

Essential JavaScript & jQuery Design Patterns For Beginners

Authored By [Addy Osmani](#)

Copyright 2010 © Addy Osmani.

Creative Commons [Attribution-NonCommercial-ShareAlike 3.0](#) unported license. You are free to remix, tweak, and build upon this work non-commercially, as long as you credit Addy Osmani (the copyright holder) and license your new creations under the identical terms. Any of the above conditions can be waived if you get permission from the copyright holder. For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a [link to the license](#).

Foreword

I would like to thank [Rebecca Murphy](#) for inspiring me to open-source this mini-book and release it for free download and distribution - making knowledge both open and easily available is something we should all strive for where possible. I would also like to extend my thanks to the very talented [Alex Sexton](#) who was kind enough to be the technical reviewer for this publication. I hope that it helps you learn more about design patterns and the usefulness of their application to JavaScript code.

Introduction

At the beginning of this book I will focusing on a discussion about the importance and history of design patterns in any programming language. If you're already sold or are familiar with this history, feel free to the chapter '[What is a Pattern?](#)' to continue reading.

One the most important aspects of writing maintainable code is being able to notice the recurring themes in that code and optimize them. This is an area where knowledge of design patterns can prove invaluable.

Design patterns can be traced back to the early work of a civil engineer named [Christopher Alexander](#). He would often write publications about his experience in solving design issues and how they related to buildings and towns. One day, it occurred to Alexander that when used time and time again, certain design constructs lead to a desired optimal effect.

In collaboration with Sarah Ishikawra and Murray Silverstein, Alexander produced a pattern language that would help empower anyone wishing to design and build at any scale,. This was published back in 1977 in a paper titled 'A Pattern Language'.

Some 30 years ago, software engineers began to incorporate the principles Alexander had written about into the first documentation about design patterns, which was to be a guide for novice developers looking to improve their coding skills. It's important to note that the concepts behind design patterns have actually been around in the programming industry more than likely since it's inception, albeit in a less formalized form.

One of the first and arguably most iconic formal works published on design patterns in software engineering was a book in 1995 called '*Design Patterns: Elements Of Reusable Object-Oriented Software*'. This was written by [Erich Gamma](#), [Richard Helm](#), [Ralph Johnson](#) and [John Vlissides](#) - a group that became known as the Gang of Four (or GoF for short).

The GoF's publication is considered quite instrumental to pushing the concept of design patterns further in our field as it describes a number of development techniques and pitfalls as well as providing twenty-three core Object-Oriented design patterns frequently used around the world today. We will be covering these patterns in more detail in the section 'Categories of Design Patterns'.

In this book, we will take a look at a number of popular JavaScript design patterns and explore why certain patterns may be more suitable for your projects than others. Remember that patterns can be applied not just to vanilla JavaScript, but also to abstracted libraries such as jQuery or Dojo as well. Before we begin, let's look at the exact definition of a 'pattern' in software design.

Contents

What is a Pattern?

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

The Structure Of A Design Pattern

Writing Design Patterns

Anti-Patterns

Categories Of Design Pattern

Design Patterns in jQuery

- Lazy Initialization
- Composite Pattern
- Wrapper Pattern
- Facade Pattern
- Observer Pattern
- Iterator Pattern
- Strategy Pattern
- Proxy Pattern
- Builder Pattern
- Prototype Pattern
- Flyweight Pattern

Design Patterns in JavaScript

- Creational Pattern
- Constructor Pattern
- Singleton Pattern
- Module Pattern
- Revealing Module Pattern
- Prototype Pattern
- DRY Pattern
- Facade Pattern
- Factory Pattern
- Decorator Pattern

Conclusions

References

What is a Pattern?

A pattern is a reusable solution that can be applied to commonly occurring problems in software design - in our particular case, in writing JavaScript applications. Another way of looking at patterns are as templates for how you solve problems - ones which can be used in quite a few different situations. To consider how useful a pattern may be, let us consider that if you were to write a script where you said 'for each item, sound an alert', if sounding an alert was complex in nature, it would always result in more maintainable code doing the above over saying 'do this for item 1', 'do this for item 2', 'do the same again for item 3', i.e. If the code performing the bulk of the work exists in fewer places it becomes significantly easier to maintain.

You may ask why it's 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 solutions that reflect the experience and insights the developers that helped define and improve them bring to the pattern
2. Patterns can be easily re-used: A pattern usually reflects an out of the box solution that can be adapted to suit your own needs. This feature makes them quite robust.
3. Patterns can be expressive: When you 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 can offer us.

Reusing patterns assists in preventing minor issues that can cause major problems in the application development process. What this means is that when your code relies more on proven patterns, you can afford to spend less time worrying about your code architecture and more time focusing on the quality of your overall solution. This is because patterns can encourage you to code in a more structured and organized fashion so the need to refactor it for cleanliness purposes in the future can be significantly decreased.

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) you are working with, design patterns can be applied to improve the structure of your code.

Certain patterns can actually decrease the overall file-size footprint of your 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 your codebase can be decreased.

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.

'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 weary 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 the process should create the 'thing'. This is a reason why patterns generally focus on addressing a visually identifiable structure i.e you should be able to visually depict (or draw) a picture representing the structure that putting the pattern into practice results in.

In studying design patterns, you may come across the term 'proto-pattern' quite frequently. 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 is worthy of sharing with the community, but may not have yet had the opportunity to have been vetted heavily due to it's 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 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 - you can often find that problem-solving techniques 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 it's relationship to code.

You wouldn't be blamed for thinking that a proto-pattern that doesn't meet the guidelines for a complete pattern isn't worth investigating, but 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 you 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

When studying design patterns, you may wonder what teams that create them have to put in their design pattern descriptions. Every pattern has to initially be formulated in a 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.

I find that a lot of the information available out there about the structure of a good pattern can be condensed down to something more easily digestible. With this in mind, let's now take a look at a summary of the component elements for a design pattern below.

A design pattern must have a:

Pattern Name and a description

Context Outline – the contexts in which the pattern is effective in responding to the user's 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 (eg. 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 you or your company ever consider working on your own pattern, 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 you a number of useful advantages. For starters, you can gain a deeper appreciation for the reason behind a pattern being needed but can also learn how to tell if a pattern (or proto-pattern) is up to scratch when reviewing it for your own needs.

Writing good patterns is a challenging task. Patterns not only need to provide a substantial quantity of reference material for end-users (such as the items found in the *structure* section above), but they also need to be able to almost tell a 'story' that describes the experience they are trying to convey. If you've already read the previous section on 'what' a pattern is, you may think that this in itself should help you identify patterns when you see them in the wild. This is actually quite the opposite - you can't always tell if a piece of code you're inspecting follows a pattern.

When looking at a body of code that you think may be using a pattern, you might write down some of the aspects of the code that you believe falls under a particular existing pattern, but it may not be a one at all. In many cases of pattern-analysis you'll find that you'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 you're 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 books and take in what's meaningful to you - this will help you accomplish the goals you've got of designing the pattern you want to achieve. You'll probably also want to examine the structure and semantics of existing patterns - this can be begun 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 you've exposed yourself to a wealth of information on pattern literature, you may wish to begin your pattern using an *existing* format and see if you can brainstorm new ideas for improving it or integrating your ideas in there. An example of someone that did this quite recently is JavaScript developer Christian Heilmann, who took an existing pattern called the *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).

If you would like to try your hand at writing a design pattern (even if just for the learning experience of going through the process), the tips I have for doing so would be as follows:

Bear in mind practicability: Ensure that your pattern describes proven solutions to recurring problems rather than just speculative solutions which haven't been qualified.

Ensure that you draw upon best practices: The design decisions you make should be based on principles you derive from an understanding of best practices. Your 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 behaviour 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, do you not need to be the original discoverer of the solutions being documented nor do you have to worry about your design overlapping with minor pieces of other patterns. If your design is strong enough to have broad useful applicability, it has a chance of being recognized as a proper pattern

Know the differences between patterns and design: A design pattern generally draws from proven best practice and serves as a model for a designer to create a solution. *The role of the pattern is to give designers guidance to make the best design choices so they can cater to the needs of their users.*

Your pattern needs to have 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 your 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. It was inspired by the Gang of Four's book *Design Patterns*, that developed the concept of design patterns in the software field. 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 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.

Prototyping against the `Object` object (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 you're on, whilst `document.createElement` does not. You can see [here](#) 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 you are able to recognize such anti-patterns, you will be able to refactor your code to negate them so that the overall quality of your 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 you are working in. The basic approach to object creation might otherwise lead to added complexity in a project whilst creational patterns aim to solve this problem by *controlling* the creation of such objects.

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

Structural Design Patterns

Structural patterns focus on the composition of classes and objects. Structural 'class' creation patterns use inheritance to compose interfaces whilst 'object' patterns define methods to create objects to obtain new functionality.

Patterns that fall under this category include: Decorator, Facade, Composite, Adapter and Bridge

Behavioral Design Patterns

The main focus behind this category of patterns is the communication between a class's objects. By specifically targeting this problem, these patterns are able to increase the flexibility in carrying out this communication.

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

Summary Table Of 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 recommending using this table as reference, but do remember that there are a number of additional patterns that are not mentioned on this table but will be discussed later in the book. That said, it's a great starting point for learning.

Creational	Based on the concept of creating an object.
<i>Class</i>	
<i>Factory Method</i>	This makes an instance of several derived classes based on interfaced data or events.
<i>Object</i>	
<i>Abstract Factory</i>	Creates an instance of several families of classes without detailing concrete classes.
<i>Builder</i>	Separates object construction from its representation, always creates the same type of object.
<i>Prototype</i>	A fully initialized instance used for copying or cloning.
<i>Singleton</i>	A class with only a single instance with global access points.
Structural	Based on the idea of building blocks of objects
<i>Class</i>	
<i>Adapter</i>	Match interfaces of different classes therefore classes can work together despite incompatible interfaces
<i>Object</i>	
<i>Adapter</i>	Match interfaces of different classes therefore classes can work together despite incompatible interfaces
<i>Bridge</i>	Separates an object's interface from its implementation so the two can vary independently
<i>Composite</i>	A structure of simple and composite objects which makes the total object more than just the sum of its parts.
<i>Decorator</i>	Dynamically add alternate processing to objects.
<i>Facade</i>	A single class that hides the complexity of an entire subsystem.
<i>Flyweight</i>	A fine-grained instance used for efficient sharing of information that is contained elsewhere.
<i>Proxy</i>	A place holder object representing the true object
Behavioral	Based on the way objects play and work together.
<i>Class</i>	
<i>Interpreter</i>	A way to include language elements in an application to match the grammer of the intended language.
<i>Template Method</i>	Creates the shell of an algorithm in a method, then defer the exact steps to a subclass.
<i>Object</i>	

<i>Chain of Responsibility</i>	A way of passing a request between a chain of objects to find the object that can handle the request.
<i>Command</i>	Encapsulate a command request as an object to enable, logging and/or queuing of requests, and provides error-handling for unhandled requests.
<i>Iterator</i>	Sequentially access the elements of a collection without knowing the inner workings of the collection.
<i>Mediator</i>	Defines simplified communication between classes to prevent a group of classes from referring explicitly to each other.
<i>Memento</i>	Capture an object's internal state to be able to restore it later.
<i>Observer</i>	A way of notifying change to a number of classes to ensure consistency between the classes.
<i>State</i>	Alter an object's behavior when its state changes
<i>Strategy</i>	Encapsulates an algorithm inside a class separating the selection from the implementation
<i>Visitor</i>	Adds a new operation to a class without changing the class

Design Patterns In JavaScript

Next we're going to take a look at 10 popular design patterns that I've personally found very useful to apply in JavaScript applications over the years.

Note that there is no 'ideal' pattern to use from this selection as developers often use best judgment when deciding on the pattern, which is the best 'fit' for their needs.

Each pattern varies in complexity, however I have tried to keep my explanations as simple as possible so that both beginners and intermediate developers can benefit from the material.

The patterns we will be exploring are the:

- Creational Pattern
- Constructor Pattern
- Singleton Pattern
- Module Pattern
- Revealing Module Pattern
- Prototype Pattern
- DRY Pattern
- Facade Pattern
- Factory Pattern
- Decorator Pattern

The Creational Pattern

This pattern is the basis for a number of other patterns in this article and is actually quite straightforward to understand. As you might guess, a creational pattern deals with creating objects within an application. The most common way of doing this in JavaScript is as follows:

```
var newObject = new MyClass();
```

A lot of the time, you won't have a reason to approach this in another way. You just define your class and instantiate it later on when you need it. There are however situations where this is neither an advantage nor a desired feature.

The Constructor Pattern

The phrase 'constructor' is familiar to most developers, however if you're a beginner it can be useful to review what a constructor is before we get into talking about a pattern dedicated to it. Constructors are used to create specific types of objects. This paradigm can be found in many programming languages, including JavaScript. Some of the native constructors you may be familiar with include Object and Array.

You're also able to define custom constructors that define properties and methods for your own types of objects. Let's look at a constructor for a car.


```
function Car(model, year, miles){
  this.model = model;
  this.year   = year;
  this.miles  = miles;
  this.whatCar = function(){
    console.log(this.model);
  };
}
```

```
var civic = new Car("Honda Civic", 2009, 20000);
var mondeo = new Car("Ford Mondeo", 2010, 5000);
```

Side-note: [Douglas Crockford](#) recommends capitalizing your constructor functions so that it is easier to distinguish between them and normal functions.

The Singleton Pattern

At its core, 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. The singleton pattern is thus known because traditionally, it restricts instantiation of a class to a single object.

The singleton doesn't provide a way for code that doesn't know about a previous reference to the singleton to easily retrieve it - it is not the object or class that's returned by a singleton, it's a structure.

Think of how closed variables aren't actually closures - the function scope that provides the closure is the closure.

So, where is the singleton pattern useful?. Well, it's quite useful when exactly one object is needed to coordinate patterns across the system. Here's an example of the singleton pattern being used:

```

var SingletonTester = (function(){

    //args: an object containing arguments for the singleton
    function Singleton(args) {

        //set args variable to args passed or empty object if
        none provided.
        var args = args || {};
        //set the name parameter
        this.name = 'SingletonTester';
        //set the value of pointX
        this.pointX = args.pointX || 6; //get parameter from
        arguments or set default
        //set the value of pointY
        this.pointY = args.pointY || 10;

    }

    //this is our instance holder
    var instance;

    //this is an emulation of static variables and methods
    var _static = {
        name: 'SingletonTester',
        //This is a method for getting an instance

        //It returns a singleton instance of a singleton object
        getInstance: function (args){
            if (instance === undefined) {
                instance = new Singleton(args);
            }
            return instance;
        }
    };

    return _static;
})();

var singletonTest = SingletonTester.getInstance({pointX:
5});
console.log(singletonTest.pointX); // outputs 5

```

The Module Pattern

Let's now look at the popular *module* pattern. The module pattern was defined as a way to provide both private and public encapsulation for the idea of JavaScript 'classes'. They work under the premise of a 'class' actually being defined as a function (which we'll refer to as a class for simplicity sakes).

The parameters that you decide to use for this class are actually the parameters for the constructor. Both the local variables and functions defined inside your class become private members. The return method for your class (ie. still a function) returns an object that contains your public methods and variables.

A piece of trivia is that the module pattern was originally formally defined by Douglas Crockford (famous for his book 'JavaScript: The Good Parts, and more), although it is likely that variations of this pattern were used long before this. Another piece of trivia is that if you've ever played with Yahoo's YUI library, some of it's 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.

So, you've seen why the singleton 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 your 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).

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

You 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 your application.

```

var someModule = (function(){

    //private attributes
    var privateVar = 5;

    //private methods
    var privateMethod = function(){
    return 'Private Test';
    };

    return {
        //public attributes
        publicVar    : 10,
        //public methods
        publicMethod : function(){
        return ' Followed By Public Test ';
        },

        //let's access the private members
        getData : function(){
        return privateMethod() + this.publicMethod() +
privateVar;
        }
    }
})(); //the parens here cause the anonymous function to
execute and return

someModule.getData();

```

The Revealing Module Pattern

Now you're probably a little more familiar with what the Module pattern is. Let's take a look at a slightly improved version - Christian Heilmann's Revealing Module pattern, often described as a neat extension to a rather robust pattern. The Revealing Module Pattern came about as Heilmann (now at Mozilla) was frustrated with the fact that in you had to repeat the name of the main object when you 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 you wished to make public. The result of his efforts were an updated pattern where you would simply define all of your functions and variables in the private scope and return an anonymous object at the end of the module along with pointers to both the private variables and functions you wished to reveal as public.

Once again, you're probably wondering what the benefits of this approach are. The RMP allows the syntax of your script to be fairly consistent - it also makes it very clear at the end which of your functions and variables may be accessed publicly, something that is quite useful. In addition, you are also able to reveal private functions with more specific names if you wish.

An example of how to use the revealing module pattern can be found below:

```
/*
The idea here is that you have private methods
which you want to expose as public methods.

What are are doing below is effectively defining
a self-executing function and immediately returning
the object.
*/
var myRevealingModule = function(){

    var name = 'John Smith';
    var age = 40;

    function updatePerson(){
        name = 'John Smith Updated';
    }
    function setPerson () {
        name = 'John Smith Set';
    }
    function getPerson () {
        return name;
    }
    return{
        set: setPerson,
        get: getPerson
    }
}();

// Sample usage:
myRevealingModule.get();
```

The Prototype Pattern

The prototype pattern is based on the concept of 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 below), then each object created by that same constructor will also have this same property.

Looking at the definitions for the prototype pattern in existing literature non-specific to JavaScript, you *may* find references to concepts outside the scope of the language such as classes. 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 core benefits of using the prototype pattern is that we're working with the strengths JavaScript has to offer natively rather than attempting to imitate features of other languages (something a few design pattern implementations do).

Not only is this an easy way to implement inheritance, but this also comes 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 is a recent newly native method. `Object.create` creates an object which has a specified prototype and which optionally contains specified properties (i.e `Object.create(prototype, optionalDescriptorObjects)`).

We can also see this being demonstrated in the example below:

```
// No need for capitalization as it's not a constructor
var someCar = {
  drive: function() {};
  name: 'Mazda 3'
};

// Use Object.create to generate a new car
var anotherCar = Object.create(someCar);
anotherCar.name = 'Toyota Camry';
```

The DRY Pattern

Disclaimer: DRY is essentially a way of thinking and many patterns aim to achieve a level of DRY-ness with their design. In this section we'll be covering what it means for code to be DRY but also covering the DRY design pattern based on these same concepts.

A challenge that developers writing large applications frequently have is writing similar code multiple times. Sometimes this occurs because your script or application may have multiple similar ways of performing something.

Repetitive code writing generally reduces productivity and leaves you open to having to re-write code you've already written similar times before, thus leaving you with less time to add in new functionality.

DRY (don't repeat yourself) was created to simplify this - it's based on the idea that each part of your code should ideally only have one representation of each piece of knowledge in it that applies to your system.

The key concept to take away here is that if you have code that performs a specific task, you shouldn't write that code multiple times through your applications or scripts.

When DRY is applied successfully, the modification of any element in the system doesn't change other logically-unrelated elements. Elements in your code that are logically related change uniformly and are thus kept in sync.

As other patterns covered display aspects of DRY-ness with JavaScript, let's take a look at how to write DRY code using jQuery. Note that where jQuery is used, you can easily substitute selections using vanilla JavaScript because jQuery is just JavaScript at an abstracted level.

Non-DRY

```
/*Let's store some default values in an array*/  
var defaultSettings = {};  
defaultSettings['carModel']    = 'Mercedes';  
defaultSettings['carYear']    = 2010;  
defaultSettings['carMiles']   = 5000;  
defaultSettings['carTint']    = 'Metallic Blue';
```

```

Let's do something with this data if a checkbox is
clicked*/
$('.someCheckbox').click(function(){

    if (this.checked)
    {
        $('#input_carModel').val(activeSettings.carModel);
        $('#input_carYear').val(activeSettings.carYear);
        $('#input_carMiles').val(activeSettings.carMiles);
        $('#carTint').val(activeSettings.carTint);

    } else {

        $('#input_carModel').val('');
        $('#input_carYear').val('');
        $('#input_carMiles').val('');
        $('#input_carTint').val('');

    }
});

```

DRY

```

$('.someCheckbox').click(function(){
    var checked = this.checked;
    /*
        What are we repeating?
        1. input_ precedes each field name
        2. accessing the same array for settings
        3. repeating value resets

        What can we do?
        1. programmatically generate the field names
        2. access array by key
        3. merge this call using terse coding (ie. if
checked,
        set a value, otherwise don't)
    */
    $.each(['carModel', 'carYear', 'carMiles',
'carTint'], function(i,key){
        $('#input_' + v).val(checked ?
defaultSettings[key] : '');
    });
});

```


The Facade Pattern

This pattern both simplifies the interface of a class and it also decouples the class from the code that utilizes it. Facades are often considered an essential part of a developer's pattern toolkit - they can make library utilities significantly easier to understand by creating convenience routines that simplify the use of complex systems.

An example of where the Facade pattern can be found is in the creation of uniform JavaScript APIs which often seek to provide consistent experiences across all browsers. Facades provide us with an ability to indirectly interact with subsystems in a way that may be less prone to error than accessing the subsystem directly.

Facade's advantages include ease of use and often a small size-footprint in implementing the pattern. It does however have some pitfalls - Facade is inefficient when used consecutively and each time it's called a new check must be made to determine the features available for attaching event listeners. Let's take a look at the pattern in action:

This is an unoptimized code example but here we utilize Facade to simplify an interface for attaching events.

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.

```
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;  
    }  
};
```

The Factory Pattern

Similar to other creational patterns, the Factory Pattern deals with the problem of creating objects (which we can think of as 'factory products') without the need to specify the exact class of object being created.

Specifically, the Factory Pattern suggests defining an interface for creating an object where you allow the subclasses to decide which class to instantiate. This pattern handles the problem by defining a completely separate method for the creation of objects and which sub-classes are able to override so they can specify the 'type' of factory product that will be created.

This can come in quite useful, in particular if the creation process involved is quite complex. eg. if it strongly depends on the settings in configuration files. You can often find factory methods in frameworks where the code for a library may need to create objects of particular types which may be subclassed by scripts using the frameworks.

In our example, let's take the code used in the original Constructor pattern example and see what this would look like were we to optimize it using the Factory Pattern:

```
var Car = (function() {  
    var Car = function (model, year, miles){  
        this.model = model;  
        this.year   = year;  
        this.miles  = miles;  
    };  
    return function (model, year, miles) {  
        return new Car(model, year, miles);  
    }  
})();  
  
var civic = new Car("Honda Civic", 2009, 20000);  
var mondeo = new Car("Ford Mondeo", 2010, 5000);
```

The Decorator Pattern

Decorator patterns are an alternative to creating subclasses. This pattern can be used to wrap objects within another object of the same interface and allows you to both add behaviour to methods and also pass the method call to the original object (ie the constructor of the decorator).

The decorator pattern is used when you need to keep adding new functionality to overridden methods. This can be achieved by stacking multiple decorators on top of one another.

What is the main benefit of using a decorator pattern? Well, if we examine our first definition, I mentioned that decorators are an alternative to subclassing. When a script is being run, subclassing adds behaviour that affects all the instances of the original class, whilst decorating does not.

It instead can add new behaviour for individual objects, which can be of benefit depending on the application in question.

Let's take a look at some code that implements the decorator pattern:

```
//The class we're going to decorate
function Macbook(){
    this.cost = function(){
        return 1000;
    };
}

function Memory(macbook){
    this.cost = function(){
        return macbook.cost() + 75;
    };
}

function BlurayDrive(macbook){
    this.cost = function(){
        return macbook.cost() + 300;
    };
}

function Insurance(macbook){
    this.cost = function(){
        return macbook.cost() + 250;
    };
}

// Sample usage
var myMacbook = new Insurance(new BlurayDrive(new
Memory(new Macbook())));
console.log( myMacbook.cost() );
```

Design Patterns in jQuery

Now that we've taken a look at vanilla-JavaScript implementations of popular design patterns, let's switch gears and find out what of these design patterns might look like when implemented using jQuery. jQuery (as you may know) is currently the most popular JavaScript library and provides a layer of 'sugar' on top of regular JavaScript with a syntax that can be easier to understand at a glance.

Before we dive into this section, it's important to remember that many vanilla-JavaScript design patterns can be intermixed with jQuery when used correctly because jQuery is still essentially JavaScript itself.

jQuery is an interesting topic to discuss in the realm of patterns because the library actually uses a number of design patterns itself. What impresses me is just how cleanly all of the patterns it uses have been implemented so that they exist in harmony.

Let's take a look at what some of these patterns are and how they are used.

Lazy Initialization

Lazy Initialization is a design pattern where you employ a tactic of delaying any expensive processes (eg. the creation of objects) until the first instance they are needed. An example of this is the `.ready()` function in jQuery that only executes a function once the DOM has fully loaded.

```
$(document).ready(function(){
    $('#content').fadeIn();
});
```

The Composite Pattern

The Composite Pattern describes a group of objects that can be treated in the same way a single instance of an object can. Implementing this pattern allows you to treat both individual objects and compositions in a uniform manner. In jQuery, when we're accessing or performing actions on a single DOM element or a group of DOM elements, we can treat both in a uniform manner. This is demonstrated by the code sample below:

```
$('#someDiv').addClass('active'); // a single element
$('div').addClass('active');      // a collection of
elements
```

The Wrapper Pattern

The Wrapper Pattern is a pattern which translates an *interface* for a class into a compatible interface. Wrappers basically allow classes to function together which normally couldn't due to their incompatible interfaces. The wrapper translates calls to it's interface into calls to the original interface and the code required to achieve this is usually quite minimal.

```
$('.container').css({  
    opacity: .5 //apply opacity in modern browsers (eg.  
    Chrome, FireFox) but use filter for IE  
});
```

The Facade Pattern

The Facade Pattern is quite commonly used with OOP (Object-oriented programming) where a facade is an object which provides a simpler interface to a larger piece of code (eg. a class library). Facades can be frequently found across the jQuery library and make methods both easier to use and understand, but also more readable. The following are facades for jQuery's \$.ajax():

```
$.get();  
$.post();  
$.getJSON();  
$.getScript();
```

The Observer pattern

The Observer pattern is where a subject (the object), keeps a list of it's dependants, which are known as observers, and notifies them automatically of any changes in state. This is commonly done by calling one of their methods. The Observer pattern can be considered a subset of PubSub (publish/subscribe pattern).

```
//Here jQuery makes use of it's event system on top of DOM  
events  
$('.button').click(function(){})  
$('.button').trigger('click', function(){})
```

The Iterator Pattern

The Iterator Pattern is a design pattern where iterators (objects that allow us to traverse through all the elements of a collection) access the elements of an aggregate object sequentially without needing to expose its underlying form.

Iterators encapsulate the internal structure of how that particular iteration occurs - in the case of jQuery's `.each()` iterator, you are actually able to use the underlying code behind `.each()` to iterate through a collection, without needing to see or understand the code working behind the scenes that's providing this capability.

```
$.each(function(){});  
$('.items').each(function(){});
```

An interesting side-note is that jQuery's 'each' method is backwards from the ECMAScript 5 way of doing this but may change at some point in the future.

The Strategy Pattern

The Strategy Pattern is a pattern where a script may select a particular algorithm at runtime. The purpose of this pattern is that it's able to provide a way to clearly define families of algorithms, encapsulate each as an object and make them easily interchangeable.

You could say that the biggest benefit this pattern offers is that it allows algorithms to vary independent of the clients that utilize them. An example of this is where jQuery's `toggle()` allows you to bind two or more handlers to the matched elements, to be executed on alternate clicks.

The strategy pattern allows for alternative algorithms to be used independent of the client internal to the function.

```
$('#container').toggle(function(){}, function(){});
```

The Proxy Pattern

The Proxy Pattern - a proxy is basically a class that functions as an interface to something else. The proxy can be an interface to almost anything: a file, a resource, an object in memory, something else that is difficult to duplicate etc. jQuery's `.proxy()` function takes as input a function and returns a new one that will always have a particular context.

This is parallel to the idea of providing an interface as per the proxy pattern.

```
$.proxy(function(){}, obj);
```

The Builder Pattern

The Builder Pattern's main concept is that it abstracts the steps involved in creating objects so that different implementations of these steps have the ability to construct different representations of objects.

Below is an example of how jQuery utilizes this pattern to allow an element which you may wish to append to the document body to be constructed using a string definition.

```
$('< div class= "foo"> bar < /div>');
```

The Prototype Pattern

The Prototype Pattern is used when the objects you wish to create are determined by a prototypal instance that is cloned to produce the new objects. Essentially this pattern is used to avoid creating a new object in a standard manner when this process may be expensive or overly complex. In the following code sample which extends the jQuery.fn object for a minimal plugin, underlying prototypal code makes this possible:

```
$.fn.plugin = function(){}  
$('#container').plugin();
```

The Flyweight Pattern

The Flyweight Pattern is a design pattern where an object attempts to minimize the amount of memory used by sharing as much information as possible with other objects that are similar in nature. It's a way to utilize objects in large numbers when a simple repeated representation may use an amount of memory deemed unacceptable. There are often aspects of an object state that can be shared and it's commonplace that these be stored in external data-structures that are passed to the flyweight objects temporarily when needed.

// The userConfig is shared here:

```
$.fn.plugin = function(userConfig){  
    userConfig = $.extend({  
        content: 'Hello user!'  
    }, userConfig);  
    return this.html(userConfig.content);  
});
```

A side-note here is that prototypal inheritance in JavaScript uses differential inheritance to only define objects once in a prototype chain until they are overridden. This makes it easier to save memory.

Conclusions

That's it for this introduction to the world of design patterns in JavaScript & jQuery– I hope you've found it useful. The contents of this book are in no way an extensive look at the field of patterns, but should give you enough information to get started using the patterns covered in your day-to-day projects.

Design patterns make it easier to reuse successful designs and architectures. It's important for every developer to be aware of design patterns but it's also essential to know how and when to use them. Implementing the right patterns intelligently can be worth the effort but the opposite is also true. A badly implementing pattern can yield little benefit to a project.

Also bare in mind that it's not the number of patterns you implement that's important but how you choose to implement them. For example, don't choose a pattern just for the sake of using 'one' but rather try understanding the pros and cons of what particular patterns have to offer and make a judgement based on it's fitness for your application.

If I've encouraged your interest in this area further and you would like to learn more about design patterns, there are a number of excellent titles on this area available for generic software development but also those that cover specific languages.

For JavaScript developers, I recommend checking out two books:

1. '[Pro JavaScript Design Patterns](#)' by Ross Harmes and Dustin Diaz.
2. '[JavaScript Patterns](#)' by Stoyan Stefanov

If you've managed to absorb most of the information in my mini-book, I think you'll find reading these the next logical step in your learning process (beyond trying out some pattern examples for yourself of course) :)

Thanks for reading *Essential JavaScript & jQuery Design Patterns*. For more free learning material on JavaScript, jQuery and User-Interface Design, check out my official site over at <http://addyosmani.com> for my latest educational resources.

References

1. Design Principles and Design Patterns - Robert C Martin http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
2. Ralph Johnson - Special Issue of ACM On Patterns and Pattern Languages - <http://www.cs.wustl.edu/~schmidt/CACM-editorial.html>
3. Hillside Engineering Design Patterns Library - <http://hillside.net/patterns/>
4. Pro JavaScript Design Patterns - Ross Harmes and Dustin Diaz <http://jsdesignpatterns.com/>
5. Design Pattern Definitions - http://en.wikipedia.org/wiki/Design_Patterns
6. Patterns and Software Terminology <http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html>
7. Reap the benefits of Design Patterns - Jeff Juday http://articles.techrepublic.com.com/5100-10878_11-5173591.html
8. JavaScript Design Patterns - Subramanyan Guhan <http://www.slideshare.net/rmsguham/javascript-design-patterns>
9. What Are Design Patterns and Do I Need Them? - James Moaoriello <http://www.developer.com/design/article.php/1474561>
10. Software Design Patterns - Alex Barnett <http://alexbarnett.net/blog/archive/2007/07/20/software-design-patterns.aspx>
11. Evaluating Software Design Patterns - Gunni Rode <http://www.rode.dk/thesis/>
12. SourceMaking Design Patterns http://sourcemaking.com/design_patterns
13. The Singleton - Prototyp.ical <http://prototyp.ical.ly/index.php/2007/03/01/javascript-design-patterns-1-the-singleton/>
14. JavaScript Patterns - Stoyan Stevanov - <http://www.slideshare.net/stoyan/javascript-patterns>
15. Stack Overflow - Design Pattern Implementations in JavaScript (discussion) <http://stackoverflow.com/questions/24642/what-are-some-examples-of-design-pattern-implementations-using-javascript>
16. The Elements of a Design Pattern - Jared Spool http://www.uie.com/articles/elements_of_a_design_pattern/
17. Stack Overflow - Examples of Practical JS Design Patterns (discussion) <http://stackoverflow.com/questions/3722820/examples-of-practical-javascript-object-oriented-design-patterns>
18. Design Patterns in JavaScript Part 1 - Nicholas Zakkas <http://www.webreference.com/programming/javascript/ncz/column5/>
19. Stack Overflow - Design Patterns in jQuery <http://stackoverflow.com/questions/3631039/design-patterns-used-in-the-jquery-library>
20. Classifying Design Patterns By AntiClue - Elyse Neilson <http://www.anticlue.net/archives/000198.htm>
21. Design Patterns, Pattern Languages and Frameworks - Douglas Schmidt <http://www.cs.wustl.edu/~schmidt/patterns.html>

22. Show Love To The Module Pattern - Christian Heilmann <http://www.wait-till-i.com/2007/07/24/show-love-to-the-module-pattern/>
23. JavaScript Design Patterns - Mike G. <http://www.lovemikeg.com/2010/09/29/javascript-design-patterns/>
24. Software Designs Made Simple - Anoop Mashudanan <http://www.scribd.com/doc/16352479/Software-Design-Patterns-Made-Simple>
25. JavaScript Design Patterns - Klaus Komenda <http://www.klauskomenda.com/code/javascript-programming-patterns/>
26. Design Patterns Explained - <http://c2.com/cgi/wiki?DesignPatterns>
27. Working with GoF's Design Patterns In JavaScript http://aspalliance.com/1782_Working_with_GoFs_Design_Patterns_in_JavaScript_Programming.all
28. Design Patterns by Gamma, Helm supplement <http://exciton.cs.rice.edu/javaresources/DesignPatterns/>