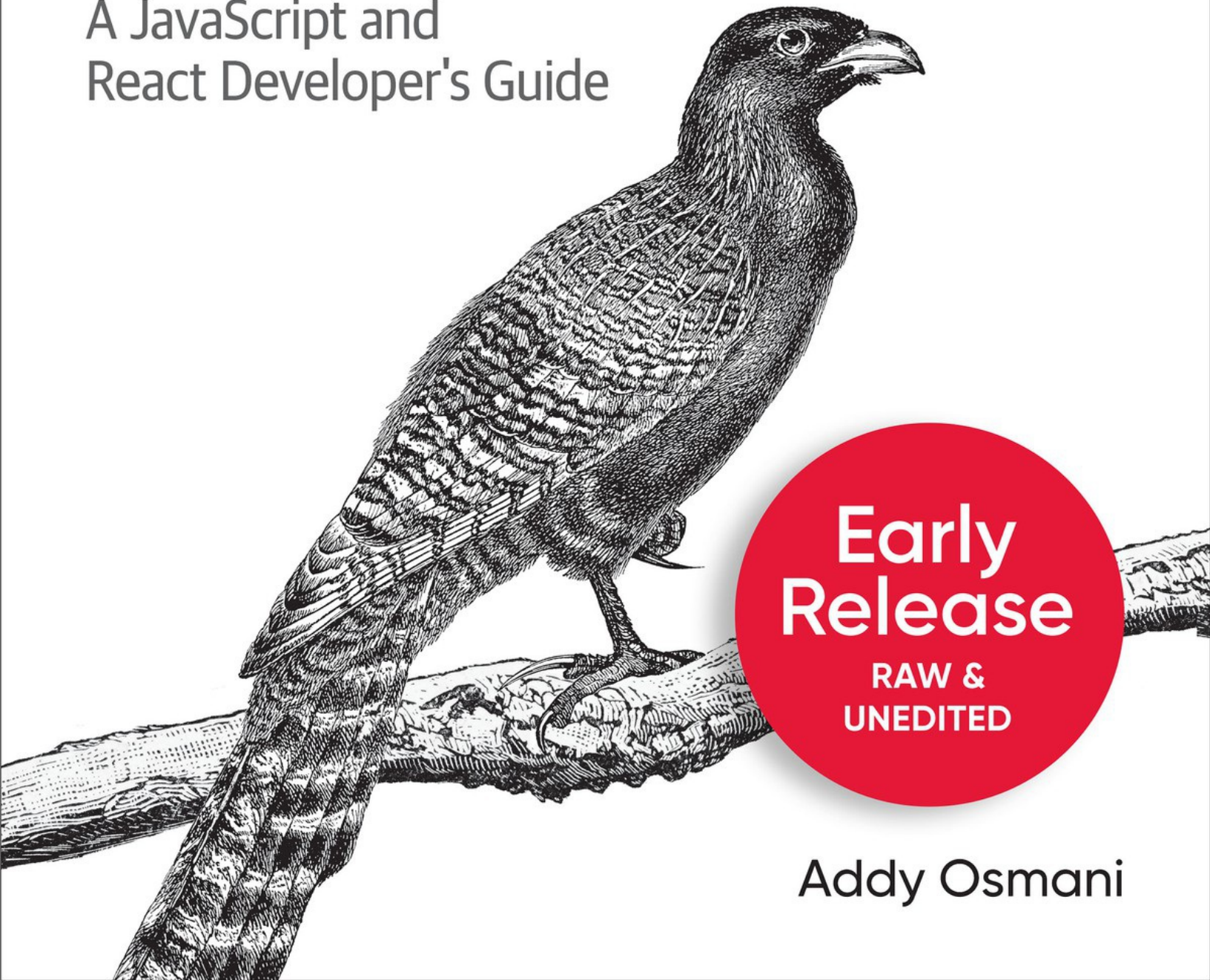
O'REILLY®

Second Edition

Learning JavaScript Design Patterns

A JavaScript and React Developer's Guide



Early Release

RAW & UNEDITED

Addy Osmani

Learning JavaScript Design Patterns

SECOND EDITION

A JavaScript and React Developer's Guide

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Addy Osmani

Learning JavaScript Design Patterns

by Addy Osmani

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Amanda Quinn

Development Editor: Michele Cronin

Production Editor: Gregory Hyman

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

June 2023: Second Edition

Revision History for the Early Release

- 2022-12-05: First Release
- 2023-01-19: Second Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098139872 for release details.

978-1-098-13987-2

Chapter 1. Introduction to Design Patterns

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the first chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Good code is like a love letter to the next developer who will maintain it!

Design patterns provide a common vocabulary to structure code, making it easier to understand. They help enhance the quality of this connection to other developers. Knowledge of design patterns helps us identify recurring themes in requirements and map them to definitive solutions. We can rely on the experience of others who have encountered a similar problem and devised an optimized method to address it. This knowledge is invaluable as it paves the way for writing or refactoring code to make it maintainable.

Whether on the server or client, JavaScript is a cornerstone of modern web application development. The previous edition of this book focused on several popular design patterns in the JavaScript context. Over the years, JavaScript has significantly evolved as a language in terms of features and syntax. It now supports modules, classes, arrow functions, and template literals that were not there earlier. We also have advanced JavaScript libraries and frameworks that have made life easy for many web developers. How

relevant, then, are Design Patterns in the modern JavaScript context?

It's important to note that traditionally design patterns are neither prescriptive nor language specific. You can apply them when you think they fit, but you don't have to. Like data structures or algorithms, you can still apply classic design patterns using modern programming languages, including JavaScript. You may not need some of these design patterns in modern frameworks or libraries where they are already abstracted. Conversely, the use of specific patterns may even be encouraged by some frameworks.

In this edition, we are taking a pragmatic approach to patterns. We will explore why specific patterns may be the right fit for implementing certain features and if a pattern it is still recommended in the modern JavaScript context.

As applications got more interactive, requiring a large amount of JavaScript, the language came under constant criticism for its negative impact on performance. Developers are continuously looking for new patterns that can optimize JavaScript performance. This edition highlights such improvements wherever relevant. We will also discuss framework-specific patterns such as React Hooks and Higher Order Components that have become increasingly popular in the age of React.js.

Going back a step, let us start by exploring the history and importance of design patterns. If you are already familiar with this history, feel free to skip to "What Is a Pattern?" to continue reading.

History of Design Patterns

Design patterns can be traced back to the early work of an architect named Christopher Alexander. He often wrote about his experiences in solving design issues and how they related to buildings and towns. One day, it occurred to Alexander that certain design constructs lead to a desired optimal effect when used repeatedly.

Alexander produced a pattern language in collaboration with Sara Ishikawa and Murray Silverstein. This language would help empower anyone wishing

to design and build at any scale. They published it in 1977 in a paper titled "A Pattern Language," later released as a complete hardcover book.

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

One of the first and arguably the 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* - written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Most engineers today recognize this group as the Gang of Four (or GoF for short).

The GoF's publication is particularly instrumental in pushing the concept of design patterns further in our field. It describes several development techniques and pitfalls and provides 23 core object-oriented design patterns frequently used worldwide today. We will cover these patterns in more detail in Chapter 6, and they also form the basis for our discussion of Chapter 7.

What Is a Pattern?

A pattern is a reusable solution template that you can apply to recurring problems and themes in software design. When building a JavaScript web application, you can use the template to structure your JavaScript code in different situations where you think it will help.

Learning and using Design patterns is mainly advantageous for developers because:

Patterns are proven solutions.

> They are the result of the combined experience and insights of developers who helped define them. They are time-tested approaches known to work when solving specific issues in software development.

Patterns can be easily reused.

A pattern usually delivers an out-of-the-box solution you can adopt and adapt to suit your needs. This feature makes them quite robust.

Patterns can be expressive.

Patterns can help express elegant solutions to extensive problems using a set structure and a shared *vocabulary*.

Additional advantages that patterns offer are:

- **Patterns assist in preventing minor issues that can cause significant problems in the application development process.** When you use established patterns to build code, you can relax about getting the structure wrong and focus on the quality of the overall solution. The pattern encourages you to write more structured and organized code naturally, avoiding the need to refactor it for cleanliness in the future.
- **Patterns provide generalized solutions, documented in a fashion that doesn't require them to be tied to a specific problem.** This generalized approach means you can apply design patterns to improve code structure regardless of the application (and, in many cases, the programming language).
- **Some patterns can decrease the overall code file-size footprint by avoiding repetition.** Design patterns encourage developers to look more closely at their solutions for areas where they can achieve instant reductions in duplication. For example, you can reduce the number of functions performing similar processes in favor of a single generalized function to decrease the size of your codebase. This is also known as making code more *dry*.
- **Patterns add to a developer's vocabulary, which makes communication faster.** Developers can reference the pattern when communicating with their team, discussing it in the design patterns community, or indirectly when another developer later maintains the

code.

- **Popular design patterns can be improvised further by harnessing the collective experiences of developers using those patterns and contributing back to the community.** In some cases, this leads to the creation of entirely new design patterns, while in others, it can lead to improved guidelines on the usage of specific patterns. This can ensure that pattern-based solutions continue to become more robust than ad hoc ones.

> **NOTE**
>
> Patterns are *not* exact solutions. The role of a pattern is merely to provide us with a solution scheme. Patterns don't solve all design problems nor replace good software designers. You still need sound designers to choose the correct patterns that can enhance the overall design.

An everyday use case for design patterns

If you have used React.js, you have probably come across the Provider pattern. If not, you may have experienced the following situation.

The component tree in web applications often needs access to shared data like user information or user access permissions. The traditional way to do this in JavaScript is to set these properties for the root level component and then pass them down from parent to child components. As the component hierarchy deepens and becomes more nested, you drill down it with your data resulting in the practice of prop-drilling. This leads to unmaintainable code where the property setting and passing will get repeated in every child component, which relies on that data.

React and a few other frameworks address this problem using the Provider Pattern. With the Provider pattern, the React Context API can broadcast the state/data to multiple components via a context provider. Child components needing the shared data can tap into this provider as a context consumer or

use the useContext Hook.

This is an excellent example of a design pattern used to optimize the solution to a very common problem. We will cover this and many such patterns in a lot of detail in this book.

Summary

With that introduction to the importance of design patterns and their relevance to modern JavaScript, we can now deep dive into learning JavaScript Design Patterns. The first few chapters in this book talk about structuring and classifying patterns and identifying anti-patterns before we go into the specifics of design patterns for JavaScript. But first, let us see what it takes for a proposed “proto-pattern” to be recognized as a pattern in the next chapter.

Chapter 2. “Pattern”-ity Testing, Proto-Patterns, and the Rule of Three

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the second chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

From the moment a new pattern is proposed to its potential widespread adoption, a pattern may have to go through multiple rounds of deep inspection by the design community and software developers. This chapter talks about this journey of a newly introduced “Proto-Pattern” through a “pattern”-ity test till it is eventually recognized as a pattern if it meets the *Rule of Three*.

This and the next chapter explore the approach to structuring, writing, presenting, and reviewing nascent design patterns. If you’d prefer to learn established design patterns first, you can skip these two chapters for the time being.

What are Proto-Patterns?

Remember that not every algorithm, best practice, or solution represents what might be considered a complete pattern. There may be a few key ingredients missing, and the pattern community is generally wary of something claiming to be one without an extensive and critical evaluation. Even if something is presented to us which *appears* to meet the criteria for a pattern, we should not consider it as one until it has undergone suitable periods of scrutiny and testing by others.

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

The "Pattern" Tests

You may often come across the term "proto-pattern" when studying design patterns. What is this? Well, a pattern that has not yet conclusively passed the "pattern"-ity tests is usually referred to as a proto-pattern. Proto-patterns may result from the work of someone who has established a particular solution worthy of sharing with the community. However, due to its relatively young age, the community has not had the opportunity to vet the proposed solution suitably.

Alternatively, the individual(s) sharing the pattern may not have the time or interest in 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 comprehensively documenting a qualified pattern can be pretty 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.

- **Does not have an obvious solution.** We can find that problem-solving techniques often attempt to derive from well-known first principles. The best design patterns usually provide solutions to issues indirectly - this is considered a necessary approach for the most challenging problems related to design.
- **Describes a proven concept.** 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 will attempt to use it.
- **Describes a relationship.** In some cases, it may appear that a pattern describes a type of module. Despite what the implementation looks like, 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 that fails to meet guidelines isn't worth learning from; however, this is far from the truth. Many proto-patterns are actually quite good. I am 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 your best judgment with the above list in mind, and you'll be fine in your selection process.

Rule of Three

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

Fitness of purpose

How is the pattern considered successful?

Usefulness

Why is the pattern considered successful?

Applicability

Is the design worthy of being a pattern because it has broader applicability? If so, this needs to be explained. When reviewing or defining a pattern, it is vital to keep the above in mind.

Summary

This chapter has shown how every proposed proto-pattern may not always be accepted as a pattern. The next chapter shares the essential elements and best practices for structuring and documenting patterns so the community can easily understand and consume them.

Chapter 3. Structuring and Writing Patterns

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the third chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

The success of a new idea depends on its utility and also on how you present it to those it is trying to help. For developers to understand and adopt a design pattern, it should be presented with relevant information about the context, circumstances, prerequisites, and significant examples. This chapter applies to those trying to understand a specific pattern and those trying to introduce a new one as it provides essential information on how patterns are structured and written.

The Structure of a Design Pattern

Pattern authors would be unable to successfully create and publish a pattern if they cannot define its purpose. Similarly, developers will find understanding or implementing a pattern challenging if they do not have a background or context.

Pattern authors must outline a new pattern's design, implementation, and

purpose. Authors initially present a new pattern in the form of a *rule* that establishes a relationship between:

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

With this in mind, let‘s now summarize the component elements for a design pattern. A design pattern should have the following, with the first five elements being the most important:

Pattern name

A unique name representative of the purpose of the pattern

Description

A brief description of what the pattern helps achieve.

Context outline

The contexts in which the pattern effectively responds to its users’ needs.

Problem statement

A statement of the problem addressed so that we understand the pattern’s intent.

Solution

A description of how the user’s problem is 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 developers would implement the pattern.

Illustrations

Visual representations of classes in the pattern (e.g., a diagram).

Examples

Implementations of the pattern in a minimal form.

Corequisites

What other patterns may be needed to support the 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.

Well Written Patterns

Understanding the structure and purpose of a design pattern can help us gain a deeper appreciation for the reasoning behind why a pattern is needed. It also helps us evaluate the pattern for our own needs.

A good pattern should ideally provide a substantial quantity of reference material for end users. Patterns should also provide evidence of why they are necessary.

Just having an overview of a pattern is not enough to help us identify them in the code we may encounter day-to-day. It's not always clear if a piece of

code we're looking at follows a set pattern or accidentally resembles one.

If you suspect that the code you see uses a pattern, consider writing down some aspects of the code that fall under a particular existing pattern or set of patterns. It may be that the code follows sound principles and design practices that coincidentally overlap with the rules for a specific pattern.

> **TIP**
>
> Solutions in which neither interactions nor defined rules appear are not patterns.

Although patterns may have a high initial cost in the planning and write-up phases, the value returned from that investment can be worth it. Patterns are valuable because they help to get all the developers in an organization or team on the same page when creating or maintaining solutions. If you are considering working on a pattern of your own, research beforehand, as you may find it more beneficial to use or extend existing, proven patterns rather than starting afresh.

Writing a pattern

If you are trying to develop a design pattern yourself, I recommend learning from others who have already been through the process and done it well. Spend time absorbing the information from several different design pattern descriptions and take in what's meaningful to you. Explore structure and semantics—you can do this by examining the interactions and context of the patterns you are interested in to identify the principles that assist in organizing those patterns together in valuable configurations.

You can utilize the *existing* format to write your own pattern or see if there are ways to improve it by integrating your ideas. An example of a developer that did this in recent years is Christian Heilmann, who took the existing *Module* pattern and made some fundamentally valuable changes to it to create the *Revealing Module* pattern.

Adhering to the following checklist would help if you are interested in creating a new design pattern or adapting an existing one:

How practical is the pattern?

> Ensure that the pattern describes proven solutions to recurring problems rather than just speculative solutions that haven't been qualified.

Keep best practices in mind.

> Our design decisions should be based on principles we derive from understanding best practices.

Our design patterns should be transparent to the user.

> Design patterns should be entirely transparent to the end-user experience. They primarily serve the developers using them and should not force changes to the expected user experience.

Remember that originality is not key in pattern design.

> When writing a pattern, you do not need to be the original discoverer of the documented solutions, nor do you have to worry about your design overlapping with minor pieces of other patterns. If the approach is strong enough to have broadly applicable, 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 effective set of examples demonstrating the successful application of the pattern. To show broad usage, examples that exhibit sound 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 you comprehensively cover all possible application areas when writing a pattern.

Whether you write a pattern or not, I hope this brief introduction to writing

patterns has given you some insights that will assist your learning process and help you rationalize the patterns covered in the following sections of this book.

Summary

This chapter painted a picture of the ideal “good” pattern. It is equally important to understand that “bad” patterns exist, too, so we can identify and avoid them. And that is why we have the next chapter about “Anti-Patterns”.

Chapter 4. Anti-Patterns

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fourth chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

As engineers, we may run into situations when we are on a deadline to deliver a solution or where code gets included as a series of patches without a code review. The code in such cases may not always be well thought out and may propagate what we call *Anti-Patterns*. This chapter describes what Anti-Patterns are and why it is essential to understand and identify them. We also look at some typical anti-patterns in JavaScript.

What are Anti-Patterns

If a pattern represents a best practice, an anti-pattern represents the lesson learned when a proposed pattern goes wrong. Inspired by the GoF's book *Design Patterns*, Andrew Koenig first coined the term Anti-Pattern in 1995 in his article in the Journal of Object-Oriented Programming - Volume 8. He described Anti-Patterns as:

> *An antipattern is just like a pattern, except that instead of a solution, it gives something that looks superficially like a solution but isn't one.*

He presented two notions of anti-patterns. Anti-patterns:

- Describe a *bad* solution to a particular problem that resulted in an unfavorable situation occurring
- Describe *how* to get out of the said situation and go 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.*

Understanding anti-patterns is as essential as being aware of design patterns. Let us qualify the reason behind this. When creating an application, a project's lifecycle begins with construction. At this stage, you are likely to choose from the available *good* design patterns as you see fit. However, after the initial release, it needs to be maintained.

Maintenance of an application already in production can be particularly challenging. Developers who haven't worked on the application before may accidentally introduce a *bad* design into the project. If said *bad* practices have already been identified as anti-patterns, developers will recognize these in advance and avoid the known common mistakes. This is similar to how knowledge of design patterns allows us to recognize areas where we could apply *known* and *helpful* standard techniques.

The quality of the solution as it evolves will either be *good* or *bad*, depending on the skill level and time the team has 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.

To summarize, an anti-pattern is a bad design that is worthy of documenting.

Anti-Patterns in JavaScript

Developers sometimes knowingly opt for shortcuts and temporary solutions

to expedite code deliveries. These tend to become permanent and accumulate as technical debt that is essentially made up of anti-patterns. JavaScript is a weakly typed or untyped language that makes taking certain shortcuts easier. Following are some examples of anti-patterns that you might come across in JavaScript:

- Polluting the global namespace by defining numerous 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. If it's executed after the page has been loaded, it can overwrite the page we're on, which makes `document.createElement` a far better choice. Visit this site 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 learn to recognize such anti-patterns, we can refactor our code to negate them so that the overall quality of our solutions improves instantly.

Summary

This chapter covered patterns that could cause problems known as Anti-Patterns and examples of JavaScript Anti-Patterns. Before we cover JavaScript design patterns in detail, we must touch upon some critical

modern JavaScript concepts that will prove relevant to our discussion on patterns. This is the subject for the next chapter that introduces modern JavaScript features and syntax.

Chapter 5. Modern JavaScript Syntax and Features

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fifth chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

JavaScript has been around for many decades now and has undergone multiple revisions. This book explores design patterns in the modern JavaScript context and uses modern ES2015+ syntax for all the examples discussed. This chapter discusses ES2015+ JavaScript features and syntax essential to further our discussion of design patterns in the current JavaScript context.

NOTE

Some fundamental changes were introduced to JavaScript syntax with ES2015 that are especially relevant to our discussion on patterns. These are covered well in the BabelJS ES2015 guide.

The Importance Of Decoupling Applications

Modular JavaScript allows you to logically split your application into smaller pieces called modules. A module can be imported by other modules that, in turn, can be imported by more modules. Thus, the application can be composed of many nested modules.

In the world of scalable JavaScript, when we say an application is **modular**, we often mean it's composed of a set of highly decoupled, distinct pieces of functionality stored in modules. Loose coupling facilitates easier maintainability of apps by removing *dependencies* where possible. If implemented efficiently, it allows you to see how changes to one part of a system may affect another.

Unlike some more traditional programming languages, the older iterations of JavaScript till ES5 (Standard ECMA-262 5.1 Edition) did not provide developers with the means to organize and import code modules cleanly. It was one of the concerns with the specifications that had not required great thought until more recent years when the need for more organized JavaScript applications became apparent. AMD (Asynchronous Module Definition) and CommonJS Modules were the most popular patterns to decouple applications in the initial versions of JavaScript.

Native solutions to these problems arrived with ES6 or ES2015. TC39, the standards body charged with defining the syntax and semantics of ECMAScript and its future iterations, had been keeping a close eye on the evolution of JavaScript usage for large-scale development and was acutely aware of the need for better language features for writing more modular JS.

The syntax to create modules in JavaScript was developed and standardized with the release of ECMAScript Modules in ES2015. Today, all major browsers support JavaScript modules. They have become the de-facto method of implementing modern-day modular programming in JavaScript. In this section, we'll explore code samples using the syntax for modules in ES2015+.

Modules With Imports And Exports

Modules allow us to separate our application code into independent units, each containing code for one aspect of the functionality. Modules also encourage code reusability and expose features that can be integrated into different applications.

A language should have features that allow you to `import` module dependencies and `export` the module interface (the public API/variables we allow other modules to consume) to support modular programming. The support for JavaScript Modules (also referred to as ES Modules) was introduced to JavaScript in ES2015, allowing you to specify module dependencies using an `import` keyword. Similarly, you can use the `export` keyword to export just about anything from within the module.

- **import** declarations bind a module's exports as local variables and may be renamed to avoid name collisions/conflicts.
- **export** declarations declare that a local binding of a module is externally visible such that other modules may read the exports but can't modify them. Interestingly, modules may export child modules but can't export modules that have been defined elsewhere. We can also rename exports so their external name differs from their local names.

> **NOTE**
>
> `.mjs` is an extension used for JavaScript modules that helps us distinguish between module files and classic scripts (.js). The `.mjs` extension ensures that corresponding files are parsed as a module by runtimes and build tools. (For example, Node.js, Babel)

The following example shows three modules for bakery staff, the functions they perform while baking, and the bakery itself. We see how functionality that is exported by one module is imported and used by the other.

```
// Filename: staff.mjs
// ===================================
```

```
// specify (public) exports that can be consumed by other
modules
export const baker = {
   bake(item) {
      console.log( `Woo! I just baked ${item}` );
    }
};

// Filename: cakeFactory.mjs
// ========================================
// specify dependencies
import baker from "/modules/staff.mjs";

export const oven = {
    makeCupcake(toppings) {
       baker.bake( "cupcake", toppings );
    },
    makeMuffin(mSize) {
        baker.bake( "muffin", size );
    }
}

// Filename: bakery.mjs
// ========================================
import {cakeFactory} from "/modules/cakeFactory.mjs";
cakeFactory.oven.makeCupcake( "sprinkles" );
cakeFactory.oven.makeMuffin( "large" );
```

Typically, a module file contains several related functions, constants, and variables. You can collectively export these at the end of the file using a single export statement followed by a comma-separated list of the module resources you want to export.

```
// Filename: staff.mjs
// ========================================
const baker = {
  //baker functions
};
const pastryChef = {
  //pastry chef functions
};
```

```
const assistant = {
  //assistant functions
};

export { baker, pastryChef, assistant };
```

Similarly, you can only import the functions you need.

```
import {baker, assistant} from "/modules/staff.mjs";
```

You can tell browsers to accept 'script' tags that contain JavaScript modules by specifying the `type` attribute with a value of `module`:

```
<script type="module" src="main.mjs"></script>
<script nomodule src="fallback.js"></script>
```

The `nomodule` attribute tells modern browsers not to load a classic script as a module. This is useful for fallback scripts that don't use the module syntax. It allows you to use the module syntax in your HTML and have it work in browsers that don't support it. This is useful for several reasons, including performance. Modern browsers don't require polyfilling for modern features, allowing you to serve the larger transpiled code to legacy browsers alone.

Module Objects

A cleaner approach to importing and using module resources is to import the module as an object. This makes all the exports available as members of the object.

```
// Filename: cakeFactory.mjs

import * as Staff from "/modules/staff.mjs";

export const oven = {
    makeCupcake(toppings) {
       Staff.baker.bake( "cupcake", toppings );
    },
```

```
    makePastry(mSize) {
        Staff.pastryChef.make( "pastry", type );
    }
}
```

Modules Loaded From Remote Sources

ES2015+ also supports remote modules (e.g., third-party libraries), making it simplistic to load modules from external locations. Here's an example of pulling in the module we defined above and utilizing it:

```
import {cakeFactory} from
"https://example.com/modules/cakeFactory.mjs"; // eagerly
loaded static import

cakeFactory.oven.makeCupcake( "sprinkles" );
cakeFactory.oven.makeMuffin( "large" );
```

Static Imports

The type of import discussed above is called static import. The module graph needs to be downloaded and executed with static import before the main code can run. This can sometimes lead to the eager loading of a lot of code up front on the initial page load, which can be expensive and delay key features being available earlier.

```
import {cakeFactory} from "/modules/cakeFactory.mjs"; //
eagerly loaded static import

cakeFactory.oven.makeCupcake( "sprinkles" );
cakeFactory.oven.makeMuffin( "large" );
```

Dynamic Imports

Sometimes, you don't want to load a module up-front but on-demand when needed. Lazy-loading modules allows you to load what you need when

needed. For example, when the user clicks a link or a button. This improves the initial load-time performance. Dynamic import was introduced to make this possible.

Dynamic import introduces a new function-like form of import. `import(url)` returns a promise for the module namespace object of the requested module, which is created after fetching, instantiating, and evaluating all of the module's dependencies, as well as the module itself. Here is an example that shows dynamic imports for the `cakeFactory` module.

```
form.addEventListener("submit", e => {
  e.preventDefault();
  import("/modules/cakeFactory.js")
    .then((module) => {
      // Do something with the module.
      module.oven.makeCupcake("sprinkles");
      module.oven.makeMuffin("large");
    });
});
```

Dynamic import can also be supported using the `await` keyword:

```
let module = await import("/modules/cakeFactory.js");
```

With dynamic import, the module graph is only downloaded and evaluated when the module is used.

Popular patterns like Import on Interaction and Import on Visibility can be easily implemented in vanilla JavaScript using the Dynamic Import feature.

Import on Interaction

Some libraries may only be required when a user starts interacting with a particular feature on the web page. Typical examples are chat widgets, complex dialog boxes, or video embeds. Libraries for these features need not be imported on page load but can be loaded when the user interacts with them

(for example, by clicking on the component facade or placeholder). The action can trigger the dynamic import of respective libraries followed by the function call to activate the desired functionality.

For example, you can implement an onscreen sort function using the external lodash.sortby module, which is loaded dynamically.

```
const btn = document.querySelector('button');

btn.addEventListener('click', e => {
  e.preventDefault();
  import('lodash.sortby')
    .then(module => module.default)
    .then(sortInput()) // use the imported dependency
    .catch(err => { console.log(err) });
});
```

Import on visibility

Many components are not visible on the initial page load but become visible as the user scrolls down. Since users may not always scroll down, modules corresponding to these components can be lazy-loaded when they become visible. The IntersectionObserver API can detect when a component placeholder is about to become visible, and a Dynamic import can load the corresponding modules.

Modules For The Server

Node 15.3.0 onwards supports JavaScript modules. They function without an experimental flag and are compatible with the rest of the npm package ecosystem. Node treats files ending in .mjs and .js with a top-level type field value of module as JavaScript Modules.

```
{
  "name": "js-modules",
  "version": "1.0.0",
  "description": "A package using JS Modules",
```

```
  "main": "index.js",
  "type": "module",
  "author": "",
  "license": "MIT"
}
```

Advantages of using Modules

Modular programming and the use of modules offer several unique advantages. Some of these are as follows:

- **Modules scripts are evaluated only once:** The browser evaluates module scripts only once, while classic scripts get evaluated as often as they are added to the DOM. This means that with JS modules, if you have an extended hierarchy of dependent modules, the module that depends on the innermost module will be evaluated first. This is a good thing because it means that the innermost module will be evaluated first and will have access to the exports of the modules that depend on it.
- **Modules are auto-deferred:** Unlike other script files, where you have to include the `defer` attribute if you don't want to load them immediately, browsers automatically defer the loading of modules.
- **Modules are easy to maintain and reuse:** Modules promote decoupling pieces of code that can be maintained independently without significant changes to other modules. They also allow you to reuse the same code in multiple different functions.
- **Modules provide namespacing:** Modules create a private space for related variables and constants so that they can be referenced via the module without polluting the global namespace.
- **Modules enable dead code elimination:** Before the introduction of modules, unused code files had to be manually removed from projects. With module imports, bundlers such as Webpack and rollup can automatically identify unused modules and eliminate them. Dead code may be removed before adding it to the bundle. This is known as tree-

shaking.

All modern browsers support module import and export, and you can use them without any fallback.

Classes With Constructors, Getters & Setters

In addition to modules, ES2015+ also allows defining classes with constructors and some sense of privacy. JavaScript Classes are defined with the `class` keyword. In the following example, we define a class `Cake` with a constructor and two getters and setters:

```
class Cake{

    // We can define the body of a class constructor
    // function by using the keyword constructor
    // with a list of class variables.
    constructor( name, toppings, price, cakeSize ){
        this.name = name;
        this.cakeSize = cakeSize;
        this.toppings = toppings;
        this.price = price;
    }

    // As a part of ES2015+ efforts to decrease the
unnecessary
    // use of function for everything, you will notice that
it is
    // dropped for cases such as the following. Here an
identifier
    // followed by an argument list and a body defines a
new method

    addTopping( topping ){
        this.toppings.push( topping );
    }

    // Getters can be defined by declaring get before
    // an identifier/method name and a curly body.
    get allToppings(){
```

```
        return this.toppings;
    }

    get qualifiesForDiscount(){
        return this.price > 5;
    }

    // Similar to getters, setters can be defined by using
    // the set keyword before an identifier
    set cakeSize( size ){
        if ( size < 0){
            throw new Error( "Cake must be a valid size -
either small, medium or large");
        }
       this.cakeSize = size;
    }
}

// Usage
let cake = new Cake( "chocolate", ["chocolate chips"], 5,
"large" );
```

JavaScript classes are built on prototypes and are a special category of Javascript functions that need to be defined before they can be referenced.

You can also use the `extends` keyword to indicate that a class inherits from another class.

```
class BirthdayCake extends Cake {
  surprise() {
    console.log(`Happy Birthday!`);
  }
}

let birthdayCake = new BirthdayCake( "chocolate",
["chocolate chips"], 5, "large" );
birthdayCake.surprise();
```

All modern browsers and Node support ES2015 classes. They are also compatible with the new-style class syntax introduced in ES6.

The difference between JavaScript Modules and Classes is that Modules are imported and exported, and Classes are defined with the `class` keyword.

Reading through, one may also notice the lack of the word 'function' in the above examples. This isn't a typo error: TC39 has made a conscious effort to decrease our abuse of the `function` keyword for everything, hoping it will help simplify how we write code.

JavaScript classes also support the `super` keyword, which allows you to call a parent class' constructor. This is useful for implementing the self-inheritance pattern. You can use `super` to call the methods of the superclass.

```
class Cookie {
  constructor(flavor) {
    this.flavor = flavor;
  }

  showTitle() {
    console.log(`The flavor of this cookie is
${this.flavor}.`);
  }
}

class FavoriteCookie extends Cookie {
  showTitle() {
    super.showTitle();
    console.log(`${this.flavor} is amazing.`);
  }
}

let myCookie = new FavoriteCookie('chocolate');
myCookie.showTitle();
// The flavor of this cookie is chocolate.
// chocolate is amazing.
```

Modern JavaScript supports public and private class members. Public class members are accessible to other classes. Private class members are accessible only to the class in which they are defined. Class fields are, by default, public. Private class fields can be created by using the # hash prefix.

```
class CookieWithPrivateField {
  #privateField;
}

class CookieWithPrivateMethod {
  #privateMethod() {
    return 'delicious cookies';
  }
}
```

JavaScript classes support static methods and properties using the `static` keyword. Static members can be referenced without instantiating the class. You can use static methods to create utility functions and static properties for holding configuration or cached data.

```
class Cookie {
  constructor(flavor) {
    this.flavor = flavor;
  }
  static brandName = "Best Bakes";
  static discountPercent = 5;
}
console.log(Cookie.brandName); //output = "Best Bakes"
```

Classes in JavaScript frameworks

Over the last few years, some modern JavaScript libraries and frameworks - notably React - have introduced alternatives to classes. React Hooks make it possible to use React state and lifecycle methods without an ES2015 class component. Before Hooks, React developers had to refactor functional components as class components so that they handle state and lifecycle methods. This was often tricky and required an understanding of how ES2015 classes work. React Hooks are functions that allow you to manage a component‘s state and lifecycle methods without relying on classes.

Note that several other approaches to building for the web, such as the Web Components community, continue to use classes as a base for component development.

Summary

This chapter introduced JavaScript language syntax for Modules and Classes. These features allow us to write code while adhering to object-oriented design and modular programming principles. We will also utilize these concepts to categorize and describe different design patterns. The next chapter talks about the different categories of design patterns.

Related Reading

- JavaScript Modules on v8
- JavaScript Modules on MDN

Chapter 6. Categories of Design Patterns

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the sixth chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

This chapter documents the three main categories of design patterns and the different patterns that fall under them. While every design pattern addresses a specific object-oriented design problem or issue, we can draw parallels between the solutions based on how they solve these issues. This forms the basis for the categorization of design patterns.

Background

Gamma, Helm, Johnson & Vlissides (1995), in their book, Design Patterns: Elements of Reusable Object-Oriented Software, describe a design pattern as:

> *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 categorized based on the type of problem they solve. The three principal categories of design patterns are:

- Creational design patterns
- Structural design patterns
- Behavioral design patterns

In the following sections, we'll review these three with a few examples of the patterns that fall into each category.

Creational Design Patterns

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

Some 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 need not change. They also assist in recasting parts of the system that 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. They identify common communication patterns among objects and provide solutions that distribute the responsibility of communication among different objects, thereby increasing communication flexibility. Essentially, behavioral patterns abstract actions from objects that take the action

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

Design Pattern Classes

Elyse Nielsen in 2004 created a "classes" table to summarize the 23 GOF design patterns. I found this table extremely useful in my early days of learning about design patterns. I've modified it where necessary to suit our discussion on design patterns.

I recommend using this table as a reference, but remember that we will discuss several other patterns not mentioned here later in the book.

> **NOTE**
>
> We discussed JavaScript ES2015+ classes in Chapter 5. JavaScript classes and objects will be relevant when you review the following table.

Let us now proceed to review the table.

Creational	Based on the concept of creating an object
Class	
Factory method	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	Matches interfaces of different classes so classes can work together despite incompatible interfaces
Object	
Adapter	Matches interfaces of different classes so 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 that makes the total object more than just the sum of its parts
Decorator	Dynamically adds 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 placeholder 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 defers 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	A way to separate the execution of a command from its invoker
Iterator	Sequentially accesses 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	Captures 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	Alters 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

Summary

This chapter introduced categories of design patterns and explained the distinction between creational, structural, and behavioral patterns. We discussed the differences between these three categories and GOF patterns in each category. We also reviewed the “classes” table that shows how the GOF patterns relate to the concepts of classes and objects.

These first few chapters have covered theoretical details about design patterns and some basics of JavaScript syntax. With this background, we are now in a position to job into some practical examples of design patterns in JavaScript.

Chapter 7. JavaScript Design Patterns

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the seventh chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Not every design pattern may be relevant or required in the web development context. I have identified a few timeless patterns that can be helpful when applied in JavaScript. In this chapter, we will explore JavaScript implementations of different classic and modern design patterns such as the factory pattern, the command pattern, the flyweight pattern, and many others.

Choosing a Pattern

Developers commonly wonder whether there is an ideal pattern or set of patterns they should use in their workflow. There isn't a single correct answer to this question; each script and web application we work on will likely have distinct individual needs. We must consider whether 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). At the same time, others may be too small for decoupling to be a concern.

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 Constructor Pattern”
- “The Module Pattern”
- “The Revealing Module Pattern”
- “The Singleton Pattern”
- “The Observer Pattern”
- “The Mediator Pattern”
- “The Prototype Pattern”
- “The Command Pattern”
- “The Facade Pattern”
- “The Factory Pattern”
- “The Mixin Pattern”
- “The Decorator Pattern”
- “Flyweight”

The Constructor Pattern

A constructor is a special method used to initialize a newly created object once the memory has been allocated for it. With ES2015+, the syntax for creating classes with constructors was introduced to JavaScript. This enables the creation of objects as an instance of a class using the default constructor.

In JavaScript, almost everything is an object, and classes are syntactic sugar

for JavaScript's prototypal approach to inheritance. With classic JavaScript, we were 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 to set the values of member properties and methods when the object is first created.

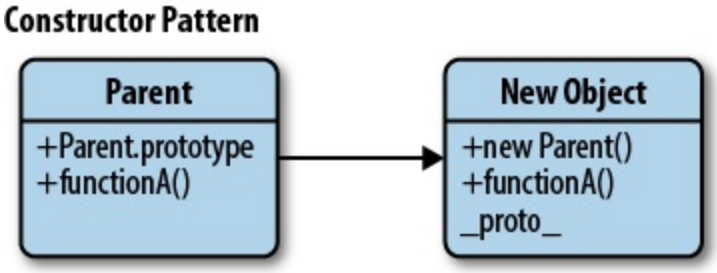


Figure 7-1. Constructor pattern

Object Creation

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

```
// Each of the following options will create a new empty
object
const newObject = {};

// or
const newObject = Object.create(Object.prototype);

// or
const newObject = new Object();
```

Here, we have declared each object as a constant, which creates a read-only block-scoped variable. In the final example, the `Object` constructor creates an object wrapper for a specific value, or where no value is passed; it creates an empty object and returns it.

You can now assign keys and values to an object in the following ways:

```
// ECMAScript 3 compatible approaches
```

```
// 1. Dot syntax

// Set properties
newObject.someKey = "Hello World";

// Get properties
var key = newObject.someKey;



// 2. Square bracket syntax

// Set properties
newObject["someKey"] = "Hello World";

// Get properties
var key = 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
});

// 4. If the above feels a little difficult to read, a
short-hand could
// be written as follows:

var defineProp = function ( obj, key, value ){
  config.value = value;
  Object.defineProperty( obj, key, config );
};
```

```
// To use, we then create a new empty "person" object
var person = Object.create( null );

// Populate the object with properties
defineProp( person, "car",  "Delorean" );
defineProp( person, "dateOfBirth", "1981" );
defineProp( person, "hasBeard", false );


// 5. 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.
```

You can even use these methods for inheritance as follows:

```
// ES2015+ keywords/syntax used: const
// Usage:

// Create a race car driver that inherits from the person
object
const 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 discussed earlier in Chapter 5, JavaScript classes were introduced in ES2015, allowing us to define templates for JavaScript Objects and implement encapsulation and inheritance using JavaScript.

To recap, classes must include and declare a method named constructor() which will be used to instantiate a new object. The keyword `new` allows us to call the constructor. The keyword `this` inside a constructor references the new object created. The following example shows a basic constructor.

```
class Car {
    constructor(model, year, miles) {
        this.model = model;
        this.year = year;
        this.miles = miles;
    }

    toString() {
        return `${this.model} has done ${this.miles}
miles`;
    }
}

// Usage:

// We can create new instances of the car
let civic = new Car('Honda Civic', 2009, 20000);
let 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 suffers from some problems. One is that it makes inheritance difficult, and the other is that functions such as toString() are redefined for each new object created using the Car constructor. This isn't optimal as all of the instances of the Car type should ideally share the same function.

Constructors with Prototypes

Prototypes in JavaScript allow you to easily define methods for all instances of a particular object, be it a function or a class. 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, you can have multiple `Car` objects that access the same prototype. We can thus extend the original example as follows:

```
class Car {
    constructor(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
// We still could use Object.prototype for adding new
methods,
// because internally we use the same structure

Car.prototype.toString = function() {
    return `${this.model} has done ${this.miles} miles`;
};

// Usage:
let civic = new Car('Honda Civic', 2009, 20000);
```

```
let mondeo = new Car('Ford Mondeo', 2010, 5000);

console.log(civic.toString());
console.log(mondeo.toString());
```

Above, all Car objects will now share a single instance of the `toString()` method.

The Module Pattern

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

Classic JavaScript had several options for implementing modules, such as:

- Object literal notation
- The Module pattern
- AMD modules
- CommonJS modules

We have already discussed modern JavaScript modules (also known as "ES modules" or "ECMAScript modules,") in the earlier chapter about Chapter 5. We will primarily use ES modules for the examples in this section.

Before ES2015, CommonJS modules or AMD modules were popular alternatives as they allowed you to export the contents of a module. We will be exploring AMD, CommonJS, and UMD modules later on in the book in the section [Link to Come]. First, let us understand the Module Pattern and its origins.

The Module pattern is based partly on object literals, 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 followed by a colon. It would be best if you did not use a comma after the final name/value pair in the object, as this may result in errors.

```
const myObjectLiteral = {
    variableKey: variableValue,
    functionKey() {
        // ...
    }
};
```

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 new block. Outside of an object, new members may be added to it using the assignment as follows `myModule.property = "someValue";`.

We can see a complete example of a module defined using object literal notation below:

```
const 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() {
        console.log('Where is Paul Irish debugging
today?');
    },
    // output a value based on the current configuration
    reportMyConfig() {
        console.log(
```

```
            `Caching is: ${this.myConfig.useCaching ?
'enabled' : 'disabled'}`
        );
    },
    // override the current configuration
    updateMyConfig(newConfig) {
        if (typeof newConfig === 'object') {
            this.myConfig = newConfig;
            console.log(this.myConfig.language);
        }
    },
};

// Outputs: What is Paul Irish debugging 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 provided a way to encapsulate and organize code. Rebecca Murphey has previously written about this topic in depth should you wish to read into object literals further.

The Module Pattern

The Module pattern was initially defined to provide private and public encapsulation for classes in conventional software engineering.

At one point, organizing a JavaScript application of any reasonable size was a challenge. Developers would rely on separate scripts to split and manage reusable chunks of logic, and it wasn‘t surprising to find 10-20 scripts being

imported manually in an HTML file to keep things tidy. Using objects, the module pattern was just one way to encapsulate logic in a file with both public and “private” methods. Over time, several custom module systems came about to make this smoother. Now, developers can use JavaScript modules to organize objects, functions, classes, or variables such that they can be easily exported or imported into other files. This helps prevent conflicts between classes or function names included in different modules.

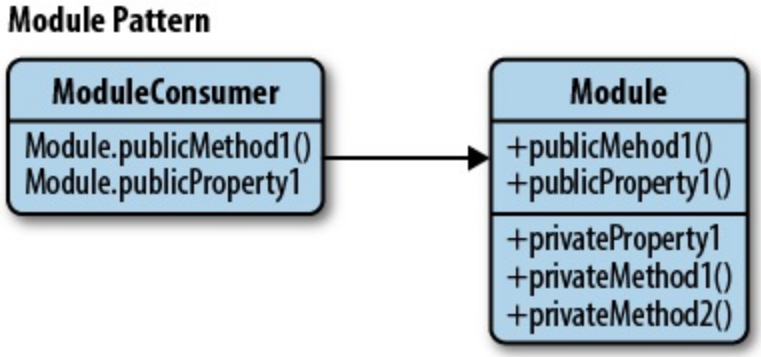


Figure 7-2. Module pattern

Privacy

The Module pattern encapsulates the “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, you only expose the public API, keeping everything else within the closure private.

This gives us a clean solution where the shielding logic does the heavy lifting while we only expose an interface we wish other parts of our application to use. The pattern utilizes an immediately-invoked function expression[1] where an object is returned.

Note that there isn’t an explicitly true sense of “privacy” inside JavaScript because it doesn’t have access modifiers, unlike some traditional languages. You can’t technically declare variables as public or private, 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. However, variables or methods defined within the returning object are available to everyone.

A workaround to implement privacy of variables in returned objects uses WeakMap() discussed later in this chapter in Modern Module Pattern with WeakMap. WeakMap() only takes objects as keys and cannot be iterated. Thus, the only way to access the object inside a module is through its reference. Outside the module, you can only access it through a public method defined within it. Thus, it ensures privacy for the object.

History

From a historical perspective, the Module pattern was originally developed in 2003 by several people, including Richard Cornford. Douglas Crockford later popularized it in his lectures. Another piece of trivia is that some of its features may appear quite familiar if you've ever played with Yahoo's YUI library. The reason for this is that the Module pattern was a strong influence on YUI when creating its components.

Examples

Let's begin looking at implementing the Module pattern by creating a self-contained module. We use the `import` and `export` keywords in our implementation. To recap our previous discussion, `export` allows you to provide access to module features outside the module. At the same time, `import` enables us to import bindings exported by a module to our script.

```
let counter = 0;

const testModule = {
  incrementCounter() {
    return counter++;
  },
  resetCounter() {
    console.log(`counter value prior to reset:
${counter}`);
    counter = 0;
  },
};
```

```
// Default export module, without name
export default testModule;

// Usage:

// Import module from path
import testModule from './testModule';

// 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 cannot directly read the value of our `incrementCounter()` or `resetCounter()`. The `counter` variable is entirely 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 two functions are the only code able to access its scope. Our methods are effectively namespaced, so in the test section of our code, we need to prefix any calls with the module's name (e.g., `testModule`).

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

```
// A private counter variable
let myPrivateVar = 0;

// A private function which logs any arguments
const myPrivateMethod = foo => {
  console.log(foo);
};

const myNamespace = {
  // A public variable
  myPublicVar: 'foo',
```

```
  // A public function utilizing privates
  myPublicFunction(bar) {
    // Increment our private counter
    myPrivateVar++;

    // Call our private method using bar
    myPrivateMethod(bar);
  },
};

export default myNamespace;
```

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, so other parts of our application cannot directly read it. It only exists within the module's closure, and so the only methods able to access it are those with access to its scope (i.e., `addItem()`, `getItem()`, etc.).

```
// privates

const basket = [];

const doSomethingPrivate = () => {
  //...
};

const doSomethingElsePrivate = () => {
  //...
};

// Create an object exposed to the public
const basketModule = {
  // Add items to our basket
  addItem(values) {
    basket.push(values);
  },

  // Get the count of items in the basket
```

```
  getItemCount() {
    return basket.length;
  },

  // Public alias to a private function
  doSomething() {
    doSomethingPrivate();
  },

  // Get the total value of items in the basket
  // The reduce() method applies a function against an
accumulator and each element in the array (from left to
right) to reduce it to a single value.
  getTotal() {
    return basket.reduce((currentSum, item) => item.price +
currentSum, 0);
  },
};

export default basketModule;
```

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

```
// Import module from path
import basketModule from './basketModule';

// 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`. All our functions are wrapped in this module, giving us several advantages, such as:

- The freedom to have private functions that can only be consumed by our module. They aren't exposed to the rest of the page (only our exported API is), so they're considered truly private.
- Given that functions are usually declared 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 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 pattern variation demonstrates how you can pass globals (e.g., `jQuery`, `Underscore`) as arguments to our module's anonymous function. This effectively allows us to *import* and locally alias them as we wish.

```
const privateMethod1 = () => {
    $(".container").html("test");
}

const privateMethod2 = () => {
    console.log(_.min([10, 5, 100, 2, 1000]));
}

const myModule = {
    publicMethod() {
        privateMethod1();
    }
};

// Default export module, without name
export default myModule;


// Import module from path
import myModule from './MyModule';

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.

```
// Module object
const module = {};
const privateVariable = 'Hello World';

const privateMethod = () => {
  // ...
```

```
};

module.publicProperty = 'Foobar';
module.publicMethod = () => {
  console.log(privateVariable);
};

// Default export module, without name
export default module;
```

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, we only have access to the values that we explicitly exported using the `export` keyword. Values we didn't expressly export are private and only available within the module. This reduces the risk of accidentally polluting the global scope. You don't have to fear that you will accidentally overwrite values created by developers using your module that may have had the same name as your private value: it prevents naming collisions and global scope pollution.

With the module pattern, we can encapsulate parts of our code that should not be publicly exposed. They make working with multiple dependencies and namespaces less risky. Note that a transpiler such as Babel is needed to use ES2015 modules in all JavaScript runtimes.

Disadvantages

The disadvantages of the Module pattern are that we access both public and private members differently. When we wish to change the visibility, we must make changes to each place we use the member.

We also can't access private members in methods we added to the object later. That said, in many cases, the Module pattern is still quite helpful 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 interacting 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.

Modern Module Pattern with WeakMap

Introduced to JavaScript in ES6, the WeakMap object is a collection of key/value pairs in which the keys are weakly referenced. The keys must be objects, and the values can be arbitrary. The object is essentially a map where keys are held weakly. This means that keys will be a target for garbage collection (GC) if there is no active reference to the object. Let us look at an implementation of the module pattern that uses the WeakMap object.

Basic Module Definition

```
let _counter = new WeakMap();

class Module {
    constructor() {
        _counter.set(this, 0);
    }
    incrementCounter() {
        let counter = _counter.get(this);
        counter++;
        _counter.set(this, counter);

        return _counter.get(this);
    }
    resetCounter() {
        console.log(`counter value prior to reset:
${_counter.get(this)}`);
```

```
            _counter.set(this, 0);
        }
    }

    const testModule = new Module();

    // Usage:

    // Increment our counter
    testModule.incrementCounter();
    // Check the counter value and reset
    // Outputs: counter value prior to reset: 1
    testModule.resetCounter();
```

Namespaces with Public/Private variables

```
    const myPrivateVar = new WeakMap();
    const myPrivateMethod = new WeakMap();

    class MyNamespace {
        constructor() {
            // A private counter variable
            myPrivateVar.set(this, 0);
            // A private function which logs any arguments
            myPrivateMethod.set(this, foo => console.log(foo));
            // A public variable
            this.myPublicVar = 'foo';
        }
        // A public function utilizing privates
        myPublicFunction(bar) {
            let privateVar = myPrivateVar.get(this);
            const privateMethod = myPrivateMethod.get(this);
            // Increment our private counter
            privateVar++;
            myPrivateVar.set(this, privateVar);
            // Call our private method using bar
            privateMethod(bar);
        }
    }
```

Shopping Basket Implementation

```
const basket = new WeakMap();
const doSomethingPrivate = new WeakMap();
const doSomethingElsePrivate = new WeakMap();

class BasketModule {
    constructor() {
        // privates
        basket.set(this, []);
        doSomethingPrivate.set(this, () => {
            //...
        });
        doSomethingElsePrivate.set(this, () => {
            //...
        });
    }
    // Public aliases to a private functions
    doSomething() {
        doSomethingPrivate.get(this)();
    }
    doSomethingElse() {
        doSomethingElsePrivate.get(this)();
    }
    // Add items to our basket
    addItem(values) {
        const basketData = basket.get(this);
        basketData.push(values);
        basket.set(this, basketData);
    }
    // Get the count of items in the basket
    getItemCount() {
        return basket.get(this).length;
    }
    // Get the total value of items in the basket
    getTotal() {
        return basket
            .get(this)
            .reduce((currentSum, item) => item.price +
currentSum, 0);
    }
}
```

Modules with modern libraries

You can use the Module pattern when building applications with JavaScript libraries such as React. Let‘s say you have a large number of custom components created by your team. In that case, you can separate each component in its own file, essentially creating a module for every component. Here is an example of a button component customized from the material-ui button component and exported as a module.

```
import React from "react";
import Button from "@material-ui/core/Button";

const style = {
  root: {
    borderRadius: 3,
    border: 0,
    color: "white",
    margin: "0 20px"
  },
  primary: {
    background: "linear-gradient(45deg, #FE6B8B 30%,
#FF8E53 90%)"
  },
  secondary: {
    background: "linear-gradient(45deg, #2196f3 30%,
#21cbf3 90%)"
  }
};

export default function CustomButton(props) {
  return (
    <Button {...props} style={{ ...style.root,
...style[props.color] }}>
      {props.children}
    </Button>
  );
}
```

The Revealing Module Pattern

Now that we are a little more familiar with the Module pattern let’s look at a

slightly improved version — Christian Heilmann's Revealing Module pattern.

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

His efforts resulted in an updated pattern where we can simply define all functions and variables in the private scope and return an anonymous object with pointers to the private functionality we wished to reveal as public.

With the modern way of implementing modules in ES2015+, the scope of functions and variables defined in the module is already private. Also, we use export and import to reveal whatever needs to be revealed.

An example of the use of the Revealing Module pattern with ES2015+ is as follows:

```
let privateVar = 'Ben Cherry';
const publicVar = 'Hey there!';

const privateFunction = () => {
  console.log(`Name:${privateVar}`);
};

const publicSetName = strName => {
  privateVar = strName;
};

const publicGetName = () => {
  privateFunction();
};

// Reveal public pointers to
// private functions and properties
const myRevealingModule = {
  setName: publicSetName,
  greeting: publicVar,
  getName: publicGetName,
};
```

```
export default myRevealingModule;

// Usage:
import myRevealingModule from './myRevealingModule';

myRevealingModule.setName('Paul Kinlan');
```

In the above example, we reveal the private variable `privateVar` through its public get and set methods, publicSetName, and publicGetName.

You can also use the pattern to reveal private functions and properties with a more specific naming scheme.

```
let privateCounter = 0;

const privateFunction = () => {
    privateCounter++;
}

const publicFunction = () => {
    publicIncrement();
}

const publicIncrement = () => {
    privateFunction();
}

const publicGetCount = () => privateCounter;

// Reveal public pointers to
// private functions and properties
const myRevealingModule = {
    start: publicFunction,
    increment: publicIncrement,
    count: publicGetCount
};

export default myRevealingModule;

// Usage:
import myRevealingModule from './myRevealingModule';
```

```
myRevealingModule.start();
```

Advantages

This pattern allows the syntax of our scripts to be more consistent. It also makes it easier to understand 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.

As a result, modules created with the Revealing Module pattern may be more fragile than those created with the original Module pattern, and you should take care when using it.

The Singleton Pattern

The Singleton pattern is a design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. Classically, you can implement the Singleton pattern by creating a class with a method that creates a new instance of the class only if one doesn't already exist. If an instance already exists, it simply returns a reference to that object.

Singletons differ from static classes (or objects) as we can delay their initialization because they require certain information that may not be available during initialization time. Any code that is unaware of a previous

reference to the Singleton class cannot easily retrieve it. This is because it is neither the object nor “class” that a Singleton returns; it’s a structure. Think of how closured variables aren’t actually closures - the function scope that provides the closure is the closure.

ES2015+ allows us to implement the singleton pattern to create a global instance of a JavaScript class that is instantiated once. You can expose the singleton instance through a module export. This makes access to it more explicit and controlled and differentiates it from other global variables. You cannot create a new class instance but can read/modify the instance using public get and set methods defined in the class.

We can implement a Singleton as follows:

```
// Instance stores a reference to the Singleton
let instance;

// Private methods and variables
const privateMethod = () => {
    console.log('I am private');
  };
const privateVariable = 'Im also private';
const randomNumber = Math.random();

// Singleton
class MySingleton {
  // Get the Singleton instance if one exists
  // or create one if it doesn't
  constructor() {
    if (!instance) {
      // Public property
      this.publicProperty = 'I am also public';
      instance = this;
    }

    return instance;
  }

  // Public methods
  publicMethod() {
    console.log('The public can see me!');
```

```
  }

  getRandomNumber() {
    return randomNumber;
  }
}
// [ES2015+] Default export module, without name
export default MySingleton;


// Instance stores a reference to the Singleton
let instance;

// Singleton
class MyBadSingleton {
    // Always create a new Singleton instance
    constructor() {
        this.randomNumber = Math.random();
        instance = this;

        return instance;
    }

    getRandomNumber() {
        return this.randomNumber;
    }
}

export default MyBadSingleton;


// Usage:
import MySingleton from './MySingleton';
import MyBadSingleton from './MyBadSingleton';

const singleA = new MySingleton();
const singleB = new MySingleton();
console.log(singleA.getRandomNumber() ===
singleB.getRandomNumber());
// true

const badSingleA = new MyBadSingleton();
const badSingleB = new MyBadSingleton();
```

```
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. The GoF book describes the *applicability* of the Singleton pattern as follows:

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- 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:

```
constructor() {
    if (this._instance == null) {
        if (isFoo()) {
            this._instance = new FooSingleton();
        } else {
            this._instance = new BasicSingleton();
        }
    }

    return this._instance;
}
```

Here, the `constructor` 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 significant for a Singleton?:

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

It is essential to note the difference between a static instance of a class (object) and a Singleton. While you can implement a Singleton as a static instance, it can also be constructed lazily, without the need for resources or memory until it is needed.

Suppose we have a static object that we can initialize directly. In that case, 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 shouldn't be overused - the same way we shouldn't overuse other patterns.

In practice, it helps to use the Singleton pattern when exactly one object is needed to coordinate others across a system. Here is one example that uses the pattern in this context:

```
// options: an object containing configuration options for
the singleton
// e.g const options = { name: "test", pointX: 5};
class Singleton {
    constructor(options = {}) {
      // set some properties for our singleton
      this.name = 'SingletonTester';
      this.pointX = options.pointX || 6;
      this.pointY = options.pointY || 10;
    }
  }

  // our instance holder
  let instance;

  // an emulation of static variables and methods
  const SingletonTester = {
    name: 'SingletonTester',
    // Method for getting an instance. It returns
    // a singleton instance of a singleton object
    getInstance(options) {
```

```
    if (instance === undefined) {
      instance = new Singleton(options);
    }

    return instance;
  },
};

const singletonTest = SingletonTester.getInstance({
  pointX: 5,
});

// Log the output of pointX just to verify it is correct
// Outputs: 5
console.log(singletonTest.pointX);
```

While 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. Unlike C++ or Java, where you have to define a class to create an object, JavaScript allows you to create objects directly. Thus, you can create one such object directly instead of defining a singleton class. In contrast, using singleton classes in JavaScript has some disadvantages.

Identifying Singletons can be difficult

If you are importing a large module, you will be unable to recognize that a particular class is a Singleton. As a result, you may accidentally use it as a regular class to instantiate multiple objects and incorrectly update it instead.

Challenging to test

Singletons can be more difficult to test due to issues ranging from hidden dependencies, difficulty creating multiple instances, difficulty in stubbing dependencies, and so on.

Need for careful orchestration

An everyday use case for Singletons would be to store data that will be required across the global scope, such as user credentials or cookie data

> that can be set once and consumed by multiple components. Implementing the correct execution order becomes essential so that data is always consumed after it becomes available and not the other way around. This may become challenging as the application grows in size and complexity.

Miller Medeiros has previously recommended this excellent article on the Singleton and its various issues for further reading, and the comments to this article discuss 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.

State management in React

Developers using React for web development can rely on the global state through state management tools such as Redux or React Context instead of Singletons. Unlike Singletons, these tools provide a read-only state rather than the mutable state.

Although the downsides to having a global state don't magically disappear by using these tools, we can at least ensure that the global state is mutated the way we intend it to since components cannot update it directly.

The Observer Pattern

The Observer pattern allows you to notify one object when another object changes without requiring the object to know about its dependents. Often this is a pattern where an object (known as a subject) maintains a list of objects depending on it (observers), automatically notifying them of any changes to its state. In modern frameworks, the observer pattern is used to inform components of state changes.

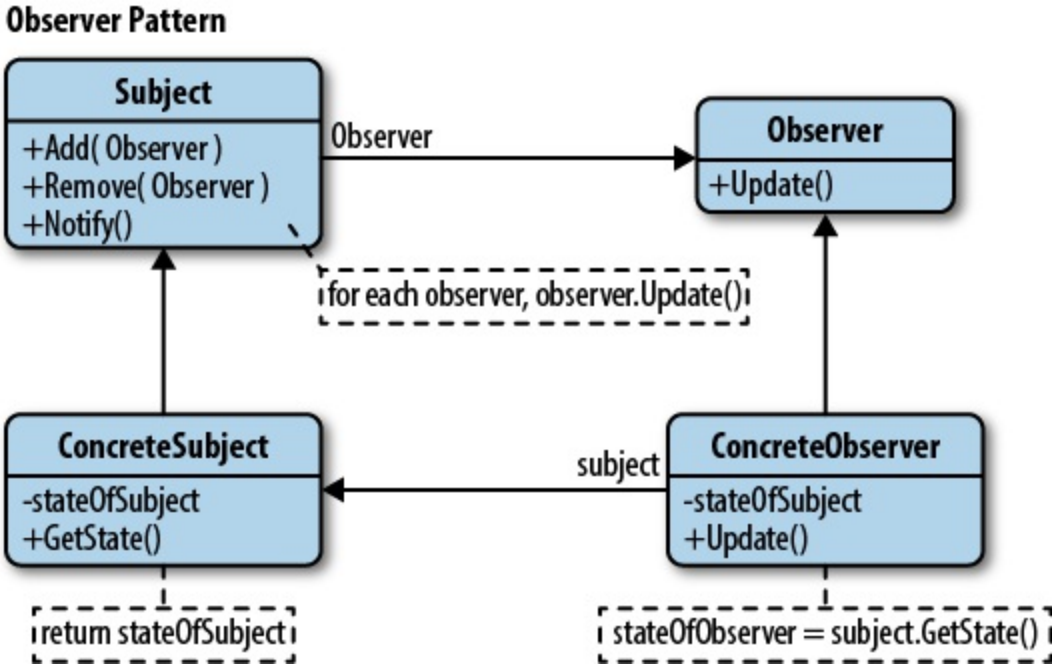


Figure 7-3. Observer pattern

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). When an observer no longer wishes to be notified of changes by the subject, they can be removed from the list of observers.

It's helpful 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 an `update` interface for objects that need to be notified of a Subject's changes in 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 the state is consistent with the Subject's

ES2015+ allows us to implement the observer pattern using JavaScript classes for observers and subjects with methods for `notify` and `update`.

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

```
class ObserverList {
    constructor() {
        this.observerList = [];
    }

    add(obj) {
        return this.observerList.push(obj);
    }

    count() {
        return this.observerList.length;
    }

    get(index) {
        if (index > -1 && index < this.observerList.length)
{
```

```
            return this.observerList[index];
        }
    }

    indexOf(obj, startIndex) {
        let i = startIndex;

        while (i < this.observerList.length) {
            if (this.observerList[i] === obj) {
                return i;
            }
            i++;
        }

        return -1;
    }

    removeAt(index) {
        this.observerList.splice(index, 1);
    }
}
```

Next, let’s model the Subject class that can add, remove, or notify observers on the observer list.

```
  class Subject {
    constructor() {
      this.observers = new ObserverList();
    }

    addObserver(observer) {
      this.observers.add(observer);
    }

    removeObserver(observer) {

  this.observers.removeAt(this.observers.indexOf(observer,
  0));
    }

    notify(context) {
      const observerCount = this.observers.count();
```

```
        for (let i = 0; i < observerCount; i++) {
        this.observers.get(i).update(context);
        }
    }
}
```

We then define a skeleton for creating new Observers. We will overwrite the `Update` functionality here later with custom behavior.

```
// The Observer
class Observer {
    constructor() {}
    update() {
        // ...
    }
}
```

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 will act as a subject, notifying other checkboxes that they should update to the checked state.
- A container for the new checkboxes added.

We then define `ConcreteSubject` and `ConcreteObserver` handlers to add new observers to the page and implement the updating interface. For this, we use inheritance to extend our Subject and Observer classes. The `ConcreteSubject` class encapsulates a checkbox and generates a notification when the main checkbox is clicked. `ConcreteObserver` encapsulates each of the observing checkboxes and implements the `Update` interface by changing the checked value of the checkboxes. See below for inline comments on how these work together in the context of our example.

Here is the HTML code:

```
<button id="addNewObserver">Add New Observer
```

```
checkbox</button>
<input id="mainCheckbox" type="checkbox"/>
<div id="observersContainer"></div>
```

Here is a sample script:

```
// References to our DOM elements

// Concrete Subject
class ConcreteSubject extends Subject {
    constructor(element) {
      // Call the constructor of the super class.
      super();
      this.element = element;

      // Clicking the checkbox will trigger notifications
to its observers
      this.element.onclick = () => {
        this.notify(this.element.checked);
      };
    }
  }

  // Concrete Observer

  class ConcreteObserver extends Observer {
    constructor(element) {
      super();
      this.element = element;
    }

    // Override with custom update behaviour
    update(value) {
      this.element.checked = value;
    }
  }

  // References to our DOM elements
  const addBtn = document.getElementById('addNewObserver');
  const container =
document.getElementById('observersContainer');
  const controlCheckbox = new ConcreteSubject(
```

```
    document.getElementById('mainCheckbox')
  );

  const addNewObserver = () => {
    // Create a new checkbox to be added
    const check = document.createElement('input');
    check.type = 'checkbox';
    const checkObserver = new ConcreteObserver(check);

    // Add the new observer to our list of observers
    // for our main subject
    controlCheckbox.addObserver(checkObserver);

    // Append the item to the container
    container.appendChild(check);
  };

  addBtn.onclick = addNewObserver;
}
```

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

While it's helpful to be aware of the Observer pattern quite often in the JavaScript world, we'll find it commonly implemented using a variation known as the Publish/Subscribe pattern. Although the two patterns are pretty similar, there are differences 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), as seen in Figure 7-4.

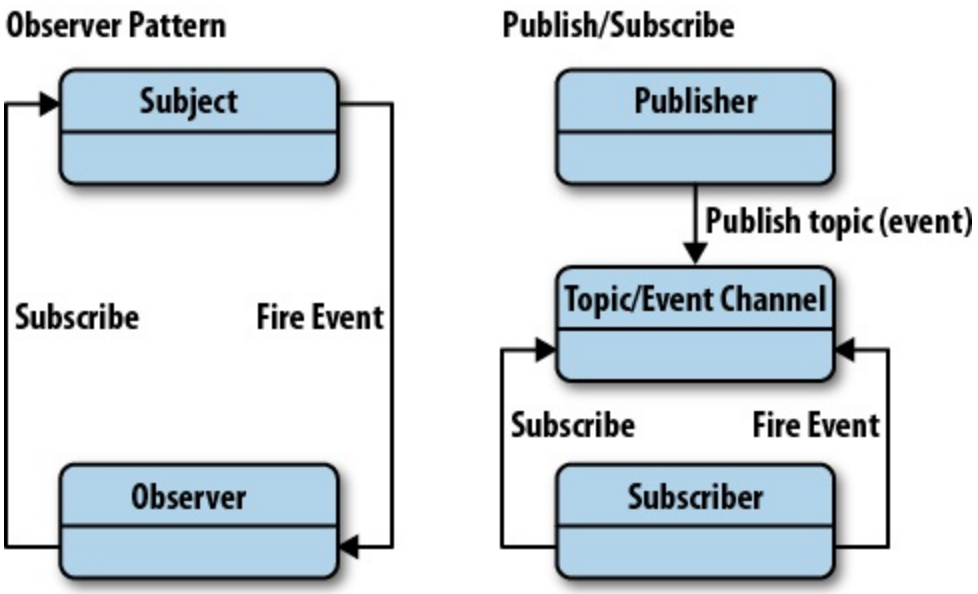


Figure 7-4. Publish/Subscribe

The Publish/Subscribe pattern, however, uses a topic/event channel that 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 pattern if provided with a functional implementation powering `publish()`, `subscribe()`, and `unsubscribe()` behind the scenes:

```
// A very simple new mail handler

// A count of the number of messages received
let mailCounter = 0;

// Initialize subscribers that will listen out for a topic
// with the name "inbox/newMessage".
```

```
// Render a preview of new messages
const subscriber1 = subscribe('inbox/newMessage', (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

const subscriber2 = subscribe('inbox/newMessage', (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 application parts. They also help us identify layers containing direct relationships that could 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 potential for reuse.

Further motivation for using the Observer pattern is in situations 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 excellent flexibility that may not be as easy to implement when disparate parts of our application are tightly coupled.

While 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 essential 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 assume 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 drawback of the pattern is that subscribers are entirely ignorant of the existence of each other and are blind to the cost of switching publishers. Due to the dynamic relationship between subscribers and publishers, it can be

difficult to track an update dependency.

Publish/Subscribe Implementations

Publish/Subscribe fits in very well in JavaScript ecosystems, primarily 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 event systems in implementation code (except for perhaps the DOM3 CustomEvent, which is bound to the DOM and is thus not generically applicable).

Luckily, many modern JavaScript libraries support utilities for implementing a Publish/Subscribe system with little effort. For those wishing to use the Publish/Subscribe pattern with vanilla JavaScript (or another library), PubSubJS is a topic-based publish/subscribe library written in JavaScript. Postal.js is an in-memory message bus that implements channels and topic-based messaging. AmplifyJS includes a clean, library-agnostic implementation that you can be use with any library or toolkit.

To better appreciate how many of the vanilla JavaScript implementations of the Observer pattern might work, let's take a walkthrough of a minimalist version of Publish/Subscribe I released on GitHub under a project called pubsubz. This demonstrates the core concepts of subscribe and publish, and the idea of unsubscribing.

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

A Publish/Subscribe implementation

```
class PubSub {
    constructor() {
        // Storage for topics that can be broadcast
        // or listened to
```

```
        this.topics = {};

        // A topic identifier
        this.subUid = -1;
    }

    publish(topic, args) {
        if (!this.topics[topic]) {
            return false;
        }

        const subscribers = this.topics[topic];
        let len = subscribers ? subscribers.length : 0;

        while (len--) {
            subscribers[len].func(topic, args);
        }

        return this;
    }

    subscribe(topic, func) {
        if (!this.topics[topic]) {
            this.topics[topic] = [];
        }

        const token = (++this.subUid).toString();
        this.topics[topic].push({
            token,
            func,
        });
        return token;
    }

    unsubscribe(token) {
        for (const m in this.topics) {
            if (this.topics[m]) {
                for (let i = 0, j = this.topics[m].length;
i < j; i++) {
                    if (this.topics[m][i].token === token)
{
                        this.topics[m].splice(i, 1);
```

```
                        return token;
                    }
                }
            }
        }
        return this;
    }
}

const pubsub = new PubSub();

pubsub.publish('/addFavorite', ['test']);
pubsub.subscribe('/addFavorite', (topic, args) => {
    console.log('test', topic, args);
});
```

Here we have defined a basic PubSub class that contains * A list of topics with subscribers who have subscribed to it. * The Subscribe method creates a new subscriber to a topic using the function to be called when publishing a topic and a unique token. * The Unsubscribe method removes a subscriber from the list based on the token value passed. The * The Publish method publishes content on a given topic to all its subscribers by calling the registered function.

Using our implementation

We can now use the implementation to publish and subscribe to events of interest as follows (Example 7-1):

Example 7-1. Using our implementation

```
// Another simple message handler

// A simple message logger that logs any topics and data
received through our
// subscriber
const messageLogger = (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
const 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?');
```

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 displaying the stock stats and a counter indicating the last update point. The application must update the grid and counter when the data model changes. 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 are notified that the model 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 the *last updated* counter to log the last time that data was added. (Example 7-2).

Example 7-2. User-interface notifications

```
// Return the current local time to be used in our UI later
getCurrentTime = () => {
    const date = new Date();
    const m = date.getMonth() + 1;
    const d = date.getDate();
    const y = date.getFullYear();
    const t = date.toLocaleTimeString().toLowerCase();

    return `${m}/${d}/${y} ${t}`;
  };

  // Add a new row of data to our fictional grid component
  const 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
  const 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
  const gridUpdate = (topic, data) => {
    if (data !== undefined) {
      addGridRow(data);
      updateCounter(data);
    }
  };
```

```
// Create a subscription to the newDataAvailable topic
const 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,
});
```

Decoupling applications using Ben Alman's Pub/Sub implementation

In the following movie rating example, we'll use 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 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.

Here is the HTML/Templates code (Example 7-3):

Example 7-3. HTML/Templates code for Pub/Sub

```
<script id="userTemplate" type="text/html">
```

```
   <li><%= name %></li>
</script>


<script id="ratingsTemplate" type="text/html">
   <li><strong><%= title %></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>
```

Here is the JavaScript code (Example 7-4):

Example 7-4. JavaScript code for Pub/Sub

```
;($ => {
    // Pre-compile templates and "cache" them using closure
    const userTemplate =
_.template($('#userTemplate').html());

    const 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', (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', (e, data) => {
      if (data) {
        $('#ratings').append(ratingsTemplate(data));
      }
    });

    // Handler for adding a new user
    $('#add').on('click', e => {
      e.preventDefault();

      const strUser = $('#twitter_handle').val();
```

```
      const strMovie = $('#movie_seen').val();
      const 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);
```

Decoupling an Ajax-based jQuery application

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

Often in Ajax-heavy applications, we want to achieve more than just one unique action once we've received a response to a request. We could add all of the 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. Keeping our post-request logic hardcoded in a callback might be okay if we just try to grab a result set once. However, 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 going 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 that can be less elegantly done using other patterns.

Notice how in our sample below, one topic notification is made when a user indicates that he wants to make a search query. 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 using 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 and 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.

Here is the HTML/Templates code (Example 7-5):

Example 7-5. HTML/Templates code for Ajax

```
<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>
```

Here is the JavaScript code (Example 7-6):

Example 7-6. JavaScript code for Ajax

```
($ => {
    // Pre-compile template and "cache" it using closure
    const resultTemplate =
```

```
_.template($('#resultTemplate').html());

    // Subscribe to the new search tags topic
    $.subscribe('/search/tags', (e, tags) => {
      $('#lastQuery').html(`Searched for: ${tags}`);
    });

    // Subscribe to the new results topic
    $.subscribe('/search/resultSet', (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();
      const 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
    // We used the destructuring assignment syntax that makes
it possible to
    // unpack values from data structures into distinct
variables.

    $.subscribe('/search/tags', (e, tags) => {
      $.getJSON(

'http://api.flickr.com/services/feeds/photos_public.gne?
jsoncallback=?',
```

```
        {
          tags,
          tagmode: 'any',
          format: 'json',
        },
        // The destructuring assignment as function parameter
        ({ items }) => {
          if (!items.length) {
            return;
          }
          //shorthand property names in object creation,
          // if variable name equal to object key
          $.publish('/search/resultSet', { items });
        }
      );
    });
  })(jQuery);
```

Observer Pattern in the React ecosystem

A popular library that uses the observable pattern is RxJS.

> *ReactiveX combines the Observer pattern with the Iterator pattern and functional programming with collections to fill the need for an ideal way of managing sequences of events. - RxJS*

With RxJS, we can create observers and subscribe to certain events! Let's look at an example from their documentation, which logs whether a user was dragging in the document.

```
import ReactDOM from "react-dom";
import { fromEvent, merge } from "rxjs";
import { sample, mapTo } from "rxjs/operators";

import "./styles.css";

merge(
  fromEvent(document, "mousedown").pipe(mapTo(false)),
  fromEvent(document, "mousemove").pipe(mapTo(true))
)
  .pipe(sample(fromEvent(document, "mouseup")))
  .subscribe(isDragging => {
```

```
    console.log("Were you dragging?", isDragging);
  });

ReactDOM.render(
  <div className="App">Click or drag anywhere and check the
console!</div>,
  document.getElementById("root")
);
```

The Observer pattern helps decouple several different scenarios in application design. If you haven't been using it, I recommend picking up one of the prewritten implementations mentioned here and giving it a try. It's one of the easier design patterns to get started with but also one of the most powerful.

The Mediator Pattern

The Mediator Pattern is a design pattern that allows one object to notify a set of other objects when an event occurs. The difference between the Mediator and Observer patterns is that the Mediator pattern allows one object to be notified of events that occur in other objects. In contrast, the Observer pattern allows one object to subscribe to multiple events that occur in other objects.

In the section on the Observer pattern, we discussed 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."[2] 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 interactions between components are managed centrally instead of having components refer to each other explicitly. 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) take place from the aircraft to the control tower rather than from plane to plane. A centralized controller is a key to the success of this system, and that's the role a Mediator plays in software design (Figure 7-5).

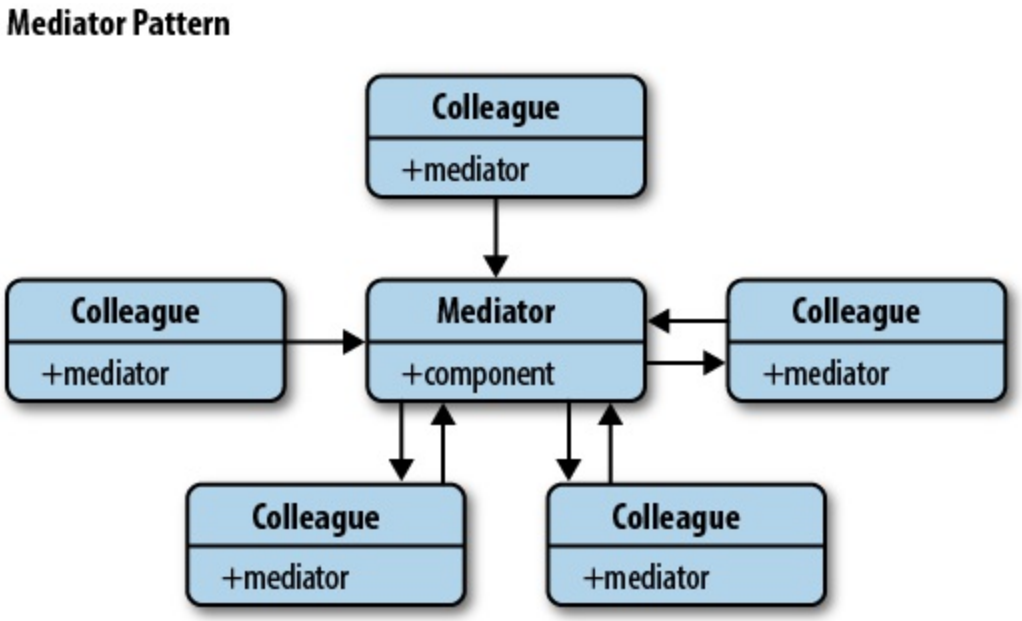


Figure 7-5. Mediator pattern

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, sometimes 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. They should not be

interchanged or confused in communication because of their differences.

A Simple Mediator

A Mediator is an object that coordinates interactions (logic and behavior) between multiple objects. It decides 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:

```
const mediator = {};
```

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

The following example shows a basic implementation of a mediator object with some utility methods that can trigger and subscribe to events. The orgChart object here is a mediator that assigns actions to be taken on the occurrence of a particular event. Here, a manager is assigned to the employee on completing the details of a new employee, and the employee record is saved.

```
const orgChart = {
    addNewEmployee() {
        // getEmployeeDetail provides a view that users
interact with
        const employeeDetail = this.getEmployeeDetail();

        // when the employee detail is complete, the
mediator (the 'orgchart'
        // object) decides what should happen next
        employeeDetail.on('complete', employee => {
            // set up additional objects that have
additional events, which are
            // used by the mediator to do additional things
            const managerSelector =
this.selectManager(employee);
```

```
            managerSelector.on('save', employee => {
                employee.save();
            });
        });
    },

    // ...
};
```

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 a 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 pattern's intent and see that the implementations can be dramatically different, the nature of the patterns becomes 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 web app 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 using several 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

By design, the event aggregator and mediator 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 primarily depends on where the application logic and workflow are 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 mediator's case, the business logic and workflow are aggregated into the mediator itself. The mediator decides when an object should have its methods called and attributes updated based on factors the mediator knows about. It encapsulates the workflow and process, coordinating multiple objects to produce the desired system behavior. The individual objects involved in this workflow know how to perform their task. But the mediator 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 might use events to make decisions, but it is definitely not "fire and forget". A mediator pays attention to a known set of inputs or activities so that it can facilitate and coordinate other behavior with a known set of actors (objects).

Relationships: When To Use Which

Understanding the similarities and differences between an event aggregator and a mediator is essential for semantic reasons. It's equally important to know when to use which pattern, though. The basic semantics and intent of

the patterns inform the question of when, but experience in using the patterns will help you understand the more subtle points and nuanced decisions that must be made.

Event Aggregator Use

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

When two objects already have a direct relationship – for example, a parent view and a child view – there may be a benefit in using an event aggregator. Have the child view trigger an event, and the parent view can handle the event. This is most commonly seen in Backbone's Collection and Model in JavaScript framework terms, 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 an excellent example.

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 the performance of the application and user experience. Instead, using jQuery's on method allows us to aggregate all 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 ubiquitous 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 showing all 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 an excellent example of this, as shown in the “orgChart” the example above. Multiple views 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 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, is best illustrated by showing how they can be used together. The menu example for an event aggregator is the perfect place to introduce a mediator.

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 makes sense. Some of these changes may be internally related, though, and may use a mediator to enact those changes.

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

```
const MenuItem = MyFrameworkView.extend({
    events: {
        'click .thatThing': 'clickedIt',
    },
```

```
    clickedIt(e) {
        e.preventDefault();

        // assume this triggers "menu:click:foo"

MyFramework.trigger(`menu:click:${this.model.get('name')}`)

    },
});

// ... somewhere else in the app

class MyWorkflow {
    constructor() {
        MyFramework.on('menu:click:foo', this.doStuff,
this);
    }

    static doStuff() {
        // 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` class will handle this specific event and coordinate all of the objects it knows about to create the desired user experience and workflow.

We have thus combined an event aggregator and a mediator to create a meaningful experience in both code and application. We now have a clean separation between the menu and the workflow through an event aggregator, and we are still keeping the workflow clean and maintainable through a mediator.

Mediator/Middleware in modern JavaScript

Express.js is a popular web application server framework. We can add callbacks to certain routes that the user can access.

Say we want to add a header to the request if the user hits the root (/). We can add this header in a middleware callback.

```
const app = require("express")();

app.use("/", (req, res, next) => {
  req.headers["test-header"] = 1234;
  next();
});
```

The `next()` method calls the next callback in the request-response cycle. We'd create a chain of middleware functions that sit between the request and the response or vice versa. We can track and modify the request object all the way to the response through one or multiple middleware functions.

The middleware callbacks will be invoked whenever the user hits a root endpoint (/).

```
const app = require("express")();
  const html = require("./data");

  app.use(
    "/",
    (req, res, next) => {
      req.headers["test-header"] = 1234;
      next();
    },
    (req, res, next) => {
      console.log(`Request has test header:
${!!req.headers["test-header"]}`);
      next();
    }
  );

  app.get("/", (req, res) => {
    res.set("Content-Type", "text/html");
    res.send(Buffer.from(html));
```

```
});

app.listen(8080, function() {
  console.log("Server is running on 8080");
});
```

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 both abstract the functionality of existing modules, but there are some subtle differences.

The Mediator centralizes communication between modules where these modules explicitly reference it. In a sense, this is multidirectional. The Facade, however, defines a more straightforward 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 refers to the Prototype pattern as one that 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 is effectively used as a blueprint for each object the constructor creates. For example, if the prototype of the constructor function used contains a property called `name` (as per the code sample that follows), then each object created by that constructor will also have this same property (Figure 7-6).

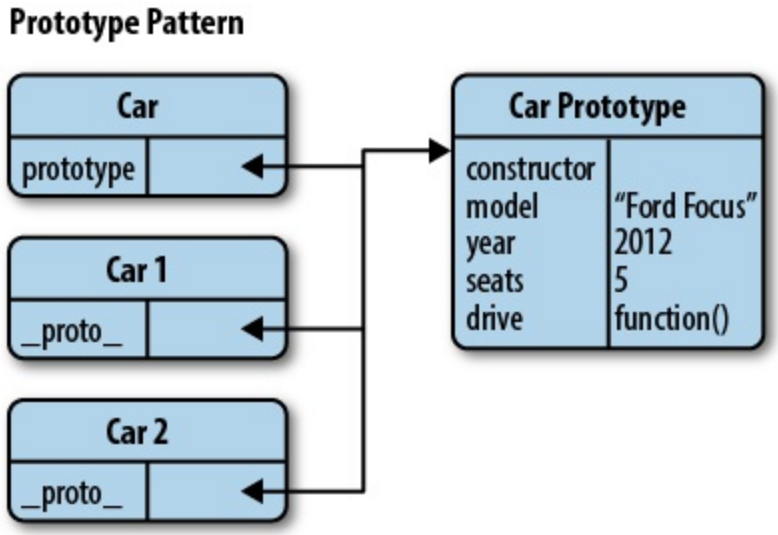


Figure 7-6. Prototype pattern

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. When defining functions in an object, they're all created by reference (so all child objects point to the same functions), instead of creating individual copies.

With ES2015+, we can use classes and constructors to create objects. While this ensures that our code looks cleaner and follows OOAD principles, the classes and constructors get compiled down to functions and prototypes internally. This ensures that we are still working with the prototypal strengths of JavaScript and the accompanying performance boost.

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 review, `Object.create` creates an object with a specified prototype and optionally contains specified properties (e.g., `Object.create( prototype, optionalDescriptorObjects )`).

We can see this demonstrated in the following example:

```
const myCar = {
    name: 'Ford Escort',

    drive() {
        console.log("Weeee. I'm driving!");
    },

    panic() {
        console.log('Wait. How do you stop this thing?');
    },
};

// Use Object.create to instantiate a new car
const 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 initialize object properties using the second supplied argument. For example:

```
const vehicle = {
    getModel() {
        console.log(`The model of this vehicle
is..${this.model}`);
    },
};

const car = Object.create(vehicle, {
    id: {
        value: MY_GLOBAL.nextId(),
```

```
        // writable:false, configurable:false by default
        enumerable: true,
    },

    model: {
        value: 'Ford',
        enumerable: true,
    },
});
```

Here, you can initialize the properties 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:

```
class VehiclePrototype {
    constructor(model) {
        this.model = model;
    }

    getModel() {
        console.log('The model of this vehicle is..' +
this.model);
    }

    Clone() {}
}

class Vehicle extends VehiclePrototype {
    constructor(model) {
        super(model);
    }
    Clone() {
```

```
            return new Vehicle(this.model);
        }
    }

    const car = new Vehicle('Ford Escort');
    const car2 = car.Clone();
    car2.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:

```
const beget = (() => {
    class F {
        constructor() {}
    }

    return proto => {
        F.prototype = proto;
        return new F();
    };
})();
```

One could reference this method from the `vehicle` function. However, note that `vehicle` here emulates 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 allows us to both parameterize and pass

method calls that can be executed at our discretion. In addition, it enables us to decouple objects invoking the action from the objects that implement them, giving us greater 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 its member functions. It acts as a base class from which others are derived. A derived class that implements the missing functionality is called a *concrete* class (Figure 7-7). Base and concrete classes can be implemented in JavaScript(ES2015+) using the extends keyword applicable to the JavaScript classes.

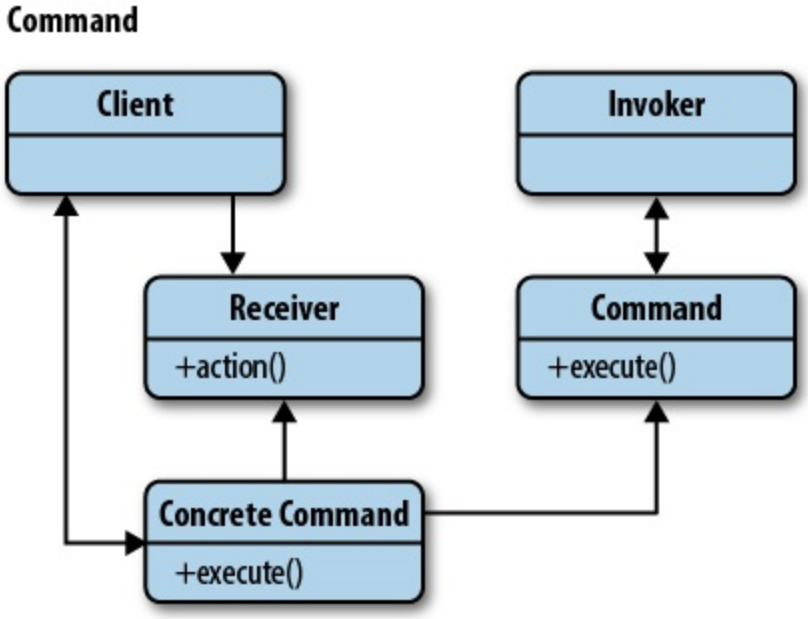


Figure 7-7. Command pattern

The general idea behind the Command pattern is that it provides 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 the 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, which is one of the vital

benefits of the pattern.

To demonstrate the Command pattern we will create a simple car purchasing service.

```
const CarManager = {
        // request information
        requestInfo(model, id) {
            return `The information for ${model} with ID
${id} is foobar`;
        },

        // purchase the car
        buyVehicle(model, id) {
            return `You have successfully purchased Item
${id}, a ${model}`;
        },

        // arrange a viewing
        arrangeViewing(model, id) {
            return `You have booked a viewing of ${model} (
${id} ) `;
        },
    };
```

The `CarManager` object is our command object responsible for issuing commands to request information about a car, buy a car, and arrange a viewing. It would be trivial to invoke our `CarManager` methods by directly accessing the object. It would be forgivable to assume nothing is 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 be modified. It is a type of coupling that 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 the `CarManager` so that our Command pattern

application 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:

```
CarManager.execute('buyVehicle', 'Ford Escort', '453543');
```

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

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

Our final sample calls would thus look as follows:

```
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 inspired the name for the next pattern we'll 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, a quality that almost always improves usability (Figure 7-8).

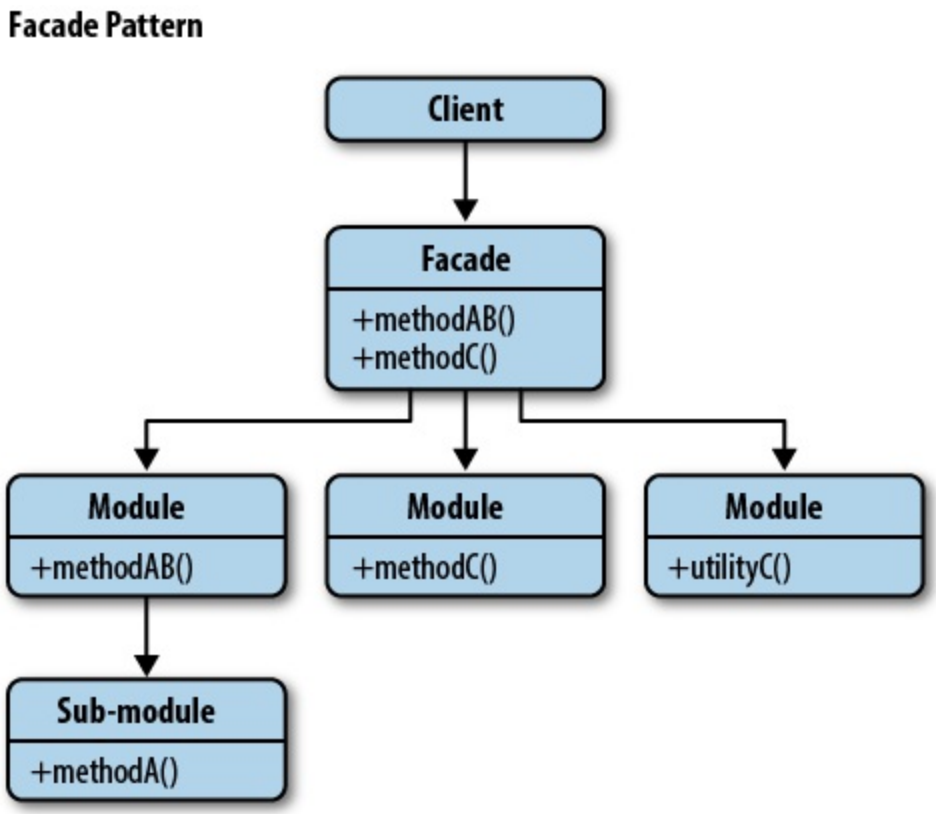


Figure 7-8. Facade pattern

Facades are a structural pattern that can often be seen in JavaScript libraries such as 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 using a Facade: the simpler public interface that lets us avoid manually calling the many internal methods in jQuery core required to get some behavior working. This also circumvents the need to interact manually with DOM APIs and maintain state variables.

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

To build on what we've learned, the Facade pattern simplifies a class's interface and decouples the class from the code that utilizes it. This allows us

to interact indirectly with subsystems in a way that can sometimes be less error-prone than accessing the subsystem directly. A Facade's advantages include ease of use and often a small-sized 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 using a Facade to simplify an interface for listening to events across browsers. We do this by creating a common method that does the task of checking for the existence of features so that it can provide a safe and cross-browser-compatible solution.

```
const addMyEvent = (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 powered by a method called `bindReady()`, which is doing this:

```
function bindReady() {
    // ...
    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 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. You can also integrate them with other patterns, such as the Module pattern. As we can see below, our instance of the Module pattern contains a number of methods that have been privately defined. A Facade is then used to supply a much simpler API for accessing these methods:

```
const _private = {
    i: 5,
    get() {
      console.log(`current value:${this.i}`);
    },
    set(val) {
      this.i = val;
    },
    run() {
      console.log('running');
    },
    jump() {
      console.log('jumping');
    },
  };

  // We used the destructuring assignment syntax that makes
it possible to
  // unpack values from data structures into distinct
variables.
```

```
const module = {
  facade({ val, run }) {
    _private.set(val);
    _private.get();
    if (run) {
      _private.run();
    }
  },
};

export default module;

import module from './module.js';
// Outputs: "current value: 10" and "running"
module.facade({
  run: true,
  val: 10,
});
```

In this example, calling `module.facade()` will trigger a set of private behavior within the module, but the users aren't concerned with this. We've made it much easier for them to consume a feature without worrying about implementation-level details.

Notes on Abstraction

Facades generally have a 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 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*. 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 in the past, a simple Facade worked out very well for the team.

When using the Facade pattern, try to be aware of any performance costs involved and decide whether they are worth the level of abstraction offered.

The Factory Pattern

The Factory pattern is another creational pattern for creating objects. It differs from the other patterns in its category because it doesn't explicitly require us to use a constructor. Instead, a Factory can provide a generic interface for creating objects, where we can specify the type of factory object we want to create (Figure 7-9).

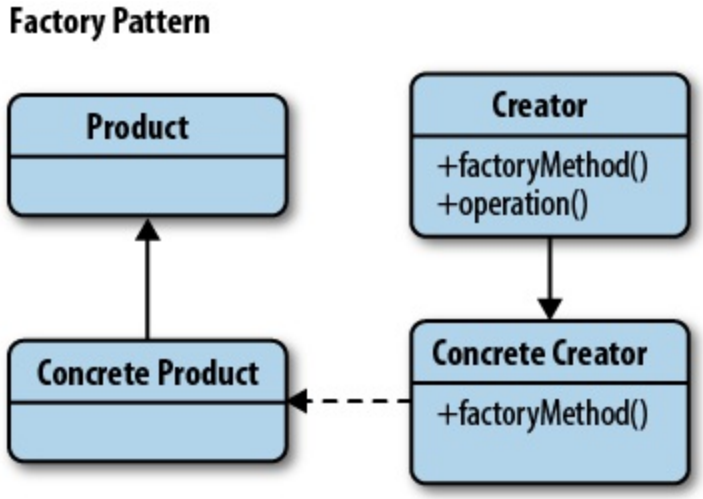


Figure 7-9. Factory pattern

Imagine a UI factory where we want to create a type of UI component. Rather than creating this component directly using the `new` operator or

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 it and returns 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.

You can find examples of this pattern in UI libraries such as ExtJS, where the methods for creating objects or components may be further subclassed.

The following example builds upon our previous snippets using the Constructor pattern logic to define cars. It demonstrates how a `VehicleFactory` may be implemented using the Factory pattern:

```
// Types.js - Constructors used behind the scenes

// A constructor for defining new cars
class Car {
    constructor({
        doors,
        state,
        color
    }) {
        // some defaults
        this.doors = doors || 4;
        this.state = state || 'brand new';
        this.color = color || 'silver';
    }
}
// A constructor for defining new trucks
class Truck {
    constructor({
        state,
        wheelSize,
        color
    }) {
        this.state = state || 'used';
        this.wheelSize = wheelSize || 'large';
        this.color = color || 'blue';
    }
}
```

```
// FactoryExample.js

// Define a vehicle factory
class VehicleFactory {
    // Define the prototypes and utilities for this factory

    // Our default vehicleClass is Car
    constructor() {
        this.vehicleClass = Car;
    }
    // Our Factory method for creating new Vehicle
instances
    createVehicle(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
const carFactory = new VehicleFactory();
const 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);
```

We have defined the car and truck classes with constructors that set properties relevant to the respective vehicle. The `VehicleFactory` can create a new vehicle object, `Car`, or `Truck` based on the vehicleType passed.

There are two possible approaches to building trucks using the VehicleFactory class.

In Approach 1, we modify a `VehicleFactory` instance to use the `Truck` class:

```
const 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);
```

In Approach 2, we subclass `VehicleFactory` to create a factory class that builds `Trucks`:

```
class TruckFactory extends VehicleFactory {
    constructor() {
        super();
        this.vehicleClass = Truck;
    }
```

```
}
const truckFactory = new TruckFactory();
const 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 beneficial when applied to the following situations:

- When our object or component setup involves a high level of complexity
- When we need a convenient way to 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 a large amount of unnecessary 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 undue overhead.

Since 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 worthwhile 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 implementing a set of objects from their general usage.

You can use an Abstract Factory when a system must be independent of how the objects it creates are generated, or it needs to work with multiple types of objects.

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

```
class AbstractVehicleFactory {
    constructor() {
        // Storage for our vehicle types
        this.types = {};
    }

    static getVehicle(type, customizations) {
        const Vehicle = this.types[type];

        return Vehicle ? new Vehicle(customizations) :
null;
    }
```

```
    static registerVehicle(type, Vehicle) {
        const proto = Vehicle.prototype;

        // only register classes that fulfill the vehicle
contract
        if (proto.drive && proto.breakDown) {
            this.types[type] = Vehicle;
        }

        return abstractVehicleFactory;
    }
}

// Usage:

abstractVehicleFactory.registerVehicle('car', Car);
abstractVehicleFactory.registerVehicle('truck', Truck);

// Instantiate a new car based on the abstract vehicle type
const car = abstractVehicleFactory.getVehicle('car', {
    color: 'lime green',
    state: 'like new',
});

// Instantiate a new truck in a similar manner
const truck = abstractVehicleFactory.getVehicle('truck', {
    wheelSize: 'medium',
    color: 'neon yellow',
});
```

The Mixin Pattern

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

Subclassing

We have already introduced the ES2015+ features that allow us to extend a

base or superclass and call the methods in the superclass. The child class that extends the superclass is known as a sub-class.

Sub-classing refers to inheriting properties for a new object from a base or superclass object. A sub-class can still define its methods, including those that override methods initially defined in the superclass. The method in the sub-class can invoke an overridden method in the superclass, known as method chaining. Similarly, it can invoke the superclass's constructor, which is known as constructor chaining.

To demonstrate sub-classing, we first need a base class that can have new instances of itself created. Let's model this around the concept of a person.

```
class Person{
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.gender = "male";
    }
}
// a new instance of Person can then easily be created as
follows:
const clark = new Person( 'Clark', 'Kent' );
```

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

```
class Superhero extends Person {
    constructor(firstName, lastName, powers) {
        // Invoke the superclass constructor
        super(firstName, lastName);
        this.powers = powers;
    }
}
```

```
// A new instance of Superhero can be created as follows

const SuperMan = new Superhero('Clark','Kent',
['flight','heat-vision']);
console.log(SuperMan);

// Outputs Person attributes as well as power
```

The `Superhero` constructor creates an instance of the Superhero class, which is an extension of the `Person` class. Objects of this type have attributes of the classes above it in the chain. If we had set default values in the `Person` class, `Superhero` could override any inherited values with values specific to its class.

Mixins

In JavaScript, we can look at inheriting from Mixins to collect functionality through extension. Each new class we define can have a superclass from which it can inherit methods and properties. Classes can also define their own properties and methods. We can leverage this fact to promote function reuse (Figure 7-10).

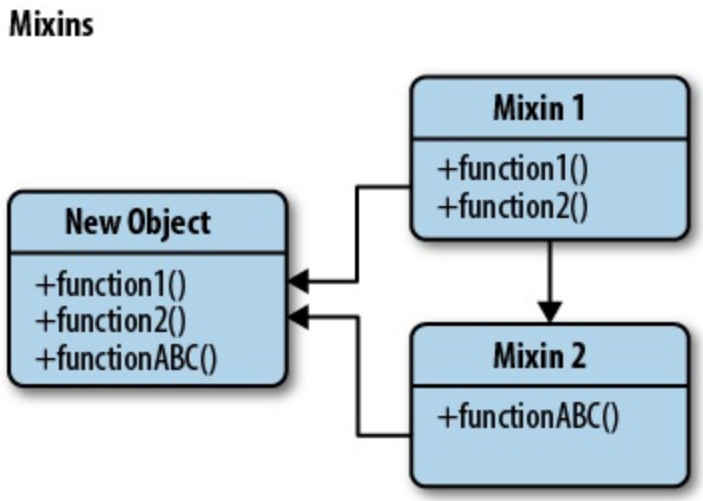


Figure 7-10. Mixins

Mixins allow objects to borrow (or inherit) functionality from them with minimal complexity. Thus, Mixins are classes with attributes and methods

that can be easily shared across several other classes.

While JavaScript classes cannot inherit from multiple superclasses, we can still mix functionality from various classes. A class in JavaScript can be used as an expression as well as a statement. As an expression, it returns a new class each time it's evaluated. The extends clause can also accept arbitrary expressions that return classes or constructors. These features enable us to define a mixin as a function that accepts a superclass and creates a new subclass from it.

Imagine that we define a Mixin containing utility functions in a standard JavaScript class as follows:

```
const MyMixins = superclass =>
    class extends superclass {
        moveUp() {
            console.log('move up');
        }
        moveDown() {
            console.log('move down');
        }
        stop() {
            console.log('stop! in the name of love!');
        }
    };
```

Above, we created a MyMixins function that can extend a dynamic superclass. We will now create two classes, CarAnimator and PersonAnimator, from which MyMixins can extend and return a sub-class with methods defined in MyMixins and those in the class being extended.

```
// A skeleton carAnimator constructor
class CarAnimator {
    moveLeft() {
        console.log('move left');
    }
}
// A skeleton personAnimator constructor
class PersonAnimator {
```

```
    moveRandomly() {
        /*..*/
    }
}

// Extend MyMixins using CarAnimator
class MyAnimator extends MyMixins(CarAnimator) {}

// Create a new instance of carAnimator
const myAnimator = new MyAnimator();
myAnimator.moveLeft();
myAnimator.moveDown();
myAnimator.stop();

// Outputs:
// move left
// move down
// stop! in the name of love!
```

As we can see, this makes mixing similar behaviour into classes reasonably trivial.

The following example has two classes: 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 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.

```
// Define a simple Car constructor
class Car {
    constructor({ model, color }) {
        this.model = model || 'no model provided';
        this.color = color || 'no colour provided';
    }
}

// Mixin
const Mixin = superclass =>
```

```
        class extends superclass {
            driveForward() {
                console.log('drive forward');
            }
            driveBackward() {
                console.log('drive backward');
            }
            driveSideways() {
                console.log('drive sideways');
            }
        };

    class MyCar extends Mixin(Car) {}

    // Create a new Car
    const myCar = new MyCar({
        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

    const mySportCar = new MyCar({
        model: 'Porsche',
        color: 'red',
    });

    mySportsCar.driveSideways();

    // Outputs:
    // drive sideways
```

Advantages and Disadvantages

Mixins assist in decreasing functional repetition and increasing function reuse in a system. Where an application is likely to require shared behavior across

object instances, we can easily avoid 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 a class or 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.

Even with React, Mixins were often used to add functionality to components before the introduction of ES6 classes. The React team discourages mixins as it adds unnecessary complexity to a component, making it hard to maintain and reuse. The React team encouraged using higher order components and Hooks instead.

I would argue that solid documentation can assist in minimizing the amount of confusion regarding the source of mixed-in functions. Still, as with every pattern, we should be okay if we take care during implementation.

The Decorator Pattern

Decorators are a structural design pattern that aims to promote code reuse. Like Mixins, you can think of them as another viable alternative to object sub-classing.

Classically, Decorators offered the ability to add behavior 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, we could bake it into the *superclass* itself.

We can use them to modify existing systems where we wish to add additional features to objects without heavily changing the underlying code that uses them. A common reason developers use them is that their applications may contain features requiring many distinct types of objects. Imagine defining hundreds of different object constructors for, say, a JavaScript game (Figure 7-11).

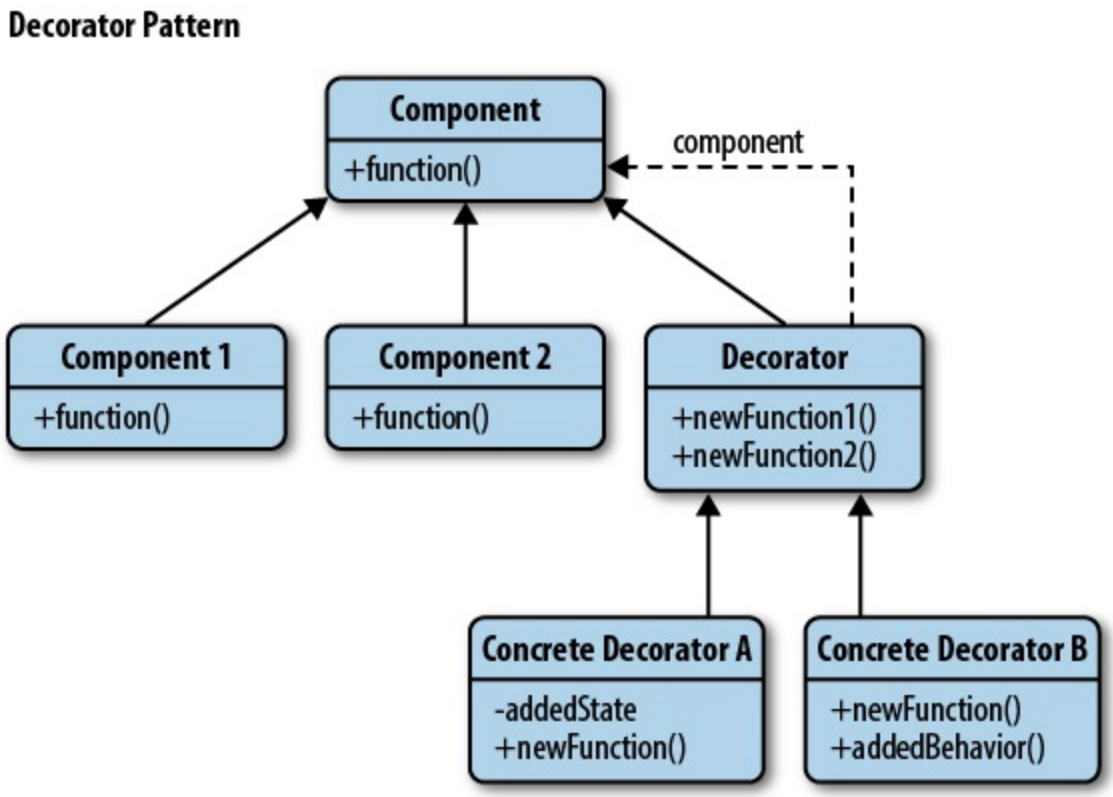


Figure 7-11. Decorator pattern

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 types, e.g., `HobbitWithRing`, `HobbitWithSword`, `HobbitWithRingAndSword`, and so on. This isn't very practical and certainly isn't manageable when we factor in an increasing 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 class and progressively add decorator objects which provide 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.

We can use JavaScript classes to create the base classes that can be decorated. Adding new attributes or methods to object instances of the class in JavaScript is a very straightforward process. With this in mind, we can implement a very simplistic decorator as follows (Examples 7-7 and 7-8).

Example 7-7. Decorating Constructors with New Functionality

```
// A vehicle constructor
class Vehicle {
    constructor(vehicleType) {
        // some sane defaults
        this.vehicleType = vehicleType || 'car';
        this.model = 'default';
        this.license = '00000-000';
    }
}

// Test instance for a basic vehicle
const 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
const 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
const secondInstance = new Vehicle('car');
console.log(secondInstance);

// Outputs:
// vehicle: car, model:default, license: 00000-000
```

Here, `truck` is an instance of the class `Vehicle`, and we also decorate it with additional methods `setColor` and `setModel`.

This type of simplistic implementation is functional, but it doesn't demonstrate all the strengths that Decorators offer. For this, we're 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 7-8. Decorating Objects with Multiple Decorators

```
// The constructor to decorate
class MacBook {
    constructor() {
        this.cost = 997;
        this.screenSize = 11.6;
    }
    getCost() {
        return this.cost;
    }
    getScreenSize() {
        return this.screenSize;
    }
}

// Decorator 1
class Memory extends MacBook {
    constructor(macBook) {
        super();
        this.macBook = macBook;
    }

    getCost() {
```

```
        return this.macBook.getCost() + 75;
    }
}

// Decorator 2
class Engraving extends MacBook {
    constructor(macBook) {
        super();
        this.macBook = macBook;
    }

    getCost() {
        return this.macBook.getCost() + 200;
    }
}

// Decorator 3
class Insurance extends MacBook {
    constructor(macBook) {
        super();
        this.macBook = macBook;
    }

    getCost() {
        return this.macBook.getCost() + 250;
    }
}

// init main object
let mb = new MacBook();

// init decorators
mb = new Memory(mb);
mb = new Engraving(mb);
mb = new Insurance(mb);

// Outputs: 1522
console.log(mb.getCost());

// Outputs: 11.6
console.log(mb.getScreenSize());
```

In the above example, our Decorators are overriding the `MacBook`

superclass objects `.cost()` function to return the current price of the Macbook plus the cost of the upgrade.

It's considered 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 the `Macbook`, remain unchanged and intact.

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

Pseudoclassical 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 you find it overly complex, I recommend opting for one of the straightforward implementations covered earlier.

Interfaces

PJDP describes the Decorator pattern as one 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 directly specify how you should implement those methods. Interfaces can also

optionally indicate what parameters the methods take.

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

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

```
// 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
const reminder = new Interface('List', ['summary',
'placeOrder']);

const properties = {
    name: 'Remember to buy the milk',
    date: '05/06/2016',
    actions: {
        summary() {
            return 'Remember to buy the milk, we are almost
out!';
        },
        placeOrder() {
            return 'Ordering milk from your local grocery
store';
        },
    },
};

// Now create a constructor implementing the above
properties
// and methods

class Todo {
```

```
    constructor({ actions, name }) {
        // State the methods we expect to be supported
        // as well as the Interface instance being checked
        // against

        Interface.ensureImplements(actions, reminder);

        this.name = name;
        this.methods = actions;
    }
}

// Create a new instance of our Todo constructor

const 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
```

Both classic JavaScript, as well as ES2015+, do not support interfaces. However, we can create our Interface class. In the above, `Interface.ensureImplements` provides strict functionality checking, and you can find code for both this and the `Interface` constructor here.

The biggest problem with interfaces is that since there isn't built-in support for them in JavaScript, there is a risk that we will attempt to emulate a feature of another language that may not be an ideal fit. You can use lightweight interfaces without a significant 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 4 GB or 8 GB of RAM, engraving, Parallels, or a case. Now, if we were to model this using an individual subclass for each combination of enhancement options, it might look something like this:

```
const Macbook = class {
    //...
};

const MacbookWith4GBRam = class {};
const MacbookWith8GBRam = class {};
const MacbookWith4GBRamAndEngraving = class {};
const MacbookWith8GBRamAndEngraving = class {};
const MacbookWith8GBRamAndParallels = class {};
const MacbookWith4GBRamAndParallels = class {};
const MacbookWith8GBRamAndParallelsAndCase = class {};
const MacbookWith4GBRamAndParallelsAndCase = class {};
const MacbookWith8GBRamAndParallelsAndCaseAndInsurance =
class {};
const MacbookWith4GBRamAndParallelsAndCaseAndInsurance =
class {};
```

… and so on.

The above solution would be impractical 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 we can use decorators to solve this problem better.

Rather than requiring all of the combinations we saw earlier, we will only create five new decorator classes. Methods called on these enhancement classes would be passed on to our `Macbook` class.

In the following example, decorators transparently wrap around their components and can be interchanged as they use the same interface. Here’s the interface we’re going to define for the Macbook:

```
const Macbook = new Interface('Macbook', [
    'addEngraving',
    'addParallels',
    'add4GBRam',
    'add8GBRam',
    'addCase',
]);

// A Macbook Pro might thus be represented as follows:
class MacbookPro {
    // implements Macbook
}

// ES2015+: We still could use Object.prototype for adding
new methods,
// because internally we use the same structure

MacbookPro.prototype = {
    addEngraving() {},
    addParallels() {},
    add4GBRam() {},
    add8GBRam() {},
    addCase() {},
    getPrice() {
        // Base price
        return 900.0;
    },
};
```

To make it easier for us to add 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 subclass. 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.

```
// Macbook decorator abstract decorator class

class MacbookDecorator {
```

```
    constructor(macbook) {
        Interface.ensureImplements(macbook, Macbook);
        this.macbook = macbook;
    }

    addEngraving() {
        return this.macbook.addEngraving();
    }

    addParallels() {
        return this.macbook.addParallels();
    }

    add4GBRam() {
        return this.macbook.add4GBRam();
    }

    add8GBRam() {
        return this.macbook.add8GBRam();
    }

    addCase() {
        return this.macbook.addCase();
    }

    getPrice() {
        return this.macbook.getPrice();
    }
}
```

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

```
// Let's now extend (decorate) the CaseDecorator
// with a MacbookDecorator

class CaseDecorator extends MacbookDecorator {
    constructor(macbook) {
```

```
        super(macbook);
    }

    addCase() {
        return `${this.macbook.addCase()}Adding case to
macbook`;
    }

    getPrice() {
        return this.macbook.getPrice() + 45.0;
    }
}
```

We are overriding the `addCase()` and `getPrice()` methods that we want to decorate, and we'r 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.

```
// Instantiation of the macbook
const myMacbookPro = new MacbookPro();

// Outputs: 900.00
console.log(myMacbookPro.getPrice());

// Decorate the macbook
const decoratedMacbookPro = new
CaseDecorator(myMacbookPro);

// This will return 945.00
console.log(decoratedMacbookPro.getPrice());
```

As decorators can 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 subclasses for each object type. This makes maintaining applications that may require many

subclassed objects significantly more straightforward.

You can find a functional version of this example on JSBin.

Advantages and Disadvantages

Developers enjoy using this pattern as it can be used transparently and is somewhat flexible. As we've seen, objects can be wrapped or "decorated" with new behavior and continue to be used without worrying 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. It can significantly complicate our application architecture if poorly managed, as it introduces many small but similar objects into our namespace. The concern is that other developers unfamiliar with the pattern may have difficulty grasping why it's being used, making it hard to manage.

Sufficient commenting or pattern research should assist with the latter. However, as long as we handle how widely 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—see Figure 7-12).

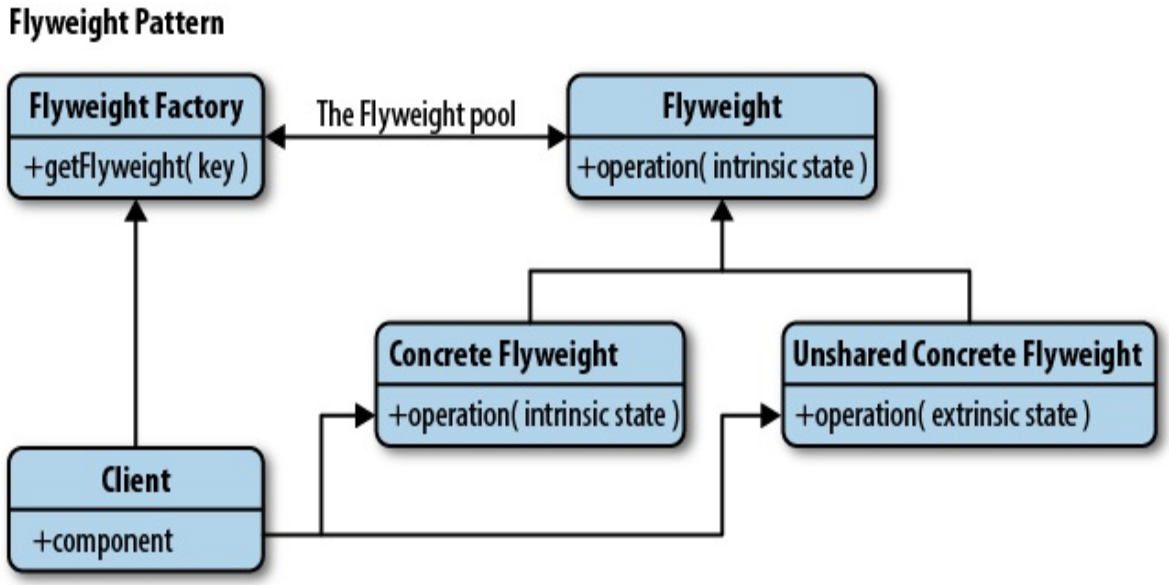


Figure 7-12. Flyweight pattern

Paul Calder and Mark Linton first conceived the pattern in 1990 and named it after the boxing weight class that includes fighters weighing less than 112lb. The name Flyweight 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 many objects and placing this data into a single external object. We can pass this object to those depending on this data rather than storing identical data across each one.

Using Flyweights

There are two ways in which you can apply the Flyweight pattern. 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.

You can also apply the Flyweight at the DOM layer as a central event manager to avoid attaching event handlers to every child element in a parent container with similar behavior.

Traditionally the flyweight pattern has been used most at the data layer, so we'll take a look at this first.

Flyweights and Sharing Data

For this application, we need to be aware of a few more concepts around the classical Flyweight pattern. 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, without which they absolutely cannot function. Extrinsic information can, however, be removed and stored externally.

You can replace objects with the same intrinsic data 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 is that we can 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. You can implement this in various ways, but one approach is to have the manager object contain a central database of the extrinsic states and the flyweight objects to which they belong.

Implementing Classical Flyweights

As the Flyweight pattern hasn't been heavily used in JavaScript recently, 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 the intrinsic states. Concrete Flyweights need to be sharable and capable of manipulating the extrinsic state.

Flyweight factory

> Manages flyweight objects and creates them too. It ensures that our flyweights are shared and manages them as a group of objects that can be queried if we require individual instances. If an object has already been 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 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).

```
// 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 explicitly having a function inherit an interface. Below, `CoffeeFlavor` implements the `CoffeeOrder` interface and must contain its interface methods for us to assign the functionality powering these implementations to an object.

```
// Flyweight object
const CoffeeOrder = {
    // Interfaces
    serveCoffee(context) {},
    getFlavor() {},
};

// ConcreteFlyweight object that creates ConcreteFlyweight
// Implements CoffeeOrder
function CoffeeFlavor(newFlavor) {
    const flavor = newFlavor;

    // If an interface has been defined for a feature
    // implement the feature
    if (typeof this.getFlavor === 'function') {
        this.getFlavor = () => flavor;
    }
```

```
    if (typeof this.serveCoffee === 'function') {
        this.serveCoffee = context => {
            console.log(
                `Serving Coffee flavor ${flavor} to table
${context.getTable()}`
            );
        };
    }
}

// Implement interface for CoffeeOrder
CoffeeFlavor.implementsFor(CoffeeOrder);

// Handle table numbers for a coffee order
const CoffeeOrderContext = tableNumber => ({
    getTable() {
        return tableNumber;
    }
});

const CoffeeFlavorFactory = () => {
    const flavors = {};
    let length = 0;

    return {
        getCoffeeFlavor(flavorName) {
            let flavor = flavors[flavorName];
            if (typeof flavor === 'undefined') {
                flavor = new CoffeeFlavor(flavorName);
                flavors[flavorName] = flavor;
                length++;
            }
            return flavor;
        },

        getTotalCoffeeFlavorsMade() {
            return length;
        },
    };
};

// Sample usage:
// testFlyweight()
```

```
const testFlyweight = () => {
    // The flavors ordered.
    const flavors = [];

    // The tables for the orders.
    const tables = [];

    // Number of orders made
    let ordersMade = 0;

    // The CoffeeFlavorFactory instance
    const flavorFactory = new CoffeeFlavorFactory();

    function takeOrders(flavorIn, table) {

flavors.push(flavorFactory.getCoffeeFlavor(flavorIn));
        tables.push(new CoffeeOrderContext(table));
        ordersMade++;
    }

    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 (let 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 books in a library. You could list the essential meta-data for each book could probably as follows:

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

We’ll also require the following properties to track which member has checked out a particular book, the date they’ve checked it out, and the expected return date.

- `checkoutDate`
- `checkoutMember`
- `dueReturnDate`
- `availability`

We create a `Book` class to represent each book as follows before any optimization using the Flyweight pattern. The constructor takes in all the properties related directly to the book and those required for tracking it.

```
class Book {
  constructor(
    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;
  }

  getTitle() {
    return this.title;
  }

  getAuthor() {
    return this.author;
  }

  getISBN() {
    return this.ISBN;
  }

  // For brevity, other getters are not shown
  updateCheckoutStatus(
    bookID,
    newStatus,
```

```
      checkoutDate,
      checkoutMember,
      newReturnDate
    ) {
      this.id = bookID;
      this.availability = newStatus;
      this.checkoutDate = checkoutDate;
      this.checkoutMember = checkoutMember;
      this.dueReturnDate = newReturnDate;
    }

    extendCheckoutPeriod(bookID, newReturnDate) {
      this.id = bookID;
      this.dueReturnDate = newReturnDate;
    }

    isPastDue(bookID) {
      const 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 more extensive inventory with multiple versions and copies of each book, 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 : data relevant to the book object (`title`, `author`, etc.) is intrinsic, while 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 number of objects, but significantly fewer than we had previously.

An instance of the following book meta-data combination will be created for all required copies of the book object with a particular title/ISBN.

```
// Flyweight optimized version
class Book {
  constructor(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, we can use a factory for instantiation.

A Basic Factory

Let's now define a very basic factory. We will have it check 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:

```
// Book Factory singleton
const existingBooks = {};

class BookFactory {
    constructor(title, author, genre, pageCount,
publisherID, ISBN) {

        // Find if a particular book + metadata combination
already exists
        // !! or (bang bang) forces a boolean to be
returned
        this.existingBook = existingBooks[ISBN];
        if (!!this.existingBook) {
            return this.existingBook;
        } else {
```

```
            // if not, let's create a new instance of the
book and store it
            const 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 who's checked it out will be called Book record. Our manager will be storing both and will include checkout-related logic we stripped out during our flyweight optimization of the `Book` class.

```
// BookRecordManager singleton
const bookRecordDatabase = {};

class BookRecordManager {
    // add a new book into the library system
    constructor(id, title, author, genre, pageCount,
publisherID, ISBN,
            checkoutDate, checkoutMember, dueReturnDate,
availability) {

        this.book = new BookFactory(title, author, genre,
pageCount,
                    publisherID, ISBN);

        bookRecordDatabase[id] = {
            checkoutMember,
            checkoutDate,
            dueReturnDate,
            availability,
```

```
            book: this.book,
        };
    }

    updateCheckoutStatus(bookID, newStatus, checkoutDate,
                         checkoutMember, newReturnDate)
{

        const record = bookRecordDatabase[bookID];
        record.availability = newStatus;
        record.checkoutDate = checkoutDate;
        record.checkoutMember = checkoutMember;
        record.dueReturnDate = newReturnDate;
    }

    extendCheckoutPeriod(bookID, newReturnDate) {
        bookRecordDatabase[bookID].dueReturnDate =
newReturnDate;
    }

    isPastDue(bookID) {
        const currentDate = new Date();
        return currentDate.getTime() >

Date.parse(bookRecordDatabase[bookID].dueReturnDate);
    }
}
```

The result of these changes is that all of the data 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 extrinsic rather than intrinsic data.

This process does add a little complexity to our final solution. However, it's a minor concern compared to the performance issues that we have 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 unoptimized version flyweight mentioned above, we store just a link to the function object as we used the Book constructor's prototype. Still, if we implemented it another 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.

Gary Chisholm wrote one of the best metaphors for describing Flyweights in this context, 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 in which 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 further tweak the event bubbling process, as we will see shortly (Example 7-9).

Example: Centralized event handling

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

Usually, when constructing our accordion component, menu, or other list-based widgets, we 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 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 clicked will be pretty 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, while jQuery is used to bind the initial click to a container `div`. An `unbind` event is first applied to ensure that no other logic on the page attaches similar handles to the container.

To establish exactly what child element in the container is clicked, we use 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.

Example 7-9. Centralized event handling

```
<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="MAPS_URL"></iframe>
       </span>
   </div>
```

```
</div>
const stateManager = {
    fly() {
        const self = this;

        $('#container')
            .unbind()
            .on('click', 'div.toggle', ({
                target
            }) => {
                self.handleClick(target);
            });
    },

    handleClick(elem) {
        $(elem)
            .find('span')
            .toggle('slow');
    },
};
```

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

Summary

With that, we can conclude our discussion of traditional design patterns you can use when designing classes, objects, and modules. I have tried incorporating an ideal mix of creational, structural, and behavioral patterns. We have also studied patterns created for classic object-oriented programming languages such as Java and C++ and adapted them for JavaScript.

These patterns will help us design many domain-specific objects (e.g., shopping cart, vehicle, or book) that make up our applications' business model. In the next chapter, we will look at the larger picture of how we can structure applications so that this model delivers to the other application layers, such as the view or the presenter.

1 IIFE. See [Link to Come] for more on this.

2 Wikipedia; Dictionary.com

About the Author

Addy Osmani is an Engineering Leader working on Google Chrome. He leads Chrome's Developer Experience teams, helping to keep the web fast and delightful to build. Addy has authored several open-source projects as well as a number of books including *Learning Patterns*, *Learning JavaScript Design Patterns*, and *Image Optimization*. His personal blog is *addyosmani.com*.