the function should ideally be shared between all of the instances of the Car type.

Thankfully as there are a number of both ES3 and ES5-compatible alternatives to constructing objects, it's trivial to work around this limitation.

Constructors With Prototypes

Functions, like almost all objects in JavaScript, contain a "prototype" object. When we call a JavaScript constructor to create an object, all the properties of the constructor's prototype are then made available to the new object. In this fashion, multiple Car objects can be created which access the same prototype. We can thus extend the original example as follows:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | function Car( model, year, miles ) {      this.model = model;    this.year = year;    this.miles = miles;    }      // Note here that we are using Object.prototype.newMethod rather than  // Object.prototype so as to avoid redefining the prototype object  Car.prototype.toString = function () {    return this.model + " has done " + this.miles + " miles";  };    // Usage:    var civic = new Car( "Honda Civic", 2009, 20000 );  var mondeo = new Car( "Ford Mondeo", 2010, 5000 );    console.log( civic.toString() );  console.log( mondeo.toString() ); |

Above, a single instance of toString() will now be shared between all of the Car objects.

The Module Pattern

Modules

Modules are an integral piece of any robust application's architecture and typically help in keeping the units of code for a project both cleanly separated and organized.

In JavaScript, there are several options for implementing modules. These include:

* The Module pattern
* Object literal notation
* AMD modules
* CommonJS modules
* ECMAScript Harmony modules

We will be exploring the latter three of these options later on in the book in the section *Modern Modular JavaScript Design Patterns*.

The Module pattern is based in part on object literals and so it makes sense to refresh our knowledge of them first.

Object Literals

In object literal notation, an object is described as a set of comma-separated name/value pairs enclosed in curly braces ({}). Names inside the object may be either strings or identifiers that are followed by a colon. There should be no comma used after the final name/value pair in the object as this may result in errors.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var myObjectLiteral = {        variableKey: variableValue,        functionKey: function () {        // ...      }  }; |

Object literals don't require instantiation using the new operator but shouldn't be used at the start of a statement as the opening { may be interpreted as the beginning of a block. Outside of an object, new members may be added to it using assignment as follows myModule.property = "someValue";

Below we can see a more complete example of a module defined using object literal notation:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | var myModule = {      myProperty: "someValue",      // object literals can contain properties and methods.    // e.g we can define a further object for module configuration:    myConfig: {      useCaching: true,      language: "en"    },      // a very basic method    saySomething: function () {      console.log( "Where in the world is Paul Irish today?" );    },      // output a value based on the current configuration    reportMyConfig: function () {      console.log( "Caching is: " + ( this.myConfig.useCaching ? "enabled" : "disabled") );    },      // override the current configuration    updateMyConfig: function( newConfig ) {        if ( typeof newConfig === "object" ) {        this.myConfig = newConfig;        console.log( this.myConfig.language );      }    }  };    // Outputs: Where in the world is Paul Irish today?  myModule.saySomething();    // Outputs: Caching is: enabled  myModule.reportMyConfig();    // Outputs: fr  myModule.updateMyConfig({    language: "fr",    useCaching: false  });    // Outputs: Caching is: disabled  myModule.reportMyConfig(); |

Using object literals can assist in encapsulating and organizing your code and Rebecca Murphey has previously written about this topic in [depth](http://rmurphey.com/blog/2009/10/15/using-objects-to-organize-your-code/) should you wish to read into object literals further.

That said, if we're opting for this technique, we may be equally as interested in the Module pattern. It still uses object literals but only as the return value from a scoping function.

The Module Pattern

The Module pattern was originally defined as a way to provide both private and public encapsulation for classes in conventional software engineering.

In JavaScript, the Module pattern is used to further *emulate* the concept of classes in such a way that we're able to include both public/private methods and variables inside a single object, thus shielding particular parts from the global scope. What this results in is a reduction in the likelihood of our function names conflicting with other functions defined in additional scripts on the page.

Privacy

The Module pattern encapsulates "privacy", state and organization using closures. It provides a way of wrapping a mix of public and private methods and variables, protecting pieces from leaking into the global scope and accidentally colliding with another developer's interface. With this pattern, only a public API is returned, keeping everything else within the closure private.

This gives us a clean solution for shielding logic doing the heavy lifting whilst only exposing an interface we wish other parts of our application to use. The pattern is quite similar to an immediately-invoked functional expression ([IIFE](http://benalman.com/news/2010/11/immediately-invoked-function-expression/) - see the section on namespacing patterns for more on this) except that an object is returned rather than a function.

It should be noted that there isn't really an explicitly true sense of "privacy" inside JavaScript because unlike some traditional languages, it doesn't have access modifiers. Variables can't technically be declared as being public nor private and so we use function scope to simulate this concept. Within the Module pattern, variables or methods declared are only available inside the module itself thanks to closure. Variables or methods defined within the returning object however are available to everyone.

History

From a historical perspective, the Module pattern was originally developed by a number of people including [Richard Cornford](http://groups.google.com/group/comp.lang.javascript/msg/9f58bd11bd67d937) in 2003. It was later popularized by Douglas Crockford in his lectures. Another piece of trivia is that if you've ever played with Yahoo's YUI library, some of its features may appear quite familiar and the reason for this is that the Module pattern was a strong influence for YUI when creating their components.

Examples

Let's begin looking at an implementation of the Module pattern by creating a module which is self-contained.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | var testModule = (function () {      var counter = 0;      return {        incrementCounter: function () {        return counter++;      },        resetCounter: function () {        console.log( "counter value prior to reset: " + counter );        counter = 0;      }    };    })();    // Usage:    // Increment our counter  testModule.incrementCounter();    // Check the counter value and reset  // Outputs: counter value prior to reset: 1  testModule.resetCounter(); |

Here, other parts of the code are unable to directly read the value of our incrementCounter() orresetCounter(). The counter variable is actually fully shielded from our global scope so it acts just like a private variable would - its existence is limited to within the module's closure so that the only code able to access its scope are our two functions. Our methods are effectively namespaced so in the test section of our code, we need to prefix any calls with the name of the module (e.g. "testModule").

When working with the Module pattern, we may find it useful to define a simple template that we use for getting started with it. Here's one that covers namespacing, public and private variables:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | var myNamespace = (function () {      var myPrivateVar, myPrivateMethod;      // A private counter variable    myPrivateVar = 0;      // A private function which logs any arguments    myPrivateMethod = function( foo ) {        console.log( foo );    };      return {        // A public variable      myPublicVar: "foo",        // A public function utilizing privates      myPublicFunction: function( bar ) {          // Increment our private counter        myPrivateVar++;          // Call our private method using bar        myPrivateMethod( bar );        }    };    })(); |

Looking at another example, below we can see a shopping basket implemented using this pattern. The module itself is completely self-contained in a global variable called basketModule. The basket array in the module is kept private and so other parts of our application are unable to directly read it. It only exists with the module's closure and so the only methods able to access it are those with access to its scope (ie. addItem(), getItemCount() etc).

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | var basketModule = (function () {      // privates      var basket = [];      function doSomethingPrivate() {      //...    }      function doSomethingElsePrivate() {      //...    }      // Return an object exposed to the public    return {        // Add items to our basket      addItem: function( values ) {        basket.push(values);      },        // Get the count of items in the basket      getItemCount: function () {        return basket.length;      },        // Public alias to a private function      doSomething: doSomethingPrivate,        // Get the total value of items in the basket      getTotal: function () {          var q = this.getItemCount(),            p = 0;          while (q--) {          p += basket[q].price;        }          return p;      }    };  })(); |

Inside the module, you may have noticed that we return an object. This gets automatically assigned tobasketModule so that we can interact with it as follows:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | // basketModule returns an object with a public API we can use    basketModule.addItem({    item: "bread",    price: 0.5  });    basketModule.addItem({    item: "butter",    price: 0.3  });    // Outputs: 2  console.log( basketModule.getItemCount() );    // Outputs: 0.8  console.log( basketModule.getTotal() );    // However, the following will not work:    // Outputs: undefined  // This is because the basket itself is not exposed as a part of our  // public API  console.log( basketModule.basket );    // This also won't work as it only exists within the scope of our  // basketModule closure, but not in the returned public object  console.log( basket ); |

The methods above are effectively namespaced inside basketModule.

Notice how the scoping function in the above basket module is wrapped around all of our functions, which we then call and immediately store the return value of. This has a number of advantages including:

* The freedom to have private functions and private members which can only be consumed by our module. As they aren't exposed to the rest of the page (only our exported API is), they're considered truly private.
* Given that functions are declared normally and are named, it can be easier to show call stacks in a debugger when we're attempting to discover what function(s) threw an exception.
* As T.J Crowder has pointed out in the past, it also enables us to return different functions depending on the environment. In the past, I've seen developers use this to perform UA testing in order to provide a code-path in their module specific to IE, but we can easily opt for feature detection these days to achieve a similar goal.

Module Pattern Variations

**Import mixins**

This variation of the pattern demonstrates how globals (e.g jQuery, Underscore) can be passed in as arguments to our module's anonymous function. This effectively allows us to *import* them and locally alias them as we wish.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | // Global module  var myModule = (function ( jQ, \_ ) {        function privateMethod1(){          jQ(".container").html("test");      }        function privateMethod2(){        console.log( \_.min([10, 5, 100, 2, 1000]) );      }        return{          publicMethod: function(){              privateMethod1();          }      };    // Pull in jQuery and Underscore  })( jQuery, \_ );    myModule.publicMethod(); |

**Exports**

This next variation allows us to declare globals without consuming them and could similarly support the concept of global imports seen in the last example.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | // Global module  var myModule = (function () {      // Module object    var module = {},      privateVariable = "Hello World";      function privateMethod() {      // ...    }      module.publicProperty = "Foobar";    module.publicMethod = function () {      console.log( privateVariable );    };      return module;    })(); |

Toolkit And Framework-specific Module Pattern Implementations

**Dojo**

Dojo provides a convenience method for working with objects called dojo.setObject(). This takes as its first argument a dot-separated string such as myObj.parent.child which refers to a property called "child" within an object "parent" defined inside "myObj". Using setObject() allows us to set the value of children, creating any of the intermediate objects in the rest of the path passed if they don't already exist.

For example, if we wanted to declare basket.core as an object of the store namespace, this could be achieved as follows using the traditional way:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | var store = window.store || {};    if ( !store["basket"] ) {    store.basket = {};  }    if ( !store.basket["core"] ) {    store.basket.core = {};  }    store.basket.core = {    // ...rest of our logic  }; |

Or as follows using Dojo 1.7 (AMD-compatible version) and above:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | require(["dojo/\_base/customStore"], function( store ){      // using dojo.setObject()    store.setObject( "basket.core", (function() {          var basket = [];          function privateMethod() {            console.log(basket);        }          return {            publicMethod: function(){                    privateMethod();            }        };      })());    }); |

For more information on dojo.setObject(), see the official [documentation](http://dojotoolkit.org/reference-guide/1.7/dojo/setObject.html).

**ExtJS**

For those using Sencha's ExtJS, an example demonstrating how to correctly use the Module pattern with the framework can be found below.

Here, we see an example of how to define a namespace which can then be populated with a module containing both a private and public API. With the exception of some semantic differences, it's quite close to how the Module pattern is implemented in vanilla JavaScript:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | // create namespace  Ext.namespace("myNameSpace");    // create application  myNameSpace.app = function () {      // do NOT access DOM from here; elements don't exist yet      // private variables    var btn1,        privVar1 = 11;      // private functions    var btn1Handler = function ( button, event ) {        console.log( "privVar1=" + privVar1 );        console.log( "this.btn1Text=" + this.btn1Text );      };      // public space    return {      // public properties, e.g. strings to translate      btn1Text: "Button 1",        // public methods      init: function () {          if ( Ext.Ext2 ) {            btn1 = new Ext.Button({            renderTo: "btn1-ct",            text: this.btn1Text,            handler: btn1Handler          });          } else {            btn1 = new Ext.Button( "btn1-ct", {            text: this.btn1Text,            handler: btn1Handler          });          }      }    };  }(); |

**YUI**

Similarly, we can also implement the Module pattern when building applications using YUI3. The following example is heavily based on the original YUI Module pattern implementation by Eric Miraglia, but again, isn't vastly different from the vanilla JavaScript version:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | Y.namespace( "store.basket" ) ;  Y.store.basket = (function () {        var myPrivateVar, myPrivateMethod;        // private variables:      myPrivateVar = "I can be accessed only within Y.store.basket.";        // private method:      myPrivateMethod = function () {          Y.log( "I can be accessed only from within YAHOO.store.basket" );      }        return {          myPublicProperty: "I'm a public property.",            myPublicMethod: function () {              Y.log( "I'm a public method." );                // Within basket, I can access "private" vars and methods:              Y.log( myPrivateVar );              Y.log( myPrivateMethod() );                // The native scope of myPublicMethod is store so we can              // access public members using "this":              Y.log( this.myPublicProperty );          }      };    })(); |

**jQuery**

There are a number of ways in which jQuery code unspecific to plugins can be wrapped inside the Module pattern. Ben Cherry previously suggested an implementation where a function wrapper is used around module definitions in the event of there being a number of commonalities between modules.

In the following example, a library function is defined which declares a new library and automatically binds up the init function to document.ready when new libraries (ie. modules) are created.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | function library( module ) {      $( function() {      if ( module.init ) {        module.init();      }    });      return module;  }    var myLibrary = library(function () {      return {      init: function () {        // module implementation      }    };  }()); |

Advantages

We've seen why the Constructor pattern can be useful, but why is the Module pattern a good choice? For starters, it's a lot cleaner for developers coming from an object-oriented background than the idea of true encapsulation, at least from a JavaScript perspective.

Secondly, it supports private data - so, in the Module pattern, public parts of our code are able to touch the private parts, however the outside world is unable to touch the class's private parts (no laughing! Oh, and thanks to David Engfer for the joke).

Disadvantages

The disadvantages of the Module pattern are that as we access both public and private members differently, when we wish to change visibility, we actually have to make changes to each place the member was used.

We also can't access private members in methods that are added to the object at a later point. That said, in many cases the Module pattern is still quite useful and when used correctly, certainly has the potential to improve the structure of our application.

Other disadvantages include the inability to create automated unit tests for private members and additional complexity when bugs require hot fixes. It's simply not possible to patch privates. Instead, one must override all public methods which interact with the buggy privates. Developers can't easily extend privates either, so it's worth remembering privates are not as flexible as they may initially appear.

For further reading on the Module pattern, see Ben Cherry's excellent in-depth [article](http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth) on it.

The Revealing Module Pattern

Now that we're a little more familiar with the module pattern, let’s take a look at a slightly improved version - Christian Heilmann’s Revealing Module pattern.

The Revealing Module pattern came about as Heilmann was frustrated with the fact that he had to repeat the name of the main object when we wanted to call one public method from another or access public variables.  He also disliked the Module pattern’s requirement for having to switch to object literal notation for the things he wished to make public.

The result of his efforts was an updated pattern where we would simply define all of our functions and variables in the private scope and return an anonymous object with pointers to the private functionality we wished to reveal as public.

An example of how to use the Revealing Module pattern can be found below:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | var myRevealingModule = (function () {            var privateVar = "Ben Cherry",              publicVar = "Hey there!";            function privateFunction() {              console.log( "Name:" + privateVar );          }            function publicSetName( strName ) {              privateVar = strName;          }            function publicGetName() {              privateFunction();          }              // Reveal public pointers to          // private functions and properties            return {              setName: publicSetName,              greeting: publicVar,              getName: publicGetName          };        })();    myRevealingModule.setName( "Paul Kinlan" ); |

The pattern can also be used to reveal private functions and properties with a more specific naming scheme if we would prefer:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | var myRevealingModule = (function () {            var privateCounter = 0;            function privateFunction() {              privateCounter++;          }            function publicFunction() {              publicIncrement();          }            function publicIncrement() {              privateFunction();          }            function publicGetCount(){            return privateCounter;          }            // Reveal public pointers to          // private functions and properties           return {              start: publicFunction,              increment: publicIncrement,              count: publicGetCount          };        })();    myRevealingModule.start(); |

**Advantages**

This pattern allows the syntax of our scripts to be more consistent. It also makes it more clear at the end of the module which of our functions and variables may be accessed publicly which eases readability.

**Disadvantages**

A disadvantage of this pattern is that if a private function refers to a public function, that public function can't be overridden if a patch is necessary. This is because the private function will continue to refer to the private implementation and the pattern doesn't apply to public members, only to functions.

Public object members which refer to private variables are also subject to the no-patch rule notes above.

As a result of this, modules created with the Revealing Module pattern may be more fragile than those created with the original Module pattern, so care should be taken during usage.

The Singleton Pattern

The Singleton pattern is thus known because it restricts instantiation of a class to a single object. Classically, the Singleton pattern can be implemented by creating a class with a method that creates a new instance of the class if one doesn't exist. In the event of an instance already existing, it simply returns a reference to that object.

Singletons differ from static *classes* (or objects) as we can delay their initialization, generally because they require some information that may not be available during initialization time. They don't provide a way for code that is unaware of a previous reference to them to easily retrieve them. This is because it is neither the object or "class" that's returned by a Singleton, it's a structure. Think of how closured variables aren't actually closures - the function scope that provides the closure is the closure.

In JavaScript, Singletons serve as a shared resource namespace which isolate implementation code from the global namespace so as to provide a single point of access for functions.

We can implement a Singleton as follows:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102 | var mySingleton = (function () {      // Instance stores a reference to the Singleton    var instance;      function init() {        // Singleton        // Private methods and variables      function privateMethod(){          console.log( "I am private" );      }        var privateVariable = "Im also private";        var privateRandomNumber = Math.random();        return {          // Public methods and variables        publicMethod: function () {          console.log( "The public can see me!" );        },          publicProperty: "I am also public",          getRandomNumber: function() {          return privateRandomNumber;        }        };      };      return {        // Get the Singleton instance if one exists      // or create one if it doesn't      getInstance: function () {          if ( !instance ) {          instance = init();        }          return instance;      }      };    })();    var myBadSingleton = (function () {      // Instance stores a reference to the Singleton    var instance;      function init() {        // Singleton        var privateRandomNumber = Math.random();        return {          getRandomNumber: function() {          return privateRandomNumber;        }        };      };      return {        // Always create a new Singleton instance      getInstance: function () {          instance = init();          return instance;      }      };    })();      // Usage:    var singleA = mySingleton.getInstance();  var singleB = mySingleton.getInstance();  console.log( singleA.getRandomNumber() === singleB.getRandomNumber() ); // true    var badSingleA = myBadSingleton.getInstance();  var badSingleB = myBadSingleton.getInstance();  console.log( badSingleA.getRandomNumber() !== badSingleB.getRandomNumber() ); // true    // Note: as we are working with random numbers, there is a  // mathematical possibility both numbers will be the same,  // however unlikely. The above example should otherwise still  // be valid. |

What makes the Singleton is the global access to the instance (generally throughMySingleton.getInstance()) as we don't (at least in static languages) call new MySingleton() directly. This is however possible in JavaScript.

In the GoF book, the *applicability* of the Singleton pattern is described as follows:

* There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
* When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

The second of these points refers to a case where we might need code such as:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | mySingleton.getInstance = function(){    if ( this.\_instance == null ) {      if ( isFoo() ) {         this.\_instance = new FooSingleton();      } else {         this.\_instance = new BasicSingleton();      }    }    return this.\_instance;  }; |

Here, getInstance becomes a little like a Factory method and we don't need to update each point in our code accessing it. FooSingleton above would be a subclass of BasicSingleton and implement the same interface.

Why is deferring execution considered important for a Singleton?:

In C++ it serves to isolate from the unpredictability of the order of dynamic initialization, returning control to the programmer.

It is important to note the difference between a static instance of a class (object) and a Singleton: whilst a Singleton can be implemented as a static instance, it can also be constructed lazily, without the need for resources nor memory until this is actually needed.

If we have a static object that can be initialized directly, we need to ensure the code is always executed in the same order (e.g in case objCar needs objWheel during its initialization) and this doesn't scale when you have a large number of source files.

Both Singletons and static objects are useful but they shouldn't be overused - the same way in which we shouldn't overuse other patterns.

In practice, the Singleton pattern is useful when exactly one object is needed to coordinate others across a system. Here is one example with the pattern being used in this context:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | var SingletonTester = (function () {      // options: an object containing configuration options for the singleton    // e.g var options = { name: "test", pointX: 5};    function Singleton( options ) {        // set options to the options supplied      // or an empty object if none are provided      options = options || {};        // set some properties for our singleton      this.name = "SingletonTester";        this.pointX = options.pointX || 6;        this.pointY = options.pointY || 10;      }      // our instance holder    var instance;      // an emulation of static variables and methods    var \_static = {        name: "SingletonTester",        // Method for getting an instance. It returns      // a singleton instance of a singleton object      getInstance: function( options ) {        if( instance === undefined ) {          instance = new Singleton( options );        }          return instance;        }    };      return \_static;    })();    var singletonTest = SingletonTester.getInstance({    pointX: 5  });    // Log the output of pointX just to verify it is correct  // Outputs: 5  console.log( singletonTest.pointX ); |

Whilst the Singleton has valid uses, often when we find ourselves needing it in JavaScript it's a sign that we may need to re-evaluate our design.

They're often an indication that modules in a system are either tightly coupled or that logic is overly spread across multiple parts of a codebase. Singletons can be more difficult to test due to issues ranging from hidden dependencies, the difficulty in creating multiple instances, difficulty in stubbing dependencies and so on.

Miller Medeiros has previously recommended [this](http://www.ibm.com/developerworks/webservices/library/co-single/index.html) excellent article on the Singleton and its various issues for further reading as well as the comments to [this](http://misko.hevery.com/2008/10/21/dependency-injection-myth-reference-passing/) article, discussing how Singletons can increase tight coupling. I'm happy to second these recommendations as both pieces raise many important points about this pattern that are also worth noting.

The Observer Pattern

The Observer is a design pattern where an object (known as a subject) maintains a list of objects depending on it (observers), automatically notifying them of any changes to state.

When a subject needs to notify observers about something interesting happening, it broadcasts a notification to the observers (which can include specific data related to the topic of the notification).

When we no longer wish for a particular observer to be notified of changes by the subject they are registered with, the subject can remove them from the list of observers.

It's often useful to refer back to published definitions of design patterns that are language agnostic to get a broader sense of their usage and advantages over time. The definition of the Observer pattern provided in the GoF book, *Design Patterns: Elements of Reusable Object-Oriented Software*, is:

*"One or more observers are interested in the state of a subject and register their interest with the subject by attaching themselves. When something changes in our subject that the observer may be interested in, a notify message is sent which calls the update method in each observer. When the observer is no longer interested in the subject's state, they can simply detach themselves."*

We can now expand on what we've learned to implement the Observer pattern with the following components:

* **Subject**: maintains a list of observers, facilitates adding or removing observers
* **Observer**: provides a update interface for objects that need to be notified of a Subject's changes of state
* **ConcreteSubject**: broadcasts notifications to observers on changes of state, stores the state of ConcreteObservers
* **ConcreteObserver**: stores a reference to the ConcreteSubject, implements an update interface for the Observer to ensure state is consistent with the Subject's

First, let's model the list of dependent Observers a subject may have:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | function ObserverList(){    this.observerList = [];  }    ObserverList.prototype.add = function( obj ){    return this.observerList.push( obj );  };    ObserverList.prototype.count = function(){    return this.observerList.length;  };    ObserverList.prototype.get = function( index ){    if( index > -1 && index < this.observerList.length ){      return this.observerList[ index ];    }  };    ObserverList.prototype.indexOf = function( obj, startIndex ){    var i = startIndex;      while( i < this.observerList.length ){      if( this.observerList[i] === obj ){        return i;      }      i++;    }      return -1;  };    ObserverList.prototype.removeAt = function( index ){    this.observerList.splice( index, 1 );  }; |

Next, let's model the Subject and the ability to add, remove or notify observers on the observer list.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | function Subject(){    this.observers = new ObserverList();  }    Subject.prototype.addObserver = function( observer ){    this.observers.add( observer );  };    Subject.prototype.removeObserver = function( observer ){    this.observers.removeAt( this.observers.indexOf( observer, 0 ) );  };    Subject.prototype.notify = function( context ){    var observerCount = this.observers.count();    for(var i=0; i < observerCount; i++){      this.observers.get(i).update( context );    }  }; |

We then define a skeleton for creating new Observers. The update functionality here will be overwritten later with custom behaviour.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // The Observer  function Observer(){    this.update = function(){      // ...    };  } |

In our sample application using the above Observer components, we now define:

* A button for adding new observable checkboxes to the page
* A control checkbox which will act as a subject, notifying other checkboxes they should be checked
* A container for the new checkboxes being added

We then define ConcreteSubject and ConcreteObserver handlers for both adding new observers to the page and implementing the updating interface. See below for inline comments on what these components do in the context of our example.

**HTML:**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3 | <button id="addNewObserver">Add New Observer checkbox</button>  <input id="mainCheckbox" type="checkbox"/>  <div id="observersContainer"></div> |

**Sample script:**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | // Extend an object with an extension  function extend( obj, extension ){    for ( var key in extension ){      obj[key] = extension[key];    }  }    // References to our DOM elements    var controlCheckbox = document.getElementById( "mainCheckbox" ),    addBtn = document.getElementById( "addNewObserver" ),    container = document.getElementById( "observersContainer" );      // Concrete Subject    // Extend the controlling checkbox with the Subject class  extend( controlCheckbox, new Subject() );    // Clicking the checkbox will trigger notifications to its observers  controlCheckbox.onclick = function(){    controlCheckbox.notify( controlCheckbox.checked );  };    addBtn.onclick = addNewObserver;    // Concrete Observer    function addNewObserver(){      // Create a new checkbox to be added    var check = document.createElement( "input" );    check.type = "checkbox";      // Extend the checkbox with the Observer class    extend( check, new Observer() );      // Override with custom update behaviour    check.update = function( value ){      this.checked = value;    };      // Add the new observer to our list of observers    // for our main subject    controlCheckbox.addObserver( check );      // Append the item to the container    container.appendChild( check );  } |

In this example, we looked at how to implement and utilize the Observer pattern, covering the concepts of a Subject, Observer, ConcreteSubject and ConcreteObserver.

Differences Between The Observer And Publish/Subscribe Pattern

Whilst the Observer pattern is useful to be aware of, quite often in the JavaScript world, we'll find it commonly implemented using a variation known as the Publish/Subscribe pattern. Whilst very similar, there are differences between these patterns worth noting.

The Observer pattern requires that the observer (or object) wishing to receive topic notifications must subscribe this interest to the object firing the event (the subject).

The Publish/Subscribe pattern however uses a topic/event channel which sits between the objects wishing to receive notifications (subscribers) and the object firing the event (the publisher). This event system allows code to define application specific events which can pass custom arguments containing values needed by the subscriber. The idea here is to avoid dependencies between the subscriber and publisher.

This differs from the Observer pattern as it allows any subscriber implementing an appropriate event handler to register for and receive topic notifications broadcast by the publisher.

Here is an example of how one might use the Publish/Subscribe if provided with a functional implementation powering publish(),subscribe() and unsubscribe() behind the scenes:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | // A very simple new mail handler    // A count of the number of messages received  var mailCounter = 0;    // Initialize subscribers that will listen out for a topic  // with the name "inbox/newMessage".    // Render a preview of new messages  var subscriber1 = subscribe( "inbox/newMessage", function( topic, data ) {      // Log the topic for debugging purposes    console.log( "A new message was received: ", topic );      // Use the data that was passed from our subject    // to display a message preview to the user    $( ".messageSender" ).html( data.sender );    $( ".messagePreview" ).html( data.body );    });    // Here's another subscriber using the same data to perform  // a different task.    // Update the counter displaying the number of new  // messages received via the publisher    var subscriber2 = subscribe( "inbox/newMessage", function( topic, data ) {      $('.newMessageCounter').html( ++mailCounter );    });    publish( "inbox/newMessage", [{    sender: "hello@google.com",    body: "Hey there! How are you doing today?"  }]);    // We could then at a later point unsubscribe our subscribers  // from receiving any new topic notifications as follows:  // unsubscribe( subscriber1 );  // unsubscribe( subscriber2 ); |

The general idea here is the promotion of loose coupling. Rather than single objects calling on the methods of other objects directly, they instead subscribe to a specific task or activity of another object and are notified when it occurs.

Advantages

The Observer and Publish/Subscribe patterns encourage us to think hard about the relationships between different parts of our application. They also help us identify what layers containing direct relationships which could instead be replaced with sets of subjects and observers. This effectively could be used to break down an application into smaller, more loosely coupled blocks to improve code management and potentials for re-use.

Further motivation behind using the Observer pattern is where we need to maintain consistency between related objects without making classes tightly coupled. For example, when an object needs to be able to notify other objects without making assumptions regarding those objects.

Dynamic relationships can exist between observers and subjects when using either pattern. This provides a great deal of flexibility which may not be as easy to implement when disparate parts of our application are tightly coupled.

Whilst it may not always be the best solution to every problem, these patterns remain one of the best tools for designing decoupled systems and should be considered an important tool in any JavaScript developer's utility belt.

Disadvantages

Consequently, some of the issues with these patterns actually stem from their main benefits. In Publish/Subscribe, by decoupling publishers from subscribers, it can sometimes become difficult to obtain guarantees that particular parts of our applications are functioning as we may expect.

For example, publishers may make an assumption that one or more subscribers are listening to them. Say that we're using such an assumption to log or output errors regarding some application process. If the subscriber performing the logging crashes (or for some reason fails to function), the publisher won't have a way of seeing this due to the decoupled nature of the system.

Another draw-back of the pattern is that subscribers are quite ignorant to the existence of each other and are blind to the cost of switching publishers. Due to the dynamic relationship between subscribers and publishers, the update dependency can be difficult to track.

Publish/Subscribe Implementations

Publish/Subscribe fits in very well in JavaScript ecosystems, largely because at the core, ECMAScript implementations are event driven. This is particularly true in browser environments as the DOM uses events as its main interaction API for scripting.

That said, neither ECMAScript nor DOM provide core objects or methods for creating custom events systems in implementation code (with the exception of perhaps the DOM3 CustomEvent, which is bound to the DOM and is thus not generically useful).

Luckily, popular JavaScript libraries such as dojo, jQuery (custom events) and YUI already have utilities that can assist in easily implementing a Publish/Subscribe system with very little effort. Below we can see some examples of this:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | // Publish    // jQuery: $(obj).trigger("channel", [arg1, arg2, arg3]);  $( el ).trigger( "/login", [{username:"test", userData:"test"}] );    // Dojo: dojo.publish("channel", [arg1, arg2, arg3] );  dojo.publish( "/login", [{username:"test", userData:"test"}] );    // YUI: el.publish("channel", [arg1, arg2, arg3]);  el.publish( "/login", {username:"test", userData:"test"} );      // Subscribe    // jQuery: $(obj).on( "channel", [data], fn );  $( el ).on( "/login", function( event ){...} );    // Dojo: dojo.subscribe( "channel", fn);  var handle = dojo.subscribe( "/login", function(data){..} );    // YUI: el.on("channel", handler);  el.on( "/login", function( data ){...} );      // Unsubscribe    // jQuery: $(obj).off( "channel" );  $( el ).off( "/login" );    // Dojo: dojo.unsubscribe( handle );  dojo.unsubscribe( handle );    // YUI: el.detach("channel");  el.detach( "/login" ); |

For those wishing to use the Publish/Subscribe pattern with vanilla JavaScript (or another library)[AmplifyJS](http://amplifyjs.com/) includes a clean, library-agnostic implementation that can be used with any library or toolkit. Radio.js (<http://radio.uxder.com/>), PubSubJS (<https://github.com/mroderick/PubSubJS>) or Pure JS PubSub by Peter Higgins (<https://github.com/phiggins42/bloody-jquery-plugins/blob/55e41df9bf08f42378bb08b93efcb28555b61aeb/pubsub.js>) are also similar alternatives worth checking out.

jQuery developers in particular have quite a few other options and can opt to use one of the many well-developed implementations ranging from Peter Higgins's jQuery plugin to Ben Alman's (optimized) Pub/Sub jQuery gist on GitHub. Links to just a few of these can be found below.

* Ben Alman's Pub/Sub gist <https://gist.github.com/661855> (recommended)
* Rick Waldron's jQuery-core style take on the above <https://gist.github.com/705311>
* Peter Higgins" plugin <http://github.com/phiggins42/bloody-jquery-plugins/blob/master/pubsub.js>.
* AppendTo's Pub/Sub in AmplifyJS [http://amplifyjs.com](http://amplifyjs.com/)
* Ben Truyman's gist <https://gist.github.com/826794>

So that we are able to get an appreciation for how many of the vanilla JavaScript implementations of the Observer pattern might work, let's take a walk through of a minimalist version of Publish/Subscribe I released on GitHub under a project called [pubsubz](http://github.com/addyosmani/pubsubz). This demonstrates the core concepts of subscribe, publish as well as the concept of unsubscribing.

I've opted to base our examples on this code as it sticks closely to both the method signatures and approach of implementation I would expect to see in a JavaScript version of the classic Observer pattern.

A Publish/Subscribe Implementation

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | var pubsub = {};    (function(myObject) {        // Storage for topics that can be broadcast      // or listened to      var topics = {};        // An topic identifier      var subUid = -1;        // Publish or broadcast events of interest      // with a specific topic name and arguments      // such as the data to pass along      myObject.publish = function( topic, args ) {            if ( !topics[topic] ) {              return false;          }            var subscribers = topics[topic],              len = subscribers ? subscribers.length : 0;            while (len--) {              subscribers[len].func( topic, args );          }            return this;      };        // Subscribe to events of interest      // with a specific topic name and a      // callback function, to be executed      // when the topic/event is observed      myObject.subscribe = function( topic, func ) {            if (!topics[topic]) {              topics[topic] = [];          }            var token = ( ++subUid ).toString();          topics[topic].push({              token: token,              func: func          });          return token;      };        // Unsubscribe from a specific      // topic, based on a tokenized reference      // to the subscription      myObject.unsubscribe = function( token ) {          for ( var m in topics ) {              if ( topics[m] ) {                  for ( var i = 0, j = topics[m].length; i < j; i++ ) {                      if ( topics[m][i].token === token ) {                          topics[m].splice( i, 1 );                          return token;                      }                  }              }          }          return this;      };  }( pubsub )); |

Example: Using Our Implementation

We can now use the implementation to publish and subscribe to events of interest as follows:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | // Another simple message handler    // A simple message logger that logs any topics and data received through our  // subscriber  var messageLogger = function ( topics, data ) {      console.log( "Logging: " + topics + ": " + data );  };    // Subscribers listen for topics they have subscribed to and  // invoke a callback function (e.g messageLogger) once a new  // notification is broadcast on that topic  var subscription = pubsub.subscribe( "inbox/newMessage", messageLogger );    // Publishers are in charge of publishing topics or notifications of  // interest to the application. e.g:    pubsub.publish( "inbox/newMessage", "hello world!" );    // or  pubsub.publish( "inbox/newMessage", ["test", "a", "b", "c"] );    // or  pubsub.publish( "inbox/newMessage", {    sender: "hello@google.com",    body: "Hey again!"  });    // We can also unsubscribe if we no longer wish for our subscribers  // to be notified  pubsub.unsubscribe( subscription );    // Once unsubscribed, this for example won't result in our  // messageLogger being executed as the subscriber is  // no longer listening  pubsub.publish( "inbox/newMessage", "Hello! are you still there?" ); |

Example: User-Interface Notifications

Next, let's imagine we have a web application responsible for displaying real-time stock information.

The application might have a grid for displaying the stock stats and a counter for displaying the last point of update. When the data model changes, the application will need to update the grid and counter. In this scenario, our subject (which will be publishing topics/notifications) is the data model and our subscribers are the grid and counter.

When our subscribers receive notification that the model itself has changed, they can update themselves accordingly.

In our implementation, our subscriber will listen to the topic "newDataAvailable" to find out if new stock information is available. If a new notification is published to this topic, it will trigger gridUpdate to add a new row to our grid containing this information. It will also update a *last updated* counter to log the last time data was added

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | // Return the current local time to be used in our UI later  getCurrentTime = function (){       var date = new Date(),           m = date.getMonth() + 1,           d = date.getDate(),           y = date.getFullYear(),           t = date.toLocaleTimeString().toLowerCase();            return (m + "/" + d + "/" + y + " " + t);  };    // Add a new row of data to our fictional grid component  function addGridRow( data ) {       // ui.grid.addRow( data );     console.log( "updated grid component with:" + data );    }    // Update our fictional grid to show the time it was last  // updated  function updateCounter( data ) {       // ui.grid.updateLastChanged( getCurrentTime() );     console.log( "data last updated at: " + getCurrentTime() + " with " + data);    }    // Update the grid using the data passed to our subscribers  gridUpdate = function( topic, data ){      if ( data !== undefined ) {       addGridRow( data );       updateCounter( data );     }    };    // Create a subscription to the newDataAvailable topic  var subscriber = pubsub.subscribe( "newDataAvailable", gridUpdate );    // The following represents updates to our data layer. This could be  // powered by ajax requests which broadcast that new data is available  // to the rest of the application.    // Publish changes to the gridUpdated topic representing new entries  pubsub.publish( "newDataAvailable", {    summary: "Apple made $5 billion",    identifier: "APPL",    stockPrice: 570.91  });    pubsub.publish( "newDataAvailable", {    summary: "Microsoft made $20 million",    identifier: "MSFT",    stockPrice: 30.85  }); |

Example: Decoupling applications using Ben Alman's Pub/Sub implementation

In the following movie ratings example, we'll be using Ben Alman's jQuery implementation of Publish/Subscribe to demonstrate how we can decouple a user interface. Notice how submitting a rating only has the effect of publishing the fact that new user and rating data is available.

It's left up to the subscribers to those topics to then delegate what happens with that data. In our case we're pushing that new data into existing arrays and then rendering them using the Underscore library's.template() method for templating.

**HTML/Templates**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | <script id="userTemplate" type="text/html">     <li><%= name="" %=""></li>  </script>      <script id="ratingsTemplate" type="text/html">     <li><strong><%= %=""></strong> was rated <%= rating="" %="">/5</li>  </script>      <div id="container">       <div class="sampleForm">         <p>             <label for="twitter\_handle">Twitter handle:</label>             <input type="text" id="twitter\_handle" />         </p>         <p>             <label for="movie\_seen">Name a movie you've seen this year:</label>             <input type="text" id="movie\_seen" />         </p>         <p>               <label for="movie\_rating">Rate the movie you saw:</label>             <select id="movie\_rating">                   <option value="1">1</option>                    <option value="2">2</option>                    <option value="3">3</option>                    <option value="4">4</option>                    <option value="5" selected>5</option>              </select>          </p>          <p>                <button id="add">Submit rating</button>          </p>      </div>            <div class="summaryTable">          <div id="users"><h3>Recent users</h3></div>          <div id="ratings"><h3>Recent movies rated</h3></div>      </div>     </div>    <!--%=--><!--%=--><!--%=--> |

**JavaScript**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | ;(function( $ ) {      // Pre-compile templates and "cache" them using closure    var      userTemplate = \_.template($( "#userTemplate" ).html()),      ratingsTemplate = \_.template($( "#ratingsTemplate" ).html());      // Subscribe to the new user topic, which adds a user    // to a list of users who have submitted reviews    $.subscribe( "/new/user", function( e, data ){        if( data ){          $('#users').append( userTemplate( data ));        }      });      // Subscribe to the new rating topic. This is composed of a title and    // rating. New ratings are appended to a running list of added user    // ratings.    $.subscribe( "/new/rating", function( e, data ){        if( data ){          $( "#ratings" ).append( ratingsTemplate( data ) );        }      });      // Handler for adding a new user    $("#add").on("click", function( e ) {        e.preventDefault();        var strUser = $("#twitter\_handle").val(),         strMovie = $("#movie\_seen").val(),         strRating = $("#movie\_rating").val();        // Inform the application a new user is available      $.publish( "/new/user", { name: strUser } );        // Inform the app a new rating is available      $.publish( "/new/rating", { title: strMovie, rating: strRating} );        });    })( jQuery ); |

Example: Decoupling an Ajax-based jQuery application

In our final example, we're going to take a practical look at how decoupling our code using Pub/Sub early on in the development process can save us some potentially painful refactoring later on.

Quite often in Ajax-heavy applications, once we've received a response to a request we want to achieve more than just one unique action. One could simply add all of their post-request logic into a success callback, but there are drawbacks to this approach.

Highly coupled applications sometimes increase the effort required to reuse functionality due to the increased inter-function/code dependency. What this means is that although keeping our post-request logic hardcoded in a callback might be fine if we're just trying to grab a result set once, it's not as appropriate when we want to make further Ajax-calls to the same data source (and different end-behavior) without rewriting parts of the code multiple times. Rather than having to go back through each layer that calls the same data-source and generalizing them later on, we can use pub/sub from the start and save time.

Using Observers, we can also easily separate application-wide notifications regarding different events down to whatever level of granularity we're comfortable with - something which can be less elegantly done using other patterns.

Notice how in our sample below, one topic notification is made when a user indicates they want to make a search query and another is made when the request returns and actual data is available for consumption. It's left up to the subscribers to then decide how to use knowledge of these events (or the data returned). The benefits of this are that, if we wanted, we could have 10 different subscribers utilizing the data returned in different ways but as far as the Ajax-layer is concerned, it doesn't care. Its sole duty is to request and return data then pass it on to whoever wants to use it. This separation of concerns can make the overall design of our code a little cleaner.

**HTML/Templates**:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | <form id="flickrSearch">       <input type="text" name="tag" id="query"/>       <input type="submit" name="submit" value="submit"/>    </form>        <div id="lastQuery"></div>    <ol id="searchResults"></ol>        <script id="resultTemplate" type="text/html">      <% \_.each(items,="" function(="" item="" ){="" %="">          <li><img src="<%= item.media.m="" %="">"/></li>      <% });%="">  </script>    <!--%--><!--%=--><!--%--> |

**JavaScript**:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | ;(function( $ ) {       // Pre-compile template and "cache" it using closure     var resultTemplate = \_.template($( "#resultTemplate" ).html());       // Subscribe to the new search tags topic     $.subscribe( "/search/tags", function( e, tags ) {         $( "#lastQuery" )                  .html("<p>Searched for:<strong>" + tags + "</strong></p>");     });       // Subscribe to the new results topic     $.subscribe( "/search/resultSet", function( e, results ){           $( "#searchResults" ).empty().append(resultTemplate( results ));       });       // Submit a search query and publish tags on the /search/tags topic     $( "#flickrSearch" ).submit( function( e ) {           e.preventDefault();         var tags = $(this).find( "#query").val();           if ( !tags ){          return;         }           $.publish( "/search/tags", [ $.trim(tags) ]);       });         // Subscribe to new tags being published and perform     // a search query using them. Once data has returned     // publish this data for the rest of the application     // to consume       $.subscribe("/search/tags", function( e, tags ) {           $.getJSON( "<http://api.flickr.com/services/feeds/photos_public.gne?jsoncallback=?>", {                tags: tags,                tagmode: "any",                format: "json"              },              function( data ){                  if( !data.items.length ) {                  return;                }                  $.publish( "/search/resultSet", { items: data.items } );         });       });      })( jQuery ); |

The Observer pattern is useful for decoupling a number of different scenarios in application design and if you haven't been using it, I recommend picking up one of the pre-written implementations mentioned today and just giving it a try out. It's one of the easier design patterns to get started with but also one of the most powerful.

The Mediator Pattern

In the section on the Observer pattern, we were introduced to a way of channeling multiple event sources through a single object. This is also known as Publish/Subscribe or Event Aggregation. It's common for developers to think of Mediators when faced with this problem, so let's explore how they differ.

The dictionary refers to a mediator as *a neutral party that assists in negotiations and conflict resolution*. In our world, a mediator is a behavioral design pattern that allows us to expose a unified interface through which the different parts of a system may communicate.

If it appears a system has too many direct relationships between components, it may be time to have a central point of control that components communicate through instead. The Mediator promotes loose coupling by ensuring that instead of components referring to each other explicitly, their interaction is handled through this central point. This can help us decouple systems and improve the potential for component reusability.

A real-world analogy could be a typical airport traffic control system. A tower (Mediator) handles what planes can take off and land because all communications (notifications being listened out for or broadcast) are done from the planes to the control tower, rather than from plane-to-plane. A centralized controller is key to the success of this system and that's really the role a Mediator plays in software design.

Another analogy would be DOM event bubbling and event delegation. If all subscriptions in a system are made against the document rather than individual nodes, the document effectively serves as a Mediator. Instead of binding to the events of the individual nodes, a higher level object is given the responsibility of notifying subscribers about interaction events.

When it comes to the Mediator and Event Aggregator patterns, there are some times where it may look like the patterns are interchangeable due to implementation similarities. However, the semantics and intent of these patterns are very different.

And even if the implementations both use some of the same core constructs, I believe there is a distinct difference between them. I also believe they should not be interchanged or confused in communication because of the differences.

**A Simple Mediator**

A Mediator is an object that coordinates interactions (logic and behavior) between multiple objects. It makes decisions on when to call which objects, based on the actions (or inaction) of other objects and input.

You can write a mediator using a single line of code:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1 | var mediator = {}; |

Yes, of course this is just an object literal in JavaScript. Once again, we’re talking about semantics here. The purpose of the mediator is to control the workflow between objects and we really don’t need anything more than an object literal to do this.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | var orgChart = {      addNewEmployee: function(){        // getEmployeeDetail provides a view that users interact with      var employeeDetail = this.getEmployeeDetail();        // when the employee detail is complete, the mediator (the 'orgchart' object)      // decides what should happen next      employeeDetail.on("complete", function(employee){          // set up additional objects that have additional events, which are used        // by the mediator to do additional things        var managerSelector = this.selectManager(employee);        managerSelector.on("save", function(employee){          employee.save();        });        });    },      // ...  } |

This example shows a very basic implementation of a mediator object with some utility methods that can trigger and subscribe to events.

I’ve often referred to this type of object as a “workflow” object in the past, but the truth is that it is a mediator. It is an object that handles the workflow between many other objects, aggregating the responsibility of that workflow knowledge into a single object. The result is workflow that is easier to understand and maintain.

**Similarities And Differences**

There are, without a doubt, similarities between the event aggregator and mediator examples that I’ve shown here. The similarities boil down to two primary items: events and third-party objects. These differences are superficial at best, though. When we dig into the intent of the pattern and see that the implementations can be dramatically different, the nature of the patterns become more apparent.

**Events**

Both the event aggregator and mediator use events, in the above examples. An event aggregator obviously deals with events – it’s in the name after all. The mediator only uses events because it makes life easy when dealing with modern JavaScript webapp frameworks. There is nothing that says a mediator must be built with events. You can build a mediator with callback methods, by handing the mediator reference to the child object, or by any of a number of other means.

The difference, then, is why these two patterns are both using events. The event aggregator, as a pattern, is designed to deal with events. The mediator, though, only uses them because it’s convenient.

**Third-Party Objects**

Both the event aggregator and mediator, by design, use a third-party object to facilitate things. The event aggregator itself is a third-party to the event publisher and the event subscriber. It acts as a central hub for events to pass through. The mediator is also a third party to other objects, though. So where is the difference? Why don’t we call an event aggregator a mediator? The answer largely comes down to where the application logic and workflow is coded.

In the case of an event aggregator, the third party object is there only to facilitate the pass-through of events from an unknown number of sources to an unknown number of handlers. All workflow and business logic that needs to be kicked off is put directly into the object that triggers the events and the objects that handle the events.

In the case of the mediator, though, the business logic and workflow is aggregated into the mediator itself. The mediator decides when an object should have its methods called and attributes updated based on factors that the mediator knows about. It encapsulates the workflow and process, coordinating multiple objects to produce the desired system behaviour. The individual objects involved in this workflow each know how to perform their own task. But it’s the mediator that tells the objects when to perform the tasks by making decisions at a higher level than the individual objects.

An event aggregator facilitates a “fire and forget” model of communication. The object triggering the event doesn’t care if there are any subscribers. It just fires the event and moves on. A mediator, though, might use events to make decisions, but it is definitely not “fire and forget”. A mediator pays attention to a known set of input or activities so that it can facilitate and coordinate additional behavior with a known set of actors (objects).

**Relationships: When To Use Which**

Understanding the similarities and differences between an event aggregator and mediator is important for semantic reasons. It’s equally as important to understand when to use which pattern, though. The basic semantics and intent of the patterns does inform the question of when, but actual experience in using the patterns will help you understand the more subtle points and nuanced decisions that have to be made.

**Event Aggregator Use**

In general, an event aggregator is used when you either have too many objects to listen to directly, or you have objects that are entirely unrelated.

When two objects have a direct relationship already – say, a parent view and child view – then there might be little benefit in using an event aggregator. Have the child view trigger an event and the parent view can handle the event. In JavaScript framework terms, this is most commonly seen in Backbone’s Collection and Model, where all Model events are bubbled up to and through its parent Collection. A Collection often uses model events to modify the state of itself or other models. Handling “selected” items in a collection is a good example of this.

jQuery’s on method as an event aggregator is a great example of too many objects to listen to. If you have 10, 20 or 200 DOM elements that can trigger a “click” event, it might be a bad idea to set up a listener on all of them individually. This could quickly deteriorate performance of the application and user experience. Instead, using jQuery’s on method allows us to aggregate all of the events and reduce the overhead of 10, 20, or 200 event handlers down to 1.

Indirect relationships are also a great time to use event aggregators. In modern applications, it is very common to have multiple view objects that need to communicate, but have no direct relationship. For example, a menu system might have a view that handles the menu item clicks. But we don’t want the menu to be directly tied to the content views that show all of the details and information when a menu item is clicked. Having the content and menu coupled together would make the code very difficult to maintain, in the long run. Instead, we can use an event aggregator to trigger “menu:click:foo” events, and have a “foo” object handle the click event to show its content on the screen.

**Mediator Use**

A mediator is best applied when two or more objects have an indirect working relationship, and business logic or workflow needs to dictate the interactions and coordination of these objects.

A wizard interface is a good example of this, as shown with the “orgChart” example, above. There are multiple views that facilitate the entire workflow of the wizard. Rather than tightly coupling the view together by having them reference each other directly, we can decouple them and more explicitly model the workflow between them by introducing a mediator.

The mediator extracts the workflow from the implementation details and creates a more natural abstraction at a higher level, showing us at a much faster glance what that workflow is. We no longer have to dig into the details of each view in the workflow, to see what the workflow actually is.

**Event Aggregator (Pub/Sub) And Mediator Together**

The crux of the difference between an event aggregator and a mediator, and why these pattern names should not be interchanged with each other, is illustrated best by showing how they can be used together. The menu example for an event aggregator is the perfect place to introduce a mediator as well.

Clicking a menu item may trigger a series of changes throughout an application. Some of these changes will be independent of others, and using an event aggregator for this makes sense. Some of these changes may be internally related to each other, though, and may use a mediator to enact those changes.

A mediator, then, could be set up to listen to the event aggregator. It could run its logic and process to facilitate and coordinate many objects that are related to each other, but unrelated to the original event source.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | var MenuItem = MyFrameworkView.extend({      events: {      "click .thatThing": "clickedIt"    },      clickedIt: function(e){      e.preventDefault();        // assume this triggers "menu:click:foo"      MyFramework.trigger("menu:click:" + this.model.get("name"));    }    });    // ... somewhere else in the app    var MyWorkflow = function(){    MyFramework.on("menu:click:foo", this.doStuff, this);  };    MyWorkflow.prototype.doStuff = function(){    // instantiate multiple objects here.    // set up event handlers for those objects.    // coordinate all of the objects into a meaningful workflow.  }; |

In this example, when the MenuItem with the right model is clicked, the “menu:click:foo” event will be triggered. An instance of the “MyWorkflow” object, assuming one is already instantiated, will handle this specific event and will coordinate all of the objects that it knows about, to create the desired user experience and workflow.

An event aggregator and a mediator have been combined to create a much more meaningful experience in both the code and the application itself. We now have a clean separation between the menu and the workflow through an event aggregator and we are still keeping the workflow itself clean and maintainable through the use of a mediator.

Advantages & Disadvantages

The largest benefit of the Mediator pattern is that it reduces the communication channels needed between objects or components in a system from many to many to just many to one. Adding new publishers and subscribers is relatively easy due to the level of decoupling present.

Perhaps the biggest downside of using the pattern is that it can introduce a single point of failure. Placing a Mediator between modules can also cause a performance hit as they are always communicating indirectly. Because of the nature of loose coupling, it's difficult to establish how a system might react by only looking at the broadcasts.

That said, it's useful to remind ourselves that decoupled systems have a number of other benefits - if our modules communicated with each other directly, changes to modules (e.g another module throwing an exception) could easily have a domino effect on the rest of our application. This problem is less of a concern with decoupled systems.

At the end of the day, tight coupling causes all kinds of headaches and this is just another alternative solution, but one which can work very well if implemented correctly.

Mediator Vs. Facade

We will be covering the Facade pattern shortly, but for reference purposes some developers may also wonder whether there are similarities between the Mediator and Facade patterns. They do both abstract the functionality of existing modules, but there are some subtle differences.

The Mediator centralizes communication between modules where it's explicitly referenced by these modules. In a sense this is multidirectional. The Facade however just defines a simpler interface to a module or system but doesn't add any additional functionality. Other modules in the system aren't directly aware of the concept of a facade and could be considered unidirectional.

The Prototype Pattern

The GoF refer to the prototype pattern as one which creates objects based on a template of an existing object through cloning.

We can think of the prototype pattern as being based on prototypal inheritance where we create objects which act as prototypes for other objects. The prototype object itself is effectively used as a blueprint for each object the constructor creates. If the prototype of the constructor function used contains a property called name for example (as per the code sample lower down), then each object created by that same constructor will also have this same property.

Reviewing the definitions for this pattern in existing (non-JavaScript) literature, we **may** find references to classes once again. The reality is that prototypal inheritance avoids using classes altogether. There isn't a "definition" object nor a core object in theory. We're simply creating copies of existing functional objects.

One of the benefits of using the prototype pattern is that we're working with the prototypal strengths JavaScript has to offer natively rather than attempting to imitate features of other languages. With other design patterns, this isn't always the case.

Not only is the pattern an easy way to implement inheritance, but it can also come with a performance boost as well: when defining a function in an object, they're all created by reference (so all child objects point to the same function) instead of creating their own individual copies.

For those interested, real prototypal inheritance, as defined in the ECMAScript 5 standard, requires the use of Object.create (which we previously looked at earlier in this section). To remind ourselves,Object.create creates an object which has a specified prototype and optionally contains specified properties as well (e.g Object.create( prototype, optionalDescriptorObjects )).

We can see this demonstrated in the example below:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | var myCar = {      name: "Ford Escort",      drive: function () {      console.log( "Weeee. I'm driving!" );    },      panic: function () {      console.log( "Wait. How do you stop this thing?" );    }    };    // Use Object.create to instantiate a new car  var yourCar = Object.create( myCar );    // Now we can see that one is a prototype of the other  console.log( yourCar.name ); |

Object.create also allows us to easily implement advanced concepts such as differential inheritance where objects are able to directly inherit from other objects. We saw earlier that Object.create allows us to initialise object properties using the second supplied argument. For example:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | var vehicle = {    getModel: function () {      console.log( "The model of this vehicle is.." + this.model );    }  };    var car = Object.create(vehicle, {      "id": {      value: MY\_GLOBAL.nextId(),      // writable:false, configurable:false by default      enumerable: true    },      "model": {      value: "Ford",      enumerable: true    }    }); |

Here the properties can be initialized on the second argument of Object.create using an object literal with a syntax similar to that used by the Object.defineProperties and Object.defineProperty methods that we looked at previously.

It is worth noting that prototypal relationships can cause trouble when enumerating properties of objects and (as Crockford recommends) wrapping the contents of the loop in a hasOwnProperty() check.

If we wish to implement the prototype pattern without directly using Object.create, we can simulate the pattern as per the above example as follows:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | var vehiclePrototype = {      init: function ( carModel ) {      this.model = carModel;    },      getModel: function () {      console.log( "The model of this vehicle is.." + this.model);    }  };      function vehicle( model ) {      function F() {};    F.prototype = vehiclePrototype;      var f = new F();      f.init( model );    return f;    }    var car = vehicle( "Ford Escort" );  car.getModel(); |

**Note:** This alternative does not allow the user to define read-only properties in the same manner (as the vehiclePrototype may be altered if not careful).

A final alternative implementation of the Prototype pattern could be the following:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var beget = (function () {        function F() {}        return function ( proto ) {          F.prototype = proto;          return new F();      };  })(); |

One could reference this method from the vehicle function. Note, however that vehicle here is emulating a constructor, since the prototype pattern does not include any notion of initialization beyond linking an object to a prototype.

The Command Pattern

The Command pattern aims to encapsulate method invocation, requests or operations into a single object and gives us the ability to both parameterize and pass method calls around that can be executed at our discretion. In addition, it enables us to decouple objects invoking the action from the objects which implement them, giving us a greater degree of overall flexibility in swapping out concrete *classes*(objects).

*Concrete* classes are best explained in terms of class-based programming languages and are related to the idea of abstract classes. An *abstract* class defines an interface, but doesn't necessarily provide implementations for all of its member functions. It acts as a base class from which others are derived. A derived class which implements the missing functionality is called a *concrete* class.

The general idea behind the Command pattern is that it provides us a means to separate the responsibilities of issuing commands from anything executing commands, delegating this responsibility to different objects instead.

Implementation wise, simple command objects bind together both an action and the object wishing to invoke the action. They consistently include an execution operation (such as run() or execute()). All Command objects with the same interface can easily be swapped as needed and this is considered one of the larger benefits of the pattern.

To demonstrate the Command pattern we're going to create a simple car purchasing service.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | (function(){      var carManager = {        // request information      requestInfo: function( model, id ){        return "The information for " + model + " with ID " + id + " is foobar";      },        // purchase the car      buyVehicle: function( model, id ){        return "You have successfully purchased Item " + id + ", a " + model;      },        // arrange a viewing      arrangeViewing: function( model, id ){        return "You have successfully booked a viewing of " + model + " ( " + id + " ) ";      }      };    })(); |

Taking a look at the above code, it would be trivial to invoke our carManager methods by directly accessing the object. We would all be forgiven for thinking there is nothing wrong with this - technically, it's completely valid JavaScript. There are however scenarios where this may be disadvantageous.

For example, imagine if the core API behind the carManager changed. This would require all objects directly accessing these methods within our application to also be modified. This could be viewed as a layer of coupling which effectively goes against the OOP methodology of loosely coupling objects as much as possible. Instead, we could solve this problem by abstracting the API away further.

Let's now expand on our carManager so that our application of the Command pattern results in the following: accept any named methods that can be performed on the carManager object, passing along any data that might be used such as the Car model and ID.

Here is what we would like to be able to achieve:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1 | carManager.execute( "buyVehicle", "Ford Escort", "453543" ); |

As per this structure we should now add a definition for the carManager.execute method as follows:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3 | carManager.execute = function ( name ) {      return carManager[name] && carManager[name].apply( carManager, [].slice.call(arguments, 1) );  }; |

Our final sample calls would thus look as follows:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4 | carManager.execute( "arrangeViewing", "Ferrari", "14523" );  carManager.execute( "requestInfo", "Ford Mondeo", "54323" );  carManager.execute( "requestInfo", "Ford Escort", "34232" );  carManager.execute( "buyVehicle", "Ford Escort", "34232" ); |

The Facade Pattern

When we put up a facade, we present an outward appearance to the world which may conceal a very different reality. This was the inspiration for the name behind the next pattern we're going to review - the Facade pattern. This pattern provides a convenient higher-level interface to a larger body of code, hiding its true underlying complexity. Think of it as simplifying the API being presented to other developers, something which almost always improves usability.

Facades are a structural pattern which can often be seen in JavaScript libraries like jQuery where, although an implementation may support methods with a wide range of behaviors, only a "facade" or limited abstraction of these methods is presented to the public for use.

This allows us to interact with the Facade directly rather than the subsystem behind the scenes. Whenever we use jQuery's $(el).css() or $(el).animate() methods, we're actually using a Facade - the simpler public interface that avoids us having to manually call the many internal methods in jQuery core required to get some behavior working. This also avoids the need to manually interact with DOM APIs and maintain state variables.

The jQuery core methods should be considered intermediate abstractions. The more immediate burden to developers is the DOM API and facades are what make the jQuery library so easy to use.

To build on what we've learned, the Facade pattern both simplifies the interface of a class and it also decouples the class from the code that utilizes it. This gives us the ability to indirectly interact with subsystems in a way that can sometimes be less prone to error than accessing the subsystem directly. A Facade's advantages include ease of use and often a small size-footprint in implementing the pattern.

Let’s take a look at the pattern in action. This is an unoptimized code example, but here we're utilizing a Facade to simplify an interface for listening to events cross-browser. We do this by creating a common method that can be used in one’s code which does the task of checking for the existence of features so that it can provide a safe and cross-browser compatible solution.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var addMyEvent = function( el,ev,fn ){       if( el.addEventListener ){              el.addEventListener( ev,fn, false );        }else if(el.attachEvent){              el.attachEvent( "on" + ev, fn );        } else{             el["on" + ev] = fn;      }    }; |

In a similar manner, we're all familiar with jQuery's $(document).ready(..). Internally, this is actually being powered by a method called bindReady(), which is doing this:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | bindReady: function() {      ...      if ( document.addEventListener ) {        // Use the handy event callback        document.addEventListener( "DOMContentLoaded", DOMContentLoaded, false );          // A fallback to window.onload, that will always work        window.addEventListener( "load", jQuery.ready, false );        // If IE event model is used      } else if ( document.attachEvent ) {          document.attachEvent( "onreadystatechange", DOMContentLoaded );          // A fallback to window.onload, that will always work        window.attachEvent( "onload", jQuery.ready );                 ... |

This is another example of a Facade, where the rest of the world simply uses the limited interface exposed by $(document).ready(..) and the more complex implementation powering it is kept hidden from sight.

Facades don't just have to be used on their own, however. They can also be integrated with other patterns such as the Module pattern. As we can see below, our instance of the module patterns contains a number of methods which have been privately defined. A Facade is then used to supply a much simpler API to accessing these methods:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | var module = (function() {        var \_private = {          i: 5,          get: function() {              console.log( "current value:" + this.i);          },          set: function( val ) {              this.i = val;          },          run: function() {              console.log( "running" );          },          jump: function(){              console.log( "jumping" );          }      };        return {            facade: function( args ) {              \_private.set(args.val);              \_private.get();              if ( args.run ) {                  \_private.run();              }          }      };  }());      // Outputs: "current value: 10" and "running"  module.facade( {run: true, val: 10} ); |

In this example, calling module.facade() will actually trigger a set of private behavior within the module, but again, the user isn't concerned with this. we've made it much easier for them to consume a feature without needing to worry about implementation-level details.

Notes on abstraction

Facades generally have few disadvantages, but one concern worth noting is performance. Namely, one must determine whether there is an implicit cost to the abstraction a Facade offers to our implementation and if so, whether this cost is justifiable. Going back to the jQuery library, most of us are aware that both getElementById("identifier") and $("#identifier") can be used to query an element on a page by its ID.

Did you know however that getElementById() on its own is significantly faster by a high order of magnitude? Take a look at this jsPerf test to see results on a per-browser level:<http://jsperf.com/getelementbyid-vs-jquery-id>. Now of course, we have to keep in mind that jQuery (and Sizzle - its selector engine) are doing a lot more behind the scenes to optimize our query (and that a jQuery object, not just a DOM node is returned).

The challenge with this particular Facade is that in order to provide an elegant selector function capable of accepting and parsing multiple types of queries, there is an implicit cost of abstraction. The user isn't required to access jQuery.getById("identifier") or jQuery.getByClass("identifier") and so on. That said, the trade-off in performance has been tested in practice over the years and given the success of jQuery, a simple Facade actually worked out very well for the team.

When using the pattern, try to be aware of any performance costs involved and make a call on whether they are worth the level of abstraction offered.

The Factory Pattern

The Factory pattern is another creational pattern concerned with the notion of creating objects. Where it differs from the other patterns in its category is that it doesn't explicitly require us use a constructor. Instead, a Factory can provide a generic interface for creating objects, where we can specify the type of factory object we wish to be created.

Imagine that we have a UI factory where we are asked to create a type of UI component. Rather than creating this component directly using the new operator or via another creational constructor, we ask a Factory object for a new component instead. We inform the Factory what type of object is required (e.g "Button", "Panel") and it instantiates this, returning it to us for use.

This is particularly useful if the object creation process is relatively complex, e.g. if it strongly depends on dynamic factors or application configuration.

Examples of this pattern can be found in UI libraries such as ExtJS where the methods for creating objects or components may be further subclassed.

The following is an example that builds upon our previous snippets using the Constructor pattern logic to define cars. It demonstrates how a Vehicle *Factory* may be implemented using the Factory pattern:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | // Types.js - Constructors used behind the scenes    // A constructor for defining new cars  function Car( options ) {      // some defaults    this.doors = options.doors || 4;    this.state = options.state || "brand new";    this.color = options.color || "silver";    }    // A constructor for defining new trucks  function Truck( options){      this.state = options.state || "used";    this.wheelSize = options.wheelSize || "large";    this.color = options.color || "blue";  }      // FactoryExample.js    // Define a skeleton vehicle factory  function VehicleFactory() {}    // Define the prototypes and utilities for this factory    // Our default vehicleClass is Car  VehicleFactory.prototype.vehicleClass = Car;    // Our Factory method for creating new Vehicle instances  VehicleFactory.prototype.createVehicle = function ( options ) {      switch(options.vehicleType){      case "car":        this.vehicleClass = Car;        break;      case "truck":        this.vehicleClass = Truck;        break;      //defaults to VehicleFactory.prototype.vehicleClass (Car)    }      return new this.vehicleClass( options );    };    // Create an instance of our factory that makes cars  var carFactory = new VehicleFactory();  var car = carFactory.createVehicle( {              vehicleType: "car",              color: "yellow",              doors: 6 } );    // Test to confirm our car was created using the vehicleClass/prototype Car    // Outputs: true  console.log( car instanceof Car );    // Outputs: Car object of color "yellow", doors: 6 in a "brand new" state  console.log( car ); |

**Approach #1: Modify a VehicleFactory instance to use the Truck class**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | var movingTruck = carFactory.createVehicle( {                        vehicleType: "truck",                        state: "like new",                        color: "red",                        wheelSize: "small" } );    // Test to confirm our truck was created with the vehicleClass/prototype Truck    // Outputs: true  console.log( movingTruck instanceof Truck );    // Outputs: Truck object of color "red", a "like new" state  // and a "small" wheelSize  console.log( movingTruck ); |

**Approach #2: Subclass VehicleFactory to create a factory class that builds Trucks**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | function TruckFactory () {}  TruckFactory.prototype = new VehicleFactory();  TruckFactory.prototype.vehicleClass = Truck;    var truckFactory = new TruckFactory();  var myBigTruck = truckFactory.createVehicle( {                      state: "omg..so bad.",                      color: "pink",                      wheelSize: "so big" } );    // Confirms that myBigTruck was created with the prototype Truck  // Outputs: true  console.log( myBigTruck instanceof Truck );    // Outputs: Truck object with the color "pink", wheelSize "so big"  // and state "omg. so bad"  console.log( myBigTruck ); |

When To Use The Factory Pattern

The Factory pattern can be especially useful when applied to the following situations:

* When our object or component setup involves a high level of complexity
* When we need to easily generate different instances of objects depending on the environment we are in
* When we're working with many small objects or components that share the same properties
* When composing objects with instances of other objects that need only satisfy an API contract (aka, duck typing) to work. This is useful for decoupling.

When Not To Use The Factory Pattern

When applied to the wrong type of problem, this pattern can introduce an unnecessarily great deal of complexity to an application. Unless providing an interface for object creation is a design goal for the library or framework we are writing, I would suggest sticking to explicit constructors to avoid the unnecessary overhead.

Due to the fact that the process of object creation is effectively abstracted behind an interface, this can also introduce problems with unit testing depending on just how complex this process might be.

Abstract Factories

It is also useful to be aware of the **Abstract Factory** pattern, which aims to encapsulate a group of individual factories with a common goal. It separates the details of implementation of a set of objects from their general usage.

An Abstract Factory should be used where a system must be independent from the way the objects it creates are generated or it needs to work with multiple types of objects.

An example which is both simple and easier to understand is a vehicle factory, which defines ways to get or register vehicles types. The abstract factory can be named abstractVehicleFactory. The Abstract factory will allow the definition of types of vehicle like "car" or "truck" and concrete factories will implement only classes that fulfill the vehicle contract (e.g Vehicle.prototype.drive andVehicle.prototype.breakDown).

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | var abstractVehicleFactory = (function () {      // Storage for our vehicle types    var types = {};      return {        getVehicle: function ( type, customizations ) {            var Vehicle = types[type];              return (Vehicle ? new Vehicle(customizations) : null);        },          registerVehicle: function ( type, Vehicle ) {            var proto = Vehicle.prototype;              // only register classes that fulfill the vehicle contract            if ( proto.drive && proto.breakDown ) {                types[type] = Vehicle;            }              return abstractVehicleFactory;        }    };  })();      // Usage:    abstractVehicleFactory.registerVehicle( "car", Car );  abstractVehicleFactory.registerVehicle( "truck", Truck );    // Instantiate a new car based on the abstract vehicle type  var car = abstractVehicleFactory.getVehicle( "car", {              color: "lime green",              state: "like new" } );    // Instantiate a new truck in a similar manner  var truck = abstractVehicleFactory.getVehicle( "truck", {              wheelSize: "medium",              color: "neon yellow" } ); |

The Mixin Pattern

In traditional programming languages such as C++ and Lisp, Mixins are classes which offer functionality that can be easily inherited by a sub-class or group of sub-classes for the purpose of function re-use.

Sub-classing

For developers unfamiliar with sub-classing, we will go through a brief beginners primer on them before diving into Mixins and Decorators further.

*Sub-classing* is a term that refers to inheriting properties for a new object from a base or *superclass* object. In traditional object-oriented programming, a class B is able to extend another class A. Here we consider A a superclass and B a subclass of A. As such, all instances of B inherit the methods from A.B is however still able to define its own methods, including those that override methods originally defined by A.

Should B need to invoke a method in A that has been overridden, we refer to this as method chaining. Should B need to invoke the constructor A (the superclass), we call this constructor chaining.

In order to demonstrate sub-classing, we first need a base object that can have new instances of itself created. let's model this around the concept of a person.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var Person = function( firstName, lastName ){      this.firstName = firstName;    this.lastName = lastName;    this.gender = "male";    }; |

Next, we'll want to specify a new class (object) that's a subclass of the existing Person object. Let us imagine we want to add distinct properties to distinguish a Person from a Superhero whilst inheriting the properties of the Person "superclass". As superheroes share many common traits with normal people (e.g. name, gender), this should hopefully illustrate how sub-classing works adequately.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | // a new instance of Person can then easily be created as follows:  var clark = new Person( "Clark", "Kent" );    // Define a subclass constructor for for "Superhero":  var Superhero = function( firstName, lastName, powers ){        // Invoke the superclass constructor on the new object      // then use .call() to invoke the constructor as a method of      // the object to be initialized.        Person.call( this, firstName, lastName );        // Finally, store their powers, a new array of traits not found in a normal "Person"      this.powers = powers;  };    Superhero.prototype = Object.create( Person.prototype );  var superman = new Superhero( "Clark", "Kent", ["flight","heat-vision"] );  console.log( superman );    // Outputs Person attributes as well as powers |

The Superhero constructor creates an object which descends from Person. Objects of this type have attributes of the objects that are above it in the chain and if we had set default values in the Personobject, Superhero is capable of overriding any inherited values with values specific to it's object.

Mixins

In JavaScript, we can look at inheriting from Mixins as a means of collecting functionality through extension. Each new object we define has a prototype from which it can inherit further properties. Prototypes can inherit from other object prototypes but, even more importantly, can define properties for any number of object instances. We can leverage this fact to promote function re-use.

Mixins allow objects to borrow (or inherit) functionality from them with a minimal amount of complexity. As the pattern works well with JavaScripts object prototypes, it gives us a fairly flexible way to share functionality from not just one Mixin, but effectively many through multiple inheritance.

They can be viewed as objects with attributes and methods that can be easily shared across a number of other object prototypes. Imagine that we define a Mixin containing utility functions in a standard object literal as follows:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | var myMixins = {      moveUp: function(){      console.log( "move up" );    },      moveDown: function(){      console.log( "move down" );    },      stop: function(){      console.log( "stop! in the name of love!" );    }    }; |

We can then easily extend the prototype of existing constructor functions to include this behavior using a helper such as the Underscore.js \_.extend() method:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | // A skeleton carAnimator constructor  function CarAnimator(){    this.moveLeft = function(){      console.log( "move left" );    };  }    // A skeleton personAnimator constructor  function PersonAnimator(){    this.moveRandomly = function(){ /\*..\*/ };  }    // Extend both constructors with our Mixin  \_.extend( CarAnimator.prototype, myMixins );  \_.extend( PersonAnimator.prototype, myMixins );    // Create a new instance of carAnimator  var myAnimator = new CarAnimator();  myAnimator.moveLeft();  myAnimator.moveDown();  myAnimator.stop();    // Outputs:  // move left  // move down  // stop! in the name of love! |

As we can see, this allows us to easily "mix" in common behaviour into object constructors fairly trivially.

In the next example, we have two constructors: a Car and a Mixin. What we're going to do is augment (another way of saying extend) the Car so that it can inherit specific methods defined in the Mixin, namely driveForward() and driveBackward(). This time we won't be using Underscore.js.

Instead, this example will demonstrate how to augment a constructor to include functionality without the need to duplicate this process for every constructor function we may have.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87 | // Define a simple Car constructor  var Car = function ( settings ) {        this.model = settings.model || "no model provided";      this.color = settings.color || "no colour provided";    };    // Mixin  var Mixin = function () {};    Mixin.prototype = {        driveForward: function () {          console.log( "drive forward" );      },        driveBackward: function () {          console.log( "drive backward" );      },        driveSideways: function () {          console.log( "drive sideways" );      }    };      // Extend an existing object with a method from another  function augment( receivingClass, givingClass ) {        // only provide certain methods      if ( arguments[2] ) {          for ( var i = 2, len = arguments.length; i < len; i++ ) {              receivingClass.prototype[arguments[i]] = givingClass.prototype[arguments[i]];          }      }      // provide all methods      else {          for ( var methodName in givingClass.prototype ) {                // check to make sure the receiving class doesn't              // have a method of the same name as the one currently              // being processed              if ( !Object.hasOwnProperty.call(receivingClass.prototype, methodName) ) {                  receivingClass.prototype[methodName] = givingClass.prototype[methodName];              }                // Alternatively (check prototype chain as well):              // if ( !receivingClass.prototype[methodName] ) {              // receivingClass.prototype[methodName] = givingClass.prototype[methodName];              // }          }      }  }      // Augment the Car constructor to include "driveForward" and "driveBackward"  augment( Car, Mixin, "driveForward", "driveBackward" );    // Create a new Car  var myCar = new Car({      model: "Ford Escort",      color: "blue"  });    // Test to make sure we now have access to the methods  myCar.driveForward();  myCar.driveBackward();    // Outputs:  // drive forward  // drive backward    // We can also augment Car to include all functions from our mixin  // by not explicitly listing a selection of them  augment( Car, Mixin );    var mySportsCar = new Car({      model: "Porsche",      color: "red"  });    mySportsCar.driveSideways();    // Outputs:  // drive sideways |

Advantages & Disadvantages

Mixins assist in decreasing functional repetition and increasing function re-use in a system. Where an application is likely to require shared behaviour across object instances, we can easily avoid any duplication by maintaining this shared functionality in a Mixin and thus focusing on implementing only the functionality in our system which is truly distinct.

That said, the downsides to Mixins are a little more debatable. Some developers feel that injecting functionality into an object prototype is a bad idea as it leads to both prototype pollution and a level of uncertainty regarding the origin of our functions. In large systems this may well be the case.

I would argue that strong documentation can assist in minimizing the amount of confusion regarding the source of mixed in functions, but as with every pattern, if care is taken during implementation we should be okay.

The Decorator Pattern

Decorators are a structural design pattern that aim to promote code re-use. Similar to Mixins, they can be considered another viable alternative to object sub-classing.

Classically, Decorators offered the ability to add behaviour to existing classes in a system dynamically. The idea was that the *decoration* itself wasn't essential to the base functionality of the class, otherwise it would be baked into the *superclass* itself.

They can be used to modify existing systems where we wish to add additional features to objects without the need to heavily modify the underlying code using them. A common reason why developers use them is their applications may contain features requiring a large quantity of distinct types of object. Imagine having to define hundreds of different object constructors for say, a JavaScript game.

The object constructors could represent distinct player types, each with differing capabilities. A *Lord of the Rings* game could require constructors for Hobbit, Elf, Orc, Wizard, Mountain Giant, Stone Giant and so on, but there could easily be hundreds of these. If we then factored in capabilities, imagine having to create sub-classes for each combination of capability type e.g HobbitWithRing,HobbitWithSword,HobbitWithRingAndSword and so on.This isn't very practical and certainly isn't manageable when we factor in a growing number of different abilities.

The Decorator pattern isn't heavily tied to how objects are created but instead focuses on the problem of extending their functionality. Rather than just relying on prototypal inheritance, we work with a single base object and progressively add decorator objects which provide the additional capabilities. The idea is that rather than sub-classing, we add (decorate) properties or methods to a base object so it's a little more streamlined.

Adding new attributes to objects in JavaScript is a very straight-forward process so with this in mind, a very simplistic decorator may be implemented as follows:

Example 1: Decorating Constructors With New Functionality

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | // A vehicle constructor  function Vehicle( vehicleType ){        // some sane defaults      this.vehicleType = vehicleType || "car";      this.model = "default";      this.license = "00000-000";    }    // Test instance for a basic vehicle  var testInstance = new Vehicle( "car" );  console.log( testInstance );    // Outputs:  // vehicle: car, model:default, license: 00000-000    // Lets create a new instance of vehicle, to be decorated  var truck = new Vehicle( "truck" );    // New functionality we're decorating vehicle with  truck.setModel = function( modelName ){      this.model = modelName;  };    truck.setColor = function( color ){      this.color = color;  };    // Test the value setters and value assignment works correctly  truck.setModel( "CAT" );  truck.setColor( "blue" );    console.log( truck );    // Outputs:  // vehicle:truck, model:CAT, color: blue    // Demonstrate "vehicle" is still unaltered  var secondInstance = new Vehicle( "car" );  console.log( secondInstance );    // Outputs:  // vehicle: car, model:default, license: 00000-000 |

This type of simplistic implementation is functional, but it doesn't really demonstrate all of the strengths Decorators have to offer. For this, we're first going to go through my variation of the Coffee example from an excellent book called *Head First Design Patterns* by Freeman, Sierra and Bates, which is modeled around a Macbook purchase.

Example 2: Decorating Objects With Multiple Decorators

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | // The constructor to decorate  function MacBook() {      this.cost = function () { return 997; };    this.screenSize = function () { return 11.6; };    }    // Decorator 1  function memory( macbook ) {      var v = macbook.cost();    macbook.cost = function() {      return v + 75;    };    }    // Decorator 2  function engraving( macbook ){      var v = macbook.cost();    macbook.cost = function(){      return v + 200;    };    }    // Decorator 3  function insurance( macbook ){      var v = macbook.cost();    macbook.cost = function(){       return v + 250;    };    }    var mb = new MacBook();  memory( mb );  engraving( mb );  insurance( mb );    // Outputs: 1522  console.log( mb.cost() );    // Outputs: 11.6  console.log( mb.screenSize() ); |

In the above example, our Decorators are overriding the MacBook() super-class objects .cost() function to return the current price of the Macbook plus the cost of the upgrade being specified.

It's considered a decoration as the original Macbook objects constructor methods which are not overridden (e.g. screenSize()) as well as any other properties which we may define as a part of theMacbook remain unchanged and intact.

There isn't really a defined *interface* in the above example and we're shifting away the responsibility of ensuring an object meets an interface when moving from the creator to the receiver.

Pseudo-classical Decorators

We're now going to examine a variation of the Decorator first presented in a JavaScript form in *Pro JavaScript Design Patterns* (PJDP) by Dustin Diaz and Ross Harmes.

Unlike some of the examples from earlier, Diaz and Harmes stick more closely to how decorators are implemented in other programming languages (such as Java or C++) using the concept of an "interface", which we will define in more detail shortly.

**Note:** This particular variation of the Decorator pattern is provided for reference purposes. If finding it overly complex, I recommend opting for one of the simpler implementations covered earlier.

Interfaces

PJDP describes the Decorator as a pattern that is used to transparently wrap objects inside other objects of the same interface. An interface is a way of defining the methods an object **should** have, however, it doesn't actually directly specify how those methods should be implemented.

They can also indicate what parameters the methods take, but this is considered optional.

So, why would we use an interface in JavaScript? The idea is that they're self-documenting and promote reusability. In theory, interfaces also make code more stable by ensuring changes to them must also be made to the objects implementing them.

Below is an example of an implementation of interfaces in JavaScript using duck-typing - an approach that helps determine whether an object is an instance of constructor/object based on the methods it implements.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | // Create interfaces using a pre-defined Interface  // constructor that accepts an interface name and  // skeleton methods to expose.    // In our reminder example summary() and placeOrder()  // represent functionality the interface should  // support  var reminder = new Interface( "List", ["summary", "placeOrder"] );    var properties = {    name: "Remember to buy the milk",    date: "05/06/2016",    actions:{      summary: function (){        return "Remember to buy the milk, we are almost out!";     },      placeOrder: function (){        return "Ordering milk from your local grocery store";      }    }  };    // Now create a constructor implementing the above properties  // and methods    function Todo( config ){      // State the methods we expect to be supported    // as well as the Interface instance being checked    // against      Interface.ensureImplements( config.actions, reminder );      this.name = config.name;    this.methods = config.actions;    }    // Create a new instance of our Todo constructor    var todoItem = new Todo( properties );    // Finally test to make sure these function correctly    console.log( todoItem.methods.summary() );  console.log( todoItem.methods.placeOrder() );    // Outputs:  // Remember to buy the milk, we are almost out!  // Ordering milk from your local grocery store |

In the above, Interface.ensureImplements provides strict functionality checking and code for both this and the Interface constructor can be found [here](https://gist.github.com/1057989).

The biggest problem with interfaces is that, as there isn't built-in support for them in JavaScript, there is a danger of us attempting to emulate a feature of another language that may not be an ideal fit. Lightweight interfaces can be used without a great performance cost however and we will next look at*Abstract* Decorators using this same concept.

Abstract Decorators

To demonstrate the structure of this version of the Decorator pattern, we're going to imagine we have a superclass that models a Macbook once again and a store that allows us to "decorate" our Macbook with a number of enhancements for an additional fee.

Enhancements can include upgrades to 4GB or 8GB Ram, engraving, Parallels or a case. Now if we were to model this using an individual sub-class for each combination of enhancement options, it might look something like this:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | var Macbook = function(){          //...  };    var  MacbookWith4GBRam = function(){},       MacbookWith8GBRam = function(){},       MacbookWith4GBRamAndEngraving = function(){},       MacbookWith8GBRamAndEngraving = function(){},       MacbookWith8GBRamAndParallels = function(){},       MacbookWith4GBRamAndParallels = function(){},       MacbookWith8GBRamAndParallelsAndCase = function(){},       MacbookWith4GBRamAndParallelsAndCase = function(){},       MacbookWith8GBRamAndParallelsAndCaseAndInsurance = function(){},       MacbookWith4GBRamAndParallelsAndCaseAndInsurance = function(){}; |

and so on.

This would be an impractical solution as a new subclass would be required for every possible combination of enhancements that are available. As we would prefer to keep things simple without maintaining a large set of subclasses, let's look at how decorators may be used to solve this problem better.

Rather than requiring all of the combinations we saw earlier, we should simply have to create five new decorator classes. Methods that are called on these enhancement classes would be passed on to ourMacbook class.

In our next example, decorators transparently wrap around their components and can interestingly be interchanged as they use the same interface.

Here's the interface we're going to define for the Macbook:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | var Macbook = new Interface( "Macbook",    ["addEngraving",    "addParallels",    "add4GBRam",    "add8GBRam",    "addCase"]);    // A Macbook Pro might thus be represented as follows:  var MacbookPro = function(){      // implements Macbook  };    MacbookPro.prototype = {      addEngraving: function(){      },      addParallels: function(){      },      add4GBRam: function(){      },      add8GBRam:function(){      },      addCase: function(){      },      getPrice: function(){        // Base price        return 900.00;      }  }; |

To make it easier for us to add as many more options as needed later on, an Abstract Decorator class is defined with default methods required to implement the Macbook interface, which the rest of the options will sub-class. Abstract Decorators ensure that we can decorate a base class independently with as many decorators as needed in different combinations (remember the example earlier?) without needing to derive a class for every possible combination.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | // Macbook decorator abstract decorator class    var MacbookDecorator = function( macbook ){        Interface.ensureImplements( macbook, Macbook );      this.macbook = macbook;    };    MacbookDecorator.prototype = {      addEngraving: function(){          return this.macbook.addEngraving();      },      addParallels: function(){          return this.macbook.addParallels();      },      add4GBRam: function(){          return this.macbook.add4GBRam();      },      add8GBRam:function(){          return this.macbook.add8GBRam();      },      addCase: function(){          return this.macbook.addCase();      },      getPrice: function(){          return this.macbook.getPrice();      }  }; |

What's happening in the above sample is that the Macbook Decorator accepts an object (a Macbook) to use as our base component. It's using the Macbook interface we defined earlier and for each method is just calling the same method on the component. We can now create our option classes for what can be added, just by using the Macbook Decorator.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | // First, define a way to extend an object a  // with the properties in object b. We'll use  // this shortly!  function extend( a, b ){      for( var key in b )          if( b.hasOwnProperty(key) )              a[key] = b[key];      return a;  }    var CaseDecorator = function( macbook ){     this.macbook = macbook;  };    // Let's now extend (decorate) the CaseDecorator  // with a MacbookDecorator  extend( CaseDecorator, MacbookDecorator );    CaseDecorator.prototype.addCase = function(){      return this.macbook.addCase() + "Adding case to macbook";  };    CaseDecorator.prototype.getPrice = function(){      return this.macbook.getPrice() + 45.00;  }; |

What we're doing here is overriding the addCase() and getPrice() methods that need to be decorated and we're achieving this by first calling these methods on the original macbook and then simply appending a string or numeric value (e.g 45.00) to them accordingly.

As there's been quite a lot of information presented in this section so far, let's try to bring it all together in a single example that will hopefully highlight what we have learned.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // Instantiation of the macbook  var myMacbookPro = new MacbookPro();    // Outputs: 900.00  console.log( myMacbookPro.getPrice() );    // Decorate the macbook  var decoratedMacbookPro = new CaseDecorator( myMacbookPro );    // This will return 945.00  console.log( decoratedMacbookPro.getPrice() ); |

As decorators are able to modify objects dynamically, they're a perfect pattern for changing existing systems. Occasionally, it's just simpler to create decorators around an object versus the trouble of maintaining individual sub-classes for each object type. This makes maintaining applications that may require a large number of sub-classed objects significantly more straight-forward.

A functional version of this example can be found on [JSBin](http://jsbin.com/UMEJaXu/1/edit).

Decorators With jQuery

As with other patterns we've covered, there are also examples of the Decorator pattern that can be implemented with jQuery. jQuery.extend() allows us to extend (or merge) two or more objects (and their properties) together into a single object at run-time.

In this scenario, a target object can be decorated with new functionality without necessarily breaking or overriding existing methods in the source/superclass object (although this can be done).

In the following example, we define three objects: defaults, options and settings. The aim of the task is to decorate the defaults object with additional functionality found in optionssettings. We must:

(a) Leave "defaults" in an untouched state where we don't lose the ability to access the properties or functions found in it a later point (b) Gain the ability to use the decorated properties and functions found in "options"

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | var decoratorApp = decoratorApp || {};    // define the objects we're going to use  decoratorApp = {        defaults: {          validate: false,          limit: 5,          name: "foo",          welcome: function () {              console.log( "welcome!" );          }      },        options: {          validate: true,          name: "bar",          helloWorld: function () {              console.log( "hello world" );          }      },        settings: {},        printObj: function ( obj ) {          var arr = [],              next;          $.each( obj, function ( key, val ) {              next = key + ": ";              next += $.isPlainObject(val) ? printObj( val ) : val;              arr.push( next );          } );            return "{ " + arr.join(", ") + " }";      }    };    // merge defaults and options, without modifying defaults explicitly  decoratorApp.settings = $.extend({}, decoratorApp.defaults, decoratorApp.options);    // what we have done here is decorated defaults in a way that provides  // access to the properties and functionality it has to offer (as well as  // that of the decorator "options"). defaults itself is left unchanged    $("#log")      .append( decoratorApp.printObj(decoratorApp.settings) +            + decoratorApp.printObj(decoratorApp.options) +            + decoratorApp.printObj(decoratorApp.defaults));    // settings -- { validate: true, limit: 5, name: bar, welcome: function (){ console.log( "welcome!" ); },  // helloWorld: function (){ console.log( "hello world" ); } }  // options -- { validate: true, name: bar, helloWorld: function (){ console.log( "hello world" ); } }  // defaults -- { validate: false, limit: 5, name: foo, welcome: function (){ console.log("welcome!"); } } |

Advantages & Disadvantages

Developers enjoy using this pattern as it can be used transparently and is also fairly flexible - as we've seen, objects can be wrapped or "decorated" with new behavior and then continue to be used without needing to worry about the base object being modified. In a broader context, this pattern also avoids us needing to rely on large numbers of subclasses to get the same benefits.

There are however drawbacks that we should be aware of when implementing the pattern. If poorly managed, it can significantly complicate our application architecture as it introduces many small, but similar objects into our namespace. The concern here is that in addition to becoming hard to manage, other developers unfamiliar with the pattern may have a hard time grasping why it's being used.

Sufficient commenting or pattern research should assist with the latter, however as long as we keep a handle on how widespread we use the decorator in our applications we should be fine on both counts.

Flyweight

The Flyweight pattern is a classical structural solution for optimizing code that is repetitive, slow and inefficiently shares data. It aims to minimize the use of memory in an application by sharing as much data as possible with related objects (e.g application configuration, state and so on).

The pattern was first conceived by Paul Calder and Mark Linton in 1990 and was named after the boxing weight class that includes fighters weighing less than 112lb. The name Flyweight itself is derived from this weight classification as it refers to the small weight (memory footprint) the pattern aims to help us achieve.

In practice, Flyweight data sharing can involve taking several similar objects or data constructs used by a number of objects and placing this data into a single external object. We can pass through this object to those depending on this data, rather than storing identical data across each one.

Using Flyweights

There are two ways in which the Flyweight pattern can be applied. The first is at the data-layer, where we deal with the concept of sharing data between large quantities of similar objects stored in memory.

The second is at the DOM-layer where the Flyweight can be used as a central event-manager to avoid attaching event handlers to every child element in a parent container we wish to have some similar behavior.

As the data-layer is where the flyweight pattern is most used traditionally, we'll take a look at this first.

Flyweights and sharing data

For this application, there are a few more concepts around the classical Flyweight pattern that we need to be aware of. In the Flyweight pattern there's a concept of two states - intrinsic and extrinsic. Intrinsic information may be required by internal methods in our objects which they absolutely cannot function without. Extrinsic information can however be removed and stored externally.

Objects with the same intrinsic data can be replaced with a single shared object, created by a factory method. This allows us to reduce the overall quantity of implicit data being stored quite significantly.

The benefit of this is that we're able to keep an eye on objects that have already been instantiated so that new copies are only ever created should the intrinsic state differ from the object we already have.

We use a manager to handle the extrinsic states. How this is implemented can vary, but one approach to this to have the manager object contain a central database of the extrinsic states and the flyweight objects which they belong to.

Implementing Classical Flyweights

As the Flyweight pattern hasn't been heavily used in JavaScript in recent years, many of the implementations we might use for inspiration come from the Java and C++ worlds.

Our first look at Flyweights in code is my JavaScript implementation of the Java sample of the Flyweight pattern from Wikipedia (<http://en.wikipedia.org/wiki/Flyweight_pattern>).

We will be making use of three types of Flyweight components in this implementation, which are listed below:

* **Flyweight** corresponds to an interface through which flyweights are able to receive and act on extrinsic states
* **Concrete Flyweight** actually implements the Flyweight interface and stores intrinsic state. Concrete Flyweights need to be sharable and capable of manipulating state that is extrinsic
* **Flyweight Factory** manages flyweight objects and creates them too. It makes sure that our flyweights are shared and manages them as a group of objects which can be queried if we require individual instances. If an object has been already created in the group it returns it, otherwise it adds a new object to the pool and returns it.

These correspond to the following definitions in our implementation:

* CoffeeOrder: Flyweight
* CoffeeFlavor: Concrete Flyweight
* CoffeeOrderContext: Helper
* CoffeeFlavorFactory: Flyweight Factory
* testFlyweight: Utilization of our Flyweights

Duck punching "implements"

Duck punching allows us to extend the capabilities of a language or solution without necessarily needing to modify the runtime source. As this next solution requires the use of a Java keyword (implements) for implementing interfaces and isn't found in JavaScript natively, let's first duck punch it.

Function.prototype.implementsFor works on an object constructor and will accept a parent class (function) or object and either inherit from this using normal inheritance (for functions) or virtual inheritance (for objects).

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | // Simulate pure virtual inheritance/"implement" keyword for JS  Function.prototype.implementsFor = function( parentClassOrObject ){      if ( parentClassOrObject.constructor === Function )      {          // Normal Inheritance          this.prototype = new parentClassOrObject();          this.prototype.constructor = this;          this.prototype.parent = parentClassOrObject.prototype;      }      else      {          // Pure Virtual Inheritance          this.prototype = parentClassOrObject;          this.prototype.constructor = this;          this.prototype.parent = parentClassOrObject;      }      return this;  }; |

We can use this to patch the lack of an implements keyword by having a function inherit an interface explicitly. Below, CoffeeFlavor implements the CoffeeOrder interface and must contain its interface methods in order for us to assign the functionality powering these implementations to an object.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119 | // Flyweight object  var CoffeeOrder = {      // Interfaces    serveCoffee:function(context){},      getFlavor:function(){}    };      // ConcreteFlyweight object that creates ConcreteFlyweight  // Implements CoffeeOrder  function CoffeeFlavor( newFlavor ){        var flavor = newFlavor;        // If an interface has been defined for a feature      // implement the feature      if( typeof this.getFlavor === "function" ){        this.getFlavor = function() {            return flavor;        };      }        if( typeof this.serveCoffee === "function" ){        this.serveCoffee = function( context ) {          console.log("Serving Coffee flavor "            + flavor            + " to table number "            + context.getTable());      };      }    }      // Implement interface for CoffeeOrder  CoffeeFlavor.implementsFor( CoffeeOrder );      // Handle table numbers for a coffee order  function CoffeeOrderContext( tableNumber ) {     return{        getTable: function() {           return tableNumber;       }     };  }      function CoffeeFlavorFactory() {      var flavors = {},      length = 0;        return {          getCoffeeFlavor: function (flavorName) {                var flavor = flavors[flavorName];              if (typeof flavor === "undefined") {                  flavor = new CoffeeFlavor(flavorName);                  flavors[flavorName] = flavor;                  length++;              }              return flavor;          },            getTotalCoffeeFlavorsMade: function () {              return length;          }      };  }    // Sample usage:  // testFlyweight()    function testFlyweight(){        // The flavors ordered.    var flavors = new CoffeeFlavor(),      // The tables for the orders.      tables = new CoffeeOrderContext(),      // Number of orders made      ordersMade = 0,      // The CoffeeFlavorFactory instance      flavorFactory;      function takeOrders( flavorIn, table) {       flavors[ordersMade] = flavorFactory.getCoffeeFlavor( flavorIn );       tables[ordersMade++] = new CoffeeOrderContext( table );    }       flavorFactory = new CoffeeFlavorFactory();       takeOrders("Cappuccino", 2);     takeOrders("Cappuccino", 2);     takeOrders("Frappe", 1);     takeOrders("Frappe", 1);     takeOrders("Xpresso", 1);     takeOrders("Frappe", 897);     takeOrders("Cappuccino", 97);     takeOrders("Cappuccino", 97);     takeOrders("Frappe", 3);     takeOrders("Xpresso", 3);     takeOrders("Cappuccino", 3);     takeOrders("Xpresso", 96);     takeOrders("Frappe", 552);     takeOrders("Cappuccino", 121);     takeOrders("Xpresso", 121);       for (var i = 0; i < ordersMade; ++i) {         flavors[i].serveCoffee(tables[i]);     }     console.log(" ");     console.log("total CoffeeFlavor objects made: " + flavorFactory.getTotalCoffeeFlavorsMade());  } |

Converting code to use the Flyweight pattern

Next, let's continue our look at Flyweights by implementing a system to manage all of the books in a library. The important meta-data for each book could probably be broken down as follows:

* ID
* Title
* Author
* Genre
* Page count
* Publisher ID
* ISBN

We'll also require the following properties to keep track of which member has checked out a particular book, the date they've checked it out on as well as the expected date of return.

* checkoutDate
* checkoutMember
* dueReturnDate
* availability

Each book would thus be represented as follows, prior to any optimization using the Flyweight pattern:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | var Book = function( id, title, author, genre, pageCount,publisherID, ISBN, checkoutDate, checkoutMember, dueReturnDate,availability ){       this.id = id;     this.title = title;     this.author = author;     this.genre = genre;     this.pageCount = pageCount;     this.publisherID = publisherID;     this.ISBN = ISBN;     this.checkoutDate = checkoutDate;     this.checkoutMember = checkoutMember;     this.dueReturnDate = dueReturnDate;     this.availability = availability;    };    Book.prototype = {      getTitle: function () {       return this.title;    },      getAuthor: function () {       return this.author;    },      getISBN: function (){       return this.ISBN;    },      // For brevity, other getters are not shown    updateCheckoutStatus: function( bookID, newStatus, checkoutDate, checkoutMember, newReturnDate ){         this.id = bookID;       this.availability = newStatus;       this.checkoutDate = checkoutDate;       this.checkoutMember = checkoutMember;       this.dueReturnDate = newReturnDate;      },      extendCheckoutPeriod: function( bookID, newReturnDate ){          this.id = bookID;        this.dueReturnDate = newReturnDate;      },      isPastDue: function(bookID){         var currentDate = new Date();       return currentDate.getTime() > Date.parse( this.dueReturnDate );       }  }; |

This probably works fine initially for small collections of books, however as the library expands to include a larger inventory with multiple versions and copies of each book available, we may find the management system running slower and slower over time. Using thousands of book objects may overwhelm the available memory, but we can optimize our system using the Flyweight pattern to improve this.

We can now separate our data into intrinsic and extrinsic states as follows: data relevant to the book object (title, author etc) is intrinsic whilst the checkout data (checkoutMember, dueReturnDate etc) is considered extrinsic. Effectively this means that only one Book object is required for each combination of book properties. it's still a considerable quantity of objects, but significantly fewer than we had previously.

The following single instance of our book meta-data combinations will be shared among all of the copies of a book with a particular title.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // Flyweight optimized version  var Book = function ( title, author, genre, pageCount, publisherID, ISBN ) {        this.title = title;      this.author = author;      this.genre = genre;      this.pageCount = pageCount;      this.publisherID = publisherID;      this.ISBN = ISBN;    }; |

As we can see, the extrinsic states have been removed. Everything to do with library check-outs will be moved to a manager and as the object data is now segmented, a factory can be used for instantiation.

A Basic Factory

Let's now define a very basic factory. What we're going to have it do is perform a check to see if a book with a particular title has been previously created inside the system; if it has, we'll return it - if not, a new book will be created and stored so that it can be accessed later. This makes sure that we only create a single copy of each unique intrinsic piece of data:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | // Book Factory singleton  var BookFactory = (function () {    var existingBooks = {}, existingBook;      return {      createBook: function ( title, author, genre, pageCount, publisherID, ISBN ) {          // Find out if a particular book meta-data combination has been created before        // !! or (bang bang) forces a boolean to be returned        existingBook = existingBooks[ISBN];        if ( !!existingBook ) {          return existingBook;        } else {            // if not, let's create a new instance of the book and store it          var book = new Book( title, author, genre, pageCount, publisherID, ISBN );          existingBooks[ISBN] = book;          return book;          }      }    };    })(); |

Managing the extrinsic states

Next, we need to store the states that were removed from the Book objects somewhere - luckily a manager (which we'll be defining as a Singleton) can be used to encapsulate them. Combinations of a Book object and the library member that's checked them out will be called Book records. Our manager will be storing both and will also include checkout related logic we stripped out during our flyweight optimization of the Book class.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | // BookRecordManager singleton  var BookRecordManager = (function () {      var bookRecordDatabase = {};      return {      // add a new book into the library system      addBookRecord: function ( id, title, author, genre, pageCount, publisherID, ISBN, checkoutDate, checkoutMember, dueReturnDate, availability ) {          var book = bookFactory.createBook( title, author, genre, pageCount, publisherID, ISBN );          bookRecordDatabase[id] = {          checkoutMember: checkoutMember,          checkoutDate: checkoutDate,          dueReturnDate: dueReturnDate,          availability: availability,          book: book        };      },      updateCheckoutStatus: function ( bookID, newStatus, checkoutDate, checkoutMember, newReturnDate ) {          var record = bookRecordDatabase[bookID];        record.availability = newStatus;        record.checkoutDate = checkoutDate;        record.checkoutMember = checkoutMember;        record.dueReturnDate = newReturnDate;      },        extendCheckoutPeriod: function ( bookID, newReturnDate ) {        bookRecordDatabase[bookID].dueReturnDate = newReturnDate;      },        isPastDue: function ( bookID ) {        var currentDate = new Date();        return currentDate.getTime() > Date.parse( bookRecordDatabase[bookID].dueReturnDate );      }    };    })(); |

The result of these changes is that all of the data that's been extracted from the Book *class* is now being stored in an attribute of the BookManager singleton (BookDatabase) - something considerably more efficient than the large number of objects we were previously using. Methods related to book checkouts are also now based here as they deal with data that's extrinsic rather than intrinsic.

This process does add a little complexity to our final solution, however it's a small concern when compared to the performance issues that have been tackled. Data wise, if we have 30 copies of the same book, we are now only storing it once. Also, every function takes up memory. With the flyweight pattern these functions exist in one place (on the manager) and not on every object, thus saving on memory use. For the above-mentioned flyweight unoptimized version we store just link to the function object as we used Book constructor's prototype but if it was implemented in other way, functions would be created for every book instance.

The Flyweight pattern and the DOM

The DOM (Document Object Model) supports two approaches that allow objects to detect events - either top down (event capture) or bottom up (event bubbling).

In event capture, the event is first captured by the outer-most element and propagated to the inner-most element. In event bubbling, the event is captured and given to the inner-most element and then propagated to the outer-elements.

One of the best metaphors for describing Flyweights in this context was written by Gary Chisholm and it goes a little like this:

Try to think of the flyweight in terms of a pond. A fish opens its mouth (the event), bubbles rise to the surface (the bubbling) a fly sitting on the top flies away when the bubble reaches the surface (the action). In this example we can easily transpose the fish opening its mouth to a button being clicked, the bubbles as the bubbling effect and the fly flying away to some function being run

Bubbling was introduced to handle situations where a single event (e.g a click) may be handled by multiple event handlers defined at different levels of the DOM hierarchy. Where this happens, event bubbling executes event handlers defined for specific elements at the lowest level possible. From there on, the event bubbles up to containing elements before going to those even higher up.

Flyweights can be used to tweak the event bubbling process further, as we will see shortly.

Example 1: Centralized event handling

For our first practical example, imagine we have a number of similar elements in a document with similar behavior executed when a user-action (e.g click, mouse-over) is performed against them.

Normally what we do when constructing our own accordion component, menu or other list-based widget is bind a click event to each link element in the parent container (e.g $('ul li a').on(..). Instead of binding the click to multiple elements, we can easily attach a Flyweight to the top of our container which can listen for events coming from below. These can then be handled using logic that is as simple or complex as required.

As the types of components mentioned often have the same repeating markup for each section (e.g. each section of an accordion), there's a good chance the behavior of each element that may be clicked is going to be quite similar and relative to similar classes nearby. We'll use this information to construct a very basic accordion using the Flyweight below.

A stateManager namespace is used here to encapsulate our flyweight logic whilst jQuery is used to bind the initial click to a container div. In order to ensure that no other logic on the page is attaching similar handles to the container, an unbind event is first applied.

Now to establish exactly what child element in the container is clicked, we make use of a target check which provides a reference to the element that was clicked, regardless of its parent. We then use this information to handle the click event without actually needing to bind the event to specific children when our page loads.

**HTML**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <div id="container">     <div class="toggle" href="#">More Info (Address)         <span class="info">             This is more information         </span></div>     <div class="toggle" href="#">Even More Info (Map)         <span class="info">            <iframe src="<http://www.map-generator.net/extmap.php?name=London&amp;address=london%2C%20england&amp;width=500...gt;>"</iframe>         </span>     </div>  </div> |

**JavaScript**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | var stateManager = {      fly: function () {        var self = this;        $( "#container" )            .unbind()            .on( "click", "div.toggle", function ( e ) {              self.handleClick( e.target );            });    },      handleClick: function ( elem ) {      elem.find( "span" ).toggle( "slow" );    }  }; |

The benefit here is that we're converting many independent actions into a shared ones (potentially saving on memory).

Example 2: Using the Flyweight for performance optimization

In our second example, we'll reference some further performance gains that can be achieved using Flyweights with jQuery.

James Padolsey previously wrote an article called *76 bytes for faster jQuery* where he reminded us that each time jQuery fires off a callback, regardless of type (filter, each, event handler), we're able to access the function's context (the DOM element related to it) via the this keyword.

Unfortunately, many of us have become used to the idea of wrapping this in $() or jQuery(), which means that a new instance of jQuery is unnecessarily constructed every time, rather than simply doing this:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | $("div").on( "click", function () {    console.log( "You clicked: " + $( this ).attr( "id" ));  });    // we should avoid using the DOM element to create a  // jQuery object (with the overhead that comes with it)  // and just use the DOM element itself like this:    $( "div" ).on( "click", function () {    console.log( "You clicked:"  + this.id );  }); |

James had wanted to use jQuery's jQuery.text in the following context, however he disagreed with the notion that a new jQuery object had to be created on each iteration:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3 | $( "a" ).map( function () {    return $( this ).text();  }); |

Now with respect to redundant wrapping, where possible with jQuery's utility methods, it's better to use jQuery.methodName (e.g jQuery.text) as opposed to jQuery.fn.methodName (e.g jQuery.fn.text) where methodName represents a utility such as each() or text. This avoids the need to call a further level of abstraction or construct a new jQuery object each time our function is called as as jQuery.methodName is what the library itself uses at a lower-level to power jQuery.fn.methodName.

Because however not all of jQuery's methods have corresponding single-node functions, Padolsey devised the idea of a jQuery.single utility.

The idea here is that a single jQuery object is created and used for each call to jQuery.single (effectively meaning only one jQuery object is ever created). The implementation for this can be found below and as we're consolidating data for multiple possible objects into a more central singular structure, it is technically also a Flyweight.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | jQuery.single = (function( o ){       var collection = jQuery([1]);     return function( element ) {           // Give collection the element:         collection[0] = element;            // Return the collection:         return collection;       };  })(); |

An example of this in action with chaining is:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | $( "div" ).on( "click", function () {       var html = jQuery.single( this ).next().html();     console.log( html );    }); |

Note: Although we may believe that simply caching our jQuery code may offer just as equivalent performance gains, Padolsey claims that $.single() is still worth using and can perform better. That's not to say don't apply any caching at all, just be mindful that this approach can assist. For further details about $.single, I recommend reading Padolsey's full post.

JavaScript MV\* Patterns

In this section, we're going to review three very important architectural patterns - MVC (Model-View-Controller), MVP (Model-View-Presenter) and MVVM (Model-View-ViewModel). In the past, these patterns have been heavily used for structuring desktop and server-side applications but it's only been in recent years that come to being applied to JavaScript.

As the majority of JavaScript developers currently using these patterns opt to utilize libraries such as Backbone.js for implementing an MVC/MV\*-like structure, we will compare how modern solutions such as it differ in their interpretation of MVC compared to classical takes on these patterns.

Let us first now cover the basics.

MVC

MVC is an architectural design pattern that encourages improved application organization through a separation of concerns. It enforces the isolation of business data (Models) from user interfaces (Views), with a third component (Controllers) traditionally managing logic and user-input. The pattern was originally designed by [Trygve Reenskaug](http://en.wikipedia.org/wiki/Trygve_Reenskaug) during his time working on Smalltalk-80 (1979) where it was initially called Model-View-Controller-Editor. MVC went on to be described in depth in 1995's [“Design Patterns: Elements of Reusable Object-Oriented Software”](http://www.amazon.co.uk/Design-patterns-elements-reusable-object-oriented/dp/0201633612) (The "GoF" book), which played a role in popularizing its use.

Smalltalk-80 MVC

It's important to understand what the original MVC pattern was aiming to solve as it's mutated quite heavily since the days of its origin. Back in the 70's, graphical user-interfaces were few and far between and a concept known as [Separated Presentation](http://martinfowler.com/eaaDev/uiArchs.html) began to be used as a means to make a clear division between domain objects which modeled concepts in the real world (e.g a photo, a person) and the presentation objects which were rendered to the user's screen.

The Smalltalk-80 implementation of MVC took this concept further and had an objective of separating out the application logic from the user interface. The idea was that decoupling these parts of the application would also allow the reuse of models for other interfaces in the application. There are some interesting points worth noting about Smalltalk-80's MVC architecture:

* A Model represented domain-specific data and was ignorant of the user-interface (Views and Controllers). When a model changed, it would inform its observers.
* A View represented the current state of a Model. The Observer pattern was used for letting the View know whenever the Model was updated or modified.
* Presentation was taken care of by the View, but there wasn't just a single View and Controller - a View-Controller pair was required for each section or element being displayed on the screen.
* The Controllers role in this pair was handling user interaction (such as key-presses and actions e.g. clicks), making decisions for the View.

Developers are sometimes surprised when they learn that the Observer pattern (nowadays commonly implemented as the Publish/Subscribe variation) was included as a part of MVC's architecture many decades ago. In Smalltalk-80's MVC, the View observes the Model. As mentioned in the bullet point above, anytime the Model changes, the Views react. A simple example of this is an application backed by stock market data - in order for the application to be useful, any change to the data in our Models should result in the View being refreshed instantly.

Martin Fowler has done an excellent job of writing about the [origins](http://martinfowler.com/eaaDev/uiArchs.html) of MVC over the years and if interested in some further historical information about Smalltalk-80's MVC, I recommend reading his work.

MVC For JavaScript Developers

We've reviewed the 70's, but let us now return to the here and now. In modern times, the MVC pattern has been applied to a diverse range of programming languages including of most relevance to us: JavaScript. JavaScript now has a number of frameworks boasting support for MVC (or variations on it, which we refer to as the MV\* family), allowing developers to easily add structure to their applications without great effort.

These frameworks include the likes of Backbone, Ember.js and AngularJS. Given the importance of avoiding "spaghetti" code, a term which describes code that is very difficult to read or maintain due to its lack of structure, it's imperative that the modern JavaScript developer understand what this pattern provides. This allows us to effectively appreciate what these frameworks enable us to do differently.

We know that MVC is composed of three core components:

Models

Models manage the data for an application. They are concerned with neither the user-interface nor presentation layers but instead represent unique forms of data that an application may require. When a model changes (e.g when it is updated), it will typically notify its observers (e.g views, a concept we will cover shortly) that a change has occurred so that they may react accordingly.

To understand models further, let us imagine we have a JavaScript photo gallery application. In a photo gallery, the concept of a photo would merit its own model as it represents a unique kind of domain-specific data. Such a model may contain related attributes such as a caption, image source and additional meta-data. A specific photo would be stored in an instance of a model and a model may also be reusable. Below we can see an example of a very simplistic model implemented using Backbone.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | var Photo = Backbone.Model.extend({        // Default attributes for the photo      defaults: {        src: "placeholder.jpg",        caption: "A default image",        viewed: false      },        // Ensure that each photo created has an `src`.      initialize: function() {         this.set( { "src": this.defaults.src} );      }    }); |

The built-in capabilities of models vary across frameworks, however it is quite common for them to support validation of attributes, where attributes represent the properties of the model, such as a model identifier. When using models in real-world applications we generally also desire model persistence. Persistence allows us to edit and update models with the knowledge that its most recent state will be saved in either: memory, in a user's localStorage data-store or synchronized with a database.

In addition, a model may also have multiple views observing it. If say, our photo model contained meta-data such as its location (longitude and latitude), friends that were present in the photo (a list of identifiers) and a list of tags, a developer may decide to provide a single view to display each of these three facets.

It is not uncommon for modern MVC/MV\* frameworks to provide a means to group models together (e.g. in Backbone, these groups are referred to as "collections"). Managing models in groups allows us to write application logic based on notifications from the group should any model it contains be changed. This avoids the need to manually observe individual model instances.

A sample grouping of models into a simplified Backbone collection can be seen below.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | var PhotoGallery = Backbone.Collection.extend({        // Reference to this collection's model.      model: Photo,        // Filter down the list of all photos      // that have been viewed      viewed: function() {          return this.filter(function( photo ){             return photo.get( "viewed" );          });      },        // Filter down the list to only photos that      // have not yet been viewed      unviewed: function() {        return this.without.apply( this, this.viewed() );      }  }); |

Older texts on MVC may also contain reference to a notion of models managing application *state*.In JavaScript applications *state* has a different connotation, typically referring to the current "state" i.e view or sub-view (with specific data) on a users screen at a fixed point. State is a topic which is regularly discussed when looking at Single-page applications, where the concept of state needs to be simulated.

So to summarize, models are primarily concerned with business data.

Views

Views are a visual representation of models that present a filtered view of their current state. Whilst Smalltalk views are about painting and maintaining a bitmap, JavaScript views are about building and maintaining a DOM element.

A view typically observes a model and is notified when the model changes, allowing the view to update itself accordingly. Design pattern literature commonly refers to views as "dumb" given that their knowledge of models and controllers in an application is limited.

Users are able to interact with views and this includes the ability to read and edit (i.e get or set the attribute values in) models. As the view is the presentation layer, we generally present the ability to edit and update in a user-friendly fashion. For example, in the former photo gallery application we discussed earlier, model editing could be facilitated through an "edit' view where a user who has selected a specific photo could edit its meta-data.

The actual task of updating the model falls to controllers (which we will be covering shortly).

Let's explore views a little further using a vanilla JavaScript sample implementation. Below we can see a function that creates a single Photo view, consuming both a model instance and a controller instance.

We define a render() utility within our view which is responsible for rendering the contents of thephotoModel using a JavaScript templating engine (Underscore templating) and updating the contents of our view, referenced by photoEl.

The photoModel then adds our render() callback as one of its subscribers so that through the Observer pattern we can trigger the view to update when the model changes.

One may wonder where user-interaction comes into play here. When users click on any elements within the view, it's not the view's responsibility to know what to do next. It relies on a controller to make this decision for it. In our sample implementation, this is achieved by adding an event listener to photoElwhich will delegate handling the click behavior back to the controller, passing the model information along with it in case it's needed.

The benefit of this architecture is that each component plays its own separate role in making the application function as needed.

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | var buildPhotoView = function ( photoModel, photoController ) {      var base = document.createElement( "div" ),        photoEl = document.createElement( "div" );      base.appendChild(photoEl);      var render = function () {            // We use a templating library such as Underscore            // templating which generates the HTML for our            // photo entry            photoEl.innerHTML = \_.template( "#photoTemplate", {                src: photoModel.getSrc()            });        };      photoModel.addSubscriber( render );      photoEl.addEventListener( "click", function () {      photoController.handleEvent( "click", photoModel );    });      var show = function () {      photoEl.style.display = "";    };      var hide = function () {      photoEl.style.display = "none";    };      return {      showView: show,      hideView: hide    };    }; |

**Templating**

In the context of JavaScript frameworks that support MVC/MV\*, it is worth briefly discussing JavaScript templating and its relationship to views as we briefly touched upon it in the last section.

It has long been considered (and proven) a performance bad practice to manually create large blocks of HTML markup in-memory through string concatenation. Developers doing so have fallen prey to inperformantly iterating through their data, wrapping it in nested divs and using outdated techniques such as document.write to inject the "template" into the DOM. As this typically means keeping scripted markup inline with our standard markup, it can quickly become both difficult to read and more importantly, maintain such disasters, especially when building non-trivially sized applications.

JavaScript templating solutions (such as Handlebars.js and Mustache) are often used to define templates for views as markup (either stored externally or within script tags with a custom type - e.g text/template) containing template variables. Variables may be delimitated using a variable syntax (e.g {{name}}) and frameworks are typically smart enough to accept data in a JSON form (which model instances can be converted to) such that we only need be concerned with maintaining clean models and clean templates. Most of the grunt work to do with population is taken care of by the framework itself. This has a large number of benefits, particularly when opting to store templates externally as this can give way to templates being dynamically loaded on an as-needed basis when it comes to building larger applications.

Below we can see two examples of HTML templates. One implemented using the popular Handlebars.js framework and another using Underscore's templates.

**Handlebars.js:**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <li class="photo">    <h2>{{caption}}</h2>    <img class="source" src="{{src}}"/>    <div class="meta-data">      {{metadata}}    </div>  </li> |

**Underscore.js Microtemplates:**

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <li class="photo">    <h2><%= caption %></h2>    <img class="source" src="<%= src %>"/>    <div class="meta-data">      <%= metadata %>    </div>  </li> |

Note that templates are not themselves views. Developers coming from a Struts Model 2 architecture may feel like a template \*is\* a view, but it isn't. A view is an object which observes a model and keeps the visual representation up-to-date. A template \*might\* be a declarative way to specify part or even all of a view object so that it can be generated from the template specification.

It is also worth noting that in classical web development, navigating between independent views required the use of a page refresh. In Single-page JavaScript applications however, once data is fetched from a server via Ajax, it can simply be dynamically rendered in a new view within the same page without any such refresh being necessary.

The role of navigation thus falls to a "router", which assists in managing application state (e.g allowing users to bookmark a particular view they have navigated to). As routers are, however, neither a part of MVC nor present in every MVC-like framework, I will not be going into them in greater detail in this section.

To summarize, views are a visual representation of our application data.

Controllers

Controllers are an intermediary between models and views which are classically responsible for updating the model when the user manipulates the view.

In our photo gallery application, a controller would be responsible for handling changes the user made to the edit view for a particular photo, updating a specific photo model when a user has finished editing.

Remember that the controllers fulfill one role in MVC: the facilitation of the Strategy pattern for the view. In the Strategy pattern regard, the view delegates to the controller at the view's discretion. So, that's how the strategy pattern works. The view could delegate handling user events to the controller when the view sees fit. The view \*could\* delegate handling model change events to the controller if the view sees fit, but this is not the traditional role of the controller.

In terms of where most JavaScript MVC frameworks detract from what is conventionally considered "MVC" however, it is with controllers. The reasons for this vary, but in my honest opinion, it is that framework authors initially look at the server-side interpretation of MVC, realize that it doesn't translate 1:1 on the client-side and re-interpret the C in MVC to mean something they feel makes more sense. The issue with this however is that it is subjective, increases the complexity in both understanding the classical MVC pattern and of course the role of controllers in modern frameworks.

As an example, let's briefly review the architecture of the popular architectural framework Backbone.js. Backbone contains models and views (somewhat similar to what we reviewed earlier), however it doesn't actually have true controllers. Its views and routers act a little similar to a controller, but neither are actually controllers on their own.

In this respect, contrary to what might be mentioned in the official documentation or in blog posts, Backbone is neither a truly MVC/MVP nor MVVM framework. It's in fact better to consider it a member of the MV\* family which approaches architecture in its own way. There is of course nothing wrong with this, but it is important to distinguish between classical MVC and MV\* should we begin relying on advice from classical literature on the former to help with the latter.

Controllers in another library (Spine.js) vs Backbone.js

**Spine.js**

We now know that controllers are traditionally responsible for updating the model when the user updates the view. It's interesting to note that the most popular JavaScript MVC/MV\* framework at the time of writing (Backbone) does not have it's **own** explicit concept of controllers.

It can thus be useful for us to review the controller from another MVC framework to appreciate the difference in implementations and further demonstrate how nontraditionally frameworks approach the role of the controller. For this, let's take a look at a sample controller from Spine.js:

In this example, we're going to have a controller called PhotosController which will be in charge of individual photos in the application. It will ensure that when the view updates (e.g a user edited the photo meta-data) the corresponding model does too.

Note: We won't be delving heavily into Spine.js at all, but will just take a ten-foot view of what its controllers can do:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | // Controllers in Spine are created by inheriting from Spine.Controller    var PhotosController = Spine.Controller.sub({      init: function () {      this.item.bind( "update", this.proxy( this.render ));      this.item.bind( "destroy", this.proxy( this.remove ));    },      render: function () {      // Handle templating      this.replace( $( "#photoTemplate" ).tmpl( this.item ) );      return this;    },      remove: function () {      this.el.remove();      this.release();    }  }); |

In Spine, controllers are considered the glue for an application, adding and responding to DOM events, rendering templates and ensuring that views and models are kept in sync (which makes sense in the context of what we know to be a controller).

What we're doing in the above example is setting up listeners in the update and destroy events usingrender() and remove(). When a photo entry gets updated, we re-render the view to reflect the changes to the meta-data. Similarly, if the photo gets deleted from the gallery, we remove it from the view. In therender() function, we're using Underscore micro-templating (via \_.template()) to render a JavaScript template with the ID #photoTemplate. This simply returns a compiled HTML string used to populate the contents of photoEl.

What this provides us with is a very lightweight, simple way to manage changes between the model and the view.

**Backbone.js**

Later on in this section we're going to revisit the differences between Backbone and traditional MVC, but for now let's focus on controllers.

In Backbone, one shares the responsibility of a controller with both the Backbone.View andBackbone.Router. Some time ago Backbone did once come with its own Backbone.Controller, but as the naming for this component didn't make sense for the context in which it was being used, it was later renamed to Router.

Routers handle a little more of the controller responsibility as it's possible to bind the events there for models and have our view respond to DOM events and rendering. As Tim Branyen (another Bocoup-based Backbone contributor) has also previously pointed out, it's possible to get away with not needingBackbone.Router at all for this, so a way to think about it using the Router paradigm is probably:

[?](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var PhotoRouter = Backbone.Router.extend({    routes: { "photos/:id": "route" },      route: function( id ) {      var item = photoCollection.get( id );      var view = new PhotoView( { model: item } );        $('.content').html( view.render().el );    }  }); |

To summarize, the takeaway from this section is that controllers manage the logic and coordination between models and views in an application.

What does MVC give us?

This separation of concerns in MVC facilitates simpler modularization of an application's functionality and enables:

* Easier overall maintenance. When updates need to be made to the application it is very clear whether the changes are data-centric, meaning changes to models and possibly controllers, or merely visual, meaning changes to views.
* Decoupling models and views means that it is significantly more straight-forward to write unit tests for business logic
* Duplication of low-level model and controller code (i.e what we may have been using instead) is eliminated across the application
* Depending on the size of the application and separation of roles, this modularity allows developers responsible for core logic and developers working on the user-interfaces to work simultaneously

Smalltalk-80 MVC In JavaScript

Although the majority of modern-day JavaScript frameworks attempt to evolve the MVC paradigm to better fit the differing needs of web application development, there is one framework which attempts to adhere to the pure form of the pattern found in Smalltalk-80. Maria.js (<https://github.com/petermichaux/maria>) by Peter Michaux offers an implementation which is faithful to MVCs origins - Models are models, Views are views and Controllers are nothing but controllers. Whilst some developers might feel an MV\* framework should address more concerns, this is a useful reference to be aware of in case you would like a JavaScript implementation of the original MVC.

Delving deeper

At this point in the book, we should have a basic understanding of what the MVC pattern provides, but there's still some fascinating information about it worth noting.

The GoF do not refer to MVC as a design pattern, but rather consider it a *set of classes to build a user interface*. In their view, it's actually a variation of three classical design patterns: the Observer, Strategy and Composite patterns. Depending on how MVC has been implemented in a framework, it may also use the Factory and Template patterns. The GoF book mentions these patterns as useful extras when working with MVC.

As we have discussed, models represent application data whilst views are what the user is presented on screen. As such, MVC relies on the Observer pattern for some of its core communication (something that surprisingly isn't covered in many articles about the MVC pattern). When a model is changed it notifies its observers (Views) that something has been updated - this is perhaps the most important relationship in MVC. The observer nature of this relationship is also what facilitates multiple views being attached to the same model.

For developers interested in knowing more about the decoupled nature of MVC (once again, depending on the implementation), one of the goals of the pattern is to help define one-to-many relationships between a topic (data object) and its observers. When a topic changes, its observers are updated. Views and controllers have a slightly different relationship. Controllers facilitate views to respond to different user input and are an example of the Strategy pattern.

Summary

Having reviewed the classical MVC pattern, we should now understand how it allows us to cleanly separate concerns in an application. We should also now appreciate how JavaScript MVC frameworks may differ in their interpretation of the MVC pattern, which although quite open to variation, still shares some of the fundamental concepts the original pattern has to offer.

When reviewing a new JavaScript MVC/MV\* framework, remember - it can be useful to step back and review how it's opted to approach architecture (specifically, how it supports implementing models, views, controllers or other alternatives) as this can better help us grok how the framework expects to be used.

MVP

Model-view-presenter (MVP) is a derivative of the MVC design pattern which focuses on improving presentation logic. It originated at a company named [Taligent](http://en.wikipedia.org/wiki/Taligent) in the early 1990s while they were working on a model for a C++ CommonPoint environment. Whilst both MVC and MVP target the separation of concerns across multiple components, there are some fundamental differences between them.

For the purposes of this summary we will focus on the version of MVP most suitable for web-based architectures.

Models, Views & Presenters

The P in MVP stands for presenter. It's a component which contains the user-interface business logic for the view. Unlike MVC, invocations from the view are delegated to the presenter, which are decoupled from the view and instead talk to it through an interface. This allows for all kinds of useful things such as being able to mock views in unit tests.

The most common implementation of MVP is one which uses a Passive View (a view which is for all intents and purposes "dumb"), containing little to no logic. If MVC and MVP are different it is because the C and P do different things. In MVP, the P observes models and updates views when models change. The P effectively binds models to views, a responsibility which was previously held by controllers in MVC.

Solicited by a view, presenters perform any work to do with user requests and pass data back to them. In this respect, they retrieve data, manipulate it and determine how the data should be displayed in the view. In some implementations, the presenter also interacts with a service layer to persist data (models). Models may trigger events but it's the presenters role to subscribe to them so that it can update the view. In this passive architecture, we have no concept of direct data binding. Views expose setters which presenters can use to set data.

The benefit of this change from MVC is that it increases the testability of our application and provides a more clean separation between the view and the model. This isn't however without its costs as the lack of data binding support in the pattern can often mean having to take care of this task separately.

Although a common implementation of a [Passive View](http://martinfowler.com/eaaDev/PassiveScreen.html) is for the view to implement an interface, there are variations on it, including the use of events which can decouple the View from the Presenter a little more. As we don't have the interface construct in JavaScript, we're using more a protocol than an explicit interface here. It's technically still an API and it's probably fair for us to refer to it as an interface from that perspective.