Community Experience Distilled

Mastering JavaScript Design Patterns

Discover how to use JavaScript design patterns to create powerful applications with reliable and maintainable code

Simon Timms

[PACKT] PUBLISHING open source* community experience distilled

Mastering JavaScript Design Patterns

Discover how to use JavaScript design patterns to create powerful applications with reliable and maintainable code

Simon Timms

BIRMINGHAM - MUMBAI

Mastering JavaScript Design Patterns

First published: November 2014

Production reference: 1151114

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-798-6

www.packtpub.com

Credits

Author
Simon Timms

Reviewers
Amrita Chaturvedi
Philippe Renevier Gonin
Pedro Miguel Barros Morgado
Mani Nilchiani

Commissioning Editor
Kunal Parikh

Acquisition Editor
Meeta Rajani

Content Development Editor
Sweny M. Sukumaran

Technical Editor
Siddhi Rane

Copy Editor
Laxmi Subramanian

Project Coordinator
Rashi Khivansara

Proofreaders
Simran Bhogal
Lawrence A. Herman
Elinor Perry-Smith

Indexer
Hemangini Bari

Graphics
Sheetal Aute
Ronak Dhruv
Valentina D'silva
Disha Haria
Abhinash Sahu

Production Coordinator
Nitesh Thakur

Cover Work
Nitesh Thakur

About the Author

Simon Timms is a developer who loves to write code. He writes in a variety of languages using a number of tools. For the most part, he develops web applications with .NET backends. He is very interested in visualizations and cloud computing. A background in build and system administration keeps him on the straight and narrow when it comes to DevOps.

He is the author of *Social Data Visualization with HTML5 and JavaScript*, *Packt Publishing*. He blogs at `http://blog.simontimms.com/` and is also a frequent contributor to the Canadian Developer Connection, where his latest series explores evolving ASP.NET applications.

He is the President of the Calgary .NET User Group and a member of half a dozen others. He speaks on a variety of topics from DevOps to how the telephone system works.

He works as a web developer for Pacesetter Directional Drilling, the friendliest performance drilling company around.

I would like to thank my wonderful wife for all her support and my children who provided a welcome distraction from writing. I would also like to thank the Prime team at Pacesetter for their sense of humor and for putting up with me.

About the Reviewers

Amrita Chaturvedi is a PhD researcher in the Department of Computer Science and Engineering at Indian Institute of Technology, Kanpur, Uttar Pradesh, India (`http://www.cse.iitk.ac.in/users/amrita/index.htm`). She was schooled (kindergarten to intermediate) at City Montessori School, Aliganj, Lucknow, Uttar Pradesh, India. She received a Bachelor of Technology degree in Information Technology from Institute of Engineering and Technology, Lucknow, Uttar Pradesh, India and a Master of Technology degree in Information Technology (with a specialization in Software Engineering) from Indian Institute of Information Technology, Allahabad (Deemed University), Uttar Pradesh, India. She has worked in Nucleus Software Exports Ltd., Noida, Uttar Pradesh, India as a software engineer. She was also employed as a faculty in Institute of Engineering and Technology, Lucknow, Uttar Pradesh, India. She has worked in user interface design as a senior project associate at Indian Institute of Technology, Kanpur, Uttar Pradesh, India. She was selected as the first female PhD student from Asia under EURECA (European Research and Educational Collaboration with Asia) project 2009 to conduct research at VU University Amsterdam, the Netherlands. Her areas of specialization are software engineering, software architecture, software design patterns, and ontologies. Her research interests include software architecture and design, ontologies-based software engineering, service-oriented and model-driven architecture, semantic web, Internet technologies, and mobile agents.

She has given several talks and seminars as well as conference welcome/key notes at international conferences. She has also earned various awards such as best paper award for her research paper in an international conference, ACM SIGAPP award, and has also been a Physics Olympiad topper. She has traveled several European, North American, African, and Asian countries for educational/conference/research purposes. She has teaching as well as research experience and has worked on several implementation-based projects both jointly in a team as well as independently. She has acted as a session chair and program committee member, as well as research paper reviewer for various international conferences.

I would like to thank my incredible and beloved husband, Bhartendu Chaturvedi, for his constant support.

Philippe Renevier Gonin has been an assistant professor at the University of Nice Sophia-Antipolis (UNS), France, since 2005. He teaches web technologies, software engineering (architecture, development), and HCI (Human Computer Interaction).

In the research area, he works on connections between user-centered design (for example, user and task models) and software engineering (for example, component architecture and UI development).

Pedro Miguel Barros Morgado holds a Master's degree in Informatics and Computing Engineering at FEUP (Faculdade de Engenharia da Universidade do Porto) and did his master thesis on Object-Oriented Patterns and Service-Oriented Patterns.

Since 2009, he has been working with several different programming languages, frameworks, and technologies, which included the main OO programming languages such as PHP, Python, C/C++, Java, and JavaScript as well as web languages such as HTML, JSON, and XML. He has worked with different database technologies such as MySQL, PostgreSQL, Oracle SQL, and SQL Server and also with different caching systems and search engines.

He has worked as an IT consultant in the banking field for a year, and built a recommendation system (data mining and text mining) as a research assistant at INESC (Technology and Science-Associated Laboratory) for a period of 1 year. Finally, he focused on web projects as a technical lead at Rocket Internet AG, for which he built scalable systems for FoodPanda, CupoNation, Camudi, and Lamudi. Due to his experience, he has specialized in project management and product development based on an e-commerce area. For more information, take a look at his LinkedIn account at `https://www.linkedin.com/in/pedrombmorgado`.

Mani Nilchiani is a developer, an artist, and a teacher based in Brooklyn, New York. He holds an MFA degree in Design and Technology from Parsons The New School for Design. He is a frontend engineer at The Daily Beast where he focuses on UI Development, API design, integration, and architecture, and works as an adjunct faculty at Parsons The New School for Design, where he teaches a graduate-level curriculum of JavaScript development and design patterns. As a digital artist, he approaches code as an expressive medium to create interactive, site-specific, and generative works of art.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

To my wife, Melissa, and children, Oliver and Sebastian, who have been with me every step of the way. Without their support, I would be but half a person.

Table of Contents

Preface

JavaScript is starting to become one of the most popular languages in the world. However, its history as a bit of a toy language, means that developers are tempted to ignore good design. Design patterns are a great tool to suggest some well-tried solutions.

What this book covers

This book is divided into two main parts, each of which contains a number of chapters. The first part of the book, which I'm calling *Part 1*, covers the classical design patterns, which are found in the GoF book. *Part 2* looks at patterns, which are either not covered in the GoF book or the ones that are more specific to JavaScript.

Chapter 1, *Designing for Fun and Profit*, provides an introduction to what design patterns are and why we are interested in using design patterns. We will also talk about the history of JavaScript to give a historical context.

Chapter 2, *Organizing Code*, explains how to create the classical structures used to organize code: namespaces or modules and classes as JavaScript lack these constructs as first class citizens.

Chapter 3, *Creational Patterns*, covers the creational patterns outlined in the Gang of Four book. We'll discuss how these patterns apply to JavaScript, as opposed to the languages which were popular at the time when the Gang of Four wrote their book.

Chapter 4, *Structural Patterns*, examines the structural patterns from the Gang of Four book following on our look at creational patterns.

Chapter 5, *Behavioral Patterns*, covers the final set of patterns from the Gang of Four book that we'll examine. These patterns govern different ways to link classes together.

Chapter 6, *Functional Programming*, explains some of the patterns which can be found in functional programming languages. We'll look at how these patterns can be used in JavaScript to improve code.

Chapter 7, *Model View Patterns*, examines the confusing variety of different patterns to create single-page applications. We'll provide clarity and look at how to use libraries which use each of the existing patterns, as well as create their own lightweight framework.

Chapter 8, *Web Patterns*, covers a number of patterns which have specific applicability to web applications. We'll also look at some patterns around deploying code to remote runtimes such as the browser.

Chapter 9, *Messaging Patterns*, explains messaging which is a powerful technique to communicate inside, and even between, applications. We'll also look at some common structures around messaging and discuss why messaging is so useful.

Chapter 10, *Patterns for Testing*, focuses on some patterns which make for easier testing, giving you more confidence that your application is working as it should.

Chapter 11, *Advanced Patterns*, includes some patterns, such as aspect-oriented programming, that are rarely applied in JavaScript. We'll look at how these patterns can be applied in JavaScript and discuss if we should apply them.

Chapter 12, *ES6 Solutions Today*, discusses some of the tools available to allow you to use features from future versions of JavaScript today. We'll examine Microsoft's TypeScript as well as Traceur.

Appendix, *Conclusion*, covers what you have learned, in general, in the book, and you will be reminded of the goal of using patterns.

What you need for this book

There is no specialized software needed for this book. JavaScript runs on all modern browsers. There are standalone JavaScript engines written in C++ (V8) and Java (Rhino), and these are used to power all sorts of tools such as Node.js, CouchDB, and even Elasticsearch. These patterns can be applied to any of these technologies.

Who this book is for

The intended audience is developers who have some experience with JavaScript, but not necessarily with entire applications written in JavaScript. Also, developers who are interested in creating easily maintainable applications that can grow and change with need.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The next item of interest is that we need to make use of the `this` qualifier to address the `greeting` variable from within the `doThings` function."

A block of code is set as follows:

```
var Wall = (function () {
  function Wall() {
    console.log("Wall constructed");
  }
  return Wall;
})();
Structures.Wall = Wall;
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
var Wall = (function () {
  function Wall() {
    console.log("Wall constructed");
  }
  return Wall;
})();
Structures.Wall = Wall;
```

Any command-line input or output is written as follows:

```
npm install -g traceur
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can also download the example code files for this book from GitHub at `https://github.com/stimms/JavaScriptPatterns`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata, under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Designing for Fun and Profit

JavaScript is an evolving language that has come a long way from its inception. Possibly more than any other programming language, it has grown and changed with the growth of the World Wide Web. The exploration of how JavaScript can be written using good design principles is the topic of this book. The preface of this book contains a detailed explanation of the sections of the book.

In the first half of this chapter, we'll explore the history of JavaScript and how it came to be the important language that it is today. As JavaScript has evolved and grown in importance, the need to apply rigorous methods to its construction has also grown. Design patterns can be a very useful tool to assist in developing maintainable code. The second half of the chapter will be dedicated to the theory of design patterns. Finally, we'll look briefly at antipatterns.

The topics covered in this chapter are:

- The history of JavaScript
- What is a design pattern?
- Antipatterns

The road to JavaScript

We'll never know how language first came into being. Did it slowly evolve from a series of grunts and guttural sounds made during grooming rituals? Perhaps it developed to allow mothers and their offsprings to communicate. Both of these are theories, all but impossible to prove. Nobody was around to observe our ancestors during that important period. In fact, the general lack of empirical evidence lead the Linguistic Society of Paris to ban further discussions on the topic, seeing it as unsuitable for serious study.

The early days

Fortunately, programming languages have developed in recent history and we've been able to watch them grow and change. JavaScript has one of the more interesting histories in modern programming languages. During what must have been an absolutely frantic 10 days in May of 1995, a programmer at Netscape wrote the foundation for what would grow up to be modern JavaScript.

At that time, Netscape was involved in the first of the browser wars with Microsoft. The vision for Netscape was far grander than simply developing a browser. They wanted to create an entire distributed operating system making use of Sun Microsystems' recently released Java programming language. Java was a much more modern alternative to C++ that Microsoft was pushing. However, Netscape didn't have an answer to Visual Basic. Visual Basic was an easier to use programming language, which was targeted at developers with less experience. It avoided some of the difficulties around memory management which makes C and C++ notoriously difficult to program. Visual Basic also avoided strict typing and overall allowed more leeway.

Brendan Eich was tasked with developing Netscape repartee to VB. The project was initially codenamed Mocha but was renamed LiveScript before Netscape 2.0 beta was released. By the time the full release was available, Mocha/LiveScript had been renamed JavaScript to tie it into the Java applet integration. Java applets were small applications that ran on the browser. They had a different security model from the browser itself and so were limited in how they could interact with both the browser and the local system. It is quite rare to see applets these days, as much of their functionality has become part of the browser. Java was riding a popular wave at that time and any relationship to it was played up.

The name has caused much confusion over the years. JavaScript is a very different language from Java. JavaScript is an interpreted language with loose typing that runs primarily on the browser. Java is a language that is compiled to bytecode, which is then executed on the Java Virtual Machine. It has applicability in numerous scenarios from the browser (through the use of Java applets) to the server (Tomcat, JBoss, and so on) to full desktop applications (Eclipse, OpenOffice). In most laypeople's minds, the confusion remains.

JavaScript turned out to be really quite useful for interacting with the web browser. It was not long until Microsoft had also adopted JavaScript in their Internet Explorer to complement VBScript. The Microsoft implementation was known as JScript.

By late 1996, it was clear that JavaScript was going to be the winning web language for the near future. In order to limit the amount of language deviation between implementations, Sun and Netscape began working with the **European Computer Manufacturers Association** (**ECMA**) to develop a standard to which future versions of JavaScript would need to comply. The standard was released very quickly (very quickly in terms of how rapidly standard organizations move) in July of 1997. On the off chance that you have not seen enough names yet for JavaScript, the standard version was called ECMAScript, a name which still persists in some circles.

Unfortunately, the standard only specified the very core parts of JavaScript. With the browser wars raging, it was apparent that any vendor that stuck with only the basic implementation of JavaScript would quickly be left behind. At the same time, there was much work going on to establish a standard **document object model** (**DOM**) for browsers. The DOM was, in effect, an API for a web page that could be manipulated using JavaScript.

For many years, every JavaScript script would start by attempting to determine the browser on which it was running. This would dictate how to address elements in the DOM, as there were dramatic deviations between each browser. The spaghetti of code that was required to perform simple actions was legendary. I remember reading a year-long 20 part series on developing **Dynamic HTML** (**DHTML**) drop-down menus such that they would work on both Internet Explorer and Netscape Navigator. The same functionality can now be achieved with pure CSS without even having to resort to JavaScript.

DHTML was a popular term in the late 1990s and early 2000s. It really referred to any web page that had some sort of dynamic content that was executed on the client side. It has fallen out of use as the popularity of JavaScript has made almost every page a dynamic one.

Fortunately, the efforts to standardize JavaScript continued behind the scenes. Versions 2 and 3 of ECMAScript were released in 1998 and 1999. It looked like there might finally be some agreement between the various parties interested in JavaScript. Work began in early 2000 on ECMAScript 4, which was to be a major new release.

A pause

Then, disaster struck! The various groups involved in the ECMAScript effort had major disagreements about the direction JavaScript was to take. Microsoft seemed to have lost interest in the standardization effort. It was somewhat understandable as it was around that time that Netscape self-destructed and Internet Explorer became the de facto standard. Microsoft implemented parts of ECMAScript 4 but not all of it. Others implemented more fully featured support but, without the market leader on board, developers didn't bother using them.

Years passed without consensus and without a new release of ECMAScript. However, as frequently happens, the evolution of the Internet could not be stopped by a lack of agreement between major players. Libraries such as jQuery, Prototype, Dojo, and MooTools papered over the major differences in browsers, making cross-browser development far easier. At the same time, the amount of JavaScript used in applications increased dramatically.

The way of Gmail

The turning point was, perhaps, the release of Google's Gmail application in 2004. Although XMLHttpRequest, the technology behind **Asynchronous JavaScript and XML** (**AJAX**), had been around for about 5 years when Gmail was released, it had not been well used. When Gmail was released, I was totally knocked off my feet by how smooth it was. We've grown used to applications that avoid full reloads, but at that time it was a revolution. To make applications like that work, a great deal of JavaScript is needed.

AJAX is a method by which small chunks of data are retrieved from the server by a client instead of refreshing the entire page. The technology allows for more interactive pages that avoid the jolt of full page reloads.

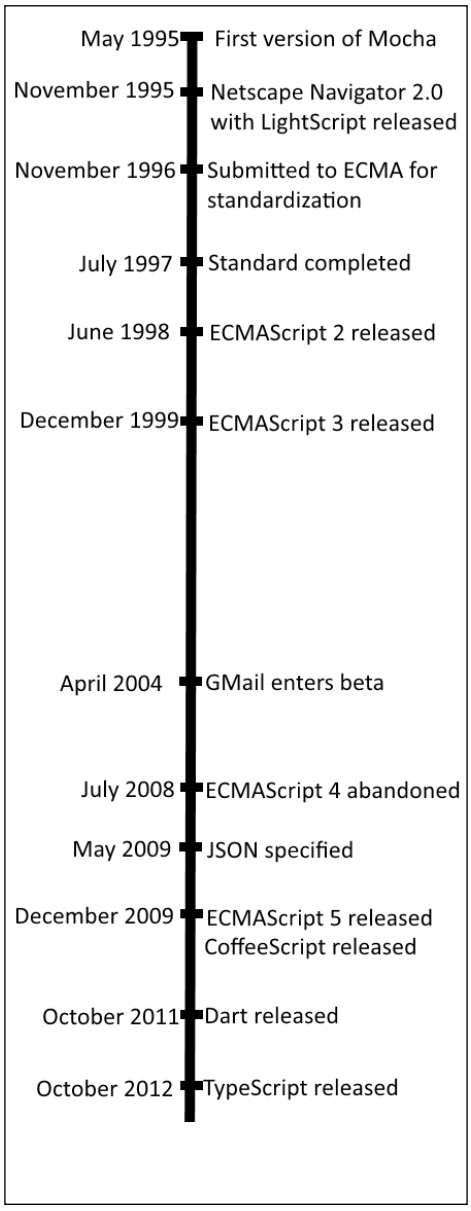
The popularity of Gmail was the trigger for a change that had been brewing for a while. Increasing JavaScript acceptance and standardization pushed us past the tipping point for the acceptance of JavaScript as a proper language. Up until that point, much of the use of JavaScript was for performing minor changes to the page and for validating form input. I joke with people that in the early days of JavaScript the only function name that was used was `Validate()`.

Applications such as Gmail that have a heavy reliance on AJAX and avoid full-page reloads are known as **single page applications** (**SPAs**). By minimizing the changes to the page contents, users have a more fluid experience. By transferring only **JavaScript Object Notation** (**JSON**) payload, instead of HTML, the amount of bandwidth required is also minimized. This makes applications appear to be snappier. In recent years, there have been great advances in frameworks that ease the creation of SPAs. AngularJS, Backbone.js, and Ember.js are all Model View Controller style frameworks. They have gained great popularity in the past two to three years and provide some interesting use of patterns. These frameworks are the evolution of years of experimentation with JavaScript best practices by some very smart people.

JSON is a human readable serialization format for JavaScript.
It has become very popular in recent years as it is easier and less cumbersome than previously popular formats such as XML.
It lacks many of the companion technologies and strict grammatical rules of XML, but makes up for it with simplicity.

At the same time as the frameworks using JavaScript are evolving, the language is too. A much vaunted new version of JavaScript has been under development for some years. ECMAScript 6 will bring some great improvements to the ecosystem. A number of other languages that transcompile to JavaScript are also gaining popularity. CoffeeScript is a Python-like language that strives to improve the readability and brevity of JavaScript. Developed by Google, Dart is being pushed by Google as an eventual replacement for JavaScript. Its construction addresses some of the optimizations that are impossible in traditional JavaScript. Until a Dart runtime is sufficiently popular, Google provides a Dart to JavaScript transcompiler. TypeScript is a Microsoft project that adds some ECMAScript 6 syntax as well as an interesting typing system to JavaScript. It aims to address some of the issues that large JavaScript projects present.

You can see the growth of JavaScript in the following timeline:



The point of this discussion about the history of JavaScript is twofold: first it is important to remember that languages do not develop in a vacuum. Both human languages and computer programming languages mutate based on the environments in which they are used. It is a popularly held belief that the Inuit people have a great number of words for "snow" as it was so prevalent in their environment. This may or may not be true depending on your definition for the word and exactly who makes up the Inuit people. There are, however, a great number of examples of domain-specific lexicons evolving to meet the requirements for exact definitions in narrow fields. One need look no further than a specialty cooking store to see the great number of variants of items which a layperson, such as me, would call a pan.

The Sapir–Whorf hypothesis is a hypothesis within the linguistics domain that suggests that not only is language influenced by the environment in which it is used but also that language influences its environment. Also known as linguistic relativity, the theory is that one's cognitive processes differ based on how the language is constructed. Cognitive psychologist Keith Chen has proposed a fascinating example of this. In a very highly viewed TED talk, Dr. Chen suggested that there is a strong positive correlation between languages that lack a future tense and those that have high saving rates (`https://www.ted.com/talks/keith_chen_could_your_language_affect_your_ability_to_save_money/transcript`). The hypothesis at which Dr. Chen arrived is that when your language does not have a strong sense of connection between the present and the future, it leads to more reckless behavior in the present.

Thus, understanding the history of JavaScript puts one in a better position to understand how and where to make use of JavaScript.

The second reason I explored the history of JavaScript is because it is absolutely fascinating to see how quickly such a popular tool has evolved. At the time of this writing, it has been less than 20 years since JavaScript was first built and its rise to popularity has been explosive. What is more exciting than to work in an ever evolving language?

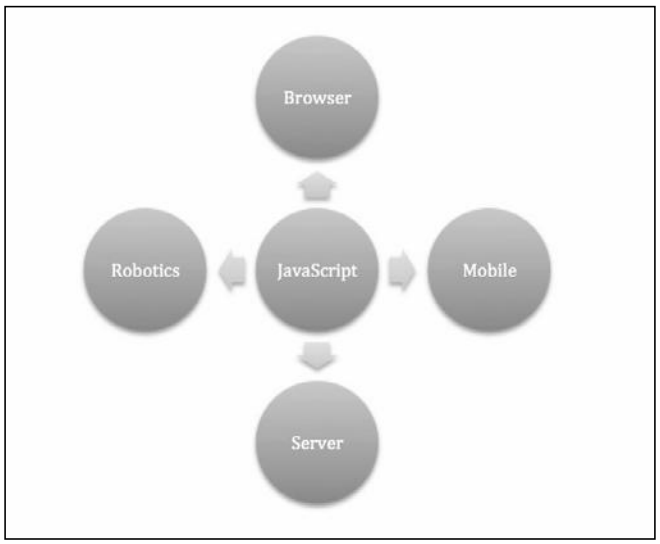
JavaScript everywhere

Since the Gmail revolution, JavaScript has grown immensely. The renewed browser wars, which pit Internet Explorer against Chrome against Firefox, have lead to building a number of very fast JavaScript interpreters. Brand new optimization techniques have been deployed and it is not unusual to see JavaScript compiled to machine native code for the added performance it gains. However, as the speed of JavaScript has increased, so has the complexity of the applications built using it.

JavaScript is no longer simply a language for manipulating the browser either. The JavaScript engine behind the popular Chrome browser has been extracted and is now at the heart of a number of interesting projects such as Node.js. Node.js started off as a highly asynchronous method of writing server-side applications. It has grown greatly and has a very active community supporting it. A wide variety of applications have been built using the Node.js runtime. Everything from build tools to editors have been built on the base of Node.js.

JavaScript can even be used to control microcontrollers. The Johnny-Five framework is a programming framework for the very popular Arduino. It brings a much simpler approach to programming the devices than the traditional low-level languages used for programming these devices. Using JavaScript and Arduino opens up a world of possibilities, from building robots to interacting with real-world sensors.

All of the major smartphone platforms (iOS, Android, and Windows Phone) have an option to build applications using JavaScript. The tablet space is much the same, with tablets supporting programming using JavaScript. Even the latest version of Windows provides a mechanism for building applications using JavaScript. JavaScript is used everywhere, as shown in the following diagram:



JavaScript is becoming one of the most important languages in the world. Although language usage statistics are notoriously difficult to calculate, every single source that attempts to develop a ranking puts JavaScript in the top 10:

Language index	Rank of JavaScript
Langpop.com	4
Statisticbrain.com	5
Codeval.com	5
TIOBE	9

What is more interesting is that each one of these rankings suggests that the usage of JavaScript popularity is on the rise.

The long and short of it is that JavaScript is going to be a major language in the next few years. More and more applications are being written in JavaScript and it is the *lingua franca* for any sort of web development. The developer of the popular Stack Overflow website, Jeff Atwood, created Atwood's Law regarding the wide adoption of JavaScript:

> *Any application that can be written in JavaScript, will eventually be written in JavaScript.*

This insight has been proven to be correct time and time again. There are now compilers, spreadsheets, word processors—you name it—all written in JavaScript.

As the applications that make use of JavaScript increase in complexity, the developers stumble upon many of the same issues as have been encountered in traditional programming languages: How can we write this application to be adaptable to change?

This brings us to the need for properly designing applications. No longer can we simply throw a bunch of JavaScript into a file and hope that it works properly. Nor can we rely on libraries such as jQuery to save ourselves. Libraries can only provide additional functionality, and they contribute nothing to the structure of an application. At least some attention must now be paid to how to construct the application to be extensible and adaptable. The real world is ever changing and any application that is unable to change to suit the changing world is likely to be left in the dust. Design patterns provide some guidance in building adaptable applications, which can shift with changing business needs.

What is a design pattern?

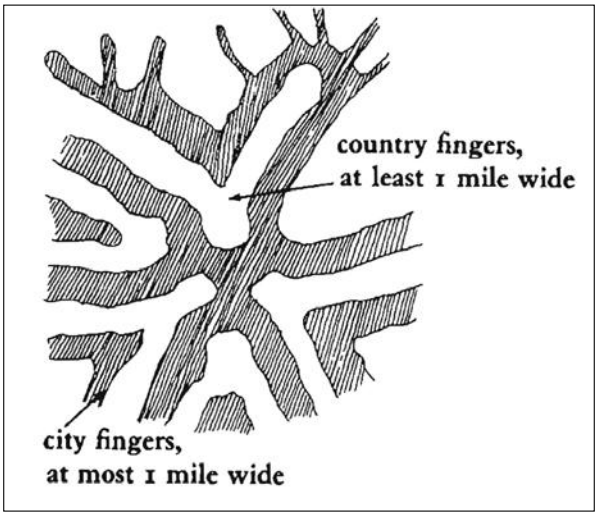
For the most part, ideas are only applicable in one place. Adding peanut butter is really only a great idea in cooking and not in sewing. However, from time to time, it is possible to find applicability for a great idea outside of its original purpose. This is the story behind design patterns.

In 1977, Christopher Alexander, Sara Ishikawa, and Murray Silverstein authored a seminal book on what they called design patterns in urban planning called *A Pattern Language: Towns, Buildings, Construction, Oxford*.

The book described a language for talking about the commonalities of design. In the book, a pattern is described by Christopher Alexander as follows:

> *The elements of this language are entities called patterns. Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*

These design patterns included such things as how to lay out cities to provide a mixture of city and country living or how to build roads in loops as a traffic calming measure in residential areas. This is shown in the following image taken from the book:



Even for those without a strong interest in urban planning, the book presents some fascinating ideas about how to structure our world to promote healthy societies.

Using the work of Christopher Alexander and the other authors as a source of inspiration, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides wrote a book called *Design Patterns: Elements of Reusable Object-Oriented Software*, *Addison-Wesley*. When a book is very influential in computing science curriculum, it is often given a pet name. For instance, most computing science graduates will know of which book you speak if you talk about "the dragon book" (*Principles of Compiler Design*, *Addison-Wesley*, 1986). In enterprise software, "the blue book" is well known to be Eric Evan's book on domain-driven design. The design patterns book has been so important that it is commonly referred do as the **Gang of Four** (**GoF**) book for its four authors.

This book outlined 23 patterns for use in object-oriented design. It divided the patterns into three major groups:

- **Creational**: These patterns outline a number of ways in which objects could be created and their lifecycles managed
- **Behavioral**: These patterns describe how objects interact with each other
- **Structural**: These patterns describe a variety of different ways to add functionality to existing objects

The purpose of design patterns is not to instruct you how to build software but rather to give guidance on ways in which to solve common problems. For instance, many applications have a need to provide some sort of an undo function. The problem is common to text editors, drawing programs, and even e-mail clients. Solving this problem has been done many times before, so it would be great to have a common solution. The command pattern provides just such a common solution. It suggests keeping track of all the actions performed in an application as instances of a command. This command will have forward and reverse actions. Every time a command is processed, it is placed onto a queue. When the time comes to undo a command, it is as simple as popping the top command off of the command queue and executing the undo action on it.

Design patterns provide some hints about how to solve common problems such as the undo problem. They have been distilled from performing hundreds of iterations of solving the same problem. The design pattern may not be exactly the correct solution for the problem you have, but it should, at the very least, provide some guidance to implement a solution more easily.

A consultant friend of mine once told me a story about starting an assignment at a new company. The manager told them that he didn't think there would be a lot of work to do with the team because they had bought the GoF design pattern book for the developers early on and they'd implemented every last design pattern. My friend was delighted about hearing this because he charges by the hour. The misapplication of design patterns paid for much of his first-born's college education.

Since the GoF book, there has been a great proliferation of literature dealing with enumerating and describing design patterns. There are books on design patterns that are specific to a certain domain and books that deal with patterns for large enterprise systems. The Wikipedia category for software design patterns contains 130 entries for different design patterns. I would, however, argue that many of the entries are not true design patterns but rather programming paradigms.

For the most part, design patterns are simple constructs that don't need complicated support from libraries. While there do exist pattern libraries for most languages, you need not go out and spend a lot of money to purchase the libraries. Implement the patterns as you find the need. Having an expensive library burning a hole in your pocket encourages blindly applying patterns just to justify having spent the money. Even if you did have the money, I'm not aware of any libraries for JavaScript whose sole purpose is to provide support for patterns. Of course, GitHub has a wealth of interesting JavaScript projects, so there may well be a library on there of which I'm unaware.

There are some who suggest that design patterns should be emergent. That is to say that by simply writing software in an intelligent way one can see the patterns emerge from the implementation. I think that may be an accurate statement; however, it ignores the actual cost of getting to those implementations by trial and error. Those with an awareness of design patterns are much more likely to spot emergent patterns early on. Teaching junior programmers about patterns is a very useful exercise. Knowing early on which pattern or patterns can be applied acts as a shortcut. The full solution can be arrived at earlier and with fewer missteps.

Antipatterns

If there are common patterns to be found in good software design, are there also patterns that can be found in bad software design? Absolutely! There is any number of ways to do things incorrectly but most of them have been done before. It takes real creativity to screw up in a hitherto unknown way.

The shame of it is that it is very difficult to remember all the ways in which people have gone wrong over the years. At the end of many major projects, the team will sit down and put together a document called **lessons learned**. This document contains a list of things that could have gone better on the project and may even outline some suggestions as to how these issues can be avoided in the future. That these documents are only constructed at the end of a project is unfortunate. By that time, many of the key players have moved on and those who are left must try to remember lessons from the early stages of the project, which could be years ago. It is far better to construct the document as the project progresses.

Once complete, the document is filed away ready for the next project to make use of it. At least that is the theory. For the most part, the document is filed away and never used again. It is difficult to create lessons that are globally applicable. The lessons learned tend to only be useful for the current project or an exactly identical project, which almost never happens.

However, by looking at a number of these documents from various projects, patterns start to emerge. It was by following such an approach that William Brown, Raphael Malveau, Skip McCormick, and Tom Mowbray, collectively known as the **Upstart Gang of Four** in reference to the original Gang of Four, wrote the initial book on antipatterns. This book, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, John Wiley & Sons, Inc.*, outlined antipatterns for not just issues in code but also in the management process that surrounds code.

Patterns outlined include such humorously named patterns as The Blob and Lava Flow. The Blob, also known as the **god object**, is a pattern in which one object grows to take on the responsibility for vast swaths of the application logic. Lava Flow is a pattern that emerges as a project ages and nobody knows if code is still used. Developers are nervous about deleting the code because it might be used somewhere or may become useful again. There are many other patterns described in the book that are worth exploring. Just as with patterns, antipatterns are emergent from writing code, but in this case code that gets out of hand.

This book will not cover JavaScript antipatterns, but it is useful to remember that one of the antipatterns is an overapplication of design patterns.

Summary

Design patterns have a rich and interesting history. From their origin as tools for helping to describe how to build the structures to allow people to live together, they have grown to be applicable to a number of domains.

It has now been a decade since the seminal work on applying design patterns to programming. Since then a vast number of new patterns have been developed. Some of these patterns are general-purpose patterns such as those outlined in the GoF book, but a larger number are very specific patterns that are designed for use in a narrow domain.

JavaScript has an interesting history and is really coming of age. With server-side JavaScript taking off and large JavaScript applications becoming common, there is a need for more diligence in building JavaScript applications. It is rare to see patterns being properly exploited in most modern JavaScript code.

Leaning on the teachings provided by design patterns to build modern JavaScript patterns gives one the best of both worlds. As Isaac Newton famously wrote:

> *If I have seen further it is by standing on ye shoulders of Giants.*

Patterns give us easily accessible shoulders on which to stand.

In the next chapter, we will look at some techniques for building structure into JavaScript. The inheritance system in JavaScript is unlike that of most other object-oriented languages and that provides us both opportunities and limits. We'll see how to build classes and modules in the JavaScript world.

Part 1

Classical Design Patterns

Organizing Code

Creational Patterns

Structural Patterns

Behavioral Patterns

2
Organizing Code

In this chapter, we'll look at how to organize JavaScript code into reusable, understandable chunks. The language doesn't lend itself well to this sort of modularization, but a number of methods of organizing JavaScript code have emerged over the years. This chapter will argue the need to break down code, and then work through the methods of creating JavaScript modules.

We will cover the following topics:

- Global scope
- Objects
- Prototype inheritance
- ECMAScript 6 classes

Chunks of code

The first thing anybody learns to program is the ubiquitous **hello world** application. This simple application prints some variation of "hello world" to the screen. Depending on who you ask, the phrase hello world dates back to the early 1970s where it is used to demonstrate the B programming language, or even to 1967 where it appears in a BCL programming guide. In such a simple application, there is no need to worry about the structure of code. Indeed in many programming languages, hello world needs no structure at all, as shown in the following two languages:

- In Ruby:

```
#!/usr/bin/ruby
puts "hello world"
```

- In JavaScript (via Node.js):

```
#!/usr/local/bin/node
console.log("Hello world")
```

Programming modern computers was originally done using brutally simplistic techniques. Many of the first computers had the problems they were attempting to solve hardwired into them. They were not general purpose computing machines the likes of which we have today. Instead, they were built to solve just one problem of decoding encrypted text. Stored program computers were first developed in the late 1940s.

The languages used to program these computers were complicated at first, usually very closely tied to binary. Eventually, higher and higher-level abstractions were created to make programming more accessible. As these languages started to take shape through the '50s and '60s, it quickly became apparent that there needed to be some way to divide up large blocks of code.

In part, this was simply to maintain the sanity of programmers who could not keep an entire large program in their heads at any one time. However, creating reusable modules also allowed for code to be shared within the application and even between applications. The initial solution was to make use of statements, which jumped the flow control of the program from one place to another. For a number of years, these GOTO statements were heavily relied upon. To a modern programmer who has been fed a continual stream of warnings about the use of GOTO statements, this seems like insanity. However, it was not until some years after the first programming languages emerged that structured programming grew to replace the GOTO syntax.

Structured programming is based on the Böhm-Jacopini theorem, which states that there is a rather large class of problems, the answer to which can be computed using three very simple constructs:

- Sequential execution of subprograms
- Conditional execution of two subprograms
- Repeated execution of a subprogram until a condition is true

Astute readers will recognize these constructs as being the normal flow of execution, a branch or `if` statement and a loop.

Fortran was one of the earliest languages and was initially built without support for structured programming. However, structured programming was soon adopted as it helped avoid spaghetti code.

Code in Fortran was organized into modules. Modules were loosely coupled collections of procedures. For those coming from a modern object-oriented language, the closest concept might be that a module was like a class that contains only static methods.

Modules were useful for dividing code into logical groupings. However, it didn't provide for any sort of structure for the actual applications. The structure for object-oriented languages, that is classes and subclasses, can be traced to a 1967 paper written by Ole-Johan Dahl and Kristen Nygaard. This paper would go on to form the basis of Simula-67, the first language with the support of object-oriented programming.

While Simula-67 was the first language to have classes, the language most talked about in relation to early object-oriented programming is Smalltalk. This language was developed (behind closed doors) at the famous Xerox **Palo Alto Research Center** (**PARC**) during the 1970s. It was released to the public in 1980 as Smalltalk-80 (it seems like all historically relevant programming languages were suffixed with the year of release as a version number). What Smalltalk brought was that everything in the language was an object, even literal numbers such as 3, could have operations performed on them.

Almost every modern programming language has some concept of classes to organize code. Often these classes will fall into a higher-level structure commonly called a **namespace** or **modules**. Through the use of these structures, even very large programs can be divided into manageable and understandable chunks.

Despite the rich history and obvious utility of classes and modules, JavaScript does not support them as first class constructs. To understand why, one has to simply look back at the history of JavaScript from *Chapter 1*, *Designing for Fun and Profit*, and realize that for its original purpose, having such constructs would have been overkill. Classes were a part of the ill-fated ECMAScript 4 standard and they are a part of the upcoming ECMAScript 6, but for the moment they don't exist.

In this chapter, we'll explore some of the ways to recreate the well-worn class structure of other modern programming languages in JavaScript.

What's the matter with global scope anyway?

In browser-based JavaScript, every object you create is assigned to the global scope. For the browser, this object is simply known as **window**. It is simple to see this behavior in action, by opening up the development console in your favorite browser.

Opening the development console

Modern browsers have, built into them, some very advanced debugging and auditing tools. To access them, there is a menu item, which is located under **More tools** | **Developer tools** in Chrome, **Tools** | **Web Developer** in Firefox, and directly under the menu as **F12 developer tools** in Internet Explorer. Keyboard shortcuts also exist for accessing the tools. On Windows and Linux, *F12* is standard and on OSX, *option* + *command* + *I* is used.

Within the developer tools, there is a console window that provides direct access to the current page's JavaScript. This is a very handy place to test out small snippets of code or to access the page's JavaScript.

Once you have the console open, enter the following code:

```
var words = "hello world"
console.log(window.words);
```

The result of this will be that `hello world` is printed to the console. By declaring words globally, it is automatically attached to the top-level container: window.

In Node.js, the situation is somewhat different. Assigning a variable in this fashion will actually attach it to the current module. Not including `var` will attach the variable to the `global` object.

For years, you've probably heard that making use of global variables is a bad thing. This is because globals are very easily polluted by other code.

Consider a very commonly named variable such as `index`. It is likely that in any application of appreciable size, this variable name would be used in several places. When either piece of code makes use of the variable, it will cause unexpected results in the other piece of code. It is certainly possible to reuse variables, and can even be useful in systems with very limited memory, such as embedded systems; however, in most applications, reusing variables to mean different things within a single scope is difficult to understand and is also a source of errors.

Applications that make use of global scoped variables also open themselves to being attacked on purpose by other code. It is trivial to alter the state of global variables from other code, which could expose secrets (such as login information) to attackers.

Finally, global variables add a great deal of complexity to applications. Reducing the scope of variables to a small section of code, allows developers to more easily understand the ways in which the variable is used. When the scope is global, changes to that variable may have an effect far outside of that particular section of code. A simple change to a variable can cascade into the entire application.

As a general rule, global variables should be avoided.

Objects in JavaScript

JavaScript is an object-oriented language, but most people don't make use of the object-oriented features of it except in passing. JavaScript uses a mixed object model in that it has some primatives as well as objects. JavaScript has five primitive types:

- Undefined
- Null
- Boolean
- String
- Number

Of these five, really only three of them are what one would expect to be an object anyway. The other three, boolean, string, and number all have wrapped versions, which are objects: Boolean, String, and Number. They are distinguished by starting with uppercase. This is the same sort of model used by Java: a hybrid of objects and primitives.

JavaScript will also box and unbox the primitives as needed.

In the following code, you can see the boxed and unboxed versions of JavaScript primitives at work:

```
var numberOne = new Number(1);
var numberTwo = 2;
typeof numberOne; //returns 'object'
typeof numberTwo; //returns 'number'
var numberThree = numberOne + numberTwo;
typeof numberThree; //returns 'number'
```

Creating objects in JavaScript is trivial. This can be seen in the following code for creating an object in JavaScript:

```
var objectOne = {};
typeof objectOne; //returns 'object'
var objectTwo = new Object();
typeof objectTwo; //returns 'object'
```

Because JavaScript is a dynamic language, adding properties to objects is also quite easy. This can be done even after the object has been created. The following code creates:

```
var objectOne = { value: 7 };
var objectTwo = {};
objectTwo.value = 7;
```

Objects contain both data and functionality. We've only seen the data part so far. Fortunately, in JavaScript, functions are first class objects. Functions can be passed around and assigned to variables. Let's try adding some functions to an object, as seen in the following code:

```
var functionObject = {};
functionObject.doThings = function() {
  console.log("hello world");
}
functionObject.doThings(); //writes "hello world" to the console
```

This syntax is a bit painful, building up objects an assignment at a time. Let's see if we can improve upon the syntax for creating objects:

```
var functionObject = {
  doThings: function() {
    console.log("hello world");
  }
}
functionObject.doThings();//writes "hello world" to the console
```

This syntax seems, at least to me, to be a much cleaner, more traditional way of building objects. Of course, it is possible to mix data and functionality in an object in the following fashion:

```
var functionObject = {
  greeting: "hello world",
  doThings: function() {
    console.log(this.greeting);
  }
}
functionObject.doThings();//prints hello world
```

There are a couple of things to note in this piece of code. The first is that the different items in the object are separated using a comma, not a semicolon. Those coming from other languages such as C# or Java are likely to make this mistake, I know that I did.

The next item of interest is that we need to make use of the `this` qualifier to address the `greeting` variable from within the `doThings` function. This would also be true if we had a number of functions within the object, as shown in the following code:

```
var functionObject = {
  greeting: "hello world",
  doThings: function() {
    console.log(this.greeting);
    this.doOtherThings();
  },
  doOtherThings: function() {
    console.log(this.greeting.split("").reverse().join(""));
  }
}
functionObject.doThings();//prints hello world then dlrow olleh
```

The `this` keyword behaves differently in JavaScript from what you might expect coming from other C-syntax languages. This is bound to the owner of the function in which it is found. However, the owner of the function is sometimes not what you expect. In the preceding example, this is bound to `functionObject`; however, if the function was declared outside of an object, this would refer to the global object. In certain circumstances, typically event handlers, this is rebound to the object firing the event.

Thus, in the following code, `this` takes on the value of target. Getting used to the value of `this` is, perhaps, one of the trickiest things in JavaScript:

```
var target = document.getElementById("someId");
target.addEventListener("click", function() {
  console.log(this);
}, false);
```

We have built up a pretty complete model of how to build objects within JavaScript. However, objects are not the same thing as classes. Objects are instances of classes. If we want to create multiple instances of our `functionObject` instance, we're out of luck. Attempting to do so will result in an error. In the case of Node.js, the error will be as follows:

```
var obj = new functionObject();
TypeError: object is not a function
    at repl:1:11
```

```
    at REPLServer.self.eval (repl.js:110:21)
    at repl.js:249:20
    at REPLServer.self.eval (repl.js:122:7)
    at Interface.<anonymous> (repl.js:239:12)
    at Interface.EventEmitter.emit (events.js:95:17)
    at Interface._onLine (readline.js:202:10)
    at Interface._line (readline.js:531:8)
    at Interface._ttyWrite (readline.js:760:14)
    at ReadStream.onkeypress (readline.js:99:10)
```

The stack trace here shows an error in a module called `repl`. This is the **read-execute-print loop** that is loaded by default when starting Node.js.

Each time that a new instance is required, the object must be reconstructed. To get around `this`, we can define the object using a function, as shown in the following code:

```
var ThingDoer = function(){
  this.greeting = "hello world";
  this.doThings = function() {
    console.log(this.greeting);
    this.doOtherThings();
  };
  this.doOtherThings = function() {
    console.log(this.greeting.split("").reverse().join(""));
  };
}
var instance = new ThingDoer();
instance.doThings();//prints hello world then dlrow olleh
```

This syntax allows for a constructor to be defined and for new objects to be created from this function. Constructors without return values are functions that are called as an object is created. In JavaScript, the constructor actually returns the object created. You can even assign internal properties using the constructor by making them part of the initial function, as follows:

```
var ThingDoer = function(greeting){
  this.greeting = greeting;
  this.doThings = function() {
    console.log(this.greeting);
  };
}
var instance = new ThingDoer("hello universe");
instance.doThings();
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Build me a prototype

As I've previously mentioned, there is currently no support for creating true classes in JavaScript. Objects created using the structure in the previous section have a fairly major drawback: creating multiple objects is not only time-consuming but also memory intensive. Each object is completely distinct from other objects created in the same fashion. This means that the memory used to hold the function definitions is not shared between all instances. What is even more fun is that you can redefine individual instances of a class without changing all of the instances. This is demonstrated in the following code:

```
var Castle = function(name){
  this.name = name;
  this.build = function() {
    console.log(this.name);
  };
}
var instance1 = new Castle("Winterfell");
var instance2 = new Castle("Harrenhall");
instance1.build = function(){ console.log("Moat Cailin");}
instance1.build(); //prints "Moat Cailin"
instance2.build(); //prints "Harrenhall" to the console
```

Altering functionality of a single instance, or really of any already defined object in this fashion, is known as **monkey patching**. There is some division over whether or not this is good practice. It can certainly be useful when dealing with library code but it adds great confusion. It is generally considered to be better practice to extend the existing class.

Without a proper class system, JavaScript, of course, has no concept of inheritance. What it does have is a prototype. At the most basic level, an object in JavaScript is an associative array of keys and values. Each property or function on an object is simply defined as part of this array.

You can even see this in action by accessing members of an object using the array syntax, as is shown in the following code:

```
var thing = { a: 7};
console.log(thing["a"]);
```

Accessing members of an object using the array syntax can be a very handy way to avoid using the `eval` function. For instance, if I had the name of the function I wanted to call in a string called `funcName` and I wanted to call it on an object, `obj1`, then I could do so by doing `obj1[funcName]()` instead of using a potentially dangerous call to `eval`. The `eval` function allows for arbitrary code to be executed. Allowing this on a page means that an attacker may be able to enter malicious scripts on other people's browsers.

When an object is created, its definition is inherited from a prototype. Weirdly, each prototype is also an object, so even prototypes have prototypes. Well, except for object, which is the top-level prototype. The advantage to attaching functions to the prototype is that only a single copy of the function is created; saving on memory. There are some complexities to prototypes, but you can certainly survive without knowing about them. To make use of a prototype, you need to assign functions to it, as shown in the following code:

```
var Castle = function(name){
  this.name = name;
}
Castle.prototype.build = function(){ console.log(this.name);}
var instance1 = new Castle("Winterfell");
instance1.build();
```

One thing to note is that only the functions are assigned to the prototype. Instance variables such as `name` are still assigned to the instance. As these are unique to each instance, there is no real impact on the memory usage.

In many ways, a prototypical language is more powerful than a class-based inheritance model.

If you make a change to the prototype of an object at a later date, then all the objects which share that prototype, are updated with the new function. This removes some of the concerns expressed about monkey typing. An example of this behavior is shown in the following code:

```
var Castle = function(name){
  this.name = name;
};
```

```
Castle.prototype.build = function(){
  console.log(this.name);
};
var instance1 = new Castle("Winterfell");
Castle.prototype.build = function(){
  console.log(this.name.replace("Winterfell", "Moat Cailin"));
}
instance1.build();//prints "Moat Cailin" to the console
```

When building up objects, you should be sure to take advantage of the prototype object whenever possible.

Now we know about prototypes. There is an alternative approach to building objects in JavaScript, and that is to use the `Object.create` function. This is a new syntax introduced in ECMAScript 5, which is:

```
Object.create(prototype [, propertiesObject ] )
```

The `create` syntax will build a new object based on the given prototype. You can also pass in a `propertiesObject` parameter that describes additional fields on the created object. These descriptors consist of a number of optional fields:

- `writable`: This dictates whether the field should be writable
- `configurable`: This dictates whether the files should be removable from the object, or support further configuration after creation
- `enumerable`: This checks whether the property can be listed during an enumeration of the object's properties
- `value`: This is the default value of the field

It is also possible to assign a get and set function within the descriptor that acts as getters and setters for some other internal property.

Using `object.create` for our `Castle` class, we can build an instance using `Object.create`, as shown in the following code:

```
var instance3 = Object.create(Castle.prototype, {name: { value:
"Winterfell", writable: false}});
instance3.build();
instance3.name="Highgarden";
instance3.build();
```

You'll notice that we explicitly define the `name` field. The `Object.create` function bypasses the constructor, so the initial assignment we described in the preceding code won't be called. You might also notice that `writeable` is set to `false`. The result of this is that the reassignment of `name` to `"Highgarden"` has no effect. The output is:

```
Winterfell
Winterfell
```

Inheritance

One of the niceties of objects is that they can be built upon to create increasingly more complex objects. This is a common pattern, which is used for any number of things. There is no inheritance in JavaScript because of its prototypical nature. However, you can combine functions from one prototype into another.

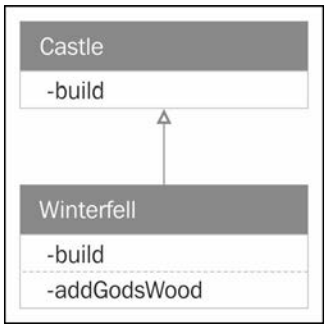
Let's say that we have a base class called `Castle` and we want to customize it into a more specific class called `Winterfell`. We can do so by first copying all of the properties from the `Castle` prototype onto the `Winterfell` prototype. This can be done as shown in the following code:

```
var Castle = function(){};
Castle.prototype.build = function(){console.log("Castle built");}

var Winterfell = function(){};
Winterfell.prototype.build = Castle.prototype.build;
Winterfell.prototype.addGodsWood = function(){}
var winterfell = new Winterfell();
winterfell.build(); //prints "Castle built" to the console
```

Of course, this is a very painful way to build objects. You're forced to know exactly which functions the base class has to copy them. It can be abstracted in a rather naive fashion as follows:

```
function clone(source, destination) {
  for(var attr in source.prototype){ destination.prototype[attr] =
  source.prototype[attr];}
}
```

The following is the class diagram of the `Castle` class:



The following class code can be used quite simply:

```
var Castle = function(){};
Castle.prototype.build = function(){console.log("Castle built");};

var Winterfell = function(){};
clone(Castle, Winterfell);
var winterfell = new Winterfell();
winterfell.build();
```

We say that this is naive because it fails to take into account a number of potential failure conditions. A fully fledged implementation is quite extensive. The jQuery library provides a function called `extend` which implements prototype inheritance in a robust fashion. It is about 50 lines long and deals with deep copies and null values. The function is internally used in jQuery extensively, but it can be a very useful function in your own code. We mentioned that prototype inheritance is more powerful than the traditional methods of inheritance. This is because it is possible to mix and match bits from many base classes to create a new class. In most modern languages, there is support for only single inheritance: a class can have only one direct parent. There are some languages where there is multiple inheritance, however, it is a practice that adds a great deal of complexity when attempting to decide which version of a method to call at runtime. Prototype inheritance avoids many of these issues by forcing selection of a method at assembly time.

Composing objects in this fashion permits taking properties from two or more different bases. There are many times when this can be useful. For example, a class representing a wolf might take some of its properties from a class describing a dog and some from another class describing a quadruped.

By using classes built in this way, we can meet pretty much all of the requirements for constructing a system of classes including inheritance. However, inheritance is a very strong form of coupling. In almost all cases, it is better to avoid inheritance in favor of a looser form of coupling. This will allow for classes to be replaced or altered with a minimum impact on the rest of the system.

Modules

Now that we have a complete class system, it would be good to address the global namespace discussed earlier. Again there is no first class support for namespaces but we can easily isolate functionality to the equivalent of a namespace. There are a number of different approaches to creating modules in JavaScript. We'll start with the simplest and add some functionality as we go along.

To start, we simply need to attach an object to the global namespace. This object will contain our root namespace. We'll name our namespace `Westeros`; the code simply looks like this:

```
Westeros = {}
```

This object is, by default, attached to the top-level object, so we need not do anything more than that. A typical usage is to first check if the object already exists, and use that version instead of reassigning the variable. This allows you to spread your definitions over a number of files. In theory, you could define a single class in each file and then bring them all together as part of the build process, before delivering them to the client or using them in an application. The short form of this is:

```
Westeros = Westeros || {}
```

Once we have the object, it is simply a question of assigning our classes as properties of that object. If we continue to use the `Castle` object, then it would look like this:

```
var Westeros = Westeros || {};
Westeros.Castle = function(name){this.name = name}; //constructor
Westeros.Castle.prototype.Build = function(){console.log("Castle
built: " +  this.name)};
```

If we want to build a hierarchy of namespaces that is more than a single-level deep, that too is easily accomplished, as shown in the following code:

```
var Westeros = Westeros || {};
Westeros.Structures = Westeros.Structures || {};
Westeros.Structures.Castle = function(name){ this.name = name};
//constructor
Westeros.Structures.Castle.prototype.Build =
function(){console.log("Castle built: " +  this.name)};
```

This class can be instantiated and used in much the same way as in previous examples:

```
var winterfell = new Westeros.Structures.Castle("Winterfell");
winterfell.Build();
```

Of course, with JavaScript, there is more than one way to build the same code structure. An easy way to structure the preceding code is to make use of the ability to create and immediately execute a function:

```
var Castle = (function () {
  function Castle(name) {
    this.name = name;
  }
  Castle.prototype.Build = function () {
    console.log("Castle built: " + this.name);
  };
  return Castle;
})();
Westros.Structures.Castle = Castle;
```

This code seems to be a bit longer than the previous code sample but I find it easier to follow due to its hierarchical nature. We can create a new `Castle` class using the same structure as the previous one:

```
var winterfell = new Westeros.Structures.Castle("Winterfell");
winterfell.Build();
```

Inheritance using this structure is also relatively easily done. If we were to define a `BaseStructure` class which was to be in the ancestor of all structures, then making use of it would look like this:

```
var BaseStructure = (function () {
  function BaseStructure() {
  }
  return BaseStructure;
})();
Structures.BaseStructure = BaseStructure;
var Castle = (function (_super) {
  __extends(Castle, _super);
  function Castle(name) {
    this.name = name;
    _super.call(this);
  }
```

```
    Castle.prototype.Build = function () {
      console.log("Castle built: " + this.name);
    };
    return Castle;
})(BaseStructure);
```

You'll note that the base structure is passed into the castle when the closure is evaluated. The highlighted line of code makes use of a helper method called `__extends`. This method is responsible for copying the functions over from the base prototype to the derived class. This particular piece of code was generated from a TypeScript compiler which also, helpfully, generated an `extends` method, which looks like this:

```
var __extends = this.__extends || function (d, b) {
  for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
  function __() { this.constructor = d; }
  __.prototype = b.prototype;
  d.prototype = new __();
};
```

We can continue the rather nifty closure syntax we've adopted for a class here to implement an entire module. This is shown in the following code:

```
var Westeros;
(function (Westeros) {
  (function (Structures) {
    var Castle = (function () {
      function Castle(name) {
        this.name = name;
      }
      Castle.prototype.Build = function () {
        console.log("Castle built " + this.name);
      };
      return Castle;
    })();
    Structures.Castle = Castle;
    })(Westeros.Structures || (Westeros.Structures = {}));
  var Structures = Westeros.Structures;
})(Westeros || (Westeros = {}));
```

Within this structure, you can see the same code for creating modules that we explored earlier. It is also relatively easy to define multiple classes inside a single module. This can be seen in the following code:

```
var Westeros;
(function (Westeros) {
    (function (Structures) {
        var Castle = (function () {
            function Castle(name) {
                this.name = name;
            }
            Castle.prototype.Build = function () {
                console.log("Castle built: " + this.name);
                var w = new Wall();
            };
            return Castle;
        })();
        Structures.Castle = Castle;
        var Wall = (function () {
            function Wall() {
                console.log("Wall constructed");
            }
            return Wall;
        })();
        Structures.Wall = Wall;
    })(Westeros.Structures || (Westeros.Structures = {}));
    var Structures = Westeros.Structures;
})(Westeros || (Westeros = {}));
```

The highlighted code creates a second class inside of the module. It is also perfectly permissible to define one class in each file. Because the code checks to get the current value of Westeros before blindly reassigning it, we can safely split the module definition across multiple files.

The last few lines of highlighted code expose the class outside of the closure. If we want to make private classes that are only available within the module, then we need to exclude only that line. This is actually known as the **revealing module pattern**. We only "reveal" the classes that need to be globally available. It is good practice to keep as much functionality out of the global accessible namespace as possible.

ECMAScript 6 classes and modules

We've seen so far that it is perfectly possible to build classes and even modules in JavaScript. The syntax is, obviously, a bit more involved than in a language such as C# or Java. Fortunately, the next version of JavaScript, ECMAScript 6 (also known as Harmony), brings support for some syntactic sugar for making classes:

```
class Castle extends Westeros.Structures.BaseStructure {
  constructor(name, allegience) {
    super(name);
    ...
  }

  Build() {
    ...
    super.Build();
  }
}
```

ECMAScript 6 also brings a well thought out module system for JavaScript. There is syntactic sugar for creating modules which looks like:

```
module 'Westeros' {
    export function Rule(rulerName, house) {
     ...
        return "Long live " + rulerName + " of house " + house;
    }
}
```

As modules can contain functions, they can, of course, contain classes. ECMAScript 6 also defines a module import syntax, and support for retrieving modules from remote locations. Importing a module looks like this:

```
import westeros from 'Westeros';
module JSON from 'http://json.org/modules/json2.js';
westeros.Rule("Rob Stark", "Stark");
```

Some of this syntactic sugar is available with current JavaScript, but it does require some additional tooling, which is outlined in *Chapter 12*, *ES6 Solutions Today*.

Best practices and troubleshooting

In an ideal world, everybody would get to work on greenfield projects, where they can put in standards right from the get go. However, that isn't the case. Frequently, you may find yourself in a situation where you have a bunch of non-modular JavaScript code as part of a legacy system.

In these situations, it may be advantageous to simply ignore the non-modular code until there is an actual need to upgrade it. Despite the popularity of JavaScript, much of the tooling for JavaScript is still immature, making it difficult to rely on a compiler to find errors introduced by JavaScript refactoring. Automatic refactoring tools are also complicated by the dynamic nature of JavaScript. However, for new code, proper use of modular JavaScript can be very helpful to avoid namespace conflicts and improve testability.

How to arrange JavaScript is an interesting question. From a web perspective, I have taken the approach of arranging my JavaScript in line with the web pages. So each page has an associated JavaScript file, which is responsible for the functionality of that page. In addition, components which are common between pages, say a grid control, are placed into a separate file. At compile time, all the files are combined into a single JavaScript file. This helps strike a balance between having small code files to work with, and reducing the number of requests to the server from the browser.

Summary

It has been said that there are only two really hard things in computing science. What those issues are varies depending on who is speaking. Frequently, it is some variation of cache invalidation and naming. How to organize your code is a large part of that naming problem.

As a group, we seem to have settled quite firmly on the idea of namespaces and classes. As we've seen, there is no direct support for either of these two concepts in JavaScript. However, there are a myriad of ways to work around the problem, some of which actually provide more power than one would get through a traditional namespace/class system.

The primary concern with JavaScript is to avoid polluting the global namespace with a large number of similarly named, unconnected objects. Encapsulating JavaScript into modules is a key step on the road to writing maintainable and reusable code.

As we move forward, we'll see that many of the patterns which are quite complex arrangements of interfaces become far simpler in the land of JavaScript. Prototype-based inheritance, which seems difficult at the onset, is a tremendous tool for aiding in the simplification of design patterns.

3
Creational Patterns

In the last chapter, we took a long look at how to fashion a class. In this chapter, we'll look at how to create instances of classes. On the surface it seems like a simple concern, but how we create instances of a class can be of great importance.

We take great pains in creating our code such that it can be as decoupled as much as possible. Ensuring that classes have minimal dependence on other classes is the key to building a system that can change fluently with the changing needs of those using the software. Allowing classes to be too closely related means that changes ripple through them.

One ripple isn't a huge problem but as you throw more and more changes into the mix, the ripples add up and create interference patterns. Soon the once placid surface is an unrecognizable mess of additive and destructive nodes. This problem also occurs in our applications: the changes magnify and interact in unexpected ways. One situation in which we tend to forget about coupling is when creating objects as can be seen in the following code:

```
var Westeros;
(function (Westeros) {
  var Ruler = (function () {
    function Ruler() {
      this.house = new Westeros.Houses.Targaryen();
    }
    return Ruler;
  })();
  Westeros.Ruler = Ruler;
})(Westeros || (Westeros = {}));
```

You can see in this class that the `Ruler` variable's house is strongly coupled to the `Targaryen` class. If this were ever to change, then this tight coupling would have to change in a great number of places. This chapter discusses a number of patterns, which were originally presented in the Gang of Four book, *Design Patterns: Elements of Reusable Object-Oriented Software*, *Addison-Wesley*. The goal of these patterns is to improve the degree of coupling in applications and increase the opportunities for code reuse.

The patterns are as follows:

- Abstract Factory
- Builder
- Factory Method
- Singleton
- Prototype

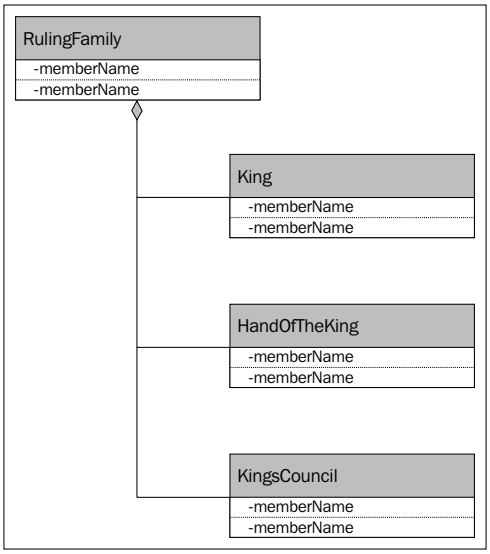
Of course not all of these are applicable to JavaScript, but we'll see all about that as we work through the creational patterns.

Abstract Factory

The first pattern presented here is a method to create kits of objects without knowing the concrete types of the objects. Let's continue on with the system presented earlier for ruling a kingdom.

In the kingdom in question, the ruling house changes with some degree of frequency. In all likelihood, there is a degree of battling and fighting during the change of house but we'll ignore that for the moment. Each house will rule the kingdom differently. Some value peace and tranquility, and rule as benevolent leaders, while others rule with an iron fist. The rule of a kingdom is too large for a single individual, so the king defers some of his decisions to a second-in-command known as the **hand of the king**. The king is also advised on matters by a council, which consists of some of the more savvy lords and ladies of the land.

A diagram of the classes in our description is as follows:

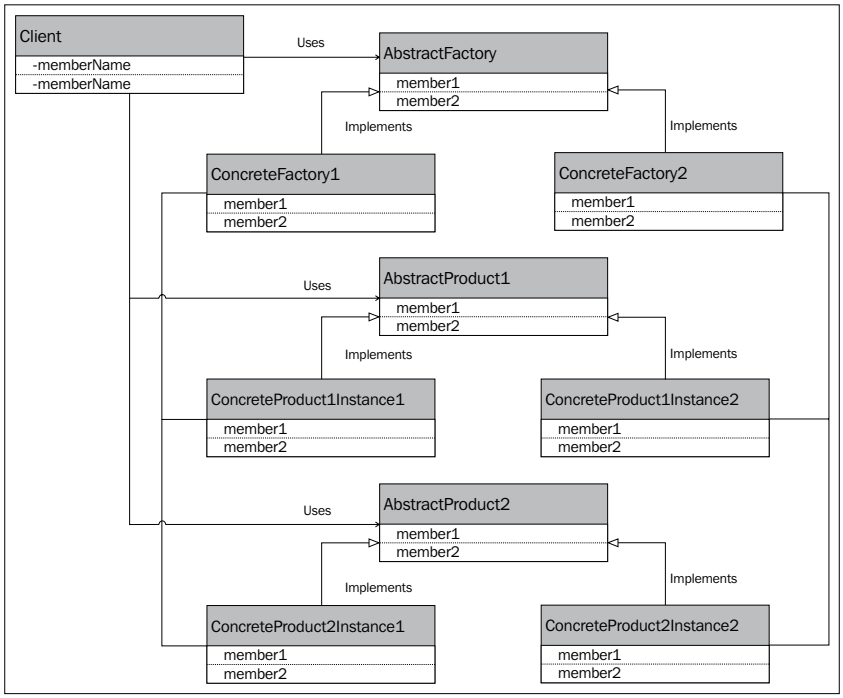


Unified Modeling Language (**UML**) is a standardized language developed by the **Object Management Group** (**OMG**), which describes computer systems. There is vocabulary in the language to create user interaction diagrams, sequence diagrams, and state machines, among others. For the purposes of this book, we're most interested in class diagrams, which describe the relationship between a set of classes.

The entire UML class diagram vocabulary is extensive and is beyond the scope of this book. However, the Wikipedia article at `https://en.wikipedia.org/wiki/Class_diagram` acts as a great introduction, as does Derek Banas's excellent video tutorial on class diagrams at `https://www.youtube.com/watch?v=3cmzqZzwNDM`.

The issue is that with the ruling family, and even the member of the ruling family on the throne changing so frequently, coupling to a concrete family such as `Targaryen` or `Lannister` makes our application brittle. Brittle applications do not fare well in an ever-changing world.

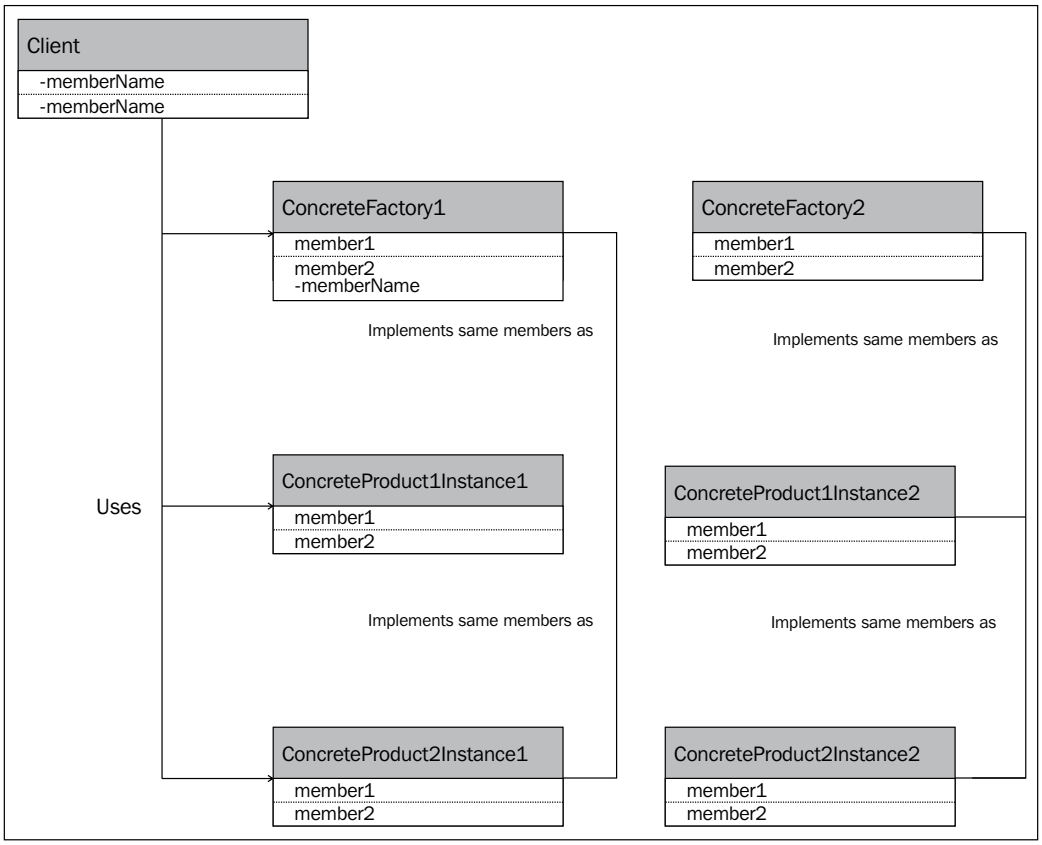
An approach to fixing this is to make use of the **Abstract Factory** pattern. The Abstract Factory pattern declares an interface to create each of the various classes related to the ruling family, as shown in the following diagram:



The Abstract Factory class may have multiple implementations for each of the various ruling families. These are known as **concrete factories** and each of them will implement the interface provided by the Abstract Factory. The concrete factories, in return, will return concrete implementations of the various ruling classes. These concrete classes are known as **products**.

Let's start by looking at the code for the interface for the Abstract Factory.

No code? Well, actually that is exactly the case. The lack of classes in JavaScript precludes the need for interfaces to describe classes. Instead of having interfaces, we'll move directly to creating the classes, as shown in the following diagram:



Instead of interfaces, JavaScript trusts that the class you provide implements all the appropriate methods. At runtime, the interpreter will attempt to call the method you request, and call it, if it is found. The interpreter simply assumes that if your class implements the method, then it is that class. This is known as **duck typing**.

Duck typing

The name duck typing originates a post made by Alex Martelli in the year 2000 to the news group comp.lang.python, where he wrote:

> *In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with.*

I enjoy the possibility that Martelli took the term from the witch hunt sketch from Monty Python's search for the Holy Grail. Although I can find no evidence of that, I find it quite likely as the Python programming language takes its name from Monty Python.

Duck typing is a powerful tool in dynamic languages allowing, for much less overhead in implementing a class hierarchy. It does, however, introduce some uncertainty. If two classes implement an identically named method that have radically different meanings, then there is no way to know whether the one being called is the correct one. For example, consider the following code:

```
class Boxer{
  function punch(){}
}
class TicketMachine{
  function punch(){}
}
```

Both classes have a `punch()` method but they clearly have different meanings. The JavaScript interpreter has no idea that they are different classes and will happily call `punch` on either class, even when one doesn't make sense.

In some dynamic languages, there is support for the generic method, which is called whenever an undefined method is called. Ruby, for instance, has `missing_method`, which has proven to be very useful in a number of scenarios. As of this writing, there is no support for `missing_method` in JavaScript. However, it may be possible to implement such a feature in ECMAScript 6.

Implementation

To demonstrate an implementation of the Abstract Factory pattern, the first thing we'll need is an implementation of the `King` class. The following code provides that implementation:

```
var KingJoffery= (function () {
  function KingJoffery() {
  }
  KingJoffery.prototype.makeDecision = function () {
    …
  };
  KingJoffery.prototype.marry = function () {
    …
  };
  return KingJoffery;
})();
```

This code does not include the module structure suggested in *Chapter 2, Organizing Code*. Including the boiler-plate module code in every example is tedious and you are all smart cookies, so you know to put this in modules if you're going to actually use it. The fully modularized code is available in the distributed source code.

This is just a regular concrete class and could really contain any implementation details. Similarly, we'll need an implementation of the *HandOfTheKing* class that is equally unexciting:

```
var LordTywin = (function () {
  function LordTywin() {
  }
  LordTywin.prototype.makeDecision = function () {
  };
  return LordTywin;
})();
```

The concrete factory method looks like this:

```
var LannisterFactory = (function () {
  function LannisterFactory() {
  }
  LannisterFactory.prototype.getKing = function () {
    return new KingJoffery();
```

```
    };
    LannisterFactory.prototype.getHandOfTheKing = function ()
    {
      return new LordTywin();
    };
    return LannisterFactory;
  })();
```

The preceding code simply instantiates new instances of each of the required classes and returns them. An alternative implementation for a different ruling family would follow the same general form and might look like the following code:

```
var TargaryenFactory = (function () {
  function TargaryenFactory() {
  }
  TargaryenFactory.prototype.getKing = function () {
    return new KingAerys();
  };
  TargaryenFactory.prototype.getHandOfTheKing = function () {
    return new LordConnington();
  };
  return TargaryenFactory;
})();
```

The implementation of the Abstract Factory pattern in JavaScript is much easier than in other languages. However, the penalty for this is that you lose the compiler checks, which force a full implementation of either the factory or the products. As we proceed through the rest of the patterns, you'll notice that this is a common theme. Patterns that have a great deal of plumbing in statically typed languages are far simpler but create a greater risk of runtime failure. Appropriate unit tests or a JavaScript compiler can ameliorate this situation.

To make use of the Abstract Factory pattern, we'll first need a class that requires the use of some ruling family. The following is the code for this class:

```
var CourtSession = (function () {
  function CourtSession(abstractFactory) {
    this.abstractFactory = abstractFactory;
    this.COMPLAINT_THRESHOLD = 10;
  }
  CourtSession.prototype.complaintPresented = function (complaint)
  {
    if (complaint.severity < this.COMPLAINT_THRESHOLD) {
```

```
        this.abstractFactory.getHandOfTheKing().makeDecision();
      } else
        this.abstractFactory.getKing().makeDecision();
      };
    return CourtSession;
})();
```

We can now call the `CourtSession` class and inject different functionality depending on which factory we pass in:

```
var courtSession1 = new CourtSession(new TargaryenFactory());
courtSession1.complaintPresented({ severity: 8 });
courtSession1.complaintPresented({ severity: 12 });

var courtSession2 = new CourtSession(new LannisterFactory());
courtSession2.complaintPresented({ severity: 8 });
courtSession2.complaintPresented({ severity: 12 });
```

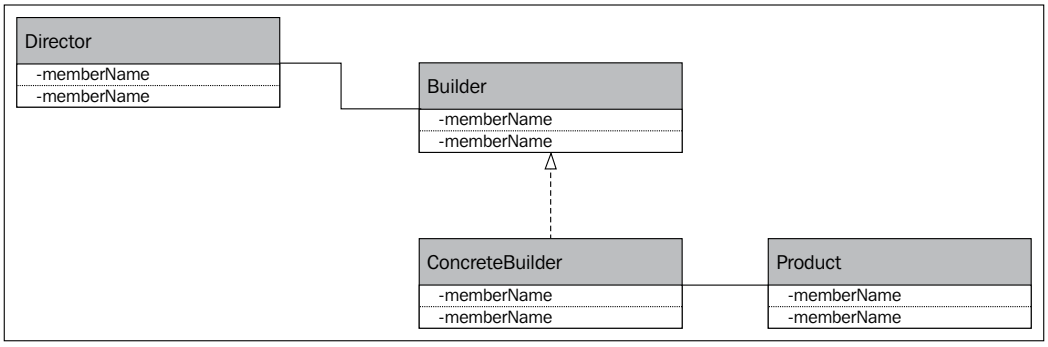
Despite the differences between a static language and JavaScript, this pattern remains applicable and useful in JavaScript applications. Creating a kit of objects, that work together is useful in a number of situations: any time when a group of objects needs to collaborate to provide functionality but may need to be replaced wholesale. It may also be a useful pattern when attempting to ensure that a set of objects be used together without substitutions.

Builder

In our fictional world, we sometimes have some rather complicated classes that need to be constructed. The classes contain different implementations of an interface depending on how they are constructed. In order to simplify the building of these classes and encapsulate the knowledge of building the class away from the consumers, a **builder** may be used. Multiple concrete builders reduce the complexity of the constructor in the implementation. When new builders are required, a constructor does not need to be added, a new builder just needs to be plugged in.

Tournaments are an example of a complicated class. Each tournament has a complicated setup involving the events, the attendees, and the prizes. Much of the setup for these tournaments is similar: each one has a joust, archery, and a melee. Creating a tournament from multiple places in the code means that the responsibility of knowing how to construct a tournament is distributed. If there is a need to change the initiation code, then it must be done in a lot of different places.

Employing a Builder pattern avoids this issue by centralizing the logic necessary to build the object. Different concrete builders can be plugged into the builder to construct different complicated objects, as shown in the following diagram:



Implementation

Let's drop in and look at some of the code. To start with, we'll create a number of utility classes, which will represent the parts of a tournament. We can see this in the following code:

```
var Event = (function () {
  function Event(name) {
    this.name = name;
  }
  return Event;
})();
Westeros.Event = Event;

var Prize = (function () {
  function Prize(name) {
    this.name = name;
  }
  return Prize;
})();
Westeros.Prize = Prize;

var Attendee = (function () {
  function Attendee(name) {
    this.name = name;
  }
  return Attendee;
})();
Westeros.Attendee = Attendee;
```

The tournament itself is a very simple class as we don't need to assign any of the public properties explicitly, as shown in the following code:

```
var Tournament = (function () {
  this.Events = [];
  function Tournament() {
  }
  return Tournament;
})();
Westeros.Tournament = Tournament;
```

We'll implement two builders that create different tournaments. This can be seen in the following code:

```
var LannisterTournamentBuilder     = (function () {
  function LannisterTournamentBuilder() {
  }
  LannisterTournamentBuilder.prototype.build = function () {
    var tournament = new Tournament();
    tournament.events.push(new Event("Joust"));
    tournament.events.push(new Event("Melee"));

    tournament.attendees.push(new Attendee("Jamie"));

    tournament.prizes.push(new Prize("Gold"));
    tournament.prizes.push(new Prize("More Gold"));

    return tournament;
  };
  return LannisterTournamentBuilder;
})();
Westeros.LannisterTournamentBuilder = LannisterTournamentBuilder;

var BaratheonTournamentBuilder = (function () {
  function BaratheonTournamentBuilder() {
  }
  BaratheonTournamentBuilder.prototype.build = function () {
    var tournament = new Tournament();
    tournament.events.push(new Event("Joust"));
```

```
        tournament.events.push(new Event("Melee"));

        tournament.attendees.push(new Attendee("Stannis"));
        tournament.attendees.push(new Attendee("Robert"));

        return tournament;
      };
      return BaratheonTournamentBuilder;
    })();
    Westeros.BaratheonTournamentBuilder = BaratheonTournamentBuilder;
```

Finally, the director, or as we're calling it `TournamentBuilder`, simply takes a builder and executes it:

```
    var TournamentBuilder = (function () {
      function TournamentBuilder() {
      }
      TournamentBuilder.prototype.build = function (builder) {
        return builder.build();
      };
      return TournamentBuilder;
    })();
    Westeros.TournamentBuilder = TournamentBuilder;
```

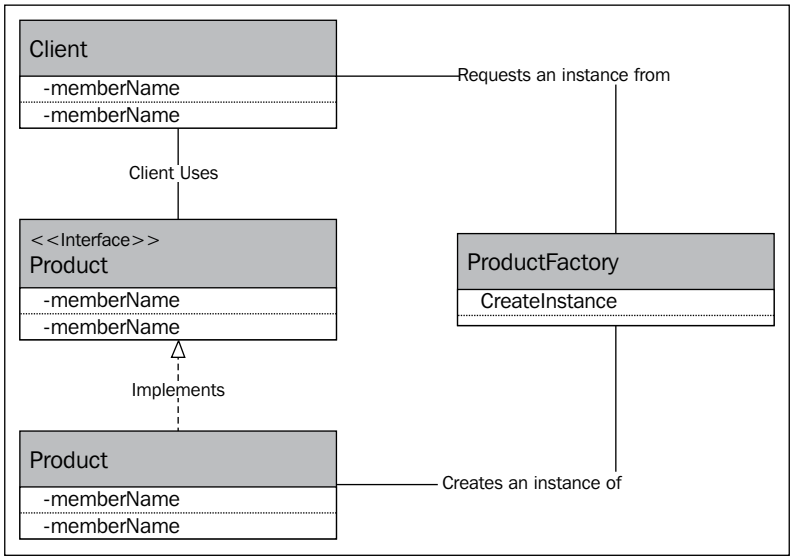
Again, you'll see that the JavaScript implementation is far simpler than the traditional implementation as there is no need for interfaces.

Builders need not return a fully realized object. This means that you can create a builder that partially hydrates an object, then allows the object to be passed onto another builder for it to finish. This approach allows us to divide the work of building an object amongst several classes with limited responsibility. In our preceding example, we could have a builder that is responsible for populating the events and another that is responsible for populating the attendees.

Does the builder pattern still make sense in view of JavaScript's prototype extension model? I believe so. There are still cases where a complicated object needs to be created according to different approaches.

Factory Method

We've already looked at the Abstract Factory and Builder patterns. The Abstract Factory pattern builds a family of related classes and the builder creates complicated objects using different strategies. The Factory Method pattern allows a class to request a new instance of an interface without the class making decisions about which implementation of the interface to use. The factory may use some strategy to select which implementation to return. This is shown in the following diagram:



Sometimes, this strategy is simply to take a string parameter or to examine some global setting to act as a switch.

Implementation

In our example world of `Westeros`, there are plenty of times when we would like to defer the choice of implementation to a factory. Just like the real world, Westeros has a vibrant religious culture with dozens of competing religions worshiping a wide variety of gods. When praying in each religion, different rules must be followed. Some religions demand sacrifices while others demand only that a gift be given. The prayer class doesn't want to know about all the different religions and how to construct them.

Let's start with creating a number of different gods to which prayers can be offered. The following code creates three gods, including a default god to whom prayers fall if no other god is specified:

```
var WateryGod = (function () {
  function WateryGod() {
  }
  WateryGod.prototype.prayTo = function () {
  };
  return WateryGod;
})();
Religion.WateryGod = WateryGod;
var AncientGods = (function () {
  function AncientGods() {
  }
  AncientGods.prototype.prayTo = function () {
  };
  return AncientGods;
})();
Religion.AncientGods = AncientGods;

var DefaultGod = (function () {
  function DefaultGod() {
  }
  DefaultGod.prototype.prayTo = function () {
  };
  return DefaultGod;
})();
Religion.DefaultGod = DefaultGod;
```

I've avoided any sort of implementation details for each god. You may imagine whatever traditions you want to populate the `prayTo` methods. There is also no need to ensure that each of the gods implements an `IGod` interface. Next, we'll need a factory that is responsible for constructing each of the different gods, as shown in the following code:

```
var GodFactory = (function () {
  function GodFactory() {
  }
  GodFactory.Build = function (godName) {
    if (godName === "watery")
```

```
    return new WateryGod();
    if (godName === "ancient")
    return new AncientGods();
    return new DefaultGod();
  };
  return GodFactory;
})();
```

You can see that, in this example, we're taking in a simple string to decide how to create a god. It could be done via a global or via a more complicated object. In some polytheistic religions in Westeros, gods have defined roles as gods of courage, beauty, or some other aspect. The god to which one must pray is determined by not just the religion but the purpose of the prayer. We can represent this with a `GodDeterminant` class, as shown in the following code:

```
var GodDeterminant = (function () {
  function GodDeterminant(religionName, prayerPurpose) {
    this.religionName = religionName;
    this.prayerPurpose = prayerPurpose;
  }
  return GodDeterminant;
})();
```

The factory would be updated to take this class instead of the simple string.

Finally, the last step is to see how this factory would be used. It is quite simple; we just need to pass in a string that denotes which religion we wish to observe and the factory will construct the correct god and return it. The following code demonstrates how to call the factory:

```
var Prayer = (function () {
  function Prayer() {
  }
  Prayer.prototype.pray = function (godName) {
    GodFactory.Build(godName).prayTo();
  };
  return Prayer;
})();
```

Once again, there is certainly need for a pattern such as this in JavaScript. There are plenty of times when separating the instantiation from the use is useful. Testing the instantiation is also very simple thanks to the separation of concerns, and the ability to inject a fake factory to allow testing of the `Prayer` class is also easy.

Continuing the trend of creating simpler patterns without interfaces, we can ignore the interface portion of the pattern and work directly with the types, thanks to duck typing.

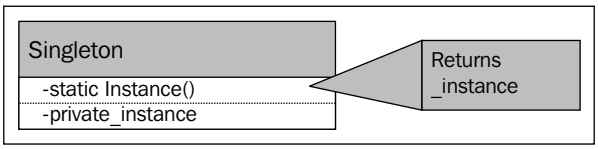
Factory Method is a very useful pattern; it allows classes to defer the selection of the implementation of an instantiation to another class. This pattern is very useful when there are multiple similar implementations such as the strategy pattern (see *Chapter 5*, *Behavioral Patterns*), and is commonly used in conjunction with the Abstract Factory pattern. The Factory Method pattern is used to build the concrete objects within a concrete implementation of the Abstract Factory. An Abstract Factory may contain a number of factory methods. Factory Method is certainly a pattern that remains applicable in the field of JavaScript.

Singleton

The Singleton pattern is perhaps the most overused pattern. It is also a pattern that has fallen out of favor in recent years. To see why people are starting to advise against using Singleton, let's take a look at how the pattern works.

Singleton is used when a global variable is desirable, but Singleton provides protection against accidentally creating multiple copies of a complex object. It also allows for the deferral of object instantiation until the first use.

The UML diagram for Singleton is as follows:



It is clearly a very simple pattern. The Singleton pattern acts as a wrapper around an instance of the class and the Singleton itself lives as a global variable. When accessing the instance, we simply ask Singleton for the current instance of the wrapped class. If the class does not yet exist within the Singleton, it is common to create a new instance at that time.

Implementation

Within our ongoing example in the world of Westeros, we need to find a case where there can only ever be one of an item. Unfortunately, it is a land with frequent conflicts and rivalries, and so my first idea of using the king as the Singleton pattern is simply not going to fly. This split also means that we cannot make use of any of the other obvious candidates (capital city, queen, general…) as there may be many instances of each of those too. However, in the far north of Westeros, there is a giant wall constructed to keep an ancient enemy at bay. There is only one of these walls and it should pose no issue having it in the global scope.

Let's go ahead and create a Singleton class in JavaScript:

```
var Westeros;
(function (Westeros) {
  var Wall = (function () {
    function Wall() {
      this.height = 0;
      if (Wall._instance)
        return Wall._instance;
      Wall._instance = this;
    }
    Wall.prototype.setHeight = function (height) {
      this.height = height;
    };
    Wall.prototype.getStatus = function () {
      console.log("Wall is " + this.height + " meters tall");
    };
    Wall.getInstance = function () {
      if (!Wall._instance) {
        Wall._instance = new Wall();
      }
      return Wall._instance;
    };
    Wall._instance = null;
    return Wall;
  })();
  Westeros.Wall = Wall;
})(Westeros || (Westeros = {}));
```

The code creates a lightweight representation of the `Wall` class. The Singleton pattern is demonstrated in the two highlighted sections. In a language like C# or Java, we would normally just set the constructor to be private so that it could only be called by the static `getInstance` method. However, we don't have that ability in JavaScript: constructors cannot be private. Thus, we do the best we can and return the current instance from the constructor. This may appear strange, but in the way we've constructed our classes the constructor is no different from any other method, so it is possible to return something from it.

In the second highlighted section, we set a static variable, `_instance`, to be a new instance of `Wall` when one is not already there. In the case that `_instance` already exists, we return that. In C# and Java, there will be a need for some complicated locking logic in this function to avoid race conditions as two different threads attempt to access the instance at the same time. Fortunately, there is no need to worry about this in JavaScript, where the multithreading story is different.

Disadvantages

Singletons have gained a somewhat bad reputation in the last few years. They are, in effect, glorified global variables. As we've discussed, global variables are ill conceived and the potential cause of numerous bugs. They are also difficult to test with unit tests, as the creation of the instance cannot easily be overridden. The single largest concern I have with them is that singletons have too much responsibility. They control not just themselves but also their instantiation. This is a clear violation of the single responsibility principle. Almost every problem that can be solved by using a Singleton pattern is better solved using some other mechanism.

JavaScript makes the problem even worse. It isn't possible to create a clean implementation of the Singleton pattern due to the restrictions on the constructor. This, coupled with the general problems around the Singleton pattern, lead me to suggest that this pattern should be avoided in JavaScript.

Prototype

The final creational pattern in this chapter is the Prototype pattern. Perhaps this name sounds familiar. It certainly should: it is the mechanism through which JavaScript inheritance is supported.

We looked at prototypes for inheritance but the applicability of prototypes need not be limited to inheritance. Copying existing objects can be a very useful pattern. There are numerous cases when being able to duplicate a constructed object is handy. For instance, maintaining a history of the state of an object is easily done by saving previous instances created by leveraging some sort of cloning.

Implementation

In Westeros, we find that members of a family are frequently very similar: as the adage goes, "like father, like son." As each generation is born, it is easier to create the new generation by copying and modifying an existing family member than to build one from scratch.

In *Chapter 2*, *Organizing Code*, we looked at how to copy existing objects and presented a very simple piece of code for cloning:

```
function clone(source, destination) {
  for(var attr in source.prototype){
    destination.prototype[attr] = source.prototype[attr];}
  }
```

The following code can easily be altered to be used inside a class to return a copy of itself:

```
var Westeros;
(function (Westeros) {
  (function (Families) {
    var Lannister = (function () {
      function Lannister() {
      }
      Lannister.prototype.clone = function () {
        var clone = new Lannister();
        for (var attr in this) {
          clone[attr] = this[attr];
        }
        return clone;
      };
      return Lannister;
    })();
    Families.Lannister = Lannister;
  })(Westeros.Families || (Westeros.Families = {}));
  var Families = Westeros.Families;
})(Westeros || (Westeros = {}));
```

The highlighted section of the preceding code is the modified `clone` method. It can be used as follows:

```
var jamie = new Westeros.Families.Lannister();
jamie.swordSkills = 9;
jamie.charm = 6;
jamie.wealth = 10;

var tyrion = jamie.clone();
tyrion.charm = 10;
//tyrion.wealth == 10
//tyrion.swordSkill == 9
```

The Prototype pattern allows for a complex object to be constructed only once and is then cloned into any number of objects that vary only slightly. If the source object is not complicated, there is little to be gained from taking a cloning approach. Care must be taken when using the prototype approach to think about dependent objects. Should the clone be a deep one?

Prototype is obviously a useful pattern and one that forms an integral part of JavaScript from the get go. As such, it is certainly a pattern that will see some use in any JavaScript application of appreciable size.

Hints and tips

Creational patterns allow for specialized behavior in creating objects. In many cases, such as the factory, they provide extension points into which crosscutting logic can be placed. This means the logic applies to a number of different types of objects. If you're looking for a way to inject, say, logging throughout your application, then being able to hook into a factory is of great utility.

Despite the utility of these creational patterns, they should not be used very frequently. The vast majority of your object instantiations should still be just the normal method of newing up objects. Although it is tempting to treat everything as a nail when you've got a new hammer, the truth is that each situation needs to have a specific strategy. All these patterns are more complicated than simply using `new`, and complicated code is more liable to have bugs than simple code. Use `new` whenever possible.

Summary

This chapter presented a number of different strategies for creating objects. These methods provide abstractions over the typical methods for creating objects. The Abstract Factory pattern provides for a method to build interchangeable kits or collections of related objects. The Builder pattern provides for a solution to the telescoping parameters issue. It makes the construction of large complicated objects easier. The Factory Method pattern, which is a useful complement to Abstract Factory, allows for different implementations to be created though a static factory. Singleton is a pattern to provide a single copy of a class that is available to the entire solution. It is the only pattern we've seen so far that has presented some questions around applicability in modern software. The Prototype pattern is a commonly used pattern in JavaScript to build objects based on other existing objects.

We'll continue our examination of classical design patterns in the next chapter by looking at structural patterns.

4
Structural Patterns

In the previous chapter, we looked at a number of ways to create objects in order to optimize, for reuse. In this chapter, we'll take a look at structural patterns; these are patterns that are concerned with easing the design by describing simple ways in which objects can interact.

Again, we will limit ourselves to the patterns described in the GoF book. There are a number of other interesting structural patterns that have been identified since the publication of the GoF, and we'll look at those in *Part 2* of the book.

The patterns we'll examine in this chapter are:

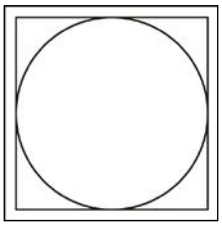
- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Once again, we'll discuss if the patterns that were described years ago are still relevant for a different language and a different time.

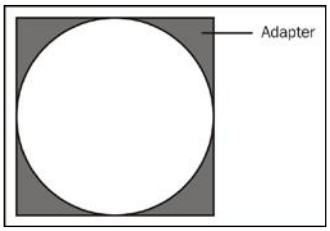
Adapter

From time to time, we will need to fit a round peg in a square hole. If you've ever played with a child's shape sorting toy, then you may have discovered that you can, in fact, put a round peg in a square hole. The hole is not completely filled and getting the peg in there can be difficult.

This is shown in the following diagram:

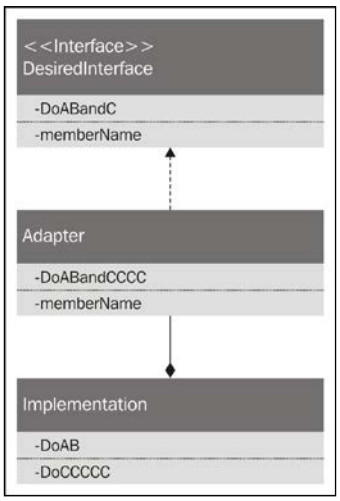


To improve the fit of the peg, an adapter can be used. This adapter fills the hole in completely resulting in a perfect fit, as shown in the following diagram:



In software, a similar approach is often needed. We may need to make use of a class that does not perfectly fit the required interface. The class may be missing methods or may have additional methods we would like to hide. This occurs frequently when dealing with third-party code. In order to make it comply with the interface needed in your code, an adapter may be required.

The class diagram for an adapter is very simple as follows:



The interface of the implementation does not look the way we would like it to for use in our code. Normally, the solution to this is to simply refactor the implementation so it looks the way we would like it to. However, there are a number of possible reasons that this cannot be done: perhaps the implementation exists inside third-party code to which we have no access. It is also possible that the implementation is used elsewhere in the application, where the interface is exactly as we would like it to be.

The adapter class is a thin piece of code that implements the required interface. It typically wraps a private copy of the implementation class and proxies calls through to it. Frequently, the adapter pattern is used to change the abstraction level of the code. Let's take a look at a quick example.

Implementation

In the land of Westeros, much of the trade and travel is done by boat. It is more dangerous to travel by ship than to walk or travel by horse, but also riskier due to the constant presence of storms and pirates. These ships are not the sort which might be used by Royal Caribbean to cruise around the Caribbean; they are crude animals which might look more at home captained by the 15th century European explorers.

While I am aware that ships exist, I have very little knowledge of how they work or how I might go about navigating one. I imagine that many people are in the same, (*cough!*), boat as me. If we look at the interface for `Ship` in `Westeros`, it looks intimidating:

```
interface Ship{
  SetRudderAngleTo(angle: number);
  SetSailConfiguration(configuration: SailConfiguration);
  SetSailAngle(sailId: number, sailAngle: number);
  GetCurrentBearing(): number;
  GetCurrentSpeedEstimate(): number;
  ShiftCrewWeightTo(weightToShift: number, locationId: number);
}
```

I would really like a much simpler interface that abstracts away all the fiddly little details. Ideally, something like the following code:

```
interface SimpleShip{
  TurnLeft();
  TurnRight();
  GoForward();
}
```

This looks like something I could probably figure out even living in a city that is over 1,000 kilometers from the nearest ocean. In short, what I'm looking for is a higher-level abstraction around the `Ship` interface. In order to transform a ship into a simple ship, we need an adapter.

The adapter will have the interface of `SimpleShip`, but it will perform actions on a wrapped instance of `Ship`. The code might look similar to the following:

```
var ShipAdapter = (function () {
  function ShipAdapter() {
    this._ship = new Ship();
  }
  ShipAdapter.prototype.TurnLeft = function () {
    this._ship.SetRudderAngleTo(-30);
    this._ship.SetSailAngle(3, 12);
  };
  ShipAdapter.prototype.TurnRight = function () {
    this._ship.SetRudderAngleTo(30);
    this._ship.SetSailAngle(5, -9);
  };
  ShipAdapter.prototype.GoForward = function () {
    //do something else to the _ship
  };
  return ShipAdapter;
})();
```

In reality, these functions would be far more complex but it should not matter much, because we've got a nice simple interface to present to the world. The presented interface can also be set up so as to restrict access to certain methods on the underlying type. When building library code, adapters can be used to mask the internal method and only present the limited functions needed by the end user.

To use this pattern, the code might look like the following:

```
var ship = new ShipAdapter();
ship.GoForward();
ship.TurnLeft();
```

You would likely not want to use "adapter" in the name of your client class as it leaks some information about the underlying implementation. Clients should be unaware that they are talking to an adapter.

The adapter itself can grow to be quite complex to adjust one interface into another. Care must be taken in order to avoid creating very complex adapters. It is certainly not inconceivable to build several adapters, one atop another. If you find an adapter becoming too large, then it is a good idea to stop and examine if the adapter is following the single responsibility principle. That is to say, ensure that each class has only one thing for which it has some responsibility. A class that looks up users from a database should itself not contain functionality to send e-mails to these users. That is too much responsibility. Complex adapters can be replaced with a composite object that will be explored later in this chapter.

From the testing perspective, adapters can be used to totally wrap third-party dependencies. In this scenario, they provide a place into which to hook tests. Unit tests should avoid testing libraries, but they can certainly test the adapters to ensure that they are proxying through the correct calls.

The adapter is a very powerful pattern to simplify code interfaces. Massaging interfaces to better match a requirement is useful in countless places. The pattern is certainly useful in JavaScript. The applications I write in JavaScript tend to make use of a large number of small libraries. By wrapping up these libraries in adapters, I'm able to limit the number of places I interact with the libraries directly; this means that the libraries can easily be replaced.

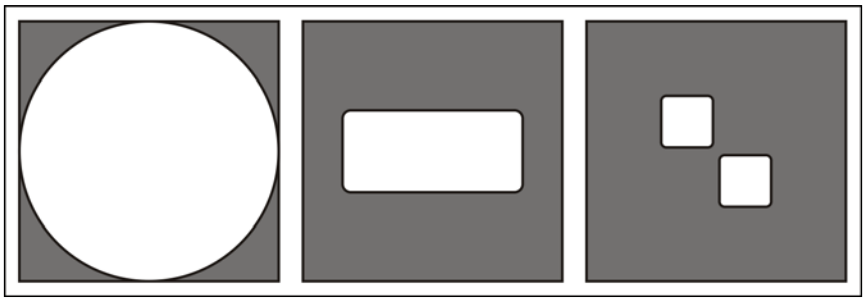
The adapter pattern can be slightly modified to provide consistent interfaces over a number of different implementations. This is usually known as the **bridge pattern**.

Bridge

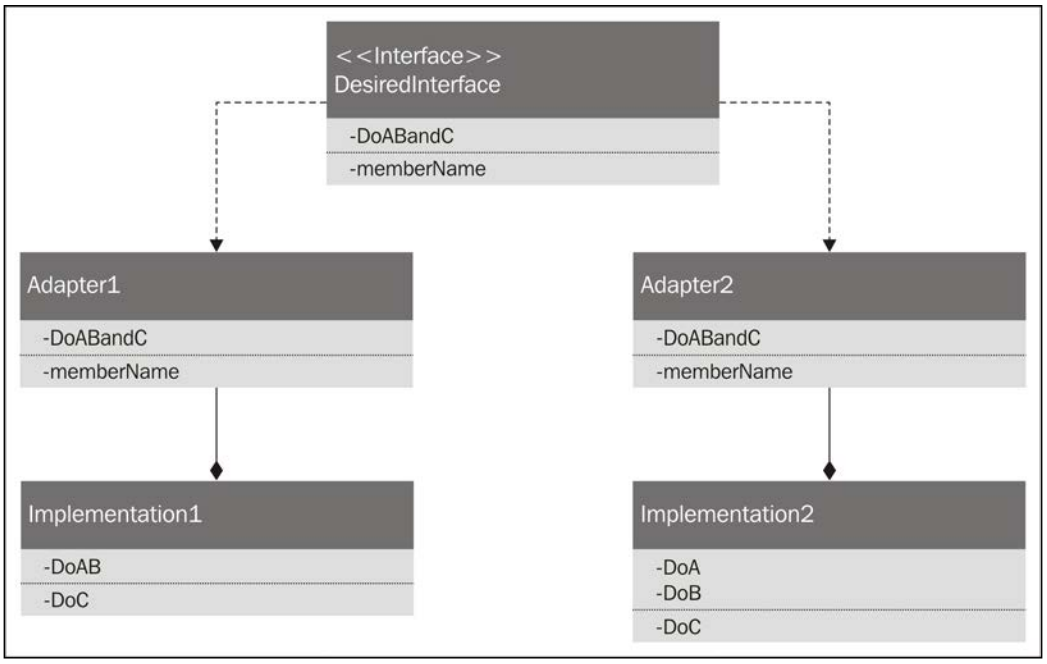
The bridge pattern takes the adapter pattern to a new level. Given an interface, we can build multiple adapters, each one of which acts as an intermediary to a different implementation.

An excellent example across which I've run, is dealing with two different services that provide more or less the same functionality and are used in a failover configuration. Neither service provides exactly the interface required by the application and both services provide different APIs. In order to simplify the code, adapters are written to provide a consistent interface. The adapters implement a consistent interface and provide fills so that each API can be called consistently. To expand on the shape sorter metaphor a bit more, we can imagine that we have a variety of different pegs we would like to use to fill the square hole.

Each adapter fills in the missing bits and helps us get a good fit.



The bridge is a very useful pattern. Let's take a look at how to implement it:



The adapters shown in the preceding diagram sit between the implementation and the desired interface. They modify the implementation to fit in with the desired interface.

Implementation

We've already discussed that in the land of Westeros, the people practice a number of disparate religions. Each one has a different way of praying and making offerings. There is a lot of complexity around making the correct prayers at the correct time and we would like to avoid exposing this complexity. Instead, we'll write a series of adapters that can simplify prayers.

The first thing we need is a number of different gods to which we can pray. For this, we have the following code:

```
var OldGods = (function () {
  function OldGods() {
  }
  OldGods.prototype.prayTo = function (sacrifice) {
    console.log("We Old Gods hear your prayer");
  };
  return OldGods;
})();
Religion.OldGods = OldGods;

var DrownedGod = (function () {
  function DrownedGod() {
  }
  DrownedGod.prototype.prayTo = function (humanSacrifice) {
    console.log("*BUBBLE* GURGLE");
  };
  return DrownedGod;
})();
Religion.DrownedGod = DrownedGod;

var SevenGods = (function () {
  function SevenGods() {
  }
  SevenGods.prototype.prayTo = function (prayerPurpose) {
    console.log("Sorry there are a lot of us, it gets confusing
    here.  Did you pray for something?");
  };
  return SevenGods;
})();
Religion.SevenGods = SevenGods;
```

These classes should look familiar, as they are basically the same classes that we found in the previous chapter, where they were used as examples for the Factory Method pattern. You may notice, however, that the signature for the `prayTo` method for each religion is slightly different. This proves to be something of an issue when building a consistent interface, like the one shown in the following pseudo code:

```
interface God
{
  prayTo():void;
}
```

So, let's slot in a few adapters to act as a bridge between the classes we have and the signature we would like:

```
var OldGodsAdapter = (function () {
  function OldGodsAdapter() {
    this._oldGods = new OldGods();
  }
  OldGodsAdapter.prototype.prayTo = function () {
    var sacrifice = new Sacrifice();
    this._oldGods.prayTo(sacrifice);
  };
  return OldGodsAdapter;
})();
Religion.OldGodsAdapter = OldGodsAdapter;

var DrownedGodAdapter = (function () {
  function DrownedGodAdapter() {
    this._drownedGod = new DrownedGod();
  }
  DrownedGodAdapter.prototype.prayTo = function () {
    var sacrifice = new HumanSacrifice();
    this._drownedGod.prayTo(sacrifice);
  };
  return DrownedGodAdapter;
})();
Religion.DrownedGodAdapter = DrownedGodAdapter;

var SevenGodsAdapter = (function () {
  function SevenGodsAdapter() {
    this.prayerPurposeProvider = new PrayerPurposeProvider();
    this._sevenGods = new SevenGods();
  }
```

```
  SevenGodsAdapter.prototype.prayTo = function () {
    this._sevenGods.prayTo(this.prayerPurposeProvider.
    GetPurpose());
  };
  return SevenGodsAdapter;
})();
Religion.SevenGodsAdapter = SevenGodsAdapter;
```

Each one of these adapters implements the `God` interface we wanted and abstracts away the complexity of dealing with three different interfaces: one for each God.

To use the bridge pattern, we could write the following code:

```
var god1 = new Religion.SevenGodsAdapter();
var god2 = new Religion.DrownedGodAdapter();
var god3 = new Religion.OldGodsAdapter();

var gods = [god1, god2, god3];
for(var i =0; i<gods.length; i++){
  gods[i].prayTo();
}
```

This code uses the bridges to provide a consistent interface to the gods so that they can all be treated as equals.

In this case, we are simply wrapping the individual gods and proxying method calls through to them. The adapters could each wrap a number of objects and this is another useful place in which to use the adapter. If a complex series of objects needs to be orchestrated, then an adapter can take some responsibility for that orchestration, providing a simpler interface to other classes.

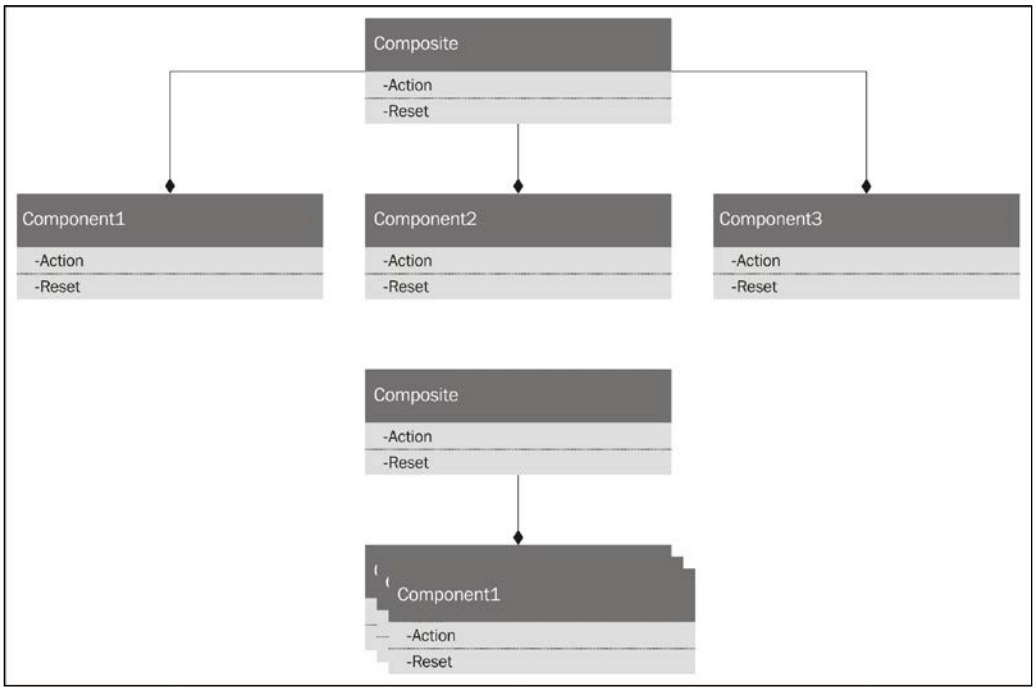
You can imagine how useful the bridge pattern is. It can be used well in conjunction with the Factory Method pattern presented in the previous chapter.

This pattern certainly remains a very useful one for use in JavaScript. As I mentioned at the start of this section, it is handy to deal with different APIs in a consistent fashion. I have used it to swap in different third-party components such as different graphing libraries or phone system integration points. If you're building applications on a mobile platform using JavaScript, then the bridge pattern is going to be a great friend for you, allowing you to separate your common and platform-specific code cleanly. Because there are no interfaces in JavaScript, the bridge pattern is far closer to the adapter in JavaScript than in other languages. In fact, it is basically, exactly the same.

Bridge also makes testing easier. We are able to implement a fake bridge and use this to ensure that the calls to the bridge are correct.

Composite

In the previous chapter, I mentioned that we would like to avoid coupling our objects together tightly. Inheritance is a very strong form of coupling and I suggested that, instead, composites should be used. The composite pattern is a special case of this in which the composite is treated as interchangeable with the components. Let's explore how the composite pattern works:



The preceding class diagram contains two different ways to build a composite. In the first one, the composite component is built from a fixed number of a variety of components. The second component is constructed from a collection of indeterminate length. In both cases, the components contained within the parent composition could be of the same type as the composition. So a composition may contain instances of its own type.

The key feature of the composite pattern is the interchangeability of a component with its children. So if we have a composite which implements `IComponent`, then all of the components of the composite will also implement `IComponent`. This is, perhaps, best illustrated with an example.

An example

Tree structures are very useful in computing. It turns out that a hierarchical tree can represent many things. A tree is made up of a series of nodes and edges and is acyclical. In a binary tree, each node contains a left and a right child until we get down to the terminal nodes known as **leaves**.

While life is difficult in Westeros, there is opportunity for taking joy in things like religious holidays or weddings. At these events, there is typically a great deal of feasting on delicious food. The recipes for this food is much as you would find in your own set of recipes. A simple dish like baked apples contains a list of ingredients:

- Baking apples
- Honey
- Butter
- Nuts

Each one of these ingredients implements an interface which we'll refer to as `Ingredient`. More complex recipes contain more ingredients but in addition to that, more complex recipes may contain complex ingredients that are themselves made from other ingredients.

A popular dish in a southern part of Westeros is a dessert which is not at all unlike what we would call tiramisu. It is a complex recipe with the following:

- Custard
- Cake
- Whipped cream
- Coffee

Of course, custard itself is made from:

- Milk
- Sugar
- Eggs
- Vanilla

Custard is a composite as are coffee and cake.

Operations on the composite object are typically proxied through to all of the contained objects.

Implementation

A simple ingredient, one which would be a leaf node, is shown in the following code:

```
var SimpleIngredient = (function () {
  function SimpleIngredient(name, calories, ironContent,
  vitaminCContent) {
    this.name = name;
    this.calories = calories;
    this.ironContent = ironContent;
    this.vitaminCContent = vitaminCContent;
  }
  SimpleIngredient.prototype.GetName = function () {
    return this.name;
  };
  SimpleIngredient.prototype.GetCalories = function () {
    return this.calories;
  };
  SimpleIngredient.prototype.GetIronContent = function () {
    return this.ironContent;
  };
  SimpleIngredient.prototype.GetVitaminCContent = function () {
    return this.vitaminCContent;
  };
  return SimpleIngredient;
})();
```

It can be used interchangeably with a compound ingredient which has a list of ingredients, as shown in the following code:

```
var CompoundIngredient = (function () {
  function CompoundIngredient(name) {
    this.name = name;
    this.ingredients = new Array();
  }
  CompoundIngredient.prototype.AddIngredient =
  function (ingredient) {
    this.ingredients.push(ingredient);
  };

  CompoundIngredient.prototype.GetName = function () {
    return this.name;
  };
```

```
  CompoundIngredient.prototype.GetCalories = function () {
    var total = 0;
    for (var i = 0; i<this.ingredients.length; i++) {
      total += this.ingredients[i].GetCalories();
    }
    return total;
  };
  CompoundIngredient.prototype.GetIronContent = function () {
    var total = 0;
    for (var i = 0; i<this.ingredients.length; i++) {
      total += this.ingredients[i].GetIronContent();
    }
    return total;
  };
  CompoundIngredient.prototype.GetVitaminCContent = function () {
    var total = 0;
    for (var i = 0; i<this.ingredients.length; i++) {
      total += this.ingredients[i].GetVitaminCContent();
    }
    return total;
  };
  return CompoundIngredient;
})();
```

The composite ingredient loops over its internal ingredients and performs the same operation on each of them. There is, of course, no need to define an interface due to the prototype model.

To make use of this compound ingredient, we might use the following code:

```
var egg = new SimpleIngredient("Egg", 155, 6, 0);
var milk = new SimpleIngredient("Milk", 42, 0, 0);
var sugar = new SimpleIngredient("Sugar", 387, 0,0);
var rice = new SimpleIngredient("Rice", 370, 8, 0);

var ricePudding = new CompoundIngredient("Rice Pudding");
ricePudding.AddIngredient(egg);
ricePudding.AddIngredient(rice);
ricePudding.AddIngredient(milk);
ricePudding.AddIngredient(sugar);

console.log("A serving of rice pudding contains:");
console.log(ricePudding.GetCalories() + " calories");
```

Of course, this only shows part of the power of the pattern. We could use rice pudding as an ingredient in an even more complicated recipe: rice pudding stuffed buns (they have some strange foods in Westeros). As both the simple and compound version of the ingredients have the same interface, the caller does not need to know that there is any difference between the two ingredient types.

Composite is a heavily used pattern in JavaScript code which deals with HTML elements, as they are a tree structure. For example, the jQuery library provides a common interface if you have selected a single element or a collection of elements. When a function is called, it is actually called on all the children. For instance, the following code will hide all the links on a page, regardless of how many elements are actually found by calling `$("a")`:

```
$("a").hide()
```

The composite is a very useful pattern for JavaScript development.

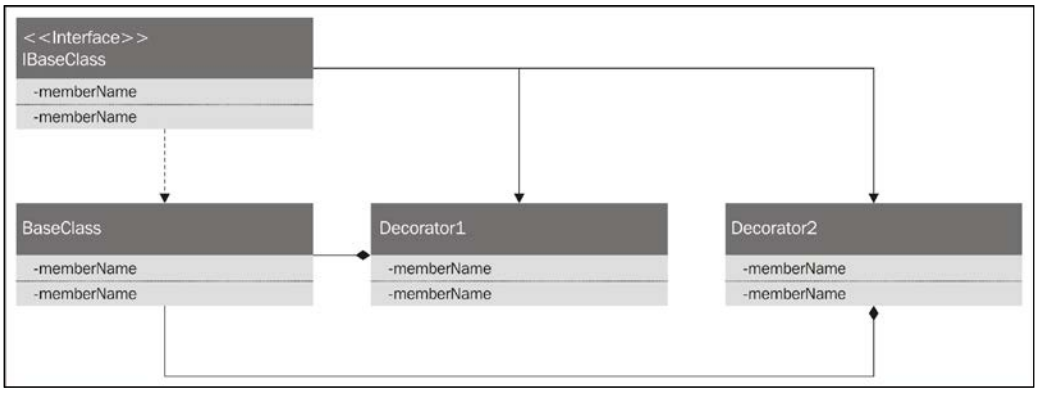
Decorator

The decorator pattern is used to wrap and augment an existing class. Using a decorator pattern is an alternative to subclassing an existing component. Subclassing is typically a compile-time operation and is a tight coupling. This means that once a subclassing is performed, there is no way to alter it at runtime. In cases where there are many possible subclassings that can act in combination, the number of combinations of subclassings explodes. Let's look at an example:

The armor worn by knights in Westeros can be quite configurable. Armor can be fabricated in a number of different styles: scale, lamellar, chain mail, and so on. In addition to the style of armor, there is also a variety of different face guards, knee and elbow joints and, of course, colors. The behavior of armor made from lamellar and a grill is different from chain mail with a face visor. You can see, however, that there are a large number of possible combinations; far too many combinations to explicitly code.

What we do instead is implement the different styles of armor using the decorator pattern. Decorator works using a similar theory to the adapter and bridge patterns, in that it wraps another instance and proxies calls through. The decorator pattern, however, performs the redirections at runtime by having the instance to wrap passed into it. Typically, a decorator will act as a simple pass through for some methods, and for others, it will make some modifications. These modifications could be limited to performing an additional action before passing the call off to the wrapped instance, or could go so far as to change the parameters passed in.

The following is the class diagram of the decorator pattern:



This allows for very granular control over which methods are altered by the decorator and which remain as mere pass-throughs. Let's take a look at an implementation of the pattern in JavaScript.

Implementation

For our armor example, the code looks like:

```
var BasicArmor = (function () {
  function BasicArmor() {
  }
  BasicArmor.prototype.CalculateDamageFromHit = function (hit) {
    return 1;
  };
  BasicArmor.prototype.GetArmorIntegrity = function () {
    return 1;
  };
   return BasicArmor;
})();


var ChainMail = (function () {
  function ChainMail(decoratedArmor) {
    this.decoratedArmor = decoratedArmor;
  }
  ChainMail.prototype.CalculateDamageFromHit = function (hit) {
    hit.Strength = hit.Strength * .8;
    return this.decoratedArmor.CalculateDamageFromHit(hit);
  };
```

```
    ChainMail.prototype.GetArmorIntegrity = function () {
      return .9 * this.decoratedArmor.GetArmorIntegrity();
    };
    return ChainMail;
})();
```

The `ChainMail` armor takes in an instance of armor that complies with an interface, like the following code:

```
export interface IArmor{
  CalculateDamageFromHit(hit: Hit):number;
  GetArmorIntegrity():number;
}
```

That instance is wrapped and calls proxied through. The `GetArmorIntegiry` method modifies the result from the underlying class, while the `CalculateDamageFromHit` method modifies the arguments that are passed into the decorated class. This `ChainMail` class could, itself, be decorated with several more layers of decorators, until a long chain of methods is actually called for each method call. This behavior, of course, remains invisible to outside callers.

To make use of this armor, you simply use the following code:

```
var armor = new ChainMail(new BasicArmor());
console.log(armor.CalculateDamageFromHit({Location: "head",
Weapon: "Sock filled with pennies", Strength: 12}));
```

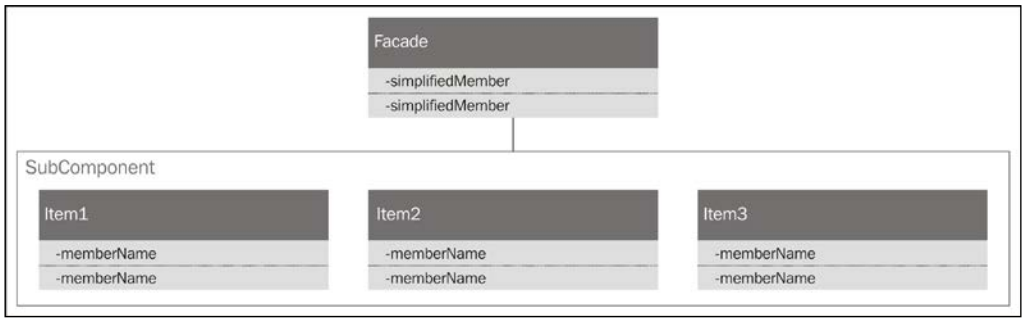
It is tempting to make use of JavaScript's ability to rewrite individual methods on classes to implement this pattern. Indeed, in an earlier draft of this section, I had intended to suggest just that. However, doing so is syntactically messy and not a common way of doing things. One of the most important things to keep in mind when programming is that code must be maintainable, not only by you but also by others. Complexity breeds confusion and confusion breeds bugs.

The decorator pattern is a valuable pattern for scenarios where inheritance is too limiting. These scenarios still exist in JavaScript so the pattern remains useful.

Façade

The façade pattern is a special case of the adapter pattern that provides a simplified interface over a collection of classes. I mentioned such a scenario in the *Adapter* section but only within the context of a single class, `SimpleShip`. This same idea can be expanded to provide an abstraction around a group of classes or an entire subsystem.

The following is the class diagram of the façade pattern:



Implementation

If we take the same `SimpleShip` class as before and expand it to an entire fleet, we have a great example of the use to create a façade. If it was difficult to sail a single ship, it would be far more difficult to command an entire fleet of ships. There is a great deal of nuance required; commands to individual ships would have to be made. In addition to the individual ships, there must also be a fleet `Admiral` class and a degree of coordination between the ships in order to distribute supplies. All of this can be abstracted away. If we have a collection of classes representing the aspects of a fleet such as the following code:

```
var Ship = (function () {
  function Ship() {
  }
  Ship.prototype.TurnLeft = function () {
  };
  Ship.prototype.TurnRight = function () {
  };
  Ship.prototype.GoForward = function () {
  };
  return Ship;
})();
Transportation.Ship = Ship;

var Admiral = (function () {
  function Admiral() {
  }
  return Admiral;
 })();
```

```
Transportation.Admiral = Admiral;

var SupplyCoordinator = (function () {
  function SupplyCoordinator() {
  }
  return SupplyCoordinator;
})();
Transportation.SupplyCoordinator = SupplyCoordinator;
```

Then, we might build a façade as follows:

```
var Fleet = (function () {
  function Fleet() {
  }
  Fleet.prototype.setDestination = function (destination) {
    //pass commands to a series of ships, admirals and whoever
    else needs it
  };

  Fleet.prototype.resupply = function () {
  };

  Fleet.prototype.attack = function (destination) {
    //attack a city
  };
  return Fleet;
})();
```

Façades are very useful abstractions, especially in dealing with APIs. Using a façade around a granular API can create an easier interface. The level of abstraction at which the API works can be raised, so that it is more in sync with how your application works. For instance, if you're interacting with the Azure blob storage API, you could raise the level of abstraction from working with individual files to working with a collection of files. Instead of writing the following code:

```
$.ajax({method: "PUT",
url: "https://settings.blob.core.windows.net/container/set1",
data: "setting data 1"});
$.ajax({method: "PUT",
url: "https://settings.blob.core.windows.net/container/set2",
data: "setting data 2"});

$.ajax({method: "PUT",
url: "https://settings.blob.core.windows.net/container/set3",
data: "setting data 3"});
```

A façade could be written which encapsulates all of these calls and provides an interface like:

```
public interface SettingSaver{
  Save(settings: Settings); //previous code in this method
  Retrieve():Settings;
}
```

As you can see, façades remain useful in JavaScript and should be a pattern that remain in your toolbox.

Flyweight

In boxing, there is a lightweight division between 49 to 52 kgs known as the **flyweight division**. It was one of the last divisions to be established and was named, I imagine, for the fact that the fighters in it were tiny like flies.

The flyweight pattern is used in instances when there is a large number of instances of objects which only vary slightly. I should perhaps pause here to mention that a large number, in this situation, is probably around 10,000 objects rather than 50 objects. However, the cutoff for the number of instances is highly dependent on how expensive the object is to create. In some cases, the object may be so expensive that only a handful of objects are required before they overload the system. In this case, introducing flyweight at a smaller number would be beneficial. Maintaining a full object for each object consumes a lot of memory. It seems that the memory is largely consumed wastefully too, as most of the instances have the same value for their fields. Flyweight offers a way to compress this data, by only keeping track of the values that differ from some prototype in each instance.

JavaScript's prototype model is ideal for this scenario. We can simply assign the most common value to the prototype and have individual instances override them as needed. Let's see how that looks with an example.

Implementation

Returning once more to Westeros (aren't you glad I've opted for a single overarching problem domain?), we find that their armies are full of ill-equipped fighting people. Within this set of people, there is really very little difference from the perspective of the generals. Certainly, each person has their own life, ambitions, and dreams but they have all been adapted into simple fighting automatons in the eyes of the general. The general is only concerned with how well the soldiers fight, and whether they're healthy and well fed.

We can see the simple set of fields in the following code:

```
var Soldier = (function () {
  function Soldier() {
    this.Health = 10;
    this.FightingAbility = 5;
    this.Hunger = 0;
  }
  return Soldier;
})();
```

Of course, with an army of 10,000 soldiers, keeping track of all of this requires quite some memory. Let's take a different approach and use prototypes:

```
var Soldier = (function () {
  function Soldier() { }
  Soldier.prototype.Health = 10;
  Soldier.prototype.FightingAbility = 5;
  Soldier.prototype.Hunger = 0;

  return Soldier;
})();
```

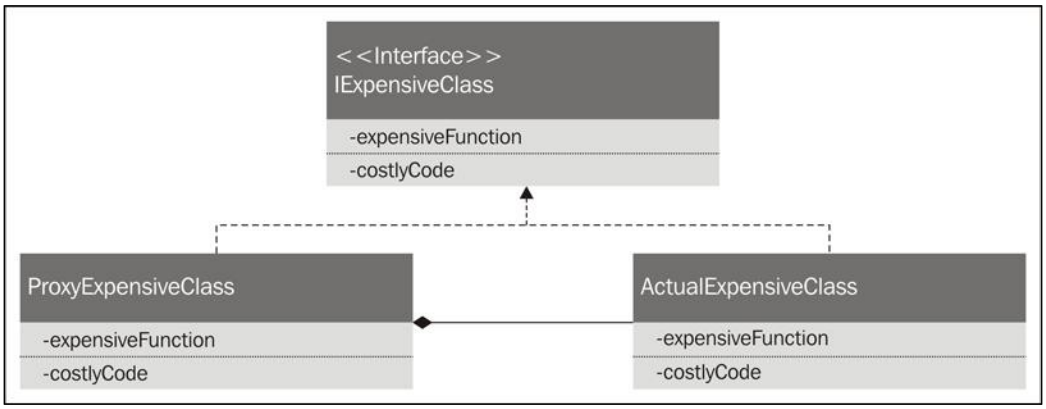
Using this approach, we are able to defer all requests for the soldier's health to the prototype. Setting the value is easy too, as shown in the following code:

```
var soldier1 = new Soldier();
var soldier2 = new Soldier();
console.log(soldier1.Health); //10
soldier1.Health = 7;
console.log(soldier1.Health); //7
console.log(soldier2.Health); //10
delete soldier1.Health;
console.log(soldier1.Health); //10
```

You'll note that we make a call to remove the property override and return the value back to the parent value.

Proxy

The final pattern presented in this chapter is the proxy. In the previous section, I mentioned how it is expensive to create objects and how we would like to avoid creating too many of them. The proxy pattern provides a method of controlling the creation and use of expensive objects. The following is the class diagram of the proxy pattern:



As you can see, the proxy mirrors the interface of the actual instance. It is substituted for the instance in all the clients and, typically, wraps a private instance of the class. There are a number of places where the proxy pattern can be of use:

- Lazy instantiation of an expensive object
- Protection of secret data
- Stubbing for remote method invocation
- Interposing additional actions before or after method invocation

Many times, an object is expensive to instantiate and we don't want to prematurely create instance if they're not actually used. In this case, the proxy can check its internal instance, and if not yet initiated, create it before passing on the method call. This is known as **lazy instantiation**.

If a class has been designed without any security in mind but now requires some, then this can be provided through the use of a proxy. The proxy will check the call and only pass the method call on in cases where the security checks out.

The proxy may be used to simply provide an interface to methods that are invoked somewhere else. In fact, this is exactly how a number of web socket libraries function, proxying calls back to the web server.

Finally, there may be cases where it is useful to interpose some functionality into the method invocation. This could be logging of parameters, validating of parameters, altering results, and so on.

Implementation

Let's take a look at the Westeros example where method interposition is needed. As it tends to happen, the units of measurement for liquids varies greatly from one side of the land to the other. In the north, one might buy a pint of beer, while in the south one would buy it by the dragon. This causes no end of confusion and code duplication but can be solved by wrapping classes that care about measurement in proxies.

For example, the following code is for a barrel calculator which estimates the number of barrels needed to ship a particular quantity of liquid:

```
var BarrelCalculator = (function () {
  function BarrelCalculator() {
  }
  BarrelCalculator.prototype.calculateNumberNeeded =
  function (volume) {
  return Math.ceil(volume / 357);
  };
  return BarrelCalculator;
})();
```

Although it is not well documented here, this version takes pints as a volume parameter. A proxy is created which deals with the transformation as follows:

```
var DragonBarrelCalculator = (function () {
  function DragonBarrelCalculator() {
  }
  DragonBarrelCalculator.prototype.calculateNumberNeeded =
  function (volume) {
    if (this._barrelCalculator == null)
      this._barrelCalculator = new BarrelCalculator();
    return this._barrelCalculator.calculateNumberNeeded
    (volume * .77);
  };
  return DragonBarrelCalculator;
})();
```

This proxy class does the unit conversion for us and helps alleviate some confusion.

Proxy is absolutely a useful pattern within JavaScript. I already mentioned that it is used by web socket libraries when generating stubs, but it finds itself useful in countless other locations.

Hints and tips

Many of the patterns presented in this chapter provide methods of abstracting functionality, and of molding interfaces to look the way you want. Keep in mind that with each layer of abstraction, a cost is introduced. Function calls take longer, but it is also much more confusing for people who need to understand your code. Tooling can help a little, but tracking a function call through nine layers of abstraction is never fun.

Also be wary of doing too much in the façade pattern. It is very easy to turn the façade into a fully fledged management class, and that degrades easily into a god object that is responsible to coordinate and do everything.

Summary

In this chapter, we've looked at a number of patterns used to structure the interaction between objects. Some of them are quite similar to each other but they are all useful in JavaScript, although bridge is effectively reduced to adapter.

In the next chapter, we'll finish our examination of the original GoF patterns by looking at behavioral patterns.

5
Behavioral Patterns

In the last chapter, we looked at structural patterns that describe ways in which objects can be constructed to ease interaction.

In this chapter, we'll take a look at the final, and largest, grouping of GoF patterns: behavioral patterns. These patterns are the ones that provide guidance on how objects share data, or from a different perspective, how data flows between objects.

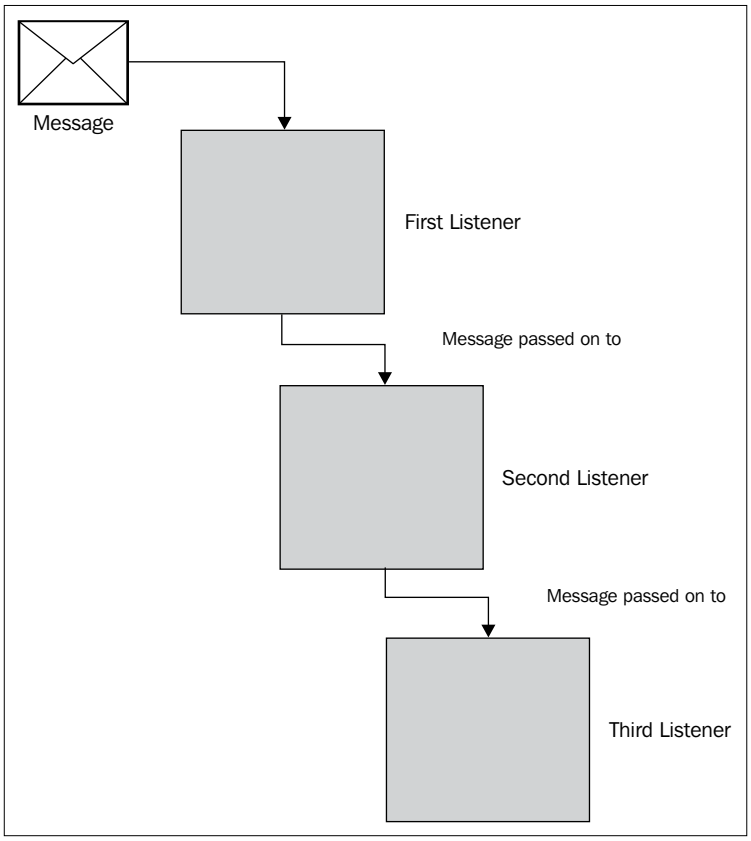
The patterns we'll look at are:

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

Once again, there are a number of more recently identified patterns that could well be classified as behavioral patterns. We'll defer looking at those until a later chapter, instead keeping to the GoF patterns.

Chain of responsibility

We can think of a function call on an object as sending that object a message. Indeed this message-passing mentality was one that dates back to the days of Smalltalk. The chain of responsibility pattern describes an approach in which a message trickles down from one class to another. A class can either act on the message or allow it to be passed onto the next member of the chain. Depending on the implementation, there are a few different rules that can be applied to the message passing. In some situations, only the first matching link in the chain is permitted to act. In others, every matching link acts on the message. Sometimes, the links are permitted to stop processing or even to mutate the message as it continues down the chain. A typical message flow is shown in this diagram:

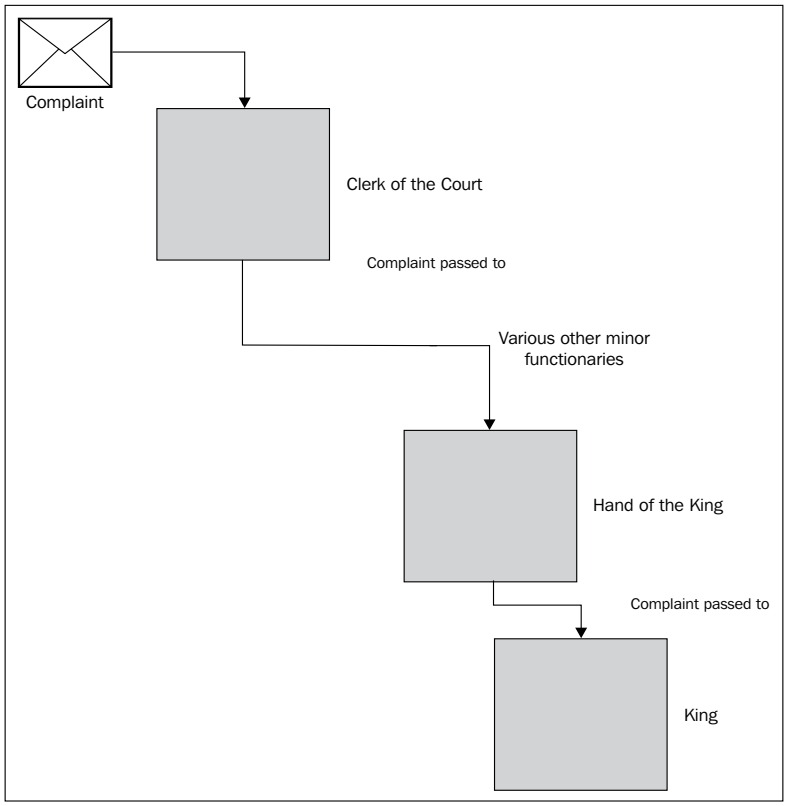


Let's see if we can find a good example of this pattern in our go to example land of Westeros.

Implementation

There is very little in the way of a legal system in Westeros. Certainly, there are laws and even city guards who enforce them but the judicial system is scant. The law of the land is really decided by the king and his advisors. Those with the time and money can petition for an audience with the king who will listen to their complaint and pass a ruling. This ruling is law. Of course, any king who spent his entire day listening to the complaints of peasants would go mad. For this reason, many of the cases are caught and solved by his advisors before they reach his ears.

To represent this in code, we'll need to start by thinking about how the chain of responsibility would work. A complaint comes in and it starts with the lowest possible person who can solve it. If that person cannot or will not solve the problem, it trickles up to a more senior member of the ruling class. Eventually, the problem reaches the king, who is the final arbiter of disputes. We can think of him as the default dispute solver who is called upon when all else fails. This is explained in the following diagram:



We'll start with an interface to describe those who might listen to complaints. Again, this is just pseudo code as there are no interfaces in JavaScript (all these interfaces are actually written in TypeScript, but that can be our little secret):

```
export interface ComplaintListener{
  IsAbleToResolveComplaint(complaint: Complaint): boolean;
  ListenToComplaint(complaint: Complaint): string;
}
```

The interface requires two methods. The first is a simple check to see if the class is able to resolve a given complaint. The second listens to and resolves the complaint. Next, we'll need to describe what constitutes a complaint. The following is the code of the `Complaint` class:

```
var Complaint = (function () {
  function Complaint() {
    this.ComplainingParty = "";
    this.ComplaintAbout = "";
    this.Complaint = "";
  }
  return Complaint;
})();
```

Next, we need a couple of different classes that implement `ComplaintListener` and are able to solve complaints:

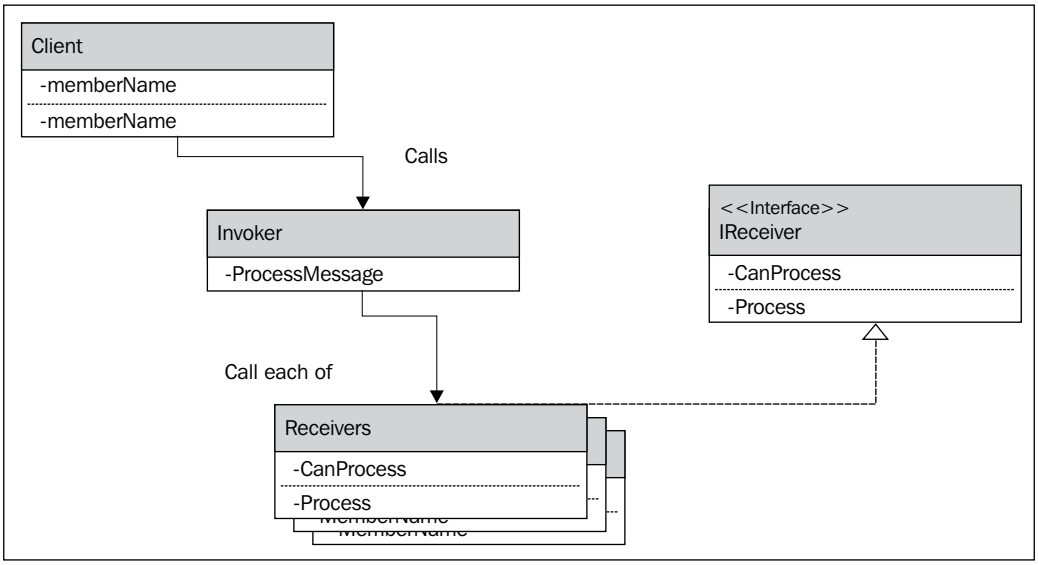
```
var ClerkOfTheCourt = (function () {
  function ClerkOfTheCourt() {
  }
  ClerkOfTheCourt.prototype.IsAbleToResolveComplaint = function
  (complaint) {
    //decide if this is a complaint that can be solved by the
    clerk
    return false;
  };

  ClerkOfTheCourt.prototype.ListenToComplaint = function
  (complaint) {
    //perform some operation
    //return solution to the complaint
    return "";
  };
```

```
  return ClerkOfTheCourt;
})();
JudicialSystem.ClerkOfTheCourt = ClerkOfTheCourt;

var King = (function () {
  function King() {
  }
  King.prototype.IsAbleToResolveComplaint = function (complaint) {
    return true;
  };

  King.prototype.ListenToComplaint = function (complaint) {
    //perform some operation
    //return solution to the complaint
    return "";
  };
  return King;
})();
JudicialSystem.King = King;
```

Each one of these classes implements a different approach to solving the complaint. We need to chain them together making sure that the `King` class is in the default position. This can be seen in the following code:

```
var ComplaintResolver = (function () {
  function ComplaintResolver() {
    this.complaintListeners = new Array();
    this.complaintListeners.push(new ClerkOfTheCourt());
    this.complaintListeners.push(new King());
  }
  ComplaintResolver.prototype.ResolveComplaint = function
  (complaint) {
    for (var i = 0; i < this.complaintListeners.length; i++) {
      if (this.complaintListeners[i].
      IsAbleToResolveComplaint(complaint)) {
        return this.complaintListeners[i].
        ListenToComplaint(complaint);
      }
    }
  };
  return ComplaintResolver;
})();
```

This code will work its way through each of the listeners until it finds one that is interested in hearing the complaint. In this version, the result is returned immediately, halting any further processing. There are variations of this pattern in which multiple listeners could fire, allowing the listeners to mutate the parameters for the next listener. A chain of listeners is shown here:

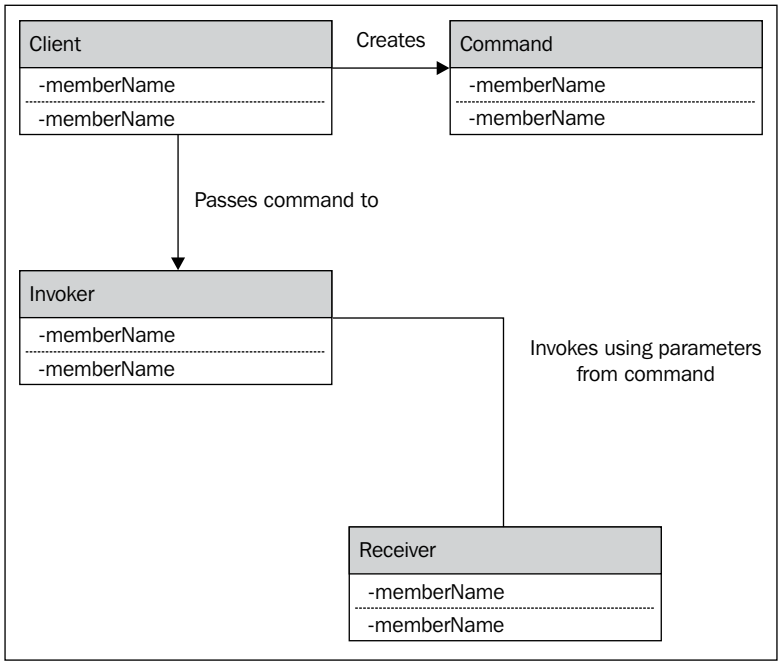


Chain of responsibility is a highly useful pattern in JavaScript. In browser-based JavaScript, the events that are fired fall through a chain of responsibility. For instance, you can attach multiple listeners to the click event on a link and each of them will fire until finally the default navigation listener fires. It is likely that you're using the chain of responsibility in most of your code without even knowing it.

Command

The command pattern is a method of encapsulating both the parameters to a method and the current object state, and the method to be called. In effect, the command pattern packs up everything needed to call a method at a later date into a nice little package. Using this approach, one can issue a command and wait until a later date to decide which piece of code will execute the command. This package can then be queued or even serialized for later execution. Having a single point of command execution also allows us to easily add functionality such as undo or command logging.

This pattern can be a bit difficult to imagine, so let's break it down into its components, as shown in the following diagram:



The command message

The first component of the command pattern is, predictably, the command itself. As I mentioned, the command encapsulates everything needed to invoke a method. This includes the method name, the parameters, and any global state. As you can imagine, keeping track of the global state in each command is very difficult. What happens if the global state changes after the command has been created? This dilemma is yet another reason why using a global state is problematic and should be avoided.

There are a couple of options to set up commands. At the simple end of the scale all that is needed is to track a function and a set of parameters. Because functions are first class objects in JavaScript, they can easily be saved into an object. We can also save the parameters to the function into a simple array. Let's build a command using this very simple approach.

The deferred nature of commands suggests an obvious metaphor in the land of Westeros. There are no methods of communicating quickly in Westeros. The best method is to attach small messages to birds and release them. The birds have a tendency to want to return to their homes so each lord raises a number of birds in their home and, when they come of age, sends them to other lords who might wish to communicate with them. The lords keep an aviary of birds and retain records of which bird will travel to which other lord. The king of Westeros sends many of his commands to his loyal lords through this method.

The commands sent by the king contain all the necessary instructions for the lords. The command may be something like "bring your troops" and the arguments to that command may be a number of troops, a location, and a date by which the command must be carried out.

In JavaScript, the simplest way of representing this is through an array:

```
var simpleCommand = new Array();
simpleCommand.push(new LordInstructions().BringTroops);
simpleCommand.push("King's Landing");
simpleCommand.push(500);
simpleCommand.push(new Date());
```

This array can be passed around and invoked at will. To invoke it, a generic function can be used, as shown in the following code:

```
simpleCommand[0](simpleCommand[1], simpleCommand[2],
simpleCommand[3]);
```

As you can see, this function only works for commands with three arguments. You can, of course, expand this to any number:

```
simpleCommand[0](simpleCommand[1], simpleCommand[2],
simpleCommand[3], simpleCommand[4], simpleCommand[5],
simpleCommand[6]);
```

The additional parameters are undefined, but the function doesn't use them, so there is no ill effect. Of course, this is not at all an elegant solution.

It is desirable to build a class for each sort of command. This allows you to ensure the correct arguments have been supplied and easily distinguish the different sorts of commands in a collection. Typically, commands are named using the imperative, as they are instructions. Examples of this are `BringTroops`, `Surrender`, `SendSupplies`, and so on.

Let's transform our ugly, simple command into a proper class:

```
var BringTroopsCommand = (function () {
  function BringTroopsCommand(location, numberOfTroops, when) {
    this._location = location;
    this._numberOfTroops = numberOfTroops;
    this._when = when;
  }
  BringTroopsCommand.prototype.Execute = function () {
    var receiver = new LordInstructions();
    receiver.BringTroops(this._location, this._numberOfTroops,
    this._when);
  };
  return BringTroopsCommand;
})();
```

We may wish to implement some logic to ensure that the parameters passed into the constructor are correct. This will ensure that the command fails on creation instead of on execution. It is easier to debug the issue during creation rather than during execution as execution could be delayed, even for days. The validation won't be perfect, but even if it catches only a small portion of errors it is helpful.

As mentioned, these commands can be saved for later use in memory or even written to disk.

The invoker

The invoker is the part of the command pattern that instructs the command to execute its instructions. The invoker can really be anything; a timed event, a user interaction, or just the next step in the process may all trigger invocation. When we executed `simpleCommand` earlier, we were playing at being the invoker. In the more rigorous command, the invoker might look something like:

```
command.Execute()
```

As you can see, invoking a command is very easy. Commands may be invoked at once or at some later date. One popular approach is to defer the execution of the command to the end of the event loop. This can be done in the node with:

```
process.nextTick(function(){command.Execute();});
```

The `process.nextTick` function defers the execution of a command to the end of the event loop such that it is executed next time the process has nothing to do.

The receiver

The final component in the command pattern is the receiver. This is the target of the command execution. In our example, we created a receiver called `LordInstructions`:

```
var LordInstructions = (function () {
  function LordInstructions() {
  }
  LordInstructions.prototype.BringTroops = function (location,
  numberOfTroops, when) {
    console.log("You have been instructed to bring " +
    numberOfTroops + " troops to " + location + " by " + when);
  };
  return LordInstructions;
})();
```

The receiver knows how to perform the action that the command has deferred. There need not be anything special about the receiver; in fact it may be any class.

Together these components make up the command pattern. A client will generate a command and pass it off to an invoker that may delay the command or execute it at once and the command will act upon a receiver.

In the case of building an undo stack, the commands are special, in that they have both an `Execute` and an `Undo` method. One takes the application state forward and the other backward. To perform an undo, simply pop the command off the undo stack, execute the `Undo` function, and push it onto a redo stack. For redo, pop from redo, run `Execute`, and push to the undo stack. Simple as that, although one must make sure all state mutations are performed through commands.

The GoF book outlines a slightly more complicated set of players for the command pattern. This is largely due to the reliance on interfaces that we've avoided in JavaScript. The pattern becomes much simpler thanks to the prototype inheritance model in JavaScript.

The command pattern is a very useful one to defer the execution of some piece of code. We'll actually explore the command pattern and some useful companion patterns in *Chapter 9*, *Messaging Patterns*.

Interpreter

The interpreter pattern is an interesting pattern as it allows for the creation of your own language. This might sound like a crazy idea: we're already writing JavaScript, why would we want to create a new language? Since the publication of the GoF book, **domain specific languages** (**DSLs**) have had something of a renaissance. There are situations where it is quite useful to create a language that is specific to one requirement. For instance, **Structured Query Language** (**SQL**) is very good at describing the querying of relational databases. Equally, regular expressions have proven themselves to be highly effective for the parsing and manipulation of text.

There are many scenarios in which being able to create a simple language is useful. That's really the key: a simple language. Once the language gets more complicated, the advantages are quickly lost to the difficulty of creating what is, in effect, a compiler.

This pattern is different from those we've seen to this point as there is no real class structure that is defined by the pattern. You can design your language interpreter as you wish.

An example

For our example, let's define a language that can be used to describe historical battles in the land of Westeros. The language must be simple for clerics to write and easy to read. We'll start by creating a simple grammar such as `(aggressor -> battle ground <- defender) -> victor`.

In the preceding line, you can see that we're just writing out a rather nice syntax that will let people describe battles. A battle between Robert Baratheon and Rhaegar Targaryen at the river Trident would look like `(Robert Baratheon -> River Trident <- RhaegarTargaryen) -> Robert Baratheon`.

Using this grammar, we would like to build some code that is able to query a list of battles for answers. In order to do this, we're going to rely on regular expressions. For most languages, this wouldn't be a good approach as the grammar is too complicated. In those cases, one might wish to create a lexor and a parser and build up syntax trees; however, by that point, you may wish to reexamine whether creating a DSL is really a good idea. For our language, the syntax is very simple, so we can get away with regular expressions.

Implementation

The first thing we establish is a JavaScript data model for the battle as follows:

```
var Battle = (function () {
  function Battle(battleGround, agressor, defender, victor) {
    this.battleGround = battleGround;
    this.agressor = agressor;
    this.defender = defender;
    this.victor = victor;
  }
  return Battle;
})();
```

Next, we need a parser:

```
var Parser = (function () {
  function Parser(battleText) {
    this.battleText = battleText;
    this.currentIndex = 0;
    this.battleList = battleText.split("\n");
  }
  Parser.prototype.nextBattle = function () {
    if (!this.battleList[0])
    return null;
    var segments = this.battleList[0].
    match(/\((.+?)\s?->\s?(.+?)\s?<-\s?(.+?)\s?->\s?(.+)/);
    return new Battle(segments[2], segments[1], segments[3],
    segments[4]);
  };
  return Parser;
})();
```

It is likely best that you don't think too much about that regular expression. However, the class does take in a list of battles (one per line) and using `nextBattle` allows one to parse them. To use the class, we simply need the following code:

```
var text = "(Robert Baratheon -> River Trident <-
  RhaegarTargaryen) -> Robert Baratheon";
var p = new Parser(text);
p.nextBattle()
```

Its output will be:

```
{ battleGround: 'River Trident',
  agressor: 'Robert Baratheon',
  defender: 'RhaegarTargaryen)',
  victor: 'Robert Baratheon' }
```

This data structure can now be queried like one would for any other structure in JavaScript.

As I mentioned earlier, there is no fixed way to implement this pattern, so the previous implementation is provided simply as an example. Your implementation will very likely look very different and that is just fine.

Interpreter can be a useful pattern in JavaScript. It is, however, a pretty infrequently used pattern in most situations. The best example of a language interpreted in JavaScript is the Less language that is compiled, by JavaScript, to CSS.

Iterator

Traversing collections of objects is an amazingly common problem, so much so that many languages provide for special constructs just to move through collections. For example, C# has a `foreach` loop and Python has `for x in`. These looping constructs are frequently built on top of an iterator. An iterator is a pattern that provides a simple method to select, sequentially, the next item in a collection.

The interface for the iterator looks like the following code:

```
interface Iterator{
    next();
}
```

Implementation

In the land of Westeros, there is a well-known sequence of people in line for the throne in the very unlikely event that the king were to die. We can set up a handy iterator on top of this collection and simply call `next` on it should the ruler die. The following code keeps track of a pointer to the current element in the iteration:

```
var KingSuccession = (function () {
  function KingSuccession(inLineForThrone) {
    this.inLineForThrone = inLineForThrone;
    this.pointer = 0;
  }
```

```
  KingSuccession.prototype.next = function () {
    return this.inLineForThrone[this.pointer++];
  };
  return KingSuccession;
})();
```

This is primed with an array and then we can call it with the following code:

```
var king = new KingSuccession(["Robert Baratheon" ,
"JofferyBaratheon", "TommenBaratheon"]);
king.next() //'Robert Baratheon'
king.next() //'JofferyBaratheon'
king.next() //'TommenBaratheon'
```

An interesting application of iterators is to iterate over a collection that is not fixed. For instance, an iterator can be used to generate sequential members of an infinite set like the Fibonacci sequence:

```
var FibonacciIterator = (function () {
  function FibonacciIterator() {
    this.previous = 1;
    this.beforePrevious = 1;
  }
  FibonacciIterator.prototype.next = function () {
    var current = this.previous + this.beforePrevious;
    this.beforePrevious = this.previous;
    this.previous = current;
    return current;
  };
  return FibonacciIterator;
})();
```

This is used as follows:

```
var fib = new FibonacciIterator()
fib.next() //2
fib.next() //3
fib.next() //5
fib.next() //8
fib.next() //13
fib.next() //21
```

Iterators are handy constructs allowing to explore not just arrays but any collection or even any generated list. There are a ton of places where this can be used to great effect.

ECMAScript 6 iterators

Iterators are so useful that they are actually part of the next generation of JavaScript. The iterator pattern used in ECMAScript 6 is a single method that returns an object that contains a `done` and `value` parameters. The `done` parameter is `true` when the iterator is at the end of the collection. What is nice about the ECMAScript 6 iterators is that the array collection in JavaScript will support the iterator. This opens up a new syntax that can largely replace the `for` loop:

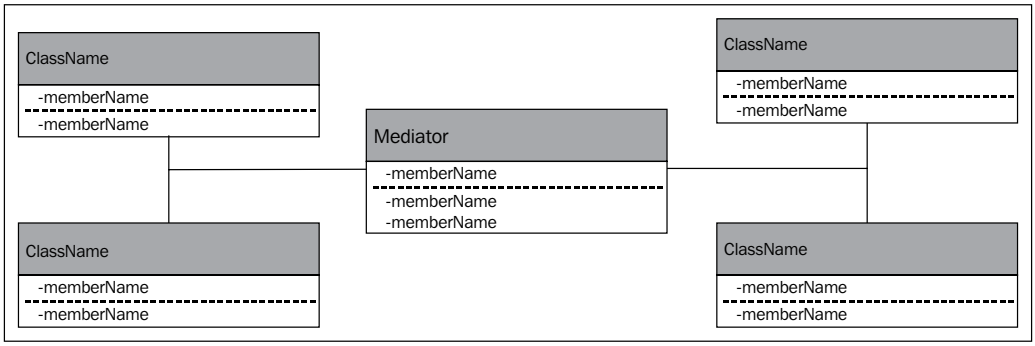
```
var kings = new KingSuccession(["Robert Baratheon"
,"JofferyBaratheon", "TommenBaratheon"]);
for(var king of kings){
  //act on members of kings
}
```

Iterators are a syntactic nicety that has long been missing from JavaScript.

Mediator

Managing many-to-many relationships in classes can be a complicated prospect. Let's consider a form that contains a number of controls, each of which wants to know if other controls on the page are valid before performing their action. Unfortunately, having each control know about every other control creates a maintenance nightmare. Each time a new control is added every other control needs to be modified.

A mediator will sit between the various components and act as a single place in which message routing changes can be made. By doing so, the mediator simplifies the otherwise complex work needed to maintain the code. In the case of controls on a form, the mediator is likely to be the form itself. The mediator acts much like a real-life mediator would, clarifying and routing information exchange between a number of parties. The following diagram illustrates the mediator pattern:



Implementation

In the land of Westeros, there are many times when a mediator is needed. Frequently, the mediator ends up deceased, but I'm sure that won't be the case with our example.

There are a number of great families in Westeros who own large castles and vast tracts of land. Lesser lords swear themselves to the great houses, forming alliances, frequently supported through marriage.

When coordinating the various houses sworn to them, the great lord will act as a mediator, communicating information back and forth between the lesser lords and resolving any disputes they may have amongst themselves.

In this example, we'll greatly simplify the communication between the houses and say that all messages pass through the great lord. In this case, we'll use the house of Stark as our great lord. They have a number of other houses which talk with them. Each of the houses looks roughly like:

```
var Karstark = (function () {
  function Karstark(greatLord) {
    this.greatLord = greatLord;
  }
  Karstark.prototype.receiveMessage = function (message) {
  };
  Karstark.prototype.sendMessage = function (message) {
    this.greatLord.routeMessage(message);
  };
  return Karstark;
})();
```

They have two functions, one of which receives messages from a third party and one of which sends messages out to their great lord that is set upon instantiation. The `HouseStark` class looks like the following code:

```
var HouseStark = (function () {
  function HouseStark() {
    this.karstark = new Karstark(this);
    this.bolton = new Bolton(this);
    this.frey = new Frey(this);
    this.umber = new Umber(this);
  }
  HouseStark.prototype.routeMessage = function (message) {
  };
  return HouseStark;
})();
```

By passing all messages through the `HouseStark` class, the various other houses do not need to concern themselves with how their messages are routed. This responsibility is handed off to `HouseStark`, which acts as the mediator.

Mediators are best used when the communication is both complex and well defined. If the communication is not complex, then the mediator adds extra complexity. If the communication is ill defined, then it becomes difficult to codify the communication rules in a single place.

Simplifying communication between many-to-many objects is certainly useful in JavaScript. I would actually argue that in many ways jQuery acts as a mediator. When acting on a set of items on the page, jQuery serves to simplify communication by abstracting away the need to know exactly which objects on the page are being changed. For instance, refer to the following code:

```
$(".error").slideToggle();
```

This is jQuery shorthand to toggle the visibility of all the elements on the page that have the `error` class.

Memento

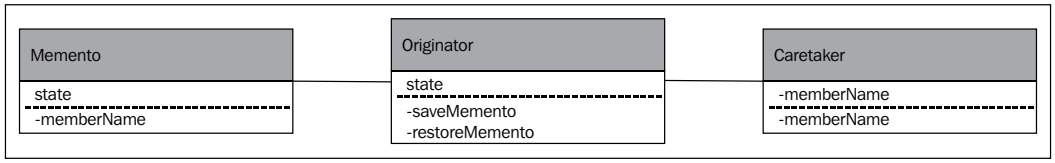
In the *Command* section, we talked briefly about the ability to undo operations. Creating reversible commands is not always possible. For many operations, there is no apparent reversing operation that can restore the original state. For instance, imagine the code that squares a number:

```
var SquareCommand = (function () {
  function SquareCommand(numberToSquare) {
    this.numberToSquare = numberToSquare;
  }
  SquareCommand.prototype.Execute = function () {
    this.numberToSquare *= this.numberToSquare;
  };
  return SquareCommand;
})();
```

Giving this code, -9 will result in 81 but giving it 9 will also result in 81. There is no way to reverse this command without additional information.

The memento pattern provides an approach to restore the state of objects to a previous state. The memento keeps a record of the previous values of a variable and provides the functionality to restore them. Keeping a memento around for each command allows for easy restoration of irreversible commands.

In addition to an undo-stack, there are many instances where having the ability to roll back the state of an object is useful. For instance, doing "what-if" analysis requires that you make some hypothetical changes to a state and then observe how things change. The changes are generally not permanent so they could be rolled back using the memento pattern or, if the projects are desirable, left in place. The following diagram explains a typical memento implementation:



A typical memento implementation involves three players:

- **Originator**: The originator holds some form of the state and provides an interface to generate new mementos.
- **Caretaker**: This is the client of the pattern. It is what requests that new mementos be taken and governs when they are to be restored.
- **Memento**: This is a representation of the saved state of the originator. This is what can be saved to storage to allow for rolling back.

It can help to think of the members of the memento pattern as a boss and a secretary taking notes. The boss (caretaker) dictates some memo to the secretary (originator) who writes notes in a notepad (memento). From time to time, the boss may request that the secretary cross out what he has just written.

The involvement of the caretaker can be varied slightly with the memento pattern. In some implementation, the originator will generate a new memento each time a change is made to its state. This is commonly known as copy on write, as a new copy of the state is created and the change is applied to it. The old version can be saved to a memento.

Implementation

In the land of Westeros, there are a number of soothsayers/foretellers of the future. They work by using magic to peer into the future and examining how certain changes in the present will play out in the future. Often, there is need for numerous foretellings with slightly different starting conditions. When setting their starting conditions, a memento pattern is invaluable.

We start off with a world state that gives information on the state of the world for a certain starting point:

```
var WorldState = (function () {
  function WorldState(numberOfKings, currentKingInKingsLanding,
  season) {
    this.numberOfKings = numberOfKings;
    this.currentKingInKingsLanding = currentKingInKingsLanding;
    this.season = season;
  }
  return WorldState;
})();
```

This world state is used to track all the conditions that make up the world. It is what is altered by the application every time a change to the starting conditions is made. Because this world state encompasses the whole state for the application, it can be used as a memento. We can serialize this object and save it to the disk or send it back to some history server somewhere.

The next thing we need is a class that provides the same state as the memento and allows for the creation and restoration of mementos. In our example, we've called this a world state provider:

```
var WorldStateProvider = (function () {
  function WorldStateProvider() {
  }
  WorldStateProvider.prototype.saveMemento = function () {
    return new WorldState(this.numberOfKings,
    this.currentKingInKingsLanding, this.season);
  };
  WorldStateProvider.prototype.restoreMemento = function (memento)
  {
    this.numberOfKings = memento.numberOfKings;
    this.currentKingInKingsLanding =
    memento.currentKingInKingsLanding;
    this.season = memento.season;
  };
  return WorldStateProvider;
})();
```

Finally, we need a client for the foretelling, which we'll call soothsayer:

```
var Soothsayer = (function () {
  function Soothsayer() {
    this.startingPoints = [];
```

```
    this.currentState = new WorldStateProvider();
  }
  Soothsayer.prototype.setInitialConditions = function
  (numberOfKings, currentKingInKingsLanding, season) {
    this.currentState.numberOfKings = numberOfKings;
    this.currentState.currentKingInKingsLanding =
    currentKingInKingsLanding;
    this.currentState.season = season;
  };
  Soothsayer.prototype.alterNumberOfKingsAndForetell = function
  (numberOfKings) {
    this.startingPoints.push(this.currentState.saveMemento());
    this.currentState.numberOfKings = numberOfKings;
    //run some sort of prediction
  };
  Soothsayer.prototype.alterSeasonAndForetell = function (season)
  {
    //as above
  };
  Soothsayer.prototype.alterCurrentKingInKingsLandingAndForetell =
  function (currentKingInKingsLanding) {
    //as above
  };
  Soothsayer.prototype.tryADifferentChange = function () {
    this.currentState.restoreMemento(this.startingPoints.pop());
  };
  return Soothsayer;
})();
```

This class provides a number of convenient methods that alter the state of the world and then run a foretelling. Each of these methods pushes the previous state into the history array, `startingPoints`. There is also a method, `tryADifferentChange`, which undoes the previous state change and gets ready to run another foretelling. The undo is performed by loading back the memento that is stored in an array.

Despite a great pedigree, it is very rare that client-side JavaScript applications provide an undo function. I'm sure there are various reasons for this, but for the most part it is likely that people do not expect such functionality. However, in most desktop applications, having an undo function is expected. I imagine that as a client-side application continues to grow in its capabilities, the undo functionality will become more important. When it does, the memento pattern is a fantastic way of implementing the undo stack.

Observer

The observer pattern is perhaps the most used pattern in the JavaScript world. The pattern is used especially with modern single page applications; it is a big part of the various libraries that provide the **Model-View-ViewModel** (**MVVM**) functionality. We'll explore those patterns in some detail in *Chapter 7*, *Model View Patterns*.

It is frequently useful to know when the value of an object has changed. In order to do so, you could wrap up the property of interest with a getter and a setter:

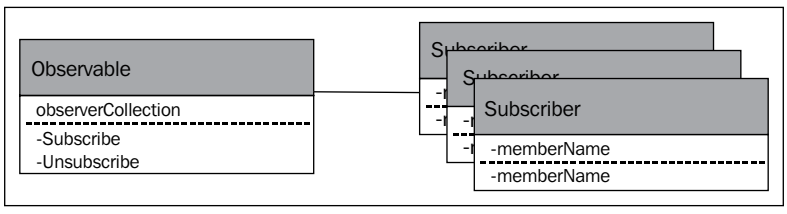
```
var GetterSetter = (function () {
  function GetterSetter() {
  }
  GetterSetter.prototype.GetProperty = function () {
    return this._property;
  };
  GetterSetter.prototype.SetProperty = function (value) {
    this._property = value;
  };
  return GetterSetter;
})();
```

The setter function can now be augmented with a call to some other object that is interested in knowing that a value has changed:

```
GetterSetter.prototype.SetProperty = function (value) {
  var temp = this._property;
  this._property = value;
  this._listener.Event(value, temp);
};
```

This setter will now notify the listener that a property change has occurred. In this case, both the old and new value have been included. This is not necessary as the listener can be tasked with keeping track of the previous value.

The observer pattern generalizes and codifies this idea. Instead of having a single call to the listener, the observer pattern allows interested parties to subscribe to change notifications. The following diagram explains the observer pattern:



Implementation

The court of Westeros is a place of great intrigue and trickery. Controlling who is on the throne and what moves they make is a complex game. Many of the players in the game of thrones employ numerous spies to discover what moves others are making. Frequently, these spies are employed by more than one player and must report what they have found to all of the players.

The spy is a perfect place to employ the observer pattern. In our particular example, the spy being employed is the official doctor to the king and the players are very interested in knowing how much pain killer is being prescribed to the ailing king. This can give a player advanced knowledge of when the king might die—a most useful piece of information.

The spy looks like the following code:

```
var Spy = (function () {
  function Spy() {
    this._partiesToNotify = [];
  }
  Spy.prototype.Subscribe = function (subscriber) {
    this._partiesToNotify.push(subscriber);
  };

  Spy.prototype.Unsubscribe = function (subscriber) {
    this._partiesToNotify.remove(subscriber);
  };

  Spy.prototype.SetPainKillers = function (painKillers) {
    this._painKillers = painKillers;
    for (var i = 0; i < this._partiesToNotify.length; i++) {
      this._partiesToNotify[i](painKillers);
    }
  };
  return Spy;
})();
```

In other languages, the subscriber usually has to comply with a certain interface and the observer will call only the interface method. The encumbrance doesn't exist with JavaScript and, in fact, we just give the `Spy` class a function. This means that there is no strict interface required for the subscriber.

The following code is an example:

```
var Player = (function () {
  function Player() {
  }
  Player.prototype.OnKingPainKillerChange = function
  (newPainKillerAmount) {
    //perform some action
  };
  return Player;
})();
```

This can be used as follows:

```
var s = new Spy();
var p = new Player();
s.Subscribe(p.OnKingPainKillerChange); //p is now a subscriber
s.SetPainKillers(12); //s will notify all subscribers
```

This provides a very simple and highly effective way of building observers. Having subscribers decouples the subscriber from the observable object.

The observer pattern can also be applied to methods as well as properties. In doing so, you can provide hooks for additional behavior to happen. This is a common method of providing a plugin infrastructure for JavaScript libraries.

In browsers, all the event listeners on various items in the DOM are implemented using the observer pattern. For instance, using the popular jQuery library, one can subscribe to all the click events on buttons on a page with the following line:

```
$("body").on("click", "button", function(){/*do something*/})
```

Even in Vanilla JavaScript, the same pattern applies:

```
var buttons = document.getElementsByTagName("button");
for(var i =0; i< buttons.length; i++)
{
  buttons[i].onclick = function(){/*do something*/}
}
```

Clearly, the observer pattern is a very useful one when dealing with JavaScript. There is no need to change the pattern in any significant fashion.

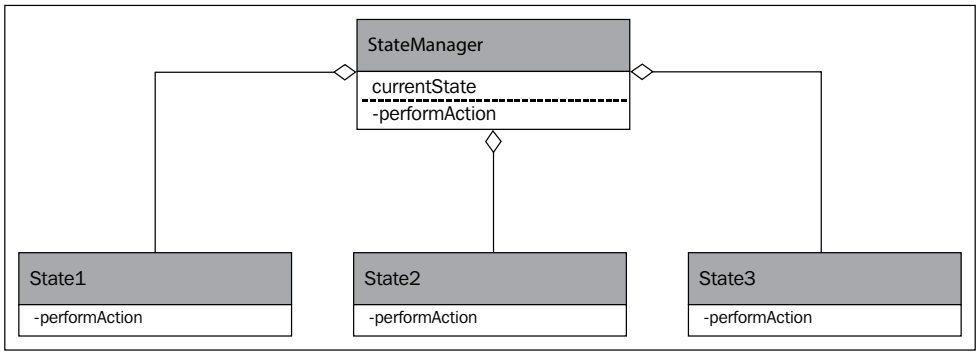
State

State machines are an amazingly useful device in computer programming. Unfortunately, they are not used very frequently by most programmers. I'm sure that at least some of the objection to state machines is that many people implement them as a giant `if` statement, as shown in the following code:

```
function (action, amount) {
  if (this.state == "overdrawn" && action == "withdraw") {
    this.state = "on hold";
  }
  if (this.state == "on hold" && action != "deposit") {
    this.state = "on hold";
  }
  if (this.state == "good standing" && action == "withdraw" &&
  amount <= this.balance) {
    this.balance -= amount;
  }
  if (this.state == "good standing" && action == "withdraw" &&
  amount >this.balance) {
    this.balance -= amount;
    this.state = "overdrawn";
  }
};
```

This is just a sample of what could be a much more complicated workflow. `if` statements of this length are painful to debug and highly error prone. Simply flipping a greater than sign is enough to drastically change how the `if` statement works.

Instead of using a single giant `if` statement block, we can make use of the state pattern. The state pattern is characterized by having a state manager that abstracts away the internal state and proxies message through to the appropriate state that is implemented as a class. All the logic within states and governing state transitions is governed by the individual state classes. The following diagram explains the state pattern:



Splitting state into a class per state allows for much smaller blocks of code to debug and makes testing much easier.

The interface for the state manager is fairly simple and usually just provides the methods needed to communicate with the individual states. The manager may also contain some shared state variables.

Implementation

As alluded to in the `if` statement example, Westeros has a banking system. Much of it is centered on the island of Braavos. Banking there runs in much the same way as banking here, with accounts, deposits, and withdrawals. Managing the state of a bank account involves keeping an eye on all of the transactions and changing the state of the bank account in accordance with the transactions.

Let's take a look at some of the code that is needed to manage a bank account at the Iron Bank of Braavos. First is the state manager:

```
var BankAccountManager = (function () {
  function BankAccountManager() {
    this.currentState = new GoodStandingState(this);
  }
  BankAccountManager.prototype.Deposit = function (amount) {
    this.currentState.Deposit(amount);
  };

  BankAccountManager.prototype.Withdraw = function (amount) {
    this.currentState.Withdraw(amount);
  };
  BankAccountManager.prototype.addToBalance = function (amount) {
    this.balance += amount;
  };
  BankAccountManager.prototype.getBalance = function () {
    return this.balance;
  };
  BankAccountManager.prototype.moveToState = function (newState) {
  this.currentState = newState;
  };
  return BankAccountManager;
})();
```

The bank account manager class provides a state for the current balance and also the current state. To protect the balance, it provides an accessor to read the balance and another to add to the balance. In a real banking application, I would rather expect that the function to set the balance would have more protection than this. In this version of the bank manager, the ability to manipulate the current state is accessible to the states. They have the responsibility to change states. This functionality can be centralized in the manger but that increases the complexity of adding new states.

We've identified three simple states for the bank account: overdrawn, on hold, and good standing. Each one is responsible to deal with withdrawals and deposits when in that state. The `GoodStandingState` class looks like the following code:

```
var GoodStandingState = (function () {
  function GoodStandingState(manager) {
    this.manager = manager;
  }
  GoodStandingState.prototype.Deposit = function (amount) {
    this.manager.addToBalance(amount);
  };
  GoodStandingState.prototype.Withdraw = function (amount) {
    if (this.manager.getBalance() < amount) {
      this.manager.moveToState(new OverdrawnState(this.manager));
    }

    this.manager.addToBalance(-1 * amount);
  };
  return GoodStandingState;
})();
```

The overdrawn state looks like the following code:

```
var OverdrawnState = (function () {
  function OverdrawnState(manager) {
    this.manager = manager;
  }
  OverdrawnState.prototype.Deposit = function (amount) {
    this.manager.addToBalance(amount);
    if (this.manager.getBalance() > 0) {
      this.manager.moveToState(new
      GoodStandingState(this.manager));
    }
  };
```

```
  OverdrawnState.prototype.Withdraw = function (amount) {
    this.manager.moveToState(new OnHold(this.manager));
    throw "Cannot withdraw money from an already overdrawn bank
    account";
  };
  return OverdrawnState;
})();
```

Finally, the `OnHold` state looks like the following code:

```
var OnHold = (function () {
  function OnHold(manager) {
    this.manager = manager;
  }
  OnHold.prototype.Deposit = function (amount) {
    this.manager.addToBalance(amount);
    throw "Your account is on hold and you must go to the bank to
    resolve the issue";
  };
  OnHold.prototype.Withdraw = function (amount) {
    throw "Your account is on hold and you must go to the bank to
    resolve the issue";
  };
  return OnHold;
})();
```

You can see that we've managed to reproduce all the logic of the confusing `if` statement in a number of simple classes. The amount of code here looks to be far more than the `if` statements, but in the long run encapsulating the code into individual classes will pay off.

There is plenty of opportunity to make use of this pattern within JavaScript. Keeping track of the state is a typical problem in most applications. When the transitions between states is complex, then wrapping it up in a state pattern is one method of simplifying things. It is also possible to build up a simple workflow by registering events as sequential. A nice interface for this might be a fluent one so that you could register states as follows:

```
goodStandingState
.on("withdraw")
.when(function(manager){return manager.balance > 0;})
.transitionTo("goodStanding")
.when(function(manager){return mangaer.balance <=0;})
.transitionTo("overdrawn");
```

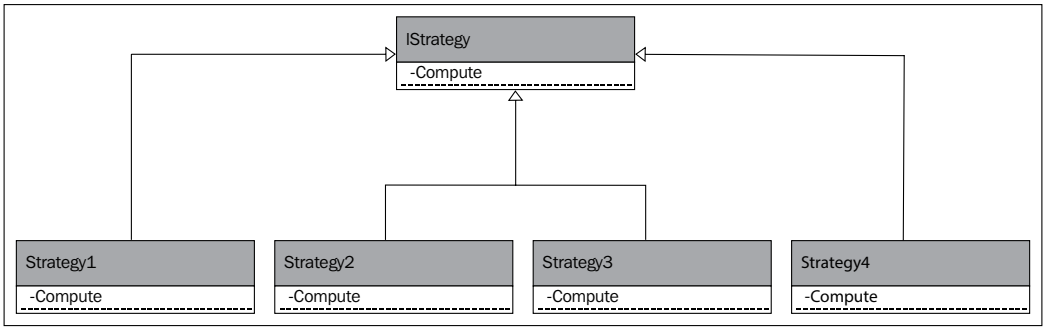
Strategy

It has been said that there is more than one way to skin a cat. I have, wisely, never looked into how many ways there are. The same is frequently true for algorithms in computer programming. Frequently, there are numerous versions of an algorithm that trade off memory usage for CPU usage. Sometimes, there are different approaches that provide different levels of fidelity. For example, performing a geolocation on a smart phone typically uses one of three different sources of data:

- GPS chips
- Cell phone triangulation
- Nearby WiFi points

Using the GPS chip provides the highest level of fidelity; however, it is also the slowest and requires the most battery. Looking at the nearby WiFi points requires very little energy and is very quick; however, it provides poor fidelity.

The strategy pattern provides a method of swapping these strategies out in a transparent fashion. In a traditional inheritance model, each strategy would implement the same interface, which would allow for any of the strategies to be swapped in. The following diagram describes the strategy pattern:



Selecting the correct strategy to use can be done in a number of different ways. The simplest method is to select the strategy statically. This can be done through a configuration variable or can even be hardcoded. This approach is best for times when the strategy changes infrequently or is specific to a single customer or user.

Alternately, an analysis can be run on the dataset on which the strategy is to be run and then a proper strategy selected. If it is known that strategy A works better than strategy B when the data passed in is clustered around a mean, then a fast algorithm to analyze the spread could be run first and then the appropriate strategy selected.

If a particular algorithm fails on data of a certain type, this too can be taken into consideration when choosing a strategy. In a web application, this can be used to call a different API depending on the shape of data. It can also be used to provide a fallback mechanism should one of the API endpoints be down.

Another interesting approach is to use progressive enhancement. The fastest and least accurate algorithm is run first to provide rapid user feedback. At the same time, a slower algorithm is also run, and when it is finished the superior results are used to replace the existing results. This approach is frequently used in the GPS situation outlined earlier. You may notice when using a map on a mobile device that your location is updated a moment after the map loads; this is an example of progressive enhancement.

Finally, the strategy can be chosen completely at random. It sounds like a strange approach but can be useful when comparing the performance of two different strategies. In this case, statistics would be gathered about how well each approach works and an analysis run to select the best strategy. The strategy pattern can be the foundation for A/B testing.

Selecting which strategy to use can be an excellent place to apply the factory pattern.

Implementation

In the land of Westeros, there are no planes, trains, or automobiles but there is still a wide variety of different ways to travel. One can walk, ride a horse, sail on a sea-going vessel, or even take a boat down the river. Each one has different advantages and drawbacks but in the end they still take a person from point A to point B. The interface might look something like the following code:

```
export interface ITravelMethod{
  Travel(source: string, destination: string) : TravelResult;
}
```

The travel result communicates back to the caller some information about the method of travel. In our case, we track how long the trip will take, what the risks are, and how much it will cost:

```
var TravelResult = (function () {
  function TravelResult(durationInDays, probabilityOfDeath, cost)
  {
    this.durationInDays = durationInDays;
    this.probabilityOfDeath = probabilityOfDeath;
    this.cost = cost;
  }
```

```
    return TravelResult;
})();
Travel.TravelResult = TravelResult;
```

In this scenario, we might like to have an additional method that predicts some of the risks, to allow for automating selection of a strategy.

Implementing the strategies is as simple as:

```
var SeaGoingVessel = (function () {
  function SeaGoingVessel() {
  }
  SeaGoingVessel.prototype.Travel = function (source, destination)
  {
    return new TravelResult(15, .25, 500);
  };
  return SeaGoingVessel;
})();

var Horse = (function () {
  function Horse() {
  }
  Horse.prototype.Travel = function (source, destination) {
    return new TravelResult(30, .25, 50);
  };
  return Horse;
})();

var Walk = (function () {
  function Walk() {
  }
  Walk.prototype.Travel = function (source, destination) {
    return new TravelResult(150, .55, 0);
  };
  return Walk;
})();
```

In a traditional implementation of the strategy pattern, the method signature for each strategy should be the same. In JavaScript, there is a bit more flexibility, as excess parameters to a function are ignored and missing parameters can be given default values.

Obviously, the actual calculations around risk, cost, and duration would not be hardcoded in an actual implementation. To make use of these, one needs only to use the following code:

```
var currentMoney = getCurrentMoney();
var strat;
if (currentMoney > 500)
  strat = new SeaGoingVessel();
else if (currentMoney > 50)
  strat = new Horse();
else
  strat = new Walk();
var travelResult = strat.Travel();
```

To improve the level of abstraction for this strategy, we might replace the specific strategies with more generally named ones that describe what it is we're optimizing for:

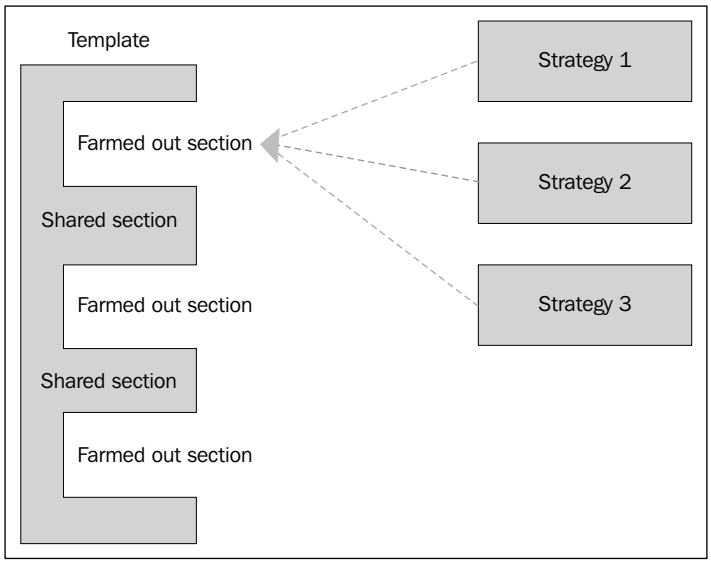
```
var currentMoney = getCurrentMoney();
var strat;
if (currentMoney > 500)
  strat = new FavorFastestAndSafestStrategy();
else
  strat = new FavorCheapest();
var travelResult = strat.Travel();
```

Strategy is a very useful pattern in JavaScript. We're also able to make the approach much simpler than in a language that doesn't use prototype inheritance: there is no need for an interface. We don't need to return the same shaped object from each of the different strategies. So long as the caller is somewhat aware that the returned object may have additional fields, this is a perfectly reasonable, if difficult to maintain, approach.

Template method

The strategy pattern allows replacing an entire algorithm with a complimentary one. Frequently replacing the entire algorithm is overkill: the vast majority of the algorithm remains the same in every strategy with only minor variations in specific sections.

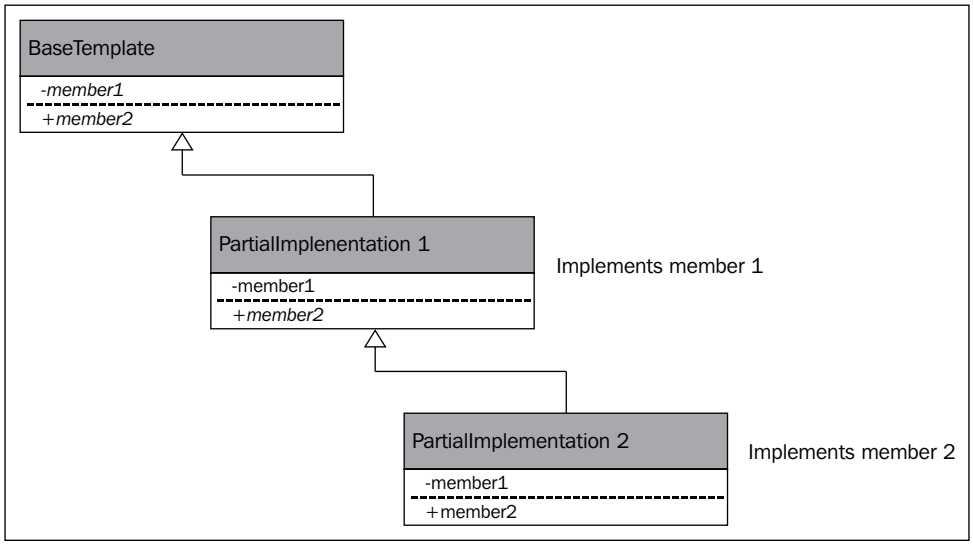
The template method pattern is an approach that allows for some sections of an algorithm to be shared and other sections be implemented using different approaches. These farmed out sections can be implemented by any one of a family of methods. The following diagram describes the template method pattern:



The template class implements parts of the algorithm and leaves other parts as abstract to be overridden later by classes that extend it. The inheritance hierarchy can be several layers deep, with each level implementing more and more of the template class.

An abstract class is one that contains abstract methods. Abstract methods are simply methods that have no body to them. The abstract class cannot be used directly and must, instead, be extended by another class that implements the abstract methods. An abstract class may extend another abstract class so that not all methods need to be implemented by the extending class.

This approach applies the principles of progressive enhancement to an algorithm. We move closer and closer to a fully implemented algorithm, and at the same time build up an interesting inheritance tree. The template method helps keep identical code to a single location while still allowing for some deviation. This diagram shows a chain of partial implementations:



Overriding methods left as abstract is a quintessential part of object-oriented programming. It is likely that this pattern is one which you've used frequently without even being aware that it had a name.

Implementation

I have been told, by those in the know, that there are many different ways to produce beer. These beers differ in their choice of ingredients and in their method of production. In fact, beer does not even need to contain hops—it can be made from any number of grains. However, there are similarities between all beers. They are all created through the fermentation process and all proper beers contain some amount of alcohol.

In Westeros, there are a large number of craftsmen who pride themselves on creating top notch beers. We would like to describe their processes as a set of classes, each one describing a different beer-making methodology. We start with a simplified implementation of creating beer:

```
var BasicBeer = (function () {
  function BasicBeer() {
  }
  BasicBeer.prototype.Create = function () {
    this.AddIngredients();
    this.Stir();
    this.Ferment();
    this.Test();
    if (this.TestingPassed()) {
      this.Distribute();
    }
  };

  BasicBeer.prototype.AddIngredients = function () {
    throw "Add ingredients needs to be implemented";
  };

  BasicBeer.prototype.Stir = function () {
    //stir 15 times with a wooden spoon
  };

  BasicBeer.prototype.Ferment = function () {
    //let stand for 30 days
  };

  BasicBeer.prototype.Test = function () {
    //draw off a cup of beer and taste it
  };

  BasicBeer.prototype.TestingPassed = function () {
    throw "Conditions to pass a test must be implemented";
  };

  BasicBeer.prototype.Distribute = function () {
    //place beer in 50L casks
  };
  return BasicBeer;
})();
```

As there is no concept of abstract in JavaScript, we've added exceptions to the various methods that must be overridden. The remaining methods can be changed but do not require it. An implementation of this for raspberry beer would look like the following code:

```
var RaspberryBeer = (function (_super) {
  __extends(RaspberryBeer, _super);
  function RaspberryBeer() {
    _super.apply(this, arguments);
  }
  RaspberryBeer.prototype.AddIngredients = function () {
    //add ingredients, probably including raspberries
  };

  RaspberryBeer.prototype.TestingPassed = function () {
    //beer must be reddish and taste of raspberries
  };
  return RaspberryBeer;
})(BasicBeer);
```

Additional subclassing may be performed at this stage for more specific raspberry beers.

The template method remains a fairly useful pattern in JavaScript. There is some added syntactic sugar around creating classes, but it isn't anything we haven't already seen in a previous chapter. The only warning I would give is that the template method uses inheritance and thus strongly couples the inherited classes with the parent class. This is generally not a desirable state of affairs.

Visitor

The final pattern in this section is the visitor pattern. Visitor provides for a method of decoupling an algorithm from the object structure on which it operates. If we wanted to perform some action over a collection of objects that differ in type and we want to perform a different action depending on the object type, we would typically need to make use of a large `if` statement.

Let's get right into an example of this in Westeros. An army is made up of a few different classes of fighting persons (it is important that we be politically correct as there are many notable female fighters in Westeros). However, each member of the army implements a hypothetical interface called `IMemberOfArmy`:

```
interface IMemberOfArmy{
  printName();
}
```

A simple implementation of this might be:

```
var Knight = (function () {
  function Knight() {
  }
  Knight.prototype.printName = function () {
    console.log("Knight");
  };
  return Knight;
})();
```

Now that we have a collection of these different types, we can use an `if` statement to only call the `printName` function on the knights:

```
var collection = [];
collection.push(new Knight());
collection.push(new FootSoldier());
collection.push(new Lord());
collection.push(new Archer());

for (var i = 0; i < collection.length; i++) {
  if (typeof (collection[i]) == 'Knight')
    collection[i].printName();
  else
    console.log("Not a knight");
}
```

Except if you run the preceding code, you'll actually find that all we get is:

```
Not a knight
Not a knight
Not a knight
Not a knight
```

This is because, despite an object being a knight, it is still an object and `typeof` will return the object in all cases.

An alternative approach is to use `instanceof` instead of `typeof`:

```
var collection = [];
collection.push(new Knight());
collection.push(new FootSoldier());
collection.push(new Lord());
collection.push(new Archer());
```

```
for (var i = 0; i < collection.length; i++) {
  if (collection[i] instanceof Knight)
    collection[i].printName();
  else
    console.log("No match");
}
```

The instance of approach works great until we run into somebody who makes use of the `Object.create` syntax:

```
collection.push(Object.create(Knight));
```

Despite being the `Knight` class, this will return `false` when asked if it is an instance of the `Knight` class.

This poses something of a problem for us. The problem is exacerbated by the visitor pattern as it requires that the language support method overloading. JavaScript does not really support this. There are various hacks that can be used to make JavaScript somewhat aware of overloaded methods, but the usual advice is to simply not bother and create methods with different names.

Let's not abandon this pattern just yet, though; it is a useful pattern. What we need is a way to reliably distinguish one type from another. The simplest approach is to just define a variable on the class that denotes its type:

```
var Knight = (function () {
  function Knight() {
    this._type = "Knight";
  }
  Knight.prototype.printName = function () {
    console.log("Knight");
  };
  return Knight;
})();
```

Given the new `_type` variable, we can now fake having real method overrides:

```
var collection = [];
collection.push(new Knight());
collection.push(new FootSoldier());
collection.push(new Lord());
collection.push(new Archer());

for (var i = 0; i < collection.length; i++) {
  if (collection[i]._type == 'Knight')
collection[i].printName();
  else
console.log("No match");
}
```

Given this approach, we can now implement the visitor pattern. The first step is to expand the various members of our army to have a generic method on them that takes a `visitor` and applies it:

```
var Knight = (function () {
  function Knight() {
    this._type = "Knight";
  }
  Knight.prototype.printName = function () {
    console.log("Knight");
  };
  Knight.prototype.visit = function (visitor) {
    visitor.visit(this);
  };
  return Knight;
})();
```

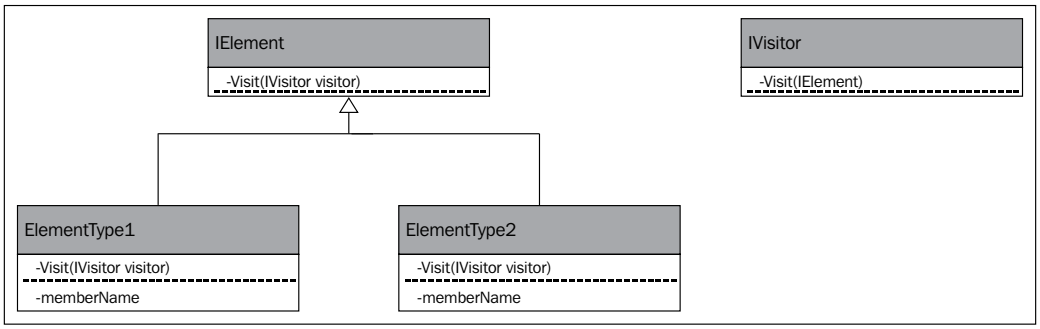
Now, we need to build a visitor. The following code approximates the `if` statements we had in the previous code:

```
var SelectiveNamePrinterVisitor = (function () {
  function SelectiveNamePrinterVisitor() {
  }
  SelectiveNamePrinterVisitor.prototype.Visit = function
  (memberOfArmy) {
    if (memberOfArmy._type == "Knight") {
      this.VisitKnight(memberOfArmy);
    } else {
      console.log("Not a knight");
    }
  };

  SelectiveNamePrinterVisitor.prototype.VisitKnight = function
  (memberOfArmy) {
    memberOfArmy.printName();
  };
  return SelectiveNamePrinterVisitor;
})();
```

This visitor would be used as such:

```
var collection = [];
collection.push(new Knight());
collection.push(new FootSoldier());
collection.push(new Lord());
collection.push(new Archer());
var visitor = new SelectiveNamePrinterVisitor();
for (var i = 0; i < collection.length; i++) {
  collection[i].visit(visitor);
}
```

As you can see, we've pushed the decisions about what is the type of the item in the collection down to the visitor. This decouples the items themselves from the visitor, as shown in the following diagram:



If we allow the visitor itself to make decisions about what methods are called on the visited objects, there is a fair bit of trickery required. If we can provide a constant interface for the visited objects, then all the visitor needs do is call the interface method. This does, however, move logic from the visitor into the objects that are visited, which is contrary to the idea that the objects shouldn't know that they are part of a visitor.

Whether suffering through the trickery is worthwhile or not, it is definitely an exercise for you. Personally, I would tend to avoid using the visitor pattern in JavaScript, as the requirements to get it working are complicated and non-obvious.

Hints and tips

Here are a couple of brief tips to keep in mind about some of the patterns we've seen in this chapter:

- When implementing the interpreter pattern, you may be tempted to use JavaScript properly as your DSL and then use the `eval` function to execute the code. This is actually a very dangerous idea as `eval` opens up an entire world of security issues. It is generally considered to be very bad form to use `eval` in JavaScript.
- If you find yourself in a position to audit the changes to data in your project, then the memento pattern can easily be modified to suit this. Instead of just keeping track of the state changes, you can also track when the change was made and who made the change. Saving these mementos to disk somewhere allows you to go back and, rapidly, build an audit log pointing to precisely what happened to change the object.
- The observer pattern is notorious as it causes memory leaks when listeners aren't properly unregistered. This can happen even in a memory managed environment such as JavaScript. Be wary of failing to unhook observers.

Summary

In this chapter, we've looked at a bunch of behavioral patterns. Some of these patterns, such as observers and iterators, will be ones you'll use almost every day, while others, such as interpreters, you might use no more than a handful of times in your entire career. Learning about these patterns should help you identify well-defined solutions to common problems.

Most of the patterns are directly applicable to JavaScript and some of them, such as the strategy pattern, become more powerful in a dynamic language. The only pattern we found that has some limitations is the visitor pattern. The lack of static classes and polymorphism makes this pattern difficult to implement without breaking proper separation of concerns.

These are, by no means, all of the behavioral patterns in existence. The programming community has spent the past two decades building on the ideas of the GoF book and identifying new patterns. The remainder of this book is dedicated to these newly identified patterns. The solutions may be very old ones but were not generally recognized as common solutions until more recently. As far as I'm concerned, this is the point where the book starts to get very interesting, as we start looking at less well-known and more JavaScript-specific patterns.

Part 2

Other Patterns

Functional Programming

Model View Patterns

Web Patterns

Messaging Patterns

Patterns for Testing

Advanced Patterns

ES6 Solutions Today

In *Part 1*, we focused on patterns that were identified in the GoF book that was the original impetus behind patterns in software design. In this part of the book, we'll expand beyond those patterns to look at patterns that are related to functional programming, large scale patterns to structure an entire application, patterns that are specific to the web, and messaging patterns. In addition, we'll look at patterns for the purpose of testing and some rather interesting advanced patterns. Finally, we'll look at how we can get many of the features of the next version of JavaScript today.

6
Functional Programming

Functional programming is a different approach to development than the heavily object-oriented approach that we have focused on so far. Object-oriented programming is a fantastic tool to solve a great number of problems but it has some issues. Parallel programming within an object-oriented context is difficult, as the state can be changed by various different threads with unknown side effects. Functional programming does not permit state or mutable variables. Functions act as primary building blocks in functional programming. Places where you might have used a variable in the past will now use a function.

Even in a single-threaded program, functions can have side effects that change the global state. This means that calling an unknown function can alter the whole flow of the program. This makes debugging a program quite difficult.

JavaScript is not a functional programming language but we can still apply some functional principles to our code. We'll look at a number of patterns in the functional space:

- Function passing
- Filters and pipes
- Accumulators
- Memoization
- Immutability
- Lazy instantiation

Functional functions are side-effect free

A core tenant of functional programming is that functions should not change state. Values local to the function may be set but nothing outside of the function may change. This approach is very useful to make code more maintainable. There need no longer be any concern that passing an array into a function might play havoc with its contents. This is especially a concern when using libraries that are not under your control.

There is no mechanism within JavaScript to prevent you from changing the global state. Instead, you must rely on developers to write side-effect free functions. This could be difficult depending on the maturity of the team.

It may not be desirable to put all of the code from your application into functions, but separating as much as possible is desirable. There is a pattern called command query separation that suggests that methods should fall into two categories. Either a method is a function that reads a value or it is a command that sets a value. Never the twain should meet. Keeping methods categorized like this eases in debugging and in code reuse.

One of the consequences of side-effect free functions is that they can be called any number of times with the same inputs and the result will be the same. Furthermore, because there are no changes to state, calling the function many times will not cause any ill side effects, other than running slower.

Function passing

In functional programming languages, functions are first-class citizens. Functions can be assigned to variables and passed around just like you would with any other variable. This is not entirely a foreign concept. Even languages such as C had function pointers that could be treated just like other variables. C# has delegates and, in more recent versions, lambdas. The latest release of Java has also added support for lambdas, as they have proven to be so useful.

JavaScript allows for functions to be treated as variables and even as objects and strings. In this way, JavaScript is functional in nature.

Because of JavaScript's single-threaded nature, callbacks are a common convention and you can find them pretty much everywhere. Consider calling a function at a later date on a web page. This is done by setting a timeout on the window object as follows:

```
setTimeout(function(){alert("Hello from the past")}, 5 * 1000);
```

The arguments for the set timeout function are a function to call and a time to delay in milliseconds.

No matter the JavaScript environment in which you're working, it is almost impossible to avoid functions in the shape of callbacks. The asynchronous processing model of Node.js is highly dependent on being able to call a function and pass in something to be completed at a later date. Making calls to external resources in a browser is also dependent on a callback to notify the caller that some asynchronous operation has completed. In basic JavaScript, this looks like the following code:

```
var xmlhttp = new XMLHttpRequest()
xmlhttp.onreadystatechange=function()
if (xmlhttp.readyState==4 &&xmlhttp.status==200){
  //process returned data
}
};
xmlhttp.open("GET", http://some.external.resource, true);
xmlhttp.send();
```

You may notice that we assign `onreadystatechange` before we even send the request. This is because assigning it later may result in a race condition in which the server responds before the function is attached to the ready state change. In this case, we've used an inline function to process the returned data. Because functions are first class citizens, we can change this to look like the following code:

```
var xmlhttp;
function requestData(){
  xmlhttp = new XMLHttpRequest()
  xmlhttp.onreadystatechange=processData;
  xmlhttp.open("GET", http://some.external.resource, true);
  xmlhttp.send();
}

function processData(){
  if (xmlhttp.readyState==4 &&xmlhttp.status==200){
    //process returned data
  }
}
```

This is typically a cleaner approach and avoids performing complex processing in line with another function.

However, you might be more familiar with the jQuery version of this, which looks something like this:

```
$.getJSON('http://some.external.resource', function(json){
  //process returned data
});
```

In this case, the boiler plate of dealing with ready state changes is handled for you. There is even convenience provided for you should the request for data fail with the following code:

```
$.ajax('http://some.external.resource',
  { success: function(json){
    //process returned data
  },
  error: function(){
    //process failure
  },
dataType: "json"
});
```

In this case, we've passed an object into the `ajax` call, which defines a number of properties. Amongst these properties are function callbacks for success and failure. This method of passing numerous functions into another suggests a great way of providing expansion points for classes.

Likely, you've seen this pattern in use before without even realizing it. Passing functions into constructors as part of an options object is a commonly used approach to providing extension hooks in JavaScript libraries. We have seen some treatment of functions in *Chapter 5*, *Behavioral Patterns*, when passing the function into the observer.

Implementation

In Westeros, the tourism industry is almost nonextant. There are great difficulties with bandits killing tourists and tourists becoming entangled in regional conflicts. Nonetheless, some enterprising folks have started to advertise a grand tour of Westeros in which they will take those with the means on a tour of all the major attractions. From King's Landing to Eyrie, to the great mountains of Dorne, the tour will cover it all. In fact, a rather mathematically inclined member of the tourism board has taken to calling it a Hamiltonian tour, as it visits everywhere once.

The `HamiltonianTour` class provides an `options` object that allows the definition of an `options` object. This object contains the various places to which a callback can be attached. In our case, the interface for it would look something like the following code:

```
export class HamiltonianTourOptions{
  onTourStart: Function;
  onEntryToAttraction: Function;
  onExitFromAttraction: Function;
  onTourCompletion: Function;
}
```

The full `HamiltonianTour` class looks like the following code:

```
var HamiltonianTour = (function () {
  function HamiltonianTour(options) {
    this.options = options;
  }
  HamiltonianTour.prototype.StartTour = function () {
    if (this.options.onTourStart&&typeof (this.options.onTourStart)
    === "function")
    this.options.onTourStart();
    this.VisitAttraction("King's Landing");
    this.VisitAttraction("Winterfell");
    this.VisitAttraction("Mountains of Dorne");
    this.VisitAttraction("Eyrie");
    if (this.options.onTourCompletion&&typeof
    (this.options.onTourCompletion) === "function")
    this.options.onTourCompletion();
  };

  HamiltonianTour.prototype.VisitAttraction = function
  (AttractionName) {
    if (this.options.onEntryToAttraction&&typeof
    (this.options.onEntryToAttraction) === "function")
    this.options.onEntryToAttraction(AttractionName);

    //do whatever one does in a Attraction
    if (this.options.onExitFromAttraction&&typeof
    (this.options.onExitFromAttraction) === "function")
    this.options.onExitFromAttraction(AttractionName);
  };
  return HamiltonianTour;
})();
```

You can see in the highlighted code how we check the options and then execute the callback as needed. This can be done by simply using the following code:

```
var tour = new HamiltonianTour({
  onEntryToAttraction: function(cityname){console.log("I'm
  delighted to be in " + cityname)}});
tour.StartTour();
```

The output of the preceding code will be:

```
I'm delighted to be in King's Landing
I'm delighted to be in Winterfell
I'm delighted to be in Mountains of Dorne
I'm delighted to be in Eyrie
```

Passing functions is a great approach to solving a number of problems in JavaScript and tends to be used extensively by libraries such as jQuery and frameworks such as Express. It is so commonly adopted that using it provides no added barriers to your code's readability.

Filters and pipes

If you're at all familiar with the Unix command line, or to a lesser extent, the Windows command line, then you'll have probably made use of pipes. A pipe, which is represented by the `|` character, is short hand for "take the output of program A and put it into program B." This relatively simple idea makes the Unix command line incredibly powerful. For instance, if you wanted to list all the files in a directory and then sort them and filter for any files that start with either the letter b or g and end with an f, then the command might look like this:

```
ls|sort|grep "^[gb].*f$"
```

The `ls` command lists all files and directories, the `sort` command sorts them, and the `grep` command matches filenames against a regular expression. Running these commands in the `etc` directory on an Ubuntu box in `/etc` would give a result that looks something like this:

```
stimms@ubuntu1:/etc$ ls|sort|grep "^[gb].*f$"
blkid.conf
bogofilter.cf
brltty.conf
```

gai.conf

gconf

groff

gssapi_mech.conf

Some functional programming languages such as F# offer a special syntax for piping between functions. In F# filtering, a list for the even numbers can be done as follows:

```
[1..10] |>List.filter (fun n -> n% 2 = 0);;
```

This syntax is very nice looking especially when used for long chains of functions. As an example, taking a number, casting it to a float, square rooting it, and then rounding it would look like this:

```
10.5 |> float |>Math.Sqrt |>Math.Round
```

This is a clearer syntax than the C-style syntax, which would look more like this:

```
Math.Round(Math.Sqrt((float)10.5))
```

Unfortunately, there is no ability to write pipes in JavaScript using a nifty F# style syntax, but we can still improve upon the preceding normal method by using method chaining.

Everything in JavaScript is an object, which means that we can have some real fun adding functionality to the existing objects to improve their look. Operating on collections of objects is a space in which functional programming provides some powerful features. Let's start by adding a simple filtering method to the array object. You can think of these queries as being like SQL database queries written in a functional fashion.

Implementation

We would like to provide a function that performs a match against each member of the array and returns a set of results:

```
Array.prototype.where = function (inclusionTest) {
  var results = [];
  for (var i = 0; i<this.length; i++) {
    if (inclusionTest(this[i]))
    results.push(this[i]);
  }
  return results;
};
```

The rather simple looking function allows us to quickly filter an array:

```
var items = [1,2,3,4,5,6,7,8,9,10];
items.where(function(thing){ return thing % 2 ==0;});
```

What we return is also an object, an array object in this case; we can continue to chain methods onto it like this:

```
items.where(function(thing){ return thing % 2 ==0;})
  .where(function(thing){ return thing % 3 == 0;});
```

The result of this is an array containing only the number 6, as it is the only number between 1 and 10 which is both even and divisible by three. This method of returning a modified version of the original object without changing the original is known as a **fluent interface**. By not changing the original items array, we've introduced a small degree of immutability into our variables.

If we add another function to our library of `Array` extensions, we can start to see how useful these pipes can be:

```
Array.prototype.select=function(projection){
  var results = [];
  for(var i = 0; i<this.length;i++){
    results.push(projection(this[i]));
  }
  return results;
};
```

This extension allows for projections of the original items based on an arbitrary projection function. Given a set of objects that contain IDs and names, we can use our fluent extensions to arrays to perform complex operations:

```
var children = [{ id: 1, Name: "Rob" },
{ id: 2, Name: "Sansa" },
{ id: 3, Name: "Arya" },
{ id: 4, Name: "Brandon" },
{ id: 5, Name: "Rickon" }];
var filteredChildren = children.where(function (x) {
  return x.id % 2 == 0;
}).select(function (x) {
  return x.Name;
});
```

This code will build a new array that contains only children with even IDs, and instead of full objects the array will contain only their names: `"Sansa"` and `"Brandon"`. For those familiar with .NET, these functions may look very familiar. The **Language Integrated Queries** (**LINQ**) library on .NET provides similarly named functional-inspired functions for the manipulation of collections.

Chaining functions in this manner can be both easier to understand and easier to build than alternatives: temporary variables are avoided and the code made terser. Consider the previous example reimplemented using loops and temporary variables:

```
var children = [{ id: 1, Name: "Rob" },
{ id: 2, Name: "Sansa" },
{ id: 3, Name: "Arya" },
{ id: 4, Name: "Brandon" },
{ id: 5, Name: "Rickon" }];
var evenIds = [];
for(var i=0; i<children.length; i++)
{
  if(children[i].id%2==0)
    evenIds.push(children[i]);
}
var names = [];
for(var i=0; i< evenIds.length; i++)
{
  names.push(evenIds[i].name);
}
```

A number of JavaScript libraries such as d3 are constructed to encourage this sort of programming. At first, it seems like the code created following this convention is bad due to very long line length. I would argue that this is a function of line length not being a very good tool to measure complexity rather than an actual problem with the approach.

Accumulators

We've looked at some simple array functions that add filtering and pipes to arrays. Another useful tool is the accumulator. Accumulators aide in building up a single result by iterating over a collection. Many common operations such as summing up the elements of an array can be implemented using an accumulator instead of a loop.

Recursion is popular within functional programming languages and many of them actually offer an optimization called **tail call optimization**. A language that supports this provides optimizations for functions that use recursion in which the stack frame is reused. This is very efficient and can easily replace most loops. Details on whether tail call optimization is supported in any JavaScript interpreter are sketchy. For the most part, it doesn't seem like it is, but we can still make use of recursion.

The problem with `for` loops is that the control flow through the loop is mutable. Consider the following code; it rather easy to make a mistake:

```
var result = "";
var multiArray = [[1,2,3], ["a", "b", "c"]];
for(var i=0; i<multiArray.length; i++)
  for(var j=0; i<multiArray[i].length; j++)
    result += multiArray[i][j];
```

Did you spot the error? It took me several attempts to get a working version of this code I could break. The problem is in the loop counter in the second loop; it should read:

```
var result = "";
var multiArray = [[1,2,3], ["a", "b", "c"]];
for(var i=0; i<multiArray.length; i++)
  for(var j=0; j<multiArray[i].length; j++)
    result +=multiArray[i][j];
```

Obviously, this could be somewhat mitigated through better variable naming but we would like to avoid the problem completely.

Instead we can make use of an accumulator, a tool used to combine multiple values from a collection into a single value. We've rather missed Westeros for a couple of patterns, so let's get back to our mythical example land. Wars cost a great deal of money, but fortunately there are a great number of peasants to pay taxes and finance the lords in their games for the throne.

Implementation

Our peasants are represented by a simple model that looks like this:

```
var peasants = [
{name: "Jory Cassel", taxesOwed: 11, bankBalance: 50},
{name: "Vardis Egen", taxesOwed: 15, bankBalance: 20}];
```

Over this set of peasants, we have an accumulator that looks like this:

```
TaxCollector.prototype.collect = function (items, value,
projection) {
  if (items.length > 1)
    return projection(items[0]) + this.collect(items.slice(1),
    value, projection);
  return projection(items[0]);
};
```

This code takes a list of items, an accumulator value, and a function that projects the value to be integrated into the accumulation.

The projection function looks something like this:

```
function (item) {
  return Math.min(item.moneyOwed, item.bankBalance);
}
```

In order to prime this function, we simply need to pass in an initial value for the accumulator along with the array and projection. The priming value will vary but more often than not it will be an identity; an empty string in the case of a string accumulator and a 0 or 1 in the case of mathematical ones.

Each pass through the accumulator shrinks the size of the array over which we are operating. All this is done without a single mutable variable.

The inner accumulation can really be any function such as string appending, addition, or something more complicated. The accumulator is somewhat like the visitor pattern except that modifying values in the collection inside an accumulator is frowned upon. Remember that functional programming is side-effect free.

Memoization

Not to be confused with memorization, memoization is a specific term to retain a number of previously calculated values from a function.

As we have seen earlier, side-effect free functions can be called multiple times without causing problems. The corollary to this is that a function can also be called fewer times than needed. Consider an expensive function that does some complex or, at least, time-consuming math. We know that the result of the function is entirely predicated on the inputs to the function. So the same inputs will always produce the same outputs. Why, then, would we need to call the function multiple times? If we saved the output of the function, we could retrieve that instead of redoing the time-consuming math.

Trading off space for time is a classic computing science problem. By caching the result, we make the application faster but we will consume more memory. Deciding when to perform caching and when to simply recalculate the result is a difficult problem.

Implementation

In the land of Westeros, learned men—known as Maesters—have long had a fascination with a sequence of numbers which seems to reappear a great deal in the natural world. In a strange coincidence, they call this sequence the Fibonacci sequence. It is defined by adding the two previous terms in the sequence to get the next one. The sequence is bootstrapped by defining the first few terms as 0, 1, and 1. So, to get the next term, we would simply add 1 and 1 to get 2. The next term would add 2 and 1 to get 3, and so forth. Finding an arbitrary member of the sequence requires finding the two previous, so it can end up being quite a bit of a calculation.

In our world, we have discovered a closed form which avoids much of this calculation, but in Westeros no such discovery has been made.

A naive approach is to simply calculate every term like this:

```
var Fibonacci = (function () {
  function Fibonacci() {
  }
  Fibonacci.prototype.NaieveFib = function (n) {
    if (n == 0)
    return 0;
    if (n <= 2)
    return 1;
    return this.NaieveFib(n - 1) + this.NaieveFib(n - 2);
  };
  return Fibonacci;
})();
```

This solution works very quickly for small numbers such as 10. However, for larger numbers, say greater than 40, there is a substantial slowdown. This is because the base case is called 102,334,155 times.

Let's see if we can improve things by memoizing some values:

```
var Fibonacci = (function () {
  function Fibonacci() {
    this.memoizedValues = [];
  }
```

```
    Fibonacci.prototype.MemetoFib = function (n) {
      if (n == 0)
        return 0;
      if (n <= 2)
        return 1;
      if (!this. memoizedValues[n])
        this. memoizedValues[n] = this.MemetoFib(n - 1) +
        this.MemetoFib(n - 2);
      return this. memoizedValues[n];
    };
    return Fibonacci;
  })();
```

We have just memoized every item we encounter. As it turns out for this algorithm, we store $n + 1$ items, which is a pretty good trade off. Without memoization, calculating the 40th Fibonacci number took 963 milliseconds, while the memoization version took only 11 milliseconds. The difference is far more pronounced when the functions become more complex to calculate. The 140th Fibonacci number took 12 milliseconds for the memoization version while the naive version took… well, it is has been a day and it is still running.

The best part of this memoization is that subsequent calls to the function with the same parameter will be lightning fast, as the result is already computed.

In our example, only a very small cache was needed. In more complex examples, it is difficult to know how large a cache should be or how frequently a value will need to be recomputed. Ideally, your cache will be large enough that there will always be room to put more results in. However, this may not be realistic and tough decisions will need to be made about which members of the cache should be removed to save space. There are a plethora of methods to perform cache invalidation. It has been said that cache invalidation is one of the toughest problems in computing science, the reason being that we're effectively trying to predict the future. If anybody has perfected a method of telling the future, it is likely that they are applying their skills in a more important domain than cache invalidation. Two options are to prey on the least recently used member of the cache or the least frequently used member. It is possible that the shape of the problem may dictate a better strategy.

Memoization is a fantastic tool to speed up calculations that need to be performed multiple times or even calculations that have common subcalculations. One can consider memoization as just a special case of caching, which is a commonly used technique when building web servers or browsers. It is certainly worthwhile exploring in more complex JavaScript applications.

Immutability

One of the cornerstones of functional programming is that the so-called variables can be assigned only once. This is known as **immutability**. Currently, there is no real support for immutability in JavaScript. However, ECMAScript 6 will support a new keyword: `const`. The `const` keyword can be used in the same way as `var` except that variables assigned with `const` will be immutable. For instance, the following code shows a variable and `const` that are both manipulated in the same way:

```
var numberOfQueens = 1;
const numberOfKings = 1;
numberOfQueens++;
numberOfKings++;
console.log(numberOfQueens);
console.log(numberOfKings);
```

The output of running the preceding code is:

```
1
2
```

As you can see, the results for `const` and `var` are different.

I have never seen a reliable way to simulate this behavior in the current editions of JavaScript. At the time of this writing, there is limited support for `const` on certain browsers. A possible approach is to make use of the `Object.freeze` functionality, which is more widely adopted:

```
var consts = Object.freeze({ pi : 3.141});
consts.pi = 7;
console.log(consts.pi);//outputs 3.141
```

As you can see, the syntax here is not very user friendly. Also, an issue is that attempting to assign to an already assigned `const` keyword simply fails silently instead of throwing an error. Failing silently in this fashion is not at all a desirable behavior; a full exception should be thrown. If you enable the strict mode, a more rigorous parsing mode added in ECMAScript 5, then an exception is actually thrown:

```
"use strict";
var consts = Object.freeze({ pi : 3.141});
consts.pi = 7;
```

This will throw the following exception:

```
consts.pi = 7;
          ^
TypeError: Cannot assign to read only property 'pi' of #<Object>
```

An alternative is the `Object.create` syntax we spoke about earlier. When creating properties on the object, one can specify `writable: false` to make the property immutable:

```
var t = Object.create(Object.prototype,
{ value: { writable: false,
  value: 10}
});
t.value = 7;
console.log(t.value);//prints 10
```

However, even in strict mode, no exception is thrown when attempting to write to a nonwritable property. Thus, I would claim that the `const` keyword is not perfect to implement immutable objects. You're better off using freeze.

Lazy instantiation

If you go into a high-end coffee shop and place an order for some overly complex beverage (Grande Chai Tea Latte, Three Pump, Skim Milk, Lite Water, No Foam, Extra Hot anybody?), then that beverage is going to be made on the fly and not in advance. Even if the coffee shop knew what all the orders that were going to come in that day would be, they would still not make all the beverages up front. Firstly, because it would result in a large number of ruined, cold beverages, and secondly, it would be a very long time for the first customer to get their order if they had to wait for all the orders of the day to be completed.

Instead coffee shops, follow a just-in-time approach to craft beverages. They make them when they're ordered. We can apply a similar approach to our code through the use of a technique known as lazy instantiation or lazy initialization.

Consider an object that is expensive to create, that is to say that it takes a great deal of time to create the object. If we are unsure if the object's value will be needed, we can defer its full creation until later.

Implementation

Let's jump into an example of this. Westeros isn't really big on expensive coffee shops but they do love a good bakery. This bakery takes requests for different bread types in advance and then bakes them all at once should they get an order. However, creating the bread object is an expensive operation, so we would like to defer that until somebody actually comes to pick up the bread:

```
var Bread = (function () {
  function Bread(breadType) {
```

```
    this.breadType = breadType;
    //some complex, time consuming operation
    console.log("Bread " + breadType + " created.");
  }
  return Bread;
})();
```

We start by creating a list of bread types to create as needed. This list is appended to by ordering a bread type:

```
var Bakery = (function () {
  function Bakery() {
    this.requiredBreads = [];
  }
  Bakery.prototype.orderBreadType = function (breadType) {
    this.requiredBreads.push(breadType);
  };
```

This allows for breads to be rapidly added to the required breads list without paying the price for each bread to be created.

Now, when `pickUpBread` is called, we'll actually create the breads:

```
Bakery.prototype.pickUpBread = function (breadType) {
  console.log("Pickup of bread " + breadType + " requested");
  if (!this.breads) {
    this.createBreads();
  }
  for (var i = 0; i<this.breads.length; i++) {
    if (this.breads[i].breadType == breadType)
    return this.breads[i];
  }
};
Bakery.prototype.createBreads = function () {
  this.breads = [];
  for (var i = 0; i<this.requiredBreads.length; i++) {
    this.breads.push(new Bread(this.requiredBreads[i]));
  }
};
```

Then we will call a series of operations:

```
var bakery = new Westeros.FoodSuppliers.Bakery();
bakery.orderBreadType("Brioche");
bakery.orderBreadType("Anadama bread");
```

```
bakery.orderBreadType("Chapati");
bakery.orderBreadType("Focaccia");

console.log(bakery.pickUpBread("Brioche").breadType + "picked
up");
```

This will result in the following output:

```
Pickup of bread Brioche requested.
Bread Brioche created.
Bread Anadama bread created.
Bread Chapati created.
Bread Focaccia created.
Brioche picked up
```

You can see that the collection of actual breads is left until after the pickup has been requested.

Lazy instantiation can be used to simplify asynchronous programming. Promises are an approach to simplifying callbacks, which are common in JavaScript. Instead of building up complicated callbacks, a promise is an object that contains a state and a result. When first called, the promise is in an unresolved state; once the `async` operation completes, the state is updated to complete and the result filled in. You can think of the result as being lazily instantiated. We'll look at promises and promise libraries in more detail in *Chapter 8*, *Web Patterns*.

Being lazy can save you quite a bit of time in creating expensive objects that end up never being used.

Hints and tips

Although callbacks are the standard way of dealing with asynchronous methods in JavaScript, they can get out of hand easily. There are a number of approaches to solve this spaghetti code: promise libraries provide a more fluent way of handling callbacks and future versions of JavaScript may adopt an approach similar to the C# `async/await` syntax.

I really like accumulators but they can be inefficient in terms of memory use. The lack of tail recursion means that each pass through adds another stack frame, so this approach may result in memory pressure. All things are a trade off in this case between memory and code maintainability.

Summary

JavaScript is not a functional programming language. That is not to say that it isn't possible to apply some of the ideas from functional programming to it. These approaches enable cleaner, easier-to-debug code. Some might even argue that the number of issues will be reduced, although I have never seen any convincing studies on that.

In this chapter, we looked at six different patterns. Lazy instantiation, memoization, and immutability are all creational patterns. Function passing is a structural pattern as well as a behavioral one. Accumulators are also behavioral in nature. Filters and pipes don't really fall into any of the GoF categories, so one might think of these as a style pattern.

In the next chapter, we'll look at a number of patterns to divide the logic and presentation in applications. These patterns have become more important as JavaScript applications have grown.

7
Model View Patterns

Thus far, we have spent a great deal of time examining patterns that are used to solve local problems, that span only a handful of classes and not the whole application. These patterns have been narrow in scope. They frequently only relate to two or three classes and might be used but a single time in any given application. As you can imagine, there are also larger scale patterns that are applicable to the application as a whole. You might think of *toolbar* as a general pattern, which is used in many places in an application. What's more, it is a pattern that is used in a great number of applications to give them a similar look and feel. Patterns can help guide how the whole application is assembled.

In this chapter, we're going to look at a family of patterns, which I've taken to calling the MV* family. This family includes MVC, MVVM, MVP, and even PAC. Just like their names, the patterns themselves are pretty similar. The chapter will cover each of these patterns and show how, or if, we can apply them to JavaScript. We'll also pay special attention to how the patterns differ one from the other. By the end of the chapter, you should be able to thrill guests at a cocktail party with an explanation of the nuances of MVP versus MVC.

The following topics will be covered:

- History of model view patterns
- Model View Controller
- Model View Presenter
- Model View ViewModel

First, some history

Separating concerns inside an application is a very important idea. We live in a complex and ever changing world. This means that not only is it nearly impossible to formulate a computer program which works in exactly the way users want, but that what users want is an ever shifting maze. Couple this with the fact that an ideal program for user A is totally different from an ideal program for user B, and we're guaranteed to end up in a mess. Our applications need to change as frequently as we change our socks: at least once a year.

Layering an application and maintaining modularity reduces the impact of a change. The less each layer knows about the other layers the better. Maintaining simple interfaces between the layers reduces the chances that a change to one layer will percolate to another layer.

If you've ever taken a close look at a high quality piece of nylon (from a hot air balloon, parachute, or expensive jacket), you may have noticed that the fabric seems to form tiny squares. This is because every few millimeters a thick reinforcing thread is added to the weave to form a crosshatch pattern. If the fabric is ripped, then the rip will be stopped or at least slowed by the reinforcement. This limits the damage to a small area and prevents it from spreading.

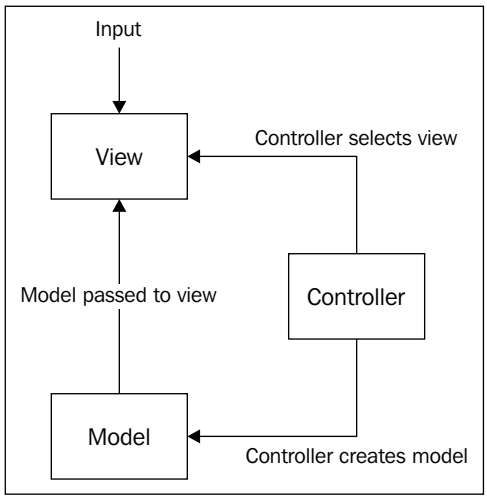
Layers and modules in an application are exactly the same: they limit the spread of damage from a change.

In the early chapters of this book, we talked a bit about the seminal language Smalltalk. It was the language that made classes famous. Like many of these patterns, the original MV* pattern, **Model View Controller** (**MVC**), was used long before it was ever identified. Although difficult to prove, it seems that Model View Controller was originally suggested in the late 1970s by Trygve Reenskaug, a Norwegian computer scientist, during a visit to Xerox PARC. Through the 1980s, the pattern became heavily used in the Smalltalk application. However, it was not until 1988 that the pattern was more formally documented in an article entitled *A cookbook for using the model-view-controller user interface paradigm* by Krasner and Pope.

Model View Controller

MVC is a pattern that is used to create rich, interactive user interfaces: just the sort of interfaces that are becoming more and more popular on the Web.

The astute amongst you will have already figured out that the pattern is made up of three major components: model, view, and controller:



The preceding diagram shows the relationship between the three components in MVC.

The model contains the state of the program. In many applications, this model is contained in some form in a database. The model may be rehydrated from a persistent store (such as the database) or it can be transient. Ideally, the model is the only mutable part of the pattern. Neither the view nor the controller have any state associated with them.

For a simple login screen, the model might look like the following code:

```
class LoginModel{
  UserName: string;
  Password: string;
  RememberMe: bool;

  LoginSuccessful: bool;
  LoginErrorMessage: string;
}
```

You'll notice that not only do we have fields for the inputs shown to the user, but also for the state of the login. This would not be apparent to the user but it is still part of the application state.

The model is usually modeled as a simple container for information. Typically, there are no real functions in the model. It simply contains data fields and may also contain validation. In some implementations of the MVC pattern, the model also contains metadata about the fields, such as validation rules.

The Naked Object pattern is a deviation from the typical MVC pattern. It augments the model with extensive business information as well as hits about the display and editing of data. It even contains methods to persist the model to storage.

The views in the Naked Object pattern are automatically generated from these models. The controller is also automatically generated by examining the model. This centralizes the logic to display and manipulate application state and saves the developer from having to write their own views and controllers. So while the view and the controller still exist, they are not actual objects but are dynamically created from the model.

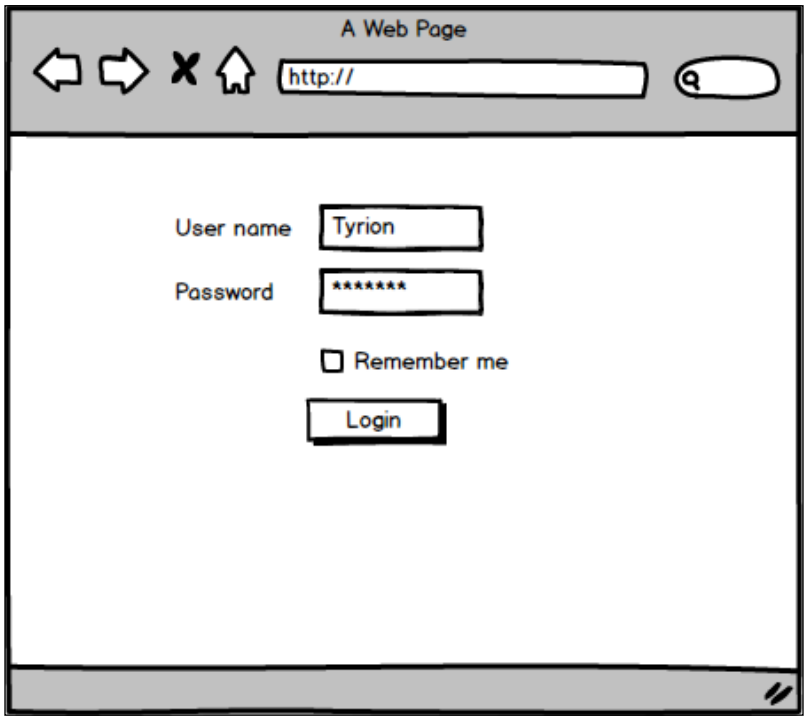
Several systems have been successfully deployed using this pattern. Some criticisms have emerged around the ability to generate an attractive user interface from just the models, as well as how to properly coordinate multiple views.

In a foreword to the PhD thesis presenting Naked Object, Reenskaug suggests that the Naked Object pattern is actually closer to his original vision for MVC than most of the derivations of MVC in the wild.

Updates to the model are communicated to the view whenever the state has changed. This is typically done through the use of an observer pattern. The model does not typically know about either the controller or the view. The first is an anonymous source of change and the second is only updated through the observer pattern, so the model doesn't have direct knowledge of it.

The view does pretty much what you would expect: communicate the model state to a target. I hesitate to suggest that the view must be a visual or graphical representation of the model, as the view is being frequently communicated to another computer and may be in the form of XML, JSON, or some other data format. In most cases, especially those related to JavaScript, the view will be a graphical object. In a web context, this will typically be HTML which is rendered by the browser. JavaScript is also gaining popularity on phones and on the desktop, so the view could also be a screen on a phone or on the desktop.

The view for the model presented earlier might look like the following diagram:



In cases where the observer pattern is not used, the view may poll the model at some interval looking for changes. In this case, the view may have to keep a representation of the state itself or at least a version number. It is important that the view not unilaterally update this state without passing the updates to the controller, otherwise the model and the copy in the view will get out of sync.

Finally, the state of the model is updated by the controller. The controller usually contains all the logic and business rules to update fields on the model. A simple controller for our login page might look like the following code:

```
var LoginController = (function () {
  function LoginController(model) {
    this.model = model;
  }
  LoginController.prototype.Login = function (userName, password,
  rememberMe) {
    this.model.UserName = userName;
    this.model.Password = password;
```

```
        this.model.RememberMe = rememberMe;
        if (this.checkPassword(userName, password))
          this.model.LoginSuccessful;
        else {
          this.model.LoginSuccessful = false;
          this.model.LoginErrorMessage = "Incorrect username or
          password";
        }
      };
      return LoginController;
    })();
```

The controller knows about the existence of the model and is typically aware of the view's existence as well. It coordinates the two of them. A controller may be responsible to initialize more than one view. For instance, a single controller may provide a list view of all the instances of a model as well as a view that simply provides details. In many systems, a controller will have **create, read, update, and delete** (**CRUD**) operations on it that work over a model. The controller is responsible for choosing the correct view and to wire up the communication between the model and the view.

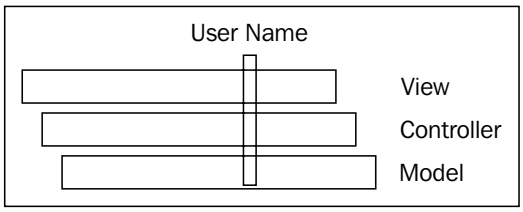
When there is a need for a change to the application, then the location of the code should be immediately apparent. Refer to the following table for examples:

Change	Location
Elements don't appear well spaced on the screen, change spacing	View
No users are able to log in due to a logical error in password validation	Controller
New field to be added	All layers

Presentation-Abstraction-Control (**PAC**) is another pattern that makes use of a triad of components. In this case, its goal is to describe a hierarchy of encapsulated triples that more closely match how we think of the world. The control, similar to an MVC controller, passes interactions up in the hierarchy of encapsulated components allowing for information to flow between them. The abstraction is similar to a model but may represent only a few fields that are important for that specific PAC, instead of the entire model. Finally, the presentation is effectively the same as a view.

The hierarchical nature of PAC allows for parallel processing of the components, meaning that it can be a powerful tool in today's multiprocessor systems.

You might notice that the last item in the preceding table requires a change in all the layers of the application. The multiple locations for responsibility are something that the Naked Object pattern attempts to address by dynamically creating views and controllers. The MVC pattern splits code into locations by dividing the code by its role in user interaction. This means that a single data field lives in all the layers, as shown in the following diagram:



Some might call this a cross-cutting concern but really it doesn't span a sufficient amount of the application to be called so. Data access and logging are cross-cutting concerns as they are pervasive and difficult to centralize. This pervasion of a field through the different layers is really not a major problem. However, if it is bugging you, then you might be an ideal candidate to use the Naked Object pattern.

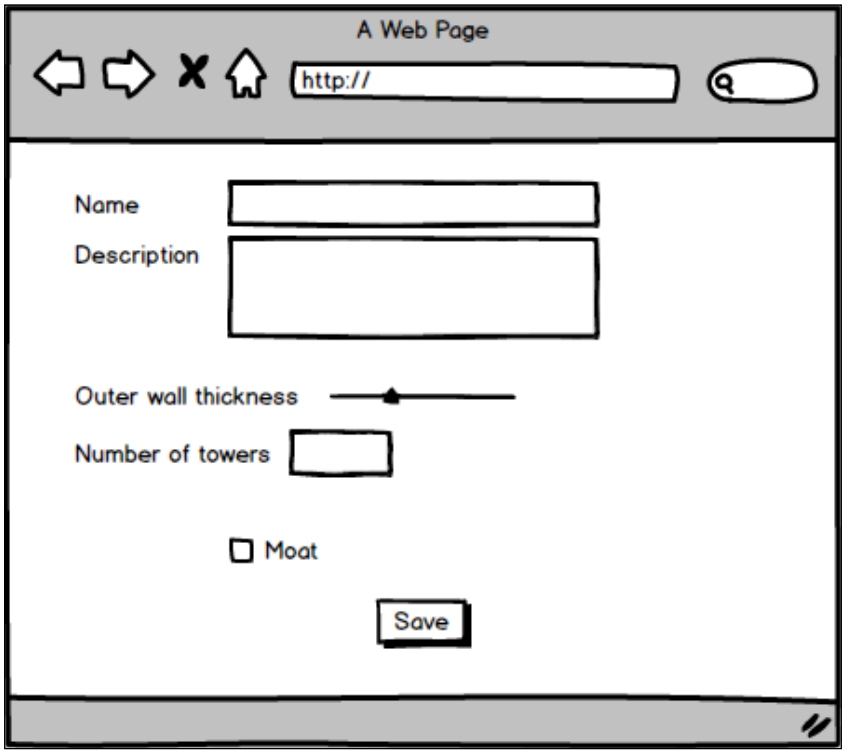
Let's step into building some code to represent an MVC pattern in JavaScript.

The MVC code

Let's start with a simple scenario where we can apply MVC. Unfortunately, Westeros has very few computers, probably due to the lack of electricity. Thus, applying application structuring patterns using Westeros as an example is difficult. Sadly, we'll have to take a step back and talk about an application which controls Westeros. Let's assume it to be a web application and implement the entirety of MVC on the client side.

It is possible to implement MVC by splitting the model, view, and controller between client and server. Typically, the controller would sit on the server and provide an API which is known by the view. The model serves as a communication method both to the view which resides on the web browser and to the data store, probably a database of some form. It is also common that the controller be split between the server and the client in cases where some additional control is required on the client.

In our example, we would like to create a screen that controls the properties of a castle. Fortunately, you're lucky that this is not a book on designing user interfaces with HTML, as I would certainly fail. We'll stick to a diagram in place of the HTML:



For the most part, the view simply provides a set of controls and data for the end user. In this example, the view will need to know how to call the save function on the controller. Let's set that up:

```
var CreateCastleView = (function () {
  function CreateCastleView(document, controller, model,
  validationResult) {
    this.document = document;
    this.controller = controller;
    this.model = model;
    this.validationResult = validationResult;
    var _this = this;
    this.document.getElementById("saveButton").addEventListener
    ("click", function(){return _this.saveCastle();});
    this.document.getElementById("castleName").value = model.name;
```

```
      this.document.getElementById("description").value =
      model.description;
      this.document.getElementById("outerWallThickness").value =
      model.outerWallThickness;
      this.document.getElementById("numberOfTowers").value =
      model.numberOfTowers;
      this.document.getElementById("moat").value = model.moat;
    }
    CreateCastleView.prototype.saveCastle = function () {
      var data = {
        name: this.document.getElementById("castleName").value,
        description:
        this.document.getElementById("description").value,
        outerWallThickness:
        this.document.getElementById("outerWallThickness").value,
        numberOfTowers:
        this.document.getElementById("numberOfTowers").value,
        moat: this.document.getElementById("moat").value
      };
      this.controller.saveCastle(data);
    };
    return CreateCastleView;
  })();
```

You'll notice that the constructor for this view contains both a reference to the document and to the controller. The document contains both HTML and styling, provided by CSS. We can get away with not passing in a reference to the document, but injecting the document in this fashion allows for easier testability. We'll look at testability in detail in a later chapter. It also permits reusing the view multiple times on a single page, without worrying about conflicts between the two instances.

The constructor also contains a reference to the model that is used to add data to fields on the page as needed. Finally, the constructor also references a collection of errors. This allows for validation errors from the controller to be passed back to the view that needs to be handled. We have set the validation result to be a wrapped collection that looks something like this:

```
class ValidationResult{
  public IsValid: boolean;
  public Errors: Array<String>;
  public constructor(){
    this.Errors = new Array<String>();
  }
}
```

The only functionality here is that the button's `onclick` method is bound to calling save on the controller. Instead of passing in a large number of parameters to the `saveCastle` function on the controller, we build a lightweight object and pass that in. This makes the code more readable, especially in cases where some of the parameters are optional. No real work is done in the view and all the input goes directly to the controller.

The controller contains the real functionality of the application:

```
var Controller = (function () {
  function Controller(document) {
    this.document = document;
  }
  Controller.prototype.createCastle = function () {
    this.setView(new CreateCastleView(this.document, this));
  };

  Controller.prototype.saveCastle = function (data) {
    var validationResult = this.validate(data);
    if (validationResult.IsValid) {
      //save castle to storage
    this.saveCastleSuccess(data);
    } else {
      this.setView(new CreateCastleView(this.document, this, data,
      validationResult));
    }
  };

  Controller.prototype.saveCastleSuccess = function (data) {
    this.setView(new CreateCastleSuccess(this.document, this,
    data));
  };

  Controller.prototype.setView = function (view) {
    //send the view to the browser
  };

  Controller.prototype.validate = function (model) {
    var validationResult = new validationResult();
    if (!model.name || model.name === "") {
      validationResult.IsValid = false;
      validationResult.Errors.push("Name is required");
    }
    return;
  };
  return Controller;
})();
```

The controller here does a number of things. The first thing is that it has a `setView` function which instructs the browser to set the given view as the current one. This is probably done through the use of the template. The mechanics of how that works are not important to the pattern, so I'll leave that up to your imagination.

Next, the controller implements a `validate` method. This method checks to make sure that the model is valid. Some validation may be performed on the client, such as testing the format of a postal code, but other validation requires a server trip. If a username must be unique, then there is no reasonable way to test that on the client without communicating with the server. In some cases, the validation functionality may exist on the model rather than in the controller.

Methods to set up various different views are also found in the controller. In this case, we have a bit of a workflow with a view to create a castle, then views for both success and failure. The failure case just returns the same view with a collection of validation errors attached to it. The success case returns a whole new view.

The logic to save the model to some sort of persistent storage is also located in the controller. Again, the implementation of this is less important than to see that the logic to communicate with the storage system is located in the controller.

The final letter in MVC is the model, in this case, it is a very lightweight one:

```
var Model = (function () {
  function Model(name, description, outerWallThickness,
  numberOfTowers, moat) {
    this.name = name;
    this.description = description;
    this.outerWallThickness = outerWallThickness;
    this.numberOfTowers = numberOfTowers;
    this.moat = moat;
  }
  return Model;
})();
```

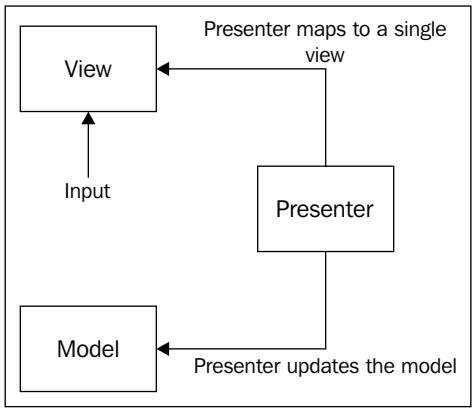
As you can see, all it does is keep track of the variables that make up the state of the application.

Concerns are well separated in this pattern allowing for changes to be made with relative ease.

Model View Presenter

The **Model View Presenter** (**MVP**) pattern is very similar to MVC. It is a fairly well known pattern in the Microsoft world and is generally used to structure WPF and Silverlight applications. It can be used in pure JavaScript as well. The key difference comes down to how the different parts of the system interact and where their responsibility ends.

The first difference is that with the presenter there is a one-to-one mapping between the presenter and the view. This means that the logic that existed in the controller in the MVC pattern that selected the correct view to render, doesn't exist. Or rather, it exists at a higher level outside the concern of the pattern. The selection of the correct presenter may be handled by a routing tool. Such a router will examine the parameters and provide the best choice for the presenter. This is explained in the following diagram:



The presenter is aware of both the view and the model, but the view is unaware of the model and the model is unaware of the view. All communication is passed through the presenter.

The presenter pattern is often characterized by a great deal of two-way dispatch. A click will fire in the presenter then the presenter will update the model with the change and then update the view. The preceding diagram suggests that the input first passes through the view. In a passive version of the MVP pattern, the view has little to no interaction with the messages as they are passed onto the presenter. However, there is also a variation called active MVP that allows the view to contain some additional logic.

This active version of MVP can be more useful for use in web applications. It permits adding validation and other simple logic to the view. This reduces the number of requests that need to pass from the client back to the web server.

Let's update our existing code sample to use MVP instead of MVC.

The MVP code

Let's start again with the view having the following code:

```
var CreateCastleView = (function () {
  function CreateCastleView(document, presenter) {
    this.document = document;
    this.presenter = presenter;
    this.document.getElementById("saveButton").
    addEventListener("click", this.saveCastle);
  }
  CreateCastleView.prototype.setCastleName = function (name) {
    this.document.getElementById("castleName").value = name;
  };

  CreateCastleView.prototype.getCastleName = function () {
    return this.document.getElementById("castleName").value;
  };

  CreateCastleView.prototype.setDescription = function
  (description) {
    this.document.getElementById("description").value =
    description;
  };

  CreateCastleView.prototype.getDescription = function () {
    return this.document.getElementById("description").value;
  };

  CreateCastleView.prototype.setOuterWallThickness = function
  (outerWallThickness) {
    this.document.getElementById("outerWallThickness").value =
    outerWallThickness;
  };
```

```
    CreateCastleView.prototype.getOuterWallThickness = function () {
      return this.document.getElementById
      ("outerWallThickness").value;
    };

    CreateCastleView.prototype.setNumberOfTowers = function
    (numberOfTowers) {
      this.document.getElementById("numberOfTowers").value =
      numberOfTowers;
    };

    CreateCastleView.prototype.getNumberOfTowers = function () {
      return parseInt(this.document.getElementById
      ("numberOfTowers").value);
    };

    CreateCastleView.prototype.setMoat = function (moat) {
      this.document.getElementById("moat").value = moat;
    };

    CreateCastleView.prototype.getMoat = function () {
      return this.document.getElementById("moat").value;
    };

    CreateCastleView.prototype.setValid = function
    (validationResult) {
    };

    CreateCastleView.prototype.saveCastle = function () {
      this.presenter.saveCastle();
    };
    return CreateCastleView;
  })();
  CastleDesign.CreateCastleView = CreateCastleView;
```

As you can see, the constructor for the view no longer takes a reference to the model. This is because the view in MVP doesn't have any idea about what model is being used. That information is abstracted away by the presenter. The reference to the presenter remains in the constructor to allow sending messages back to the presenter.

Without the model, there is an increase in the number of public setter and getter methods. These setters allow the presenter to make updates to the state of the view. The getters provide an abstraction over how the view stores the state and gives the presenter a way to get the information. The `saveCastle` function no longer passes any values to the presenter.

The presenter's code looks like this:

```
var CreateCastlePresenter = (function () {
  function CreateCastlePresenter(document) {
    this.document = document;
    this.model = new CreateCastleModel();
    this.view = new CreateCastleView(document, this);
  }
  CreateCastlePresenter.prototype.saveCastle = function () {
    var data = {
      name: this.view.getCastleName(),
      description: this.view.getDescription(),
      outerWallThickness: this.view.getOuterWallThickness(),
      numberOfTowers: this.view.getNumberOfTowers(),
      moat: this.view.getMoat()
    };

    var validationResult = this.validate(data);
    if (validationResult.IsValid) {
      //write to the model
      this.saveCastleSuccess(data);
    } else {
      this.view.setValid(validationResult);
    }
  };

  CreateCastlePresenter.prototype.saveCastleSuccess = function
  (data) {
    //redirect to different presenter
  };

  CreateCastlePresenter.prototype.validate = function (model) {
    var validationResult = new validationResult();
    if (!model.name || model.name === "") {
      validationResult.IsValid = false;
      validationResult.Errors.push("Name is required");
    }
    return;
  };
  return CreateCastlePresenter;
})();
CastleDesign.CreateCastlePresenter = CreateCastlePresenter;
```

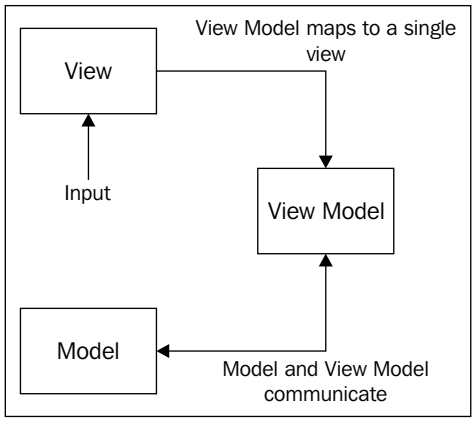
You can see that the view is now referenced in a persistent fashion in the presenter. The `saveCastle` method calls into the view to get its values. However, the presenter does make sure to use the public methods of the view instead of referencing the document directly. The `saveCastle` method updates the model. If there are validation errors, then it will call back into the view to update the `IsValid` flag. This is an example of the double dispatch I mentioned earlier.

Finally, the model remains unchanged from before. We've kept the validation logic in the presenter. At which level the validation is done, model or presenter, matters less than being consistent in where the validation is done through your application.

The MVP pattern is again a fairly useful pattern for building user interfaces. The larger separation between the view and the model creates a stricter API, allowing for better adaptation to change. However, this comes at the expense of more code. With more code comes more opportunity for bugs.

Model View ViewModel

The final pattern we'll look at in this chapter is the **Model View ViewModel** pattern, more commonly known as **MVVM**. By now, this sort of pattern should be getting quite familiar. The following diagram explains the MVVM pattern:



You can see here that many of the same constructs have returned, but that the communication between them is somewhat different.

In this variation, what has previously been the controller and presenter is now the view model. Just like with MVC and MVP, the majority of the logic is held within the central component, in this case, the view model. The model itself is actually very simple in MVVM. Typically, it acts as an envelope that just holds data. Validation is done within the view model.

Just like with MVP, the view is totally unaware of the existence of the model. The difference is that, with MVP, the view was aware that it was talking to some intermediate class. It called methods rather than simply setting values. In MVVM, the view believes that the view model is its view. Instead of calling operations such as `saveCastle` and passing in data or waiting for data to be requested, the view updates fields on the view model as they change. In effect, the fields on the view are bound to the view model. The view model may proxy these values through to the model or wait until a commit-like operation such as save is called to pass the data along.

Equally, changes to the view model should be reflected at once in the view. A single view may have a number of view models. Each of these view models may push updates to the view or have changes pushed to it via the view.

Let's take a look at a really rudimentary implementation of this and then we'll discuss how to make it better.

The MVVM code

The naive view implementation is, frankly, a huge mess:

```
var CreateCastleView = (function () {
  function CreateCastleView(document, viewModel) {
    this.document = document;
    this.viewModel = viewModel;
    var _this = this;
    this.document.getElementById("saveButton").
    addEventListener("click", function () {
      return _this.saveCastle();
    });
    this.document.getElementById("name").
    addEventListener("change", this.nameChangedInView);
    this.document.getElementById("description").
    addEventListener("change", this.descriptionChangedInView);
    this.document.getElementById("outerWallThickness").
    addEventListener("change",
    this.outerWallThicknessChangedInView);
    this.document.getElementById("numberOfTowers").
    addEventListener("change", this.numberOfTowersChangedInView);
    this.document.getElementById("moat").
    addEventListener("change", this.moatChangedInView);
  }
  CreateCastleView.prototype.nameChangedInView = function (name) {
    this.viewModel.nameChangedInView(name);
```

```
    };

    CreateCastleView.prototype.nameChangedInViewModel = function
    (name) {
      this.document.getElementById("name").value = name;
    };
    //snipped more of the same
    CreateCastleView.prototype.isValidChangedInViewModel = function
    (validationResult) {
      this.document.getElementById("validationWarning").innerHtml =
      validationResult.Errors;
      this.document.getElementById("validationWarning").className =
      "visible";
    };
    CreateCastleView.prototype.saveCastle = function () {
      this.viewModel.saveCastle();
    };
    return CreateCastleView;
  })();
  CastleDesign.CreateCastleView = CreateCastleView;
```

It is highly repetitive and each property must be proxied back to the ViewModel. I've truncated most of this code but it adds up to a good 70 lines. The following code inside the view model is equally terrible:

```
  var CreateCastleViewModel = (function () {
    function CreateCastleViewModel(document) {
      this.document = document;
      this.model = new CreateCastleModel();
      this.view = new CreateCastleView(document, this);
    }
    CreateCastleViewModel.prototype.nameChangedInView = function
    (name) {
      this.name = name;
    };

    CreateCastleViewModel.prototype.nameChangedInViewModel =
    function (name) {
      this.view.nameChangedInViewModel(name);
    };

    //snip
```

```
  CreateCastleViewModel.prototype.saveCastle = function () {
    var validationResult = this.validate();
    if (validationResult.IsValid) {
      //write to the model
      this.saveCastleSuccess();
    } else {
      this.view.isValidChangedInViewModel(validationResult);
    }
  };

  CreateCastleViewModel.prototype.saveCastleSuccess = function ()
  {
    //do whatever is needed when save is successful.
    //Possibly update the view model
  };

  CreateCastleViewModel.prototype.validate = function () {
    var validationResult = new validationResult();
    if (!this.name || this.name === "") {
      validationResult.IsValid = false;
      validationResult.Errors.push("Name is required");
    }
    return;
  };
  return CreateCastleViewModel;
})();
```

One look at this code should send you running for the hills. It is set up in a way that will encourage copy and paste programming: a fantastic way to introduce errors into a code base. I sure hope there is a better way to transfer changes between the model and the view.

A better way to transfer changes between the model and the view

It may not have escaped your notice that there are a number of MVVM style frameworks for JavaScript in the wild. Obviously, they would not have been readily adopted if they followed the previous approach. Instead they follow one of the two different approaches.

The first approach is known as **dirty checking**. In this approach, after every interaction with the view model, we loop over all of its properties looking for changes. When changes are found, the related value in the view is updated with the new value. For changes to values in the view, change actions are attached to all the controls. These then trigger updates to the view model.

This approach can be slow for large models as it is expensive to iterate over all the properties of a large model. The number of things that can cause a model to change is high, and there is no real way to tell if a distant field in a model has been changed by changing another without validating it and validating it. On the up side, dirty checking allows you to use plain old JavaScript objects. There is no need to write your code any differently than before. The same is not true of the other approach: container objects.

With a container object, a special interface is provided to wrap existing objects so that changes to the object may be directly observed. Basically, this is an application of the observer pattern, but applied dynamically so the underlying object has no idea it is being observed. The spy pattern, perhaps?

An example might be helpful here. Let's say that we have the model object we've been using up until now:

```
var CreateCastleModel = (function () {
  function CreateCastleModel(name, description,
  outerWallThickness, numberOfTowers, moat) {
    this.name = name;
    this.description = description;
    this.outerWallThickness = outerWallThickness;
    this.numberOfTowers = numberOfTowers;
    this.moat = moat;
  }
  return CreateCastleModel;
})();
```

Then instead of `model.name` being a simple string, we would wrap some function around it. In the case of the Knockout library, this would look like the following code:

```
var CreateCastleModel = (function () {
  function CreateCastleModel(name, description,
  outerWallThickness, numberOfTowers, moat) {
    this.name = ko.observable(name);
    this.description = ko.observable(description);
    this.outerWallThickness = ko.observable(outerWallThickness);
    this.numberOfTowers = ko.observable(numberOfTowers);
```

```
    this.moat = ko.observable(moat);
  }
  return CreateCastleModel;
})();
```

In the highlighted code, the various properties of the model are being wrapped with an observable. This means that they must now be accessed differently:

```
var model = new CreateCastleModel();
model.name("Winterfell"); //set
model.name(); //get
```

This approach obviously adds some friction to your code and makes changing frameworks quite involved.

Current MVVM frameworks are split on their approach to container objects versus dirty checking. AngularJS uses dirty checking while Backbone, Ember, and Knockout all make use of container objects. There is currently no clear winner.

Observing view changes

Fortunately, the pervasiveness of MV* patterns on the Web and the difficulties with observing model changes has not gone unnoticed. You might be expecting me to say that this will be solved in ECMAScript 6, as is my normal approach. Weirdly, the solution to all of this, `Object.observe`, is a feature under discussion for ECMAScript 7. However, at the time of this writing, at least one major browser already supports it.

The `Object.observe` method can be used as shown in the following code:

```
var model = { };
Object.observe(model, function(changes){
  changes.forEach(function(change) {
    console.log("A " + change.type + " occured on " +
    change.name + ".");
    if(change.type=="update")
      console.log("\tOld value was " + change.oldValue );
  });
});
model.item = 7;
model.item = 8;
delete model.item;
```

Having this simple interface to monitor changes to objects removes much of the logic provided by large MV* frameworks. It will be easier to roll your own functionality for MV* and there may, in fact, be no need to use external frameworks.

Hints and tips

The different layers of the various MV* patterns need not all be on the browser nor do they all need to be written in JavaScript. Many popular frameworks allow the maintenance of a model on the server and communicating with it, using JSON.

The `Object.observe` method may not be available on all browsers yet, but there are polyfills that can be used to create a similar interface. The performance is not as good as the native implementation, but it is still usable.

Summary

Separating concerns to a number of layers ensures that changes to the application are isolated like a ripstop. The various MV* patterns allow separation of the concerns in a graphical application. The differences between the various patterns come down to how the responsibilities are separated and how information is communicated.

In the next chapter, we'll look at a number of patterns and techniques to improve the experience of developing and deploying JavaScript to the Web.

8
Web Patterns

The rise of Node.js has proven that JavaScript has a place on web servers, even on very high throughput servers. There is no denying that JavaScript's pedigree remains in the browser for client-side programming.

In this chapter, we're going to look at a number of patterns to improve the performance and usefulness of JavaScript on the client. I'm not sure that all of these can be thought of as patterns in the strictest sense. They are, however, important and worth mentioning.

The concepts we'll examine in this chapter are:

- Sending JavaScript
- Plugins
- Multithreading
- Circuit breaker pattern
- Back-off
- Promises

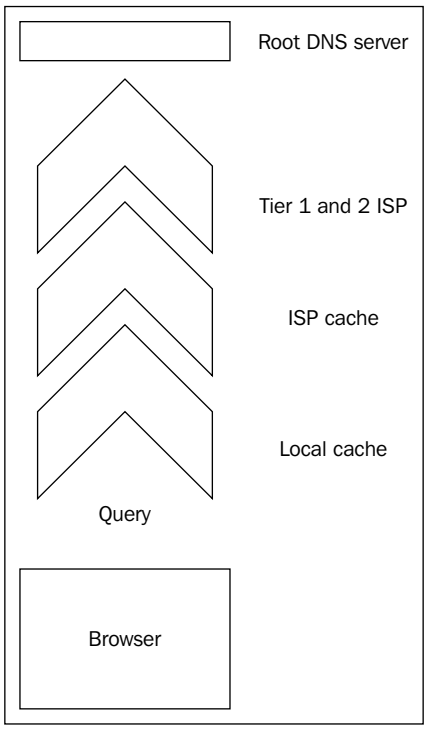
Sending JavaScript

Communicating JavaScript to the client seems to be a simple proposition: so long as you can get the code to the client, it doesn't matter how that happens, right? Well, not exactly. There are actually a number of things that need to be considered when sending JavaScript to the browser.

Combining files

Way back in *Chapter 2*, *Organizing Code*, we looked at how to build objects using JavaScript. Although opinions on this vary, I consider it to be good form to have a one class to one file organization of my JavaScript or really any of my object-oriented code. Doing this makes finding code easy. Nobody needs to hunt through a 9,000-line-long JavaScript file to locate one method. It also allows for a hierarchy to be established, again allowing for good code organization. However, good organization for a developer is not necessarily good organization for a computer. In our case, having a lot of small files is actually highly detrimental. To understand why, you need to know a little bit about how browsers ask for and receive content.

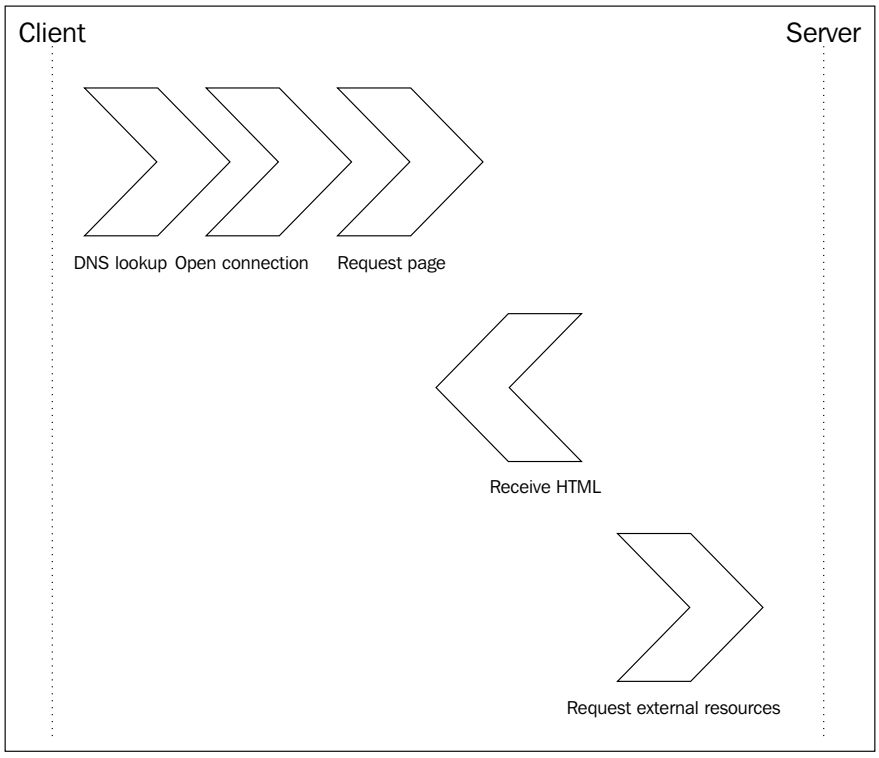
When you type a URL into the address bar of a browser and hit *Enter*, a cascading series of events happens. The first thing is that the browser will ask the operating system to resolve the website name to an IP address. In both Windows and Linux (and Mac OS X), the standard C library function, `gethostbyname`, is used. This function will check the local DNS cache to see if the mapping from name to address is already known. If it is, then that information is returned. If not, then the computer makes a request to the DNS server one step up from it. Typically, this is the DNS server provided by the ISP but on a larger network; it could also be a local DNS server. A typical DNS query is shown here:



If a record doesn't exist on that server, then the request is propagated up a chain of DNS servers in an attempt to find one that knows about the domain. Eventually, the propagation stops at the root servers. These root servers are the stopping point for queries—if they don't know who is responsible for DNS information for a domain, then the lookup is deemed to have failed.

Once the browser has an address for the site, it opens up a connection and sends a request for the document. If no document is provided, then a / character is sent. Should the connection be a secure one, then negotiation of SSL/TSL is performed at this time. There is some computational expense to setting up an encrypted connection but this is slowly being fixed.

The server will respond with a blob of HTML. As the browser receives this HTML, it starts to process it; the browser does not wait for the entire HTML document to be downloaded before it goes to work. If the browser encounters a resource that is external to HTML, it will kick off a new request to open another connection to the web server and download that resource. The maximum number of connections to a single domain is limited so that the web server isn't flooded. It should also be mentioned that setting up a new connection to the web server carries overhead. The following diagram explains this process:



Connections to the web server should be limited to avoid paying the connection setup costs repeatedly. This brings us to our first concept: combining files.

If you've followed the advice to leverage namespaces and classes in your JavaScript, then putting all of your JavaScript together in a single file is a trivial step. One need only concatenate the files together and everything should continue to work as normal. Some minor care and attention may need to be paid to the order of inclusion but not typically.

The previous code we've written has been pretty much one file per pattern. If there is a need for multiple patterns to be used, then we could simply concatenate the files together. For instance, the combined builder and factory method patterns might look like this:

```
var Westeros;
  (function (Westeros) {
    (function (Religion) {
      …
    })(Westeros.Religion || (Westeros.Religion = {}));
    var Religion = Westeros.Religion;
  })(Westeros || (Westeros = {}));
  (function (Westeros) {
    var Tournament = (function () {
      function Tournament() {
      }
    return Tournament;
  })();
  Westeros.Tournament = Tournament;
  …
  })();
  Westeros.Attendee = Attendee;
})(Westeros || (Westeros = {}));
```

The question may arise as to how much of your JavaScript should be combined and loaded at once. It is a surprisingly difficult question to answer. On one hand, it is desirable to front load all the JavaScript for the entire site when users first arrive at the site. This means that users will pay a price initially but will not have to download any additional JavaScript as they travel about the site. This is because the browser will cache the script and reuse it instead of downloading it from the server again. However, if users only visit a small subset of the pages on the site, then they would have loaded a great deal of JavaScript that was not needed.

On the other hand, splitting up the JavaScript means that additional page visits incur a penalty to retrieve additional JavaScript files. There is a sweet spot somewhere in the middle of these two approaches. Script can be organized into blocks that map to different sections of the website. This can be a place where using proper namespacing will come in handy once again. Each namespace can be combined into a single file and then loaded as users visit that part of the site.

In the end, the only approach that makes sense is to maintain statistics about how users move about the site. Based on this information, an optimal strategy to find the sweet spot can be established.

Minification

Combining JavaScript into a single file solves the problem of limiting the number of requests. However, each request may still be large. Again, we come to a schism between what makes code fast and readable by humans and what makes it fast and readable by computers.

We humans like descriptive variable names, bountiful whitespace, and proper indentation. Computers don't care about descriptive names, whitespace, or proper indentation. In fact, these things increase the size of the file and thus decrease the speed at which the code can be read.

Minification is a compile step that transforms the human readable code into smaller but equivalent code. External variables remain named the same as the minifier has no way to know what other code may be relying on the variable names remaining unchanged.

As an example, if we start with the composite code from *Chapter 4*, *Structural Patterns*, the minified code looks like this:

```
var Westros;(function(Westros){(function(Food){var
SimpleIngredient=(function(){function SimpleIngredient(name
,calories,ironContent,vitaminCContent){this.name=name;this.
calories=calories;this.ironContent=ironContent;this.
vitaminCContent=vitaminCContent}SimpleIngredient.prototype.
GetName=function(){return this.name};SimpleIngredient.prototype.
GetCalories=function(){return this.calories};SimpleIngredient.
prototype.GetIronContent=function(){return this.
ironContent};SimpleIngredient.prototype.GetVitaminCContent=function()
{return this.vitaminCContent};return SimpleIngredient})();Food.Sim
pleIngredient=SimpleIngredient;var CompoundIngredient=(function()
{function CompoundIngredient(name){this.name=name;this.ingredients=new
Array()}CompoundIngredient.prototype.AddIngredient=function(ing
redient){this.ingredients.push(ingredient)};CompoundIngredient.
prototype.GetName=function(){return this.name};CompoundIngredient.
```

```
prototype.GetCalories=function(){var total=0;for(var i=0;i<this.
ingredients.length;i++){total+=this.ingredients[i].GetCalories()}
return total};CompoundIngredient.prototype.GetIronContent=function()
{var total=0;for(var i=0;i<this.ingredients.length;i++){total+=this.
ingredients[i].GetIronContent()}return total};CompoundIngredient.
prototype.GetVitaminCContent=function(){var total=0;for(var
i=0;i<this.ingredients.length;i++){total+=this.ingredients[i].
GetVitaminCContent()}return total};return CompoundIngredient})();Food.
CompoundIngredient=CompoundIngredient})(Westros.Food||(Westros.
Food={}));var Food=Westros.Food})(Westros||(Westros={}));
```

You'll notice that all the spacing has been removed and that all the internal variables have been replaced with smaller versions. At the same time, you can spot that some well-known variable names have remained unchanged.

Minification shortened this particular piece of code by 40 percent. Compressing the content stream from the server using **gzip**, a popular approach, is lossless compression. That means that there is a perfect bijection between compressed and uncompressed. Minification, on the other hand, is a lossy compression. There is no way to get back to the unminified code from just the minified code once it has been minified.

You can read more about gzip compression at `http://betterexplained.com/articles/how-to-optimize-your-site-with-gzip-compression/`.

If there is need to return to the original code, then source maps can be used. A source map is a file that provides a translation from one format of code to another. It can be loaded by the debugging tools in modern browsers to allow you to debug the original code instead of unintelligible minified code. Multiple source maps can be combined to allow for translation from, say, minified code to unminified JavaScript to TypeScript.

Content delivery networks

The final delivery trick is to make use of **content delivery networks** (**CDNs**). CDNs are distributed networks of hosts whose only purpose is to serve out static content. In much the same way that the browser will cache JavaScript between pages on the site, it will also cache JavaScript that is shared between multiple web servers. Thus, if your site makes use of jQuery, then pulling jQuery from a well-known CDN such as `https://code.jquery.com` or Microsoft's ASP.NET CDN may be faster as it is already cached. Pulling from a CDN also means that the content is coming from a different domain and doesn't count against the limited connections to your server. Referencing a CDN is as simple as setting the source of the script tag to point at the CDN.

Once again, some metrics will need to be gathered to see whether it is better to use a CDN or simply roll libraries into the JavaScript bundle. Examples of such metrics may include the added time to perform additional DNS lookup and the difference in the download sizes. The best approach is to use the timing APIs in the browser.

The long and short of distributing JavaScript to the browser is that experimentation is required. Testing a number of approaches and measuring the results will give the best result for end users.

Plugins

There are a great number of really impressively good JavaScript libraries in the wild. For me, the library that changed how I look at JavaScript was jQuery. For others, it may have been one of the other popular libraries such as MooTools, Dojo, Prototype, or YUI. However, jQuery has exploded in popularity and has, at the time of writing, won the JavaScript library wars. 78.5 percent of the top 10,000 websites by traffic on the Internet make use of some version of jQuery. None of the rest of the libraries even break 1 percent.

Many developers have seen fit to implement their own libraries on top of these foundational libraries in the form of plugins. A plugin typically modifies the prototype exposed by the library and adds additional functionality. The syntax is such that, to the end developer, it appears to be part of the core library.

How plugins are built vary depending on the library you're trying to extend. Nonetheless, let's take a look at how we can build a plugin for jQuery and then for one of my favorite libraries, d3. We'll see if we can extract some commonalities.

jQuery

At jQuery's core is the CSS selector library called Sizzle.js. It is Sizzle.js that is responsible for all the really nifty ways jQuery can select items on a page using CSS3 selectors. jQuery can be used to select elements on a page like this:

```
$(":input").css("background-color", "blue");
```

What is returned is a jQuery object. The jQuery object acts a lot like, although not completely like, an array. This is achieved by creating a series of keys on the jQuery object numbered 0 to *n* - 1, where *n* is the number of elements matched by the selector. This is actually pretty smart, as it enables array like accessors, while also providing a bunch of additional functions:

```
$($(":input")[2]).css("background-color", "blue");
```

The items at the indices are plain HTML elements and not wrapped with jQuery, hence the use of the second `$()` parameter.

For jQuery plugins, we typically want to make our plugins extend this jQuery object. Because it is dynamically created every time the selector is fired, we actually extend an object called `$.fn`. This object is used as the basis to create all jQuery objects. Thus creating a plugin that transforms all the text in inputs on the page into uppercase is nominally as simple as this:

```
$.fn.yeller = function(){
  this.each(function(_, item){
    $(item).val($(item).val().toUpperCase());
    return this;
  });
};
```

This plugin is particularly useful to post to bulletin boards and for whenever my boss fills in a form. The plugin iterates over all the objects selected by the selector and converts their content to uppercase. It also returns `this`. By doing so, we allow chaining additional functions. You can use the function like this:

```
$(function(){$("input").yeller();});
```

It does rather depend on the `$` variable being assigned to jQuery. This isn't always the case, as `$` is a popular variable in JavaScript libraries, likely because it is the only character that isn't a letter or a number and doesn't really have a special meaning.

To combat this, we can use an immediately evaluated function in much the same way we did way back in *Chapter 2*, *Organizing Code*:

```
(function($){
  $.fn.yeller2 = function(){
    this.each(function(_, item){
      $(item).val($(item).val().toUpperCase());
      return this;
    });
  };
})(jQuery);
```

The added advantage here is that, should our code require helper functions or private variables, they can be set inside the same function. You can also pass in any options required. jQuery provides a very helpful `$.extend` function that copies properties between objects, making it ideal to extend a set of default options with those passed in. We looked at it in some detail in a previous chapter.

The jQuery plugin documentation recommends that the jQuery object be polluted as little as possible with plugins. This is to avoid conflicts between multiple plugins that want to use the same names. Their solution is to have a single function that has different behaviors depending on the parameters passed in. For instance, the jQuery UI plugin uses this approach for dialog:

```
$(".dialog").dialog("open");
$(".dialog").dialog("close");
```

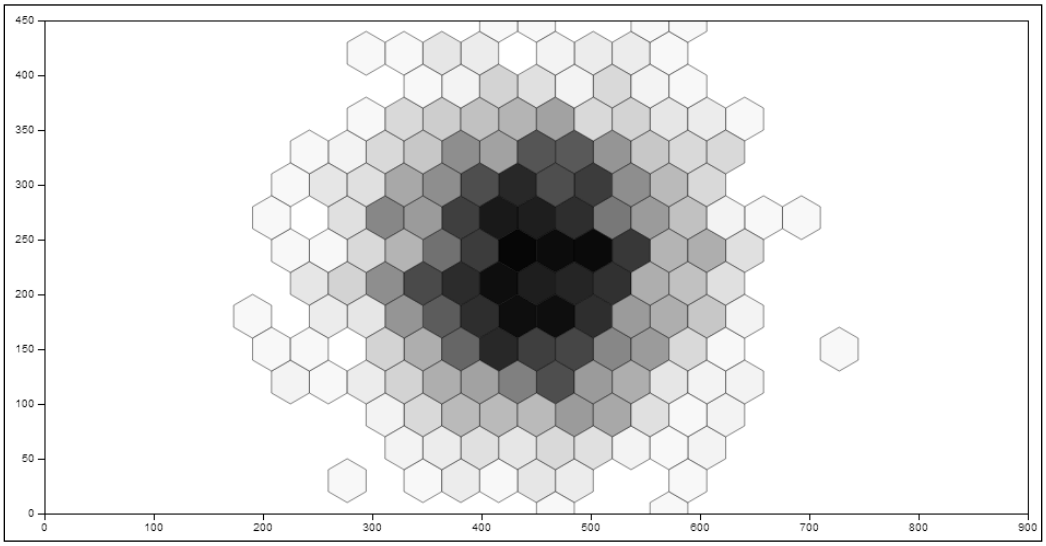
I would much rather call these like this:

```
$(".dialog").dialog().open();
$(".dialog").dialog().close();
```

With dynamic languages, there really isn't a great deal of difference but I would much rather have well-named functions that can be discovered by tooling than magic strings.

d3

d3 is a great JavaScript library that is used to create and manipulate visualizations. For the most part, people use d3 in conjunction with scalable vector graphics to produce graphics such as the following hexbin graph by Mike Bostock:



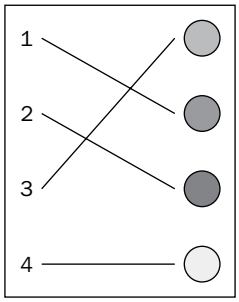
d3 attempts to be unopinionated about the sorts of visualizations it creates. Thus, there is no built-in support to create bar charts. There is, however, a collection of plugins that can be added to d3 to enable a wide variety of graphs including the hexbin one shown in the preceding graph.

jQuery d3 places emphasis on creating chainable functions. For example, the following code is a snippet that creates a column chart. You can see that all the attributes are being set through chaining:

```
var svg = d3.select(containerId).append("svg")
var bar = svg.selectAll("g").data(data).enter().append("g");
bar.append("rect")
  .attr("height", yScale.rangeBand()).attr("fill", function (d, _)
  {
    return colorScale.getColor(d);
  })
  .attr("stroke", function (d, _){
    return colorScale.getColor(d);
  })
  .attr("y", function (d, i) {
    return yScale(d.Id) + margins.height;
  })
```

The core of d3 is the d3 object. The d3 object contains a number of namespaces for layouts, scales geometry, and numerous others. As well as whole namespaces, there are functions to carry out array manipulation and loading data from external sources.

Creating a plugin for d3 starts with deciding where we're going to plug into the code. Let's build a plugin that creates a new color scale. A color scale is used to map a domain of values to a range of colors. For instance, we might wish to map the domain of four values below onto a range of four colors, as shown in the following diagram:



Let's plug in a function to provide a new color scale, in this case one that supports grouping elements. A scale is a function that maps a domain to a range; for a color scale, the range is a set of colors. An example might be a function that maps all even numbers to red and all odd to white. Using the following scale on a table would result in zebra striping:

```
d3.scale.groupedColorScale = function () {
  var domain, range;

  function scale(x) {
    var rangeIndex = 0;
    domain.forEach(function (item, index) {
      if (item.indexOf(x) > 0)
        rangeIndex = index;
    });
    return range[rangeIndex];
  }

  scale.domain = function (x) {
    if (!arguments.length)
      return domain;
    domain = x;
    return scale;
  };

  scale.range = function (x) {
    if (!arguments.length)
      return range;
    range = x;
    return scale;
  };

  return scale;
};
```

We simply attach this plugin to the existing `d3.scale` object. This can be used by simply giving an array of arrays as a domain and an array as a range:

```
var s = d3.scale.groupedColorScale().domain([[1, 2, 3], [4,
5]]).range(["#111111", "#222222"]);
s(3); //#111111
s(4); //#222222
```

This simple plugin extends the functionality of d3's scale. We could have replaced the existing functionality or even wrapped it such that calls to the existing functionality would be proxied through our plugin.

Plugins are generally not that difficult to build but they do vary from library to library. It is important to keep an eye on the existing variable names in libraries so we don't end up clobbering them or even clobbering the functionality provided by other plugins. Some suggest prefixing functions with a string to avoid clobbering.

If the library has been designed with plugins in mind, there may be additional places to which we can hook. A popular approach is to provide an options object that contains optional fields to hook in our own functions as event handlers. If nothing is provided, the function continues as normal.

Doing two things at once – multithreading

Doing two things at once is hard. For many years, the solution in the computer world was to use either multiple processes or multiple threads. The difference between the two is fuzzy due to implementation differences on different operating systems but threads are typically lighter weight versions of processes. JavaScript on the browser supports neither of these approaches.

Historically, there has been no real need for multithreading on the browser. JavaScript was used to manipulate the user interface. When manipulating a UI, even in other languages and windowing environments, only one thread is permitted to act at a time. This avoids race conditions that would be very obvious to users.

However, as JavaScript grows in popularity, more and more complicated software is being written to run inside the browser. Sometimes, that software could really benefit from performing complex calculations in the background.

Web workers provide a mechanism to do two things at once in a browser. Although a fairly recent innovation, web workers now have good support in mainstream browsers. In effect, a worker is a background thread that can communicate with the main thread using messages. Web workers must be self-contained in a single JavaScript file.

To make use of web workers is fairly easy. We'll revisit our example from a few chapters ago when we looked at the Fibonacci sequence. The worker process listens for messages like this:

```
self.addEventListener('message', function(e) {
  var data = e.data;
  if(data.cmd == 'startCalculation'){
    self.postMessage({event: 'calculationStarted'});
    var result = fib(data.parameters.number);
    self.postMessage({event: 'calculationComplete', result:
    result});
  };
}, false);
```

Here we start a new instance of `fib` any time we get a `startCalculation` message. `fib` is simply the naive implementation from earlier.

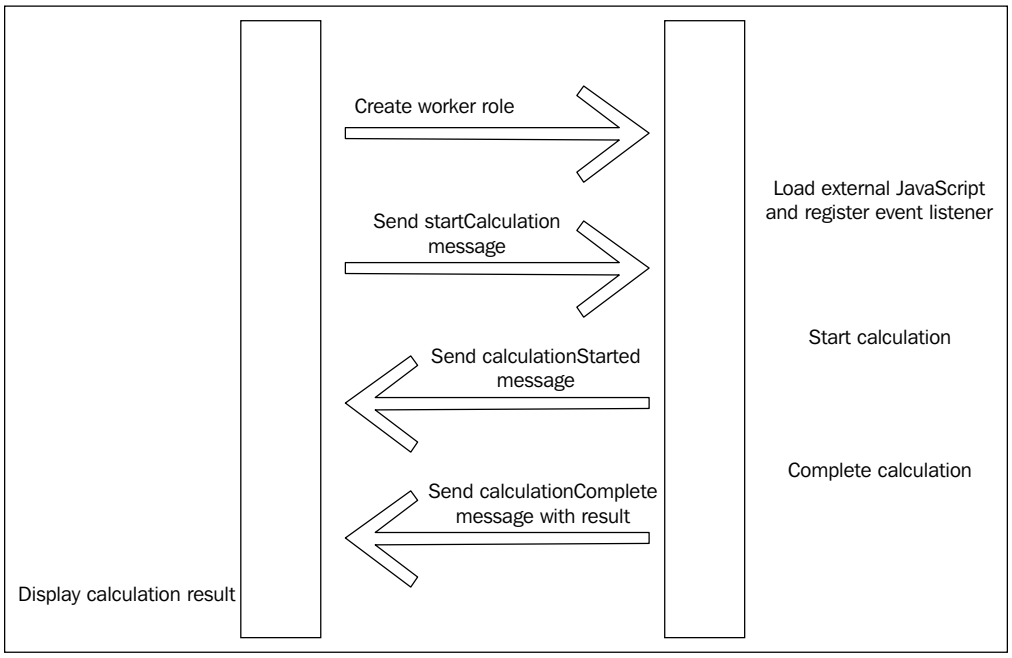
The main thread loads the worker process from its external file and attaches a number of listeners:

```
function startThread(){
  worker = new Worker("worker.js");
  worker.addEventListener('message', function(message) {
    logEvent(message.data.event);
    if(message.data.event == "calculationComplete"){
      writeResult(message.data.result);
    }
    if(message.data.event == "calculationStarted"){
      document.getElementById("result").innerHTML = "working";
    }
  });
};
```

In order to start the calculation, all that is needed is to send a command:

```
worker.postMessage({cmd: 'startCalculation',
parameters: { number: 40}});
```

Here we pass the number of the term in the sequence we want to calculate. While the calculation is running in the background, the main thread is free to do whatever it likes. When the message is received back from the worker, it is placed in the normal event loop to be processed, as shown in the following diagram:



Web workers may be useful to you if you have to do any time-consuming calculations in JavaScript.

If you're making use of server-side JavaScript through the use of Node.js, then there is a different approach to doing more than one thing at a time. Node.js offers the ability to fork child processes and provides an interface not dissimilar to the web worker one to communicate between the child and parent processes. This method forks an entire process, though, which is much more resource intensive than using lightweight threads.

Some other tools exist that create lighter weight background workers in Node.js. These are probably a closer parallel to what exists on the client side than forking a child process.

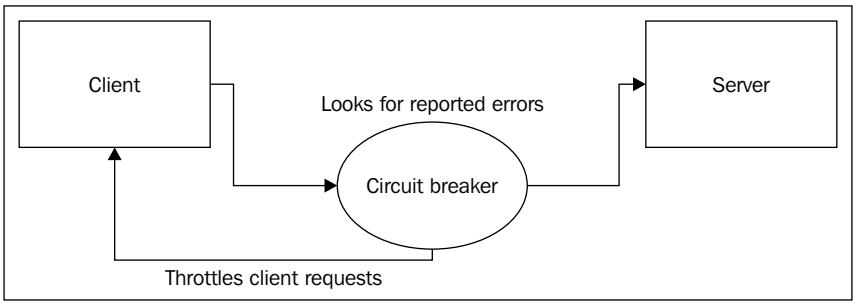
The circuit breaker pattern

Systems, even the best designed systems, fail. The larger and more distributed a system, the higher the probability of failure. Many large systems such as Netflix or Google have extensive built-in redundancies. The redundancies don't decrease the chance of a failure of a component, but they do provide a backup. Switching to the backup is frequently transparent to the end user.

The circuit breaker pattern is a common component of a system that provides this sort of redundancy. Let's say that your application queries an external data source every five seconds; perhaps you're polling for some data that you're expecting to change. What happens when this polling fails? In many cases, the failure is simply ignored and the polling continues. This is actually a pretty good behavior on the client side as data updates are not always crucial. In some cases, a failure will cause the application to retry the request immediately. Retrying server requests in a tight loop can be problematic for both the client and the server. The client may become unresponsive as it spends more time in a loop requesting data.

On the server side, a system that is attempting to recover from a failure is being slammed every five seconds by, what could be, thousands of clients. If the failure is due to the system being overloaded, then continuing to query it will only make matters worse.

The circuit breaker pattern stops attempting to communicate with a system that is failing once a certain number of failures have been reached. Basically, repeated failures result in the circuit being broken and the application ceasing to query, as shown in the following diagram:



For the server, having the number of clients drop off as failures pile up allows for some breathing room to recover. The chances of a storm of requests coming in and keeping the system down is minimized.

Of course, we would like the circuit breaker to reset at some point so that the service can be restored. The two approaches for this are that either the client polls periodically (less frequently than before) and resets the breaker or that the external system communicates back to its clients that the service has been restored.

Back-off

A variation on the circuit breaker pattern is to use some form of back-off instead of cutting out communication to the server completely. This is an approach that is suggested by many database vendors and cloud providers. If our original polling was a 5-second interval, then when a failure is detected, change the interval to every 10 seconds. Repeat this process using longer and longer intervals.

When requests start to work again, then the pattern of changing the time interval is reversed. Requests are sent closer and closer together until the original polling interval is resumed.

Monitoring the status of the external resource availability is a perfect place to use background worker roles. The work is not complex but it is totally detached from the main event loop.

Again, this reduces the load on the external resource, giving it more breathing room. It also keeps the clients unburdened by too much polling.

An example using jQuery's `ajax` function looks like the following code:

```
$.ajax({
  url : 'someurl',
  type : 'POST',
  data : ....,
  tryCount : 0,
  retryLimit : 3,
  success : function(json) {
    //do something
  },
  error : function(xhr, textStatus, errorThrown ) {
    if (textStatus == 'timeout') {
      this.tryCount++;
      if (this.tryCount <= this.retryLimit) {
        //try again
```

```
            $.ajax(this);
            return;
          }
          return;
        }
        if (xhr.status == 500) {
          //handle error
        } else {
          //handle error
        }
      }
    });
```

You can see that the highlighted section retries the query.

This style of back-off is actually used in Ethernet to avoid repeated packet collisions.

Degraded application behavior

There is likely a very good reason that your application is calling out to external resources. Backing off and not querying the data source is perfectly reasonable but it is still desirable that users have some ability to interact with the site. One solution to this problem is to degrade the behavior of the application.

For instance, if your application shows real-time stock quote information but the system to deliver stock information is broken, then a less than real-time service could be swapped in. Modern browsers have a whole raft of different technologies that allow storing small quantities of data on the client computer. This storage space is ideal to cache old versions of some data should the latest versions be unavailable.

Even in cases where the application is sending data to the server, it is possible to degrade behavior. Saving the data updates locally and then sending them en masse when the service is restored is generally acceptable. Of course, once a user leaves a page, then any background works will terminate. If the user never returns to the site, then whatever updates they had queued to send to the server will be lost.

A word of warning: If this is an approach you take, it might be best to warn users that their data is old—especially if your application is a stock trading application.

The promise pattern

I said earlier that JavaScript is single threaded. This is not entirely accurate. There is a single event loop in JavaScript. Blocking this event loop with a long running process is considered to be bad form. Nothing else can happen while your greedy algorithm is taking up all the CPU cycles.

When you launch an asynchronous function in JavaScript, such as fetching data from a remote server, then much of this activity happens in a different thread. The success or failure handler functions are executed in the main event thread. This is part of the reason that success handlers are written as functions: it allows them to be easily passed back and forth between contexts.

Thus, there are activities that truly do happen in an asynchronous, parallel fashion. When the `async` method has completed, then the result is passed into the handler we provided and the handler is put into the event queue to be picked up next time the event loop repeats. Typically, the event loop runs many hundreds or thousands of times a second, depending on how much work there is to do on each iteration.

Syntactically, we write the message handlers as functions and hook them up:

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
if (xmlhttp.readyState === 4){
  alert(xmlhttp.readyState);
}
;};
```

This is reasonable if the situation is simple. However, if you would like to perform some additional asynchronous actions with the results of the callback, then you end up with nested callbacks. If you need to add error handling, that too is done using callbacks. The complexity of waiting for multiple callbacks to return and orchestrating your response rises quickly.

The promise pattern provides some syntactic help to clean up the asynchronous difficulties. If we take a common asynchronous operation such as retrieving data over XMLHttpRequest using jQuery, then the code takes both an error and a success function. It might look something like this:

```
$.ajax("/some/url",
{ success: function(data, status){},
  error: function(jqXHR, status){}
});
```

Using a promise instead would transform the code to look more like this:

```
$.ajax("/some/url").then(successFunction, errorFunction);
```

In this case, the `$.ajax` method returns a promise object that contains a value and a state. The value is populated when the `async` call completes. The status provides some information about the current state of the request: has it completed, was it successful?

The promise also has a number of functions called on it. The `then()` function takes a success and an error function; it returns an additional promise. Should the success function run synchronously, then the promise is returned as already fulfilled. Otherwise, it remains in a working state, known as pending, until the asynchronous success has fired.

In my mind, the method in which jQuery implements promises is not ideal. Their error handing doesn't properly distinguish between a promise that has failed to be fulfilled and a promise that has failed but has been handled. This renders jQuery promises incompatible with the general idea of promises. For instance, it is not possible to use the following code:

```
$.ajax("/some/url").then(
  function(data, status){},
  function(jqXHR, status){
    //handle the error here and return a new promise
  }
).then(/*continue*/);
```

Even though the error has been handled and a new promise has been returned, processing will discontinue. It would be much better if the function could be written as:

```
$.ajax("/some/url").then(function(data, status){})
  .catch(function(jqXHR, status){
    //handle the error here and return a new promise
  })
  .then(/*continue*/);
```

There has been much discussion about the implementation of promises in jQuery and other libraries. As a result of this discussion, the current proposed promise specification is different from jQuery's promises and is incompatible. Promises/A+ is the certification that is met by numerous promise libraries such as When.js and Q, which also form the foundation of the specification of the promises that will be fulfilled by ECMAScript 6.

Promises provide a bridge between synchronous and asynchronous functions, in effect turning the asynchronous functions into something that can be manipulated as if it were synchronous.

If promises sound a lot like the lazy evaluation pattern we saw some chapters ago, then you're exactly correct. Promises are constructed using lazy evaluation; the actions called on them are queued inside the object rather than being evaluated at once. This is a wonderful application of a functional pattern and even enables scenarios like this:

```
when(function(){return 2+2;})
.delay(1000)
.then(function(promise){ console.log(promise());})
```

Promises greatly simplify asynchronous programming in JavaScript and should certainly be considered for any project that is heavily asynchronous in nature.

Hints and tips

ECMAScript 6 promises are not, at the time of this writing, available on all browsers or JavaScript environments. There are some great shims out there that can add the functionality with a minimum of overhead.

When examining the performance of retrieving JavaScript from a remote server, there are tools provided in most modern browsers to view a timeline of resource loading. This timeline will show when the browser is waiting for scripts to be downloaded and when it is parsing the scripts. Using this timeline allows us to experiment and find the best way to load a script or series of scripts.

Summary

In this chapter, we've looked at a number of patterns or approaches that improve the experience of developing JavaScript. We looked at a number of concerns around delivery to the browser. We also looked at how to implement plugins against a couple of libraries and extrapolated general practices. Next, we took a look at how to work with background processes in JavaScript. Circuit breakers were suggested as a method of keeping remote resource fetching sane. Finally, we examined how promises can improve the writing of asynchronous code.

In the next chapter, we'll spend quite a bit more time looking at messaging patterns. We saw a little about messing with web workers, but we'll expand quite heavily on them in the next section.

9
Messaging Patterns

When Smalltalk was first envisioned, the communication between classes was envisioned as being messages. Somehow we've moved away from this pure idea of messages. We spoke a bit about how functional programming avoids side effects; well, much the same is true of messaging-based systems.

Messaging also allows for impressive scalability as messages can be fanned out to dozens or even hundreds of computers. Within a single application, messaging promotes low coupling and eases testing.

In this chapter, we're going to look at a number of patterns related to messaging. By the end of the chapter, you should be aware of how messages work. When I first learned about messaging, I wanted to rewrite everything using it.

We will be covering:

- What's a message anyway?
    - Commands
    - Events
- Request-reply
- Publish-subscribe
    - Fan out and fan in
- Dead-letter queues
- Message replay
- Pipes and filters

What's a message anyway?

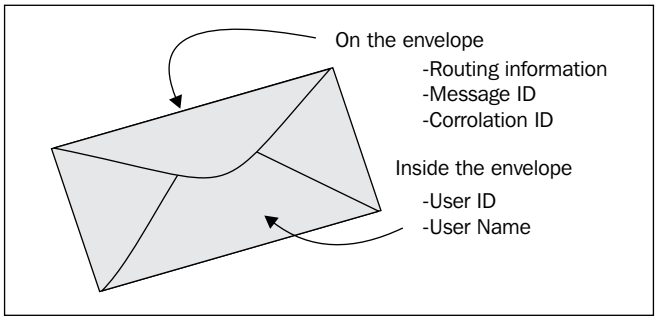
In the simplest definition, a message is a collection of related bits of data that have some meaning together. The message is named in a way that provides some additional meaning to it. For instance, both an `AddUser` message and a `RenameUser` message might have the following fields:

- User ID
- Username

But the fact that the fields exist inside a named container gives them a different meaning.

Messages are usually related to some action in the application or some action in the business. A message contains all the information needed for a receiver to act upon the action. In the case of the `RenameUser` message, the message contains enough information for any component that keeps track of a relationship between a user ID and a username to update its value for username.

Many messaging systems, especially those that communicate between application boundaries, also define an envelope. The envelope has metadata in it that could help with message auditing, routing, and security. The information on the envelope is not part of the business process but is part of the infrastructure. So having a security annotation on the envelope is fine as security exists outside of the normal business workflow and is owned by a different part of the application. An example message might contain:



Messages should be sealed so that no changes can be made to them once they have been created. This makes certain operations such as auditing and replaying much easier.

Messaging can be used to communicate inside a single process or it can be used between applications. For the most part, there is no difference between sending a message within an application and sending it between applications. One difference is the treatment of synchronicity. Within a single process, messages can be handled in a synchronous fashion. This means that the main processing effectively waits for the handling of the message to complete before continuing.

In an asynchronous scenario, the handling of the message may occur at a later date. Sometimes, the later date is far in the future. When calling out to an external server, the asynchronous approach will almost certainly be the correct one. Even within a single process, the single-threaded nature of JavaScript encourages using asynchronous messaging. While using asynchronous messaging, some additional care and attention needs to be taken, as some of the assumptions made for synchronous messaging cease to be safe. For instance, assuming the messages will be replied to in the same order in which they were sent is no longer safe.

There are two different flavors of messages: commands and events. Commands instruct things to happen while events notify about something that has happened.

Commands

A command is simply an instruction from one part of a system to another. It is a message so it is really just a simple data transfer object. If you think back to the command pattern introduced in *Chapter 5*, *Behavioral Patterns*, this is exactly what it uses.

As a matter of convention, commands are named using the imperative. The format is usually `<verb><object>`. Thus, a command might be called `InvadeCity`. Typically, when naming a command, you want to avoid generic names and focus on exactly what is causing the command.

As an example, consider a command that changes the address of a user. You might be tempted to simply call the command `ChangeAddress`, but doing so does not add any additional information. It would be better to dig deeper and see why the address is being changed. Did the person move or was the original address entered incorrectly? Each case might include a different behavior. Users that have moved could be sent a moving present, while those correcting their address would not.

Messages should have a component of business meaning to increase their utility. Defining messages and how they can be constructed within a complex business is a whole field of study on its own. Those interested might do well to investigate **domain-driven design** (**DDD**).

Commands are an instruction targeted at one specific component, giving it instructions to perform a task.

Within the context of a browser, you might consider that a command would be the click that is fired on a button. The command is transformed into an event and that event is what is passed to your event listeners.

Only one end point should receive a specific command. This means that only one component is responsible for an action taking place. As soon as a command is acted upon by more than one end point, any number of race conditions are introduced. What if one of the end points accepts the command and another rejects it as invalid? Even in cases where several near identical commands are issued, they should not be aggregated. For instance, sending a command from a king to all his generals should send one command to each general.

Because there is only one end point for a command, it is possible for that end point to validate and even cancel the command. The cancellation of the command should have no impact on the rest of the application.

When a command is acted upon, then one or more events may be published.

Events

An event is a special message that notifies that something has happened. There is no use in trying to change or cancel an event because it is simply a notification that something has happened. You cannot change the past unless you own a Delorian.

The naming convention for events is that they are written in the past tense. You might see a reversal in the ordering of the words in the command, so we could end up with `CityInvaded` once the `InvadeCity` command has succeeded.

Unlike commands, events may be received by any number of components. There are no real-race conditions presented by this approach. As no message handler can change the message or interfere with the delivery of other copies of the message, each handler is siloed away from all others.

You may be familiar with events from having done user interface work. When a user clicks on a button, an event is "raised." In effect, the event is broadcast to a series of listeners. You subscribe to a message by hooking into that event:

```
document.getElementById("button1").addEventListener("click",
doSomething);
```

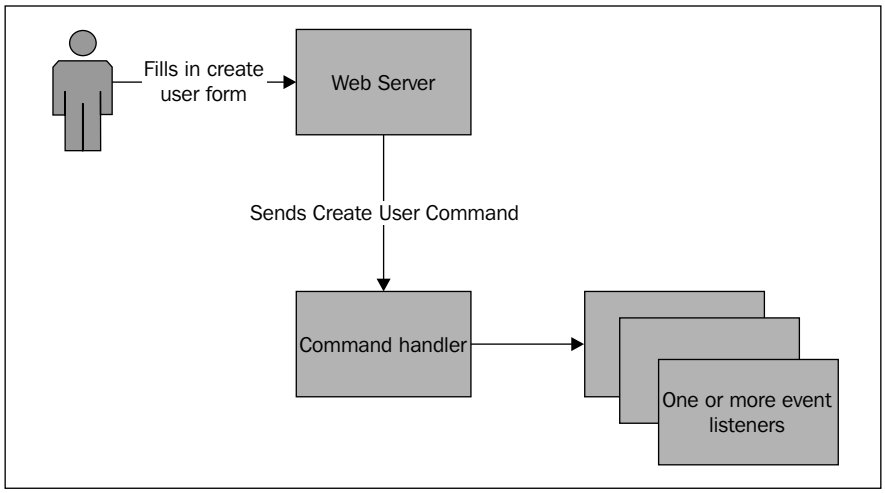
The events in the browsers don't quite meet the definition of an event I gave earlier. This is because event handlers in the browser can cancel events and stop them from propagating to the next handler. That is to say that when there are a series of event handlers for the same message, one of them can completely consume the message and not pass it onto subsequent handlers. There is certainly utility to an approach like this but it does introduce some confusion. Fortunately, for UI messages the number of handlers is typically quite small.

In some systems, events can be polymorphic in nature. That is to say that if I had an event called `IsHiredSalary` that is fired when somebody is hired in a salaried role, I could make it a descendant of the message `IsHired`. Doing so would allow for both handlers subscribed to `IsHiredSalary` and `IsHired` to be fired upon receipt of an `IsHiredSalary` event. JavaScript doesn't have polymorphism in the true sense, so such things aren't particularly useful. You can add a message field that takes the place of polymorphism but it looks somewhat messy:

```
var IsHiredSalary = { __name: "isHiredSalary",
  __alsoCall: ["isHired"],
  employeeId: 77,
  …
}
```

In this case, I've used __ to denote fields that are part of the envelope. You could also construct the message with separate fields for message and envelope; it really doesn't matter all that much.

Let's take a look at a simple operation such as creating a user, so we can see how commands and events interact:



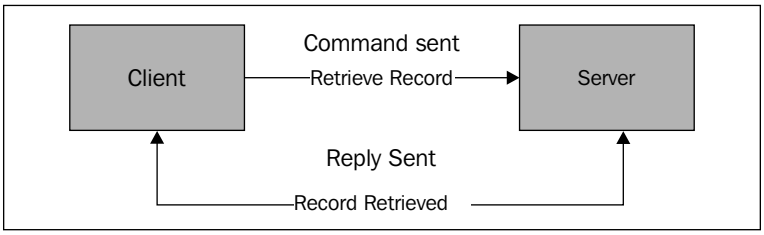
Here, a user enters data into a form and submits it. The web server takes in the input, validates it, and, if it is correct, creates a command. The command is now sent to the command handler. The command handler performs some action, perhaps writing to a database; it then publishes an event that is consumed by a number of event listeners. These event listeners might send confirmation e-mails, notify system administrators, or any number of things.

All of this looks familiar because systems already contain commands and events. The difference is that we are now modeling the commands and events explicitly.

Request-reply

The simplest pattern you'll see with messaging is the request-reply pattern. Also known as request-response, this is a method of retrieving data that is owned by another part of the application.

In many cases, the sending of a command is an asynchronous operation. A command is fired and the application flow continues on. Because of this, there is no easy way to do things such as look up a record by ID. Instead, one needs to send a command to retrieve a record and then wait for the associated event to be returned. The workflow might look like:



Most events can be subscribed to by any number of listeners. While it is possible to have multiple event listeners for a request-response pattern, it is unlikely and is probably not advisable.

We can implement a very simple request-response pattern here. In Westeros, there are some issues with sending messages in a timely fashion. Without electricity, sending messages over long distances rapidly can really only be accomplished by attaching tiny messages to the legs of crows. Thus, we have a **crow messaging system**.

We'll start with building out what we'll call the bus. A bus is simply a distribution mechanism for messages. It can be implemented in process, as we've done here, or out of process.

If implementing it out of process, there are many options from ZeroMQ to RabbitMQ to a wide variety of systems built on top of databases and in the cloud. Each of these systems exhibits some different behaviors when it comes to message reliability and durability. It is important to do some research into the way the message distribution systems work as they may dictate how the application is constructed. They also implement different approaches to dealing with the underlying unreliability of applications:

```
var CrowMailBus = (function () {
  function CrowMailBus(requestor) {
    this.requestor = requestor;
    this.responder = new CrowMailResponder(this);
  }
  CrowMailBus.prototype.Send = function (message) {
    if (message.__from == "requestor") {
      this.responder.processMessage(message);
    } else {
      this.requestor.processMessage(message);
    }
  };
  return CrowMailBus;
})();
```

One thing that is a potential trip up is that the order in which messages are received back on the client is not necessarily the order in which they were sent. To deal with this, it is typical to include some sort of a correlation ID. When the event is raised, it includes a known ID from the sender so that the correct event handler is used.

This bus is a highly naïve one as it has its routing hardcoded. A real bus would probably allow the sender to specify the address of the end point for delivery. Alternately, the receivers could register themselves as interested in a specific sort of message. The bus would then be responsible for doing some limited routing to direct the message. Our bus is even named after the messages it deals with—certainly not a scalable approach.

Next, we'll implement the requestor. The requestor contains only two methods: one to send a request and the other to receive a response from the bus, as shown in the following code:

```
var CrowMailRequestor = (function () {
  function CrowMailRequestor() {
  }
```

```
  CrowMailRequestor.prototype.Request = function () {
    var message = {
      __messageDate: new Date(),
      __from: "requestor",
      __corrolationId: new Guid(),
      body: "Invade Moat Cailin"
    };
    var bus = new CrowMailBus(this);
    bus.Send(message);
  };

  CrowMailRequestor.prototype.processMessage = function (message)
  {
    console.dir(message);
  };
  return CrowMailRequestor;
})();
```

The process message function currently just logs the response but it would likely do more in a real-world scenario such as updating the UI or dispatching another message. The correlation ID is invaluable in order to understand to which `Send` message the reply is related.

Finally, the responder simply takes in the message and replies to it with another message:

```
var CrowMailResponder = (function () {
  function CrowMailResponder(bus) {
    this.bus = bus;
  }
  CrowMailResponder.prototype.processMessage = function (message)
  {
    var response = {
      __messageDate: new Date(),
      __from: "responder",
      __corrolationId: message.__corrolationId,
      body: "Okay, invaded
    };
    this.bus.Send(response);
  };
  return CrowMailResponder;
})();
```

Everything in our example is synchronous but all it would take to make it asynchronous is to swap out the bus. If we're working in Node.js, then we can do this using `process.nextTick`, which simply defers a function to the next time through the event loop. If we're in a web context, then web workers may be used to do the processing in another thread. In fact, when starting a web worker, the communication back and forth to it takes the form of a message:

```
CrowMailBus.prototype.Send = function (message) {
  var _this = this;
  if (message.__from == "requestor") {
    process.nextTick(function () {
      return _this.responder.processMessage(message);
    });
  } else {
    process.nextTick(function () {
      return _this.requestor.processMessage(message);
    });
  }
};
```

This approach now allows other code to run before the message is processed. If we weave in some print statements after each bus `Send`, then we get the following output:

```
Request sent!
Reply sent
{ __messageDate: Mon Aug 11 2014 22:43:07 GMT-0600 (MDT),
  __from: 'responder',
  __corrolationId: 0.5604551520664245,
  body: 'Okay, invaded.' }
```

You can see that the print statements are executed before the message processing, as that processing happens next to iteration.

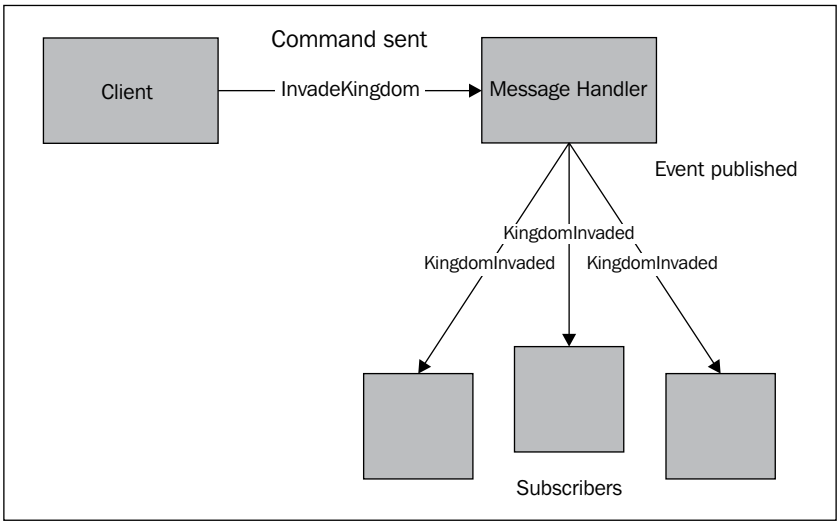
Publish-subscribe

I've alluded to the publish-subscribe model elsewhere in this chapter. Publish-subscribe is a powerful tool to decouple events from processing code.

At the crux of the pattern is the idea that, as a message publisher, my responsibility for the message should end as soon as I send it. I should not know who is listening to messages or what they will do with the messages. So long as I am fulfilling a contract to produce correctly formatted messages, the rest shouldn't matter.

It is the responsibility of the listener to register its interest in the message type. You'll, of course, wish to register some sort of security to disallow registration of rogue services.

We can update our service bus to do a more complete job of routing and sending multiple messages. Let's call our new method `Publish` instead of `Send`. We'll keep `Send` around to do the sending functionality, as shown in the following diagram:



The crow mail analogy we used in the previous section starts to fall apart here as there is no way to broadcast a message using crows. Crows are too small to carry large banners and it is very difficult to train them to do sky writing. I'm unwilling to totally abandon the idea of crows, so let's assume that there exists a sort of crow broadcast center. Sending a message here allows for it to be fanned out to numerous interested parties who have signed up for updates. This center will be more or less synonymous with a bus.

We'll write our router so that it works as a function of the name of the message. One could route a message using any of its attributes. For instance, a listener could subscribe to all the messages called `invoicePaid` where the `amount` field is greater than $10,000. Adding this sort of logic to the bus will slow it down and make it far harder to debug. Really, this is more the domain of business process orchestration engines than a bus. We'll continue on without that complexity.

The first thing to set up is the ability to subscribe to published messages:

```
CrowMailBus.prototype.Subscribe = function (messageName,
subscriber) {
  this.responders.push({ messageName: messageName, subscriber:
  subscriber });
};
```

The `Subscribe` function just adds a message handler and the name of a message to consume. The responders array is simply an array of handlers.

When a message is published, we loop over the array and fire each of the handlers that has registered for messages with that name:

```
CrowMailBus.prototype.Publish = function (message) {
  for (var i = 0; i < this.responders.length; i++) {
    if (this.responders[i].messageName == message.__messageName) {
      (function (b) {
        process.nextTick(function () {
          return b.subscriber.processMessage(message);
        });
      })(this.responders[i]);
    }
  }
};
```

The execution here is deferred to the next tick. This is done using a closure to ensure that the correctly scoped variables are passed through. We can now change `CrowMailResponder` to use the new `Publish` method instead of `Send`:

```
CrowMailResponder.prototype.processMessage = function (message) {
  var response = {
    __messageDate: new Date(),
    __from: "responder",
    __corrolationId: message.__corrolationId,
    __messageName: "KingdomInvaded",
    body: "Okay, invaded"
  };
  this.bus.Publish(response);
  console.log("Reply published");
};
```

Instead of allowing `CrowMailRequestor` to create its own bus as earlier, we need to modify it to accept an instance of bus from outside. We simply assign it to a local variable in `CrowMailRequestor`. Similarly, `CrowMailResponder` should also take in an instance of bus.

In order to make use of this, we simply need to create a new bus instance and pass it into the requestor:

```
var bus = new CrowMailBus();
bus.Subscribe("KingdomInvaded", new TestResponder1());
bus.Subscribe("KingdomInvaded", new TestResponder2());
var requestor = new CrowMailRequestor(bus);
requestor.Request();
```

In the preceding code, we've also passed in two other responders that are interested in knowing about `KingdomInvaded` messages. They look like this:

```
var TestResponder1 = (function () {
  function TestResponder1() {}
  TestResponder1.prototype.processMessage = function (message) {
    console.log("Test responder 1: got a message");
  };
  return TestResponder1;
})();
```

By running this code, we will now get this:

```
Message sent!
Reply published
Test responder 1: got a message
Test responder 2: got a message
Crow mail responder: got a message
```

You can see that the messages are sent using `Send`. The responder or handler does its work and publishes a message that is passed onto each of the subscribers.

Fan out and fan in

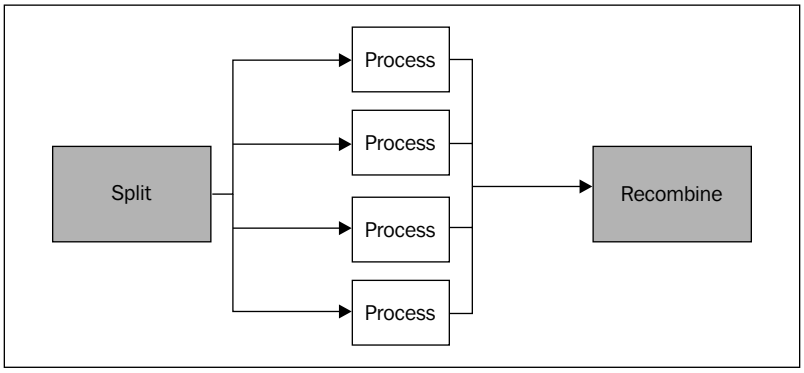
A fantastic use of the publish-subscribe pattern allows you to fan out a problem to a number of different nodes. Moore's law has always been about the doubling of the number of transistors per square unit of measure. If you've been paying attention to processor clock speeds, you may have noticed that there hasn't really been any significant change in clock speeds for a decade. In fact, clock speeds are not lower than they were in 2005.

This is not to say that processors are "slower" than they once were. The work that is performed in each clock tick has increased. The number of cores has also jumped up. It is now unusual to see a single core processor; even in cellular phones, dual core processors are becoming common. It is the rule, rather than the exception, to have computers that are capable of doing more than one thing at a time.

At the same time, cloud computing is taking off. The computers you purchase outright are faster than the ones available to rent from the cloud. The advantage of cloud computing is that you can scale it out easily. It is nothing to provision a hundred or even a thousand computers from a cloud provider.

Writing software that can take advantage of multiple cores is the great computing problem of our time. Dealing directly with threads is a recipe for disaster. Locking and contention is a far too difficult problem for most developers: me included! For a certain class of problems, they can easily be divided up into subproblems and distributed. Some call this class of problems "embarrassingly parallelizable."

Messaging provides a mechanism to communicate the inputs and outputs from a problem. If we had one of these easily parallelized problems, such as searching, then we would bundle up the inputs into one message. In this case, it would contain our search terms. The message might also contain the set of documents to search. If we had 10,000 documents, then we could divide the search space up into, say, four collections of 2,500 documents. We would publish five messages with the search terms and the range of documents to search. A message span-out might look like:



Different search nodes will pick up the messages and perform the search. The results will then be sent back to a node that will collect the messages and combine them into one. This is what will be returned to the client.

Of course, this is a bit of an oversimplification. It is likely that the receiving nodes themselves would maintain a list of documents over which they had responsibility. This would prevent the original publishing node from having to know anything about the documents it was searching through. The search results could even be returned directly to the client that would do the assembling.

Even in a browser, the fan-out-and-in approach can be used to distribute a calculation over a number of cores through the use of web workers. A simple example might take the form of creating a potion. A potion might contain a number of ingredients that can be combined to create a final product. It is quite computationally complicated combining ingredients, so we would like to farm a process out to a number of workers.

We start with a combiner that contains a `combine()` method as well as a `complete()` function that is called once all the distributed ingredients are combined:

```
var Combiner = (function () {
  function Combiner() {
    this.waitingForChunks = 0;
  }
  Combiner.prototype.combine = function (ingredients) {
    var _this = this;
    console.log("Starting combination");
    if (ingredients.length > 10) {
      for (var i = 0; i < Math.ceil(ingredients.length / 2); i++)
      {
        this.waitingForChunks++;
        console.log("Dispatched chunks count at: " +
        this.waitingForChunks);
        var worker = new Worker("FanOutInWebWorker.js");
        worker.addEventListener('message', function (message) {
          return _this.complete(message);
        });
        worker.postMessage({ ingredients: ingredients.slice(i, i *
        2) });
      }
    }
  };
  Combiner.prototype.complete = function (message) {
    this.waitingForChunks--;
    console.log("Outstanding chunks count at: " +
    this.waitingForChunks);
    if (this.waitingForChunks == 0)
    console.log("All chunks received");
  };
  return Combiner;
})();
```

In order to keep track of the number of workers outstanding, we use a simple counter. Because the main section of code is single threaded, we have no risk of race conditions. Once the counter shows no remaining workers, we can take whatever step is necessary. The web worker looks like this:

```
self.addEventListener('message', function (e) {
  var data = e.data;
  var ingredients = data.ingredients;
  combinedIngredient = new Westeros.Potion.CombinedIngredient();
  for (var i = 0; i < ingredients.length; i++) {
    combinedIngredient.Add(ingredients[i]);
  }
  console.log("calculating combination");
  setTimeout(combinationComplete, 2000);
}, false);

function combinationComplete() {
  console.log("combination complete");
  (self).postMessage({ event: 'combinationComplete', result:
  combinedIngredient });
}
```

In this case, we simply put in a timeout to simulate the complex calculation needed to combine ingredients.

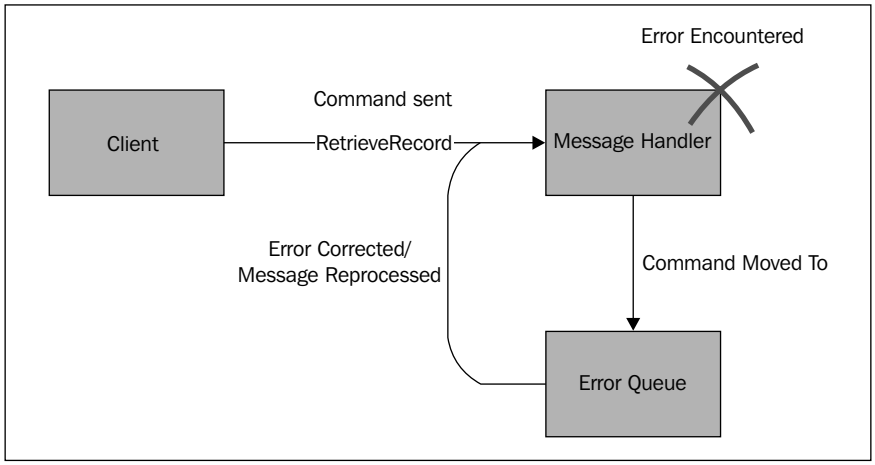
The subproblems that are farmed out to a number of nodes do not have to be identical problems. However, they should be sufficiently complicated that the cost savings of farming them out is not consumed by the overhead of sending out messages.

Dead-letter queues

No matter how hard I try, I have yet to write any significant block of code that does not contain any errors. Nor have I been very good at predicting the wide range of crazy things users do with my applications. Why would anybody click on that link 73 times in a row? I'll never know.

Dealing with failures in a messaging scenario is very easy. The core of the failure strategy is to embrace errors. We have exceptions for a reason and to spend all of our time trying to predict and catch exceptions is counterproductive. You'll invariably spend time building in catches for errors that never happen and miss errors that happen frequently.

In an asynchronous system, errors need not be handled as soon as they occur. Instead, the message that caused an error can be put aside to be examined by an actual human later. The message is stored in a dead letter or error queue. From there, the message can easily be reprocessed after it has been corrected, or the handler has been corrected. Ideally, the message handler is changed to deal with messages exhibiting whatever property caused the errors. This prevents future errors and is preferable to fixing whatever generates the message, as there is no guarantee that other messages with the same problem aren't lurking somewhere else in the system:



As more and more errors are caught and fixed, the quality of the message handlers increases. Having an error queue of messages ensures that nothing important, such as a `BuySimonsBook` message, is missed. This means that getting to a correct system becomes a marathon instead of a sprint. There is no need to rush a fix into production before it is properly tested. Progress towards a correct system is constant and reliable.

Using a dead-letter queue also improves the catching of intermittent errors. These are errors that result from an external resource being unavailable or incorrect. Imagine a handler that calls out to an external web service. In a traditional system, a failure in the web service guarantees a failure in the message handler. However, with a message-based system, the command can be moved back to the end of the input queue and tried again whenever it reaches the front of the queue. On the envelope, we write down the number of times the message has been dequeued (processed). Once this dequeue count reaches a limit, such as 5, only then is the message moved into the true error queue.

This approach improves the overall quality of the system by smoothing over the small failures and stopping them from becoming large failures. In effect, the queues provide failure bulkheads to prevent small errors from overflowing and becoming large errors that might have an impact on the system as a whole.

Message replay

When developers are working with a set of messages that produce an error, the ability to reprocess messages is also useful. Developers can take a snapshot of the dead-letter queue and reprocess it in debug mode again and again until they have correctly processed the messages. A snapshot of the message can also make up a part of the testing for a message handler.

Even without there being an error, the messages sent to a service on a daily basis are representative of the normal workflows of users. These messages can be mirrored to an audit queue as they enter into the system. The data from the audit queue can be used for testing. If a new feature is introduced, then a normal day's workload can be played back to ensure than there has been no degradation in either correct behavior or performance.

Of course, if the audit queue contains a list of all the messages, then it becomes trivial to understand how the application arrived at its current state. Frequently, people implement history by plugging in a lot of custom code or by using triggers and audit tables. Neither of these approaches do as good a job as messaging at understanding not only which data has changed but why it has changed. Once again, consider the address change scenario; without messaging we will likely never know why an address for a user is different from the pervious day.

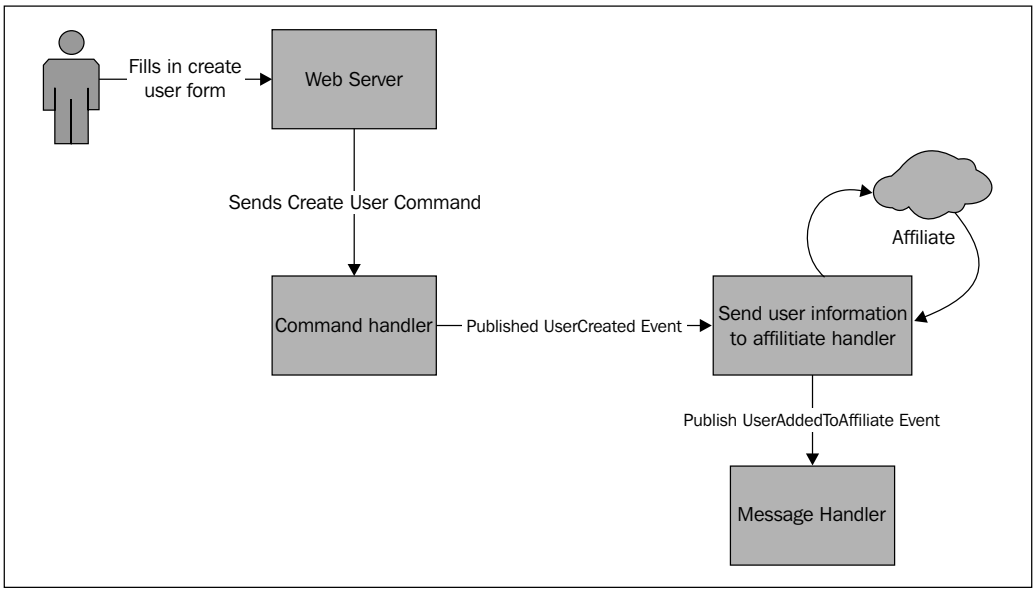
Maintaining a good history of changes to system data is storage intensive but easily paid for by improving the life of even one auditor who can now see how and why each change was made. Well-constructed messages also allow for the history to contain the intent of the user making the change.

While it is possible to implement this sort of messaging system in a single process, it is difficult. Ensuring that messages are properly saved in the event of errors is difficult, as the entire process dealing with messages may crash, taking the internal message bus with it. Realistically, if the replaying of messages sounds like something worth investigating, then external message busses are the solution.

Pipes and filters

I mentioned earlier that messages should be considered to be immutable. This is not to say that messages cannot be rebroadcast with some properties changed or even broadcast as a new type of message. In fact, many message handlers may consume an event and then publish a new event after having performed some task.

As an example, you might consider the workflow to add a new user to a system:



In this case, the `CreateUser` command triggers a `UserCreated` event. That event is consumed by a number of different services. One of these services passes on user information to a select number of affiliates. As this service runs, it publishes its own set of events, one for each affiliate that receives the new user's details. These events may, in turn, be consumed by other services that could trigger their own events. In this way, changes can ripple through the entire application. However, no service knows more than what starts it and what events it publishes. This system has very low coupling. Plugging in new functionality is trivial and even removing functionality is easy, certainly easier than in a monolithic system.

Systems constructed using messaging and autonomous components are frequently referred to as using **Service Oriented Architecture** (**SOA**) or **Microservices**. There remains a great deal of debate as to the differences, if indeed there are any, between SOA and Microservices. We won't delve into it anymore here. Perhaps, by the time you're reading this book, the question would have been answered to everybody's satisfaction.

The altering and rebroadcasting of messages can be thought of as being a pipe or a filter. A service can proxy messages through to other consumers just as a pipe would do or can selectively republish messages as would be done by a filter.

Versioning messages

As systems evolve, the information contained in a message may also change. In our user creation example, we might have originally been asking for a name and e-mail address. However, the marketing department would like to be able to send e-mails addressed to Mr. Jones or Mrs. Jones, so we need to also collect the user's title. This is where message versioning comes in handy.

We can now create a new message that extends the previous message. The message can contain additional fields and might be named using the version number or a date. Thus, a message such as `CreateUser` might become `CreateUserV1` or `CreateUser20140101`. Earlier I mentioned polymorphic messages. This is one approach to versioning messages. The new message extends the old, so all the old message handlers still fire. However, we also talked about how there are no real polymorphic capabilities in JavaScript.

Another option is to use upgrading message handlers. These handers will take in a version of the new message and modify it to be the old version. Obviously, the newer messages need to have at least as much data in them as the old version or have data that permits converting one message type to another.

If we had a `v1` message that looked like this:

```
class CreateUserv1Message implements IMessage{
  __messageName: string
  UserName: string;
  FirstName: string;
  LastName: string;
  EMail: string;
}
```

And a `v2` message that extended it adding a user title:

```
class CreateUserv2Message extends CreateUserv1Message implements
IMessage{
  UserTitle: string;
}
```

Then we would be able to write a very simple upgrader or downgrader that looks like this:

```
var CreateUserv2tov1Downgrader = (function () {
  function CreateUserv2tov1Downgrader (bus) {
    this.bus = bus;
  }
  CreateUserv2tov1Downgrader.prototype.processMessage = function
  (message) {
    message.__messageName = "CreateUserv1Message";
    delete message.UserTitle;
    this.bus.publish(message);
  };
  return CreateUserv2tov1Downgrader;
})();
```

You can see that we simply modified the message and rebroadcasted it.

Hints and tips

Messages create a well-defined interface between two different systems. Defining messages should be done by members of both teams. Establishing a common language can be tricky especially as terms are overloaded between different business units. What sales considers a customer may be totally different from what shipping considers a customer. Domain-driven design provides some hints as to how boundaries can be established to avoid mixing terms.

There is a huge preponderance of queue technologies available. Each of them have a bunch of different properties around reliability, durability, and speed. Some of the queues support reading and writing JSON over HTTP, ideal for those interested in building JavaScript applications. Which queue is appropriate for your application is a topic for some research.

Summary

Messaging and the associated patterns are a large topic. Delving too deeply into messages will bring you into contact with DDD and **command query responsibility segregation** (**CQRS**), as well as touching on high performance computing solutions.

There is substantial research and discussion ongoing as to the best way to build large systems. Messaging is one possible solution that avoids creating a big ball of mud that is difficult to maintain and fragile to change. Messaging provides natural boundaries between components in a system and the messages themselves provide for a consistent API.

Not every application benefits from messaging. There is additional overhead to building a loosely coupled application such as this. Applications that are collaborative, ones where losing data is especially undesirable and those that benefit from a strong history story, are good candidates for messaging. In most cases, a standard CRUD application will be sufficient. It is still worthwhile to know about messaging patterns, as they will offer alternative thinking.

In this chapter, we've taken a look at a number of different messaging patterns and how they can be applied to common scenarios. The differences between commands and events were also explored.

In the next chapter, we'll look at some patterns to make testing code a little bit easier. Testing is jolly important, so read on!

10
Patterns for Testing

All throughout this book, we've been pushing the idea that JavaScript is no longer a toy language with which we can't do useful things. Real-world software is being written in JavaScript right now and the percentage of applications using JavaScript is only likely to grow over the next decade.

With real software comes concerns about correctness. Manually testing software is painful and, weirdly, error prone. It is far cheaper and easier to produce unit and integration tests that run automatically and test various aspects of the application.

There are countless tools available to test JavaScript. From test runners to testing framework, the ecosystem is a rich one. We'll try to maintain a more or less tool-agnostic approach to testing in this chapter. This book does not concern itself with which framework is the best or friendliest. There are overarching patterns that are common to testing as a whole. It is those that we'll examine. We will touch on some specific tools but only as a shortcut to having to write all our own testing tools.

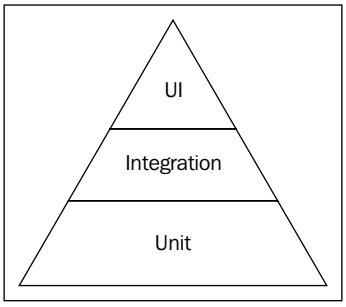
In this chapter, we'll look at

- Fake objects
- Monkey patching
- Interacting with the user interface

The testing pyramid

We computer programmers are, as a rule, highly analytical people. This means that we're always striving to categorize and understand concepts. This has lead to us developing some very interesting global techniques that can be applied outside of computer programming. For instance, agile development has applications in general society but can trace its roots back to computing. One might even argue that the idea of patterns owes much of its popularity to it being used by computer programmers in other walks of life.

This desire to categorize has lead to the concept of testing code being divided up into a number of different types of tests. I've seen as many as eight different categories of tests from unit tests right up to workflow tests and GUI tests. This is, perhaps, overkill. It is much more common to think about having three different categories of tests: unit, integration, and user interface:



Unit tests form the foundation of the pyramid. They are the most numerous, the easiest to write, and the most granular in the errors they give. An error in a unit test will allow you to find the individual method that has an error in it. As we move up the pyramid, the number of tests falls along with the granularity, while the complexity of each test increases. At a higher level, when a test fails, we might only be able to say *there is an issue with adding an order to the system*.

Test in the small with unit tests

To many, unit testing is a foreign concept. This is understandable, as it is a topic that is poorly taught in many schools. I know that I've done six years of higher education in computing science without it being mentioned. It is unfortunate because delivering a quality product is a pretty important part of any project.

For those who know about unit testing, there is a big barrier to adoption. Managers and even developers frequently see unit testing and automated testing on the whole as a waste of time. After all, you cannot ship a unit test to your customer nor are most customers interested that their product has been properly unit tested.

Unit testing is notoriously difficult to define. It is close enough to integration testing that people slip back and forth between the two easily. In the seminal book, *The Art of Unit Testing*, Roy Osherove defines a unit test thusly:

> *A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work.*

The exact size of a unit of work is up for some debate. Some people restrict it to a single function or a single class, while others allow a unit of work to span multiple classes. I tend to think that a unit of work that spans multiple classes can actually be broken into smaller, testable units.

The key to unit testing is that it tests a small piece of functionality,? and it quickly tests the functionality in a repeatable, automated fashion. Unit tests written by one person should be easily runnable by any other member of the team.

For unit testing, we want to test small pieces of functionality, because we believe that if all the components of a system work correctly, then the system as a whole will work. This is not completely true. The communication between modules is just as likely to fail as a function within the unit. This is why we want to write tests on several levels. Unit tests check that the code we're writing right now is correct. Integration testing tests entire workflows through the application and will uncover problems in the interaction of units.

The test-driven development approach suggests writing tests at the same time as we write code. While this gives us great confidence that the code we're writing is correct, the real advantage is that it helps drive good architecture. When the code has too many interdependencies, it is far harder to test than well-separated modular code. A lot of the code developers write goes unread by anybody ever again. Unit tests provide a useful way of keeping developers on the right path even in cases where they know that nobody will ever see this code. There is no better way to produce a quality product than to tell people they are going to be checked on it, even if the checker happens to be an automated test.

The test can be run both while developing new code and in an automatic fashion on the build machines. If every time a developer checks in a change the entire project is built and tested, then some reassurance can be provided that the newly checked in code is correct. From time to time, the build will break and that will be a flag that something we just added was in error. Often the code broken may not even be proximal to the code changed. An altered return value may percolate through the system and manifest itself somewhere wholly unexpected. Nobody can keep anything more than the most trivial system in their mind at any one time. Testing acts as a sort of second memory, checking and rechecking assumptions made previously.

Failing the build as soon as an error occurs shortens the time it takes between an error being made in the code and it being found and fixed. Ideally, the problem will still be fresh in the developer's mind, so the fix can easily be found. If the error were not discovered until months down the road, the developer will certainly have forgotten what they were working on at the time. The developer may not even be around to help solve the problem, throwing somebody who has never seen the code in to fix it.

Arrange-Act-Assert

When building a test for any piece of code, a very common approach to follow is that of **Arrange-Act-Assert**. This describes the different steps that take place inside of a single unit test.

The first thing we do is set up a test scenario (**arrange**). This step can consist of a number of actions and may involve putting in place fake objects to simulate real objects as well as creating new instances of the subject under test. If you find that your test setup code is long or involved, it is likely a smell and you should consider refactoring your code. As mentioned in the previous section, testing is helpful to drive not just correctness but also architecture. Difficult-to-write tests are indicative that the architecture is not sufficiently modular.

Once the test is set up, then the next step is to actually execute the function we would like to test (**act**). The act step is usually very short, in many cases no more than a single line of code.

The final part is to check to make sure that the result of the function or the state of the world is as you would expect (**assert**).

A very simple example of this might be a castle builder:

```
var CastleBuilder = (function () {
  function CastleBuilder() {
  }
  CastleBuilder.prototype.buildCastle = function (size) {
    var castle = new Castle();
    castle.size = size;
    return castle;
  };
  return CastleBuilder;
})();
```

The preceding class simply builds a new castle of a specific size. We want to make sure that no shenanigans are going on and that when we build a castle of size 10, we get a castle of size 10:

```
function When_building_a_castle_size_should_be_correctly_set() {
  var castleBuilder = new CastleBuilder();
  var expectedSize = 10;

  var builtCastle = castleBuilder.buildCastle(10);

  assertEqual(expectedSize, builtCastle.size);
}
```

Asserts

You may have noticed that in the last example we made use of a function called `assertEqual`. An assert is a test that, when it fails, throws an exception. There is currently no built-in assert functionality in JavaScript, although there is a proposal in the works to add it.

Fortunately, building an assert is pretty simple:

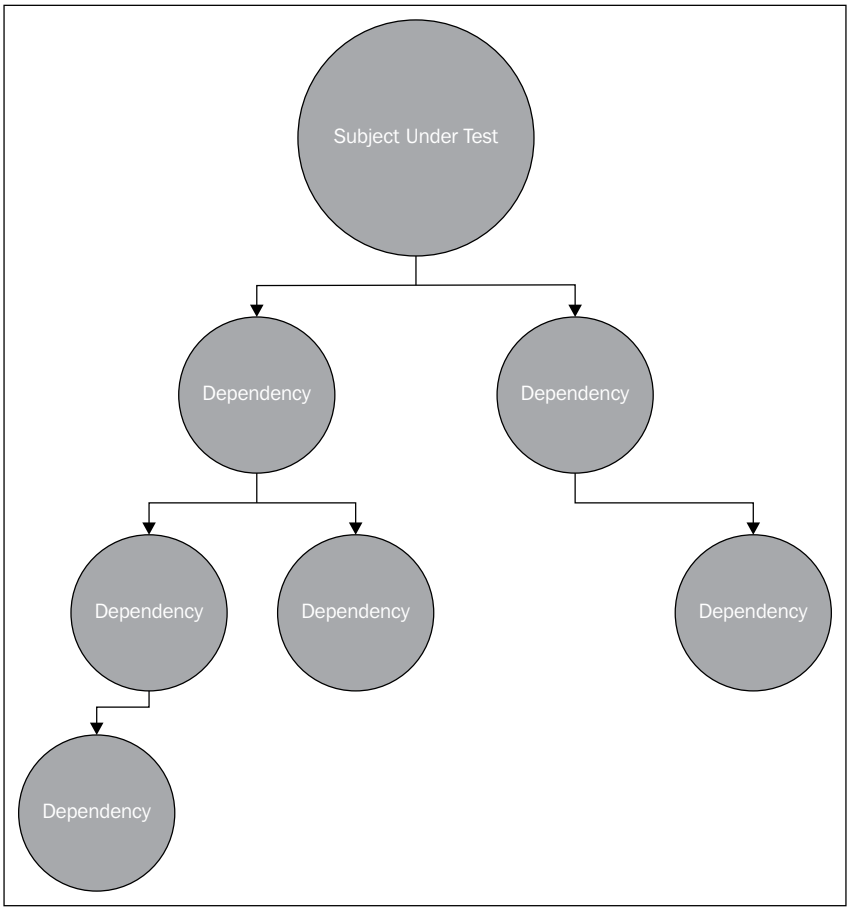
```
function assertEqual(expected, actual){
  if(expected !== actual)
    throw "Got " + actual + " but expected " + expected;
}
```

It is helpful to mention, in the error, the actual value as well as the expected value.

There are a great number of assertion libraries in existence. Node.js ships with one creatively called assert.js. If you end up using a testing framework for JavaScript, it is likely that it will also contain an assertion library.

Fake objects

If we think of the interdependencies between objects in an application as a graph, it becomes quickly apparent that there are a number of nodes that have dependencies on not just one but many other objects. Attempting to place an object with a lot of dependencies under a test is challenging. Each of the dependent objects must be constructed and included in the test. When these dependencies interact with external resources such as the network or filesystem, the problem becomes intractable. Pretty soon we're testing the entire system at a time. This is a legitimate testing strategy, known as **integration testing**, but we're really just interested in ensuring that the functionality of a single class is correct. Integration testing tends to be slower to execute than unit tests:



We need to find a way to isolate the class under test so that we don't have to recreate all the dependencies, including the network. We can think of this approach as adding bulkheads to our code. We will insert bulkheads to stop tests from flowing over from one class to many, as shown in the following figure:

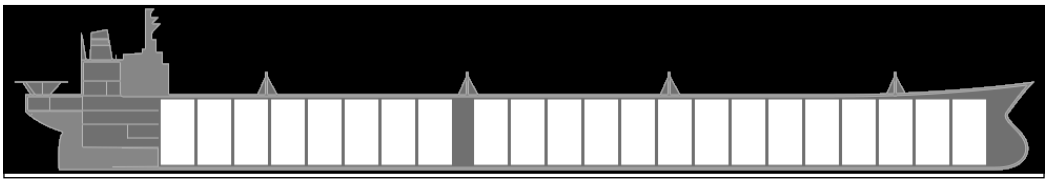


Image courtesy `http://www.reactivemanifesto.org/`

To this end, we can use fake objects that have a limited set of functionality in place of the real objects. We'll look at three different methods of creating fake objects.

The first is the rather niftily named **test spy**.

Test spies

A spy is an approach that wraps all the methods of an object and records the inputs and outputs from that method as well as the number of calls. By wrapping the calls, it is possible to examine exactly what was passed in and what came out of the function. Test spies can be used when the exact inputs into a function are not known beforehand.

In other languages, building test spies requires reflection and can be quite complicated. We can actually get away with making a basic test spy in no more than a couple of lines of code. Let's experiment.

To start, we'll need a class to intercept:

```
var SpyUpon = (function () {
  function SpyUpon() {
  }
  SpyUpon.prototype.write = function (toWrite) {
    console.log(toWrite);
    return 7;
  };
  return SpyUpon;
})();
```

Now, we would like to spy on this function. Because functions are first class objects in JavaScript, we can simply rejigger the `SpyUpon` object:

```
var spyUpon = new SpyUpon();
spyUpon._write = spyUpon.write;
spyUpon.write = function (arg1) {
  console.log("intercepted");
  this.called = true;
  this.args = arguments;
  this.result = this._write(arg1, arg2, arg3, arg4, arg5);
  return this.result;
};
```

Here we take the existing function and give it a new name. Then we create a new function that calls the renamed function and also records some things. After the function has been called, we can examine the various properties:

```
console.log(spyUpon.write("hello world"));
console.log(spyUpon.called);
console.log(spyUpon.args);
console.log(spyUpon.result);
```

Running the preceding code in Node.js gets us this:

```
hello world
7
true
{ '0': 'hello world' }
7
```

Using this technique, it is possible to get all sorts of insight into how a function is used. There are a number of libraries that support creating test spies in a more robust way than our simple version here. Some provide tools to record exceptions, the number of times the method was called, and the arguments for each call.

Stub

A stub is another example of a fake object. We can use stubs when we have some dependencies in the subject being tested that need to be satisfied with an object that returns a value. They can also be used to provide a bulkhead to stop computationally-expensive or I/O-reliant functions from being run.

Stubs can be implemented in much the same way that we implemented spies. We just need to intercept the call to the method and replace it with a version that we wrote. However, with stubs we actually don't call the replaced function. It can be useful to keep the replaced function around just in case we need to restore the functionality of the stubbed out class.

Let's start with an object that depends on another object for part of its functionality:

```
var Knight = (function () {
  function Knight(credentialFactory) {
    this.credentialFactory = credentialFactory;
  }
  Knight.prototype.presentCredentials = function (toRoyalty) {
    console.log("Presenting credentials to " + toRoyalty);
    toRoyalty.send(this.credentialFactory.Create());
    return true;
  };
  return Knight;
})();
```

This `Knight` object takes a `credentialFactory` parameter as part of its constructor. By passing in the object, we *decouple* the dependency and remove the responsibility for creating a `credentialFactory` parameter from the knight. We've seen this sort of inversion of control previously and we'll look at it in more detail in the next chapter. This makes our code more modular and it makes testing far easier.

Now when we want to test the knight without worrying about how a credential factory works, we can use a fake object, in this case, a stub:

```
var StubCredentialFactory = (function () {
  function StubCredentialFactory() {
  }
  StubCredentialFactory.prototype.Create = function () {
    //manually create a credential
  };
  return StubCredentialFactory;
})();
```

This stub is a very simple one that simply returns a standard new credential. Stubs can be made to be quite complicated if there needs to be multiple calls to it. For instance, we could rewrite our preceding simple stub as:

```
var StubCredentialFactory = (function () {
  function StubCredentialFactory() {
    this.callCounter = 0;
```

```
    }
    StubCredentialFactory.prototype.Create = function () {
      if (this.callCounter == 0)
        return new SimpleCredential();
      if (this.callCounter == 1)
        return new CredentialWithSeal();
      if (this.callCounter == 2)
        return null;
      this.callCounter++;
    };
    return StubCredentialFactory;
  })();
```

This version of the stub returns a different sort of credential every time it is called. On the third call, it returns `null`. As we set up the class using an inversion of control, writing a test is as simple as:

```
var knight = new Knight(new StubCredentialFactory());
knight.presentCredentials("Queen Cersei");
```

We can now execute the test:

```
var knight = new Knight(new StubCredentialFactory());
var credentials = knight.presentCredentials("Lord Snow");
assert(credentials.type === "SimpleCredentials");
credentials = knight.presentCredentials("Queen Cersei");
assert(credentials.type === "CredentialWithSeal");
credentials = knight.presentCredentials("Lord Stark");
assert(credentials == null);
```

Because there is no hard-typing system in JavaScript, we can build stubs without worrying about implementing interfaces. There is also no need to stub an entire object but only the function in which we're interested.

Mock

The final type of fake object is a mock. The difference between a mock and a stub is where the verification is done. With a stub, our test must check if the state is correct after the act. With a mock object, the responsibility to test the asserts falls to the mock itself. Mocks are another place where it is useful to leverage a mocking library. We can, however, build the same sort of thing simply ourselves:

```
var MockCredentialFactory = (function () {
  function MockCredentialFactory() {
```

```
    this.timesCalled = 0;
  }
  MockCredentialFactory.prototype.Create = function () {
    this.timesCalled++;
  };

  MockCredentialFactory.prototype.Verify = function () {
    assert(this.timesCalled == 3);
  };
  return MockCredentialFactory;
})();
```

This `mockCredentialsFactory` class takes on the responsibility of verifying whether the correct functions were called. This is a very simple sort of approach to mocking and can be done this way:

```
var credentialFactory = new MockCredentialFactory();
var knight = new Knight(credentialFactory);

var credentials = knight.presentCredentials("Lord Snow");
credentials = knight.presentCredentials("Queen Cersei");
credentials = knight.presentCredentials("Lord Stark");

credentialFactory.Verify();
```

This is a static mock that keeps the same behavior every time it is used. It is possible to build mocks that act as recording devices. You can instruct the mock object to expect certain behaviors and then have it automatically play them back.

The syntax for this, which is taken from the documentation for the mocking library Sinon, looks like this:

```
var mock = sinon.mock(myAPI);
mock.expects("method").once().throws();
```

Monkey patching

We've seen a number of methods of creating fake objects in JavaScript. When creating the spy, we made use of a method called **monkey patching**. Monkey patching allows you to dynamically change the behavior of an object by replacing its functions. We can use this sort of approach without having to revert to full fake objects. Any existing object can have its behavior changed in isolation using this approach. This includes built-in objects such as strings and arrays.

Interacting with the user interface

A great deal of JavaScript in use today is used on the client and is used to interact with elements that are visible on the screen. Interacting with the page flows through a model of the page known as the **Document Object Model** (**DOM**).

Every element on the page is represented in the DOM. Whenever a change is made to the page, the DOM is updated. If we add a paragraph to the page, then a paragraph is added to the DOM. Thus, if our JavaScript code is to add a paragraph, checking that it does so is simply a function of checking the DOM.

Unfortunately, this requires that a DOM actually exists and that it is formed in the same way that it is on the actual page. There are a number of approaches to doing testing against a page.

Browser testing

The most naïve approach is to simply automate the browser. There are a few projects out there that can help with this task. One can either automate a fully fledged browser such as Firefox, Internet Explorer, or Chrome, using a tool such as Selenium, or one can pick a browser that is headless such as Phantom.js. The fully fledged browser approach requires that a browser be installed on the test machine and that the machine be running in a mode that has a desktop available. This is not always the case with continuous integration build machines.

Many Unix-based build servers would not have been set up to show a desktop, as it wouldn't be needed for most build tasks. Even if your build machine is a Windows one, the build account frequently runs in a mode that has no ability to open a window. Tests using full browsers also have a tendency to break, in my mind. Subtle timing issues crop up and tests are easily interrupted by unexpected changes to the browser. It is a frequent occurrence that manual intervention will be required to unstick a browser that has ended up in an incorrect state.

Fortunately, efforts have been made to decouple the graphical portions of a web browser from the DOM and JavaScript. For Chrome, this initiative has resulted in PhantomJS, and for Firefox it has resulted in SlimerJS.

Typically, the sort of tests that require a full browser need some navigation of the browser across several pages. This is provided for in the headless browsers through an API. I would tend to think of tests at this scale as integration tests rather than unit tests.

A typical test using the PhantomJS and CasperJS libraries that sits on top of the browser might look like this:

```
var casper = require('casper').create();
casper.start('http://google.com', function() {
  assert.false($("#gbqfq").attr("aria-haspopup"));
  $("#gbqfq").val("redis");
  assert.true($("#gbqfq").attr("aria-haspopup"));
});
```

This would test that entering a value into the search box on Google would change the `aria-haspopup` property from `false` to `true`.

Testing things this way puts a great deal of reliance on the DOM not changing too radically. Depending on the selectors used to find elements on the page, a simple change to the style of the page could break every test. I like to keep tests of this sort away from the look of that page by never using CSS properties to select elements. Instead make use of IDs or, better yet, data-* attributes. We don't necessarily have the luxury of that when it comes to testing existing pages but it is certainly a good plan for new pages.

Faking the DOM

Much of the time, we don't need a full page DOM to perform our tests. The page's elements we need to test are part of a section on the page instead of the entire page. A number of initiatives exist that allow for the creation of a chunk of the document in pure JavaScript. The `jsdom` project, for instance, provides a method to inject a string of HTML and receive a fake window.

In this example, modified slightly from the `jsdom` README, they create some HTML elements, load JavaScript, and test that it returns correctly:

```
var jsdom = require("jsdom");
jsdom.env( '<p><a class="the-link"
ref="https://github.com/tmpvar/jsdom">jsdom!</a></p>',
["http://code.jquery.com/jquery.js"],
function (errors, window) {
  assert.equal(window.$("a.the-link").text(), "jsdom!");
}
);
```

If your JavaScript is focused on a small section of the page, perhaps you're building custom controls or web components, and then this is an ideal approach.

Wrapping the manipulation

The final approach to dealing with graphical JavaScript is to stop interacting directly with elements on the page. This is the approach that many of the popular JavaScript frameworks today use. One simply updates a JavaScript model and this model then updates the page through the use of some sort of MV* pattern. We looked at this approach in some detail in one of the previous chapters.

Testing in this case becomes quite easy. Our complicated JavaScript can simply be tested by building a model state prior to running the code and then testing to see if the model state after running the code is as we expect.

As an example, we could have a model that looks like this:

```
class PageModel{
  titleVisible: boolean;
  users: Array<User>;
}
```

And the test code for it might look as simple as this:

```
var model = new PageModel();
model.titleVisible = false;
var controller = new UserListPageController(model);

controller.AddUser(new User());

assert.true(model.titleVisible);
```

As everything on the page is manipulated through the bindings to the model, we can be confident that changes in the model are correctly updating the page.

Some would argue that we've simply shifted the problem. Now the only place for errors is if the binding between the HTML and the model is incorrect. So we also need to test if we have bindings correctly applied to the HTML. This falls to higher-level testing that can be done more simply. We can cover far more with a higher-level test than with a lower level one, although at the cost of knowing exactly where the error occurred.

You're never going to be able to test everything about an application, but the smaller you can make the untested surface the better.

Build and test tools

There are many build and testing tools available for JavaScript. It may seem to be odd that we're building an interpreted language, but we've seen previously that there are minification and combination steps. There are also linting tools that check JavaScript for common errors such as using `==` when `===` is the correct operator.

The JavaScript world is a very quick moving one in which tools are rapidly outdated and replaced with something newer. I'm hesitant to mention any tools in particular but there are a few that have proven to be resilient:

- **Grunt**: This is a configuration-based task runner for JavaScript. It has a rich plugin infrastructure and runs on Node.js.
- **Gulp**: This is a slightly newer task runner that replaced the configuration-based approach of Grunt with a streaming approach written in JavaScript.
- **Mocha, Jasmine, and QUnit**: These are JavaScript unit testing frameworks. Each one offers a slightly different approach and syntax. Which one is better is really a decision best left up to individual preference.

Hints and tips

I have seen tests where people split up the **arrange**, **act**, and **assert** by putting in comments:

```
function testMapping(){
  //Arrange
  …
  //Act
  …
  //Assert
  …
}
```

You're going to wear your fingers to the bone typing those comments for every single test. Instead, I just split them up with a blank line. The separation is clear and anybody who knows Arrange-Act-Assert will instantly recognize what it is that you're doing. You would have seen the example code in this chapter split up in this fashion.

When writing tests, I tend to name them in a way that makes it obvious that they are tests and not production code. For most JavaScript, I follow camel case naming conventions such as `testMapping`. However, for test methods, I follow an underscored naming pattern `When_building_a_castle_size_should_be_correctly_set`. In this way, the test reads more like a specification. Others have different approaches to naming and there is no *right* answer so feel free to experiment.

Summary

Producing a quality product is always going to require extensive and repeated testing, which is exactly the sort of thing computers are really good at. Automate as much as possible.

In this chapter, we examined the general form for a test using arrange, act, and assert. We also looked at how to create fake objects to take the place of the components of the system. Finally, we looked at ways to handle interacting with the user interface, a typical pain point in testing.

Testing JavaScript code is an up and coming thing. The tooling around mocking out objects and even the tools for running tests are undergoing constant changes. Being able to use tools such as Node.js to run tests quickly and without having to boot up an entire browser is stunningly helpful. This is an area that is only going to improve over the next few years. I am enthused to see what changes come from it.

In the next chapter, we'll take a look at some advanced patterns in JavaScript that you might not want to use every day but are very handy.

11
Advanced Patterns

I hesitated when naming this chapter *Advanced Patterns*. This isn't really about patterns that are more complicated or sophisticated than other patterns. It is about patterns that you wouldn't use very frequently. Frankly, coming from a static programming language background, some of them seem crazy. Nonetheless, they are completely valid patterns and are in use in big name projects everywhere.

In this chapter, we'll be looking at:

- Dependency injection
- Live postprocessing
- Aspect-oriented programming
- Macros

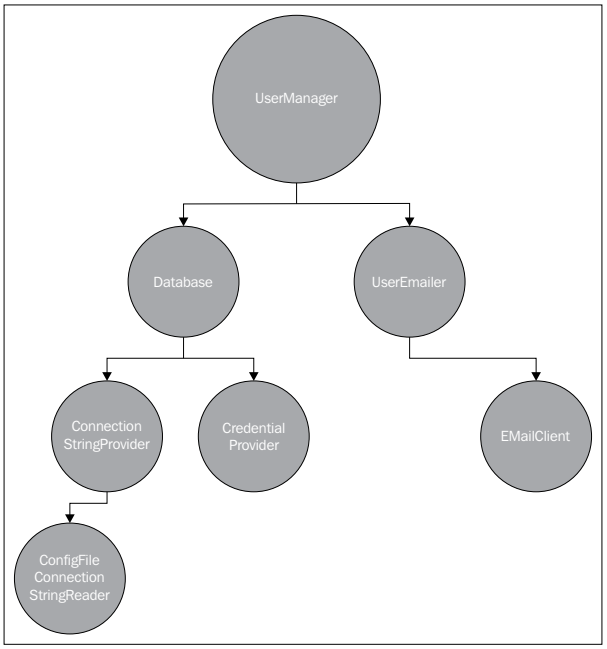
Dependency injection

One of the topics we've been talking about continuously in this book is the importance of making your code modular. Small classes are easier to test, provide for better reuse, and promote better collaboration between teams. Modular, loosely coupled code is easier to maintain, as changes can be limited. You may remember the example of a ripstop we used earlier in *Chapter 7*, *Model View Patterns*.

With modular code of this sort, we see a lot of inversion of control: classes have functionality inserted into them by passing in additional classes from their creators. The term inversion of control was popularized by Martin Fowler and Robert C. Martin (`http://martinfowler.com/bliki/InversionOfControl.html`). This moves the responsibility for how some portions of the child class work to the parent. For small projects, this is a pretty reasonable approach. As projects and dependency graphs get more complicated, manually injecting functionality becomes less tenable. We still have to new up objects and pass them into the client object–we've simply shifted the problem up a level.

If we think of object creation as a service, then a solution to this problem presents itself. We can defer the object creation to a central location. This allows us to change the implementations for a given interface in one place, simply and easily. It also allows us to control object lifetime so that we can reuse objects, or recreate them every time they are used. If we need to replace one implementation of an interface with another implementation, then we can be confident that we only need to change it in one location. Because the new implementation still fulfills the contract that is the interface, all the classes that make use of the interface can remain ignorant of the change.

What's more is that by centralizing object creation, it becomes easier to construct objects that depend on other objects. If we look at a dependency graph for a module such as the `UserManager`, it is clear that it has a number of dependencies. These dependencies may have additional dependencies and so forth. To build a `UserManager` module, we not only need to pass in `Database`, but also `ConnectionStringProvider`, `CredentialProvider`, and `ConfigFileConnectionStringReader`. Each of these classes is responsible for a small part of the login workflow and their relationship is shown in the following diagram. Goodness, that is going to be a lot of work to create instances of all of these. Instead, if we, register implementations of each of these interfaces in a registry, then we need only go to the registry to look up how to make them. This can be automated and the dependencies automatically injected to all dependencies without a need to explicitly create any of them. This method of solving dependencies is commonly referred to as solving the transitive closure.



A dependency injection framework handles the responsibility of constructing objects. On application setup, the dependency injection framework is primed with a combination of names and objects. From this, it creates a registry or a container. When constructing an object through the container, the container looks at the signature of the constructor and attempts to satisfy the arguments on the constructor.

In more statically-typed languages such as C# or Java, dependency injection frameworks are commonplace. They usually work by using reflection, a method of using code to extract structural information from other code. When building the container, one specifies an interface and one or more concrete classes that can satisfy the interface. Of course, it requires that the language support both interfaces and introspection.

There is no way to do this in JavaScript. JavaScript has neither direct introspection nor a traditional object inheritance model. A common approach is to use variable names to solve the dependency problem. Consider a class that has a constructor, as shown in the following code:

```
var UserManager = (function () {
  function UserManager(database, userEmailer) {
    this.database = database;
    this.userEmailer = userEmailer;
  }
  return UserManager;
})();
```

The constructor takes two arguments that are very specifically named. When we construct this class through the dependency injection, these two are satisfied by looking through the names registered with the container and passing them into the constructor. However, without introspection, how can we extract the names of the parameters so that we know what to pass into the constructor?

The solution is actually amazingly simple. The original text of any function in JavaScript is available by simply calling `toString` on it. So given the previous constructor, we can just use the following line:

```
UserManager.toString()
```

And now we can parse the string returned to extract the names of the parameters. Care must be taken to parse the text correctly, but it is possible. The popular JavaScript framework, Angular.js, actually uses this method to perform its dependency injection. The result remains relatively preformat. The parsing really only needs be done once and the results cached, so no additional penalty is incurred.

I won't go through how to actually implement the dependency injection, as it is rather tedious. When parsing the function, you can either parse it using a string-matching algorithm or build a lexer and parser for the JavaScript grammar. The first solution seems easier, but it is probably a better decision to build up a simple syntax tree for the code into which you're injecting. Fortunately, the entire method body can be treated as a single token, so it is vastly easier than building a fully fledged parser.

There are some limitations to this method when combined with minification as minification will change variable names to shorter versions. A number of workarounds exist, including a custom minifier and a different syntax for the method signature.

If you're willing to impose a different syntax on the user of your dependency injection framework, then you can even go so far as to create your own syntax. The Angular 2.0 dependency injection framework, di.js, supports a custom syntax to denote both places where objects should be injected and to denote the objects that satisfy some requirement.

Using it as a class into which some code needs to be injected looks like this code, taken from the di.js examples page:

```
@Inject(CoffeeMaker, Skillet, Stove, Fridge, Dishwasher)
export class Kitchen {
  constructor(coffeeMaker, skillet, stove, fridge, dishwasher) {
    this.coffeeMaker = coffeeMaker;
    this.skillet = skillet;
    this.stove = stove;
    this.fridge = fridge;
    this.dishwasher = dishwasher;
  }
}
```

The `CoffeeMaker` parameter might look like this:

```
@Provide(CoffeeMaker)
@Inject(Filter, Container)
export class BodumCoffeeMaker{
  constructor(filter, container){
    …
  }
}
```

You might also notice that this example makes use of the `class` keyword. This is because the project is very forward looking, and requires the use of Traceur.js to provide for ES6 class support. We'll learn about Traceur.js in the next chapter.

Live postprocessing

It should be apparent now that running `toString` over a function in JavaScript is a valid way to perform tasks. It seems odd, but really writing code that emits other code or meta-programming is as old as Lisp, possibly older. When I first came across how dependency injection works in Angular.js, I was both disgusted at the hack and impressed by the ingenuity of the solution.

If it is possible to do dependency injection by interpreting code on the fly, then what else could we do with it? The answer is: quite a lot. The first thing that comes to mind is that you could write domain-specific languages.

We talked about DSLs in *Chapter 5*, *Behavioral Patterns*, and even created a very simple one. With the ability to load and rewrite JavaScript, we can take advantage of a syntax that is close to JavaScript but not wholly compatible. When interpreting the DSL, our interpreter would write out additional tokens needed to convert the code to actual JavaScript.

One of the useful features of TypeScript that I've always liked, is that the parameter to the constructors that are marked as public are automatically transformed into properties on the object. For instance, the TypeScript code:

```
class Axe{
  constructor(public handleLength, public headHeight){}
}
```

Compiles to:

```
var Axe = (function () {
  function Axe(handleLength, headHeight) {
    this.handleLength = handleLength;
    this.headHeight = headHeight;
  }
  return Axe;
})();
```

We could do something similar in our DSL. Starting with the `Axe` class's definition:

```
class Axe{
  constructor(handleLength, /*public*/ headHeight){}
}
```

We've used a comment here to denote that `headHeight` should be `public`. Unlike the TypeScript version, we would like our source code to be valid JavaScript. Because comments are included in `toString`, this works just fine.

Next is to actually emit new JavaScript from this. I've taken a naïve approach and used regular expressions. This approach would quickly get out of hand and probably only works with the well-formed JavaScript in the `Axe` class:

```
function publicParameters(func){
  var stringRepresentation = func.toString();
  var parameterString = stringRepresentation.match(/^function
  .*\((.*)\)/)[1];
  var parameters = parameterString.split(",");
  var setterString = "";
  for(var i = 0; i < parameters.length; i++){
    if(parameters[i].indexOf("public") >= 0){
      var parameterName =
      parameters[i].split('/')[parameters[i].split('/').
      length-1].trim();
      setterString += "this." + parameterName + " = " +
      parameterName + ";\n";
    }
  }
  var functionParts = stringRepresentation.
  match(/(^.*{)([\s\S]*)/);
  return functionParts[1] + setterString + functionParts[2];
}

console.log(publicParameters(Axe));
```

In the preceding code, we extract the parameters to the function, and check for those that have the `public` annotation. The result of this function can be passed back into `eval` for use in the current object or written out to a file if we're using this function in a preprocessor. Typically, use of `eval` in JavaScript is discouraged.

There are tons of different things that can be done using this sort of processing. Even without string postprocessing, there are some interesting programming concepts we can explore by just wrapping methods.

Aspect-oriented programming

Modularity of software is a great feature; the majority of this book has been about modularity and its advantages. However, there are some features of software that span the entire system. Security is a great example of this.

We would like to have similar security code in all the modules of the application to check that people are, in fact, authorized to perform some action. So if we have a function of the sort:

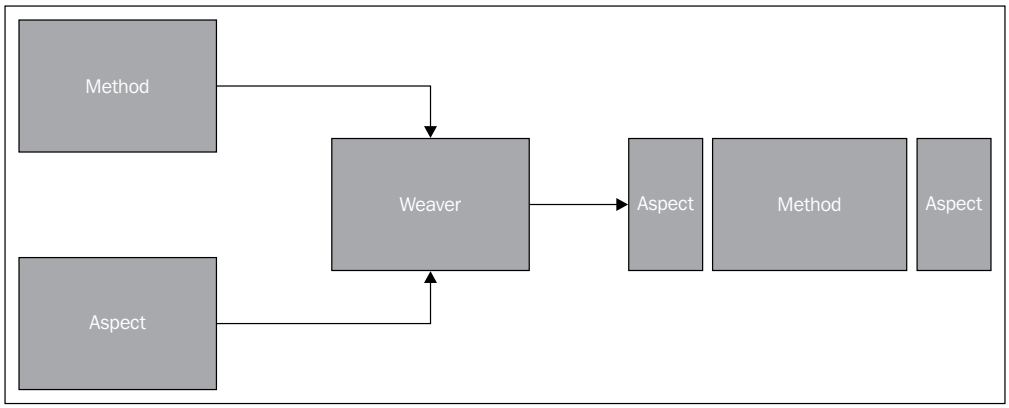
```
var GoldTransfer = (function () {
  function GoldTransfer() {
  }
  GoldTransfer.prototype.SendPaymentOfGold = function
  (amountOfGold, destination) {
    var user = Security.GetCurrentUser();
    if (Security.IsAuthorized(user, "SendPaymentOfGold")) {
      //send actual payment
    } else {
      return { success: 0, message: "Unauthorized" };
    }
  };
  return GoldTransfer;
})();
```

We can see that there is a fair bit of code in place to check if a user is authorized. This same boilerplate code is used elsewhere in the application. In fact, with this being a high security application, the security checks are in place in every public function. All is well until we have the need to make a change to the common security code. This change needs to take place in every single public function in the application. We can refactor our application all we want, but the truth remains that we need to have at least some code in each of the public methods to perform a security check. This is known as a **cross-cutting concern**.

There are other instances of cross-cutting concerns in most large applications. Logging is a great example, as is database access and performance instrumenting. **Aspect-oriented programming** (**AOP**) presents a way to minimize the repeated code through a process known as **weaving**.

An aspect is a piece of code that can intercept method calls and change them. On the .NET platform, there is a tool called PostSharp that does aspect weaving and on the Java platform, one called AspectJ. These tools hook into the build pipeline and modify the code after it has been transformed into instructions. This allows code to be injected wherever needed. The source code appears unchanged but the compiled output now includes calls to the aspect. Aspects solve the cross-cutting concern by being injected into existing code.

A common workflow for AOP is shown in the following diagram:



Of course, we don't have the luxury of a design-time compile step in most JavaScript workflows. Fortunately, we've already seen some approaches that would allow us to implement cross cuts using JavaScript. The first thing we need is the wrapping of methods that we saw in the testing chapter. The second is the string ability from earlier in this chapter.

There are some AOP libraries already in existence for JavaScript that may be a good bet to explore. You might be interested in:

- YouAreDaChef (`https://github.com/raganwald/YouAreDaChef`)
- AOP.js (`https://github.com/notejs/aop`)
- Meld (`https://github.com/cujojs/meld`)

However, we can implement a simple interceptor here. First, let's decide on a grammar for requesting injection. We'll use the same idea of comments from earlier to denote methods that require interception. We'll just make the first line in the method a comment that reads aspect (<name of aspect>).

To start, we'll take a slightly modified version of our same `GoldTransfer` class from the earlier example:

```
var GoldTransfer = (function () {
  function GoldTransfer() {
  }
  GoldTransfer.prototype.SendPaymentOfGold = function
  (amountOfGold, destination) {
    /* @aspect(Security)*/
```

```
    //send actual payment
    console.log("Payment sent");
  };
  return GoldTransfer;
})();
```

We've stripped out all the security stuff that used to exist in it and added a console log, so we can see that it actually works. Next we'll need an aspect to weave into it; this is quite similar to the decorator pattern:

```
var ToWeaveIn = (function () {
  function ToWeaveIn() {
  }
  ToWeaveIn.prototype.BeforeCall = function () {
    console.log("Before!");
  };
  ToWeaveIn.prototype.AfterCall = function () {
    console.log("After!");
  };
  return ToWeaveIn;
})();
```

For this, we use a simple class that has the `BeforeCall` and `AfterCall` methods, one which is called before and one which is called after the original method. We don't need to use `eval` in this case so the interceptions are safer:

```
function weave(toWeave, toWeaveIn, toWeaveInName) {
  for (var property in toWeave.prototype) {
    var stringRepresentation =
    toWeave.prototype[property].toString();

    console.log(stringRepresentation);
    if (stringRepresentation.indexOf("@aspect
    (" + toWeaveInName + ")") >= 0) {
      toWeave.prototype[property + "_wrapped"] =
      toWeave.prototype[property];
      toWeave.prototype[property] = function () {
        toWeaveIn.BeforeCall();
        toWeave.prototype[property + "_wrapped"]();
        toWeaveIn.AfterCall();
      };
    }
  }
}
```

This interceptor can easily be modified to shortcut and return something before the main method body is called. It can also be changed so that the output of the function is modified by simply tracking the output from the wrapped method, and then modifying it in the `AfterCall` method.

This is a fairly lightweight example of AOP. There are some frameworks in existence for JavaScript AOP, but perhaps the best approach is to make use of a precompiler or macro language.

Macros

A macro is a rule to transfer one piece of code into another, usually expanding the code and writing out new code. Preprocessing code through macros is not a new idea. It was, and probably still is, very popular for C and C++. In fact, if you take a look at some of the source code for the GNU utilities for Linux, they are written almost entirely in macros. Macros are notorious as they are hard to understand and debug. For a long time, newly created languages such as Java and C#, did not support macros for exactly this reason.

That being said, even more recent languages such as Rust and Julia have brought the idea of macros back. These languages were influenced by the macros from the Scheme language, a dialect of Lisp. The difference between C macros and Lisp/Scheme macros is that the C versions are textual, while the Lisp/Scheme ones are structural. This means that C macros are just glorified find replace tools, while Scheme macros are aware of the **abstract syntax tree** (**AST**) around them, allowing them to be much more powerful.

The AST for Scheme is a far simpler construct than that of JavaScript. Nonetheless, there is a very interesting project called Sweet.js (`http://sweetjs.org/`) that tries to create structural macros for JavaScript.

Sweet.js plugs into the JavaScript build pipeline and modified JavaScript source code using one or more macros. There are a number of fully fledged JavaScript transpilers, that is compilers that emit JavaScript. These compilers are problematic as they share code between multiple projects. The code is so different that there is no real way to share it. Sweet.js supports multiple macros being expanded in a single step. This allows for much better code sharing. The reusable bits are a smaller size and easier easy to run together.

A simple example of Sweet.js is:

```
let var = macro {
  rule { [$var (,) ...] = $obj:expr } => {
    var i = 0;
    var arr = $obj;
    $(var $var = arr[i++]) (;) ...
```

```
    }

    rule { $id } => {
      var $id
    }
  }
```

The macro here provides ES6 style deconstructors that split an array into three fields. The macro matches an array assignment and also a regular assignment. For regular assignment, the macro simply returns the identity, while for assignment of an array, it will explode the text and replace it.

For instance, if you run it over:

```
var [foo, bar, baz] = arr;
```

The result will be:

```
var i = 0;
var arr$2 = arr;
var foo = arr$2[i++];
var bar = arr$2[i++];
var baz = arr$2[i++];
```

This is just one example macro. The power of macros is really quite spectacular. Macros can create an entirely new language or change very minor things. They can be easily plugged in to fit any sided requirement.

Hints and tips

Using name-based dependency injection allows for conflicts between names. In order to avoid conflicts, it may be worth prefacing your injected arguments with some special character. For instance, Angular.js uses the `$` character to denote an injected term.

Several times in this chapter, I've mentioned the JavaScript build pipeline. It may seem odd that we have to build an interpreted language. However, there are certain optimizations and process improvements that may result from "building" JavaScript. There are a number of tools that can be used to help build JavaScript. Tools such as Grunt and Gulp are specifically designed to perform JavaScript and web tasks, but you can also make use of traditional build tools such as Rake, Ant, or even Make.

Summary

In this chapter, we covered a number of advanced JavaScript patterns. Of these patterns, it's my belief that dependency injection and macros are the most useful to us. You may not necessarily want to use them on every project. When approaching problems, simply being aware of the possible solutions may change your approach to them.

Throughout this book, I have talked extensively about the next versions of JavaScript. However, you don't need to wait until some future time to make use of many of these tools. There are ways today to compile newer versions of JavaScript up to the current version. The final chapter will explore a number of these tools and techniques.

12
ES6 Solutions Today

I cannot count the number of times I have mentioned upcoming versions of JavaScript in this book. It is somewhat frustrating that the language is not keeping pace with the requirements of application developers. Many of the approaches we've discussed become unnecessary with a newer version of JavaScript. Work is progressing on **ECMAScript 6** (**ES6**), but browser support is spotty at best (`http://caniuse.com/`). There are, however, some ways to get the next version of JavaScript working today.

In this chapter, we'll look at a couple of these, specifically:

- TypeScript
- Traceur

TypeScript

There is no shortage of languages that compile to JavaScript. CoffeeScript is perhaps the best known example of one of these languages, although the Google Web Toolkit that compiles Java to JavaScript was also once very popular. Never ones to be left behind or use somebody else's solution, Microsoft released a language called TypeScript in 2012. It is designed to be a superset of JavaScript in the same way that C++ is a superset of C. This means that all syntactically valid JavaScript code is also syntactically valid TypeScript code.

Microsoft itself is making heavy use of TypeScript in some of its larger web properties. Both Office 365 and Visual Studio Online have significant code bases written in TypeScript. These projects actually predate TypeScript by a significant margin. The transition from JavaScript to TypeScript was reportedly quite easy, due to the fact that it is a superset of JavaScript.

One of the design goals for TypeScript was to make it as compatible as possible with ECMAScript 6. This means that TypeScript supports some, although certainly not all, of the features of ECMAScript 6. The two main features it brings are support for the class syntax and arrow operators.

The class syntax

Way back in *Chapter 2*, *Organizing Code*, we explored the syntax to create classes in JavaScript, or at least a syntax to simulate classes. TypeScript will compile code down to a syntax not at all dissimilar to that from *Chapter 2*, *Organizing Code*.

If we wanted to recreate the `Castle` class's example we had in *Chapter 2*, *Organizing Code*, using TypeScript, we would only need to write this:

```
class Castle{
  constructor(public name){
  }
  public Build(){
    console.log("Castle built: " + this.name);
  }
}
```

Instead of the far more cumbersome pure JavaScript version:

```
var Castle = (function () {
  function Castle(name) {
    this.name = name;
  }
  Castle.prototype.Build = function () {
    console.log("Castle built: " + this.name);
  };
  return Castle;
})();
```

There is also support for class inheritance through the use of the `extends` keyword. If you remember, creating classes that extend other classes was something of an ordeal in Vanilla JavaScript. TypeScript makes it as easy as it would be in C# or Java:

```
class BaseStructure{
  constructor() {
    console.log("Structure built");
  }
}
```

```
class Castle extends BaseStructure{
  constructor(public name){
    super();
  }
  public Build(){
    console.log("Castle built: " + this.name);
  }
}
```

The preceding code will emit:

```
var __extends = this.__extends || function (d, b) {
  for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
  function __() { this.constructor = d; }
  __.prototype = b.prototype;
  d.prototype = new __();
};
var BaseStructure = (function () {
  function BaseStructure() {
    console.log("Structure built");
  }
  return BaseStructure;
})();
var Castle = (function (_super) {
  __extends(Castle, _super);
  function Castle(name) {
    _super.call(this);
    this.name = name;
  }
  Castle.prototype.Build = function () {
    console.log("Castle built: " + this.name);
  };
  return Castle;
})(BaseStructure);
```

Which one of these codes would you rather maintain?

The module syntax

In addition to classes, modules are also supported by TypeScript. They are as simple as using the `module` keyword. If we take the preceding class and wrap it in module{} like the following code:

```
module Westeros.Buildings{
  classBaseStructure{
      //…
```

```
    }
  }
  class Castle extends BaseStructure{
    //…
  }
}
```

Then, TypeScript will generate the following code:

```
var Westeros;
  (function (Westeros) {
    (function (Buildings) {
      var BaseStructure = (function () {
         //…
      }
      Buildings.BaseStructure = BaseStructure;
      var Castle = (function (_super) {
        //…
      })(BaseStructure);
      Buildings.Castle = Castle;
    })(Westeros.Buildings || (Westeros.Buildings = {}));
    var Buildings = Westeros.Buildings;
  })(Westeros || (Westeros = {}));
```

Arrow functions

A few years ago, Microsoft introduced lambdas, which are first class functions, into the C# language. A first class function is one that can be assigned to a variable, passed into and out of other functions, and created at runtime. Since that time, there has been something of a renaissance of languages supporting lambdas as they are so amazingly useful. The last version of Java even adds support for them.

In C# and Java, these lambdas are commonly passed into other methods as a form of inversion of control. A popular approach is to use them to operate on members of a collection. C# even defines a series of operations in a library called **Language Integrated Queries** (**LINQ**) that provides very powerful operators on top of enumerable collections.

The generally agreed upon syntax is to use `=>` to denote a lambda. The following code is an example from C#:

```
collection.ForEach((item) => {
  var testResult = item.test();
  if(testResult.State == FAILURE)
  sendErrorEmail(testResult);
});
```

By default, lambdas specified without curly braces return the statement value. These can be readily used for projections. Again in C#:

```
collection.Select(x=>x.State);
```

This will return a collection of states selected from the collection.

ECMAScript 6 has support to specify functions using the arrow lambda syntax. TypeScript also supports lambdas and can compile them backwards to earlier versions of JavaScript.

Arrow functions are not just a short form of the standard function definition. They support lexical bind of `this`. Lexical scoping means that the binding of `this` is dependent on where the lambda is defined and not where it is used. In my experience, the scoping of `this` is one of the most common sources of errors and confusion in JavaScript. This is especially true when using jQuery, which I found takes a rather liberal hand in redefining `this` for its own purposes.

If we look at a typical jQuery click handler, we can see how the `function` keyword and arrow functions differ. In TypeScript:

```
class LoginPage{
  constructor(container: JQuery) {
    container.on("click", ".login-link", (item) => {
      this.login();
    });
    container.on("click", ".login-link", function(item) {
      this.login();
    });
  }

  login(){
    console.log("logged in");
  }
}
```

The preceding code will emit:

```
var LoginPage = (function () {
  function LoginPage(container) {
    var _this = this;
    container.on("click", ".login-link", function (item) {
      _this.login();
    });
```

```
    container.on("click", ".login-link", function (item) {
      this.login();
    });
  }
  LoginPage.prototype.login = function () {
    console.log("logged in");
  };
  return LoginPage;
})();
```

Notice that the first function makes use of a variable called `_this` which is defined outside the function. This means that the arrow function will correctly call the login method on the class, while the normal function syntax will attempt to call login on whatever jQuery has redefined `this` to be. In this case, `this` is going to be bound to the HTML element that was clicked.

Arrow functions also disallow reassigning this inside them, so you can be assured that you'll always get the correct version of the variable.

I mentioned the LINQ library of C# earlier. There are similar libraries in JavaScript that provide for functional programming over collections. We've looked a bit at using function passing on collection for filtering and piping in *Chapter 6*, *Functional Programming*. In that chapter, we used the following example:

```
var items = [1,2,3,4,5,6,7,8,9,10];
items.where(function(thing){ return thing % 2 ==0;});
```

Using arrow functions, we get a cleaner bit of code that looks more like this:

```
var items = [1,2,3,4,5,6,7,8,9,10];
items.where((thing) =>thing % 2 ==0);
```

Arrow functions are not just syntactically nicer but also work more like most developers expect functions to work. I've seen many developers suggesting that arrow functions can almost entirely replace the `function` key word, and that in doing so confusion and bugs will be reduced. In my code, I still find myself using a mixture but I'll admit that I use arrow functions most of the time.

Typing

Along with the ES6 features we've mentioned earlier, TypeScript has a rather intriguing typing system incorporated into it. One of the nicest parts of JavaScript is that it is a dynamically typed language. We've seen, repeatedly, how not being burdened by types has saved us time and code. The typing system in TypeScript allows you to use as much or as little typing as you deem to be necessary.

You can give variables a type by declaring them with the following syntax:

```
var a_number: number;
var a_string: string;
var an_html_element: HTMLElement;
```

Once a variable has a type assigned to it, the TypeScript compiler will use that, not only to check that variable's usage but also to infer what other types may be derived from that class. For example, in the following code:

```
var numbers: Array<number> = [];
numbers.push(7);
numbers.push(9);
var unknown = numbers.pop();
```

The TypeScript compiler will know that unknown is a number. If you attempt to use it as something else, say a string:

```
console.log(unknown.substr(0, 1));
```

Then the compiler will throw an error. However, you don't need to assign a type to any variable. This means that you can tune the degree to which the type checking is run. While it sounds odd, it is actually a brilliant solution to introduce the rigor of type checking without losing the pliability of JavaScript. The typing is only enforced during compilation; once the code is compiled to JavaScript, any hint that there was typing information associated with a field disappears. As a result, the emitted JavaScript is actually very clean.

If you're interested in typing systems and know words such as contravariant and can discuss the various levels of gradual typing, then TypeScript's typing system may be well worth your time to investigate.

All the examples in this book were originally written in TypeScript and then compiled to JavaScript. This was done to improve the accuracy of the code and generally to save me from messing up quite so frequently. I'm horribly biased, but I think that TypeScript is really well done and certainly better than writing pure JavaScript.

There is no support to type in future versions of JavaScript. Thus, even with all the changes coming to future versions of JavaScript, I still believe that TypeScript has its place in providing compile time type checking. I never cease to be amazed by the number of times that the type checker has saved me from making silly mistakes when writing TypeScript.

Traceur

An alternative to TypeScript is to use the Traceur compiler. This is a project sponsored by Google to compile ES6 to equivalent ES5 JavaScript. A lot of the changes put in place for ES6 are syntactic niceties so they can actually be represented in ES5 JavaScript, although not as succinctly or as pleasantly. We've seen that already with using class-like structures in ES5. Traceur is written in JavaScript which means that the compilation from ES6 to ES5 is possible directly on a web page. Of course, as seems to be the trend with compilers, the source code for Traceur makes use of ES6 constructs, so Traceur must be used to compile Traceur.

At the time of writing, the list of ES6 functions that are supported by Traceur is extensive:

- Arrow functions
- Classes
- Computed property names
- Default parameters
- Destructuring assignment
- Iterators and `for of`
- Generator comprehension
- Generators
- Modules
- Numeric literals
- Property method assignment
- Object initializer shorthand
- Rest parameters
- Spread
- Template literals
- Promises

There is already a fair bit of documentation available on how each of these features works, so we won't go over all of them.

Setting up Traceur is a fairly simple exercise if you already have Node.js and npm installed:

```
npm install -g traceur
```

This will create a `traceur` binary which can do compilation with the following command:

```
traceur --script input.js --out output.js
```

Classes

By now, you should be getting sick of reading about different ways to make classes in JavaScript. Unfortunately for you, I'm the one writing this book, so let's look at one final example. We'll use the same castle example from earlier.

Modules within files are not supported in Traceur. Instead files are treated as modules, which allow for dynamic loading of modules in the fashion of Require.js. Thus, we'll drop the module definition from our castle and stick to just the classes. One other feature that exists in TypeScript and not ES6, is the prefacing a parameter with public to make it a public property on a class.

Once we've made these changes, the source ES6 file looks like this:

```
class BaseStructure {
  constructor() {
    console.log("Structure built");
  }
}

class Castle extends BaseStructure {
  constructor(name){
    this.name = name;
    super();
  }
  Build(){
    console.log("Castle built: " + this.name);
  }
}
```

The resulting ES5 JavaScript looks like this:

```
var BaseStructure = function BaseStructure() {
  "use strict";
  console.log("Structure built");
};
($traceurRuntime.createClass)(BaseStructure, {}, {});
var Castle = function Castle(name) {
  "use strict";
```

```
    this.name = name;
    $traceurRuntime.superCall(this, $Castle.prototype,
    "constructor", []);
  };
  var $Castle = Castle;
  ($traceurRuntime.createClass)(Castle, {Build: function() {
      "use strict";
      console.log("Castle built: " + this.name);
    }}, {}, BaseStructure);
```

Right away, it is apparent that the code produced by Traceur is not as clean as the code from TypeScript. You may also have noticed that there is a call to `$traceurRuntime.createClass`; this function is defined in a Traceur runtime file that must be included before making use of the Traceur compiled code. There are also a number of mentions of `"use strict";`. This is an instruction to the JavaScript engine that it should run in strict mode.

Strict mode prevents a number of dangerous JavaScript practices. For instance, in some JavaScript interpretors, it is legal to use a variable without declaring it first:

```
x = 22;
```

This will throw an error if `x` has not previously been declared:

```
var x = 22;
```

Duplicating properties in objects is disallowed, as well as double declaring a parameter. There are a number of other practices that `use strict` will treat as errors. I would like to think of `use strict` as being similar to treating all warnings as errors. It isn't, perhaps, as complete as `-Werror` in GCC, but it is still a good idea to use strict mode on new JavaScript code bases. Traceur simply enforces that for you, which is a great idea as strict mode avoids all sorts of errors.

Default parameters

Not a huge feature but a real nicety in ES6 is the introduction of default parameters. It has always been possible to call a function in JavaScript without specifying all the parameters. Parameters are simply populated from left to right, until there are no more values and all remaining parameters are given `undefined`.

The default parameters allow setting a value other than undefined for parameters that aren't filled out:

```
function CreateFeast(meat, drink = "wine"){
  console.log("The meat is: " + meat);
```

```
    console.log("The drink is: " + drink);
  }
  CreateFeast("Boar", "Beer");
  CreateFeast("Venison");
```

The preceding code will output:

```
The meat is: Boar
The drink is: Beer
The meat is: Venison
The drink is: wine
```

The JavaScript code produced is actually very simple:

```
function CreateFeast(meat) {
  var drink = arguments[1] !== (void 0) ? arguments[1] : "wine";
  console.log("The meat is: " + meat);
  console.log("The drink is: " + drink);
}
```

The only confusing part is `(void 0)`. This is a method to get the undefined value we need to check against. Weirdly, undefined is actually not a reserved word in JavaScript, so in some environments it is actually possible to assign a value to undefined.

Template literals

On the surface, template literals seem to be a solution for the lack of string interpolation in JavaScript. In some languages, such as Ruby and Python, you can inject substitutions from the surrounding code directly into a string, without having to pass them into some sort of string formatting function. For instance, in Ruby, you can use the following code:

```
name= "Stannis";
print "The one true king is ${name}"
```

This will bind the `${name}` value to the name from the surrounding scope.

ES6 supports template literals that allow something similar in JavaScript:

```
var name = "Stannis";
console.log('The one true king is ${name}');
```

It may be difficult to see, but that string is actually surrounded by backticks and not quotation marks. Tokens to bind to the scope are denoted by `${}`. Within the braces, you can put complex expressions such as:

```
var army1Size = 5000;
var army2Size = 3578;
console.log(`The surviving army will be ${army1Size > army2Size ?
"Army 1": "Army 2"}`);
```

The Traceur compiled version of the preceding code simply substitutes appending strings for string interpolation:

```
var army1Size = 5000;
var army2Size = 3578;
console.log(("The surviving army will be " + (army1Size >
army2Size ? "Army 1" : "Army 2")));
```

Template literals also solve a number of other problems. New line characters inside of a template literal are legal, meaning that you can use template literals to create multiline strings.

With the multiline string idea in mind, it seems like template literals might be useful to build domain-specific languages: a topic we've seen a number of times already. The DSL can be embedded in a template literal and then values from outside plugged in. An example might be using it to hold HTML strings (certainly a DSL), and inserting values in from a model. These could, perhaps, take the place of some of the template tools in use today. A template string might look like:

```
var template = '<span>${name}</span>
  <span>${address}</span>
  <span>${city}</span>
  <span>${postalCode}</span>'
```

Block bindings with let

The scoping of variables in JavaScript is weird. If you define a variable inside a block, say inside an `if` statement, then that variable is still available outside of the block. For example, the following code:

```
if(true)
{
  var outside = 9;
}
console.log(outside);
```

This code will print `9`, even though the variable outside is clearly out of scope. At least, it is out of scope if you assume that JavaScript is like other C-syntax languages and supports block-level scoping. The scoping in JavaScript is actually function-level scope. The variables declared in code blocks like those found attached to `if` and loop statements are hoisted to the beginning of the function. This means that they remain in scope for the entirety of the function.

ES6 introduces a new keyword, `let`, which scopes variables to the block level. This sort of variable is ideal for use in loops or to maintain proper variable values inside an `if` statement. Traceur implements support for the `for` block scoped variables. However, the support is experimental at the moment due to performance implications.

The following code:

```
if(true)
{
  var outside = 9;
  let inside = 7;
}
console.log(outside);
console.log(inside);
```

Will compile to:

```
var inside$__0;
if (true) {
  var outside = 9;
  inside$__0 = 7;
}
console.log(outside);
console.log(inside);
```

You can see that the inner variable is replaced with a renamed one. Once outside the block, the variable is no longer replaced. Running the preceding code will report that `inside$__0` is undefined when the `console.log` occurs. The variable does exist and is still accessible, so it is really just a fake local variable.

Async

Traceur actually implements some constructs that are part of ECMAScript 7 or Harmony. An example of this is the `async/await` structure. As far as I know, this method of simplifying asynchronous code was first introduced in C#. It helps eliminate some of the complexity around dealing with callbacks.

Traceur provides an implementation of promises. If you're from the Java or C# communities, you might think of a promise as a task. It is a construct that records a result as well as a state. When you first create it, you pass in a function to run, and, once the function is complete, the promise will populate its state as well as the result. You can keep checking the state which will read as unresolved until the given function is complete. Then the state will change to complete.

The `await` keyword is a short form, to wait for the promise to be resolved and unwrapping the result. Using `await` for a promise obscures the fact that there even is a promise by just returning the promise result. The `async` keyword is used to denote a method that will return a promise that must be resolved into an actual value.

An example based on the one given in the Traceur documentation is:

```
function timeout(ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}

async function asyncValue(value) {
  await timeout(1500);
  return value;
}

(async function() {
  console.log("Starting.");
  var valuePromise =
  asyncValue(42).catch(console.error.bind(console));
  console.log("Task is running in the background.");
  console.log("Awaiting the promise");
  var value = await valuePromise;
  console.log("Promise resolved");
  assert.equal(42, value);
  console.log(value);
  done();
})();
```

Here, we start an asynchronous process and get a promise from it. We keep that promise around, until we actually need the value two lines later when we call `await`. The execution will then block until the promise has been resolved, at which point execution will continue.

The compiled ES5 code for this short piece of ES6 code is quite extensive. Not only is the code more than fifty lines long, it makes heavy use of the Traceur runtime. The code involves creating a state machine that handles the state of the promise. The complexity of the state machine grows rapidly as the number of promises increase. Although you could create this by hand, it would be exceedingly painful and error prone.

The `async` and `await` features provide a very useful tool to help avoid callback hell. So long as you have no need to produce easily understandable JavaScript, then using Traceur to produce the state machine is a substantial time saving.

Conclusion

Traceur is a very powerful tool to replicate many of the structures and features of the next version of JavaScript today. However, the code generated is never going to be quite as efficient as having native support for the constructs. It may be worth benchmarking the generated code to ensure that it continues to meet the performance requirements of your project.

The next version of AngularJS is being built using Traceur for maximum compatibility with upcoming versions of JavaScript. Traceur itself is also implemented in Traceur-based ES6, so there are significant code bases in existence to prove its viability. I have great faith in the team working on AngularJS and would say that using Traceur is an entirely valid approach to create forward looking code. Eventually, you'll be able to simply remove the Traceur compile step and deploy the ES6 code directly.

Hints and tips

There are two excellent libraries to work with collections in a functional fashion in JavaScript: Underscore.js and Lo-Dash. Used in combination with TypeScript or Traceur, they have a very pleasant syntax and provide immense power.

For instance, finding all the members of a collection that satisfy a condition using Underscore.js looks like this:

```
_.filter(collection, (item) =>item.Id> 3);
```

This code will find all the items where the ID is greater than 3.

One of these libraries is one of the first things I add to a new project. Underscore.js is actually bundled with Backbone.js, an model-view framework.

Tasks for Grunt and Gulp exist to compile code written in TypeScript or Traceur. There is, of course, also good support for TypeScript in much of Microsoft's development tool chain, although Traceur is currently not supported directly.

Summary

As the functionality of JavaScript expands, the need for third-party frameworks and even transpilers starts to drop off. The language itself replaces many of these tools. The end game for tools such as jQuery is that they are no longer required as they have been absorbed into the ecosystem. For many years, the velocity of web browsers has been unable to keep pace with the rate of change of people's desires.

There is a large effort behind the next version of AngularJS, but great efforts are being made to align the new components with the upcoming web component standards. Web components won't fully replace AngularJS, but AngularJS will end up simply enhancing web components.

Of course, the idea that there won't be a need for any frameworks or tools is ridiculous. There is always going to be a new method of solving a problem, and new libraries and frameworks will show up. People's opinions on how to solve problems is also going to differ. That's why there is space in the market for the wide variety of MVVM frameworks that exist.

Working with JavaScript can be a much more pleasant experience if you make use of ES6 constructs. There are a couple of possible approaches to doing so; the best one for your specific problem is a matter for closer investigation. We looked at TypeScript, a Microsoft approach that brings some ES6 constructs along with an interesting type system. We also looked at Traceur, a project designed to bring as much of ES6 to current JavaScript as possible.

Conclusion

My wife tells me that all really good conclusions start with "so in conclusion". If she's right (which she frequently claims to be), I've already relegated this conclusion to junk status by failing to start it correctly. It is actually a great relief to have the yoke of greatness lifted from my shoulders.

The truth is that in 95 percent of software, getting things right the first time doesn't matter. Unless you're sending software into space or to the bottom of the ocean, then what really matters is how quickly you can adapt to change. Even software deployed to gaming consoles can now be patched through the use of their online services. Although slightly irritating to the end consumer, it is probably not as irritating as a game being delayed while more testing is performed.

The changes to software can stem from an actual change to a business requirement, or just that you didn't get the requirement right in the first place.

This is not to say that there is no reason to even attempt to get things right in the first place, but simply to say that it isn't unusual to get things wrong. Specifying software is a notoriously difficult problem. The best solution is to be adaptable and open to change.

The lean startup mentality that has taken root in the last few years advocates shipping as quickly as possible, by shipping the **minimum viable product** (**MVP**). If the shipped product takes off, then more can be invested to add features to the product and improve it. If the idea doesn't take off, then the amount of money invested in the failed idea is minimized. With cloud computing and a proliferation of software services that can handle time-consuming tasks such as authentication, the cost to take a product to the market is far lower now that it has been at any point in the past.

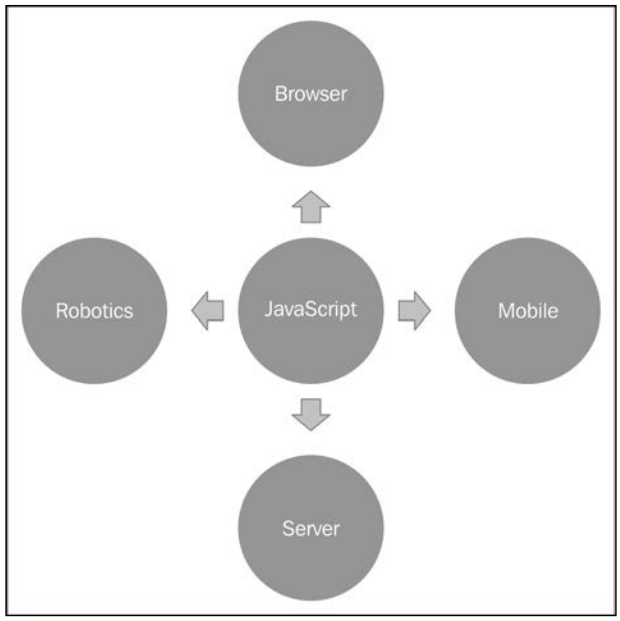
I frequently see software projects where a change in requirement as simple as adding a new field to a form, takes months. This sort of sloth-like pace is becoming increasingly unacceptable even in large corporate environments. Software is a differentiator; it lets you do your business more easily than your competitors. Being more efficient means you can do the same work at a lower cost and undercut everybody else.

I have worked on a number of projects that have a significant JavaScript code base. Almost all of these are older projects that have failed to follow patterns. JavaScript is stuffed into a directory and included in the most ad hoc way possible. I was scared to alter existing JavaScript and ended up replacing most of it. Had some proper patterns been followed, the code would have been much more maintainable and testable. I can assure you that the replacement code was more pattern rich.

Many bemoan the rapid release of small applications without required features, and insufficiently debugged as the death of quality. To a certain extent, this is true. I like to think of it more as the victory of the 80:20 rule: if you can get 80 percent of the result with 20 percent of the effort, then is the remaining 20 percent of the result really worth it?

Design patterns encourage building applications that are loosely coupled and more easily adaptable to change. By designing software using well-known and well-explored patterns, rapid iteration becomes easier.

The patterns presented in this book aim to keep you from spending your time rediscovering what those before have already invested sweat and tears discovering. As JavaScript moves into roles other than a pure browser language, the patterns previously in that space become applicable to JavaScript. JavaScript is used everywhere, as shown in the following diagram:



The lessons in this book may not be well known in JavaScript circles but they are well explored in other programming environments. As JavaScript expands in the problems to which it is applied, the lessons from other languages become more and more important.

As you work through developing applications in JavaScript, keep in mind the patterns that may be of assistance to you. Try to avoid setting out to use any particular pattern and instead apply them as the problem warrants. Most of all get out there, write interesting applications, and enjoy living through the JavaScript revolution. These are going to be an exciting few years.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Thank you for buying
Mastering JavaScript Design Patterns

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

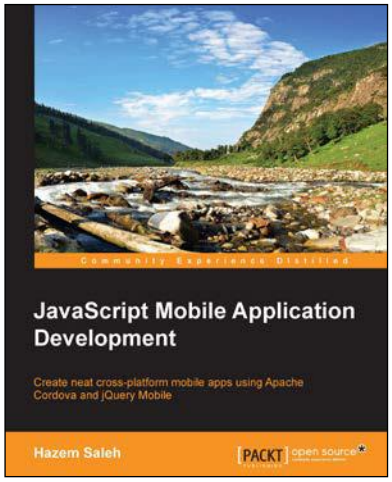
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING



JavaScript Mobile Application Development

ISBN: 978-1-78355-417-1 Paperback: 332 pages

Create neat cross-platform mobile apps using Apache Cordova and jQuery Mobile

1. Configure your Android, iOS, and Windows Phone 8 development environments.
2. Extend the power of Apache Cordova by creating your own Apache Cordova cross-platform mobile plugins.
3. Enhance the quality and the robustness of your Apache Cordova mobile application by unit testing its logic using Jasmine.

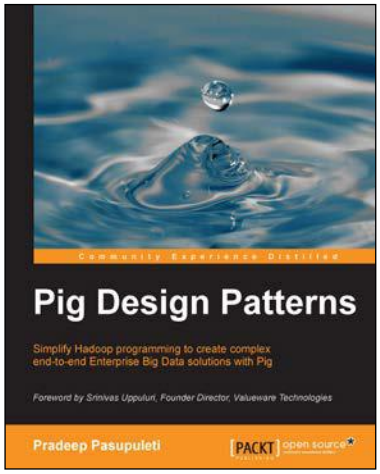
Laravel Design Patterns and Best Practices

ISBN: 978-1-78328-798-7 Paperback: 106 pages

Enhance the quality of your web applications by efficiently implementing design patterns in Laravel

1. Create fully functional web applications using design patterns in Laravel.
2. Explore various techniques to adapt different software patterns that suit your needs.
3. Get to know the best practices to utilize when making a web application.

[PACKT] open source
community experience distilled
PUBLISHING

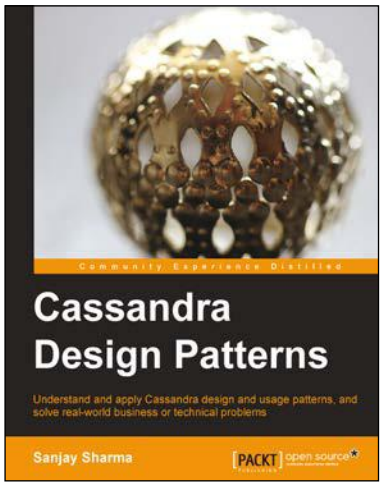


Pig Design Patterns

ISBN: 978-1-78328-555-6 Paperback: 310 pages

Simplify Hadoop programming to create complex end-to-end Enterprise Big Data solutions with Pig

1. Quickly understand how to use Pig to design end-to-end Big Data systems.
2. Implement a hands-on programming approach using design patterns to solve commonly occurring enterprise Big Data challenges.
3. Enhances users' capabilities to utilize Pig and create their own design patterns wherever applicable.



Cassandra Design Patterns

ISBN: 978-1-78328-880-9 Paperback: 88 pages

Understand and apply Cassandra design and usage patterns, and solve real-world business or technical problems

1. Learn how to identify real-world use cases that Cassandra solves easily, in order to use it effectively.
2. Identify and apply usage and design patterns to solve specific business and technical problems including technologies that work in tandem with Cassandra.
3. A hands-on guide that will show you the strengths of the technology and help you apply Cassandra design patterns to data models.