# 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

**Note:** ES2015 introduced native support for classes to JavaScript, however, they are primarily syntactical sugar over JavaScript's existing prototype-based inheritance model. We will not be covering ES2015 classes in this book, but [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes) has an excellent introduction to them.

Keep in mind that there will be patterns in this table that reference the concept of "classes". In ES5, 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/#)

|  |  |
| --- | --- |
|  | // 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:

|  |  |
| --- | --- |
|  | 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. | | |

## 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:

|  |  |
| --- | --- |
|  | // 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:

|  |  |
| --- | --- |
|  | // 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/#)

|  |  |
| --- | --- |
|  | 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/#)

|  |  |
| --- | --- |
|  | 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.