Code Responsibly

JS

99 Bottles of OOP

A Practical Guide to Object-Oriented Design

Second Edition

Sandi Metz, Katrina Owen & TJ Stankus

99 Bottles of OOP

Metz, Sandi;Owen, Katrina;Stankus, TJ

Table of Contents

Colophon

Version: 2.0.0

Version Date: 20-07-12

Published By: Potato Canyon Software, LLC

2nd Edition

Cover Design and Art by Lindsey Morris.

Edited by Julia Trimmer.

Created using Asciidoctor.

JavaScript logo: License, MIT

Your Rights As A Reader

Thank you for buying *99 Bottles of OOP*. As the authors of a self-published book, we very much appreciate your purchase.

This book is chock-full of lessons, and readers often write asking if they can share them with others. We commend your desire to pass on what you've learned, and ask only that while doing so you respect our rights are authors. This means that while we encourage you to spread the underlying ideas of the book, we restrict your use of its actual content (the specific examples, explanations, and descriptions).

Our bargain with you is as follows:

1. Your purchase entitles you to a single, non-transferable license for your personal use of the ebook related files. You may read and download the ebook you purchased to your personal devices only.

   You may not:

   - Sell the book
   - Give it away
   - Distribute it in any way
   - Print it (except for your personal use)

2. You may use *99 Bottles of OOP* as curriculum in a public education setting (university, code school, secondary school) as long as every student buys or is provided with a legal copy of the book.

   Volume discounts are available, and there's a free-book-for-a-postcard program. Contact human@99bottlesbook.com for information about bulk purchases.

3. You may share one small section (a chapter or less) at a free, public meet-up as long as the material is properly attributed.

4. You may not teach a course based on the entire book, even if this course is free and open to the public.

5. You may not use any part of *99 Bottles of OOP* in any endeavor in which you charge for your services.

Dedication

Sandi

To Amy, for everything she is and does, and to Jasper, who taught me that nothing trumps a good walk.

Katrina

To Sander, whose persistence is out of this world.

TJ

To those who have encouraged me, especially my parents and teachers. And to Graylyn, who I try to encourage most.

Preface

It turns out that everything you need to know about Object-Oriented Design (OOD) can be learned from the "99 Bottles of Beer" song.

Well, perhaps not *everything*, but quite certainly a great *many* things.

The song is simultaneously easy to understand and full of hidden complexity, which makes it the perfect skeleton upon which to hang lessons in OOD. The lessons embedded within the song are so useful, and so broad, that over the last three years it has become a core part of the curriculum of Sandi Metz's Practical Object-Oriented Design course.

The thoughts in this book reflect countless hours of discussion and collaboration between Sandi, Katrina Owen, and TJ Stankus. These ideas have been battle-tested by hundreds of students, and refined by a series of deeply thoughtful co-instructors, beginning with Katrina. While none of the authors have the hubris to claim perfect understanding, all have learned a great deal about Object-Oriented Design from teaching this song, and feel compelled to write it all down.

Therefore, this book, now in its second edition. We hope that you find it both useful and enjoyable.

What This Book Is About

This book is about writing cost-effective, maintainable, and pleasing code.

Chapter 1 explores how to decide if code is "good enough." This chapter uses metrics to compare several possible solutions to the 99 Bottles problem. It introduces a type of solution known as *Shameless Green*, and argues that although Shameless Green is neither clever nor changeable, it is the best *initial* solution to many problems.

Chapter 2 is a primer for Test-Driven Development (TDD), which is used to find Shameless Green. This chapter is concerned with deciding what to test, and with creating tests that happily tolerate changes to the underlying code.

Chapter 3 introduces a new requirement (six-pack), which leads to a discussion of how to decide where to start when changing code. This chapter examines the Open/Closed Principle, and then explores code smells. The chapter then defines a simple set of *Flocking Rules*, which guide a step-by-step refactoring of code.

Chapter 4 continues the step-by-step refactoring begun in Chapter 3. It iteratively applies the Flocking Rules, eventually stumbles across the need for the Liskov Substitution Principle, and ultimately unearths a deeply hidden abstraction.

Chapter 5 inventories the existing code for smells, chooses the most prominent one, and uses it to trigger the creation of a new class. Along the way, it takes a hard look at immutability, performance, and caching.

Chapter 6 performs a miracle that not only removes the conditionals, but also allows you to finally implement the new six-pack requirement without altering existing code.

Chapter 7 examines the tradeoffs along a continuum of six different styles of Factories. It begins by exploring a simple, hard-coded conditional, and ends with a factory whose candidate objects both self-register, and also supply the logic needed to choose them.

Chapter 8 introduces another new requirement—to vary the lyrics. It uses this requirement to introduce the idea of a programming aesthetic, or set of rules to guide you in times of uncertainty. The chapter ends with a list of specific suggestions for deciding when it's worthwhile to voluntarily improve code.

Chapter 9 comes full circle and returns to testing. It takes advantage of the improved design to write better tests, and then uses the new tests as a spur to improve the final design.

Who Should Read This Book

The lessons in the book have been found useful by programmers with a broad range of experience, from rank novice through grizzled veteran. Despite what one might predict, novices often have an easier time with this material. As they are unencumbered by prior knowledge, their minds are open, and easily absorb these ideas.

It's the veterans who struggle. Their habits are deeply ingrained. They know themselves to be good at programming. They feel quick, and efficient, and so resist new techniques, especially when those techniques temporarily slow them down.

This book *will* be useful if you are a veteran, but it cannot be denied that it teaches programming techniques that likely contradict your current practice. Changing entrenched ideas can be painful. However, you cannot make informed decisions about the value of new ideas unless you thoroughly understand them, and to understand them you must commit, wholeheartedly, to learning them.

Therefore, if you are a veteran, it's best to adopt the novice mindset before reading on. Set aside prior beliefs, and dedicate yourself to what follows. While reading, resist the urge to resist. Read the entire book, work the problems, and only then decide whether to integrate these ideas into your daily practice.

Before You Read This Book

You'll learn more from this book if you spend 30 minutes working on the "99 Bottles of Beer" problem before starting to read. See the appendix for instructions.

If you just want to read on but you don't know JavaScript, have no fear. Your purchase of this book entitles you to variants in Ruby, Javascript, and (soon-to-be-released) PHP, and each language variant comes in beer and milk flavors. Every combination of variant and flavor was available for download during your purchase, and one of them should suit.

Regardless of which version you read, be assured that this book is not about any specific language or beverage; it's about object-oriented programming and design. The technical content of every version is essentially the same.

How To Read This Book

The chapters build upon one another, and so should be read in order. While isolated sections may be useful, the whole is more than the sum of its parts. The ideas gain power in relation to one another.

To get the *most* from the book, work the code samples as you read along. With active participation, you'll learn more, understand better, and retain longer. While reading has value, doing has more.

Code Examples

The code examples in this version of the book are written in Javascript. The source code shown in the book is on GitHub. The majority of code listings are extracted from this repository; and for those, the listing numbers link to the associated code in the repo.

The exercises rely on Jest.

Errata

A current list of errata is located at sandimetz.com/99bottles-errata. If you find additional errors, please email them to errata@99bottlesbook.com.

About the Authors

Sandi Metz

Sandi is the author of Practical Object-Oriented Design in Ruby. She has thirty years of experience working on large object-oriented applications. She's spoken about programming, object-oriented design and refactoring at numerous conferences including Agile Alliance Technical Conference, Craft Conf, Øredev, RailsConf, and RubyConf. She believes in simple code and straightforward explanations, and is the proud recipient of a Ruby Hero award for her contribution to the Ruby community. She prefers working software, practical solutions and lengthy bicycle trips (not necessarily in that order). Find out more about Sandi at sandimetz.com.

Katrina Owen

Katrina works for GitHub as an Advocate on the Open Source team. Katrina has ten years of experience and works primarily in Go and Ruby. She is the creator of exercism.io, a platform for programming skill development in more than 30 languages. She's spoken about refactoring and open source at international conferences such as NordicRuby, Mix-IT, Software Craftsmanship North America, OSCON, Bath Ruby and RailsConf. She received a Ruby Hero award for her contribution to the Ruby community. When programming, her focus is on automation, workflow optimization, and refactoring. Find out more about Katrina at kytrinyx.com.

TJ Stankus

TJ works as a software developer for boldfacet LLC and co-instructs software design courses with Sandi. He began his programming career over 20 years ago by accident. By hacking together WordPerfect macros to streamline his job as a proofreader, he discovered he loved programming as much as any creative activity he'd ever pursued. He has worked in mobile applications and back-end web development. He even wrote an SMTP server back when that seemed like a good idea. Today, he works primarily with Elixir and Ruby. His main interests lie not in specific programming languages, but in the essential design ideas that span programming languages and paradigms. Find out more about TJ at tj.stank.us.

About the Translators

Tom Stuart

The JavaScript translation was done by Tom Stuart. Tom is a computer scientist, programmer and technical leader. He's the former CTO of FutureLearn and Econsultancy. He has lectured on optimising compilers at the University of Cambridge and written about technology for the Guardian.

Tom is the proud author of Understanding Computation, which was published by O'Reilly in 2013.

Introduction

This book creates a simple solution to the "99 Bottles of Beer" song problem, and then applies a series of refactorings to improve the design of the code.

Put that way, the topic sounds so painfully obvious that one might reasonably wonder if this entire tome could be replaced by a few samples of code. These refactoring "end points" would be a fraction of the size of this book, and a vastly quicker read. Unfortunately, they would teach you almost nothing about programming. Writing code is the *process* of working your way to the next stable end point, not the end point itself. You don't know the answer in advance, but instead, you are seeking it.

This book documents every step down every path of code, and so provides a guided-tour of the decisions made along the way. It not only shows how good code looks when it's done, it reveals the thoughts that produced it. It aims to leave nothing out. It flings back the veil behind which sausage is being made.

One final note before diving into the book proper. The chapters that follow apply a general, broad solution to a specific, narrow problem. The authors cheerfully stipulate the fact that you are unlikely to encounter the "99 Bottles of Beer" song in your daily work, and that problems of similar size are best solved very simply. For the purposes of this book, "99 Bottles" is convenient because it's simultaneously easily understandable and surprisingly complex, and so provides an expedient stand-in for larger problems. Once you understand the solutions here, you'll be able to apply them to the much larger real world.

With that, on to the book.

1. Rediscovering Simplicity

When you were new to programming you wrote simple code. Although you may not have appreciated it at the time, this was a great strength. Since then, you've learned new skills, tackled harder problems, and produced increasingly complex solutions. Experience has taught you that most code will someday change, and you've begun to craft it in anticipation of that day. Complexity seems both natural and inevitable.

Where you once optimized code for *understandability*, you now focus on its *changeability*. Your code is less *concrete* but more *abstract*—you've made it initially harder to understand in hopes that it will ultimately be easier to maintain.

This is the basic promise of Object-Oriented Design (OOD): that if you're willing to accept increases in the complexity of your code along *some* dimensions, you'll be rewarded with decreases in complexity along *others*. OOD doesn't claim to be free; it merely asserts that its benefits outweigh its costs.

Design decisions inevitably involve trade-offs. There's always a cost. For example, if you've duplicated a bit of code in many places, the *Don't Repeat Yourself* (DRY) principle tells you to extract the duplication into a single common method and then invoke this new method in place of the old code. DRY is a great idea, but that doesn't mean it's free. The price you pay for DRYing out code is that the invoker of the new method no longer knows the result, only the message it should send. If you're willing to pay this price, that is, you are willing to be ignorant of the actual behavior, the reward you reap is that when the behavior changes, you need alter your code in only one place. The argument that OOD makes is that this bargain will save you money.

Did you divide one large class into many small ones? You can now reuse the new classes independently of one another, but it's no longer obvious how they fit together for the original case. Have you injected a dependency instead of hard-coding the class name of a collaborator? The receiver can now freely depend on new and previously unforeseen objects, but it must remain ignorant of their actual class.

The examples above change code by increasing its level of abstraction. DRYing out code inserts a level of indirection between the place that uses behavior and the place that defines it. Breaking one large class into many forces the creation of something new to embody the relationship between the pieces. Injecting a dependency transforms the receiver into something that depends on an abstract role rather than a concrete class.

Each of these design choices has costs, and it only makes sense to pay these costs if you also accrue some offsetting benefits. Design is thus about picking the right abstractions. If you choose well, your code will be expressive, understandable and flexible, and everyone will love both it and you. However, if you get the abstractions wrong, your code will be convoluted, confusing, and costly, and your programming peers will hate you.

Unfortunately, abstractions are hard, and even with the best of intentions, it's easy to get them wrong. Well-meaning programmers tend to over-anticipate abstractions, inferring them

prematurely from incomplete information. Early abstractions are often not quite right, and therefore they create a catch-22.[1] You can't create the right abstraction until you fully understand the code, but the existence of the wrong abstraction may prevent you from ever doing so. This suggests that you should not *reach* for abstractions, but instead, you should resist them until they absolutely insist upon being created.

This book is about finding the right abstraction. This first chapter starts by peeling away the fog of complexity and defining what it means to write simple code.

1.1. Simplifying Code

The code you write should meet two often-contradictory goals. It must remain concrete enough to be understood while simultaneously being abstract enough to allow for change.

Imagine a continuum with "most concrete" at one end and "most abstract" at the other. Code at the concrete end might be expressed as a single long procedure full of `if` statements. Code at the abstract end might consist of many classes, each with one method containing a single line of code.

The best solution for most problems lies not at the extremes of this continuum, but somewhere in the middle. There's a sweet spot that represents the perfect compromise between comprehension and changeability, and it's your job as a programmer to find it.

This section discusses four different solutions to the "99 Bottles of Beer" problem. These solutions vary in complexity and thus illustrate different points along this continuum.

You must now make a decision. As you were forewarned in the preface, the best way to learn from this book is to work the exercises yourself. If you continue reading before solving the problem in your own way, your ideas will be contaminated by the code that follows. Therefore, if you plan to work along, go do the 99 Bottles exercise now. When you're finished, you'll be ready to examine the following four solutions.

1.1.1. Incomprehensibly Concise

Here's the first of four different solutions to the "99 Bottles" song.

Listing 1.1: Incomprehensibly Concise

```
class Bottles {
  song() {
    return this.verses(99, 0);
  }

  verses(hi, lo) {
    return downTo(hi, lo).map(n => this.verse(n)).join('\n');
  }

  verse(n) {
    return (
      `${n === 0 ? 'No more' : n} bottle${n === 1 ? '' : 's'}` +
      ' of beer on the wall, ' +
      `${n === 0 ? 'no more' : n} bottle${n === 1 ? '' : 's'} of beer.\n` +
```

```
15 |       `${n > 0   ? `Take ${n > 1 ? 'one' : 'it'} down and pass it around`
16 |                  : 'Go to the store and buy some more'}, ` +
17 |       `${n-1 < 0 ? 99 : n-1 === 0 ? 'no more' : n-1} bottle${n-1 === 1 ? '' : 's'}`+
18 |       ' of beer on the wall.\n'
19 |     );
20 |   }
21 |}
22 |
23 |// Here is the definition of the downTo helper function
24 |// used above. It will be omitted from subsequent listings.
25 |
26 |const downTo = (max, min) => {
27 |   const numbers = [];
28 |   for (let n = max; n >= min; n--) {
29 |     numbers.push(n);
30 |   }
31 |   return numbers;
32 |};
```

This first solution embeds a great deal of logic into the verse string. The code above performs a neat trick. It manages to be concise to the point of incomprehensibility while simultaneously retaining loads of duplication. This code is hard to understand because it is inconsistent and duplicative, and because it contains hidden concepts that it does not name.

Consistency

The style of the conditionals is inconsistent. Most use the *ternary* form, as on line 12:

```
n === 0 ? 'No more' : n
```

Finally, there's the ternary within a ternary on line 17, which is best left without comment:

```
n-1 < 0 ? 99 : n-1 === 0 ? 'no more' : n-1
```

Every time the style of the conditionals changes, the reader has to press a mental reset button and start thinking anew. Inconsistent styling makes code harder for humans to parse; it raises costs without providing benefits.

Duplication

The code duplicates both data and logic. Having multiple copies of the strings "of beer" and "on the wall" isn't great, but at least *string* duplication is easy to see and understand. Logic, however, is harder to comprehend than data, and duplicated logic is doubly so. Of course, if you want to achieve maximum confusion, you can interpolate duplicated logic *inside* strings, as does the `verse` method above.

For example, "bottle" pluralization is done in three places. The code to do this is identical in two of the places, on Lines 12 and 14:

```
n === 1 ? '' : 's'
```

But later, on line 17, the pluralization logic is subtly different. Suddenly it's not `n` that matters, but `n-1`:

```
n-1 === 1 ? '' : 's'
```

Duplication of logic suggests that there are concepts hidden in the code that are not yet visible because they haven't been isolated and named. The need to sometimes say "bottle" and other times say "bottles" *means something*, and the need to sometimes use `n` and other times use `n-1` *means something else*. The code gives no clue about what these meanings might be; you're left to figure this out for yourself.

Names

The most obvious point to be made about the names in the `verse` method of Listing 1.1: Incomprehensibly Concise is that there aren't any. The verse string contains embedded logic. Each bit of logic serves some purpose, and it is up to you to construct a mental map of what these purposes might be.

This code would be easier to understand if it did not place that burden upon you, the intrepid reader. The logic that's hidden inside the verse string should be dispersed into *methods*, and `verse` should fill itself with values by sending *messages*.

Terminology: Method versus Message

A "method" is defined on an object, and contains behavior. In the previous example, the `Bottles` class defines a method named `song`.

A "message" is sent by an object to invoke behavior. In the aforementioned example, the `song` method sends the `verses` message to the receiver `this`.

Therefore, methods are *defined*, and messages are *sent*.

The confusion between these terms comes about because it is common for the receiver of a message to define a method whose name exactly corresponds to that message. Consider the example above. The `song` method sends the `verses` *message* to `this`, which results in an invocation of the `verses` *method*. The fact that the message name and the method name are identical may make it seem as if the terms are synonymous.

They are not. Think of objects as black boxes. Methods are defined within a black box. Messages are passed between them. There are many ways for an object to cheerfully respond to a message for which it does not define a matching method. While it is common for message names to map directly to method names, there is no requirement that this be so.

Drawing a distinction between messages and methods improves your OO mindset. It allows you to isolate the intention of the sender from the implementation in the receiver. OO promises that if you send the right message, the correct behavior will occur, regardless of the names of the methods that eventually get invoked.

Creating a method requires identifying the code you'd like to extract and deciding on a method name. This, in turn, requires naming the concept, and naming things is just plain hard. In the case

above, it's especially hard. This code not only contains many hidden concepts, but those concepts are mixed together, conflated, such that their individual natures are obscured. Combining many ideas into a small section of code makes it difficult to isolate and name any single concept.

When you first write a piece of code, you obviously know what it does. Therefore, during initial development, the price you pay for poor names is relatively low. However, code is read many more times than it is written, and its ultimate cost is often very high and paid by someone else. Writing code is like writing a book; your efforts are for *other* readers. Although the struggle for good names is painful, it is worth the effort if you wish your work to survive to be read. Code clarity is built upon names.

Problems with consistency, duplication, and naming conspire to make the code in Listing 1.1: Incomprehensibly Concise likely to be costly.

Note that the above assertion is, at this point, an unsupported opinion. The best way to judge code would be to compare its *value* to its *cost*, but unfortunately it's hard to get good data. Judgments about code are therefore commonly reduced to individual opinion, and humans are not always in accord. There's no perfect solution to this problem, but the Judging Code section, later in this chapter, suggests ways to acquire empirical data about the goodness of code.

Independent of all judgment about how well a bit of code is arranged, code is also charged with doing what it's supposed to do *now* as well as being easy to alter so that it can do more *later*. While it's difficult to get exact figures for value and cost, asking the following questions will give you insight into the potential expense of a bit of code:

1. *How difficult was it to write?*
2. *How hard is it to understand?*
3. *How expensive will it be to change?*

The past ("was it") is a memory, the future ("will it be") is imaginary, but the present ("is it") is true right now. The very act of looking at a piece of code declares that you wish to understand it *at this moment*. Questions 1 and 3 above may or may not concern you, but question 2 always applies.

Code is easy to understand when it clearly reflects the problem it's solving, and thus openly exposes that problem's domain. If Listing 1.1: Incomprehensibly Concise openly exposed the "99 Bottles" domain, a brief glance at the code would answer these questions:

1. *How many verse variants are there?*
2. *Which verses are most alike? In what way?*
3. *Which verses are most different, and in what way?*
4. *What is the rule to determine which verse comes next?*

These questions reflect core concepts of the problem, yet none of their answers are apparent in this solution. The number of variants, the difference between the variants, and the algorithm for looping are distressingly obscure. This code does not reflect its domain, and therefore you can

infer that it was difficult to write and will be a challenge to change. If you had to characterize the goal of the writer of Listing 1.1: Incomprehensibly Concise, you might suggest that their highest priority was brevity. Brevity may be the soul of wit, but it quickly becomes tedious in code.

Incomprehensible conciseness is clearly not the best solution for the "99 Bottles" problem. It's time to examine one that's more verbose.

1.1.2. Speculatively General

This next solution errs in a different direction. It does many things well but can't resist indulging in unnecessary complexity. Have a look at the code below:

Listing 1.2: Speculatively General

```
const NoMore = verse =>
  'No more bottles of beer on the wall, ' +
  'no more bottles of beer.\n' +
  'Go to the store and buy some more, ' +
  '99 bottles of beer on the wall.\n';

const LastOne = verse =>
  '1 bottle of beer on the wall, ' +
  '1 bottle of beer.\n' +
  'Take it down and pass it around, ' +
  'no more bottles of beer on the wall.\n';

const Penultimate = verse =>
  '2 bottles of beer on the wall, ' +
  '2 bottles of beer.\n' +
  'Take one down and pass it around, ' +
  '1 bottle of beer on the wall.\n';

const Default = verse =>
  `${verse.number} bottles of beer on the wall, ` +
  `${verse.number} bottles of beer.\n` +
  'Take one down and pass it around, ' +
  `${verse.number - 1} bottles of beer on the wall.\n`;

class Bottles {
  song() {
    return this.verses(99, 0);
  }

  verses(finish, start) {
    return downTo(finish, start)
      .map(verseNumber => this.verse(verseNumber))
      .join('\n');
  }

  verse(number) {
    return this.verseFor(number).text();
  }

  verseFor(number) {
    switch (number) {
      case 0:  return new Verse(number, NoMore);
      case 1:  return new Verse(number, LastOne);
      case 2:  return new Verse(number, Penultimate);
      default: return new Verse(number, Default);
    }
  }
}
```

```
49 |
50 |class Verse {
51 |  constructor(number, lyrics) {
52 |    this.number = number;
53 |    this.lyrics = lyrics;
54 |  }
55 |
56 |  number() {
57 |    return this.number;
58 |  }
59 |
60 |  text() {
61 |    return this.lyrics(this);
62 |  }
63 |}
```

If you find this code less than clear, you're not alone. It's confusing enough to warrant an explanation, but because the explanation naturally reflects the code, it's confusing in its own right. Don't worry if the following paragraphs muddle things further. Their purpose is to help you appreciate the complexity rather than understand the details.

The code above first defines four anonymous functions (lines 1, 7, 13, and 19) and saves them as constants (`NoMore`, `LastOne`, `Penultimate`, and `Default`). Notice that each function takes argument `verse` but only `Default` actually refers to it. The code then defines the `song` and `verses` methods. Next comes the `verse` method, which passes the current verse number to `verseFor` and sends `text` to the result (line 37). This is the line of code that returns the correct string for a verse of the song.

Things get more interesting in `verseFor`, but before pondering that method, look ahead to the `Verse` class on line 50. `Verse` instances are initialized with two arguments, `number` and `lyrics`, and they respond to two messages, `number` and `text`. The `number` method simply returns the verse number that was passed during construction. The `text` method is more complicated; it calls `lyrics`, passing `this` as an argument.

If you now return to `verseFor` and examine lines 42-45, you can see that when instances of `Verse` are created, the `number` argument is a verse number and the `lyrics` argument is one of the anonymous functions. The `verseFor` method gets invoked for every verse of the song, and therefore, one hundred instances of `Verse` will be created, each containing a verse number and the function that corresponds to that number.

To summarize, sending `verse(number)` to an instance of `Bottles` invokes `verseFor(number)`, which uses the value of `number` to select the correct anonymous function on which to create and return an instance of `Verse`. The `verse` method then sends `text` to the returned `Verse`, which in turn calls the function, passing `this` as an argument. This invokes the function, which may or may not actually use the argument that was passed. Regardless, executing the function returns a string that contains the lyrics for one verse of the song.

You can be forgiven if you suspect that this is unduly complicated. It is. However, it's curious that despite this complexity, Listing 1.2: Speculatively General does a much better job than Listing 1.1: Incomprehensibly Concise of answering the domain questions:

1. *How many verse variants are there?*
   There are four verse variants (they start on lines 1, 7, 13, and 19 above).
2. *Which verses are most alike? In what way?*
   Verses 3-99 are most alike (as evidenced by the fact that all are produced by the `Default` variant).
3. *Which verses are most different? In what way?*
   Verses 0, 1 and 2 are clearly different from 3-99, although it's not obvious in what way.
4. *What is the rule to determine which verse should be sung next?*
   Buried deep within the `NoMore` function is a hard-coded "99," which might cause one to infer that verse 99 follows verse 0.

This solution's answers to the first three questions above are quite an improvement over those of Listing 1.1: Incomprehensibly Concise. However, all is not perfect; it still does poorly on the value/cost questions:

1. *How difficult was it to write?*
   There's far more code here than is needed to pass the tests. This unnecessary code took time to write.
2. *How hard is it to understand?*
   The many levels of indirection are confusing. Their existence implies necessity, but you could study this code for a long time without discerning why they are needed.
3. *How expensive will it be to change?*
   The mere fact that indirection exists suggests that it's important. You may feel compelled to understand its purpose before making changes.

As you can see from these answers, this solution does a good job of exposing core concepts, but does a bad job of being worth its cost. This good job/bad job divide reflects a fundamental fissure in the code.

Aside from the `song` and `verses` methods, the code does two basic things. First, it defines templates for each kind of verse (lines 1-23), and second, it chooses the appropriate template for a specific verse number and renders that verse's lyrics (lines 36-63).

Notice that the verse templates contain all of the information needed to answer the domain questions. There are four templates, and therefore, there must be four verse variants. The `Default` template handles verses 3 through 99, so these verses are clearly most alike. Verses 0, 1, and 2 have their own special templates, so each must be unique. The four templates (if you ignore the fact that they're stored in anonymous functions are very straightforward, which makes answering the domain questions easy.

But it's not the templates that are costly; it's the code that chooses a template and renders the lyrics for a verse. This choosing/rendering code is overly complicated, and while complexity is not forbidden, it *is* required to pay its own way. In this case, complexity does not.

Instead of 1) defining a function to hold a template, 2) creating a new object to hold the function, and 3) invoking the function with `this` as an argument, the code could merely have put each of the four templates into a method and then used the `switch` statement on lines 42-45 to invoke the correct one. Neither the functions nor the `Verse` class are needed, and the route between them is a series of pointless jumps through needless hoops.

Given the obvious superiority of this alternative implementation, how on earth did the "calling an anonymous function" variant come about? At this remove,[2] it's difficult to be certain of the motivation, but the code gives the impression that its author feared that the logic for selecting or invoking a template would someday need to change, and so added levels of indirection in a misguided attempt to protect against that day.

They did not succeed. Relative to the alternative, Listing 1.2: Speculatively General is harder to understand without being easier to change. The additional complexity does not pay off. The author may have acted with the best of intentions, but somewhere along the way, their commitment to the plan overcame good sense.

Programmers love clever code. It's like a neat card trick that uses sleight of hand and misdirection to make magic. Writing it, or suddenly understanding it, supplies a little burst of appreciative pleasure. However, this very pleasure distracts the eye and seduces the mind, and allows cleverness to worm its way into inappropriate places.

You must resist being clever for its own sake. If you are capable of conceiving and implementing a solution as complex as Listing 1.2: Speculatively General, it is incumbent upon you to accept the *harder* task and write simpler code.

Neither Listing 1.2: Speculatively General nor Listing 1.1: Incomprehensibly Concise is the best solution for "99 Bottles". Perhaps, as was true for porridge, the third solution will be just right.[3]

1.1.3. Concretely Abstract

This solution valiantly attempts to name the concepts in the domain. Here's the code:

Listing 1.3: Concretely Abstract

```
class Bottles {
  song() {
    return this.verses(99, 0);
  }

  verses(bottlesAtStart, bottlesAtEnd) {
    return downTo(bottlesAtStart, bottlesAtEnd)
      .map(bottles => this.verse(bottles))
      .join('\n');
  }

  verse(bottles) {
    return new Round(bottles).toString();
  }
}

class Round {
  constructor(bottles) {
    this.bottles = bottles;
```

```
  }

  toString() {
    return this.challenge() + this.response();
  }

  challenge() {
    return (
      capitalize(this.bottlesOfBeer()) + ' ' +
      this.onWall() + ', ' +
      this.bottlesOfBeer() + '.\n'
    );
  }

  response() {
    return (
      this.goToTheStoreOrTakeOneDown() + ', ' +
      this.bottlesOfBeer() + ' ' +
      this.onWall() + '.\n'
    );
  }

  bottlesOfBeer() {
    return (
      this.anglicizedBottleCount() + ' ' +
      this.pluralizedBottleForm() + ' of ' +
      this.beer()
    );
  }

  beer() {
    return 'beer';
  }

  onWall() {
    return 'on the wall';
  }

  pluralizedBottleForm() {
    return this.isLastBeer() ? 'bottle' : 'bottles';
  }

  anglicizedBottleCount() {
    return this.isAllOut() ? 'no more' : this.bottles.toString();
  }

  goToTheStoreOrTakeOneDown() {
    if (this.isAllOut()) {
      this.bottles = 99;
      return this.buyNewBeer();
    } else {
      const lyrics = this.drinkBeer();
      this.bottles--;
      return lyrics;
    }
  }

  buyNewBeer() {
    return 'Go to the store and buy some more';
  }

  drinkBeer() {
    return `Take ${this.itOrOne()} down and pass it around`;
  }

```

```
85 |   itOrOne() {
86 |     return this.isLastBeer() ? 'it' : 'one';
87 |   }
88 |
89 |   isAllOut() {
90 |     return this.bottles === 0;
91 |   }
92 |
93 |   isLastBeer() {
94 |     return this.bottles === 1;
95 |   }
96 |}
97 |
98 |// Here is the definition of the capitalize helper function
99 |// used above. It will be omitted from subsequent listings.
100|
101|const capitalize =
102|   string => string.charAt(0).toUpperCase() + string.slice(1);
```

This solution is characterized by having many small methods. This is normally a good thing, but somehow in this case it's gone badly wrong. Have a look at how this solution does on the domain questions:

1. *How many verse variants are there?*
   It's almost impossible to tell.
2. *Which verses are most alike? In what way?*
   Ditto.
3. *Which verses are most different? In what way?*
   Ditto.
4. *What is the rule to determine which verse should be sung next?*
   Ditto.

It fares no better on the value/cost questions.

1. *How difficult was it to write?*
   Difficult. This clearly took a fair amount of thought and time.
2. *How hard is it to understand?*
   The individual methods are easy to understand, but despite this, it's tough to get a sense of the entire song. The parts don't seem to add up to the whole.
3. *How expensive will it be to change?*
   While changing the code inside any individual method is cheap, in many cases, one simple change will cascade and force many other changes.

It's obvious that the author of this code was committed to doing the right thing, and that they carefully followed the *Red, Green, Refactor* style of writing code. The various strings that make up the song are never repeated—it looks as though these strings were refactored into separate methods at the first sign of duplication.

The code is DRY, and DRYing out code *should* reduce costs. DRY promises that if you put a chunk of code into a method and then invoke that method instead of duplicating the code, you will save

money later if the behavior of that chunk changes. Recognize, though, that DRYing out code is not free. It adds a level of indirection, and layers of indirection make the *details* of what's happening harder to understand. DRY makes sense when it reduces the cost of change more than it increases the cost of understanding the code.

The Don't Repeat Yourself principle, like all principles of object-oriented design, is completely true. However, despite that fact that the code above is DRY, there are many ways in which it's expensive to change.

One of many possible examples is the `beer` method on line 50. This method returns the string "beer," which occurs nowhere else in the code. To change the drink to "Kool-Aid," you need only change line 51 to return "Kool-Aid" instead of "beer." As this one small change is all that's needed to meet the "Kool-Aid" requirement, on the surface, DRY has fulfilled its promise. However, step back a minute and consider the resulting method:

```
beer() {
  return 'Kool-Aid';
}
```

Or ponder some of the other method names:

```
bottlesOfBeer()
buyNewBeer()
drinkBeer()
isLastBeer()
```

In light of the "Kool-Aid" change, these names are terribly confusing. These method names no longer make sense where they are defined, and they are totally misleading in places where they are used. To mitigate this confusion, you not only have to change "beer" to "Kool-Aid" inside this method, but you also have to make the same change to every method *name* that includes the word "beer" and then again to every sender of one of those messages.

This small change in requirements forces a change in many places, which is exactly the problem DRY promises to avoid. The fault here, however, lies not with the DRY principle, but with the names of the methods.

When you choose `beer` as the name of a method that returns the string "beer," you've named the method after what it does right now. Unfortunately, when you name a method after its current implementation, you can never change that internal implementation without ruining the method name.

You should name methods not after what they do, but after what they mean, what they represent in the context of your domain. If you were to ask your customer what "beer" *is* in the context of the "99 Bottles" song, they would not answer "Beer is the *beer*," they would say something like "Beer is *the thing you drink*" or "Beer is the *beverage*."

"Beer" and "Kool-Aid" are kinds of beverages; the word "beverage" is one level of abstraction higher than "beer." Naming the method at this slightly higher level of abstraction isolates the code from changes in the implementation details. If you choose `beverage` for the method name, going from:

```
beverage() {
  return 'beer';
}
```

to:

```
beverage() {
  return 'Kool-Aid';
}
```

makes perfect sense and requires no other change.

Listing 1.3: Concretely Abstract contains many small methods, and the strings that make up the song are completely DRY. These two things exert a force for good that should result in code that's easy to change. However, in *Concretely Abstract*, this force is overcome by the high cost of dealing with methods that are named at the wrong level of abstraction. These method names raise the cost of change.

Therefore, one lesson to be gleaned from this solution is that you should name methods after the concept they represent rather than how they currently behave. However, notice that even if you edited the code to improve every method name, this code still isn't quite right.

Changing the name of the `beer` method to `beverage` makes it easy to replace the string "beer" with the string "Kool-Aid" but does nothing to improve this code's score on the domain questions. The problem goes far deeper than having methods with inadequate names. It's not just the names that are wrong, but the methods themselves. Many methods in this code represent the wrong abstractions.

The challenge of identifying the *right* abstractions is explored in future chapters, but meanwhile, it's time to consider one more solution.

1.1.4. Shameless Green

None of the solutions shown thus far do very well on the value/cost questions. *Incomprehensibly Concise* cares only for terseness. *Speculatively General* tries for extensibility but achieves unwarranted complexity. The heart of *Concretely Abstract* is in the right place, but it can't get its feet out of the mud.

Solving the "99 Bottles" problem in any of these ways requires more effort than is necessary and results in more complexity than is needed. These solutions cost too much; they do too many of the wrong things and too few of the right.

Speculatively General and *Concretely Abstract* were both written with an eye toward reducing future costs, and it is distressing to see good intentions fail so spectacularly. It's a particular shame that the abstractions are wrong because given the opportunity to do so, the code is completely willing to reveal abstractions that are right. The failure here is not bad intention—it's insufficient patience.

This next example *is* patient and so provides an antidote for all that has come before. The following solution is known as *Shameless Green*:

Listing 1.4: Shameless Green

```
class Bottles {
  song() {
    return this.verses(99, 0);
  }

  verses(upper, lower) {
    return downTo(upper, lower)
      .map(i => this.verse(i))
      .join('\n');
  }

  verse(number) {
    switch (number) {
      case 0:
        return (
          'No more bottles of beer on the wall, ' +
          'no more bottles of beer.\n' +
          'Go to the store and buy some more, ' +
          '99 bottles of beer on the wall.\n'
        );
      case 1:
        return (
          '1 bottle of beer on the wall, ' +
          '1 bottle of beer.\n' +
          'Take it down and pass it around, ' +
          'no more bottles of beer on the wall.\n'
        );
      case 2:
        return (
          '2 bottles of beer on the wall, ' +
          '2 bottles of beer.\n' +
          'Take one down and pass it around, ' +
          '1 bottle of beer on the wall.\n'
        );
      default:
        return (
          `${number} bottles of beer on the wall, ` +
          `${number} bottles of beer.\n` +
          'Take one down and pass it around, ' +
          `${number-1} bottles of beer on the wall.\n`
        );
    }
  }
}
```

The most immediately apparent quality of this code is how very simple it is. There's nothing tricky here. The code is gratifyingly easy to comprehend. Not only that, despite its lack of complexity, this solution does extremely well on the domain questions.

1. *How many verse variants are there?*
   Clearly, four.
2. *Which verses are most alike? In what way?*
   3-99, where only the verse number varies.
3. *Which verses are most different? In what way?*
   0, 1 and 2 are different from 3-99, though figuring out how requires parsing strings with your eyes.

4. *What is the rule to determine which verse should be sung next?*
   This is still not explicit. The 0 verse contains a deeply buried, hard-coded 99.

These answers are identical to those achieved by Listing 1.2: Speculatively General. *Shameless Green* and *Speculatively General* differ, though, in how they compare on the value/cost questions. *Shameless Green* is a substantial improvement.

1. *How difficult was this to write?*
   It was easy to write.
2. *How hard is it to understand?*
   It is easy to understand.
3. *How expensive will it be to change?*
   It will be cheap to change. Even though the verse strings are duplicated, if one verse changes it's easy to keep the others in sync.

By the criteria that have been established, *Shameless Green* is clearly the best solution, yet almost no one writes it. It feels embarrassingly easy, and is missing many qualities that you expect in good code. It duplicates strings and contains few named abstractions.

Most programmers have a powerful urge to do more, but sometimes it's best to stop right here. If you were charged with writing the code to produce the lyrics to the 99 Bottles song, it is difficult to imagine fulfilling that requirement in a more cost-effective way.

The Shameless Green solution is disturbing because, although the code is easy to understand, it makes no provision for change. In this particular case, the song is so unlikely to change that betting that the code is "good enough" should pay off. However, if you pretend that this problem is a proxy for a real, production application, the proper course of action is not so clear.

When you DRY out duplication or create a method to name a bit of code, you add levels of indirection that make it more abstract. In theory these abstractions make code easier to understand and change, but in practice they often achieve the opposite. One of the biggest challenges of design is knowing when to stop, and deciding well requires making judgments about code.

1.2. Judging Code

You now have access to five different solutions to the "99 Bottles of Beer" problem; the four listed in the preceding section and the one you wrote yourself.

Which is best?

You likely have an opinion on this question—one which, granted, may have been swayed by the commentary above. However, independent of that gentle influence, the sum of your experiences and expectations predispose you to assess the goodness of code in your own unique way.

You judge code constantly. Writing code requires making choices; the choices you make reflect personal, internalized criteria. You intend to write "good" code and if, in your estimation, you've

written "bad" code, you are clearly aware that you've done so. Regardless of how implicit, unachievable, or unhelpful they may be, you already have rules about code.

While having standards of *any* sort is a virtue, the chance of achieving your standards is improved if they are explicit and quantifiable. Answering the question "What makes code good?" thus requires defining goodness in concrete and actionable ways.

This is harder than one might think.

1.2.1. Evaluating Code Based on Opinion

You'd think that by now, there would exist a universally agreed-upon definition of good code that could unambiguously guide our programming behavior. The unfortunate truth is that not only are there a multitude of definitions, but these definitions generally describe how code looks when it's done without providing any concrete guidance about how to get there.

Just as "Everybody complains about the weather but nobody does anything about it,"[4] everyone has an opinion about what good code looks like, but those opinions usually don't tell us what action to take to create it. Here are a few definitions of clean code. Notice that they could describe art or wine as easily as software.

> *I like my code to be elegant and efficient.*
>
> — *Bjarne Stroustrup*
> *inventor of C++*

> *Clean code is ... full of crisp abstractions ...*
>
> — *Grady Booch*
> *author of Object Oriented Analysis and Design with Applications*

> *Clean code was written by someone who cares.*
>
> — *Michael Feathers*
> *author of Working Effectively with Legacy Code*

Your own definition probably follows along these same lines. Any pile of code can be made to *work*; good code not only works, but is also simple, understandable, expressive and changeable.

The problem with these definitions is that although they accurately describe how good code looks once it's written, they give no help with achieving this state, and provide little guidance for choosing between competing solutions. The attributes they use to describe good code are qualitative, not quantitative.

What does it mean to be "elegant?" What makes an abstraction "crisp?" Despite the fact that these definitions are undeniably correct, none are precise in a measurable way. This lack of precision means that well-meaning programmers can hold identically high standards and still have significant disagreements about relative goodness. Thus, we argue fruitlessly about code.

Since form follows function, good code can also be defined simply, and somewhat circularly, as that which provides the highest value for the lowest cost. Our sense of elegance, expressiveness

and simplicity is an outgrowth of our experiences when reading and modifying code. Code that is easy to understand and a pleasure to extend naturally feels simple and elegant.

If you could identify and measure these qualities, you could seek after them diligently and deliberately. Therefore, although your opinions about code matter, you would be well served by facts.

1.2.2. Evaluating Code Based on Facts

A "metric" is a measure of some quality of code. Metrics are, obviously, created by people, so one could argue that they merely express one individual's opinion. That assertion, however, vastly understates their worth. Measures that rise to become metrics are backed by research that has stood the test of time. They've been scrutinized by many people over many years. You can think of metrics as crowd-sourced opinions about the quality of code.

If you apply the same metric to two different pieces of source code, you can then compare that code (at least in terms of what the metric measures) by comparing the resulting numbers. While it's possible to disagree with the premise of a specific metric, and to insist that the thing it measures isn't useful, the rules of mathematics require all to concede that the numbers produced by metrics are facts.

It would be extremely handy to have agreed-upon facts with which to compare code. In search of these facts, this section examines three different metrics: Source Lines of Code, Cyclomatic Complexity, and ABC.

Source Lines of Code

In the days of yore, the desire for reproducible, reliable information about the cost of developing applications led to the creation of a metric known simply as Source Lines of Code (SLOC, sometimes shortened to just LOC). This one number has been used to predict the total effort needed to develop software, to measure the productivity of those who write it, and to predict the cost of maintaining it.

The metric has the advantage of being easily garnered and reproduced, but suffers from many flaws.

Using SLOC to predict the development effort needed for a new project is done by counting the SLOC of *existing* projects for which total effort is known, deciding which of those existing projects the new project most resembles, and then running a cost estimation model to make the prediction. If the person doing the estimating is correct about which existing project(s) the new project most closely resembles, this prediction may be accurate.

Measuring programmer productivity by counting lines of code assumes that all programmers write equally efficient code. However, novice programmers are often far more verbose than those with more experience. Despite the fact that novices write more code to produce less function, by this metric, they can seem more productive.

While the cost of maintenance is related to the size of an application, the way in which code is organized also matters. It is cheaper to maintain a well-designed application than it is to maintain

a pile of spaghetti-code.

SLOC numbers reflect code volume, and while it's useful for some purposes, knowing SLOC alone is not enough to predict code quality.

Cyclomatic Complexity

In 1976, Thomas J. McCabe, Sr. published "A Complexity Measure", in which he asserted:

> *What is needed is a mathematical technique that will provide a quantitative basis for modularization and allow us to identify software modules that will be difficult to test or maintain.*

A "mathematical technique" to identify code that is "difficult to test or maintain"—this could be the perfect tool for assessing code. In his paper, McCabe describes his Cyclomatic Complexity metric, an algorithm that counts the number of unique execution paths through a body of source code. Think of this algorithm as a little machine that ponders your code and then maps out all the possible routes through every combination of every branch of every conditional. A method with many deeply nested conditionals would score very high, while a method with no conditionals at all would score 0.

Cyclomatic complexity neither predicts application development time nor measures programmer productivity. Its desire to identify code that is *difficult to test or maintain* aims directly at code quality.

Cyclomatic complexity can be used in several ways. First, you can use it to compare code. If you have two variants of the same method, you can choose between them based on their cyclomatic complexity. Lower scores are better and so by extension the code with the lowest score is the best.

Next, you can use it to limit overall complexity. You can set standards for how high a score you're willing to accept, and require explicit dispensation before allowing code to exceed this maximum.

Finally, you can use it to determine if you've written enough tests. Cyclomatic complexity tells you the minimum number of tests needed to cover all of the logic in the code. If you have fewer tests than cyclomatic complexity recommends, you don't have complete test coverage.

Cyclomatic complexity sounds great, and it's easy to see that it could be useful, but it views the world of code through a narrow lens.

Assignments, Branches and Conditions (ABC) Metric

The problem with cyclomatic complexity is that it doesn't take everything into account. Code does more than just evaluate conditions; it also assigns values to variables and sends messages. These things add up, and as you do more and more of each, your code becomes increasingly difficult to understand.

In 1997, twenty-one years after the unveiling of cyclomatic complexity, Jerry Fitzpatrick published "Applying the ABC Metric to C, C++, and Java", in which he describes a metric that *does*

consider more than conditionals. His ABC stands for assignments, branches and conditions, where:

- *Assignments* is a count of variable assignments.
- *Branches* counts not branches of an if statement (as one could forgivably infer) but branches of control, meaning function calls or message sends.
- *Conditions* counts conditional logic.

Fitzpatrick describes the ABC metric as a measure of size, as if ABC is a more sophisticated version of SLOC. This is *his* metric so he certainly gets to say what it represents, but you will not go wrong if you think of ABC scores as reflecting cognitive as opposed to physical size. High ABC numbers indicate code that takes up a lot of mental space. In this sense, ABC is a measure of complexity. Highly complex code is difficult to understand and change, therefore ABC scores are a proxy for code quality.

At the time of writing this there are no widely adopted—or even maintained—metrics libraries for JavaScript. The JavaScript landscape changes quickly, so regardless, consider looking for tools to run against your code.

ABC scores provide an independent perspective that may challenge your ideas about complexity and design. High scores suggest that code will be hard to test and expensive to maintain. If you believe your code to be simple but the ABC score says otherwise, you should think again.

Metrics are fallible but human opinion is no more precise. Checking metrics regularly will keep you humble *and* improve your code.

1.2.3. Comparing Solutions

Now that you have some insight into code metrics, it's time to ponder how the code examples shown in this chapter compare.

Since all of the solutions have virtually identical implementations for `song` and `verses`, these can be ignored, focusing the counts on the code that is necessary for the definition of `verse`.

The following table shows each solution's total lines of code (SLOC) along with back-of-the-napkin counts of assignments, branches, and conditionals, and the resulting ABC[5] score.

Table 1.1: Metrics

Solution	SLOC	Assignments	Branches	Conditionals	ABC
Listing 1.1: Incomprehensibly Concise	32	0	0	9 (2 branches each, 2 nested)	9

Solution	SLOC	Assignments	Branches	Conditionals	ABC
Listing 1.2: Speculatively General	63	6	9-ish	1 (4 branches)	11
Listing 1.3: Concretely Abstract	102	3	22-ish	4 (2 branches each)	23
Listing 1.4: Shameless Green	44	0	0	1 (4 branches)	1

Here's a chart of the above SLOC and ABC scores.

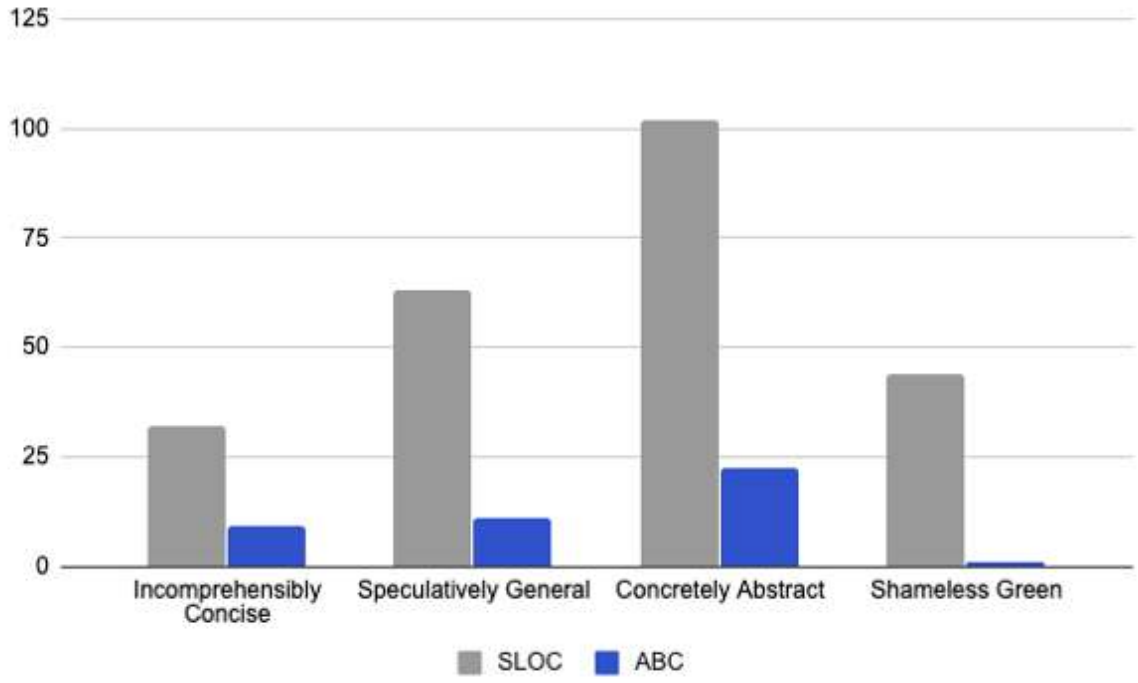

Figure 1.1: Metrics Chart

Although SLOC is not related to ABC score, both are considered a measure of size, and they appear to be roughly correlated. A larger solution in terms of SLOC is for the most part also larger in terms of its ABC score. *Shameless Green* is the notable exception—it is second lowest in SLOC, and while the ABC score is not available, given the counts it is almost certainly lower than the others by a considerable margin.

Incomprehensibly Concise and *Concretely Abstract* are interesting in that they both have characteristically complex code, but along different axes. *Incomprehensibly Concise* has the fewest lines of code, but scores highest on conditionals. It has managed to condense a lot of complexity into very few lines. In *Concretely Abstract*, on the other hand, the complexity lies in the branching

(message sends), not so much in the conditionals. The complexity in this solution is spread across the largest number of lines of code; no single method has any significant amount of complexity.

Speculatively General is shorter than both *Incomprehensily Concise* and *Concretely Abstract*, and its complexity is spread out more evenly among assignments, branches, and conditionals.

Shameless Green scores best on all counts except SLOC. *Incomprehensibly Concise* is the shortest solution, but when you look at the ratio of lines of code to conditionals, *Shameless Green* comes out more favorably.

Incomprehensibly Concise and *Shameless Green* are similar in that most of their complexity is contained in a single method. Neither has assignments or branching. Despite this similarity, if you compare their SLOC scores to their conditional counts, you'll see that they are also very different. While *Incomprehensibly Concise* has many conditionals relative to SLOC, *Shameless Green* has the opposite. *Incomprehensibly Concise* packs a lot of complexity into a few lines of code. *Shameless Green* is biased in the other direction; it has more code but is much simpler.

If your goal is to write straightforward code, these metrics point you toward *Shameless Green*.

1.3. Summary

As programmers grow, they get better at solving challenging problems, and become comfortable with complexity. This higher level of comfort sometimes leads to the belief that complexity is inevitable, as if it's the natural, inescapable state of all finished code. However, there's something beyond complexity—a higher level of simplicity. Infinitely experienced programmers do not write infinitely complex code; they write code that's blindingly simple.

This chapter examined four possible solutions to the "99 Bottles" problem as a prelude to defining what it means to write simple code. It used *metrics* as a starting point, injected a bit of common sense, and landed on *Shameless Green*.

Shameless Green is defined as the solution that quickly reaches green while prioritizing understandability over changeability. It uses tests to drive comprehension, and patiently accumulates concrete examples while awaiting insight into underlying abstractions. It doesn't dispute that DRY is good, rather, it believes that it is cheaper to manage temporary duplication than to recover from incorrect abstractions.

Writing Shameless Green is fast, and the resulting code might be "good enough." Most programmers find it embarrassingly duplicative, and the code is certainly not very object-oriented. However, if nothing ever changes, the most cost-effective strategy is to deploy this code and walk away.

The challenge comes when a change request arrives. Code that's good enough when nothing ever changes may *not* be good enough when things do. Chapter 3 introduces just such a change, and in that chapter you'll begin improving the code. Before moving on, however, it's time to take a step back, and learn how to test-drive Shameless Green.

2. Test Driving Shameless Green

The previous chapter examined four solutions to the "99 Bottles" problem, and asserted that the one known as Shameless Green is best. The Shameless Green solution consists of intention-revealing, working software, and is the result of writing simple code to pass a series of pre-supplied tests.

The provenance of the code that was written in Chapter 1 is obvious, but the tests appeared without explanation. It is now time to take a step back, and investigate how to create tests that lead to Shameless Green.

2.1. Understanding Testing

A generation ago, a handful of extreme programming (XP) practitioners began writing automated tests using a technique they called "test first development." Their ideas were so influential that automated tests are now the norm, and these tests are often written first, in prelude to writing code.

The practice of writing tests before writing code became known as *test-driven development* (TDD). In its simplest form, TDD works like this:

1. *Write a test.*
   Because the code does not yet exist, this test fails. Test runners usually display failing tests in red.
2. *Make it run.*
   Write the code to make the test pass. Test runners commonly display passing tests in green.
3. *Make it right.*
   Each time you return to green, you can refactor any code into a better shape, confident that it remains correct if the tests continue to pass.

In *Test-Driven Development by Example*, Kent Beck describes this as the *Red/Green/Refactor* cycle and calls it "the TDD mantra."

The ideas of testing, and of testing first, have won the hearts and minds of programmers. However, a commitment to writing tests doesn't make this easy. TDD presents a never-ending challenge. You must repeatedly decide which test to write next, how to arrange code so that the test passes, and how much refactoring to do once it does. Each decision requires judgment and has consequences.

If your TDD judgment is not yet fully developed, it's reasonable to temporarily adopt that of a master. Here's an excellent guiding principle:

> *Quick green excuses all sins.*
>
> — *Kent Beck*

> *Test-Driven Development by Example*

Green means safety. Green indicates that, at least as evidenced by the tests at hand, you understand the problem. Green is the wall at your back that lets you move forward with confidence. Getting to green quickly simplifies all that follows.

This chapter illustrates how to incrementally create tests and then use these tests to drive the development of code. The examples obediently follow the Red/Green/Refactor cycle, but are fairly conservative. Because the initial goal is more about reaching green than writing perfect code, the refactoring step sometimes removes duplication and other times retains it.

The plan is to create tests that thoroughly describe the "99 Bottles" problem, and then to solve the problem with the implementation known as *Shameless Green*. The Shameless Green solution strives for maximum understandability but is generally unconcerned with changeability. Shameless Green does not assert that changeability isn't important; it merely recognizes that getting to green quickly is often at odds with writing perfectly changeable code. This chapter concentrates on creating the tests and writing simple code to pass them. Future chapters refactor the resulting code to improve the design.

2.2. Writing the First Test

The first test is often the most difficult to write. At this point, you know the least about whatever it is you intend to do. Your problem is a big, fuzzy, amorphous blob, and it's challenging to reach in and carve off a single piece. It feels important to choose well, because where you start informs how you'll proceed, and ultimately determines where you'll end. The first test can therefore seem fraught with peril.

Despite its apparent import, the choice you make here hardly matters. In the beginning, you have ideas about the problem but actually know very little. Your ideas may turn out to be correct, but it's just as possible that time will prove them wrong. You can't figure out what's right until you write some tests, at which time you may realize that the best course of action is to throw everything away and start over. Therefore, the purpose of some of your tests might very well be to prove that they represent bad ideas. Learning which ideas won't work is forward progress, however disappointing it may be in the moment.

So, while it *is* important to consider the problem and to sketch out an overall plan before writing the first test, don't overthink it. Find a starting place and get going, in faith that as you proceed, the fog will clear.

If you were to sketch out a public Application Programming Interface (API) for "99 Bottles," it might look like this:

- **verse(n)** Return the lyrics for the verse number *n*
- **verses(a, b)** Return the lyrics for verses numbered *a* through *b*
- **song()** Return the lyrics for the entire song

This API allows others to request a single verse, a range of verses, or the entire song.

Now that you have a plan for the API, there are a number of possibilities for the first test. You could write a test for the entire song, for a series of contiguous verses, or for any single verse. Because the easiest way to get started is to tackle something that you thoroughly understand, it makes sense to begin by testing a single verse, and the most logical first verse to test is the first verse to be sung. Here's that test, written in Jest:

Listing 2.1: Verse 99 Test

```
1 describe('Bottles', () => {
2   test('the first verse', () => {
3     const expected =
4       '99 bottles of beer on the wall, ' +
5       '99 bottles of beer.\n' +
6       'Take one down and pass it around, ' +
7       '98 bottles of beer on the wall.\n';
8     expect(new Bottles().verse(99)).toBe(expected);
9   });
10 });
```

The test above is as simple as can be, but notice that writing it required making many decisions. It contains both a class name (`Bottles`) and a method name (`verse(n)`). This test assumes that the `Bottles` class defines a `verse` method that takes a number as an argument, and it asserts that invoking that method with an argument of `99` returns the lyrics for the 99th verse.

This test, like all tests, contains three parts:

- **Setup** Create the specific environment required for the test.
- **Do** Perform the action to be tested.
- **Verify** Confirm the result is as expected.

Lines 3-7 above define the expected result and are thus part of the setup. Setup continues on line 8, where a new bottle is created via `new Bottles()`. Line 8 also sends `verse(99)`, which is the action, and then verifies the result with `toBe`.

Running that test produces this error:

```
ReferenceError: Bottles is not defined

   5 |       'Take one down and pass it around, ' +
   6 |       '98 bottles of beer on the wall.\n';
>  7 |     expect(new Bottles().verse(99)).toBe(expected);
     |                 ^
   8 |   });
   9 | });
  10 |
```

TDD tells you to write the simplest code that will pass this test. In this case, your goal is to write only enough code to change the error message. The above error states that the `Bottles` class does not yet exist, so the first step is to define it, as follows:

```
class Bottles {
}
```

If you're new to TDD, this may seem like a ridiculously small step. Because you wrote the test, you can confidently predict that running it a second time will now produce the following error:

```
TypeError: (intermediate value).verse is not a function

   7 |       'Take one down and pass it around, ' +
   8 |       '98 bottles of beer on the wall.\n';
>  9 |     expect(new Bottles().verse(99)).toBe(expected);
     |                          ^
  10 |   });
  11 | });
  12 |
```

You can change this error message by adding a `verse` method.

```
class Bottles {
  verse() {
  }
}
```

Running the test again produces the following error:

```
expect(received).toBe(expected) // Object.is equality

Expected: "99 bottles of beer on the wall, 99 bottles of beer.
Take one down and pass it around, 98 bottles of beer on the wall.
"
Received: undefined

   7 |       'Take one down and pass it around, ' +
   8 |       '98 bottles of beer on the wall.\n';
>  9 |     expect(new Bottles().verse(99)).toBe(expected);
     |                                     ^
  10 |   });
  11 | });
  12 |
```

There's finally sufficient code so that the test fails because the output is not as expected instead of dying because of an exception.

Jest shows the difference between expected and actual output by prefixing the expected with `Expected:` and the actual with `Received:`. Therefore, you can interpret the above failure as indicating that Jest expected…

- "99 bottles of beer on the wall, 99 bottles of beer." followed by a newline, followed by
- "Take one down and pass it around, 98 bottles of beer on the wall." followed by another newline

…but instead got `undefined`.

Pay particular attention to how newlines are represented above. The expected output string contains two newlines, specified as `\n` in the test and shown as line breaks above. The final

expected line, ", represents the second newline.

Once you reach this point, it's easy to make the test pass; just copy the expected output into the `verse` method:

Listing 2.2: Verse 99 Code

```
class Bottles {
  verse() {
    return (
      '99 bottles of beer on the wall, ' +
      '99 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '98 bottles of beer on the wall.\n'
    );
  }
}
```

Although this code passes the test, it clearly doesn't solve the entire problem. As a matter of fact, writing a second test will break it. While it may seem pointless to write an obviously temporary and transitional bit of code, this is the essence of TDD.

You as the writer of tests know that the `verse` method must eventually take the value of its argument into account, but you as the writer of code must act in ignorance of this fact. When doing TDD, you toggle between wearing two hats. While in the "writing tests" hat, you keep your eye on the big picture and work your way forward with the overall plan in mind. When in the "writing code" hat, you pretend to know nothing other than the requirements specified by the tests at hand. Thus, although each individual test is correct, until all are written, the code is incomplete.

2.3. Removing Duplication

Now that the first test passes, you must decide what to test next. This next test should do the simplest, most useful thing that proves your existing code to be incorrect. While it may have been difficult to conceive of the first test because the possibilities seem infinite, this next test is often easier because it checks something relative to the first.

Verses 99 through 3 are nearly identical—they differ only in that the number changes within each verse. The test above already checks the high end of this range, and therefore it now makes sense to test the low.

The following test for verse 3 exposes the current `verse` method to be insufficient:

Listing 2.3: Verse 3 Test

```
test('another verse', () => {
  const expected =
    '3 bottles of beer on the wall, ' +
    '3 bottles of beer.\n' +
    'Take one down and pass it around, ' +
    '2 bottles of beer on the wall.\n';
  expect(new Bottles().verse(3)).toBe(expected);
});
```

TDD requires that you pass tests by writing simple code. However, most programming problems have many solutions, and it's not always clear which one is simplest. For example, the following code passes the current tests by naming the incoming argument and adding a conditional that checks the value of `number` and returns the correct string:

Listing 2.4: Conditional

```
verse(number) {
  if (number === 99) {
    return (
      '99 bottles of beer on the wall, ' +
      '99 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '98 bottles of beer on the wall.\n'
    );
  } else {
    return (
      '3 bottles of beer on the wall, ' +
      '3 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '2 bottles of beer on the wall.\n'
    );
  }
}
```

At first glance, this code appears to have achieved the ultimate in simplicity. It can produce only the lyrics for verses 99 and 3, and so does the absolute minimum needed to pass the tests.

But consider that it now contains a conditional where none existed before. This may cause you to recall the discussion on Cyclomatic Complexity in Chapter 1. This conditional adds a new execution path through the code, and additional execution paths increase complexity. This code is simple in the sense that it can't do much, but it does that one small thing in an overly complex way.

Part of the problem is that although the `if` statement switches on `number`, the true and false branches contain many things that don't vary based on `number`. The branches do differ in that one says 99/98, and the other 3/2, but they are the same for all of the other lyrics. This code conflates things that change with things that remain the same, and so forces you to parse strings with your eyes to figure out how `number` matters.

If you were to alter the `if` statement to return only the things that change, the code would look like this:

Listing 2.5: Sparse Conditional

```
verse(number) {
  let n;

  if (number === 99) {
    n = 99;
  } else {
    n = 3;
  }

  return (
    `${n} bottles of beer on the wall, ` +
    `${n} bottles of beer.\n` +
```

```
13 |     'Take one down and pass it around, ' +
14 |     `${n - 1} bottles of beer on the wall.\n`
15 |   );
16 |}
```

This code is still very specific to the two existing tests—it can produce the lyrics for verses 99 and 3, and no other. Notice, however, that it now has two parts. The first part (lines 2-8) contains the conditional, and the second (lines 10-15) contains a template that *could* correctly generate many verses. Lines 2-8 are still specific to the existing tests, but now that you've separated the things that change from the things that remain the same, lines 10-15 are generalizable to every verse between 99 and 3.

If you were to continue down the "specific" path, you would progressively add tests for the verses between 97 and 4, each time altering the `if` statement to add a condition to check for that number. Following this strategy would ultimately result in 97 nearly identical tests and 97 nearly identical verses; each would differ only in the values of the numbers.

The obvious alternative is to instead make the code more general. Because the existing template already works for every verse between 99 and 3, you could change this code to produce those verses by deleting the `if` statement and altering the template to refer to `number`, as shown here:

Listing 2.6: Interpolation

```
1 |verse(number) {
2 |  return (
3 |    `${number} bottles of beer on the wall, ` +
4 |    `${number} bottles of beer.\n` +
5 |    'Take one down and pass it around, ' +
6 |    `${number-1} bottles of beer on the wall.\n`
7 |  );
8 |}
```

Left to your own devices, your instinct would likely have been to write the code above without bothering with the intermediate steps shown in Listing 2.4: Conditional and Listing 2.5: Sparse Conditional. However, even if you would naturally have started with this more general version, it's important to understand and be able to articulate the implications of the other implementation.

The difference between the solution that adds a conditional and the solution that interpolates a variable into a string is that in the first, as the tests get more specific, the code stays equally specific. Every verse has its own personal test and its own individual code; there will never be a time when the code can do anything which is not explicitly tested.

However, in Listing 2.6: Interpolation, *as the tests get more specific, the code gets more generic.* Once the test of verse 3 is written, the code is then generalized to produce lyrics for all verses within the 3-99 range.

Remember that the purpose of this chapter is to quickly get to Shameless Green. With that goal in mind, consider the above solutions and answer this question: Which is simplest?

As previously noted, metrics aren't everything, but they can certainly be a useful *something*. In hopes that data will help answer this question, the following chart shows Source Lines Of Code, Cyclomatic Complexity and ABC metrics for the variants, calculated roughly with pencil and paper.

Table 2.1: Metrics for Code Variants After Tests of Verse 97 and 3

Solution	SLOC	Cyclomatic Complexity	ABC
Listing 2.4: Conditional	17	2	2.2
Listing 2.5: Sparse Conditional	16	2	2.8
Listing 2.6: Interpolation	8	1	1

As you can see, as the examples progress, they get shorter, and the Cyclomatic Complexity score improves. The ABC score, on the other hand, gets worse before it gets better. You must, of course, take metrics with a grain of salt, but here they cast a clear vote for replacing duplication with an abstraction as done in Listing 2.6: Interpolation.

The next section examines a nearly identical situation where the choice of what to do about duplication is not nearly so clear-cut.

2.4. Tolerating Duplication

Verses 2, 1 and 0 must still be tested, and each is unique. Having established a pattern of testing verses in the order that they appear, it makes sense to next test verse 2.

Verse 2 differs in one small way from the previous 97. The final phrase in all previous verses refers to "*n* bottles" on the wall, and thus the word "bottles" is plural. Here in verse 2, however, the final phrase reads "1 bottle." Therefore, in line 5 of the following test of verse 2, the word "bottle" is singular instead of plural.

Listing 2.7: Verse 2 Test

```
1 test('verse 2', () => {
2   const expected =
3     '2 bottles of beer on the wall, ' +
4     '2 bottles of beer.\n' +
5     'Take one down and pass it around, ' +
6     '1 bottle of beer on the wall.\n';
7   expect(new Bottles().verse(2)).toBe(expected);
8 });
```

Running that test produces the following failure:

```
-Take one down and pass it around, 1 bottle of beer on the wall.
+Take one down and pass it around, 1 bottles of beer on the wall.
```

This failure is perfect; the test expected `1 bottle`, but got `1 bottles`.

As was true with the test for verse 3, there are two fundamentally different ways to pass this test. You can add a new conditional *around* the existing code, or use the value of `number` in some way *within* it.

This next example illustrates the first possibility by wrapping the code in a new conditional:

Listing 2.8: Stark Conditional

```
verse(number) {
  if (number === 2) {
    return (
      '2 bottles of beer on the wall, ' +
      '2 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '1 bottle of beer on the wall.\n'
    );
  } else {
    return (
      `${number} bottles of beer on the wall, ` +
      `${number} bottles of beer.\n` +
      'Take one down and pass it around, ' +
      `${number-1} bottles of beer on the wall.\n`
    );
  }
}
```

In contrast, the following alternative embeds interpolated logic into the existing verse string:

Listing 2.9: Interpolated Conditional

```
verse(number) {
  return (
    `${number} bottles of beer on the wall, ` +
    `${number} bottles of beer.\n` +
    'Take one down and pass it around, ' +
    `${number-1} bottle${number-1 === 1 ? '' : 's'} of beer ` +
     'on the wall.\n'
  );
}
```

At first glance, these two solutions look a lot like the alternatives previously explored for verse 3. Listing 2.8: Stark Conditional wraps the existing code in a new conditional (as did Listing 2.4: Conditional). Moreover, Listing 2.9: Interpolated Conditional adds interpolation to the verse string (similar to Listing 2.6: Interpolation).

The choice of the best alternative for verse 3 was guided by metrics, and this might again be useful here. The following table shows metrics for the new examples:

Table 2.2: Metrics for Code Variants After Test of Verse 2

Solution	SLOC	Cyclomatic Complexity	ABC

Solution	SLOC	Cyclomatic Complexity	ABC
Listing 2.8: Stark Conditional	17	2	2
Listing 2.9: Interpolated Conditional	9	2	2

While Listing 2.9: Interpolated Conditional is about half as long as Listing 2.8: Stark Conditional, both examples contain a conditional, and neither contains any assignments. As a result, their Cyclomatic Complexity scores are identical, as are their ABC scores. The only difference between the examples, as least as far as the metrics are concerned, is that Listing 2.9: Interpolated Conditional is shorter. Shorter is often better, but, unfortunately, not in this case.

As was stated in the previous section, as tests get more specific, code should become more generic. Code becomes more generic by becoming more abstract. One way to make code more abstract is to DRY it out, that is, to extract duplicate bits of code into a single method, to give that method a name, and then to refer to the code by this new name. DRYing out code removes the duplication and thus reduces its overall size.

In Listing 2.9: Interpolated Conditional, the code has definitely gotten shorter. One would hope this happened because the code got more abstract, but sadly, this is not the case. Examine the new conditional (repeated below for convenience):

```
`${number-1} bottle${number-1 === 1 ? '' : 's'} of beer `
```

Notice that, even if an abstraction lurks here, it certainly has not been named. If forced to suggest a name, you might call the underlying concept "pluralization," asserting that the new conditional handles pluralization by adding an "s" to the string "bottle" when `number-1` is other than 1.

If pluralization is a meaningful abstraction for "99 Bottles," perhaps you should create a `pluralize` method, as follows:

```
verse() {
  // ...
  `${number-1} ${this.pluralize(number)} of beer `
  // ...
}

pluralize(number) {
  return `bottle${number-1 === 1 ? '' : 's'}`;
}
```

Unfortunately, the code above just confuses the issue. The concept of pluralization is a red herring.[6] The need for it appeared suddenly and so it feels like an important, meaningful, test-driven idea, but only because you're working with incomplete information.

Examine Listing 2.9: Interpolated Conditional and count the number of times the word "bottle" occurs, regardless of whether it's in singular or plural form. The fact that "bottle" is duplicated many times signals that there's an underlying concept that has not yet been unearthed. Within the domain of the song, "bottle/bottles" represents something important, and that thing is *not* pluralization. These words all have something in common, and hiding a single occurrence behind pluralization logic obscures this commonality. Making one look different will ultimately make it harder to see how all are the same.

Code like this `pluralize` method gets written when programmers take the DRY principle to extremes, as if they're allergic to duplication. DRY is important but if applied too early, and with too much vigor, it can do more harm than good. When faced with a situation like this, ask these questions:

- *Does the change I'm contemplating make the code harder to understand?*
  When abstractions are correct, code is easy to understand. Be suspicious of any change that muddies the waters; this suggests an insufficient understanding of the problem.
- *What is the future cost of doing nothing now?*
  Some changes cost the same regardless of whether you make them now or delay them until later. If it doesn't increase your costs, delay making changes. The day may never come when you're forced to make the change, or time may provide better information about what the change should be. Either way, waiting saves you money.
- *When will the future arrive, or how soon will I get more information?*
  If you're in the middle of writing a test suite, better information is as close as the next test. Squeezing all duplication out at the end of every test is not necessary. It's perfectly reasonable to tolerate a bit of duplication across several tests, hoping that coding up a number of slightly duplicative examples will reveal the correct abstraction. It's better to tolerate duplication than to anticipate the wrong abstraction.

Both Listing 2.8: Stark Conditional and Listing 2.9: Interpolated Conditional have identical ABC and Cyclomatic Complexity scores. From the metrics point of view, the only measurable difference between the examples is that Listing 2.9: Interpolated Conditional is shorter. Unfortunately, it isn't shorter because it contains an abstraction; it's shorter because it crams lack of understanding into a very small space. This brevity makes the code harder to understand, and obscures the concept that underlies "bottles."

Writing Shameless Green means optimizing for understandability, not changeability, and patiently tolerating duplication if doing so will help reveal the underlying abstraction. Subsequent tests, or future requirements, will provide the exact information necessary to improve the code.

Although Listing 2.8: Stark Conditional retains some duplication, it resists creating an abstraction in advance of all available information, and so is the better of these two solutions.

2.5. Hewing to the Plan

As you've seen, when working towards Shameless Green, it makes sense sometimes to eliminate duplication and other times to retain it. The Shameless Green solution is optimized to be straightforward and intention-revealing, and it doesn't much concern itself with changeability or future maintenance. The goal is to use green to maximize your understanding of the problem and to unearth *all* available information before committing to abstractions.

At some point (actually, by the end of this chapter) you will have written a full test suite for "99 Bottles," and a complete Shameless Green solution. Once that's done, you'll have two choices. You could deploy the shameless code to production and walk away, or you could refactor it into a more changeable arrangement by DRYing out duplication and extracting abstractions.

Within Shameless Green, it is perfectly acceptable to create abstractions of ideas for which you have many unambiguous examples. For example, Listing 2.6: Interpolation reduced 97 verses to a single abstraction. Having 97 examples gives you confidence that you are seeing the correct abstraction, and creating that abstraction early makes the code easier to understand.

When writing Shameless Green, you should express the unambiguous abstractions but avoid grasping for the not-quite visible ones. Listing 2.9: Interpolated Conditional jammed a conditional inside the verse string to avoid having to write a separate, mostly duplicate, copy of verse 2. In this case the new code was confusing and there were only two examples, so here it's better to take a deep breath and write down all of verse 2 while awaiting more information.

Think of the path to Shameless Green as running on a horizontal axis. Some changes propel you forward along this path and help you quickly reach green, while others are speculative and possibly distracting tangents in a vertical direction. You should complete the entire horizontal path before indulging in any vertical digressions.

Now that you have code for verses 99-2, it makes sense to continue along the horizontal path and write a test for verse 1, as follows:

Listing 2.10: Verse 1 Test

```
test('verse 1', () => {
  const expected =
    '1 bottle of beer on the wall, ' +
    '1 bottle of beer.\n' +
    'Take it down and pass it around, ' +
    'no more bottles of beer on the wall.\n';
  expect(new Bottles().verse(1)).toBe(expected);
});
```

Verse 1 is different from the others in a number of ways:

- It begins with "1 bottle" instead of "1 bottles"
- It says "Take it down" instead of "Take one down"
- It ends with "no more bottles" instead of "0 bottles"

While it's possible to pass this test by adding interpolated logic to the verse string, your experience with the prior example should dissuade you from choosing to do so. Verse 1 is even

more special than was verse 2, and having decided that verse 2 was different enough to justify adding a condition, the patient path to Shameless Green requires that you make the same decision in the case of verse 1.

The following example adds the code for verse 1. While doing so, it converts the existing `if` statement to a `switch` statement:

Listing 2.11: Verse 1 Code

```
verse(number) {
  switch (number) {
    case 1:
      return (
        '1 bottle of beer on the wall, ' +
        '1 bottle of beer.\n' +
        'Take it down and pass it around, ' +
        'no more bottles of beer on the wall.\n'
      );
    case 2:
      return (
        '2 bottles of beer on the wall, ' +
        '2 bottles of beer.\n' +
        'Take one down and pass it around, ' +
        '1 bottle of beer on the wall.\n'
      );
    default:
      return (
        `${number} bottles of beer on the wall, ` +
        `${number} bottles of beer.\n` +
        'Take one down and pass it around, ' +
        `${number-1} bottles of beer on the wall.\n`
      );
  }
}
```

Given the prior discussion, it makes sense to add a new branch to the conditional for verse 1, but this example also switched from `if` to `switch`. These keywords tell a different story.

Look at the following pseudocode and ponder the inferences a future reader might draw. Put yourself in their place; imagine that you didn't write the code and that you don't completely understand it. What does it mean to write `if` rather than `switch`?

```
if (number === 1) {
  // something
} else if (number === 2) {
  // something else
} else {
  // default
}

switch (number) {
  case 1:
    // something
  case 2:
    // something else
  default:
    // default
}
```

Use of `if` / `else if` implies that each subsequent condition varies in a meaningful way. Because `else if` is often used to test wildly different conditions, future readers will feel obliged to closely examine each one.

In contrast, use of `switch` implies that every condition checks for equality against an explicit value. Readers of `switch` statements expect conditions to be fundamentally the same.

In the 99 Bottles case above, the conditions *are* fundamentally the same. Switching from `if` to `switch` when you add the code for verse 1 implies this sameness, and so is an act of kindness towards your reader. Intention-revealing code is built from the accumulation of such thoughtful acts.

The `verse` method is accumulating lots of duplication, and this may feel troubling. However, you are very close to having code to produce every verse. While it may be tempting to veer onto the vertical path and begin DRYing out duplication, it's best to push forward horizontally.

With the end in sight, the cost of finishing the horizontal path is low. Once it's complete, you'll have an example of every different kind of verse, and therefore maximal information about the problem. When the current code is easy to understand, and more information is imminent, be shameless and *scramble* towards green.

Proceeding horizontally, then, here's the test for verse 0:

Listing 2.12: Verse 0 Test

```
test('verse 0', () => {
  const expected =
    'No more bottles of beer on the wall, ' +
    'no more bottles of beer.\n' +
    'Go to the store and buy some more, ' +
    '99 bottles of beer on the wall.\n';
  expect(new Bottles().verse(0)).toBe(expected);
});
```

Verse 0 is unique in the following ways:

- It says "No/no more bottles" instead of "0 bottles"
- It says "Go to the store and buy some more" instead of "Take it/one down and pass it around"
- It ends with "99 bottles"

At this point you will likely be unsurprised to find that verse 0 gets its own branch in the conditional, as shown here:

Listing 2.13: Verse 0 Code

```
verse(number) {
  switch (number) {
    case 0:
      return (
        'No more bottles of beer on the wall, ' +
        'no more bottles of beer.\n' +
```

```
        'Go to the store and buy some more, ' +
        '99 bottles of beer on the wall.\n'
      );
    case 1:
      return (
        '1 bottle of beer on the wall, ' +
        '1 bottle of beer.\n' +
        'Take it down and pass it around, ' +
        'no more bottles of beer on the wall.\n'
      );
    case 2:
      return (
        '2 bottles of beer on the wall, ' +
        '2 bottles of beer.\n' +
        'Take one down and pass it around, ' +
        '1 bottle of beer on the wall.\n'
      );
    default:
      return (
        `${number} bottles of beer on the wall, ` +
        `${number} bottles of beer.\n` +
        'Take one down and pass it around, ' +
        `${number-1} bottles of beer on the wall.\n`
      );
  }
}
```

This code completes the `verse` method. You now have tests for all the verse variants, and code to make each test pass.

This implementation reveals some important concepts in the domain. It's easy, for example, to see that there are 4 basic verse variants: verse 0, verse 1, verse 2 and verses 3-99. Also, verses 3-99 are so much alike that it made sense to produce them with the same bit of code.

The other verses differ, not only from the 3-99 case, but also from each other. The `switch` statement above makes it obvious that 0, 1 and 2 are special, although granted, it's difficult to see in what way. You have to read the code carefully to see how the verses are unique.

The code is easy to understand because there aren't many levels of indirection. This lack of indirection is a direct result of the dearth of abstractions. Following the horizontal path means writing code to produce every kind of verse before diverging onto tangents to DRY out small bits of code that the verses have in common. The goal is to quickly maximize the number of whole examples before extracting abstractions from their parts.

Now that you can produce any single verse, it's time to turn your attention to producing groups of verses.

2.6. Exposing Responsibilities

The plan is for the `verses(a, b)` method to take two arguments. These arguments are numbers that specify the range of verses for which the method should generate lyrics. The high-level API has been defined, but before writing the next test, you must make several more precise decisions:

- In what order do these arguments appear? Does the first argument represent the first verse to sing, such that it is always greater than the second, or vice versa? In essence, what exactly do `a` and `b` represent, and how should they be named?
- Do the arguments denote an inclusive list, that is, should you produce lyrics for the entire range specified?
- What actual argument values does it make most sense to test?

Groups of verses get sung from a higher to a lower number, so it makes sense to have the initial argument represent the first verse to sing, and thus the higher number. It also seems natural to specify an inclusive list of verse numbers. Once you make these decisions, you've finalized this part of the API and can begin considering the tests.

The first `verses` test, like the first `verse` test, should be the simplest thing imaginable. At the beginning of this chapter, when writing the initial `verse` test, it made sense to start with the first verse of the song. Following that pattern, here it makes sense to start in the same place, with verse 99. However, since the `verses` method produces a *sequence* of verses, it needs two arguments. The shortest possible sequence is two, so it's reasonable for this first test to be for the sequence from 99 to 98.

Here's the test:

Listing 2.14: Verses 99 98 Test

```
test('a couple verses', () => {
  const expected =
    '99 bottles of beer on the wall, ' +
    '99 bottles of beer.\n' +
    'Take one down and pass it around, ' +
    '98 bottles of beer on the wall.\n' +
    '\n' +
    '98 bottles of beer on the wall, ' +
    '98 bottles of beer.\n' +
    'Take one down and pass it around, ' +
    '97 bottles of beer on the wall.\n';
  expect(new Bottles().verses(99, 98)).toBe(expected);
});
```

Here's one possible way to pass that test:

Listing 2.15: Verses 99 98 Literal

```
verses() {
  return (
    '99 bottles of beer on the wall, ' +
    '99 bottles of beer.\n' +
    'Take one down and pass it around, ' +
    '98 bottles of beer on the wall.\n' +
    '\n' +
    '98 bottles of beer on the wall, ' +
    '98 bottles of beer.\n' +
    'Take one down and pass it around, ' +
    '97 bottles of beer on the wall.\n'
  );
}
```

Although the code above clearly passes the test, many programmers will find it objectionable. If asked to articulate the flaw, you might complain that it duplicates code from the `verse` method. This is certainly true. The `verse` method already contains a fair amount of duplication, and this new `verses` method repeats some of that existing code.

Some duplication is tolerable during the search for Shameless Green. However, not all duplication is helpful, and there's something about the duplication introduced above that means it should *not* be tolerated. This new code muddies rather than clarifies the waters, and it's important to understand why.

Duplication is useful when it supplies independent, specific examples of a general concept that you don't yet understand. For example, in the prior section, the `switch` statement within `verse` evolved to contain four different templates. Those templates are concrete examples of a more generic verse. Each supplies unique information, but together they point you towards the underlying abstraction.

The problem with the `verses` implementation above is that it does *not* isolate a new, independent example, but instead, it duplicates one that you've already identified. The code to produce verses 99 and 98 already exists in the `default` clause of the `switch` statement of `verse` (repeated below).

Listing 2.16: Verse Owns the Default

```
verse(number) {
  switch (number) {
    // ...
    default:
      return (
        `${number} bottles of beer on the wall, ` +
        `${number} bottles of beer.\n` +
        'Take one down and pass it around, ' +
        `${number-1} bottles of beer on the wall.\n`
      );
  }
}
```

Note that Listing 2.15: Verses 99 98 Literal is just the non-generalized version of the above pattern. Thus, this new code *duplicates an example that already exists* and so supplies no new information about the problem. In addition, duplicating this already-existing code masks the true responsibility of `verses`. This method would be more intention-revealing if this hidden responsibility were exposed instead of obscured.

The `verses` method is responsible for understanding its input arguments, and for knowing how to use these arguments to produce the correct output. Its job is not to know the exact lyrics for any verse. Its job is, rather, to repeatedly refer this question on to the `verse` method, and to accumulate the answers into a multi-verse string.

Code longs to be as ignorant as possible. While it makes perfect sense for the `verse` method to be responsible for knowing the verse templates, once `verse` assumes this responsibility, other parts of your application should not usurp it.

Here's an alternative implementation of `verses` that knows less but reveals more:

Listing 2.17: Verses 99 98 Message

```
verses() {
  return this.verse(99) + '\n' + this.verse(98);
}
```

The story this code tells is that `verses` are made up of `verses` (sorry), and that there's a relationship between a sequence of verses and an individual verse. Listing 2.15: Verses 99 98 Literal hid that relationship, while this example begins to expose it.

The code above is the simplest thing that passes this test, but you're probably chomping at the bit to do more. You are surely aware that the `verses` method must ultimately produce lyrics for all 100 verses. You recognize that the code above is incomplete and therefore temporary. You know that the real `verses` implementation will ultimately loop from upper to lower, invoking `verse` for each number and accumulating the response. Following the "simplest-thing" rule here may feel tedious and time-consuming when the real solution is so obvious.

In Chapter 28 of *Test-Driven Development by Example*, Kent Beck describes different ways to make tests pass. Three of his "Green Bar Patterns" are:

- Fake It ("Til You Make It")
- Obvious Implementation
- Triangulate

The previous two attempts at `verses` (Listing 2.15: Verses 99 98 Literal and Listing 2.17: Verses 99 98 Message) are examples of *Fake It* because although each implementation passes the current test, the tests are not yet complete. The first example was abandoned in favor of the second, but both are *Fakes* because neither does everything the final spec will require.

An *Obvious Implementation* solution is, well, obvious, and what's obvious here is that the `verses` should loop from 99 down to 0, invoking `verse` for each number and concatenating the results. When the obvious implementation is evident, it makes sense to jump straight to it. If you are absolutely certain of the correct implementation, there's no need to wear a hair shirt[7] and repetitively inch through a series of tiny steps.

Notice, however, that attractive though this idea is, it is fraught with peril. The small steps of TDD act to incrementally reveal the correct implementation. If your absolute certainty turns out to be wrong, skipping these incremental steps means you miss the opportunity of being set right. An apparently "obvious" implementation that is actually an incorrect guess will cause a world of downstream pain.

Fake It style TDD may initially seem awkward and tedious, but with practice it becomes both natural and speedy. Developing the habit of writing just enough code to pass the tests forces you to write better tests. It also provides an antidote for the hubris of thinking you know what's right when you're actually wrong. Although it sometimes makes sense to skip the small steps and

jump immediately to the final solution, exercise caution. It's best to save *Obvious Implementation* for very small leaps.

The next Green Bar Pattern is *Triangulate*, which Beck describes as a way to "conservatively drive abstraction with tests." Triangulation requires writing several tests at once, which means you'll have multiple simultaneous broken tests. The idea is to write one bit of code which makes all of the tests pass. Triangulation is meant to force you to converge upon the correct abstraction in your code.

Triangulation is such a useful idea that Shameless Green expands it from tests to code. You can expose a common, underlying abstraction through the accumulation of multiple concrete examples. These concrete code examples often contain some duplication, but this duplication is fine as long as each overall example is independent and unique.

Now that the `verses` method works for 99 and 98, the next step is to write a test that asserts it can generate other sequences. At this point, it makes sense to test the other end of the range. Here's a test for the verses from 2 down to 0:

Listing 2.18: Verses 2, 1, 0 Test

```
test('a few verses', () => {
  const expected =
    '2 bottles of beer on the wall, ' +
    '2 bottles of beer.\n' +
    'Take one down and pass it around, ' +
    '1 bottle of beer on the wall.\n' +
    '\n' +
    '1 bottle of beer on the wall, ' +
    '1 bottle of beer.\n' +
    'Take it down and pass it around, ' +
    'no more bottles of beer on the wall.\n' +
    '\n' +
    'No more bottles of beer on the wall, ' +
    'no more bottles of beer.\n' +
    'Go to the store and buy some more, ' +
    '99 bottles of beer on the wall.\n';
  expect(new Bottles().verses(2, 0)).toBe(expected);
});
```

Once again you must choose between hard-coding a new special case or generalizing the code. For example, you *could* make the test pass by explicitly adding a new conditional to the `verses` method, like so:

Listing 2.19: Verses Specific Ranges

```
verses(starting, ending) {
  if (starting === 99) {
    return this.verse(99) + '\n' + this.verse(98);
  } else {
    return this.verse(2) + '\n' + this.verse(1) +
      '\n' + this.verse(0);
  }
}
```

Alternatively, you could alter the code to make it more abstract, as follows:

Listing 2.20: Verses Within a Range

```
verses(starting, ending) {
  return downTo(starting, ending)
    .map(i => this.verse(i))
    .join('\n');
}
```

This choice between a) adding a conditional or b) making the code more abstract should remind you of an earlier discussion. Back in the Removing Duplication section, you faced the identical situation when altering `verse` to pass the test for verse 3.

In both cases, there are many existing examples of the problem and the underlying abstraction is well understood. Therefore, the arguments made in Removing Duplication apply here just as they did previously.

Relative to its alternative, Listing 2.20: Verses Within a Range is easier to understand and just as cheap to implement, and you have all the information you need to feel confident that it's correct. It is the best solution not only because it passes the test, but also because it clearly exposes the responsibility of `verses` to produce *any* range of verses. It generalizes the code, which is the best choice when you are confident that you understand the abstraction.

Now that you can generate any sequence of verses, the final task is to produce lyrics for the entire song.

2.7. Choosing Names

At the start of this chapter, the plan was to create a `Bottles` class that implemented the following API:

- `verse(n)`
- `verses(upper, lower)    // initially verses(a, b)`
- `song()`

Thus far, this plan has worked swimmingly. The `verse` and `verses` methods are complete; it's time to move on to `song`.

The code to produce the entire song is quite straightforward, as shown here:

Listing 2.21: Song Code

```
song() {
  return this.verses(99, 0);
}
```

This is a good time to reflect upon the API as a whole, and to reconsider the `song` method. The body of `song` is scarcely longer than its name. As the `verses` method is already in the public API, users of `Bottles` don't *need* the `song` method at all—they could send `verses(99,0)` and get back the same output.

Extraneous code adds costs without providing benefits, and at this point, it's quite reasonable to challenge the need for `song`. Does `song` serve a purpose independent of `verses`, or is it redundant and thus a candidate for deletion?

Answering this question requires thinking about the problem from the *message sender's point of view*. While it's true that `verses(99, 0)` and `song` return the same output, they differ widely in the amount of knowledge they require from the sender. From the sender's point of view, it is one thing to know that you want all of the lyrics to the "99 Bottles" song, but it is quite another to know how `Bottles` produces those lyrics.

Knowledge that one object has about another creates a dependency. Dependencies tie objects together, exacerbating the cost of change. Your goal as a *message sender* is to incur a limited number of dependencies, and your obligation as a *method provider* is to inflict few.

The `song` method imposes a single dependency; to use it, you need only know its name.

Using the `verses` method to request the entire song, however, requires significantly more knowledge. The sender must know:

- the name of the `verses` method
- that the method requires two arguments
- that the first argument is the verse on which to start
- that the second argument is the verse on which to end
- that the song starts on verse 99
- that the song ends on verse 0

This is a lot of knowledge. There are many ways in which the `verses` method could change that would break senders of this message.

2.8. Revealing Intentions

Kent Beck explains the difference between intention and implementation.

> *The distinction between intention and implementation [...] allows you to understand a computation first in essence and later, if necessary, in detail.*
>
> — *Kent Beck*
> *Implementation Patterns (p. 69)*

Here `song` is the intention, and `verses(99, 0)` is the implementation. There's a big difference between wanting the lyrics for a range of verses, and wanting the lyrics for the entire song. The `verses` method is in the public API, so it must continue to exist, but its existence doesn't obviate the need for `song`. Senders of the `song` message want all of the verses, and they oughtn't be forced to trouble themselves with details about how this happens.

The `song` method having defended its worth, here's the full Shameless Green for 99 Bottles.

Listing 2.22: Shameless Green Initial

```
class Bottles {
  song() {
    return this.verses(99, 0);
  }

  verses(starting, ending) {
    return downTo(starting, ending)
      .map(i => this.verse(i))
      .join('\n');
  }

  verse(number) {
    switch (number) {
      case 0:
        return (
          'No more bottles of beer on the wall, ' +
          'no more bottles of beer.\n' +
          'Go to the store and buy some more, ' +
          '99 bottles of beer on the wall.\n'
        );
      case 1:
        return (
          '1 bottle of beer on the wall, ' +
          '1 bottle of beer.\n' +
          'Take it down and pass it around, ' +
          'no more bottles of beer on the wall.\n'
        );
      case 2:
        return (
          '2 bottles of beer on the wall, ' +
          '2 bottles of beer.\n' +
          'Take one down and pass it around, ' +
          '1 bottle of beer on the wall.\n'
        );
      default:
        return (
          `${number} bottles of beer on the wall, ` +
          `${number} bottles of beer.\n` +
          'Take one down and pass it around, ' +
          `${number-1} bottles of beer on the wall.\n`
        );
    }
  }
}
```

Pleasing as this code may be, the alert reader will have noticed that the `song` method was introduced without first writing a test. This is a clear violation of TDD.

Indeed, there are a number of gaps in the tests. For example, there is no coverage for individual verses 4 through 97, and there's no guarantee that these verses appear in the correct order.

`Bottles` now produces that correct output, and it's tempting to walk away at this point. However, doing so transfers the burden of keeping this code running to some poor downstream programmer, one who has far less understanding of the problem than you do right now.

The next section, therefore, is concerned with tightening up the tests.

2.9. Writing Cost-Effective Tests

TDD promises straightforward, bug-free software that can be confidently and easily changed. TDD does not claim to be free, merely that its benefits outweigh its costs.

Belief in the value of TDD has become mainstream, and the pressure to follow this practice approaches an unspoken mandate. Acceptance of this mandate is illustrated by the fact that it's common for folks who *don't* test to tender sheepish apologies. Even those who don't test seem to believe they ought to do so.

Despite this general agreement, the sad truth is that the promise of TDD has not been universally fulfilled. Many applications have tests that are difficult to understand, challenging to change, and prohibitively time-consuming to run. Instead of enabling change, these tests actively impede it. The world is littered with test suites that are roundly hated by their maintainers, sometimes to the point of abandonment.

A great deal of this pain originates with tests that are tied too closely to code. When this is true, every improvement to the code breaks the tests, forcing them to change in turn. Therefore, the first step in learning the art of testing is to understand how to write tests that confirm *what* your code does without any knowledge of *how* your code does it.

This section explores the problem of test-to-code coupling. As a reminder of the current state of affairs, here are the current tests:

Listing 2.23: No Song Test

```
describe('Bottles', () => {
  test('the first verse', () => {
    const expected =
      '99 bottles of beer on the wall, ' +
      '99 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '98 bottles of beer on the wall.\n';
    expect(new Bottles().verse(99)).toBe(expected);
  });

  test('another verse', () => {
    const expected =
      '3 bottles of beer on the wall, ' +
      '3 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '2 bottles of beer on the wall.\n';
    expect(new Bottles().verse(3)).toBe(expected);
  });

  test('verse 2', () => {
    const expected =
      '2 bottles of beer on the wall, ' +
      '2 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '1 bottle of beer on the wall.\n';
    expect(new Bottles().verse(2)).toBe(expected);
  });

  test('verse 1', () => {
    const expected =
```

```
      '1 bottle of beer on the wall, ' +
      '1 bottle of beer.\n' +
      'Take it down and pass it around, ' +
      'no more bottles of beer on the wall.\n';
    expect(new Bottles().verse(1)).toBe(expected);
  });

  test('verse 0', () => {
    const expected =
      'No more bottles of beer on the wall, ' +
      'no more bottles of beer.\n' +
      'Go to the store and buy some more, ' +
      '99 bottles of beer on the wall.\n';
    expect(new Bottles().verse(0)).toBe(expected);
  });

  test('a couple verses', () => {
    const expected =
      '99 bottles of beer on the wall, ' +
      '99 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '98 bottles of beer on the wall.\n' +
      '\n' +
      '98 bottles of beer on the wall, ' +
      '98 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '97 bottles of beer on the wall.\n';
    expect(new Bottles().verses(99, 98)).toBe(expected);
  });

  test('a few verses', () => {
    const expected =
      '2 bottles of beer on the wall, ' +
      '2 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '1 bottle of beer on the wall.\n' +
      '\n' +
      '1 bottle of beer on the wall, ' +
      '1 bottle of beer.\n' +
      'Take it down and pass it around, ' +
      'no more bottles of beer on the wall.\n' +
      '\n' +
      'No more bottles of beer on the wall, ' +
      'no more bottles of beer.\n' +
      'Go to the store and buy some more, ' +
      '99 bottles of beer on the wall.\n';
    expect(new Bottles().verses(2, 0)).toBe(expected);
  });
});
```

2.10. Avoiding the Echo-Chamber

The output of `song` is a string of one hundred very similar verses. The method does not yet have a test. Programmers who want to remedy this omission, but who are hyper-alert to duplication, may be tempted to test `song` like this:

Listing 2.24: Whole Song Test Logic

```
test('the whole song', () => {
  const bottles = new Bottles();
  expect(bottles.song()).toBe(bottles.verses(99, 0));
});
```

The test above asserts that `song` returns the same output as does `verses(99, 0)`. On its face, this seems like a great idea. The test is short, it passes, it was easy to write, and (at least for the moment, while you're immersed in the problem) it's easy to understand. However, this test has a major flaw that can cause it to toggle from "short and sweet" to "painful and costly" in the blink of an eye. This flaw lies dormant until something changes, so the benefits of writing tests like this accrue to the writer today, while the costs are paid by an unfortunate maintainer in the future.

Understanding this flaw requires being clear about `song`'s responsibilities. From the message sender's point of view, `song` is responsible for returning the lyrics for all 100 verses. Imagine that you were tasked to test this method but knew nothing about how `Bottles` was implemented. You would be unaware of the existence of the `verses` method, and would have no choice other than to test `song` by asserting that its output matched those lyrics.

Asserting that `song` returns the expected lyrics is very different from asserting that `song` returns the same thing as `verses`. In the first case, the `song` test is independent of implementation details and so tolerates changes to other parts of the class without breaking. In the second case, the `song` test is coupled to the current `Bottles` implementation such that it will break if the signature or behavior of `verses` changes, *even if* `song` *continues to return the correct lyrics*.

There's nothing more frustrating than making a change that preserves the behavior of an application but breaks apparently unrelated tests. If you change an implementation detail while retaining existing behavior and are then confronted with a sea of red, you are right to be exasperated. This is completely avoidable, and a sign that tests are too tightly coupled to code. Such tests impede change and increase costs.

Not only is the above `song` test too tightly-coupled to the current `Bottles` implementation, it doesn't even force you to write the right code. The following badly-broken `Bottles` class passes the test suite without actually producing the correct song. Notice that the `verses` method below can only return verses 99-98, verses 2-0, or the string "ok."

Listing 2.25: Badly Broken Bottles Song

```
class Bottles {
  song() {
    return this.verses(99, 0);
  }

  verses(starting, ending) {
    if (starting === 99 && ending === 98) {
      return (
        '99 bottles of beer on the wall, ' +
        '99 bottles of beer.\n' +
        'Take one down and pass it around, ' +
        '98 bottles of beer on the wall.\n' +
        '\n' +
        '98 bottles of beer on the wall, ' +
        '98 bottles of beer.\n' +
        'Take one down and pass it around, ' +
        '97 bottles of beer on the wall.\n'
      );
    } else if (starting === 2) {
      return this.verse(2) + '\n' + this.verse(1) +
      '\n' + this.verse(0);
    } else {
```

```
      return 'ok';
    }
  }

  verse(number) {
    switch (number) {
      case 0:
        return (
          'No more bottles of beer on the wall, ' +
          'no more bottles of beer.\n' +
          'Go to the store and buy some more, ' +
          '99 bottles of beer on the wall.\n'
        );
      case 1:
        return (
          '1 bottle of beer on the wall, ' +
          '1 bottle of beer.\n' +
          'Take it down and pass it around, ' +
          'no more bottles of beer on the wall.\n'
        );
      case 2:
        return (
          '2 bottles of beer on the wall, ' +
          '2 bottles of beer.\n' +
          'Take one down and pass it around, ' +
          '1 bottle of beer on the wall.\n'
        );
      default:
        return (
          `${number} bottles of beer on the wall, ` +
          `${number} bottles of beer.\n` +
          'Take one down and pass it around, ' +
          `${number-1} bottles of beer on the wall.\n`
        );
    }
  }
}
```

The above code exploits weaknesses in the test to get to green without actually producing all of the verses. To correct this, you might be tempted to change the `song` test as follows:

Listing 2.26: Whole Song Test Logic Again

```
test('the whole song', () => {
  const bottles = new Bottles();
  const expected = downTo(99, 0)
    .map(i => bottles.verse(i))
    .join('\n');
  expect(bottles.song()).toBe(expected);
});
```

This new test succeeds in forcing `song` to produce every verse, but altering the test in this way just digs a deeper hole. Consider what just happened. The original test asserts that sending `song` produces the same result as running the code currently contained in `song`. In other words, it asserts that

```
song()
```

and

```
verses(99, 0)
```

return the same output.

This new test asserts that `song` produces the same result as running the code currently contained in `verses`. So

```
song()
```

and

```
downTo(99, 0)
  .map(i => bottles.verse(i))
  .join('\n')
```

return the same output.

Notice that although this second variant forces the production of every verse, the test continues to echo code from `Bottles`. Now, instead of asserting that the output from `song` is like the current implementation of `song`, it asserts that the output of `song` is like the current implementation of `verses`. This doesn't improve the test, but just tightly couples the test to code that's one step farther back in the stack. If that more-distant code changes, this test might break.

There's an obvious solution to this testing problem, one alluded to above. The `song` test should know nothing about how the `Bottles` class produces the song. The clear and unambiguous expectation here is that `song` return the complete set of lyrics, and the best and easiest way to test `song` is to explicitly assert that it does.

Here's that test:

Listing 2.27: Song Test

```
  test('the whole song', () => {
    const expected =
`99 bottles of beer on the wall, 99 bottles of beer.
Take one down and pass it around, 98 bottles of beer on the wall.

98 bottles of beer on the wall, 98 bottles of beer.
Take one down and pass it around, 97 bottles of beer on the wall.

97 bottles of beer on the wall, 97 bottles of beer.
Take one down and pass it around, 96 bottles of beer on the wall.

// ...

4 bottles of beer on the wall, 4 bottles of beer.
Take one down and pass it around, 3 bottles of beer on the wall.

3 bottles of beer on the wall, 3 bottles of beer.
Take one down and pass it around, 2 bottles of beer on the wall.

2 bottles of beer on the wall, 2 bottles of beer.
Take one down and pass it around, 1 bottle of beer on the wall.

1 bottle of beer on the wall, 1 bottle of beer.
Take it down and pass it around, no more bottles of beer on the wall.
```

```
25 |
26 |No more bottles of beer on the wall, no more bottles of beer.
27 |Go to the store and buy some more, 99 bottles of beer on the wall.
28 |`;
29 |    expect(new Bottles().song()).toBe(expected);
30 |  });
```

In the listing above, the `expected` string is so long that verses 96 through 5 are elided on line 12. In real life, of course, the lyrics to all 100 verses would be explicitly detailed in this test.

The text needed for 100 verses is fairly lengthy, and you may resist writing out the full string because of concerns about duplication.

2.11. Considering Options

If you find the duplication distressing, consider the alternatives. Your choices are:

1. *Assert that the expected output matches that of some other method.*

   The first two `song` test variants do this. Those tests are coupled to the current `Bottles` implementation, and so depend upon characteristics of that code.

   These dependencies mean that changes to the `Bottles` code might break the `song` test, even if there is nothing otherwise wrong with the application.

2. *Assert that the expected output matches a dynamically generated string.*

   Once you accept that the `song` test should verify specific output rather than couple to the current implementation, you must decide how to create that output. Because `song` returns a long, duplicative string, many programmers feel tempted, perhaps even obligated, to reduce this duplication by dynamically creating the verses *within the tests*.

   However, reducing string duplication inside the `song` test would of necessity require logic. This logic already exists in the `Bottles` class, so the test would be forced to invoke, copy, or re-implement it. Regardless of how you do it, using any logic here means that a change to `Bottles` might break the `song` test in an unexpected and confusing way.

3. *Assert that the expected output matches a hard-coded string.*

   In this case (as in Listing 2.27: Song Test) not only is the expected output clearly and unambiguously stated, but the test has no dependencies. These qualities combine to make it easy to understand and to tolerate changes in code.

Of these three choices, only the third is independent of the current implementation and so guaranteed to survive changes to `Bottles`. It may be difficult to reconcile yourself to writing down the entire lyrics string, but remember, DRYing out the lyrics in the test would force you to introduce an abstraction. Tests are *not* the place for abstractions—they are the place for concretions. Abstractions belong in code. If you insist on reducing duplication by adding logic to your tests, this logic by necessity must mirror the logic in your code. This binds the tests to implementation details and makes them vulnerable to breaking every time you change the code.

DRY is a very good idea in code, but much less useful in tests. When testing, the best choice is very often just to write it down.

Here again is the complete `Bottles` listing :

Listing 2.28: Shameless Green

```
class Bottles {
  song() {
    return this.verses(99, 0);
  }

  verses(starting, ending) {
    return downTo(starting, ending)
      .map(i => this.verse(i))
      .join('\n');
  }

  verse(number) {
    switch (number) {
      case 0:
        return (
          'No more bottles of beer on the wall, ' +
          'no more bottles of beer.\n' +
          'Go to the store and buy some more, ' +
          '99 bottles of beer on the wall.\n'
        );
      case 1:
        return (
          '1 bottle of beer on the wall, ' +
          '1 bottle of beer.\n' +
          'Take it down and pass it around, ' +
          'no more bottles of beer on the wall.\n'
        );
      case 2:
        return (
          '2 bottles of beer on the wall, ' +
          '2 bottles of beer.\n' +
          'Take one down and pass it around, ' +
          '1 bottle of beer on the wall.\n'
        );
      default:
        return (
          `${number} bottles of beer on the wall, ` +
          `${number} bottles of beer.\n` +
          'Take one down and pass it around, ' +
          `${number-1} bottles of beer on the wall.\n`
        );
    }
  }
}
```

The `Bottles` tests and code are now complete. The tests are straightforward, and the code is easy to understand.

2.12. Summary

Testing, done well, speeds development and lowers costs. Unfortunately it's also true that flawed tests slow you down and cost you money.

It is worth the effort, therefore, to get good at testing. TDD can prevent costly guesses, but only if you commit to writing code in small steps. Tests can make it safe and easy to refactor, but only if they are carefully de-coupled from the current code.

Good tests not only tell a story, but they lead, step by step, to a well-organized solution. The tests written in this chapter give rise (assuming proper restraint on the part of the programmer) to Shameless Green.

The Shameless Green solution is neither clever nor extensible. Its value lies in the fact that the code is easy to understand, and cheap to write. If nothing ever changes, this solution is quite certainly good enough.

Things get more interesting only if something needs to change. So, on to Chapter 3, which introduces a new requirement, and forces you to make some hard decisions about the code.

3. Unearthing Concepts

The Shameless Green solution values understandability, straight-forwardness and efficiency, with little regard for changeability. It contains duplication, and is unapologetic about leaning in the procedural direction. It's fast, and cheap, and may be good enough, at least until something changes.

However, in the real world, requirements *do* change, and when that happens, the standards for code rise.

This chapter defines a new requirement, which triggers a deeper look at the structure of the code. It then introduces a few straightforward rules to allow you to systematically and incrementally improve code, without fear of getting lost or introducing bugs. The rules are simple, but they allow complex behavior to emerge. By the end of this chapter, you'll have begun to unearth concepts that are currently hidden in the code.

3.1. Listening to Change

Code is expensive. Writing it costs time or money. It therefore behooves you to be as efficient as possible. The most cost-effective code is as good as necessary, but no better.

However, programming is an art, and programmers love elegant code. The conundrum is that once an initial, more prosaic, solution exists, the problem is solved, and the choice of whether to deliver it as is, or to improve upon it at this moment, must be weighed carefully.

If the problem is solved, and you choose to refactor now rather than later, you pay the opportunity cost[8] of not being able to work on *other* problems. Spending time "improving" code based purely on aesthetics may not be the best use of your precious time.

A good way to know that you're using limited time wisely is to be driven by changes in requirements. The arrival of a new requirement tells you two things, one very specific, the other more general.

Specifically, a new requirement tells you exactly how the code should change. Waiting for this requirement avoids the need to speculate about the future. The requirement reveals exactly how you should have initially arranged the code.

More generally, the need for change imposes higher standards on the affected code. Code that never changes obviously doesn't need to be very changeable, but once a new requirement arrives, the bar is raised. Code that needs to be changed must be changeable. Thus, a new requirement for the 99 Bottles problem will drive you to improve the code.

Here's that new requirement: users have requested that you alter the 99 Bottles code to output "1 six-pack" in each place where it currently says "6 bottles."

Here's a reminder of the current state of the code.

Listing 3.1: Shameless Green

```
class Bottles {
  song() {
    return this.verses(99, 0);
  }

  verses(starting, ending) {
    return downTo(starting, ending)
      .map(i => this.verse(i))
      .join('\n');
  }

  verse(number) {
    switch (number) {
      case 0:
        return (
          'No more bottles of beer on the wall, ' +
          'no more bottles of beer.\n' +
          'Go to the store and buy some more, ' +
          '99 bottles of beer on the wall.\n'
        );
      case 1:
        return (
          '1 bottle of beer on the wall, ' +
          '1 bottle of beer.\n' +
          'Take it down and pass it around, ' +
          'no more bottles of beer on the wall.\n'
        );
      case 2:
        return (
          '2 bottles of beer on the wall, ' +
          '2 bottles of beer.\n' +
          'Take one down and pass it around, ' +
          '1 bottle of beer on the wall.\n'
        );
      default:
        return (
          `${number} bottles of beer on the wall, ` +
          `${number} bottles of beer.\n` +
          'Take one down and pass it around, ' +
          `${number-1} bottles of beer on the wall.\n`
        );
    }
  }
}
```

In the same way that Shameless Green makes no guesses about the future, you should refrain from making up requirements. Notice the request is not to "replace every multiple of 6 with *n* six-pack(s)" nor does it mention special handling for "cases" of beer. The requirement is simply to output "1 six-pack" where it currently says "6 bottles." Knowledge of the domain may prompt you to query your customer about these other possibilities, and past experience may occasionally lead you to infer a requirement other than the one specified. But generally it's best to clarify requirements, and then write the minimum necessary code.

Despite the fact that you should rarely infer new requirements, it's true that things that change, do. Now that someone has asked for a change, you have license to improve this code. The code arrangement that was acceptable for Shameless Green is not necessarily best for enabling change.

Conditionals are the bane of OO. Shameless Green contains a `switch` statement, and within its branches, much duplication. While this was acceptable in the initial solution, consider the result if you continue down the conditional path. The following example illustrates the problem by amending the existing code to meet the "six-pack" requirement.

Listing 3.2: Compounding Conditional Sins

```
verse(number) {
  switch (number) {
    case 0:
      return (
        'No more bottles of beer on the wall, ' +
        // ...
      );
    case 1:
      return (
        '1 bottle of beer on the wall, ' +
        // ...
      );
    case 2:
      return (
        '2 bottles of beer on the wall, ' +
        // ...
      );
    case 6:
      return (
        '1 six-pack of beer on the wall, ' +
        '1 six-pack of beer.\n' +
        'Take one down and pass it around, ' +
        '5 bottles of beer on the wall.\n'
      );
    case 7:
      return (
        '7 bottles of beer on the wall, ' +
        '7 bottles of beer.\n' +
        'Take one down and pass it around, ' +
        '1 six-pack of beer on the wall.\n'
      );
    default:
      return (
        `${number} bottles of beer on the wall, ` +
        // ...
      );
  }
}
```

The `verse` `switch` statement initially contained four branches, and in the code above the number of branches has ballooned to six. This is unacceptable. Conditionals breed, and now that this one has started reproducing, you must do something to stop it.

3.2. Starting With the Open/Closed Principle

The decision about whether to refactor in the first place should be determined by whether your code is already "open" to the new requirement.

"Open" is short for "Open/Closed," which in turn is short for "open for extension and closed for modification." The "O" in open supplies the "O" in the acronym "SOLID" (see sidebar). Code is

open to a new requirement when you can meet that new requirement without changing existing code.

SOLID Design Principles

The SOLID acronym was coined by Michael Feathers and popularized by Robert Martin. Each letter stands for a well-known principle in object-oriented design. Here's a formal definition of each one:

S - Single Responsibility

The methods in a class should be cohesive around a single purpose.

O - Open-Closed

Objects should be open for extension, but closed for modification.

L - Liskov Substitution

Subclasses should be substitutable for their superclasses.

I - Interface Segregation

Objects should not be forced to depend on methods they don't use.

D - Dependency Inversion

Depend on abstractions, not on concretions.

If you find the above definitions less than enlightening, don't despair. As principles are referenced in this book, plain language explanations (like the one below) will follow.

The "open" principle says that you should not conflate the process of moving code around, of refactoring, with the act of adding new features. You should instead separate these two operations. When faced with a new requirement, first rearrange the existing code such that it's open to the new feature, and once that's complete, then add the new code.

The current `Bottles` class is not open to the "6-packs" requirement because adding new verse variants requires editing the conditional. Therefore, when faced with this new requirement, your first task is to refactor the existing code into a shape such that you can *then* implement the new requirement by merely adding code. Unfortunately, it is quite likely that you do not know how to do this, and so are at a loss about how to approach the problem.

Fortunately, you do not have to know everything in order to choose the right place to start. When faced with this situation, be guided by the following flowchart.

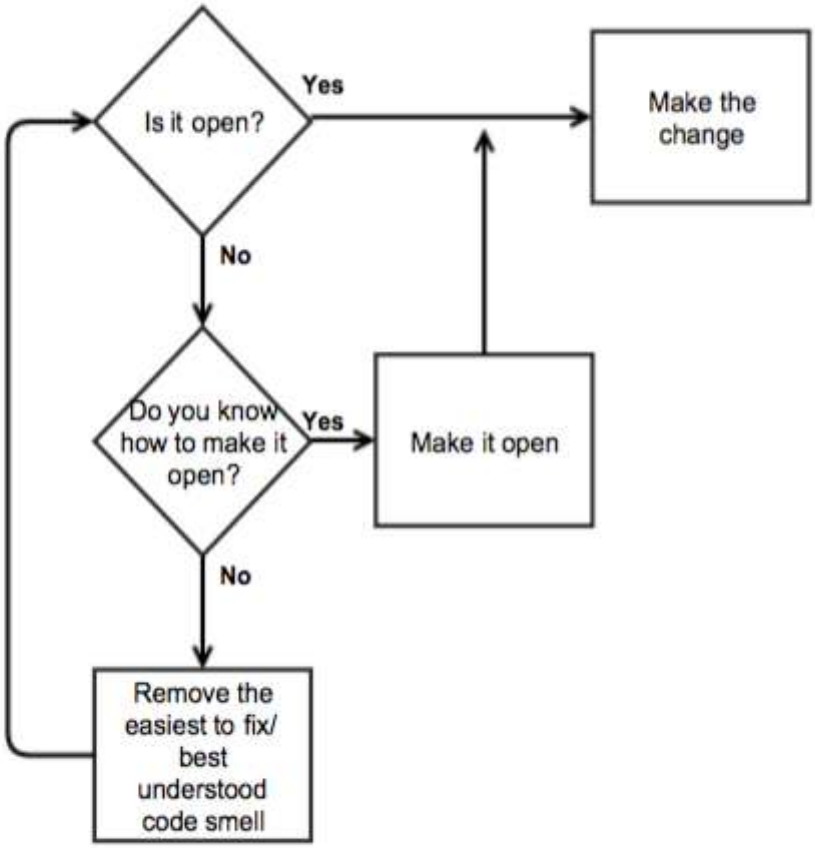

Figure 3.1: Open Closed Flowchart

As per the above flowchart, first ask yourself if the existing code is already open to the new requirement. If so, your job is simply to write the new code.

If not, next ask if you know how to alter the existing code to make it open to the new requirement. This case is also straightforward. If so, make the alteration, and then write the new code.

However, the sad truth is that the answer to both of those questions is often "no." The existing code isn't open to the new requirement, and you have no idea how to make it so. At this point "code smells" come to the rescue. If you can identify smells in code, you isolate flaws and correct them one by one.

3.3. Recognizing Code Smells

Most code is imperfect. Its flaws are many, and so thoroughly entangled that it is impossible to correct all of them at once. If you've ever tackled a bit of code, making change after change without managing to complete the task, and eventually rolling everything back, you know this problem.

The trick to successfully improving code that contains many flaws is to isolate and correct them one at a time. In his Refactoring book, Martin Fowler identifies and names many common flaws, and provides refactoring recipes to fix them. Chapter 3 (which was co-written by Kent Beck, who coined the term) calls the flaws "code smells." Thanks to Fowler's book, if you can identify a smell within code, you can look up the curative refactoring, and apply that refactoring to remove the flaw.

If you're wondering if you need to go read Fowler's book right now, the answer is, "not necessarily." Fowler's principles are introduced and demonstrated here. However, this book explores only a few of the many refactoring recipes with which you would be well-served to be familiar. Fowler's book is an excellent investment.

If asked to list a few code smells, you might suggest "duplication," or "classes that are too big," and it is indeed true that *Duplicated Code* and *Large Class* are two of the smells listed in Martin Fowler's Refactoring book. It's fairly obvious how to remove these common smells (abstract away the duplication, or divide one class into several), and so it may appear that smells are a general, hand-wavy kind of thing.

However, there are many other code smells with which you may not be as familiar. You can probably guess the definition of *Divergent Change*, but can you define *Feature Envy*? Can you recognize and specify the curative refactorings for *Primitive Obsession*, *Inappropriate Intimacy*, or *Shotgun Surgery*?

A complete exploration of every code smell is beyond the scope of this book, especially since Mr. Fowler has covered the topic so thoroughly. However, the refactorings undertaken here will be driven and guided by smells, so the task at hand is to identify the smells in the current `Bottles` class. The easiest way to unearth these smells is to make a list of the things you dislike about the code.

3.4. Identifying the Best Point of Attack

The current 99 Bottles code is not "open" to the six-pack requirement. If you are unclear about how to make it open (which is often the case), the way forward is to start removing code smells. If the smells aren't immediately obvious, start by making a list of the things you find objectionable.

Consider the `verse` method (repeated below).

Listing 3.3: Shameless Verse

```
verse(number) {
  switch (number) {
    case 0:
      return (
```

```
        'No more bottles of beer on the wall, ' +
        'no more bottles of beer.\n' +
        'Go to the store and buy some more, ' +
        '99 bottles of beer on the wall.\n'
      );
    case 1:
      return (
        '1 bottle of beer on the wall, ' +
        '1 bottle of beer.\n' +
        'Take it down and pass it around, ' +
        'no more bottles of beer on the wall.\n'
      );
    case 2:
      return (
        '2 bottles of beer on the wall, ' +
        '2 bottles of beer.\n' +
        'Take one down and pass it around, ' +
        '1 bottle of beer on the wall.\n'
      );
    default:
      return (
        `${number} bottles of beer on the wall, ` +
        `${number} bottles of beer.\n` +
        'Take one down and pass it around, ' +
        `${number-1} bottles of beer on the wall.\n`
      );
  }
}
```

This method contains a `switch` statement (the *Switch Statements* smell) whose branches contain many duplicated strings (*Duplicated Code*). Of these two smells, *Duplicated Code* is the most straightforward and so will be tackled first.

Therefore, the current task is to refactor the `verse` method to remove the duplication, in hope and expectation that the resulting code will be more open to the six-pack requirement.

Before undertaking this refactoring, it must be admitted that there is no *direct* connection between removing the duplication, and succeeding in making the code open to the six-pack requirement. That, however, is the beauty of this technique. You don't have to know how to solve the whole problem in advance. The plan is to nibble away, one code smell at a time, in faith that the path to openness will be revealed.

3.5. Refactoring Systematically

Having bandied the word around repeatedly, it's high time for a formal definition of "refactoring." According to Fowler:

> *Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.*
>
> — *Martin Fowler*
> *Refactoring*

In short, refactoring alters the arrangement of code without changing its behavior. Recall that new requirements should be implemented in two steps. First, you rearrange existing code so that

it becomes open to the new requirement. Next, you write new code to meet that requirement. The first of these steps is refactoring.

Note that safe refactoring relies upon tests. If you truly are rearranging code without changing behavior, at every step along the way the existing tests should continue to pass. Tests are a safety blanket that justifies confidence in the new arrangement of code. If they begin to fail, one of two things must be true. Either a) you've inadvertently broken the code, or b) the existing tests are flawed.

If tests fail because you've broken the code, the cure is simple. Undo the last change, make a better one and proceed merrily along your way.

However, if you rearrange code *without changing behavior* and tests begin to fail, then the tests themselves are flawed. Tests that make assertions about *how* things are done, rather than *what* actually happens, are the prime contributors to this predicament. For example, a test that makes assertions about how a method is implemented will obviously break if you change that method's implementation, even if its output is unchanged. When in this situation, there's no alternative other than to improve the tests before embarking upon a refactoring.

Tests are the wall at your back. Successful refactorings *lean* on green. Therefore, you should never change tests during a refactoring. If your tests are flawed such that they interfere with refactoring, improve them first, and then refactor.

3.6. Following the Flocking Rules

Recall that the current task is to remove duplication from the `switch` statement of the `verse` method.

The `switch` statement has four branches, each of which contains a verse template. The templates represent distinct verse variants. These variants obviously differ, but in some not-yet-identified, more-abstract way, they are also alike.

Considered from a higher viewpoint, each variant is merely a verse in the song; in that sense they are all the same. Underlying each concrete variant is a generalized verse abstraction. If you could find this abstraction, you could use it to reduce the four-branch `switch` statement to a single line of code.

The good news is that you don't have to be able to see the abstraction in advance. You can find it by iteratively applying a small set of simple rules. These rules are known as "Flocking Rules", and are as follows:

Flocking Rules

1. Select the things that are most alike.
2. Find the smallest difference between them.
3. Make the simplest change that will remove that difference.

Changes to code can be subdivided into four distinct steps:

1. parse the new code
2. parse and execute it
3. parse, execute and use its result
4. delete unused code

Making small changes means you get very precise error messages when something goes wrong, so it's useful to know how to work at this level of granularity. As you gain experience, you'll begin to take larger steps, but if you take a big step and encounter an error, you should revert the change and make a smaller one.

As you're following the flocking rules:

- For now, change only one line at a time.
- Run the tests after every change.
- If the tests fail, undo and make a better change.

Why "Flocking"?

Birds flock, fish school, and insects swarm. A flock's behavior can appear so synchronized and complex that it gives the impression of being centrally coordinated. Nothing could be further from the truth. The group's behavior is the result of a continuous series of small decisions being made by each participating individual. These decisions are guided by three simple rules.

1. Alignment - Steer towards the average heading of neighbors
2. Separation - Don't get too close to a neighbor
3. Cohesion - Steer towards the average position of the flock

Thus, complex behavior emerges from the repeated application of simple rules. In the same way that the rules in this sidebar allow birds to flock, the "Flocking Rules" for code allow abstractions to appear.

Flock of Starlings Acting As A Swarm, John Holmes, CC BY-SA 2.0

To see a beautiful example of flocking in action, watch Steven Strogatz's The Science of Sync TED talk.

3.7. Converging on Abstractions

The Flocking Rules are so atomic, and so general, that they may not yet inspire confidence. The remainder of this chapter will use them to unearth abstractions in the `verse` method, after which you may find the process more convincing.

3.7.1. Focusing on Difference

While it's true that there are problems for which the solution is obvious, those of any interesting size aren't tractable to instant understanding. They're too big or have too many parts.

When examining complicated problems, the eye is first drawn towards sameness. However, despite the fact that sameness is easier to identify, difference is more useful because it has more meaning. DRYing out sameness has *some* value, but DRYing out difference has more.

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides are commonly referred to as the "Gang of Four," in reference to their joint authorship of Design Patterns: Elements of Reusable Object-Oriented Software. This influential book describes twenty-three patterns or solutions to common OO programming problems and it explains this process thusly:

> *The focus here is encapsulating the concept that varies, a theme of many design*

“patterns.

Difference holds the key to understanding. If two concrete examples represent the same abstraction and they contain a difference, that difference must represent a smaller abstraction within the larger one. If you can name the difference, you've identified that smaller abstraction.

The good news is that a systematic application of the rules of refactoring converts difference to sameness, decomposing a problem into its constituent parts. The even better news is that this happens automatically. You don't have to identify the underlying abstractions in advance of refactoring. If you merely write the code dictated by the rules, the abstractions will follow.

The habit of believing that you understand the abstraction, and of jumping to an invented solution, is deeply ingrained. Programmers study a problem, decide on a solution, and then implement it. Solutions are crafted by intention.

If this describes your entire past experience, you may find the following code surprising. It takes many small, iterative steps, and results in a solution that is *discovered* by refactoring.

To reduce the `verse` `switch` statement to a single line of code, the rules say to first identify the things that are most alike. This means that you should select the two branches that are most alike, and focus on making them identical.

Here again is a reminder of the `switch` statement:

Listing 3.4: Verse Method Conditional

```
switch (number) {
  case 0:
    return (
      'No more bottles of beer on the wall, ' +
      'no more bottles of beer.\n' +
      'Go to the store and buy some more, ' +
      '99 bottles of beer on the wall.\n'
    );
  case 1:
    return (
      '1 bottle of beer on the wall, ' +
      '1 bottle of beer.\n' +
      'Take it down and pass it around, ' +
      'no more bottles of beer on the wall.\n'
    );
  case 2:
    return (
      '2 bottles of beer on the wall, ' +
      '2 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '1 bottle of beer on the wall.\n'
    );
  default:
    return (
      `${number} bottles of beer on the wall, ` +
      `${number} bottles of beer.\n` +
      'Take one down and pass it around, ' +
      `${number-1} bottles of beer on the wall.\n`
```

```
    );
}
```

Notice that although verse 2 contains hardcoded numbers for `2` and `1`, it could just as correctly say `number` and `number-1`, as in the `default` branch. This part looks different, but is logically the same. It may help to recall that verse 2 has but one test, which asserts that the final line says "1 bottle" instead of "1 bottles." The only real difference between the 2 and `default` cases is the word "bottle" versus the word "bottles." Therefore, these are the lines that are most alike.

3.7.2. Simplifying Hard Problems

Having found the strings that are most alike, the next task is to make them identical. It's important to focus on this specific goal without succumbing to the temptations of tangents.

Think of the process of turning these two lines into one as being on a horizontal path.[9] While walking this path, if something catches your eye in another part of the code (perhaps in the `0` or `1` cases), you may be tempted to veer off in a vertical direction. However, if you begin making changes to other parts of the code before you completely combine the `2` and `default` cases, you step off a well-trod path into a woods so dark and sinister that you might never return. While it can be useful to interleave horizontal and vertical work, it's best to finish the current journey when the terminus of the horizontal path is in sight.

Have a look at the code below, and decide what to do next.

Listing 3.5: 2 and Default Case

```
case 2:
  return (
    '2 bottles of beer on the wall, ' +
    '2 bottles of beer.\n' +
    'Take one down and pass it around, ' +
    '1 bottle of beer on the wall.\n'
  );
default:
  return (
    `${number} bottles of beer on the wall, ` +
    `${number} bottles of beer.\n` +
    'Take one down and pass it around, ' +
    `${number-1} bottles of beer on the wall.\n`
  );
```

Recall that these lines were chosen because the only real difference between them is using "bottle" versus "bottles" in the final phrase. The other apparent differences are actually similarities. The `2` and `1` in the `2` case can be replaced by `${number}` and `${number-1}` respectively, which means that these parts are logically identical.

The change needed to resolve the differences between the numbers is obvious. That part of the problem feels solved. It's boring. The "bottle/bottles" difference, however, is much more interesting. It requires more thought.

Programmers love hard problems. Many times the riskiest and most difficult bit of a larger problem feels the most interesting. It's no wonder that many programmers gravitate towards starting a problem at its most confusing part.

However, it just so happens that solving easy problems, through a magical alchemy of code, sometimes transmutes hard problems into easy ones. It is common to find that hard problems are hard only because the easy ones have not yet been solved.

Therefore, don't discount the value of solving easy problems. With that in mind, the first step towards making these lines identical is to resolve the very first difference. Scanning left to right, the very first character of the `2` case could be replaced by `${number}`. Proceeding on, the next `2` can similarly be replaced. Scanning further still, the `1` can become `${number-1}`. The result is shown below:

Listing 3.6: Replace Hard Coded Number

```
case 2:
  return (
    `${number} bottles of beer on the wall, ` +
    `${number} bottles of beer.\n` +
    'Take one down and pass it around, ' +
    `${number-1} bottle of beer on the wall.\n`
  );
default:
  return (
    `${number} bottles of beer on the wall, ` +
    `${number} bottles of beer.\n` +
    'Take one down and pass it around, ' +
    `${number-1} bottles of beer on the wall.\n`
  );
```

After making the above change (and running the tests between each, of course), the remaining difference is "bottle/bottles" on the last line:

Listing 3.7: One Difference Remains

```
case 2:
  return (
    // ...
    `${number-1} bottle of beer on the wall.\n`
  );
default:
  return (
    // ...
    `${number-1} bottles of beer on the wall.\n`
  );
```

This is the first interesting difference. Now you must decide what this difference *means*.

3.7.3. Naming Concepts

Previous sections state that if all verses are the same in some fundamental way, then an underlying verse abstraction must exist. The goal of the current refactoring is to find a way to express that more abstract verse.

If an underlying verse abstraction exists, then this small difference between verse 2 and verses 3-99 must represent a smaller abstraction within that larger one. To make these two lines the same, you must name this concept, create a method named after the concept, and replace the two differences with a common message send. Therefore, it's time to decide what the words "bottle" and "bottles" represent in the context of the song.

You may recall from the Concretely Abstract section of Chapter 1 that "bottle" is not underlying the concept. If you call the method "bottle" you are naming it after its current implementation, and you've already seen how that can go badly wrong.

Also, despite that fact that these two words differ in that one is singular and one is plural, the underlying concept is not "pluralization." Within the context of the song, "bottle/bottles" does not represent pluralization.

There are two pieces of information that can help in the struggle for a name. One is a general rule and the other is the new requirement.

First, the new requirement. Recall that the impetus for this refactoring was the need to say "six-pack" instead of "bottle/bottles" when there are 6 bottles. The string "six-pack" is one more concrete example of the underlying abstraction. This suggests that if you name the method "bottle," you will regret this decision in short order.

The general rule is that the name of a thing should be one level of abstraction higher than the thing itself. The strings "bottle/bottles/six-pack" are instances of some category, and the task is to name that category using language of the domain.

One way to identify the category is to imagine the concrete examples as rows and columns in a spreadsheet.[10] The following table illustrates this idea. This table contains three rows, one for each concrete example. Each row has two columns. The first column contains a number of bottles, and the next, the word used with that number in the song.

Table 3.1: Bottles Column Header

Number	*xxx?*
1	`bottle`
6	`six-pack`
n	`bottles`

Column 1 above contains numbers, so "Number" makes sense as a column header. The header "Number" is a level of abstraction higher than the concrete examples. "1," "6," and "n" *are* numbers.

The second column has entries for bottle, six-pack, and bottles. Bottle is an entity in this as-yet unnamed category, rather than the category itself.

It might seem as if "Unit" would be a good header. Although it's true that every example is some kind of unit, there are two problems with this name. First, it's too abstract. Unit is not *one* level of abstraction higher than the examples—it's many. There are plenty of good naming alternatives on the continuum between "bottle" and "unit." Next, unit is not in the language of the domain. The name you choose will be the name you use in conversations with your customers. Naming

things after domain concepts improves communication between you and the folks who pay the bills. Only good can come of this.

When you're struggling to find a good name but have only a few concrete instances to guide you, it can be illuminating to imagine other things that would also be in the same category.[11] For example, if the song were about wine, the wine might come in a carafe. Juice sometimes comes in small boxes. Soft drinks often come in cans.

If you were to ask your users, "What kind of thing is a bottle?," they wouldn't reply "It's a unit." Instead they might call it the *container*. In the context of "99 Bottles," container is a good name for this concept. Container is meaningful, understandable, and unambiguous.

Having named the concept, it's time to write code to remove the difference.

3.7.4. Making Methodical Transformations

Now that you've decided to create a `container` method, it's time to alter the code. It's tempting to make all of the necessary changes in one fell swoop. Doing so requires adding a new method and invoking it in two places. Here's the new method:

Listing 3.8: Guess Entire Container

```
container(number) {
  if (number === 1) {
    return 'bottle';
  } else {
    return 'bottles';
  }
}
```

This method must be invoked from both branches of the `verse` `switch` statement. Here is the code:

```
`${number-1} ${this.container(number-1)} of beer on the wall.\n`
```

But wait. Notice that the above change adds seven new lines of code, changes two existing ones, and alters code in three separate places. Any of these changes could introduce errors, which you would then be obliged to understand and correct. This small example stands in for the much bigger real-life problem where, in the process of implementing a new feature, you add many lines of code, change many others, and then run the tests, only to be confronted by an ocean of red.

Real world problems are big. Real code has bugs. Real tests are often tightly coupled to current implementations. If you simultaneously change many things and something breaks, you're forced to understand everything in order to fix anything. You could end up chasing after red, with increasing desperation, before eventually discarding all of the changes and beginning anew.

Making a slew of simultaneous changes is not refactoring—it's re*hack*toring. It would be much better to make a series of tiny changes and run the tests after each. If the tests fail, you know the exact change that caused the failure, and can undo back to green and make a better change. If

the tests pass, you know that the current code works, even if the refactoring is only partially complete.

Formal refactoring confers two additional benefits. First, because no change breaks the tests, the code can be deployed to production at any intermediate point. This allows you to avoid accumulating a large set of changes and suffering through a painful merge. Next, code that runs properly even in the midst of a long refactoring increases the bus factor. This contributes to a higher likelihood of project success even if you, personally, were to meet an untimely end.

Adding the `container` method by re*fact*oring means taking a series of small steps. As a reminder, here again are the Flocking Rules and corollaries:

Flocking Rules

1. Select the things that are most alike.
2. Find the smallest difference between them.
3. Make the simplest change to remove that difference:
    a. parse the new code
    b. parse and execute it
    c. parse, execute and use its result
    d. delete unused code

As you're following the rules:

- In general, change only one line at a time.
- Run the tests after every change.
- If you go red, undo and make a better change.

You've already followed rule 1 (you chose the `2` and `default` cases) and rule 2 (you've worked your way across to the "bottle/bottles" difference). Now you're on rule 3, ready to remove this difference. As you intend to change only one line at a time, you'll of necessity have to make small changes iteratively.

The first step is to create an empty `container` method.

Listing 3.9: Empty Container Method

```
container() {
}
```

Now run the tests.

If this admonition comes as a surprise, consider that having green tests at this point provides a very useful piece of feedback. Even though the `container` method is not yet being invoked, green tests at this point prove that the code you just wrote is syntactically correct. This means you are following rule 3a, which calls for separating *parse* from *execute*.

Now that you have written this admittedly not very exciting `container` method, the next step is to make the smallest change that will advance the code in the intended direction. Here's a reminder of the target line:

Listing 3.10: One Difference Remains Redux

```
case 2:
  return (
    // ...
    `${number-1} bottle of beer on the wall.\n`
  );
default:
  return (
    // ...
    `${number-1} bottles of beer on the wall.\n`
  );
```

The current `container` method returns `undefined`. It will eventually be called from two places. The `2` case wants the return to be "bottle," and the `default` case, "bottles." The next incremental change is to alter the method to make it usable for just one of those callers. Therefore, you must now choose which value to return first.

The default case is often a good place to start, and there's no reason not to do so here. In that spirit, change `container` to return `bottles`, like so:

Listing 3.11: Sparse Container Method

```
container() {
  return 'bottles';
}
```

From now on, it goes without saying that you should run the tests after every change.

Now that `container` returns a usable value, alter the `default` branch to send the message in place of the word "bottles," as on line 12 below:

Listing 3.12: Sparse Container Used in Default Branch

```
verse(number) {
  switch (number) {
    // ...
    case 2:
      return (
        // ...
        `${number-1} bottle of beer on the wall.\n`
      );
    default:
      return (
        // ...
        `${number-1} ${this.container()} ` +
          'of beer on the wall.\n'
      );
  }
}

container() {
  return 'bottles';
}
```

So far, so good, but consider the next step. To be usable in both the `2` and `default` cases, `container` must eventually return the correct choice between `bottle` or `bottles`. The decision between them is based on the value of `number`, which `container` does not yet know. Therefore, `container` must be changed to take an argument.

3.7.5. Refactoring Gradually

In his book *Refactoring to Patterns*, Joshua Kerievsky talks about "Gradual Cutover Refactoring," a strategy for keeping the code in a releasable state by gradually switching over a small number of pieces at a time. This type of refactoring can be done alongside other development work without affecting the release schedule. If you adopt this strategy, your colleagues and your customers will appreciate your commitment to keeping the code deployable.

In the current example, you ought not to edit all of the senders simultaneously. Therefore, to do a gradual cutover refactoring, you have to figure out how to allow some senders to pass the new argument while others remain unchanged. Conveniently, the JavaScript language is sufficiently relaxed that it treats all arguments as optional, so you may begin by simply adding an argument:

Listing 3.13: Container With Argument

```
container(number) {
  return 'bottles';
}
```

The above code takes an argument named `number`, which defaults to `undefined` if no value is provided. This default behavior is a temporary shim whose purpose is to enable a step-by-step refactoring. Once the refactor is complete, you will always be supplying a value for this argument so that `undefined` is never used.

Now that the `container` method accepts an argument, consider the next step. You could either:

- alter `container` to check the value of `number` and return "bottle" or "bottles," meaning change the receiver, or
- alter the `default` branch to add the `number` argument to `container` message, meaning change the sender.

The refactoring rules prohibit you from making both of these changes at once, so you must choose one or the other.

Because the `container` method does not yet reference `number`, changing the `default` branch to pass this argument changes almost nothing about the code. Instead of passing the argument, the better choice is to expand the code in `container` to use `number` to decide which of "bottle" or "bottles" to return, as follows:

Listing 3.14: Container With Conditional

```
container(number) {
  if (number === 1) {
    return 'bottle';
  } else {
    return 'bottles';
```

```
6 |   }
7 | }
```

There are several things to note about the above strategy.

First, notice that adding the conditional was very clearly a multi-line change. This may appear to break the "make changes on only one line" rule, but in this case, the change is obeying the spirit of the law while slightly ignoring its letter. This conditional could have been expressed in ternary form, as:

```
return number === 1 ? 'bottle' : 'bottles';
```

which would certainly have been a one-line change. The multiline `if` form above is preferred in this refactoring for reasons that will become clear in later chapters. For now, just think of these two forms as both obeying the "one line" rule.

Next, remember that this method is being invoked from only one place (the `default` branch of the `switch` statement in `verse`), and that as yet no argument is being passed. This means that the `number` argument in `container` gets set to `undefined`, which routes execution to the `false` branch. The new code in the true branch is not yet being executed, although it gets parsed when the tests run.

The act of adding a new branch to the conditional while executing only the previously existing code is a mini-example of the Open/Closed Principle. You can think of this change as making the `container` method open to a new requirement—enabling it to occasionally return the word "bottle." This splits the change into several small steps, which makes it easier to debug any errors.

The next tiny step is to change the sender to actually pass the new argument. Because `container` is being invoked from the fourth phrase of the song, the value of the argument is `number-1`, as shown on line 12 below:

Listing 3.15: Passing an Argument to Container

```
1 |verse(number) {
2 |  switch (number) {
3 |    // ...
4 |    case 2:
5 |      return (
6 |        // ...
7 |        `${number-1} bottle of beer on the wall.\n`
8 |      );
9 |    default:
10|      return (
11|        // ...
12|        `${number-1} ${this.container(number-1)} ` +
13|          'of beer on the wall.\n'
14|      );
15|  }
16|}
17|
18|container(number) {
19|  if (number === 1) {
20|    return 'bottle';
```

```
21 |   } else {
22 |     return 'bottles';
23 |   }
24 |}
```

The above step might seem so tiny as to seem pointless to isolate, but there's a real difference between executing the `false` branch because of the `undefined` default, and being routed there because of the value of the `number` argument. In the first case, you know that *if you go to the* `false` *branch* the tests pass, and in the second, you know that *the argument being passed takes you to the* `false` *branch*. Both of these things must work or the tests will break. Changing code at this level of granularity makes it easier to handle unexpected failures.

The next step is to change the 2 branch so that it also invokes the container method, as shown on line 9 below:

Listing 3.16: 2 and Default Cases Identical

```
1 |verse(number) {
2 |  switch (number) {
3 |    // ...
4 |    case 2:
5 |      return (
6 |        `${number} bottles of beer on the wall, ` +
7 |        `${number} bottles of beer.\n` +
8 |        'Take one down and pass it around, ' +
9 |        `${number-1} ${this.container(number-1)} ` +
10|          'of beer on the wall.\n'
11|      );
12|    default:
13|      return (
14|        `${number} bottles of beer on the wall, ` +
15|        `${number} bottles of beer.\n` +
16|        'Take one down and pass it around, ' +
17|        `${number-1} ${this.container(number-1)} ` +
18|          'of beer on the wall.\n'
19|      );
20|  }
21|}
22|
23|container(number) {
24|  if (number === 1) {
25|    return 'bottle';
26|  } else {
27|    return 'bottles';
28|  }
29|}
```

The above change has two consequences. First, all of the code in `container` is now being executed. Next, the code in the `2` and `default` branches of the `verse` `switch` statement are now identical.

One task remains to complete this entire horizontal refactoring. The `2` case is now obsolete, and so it can be deleted. The following example shows the resulting code:

Listing 3.17: 2 Subsumed Into Default Case

```
1 |verse(number) {
2 |  switch (number) {
```

```
    case 0:
      return (
        'No more bottles of beer on the wall, ' +
        'no more bottles of beer.\n' +
        'Go to the store and buy some more, ' +
        '99 bottles of beer on the wall.\n'
      );
    case 1:
      return (
        '1 bottle of beer on the wall, ' +
        '1 bottle of beer.\n' +
        'Take it down and pass it around, ' +
        'no more bottles of beer on the wall.\n'
      );
    default:
      return (
        `${number} bottles of beer on the wall, ` +
        `${number} bottles of beer.\n` +
        'Take one down and pass it around, ' +
        `${number-1} ${this.container(number-1)} ` +
          'of beer on the wall.\n'
      );
  }
}

container(number) {
  if (number === 1) {
    return 'bottle';
  } else {
    return 'bottles';
  }
}
```

That horizontal refactoring required a fair amount of explanation. Here's a reminder of the key actions:

1. identified `verse 2` and `default` as the most similar cases
2. worked from left to right
3. changed `verse 2` case to replace hard coded `2` with `${number}` (twice)
4. changed `verse 2` case to replace hard coded `1` with `${number-1}`
5. identified "bottle" and "bottles" as the next difference
6. chose *container* for the name of the concept represented by this difference
7. created empty `container` method
8. changed `container` to return "bottles"
9. changed `verse default` case to send `container` in place of "bottles"
10. changed `container` to take `number` argument
11. added conditional logic to `container` to return "bottle" or "bottles" based on `number`
12. changed `verse default` case to pass `number-1` to `this.container()`
13. changed `verse 2` case to send `this.container(number-1)` in place of "bottle"

14. deleted `verse 2` case

Of these 14 steps, 11 involve changes to code. The tests run after every change, so it is trivial to fix newly-introduced flaws.

The lengthy description above may have led you to fear that working in this fashion would be unbearably slow. Take another look. As you can see, there's not much code, and with practice, writing it becomes very fast. The small amount of time lost to making incremental changes is more than recouped by avoiding lengthy and frustrating debugging sessions. This style of coding is not only fast, it's also stress-free.

This first refactoring was deliberately performed using the smallest possible steps. Once you learn to work at this level of granularity, you can later combine steps if circumstances allow. Let red be your guide. If you take a giant step and the tests begin to fail, undo and fall back to making smaller changes.

There are plenty of hard problems in programming, but this isn't one of them. Real refactoring is comfortingly predictable, and saves brainpower for more thought-provoking challenges.

3.8. Summary

When faced with the need to change code, very often the hardest decision is where to start. This chapter suggested that you be guided by the Open-Closed Principle, and so separate most changes into two broad steps. First, refactor the existing code to be open to the new requirement, next, add the new code.

Sometimes the first step, refactoring to openness, requires such a large leap that it is not obvious how to achieve it. In that case, be guided by code smells. Improve code by identifying and removing smells, and have faith that as the code improves, a path to openness will appear.

Making existing code open to a new requirement often requires identifying and naming abstractions. The Flocking Rules concentrate on turning difference into sameness, and thus are useful tools for unearthing abstractions.

This chapter introduced the six-pack requirement, and in the search for openness, identified the duplication of code in the `verse` method as the first point of attack. It then dedicated a good portion of the chapter to the task of making the `default` and `2` cases identical. However, now that you've learned how to use the flocking rules to identify abstractions, resolving the differences in the `1` and `0` cases will go much faster. So, on to Chapter 4, and more extracting of abstractions.

4. Practicing Horizontal Refactoring

The previous chapter introduced the *Flocking Rules*, which it used to remove the special case for verse 2. The chapter contained plenty of explanation about how to apply the rules, but not much new code. Fortunately, the refactorings dictated by the Flocking Rules are easier done than said, and having read the prior chapter, you are now equipped to move briskly through the other special cases.

This chapter iteratively applies the Flocking Rules to the remaining special verses, and results in a single, more abstract, template that produces every possible verse.

4.1. Replacing Difference With Sameness

The refactoring rules say to start by choosing the cases that are most alike. Now that verse 2 is being produced by the `default` branch, only three different verse templates remain. Have a look at the code below, and select the two cases on which to concentrate next.

Listing 4.1: 3 Branch Conditional

```
verse(number) {
  switch (number) {
    case 0:
      return (
        'No more bottles of beer on the wall, ' +
        'no more bottles of beer.\n' +
        'Go to the store and buy some more, ' +
        '99 bottles of beer on the wall.\n'
      );
    case 1:
      return (
        '1 bottle of beer on the wall, ' +
        '1 bottle of beer.\n' +
        'Take it down and pass it around, ' +
        'no more bottles of beer on the wall.\n'
      );
    default:
      return (
        `${number} bottles of beer on the wall, ` +
        `${number} bottles of beer.\n` +
        'Take one down and pass it around, ' +
        `${number-1} ${this.container(number-1)} ` +
          'of beer on the wall.\n'
      );
  }
}
```

The `1` case differs from the `default` case in several ways. It uses a hard coded `1` as the starting number, it takes `"it"` instead of `"one"` down, and it ends with `"no more"` instead of `number-1` bottles.

The `0` case is even more different from the `default` case . It starts with `"No more"`, it says `"Go to the store and buy some more"`, and it ends with `"99"`.

Finally, the `1` and `0` cases differ from one another in lots of ways. They both have more in common with the `default` case than each other.

Of these three verse templates, the `1` and `default` cases are most alike, so they're the next to address. Start by looking at the first lines of each:

Listing 4.2: 1 and Default 1st Phrases Differ

```
case 1:
  return (
    '1 bottle of beer on the wall, ' +
    // ...
  );
default:
  return (
    `${number} bottles of beer on the wall, ` +
    // ...
  );
```

Just as with the `2` and `default` cases, the very first character is different. Remove this difference by interpolating `number` in place of the hard-coded `1` in the `1` case, as below:

Listing 4.3: 1 and Default 1st Phrases in Progress

```
case 1:
  return (
    `${number} bottle of beer on the wall, ` +
    // ...
  );
default:
  return (
    `${number} bottles of beer on the wall, ` +
    // ...
  );
```

A similar change was made in the previous chapter, where the hard-coded `"2"` was replaced by `${number}` when combining the `2` and `default` cases. The act of substituting a variable for an explicit number is so minor that it doesn't adequately reflect the enormity of the underlying idea, but step back and consider what just happened. Replacing differing concrete values with a reference to a common variable changes *difference* into *sameness*.

The fact that the argument is known to equal `1` does not matter. This substitution is important, not because it changes the resulting value, but because it increases the level of abstraction. It is this increase in abstraction that makes things the same. Without it, you are doomed to the conditional.

The next difference is "bottle" versus "bottles." This, conveniently, is the previously identified "container" concept. Each line is changed to send the `container` message, which results in the following code:

Listing 4.4: 1 and Default 1st Phrases Identical

```
case 1:
  return (
    `${number} ${this.container(number)} ` +
      'of beer on the wall, ' +
```

```
    // ...
  );
default:
  return (
    `${number} ${this.container(number)} ` +
      'of beer on the wall, ' +
    // ...
  );
```

The first phrases are now identical.

The second phrase of each case is very similar to the first, as you can see here:

Listing 4.5: 1 and Default 2nd Phrases Differ

```
case 1:
  return (
    `${number} ${this.container(number)} ` +
      'of beer on the wall, ' +
    '1 bottle of beer.\n' +
    // ...
  );
default:
  return (
    `${number} ${this.container(number)} ` +
      'of beer on the wall, ' +
    `${number} bottles of beer.\n` +
    // ...
  );
```

The second phrase is so similar to the first that repeating the same changes will make them identical. Here's the result:

Listing 4.6: 1 and Default 2nd Phrases Identical

```
case 1:
  return (
    `${number} ${this.container(number)} ` +
      'of beer on the wall, ' +
    `${number} ${this.container(number)} of beer.\n` +
    // ...
  );
default:
  return (
    `${number} ${this.container(number)} ` +
      'of beer on the wall, ' +
    `${number} ${this.container(number)} of beer.\n` +
    // ...
  );
```

After the above changes, the first two phrases of the `1` and `default` cases are identical.

4.2. Equivocating About Names

The name `container` feels right. It was fairly easy to find, in part because the underlying concept is so obvious. Once you realize that you're trying to name a category that contains bottles, juice boxes, and carafes, `container` naturally follows.

However, when concepts are fuzzier, finding a good name can be much harder. This section deals with just such a concept, and offers several suggestions for what to do when you can't find a good name.

Now that phrases one and two are the same, it's time to consider phrase three. Here's a reminder of that code:

Listing 4.7: 1 and Default 3rd Phrases Differ

```
case 1:
  return (
    // ...
    'Take it down and pass it around, ' +
    // ...
  );
default:
  return (
    // ...
    'Take one down and pass it around, ' +
    // ...
  );
```

The difference above is that `"it"` matches up with `"one"`.

If all verse variants are alike in an underlying, more abstract, way, then "it" and "one" must represent a smaller abstraction within that larger one. Once you name this concept, you can create a method with that name, and then make these lines alike by sending a message in place of the different strings.

If the previous paragraph gave you a sense of deja-vu, that's understandable. This is exactly how "bottle" and "bottles" became `container`, and how every future difference will be resolved. The process may seem too straightforward to believe, but the mechanism truly is this humble. The rules of refactoring are simple, but when followed, precise and complex behavior emerges.

The challenge, as always, is identifying the current concept and coming up with a good name. The words `"it"` and `"one"` are so innately generic that naming the underlying concept is particularly tough. Names should neither be too general nor too specific. For example, `thing` is too broad, and `itOrOne` too narrow.

If you were to ask your customer to name this category, they would likely shrug and call it `pronoun`. If you object to `pronoun` on the grounds that it's overly general, and insist that they give the category a more specific name, they might come up with something like `thingDrunk`.

Although `pronoun` *does* feel a bit too general, `thingDrunk` is just about unbearable. Neither feels perfect. This situation, unfortunately, is all too common. When the perfect name for a concept is elusive, there are three strategies for moving forward.

Some folks allot themselves five to ten minutes to ponder (usually with thesaurus in hand), and then use the best name they can come up with during that interval. Their rationale is that the name they choose *might* be good enough, and if they later discover it's not, they can always improve it. These folks have the advantage of working with code that contains names that are at

least *somewhat* useful, even if not entirely correct, but must live with the possibility that a good-enough name will persist, even after a better name becomes obvious to the humans involved.

Other folks find it more cost effective to instantly choose a meaningless name like `foo` or `namethis`. This strategy allows them to move forward quickly, and (one hopes) insures that the name *will* get improved later. These folks believe strongly in the "You'll never know less than you know right now" dictum,[12] and fully expect that a better name will occur as they work on the code. They believe there's no point in wasting time thinking about it now, when the name will be obvious later.

Finally, instead of following one of two previous strategies by yourself, you can simply ask someone else for help. Within any group of programmers, there's often someone who's good at naming things. If your group has such a person, you know who they are. Appoint them the "name guru," and leverage their strengths when you need a name.

In the case of `"it"` or `"one"` here in "99 Bottles," `pronoun` is good enough for now. If something better occurs later, you can always improve the name.

The procedure to turn `"it"` and `"one"` into `pronoun` is identical to the one that transformed "bottle" and "bottles" into `container`. Having previously practiced, this next refactoring will go quickly. The following examples step through the transitions. Remember to run the tests after each change.

First, define an empty `pronoun` method.

Listing 4.8: Empty Pronoun Method

```
pronoun() {
}
```

Alter `pronoun` to return "one," which is the value from the `default` branch.

Listing 4.9: Sparse Pronoun Method

```
pronoun() {
  return 'one';
}
```

Alter the `default` branch to call `pronoun` in place of "one":

Listing 4.10: Send Pronoun in Default Branch

```
case 1:
  return (
    // ...
    'Take it down and pass it around, ' +
    // ...
  );
default:
  return (
    // ...
    `Take ${this.pronoun()} down and pass it around, ` +
    // ...
  );
```

Add an argument to `pronoun`.

Listing 4.11: Pronoun With Argument

```
pronoun(number) {
  return 'one';
}
```

Alter `pronoun` to be open to the `1` case.

Listing 4.12: Pronoun With Conditional

```
pronoun(number) {
  if (number === 1) {
    return 'it';
  } else {
    return 'one';
  }
}
```

Alter the `default` case to pass the `number` argument to `pronoun` (line 10).

Listing 4.13: Passing an Argument to Pronoun

```
case 1:
  return (
    // ...
    'Take it down and pass it around, ' +
    // ...
  );
default:
  return (
    // ...
    `Take ${this.pronoun(number)} down and pass it around, ` +
    // ...
  );
```

Alter the `1` case to send `this.pronoun(number)` in place of "it" (line 4).

Listing 4.14: 1 and Default Cases Send Pronoun

```
case 1:
  return (
    // ...
    `Take ${this.pronoun(number)} down and pass it around, ` +
    // ...
  );
default:
  return (
    // ...
    `Take ${this.pronoun(number)} down and pass it around, ` +
    // ...
  );
```

The refactoring steps that added `pronoun` were exactly like those used to add `container`. In each case, differing strings were replaced by a common message send. Just as the `container` abstraction replaced the "bottle" and "bottles" strings, the `pronoun` abstraction replaced "it" and "one."

This completes the addition of the `pronoun` method, and makes phrase three of the `1` and `default` cases identical. It's time to move on to the fourth and final phrase.

4.3. Deriving Names From Responsibilities

Although `pronoun` may feel too general, the concept it represents is clear. If you had to describe the underlying idea, you might say something like "The `pronoun` message returns the word that is used in place of the noun 'bottles,' following the word 'Take,' in phrase 3 of each verse." Pedantic as that explanation is, it's entirely correct. That is `pronoun`'s responsibility.

The difficulty naming `pronoun` illustrates how hard it can be to choose a name, even when you understand the concept. Imagine, then, how impossible it is to choose a name when you don't. This next section addresses the fourth and final phrase, and takes on the challenge of naming a concept that is much less clear-cut.

The first difference in phrase four looks a bit, well, different, but regardless, it can be resolved using the technique you've been using. The trick to getting this next refactoring right is to trust the rules, and to write only the code that they require.

Here's a look at phrase four of the `1` and `default` cases:

Listing 4.15: 1 and Default 4th Phrases Differ

```
case 1:
  return (
    // ...
    'no more bottles of beer on the wall.\n'
  );
default:
  return (
    // ...
    `${number-1} ${this.container(number-1)} ` +
      'of beer on the wall.\n'
  );
```

Look at the code above and identify the differences. It might help to first decide what is *not* a difference. Both phases end with "of beer on the wall," so that part is clearly the same. If you disregard that sameness, you're left with:

```
"no more bottles"
```

which matches up against:

```
`${number-1} ${this.container(number-1)}`
```

If it's not clear how to proceed, look for a way to make the lines more alike (even if not yet identical), using code you've already written. Remember that the goal is to locate the next small difference, not the next *clump* of differences.

Notice the word "bottles" in the `1` case. The abstraction that underlies "bottles" has long since been identified. It's encapsulated in the `container` method, which is already being used by the `default` case.

If `"bottles"` is actually the same as `this.container(number-1)`, then "bottles" is not part of the next difference. This means that the current difference is that:

```
"no more"
```

goes with:

```
`${number-1}`
```

Until now, the differences between phrases have both been strings. Here, for the first time, one is a string and the other is interpolated code. However, it doesn't matter what form the difference takes. If each verse variant reflects a more general verse abstraction, then the differences between the variants must represent smaller concepts within that larger abstraction. Again, you can resolve this difference by following the pattern you learned from `container` and `pronoun`. Name the concept, create the method, and replace the difference with a common message send.

To help you name the new concept, remember the "what would the column header be?" technique. The following table shows a sampling of numbers and associated values:

Table 4.1: Number to XXX Column Header

Number	*XXX?*
99	`'99'`
50	`'50'`
1	`'1'`
0	`'no more'`

In the table above, the left column contains a number between 99 and 0, and the right holds the string to be sung in its place. Most times the value on the right is the direct string representation of the number on the left, so `99` becomes `"99"`, and `50` becomes `"50"`, etc. The exception is `0`, which becomes, not `"0"` as you might expect, but `"no more"`.

Phrase four is the final phrase of the song where the number gets decremented, and so the argument is always `number-1`. It's tempting, therefore, to think of "no more" and `${number-1}` as representing the number of bottles that *remain* once a verse is complete.

You could indeed name this concept "remainder," and proceed with the refactoring. However, in the interest of saving a bit of pain, take a brief peek forward. You'll soon be considering the `0` case, which says:

```
No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99 bottles of beer on the wall.
```

Notice that the `0` case *starts* with `"No more"`, just as the `1` case ends with `"no more"`. The way the song works is that whenever there are `0` bottles, you sing "no more," capitalized appropriately.

When `"No more"` comes at the beginning of the song, it's clearly not the remainder. This means that if `"no more"` and `"No more"` represent the same idea, then `remainder` isn't a good name for the underlying concept.

If you reconsider the above table, the right side is actually the *name*, or *description*, or perhaps *quantity* of bottles being sung about. It is the string to be sung in the place of any number. While not perfect, `quantity` at least *attempts* to indicate the responsibility on the method you plan to create, and so is a reasonable first attempt at a name.

Before implementing `quantity`, consider what would have happened had you named this concept `remainder`. After finishing the `1` case, you'd have advanced to the `0` case and discovered that it started with `"No more"`. This would have caused you to reconsider `remainder`. You'd likely have reverted the refactoring to this point, and re-started your search for a name.

Real life is like this, where you make the best decision you can in the moment, and reassess when you know more. Had you been doing this refactoring alone, you might well have gone down the `remainder` path, and suffered the eventual reversal. As there's enough pain in real life, here you've been left to imagine it.

Do not take this as a general license to think far ahead. While you are allowed to use common sense, it's usually best to stay horizontal and concentrate on the current goal. When creating an abstraction, first describe its responsibility *as you understand it at this moment*, then choose a name which reflects that responsibility. The effort you put into selecting good names right now pays off by making it easier to recognize perfect names later.

4.4. Choosing Meaningful Defaults

The previous few refactorings used the technique of temporarily allowing an argument to assume the default value of `undefined`. Helpful as `undefined` is, however, it won't work in every case. Sometimes circumstances conspire to force you to use a real value as a default during these refactorings. This next section delves into such a case.

Remember that the difference currently being addressed is:

```
"no more"
```

which goes with:

```
`${number-1}`
```

The underlying concept is `quantity`. To remove this difference, first add the `quantity` method:

Listing 4.16: Initial Quantity Method

```
quantity() {
}
```

The next step is to change this method to return one of the two differences. Until now, you've chosen to return the value from the `default` branch first. But in this case, the `default` branch contains interpolated code that references `number`. Therefore, you can't copy the `default` branch difference into `quantity` unless you first alter `quantity` to take `number` as an argument.

The `1` branch contains the string "no more," which is a simpler difference. That simplicity makes this a good place to explore what happens if you switch up and return the non-else value first.

Because of this change in tactics, proceeding *exactly* as you've done previously will eventually lead to an error. It's instructive to watch this happen, as shown in the following code.

Begin by returning the value from the `1` case:

Listing 4.17: Quantity Method First Return

```
quantity() {
  return 'no more';
}
```

Send `quantity` in place of `"no more"` in the `1` case:

Listing 4.18: Quantity Message First Send

```
case 1:
  return (
    // ...
    `${this.quantity()} bottles of beer on the wall.\n`
  );
```

Add the `number` argument in `quantity`:

Listing 4.19: Number Argument

```
quantity(number) {
  return 'no more';
}
```

If you're concerned about the `undefined` default above, your Spidey-sense[13] is working. Yes, everything will go terribly wrong in a minute, but until then, cast your worries aside and charge forward.

The next step is to alter `quantity` to be open to the `default` case. Remember that you're working on the final phrase of verse 1, and that the value of the passed argument will be `number-1`, or `0`. If `number` is `0`, the condition should return `"no more"`; otherwise, it should return the number.

Here's the `quantity` method, altered to contain that new conditional:

Listing 4.20: Quantity With Conditional

```
quantity(number) {
  if (number === 0) {
    return 'no more';
  } else {
    return number;
  }
}
```

If you now have additional concerns about this code, hang in there. A number of errors will arise, but they will soon get resolved.

At this point in each of the previous refactorings the tests passed, but in this case, not so. The tests are now failing with:

```
-Take it down and pass it around, no more bottles of beer on the wall.
+Take it down and pass it around, undefined bottles of beer on the wall.
```

Have a look at the `switch` statement below. Examine line 4 and try to explain what went wrong.

Listing 4.21: Using the Number Default From the 1 Case

```
case 1:
  return (
    // ...
    `${this.quantity()} bottles of beer on the wall.\n`
  );
default:
  return (
    // ...
    `${number-1} ${this.container(number-1)} ` +
      'of beer on the wall.\n'
  );
```

This failure occurs because line 4 above calls `quantity` without passing an argument. Upon invocation, the `quantity` method sets `number` to `undefined`, which sends execution to the false branch of its conditional. The false branch obediently returns `number`, which unfortunately still contains `undefined`. This result then gets interpolated back into the verse. Thus, "`undefined bottles of beer`".

The reason the `undefined` default worked in previous situations was because in those cases you *wanted* to execute the false branch. However, now you need the true branch, and therefore require a much more specific default.

The tests are failing, and the rules dictate that you must undo and return to green. Fortunately, this takes just one undo, which reverts `quantity` to the following:

Listing 4.22: Number Argument Reprise

```
quantity(number) {
  return 'no more';
}
```

Allowing missing arguments to take on the value of `undefined` can be handy, but you can use this technique only if you begin these refactorings by returning the difference from the `default` branch. While it's perfectly acceptable to begin by returning "no more" (the non-`default`

difference), doing so means that you have to think more carefully about the argument's default value. So use `undefined` arguments thoughtfully.

In this case, the default that will drive execution to the correct branch is `0`, as shown below:

Listing 4.23: Number Argument Defaults to 0

```
quantity(number=0) {
  return 'no more';
}
```

Now that the default is correct, the conditional can be re-added to `quantity` as follows:

Listing 4.24: Default Takes the True Branch

```
quantity(number=0) {
  if (number === 0) {
    return 'no more';
  } else {
    return number;
  }
}
```

Although nothing about the conditional has changed since the last attempt, the default is now correct, so the tests pass.

Taking the default caused the `true` branch to execute. Now it's time to ensure that passing an argument does the same. Line 4 below has been changed to pass `number-1` to `quantity`:

Listing 4.25: 1 Case Passes an Argument

```
case 1:
  return (
    // ...
    `${this.quantity(number-1)} bottles ` +
      'of beer on the wall.\n'
  );
default:
  return (
    // ...
    `${number-1} ${this.container(number-1)} ` +
      'of beer on the wall.\n'
  );
```

The tests still pass. The next step is to use `quantity` in the `default` case, as shown on line 10 below:

Listing 4.26: Default Case Sends Quantity

```
case 1:
  return (
    // ...
    `${this.quantity(number-1)} bottles ` +
      'of beer on the wall.\n'
  );
default:
  return (
    // ...
    `${this.quantity(number-1)} ` +
      `${this.container(number-1)} ` +
```

```
12 |       'of beer on the wall.\n'
13 |   );
```

At this point `quantity` is fully implemented. The default is no longer needed, and can be removed. The final method is shown below:

Listing 4.27: Quantity Method

```
1 |quantity(number) {
2 |  if (number === 0) {
3 |    return 'no more';
4 |  } else {
5 |    return number;
6 |  }
7 |}
```

After resolving `quantity`, one minor difference remains between the `1` and `default` cases. The final phrase of the `1` case says `"bottles"` (line 4 below) whereas in that place the `default` case sends `this.container(number-1)`.

Listing 4.28: 1 and Default Cases More Alike

```
1 |case 1:
2 |  return (
3 |    // ...
4 |    `${this.quantity(number-1)} bottles ` +
5 |      'of beer on the wall.\n'
6 |  );
7 |default:
8 |  return (
9 |    // ...
10|    `${this.quantity(number-1)} ` +
11|      `${this.container(number-1)} ` +
12|      'of beer on the wall.\n'
13|  );
```

This difference can be resolved by sending the well-known `container` message in place of the word `"bottles"`. After this change, the `1` and `default` cases are identical, as shown in their full glory below:

Listing 4.29: 1 and Default Cases Identical

```
1 |verse(number) {
2 |  switch (number) {
3 |    // ...
4 |    case 1:
5 |      return (
6 |        `${number} ${this.container(number)} ` +
7 |          'of beer on the wall, ' +
8 |        `${number} ${this.container(number)} of beer.\n` +
9 |        `Take ${this.pronoun(number)} down and pass it around, ` +
10|        `${this.quantity(number-1)} ` +
11|          `${this.container(number-1)} ` +
12|          'of beer on the wall.\n'
13|      );
14|    default:
15|      return (
16|        `${number} ${this.container(number)} ` +
17|          'of beer on the wall, ' +
18|        `${number} ${this.container(number)} of beer.\n` +
19|        `Take ${this.pronoun(number)} down and pass it around, ` +
```

```
20 |         `${this.quantity(number-1)} ` +
21 |           `${this.container(number-1)} ` +
22 |           'of beer on the wall.\n'
23 |       );
24 |   }
25 |}
```

This completely resolves the `1` case, which can now be deleted.

Two new concepts have been identified, `pronoun` and `quantity`. Although the refactoring that created `quantity` obediently follows the Flocking Rules, the *order* in which code is written differs slightly from that of previous method extractions. The earlier examples began by returning the value from the `default` branch of the `switch` statement, but the `quantity` method differs in that it initially returns the value from the `1`, or non-`default` case.

All of these refactorings extract a method. Because this is done in very small steps, the extracted methods start out simple and then gradually become more complicated. One of the complications is that each method changes to take a parameter. In order to keep the tests running green during the transition to taking a parameter, the parameter has to be assigned a default. The default is temporary, and it is meant to be deleted when the transition is complete.

When the `default` branch is implemented first, `undefined` can always be used for the default. If the non-`default` branch is implemented first, the default has to be set to something that actually meets the condition and so makes the `true` branch execute. Therefore, implementing the non-`default` branch first places a slightly greater burden on you. You have to use a specific, real value for the default, and then remember to remove the default once the transition is complete.

4.5. Seeking Stable Landing Points

At this point, the `2` and `1` cases have been removed, and three new concepts, `quantity`, `pronoun` and `container`, have been identified. To save you from having to remember, the listing below repeats the code for these concepts:

Listing 4.30: Three Abstracted Concepts

```
1 |quantity(number) {
2 |  if (number === 0) {
3 |    return 'no more';
4 |  } else {
5 |    return number;
6 |  }
7 |}
8 |
9 |container(number) {
10|  if (number === 1) {
11|    return 'bottle';
12|  } else {
13|    return 'bottles';
14|  }
15|}
16|
17|pronoun(number) {
18|  if (number === 1) {
19|    return 'it';
```

```
  } else {
    return 'one';
  }
}
```

Notice the similarities in the above methods. Each has a single responsibility. They are identical in shape. All take the same argument. Each contains a conditional and that conditional tests the argument against a specific value; it checks to see if the argument is *equal* to something, as opposed to greater or less than something. These methods are incredibly consistent, and *this did not happen by accident*—it's a direct result of the refactoring rules. The rules lead to consistent code, and consistency matters deeply.

First, it makes code easy to understand. Code is read many more times than it is written, so anything that increases understandability lowers costs. Next, and just as important, consistent code enables *future* refactorings.

Imagine yourself a child, traipsing down a stream, hopping from rock to rock. Some rocks are broad and flat and dry, others are mossy and wobbly and slick. Imagine also that you are not allowed to return home wet.

The dry rocks are stable landing points on which you can safely rest, planning your next move. The wet rocks are risky interludes that good sense suggests you traverse as quickly as possible.

Rearranging code is like rock hopping down a stream. If you follow the rules of refactoring, you'll quickly pass over the slippery places, and arrive at stable, consistent resting points. Changing code willy-nilly, however, can lead to surprising and unexpected baths.

The consistency in the code above *enables* the next refactoring. For now you must take this assertion on faith, but that faith will be rewarded in future chapters.

4.6. Obeying the Liskov Substitution Principle

Now, back to the horizontal refactoring. This chapter started with a three-branch `switch` statement. One case (the `1` case) has been removed, leaving the `0` and `default` cases still to be resolved. Here's a reminder of the current state of the code:

Listing 4.31: 0 and Default Cases Differ

```
verse(number) {
  switch (number) {
    case 0:
      return (
        'No more bottles of beer on the wall, ' +
        'no more bottles of beer.\n' +
        'Go to the store and buy some more, ' +
        '99 bottles of beer on the wall.\n'
      );
    default:
      return (
        `${number} ${this.container(number)} ` +
          'of beer on the wall, ' +
        `${number} ${this.container(number)} of beer.\n` +
        `Take ${this.pronoun(number)} down and pass it around, ` +
        `${this.quantity(number-1)} ` +
```

```
17 |           `${this.container(number-1)} ` +
18 |           'of beer on the wall.\n'
19 |       );
20 |   }
21 |}
```

Begin this next refactoring by focusing on lines 5 and 12-13 above, the first phrases of the two remaining cases. Looking for the smallest difference, both lines end with `"of beer on the wall, "`, so this is a similarity that can be ignored. The `container` method is used on line 12 in the `default` case. The word `"bottles"` on line 5 is a `container`, so "bottles" is not part of the next difference.

The remaining difference is at the very beginning of lines 5 and 12, where:

```
"No more"
```

goes with:

```
`${number}`
```

This feels like the `quantity` concept, but as it stands, that method won't work to resolve this difference. If you were to change line 5 to call `${this.quantity(number)}` in place of `"No more"`, you'd get back an all lowercase "no more," and the tests would fail.

This is a conundrum. The lowercase variant of "no more" is required by verse 1, and now verse 0 needs the same two words, except capitalized as the start of a sentence. The underlying concept is the same in both cases ("no more" is to be sung when the number of bottles is 0), but it gets expressed in slightly different ways, depending on where it falls in the song.

These words are one thing, and whether they need to be capitalized is quite another. Perhaps knowledge of the words belongs in one place, and knowledge of the capitalization requirements belongs in another.

If that's the case, capitalization can reasonably happen here in the `switch` statement. Replace `"No more"` with `${this.quantity(number)}`, and capitalize the result, as on line 3 below:

Listing 4.32: Quantity Capitalized in 0 Case

```
1 |case 0:
2 |  return (
3 |    `${capitalize(this.quantity(number))} ` +
4 |      'bottles of beer on the wall, ' +
5 |    // ...
6 |  );
7 |default:
8 |  return (
9 |    `${number} ${this.container(number)} ` +
10|      'of beer on the wall, ' +
11|    // ...
12|  );
```

The above change follows the strategy of gradually making things more alike in hopes that it will then become clear how to make them identical. When nibbling away at the problem, you don't

have to understand everything before you can do anything. Taking care of the small things often cuts the big ones down to size.

Having made the above change, the evident next step is to make a similar one in the `default` case, shown on line 9 below:

Listing 4.33: Quantity Capitalized in Default Case

```
1 case 0:
2   return (
3     `${capitalize(this.quantity(number))} ` +
4       'bottles of beer on the wall, ' +
5     // ...
6   );
7 default:
8   return (
9     `${capitalize(this.quantity(number))} ` +
10      `${this.container(number)} ` +
11      'of beer on the wall, ' +
12    // ...
13  );
```

Despite seeming reasonable, that change makes the tests fail with:

```
TypeError: string.charAt is not a function
```

Because you're working in such small steps, you *know* that the previous change caused this error. Have a look at the following code and see if you can figure out what's wrong:

Listing 4.34: Quantity Method Reprise

```
1 quantity(number) {
2   if (number === 0) {
3     return 'no more';
4   } else {
5     return number;
6   }
7 }
```

The most recent change invokes `quantity` with a non-zero argument. This causes execution to proceed to the false branch. The true branch returns a string, but the false branch returns the argument that was passed, which is a number. The implementation of `capitalize` works with a string argument, but not with a number; thus this error.

You may be itching to fix this error by making a change in the `quantity` method, but it's instructive to try attacking it here in `verse`. Go ahead and remove the error by converting the result into a string before calling `capitalize`. Line 9 below inserts `toString` into the method chain:

Listing 4.35: Default Branch Converts Result

```
1 case 0:
2   return (
3     `${capitalize(this.quantity(number))} ` +
4       'bottles of beer on the wall, ' +
5     // ...
6   );
```

```
default:
  return (
    `${capitalize(this.quantity(number).toString())} ` +
      `${this.container(number)} ` +
      'of beer on the wall, ' +
    // ...
  );
```

The above change fixes the failing test, but introduces a new difference between the phrases. To remove this difference, you must also insert `toString` into the `0` case, as on line 3 below:

Listing 4.36: Both Branches Convert Result

```
case 0:
  return (
    `${capitalize(this.quantity(number).toString())} ` +
      'bottles of beer on the wall, ' +
    // ...
  );
default:
  return (
    `${capitalize(this.quantity(number).toString())} ` +
      `${this.container(number)} ` +
      'of beer on the wall, ' +
    // ...
  );
```

Now that the difference is resolved and the tests are running, step back and consider this solution. The root of the problem is that `quantity` returns things that conform to different APIs. Senders of `quantity` expect the return to work with `capitalize`, yet `quantity` doesn't always oblige; it sometimes returns a "capitalizable," but other times does not. This inconsistency of return types forces the *sender* of the message to know more than it should.

The `verse` method above knows that it cannot trust `quantity` to return something that `capitalize` understands. The `verse` method knows that strings *do* work with `capitalize`. It knows that any object can be converted to a string by sending `toString`. Therefore, it knows that it can convert any object into something that `capitalize` understands by sending it the `toString` message.

Every piece of knowledge is a dependency, and the way that `quantity` is written requires `verse` to know too many things. If `quantity` were more trustworthy, `verse` could know less.

The idea of reducing the number of dependencies imposed upon message senders by requiring that receivers return trustworthy objects is a generalization of the Liskov Substitution Principle. The official definition of Liskov says that "subtypes must be substitutable for their supertypes." This principle was originally postulated in terms of types and subtypes, but you can think of it in terms of classes and subclasses.

Liskov, in plain terms, requires that objects be what they promise they are. When using inheritance, you must be able to freely substitute an instance of a subclass for an instance of its superclass. Subclasses, by definition, are all that their superclasses are, *plus more*, so this substitution should always work.

The Liskov Substitution Principle also applies to duck types. When relying on duck types, every object that asserts that it plays the duck's role must completely implement the duck's API. Duck types should be substitutable for one another.

Liskov prohibits you from doing anything that would force the sender of a message to test the returned result in order to know how to behave. Receivers have a contract with senders, and despite the implicit nature of this contract in dynamically typed, object-oriented languages, it must be fulfilled.

Liskov violations force message senders to have knowledge of the various return types, and to either treat them differently, or convert them into something consistent. In the `quantity` method above, one of the returns honored the "capitalizable" contract and one did not. An inconsistency like this very often forces the sender to implement a conditional to identify and fix the errant return. In this case, all JavaScript objects understand `toString`, so it was programmatically convenient to blithely convert every return into a string, even those that already were. This unconditional conversion avoids checking to see which objects need to be sent `toString`, but adds the overhead of sending `toString` to every object, even if it's already a string.

The sender's entire burden is removed if the receiver honors the contract and provides a consistent return. Instead of forcing the `verse` method to solve this problem, `quantity` should return a trustworthy object.

This is easily accomplished by doing the conversion in the `quantity` method, as shown on line 5 below:

Listing 4.37: Quantity Obeys Liskov

```
1 quantity(number) {
2   if (number === 0) {
3     return 'no more';
4   } else {
5     return number.toString();
6   }
7 }
```

Now that `quantity` always returns a "capitalizable," you can pretend that the `toString` dependency never existed in verse, which returns the code to the state shown here:

Listing 4.38: Quantity Is Trustworthy

```
 1 case 0:
 2   return (
 3     `${capitalize(this.quantity(number))} ` +
 4       'bottles of beer on the wall, ' +
 5     // ...
 6   );
 7 default:
 8   return (
 9     `${capitalize(this.quantity(number))} ` +
10       `${this.container(number)} ` +
11       'of beer on the wall, ' +
12     // ...
13   );
```

Having altered `quantity` to make it usable in all cases, the remaining difference in the first phrase is the word "bottles." This is easily resolved by sending `container` in its place:

Listing 4.39: 0 Case Sends Container

```
case 0:
  return (
    `${capitalize(this.quantity(number))} ` +
      `${this.container(number)} ` +
      'of beer on the wall, ' +
    // ...
  );
default:
  return (
    `${capitalize(this.quantity(number))} ` +
      `${this.container(number)} ` +
      'of beer on the wall, ' +
    // ...
  );
```

After that change, the first phrases of the `0` and `default` cases are identical.

4.7. Taking Bigger Steps

You've now turned small differences into message sends several times, and have likely noticed the similarity between the steps taken and the resulting code. So far, the extracted methods all have the same general shape, and are invoked in the same way.

Differences remain. However, it's beginning to feel like there's a common refactoring pattern, and one might reasonably theorize that future differences will be resolved following the same process that was used in the past. If this theory is correct, it makes sense to speed up the next refactoring by combining several steps into a single change.

The first phrase of the `0` and `default` cases are identical, so it's time to examine the second. It's repeated below:

Listing 4.40: 0 and Default 2nd Phrases Differ

```
case 0:
  return (
    // ...
    'no more bottles of beer.\n' +
    // ...
  );
default:
  return (
    // ...
    `${number} ${this.container(number)} of beer.\n` +
    // ...
  );
```

The above differences reflect the `quantity` and `container` concepts, which have long since been identified. Resolve them by changing the code as follows:

Listing 4.41: 2nd Phrases Send Quantity and Container

```
case 0:
  return (
    // ...
    `${this.quantity(number)} ` +
      `${this.container(number)} ` +
      'of beer.\n' +
    // ...
  );
default:
  return (
    // ...
    `${this.quantity(number)} ` +
      `${this.container(number)} ` +
      'of beer.\n' +
    // ...
  );
```

Now that phrases 1 and 2 are identical, here's a look at the whole `verse` method. Consider the code, and identify the next difference:

Listing 4.42: Phrases 1 and 2 Are Identical

```
verse(number) {
  switch (number) {
    case 0:
      return (
        `${capitalize(this.quantity(number))} ` +
          `${this.container(number)} ` +
          'of beer on the wall, ' +
        `${this.quantity(number)} ` +
          `${this.container(number)} ` +
          'of beer.\n' +
        'Go to the store and buy some more, ' +
        '99 bottles of beer on the wall.\n'
      );
    default:
      return (
        `${capitalize(this.quantity(number))} ` +
          `${this.container(number)} ` +
          'of beer on the wall, ' +
        `${this.quantity(number)} ` +
          `${this.container(number)} ` +
          'of beer.\n' +
        `Take ${this.pronoun(number)} down and pass it around, ` +
        `${this.quantity(number-1)} ` +
          `${this.container(number-1)} ` +
          'of beer on the wall.\n'
      );
  }
}
```

To locate the next difference, it can again be helpful to scan the verse backwards from the end. Both variants end with "of beer on the wall." On line 12, phrase 4 of case `0` begins with `"99"` followed by `"bottles"`. These seem to match up with `quantity` and `container` on lines 23-24. Ignore this fourth phrase for now and turn your thoughts to phrase 3, isolated below:

Listing 4.43: 0 and Default 3rd Phrases Differ

```
case 0:
  return (
    // ...
    'Go to the store and buy some more, ' +
```

```
    // ...
  );
default:
  return (
    // ...
    `Take ${this.pronoun(number)} down and pass it around, ` +
    // ...
  );
```

The only thing the above lines have in common is the trailing `", "`, which means that everything up to that point is a difference. If the `0` and `default` verse variants reflect a common verse abstraction, this difference must represent a smaller concept within that larger abstraction. It doesn't matter how long these strings are—their presence here in opposition means they reflect a single concept.

You must name the concept, create a method to represent it, and then replace this difference with a message send. The first step is therefore to name the category in which these two phrases are concrete examples.

This part of the song is about what happens as a result of the current number of beers. If beers exist, you drink one. If not, you go shopping. These lines describe the *action* to take, so that's a good name for this concept.

Until now, you've been doing this refactoring in the smallest possible steps. As a reminder, those steps are:

- Define a method for the concept.
- Alter it to return one of the differences.
- Replace that difference with a message send.
- Add the `number` argument to the new method, with appropriate default.
- Implement the conditional.
- Pass the `number` argument from the current sender.
- Send the message from the other branch, this time including the `number` argument.
- Clean up.

You may have noticed that the method you create during this refactoring contains code that exactly mirrors the shape of the original `switch` statement. Once this becomes apparent, it makes sense to begin plucking out methods in a single step, as shown below:

Listing 4.44: Leap Into Action

```
action(number) {
  if (number === 0) {
    return 'Go to the store and buy some more';
  } else {
    return (
      `Take ${this.pronoun(number)} down and pass it around`
    );
```

```
8 |   }
9 |}
```

This new `action` method contains a conditional that reflects the `switch` statement from whence it came. Just as the original `switch` statement switched on `number`, the new `action` method takes a `number` argument, and uses its value to choose what to return. The `true` and `false` branches of the new conditional contain code extracted directly from the `0` and `default` branches of the `switch` statement.

Once `action` exists, the original phrases can be made identical by replacing their differences with a common message send. This results in the following code:

Listing 4.45: 3rd Phrases Send Action

```
1 |case 0:
2 |  return (
3 |    // ...
4 |    `${this.action(number)}, ` +
5 |    // ...
6 |  );
7 |default:
8 |  return (
9 |    // ...
10|    `${this.action(number)}, ` +
11|    // ...
12|  );
```

The previous chapter showed an example where the entire container method was created at once. That was held up as an example of what *not* to do. The `action` method above looks a lot like that original `container` method, and it may seem as if you are now being given permission to act in a way that was previously prohibited.

However, there *is* a difference. Back when the original `container` method was first introduced, you had not yet learned how to create it using small steps. Since that time, you've practiced the Flocking Rules, refactoring bit by bit, and on several occasions have seen differences from two branches of the `switch` statement turn into a single conditional. Now that you recognize the pattern, and know how to make this change using small steps, it makes sense to start writing larger chunks of code.

However, if you take bigger steps and the tests begin to fail, there's something about the problem that you don't understand. If this happens, don't push forward and refactor under red. Undo, return to green, and make incremental changes until you regain clarity.

4.8. Discovering Deeper Abstractions

So far the `container`, `pronoun`, `quantity`, and `action` concepts have been identified, and methods have been extracted to be responsible for each. This horizontal refactoring to remove the `switch` statement is almost complete. This next section resolves the final difference, and in so doing illustrates the deep power of the Flocking Rules to unearth unanticipated abstractions.

The remaining differences are in the fourth phrases of the `0` and `default` cases, shown on lines 12 and 23-25 below:

Listing 4.46: Phrases 1, 2, and 3 Are Identical

```
verse(number) {
  switch (number) {
    case 0:
      return (
        `${capitalize(this.quantity(number))} ` +
          `${this.container(number)} ` +
          'of beer on the wall, ' +
        `${this.quantity(number)} ` +
          `${this.container(number)} ` +
          'of beer.\n' +
        `${this.action(number)}, ` +
        '99 bottles of beer on the wall.\n'
      );
    default:
      return (
        `${capitalize(this.quantity(number))} ` +
          `${this.container(number)} ` +
          'of beer on the wall, ' +
        `${this.quantity(number)} ` +
          `${this.container(number)} ` +
          'of beer.\n' +
        `${this.action(number)}, ` +
        `${this.quantity(number-1)} ` +
          `${this.container(number-1)} ` +
          'of beer on the wall.\n'
      );
  }
}
```

The trailing `"of beer on the wall"` in the lines above is a sameness, and the word `"bottles"` in line 12 is an example of the `container` abstraction, which is already used in this place in line 24. If you ignore these for now, the remaining difference is that:

```
"99"
```

seems to be set against:

```
`${this.quantity(number-1)}`
```

This may lead you to conclude that `"99"` is a third example of the `quantity` abstraction. If so, this implies that you should alter `quantity` to sometimes return `"99"`. The resulting method would look like this:

Listing 4.47: Quantity Overreaches to Handle 99

```
quantity(number) {
  switch (number) {
    case -1:
      return '99';
    case 0:
      return 'no more';
    default:
      return number.toString();
  }
}
```

If you made the alteration shown above, and then replaced `"99"` with `"${this.quantity(number-1)}"`, the tests would continue to pass. However, just because the

tests pass doesn't mean that the abstraction is correct. There's something deeply wrong with this solution, and there are many clues to the problem.

The first clue is that the above change gives `quantity` a different shape than that of the other extracted methods. Here's a reminder of how the methods looked before this alteration:

Listing 4.48: Consistent Abstractions

```
quantity(number) {
  if (number === 0) {
    return 'no more';
  } else {
    return number.toString();
  }
}

container(number) {
  if (number === 1) {
    return 'bottle';
  } else {
    return 'bottles';
  }
}

action(number) {
  if (number === 0) {
    return 'Go to the store and buy some more';
  } else {
    return (
      `Take ${this.pronoun(number)} down and pass it around`
    );
  }
}

pronoun(number) {
  if (number === 1) {
    return 'it';
  } else {
    return 'one';
  }
}
```

The proposed change alters `quantity` such that:

- its conditional has 3 branches instead of 2
- it sometimes checks `-1`, which is an invalid number of beers

These inconsistencies don't *guarantee* that something is wrong, but they should certainly motivate you to think more deeply about the underlying abstraction.

Ask yourself these two questions:

1. What is the responsibility of the `quantity` method?
2. Is there a way to make the fourth phrases more alike, even if not yet identical?

First, consider responsibilities. The `quantity` concept is responsible for knowing what to sing in the place of a number. If there are `50` beers, the quantity is `"50"`, if `5` beers, `"5"`, and if `0` beers, `"no more"`. This concept represents the mapping between the value of a number and the string that gets sung.

As the song progresses, the verse number gets decremented. It's been a while since you've seen them, so here's a reminder of the `song` and `verses` methods:

Listing 4.49: Song and Verses Reprise

```
1 song() {
2   return this.verses(99, 0);
3 }
4
5 verses(starting, ending) {
6   return downTo(starting, ending)
7     .map(i => this.verse(i))
8     .join('\n');
9 }
```

Line 2 above encodes the knowledge that the overall song starts on verse 99 and counts down to 0. Lines 6-8 decrement the verse number, which moves the song from one verse to the next. But if you are familiar with "99 Bottles," you are surely aware that the song is longer than this code suggests. The real song goes on forever (or at least until all singers become sufficiently bored).

This "forever" happens in phrase 4 of the `0` case of `verse`, repeated below:

Listing 4.50: Case 0 Handles Restart

```
1  verse(number) {
2    switch (number) {
3      case 0:
4        return (
5          // ...
6          '99 bottles of beer on the wall.\n'
7        );
8      default:
9        return (
10         // ...
11         `${this.quantity(number-1)} ` +
12           `${this.container(number-1)} ` +
13           'of beer on the wall.\n'
14       );
15   }
16 }
```

Line 6 above contains a hard-coded `99`. This is *not* a special case of the `quantity` concept, which is the rule for what to sing in place of a number.

There's something subtle about the difference above, such that the underlying concept is not immediately obvious. And this, unfortunately, is a constant of programming life. If you had perfect understanding, you'd write perfect applications. Mostly, however, you're stumbling around, suffering from insufficient information, seeing problems through a glass, darkly.[14]

When you're confused, don't try to solve the entire problem straightaway. The more confused you are, the more important it is to nibble. You already know that it becomes easier to see how

things are different if you make them more alike. Instead of trying to understand everything at once, simply search for a way to make line 6 above look more like lines 11-13 (even if not identical), using existing code.

It may help to consider these questions. When the value of `number` is `5`, what does `quantity` return? How about when `number` is `95`? And finally, what would `quantity` return if you passed in `99`?

If you just realized that you can make these lines a little bit more alike by passing the `99` into `quantity`, you've got it. Here's the resulting code:

Listing 4.51: 99 Is a Quantity

```
case 0:
  return (
    // ...
    `${this.quantity(99)} bottles ` +
      'of beer on the wall.\n'
  );
default:
  return (
    // ...
    `${this.quantity(number-1)} ` +
      `${this.container(number-1)} ` +
      'of beer on the wall.\n'
  );
```

As you can see from the above, the existing `quantity` rule is fine, and it already applies. When the number 99 appears in the song, you should sing the string "99."

At this point it makes sense to scan over to the word "bottles" and replace it with the `container` method. This is a well-understood difference, and taking it off the table now reduces mental clutter. Here's the resulting code:

Listing 4.52: Case 0 Sends Container

```
case 0:
  return (
    // ...
    `${this.quantity(99)} ${this.container(number-1)} ` +
      'of beer on the wall.\n'
  );
default:
  return (
    // ...
    `${this.quantity(number-1)} ` +
      `${this.container(number-1)} ` +
      'of beer on the wall.\n'
  );
```

Having made these lines as similar as possible, it is now obvious that:

```
"99"
```

must represent the same concept as:

```
number-1
```

As always, you must name this concept, create a method, and send the message in place of the difference.

This concept is about knowing that when `number` is `50`, the result is `49`, when it's `5`, the result is `4`, when `1`, `0`, and when `0`, `99`. It's where the song determines the *next verse to be sung*.

Mathematically speaking, a number's "successor" is the next number in the higher direction, and its "predecessor" is the next number in the lower direction. This maps nicely to most people's intuitive sense that the default direction for numbers is up.

However, in the case of verse numbers in the "99 Bottles of Beer" song, the default direction is down. Most verses are followed by the next lower numbered verse (with the exception of verse 0, which is followed by verse 99). Successor *does* feel like the right name for the current concept, but using it requires that you define successor to mean *following* rather than *higher*.

The `successor` concept was unearthed using the same refactoring rules that led to `container`, `pronoun`, `quantity`, and `action`. As this idea is a bit more abstract than the others, an abundance of caution suggests that the refactoring be done in moderately small steps. In that spirit, first create the method, and have it return the `default` branch difference. Here's that code:

Listing 4.53: Successor Handles Default

```
successor(number) {
  return number - 1;
}
```

The code in `successor` refers to `number`, so the argument must be defined from the first.

Now that `successor` exists, use it in the `default` branch in place of `number-1` (line 11 below):

Listing 4.54: Default Case Sends Successor

```
case 0:
  return (
    // ...
    `${this.quantity(99)} ${this.container(number-1)} ` +
      'of beer on the wall.\n'
  );
default:
  return (
    // ...
    `${this.quantity(this.successor(number))} ` +
      `${this.container(number-1)} ` +
      'of beer on the wall.\n'
  );
```

The next step is to make the successor open to being used in the `0` case, by adding a conditional to return the correct value:

Listing 4.55: Successor Handles Both Cases

```
  successor(number) {
    if (number === 0) {
      return 99;
```

```
4 |     } else {
5 |       return number - 1;
6 |     }
7 |   }
8 |}
```

Now that the conditional exists, the `99` can be replaced by a send of `successor`, as shown on line 4 below:

Listing 4.56: Both Cases Send Successor

```
1 |case 0:
2 |  return (
3 |    // ...
4 |    `${this.quantity(this.successor(number))} ` +
5 |      `${this.container(number-1)} ` +
6 |      'of beer on the wall.\n'
7 |  );
8 |default:
9 |  return (
10|    // ...
11|    `${this.quantity(this.successor(number))} ` +
12|      `${this.container(number-1)} ` +
13|      'of beer on the wall.\n'
14|  );
```

After this change, the `0` and `default` cases are identical. Here's a overview of the resulting `verses` method:

Listing 4.57: Identical but Incomplete

```
1 |verse(number) {
2 |  switch (number) {
3 |    case 0:
4 |      return (
5 |        `${capitalize(this.quantity(number))} ` +
6 |          `${this.container(number)} ` +
7 |          'of beer on the wall, ' +
8 |        `${this.quantity(number)} ` +
9 |          `${this.container(number)} ` +
10|          'of beer.\n' +
11|        `${this.action(number)}, ` +
12|        `${this.quantity(this.successor(number))} ` +
13|          `${this.container(number-1)} ` +
14|          'of beer on the wall.\n'
15|      );
16|    default:
17|      return (
18|        `${capitalize(this.quantity(number))} ` +
19|          `${this.container(number)} ` +
20|          'of beer on the wall, ' +
21|        `${this.quantity(number)} ` +
22|          `${this.container(number)} ` +
23|          'of beer.\n' +
24|        `${this.action(number)}, ` +
25|        `${this.quantity(this.successor(number))} ` +
26|          `${this.container(number-1)} ` +
27|          'of beer on the wall.\n'
28|      );
29|  }
30|}
```

The `successor` concept illustrates the power of iterative application of the Flocking Rules. This concept wasn't even hinted at in the solutions given in Chapter 1, and if you found it when you worked the problem yourself, you're in a minority. The concept is so subtle most programmers don't notice it, and yet it simply appears if you follow this simple set of rules.

`Successor` is important, and separating it from `quantity` gives both methods a single responsibility. If you conflate choosing-what-to-sing-for-any-number (`quantity`) with deciding-what-verse-to-sing-next (`successor`), the resulting method would be harder to understand, future refactorings would be more difficult, and attempts to change the code for one idea might accidentally break it for the other.

4.9. Depending on Abstractions

Abstractions are beneficial in many ways. They consolidate code into a single place so that it can be changed with ease. They *name* this consolidated code, allowing the name to be used as a shortcut for an idea, independent of its current implementation. These are valuable benefits, but abstractions also help in another, more subtle, way. In addition to the above, abstractions tell you where your code *relies* upon an idea. But to get this last benefit, you must refer to an abstraction in every place where it applies.

Study the code above, and consider the bits that say `${this.container(number-1)}`. When `container` is called from the `0` case, the value of the passed argument is `-1`. The `-1` causes the conditional in `container` to fall through to the false branch and return "bottles." Although this code passes the tests, it does so by accident, not by design.

The code above doesn't want the `container` of `number-1`; it wants the `container` of the following verse. The `successor` method is responsible for determining the following verse. You should now defer to that abstraction, and replace all occurrences of `number-1` with `this.successor(number)`.

That final change results in this code:

Listing 4.58: Deferring to Successor

```
case 0:
  return (
    // ...
    `${this.quantity(this.successor(number))} ` +
      `${this.container(this.successor(number))} ` +
      'of beer on the wall.\n'
  );
default:
  return (
    // ...
    `${this.quantity(this.successor(number))} ` +
      `${this.container(this.successor(number))} ` +
      'of beer on the wall.\n'
  );
```

Here's the whole `verse` method, showing the `0` and `default` cases to be identical:

Listing 4.59: Identical and Using Successor Abstraction

```
verse(number) {
  switch (number) {
    case 0:
      return (
        `${capitalize(this.quantity(number))} ` +
          `${this.container(number)} ` +
          'of beer on the wall, ' +
        `${this.quantity(number)} ` +
          `${this.container(number)} ` +
          'of beer.\n' +
        `${this.action(number)}, ` +
        `${this.quantity(this.successor(number))} ` +
          `${this.container(this.successor(number))} ` +
          'of beer on the wall.\n'
      );
    default:
      return (
        `${capitalize(this.quantity(number))} ` +
          `${this.container(number)} ` +
          'of beer on the wall, ' +
        `${this.quantity(number)} ` +
          `${this.container(number)} ` +
          'of beer.\n' +
        `${this.action(number)}, ` +
        `${this.quantity(this.successor(number))} ` +
          `${this.container(this.successor(number))} ` +
          'of beer on the wall.\n'
      );
  }
}
```

One last refactoring trick proves that this common template works for all cases. Copy the template and insert it above the `switch` statement, as follows:

Listing 4.60: Using the Same Template for Every Verse

```
verse(number) {
  return (
    `${capitalize(this.quantity(number))} ` +
      `${this.container(number)} ` +
      'of beer on the wall, ' +
    `${this.quantity(number)} ` +
      `${this.container(number)} ` +
      'of beer.\n' +
    `${this.action(number)}, ` +
    `${this.quantity(this.successor(number))} ` +
      `${this.container(this.successor(number))} ` +
      'of beer on the wall.\n'
  );

  switch (number) {
    case 0:
      // ...
    default:
      // ...
  }
}
```

The above change lets you try this one template for all cases, while preserving an easy return to green if it fails. The tests *are* green after this change, so you can safely delete the entire `switch` statement.

Here's a complete listing of the resulting code:

Listing 4.61: Final Listing

```
class Bottles {
  song() {
    return this.verses(99, 0);
  }

  verses(starting, ending) {
    return downTo(starting, ending)
      .map(i => this.verse(i))
      .join('\n');
  }

  verse(number) {
    return (
      `${capitalize(this.quantity(number))} ` +
        `${this.container(number)} ` +
        'of beer on the wall, ' +
      `${this.quantity(number)} ` +
        `${this.container(number)} ` +
        'of beer.\n' +
      `${this.action(number)}, ` +
      `${this.quantity(this.successor(number))} ` +
        `${this.container(this.successor(number))} ` +
        'of beer on the wall.\n'
    );
  }

  quantity(number) {
    if (number === 0) {
      return 'no more';
    } else {
      return number.toString();
    }
  }

  container(number) {
    if (number === 1) {
      return 'bottle';
    } else {
      return 'bottles';
    }
  }

  action(number) {
    if (number === 0) {
      return 'Go to the store and buy some more';
    } else {
      return (
        `Take ${this.pronoun(number)} down and pass it around`
      );
    }
  }

  pronoun(number) {
    if (number === 1) {
      return 'it';
    } else {
      return 'one';
    }
  }

```

```
61 |   successor(number) {
62 |     if (number === 0) {
63 |       return 99;
64 |     } else {
65 |       return number - 1;
66 |     }
67 |   }
68 |}
```

This completes the current refactoring. The `verse` `switch` statement has been reduced to a single template that refers to a series of small, consistent abstractions.

Now that you're done, it's important to ask whether this new code actually improves upon the Shameless Green from whence you began. Most programmers argue that it's better, so you may be distressed to hear that according to the metrics, it's worse; all you've accomplished is to turn one conditional into many, while simultaneously adding 55% more code.

However, be of good cheer. Despite the complexity score, this code *is* better. An improvement has been made that is invisible to static analysis tools. The `container`, `pronoun`, `quantity`, `action` and `successor` concepts were invisible in Shameless Green, but are both revealed and isolated in this new code.

4.10. Summary

This chapter finished the refactoring that began in Chapter 3. It iteratively followed the Flocking Rules to remove differences in the `verse` method, and as a result unearthed abstractions that were deeply hidden within the 99 Bottles song.

It illustrated the power of the Flocking Rules to uncover sophisticated concepts, even those which cast only dim shadows in the existing code. You don't have to understand the entire problem in order to find and express the correct abstractions—you merely apply these rules, repeatedly, and abstractions will naturally appear.

One final thought before moving on. Consider this question: If several different programmers started from Shameless Green and refactored the `verse` method according to the Flocking Rules, what would the resulting code look like? If you've guessed that everyone's code would be identical, excepting the *names* used for the concepts, you'd be right. This has enormous value.

Now on to Chapter 5, which returns to the "six-pack" problem.

5. Separating Responsibilities

The previous two chapters applied the Flocking Rules to reduce duplication in the `verse` method. The resulting code is gratifyingly consistent, and now explicitly exposes concepts that cast only faint shadows in the original code. Remember, however, that the impetus behind that entire refactoring was the arrival of the six-pack requirement. Without this change in requirements, you might very well have stopped at Shameless Green.

This chapter returns to the six-pack problem. Code smells again guide the choice of the next refactoring. A new class eventually gets created, and along the way a number of big ideas are examined. This chapter explores what it means to model abstractions and rely on messages; it considers the consequences of mutation and the perils of premature performance optimization.

5.1. Selecting the Target Code Smell

Code should be open for extension and closed for modification. It's time to reexamine the current code in light of the ongoing six-pack requirement. Recall the following flowchart (which originally appeared in Chapter 3):

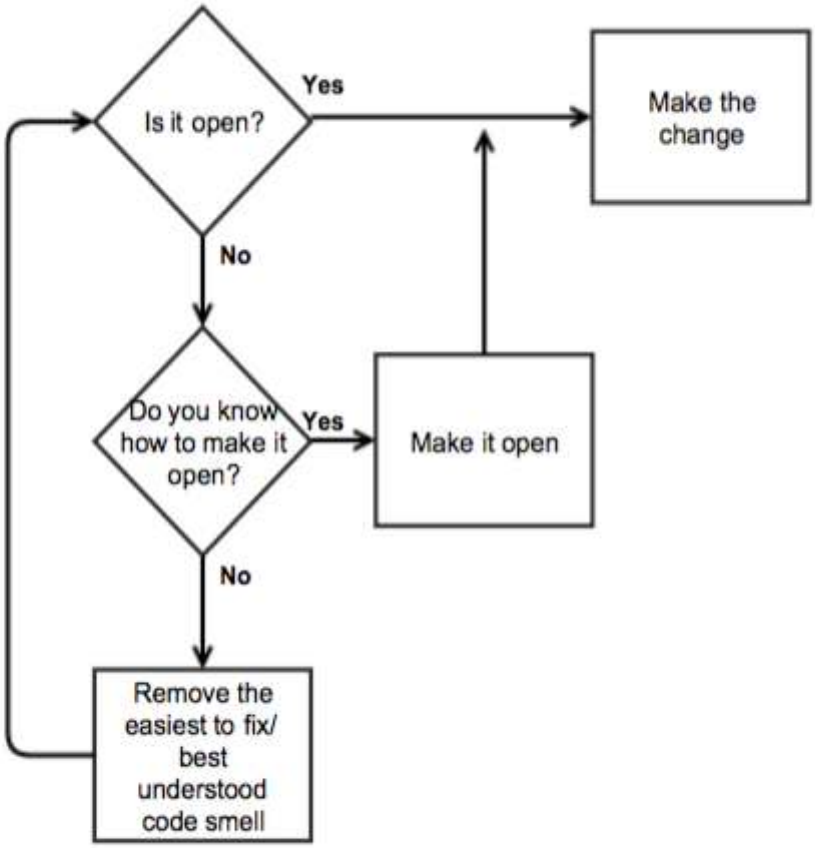

Figure 5.1: Open Closed Flowchart

Despite the fact that you've successfully replaced a fair amount of duplication with well-named methods that expose concepts in the 99 Bottles domain, the resulting code is not yet open to the six-pack requirement. If anything, the current incarnation is *less* amenable to this requirement than was Shameless Green. Within Shameless Green, you could have simply amended the `switch` statement to add a branch for verses 6 and 7. The changes needed to meet the six-pack requirement within the new code are not nearly so obvious. It may seem as if you have complicated things without making any progress towards meeting the goal.

The truth about refactoring is that it sometimes makes things worse, in which case your efforts serve gallantly to disprove an idea. The refactoring recipes don't promise to result in code that better expresses the problem—they merely make it easy to create that new expression, and just as easy to revert it. Proper refactoring allows you to explore a problem domain safely.

You've now completed one refactoring, and the resulting code is not yet open to the six-pack requirement. Not only that, but it is entirely possible that you do not yet know what change will *make* it open. At this point, you must decide whether it's better to proceed with additional modifications to the code, or better to revert the previous change and take a different tack.

The current code, although not open to the new requirement, *is* improved. This suggests that it's reasonable to continue forward, in hopes that more good things will come.

Therefore, have faith, and iterate. This means you must continue to be guided by code smells, and doing so requires that you identify the smells in the current code.

5.1.1. Identifying Patterns in Code

One way to get better at identifying smells is to practice describing the characteristics of code. Look at the `Bottles` class below and make note of the things that catch your eye. Include any patterns that you see, and things you like, hate, or don't understand. This listing is followed by a series of questions intended to inspire further thoughts, so take a minute to ponder before reading on.

Listing 5.1: DRY Bottles Class

```
class Bottles {
  song() {
    return this.verses(99, 0);
  }

  verses(starting, ending) {
    return downTo(starting, ending)
      .map(i => this.verse(i))
      .join('\n');
  }

  verse(number) {
    return (
      `${capitalize(this.quantity(number))} ` +
        `${this.container(number)} ` +
        'of beer on the wall, ' +
      `${this.quantity(number)} ` +
        `${this.container(number)} ` +
        'of beer.\n' +
      `${this.action(number)}, ` +
      `${this.quantity(this.successor(number))} ` +
        `${this.container(this.successor(number))} ` +
        'of beer on the wall.\n'
    );
  }

  quantity(number) {
    if (number === 0) {
      return 'no more';
    } else {
      return number.toString();
    }
  }

  container(number) {
    if (number === 1) {
      return 'bottle';
    } else {
```

```
      return 'bottles';
    }
  }

  action(number) {
    if (number === 0) {
      return 'Go to the store and buy some more';
    } else {
      return (
        `Take ${this.pronoun(number)} down and pass it around`
      );
    }
  }

  pronoun(number) {
    if (number === 1) {
      return 'it';
    } else {
      return 'one';
    }
  }

  successor(number) {
    if (number === 0) {
      return 99;
    } else {
      return number - 1;
    }
  }
}
```

The following questions draw attention to a number of interesting characteristics of the code as it's written so far:

1. Do any methods have the same shape?
2. Do any methods take an argument of the same name?
3. Do arguments of the same name always mean the same thing?
4. If you were going to break this class into two pieces, where's the dividing line?

For those methods created by the Flocking Rules (`quantity`, `container`, `action`, `pronoun` and `successor`, hereafter referred to as the "flocked five"):

5. Do the tests in the conditionals have anything in common?
6. How many branches do the conditionals have?
7. Do the methods contain any code *other* than the conditional?
8. Does each method depend more on the argument that got passed, or on the class as a whole?

The remainder of this section examines the above questions. If any didn't occur to you, look back at the code and try to answer them before proceeding.

5.1.2. Spotting Common Qualities

The first four questions above look at the class as a whole and expose common qualities of the code. This next section examines these questions in detail.

Question 1: Do any methods have the same shape?

Yes. The flocked five all have the same shape.

You can easily identify same-shaped methods by doing the Squint Test (see sidebar). The fact that these methods are so consistent is a tribute to the Flocking Rules. Had the methods been created at different times, by different people, for different reasons, they could easily have contained a variety of shapes. For example, the following three methods are *logically* the same:

Listing 5.2: Various Conditional Forms

```
// verbose conditional
container(number) {
  if (number === 1) {
   return 'bottle';
  } else {
   return 'bottles';
  }
}

// guard clause
quantity(number) {
  if (number === 0) {
    return 'no more';
  }

  return number.toString();
}

// ternary expression
pronoun(number) {
  return number === 1 ? 'it' : 'one';
}
```

All of the above methods pass the tests. The problem is not that the code doesn't work; it's that the non-essential variation disguises a common shape. This unnecessary variation makes the methods appear to be different when they are actually very much the same.

Programmers naturally assume that difference exists for a reason, but here there isn't one. Superfluous difference raises the cost of reading code, and increases the difficulty of future refactorings.

It's not yet clear what it means that these methods have the same shape, but it's important to notice that they do.

Squint Test

One easy way to judge code is by performing a Squint Test. This test requires no setup, and can be performed on any code at any time.

Here's how it works:

1. Put the code of interest on your screen.
2. Lean back.*
3. Squint your eyes such that you can still see the code, but can no longer read it.
4. Look for:
    a. changes in shape, and
    b. changes in color.

Changes in indentation reveal the presence of conditionals. Two or more levels of indentation expose nested conditionals. Conditionals result in multiple execution paths through the code, which add complexity and make code hard to understand.

Changes in color indicate differences in the level of abstraction. A method that intermixes many colors tells a story that will be difficult to follow.

*Instead of leaning back and squinting, it's acceptable to zoom out in your text editor until you can no longer read the code, but can still see its shape and color.

Question 2: Do any methods take an argument of the same name?

Six methods take `number` as an argument—the `verse` method and the flocked five.

Listing 5.3: Methods Which Take an Argument Named Number

```
verse(number)
quantity(number)
container(number)
action(number)
pronoun(number)
successor(number)
```

Question 3: Do arguments of the same name always mean the same thing?

The easiest way to understand what `number` represents is to follow its path through the code, beginning with `song`. Here's a reminder of that method:

Listing 5.4: Song Method

```
song() {
  return this.verses(99, 0);
}
```

When `song` sends `this.verses(99, 0)`, the `99` and `0` represent the starting and ending verse numbers to sing. You could argue that the `99` and `0` represent the starting number of *bottles* in

the verse to be sung, but that would be stretching it and you'd be in a minority. Most folks interpret the `99` and `0` as *verse* numbers.

If `song` is sending verse numbers to `verses`, the `verses` method must be receiving them. Here's that method:

Listing 5.5: Verses Method

```
verses(starting, ending) {
  return downTo(starting, ending)
    .map(i => this.verse(i))
    .join('\n');
}
```

The `upper` and `lower` arguments are verse numbers. The `verses` method iterates between them, so `i`, the argument provided to the anonymous function, must also represent a verse number. Therefore, and quite sensibly so, the argument with which `verse` is invoked must be the verse number to be sung. As received by `verse`, this argument is named `number`.

```
verse(number)
```

To repeat (with no intention to belabor the point), the `number` argument taken by `verse` represents a verse number.

Now switch your attention to the flocked five, all of which also take an argument named `number`. Here, for example, is `container`:

```
container(number)
```

The question at hand is whether `number` as received by `container` represents the same concept as `number` as received by `verse`. To answer this question, consider the entire `verse` method:

Listing 5.6: Verse Method

```
verse(number) {
  return (
    `${capitalize(this.quantity(number))} ` +
      `${this.container(number)} ` +
      'of beer on the wall, ' +
    `${this.quantity(number)} ` +
      `${this.container(number)} ` +
      'of beer.\n' +
    `${this.action(number)}, ` +
    `${this.quantity(this.successor(number))} ` +
      `${this.container(this.successor(number))} ` +
      'of beer on the wall.\n'
  );
}
```

Notice that line 4 above invokes `container` with `number`, while line 11 invokes `container` with `this.successor(number)`. Within every verse, `container` is invoked twice, on two different values.

This happens because each verse knows about two different numbers of bottles. Verse 37, for example, begins with 37 bottles of beer, and ends with 36. As you've already seen, the incoming

`number` argument to `verse` represents a *verse number*. However, the parameter that `verse` then passes on to `container` stands for something else—a *bottle number*.

The same is true for the other flocked five methods—the argument they receive is a bottle number rather than a verse number. Thus, the `verse` method and the flocked five methods use the same argument name to represent different concepts.

This is rarely a good idea.

If you have long since noticed this issue, congratulations, but you're in a minority. Most folks who work this problem name the argument taken by the flocked five methods after the parameter passed from `verse`. Initially, this made perfect sense. Back in Chapter 3, when the Flocking Rules led to the extraction of the `container` method, your grasp of the problem was less developed than it is now. Then it was clear only that:

- the `switch` statement in `verse` switched on `number`, and
- `container` needed an argument in order to decide whether to return "bottle" or "bottles."

In the interests of consistency, it was reasonable back in Chapter 3 to name the argument taken by `container` after the parameter being passed from `verse`. In the interim it hasn't mattered that `number` stands for a verse number within `verse` but a bottle number within `container`.

Now, however, it begins to. Having multiple methods that take the same argument is a code smell. It's important, however, to recognize that here the term "same" means *same concept*, not *identical name*. In an ideal world, each different concept would have its own unique, precise name, and there would be no ambiguity. Unfortunately, real world code often fails to meet this ideal. In long-lived applications, the same concept might go by several different names, or, as in this case, different concepts might hide behind a single name. These naming mistakes make it harder to notice underlying code smells, and now that you're looking for patterns in the code, you must examine the arguments and clarify the abstractions that they represent.

Having examined the use of `number` in `Bottles`, it's now clear that this argument represents a verse number to `verse`, but a bottle number to the flocked five methods.

Question 4: If you were going to break this class into two pieces, where's the dividing line?

After `verse` and before the flocked five methods.

5.1.3. Enumerating Flocked Method Commonalities

Now that you've considered the class as a whole, it's time to move on to questions five through eight, which apply only to the flocked five methods.

Question 5: Do the tests in the conditionals have anything in common?

Here's a summary of the conditionals:

Listing 5.7: Flocked Five Conditionals

```
quantity(number) {
  if (number === 0) {
    // ...
  }
}

container(number) {
  if (number === 1) {
    // ...
  }
}

action(number) {
  if (number === 0) {
    // ...
  }
}

pronoun(number) {
  if (number === 1) {
    // ...
  }
}

successor(number) {
  if (number === 0) {
    // ...
  }
}
```

In the code above, not only do all of the conditionals test the value of `number`, but they test for `number` to be *exactly equal* to another value.

These conditionals could logically have used the less than, greater than or not equal operators, and still pass the tests. The Incomprehensibly Concise example in Chapter 1 managed to use all four of these operations, and your own solution may also have had conditionals that tested for something other than equality.

Programmers tend to blithely interchange these different comparison operators, confident that if the tests pass, the code is correct. However, having tests that pass doesn't guarantee the best expression of code, and this is a case where your choice of operator affects future costs.

Testing for equality has several benefits over the alternatives. Most obviously, it narrows the range of things that meet the condition. In the above examples, if unexpected values of `number` arrive, the `else` branch executes. Knowing that the only way to get to the `true` branch is by supplying an exact value of `number` makes it easier for future readers to understand the code. This reduces the difficulty of debugging errors caused by incorrect inputs. Testing for equality also makes the code more precise, and this precision, as you will soon see, enables future refactorings.

> Question 6: How many branches do the conditionals have?

Each conditional contains two branches. This may or may not have meaning, but it's certainly a visible quality of the code and thus worth noting.

> Question 7: Do the methods contain any code *other* than the conditional?

No. Each method is named after a concept, and contains a single conditional. This conditional uses the value of `number` to choose the correct concrete expression of the concept. These methods are fiercely committed to having one responsibility and never conflating two concepts.

> Question 8: Do methods that take `number` as an argument depend more on `number`, or more on the class as a whole?

The flocked five depend only on the `number` argument, rather than on the rest of the class. This is true even for `action`, if you accept that although `action` depends on `pronoun`, `pronoun` depends only on `number`.

In conjunction, these eight questions group certain methods together. The same-shaped, same-kind-of-conditional-testing, bottle-number-taking, argument-depending, flocked five methods fall into one group, and the `song`, `verses`, and `verse` methods into another. The answers to the questions above reveal many characteristics of the code, but there's one more quality to discuss before moving on.

5.1.4. Insisting Upon Messages

This code contains a deeply non-object-oriented pattern: the flocked five methods take an argument, examine it, and then *supply behavior for it*.

As you've seen, those five methods share this common shape:

```
container(number) {
  if (number === 1) {
   return 'bottle';
  } else {
   return 'bottles';
  }
}
```

The above method was created by the Flocking Rules, and so exhibits many desirable qualities. Despite that, it's deeply flawed when considered from the point of view of an independent OO practitioner. What that practitioner would see here is that someone has gone to the trouble of injecting a dependency (`number`), but that dependency is too impaired to supply the needed behavior. Consequently, not only does `container` know about `number`, but it's also forced to understand what the specific values of `number` mean, and to know what to do in each case. The `container` method *depends* on each of these things. If any of them change, the `container` method might be forced to change in turn.

It made sense to tolerate a conditional back in Shameless Green. That solution optimized for understandability without regard for changeability. Its goal was to get to green quickly. The

resulting code was more procedural than object-oriented, but would have been good enough if nothing ever changed. However, now that you have a new requirement and are rearranging the code, you'd like to apply a full-blown OO mindset, and that mindset is deeply suspicious of conditionals.

As an OO practitioner, when you see a conditional, the hairs on your neck should stand up. Its very presence ought to offend your sensibilities. You should feel *entitled* to send messages to objects, and look for a way to write code that allows you to do so. The above pattern means that objects are missing, and suggests that subsequent refactorings are needed to reveal them. Be on the lookout for this code shape, as it implies that there's more to be done.

This is not to say that you'll never have a conditional in an object-oriented application. There *is* a place for conditionals in OO. Manageable OO applications consist of pools of small objects that collaborate to accomplish tasks. Collaborators must be brought together in useful combinations, and assembling these combinations requires knowing which objects are suitable. Some object, somewhere, must choose which objects to create, and this often involves a conditional.

However, there's a big difference between a conditional that selects the correct object and one that supplies behavior. The first is acceptable and generally unavoidable. The second suggests that you are missing objects in your domain.

Code is striving for ignorance, and preserving ignorance requires minimizing dependencies. The `container` method yearns to be injected with a smarter object to which it could merely forward the message, as shown here:

```
container(smarterNumber) {
  return smarterNumber.container();
}
```

The existing code is imploring you to create that smarter object.

5.2. Extracting Classes

The questions above identify characteristics that group methods together, and many of these groups overlap. For example, a number of methods take the same argument. Most methods that do so have the same shape, contain a conditional, could be considered private, and depend more on the argument than on the class as a whole.

Each item above acts like a vote, and these votes combine to point to *Primitive Obsession* as the dominant code smell. Built-in data types like strings, numbers, and booleans are examples of "primitives." *Primitive Obsession* is when you use one of these data types to represent a concept in your domain. Obsessing on a primitive results in code that passes built-in types around, and supplies behavior for them.

The cure for *Primitive Obsession* is to create a new class to use in place of the primitive. For this operation, the refactoring recipe is *Extract Class*.

5.2.1. Modeling Abstractions

Having decided to cure the *Primitive Obsession* code smell with the *Extract Class* refactoring, you must now choose a name for this new class.

The primitive that you're replacing represents a bottle number. Notice that it is not a *bottle*: it's a bottle *number*. A bottle is made of plastic, or glass, or aluminum, and contains water, or soda, or beer. A bottle has a shape and a volume. It exists in the physical world.

Unlike bottles, numbers aren't things—they're ideas, albeit ones so ubiquitous that you've likely forgotten how abstract and unlikely they are. Numbers are symbols used to describe quantities of things. They don't physically exist. You can pick up a bottle, but you cannot pick up a "six."

This new class does not represent a kind of bottle: it represents a kind of number. The distinction may seem subtle, but the divide between these two concepts is chasmic. A bottle is a thing, while a number is an idea. It's easy to imagine creating objects that stand in for things, but the power of OO is that it lets you model ideas.

Model-able ideas often lie dormant in the interactions between other objects. For example, an event management application might contain `Buyer` and `Ticket` classes. `Buyer` and `Ticket` are obvious because you can reach out and touch them in the real world. These objects interact in many ways: buyers buy tickets, perhaps at a discount, and may later change their minds and return the tickets for refunds.

Where, in such an application, should the logic to manage purchases, discounts, and refunds reside? You *could* jam everything into `Buyer` and `Ticket`, but the power of OO is that it allows you to create a virtual world in which `Purchase`, `Discount` and `Refund` are just as real. Embodying these concepts into discrete classes separates responsibilities and makes the overall application easier to understand, test, and change.

Experienced OO programmers deftly create virtual worlds in which ideas are as real as physical things. If you are not yet comfortable doing so, start today by thinking of the class you're about to extract not as a physical bottle, but as a symbolic number with an added bit of bottle-ish behavior.

Bearing that idea in mind, consider what to name this class. The two most obvious choices are `BottleNumber`, or `ContainerNumber`.

5.2.2. Naming Classes

You've been introduced to the rule about naming methods at one higher level of abstraction than their current implementation. Extrapolated to classes, that rule suggests this new class should be named `ContainerNumber`. However, you've also read fairly lengthy discourses about not anticipating the future, and since the existing requirements involve only bottles, you might lean towards `BottleNumber`.

`BottleNumber` is less flexible but more straightforward. `ContainerNumber` is just the opposite; it's a bit more general, and so would work for a broader range of vessels. `BottleNumber` is more concrete. `ContainerNumber` is more abstract.

The tie-breaker here is that the "name things at one higher level of abstraction" rule applies more to methods than to classes. It would be speculative to call this new class `ContainerNumber`. The rule about naming can thus be amended: while you should continue to name methods after what they *mean*, classes can be named after what they *are*.

Having two requirements for bottles firmly suggests that this class represents a bottle number, and should be named as such. As always, you can revisit this decision if things change later.

5.2.3. Extracting BottleNumber

This section extracts a new class named `BottleNumber` from the existing code. It does *not* use TDD. Instead, it creates the new class by following a slightly modified version of Martin Fowler's *Extract Class* refactoring recipe.

As you might recall, safe refactoring relies upon tests running green, so the fact that the new `BottleNumber` class will come into existence before its tests arrive has a couple of consequences. First, the existing `Bottles` tests become the safety net for this new class. They were originally written as unit tests, but using them to indirectly test `BottleNumber` transforms them into a kind of integration test. These tests must continue to run after every change.

Next, while extracting the class, code *that is known to work* is copied from `Bottles` into `BottleNumber`. It's important to put this new class fully into use before editing any of the copied code. Safety is being provided by the `Bottles` tests, so they must exercise the new code as quickly as possible.

In the previous chapters, the process of changing code was subdivided into four steps.

1. parse the new code
2. parse and execute it
3. parse, execute and use its result
4. delete unused code

These steps still apply. Start the class extraction by creating an empty `BottleNumber` class, as shown below:

Listing 5.8: BottleNumber Class Definition

```
class Bottles {
  // ...
}

class BottleNumber {
}
```

As you go through this refactoring, remember to save the code after every change, and to run the tests after every save.

Next, copy the methods that obsess on bottle `number` into the new class.

Listing 5.9: Obsessive Methods Copied to BottleNumber

```
class Bottles {
  // ...
  quantity(number) {
    if (number === 0) {
      return 'no more';
    } else {
      return number.toString();
    }
  }

  container(number) {
    // ...
  }

  action(number) {
    // ...
  }

  pronoun(number) {
    // ...
  }

  successor(number) {
    // ...
  }
}

class BottleNumber {
  quantity(number) {
    if (number === 0) {
      return 'no more';
    } else {
      return number.toString();
    }
  }

  container(number) {
    if (number === 1) {
      return 'bottle';
    } else {
      return 'bottles';
    }
  }

  action(number) {
    if (number === 0) {
      return 'Go to the store and buy some more';
    } else {
      return (
        `Take ${this.pronoun(number)} down and pass it around`
      );
    }
  }

  pronoun(number) {
    if (number === 1) {
      return 'it';
    } else {
      return 'one';
    }
  }

  successor(number) {
```

```
64 |    if (number === 0) {
65 |      return 99;
66 |    } else {
67 |      return number - 1;
68 |    }
69 |  }
70 |}
```

Remember that the `verse` method should *not* be extracted. Even though its argument is also named `number`, in this case the argument represents a verse number, not a bottle number.

Notice that the above example *copied* methods from `Bottle` to `BottleNumber`. The methods weren't moved—they were duplicated, so nothing about `Bottle` has yet been changed. This means that the old code continues to work as is and the new code is not yet being executed. Running the tests at this point merely parses the new code, proving that it's syntactically correct.

As mentioned earlier, the recipe being followed here was inspired by one from Martin Fowler. The "official" *Extract Class* recipe begins by linking the old class to the new. Then one at a time, the recipe moves attributes, and then methods, of interest. In contrast, the example above starts with Fowler's final step, and combines all of the method moves within a single change.

This may seem like a large leap, but here you can be confident that you're moving the right group of methods. These methods were created by the Flocking Rules, so they visibly share a common pattern. This common pattern makes it easy to recognize that they belong together in the extracted class. This visual similarity is a tribute to the rules, and an illustration of the value of stable landing points (remember the stream and the rocks?) The prior refactoring resulted in deeply consistent code, and here's more proof that consistent code makes the current refactoring easy.

The `BottleNumber` class needs to know the value of `number`, so add a `constructor` method to store it in a property. Here's the code:

Listing 5.10: BottleNumber Holding Onto Number

```
1 |class BottleNumber {
2 |  constructor(number) {
3 |    this.number = number;
4 |  }
5 |  // ...
6 |}
```

The `number` property is set within the `constructor` method on line 3 above. That `constructor` method gets invoked when `new BottleNumber` is called.

The `BottleNumber` class now contains all of the necessary code, but as yet this code is only being parsed. The next small step is to execute a bit of the new class without using the result.

The following example does this by altering the `quantity` method of `Bottles` to invoke the `quantity` method of `BottleNumber`:

Listing 5.11: Parse and Execute a Bit of New Code

```
1 class Bottles {
2   // ...
3   quantity(number) {
4     new BottleNumber(number).quantity(number);
5     if (number === 0) {
6       return 'no more';
7     } else {
8       return number.toString();
9     }
10  }
11  // ...
12 }
```

Line 4 above executes the new method, but then discards the result in favor of existing code. This proves that the new code can execute without blowing up, but does not prove that it returns the correct result.

It must now be admitted that the added line of code is, by any standard, ugly.

```
new BottleNumber(number).quantity(number);
```

In the above code, both `new BottleNumber()` and `quantity` require the `number` argument, so it must be passed twice. You may find this annoyingly redundant. In the newly-created `BottleNumber` class, the `quantity` method could easily make do without an argument. It can get the right number by simply reading its own `number` property. Instead of the code above, you'd prefer:

```
new BottleNumber(number).quantity();
```

However, as previously mentioned, you should refrain from altering the code in these copied methods until the new class is fully wired into the old. Regardless of how much you hate passing the parameter twice, at this point you should resist the urge to make the change shown above. First, fully connect `BottleNumber` to `Bottles`. Once that's complete, you can return and improve the methods in `BottleNumber`.

So, setting that unpleasant code temporarily aside, the next small step in the current refactoring is to use the result of the `quantity` message within the `Bottle` class. The easiest way to accomplish this is to add a `return` keyword to the beginning of line 4, like so:

Listing 5.12: Parse, Execute and Use Result

```
1 class Bottles {
2   // ...
3   quantity(number) {
4     return new BottleNumber(number).quantity(number);
5     if (number === 0) {
6       return 'no more';
7     } else {
8       return number.toString();
9     }
10  }
11  // ...
12 }
```

The tests pass, so now you can delete the old implementation from `quantity` (lines 5-9 above). This leaves the following code:

Listing 5.13: Resulting Quantity Method

```
class Bottles {
  // ...
  quantity(number) {
    return new BottleNumber(number).quantity(number);
  }
  // ...
}
```

Repeat the above procedure for each of the methods copied from the `Bottles` class. This is an extremely mechanical, wonderfully boring, and deeply comforting refactoring process.

Here's the resulting `Bottles` class:

Listing 5.14: Forwarding Messages to BottleNumber

```
class Bottles {
  // ...
  quantity(number) {
    return new BottleNumber(number).quantity(number);
  }

  container(number) {
    return new BottleNumber(number).container(number);
  }

  action(number) {
    return new BottleNumber(number).action(number);
  }

  pronoun(number) {
    return new BottleNumber(number).pronoun(number);
  }

  successor(number) {
    return new BottleNumber(number).successor(number);
  }
}
```

These methods in `Bottles` now merely forward messages along to `BottleNumber`.

5.2.4. Removing Arguments

Now that the old `Bottles` class fully uses `BottleNumber`, the existing tests serve as a safety net for changes to the new class. This means that you can now undertake improvements in the new code.

Although `BottleNumber` works, parts of it are annoyingly redundant. The problem is that even though instances of `BottleNumber` know their `number`, its methods continue to require `number` as an argument. To illustrate, here are the two `quantity` methods:

Listing 5.15: Redundant Arguments

```
class Bottles {
  // ...
```

```
3 |   quantity(number) {
4 |     return new BottleNumber(number).quantity(number);
5 |   }
6 |   // ...
7 |}
8 |
9 |class BottleNumber {
10|   constructor(number) {
11|     this.number = number;
12|   }
13|
14|   quantity(number) {
15|     if (number === 0) {
16|       return 'no more';
17|     } else {
18|       return number.toString();
19|     }
20|   }
21|   // ...
22|}
```

Line 4 above gets a new `BottleNumber` and asks for its `quantity`. Doing so requires two references to `number`. The `constructor` method (invoked by `new` and defined on line 10) and the `quantity` method (line 14) both require a `number` argument.

The point of the *Primitive Obsession*/*Extract Class* refactoring is to create a smarter object to stand in for the primitive. This smarter object, by definition, knows both the value of the primitive and its associated behavior. Because the new `BottleNumber` class holds the right number, the methods in `BottleNumber` don't need to take an argument, and invokers of these methods could be relieved of their obligation to pass a parameter.

Now that `BottleNumber` is fully connected to `Bottles`, it's safe to start making these improvements. Notice that if you're willing to simultaneously alter both the senders and the receivers of every message, it's easy to make this change. For example, you could fix the `quantity` method by changing line 4 above to remove the parameter being passed to `quantity`, while simultaneously deleting the argument from line 14 and replacing the references to it on lines 15 and 18 with `this.number`. If you make all of these changes at once, and then save the code, the tests will pass.

Keep in mind that is a multi-line change. Some problems are so simple that it's easiest to just leap in and make such a change, but others are so complex that it isn't feasible to fix everything at once. In real-world applications, the same method name is often defined several times, and a message might get sent from many different places. Learning the art of transforming code one line at a time, while keeping the tests passing at every point, lets you undertake enormous refactorings piecemeal. This small problem is a good place to practice this technique, in preparation for later tackling bigger ones.

Back in Chapter 3, you had to *add* an argument to a method that was already being called *without* one. This is the opposite problem: here you need to *remove* an argument from a method that's currently being invoked *with* one.

Consider `quantity`, repeated again below. This method takes a number *argument*. Remember, however, that the `BottleNumber` object itself has a `number` *property*.

Listing 5.16: BottleNumber Quantity Redux

```
1 class BottleNumber {
2   // ...
3   quantity(number) {
4     if (number === 0) {
5       return 'no more';
6     } else {
7       return number.toString();
8     }
9   }
10  // ...
11 }
```

The trick to working your way forward under green while making only one-line changes, is to alter the uses of the `number` argument to refer to the `number` property instead. Lines 4 and 7 below contain that change:

Listing 5.17: Unused Argument

```
1 class BottleNumber {
2   // ...
3   quantity(number) {
4     if (this.number === 0) {
5       return 'no more';
6     } else {
7       return this.number.toString();
8     }
9   }
10  // ...
11 }
```

Above, `number` has been replaced by `this.number` on lines 4 and 7. That change turns the argument reference into a property reference, which allows this method to depend upon one of the object's own properties rather than an argument passed by someone else.

Now that the argument is unused, turn your attention to senders of `quantity`. In this application there's only the one in `Bottles`, shown here:

Listing 5.18: Forward With Redundant Arguments

```
1 class Bottles {
2   // ...
3   quantity(number) {
4     return new BottleNumber(number).quantity(number);
5   }
6   // ...
7 }
```

Removing the `number` parameter from the `quantity` message invocation on line 4 results in this code:

Listing 5.19: Forward Without Redundancy

```
1 class Bottles {
2   // ...
```

```
3 |   quantity(number) {
4 |     return new BottleNumber(number).quantity();
5 |   }
6 |   // ...
7 |}
```

Because JavaScript is such a permissive language, you can't rely on it to tell you whether you've fixed all `quantity` invocations. After you remove the `number` parameter from the `quantity` method, JavaScript will silently tolerate any senders that are still providing an argument. It's important to be confident that you have found and fixed all invocations before undertaking the actual removal.

To convince yourself that you haven't missed any other senders of `quantity`, temporarily add a guard clause at the beginning of the method to check the number of arguments provided by the sender:

```
1 |class BottleNumber {
2 |  // ...
3 |  quantity(number) {
4 |    if (arguments.length !== 0) {
5 |      throw new Error('wrong number of arguments');
6 |    }
7 |
8 |    if (this.number === 0) {
9 |      return 'no more';
10|    } else {
11|      return this.number.toString();
12|    }
13|  }
14|  // ...
15|}
```

Lines 4-6 above will throw an exception if `quantity` is called with any arguments. Even so, the tests still pass, so you may now feel confident that every sender has been updated. Having gained that confidence, you may safely delete the guard clause.

Once you have located and removed the parameter from all of its senders, the `quantity` method definition no longer needs to take an argument. You can now return to `BottleNumber` and remove the `number` argument, as on line 3 below:

Listing 5.20: BottleNumber Quantity Method Without Argument

```
1 |class BottleNumber {
2 |  // ...
3 |  quantity() {
4 |    if (this.number === 0) {
5 |      return 'no more';
6 |    } else {
7 |      return this.number.toString();
8 |    }
9 |  }
10|  // ...
11|}
```

Here's a recap of the steps for removing an argument using one-line changes.

1. Alter the method body to replace occurrences of the argument with references to the property of the same name.

   In the example above:

   ```
   if (number === 0) {
     return 'no more';
   } else {
     return number.toString();
   }
   ```

   became:

   ```
   if (this.number === 0) {
     return 'no more';
   } else {
     return this.number.toString();
   }
   ```

2. Change every sender of the message to remove the parameter. In the example:

   ```
   new BottleNumber(number).quantity(number)
   ```

   became:

   ```
   new BottleNumber(number).quantity()
   ```

3. Confirm that you've updated every sender by adding a temporary guard clause to the method body. In the example above:

   ```
   quantity(number) {
     // ...
   }
   ```

   became:

   ```
   quantity(number) {
     if (arguments.length !== 0) {
       throw new Error('wrong number of arguments');
     }

     // ...
   }
   ```

4. Finally, delete the argument and guard clause from the method definition. So:

   ```
   quantity(number) {
     if (arguments.length !== 0) {
       throw new Error('wrong number of arguments');
     }

     // ...
   }
   ```

   became:

   ```
   quantity() {
     // ...
   }
   ```

As you can see, despite the length of the explanation, the technique is simple, and involves only four steps. Having practiced on `quantity`, the other methods will easily bend to your will. You can now follow this process to remove the `number` argument from the remaining methods in `BottleNumber`.

If you do this refactoring yourself, you'll find that `container` and `action` work as expected, but that when you change `pronoun`, the tests begin to fail.

5.2.5. Trusting the Process

Refactorings that lead to errors can shake your faith in the validity of the corresponding recipes. However, these recipes have proven themselves reliable for many people across many circumstances. If you adhere to a recipe and tests start failing, it's likely that there's something about the problem that you don't yet understand.

In this case, you've been using the "remove arguments via one-line changes" process. It works for `quantity`, `container`, and `action` but causes the tests to fail when applied to `pronoun`.

Specifically, if you go to the `pronoun` definition in `BottleNumber`:

```
class BottleNumber {
  // ...
  pronoun(number) {
    if (number === 1) {
      // ...
    }
  }
}
```

and change `number` to `this.number`:

```
class BottleNumber {
  // ...
  pronoun(number) {
    if (this.number === 1) {
      // ...
    }
  }
}
```

Then go to the `pronoun` method in `Bottles`:

```
class Bottles {
  // ...
  pronoun(number) {
    return new BottleNumber(number).pronoun(number);
    // ...
  }
}
```

and remove the parameter from the forward of `pronoun` to `BottleNumber`:

```
class Bottles {
  // ...
  pronoun(number) {
```

```
    return new BottleNumber(number).pronoun();
    // ...
  }
}
```

You return to the `pronoun` method definition in `BottleNumber` and add the temporary guard clause:

```
class BottleNumber {
  // ...
  pronoun(number) {
    if (arguments.length !== 0) {
      throw new Error('wrong number of arguments');
    }
  }
}
```

Then the tests begin to fail with:

```
wrong number of arguments
```

The process that worked for other methods is now failing for `pronoun`. While this error might lead you to doubt the validity of the technique, it doesn't point out a flaw in the process. Instead, it exposes a slightly more complex bit of code.

Recall the steps needed to remove parameters:

1. Alter the method definition to replace occurrences of the argument with references to the property of the same name.
2. Change every sender of the message to remove the parameter.
3. Add a temporary guard clause to the method body.
4. Delete the argument and guard clause from the method definition.

The failure appeared after step 3. The error message indicates that some caller is still passing a parameter to `pronoun`. This means step 2 isn't complete; in other words, some *sender* has not been fixed. This should trigger you to examine the source code where the failure occurred. When you do so, you'll see the following:

```
class BottleNumber {
  // ...
  action(number) {
    if (number === 0) {
      return 'Go to the store and buy some more';
    } else {
      return (
        `Take ${this.pronoun(number)} down and pass it around`
      );
    }
  }
}
```

It turns out that `pronoun` is invoked only from the `action` method of `BottleNumber`, where the message is sent to `this`. The `pronoun` method defined back in `Bottles` is no longer used (as you

can confirm by cavalierly deleting it and running the tests).

Instead of changing the unused `pronoun` method in `Bottles`, step 2 should have removed the `number` argument from the call to `pronoun` in the `action` method of `BottleNumber`, leaving:

```
class BottleNumber {
  // ...
  action(number) {
    if (number === 0) {
      return 'Go to the store and buy some more';
    } else {
      return (
        `Take ${this.pronoun()} down and pass it around`
      );
    }
  }
}
```

Once you make *that* change and then complete the steps, the code passes the tests.

The lesson here is that *the process works*, and that encountering errors while following it suggests that a closer look at the code is in order. A great benefit of these refactoring techniques is that you can accomplish quite a bit while thinking very little. Sometimes, however, thought just can't be avoided. The blessing of these techniques is that altering code in such small increments severely constrains the number of errors any change can introduce. When forced to think, you can be confident that your efforts will be narrowly focused on an opportune topic.

Now that `pronoun` works, only the `successor` method remains. It succumbs to this refactoring with no surprises. This completes the removal of extraneous arguments to methods in the `BottleNumber` class, and leaves the code at the following resting point.

Listing 5.21: Forward Messages to Smarter Number

```
class Bottles {
  song() {
    return this.verses(99, 0);
  }

  verses(starting, ending) {
    return downTo(starting, ending)
      .map(i => this.verse(i))
      .join('\n');
  }

  verse(number) {
    return (
      `${capitalize(this.quantity(number))} ` +
        `${this.container(number)} ` +
        'of beer on the wall, ' +
      `${this.quantity(number)} ` +
        `${this.container(number)} ` +
        'of beer.\n' +
      `${this.action(number)}, ` +
      `${this.quantity(this.successor(number))} ` +
        `${this.container(this.successor(number))} ` +
        'of beer on the wall.\n'
    );
  }
```

```
26 |
27 |   quantity(number) {
28 |     return new BottleNumber(number).quantity();
29 |   }
30 |
31 |   container(number) {
32 |     return new BottleNumber(number).container();
33 |   }
34 |
35 |   action(number) {
36 |     return new BottleNumber(number).action();
37 |   }
38 |
39 |   successor(number) {
40 |     return new BottleNumber(number).successor();
41 |   }
42 |}
43 |
44 |class BottleNumber {
45 |   constructor(number) {
46 |     this.number = number;
47 |   }
48 |
49 |   quantity() {
50 |     if (this.number === 0) {
51 |       return 'no more';
52 |     } else {
53 |       return this.number.toString();
54 |     }
55 |   }
56 |
57 |   container() {
58 |     if (this.number === 1) {
59 |       return 'bottle';
60 |     } else {
61 |       return 'bottles';
62 |     }
63 |   }
64 |
65 |   action() {
66 |     if (this.number === 0) {
67 |       return 'Go to the store and buy some more';
68 |     } else {
69 |       return (
70 |         `Take ${this.pronoun()} down and pass it around`
71 |       );
72 |     }
73 |   }
74 |
75 |   pronoun() {
76 |     if (this.number === 1) {
77 |       return 'it';
78 |     } else {
79 |       return 'one';
80 |     }
81 |   }
82 |
83 |   successor() {
84 |     if (this.number === 0) {
85 |       return 99;
86 |     } else {
87 |       return this.number - 1;
88 |     }
```

This completes the extraction of `BottleNumber`. The original `Bottles` class is now free of conditionals, but they didn't disappear—they just moved into this new class in a slightly simpler form. Even with the conditionals, however, the code in `BottleNumber` has a regular, orderly aspect that feels pleasing, and bodes well for future refactorings.

It's almost time to return your focus to the `Bottles` class, but before doing so, there are a few broad ideas to consider.

5.3. Appreciating Immutability

To *mutate* is to change. *State* is "the particular condition of something at a specific time." A *variable* is "that which varies," or, in maths, "a quantity which admits an infinite number of values in the same expression."

In the physical world, conditions vary over time. Your coffee cup was full, but now is empty. You've been exercising, and now you're more fit. The Himalayas are rising.

It's the same cup, you, and mountain range, but their conditions have changed. The real world is pervaded by this idea—what exists, will change.

Human agreement about the necessity and rightness of change is reflected in the choice of the word *variable* for use within computer programming languages. What purpose has a variable other than to vary? Most object-oriented programmers write code that both expects and relies upon object mutation. Objects are constructed, used, mutated, and then used again.

Regardless of how intuitive and natural it may seem, mutation is not an absolute requirement. It is perfectly possible (as programmers of functional languages will happily inform you) to construct applications from *immutable* objects, meaning objects that do not change. For those unused to this idea, it can be disorienting to imagine reality as constructed by the functional programmer. Instead of refilling your existing cup, you discard it in favor of a new one that looks identical but is full of coffee. Rather than changing yourself to be more fit, you swap yourself for the new, fitter, you. As the Himalayas rise, you replace your existing copy with a brand new mountain range that's a tiny bit taller.

If the idea of immutability is new to you, the examples in the prior paragraph may seem positively alarming. The first concern most folks have is for performance. The consequences of getting a whole new cup when all you want is more coffee don't seem so bad, but replacing an entire mountain range to handle a five-millimeter annual height change may feel excessive.

The next section will delve into those considerations, so defer performance concerns for a moment. For now, ponder the benefits of working with objects that do not change. What virtue might immutability provide, and what trouble might it avoid?

One of the best things about immutable objects is that they are easy to understand and reason about. These objects never start out one way and then secretly morph into something else. You can be confident that what you see at creation time is always what you get later.

Because they are easy to reason about, immutable objects are also easy to test. Objects that change need tests for the affected behavior. The change might be caused by a collaborating object, or triggered by a distant event, so tests could need additional collaborators, or actions triggered by apparently unrelated parts of your app. Tests for immutable objects avoid this extra setup, which makes the tests cheaper to write and easier to understand.

Another key virtue of immutable objects is that they are thread safe. Some of the most pernicious bugs in multi-threaded systems involve the inadvertent changing of shared state by different threads. These bugs are often related to the timing of thread execution, and so are notoriously difficult to reproduce, as well as costly and frustrating to debug. This class of problem is entirely avoided by immutable objects. You can't break shared state if shared state doesn't change.

Therefore, there are many good reasons to prefer objects that do not mutate. You are restrained from creating them only by the habit of mutability, and the (often unquestioned) assumption that instantiating new objects will be unacceptably more costly than reusing existing ones.

Having read this section, look back at the new `BottleNumber` class in Listing 5.21: Forward Messages to Smarter Number. The question of mutability applies directly to this new class. Imagine that you're holding onto an instance of `BottleNumber` whose `number` property contains the value `99`. The verse progresses such that it now needs bottle number 98. Is it better to mutate the value of `number` in the current instance of `BottleNumber`, or should that object be discarded in favor of `new BottleNumber(98)`?

If you lean towards mutating the existing `BottleNumber` rather than making another, it's possible that you are biased against creating new objects. This bias is often unexamined, and has its roots in the assumption that if you routinely create many new objects, your application will be too slow.

5.4. Assuming *Fast Enough*

The benefits of immutability are so great that, *if it were free*, you'd choose it every time. Immutability's offsetting costs are twofold. First, you must become reconciled to the idea, which for many programmers is no small thing. Next, achieving immutability requires the creation of more (sometimes many more) new objects.

Getting habituated to a new way of thinking need happen only once, so this cost is not a permanent concern; drinking the immutability Kool-Aid today suffices for forever. The ongoing costs of immutability are therefore mostly in the creation of new objects, and that's the topic of this section.

You may be familiar with Phil Karlton's famous saying "There are only two hard things in Computer Science: cache invalidation and naming things." You've already read a great deal about naming things, and it's finally time to discuss caching.

A *cache*, in computer science, is a local copy of something stored elsewhere. Saving a local copy of the results of an expensive operation, or *caching* it, is assumed to increase the speed of your application, and so lower costs.

The presumptions in the above statement are twofold. First, it assumes that caching will make applications faster, and next, it assumes that caching will lower costs. These statements are sometimes true, but not always.

When you send a message and save the result into a variable, you've created a simple cache. If the value in your variable becomes obsolete, you must invalidate this cache, either by discarding it, or by resending the message and saving the new result.

Caching is easy. However, figuring out that a cache needs to be updated can be hard. The code to do so is often complicated and confusing. This additional code must be tested, and inevitably, when it turns out that the tests are insufficient, debugged. The extra code needed to manage a cache can be so difficult to write, hard to understand, and expensive to run that it offsets the original benefits.

Notice that the costs of caching and mutation are interrelated. If the thing you cache doesn't mutate, your local copy is good forever. If you cache something that changes, you must write additional code to recognize that your copy is stale, and to re-run the initial operation to update the cache.

If you've ever worked on code that handles complicated cache invalidation, it will come as no surprise that the word itself comes from the French *cacher*, which means to conceal or hide. Outdated caches can be a source of opaque, expensive, and frustrating bugs. The net cost of caching can be calculated only by comparing the benefit of increases in speed to the cost of creating and *maintaining* the cache. If you require this speed increase, any cost is cheap. If you don't, every cost is too much.

Mutation and caching complicate code. This complication is often accepted as necessary and justified by the belief that it will improve performance. However, the unfortunate truth is that humans are very bad at predicting in advance whether a program will be fast enough overall, and, if not, which parts of it will be too slow.

Complicating code in order to solve performance problems, in advance of actual data about where those problems are, raises costs and very often pays nothing in return. These guesses are almost certain to be wrong, and merely serve to harm readability and impede change.

Given this, the best programming strategy is to write the simplest code possible and measure its performance once you're done. If the whole is not acceptably fast, profile the performance, and speed up the slowest parts. Increasing speed *may* require caching, but many problems can be fixed by substituting more efficient code in specific, narrow places. Once you understand precisely what's wrong, it may be possible to fix it without caching at all.

Your goal is to optimize for ease of understanding while maintaining performance that's fast enough. Don't sacrifice readability in advance of having solid performance data. The first solution to any problem should avoid caching, use immutable objects, and *treat object creation as free*. This results in speedy development of simple code, which leaves plenty of time to identify and correct the real performance problems.

Now that this somewhat theoretical discussion is complete, it's time return to the `Bottles` class, and apply ideas to actual code.

5.5. Creating BottleNumbers

Even for those comfortable with object creation, the code in `Bottles` constructs a notable number of `BottleNumbers`. Examine the methods below, and count the number of times a new `BottleNumber` is created by `verse`.

Listing 5.22: Lots of New BottleNumbers

```
class Bottles {
  // ...
  verse(number) {
    return (
      `${capitalize(this.quantity(number))} ` +
        `${this.container(number)} ` +
        'of beer on the wall, ' +
      `${this.quantity(number)} ` +
        `${this.container(number)} ` +
        'of beer.\n' +
      `${this.action(number)}, ` +
      `${this.quantity(this.successor(number))} ` +
        `${this.container(this.successor(number))} ` +
        'of beer on the wall.\n'
    );
  }

  quantity(number) {
    return new BottleNumber(number).quantity();
  }

  container(number) {
    return new BottleNumber(number).container();
  }

  action(number) {
    return new BottleNumber(number).action();
  }

  successor(number) {
    return new BottleNumber(number).successor();
  }
}
```

In the code above, a new instance of `BottleNumber` is created each time `quantity`, `container`, `action`, or `successor` are invoked. The `verse` method sends those messages a total of nine times. Therefore, over the course of the song, 900 new instances of `BottleNumber` are created (nine each in 100 verses).

This may feel excessive.

This plethora of object creation is the result of the prior refactoring. The recipe replaces the body of each original method with code that forwards the message to a new instance of the newly-extracted class.

Within `Bottles`, `verse` is the only method that sends the `quantity`, `container`, `action`, or `successor` messages, so the presence of these forwarding methods may seem like overkill. In this simple example, they probably are. In more complicated problems, however, it would not be surprising to perform an *Extract Class* refactoring and find that the resulting forwarding messages were invoked many times, from many different methods within the original class. These forwarding methods exist to provide a single place for the original class to catch these messages when sent to itself, and funnel them along to the new class.

The previous refactoring recipe makes no attempt to minimize the number of new objects, and creates a set of forwarding methods that unabashedly create new instances of the extracted class. The upshot is 900 new `BottleNumber`s.

This code works, and if you find it distressing, it's likely because it feels wasteful. There *are* alternatives. If unconstrained by the recipe, there are a number of ways to avoid such profligate object creation, and it's instructive to consider them.

For example, the first three phrases of the first verse of the song send `quantity` and `container` twice, and `action` once. This creates five instances of `BottleNumber` for the `number` 99. If the first instance were to be cached, it could be re-used four times in these three phrases.

The fourth phrase of verse 99 sends `successor` twice, which creates two additional instances of `BottleNumber` 99. The previously cached bottle number could be used here also. Therefore, `BottleNumber` 99 could be created once, and then reused six times.

The fourth phrase of verse 99 also sends `quantity` and `container`. This creates two instances of `BottleNumber` on the successor, which is 98. Caching the first instance would save another object creation within this verse. Additionally, the cached copy could be re-used in the *following* verse, saving seven more object creations for a total of eight altogether. Over the course of the song, caching could reduce the number of new `BottleNumber` instances from 900 to 100.

For those who feel the need to be even more parsimonious, it's possible to create a single instance of `BottleNumber` and reuse it 900 times. To accomplish this, one would create a `BottleNumber` for the `number` 99, and then, when the need for bottle number 98 arose, change the value of `number` from 99 to 98 in that one existing object. And just like that, you've added caching *plus* mutation.

So you can reduce the number of new `BottleNumber`s by caching existing ones, and decrease this number further if you're willing to mutate them. Doing either of these things may lower some costs, but will certainly raise others. These things are not free.

As a thought exercise, take a minute before reading on and imagine altering the existing code to use a single instance of `BottleNumber`. If you find that exercise easy, try another, this time pretending that `quantity`, `container`, `action`, and `successor` are sent from multiple methods within `Bottles`. Pause a moment if you care to, and go write the code. You'll find that the changes needed to do this add complexity. This complexity may cost more than the benefit gained by faster performance.

Having done that experiment, return to the problem at hand. In this example, the forwarding methods are invoked from only one method of `Bottles`. This means that it's possible to reduce object creation by adding a simple, automatically-invalidating, low-cost cache. The following example shows a `BottleNumber` being cached on line 4:

Listing 5.23: Caching a BottleNumber

```
1 class Bottles {
2   // ...
3   verse(number) {
4     const bottleNumber = new BottleNumber(number);
5
6     return (
7       `${capitalize(this.quantity(number))} ` +
8         `${this.container(number)} ` +
9         'of beer on the wall, ' +
10      `${this.quantity(number)} ` +
11        `${this.container(number)} ` +
12        'of beer.\n' +
13      `${this.action(number)}, ` +
14      `${this.quantity(this.successor(number))} ` +
15        `${this.container(this.successor(number))} ` +
16        'of beer on the wall.\n'
17    );
18  }
19  // ...
20 }
```

Line 4 above creates a new instance of `BottleNumber` and caches it in a temporary variable (this is the *Temporary Variable* code smell) within the `verse` method. This cache reduces object creation without adding much additional complexity, so here it's justified because the benefits outweigh the costs.

Now that this cached object exists, you can gradually alter the verse template to send messages to the new object rather than to `this`. The next example begins the transition with the simplest change possible. Line 6 below asks this new object for its `action`:

Listing 5.24: Asking the Cached Object for Its Action

```
1 verse(number) {
2   const bottleNumber = new BottleNumber(number);
3
4   return (
5     // ...
6     `${bottleNumber.action()}, ` +
7     // ...
8   );
9 }
```

In the code above, `this.action(number)` has been replaced by `bottleNumber.action()`. This sends the `action` message directly to the new `BottleNumber`, entirely bypassing the local implementation.

A similar change can be made in the first and second phrases of the `verse` template, as shown below:

Listing 5.25: Using the Cached Object in Phrases 1 and 2

```
verse(number) {
  const bottleNumber = new BottleNumber(number);

  return (
    `${capitalize(bottleNumber.quantity())} ` +
      `${bottleNumber.container()} ` +
      'of beer on the wall, ' +
    `${bottleNumber.quantity()} ` +
      `${bottleNumber.container()} ` +
      'of beer.\n' +
    `${bottleNumber.action()}, ` +
    // ...
  );
}
```

In lines 5-11 of the code above, `quantity` and `container` are now sent directly to `bottleNumber`. This, again, bypasses the local implementations in favor of sending messages to the cached object.

Now the first three phrases of the verse template send messages to a `BottleNumber` rather than to `this`. Only phrase four remains to be updated.

5.6. Recognizing Liskov Violations

Phrases 1 through 3 of the verse template refer to the same bottle number, and so can share the currently-cached `BottleNumber` instance. Phrase 4, however, uses a different bottle number. Here's a reminder of the code:

Listing 5.26: Current Phrase 4

```
verse(number) {
  const bottleNumber = new BottleNumber(number);

  return (
    // ...
    `${this.quantity(this.successor(number))} ` +
      `${this.container(this.successor(number))} ` +
      'of beer on the wall.\n'
  );
}
```

The plan is to change phrase 4 to send messages to instances of `BottleNumber` rather than to `this`. Previously, when making a similar change to phrase 1 and 2,

```
this.quantity(number)
```

was replaced with

```
bottleNumber.quantity()
```

On line 6 above, phrase 4 also invokes `quantity`, but it passes a different argument than does phrase 1:

```
this.quantity(this.successor(number))
```

The `quantity` method above is passed `this.successor(number)` because phrase 4 is about the *next* number. For example, in a verse where phrase 1 is about number 99, then phrase 4 is about number 98.

The goal here is to send the `quantity` message to an object that can answer correctly, and the problem is that you do not yet have access to such an object.

`BottleNumber`s implement `successor`, and it feels as if `successor` should return the object you need. Your object-oriented intuition is bang on[15] if you expect the `successor` of a `BottleNumber` to be another `BottleNumber`. If this were true, you could replace:

```
this.quantity(this.successor(number))
```

with:

```
bottleNumber.successor().quantity()
```

Unfortunately, as is, this code doesn't work. If you make the above change and run the tests, you'll see:

```
TypeError: bottleNumber.successor(...).quantity is not a function
```

The problem is that `successor` still returns a number, when logically it should now return the succeeding `BottleNumber`. `BottleNumber`s know `quantity`, but numbers do not.

Back when `successor` was first created, it was correct for it to return a number. This abstraction was identified by the Flocking Rules, which called for copying code from the old `verse` `switch` statement into the new `successor` method. The `switch` statement originally returned numbers, thus the `successor` method did the same. At that point, `successor` *was* a number.

However, the `successor` method has moved to a new class, and the concept once represented by a number is now represented by a `BottleNumber`. The *type* of the object has changed, but the `successor` method still returns the old type. You have every right to expect any method named `successor` to return an object that implements the same API as the receiver, but alas, this `successor` method does not.

This inconsistency is another violation of the generalized Liskov Substitution Principle. A method named `successor` implicitly promises that the thing it returns will behave like the object to which you sent the message. But this `successor` method lies. It breaks its promise, which forces the sender to know that the return is untrustworthy and to take steps to handle the violation.

As annoying as this is, you are in the middle of altering the `verse` template to send messages to objects. This current refactoring is almost complete, and it is often better to finish horizontal refactorings before undertaking vertical tangents. You *could* veer from the path and fix the Liskov violation, but in the spirit of completing the current thought before undertaking a new task, stay the course. You've already declared a temporary variable to hold bottle number 99. The

current problem can be solved by declaring another variable to hold bottle number 98 and writing some shameless code. On lines 3-4 below, the following example bravely does just that:

Listing 5.27: Caching the Successor

```
1 verse(number) {
2   const bottleNumber = new BottleNumber(number);
3   const nextBottleNumber =
4     new BottleNumber(bottleNumber.successor());
5
6   return (
7     `${capitalize(bottleNumber.quantity())} ` +
8       `${bottleNumber.container()} ` +
9       'of beer on the wall, ' +
10    `${bottleNumber.quantity()} ` +
11      `${bottleNumber.container()} ` +
12      'of beer.\n' +
13    `${bottleNumber.action()}, ` +
14    `${nextBottleNumber.quantity()} ` +
15      `${nextBottleNumber.container()} ` +
16      'of beer on the wall.\n'
17  );
18 }
```

Line 4 above creates a new `BottleNumber` on the successor of the existing `BottleNumber`. Ultimately, you'd like to improve this line of code, but at present it suffices to move the current refactoring forward. Now that `nextBottleNumber` exists, lines 14-15 can ask it for its `quantity` and `container`.

After that change, the `verse` method contains two distinct parts. Lines 7-16 above define a template which queries instances of `BottleNumber` for details. Lines 2 and 4 create new instances of `BottleNumber`. Line 2 seems reasonable, but line 4 is awkward because the Liskov violation forces you to invoke `successor` and then convert its return into a `BottleNumber` yourself.

This completes the caching of `BottleNumber`s in the `verse` method, but there's one final change to make. Now that `verse` talks directly to objects cached in temporary variables, the forwarding methods are no longer needed. Deleting them reduces the code to the following:

Listing 5.28: Obsession Cured

```
1 class Bottles {
2   song() {
3     return this.verses(99, 0);
4   }
5
6   verses(starting, ending) {
7     return downTo(starting, ending)
8       .map(i => this.verse(i))
9       .join('\n');
10  }
11
12  verse(number) {
13    const bottleNumber = new BottleNumber(number);
14    const nextBottleNumber =
15      new BottleNumber(bottleNumber.successor());
16
17    return (
```

```
18 |       `${capitalize(bottleNumber.quantity())} ` +
19 |         `${bottleNumber.container()} ` +
20 |         'of beer on the wall, ' +
21 |       `${bottleNumber.quantity()} ` +
22 |         `${bottleNumber.container()} ` +
23 |         'of beer.\n' +
24 |       `${bottleNumber.action()}, ` +
25 |       `${nextBottleNumber.quantity()} ` +
26 |         `${nextBottleNumber.container()} ` +
27 |         'of beer on the wall.\n'
28 |     );
29 |   }
30 |}
31 |
32 |class BottleNumber {
33 |   constructor(number) {
34 |     this.number = number;
35 |   }
36 |
37 |   quantity() {
38 |     if (this.number === 0) {
39 |       return 'no more';
40 |     } else {
41 |       return this.number.toString();
42 |     }
43 |   }
44 |
45 |   container() {
46 |     if (this.number === 1) {
47 |       return 'bottle';
48 |     } else {
49 |       return 'bottles';
50 |     }
51 |   }
52 |
53 |   action() {
54 |     if (this.number === 0) {
55 |       return 'Go to the store and buy some more';
56 |     } else {
57 |       return (
58 |         `Take ${this.pronoun()} down and pass it around`
59 |       );
60 |     }
61 |   }
62 |
63 |   pronoun() {
64 |     if (this.number === 1) {
65 |       return 'it';
66 |     } else {
67 |       return 'one';
68 |     }
69 |   }
70 |
71 |   successor() {
72 |     if (this.number === 0) {
73 |       return 99;
74 |     } else {
75 |       return this.number - 1;
76 |     }
```

This completes the extraction of the `BottleNumber` class, resolves the *Primitive Obsession* code smell, and heralds the end of Chapter 5.

5.7. Summary

This chapter continued the quest to make `Bottles` open to the six-pack requirement. It recognized that many methods in `Bottles` obsessed on `number`, and undertook the *Extract Class* refactoring to cure this obsession. The refactoring created a new class named `BottleNumber`.

During the course of the refactoring, conditionals were examined from an experienced OO practitioners' point of view. This chapter also explored the rewards of modeling abstractions, the trade-offs of caching, the advantages of immutability, and the benefits of deferring performance tuning.

Most programmers are happier with the current code than they were with Shameless Green, but this version is far from perfect. The total ABC score score, for example, has gone up again. From the metrics point of view, after turning one conditional into many back in Chapter 4, you've now compounded your sins by introducing a new class which adds no new behavior but increases the length of the code.

Also, there are no unit tests for `BottleNumber`. It relies entirely on `Bottle`'s tests.

The code still exudes many smells (duplication, conditionals, and temporary field, to name a few). And, finally, it commits a Liskov violation in the `successor` method.

The refactorings in this and the prior chapter were undertaken in hopes of making the code open to the six-pack requirement, but this has not yet succeeded. You've been acting in faith that removing code smells would eventually lead to openness. It's possible that your faith is being tested.

Despite the imperfections listed above, there are ways in which the code *is* better. There are now two classes, but each has focused responsibilities. While it's true that the whole is bigger, each part is easy to understand and reason about.

The code is consistent and regular, and embodies an extremely stable landing point that splendidly enables the next refactoring.

With that, on to Chapter 6.

6. Achieving Openness

Despite much refactoring, the code is still not open to the six-pack requirement. Once again, you must decide whether to continue forward with the existing code, or to retreat and strike out in a different direction.

Consider the code's present state. `BottleNumber` now contains methods that *isolate* the things that need to change. If you were willing to abandon the quest for openness and directly alter the code, you could fulfill the six-pack requirement by simply adding another branch to the conditionals in the `quantity` and `container` methods. When the value of number is `6`, the quantity could be changed to return "1," and container changed to return "six-pack."

This increasing isolation of the concepts that need to vary is an indication that the code is moving in the right direction. In optimism, then, this chapter continues forward. It removes a *Data Clump*, deals with the conditionals in `BottleNumber`, introduces a *Factory*, fixes a *Liskov* violation, and ultimately, fulfills the six-pack requirement.

6.1. Consolidating Data Clumps

The `BottleNumber` class contains conditionals, and removing them would make the code easier to understand and cheaper to maintain. Before focusing on that problem, however, there's a simpler code smell that can be addressed.

The `verse` method contains two things that always appear together. Have a look at the code (repeated below) and see if you can identify them:

Listing 6.1: Quantity and Container Form a Data Clump

```
1 class Bottles {
2   // ...
3   verse(number) {
4     const bottleNumber = new BottleNumber(number);
5     const nextBottleNumber =
6       new BottleNumber(bottleNumber.successor());
7
8     return (
9       `${capitalize(bottleNumber.quantity())} ` +
10        `${bottleNumber.container()} ` +
11        'of beer on the wall, ' +
12      `${bottleNumber.quantity()} ` +
13        `${bottleNumber.container()} ` +
14        'of beer.\n' +
15      `${bottleNumber.action()}, ` +
16      `${nextBottleNumber.quantity()} ` +
17        `${nextBottleNumber.container()} ` +
18        'of beer on the wall.\n'
19    );
20  }
21 }
```

Above, `quantity` and `container` appear together in three different places (lines 9-10, 12-13, and 16-17). The duplication of this pairing gives off a whiff of the *Data Clump* code smell. As the

name implies, *Data Clump* is officially about *data*, and is defined as the situation in which several (three or more) data fields routinely occur together.

Having a clump of data usually means you are missing a concept. When a clump gets sent as a set of parameters, the method that receives the clump can easily become polluted with clump management logic. If more than one method takes the same clump as input, some or all of this management logic will inevitably get duplicated in several places. Not only is it a pain to maintain this duplication, but over time the logic might accidentally diverge, introducing errors and confusing everyone involved.

In the present case, it's a slight stretch to call the `quantity` and `container` pairing above a *Data Clump*, but the value of removing clumps is so great that it makes sense to view this code through that lens. If these two things always appear together, it's a signal that this pairing represents a deeper concept, and that concept should be named.

Full-grown *Data Clumps* are usually removed by extracting a class, but in this small example it makes sense to simply create a new method. As always, the method should be given a name that reflects its purpose. If you're willing to take a bit of license defining this purpose, you can give the method a name that has the side effect of greatly simplifying `verse`.

This side effect requires a brief explanation. JavaScript string interpolation has a quality that you may not have considered, but which you surely already rely upon. When JavaScript finds a `${ }` within a backtick-quoted string, it evaluates the code between the curly braces, and then replaces the entire `${ }` bit with the result of that evaluation. This works, regardless of the *type* of the result. For example, the statement:

```
`five plus three = ${5+3}`
```

returns the string:

```
'five plus three = 8'
```

Interpolation works, regardless of the return type of the evaluated code, because JavaScript sends `toString` to the result of `${ }` before substituting that result into the string.

It's perfectly acceptable to override this default behavior, and many of your own classes would benefit from a custom `toString` implementation. In the current situation, if you were to implement `toString` on `BottleNumber`, you might do so as shown below:

Listing 6.2: BottleNumber Provides a String Representation

```
class BottleNumber {
  // ...
  toString() {
    return `${this.quantity()} ${this.container()}`;
  }
  // ...
}
```

Having done the above, you can now replace the `quantity`/`container` clump with a simple `toString` message send. For example, the third phrase of the `verse` template currently says:

```
`${bottleNumber.quantity()} ` +
  `${bottleNumber.container()} ` +
  'of beer.\n' +
```

Now that the custom `toString` exists, you can reduce this code to:

```
`${bottleNumber.toString()} of beer.\n`
```

However, as JavaScript is already sending `toString` to the result of the interpolation, you can further reduce the code to:

```
`${bottleNumber} of beer.\n`
```

You can make a similar change anywhere `quantity` and `container` are interpolated together into a string. Here's the entire verse method, with the clumps replaced by an implicit call to `toString` on `bottleNumber`:

Listing 6.3: Verse With Data Clumps Removed

```
1 class Bottles {
2   // ...
3   verse(number) {
4     const bottleNumber = new BottleNumber(number);
5     const nextBottleNumber =
6       new BottleNumber(bottleNumber.successor());
7
8     return (
9       capitalize(`${bottleNumber} `) +
10        'of beer on the wall, ' +
11      `${bottleNumber} ` +
12        'of beer.\n' +
13      `${bottleNumber.action()}, ` +
14      `${nextBottleNumber} ` +
15        'of beer on the wall.\n'
16    );
17  }
18 }
```

Notice that on line 9 above the `capitalize` message has been removed from the interpolation and placed outside of the string. This is both necessary (try changing the original code to `` `${capitalize(bottleNumber)}` `` if you doubt this) and arguably more correct. After all, capitalization happens to the first word of a sentence, not to a `bottleNumber`.

Removing the clump shortens the lines so much that the code can be reformatted to more accurately reflect the song. The four phrases of a verse can be seen more clearly now, as shown on lines 9-12 below:

Listing 6.4: Verse Method Template in Four Phrases

```
1 class Bottles {
2   // ...
3   verse(number) {
4     const bottleNumber = new BottleNumber(number);
5     const nextBottleNumber =
6       new BottleNumber(bottleNumber.successor());
7
8     return (
9       capitalize(`${bottleNumber} of beer on the wall, `) +
```

```
      `${bottleNumber} of beer.\n` +
      `${bottleNumber.action()}, ` +
      `${nextBottleNumber} of beer on the wall.\n`
    );
  }
}
```

Using `toString` to remove the `quantity`/`container` pair reduces the amount of code in the verse template, but, admittedly, comes perilously close to abusing the intent of `toString`. One could defensibly argue that this `toString` implementation is so specific to `verse`'s current needs that it is ill-suited for use in other situations.

In its defense, the current `toString` maximizes the effect of removing the *Data Clump* in `verse`, and so this implementation provides a great illustration of the value of clump removal. In real life, you might need a more general implementation of `toString`, in which case you'd give the new method a different name, and then explicitly send that message from within the verse template string.

The `verse` method is getting simpler, but it still has more than one responsibility. This problem is reflected by the very structure of the code—*the above method contains a blank line*. Programmers add blank lines to signify changes of topic. The presence of multiple topics suggests the existence of multiple responsibilities, which makes code harder to understand when reading, and easier to harm when changing.

Despite the fact that the `verse` method does more than one thing, it *is* improved. Its template now contains four lines, which echoes the four phrases in every verse. Each template line is short enough to display without wrapping on most reading devices. This method isn't perfect, but removing the data clump improves its readability and sheds light on its intentions.

6.2. Making Sense of Conditionals

Now that `Bottles`'s `quantity`/`container` clump is resolved, it's time to identify the next code smell.

Switch your attention to the `BottleNumber` class. It's full of conditionals, all of which have the same shape. Here's that code:

Listing 6.5: BottleNumber

```
class BottleNumber {
  constructor(number) {
    this.number = number;
  }

  toString() {
    return `${this.quantity()} ${this.container()}`;
  }

  quantity() {
    if (this.number === 0) {
      return 'no more';
    } else {
      return this.number.toString();
    }
```

```
  }

  container() {
    if (this.number === 1) {
      return 'bottle';
    } else {
      return 'bottles';
    }
  }

  action() {
    if (this.number === 0) {
      return 'Go to the store and buy some more';
    } else {
      return (
        `Take ${this.pronoun()} down and pass it around`
      );
    }
  }

  pronoun() {
    if (this.number === 1) {
      return 'it';
    } else {
      return 'one';
    }
  }

  successor() {
    if (this.number === 0) {
      return 99;
    } else {
      return this.number - 1;
    }
  }
}
```

The conditionals above are much like the ones vociferously objected to in the Insisting Upon Messages section of Chapter 5. The difference is that they now depend on the `number` property, whereas they previously depended on a `number` *argument*.

A brief review of that transition may be helpful. Here's a sample of how the methods looked in Chapter 5, when they depended on the `number` argument:

Listing 6.6: Original Container Method Takes Number Argument

```
class Bottles {
  // ...
  container(number) {
    if (number === 1) {
      return 'bottle';
    } else {
      return 'bottles';
    }
  }
  // ...
}
```

Chapter 5 argued that instead of injecting an object and conditionally supplying it with behavior, you should instead arrange code such that you can merely forward the message to the injected

object. The code below shows, hypothetically, how that might look:

Listing 6.7: Wishful Container Method

```
class Bottles {
  // ...
  container(smarterNumber) {
    return smarterNumber.container();
  }
  // ...
}
```

And indeed, Chapter 5 introduced a method very much like the one shown above. In that chapter, a `Bottles` `container(number)` forwarding method appeared early in the refactoring and lived a brief (but useful) life. Ultimately, the code was changed to cache `BottleNumber`s inside the `Bottles` `verse` method, rendering all of the forwarding methods obsolete, and leading to their deletion.

So, Chapter 5 held forth against conditionals, recognized the dependency on a repeatedly-passed argument, identified the *Primitive Obsession* code smell, and extracted the `BottleNumber` class to cure the obsession.

Extracting `BottleNumber` certainly removed the conditionals from `Bottles`, but they didn't disappear: *they just moved* to the newly extracted class. While slightly improved in that the methods now use the `number` property rather than than taking a `number` argument, they all (excepting `toString`) still contain conditionals. These conditionals characterize the class, and make *Switch Statement* the most identifiable code smell.

Fowler offers several curative refactoring recipes. The two main contenders are *Replace Conditional with State/Strategy* and *Replace Conditional with Polymorphism*.

The *Replace Conditional with State/Strategy* recipe removes conditionals by dispersing their branches into new, smaller objects, one of which is later selected and plugged back in at runtime. This recipe results in a code arrangement known as *composition*.

The *Replace Conditional with Polymorphism* recipe removes conditionals by creating one class to hold the defaults of the conditionals (the `false` branches), and adding subclasses for each specialization (the `true` branches of the various conditions). It then chooses one of these new objects to plug back in at runtime. This recipe solves the conditional problem using *inheritance*.

You can be forgiven if you find the above descriptions very similar—they are. Both recipes result in new objects that hold logic harvested from the branches of the conditionals. The main difference is that the *Polymorphism* recipe uses inheritance, and the *State/Strategy* recipe does not.

Replace Conditional with State/Strategy produces interesting results, and you are encouraged to experiment with that recipe on your own. However, *Replace Conditional with Polymorphism* leads to a code arrangement that's felicitous for the six-pack problem, and so will be followed in the next section.

The previous assertion that one recipe leads to better results than another may have piqued your curiosity. Had you been working this problem alone, how would you have known which to choose?

Skilled programmers are good at picking what best to do next. For many problems, they can immediately identify the code smell that will be most fruitful to resolve. They have excellent judgement. Their decision-making process looks intuitive and effortless, but also inimitable, which makes their actions simultaneously inspiring and disheartening. It's as if they have a secret understanding of the underlying patterns of code, one not granted to mere mortals.

Despite appearances, these programmers weren't born with magical talents. Their powerful intuition isn't innate—it's the result of a lifetime of coding experiments. Their present-day actions are informed by a diverse body of knowledge gained piecemeal, over time. Their deep familiarity with many varieties of code entanglements allows them to unconsciously map appropriate solutions onto common problems, often without the necessity of first writing code. They know what's right *before* they do it.

Or at least they do, sometimes. They also know that they don't know everything. This belief in their own fallibility lends them caution. Skilled programmers do what's right when they intuit the truth, but otherwise they engage in careful, precise, reproducible, and reversible coding experiments. You are encouraged to do the same.

The best way to figure out what will happen if you follow competing recipes *is to try it*. If battling this problem alone, tentatively identify *Switch Statement* as the code smell, look up the curative refactorings, and then, speculatively, try them all. Evaluate the results. Choose one and proceed, or revert all and try again.

Practice *builds* intuition. Do it enough, and you'll seem magical too.

6.3. Replacing Conditionals with Polymorphism

It's now time for the object-oriented miracle that turns condition-laden classes into sets of independent objects.

This miracle relies on "polymorphism," a word which combines "poly" (many) with "morphs" (forms). In OO, polymorphism refers to the idea of having many different kinds of objects that respond to the same message. *Senders* of the message needn't care with which of the possible receivers they are communicating. Polymorphism allows senders to depend on the *message* while remaining ignorant of the *type*, or class, of the receiver. Senders don't care what receivers are; instead, they depend on what receivers do.

6.3.1. Dismembering Conditionals

The current code is not polymorphic. Not only does `Bottles`'s `verse` method explicitly reference the concrete `BottleNumber` class, but `BottleNumber` itself contains many methods comprised of conditionals that return varying results. If this code *did* rely on polymorphism, the logic in those conditionals would be dispersed across several different kinds of objects, and `verse` would be ignorant of `BottleNumber`'s existence.

Polymorphism, *by definition*, involves more than one kind of object, so changing from a procedural to a polymorphic code arrangement will increase the overall number of classes. This, in turn, will force you to add new code that is aware of the existence of these new classes, and that understands which class works for what condition. Thus, as conditionals disappear from `BottleNumber`, new dependencies will arise. These new dependencies can make a mess of code, and so are managed carefully in the examples that follow.

With that, onward with removing conditionals. Recall that `BottleNumber` contains the following methods:

Listing 6.8: BottleNumber Concepts

```
class BottleNumber {
  toString() {
  quantity() {
  container() {
  action() {
  pronoun() {
  successor() {
}
```

These methods serve as a list of bottle-ish concepts. The method implementations (with the exception of `toString`) share the following shape:

Listing 6.9: BottleNumber Conditional Shape

```
class BottleNumber {
  // ...
  quantity() {
    if (this.number === 0) {
      return 'no more';
    } else {
      return this.number.toString();
    }
  }
  // ...
}
```

Methods like the above were created by following the *Flocking Rules*, and then simplified during the *Extract Class* refactoring. This conditional represents an extremely stable landing point. Once you get code into this shape, it's time to celebrate—the problem is nearly solved.

`BottleNumber` represents a smart, bottle-ish kind of number. Its logic is correct in most cases for most numbers, but not yet in every case for all. A few *specific* numbers are not yet smart enough. The consistency of the code makes it is easy to see just which numbers are lacking. The code extract below contains a very broad hint:

Listing 6.10: Some Numbers Are Special

```
class BottleNumber {
  // ...
  quantity() {
    if (this.number === 0) {
  // ...
  container() {
    if (this.number === 1) {
  // ...
```

```
  action() {
    if (this.number === 0) {
  // ...
  pronoun() {
    if (this.number === 1) {
  // ...
  successor() {
    if (this.number === 0) {
  // ...
}
```

The above makes it clear that `0` and `1` are special, and need to be smarter. The fact that this is so visible is a tribute to the benefits of checking for equality.

This code is reminiscent of primitive obsession. Here, however, the fixation is on a specific number (`0` or `1`) rather than on numbers as a whole. Obsessions are usually cured by extracting a class, and if you suspect that class extraction is called for here, you are correct.

Each conditional supplies *specific* behavior in its `true` branch and *generalized* behavior in its `false`. If you were to go into the methods and delete everything but the bodies of the `false` branches, what remained in `BottleNumber` would work for all numbers *except* for `0` and `1`. Doing so, of course, would break the tests, but at least it would leave `BottleNumber` itself free of conditionals.

Removing the conditionals without breaking the tests requires a process that carefully and systematically moves the code from each `true` branch into a new object, rather than willy-nilly deleting it. The specific logic for `0` needs to be isolated in a class of its own, as does the logic for `1`. Also, as these new classes come into existence, some additional code will have to be written to choose the correct class based on the value of `number`.

This transition is safely accomplished by the *Replace Conditional With Polymorphism* recipe. To begin, choose one of the values being explicitly tested for in one of the conditionals. All things being equal, it's reasonable to start with `0`.

Next, decide on a name for the bottle number class that will stand in for a smarter `0`. For reasons that will eventually become clear, it's expedient to name this new class `BottleNumber0`.

Having made these decisions, the next step is to create `BottleNumber0` as an empty subclass of `BottleNumber`. Here's that code:

Listing 6.11: Empty BottleNumber0 Class

```
class BottleNumber0 extends BottleNumber {
}
```

As previously stated, this recipe uses inheritance. Modern object-oriented programming is biased towards preferring composition over inheritance. However, this bias shouldn't be taken to mean that the use of inheritance is banned. The current recipe calls for it, and for the problem at hand, inheritance supplies a straightforward, cost-effective solution.

Next, copy (not cut!) one of the methods that obsesses on `0` from `BottleNumber` to `BottleNumber0`. The `quantity` method is chosen here:

Listing 6.12: BottleNumber0 Duplicates Quantity Method

```
class BottleNumber {
  // ...
  quantity() {
    if (this.number === 0) {
      return 'no more';
    } else {
      return this.number.toString();
    }
  }
  // ...
}

class BottleNumber0 extends BottleNumber {
  quantity() {
    if (this.number === 0) {
      return 'no more';
    } else {
      return this.number.toString();
    }
  }
}
```

Continuing, remove the part of `BottleNumber0`'s `quantity` method that *isn't* about `0`. This means you'll need to delete everything but the body of the `true` branch, as shown here:

Listing 6.13: BottleNumber0 Returns Correct Result

```
class BottleNumber0 extends BottleNumber {
  quantity() {
    return 'no more';
  }
}
```

`BottleNumber0` plays the bottle number *role* just as accurately as does `BottleNumber`. Unfortunately, despite the fact that there are now two equally valid players of this role, the current `verse` method is willing to use only one of them. It is tightly coupled to `BottleNumber`, which it explicitly references twice, as shown below:

Listing 6.14: Verse Method Knows BottleNumber Class Name

```
class Bottles {
  // ...
  verse(number) {
    const bottleNumber = new BottleNumber(number);
    const nextBottleNumber =
      new BottleNumber(bottleNumber.successor());
    // ...
  }
}
```

Lines 4 and 6 above both create instances of `BottleNumber`. Now that you're breaking conditionals apart, in some cases you now actually need an instance of `BottleNumber0`.

One way to ensure the right *kind* of bottle number is to alter the code to select the class based on the value of `number`, as does this next example:

```
const bottleNumber =
  new (number === 0 ? BottleNumber0 : BottleNumber)(number);

const succ = bottleNumber.successor();
const nextBottleNumber =
  new (succ === 0 ? BottleNumber0 : BottleNumber)(number);
```

This works, but it's certainly not optimal. It introduces a new, duplicated conditional into an exercise whose entire point is to remove them. This change would be counterproductive. Instead, now that more than one class plays the role of bottle number, you need shared logic to choose the correct one.

6.3.2. Manufacturing Objects

When several classes play a common role, some code, somewhere, must know how to choose the right role-playing class for any specific contingency. This *choosing* very often involves a conditional, which should exist in one and only one place. Code like this is said to "manufacture" an instance of the right kind of object, and so is commonly referred to as a *factory*. Chapter 7 takes on factories in greater detail, but for now think of a factory as a method whose job is to return the right role-playing object.

When a factory exists for a role, the factory has sole responsibility for creating objects to play that role. The factory's purpose is to isolate the names of the concrete classes, and to hide the logic needed to choose the correct one. After creating a factory, you may not refer to the names of these classes, or duplicate this choosing logic, in other parts of your application.

Now that `BottleNumber0` exists, you need a bottle number factory. The first step is to do a small refactoring to isolate the creation of bottle numbers in a single method of `Bottles`.

Here's the new method:

Listing 6.15: Simple Bottle Number Factory

```
class Bottles {
  // ...
  bottleNumberFor(number) {
    return new BottleNumber(number);
  }
}
```

The `bottleNumberFor` method inserts a level of indirection between the desire for a `BottleNumber` and its creation. It introduces a seam into the code, which makes it possible to change how the factory works without fear of breaking its invokers. It is the factory's responsibility to manufacture the right object, and the responsibility of all other code to query the factory for bottle numbers.

Once `bottleNumberFor` exists, the `verse` method can be changed to invoke it, as shown here:

Listing 6.16: Verse Method Knows About the Factory

```
class Bottles {
  // ...
  verse(number) {
    const bottleNumber = this.bottleNumberFor(number);
    const nextBottleNumber =
      this.bottleNumberFor(bottleNumber.successor());
    // ...
  }
}
```

The `bottleNumberFor` method has assumed responsibility for manufacturing bottle numbers. So far this has been a straightforward refactoring, but now that the new method is in place, you can extend its behavior.

The following example changes the factory to take the value of `number` into account when choosing which kind of bottle number to return:

Listing 6.17: Factory Method

```
class Bottles {
  // ...
  bottleNumberFor(number) {
    if (number === 0) {
      return new BottleNumber0(number);
    } else {
      return new BottleNumber(number);
    }
  }
}
```

The above code works, but it's not perfect. The problem is that the branches of the conditional combine things that differ (`BottleNumber0` and `BottleNumber`) with things that remain the same (`return new …(number)`). This conflation forces the reader to study the code to discern the difference.

The following alternative takes a different tack:

Listing 6.18: If Statement Chooses Class

```
class Bottles {
  // ...
  bottleNumberFor(number) {
    let bottleNumberClass;
    if (number === 0) {
      bottleNumberClass = BottleNumber0;
    } else {
      bottleNumberClass = BottleNumber;
    }

    return new bottleNumberClass(number);
  }
}
```

The nice thing about this version is that it isolates the things that vary, which highlights the difference between the conditions.

Now that `verse` is invoking the factory to get the appropriate bottle number, you can remove everything but the default (`false` branch) from `BottleNumber`'s `quantity` method.

Here's the resulting code:

Listing 6.19: BottleNumber Quantity Method Reduced to Default

```
class BottleNumber {
  // ...
  quantity() {
    return this.number.toString();
  }
  // ...
}
```

At this point, the tests should still pass. The fact that they do proves that the factory is manufacturing the correct bottle number for every situation.

To briefly review, `BottleNumber`'s `quantity` method initially held a conditional that checked to see if `number` was equal to `0`. This conditional supplied general behavior in its `false` branch and behavior specifically for `0` in its `true` branch. The presence of this conditional indicated the need for a new class to stand in for `0`.

`BottleNumber` was subclassed with `BottleNumber0`, into which the `quantity` method was copied. Here's a reminder of the situation at that point:

Listing 6.20: BottleNumber0 With Duplicated Quantity Method

```
class BottleNumber {
  // ...
  quantity() {
    if (this.number === 0) {
      return 'no more';
    } else {
      return this.number.toString();
    }
  }
  // ...
}

class BottleNumber0 extends BottleNumber {
  quantity() {
    if (this.number === 0) {
      return 'no more';
    } else {
      return this.number.toString();
    }
  }
}
```

The next goal was to reduce the subclass' conditional to its `true` branch, and the superclass' to its `false`. The subclass was changed without incident, but altering the superclass caused the tests to fail. This failure exposed the need for a factory to choose between these classes. So the `bottleNumberFor` factory method was created, after which the tests again passed.

The resulting code is repeated below:

Listing 6.21: Factory Chooses Polymorphic Object

```
class Bottles {
  // ...
```

```
  verse(number) {
    const bottleNumber = this.bottleNumberFor(number);
    const nextBottleNumber =
      this.bottleNumberFor(bottleNumber.successor());
    // ...
  }

  bottleNumberFor(number) {
    let bottleNumberClass;
    if (number === 0) {
      bottleNumberClass = BottleNumber0;
    } else {
      bottleNumberClass = BottleNumber;
    }

    return new bottleNumberClass(number);
  }
}

class BottleNumber {
  // ...
  quantity() {
    return this.number.toString();
  }
  // ...
}

class BottleNumber0 extends BottleNumber {
  quantity() {
    return 'no more';
  }
}
```

The above example illustrates the power of polymorphism. `BottleNumber` and `BottleNumber0` both play the *role* of bottle number. They respond to the same messages and conform to the same API, but implement `quantity` in completely different ways.

These classes are substitutable for one another. When you invoke the factory to get a bottle number, you don't need to know the class of the returned object. You merely trust that object to act like a bottle number and to respond to the messages you plan to send.

This willful ignorance of type is fundamental to object-oriented programming. It insulates code that calls a factory from changes of implementation within that factory. By refusing to be aware of the classes of the objects with which you interact, you grant others the freedom to alter your code's behavior without editing its source. In the distant future, someone could amend the factory to return newly introduced players of the bottle number role, and your existing code would happily collaborate with these unanticipated objects.

The `quantity` method is now polymorphically implemented. It's time to move on to the remaining conditionals.

6.3.3. Prevailing with Polymorphism

You've experienced one complete round of *Replace Conditional with Polymorphism*, and the remainder of this refactoring is just more of the same. Here's a list of the recipe's steps:

1. Create a subclass to stand in for the value upon which you switch.
    a. Copy one method that switches on that value into the subclass.
    b. In the subclass, remove everything but the true branch of the conditional.
        i. *At this point, create a factory if it does not yet exist, and*
        ii. *Add this subclass to the factory if not yet included.*
    c. In the superclass, remove everything but the false branch of the conditional.
    d. Repeat steps a-c until all methods that switch on the value are dispersed.
2. Iterate until a subclass exists for every different value upon which you switch.

Following those steps for the `action` and `successor` methods (both of which test for `0`) results in the following code:

Listing 6.22: 0 Has Its Own Class

```
class BottleNumber {
  constructor(number) {
    this.number = number;
  }

  toString() {
    return `${this.quantity()} ${this.container()}`;
  }

  quantity() {
    return this.number.toString();
  }

  container() {
    if (this.number === 1) {
      return 'bottle';
    } else {
      return 'bottles';
    }
  }

  action() {
    return `Take ${this.pronoun()} down and pass it around`;
  }

  pronoun() {
    if (this.number === 1) {
      return 'it';
    } else {
      return 'one';
    }
  }

  successor() {
    return this.number - 1;
  }
}

class BottleNumber0 extends BottleNumber {
  quantity() {
    return 'no more';
  }
```

```
43 |
44 |   action() {
45 |     return 'Go to the store and buy some more';
46 |   }
47 |
48 |   successor() {
49 |     return 99;
50 |   }
51 |}
```

The `quantity`, `action` and `successor` methods are now divided between `BottleNumber` and `BottleNumber0`. This completes the creation of a bottle number specific to `0`.

The next task is to repeat this entire procedure for `1`. As before, the first step is to create an empty class:

Listing 6.23: Create the BottleNumber1 Class

```
1 |class BottleNumber1 extends BottleNumber {
2 |}
```

Next, choose one method that obsesses on 1 and copy it to the subclass. The `container` method is a reasonable place to start:

Listing 6.24: Duplicate the Container Method

```
1 |class BottleNumber {
2 |  // ...
3 |  container() {
4 |    if (this.number === 1) {
5 |      return 'bottle';
6 |    } else {
7 |      return 'bottles';
8 |    }
9 |  }
10 |  // ...
11 |}
12 |
13 |class BottleNumber1 extends BottleNumber {
14 |  container() {
15 |    if (this.number === 1) {
16 |      return 'bottle';
17 |    } else {
18 |      return 'bottles';
19 |    }
20 |  }
21 |}
```

Next, remove everything but the true branch logic from the subclass:

Listing 6.25: BottleNumber1 Returns Correct Result

```
1 |class BottleNumber1 extends BottleNumber {
2 |  container() {
3 |    return 'bottle';
4 |  }
5 |}
```

Now, remove everything but the false branch logic from the superclass:

Listing 6.26: BottleNumber Container Method Reduced to Default

```
class BottleNumber {
  // ...
  container() {
    return 'bottles';
  }
  // ...
}
```

Unfortunately, the tests are now failing with:

```
expect(received).toBe(expected) // Object.is equality

- Expected
+ Received

- 1 bottle of beer on the wall, 1 bottle of beer.
+ 1 bottles of beer on the wall, 1 bottles of beer.
  Take it down and pass it around, no more bottles of beer on the wall.
  ↵

  31 |       'Take it down and pass it around, ' +
  32 |       'no more bottles of beer on the wall.\n';
> 33 |     expect(new Bottles().verse(1)).toBe(expected);
     |                                    ^
  34 |   });
  35 |
  36 |   test('verse 0', () => {
```

The test for verse 1 failed because it got `1 bottles` but expected `1 bottle` (lines 6 and 7 above). This may be confusing because you know that `BottleNumber1` correctly implements `container` to return `bottle`. The problem, however, is not that `BottleNumber1` is wrong, but that the factory does not yet return it.

As currently written, the factory must be updated every time a new bottle number class gets created. The following example thus changes `bottleNumberFor` to return an instance of `BottleNumber1` when the value of `number` is `1`:

Listing 6.27: Factory Knows About BottleNumber1

```
class Bottles {
  // ...
  bottleNumberFor(number) {
    let bottleNumberClass;
    switch (number) {
      case 0:
        bottleNumberClass = BottleNumber0;
        break;
      case 1:
        bottleNumberClass = BottleNumber1;
        break;
      default:
        bottleNumberClass = BottleNumber;
        break;
    }

    return new bottleNumberClass(number);
  }
}
```

While adding the new class, the syntax was also changed from `if` to `case`, for reasons previously discussed in the Hewing to the Plan section of Chapter 2.

The conditional above may be giving you a sense of deja vu. It's reminiscent of, although not quite identical to, the `switch` statement from the original Shameless Green solution. Think about why this might be as you finish the current refactoring. The similarity will be explored at the end of this section.

Now that instances of `BottleNumber1` are being manufactured, the tests again pass, and you can move on to `pronoun`. Once `pronoun` is resolved, the final code is as follows:

Listing 6.28: BottleNumber Hierarchy

```
class Bottles {
  song() {
    return this.verses(99, 0);
  }

  verses(starting, ending) {
    return downTo(starting, ending)
      .map(i => this.verse(i))
      .join('\n');
  }

  verse(number) {
    const bottleNumber = this.bottleNumberFor(number);
    const nextBottleNumber =
      this.bottleNumberFor(bottleNumber.successor());

    return (
      capitalize(`${bottleNumber} of beer on the wall, `) +
      `${bottleNumber} of beer.\n` +
      `${bottleNumber.action()}, ` +
      `${nextBottleNumber} of beer on the wall.\n`
    );
  }

  bottleNumberFor(number) {
    let bottleNumberClass;
    switch (number) {
      case 0:
        bottleNumberClass = BottleNumber0;
        break;
      case 1:
        bottleNumberClass = BottleNumber1;
        break;
      default:
        bottleNumberClass = BottleNumber;
        break;
    }

    return new bottleNumberClass(number);
  }
}

class BottleNumber {
  constructor(number) {
    this.number = number;
  }

  toString() {
```

```
    return `${this.quantity()} ${this.container()}`;
  }

  quantity() {
    return this.number.toString();
  }

  container() {
    return 'bottles';
  }

  action() {
    return `Take ${this.pronoun()} down and pass it around`;
  }

  pronoun() {
    return 'one';
  }

  successor() {
    return this.number - 1;
  }
}

class BottleNumber0 extends BottleNumber {
  quantity() {
    return 'no more';
  }

  action() {
    return 'Go to the store and buy some more';
  }

  successor() {
    return 99;
  }
}

class BottleNumber1 extends BottleNumber {
  container() {
    return 'bottle';
  }

  pronoun() {
    return 'it';
  }
}
```

Take a minute to admire that code. While the whole is not perfect, the `BottleNumber` hierarchy displays a pleasing symmetry that was effortlessly attained by way of a simple recipe.

The code has undergone a number of transitions. Each refactoring followed a recipe, which led to a stable landing point, which in turn enabled the next refactoring. This most recent transition arguably achieves the greatest conceptual leap by way of the least complicated recipe. The ease with which it occurred is a tribute to the efficacy of earlier refactorings.

This completes the *Replace Conditional with Polymorphism* refactoring. If introducing polymorphism improved the code, this new version ought to tell an accurate and easily understood story about the domain. One way to evaluate the story is to revisit the domain questions asked in Chapter 1. The original questions were:

1. *How many verse variants are there?*
2. *Which verses are most alike? In what way?*
3. *Which verses are most different? In what way?*
4. *What is the rule to determine which verse should be sung next?*

If you examine the code in light of the above, you'll notice that the questions revolve around *verse* variation, while the current code is more concerned with *bottle number* variation. The story the code now tells is that all verses are alike in some abstract way, and that within verses, bottle numbers vary.

Updating the questions to reflect this more nuanced understanding, they become:

1. *How many bottle number variants are there?*
   Three.
2. *Which bottle numbers are most alike? In what way?*
   Bottle numbers 2-99 are most alike.
3. *Which bottle numbers are most different? In what way?*
   Bottle numbers 0 and 1 are different from each other, and from all the others. Bottle number 0 overrides three methods, and so is slightly more different from the others than is bottle number 1.
4. *What is the rule to determine which bottle number comes next?*
   The next bottle number is the successor of the current one. This concept is clearly visible in this code. However, one would expect a `successor` to be the same type as the thing it succeeds, but here that's not the case. The `successor` of a bottle number is, disappointingly, a primitive. This seems wrong, and should be addressed.

6.4. Transitioning Between Types

The code now consists of a pleasing set of small objects with clear-cut responsibilities. However, there's one persistent problem that can no longer be ignored: the `successor` methods violate the generalized Liskov Substitution Principle. They make a promise that they fail to keep.

You have every right to expect the successor of a bottle number to *act* like a bottle number, but these successors disappoint. The `successor` methods return a result so unexpected that it's perilously close to being an outright lie. Instead of bottle numbers, they return primitive numbers, which you are then forced to convert into bottle numbers yourself.

Liskov violations are insidious, and over time cause increasing harm. As your application evolves, `successor` might get sent from many places. Each place will have to know that `successor` returns a number, and must also know how to convert that number back into a bottle number. This interconnected web of duplicated knowledge binds every sender of `successor` to its current implementation, which inflicts dependencies that make code resistant to change.

If `successor` obeyed Liskov, you could substitute the hypothetical code on line 7 below for the code on lines 5-6:

Listing 6.29: Coding by Wishful Thinking

```
1 class Bottles {
2   // ...
3   verse(number) {
4     const bottleNumber = this.bottleNumberFor(number);
5     const nextBottleNumber =
6       this.bottleNumberFor(bottleNumber.successor());
7     // const nextBottleNumber = bottleNumber.successor();
8     // ...
9   }
10  // ...
11 }
```

On line 7 above, the `successor` method returns a bottle number. This implementation avoids the Liskov violation, reduces the number of dependencies, and simplifies the code.

The Liskov violation on line 6 has existed for several refactorings, but has been ignored in favor of curing other code smells. It's instructive to recall how it originated before resolving the problem.

In Chapter 4, when the `successor` method was first created in `Bottles`, there was no violation. The method was extracted from the `verse` template using the *Flocking Rules*, and within that original template, the successor was indeed a number. In that case, it was both reasonable and correct for `successor` to return that number.

In Chapter 5, the `successor` method was identified as one that obsessed on the `number` argument, and so was migrated to `BottleNumber` during the *Extract Class* refactoring. It was at this point that the Liskov violation appeared. The root of the problem is that a new type (`BottleNumber`) was introduced, but its `successor` method continued to return the old type (primitive numbers).

The Liskov violation was troubling enough in the final code example of Chapter 5, which contained but one implementor and one sender of `successor`. Unfortunately, the refactorings in this chapter have exacerbated the problem. There are now two implementors of `successor`, one sender, and a new factory that's in charge of bottle number construction. Deferring the Liskov violation made it worse, and paradoxically, supplied a more useful example to learn to solve.

The current predicament stands in for the real-world problem of needing to change the type returned by a polymorphic method that has many implementors and many senders. Such a real-life difficulty could well contain so many parts that they couldn't all be fixed at once. The following technique can be used to solve type change problems of any size. It does this by making small, reliable, *independent* changes over time, chipping away until eventually the entire issue is resolved.

Here's a summary of the code related to `successor`:

Listing 6.30: All About Successor

```
1  class Bottles {
2    // ...
3    verse(number) {
4      const bottleNumber = this.bottleNumberFor(number);
5      const nextBottleNumber =
6        this.bottleNumberFor(bottleNumber.successor());
7      // const nextBottleNumber = bottleNumber.successor();
8
9      return (
10       capitalize(`${bottleNumber} of beer on the wall, `) +
11       `${bottleNumber} of beer.\n` +
12       `${bottleNumber.action()}, ` +
13       `${nextBottleNumber} of beer on the wall.\n`
14     );
15   }
16
17   bottleNumberFor(number) {
18     let bottleNumberClass;
19     switch (number) {
20       case 0:
21         bottleNumberClass = BottleNumber0;
22         break;
23       case 1:
24         bottleNumberClass = BottleNumber1;
25         break;
26       default:
27         bottleNumberClass = BottleNumber;
28         break;
29     }
30
31     return new bottleNumberClass(number);
32   }
33 }
34
35 class BottleNumber {
36   // ...
37   successor() {
38     return this.number - 1;
39   }
40 }
41
42 class BottleNumber0 extends BottleNumber {
43   // ...
44   successor() {
45     return 99;
46   }
47 }
```

On line 6 above, the `verse` method knows that `successor` returns a number, but wishes that it returned a bottle number as illustrated on line 7. The two `successor` methods (lines 37 and 44) ought to return bottle numbers, but to do so they must invoke the factory, and the factory is not easily within their reach. And sadly, the aforementioned difficulties are compounded by your ongoing determination to resolve problems via a series of one-line changes.

Alterations are needed in several places. Ultimately:

1. The factory should be located such that it is reachable by the `successor` methods,
2. the `successor` methods should invoke the factory, and
3. the `verse` method should expect `successor` to return a bottle number.

That's a fairly small list, but even so, it's challenging to accomplish this transition via a series of one-line changes that don't break the tests. For problems of this size, you *might* be successful at changing everything at once, but real life typically involves larger problems that require many more changes and present a much greater challenge. The following step-wise strategy is useful because it works for problems of any size. While it may be overkill on small ones, it is deeply comforting on big ones.

In that spirit, continue on with the code. Step 1 is to put the factory within `successor`'s reach. There are a number of options, but if you want to make the smallest possible change, the best choice is to make the factory a static method on an existing class. The most reasonable choice among existing classes is `BottleNumber`.

The following example copies the factory into a static method on `BottleNumber`:

Listing 6.31: BottleNumber Class Contains Factory

```
class BottleNumber {
  static for(number) {
    let bottleNumberClass;
    switch (number) {
      case 0:
        bottleNumberClass = BottleNumber0;
        break;
      case 1:
        bottleNumberClass = BottleNumber1;
        break;
      default:
        bottleNumberClass = BottleNumber;
        break;
    }

    return new bottleNumberClass(number);
  }
  // ...
}
```

As you can see, here on `BottleNumber` the method name has been reduced to simply `for`. The internal implementation hasn't changed, just the name.

It made perfect sense to name the original method `bottleNumberFor`. That name has two parts: a type (`bottleNumber`), and a generic request (`for`). Within `Bottles`, both pieces of information were relevant and arguably necessary. Now that the method is moving to `BottleNumber`, there are two good reasons to simplify its name.

First, changing the name avoids the "echo chamber" effect. `BottleNumber.bottleNumberFor` is both redundant and overly specific. It suffers from the same ailment as the beer method in Chapter 1. This name is tightly coupled to the current context, and tight coupling makes code resistant to change. For example, if you someday decide to rename the `BottleNumber` class, you'll have to change this method name too, or forever be misled.

Second, and more abstractly, `for` supports polymorphism. To illustrate how, consider `toString`. As you've seen, `Object` implements a default `toString`, and many classes supply their own

more specific implementation. Because all are named `toString`, you can confidently send this generic request to any receiver.

Imagine the consequences of including the receiver's type in the message name, as in `numberToString`, `arrayToString`, and, inevitably, `stringToString`. Adding type information defeats polymorphism, and forces you to check the type of the receiver before sending this message. This vastly complicates code for no good reason.

Just like `toString`, `for` is a generic request that works fine as the name of *any* factory. When factory-ish objects polymorphically implement `for`, you can send this message without regard for the receiver's type. Polymorphism preserves the option of constructing applications where the factories *themselves* are substitutable.

Now that the `for` factory method exists, you can alter `verse` to invoke it, as shown here:

Listing 6.32: Verse Method Uses Factory

```
1 class Bottles {
2   // ...
3   verse(number) {
4     const bottleNumber = BottleNumber.for(number);
5     const nextBottleNumber =
6       BottleNumber.for(bottleNumber.successor());
7     // ...
8   }
9   // ...
10 }
```

Lines 4 and 6 above directly invoke the factory in `BottleNumber`. This makes `bottleNumberFor` obsolete. That method can now be deleted.

The factory is now easily reachable by the two `successor` methods, so you've finished step 1.

Step 2 requires that you change the two `successor` methods to invoke the factory, but unfortunately, changing either one without simultaneously making all remaining changes will cause the tests to fail. Indeed, at this point, *every* outstanding change breaks the tests.

For example, in step 3 you'll want to change the `verse` method to read:

```
const nextBottleNumber = bottleNumber.successor();
```

instead of:

```
const nextBottleNumber =
  BottleNumber.for(bottleNumber.successor());
```

The above, however, relies upon having completed step 2, which changes the `successor` methods to return bottle numbers. You can't skip forward and do step 3 before step 2.

Returning to step 2, the `successor` methods can now invoke the factory. However, changing one (but not the other) to do so via:

```
return BottleNumber.for(99);
```

or:

```
return BottleNumber.for(number - 1);
```

also breaks the tests.

The root of the problem is that the `verse` method expects `successor` to return something that will work in the factory, and the factory, in turn, expects to receive a number. If you change one of the `successor` methods to return a bottle number, then `verse` will blithely pass that bottle number right into the bottle number factory, which breaks the tests.

The trick to moving forward using one-line changes is to temporarily alter the factory to tolerate *both* kinds of input. During the transitional period where one `successor` method returns a bottle number and the other returns a primitive, the factory will have to handle both types. This requires doing something that is anathema to your object-oriented soul: you must change the factory to check the type of its input argument.

Here's the code:

Listing 6.33: Return Argument If Already a Bottle Number

```
1 class BottleNumber {
2   static for(number) {
3     if (number instanceof BottleNumber) {
4       return number;
5     }
6
7     let bottleNumberClass;
8     switch (number) {
9       case 0:
10        bottleNumberClass = BottleNumber0;
11        break;
12      case 1:
13        bottleNumberClass = BottleNumber1;
14        break;
15      default:
16        bottleNumberClass = BottleNumber;
17        break;
18    }
19
20    return new bottleNumberClass(number);
21  }
22  // ...
23 }
```

The guard clause on lines 3-5 above bounces the input argument right back out *if it is already a bottle number*. This line is needed only while the refactoring is in progress. Once all `successor` methods return a bottle number, and all callers of `successor` expect to get a bottle number back, lines 3-5 can be deleted.

If this code seems confusing, it's because of the power of names. For example, what if the argument on line 2 above had been named `numberOrBottleNumber` instead of `number`? In that case lines 3-5 would read:

```
if (numberOrBottleNumber instanceof BottleNumber) {
  return numberOrBottleNumber;
}
```

This line tells a better story. Despite that, there's no point in actually changing the code to read like this. The guard clause is a temporary convenience that allows the factory to accept two different types of argument during a refactoring that changes from one to the other. At the beginning and end of this refactoring, the argument on line 2 is always a number. There's no point in changing `number` to `numberOrBottleNumber` only to then change it right back. Imagining this alternate name is enough to help you understand what's happening.

Now that the factory handles both input types you can continue with step 2 by altering the `successor` methods to return a bottle number. Here's the change in `BottleNumber0`:

Listing 6.34: BottleNumber0 Successor Returns a Bottle Number

```
1 class BottleNumber0 extends BottleNumber {
2   // ...
3   successor() {
4     return BottleNumber.for(99);
5   }
6 }
```

At this point, the tests pass even though one `successor` returns a number and the other returns a bottle number.

Having succeeded with `BottleNumber0`, you can proceed to `BottleNumber`, like so:

Listing 6.35: BottleNumber Successor Returns a Bottle Number

```
1 class BottleNumber {
2   // ...
3   successor() {
4     return BottleNumber.for(this.number - 1);
5   }
6 }
```

And voila, all implementors of `successor` have been updated, and you've accomplished step 2.

Step 3 requires changing the `verse` method to expect `successor` to return a bottle number. It should now be possible to do just that. The following code gingerly tests this theory by uncommenting the wishful code on line 7:

Listing 6.36: Trying Out the Wishful Code

```
1  class Bottles {
2    // ...
3    verse(number) {
4      const bottleNumber = BottleNumber.for(number);
5      let nextBottleNumber =
6        BottleNumber.for(bottleNumber.successor());
7      nextBottleNumber = bottleNumber.successor();
8
9      return (
10       capitalize(`${bottleNumber} of beer on the wall, `) +
11       `${bottleNumber} of beer.\n` +
12       `${bottleNumber.action()}, ` +
```

```
13 |       `${nextBottleNumber} of beer on the wall.\n`
14 |     );
15 |   }
16 |}
```

Lines 5-6 above are the original code, which you hope to delete. Notice that the `const` on line 5 has been temporarily changed to a `let` so the wish on line 7 can reassign the variable. They set `nextBottleNumber` the old way. Line 7 is the new code, which you'd like to keep. It overwrites `nextBottleNumber` with the result of the current bottle number's successor.

Making the transition from old code to new code by running old *and* new side-by-side is useful in situations where you're not 100% certain you got it right. If something blows up, it can ease debugging to have both variants under your eye.

In this case, the tests continue to pass, so you can confidently delete lines 5-6 above, and switch back to using `const`. This leaves the following code:

Listing 6.37: Trusting Successor

```
1 |class Bottles {
2 |  // ...
3 |  verse(number) {
4 |    const bottleNumber = BottleNumber.for(number);
5 |    const nextBottleNumber = bottleNumber.successor();
6 |
7 |    return (
8 |      capitalize(`${bottleNumber} of beer on the wall, `) +
9 |      `${bottleNumber} of beer.\n` +
10|      `${bottleNumber.action()}, ` +
11|      `${nextBottleNumber} of beer on the wall.\n`
12|    );
13|  }
14|}
```

This is definitely an improvement. However, notice that the temporary variable `nextBottleNumber` declared on line 5 is used only in one place, on line 11. This presents a further opportunity for simplification. Temporary variables that are used just once can be removed with the *Inline Temp* refactoring, which results in the following code:

Listing 6.38: Simplified Verse Method

```
1 |class Bottles {
2 |  // ...
3 |  verse(number) {
4 |    const bottleNumber = BottleNumber.for(number);
5 |
6 |    return (
7 |      capitalize(`${bottleNumber} of beer on the wall, `) +
8 |      `${bottleNumber} of beer.\n` +
9 |      `${bottleNumber.action()}, ` +
10|      `${bottleNumber.successor()} of beer on the wall.\n`
11|    );
12|  }
13|}
```

At this point, all `successor` methods return a bottle number, and all senders of `successor` expect to receive a bottle number. The only remaining issue is that the factory still contains the

guard clause:

Listing 6.39: Factory With Obsolete Guard Clause

```
class BottleNumber {
  static for(number) {
    if (number instanceof BottleNumber) {
      return number;
    }

    let bottleNumberClass;
    switch (number) {
      case 0:
        bottleNumberClass = BottleNumber0;
        break;
      case 1:
        bottleNumberClass = BottleNumber1;
        break;
      default:
        bottleNumberClass = BottleNumber;
        break;
    }

    return new bottleNumberClass(number);
  }
  // ...
}
```

That guard clause is now obsolete, and can be deleted.

This completes the correction of the Liskov violation in `successor`. Here's a full listing of the code:

Listing 6.40: Complete Listing

```
class Bottles {
  song() {
    return this.verses(99, 0);
  }

  verses(starting, ending) {
    return downTo(starting, ending)
      .map(i => this.verse(i))
      .join('\n');
  }

  verse(number) {
    const bottleNumber = BottleNumber.for(number);

    return (
      capitalize(`${bottleNumber} of beer on the wall, `) +
      `${bottleNumber} of beer.\n` +
      `${bottleNumber.action()}, ` +
      `${bottleNumber.successor()} of beer on the wall.\n`
    );
  }
}

class BottleNumber {
  static for(number) {
    let bottleNumberClass;
    switch (number) {
```

```
      case 0:
        bottleNumberClass = BottleNumber0;
        break;
      case 1:
        bottleNumberClass = BottleNumber1;
        break;
      default:
        bottleNumberClass = BottleNumber;
        break;
    }

    return new bottleNumberClass(number);
  }

  constructor(number) {
    this.number = number;
  }

  toString() {
    return `${this.quantity()} ${this.container()}`;
  }

  quantity() {
    return this.number.toString();
  }

  container() {
    return 'bottles';
  }

  action() {
    return `Take ${this.pronoun()} down and pass it around`;
  }

  pronoun() {
    return 'one';
  }

  successor() {
    return BottleNumber.for(this.number - 1);
  }
}

class BottleNumber0 extends BottleNumber {
  quantity() {
    return 'no more';
  }

  action() {
    return 'Go to the store and buy some more';
  }

  successor() {
    return BottleNumber.for(99);
  }
}

class BottleNumber1 extends BottleNumber {
  container() {
    return 'bottle';
  }

  pronoun() {
    return 'it';
```

```
92 |   }
93 |}
```

Correcting the Liskov violation is important because object-oriented programming, especially in dynamically-typed languages like JavaScript, relies on *explicit* trust in the *implicit* contracts between objects. These implicit contracts consist of expectations about the messages to which other objects respond, and presumptions about the results those messages return. Trustworthy objects are a joy to work with because they always behave as you expect.

Untrustworthy objects, however, are a different kettle of fish.[16] Objects that sometimes fail to respond to a message you plan to send, or occasionally return something you don't expect, force you into a paranoid programming style. These untrustworthy objects require senders of messages to know too much.

When your application has code that needs knowledge of the internals of other objects in order to correctly interact with them (as did `successor` above), changes to those other objects might break your code. If you have to check the type of an object in order to know what message to send, you're forced into a conditional that lists every concrete class with which you're willing to collaborate. Doing this dooms you to changing the conditional when you add a new class. Checking to see if a object responds to a message rather than checking that object's type may reduce the size of this conditional, but it doesn't ameliorate the problem.

All of the above are symptoms of an inability to trust other objects, and failures of trustworthiness are, at least by the current generous interpretation of the principle, Liskov violations. Objects made promises that they did not keep. In every case, the underlying cause is an insufficient use of polymorphism.

Having successfully fixed the problem with `successor`, it's time to return to the main issue at hand.

6.5. Making the Easy Change

The previous horizontal refactoring is complete, and it is again time to ask if the code is open to the six-pack requirement. And finally, gloriously (and only if you're willing to disregard the factory for a moment) the answer is yes. Your discipline and hard work are about to pay off.

You can now meet the six-pack requirement by *adding* a new class that stands in for bottle number 6. This new class will report its quantity as "1" and its container, "six-pack."

The factory is *not* open, and so for now must be updated to return an instance of `BottleNumber6` when the value of `number` is `6`. The next chapter will explore the costs and benefits of making the factory open for extension.

You have been refactoring for many chapters using passing tests, or green, as the wall at your back. Now that the current arrangement of code is open to the six-pack requirement, it's finally time to switch from refactoring mode back into TDD mode.

At long last, it's time to write a failing test.

The six-pack requirement changes verses 6 and 7. The simplest way to generate a test failure is to alter the `song` test to change the expected text for those verses. Here's that updated test:

Listing 6.41: Test

```
describe('Bottles', () => {
  // ...
  test('the whole song', () => {
    const expected =
`99 bottles of beer on the wall, 99 bottles of beer.
Take one down and pass it around, 98 bottles of beer on the wall.

    // ...

7 bottles of beer on the wall, 7 bottles of beer.
Take one down and pass it around, 1 six-pack of beer on the wall.

1 six-pack of beer on the wall, 1 six-pack of beer.
Take one down and pass it around, 5 bottles of beer on the wall.

    // ...

No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99 bottles of beer on the wall.
`;
    expect(new Bottles().song()).toBe(expected);
  });
});
```

Lines 11 and 13 above now assert that verses 6 and 7 read "1 six-pack" instead of "6 bottles." Running that updated test results in this error:

```
7 bottles of beer on the wall, 7 bottles of beer.
-Take one down and pass it around, 1 six-pack of beer on the wall.
+Take one down and pass it around, 6 bottles of beer on the wall.

-1 six-pack of beer on the wall, 1 six-pack of beer.
+6 bottles of beer on the wall, 6 bottles of beer.
Take one down and pass it around, 5 bottles of beer on the wall.
```

There are two problems apparent in the error message. First, the output says "6" where it should say "1." This is the `quantity` concept. Second, the output says "bottles" instead of "six-pack." This is `container`.

The `BottleNumber` inheritance hierarchy provides exemplary guidance for solving these problems. Following the pattern of `BottleNumber0` and `BottleNumber1`, first create a new `BottleNumber6` class as a subclass of `BottleNumber`:

Listing 6.42: New BottleNumber6 Class

```
class BottleNumber6 extends BottleNumber {
}
```

Next, implement one of the necessary methods. For example, you could implement `container` as follows:

Listing 6.43: BottleNumber6 Knows Container

```
class BottleNumber6 extends BottleNumber {
  container() {
    return 'six-pack';
  }
}
```

Having made the change above, you could reasonably expect the error message to change, but alas, it does not:

```
7 bottles of beer on the wall, 7 bottles of beer.
-Take one down and pass it around, 1 six-pack of beer on the wall.
+Take one down and pass it around, 6 bottles of beer on the wall.

-1 six-pack of beer on the wall, 1 six-pack of beer.
+6 bottles of beer on the wall, 6 bottles of beer.
Take one down and pass it around, 5 bottles of beer on the wall.
```

Above, the sixth bottle still reports "bottles" as its `container`. The error message hasn't changed. This would happen if the new `BottleNumber6` weren't being used, and that's exactly the case. Because the factory isn't yet open, creating the new class isn't enough—you must also update the factory.

Adding `BottleNumber6` to the factory results in the following code:

Listing 6.44: BottleNumber6 Added to Factory

```
class BottleNumber {
  static for(number) {
    let bottleNumberClass;
    switch (number) {
      case 0:
        bottleNumberClass = BottleNumber0;
        break;
      case 1:
        bottleNumberClass = BottleNumber1;
        break;
      case 6:
        bottleNumberClass = BottleNumber6;
        break;
      default:
        bottleNumberClass = BottleNumber;
        break;
    }

    return new bottleNumberClass(number);
  }
  // ...
}
```

Once you update the factory, running the tests produces the expected error:

```
7 bottles of beer on the wall, 7 bottles of beer.
-Take one down and pass it around, 1 six-pack of beer on the wall.
+Take one down and pass it around, 6 six-pack of beer on the wall.

-1 six-pack of beer on the wall, 1 six-pack of beer.
+6 six-pack of beer on the wall, 6 six-pack of beer.
Take one down and pass it around, 5 bottles of beer on the wall.
```

As shown above, the container for six bottles is now "six-pack."

The sixth bottle's quantity is still incorrect. This is easily cured by implementing the `quantity` method as so:

Listing 6.45: Final BottleNumber6

```
class BottleNumber6 extends BottleNumber {
  quantity() {
    return '1';
  }

  container() {
    return 'six-pack';
  }
}
```

Having implemented `quantity` and `container` in `BottleNumber6`, the tests now pass.

Congratulations, you've met the six-pack requirement!

You have been refactoring under green for many chapters, and now, suddenly, almost abruptly, the outstanding requirement can be met by two one-line methods in a class that has nine total lines of code. It took several refactorings to make the code open, but once so, the six-pack requirement was extraordinarily easy to fulfill.

Kent Beck describes this entire process beautifully, and sympathetically, when he says:

> *make the change easy (warning: this may be hard), then make the easy change*
>
> — *Kent Beck*
> *via Twitter*

Most of this book has been concerned with making the change easy. That hard work paid off here, where you made the easy change.

6.6. Defending the Domain

One final thought about `BottleNumber6` before moving on: it may have occurred to you to meet the six-pack requirement by simply overriding `toString` within `BottleNumber6`. For example, instead of implementing `quantity` and `container`, you could do the following:

Listing 6.46: BottleNumber6 Knows Neither Quantity nor Container

```
class BottleNumber6 extends BottleNumber {
  toString() {
    return '1 six-pack';
  }
}
```

The above code certainly passes the six-pack tests. This solution might seem attractive because it's shorter than the previous one, and so may feel more efficient. However, less code doesn't always mean better code.

Consider the meaning of `toString` versus that of `quantity` and `container`. The latter two methods reflect fundamental concepts in this domain. These concepts exist regardless of the way your application uses bottle numbers.

Extracting `BottleNumber` from `Bottles` decouples the idea of bottle number-ness from the "99 Bottles of Beer" song. Bottle numbers are now independent objects, and ought to be freely useable in contexts *other than those from which they were extracted*. If sufficiently dissociated from the song, these bottle number classes could be used in, for example, a new inventory system. It makes perfect sense to "Go to the store and buy some more" because a refrigerator reports that it contains 0 bottles of beer.

Omitting `quantity` and `container` in favor of jamming "1 six-pack" directly into `toString` corrupts `BottleNumber6` with knowledge of the inner workings of the `Bottles verse` template. The `toString` solution works only because `BottleNumber6` *knows* that `verse` implicitly sends `toString`. This expectation couples `BottleNumber6` to the context in which it was discovered, and this coupling interferes with your ability to reuse the bottle number classes when new contexts appear.

Solving the proximate problem by implementing a unique `toString` passes today's tests but misleads future programmers. If you were to override `toString`, your code would tell this story:

1. `BottleNumber6`'s rule for deriving its string representation differs from that of other bottle numbers, and
2. `BottleNumber6` has the same `quantity` and `container` as its superclass.

These claims are false, and they transfer costs from the present to the future.

Clever shortcuts are a false economy. Invest in code that tells the truth. Just write it down.

6.7. Summary

The purpose of this chapter was to produce a code arrangement that was open to the six-pack requirement. Not only did it succeed in fulfilling that requirement, but along the way it also resolved a number of other issues.

This chapter explored the *Data Clump* code smell. It replaced a *Switch Statement* with a set of polymorphic objects, which it created using a factory. It corrected the Liskov violation in `successor`, and used that problem as a jumping-off point for a more general lesson about how to change the return types of polymorphic methods.

The `BottleNumber for` factory was straightforward and most certainly did the job. While simple factories like this work great in many situations, they're not best for every case. There's a whole world of different styles of factories waiting to be explored. Therefore, on to Chapter 7.

7. Manufacturing Intelligence

In Chapter 6, on the way to achieving blissfully open code, you created a set of classes whose instances polymorphically play the role of bottle number. Each class in the `BottleNumber` hierarchy contains a simple set of code that represents the concrete implementation of a single bottle number variant. Gratifyingly, those classes contain no conditionals.

And yet, the need for conditional logic did not disappear. Some code, somewhere, has to know how to select the right bottle number class for any situation. This selection happens in the `BottleNumber.for` *factory*. In this chapter, that simple factory launches a greater exploration of factories in general.

7.1. Contrasting the Concrete Factory with Shameless Green

The *Replace Conditional with Polymorphism* refactoring in the prior chapter resulted in a small bottle number hierarchy. That hierarchy was open for extension, which made it possible to add six-pack behavior by simply adding a new `BottleNumber6` class.

Unfortunately, there was still one bit of code that had to be changed before everything would work. The `BottleNumber.for` factory contained a hard-coded conditional to pick the correct class, and this conditional had to be updated to include the new class name.

Here's a reminder of the resulting factory:

Listing 7.1: Bottle Number Factory

```
class BottleNumber {
  static for(number) {
    let bottleNumberClass;
    switch (number) {
      case 0:
        bottleNumberClass = BottleNumber0;
        break;
      case 1:
        bottleNumberClass = BottleNumber1;
        break;
      case 6:
        bottleNumberClass = BottleNumber6;
        break;
      default:
        bottleNumberClass = BottleNumber;
        break;
    }

    return new bottleNumberClass(number);
  }
  // ...
}
```

Pause for a minute to reflect upon the current code. The `for` method above contains a simple `switch` statement that chooses a class. This conditional may remind you of one contained in the original Shameless Green implementation, repeated below:

Listing 7.2: Shameless Green Conditional

```
verse(number) {
  switch (number) {
    case 0:
      return (
        'No more bottles of beer on the wall, ' +
        'no more bottles of beer.\n' +
        'Go to the store and buy some more, ' +
        '99 bottles of beer on the wall.\n'
      );
    case 1:
      return (
        '1 bottle of beer on the wall, ' +
        '1 bottle of beer.\n' +
        'Take it down and pass it around, ' +
        'no more bottles of beer on the wall.\n'
      );
    case 2:
      return (
        '2 bottles of beer on the wall, ' +
        '2 bottles of beer.\n' +
        'Take one down and pass it around, ' +
        '1 bottle of beer on the wall.\n'
      );
    default:
      return (
        `${number} bottles of beer on the wall, ` +
        `${number} bottles of beer.\n` +
        'Take one down and pass it around, ' +
        `${number-1} bottles of beer on the wall.\n`
      );
  }
}
```

As you know, the current factory handles six-packs while the original Shameless Green did not. To make this comparison more meaningful, undo the last change you made to the factory, i.e. remove the `6` branch. This reverts the factory to:

Listing 7.3: Factory Without Case 6

```
class BottleNumber {
  static for(number) {
    let bottleNumberClass;
    switch (number) {
      case 0:
        bottleNumberClass = BottleNumber0;
        break;
      case 1:
        bottleNumberClass = BottleNumber1;
        break;
      default:
        bottleNumberClass = BottleNumber;
        break;
    }

    return new bottleNumberClass(number);
  }
  // ...
}
```

Study the previous two listings. These conditionals are the only ones that exist in their respective examples. Both Shameless Green and the current code correctly generate the complete lyrics to the original song. Given that these different conditionals produce the same variability, can you explain why one contains four branches, but the other only three?

To answer that question, consider the case that is missing. Shameless Green has a special case for `2`, but the factory does not. Recall that the conditional in Shameless Green produces *verses*, but the one in the factory produces *bottle numbers*. Verse 2 is indeed special, but bottle number 2 is not. Thus, Shameless Green needs a special case for verse 2 solely because verse 2 contains bottle number 1. This explains the missing branch.

The difference in the number of branches, although highly visible, is merely an artifact of the specific domain of the "99 Bottles of Beer" song. Having explained that difference away, another yet remains, and this one has significantly more meaning.

The Shameless Green `verse` method contains a conditional that:

1. understands why you might switch (`case 0`, for example), and
2. knows the behavior needed for this case (`"No more bottles . . ."`).

The factory `for` method:

1. is similar in that it also understands the reasons for switching (again, `case 0`, etc) but
2. differs in that it knows the *name of the class that supplies the behavior* for the case.

Factories don't know what to do: instead, they know how to *choose* who does. They consolidate the choosing and separate the chosen. Shameless Green was a procedure because it combined these two things; the current code is object-oriented because it breaks them apart.

Having explored the difference between a conditional that supplies behavior and a conditional that selects an object, it's time to take a deeper look at factories.

7.2. Fathoming Factories

Object-oriented applications rely on polymorphism. Polymorphism results in multiple classes that play a common role. The power of polymorphism is that these role-playing objects are interchangeable from *the message sender's point of view*.

Message senders can confidently collaborate with polymorphic objects in faith that each honestly plays the common role even though they represent different variants. The message-sending object thinks of its collaborator as a player of a role rather than a kind of a type.

All players of a role share a common API. This API exposes a set of intentions that are public. Internally, each role-player also contains methods that implement these intentions in differing ways. The details of these alternate implementations are invisible to the outside world.

From the message sender's point of view, all players of a role are exactly the same. Message senders depend on the role player's exposed intentions while remaining studiously ignorant of their detailed internal implementations. They know what their collaborators do, but refuse to be aware of how they do it.

A system comprised of message senders who collaborate with role-playing objects can be extremely tolerant of unexpected change. For example, imagine that you create a new class that represents a new variant of bottle number. This new bottle number is interchangeable with every other one, so any class that already collaborates with an existing bottle number will be able to seamlessly collaborate with this new one. The message sender doesn't have to change; from its point of view this new bottle number is the same as all the others.

Clearly, this system works only if message senders really do treat collaborators as if they're interchangeable. If message senders are to be immune from side-effects when adding or removing role-players, these senders can't know things that are unique to specific variants. Message senders aren't allowed to know the names of the concrete variant classes, nor may they know the logic needed to choose between them.

Message senders can't know these things, but of course someone must. Knowledge of the class names of the variants, and of the logic necessary to choose the correct one, can be hidden in, you guessed it—factories.

A factory's responsibility is to manufacture the right object for a given role. Factories oughtn't know what the variants do, they merely know how to choose the right variant for any situation. This *choosing* usually involves a conditional, and putting this conditional in a factory allows you to isolate it to a single place in your code.

Thus, factories are where conditionals go to die. Isolating conditionals in factories loosens the coupling between collaborating objects, which lowers the cost of change.

Factories can be implemented in many different ways, but they tend to vary along a few interesting dimensions, each of which involves its own set of trade-offs. No one style of factory is best for every case; the right solution depends entirely on the problem at hand. Understanding these dimensions, and the tradeoffs between them, allows you to make good decisions when faced with competing requirements.

Factories vary along these dimensions:

1. The factory can be open to new variants or closed.
2. The logic that chooses a variant can be owned by the factory or by the variant.
3. The factory can be responsible for knowing/figuring out which classes are eligible to be manufactured or the variants can volunteer themselves.

The following sections explore a series of escalating factory solutions, paying particular attention to trade-offs along these dimensions.

7.3. Opening the Factory

Back to the matter at hand, the current factory is not open for extension because it contains a hard-coded conditional.

You have likely noticed that the bottle number classes follow a naming convention. The default is `BottleNumber`, and the specializations suffix that name with their own specific value of `number`, e.g. `BottleNumber0`, `BottleNumber1`, and `BottleNumber6`.

This predictable pattern makes it possible for you to dynamically derive the correct bottle number class. You can create strings that match the class *names*, and then use a tiny bit of meta-programming to turn these class name strings into actual classes.

Here's a simple way to accomplish this:

Listing 7.4: Meta Programmed Class Lookup Factory

```
1 class BottleNumber {
2   static for(number) {
3     let bottleNumberClass;
4
5     try {
6       bottleNumberClass = eval(`BottleNumber${number}`);
7     } catch (e) {
8       bottleNumberClass = BottleNumber;
9     }
10
11    return new bottleNumberClass(number);
12  }
13  // ...
14 }
```

The code above produces that same output as the original factory. Having examined it, you may find yourself afflicted with objections. If so, you are not alone—many folks find this example downright alarming. While fully acknowledging that it contains plenty of things not to like, this code also confers clear benefits. Therefore, please put any objections aside for a moment and read the following explanation of the syntax. The pros and cons of this approach will be examined afterwards.

First, syntax. Notice that the `for` method now contains a `try { .. } catch { .. }` block. Within the `try` block on line 5, the string `"BottleNumber"` gets concatenated with the `number` argument. This *might* result in a string that matches a class name. Class names are variables in JavaScript, so the `eval` method can take this string as an argument, and attempt to look up the corresponding class.

If the class exists (as `BottleNumber0`, for example, does), the `try` block returns it. If the class does not exist (`BottleNumber37`, et al.), `eval` throws a `ReferenceError`. This causes the `catch` block to execute, which returns the `BottleNumber` class.

In its favor, this factory is open to extension. As long as you honor the naming convention, the factory will cheerfully accommodate newly-added bottle number classes without having to change.

Even so, there are many things to dislike about this code. Here are a few common and thoroughly reasonable objections:

1. This version is harder to understand than the original.

   Everyone understands how the original `switch` statement works, but many folks have no idea that it's possible to locate a class using the string value of its name. Some programmers find this code unexpected and confusing.

2. `BottleNumber0`, etc are no longer explicitly referenced in the source code.

   Good luck finding references to the classes whose names are dynamically constructed. Although the factory is perfectly capable of creating new instances of `BottleNumber0`, it doesn't explicitly reference this concrete class name. Attempts to find where instances of this class are created by searching the source code for `BottleNumber0` are fruitless and therefore deeply frustrating.

   It's also conceivable that, in an excess of cleanup zeal, someone might *delete* the apparently unreferenced class. If this happens without being caught by the tests, the application will break at a far distant and maximally inconvenient time.

3. The code uses an exception for flow control.

   Controlling the flow of a program with exceptions is roundly condemned on the Internet[17] and so must be an evil to be avoided at all costs.

4. The factory ignores bottle number classes whose names do not follow the convention.

   If an unsuspecting programmer innocently creates the new class `BottleNumberSix`, the factory won't know about it. Attempts to use this new class will fail silently with nary a hint at the underlying problem. This can lead to exasperating debugging sessions.

Given the list of objections, it's logical to wonder if opening this factory could ever be worthwhile. Do the benefits of openness justify the cost of this additional complexity?

The answer, as is true for most questions about object-oriented design, is that it depends. If you frequently create new bottle number classes, the cost of repeatedly changing the factory might very well exceed that of making it open. Conversely, if you never add new bottle number classes, the factory won't ever change, so there's no justification for complicating the code.

Your goal is to minimize costs, and costs are determined by the situation at hand. There's no hard and fast rule about what's best. It just depends.

A factory's fundamental job is to manufacture the correct player of a role. Relative to this responsibility, its openness is a trivial concern that can be tweaked over time.

7.4. Supporting Arbitrary Class Names

The previous example used a simple bit of meta-programming to generate the right class name based on a convention. This, obviously, requires that all classes be named following that convention.

What if you can't enforce a convention and must manufacture instances of classes that have arbitrary names? In this situation, you could always return to the `switch` statement. That form of factory will work, but unfortunately it's not open to extension and must be updated every time someone adds a new class. If new classes get added regularly, this is annoying and expensive.

The `switch` statement factory is easy to understand because it centralizes the necessary knowledge: it knows both the names of the candidate classes and also the reason any class might be chosen. Having the condition (`number === 0`) and the name of the class (`BottleNumber0`) close together in the factory make it easy to read the code and understand how everything works.

If you like the simplicity of centralizing all knowledge in the factory but need to support arbitrary class names, it can be a challenge to make the factory open. While the following example doesn't end up *perfectly* open it gets most of the way there, and in many circumstances will be good enough.

The first step towards an open factory that both centralizes knowledge and supports arbitrary class names is to rearrange the code to increase the isolation of the names. You can do this by replacing the `switch` statement with a key/value lookup, as follows:

Listing 7.5: Key/Value Lookup Factory

```
1 class BottleNumber {
2   static for(number) {
3     let bottleNumberClass;
4
5     bottleNumberClass = {
6       0: BottleNumber0,
7       1: BottleNumber1,
8       6: BottleNumber6,
9     }[number] || BottleNumber;
10
11    return new bottleNumberClass(number);
12  }
13  // ...
14 }
```

The above example maps `number` to class name (lines 4-6), attempts to select a class based on the value of `number` (line 7), and defaults to `BottleNumber` (also line 7) when the number can't be found. It then (line 9) creates a new instance of the selected class.

This key/value lookup factory *looks* very different from the previous two examples (Listing 7.3: Factory Without Case 6 and Listing 7.4: Meta Programmed Class Lookup Factory), but despite their syntactical differences these three examples are logically very similar. Each uses the value of `number` to choose a class name. The simple `switch` statement, the meta-programmed class name selection, and the key/value lookup are simply different ways to express this set of conditionals:

If the value of number is 0, select `BottleNumber0`.
If the value of number is 1, select `BottleNumber1`.
If the value of number is 6, select `BottleNumber6`.
Otherwise, select `BottleNumber`.

The `switch` statement factory is simple and allows arbitrary class names, but is closed. The meta-programmed factory is more complicated and requires a class naming convention, but is open. The key/value factory is similar to the `switch` statement factory in that it allows arbitrary class names, but it's a bit harder to read.

It's easy to comprehend the overall behavior of the `switch` statement because it's a simple list of "if this, do that" statements. This key/value version is slightly more complicated because the *data* has been separated from the *algorithm*. In this example, the "this⇒that" bits (the data) have been grouped together in one place (the map from number to class name) and the "if" bits (the algorithm) moved to another (the `[]` lookup logic). When reading the code, you have to combine these two things in your own head in order to understand what it does.

The benefit of this separation is that you can now think of the driving data as an entity in itself, separate from the choosing algorithm. The algorithm lives in the code but you can store the data in an external file, or your database, and read it at initialization time. You might even create a nice user interface to update the database. You'll have to update the database whenever a new class is added, but that's a small price to pay for being able to change the behavior of your application without altering the actual code.

Before moving on, there's one more difference between the `switch` and key/value lookup code that's worth noting. Take another look at those two factory variants (repeated below for convenience), this time paying particular attention to the colors used in the syntax highlighting.

Listing 7.6: Simple Conditional Factory Redux

```
class BottleNumber {
  static for(number) {
    let bottleNumberClass;
    switch (number) {
      case 0:
        bottleNumberClass = BottleNumber0;
        break;
      case 1:
        bottleNumberClass = BottleNumber1;
        break;
      case 6:
        bottleNumberClass = BottleNumber6;
        break;
      default:
        bottleNumberClass = BottleNumber;
        break;
    }

    return new bottleNumberClass(number);
  }
  // ...
}
```

Listing 7.7: Key/Value Lookup Factory Redux

```
class BottleNumber {
  static for(number) {
    let bottleNumberClass;

    bottleNumberClass = {
      0: BottleNumber0,
```

```
      1: BottleNumber1,
      6: BottleNumber6,
    }[number] || BottleNumber;

    return new bottleNumberClass(number);
  }
  // ...
}
```

Notice that in the `switch` version, the colors alternate, while in the key/value version, like colors group more closely together. These groupings say something about the code.

When the colors change constantly it means that the code changes topics a lot. When the colors are more tightly grouped it means that ideas or abstractions that are alike are close together.

Procedures are often characterized by many changes of color. Even if you are completely unfamiliar with this code, you can guess that the `switch` statement factory is a procedure simply by looking at the alternating colors in the syntax highlighting. Code that is more object-oriented tends to group like things together, with fewer changes of topic. This results in more consistent colors as in the key/value factory.

The upside of procedures is that simple ones (short and without conditionals) are easy to understand. The downside is that complex ones (long and with many conditionals) are costly to change. The most efficient, expedient way to fulfill a new requirement may be to write a simple, unglamorous procedure, but if this procedure needs to change it should be converted into object-oriented code. Procedural code can save you money when used to create small, isolated features that never need to change, but this style of coding will break the bank if used on large, shared features that are core to your domain.

OO asks you to break code up into small, cohesive pieces. The benefit of having smaller pieces is that each individual piece, relative to its procedural analog, is easier to understand and change. The corollary downside is that dividing code into many small pieces can obscure the operation of the whole.

The straightforwardness of simple procedures can make them seem attractive, and indeed, they're fine as long as *nothing ever changes*. However, if your code needs to adapt and grow, it's worth paying the toll charged by OOP.

7.5. Dispersing The Choosing Logic

As stated above, the three examples shown thus far all contain the same basic underlying logic. In each case the factory knows everything. It owns the choosing logic (*the value of number is n*), it knows the things that might be chosen (*the class names*), and it contains the logic to map between the two (*number n means class x*).

Owning the choosing logic makes sense when it's simple and stable, as in the current example. But it's easy to imagine situations where the choosing logic is far more complicated. The logic needed to select the right class might be long, complex, and more closely related to the class

being chosen than to the factory itself. If the choosing logic changes in lockstep with code that lives in the class being chosen, then the choosing logic belongs in that class, not in the factory.

In this scenario, each choose-able object implements its own method to determine if it should be chosen. The factory then iterates over the possible objects and asks *them* to make the decision.

Here's a simple, closed form of this kind of factory:

Listing 7.8: Dispersing the Choosing Logic

```
1 class BottleNumber {
2   static for(number) {
3     const bottleNumberClass = [
4       BottleNumber6,
5       BottleNumber1,
6       BottleNumber0,
7       BottleNumber,
8     ].find(candidate => candidate.canHandle(number));
9
10    return new bottleNumberClass(number);
11  }
12
13  static canHandle(number) {
14    return true;
15  }
16  // ...
17 }
18
19 class BottleNumber0 extends BottleNumber {
20   static canHandle(number) {
21     return number === 0;
22   }
23   // ...
24 }
25
26 class BottleNumber1 extends BottleNumber {
27   static canHandle(number) {
28     return number === 1;
29   }
30   // ...
31 }
32
33 class BottleNumber6 extends BottleNumber {
34   static canHandle(number) {
35     return number === 6;
36   }
37   // ...
38 }
```

Each class above now implements method `canHandle(number)`. The implementations in `BottleNumber0`, `BottleNumber1`, and `BottleNumber6` (lines 21, 28 and 35) return `true` when the number is the one they represent. The `BottleNumber` implementation (line 14) unconditionally returns true because `BottleNumber` is the default.

The `for` method in `BottleNumber` on line 2 iterates over a hard-coded list of class names with `find`. The `find` method calls a function with each item and returns the first item for which that function returns `true`. This code therefore manufactures an instance of the first class on the list that responds `true` to `canHandle(number)`.

This factory disperses the choosing logic into the things chosen. In this example, that logic is so simple that this technique is excessive, but in some situations, choosing will involve lots of code, and that code will change in lockstep with the class being chosen. Those are the cases where this technique saves you money.

The structure of this factory brings up several issues. First, it's closed. Each time a new class is added you must update the list on lines 4-7.

Next, since `BottleNumber.canHandle` always returns `true`, `BottleNumber` must always be the last class on the list. If someone mistakenly adds a new class *after* `BottleNumber`, the factory will never manufacture an instance of that new class.

Finally, it's possible that more than one `canHandle` methods would return `true`. The code above stops looking the first time a candidate answers 'Me!' but it's possible to imagine scenarios where you should collect all candidates who answer yes and give each a chance to execute. In that case candidates might also want to report a priority so you can sort the resultant list in the order in which they should go.

The example above shows the simplest way to disperse choosing logic. It might be all you need. If you're concerned about keeping the list in order, or need to deal with multiple candidates wanting to volunteer, you'll have to write a bit more code.

In each case the basic issue remains the same. If your choosing logic is more closely related to the class being chosen than to the factory, the choosing logic should be co-located in that class.

7.6. Self-registering Candidates

The example above disperses the choosing logic, but the factory still has a hard-coded list of the candidate classes. The implementation requires that you manually add newly created classes to this list. If you would like the factory to simply continue working when new candidates appear, you have two basic options.

1. The factory could dynamically figure out which classes belong on its list, or
2. classes who want to be on the list could explicitly ask the factory to put them there.

Choice #1 above is possible only if there's something about the candidate classes that allows the factory to identify them, and this may not always be true. Choice #2, however, is always an option. If candidates are willing to depend on knowing the name of the factory, they can assume responsibility for putting themselves on the list. Lists like these are often referred to as *registries*.

Because choice #2 is always possible, it's the next example. The `BottleNumber` factory below:

1. holds onto the registry, and
2. provides a way for candidates to add themselves to it.

Listing 7.9: Creating a Registry

```
1 class BottleNumber {
2   // ...
3   static register(candidate) {
4     BottleNumber.registry.unshift(candidate);
5   }
6   // ...
7 }
8
9 BottleNumber.registry = [BottleNumber];
```

Line 9 above defines the `registry` property, which is initialized to an array which contains `BottleNumber` by default. Line 3 creates a `register(candidate)` method, which adds candidate arguments to the front of this registry.

Now that this registry exists, candidate classes can register themselves. Here's how that looks for `BottleNumber0` (line 5 below):

Listing 7.10: Candidate Registration

```
1 class BottleNumber0 extends BottleNumber {
2   // ...
3 }
4
5 BottleNumber.register(BottleNumber0);
```

Once you make the above change for the other bottle number classes (`BottleNumber1` and `BottleNumber6`), the hard-coded list of candidates in the factory can be replaced by a reference to the registry:

Listing 7.11: Referring to the Registry in the Factory

```
1  class BottleNumber {
2    static for(number) {
3      const bottleNumberClass = BottleNumber.registry.find(
4        candidate => candidate.canHandle(number)
5      );
6
7      return new bottleNumberClass(number);
8    }
9    // ...
10 }
```

Here's all the relevant code together in one listing:

Listing 7.12: Factory With Self Registration

```
1  class BottleNumber {
2    static for(number) {
3      const bottleNumberClass = BottleNumber.registry.find(
4        candidate => candidate.canHandle(number)
5      );
6
7      return new bottleNumberClass(number);
8    }
9
10   static register(candidate) {
11     BottleNumber.registry.unshift(candidate);
12   }
13   // ...
14 }
```

```
15 |
16 |BottleNumber.registry = [BottleNumber];
17 |
18 |class BottleNumber0 extends BottleNumber {
19 |  // ...
20 |}
21 |
22 |BottleNumber.register(BottleNumber0);
23 |
24 |class BottleNumber1 extends BottleNumber {
25 |  // ...
26 |}
27 |
28 |BottleNumber.register(BottleNumber1);
29 |
30 |class BottleNumber6 extends BottleNumber {
31 |  // ...
32 |}
33 |
34 |BottleNumber.register(BottleNumber6);
```

There are a few things to note about the listing above. First, notice that the registration lines explicitly state the class name `BottleNumber`:

```
BottleNumber.register(BottleNumber0);
```

instead of using the name of the subclass and expecting to inherit the `register` method, like so:

```
register(self)

BottleNumber0.register(BottleNumber0);
```

Since it doesn't mention `BottleNumber` by name, this second example may seem as if it contains fewer dependencies. If fact, the previous two examples contain exactly the same number of dependencies; they just depend on different things.

The `BottleNumber.register(BottleNumber0)` example sends the `register` message to `BottleNumber` and so depends on knowing the name of the factory. If that name later changes, this code will also have to change.

The `BottleNumber0.register(BottleNumber0)` example sends the `register` method to the subclass `BottleNumber0`. This code relies on finding `register` somewhere in the class hierarchy, and so depends on inheritance. If someday you move the bottle number role-playing class out of the `BottleNumber` hierarchy, you'll have to change this line of code.

Remember that any class that implements the `BottleNumber` API can play the bottle number role. In the current example, `BottleNumber0`, `1` and `6` use inheritance to acquire parts of this API, but there's no rule that says you have to use inheritance. Your situation might be such that, for good reasons, it makes sense to create new classes that implement the entire API without inheriting from `BottleNumber`.

Choosing between depending on a class name versus depending on inheritance means placing a bet on which dependency is more stable. Is it more likely that the name of the factory will change, or that role players will stop using inheritance? If you think the factory name is more

stable than the use of inheritance, you should explicitly direct the `register` message to `BottleNumber`. If you fear that the factory name might change but believe that you'll always use inheritance, your code should rely on finding `register` in the hierarchy.

No matter how you decide in the present, your choice can only be a guess. Pay attention to how your guesses turn out and they'll get better. One reason experienced programmers are good at writing change-tolerant code is that they've built up a set of internal guidelines about how to guess well. They understand that although dependencies can't be avoided, they can be deliberately chosen with an eye towards stability.

This factory is now open for extension, has dispersed the choosing logic, and allows candidates to register themselves. It can manufacture instances of classes whose names it does not know, for reasons of which it is unaware.

7.7. Summary

Maintainable OO code rests on polymorphism, on constructing applications from families of small, interchangeable objects that represent variants of a role. Instead of writing classes that contain a bunch of conditionals that choose behavior, polymorphism asks you to disperse variants of behavior into classes of their own.

Placing variants into separate classes eliminates the need for conditionals inside those classes, but it does not completely eliminate the need for conditionals; it just kicks the proverbial conditional can down the road (or back in the stack). In every situation where a role-playing object is needed, some code, somewhere has to know enough to pick the right one.

Enter factories.

Factories are where conditionals go to die. They contain conditionals that select *classes*, and they isolate those conditionals in a single, easily-tested place. They hide the names of role-playing classes and so allow the rest of your application to depend on the API of a role rather than on the concrete names of whatever classes currently exist.

This chapter explored the various forms a factory might take, and considered the trade-offs involved. No factory, whether open or closed, whether it owns the choosing logic or asks candidates if they should be chosen, or whether it reaches out for registrants or accept volunteers, is perfect for every situation. All factories, however, enable polymorphism and thus improve your code.

8. Developing a Programming Aesthetic

Until now, most changes to the "99 Bottles" code have been the result of formal refactorings. You've been an active participant in that you've selected code smells to attack and picked recipes to follow, but the resulting code was primarily dictated by those recipes. The process has been both prescriptive and proscriptive; the refactoring recipes tell you what to do while at the same time forbidding you from wandering off to tinker on tangental shiny things.

This chapter lifts its gaze and considers a few problems where the solutions aren't so clear cut. Sometimes neatly arranged, recipe-driven, fully working code remains unsatisfying because it feels like it isn't quite good enough. Perhaps it retains a bit of duplication, or has a place where adding another abstraction would make it easier to fulfill an anticipated requirement, or is implemented in a way that makes you fear for its maintainability.

Voluntarily altering working code costs money, and doing so declares that you believe that rearranging this code right now is more important than anything else on the backlog. The opportunity cost of improving existing code is that you can't simultaneously attend to other urgent things.

If you're bothered by something in code, you likely have an idea of what you'd prefer. The conundrum posed by these fuzzier situations is not in figuring out what to do, but in deciding whether you're justified in doing anything at all.

Code smells and refactoring recipes represent the distilled judgement of many deeply experienced OO practitioners. Those folks wrote piles of code, both good and bad, and over time noticed correlations between code arrangements and costs. Close attention to ultimate outcomes led them to develop a sense of what to actively do (or diligently avoid) in present code to preclude future pain. They developed a feeling, an "aesthetic", about the rightness of code, and this aesthetic guided their decisions in times of confusion and uncertainty.

Over time their aesthetic sense of what was pleasing and beneficial got codified into a set of rules, or heuristics. These heuristics became the code smells and refactoring recipes that have been bequeathed to you. The definitions and directions contained therein cover a lot of ground and solve many problems, but do not substitute for developing a programming aesthetic of your own.

The goal of this chapter is to start you down that path.

8.1. Appreciating the Mechanical Process

Before diving into programming aesthetics, take a minute to appreciate the current code. If you revert back to the simplest factory from Chapter 7, it looks like this:

Listing 8.1: Six Pack With Switch Statement Factory

```
class Bottles {
  song() {
    return this.verses(99, 0);
```

```
  }

  verses(starting, ending) {
    return downTo(starting, ending)
      .map(i => this.verse(i))
      .join('\n');
  }

  verse(number) {
    const bottleNumber = BottleNumber.for(number);

    return (
      capitalize(`${bottleNumber} of beer on the wall, `) +
      `${bottleNumber} of beer.\n` +
      `${bottleNumber.action()}, ` +
      `${bottleNumber.successor()} of beer on the wall.\n`
    );
  }
}

class BottleNumber {
  static for(number) {
    let bottleNumberClass;
    switch (number) {
      case 0:
        bottleNumberClass = BottleNumber0;
        break;
      case 1:
        bottleNumberClass = BottleNumber1;
        break;
      case 6:
        bottleNumberClass = BottleNumber6;
        break;
      default:
        bottleNumberClass = BottleNumber;
        break;
    }

    return new bottleNumberClass(number);
  }

  constructor(number) {
    this.number = number;
  }

  toString() {
    return `${this.quantity()} ${this.container()}`;
  }

  quantity() {
    return this.number.toString();
  }

  container() {
    return 'bottles';
  }

  action() {
    return `Take ${this.pronoun()} down and pass it around`;
  }

  pronoun() {
    return 'one';
  }
```

```
68 |
69 |   successor() {
70 |     return BottleNumber.for(this.number - 1);
71 |   }
72 |}
73 |
74 |class BottleNumber0 extends BottleNumber {
75 |   quantity() {
76 |     return 'no more';
77 |   }
78 |
79 |   action() {
80 |     return 'Go to the store and buy some more';
81 |   }
82 |
83 |   successor() {
84 |     return BottleNumber.for(99);
85 |   }
86 |}
87 |
88 |class BottleNumber1 extends BottleNumber {
89 |   container() {
90 |     return 'bottle';
91 |   }
92 |
93 |   pronoun() {
94 |     return 'it';
95 |   }
96 |}
97 |
98 |class BottleNumber6 extends BottleNumber {
99 |   quantity() {
100|     return '1';
101|   }
102|
103|   container() {
104|     return 'six-pack';
105|   }
106|}
```

This code is a tribute to the intense simplicity achieved by deft handling of complexity. It's a testament to the efficacy of programming by identifying smells and removing them using well-known refactoring recipes. Consider how the code above might look had you charged forward from the end-point of your original 30-minutes-of-effort solution, or even if you'd started from the Incomprehensibly Concise, Speculatively General, or Concretely Abstract implementations in Chapter 1. It's terrifyingly easy to imagine solutions that are far more complicated and far less revealing of intentions.

As you have probably long since realized, the 99 Bottles problem is more nuanced than it initially appears. Songs like "99 Bottles" are great for teaching deep lessons about dealing with complexity. The song is simple enough to be grasped quickly, yet provides fodder for endless discussions about the subtle distinctions between differing concepts. While the finely sliced code above (pleasing though it is) may seem a bit grandiose for this situation, the techniques used to create it scale to infinitely more difficult problems. Songs are great because they allow you to easily practice techniques that can then be used to conquer fiendishly complex domains.

The "99 Bottles" song, in particular, has a long history of being used for this purpose. Renowned computer scientist Donald Knuth wrote an article over 40 years ago titled *The Complexity of Songs* which directly references "*m* Bottles of Beer on the Wall." The 99 Bottles problem is so commonly tackled that there are over 1500 implementations in various programming languages. The song is perfect fodder for teaching; it's simultaneously simple enough to learn from and hard enough to make the lessons useful.

This book is just one more link in a long chain. The story the current implementation tells is that all verses are alike in an abstract way, and that bottle numbers vary within each verse. The `song` and `verses` methods of `Bottles` accumulate individual verses. The `verse` method (repeated below) turns a number into a bottle number and then uses that bottle number to produce verses.

Listing 8.2: Verse as Described With the Word And

```
1 verse(number) {
2   const bottleNumber = BottleNumber.for(number);
3
4   return (
5     capitalize(`${bottleNumber} of beer on the wall, `) +
6     `${bottleNumber} of beer.\n` +
7     `${bottleNumber.action()}, ` +
8     `${bottleNumber.successor()} of beer on the wall.\n`
9   );
10 }
```

The presence of the word "and" in the previous sentence should arouse your suspicions—it implies that the `verse` method does two things. That pesky blank line on line 3 above may cause further concern. Programmers add blank lines to indicate changes of topic, and changes of topic suggest multiple responsibilities. Blank lines smell.

Taken together, these misgivings might tempt you to proactively refactor `verse` into two smaller methods, one to convert `number` into a bottle number and the other to produce the actual verse.

Despite this very reasonable temptation, there's a good argument for leaving `verse` as is. For goodness' sake, the method works, it contains only six lines of code, and no one has asked for a new feature that would force any alterations. Breaking `verse` into multiple methods would introduce additional message sends and add extra levels of indirection. Increasing indirection makes code harder to follow.

Perhaps it's best to leave well enough alone and sneak quietly away.

In situations like this, what should you do? Is it better to voluntarily rewrite this method to separate the responsibilities, or should you leave it as is and see what the future brings? From a broader perspective, is it *ever* okay to voluntarily improve code, or should you always restrict yourself to writing the minimal code dictated by a refactoring recipe or necessary to fulfill an explicit requirement?

These are some of the most urgent questions for the practical programmer. What they have in common is uncertainty. Choosing to make a voluntary change places a bet that the change's cost will be repaid by a future offsetting reduction. Some potential changes are such good bets that they should always be made, and others such long shots as to be consistently avoided.

The choice about when to voluntarily change code relies on judgement. Judgement is informed by past experience. Experience accumulates into an intuition about how best to act in the face of uncertainty. Intuition is a form of pattern matching performed by your unconscious mind, trained throughout your career on scores of code examples.

Because your unconscious mind can't talk, this is where programming begins to feel like an art. Those feelings you have about the rightness of code are likely correct, but the big super-computer of your unconscious mind can't supply words to defend them. Sadly, advocating changes to code based on feelings you can't explain is not likely to be convincing.

Thus, while intuition helpfully supplies feelings about code to draw your attention to things that might benefit from action, it's the job of your conscious brain to figure out how to put words on those feelings. These words form your programming aesthetic, or the set of principles that underly and guide your work. Intuition drives action, justified by aesthetics, and guided by heuristics.

There's much to explore about programming aesthetics, but having briefly pondered these broader questions, put them aside for now. Just in the nick of time you're rescued by the arrival of a new requirement.

Your customer wants other songs that are similar to "99 Bottles" but contain different lyrics.

8.2. Clarifying Responsibilities with Pseudocode

Your first question should be "Similar in what way?" Recall the `verses` method:

Listing 8.3: Verses Counts Down

```
1 verses(starting, ending) {
2   return downTo(starting, ending)
3     .map(i => this.verse(i))
4     .join('\n');
5 }
```

This method is responsible for producing a range of verses of the song. The "99 Bottles" song, however, is a bit special in the universe of all songs, and this specialness is exposed by the use of `downTo` on line 2 above. Generally songs start at the bottom and count up, but this song starts at the top and counts down. When your customer asks you to produce other songs like the "99 Bottles" song, they're asking you to write code to produce other songs that also count down.

They're asking for a *variant* that can produce different verses. This means that the code has to change. As you learned in Chapter 3, the first step in deciding what code to write next is to consult the Open-Closed Flowchart.

So:

1. Is the code open to the *vary the verses* requirement?
   No.

2. Do you know how to make it open?
   Probably not.
3. What's the best-understood code smell?
   While it's clear that the `verse` method might be doing too much, it's not clear how identifying and fixing any currently-existing code smell will help.

Paradoxically, when faced with uncertainty about what to do next, it can sometimes help to sigh deeply, ignore everything you've learned, and just write a new conditional. This conditional's purpose is to supply more information about the problem, and writing it can clarify what needs to change. Once you understand what should change, you can discard the conditional and write better code.

If you were to sketch in the conditional needed to produce other kinds of verses, the `verse` method might look like this:

```
verse(number) {
  // if (99BottlesSong) {
    const bottleNumber = BottleNumber.for(number);

    return (
      capitalize(`${bottleNumber} of beer on the wall, `) +
      `${bottleNumber} of beer.\n` +
      `${bottleNumber.action()}, ` +
      `${bottleNumber.successor()} of beer on the wall.\n`
    );
  //
  // } else if (unknownSong2Verse) {
  //   ...
  //   assemble verse for unknown song 2
  //   ...
  // } else if (unknownSong3Verse) {
  //   ...
  //   assemble verse for unknown song 3
  //   ...
  // }
}
```

The pseudocode[18] above is commented out but it doesn't matter that it's not working code—the conditional's mere presence helps clarify what needs to be done. In a backwards way, the purpose of this pseudocode is to *introduce* new code smells. The `verse` method above now contains a *Switch Statement* and is a *Long Function*. These code smells provide a glimpse into the future, imparting information about what will happen if you embark down the solve-the-problem-by-adding-conditionals path. The imaginary conditional provides visible, incontrovertible proof that nothing good will come of this.

Now that you can anticipate the smells, you can preemptively choose one and apply the curative refactoring. The function is long because of the conditional so you can ignore that smell, leaving only *Switch Statement*. Each of the imaginary branches represents an unrelated set of lyrics, so the best way to cure this conditional is to apply *Extract Class* to each branch.

Discard the pseudocoded conditional and revert to the earlier `verse` method. You can think of the original code as one branch of an imaginary conditional that needs to be extracted into a

class of its own.

8.3. Extracting the Verse

You've already practiced *Extract Class*, back in the Extracting BottleNumber section of chapter 5 where a new class was created to cure the *Primitive Obsession* on `number`. Here's a reminder of that recipe:

- Choose a class name and create the new class.
- Add a property and a `constructor` method to encapsulate primitive data.
- Copy the methods from the old class to the new.
- Forward messages from the old class to the new.
- One by one, remove arguments from the methods in the new class, and corresponding parameters from the message sends in the old class.

The first concern, as always, is what to name the new class. Classes should be named for exactly what they are, so a class that represents a verse in the "99 Bottles" song could reasonably be named `BottleVerse`, as shown below.

Listing 8.5: Create BottleVerse

```
class BottleVerse {
}
```

`BottleVerse` needs a `constructor` method that accepts a `number` argument, and a property to store this number. Here's that code:

Listing 8.6: Add Attr Reader and Initialize

```
class BottleVerse {
  constructor(number) {
    this.number = number;
  }
}
```

You haven't yet changed anything in `Bottles#verse` so running tests at this point confirms that the old code still works and proves that the new `BottleVerse` class is syntactically correct. (Note that the `#` in `Bottles#verse` is a shortcut denoting that `verse` is an instance method on `Bottles`. Had `verse` been a static method, this would have been written `Bottles.verse`.)

Because you're creating a new class by following a refactoring recipe instead of doing TDD, you would normally want the new class to be fully plugged into the old before altering any of the new class's code. The safest process is to first assemble the new class from code exactly copied from the old, and next to alter the old class to invoke the code in the new. Only then should you return to the new class to make improvements or changes. Working this way minimizes the chance that you'll break something, and makes things that you do accidentally break easier to debug.

The next step, then, is to copy the entire `verse` method from `Bottles` to `BottleNumber`. This looks like:

Listing 8.7: Copy Verse to BottleNumber

```
class BottleVerse {
  // ...
  verse(number) {
    const bottleNumber = BottleNumber.for(number);

    return (
      capitalize(`${bottleNumber} of beer on the wall, `) +
      `${bottleNumber} of beer.\n` +
      `${bottleNumber.action()}, ` +
      `${bottleNumber.successor()} of beer on the wall.\n`
    );
  }
}
```

As always, run the tests after this change. They should again confirm that the old code works, and the new code is syntactically correct.

Now you can start integrating this new class back into the class from which it was extracted. Returning to `Bottles`, insert a line into its `verse` method that creates an instance of `BottleVerse` and forwards the `verse` message on, as shown on line 4 below.

Listing 8.8: Create a BottleVerse

```
class Bottles {
  // ...
  verse(number) {
    new BottleVerse(number).verse(number);
    const bottleNumber = BottleNumber.for(number);

    return (
      capitalize(`${bottleNumber} of beer on the wall, `) +
      `${bottleNumber} of beer.\n` +
      `${bottleNumber.action()}, ` +
      `${bottleNumber.successor()} of beer on the wall.\n`
    );
  }
}
```

Line 4 above executes the new code but ignores the returned verse. You can't know if the result is correct, but you *have* proven that the new `BottleVerse` code runs without blowing up.

The next step is to actually use that result. This is easily done by commenting out the old code and adding a return, as shown here:

Listing 8.9: Use Result From BottleVerse

```
class Bottles {
  // ...
  verse(number) {
    return new BottleVerse(number).verse(number);
    // const bottleNumber = BottleNumber.for(number);

    // return (
    //   capitalize(`${bottleNumber} of beer on the wall, `) +
```

```
9 |    //   `${bottleNumber} of beer.\n` +
10|    //   `${bottleNumber.action()}, ` +
11|    //   `${bottleNumber.successor()} of beer on the wall.\n`
12|    // );
13|  }
14|}
```

Running the tests confirms that you've successfully extracted the `BottleVerse` class.

At this point the code works but has an annoying quirk: line 4 above refers to `number` twice. This is clearly redundant and should be fixed. These redundancies are echoed in `BottleVerse` (repeated below for convenience). In lockstep with `Bottles`, `BottleVerse` must be initialized with `number` (line 2 below), and also has a `verse` method that requires a `number` argument (line 6).

Listing 8.10: Bottle Verse From the Recipe

```
1 |class BottleVerse {
2 |  constructor(number) {
3 |    this.number = number;
4 |  }
5 |
6 |  verse(number) {
7 |    const bottleNumber = BottleNumber.for(number);
8 |
9 |    return (
10|      capitalize(`${bottleNumber} of beer on the wall, `) +
11|      `${bottleNumber} of beer.\n` +
12|      `${bottleNumber.action()}, ` +
13|      `${bottleNumber.successor()} of beer on the wall.\n`
14|    );
15|  }
16|}
```

These redundant references are transient and exist because of the way `BottleVerse` was extracted. The recipe allows you to extract a new class without breaking an existing test. In order to keep the tests running, you have to copy code. The code copied to the new class has no test coverage so it oughtn't be changed until it's fully wired-in to the old class and safely under the protection of the existing tests.

This style of coding leans heavily upon the refactoring recipes, operating in faith that they will eventually result in working code. Since the extracted class is created using a time-tested recipe, the class doesn't have to be built test-first. Not that the extracted class doesn't need tests of its own—it surely does, and will eventually get them—but the tests don't need to be written *first*. You're isolating existing behavior, not discovering new behavior. In situations like this, the recipes supply the most straightforward and direct path to your coding goal.

You've now created a `BottleVerse` class that represents a verse. Granted, it still contains a few redundant copies of `number`, but even in its current state the `Bottles` class can use it instead of generating verses itself.

At this point, the standard recipe calls for cleaning up the extraneous references to `number`. In the spirit of developing a programming aesthetic, the next section invokes intuition and follows an alternative path.

8.4. Coding by Wishful Thinking

Trusting the *Extract Class* recipe reduces coding friction by allowing you to jump right into extracting the `BottleVerse` class, but it requires that you refrain from changing the copied code until it's fully used. This restriction on changing copied code often leaves redundant arguments, which must be subsequently dealt with.

You likely recall encountering a similar problem during the `BottleNumber` extraction in Chapter 5 where removing the redundant arguments required a series of meticulous steps (detailed in the Removing Arguments section). It's important to understand how to follow those detailed steps, but now that you do understand them and can fall back on them if trouble arises, you can allow yourself more leeway. This section explores a more advanced technique that transforms the code in bigger steps.

Software developers have long relied on structured speculation. Pseudocode is speculative, and it has already been used to reveal the code smells that would have arisen had you added a new conditional to generate other counting-down songs. That pseudocode quickly and painlessly revealed a shortcut to a better path.

TDD is yet another example. Test-driving code begins with a speculative leap—the first thing you do is write a test you wish would pass. Initially, of course, this test fails. You then shift perspectives and write the production code to pass that test. Once this succeeds, you toggle back into test writing mode and speculatively create the next test. Done well, this alternating process streamlines the proving of ideas and produces working code almost as quickly as it's conceived.

The seminal book *Structure and Interpretation of Computer Programs (SICP)* contains several references to "wishful thinking." The authors refer to it as "a powerful strategy of synthesis."

Like pseudocode and TDD, coding by wishful thinking allows you to sketch software design ideas economically, with a low level of commitment. In other words, you can guess, unburdened by penalties for being wrong. This approach to writing code can feel unruly and indulgent, but it's a bona fide, efficient, and often elegant technique for making progress, and can be applied right here.

The echo of `number` in

```
new BottleVerse(number).verse(number)
```

feels clumsy and redundant. The techniques shown in the Removing Arguments section of Chapter 5 can certainly be used to clean up this echo, but sometimes it makes sense to solve problems like this by considering the broader domain.

What code do you wish you had? What message should `Bottles` expect to send to `BottleVerse` to get back the words in a verse? Think about this from the message sender's point of view: What does `Bottles` want from `BottleVerse`?

It wants lyrics.

If `Bottles` wants lyrics from `BottleNumber` it should just ask, like so:

```
BottleVerse.new(number).lyrics
```

```
new BottleVerse(number).lyrics()
```

Now that you've expressed this wish, it's easy to make it come true. Revert back to the original `Bottles#verse` method and insert the wishful code, as shown below:

Listing 8.11: Lyrics by Wishful Thinking

```
1 class Bottles {
2   // ...
3   verse(number) {
4     // return new BottleVerse(number).lyrics();
5     const bottleNumber = BottleNumber.for(number);
6
7     return (
8       capitalize(`${bottleNumber} of beer on the wall, `) +
9       `${bottleNumber} of beer.\n` +
10      `${bottleNumber.action()}, ` +
11      `${bottleNumber.successor()} of beer on the wall.\n`
12    );
13  }
14 }
```

Line 4 above doesn't yet work and so is temporarily commented out.

The next step is to go to `BottleVerse` and rename the

`verse(number)`

method to

`lyrics`

and refer to the number property rather than referencing the parameter, as shown on line 3 and 4 below:

Listing 8.12: BottleVerse Responds to Lyrics

```
1 class BottleVerse {
2   // ...
3   lyrics() {
4     const bottleNumber = BottleNumber.for(this.number);
5
6     return (
7       capitalize(`${bottleNumber} of beer on the wall, `) +
8       `${bottleNumber} of beer.\n` +
9       `${bottleNumber.action()}, ` +
10      `${bottleNumber.successor()} of beer on the wall.\n`
11    );
12  }
```

`BottleVerse` isn't being used right now so running the tests just proves that this code is syntactically correct.

Now return to `Bottles` and uncomment the wishful line 4, as you see here:

Listing 8.13: Parse and Execute Lyrics

```
1 class Bottles {
2   // ...
3   verse(number) {
4     new BottleVerse(number).lyrics();
5     const bottleNumber = BottleNumber.for(number);
6
7     return (
8       capitalize(`${bottleNumber} of beer on the wall, `) +
9       `${bottleNumber} of beer.\n` +
10      `${bottleNumber.action()}, ` +
11      `${bottleNumber.successor()} of beer on the wall.\n`
12    );
13  }
14 }
```

Line 4 above executes the new `BottleVerse#lyrics` method but ignores its result.

Finally (drum roll), delete the original implementation from the method above and add a return. This reduces `Bottles#verse` to:

Listing 8.14: Use Result of Lyrics

```
1 class Bottles {
2   // ...
3
4   verse(number) {
5     return new BottleVerse(number).lyrics();
6   }
7 }
```

The `Bottles#verse` method now gets lyrics from a different class, `BottleVerse`.

Coding by wishing thinking led to the discovery of a more intention-revealing interface for `BottleVerse`.

Notice that while this technique, making multiple changes to `BottleVerse` before using it in `Bottles`, reduces the number of refactoring steps, it also adds risk. Running the tests after each single-line change to `BottleVerse` proves that the new code is syntactically correct, but not that it actually works. Without care, you could easily write a big pile of buggy code before plugging the `BottleVerse` class back into `Bottles` and discovering that it doesn't work.

If you plug `BottleVerse` into `Bottles` and tests fail, the rules of refactoring say you must undo and fix the offending code. If you can't immediately fix the problem, either drop back and use the original technique that takes smaller steps, or TDD the new `BottleVerse`.

Despite the added risks, this alternate technique can be very efficient. It's best used in cases where you're extracting a class and want to make a few small changes. If the extracted class needs lots of alterations you'll likely have better luck TDDing it from the start. In the future, take the opportunity to expand your programming aesthetic by trying both techniques and paying attention to how your choices turn out.

This completes the extraction of `BottleVerse`, a new class that responds to `lyrics`.

8.5. Inverting Dependencies

Recall that the impetus behind extracting `BottleVerse` was the need to produce songs with other lyrics. Despite having completed the extraction, you can't yet fulfill this vary-the-verse requirement. Why not? Because `Bottles` is currently stuck to `BottleVerse`. You have extracted the class, but not yet inverted the dependency.

The Dependency Inversion Principle (DIP) contributes the 'D' in the SOLID acronym and can be defined as "depend on abstractions, not concretions." This section demystifies the principle and employs it to loosen the coupling between `Bottles` and `BottleVerse`.

Here's a reminder of the current `Bottles` and `BottleVerse` classes:

Listing 8.15: Bottles Depends on BottleVerse

```
1  class Bottles {
2    song() {
3      return this.verses(99, 0);
4    }
5
6    verses(starting, ending) {
7      return downTo(starting, ending)
8        .map(i => this.verse(i))
9        .join('\n');
10   }
11
12   verse(number) {
13     return new BottleVerse(number).lyrics();
14   }
15 }
16
17 class BottleVerse {
18   constructor(number) {
19     this.number = number;
20   }
21
22   lyrics() {
23     const bottleNumber = BottleNumber.for(this.number);
24
25     return (
26       capitalize(`${bottleNumber} of beer on the wall, `) +
27       `${bottleNumber} of beer.\n` +
28       `${bottleNumber.action()}, ` +
29       `${bottleNumber.successor()} of beer on the wall.\n`
30     );
31   }
32 }
```

On line 13 above the `verse` method of `Bottles` knows the name of the concrete `BottleVerse` class. `Bottles` therefore depends on `BottleVerse`. Put another way, there's a tight coupling between `Bottles` and `BottleVerse`.

This coupling has consequences, the least of which is that if the name of the `BottleVerse` class changes, `Bottles` will also have to change. Being forced to go through your entire codebase and

change a bunch of explicit references from `BottleVerse` to another name would be inconvenient but not devastating.

A far worse repercussion is that `Bottles` can't collaborate with any class other than `BottleVerse`. Even if you had an entire library of objects that returned lyrics for different songs, and even if those objects looked just like `BottleVerse` from the outside, meaning that they conformed to the same API, `Bottles` couldn't use them. `Bottles` is glued to `BottleVerse` and can't acquire lyrics from any other object.

8.5.1. Injecting Dependencies

In order for `Bottles` to produce varying lyrics without resorting to a conditional, the code has to be rearranged so that `Bottles` can seamlessly talk to *any* lyrics provider. It's time to loosen the coupling between `Bottles` and `BottleVerse`.

`Bottles` depends on, or has knowledge about, two different `BottleVerse`-related things. It knows:

1. a concretion, that is, the name of the `BottleVerse` class, and
2. an abstraction, namely, the idea that there's an object that can provide a verse.

Knowing the abstraction is required. It's a fundamental part of `Bottles`'s responsibility to understand that objects exist that can provide verses, and to know how to collaborate with them. `Bottles` has to know these things in order to do its job.

Knowing about the concretion, on the other hand, is completely avoidable. `Bottles` doesn't have to know the concrete class name `BottleVerse`, this name could easily be passed into `Bottles` from the outside. Doing so not only reduces the number of dependencies inside of `Bottles`, but it also opens `Bottles` to an entire universe of existing and potential lyrics providers.

You can think of classes that provide lyrics as playing a common role. The set of messages to which `BottleVerse` responds establishes an API that defines this role. Classes that want to supply lyrics must conform to this API.

Roles need names, and this role could reasonably be named *verse template*. Line 4 below embodies the wish to depend on a player of the verse template role rather than on the `BottleVerse` class:

Listing 8.16: Wishing to Decouple From BottleVerse

```
1 class Bottles {
2   // ...
3   verse(number) {
4     // return new this.verseTemplate(number).lyrics();
5     return new BottleVerse(number).lyrics();
6   }
7 }
```

The wish above breaks the tests so it's temporarily commented out, but identifying it is worthwhile because its presence prompts you to write code to make it come true.

Notice that the wishful code makes the smallest possible change by replacing the `BottleVerse` class name with the `verseTemplate` role name. This line of code chains two dependencies together. First, it knows that `this.verseTemplate(number)` is a constructor function, and second, it knows that invoking that constructor returns an object that responds to `lyrics`. If this chaining concerns you, never fear; it's the topic of the subsequent Obeying the Law of Demeter section. For now just recognize that changing `BottleVerse` to `verseTemplate` doesn't add any additional dependencies. If you are suddenly concerned about line 4 above but weren't previously bothered by line 5, spend a minute trying to articulate your concern before you start reading the Demeter section.

Returning to the current wish, change `Bottles` so that an outside agent passes a `verseTemplate` that defaults to `BottleVerse` during creation (line 2 below). Next, create a property to store `verseTemplate`'s value (line 3):

Listing 8.17: Inject a Verse Template

```
1 class Bottles {
2   constructor(verseTemplate = BottleVerse) {
3     this.verseTemplate = verseTemplate;
4   }
5   // ...
6 }
```

Now that `verseTemplate` is being injected into `Bottles`, the wish should work. Return to the `verse` method, uncomment the wishful line, run the tests, and then delete the line that follows. This shrinks the body of `verse` to the single line of code shown on line 17 below:

Listing 8.18: Use the Verse Template

```
1  class Bottles {
2    constructor(verseTemplate = BottleVerse) {
3      this.verseTemplate = verseTemplate;
4    }
5
6    song() {
7      return this.verses(99, 0);
8    }
9
10   verses(starting, ending) {
11     return downTo(starting, ending)
12       .map(i => this.verse(i))
13       .join('\n');
14   }
15
16   verse(number) {
17     return new this.verseTemplate(number).lyrics();
18   }
19 }
20
21 class BottleVerse {
22   constructor(number) {
23     this.number = number;
24   }
25
```

```
26 |   lyrics() {
27 |     const bottleNumber = BottleNumber.for(this.number);
28 |
29 |     return (
30 |       capitalize(`${bottleNumber} of beer on the wall, `) +
31 |       `${bottleNumber} of beer.\n` +
32 |       `${bottleNumber.action()}, ` +
33 |       `${bottleNumber.successor()} of beer on the wall.\n`
34 |     );
35 |   }
36 |}
```

These changes complete the extraction and injection of `BottleVerse`, but there are a few issues with the current code that should be acknowledged.

First, you may be wondering why `verseTemplate` defaults to the `BottleVerse` class (line 2 above) rather than to an instance of that class. This perhaps brings to mind a broader question: when injecting collaborators, should you inject classes or instances of those classes?

Also, you just went to a fair amount of trouble to remove the hard-coded `BottleVerse` class name from the `verse` method in `Bottles`, and yet there's another a place in this code where a method references a class name. On line 27 above, the `BottleVerse#lyrics` method is tightly coupled to `BottleNumber`. If knowing the concrete name of a class is a bad idea, shouldn't this reference also be injected?

Both of these questions will be addressed in the Obeying the Law of Demeter section. For now, and despite these very legitimate concerns, recognize that the current code has been improved. Not only has `BottleVerse` been decoupled from `Bottles`, but you've also identified and defined a new role—verse template—for which you can create and inject entirely new players.

8.5.2. Isolating Variants

`Bottles` now thinks of itself as interacting with a player of the verse template role rather than a kind of the `BottleVerse` type. Because all role players look the same from the outside, `Bottles` can treat them as if they're identical. `Bottles` can produce as many different songs as you have different verse templates, without itself changing.

`Bottles` is now *composed* of `verseTemplates`.

Here's the process used to create the verse template role:

1. Identify the code you want to vary.
2. Name the underlying concept.
3. Extract the identified code into its own class.
4. Inject this new role-playing object back into the object from which it was extracted.
5. Forward messages from the original class to the injected object.

Here's an illustration. First you extracted the `BottleVerse` class from `Bottles` and then immediately re-injected it, as shown here:

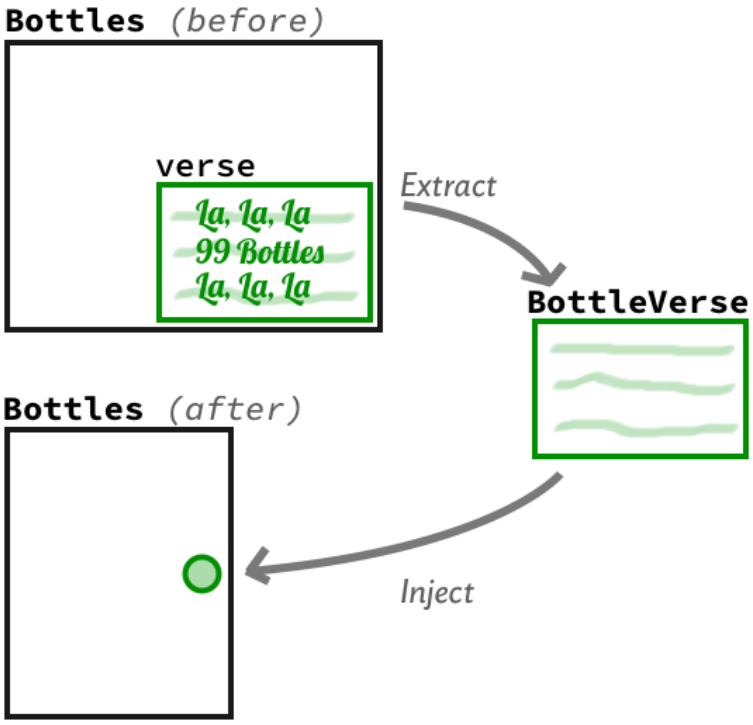

Figure 8.1: Extract and Then Inject BottleVerse

Now that `Bottles` is being passed an object that plays a role, you can invent and inject other objects to play this role. `Bottles` will happily collaborate with any verse template that responds to `lyrics`, as illustrated below:

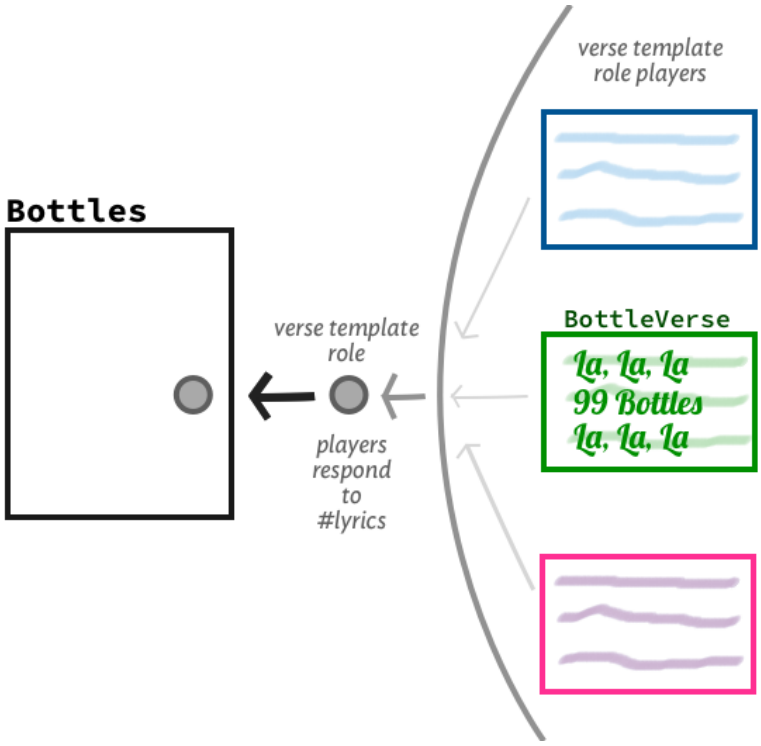

Figure 8.2: Inject a Player of the Verse Template Role

This process can be summarized in a few words: Isolate the behavior you want to vary.

That phrase bears repeating. It may deserve flashing lights.

One of the most fundamental concepts in OO is to isolate the behavior you want to vary.

When a change is needed to a small part of your code, extracting and injecting a role-playing object opens the possibility of creating and injecting other objects that play the same role. The injection point becomes a seam across which objects interact in a loosely-coupled way. These seams permit applications to expand and support new behavior without having to change existing code.

Isolating the bottles variant of the verse template opens your code to the possibility of other variants. It's now possible to fulfill the current requirement by creating and injecting a new class that plays this role.

8.5.3. Grappling with Inversion

The technical name for what happened in the prior refactoring is dependency inversion. The Dependency Inversion Principle (DIP) can be confusing because its very name contains the assumption that you already understand it. Webster's Dictionary defines invert as "to turn over, to put upside down, to place in a contrary order or direction." Inverting a dependency must therefore mean flipping it from one state to that state's opposite. The key to understanding the principle is to recognize that your code should depend on abstractions. If you stumble upon code that's in the state of depending on concretions, DIP says that you should invert those dependencies and depend upon abstractions instead.

Where `Bottles` once had a dependency on the `BottleVerse` concretion, it now has a dependency on the verse template abstraction. Thus the original dependency has been inverted.

This "depend on abstractions, not on concretions" definition distills the essence of the more verbose original DIP definition from the May 1996 issue of C++ Report, which explained it like this:

1. High-level modules should not depend upon low-level modules. Both should depend upon abstractions.
2. Abstractions should not depend upon details. Details should depend upon abstractions.

As a service to humanity, the following paragraphs rephrase that definition using concepts from this code.

First, note that the word "module" in the definition above does not refer to a specific language feature. In this definition module means an encapsulated, named unit of functionality in a program. You can substitute the words "classes" or "objects" for "modules."

`Bottles` is thus the highest-level module in the code. Another way to think about it is that `Bottles` is the outermost class. It's what you started with, and at this point it's the only class that has tests. The entire public API is currently defined in `Bottles`.

Every other class was created by extracting behavior from the high-level module `Bottles`. These extracted classes represent lower-level modules. (If you're feeling annoyed that the highest-level module in an application that produces many songs is named `Bottles`, you're probably not alone. It's time to start pondering a better name. This will be addressed in Chapter 9.)

Class names are concretions. When `Bottles` contained a hardcoded reference to `BottleVerse` it depended directly on that concretion. Thus, a higher-level module had a concrete dependency on a lower-level module. It depended upon a detail rather than an abstraction.

In terms of the current code, the official Dependency Inversion Principle definition can be rephrased as:

1. High-level modules like `Bottles` should not depend on lower-level modules like `BottleVerse`. Each should depend on abstractions.
2. `Bottles` should not depend on concrete details like the name of the `BottleVerse` class. `Bottles` should instead depend on an object that polymorphically generates song verses.

Alternatively, using the shorter form of the DIP definition from Chapter 3, you might say:

- `Bottles` should depend on the `verseTemplate` abstraction rather than the `BottleVerse` concretion.

Isolating variants often requires that you invert dependencies, and an excellent technique for inverting dependencies is to inject them. This section isolated the `BottleVerse` variant and then inverted the dependency by injecting `BottleVerse` as a player of the verse template role.

`Bottles` depends on the verse template role. `BottleVerse` plays this role. Because `Bottles` now depends upon an abstract role rather than a concrete class, you can create and inject other players of the verse template role without needing to change `Bottles`.

You can now fulfill the requirement introduced at the beginning of this chapter by simply creating and injecting a new variant that plays the abstract verse template role but returns different concrete lyrics.

Having reached this point, it seems as if you should be done, but yet again it's time to bring aesthetics into play.

8.6. Obeying the Law of Demeter

The summary of the Injecting Dependencies section voiced concerns about a line of code that contained a chain of dependencies . Lines that contain chained dependencies (often exposed by the presence of many '.'s) might violate the *Law of Demeter (LoD)*. This section defines that law,

determines where it applies, explores the consequences of ignoring it, and explains how to fix violations.

8.6.1. Understanding the Law

Have a look at the following example:

Listing 8.19: Verse Method Contains Many Dependencies

```
1 class Bottles {
2   // ...
3   verse(number) {
4     return new this.verseTemplate(number).lyrics();
5   }
6 }
```

On line 4 above the `verse` method knows:

- that `this.verseTemplate` is a constructor function which can be called with `new`
- that `new` expects an argument
- that the argument to `new` must be a number
- that the object returned from `new …(number)` responds to the message `lyrics`
- that `lyrics` returns the actual lyrics of interest

This list enumerates many things that `Bottles` knows about but doesn't control, which means they're dependencies. Dependencies are vulnerabilities—if their owner changes them, the effects of that change will roll downhill to `Bottles`, which might then be forced to change in turn. Dependencies can't be avoided but should certainly be minimized. Be alert for superfluous dependencies and remove them with extreme prejudice.

None of the extra dependencies in this code are good, but one of them falls into an especially pernicious category. This code violates the Law of Demeter. The problem solved by this law is best explained with a new example.

Consider the following:

Listing 8.20: Many Hops

```
1 class Foo {
2   durabilityOfPreferredToyOfBestFriendsPet() {
3     return this.bestFriend.pet().preferredToy().durability();
4   }
5 }
```

Note: In this and the following permutations of this exaggerated example, assume that the `bestFriend` dependency was injected.

Line 3 above contains a chain of messages, (`bestFriend.pet().preferredToy().durability()`), each of which results in an object that conforms to a different API.

Code like the above, sadly, is so common that this example may not seem surprising. Typical though it may be, describing its dependencies is awkward. Line 3 above requires that `Foo` know:

- that the `pet` message is in the API of the object held in `bestFriend` ,
- that `preferredToy` is in the API of the object that the `pet` message returns, and
- that `durability` is in the API of the object that the `preferredToy` message sent to the object returned by the `pet` message returns.

The fact that the prior sentence is painful to read reflects an underlying problem with the code.

Here's a visualization of the objects and messages:

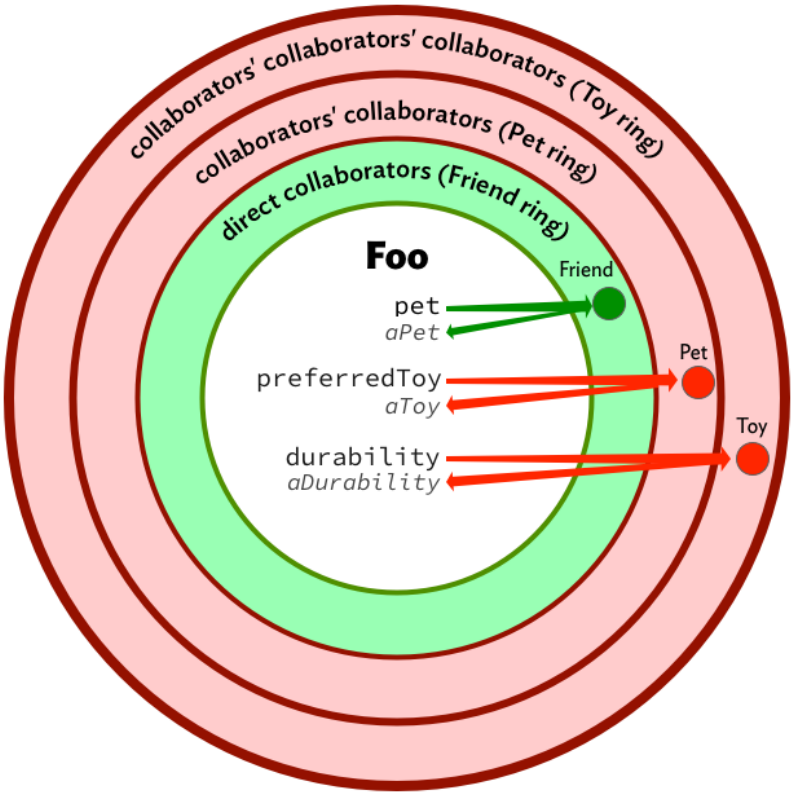

Figure 8.3: Violating the Law of Demeter

In the diagram above, `Foo` first sends `pet` to `bestFriend`. Because `bestFriend` is known to `Foo`, you can think of it as a *direct* collaborator. This returns an instance of `Pet`.

So far so good, but the next thing `Foo` does is send `preferredToy` to that returned `Pet` object. Notice that `Pet` is not a direct collaborator of `Foo`; instead it's a collaborator of `Friend`. This means that `Foo` has knowledge about, or depends on, its collaborators' collaborators.

The process then repeats when `Foo` sends `durability` to the `Toy` returned by `preferredToy`. This message requires that `Foo` know about the API of `Toy`. Consequently, `Foo` has knowledge of its collaborators' collaborators' collaborators.

The underlying code looks innocuous but the diagram makes it inescapably obvious that you can't have a `Foo` unless you can provide it with a `Friend` who has a `Pet` who has a `Toy`. The code may produce the correct output at this moment, but will not age well. This tight coupling across many objects introduces two serious problems which are then blithely lobbed into the future.

First, arranging the code in this way interferes with your ability to use `Foo` in new and unexpected contexts. If an unforeseen feature request arrives that needs `Foo`'s behavior, you won't be able to fulfill that request by merely providing a `Foo` — you'll also need to supply a `Friend` that has a `Pet` that has a `Toy`. These messages tightly couple `Foo` to a chain of different objects, all of which must be available. Nothing here stands alone—this group of objects acts like a single thing and any use requires every piece.

These consequences become obvious when you attempt to reuse `Foo`. Tests serve many purposes, one of which is to reveal how easy it is to reuse code. Tightly-coupled code is difficult to test. Tightly-coupled objects require adding lots of context, all of which must be provided in order to run any test.

If test setup involves creating a bunch of increasingly distant objects, or if you find yourself putting stubs in stubs, it means that the object you are testing is too tightly coupled to other parts of your application. An object that's hard to test is attempting to warn you that it will be difficult to reuse.

As bad as those consequences are, there's a second problem here that may be worse. Satisfying a requirement by chaining messages together allows you to make code work without figuring out what the objects actually want. In this case, `Foo` is sending `bestFriend.pet().preferredToy().durability()` *for a reason*. That reason hasn't been identified or given a name.

This application depends on a concept, an abstraction, that is implicit in the code. The current author surely understands the intention behind this code, but future maintainers are forced into mind-reading and their psychic powers may be unreliable.

Resolving this LoD violation by naming the missing concept will happen soon, but before moving on, there's one final part to understanding the Law of Demeter. It's time to have a look at the Demeter's formal definition. The Object-Oriented Programming: An Objective Sense of Style whitepaper defines the law as follows:

> *"For all classes C and for all methods M attached to C, all objects to which M sends a message must be instances of classes associated with the following classes:*
>
> 1. *The argument classes of M (including C).*
> 2. *The instance variable classes of C.*
>
> *(Objects created by M, or by functions or methods which M calls, and objects in global variables are considered as arguments of M.)*

The above is a marvel of succinct precision but its very terseness makes it difficult to understand. While it's worth having a passing familiarity with this formal definition, now that you've seen Demeter in action, the definition can be restated in more straightforward language.

The Law of Demeter says that from within a method, messages should be sent only to:

- objects that are passed in as arguments to the method
- objects that are directly available to `self`

Be aware that Demeter is slightly more subtle than that definition suggests. It cares about the APIs of the returned objects, not about each individual object. Therefore, the code

```
'AbCdE'.repeat(2).replace('C', '!').toLowerCase().slice(1) // -> 'b!deabcde'
```

contains many dots but is not a Demeter violation because each intermediate message returns an object that conforms to the same API. It's not the number of dots that matter, but the kind of object returned by each message.

The Law of Demeter effectively restricts the list of *other* objects to which an object may send a message. Its purpose is to reduce the coupling between objects. From the message-senders point of view, an object may talk to its neighbors but not to its neighbor's neighbors. Objects may only send messages to direct collaborators.

8.6.2. Curing Demeter Violations

One obvious way to cure message chains is by introducing message forwarding,[19] a technique often referred to in casual conversation as delegation.[20] The forwarding code for this example might look like the following:

```
class Friend {
  durabilityOfPreferredToyOfPet() {
    return this.pet.durabilityOfPreferredToy();
  }
}

class Pet {
  durabilityOfPreferredToy() {
    return this.preferredToy.durability();
  }
}

class Toy {
  durability() {
    return "1 hour";
  }
}

// Foo now only sends messages to bestFriend
class Foo {
  durabilityOfPreferredToyOfBestFriendsPet() {
    return this.bestFriend.durabilityOfPreferredToyOfPet();
  }
}
```

The code in each method above obeys the Law of Demeter by sending messages only to direct collaborators. The new methods in `Friend` and `Pet` supply the hops that allow `Foo` to navigate to the `duration` method of `Toy` without coupling itself to each intermediate object in that chain.

Message forwarding definitely helps, but you can be forgiven if you're feeling objections along the lines of "Hey, that just added a bunch of new levels of indirection without fundamentally changing anything." While that's true, the coupling between `Foo` and other objects has been loosened and the benefits of this loosening are immediately visible in `Foo`'s tests. Test setup now requires supplying only a `bestFriend` object that can respond to `durabilityOfPreferredToyOfPet`. You no longer have to supply instances of `Toy` and `Pet`, or stub in stubs. Adding these forwarding messages makes `Foo` easier to test, which means it will be easier to reuse.

Here's an updated visualization:

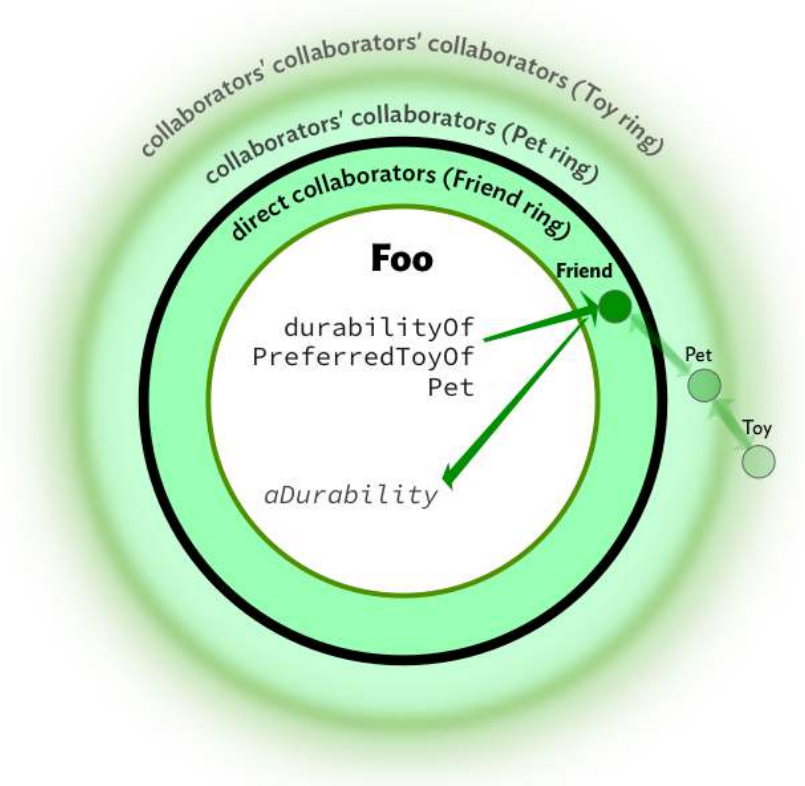

Figure 8.4: LoD Violation Cured With Forwarding

This change certainly improved the code, but `Foo` still contains troubling echos of the application's structural problems. While adding the forwarding *technically* decouples `Foo` from `Pet` and `Toy`, the new `durabilityOfPreferredToyOfPet` message very much implies their continued existence. This message name is merely a concatenation of the object types and messages from the original message chain, and it strongly suggests that `Foo` can only be used in contexts that also contain `Pets` and `Toys`.

Forwarding messages get named like this when attempts to avoid Demeter violations are hijacked by knowledge of an application's existing objects and messages. The trick to honoring the Law while simultaneously avoiding encoding the names of existing objects into the names of the forwarding messages is to *think about design from the message senders point of view*. `Foo` wants to know the durability of the favorite toy of the pet of their best friend for a reason, and the message `Foo` sends to `bestFriend` should be named to reflect `Foo`'s desires.

In this case, `Foo` is trying to figure out the right length of time to schedule for a pet playdate. The pets generally play happily until one demolishes its toy, so the playdate's time-limit should be based on how long the toy can be expected to survive.

If the message `Foo` sends to `bestFriend` is `playdateTimeLimit`, the diagram changes to look like this:

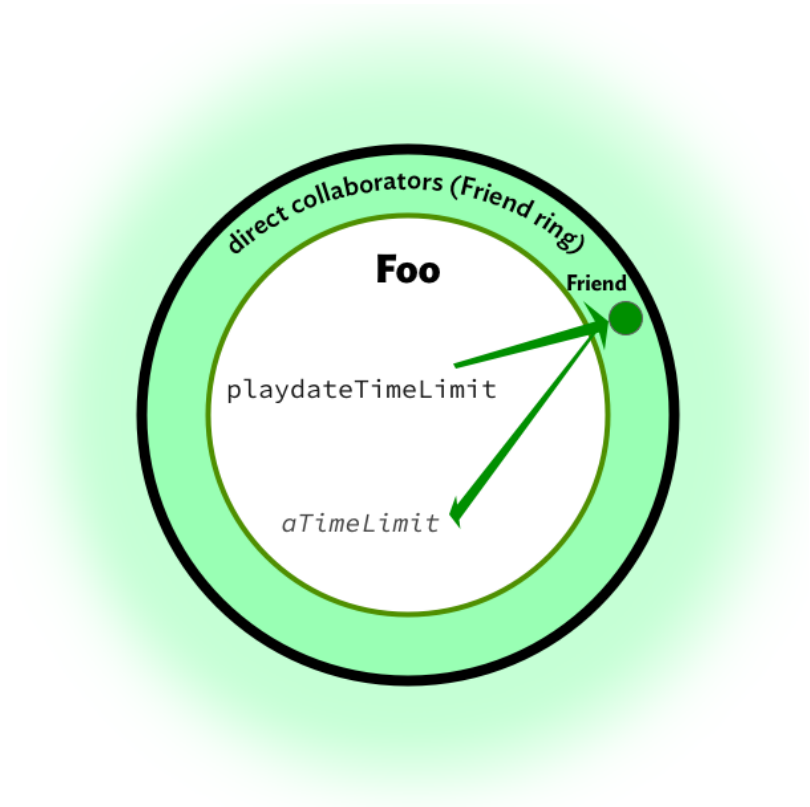

Figure 8.5: LoD Violation Cured With Abstraction

Notice that the above diagram implies nothing about the existence of objects other than `Foo` and `Friend`. As far as `Foo` is concerned, no other objects exist. The `playdateTimeLimit` message reflects `Foo`'s desire to schedule a playdate, but contains no expectation about how the `Friend` satisfies this request.

Once you decide on this new message name, fixing the code is as simple as changing `Foo` to send it and `Friend` to implement it, as shown on lines 13 and 2 below:

```
class Friend {
  playdateTimeLimit() {
    return this.pet.durabilityOfPreferredToy();
  }
}

// Pet and Toy are unchanged

// Foo now asks for what it wants instead of
// making assumptions about its collaborators' collaborators.
class Foo {
  playdateTimeLimit() {
    return this.bestFriend.playdateTimeLimit();
  }
}
```

Now that `Foo` is talking only to a direct collaborator instead poking around in distant objects, it more easily tolerates unexpected change. If a new requirement arises for playdates to sometimes involve children rather than pets, you can satisfy this requirement *without changing* `Foo`, as shown below.

```
class FriendWithPet {
  playdateTimeLimit() {
    return this.pet.durabilityOfPreferredToy();
  }
}

class Pet {
  durabilityOfPreferredToy() {
    return this.preferredToy.durability();
  }
}

class Toy {
  durability() {
    return 3600;
  }
}

class FriendWithChild {
  playdateTimeLimit() {
    return this.child.toleranceForSocialContact();
  }
}

class Child {
  toleranceForSocialContact() {
    return 1800;
  }
}

class Foo {
  playdateTimeLimit() {
    return this.bestFriend.playdateTimeLimit();
  }
}
```

In the above code, there are now two players of the friend role, one with a pet and another with a child. Both polymorphically implement `playdateTimeLimit`. The world outside of `Foo` has increased in complexity; it's been redesigned in unanticipated and perhaps alarming ways, but

`Foo`'s behavior has expanded without `Foo` itself changing at all. These two changes, injecting the `bestFriend` dependency and sending a message named after what `Foo` wants, permit `Foo` to seamlessly collaborate with new objects and tolerate unexpected change.

8.7. Identifying What The Verse Method Wants

Now that you've explored the definition and correction of Law of Demeter violations, it's time to return to the problem in the `Bottles` `verse` method. Here's a reminder of that code, where line 4 contains the problem:

Listing 8.24: Verse Method Contains Many Dependencies Redux

```
1 class Bottles {
2   // ...
3   verse(number) {
4     return new this.verseTemplate(number).lyrics();
5   }
6 }
```

You may resist the idea that line 4 above violates the Law of Demeter because you know that `verseTemplate` contains a class, and you've acquired the habit of thinking of `new` as special. Syntactically it is, but in cases like this it's useful to think of `new` as a message sent to the class. The problem here is that line 4 always needs a `verseTemplate` with a constructor that takes an argument and then creates an object that in turn responds to `lyrics`. From the new-is-just-a-message perspective, this line of code is very much a LoD violation.

The rule for injecting dependencies is that you should inject the thing you want to talk to. In other words, the receiver may directly send messages only to the injected object, not to it and all of its friends. The practical effect of this rule is to prohibit the use of injected objects in message chains that violate the Law of Demeter. If `new` counts as a real message despite its syntactical oddness, then this rule also suggests that you should inject instances, not classes to which you are forced to send `new` and then something else—in this case, `lyrics`.

The rule gets broken here because in this case it is not possible to inject an instance of `BottleVerse`. Have a look at the entire `Bottles` class (shown below) and see if you can explain why.

Listing 8.25: Bottles Generates Verse Numbers

```
1  class Bottles {
2    constructor(verseTemplate = BottleVerse) {
3      this.verseTemplate = verseTemplate;
4    }
5
6    song() {
7      return this.verses(99, 0);
8    }
9
10   verses(starting, ending) {
11     return downTo(starting, ending)
12       .map(i => this.verse(i))
13       .join('\n');
14   }
15
16   verse(number) {
```

```
17 |     return new this.verseTemplate(number).lyrics();
18 |   }
19 |}
```

Notice that when `new` is used to call `verseTemplate` on line 17 above, `number` is passed as an argument. One of `Bottles`' responsibilities is to calculate `number` (in the `map` on line 12). For `Bottles` to be independent of `BottleVerse`, `BottleVerse` has to be injected. Because it's impossible for outsiders to know the value of `number`, they can't create and inject an instance—the best they can do is pass in the class in hopes that some downstream process determines the right `number` and creates the right object.

Peculiarities of this domain have pushed you into a coding corner. You'd like to follow the rule and inject an instance of `BottleVerse`, but you can't because the correct value of `number` isn't known until later. Injecting the class condemns you to a Law of Demeter violation on line 17. You now have to decide whether it's worthwhile to fix this violation. This is where your programming aesthetic comes into play.

The code doesn't look that bad and future maintainers may not find it surprising. Many folks won't even notice the problem, or may feel comfortable ignoring it. What possible harm can come from leaving this violation in the code?

One way to get a quick handle on the consequences of a code arrangement is to attempt to test it. It's long past time for these extracted classes to have their own tests. The original `Bottles` tests are providing safety against regressions but supplying no feedback on the evolving design. Since testing everything is the topic of the next chapter, for now just *imagine* how you might test the `Bottles#verse` method.

Remember that testing is the first form of reuse, so this mental exercise will give you a hint about how easy it will be to reuse `Bottles` elsewhere in your application. The amount of test setup needed will tell you how tightly coupled `Bottles` is to other objects.

To make this exercise correspond to a real-life problem, pretend that `lyrics` is an expensive operation. Your `Bottles` tests should be able to confirm the correctness of `Bottles` without running that distant and costly code, so you'd prefer this test not execute the actual `lyrics` method of `BottleVerse`.

Because of the Demeter violation, you can't just inject a `verseTemplate` that has a simpler `lyrics` implementation. Instead you have to inject a `verseTemplate` whose `new` implementation returns a different object that contains the simpler `lyrics` implementation. Already this is almost too complicated to explain, which doesn't bode well for the tests.

Test setup seems likely to be painful. Pain in testing is a sign of a rigid application and an indication that there's something wrong with the design.

These difficulties are directly related to the Demeter violation. If `Bottles` sent `lyrics` directly to `verseTemplate`, everything would be more straightforward.

It's time to make another wish. If you have a verse template and you want its lyrics for a given verse number, as a died-in-the-wool OO practitioner you have every right to feel entitled to send your `lyrics(number)` request directly to the `verseTemplate` object you have, as shown on line 4 below:

Listing 8.26: Collaborate Directly With Verse Template

```
1 class Bottles {
2   // ...
3   verse(number) {
4     // return this.verseTemplate.lyrics(number);
5     return new this.verseTemplate(number).lyrics();
6   }
7 }
```

If you want something from an object, just ask it. If it doesn't know how to comply, teach it. Don't be trapped by what that other object currently does, but instead, loosen coupling by *designing* a conversation that embodies what the message sender wants.

The wish in line 4 above doesn't yet work, so for now it's commented out. Having conceived it, it's easy to make it come true. Add a `lyrics(number)` message to the `BottleVerse` class, as shown on lines 2-4 below:

Listing 8.27: Add Lyrics Class Method

```
1 class BottleVerse {
2   static lyrics(number) {
3     return new BottleVerse(number).lyrics();
4   }
5   // ...
6 }
```

This static method does two things. First, it uses forwarding to eradicate the extra hop and resolve the Demeter violation. Next, it establishes a new API for players of the verse template role. Objects that play this role must respond to `lyrics(number)`.

Inside of `Bottles`, the `verseTemplate` variable is now thought of as holding a player of the verse template role rather than the `BottleVerse` class. The `BottleVerse` class name is still visible as the default in the initializer, but in Chapter 9 even this will fall away.

While the `BottleVerse.lyrics(number)` static method might seem a lot like the `BottleVerse#lyrics` instance method, don't let the similarity of these names confuse you. This new `lyrics(number)` message embodies the desire of `Bottles` to get lyrics from a verse template and represents the abstraction of what `Bottles` wants. Notice that it's named `lyrics` rather than `new_lyrics`; this name is *not* the combination of the message names in the original Demeter violation. This message is more like `playdateTimeLimit` than `durabilityOfPreferredToyOfPet`.

The instance methods of `BottleVerse` are now private implementation details and have effectively disappeared from sight. Now that the `BottleVerse.lyrics(number)` static method exists, you might be tempted to move the behavior from those private instance methods into this

new static method. This would certainly work, and it would reduce the amount of code. Why bother to create an instance of `BottleVerse` at all?

Despite the fact that earlier in this chapter, classes were treated as "just another object" when deciding if `new` contributed to a Demeter violation, classes *are* different from instances in the most fundamental object-oriented way. In instances, common behavior combines with differing data to create objects that collaborate to form your application.

Putting domain behavior on the class side rather than on the instance side places a bet that this domain concept will never involve data that varies. This bet makes sense only if the value of not typing `n e` and `w` today is greater than the future cost of converting all the static methods to instance methods should you find that data needs to vary.

Since static methods resist refactoring,[21] the cost of moving domain behavior from the class to the instance can be very high, and can far outweigh the paltry savings you get from avoiding typing `new`. Because you cannot know the future, you cannot correctly guess when to follow which strategy. This suggests that the most economical overall plan is to always create instances of objects. Don't waste a minute thinking about whether or not to do this. Part of your programming aesthetic is to reflexively put domain behavior on instances.

Now that the `BottleVerse` class responds to `lyrics(number)` you can return to `Bottles` and reduce the `verse` to the wishful line of code, as shown below:

Listing 8.28: Reduce Verse to the Fulfilled Wish

```
class Bottles {
  // ...
  verse(number) {
    return this.verseTemplate.lyrics(number);
  }
}
```

Having made that change, here's an overview of the resulting `Bottles` and `BottleVerse` classes.

Listing 8.29: BottleVerse Class Responds to Lyrics

```
class Bottles {
  constructor(verseTemplate = BottleVerse) {
    this.verseTemplate = verseTemplate;
  }

  song() {
    return this.verses(99, 0);
  }

  verses(starting, ending) {
    return downTo(starting, ending)
      .map(i => this.verse(i))
      .join('\n');
  }

  verse(number) {
    return this.verseTemplate.lyrics(number);
  }
```

```
19 }
20
21 class BottleVerse {
22   static lyrics(number) {
23     return new BottleVerse(number).lyrics();
24   }
25
26   constructor(number) {
27     this.number = number;
28   }
29
30   lyrics() {
31     const bottleNumber = BottleNumber.for(this.number);
32
33     return (
34       capitalize(`${bottleNumber} of beer on the wall, `) +
35       `${bottleNumber} of beer.\n` +
36       `${bottleNumber.action()}, ` +
37       `${bottleNumber.successor()} of beer on the wall.\n`
38     );
39   }
40 }
```

Not only has the `BottleVerse` class has been extracted and injected, but the resulting Demeter violation has been fixed by adding a forwarding method.

Now that you understand the Law of Demeter, following it should become a part of your programming aesthetic. Be extremely biased towards fixing violations. The overall cost of dealing with each transgression as it occurs is guaranteed to be less than the ultimate cost of repairing the few that spiral out of control after infecting your code for several years. Admittedly, there are a few situations in which violating Demeter makes sense. For example, a report that prints all the relationships in your database should violate Demeter rather than introduce a bunch of new forwarding messages. But these situations are exceptions. In general, don't violate the Law of Demeter, and fix violations as you come upon them.

The above code looks quite reasonable, but studying it should remind you of one final concern. Despite the efforts in this chapter to depend on abstractions rather than concretions, line 31 above still contains a concretion. The `lyrics` method of `BottleVerse` contains a hard-coded reference to `BottleNumber`. The next and final section explores alternatives ways to arrange this code.

8.8. Pushing Object Creation to the Edge

The final unresolved issue is that the `lyrics` instance method in `BottleVerse` directly references the concrete `BottleNumber` class.

Here's a reminder of `BottleVerse`:

Listing 8.30: Lyrics Instance Method Depends on BottleNumber Class

```
1 class BottleVerse {
2   static lyrics(number) {
3     return new BottleVerse(number).lyrics();
4   }
5
6   constructor(number) {
```

```
7 |     this.number = number;
8 |   }
9 |
10|   lyrics() {
11|     const bottleNumber = BottleNumber.for(this.number);
12|
13|     return (
14|       capitalize(`${bottleNumber} of beer on the wall, `) +
15|       `${bottleNumber} of beer.\n` +
16|       `${bottleNumber.action()}, ` +
17|       `${bottleNumber.successor()} of beer on the wall.\n`
18|     );
19|   }
20|}
```

The above code contains a `lyrics(number)` static method (line 2) and a `lyrics` instance method (line 10). The static method is injected with a `number`. This static method creates a new instance of `BottleVerse`, which it initializes with that `number` and then sends `lyrics` to the result.

The code in the `lyrics` instance method (referred to in the following as `#lyrics` for clarity) is concerning in a number of ways.

First, line 11 of `#lyrics` knows about, or depends on, both `BottleNumber` and `for`. It would be better if the method could do its job without having to know these things.

Next, `#lyrics` contains a blank line. This suggests a change of topic, which in turn suggests that the method does more than one thing. The *Blank Line* ™ code smell tells you that `#lyrics` probably violates the Single Responsibility Principle.

Finally, pause a minute and attempt to describe how `#lyrics` uses `number`. Notice that there is only one reference to `number` in the entire method, and that the purpose of this single reference is to convert `number` into something else. This should make the hair on your OO neck stand up. If someone else knows enough to provide `#lyrics` with the right `number`, surely that someone can provide the right `BottleNumber` instead.

If you consider these observations in combination, you'll notice that they overlap. The code offset by the blank line contains the `BottleNumber` concretion, which responds to `for`, and thus turns `number` into a different object. While the three listed concerns might appear to reflect unrelated problems, each is actually a symptom of a single design issue. This is good news because it means that one fix will cure them all.

Well-designed object-oriented applications consist of loosely-coupled objects that rely on polymorphism to vary behavior. Injecting dependencies loosens coupling. Polymorphism isolates variant behavior into sets of interchangeable objects that look the same from the outside but behave differently on the inside.

From the point of view of the object whose collaborators are being injected, this is perfect. Because these injection-receiving objects can interact with new and unexpected collaborators without having to change, they are extremely flexible. Need a new variant? Just create a new class to represent that variant and inject it. The receiver knows not and cares not.

However, this way of thinking about OO introduces a concern that hasn't yet been fully addressed. Where do these injected objects get created? When? And by whom?

Applications that use dependency injection evolve, naturally and of necessity, into systems where object creation begins to separate from object use. Object creation gets pushed more towards the edges, towards the outside, and the objects themselves interact more towards the middle, or the inside.

In this case, the code would be simpler if you converted `number` to `BottleNumber` at an earlier point in the code and just used the result later in `#lyrics`. This change would move object creation back in the stack, more towards the edge of this application.

Taking a step back, consider that the current code arrangement works. You don't have a new requirement. Also, despite being bound to a concretion, the `#lyrics` method is already fairly flexible. Because `BottleNumber.for` is a factory, it's already possible to add or change bottle number role players without breaking the `#lyrics` method. The factory could manufacture new bottle numbers of a different type, and `Bottles#lyrics` could happily collaborate with these new objects without having to change.

Everything in the prior paragraph argues that this code is fine as is and suggests that you should walk away now. But there's one more rule (or perhaps more a guideline) that belongs in your aesthetic about writing object-oriented code. The hard-coded reference to the `BottleNumber` class in the `#lyrics` instance method is troubling. Experienced object-oriented programmers know that applications most easily adapt to the unknown future if they:

- resist giving instance methods knowledge of concrete class names, and
- seek opportunities to move the object creation towards the edges of the application.

These are guidelines, not hard and fast rules, so you can allow yourself some leeway. This is especially true in cases like this where the hard-coded reference is to a factory, so the coupling is already loose. Even so, you should be eternally alert for instance methods that reference class names and perpetually on the lookout for ways to remove those references.

In this case, it's easy to remove the `BottleNumber` reference from `#lyrics` while simultaneously pushing object creation back in the stack. By happy coincidence, fixing the Demeter violation in the prior section created the perfect place to convert `number` into a `BottleNumber`.

Have a look at the new wish on line 3 below:

Listing 8.31: Inject a BottleNumber Via Wishful Thinking

```
1 class BottleVerse {
2   static lyrics(number) {
3     // return new BottleVerse(BottleNumber.for(number)).lyrics();
4     return new BottleVerse(number).lyrics();
5   }
6   // ...
7 }
```

Line 3 above converts `number` to `BottleNumber` in the same method where the `BottleVerse` instance gets created. Notice how this change begins to group object creation and separate it from object use.

This wish changes the type of the object that gets passed to `BottleVerse` instances during initialization. As you may recall from Listing 6.33: Return Argument If Already a Bottle Number, the way to refactor through a type change is to alter the receiving code so that it temporarily tolerates both the old type and the new. Lines 4-6 below do just that by tolerating both `numbers` and `BottleNumbers`.

Listing 8.32: Accept Number or BottleNumber

```
1 class BottleVerse {
2   // ...
3   lyrics() {
4     const bottleNumber = this.number instanceof BottleNumber ?
5       this.number :
6       BottleNumber.for(this.number);
7     // ...
8   }
9 }
```

Now that `#verses` works with either type, you can return to the `BottleVerse.lyrics(number)` static method and uncomment the wishful code, as shown below:

Listing 8.33: Execute the Wishful Code

```
1 class BottleVerse {
2   static lyrics(number) {
3     new BottleVerse(BottleNumber.for(number)).lyrics();
4     return new BottleVerse(number).lyrics();
5   }
6   // ...
7 }
```

Running the tests proves that line 3 above executes without blowing up. Next, delete the old code on line 4, add a return to line 3, and run the tests again. This should work, reducing `BottleVerse` to:

Listing 8.34: Inject a BottleNumber Into BottleVerse at Creation

```
 1 class BottleVerse {
 2   static lyrics(number) {
 3     return new BottleVerse(BottleNumber.for(number)).lyrics();
 4   }
 5
 6   constructor(number) {
 7     this.number = number;
 8   }
 9
10   lyrics() {
11     const bottleNumber = this.number instanceof BottleNumber ?
12       this.number :
13       BottleNumber.for(this.number);
14
15     return (
16       capitalize(`${bottleNumber} of beer on the wall, `) +
```

```
17 |       `${bottleNumber} of beer.\n` +
18 |       `${bottleNumber.action()}, ` +
19 |       `${bottleNumber.successor()} of beer on the wall.\n`
20 |     );
21 |   }
22 |}
```

Now that the new type is being injected, you can remove the temporary type check in `#lyrics`. This reduces line 11 below to the following slightly confusing line of code.

Listing 8.35: Remove Temporary Type Check

```
1 |class BottleVerse {
2 |  static lyrics(number) {
3 |    return new BottleVerse(BottleNumber.for(number)).lyrics();
4 |  }
5 |
6 |  constructor(number) {
7 |    this.number = number;
8 |  }
9 |
10|  lyrics() {
11|    const bottleNumber = this.number;
12|
13|    return (
14|      capitalize(`${bottleNumber} of beer on the wall, `) +
15|      `${bottleNumber} of beer.\n` +
16|      `${bottleNumber.action()}, ` +
17|      `${bottleNumber.successor()} of beer on the wall.\n`
18|    );
19|  }
20|}
```

The above works but contains obsolete remnants of `number`. This code is tantalizingly close to being finished, and at this point it's tempting to jump ahead and make several changes at once. For example, if you were to rename `number` to `bottleNumber` on lines 6-8 and delete lines 11-12, you'd be done.

This last bit is easy if you're willing to change everything at once, but since this example stands in for bigger, real-world problems, it's worth practicing how to fix these names with smaller changes. Before doing so, however, it's finally time to amend the one-line rule.

Instead of being restricted to one-line changes, refactoring permits *one-undo* changes. The broadened one-undo rule allows you to use the find/replace feature of your text editor to make many changes at once. If you make a swath of changes in one editor command and the tests continue to pass, you can go on. If any test fails, you must undo and make a better change.

The following examples show a simple refactoring that renames `number` to `bottleNumber` in `BottleVerse`. You might find it interesting to attempt bigger leaps by changing many lines at once with your editor. Just remember, undo and try again if any change breaks the tests.

For this refactoring, first add a property to store `bottleNumber` and initialize it to the currently injected `number` variable (line 5).

Listing 8.36: Set Bottle Number to Number

```
class BottleVerse {
  // ...
  constructor(number) {
    this.number = number;
    this.bottleNumber = number;
  }
  // ...
}
```

Now that the `bottleNumber` property contains an actual bottle number, refer to that property in `#lyrics`. Next, delete `const bottleNumber = this.number` and the following blank line from `#lyrics`, which leaves:

Listing 8.37: Lyrics No Longer Uses a Local Variable

```
class BottleVerse {
  // ...
  constructor(number) {
    this.number = number;
    this.bottleNumber = number;
  }

  lyrics() {
    return (
      capitalize(`${this.bottleNumber} of beer on the wall, `) +
      `${this.bottleNumber} of beer.\n` +
      `${this.bottleNumber.action()}, ` +
      `${this.bottleNumber.successor()} of beer on the wall.\n`
    );
  }
}
```

Now, in the constructor, delete the `number` property and change `number` to `bottleNumber` (now's a good time for a multi-line one-undo change). This reduces the `BottleVerse` class to:

Listing 8.38: Consolidated Object Creation

```
class BottleVerse {
  static lyrics(number) {
    return new BottleVerse(BottleNumber.for(number)).lyrics();
  }

  constructor(bottleNumber) {
    this.bottleNumber = bottleNumber;
  }

  lyrics() {
    return (
      capitalize(`${this.bottleNumber} of beer on the wall, `) +
      `${this.bottleNumber} of beer.\n` +
      `${this.bottleNumber.action()}, ` +
      `${this.bottleNumber.successor()} of beer on the wall.\n`
    );
  }
}
```

That's the end of this refactoring. Before moving on to the chapter summary, ponder one last issue. It could be argued that `BottleVerse.lyrics(number)` is doing more than one thing. Not only does it create two new objects, a player of the bottle number role and an instance of the `BottleVerse` class, but it also sends `#lyrics` to one of those creations.

If you wanted to rigorously separate object creation from object use, you could refactor this code into the following:

```
class BottleVerse {
  static for(number) {
    return new BottleVerse(BottleNumber.for(number));
  }

  static lyrics(number) {
    return BottleVerse.for(number).lyrics();
  }
  // ...
}
```

The above code, however, may be more abstract and indirect than even your newly-expanded programming aesthetic requires. This separation can be deferred until someone asks for a change that requires direct access to a `BottleVerse` object, rather than simply requesting that object's lyrics.

8.9. Summary

This chapter pulled the lyrics of the "99 Bottles" song out of `Bottles` and put them into a new `BottleVerse` class. It then injected an instance of `BottleVerse` back into `Bottles`. Extracting `BottleVerse` reduced the `Bottles`'s responsibilities, making it easier to understand and maintain. Injecting `BottleVerse` into `Bottles` loosened the coupling between `Bottles` and the outside world. `Bottles` now thinks of itself as being injected with players of the verse template role and will happily collaborate with any newly arriving object as long as that object responds to `lyrics(number)`.

The impetus behind extracting `BottleVerse` was a new requirement to produce songs with different lyrics, that is, to vary the verse. The refactorings in this chapter satisfied that requirement by following a fundamental strategy of object-oriented design: extracting the `BottleVerse` class isolated the behavior that the new requirement needed to vary.

While continuing to lean on code smells and refactoring recipes, this chapter introduced the idea of a programming aesthetic. A programming aesthetic is the set of internal heuristics that guide your behavior in times of uncertainty. Vague feelings about the rightness of code become part of your aesthetic once you can eloquently and convincingly use actual words to explain your concerns and proposed improvements. A good programming aesthetic focuses attention on improvements that are likely to prove worthwhile.

This chapter suggested five precepts that belong in everyone's object-oriented programming aesthetic:

- Put domain behavior on instances.
- Be averse to allowing instance methods to know the names of constants.
- Seek to depend on injected abstractions rather than hard-coded concretions.

- Push object creation to the edges, expecting objects to be created in one place and used in another.
- Avoid Demeter violations, using the temptation to create them as a spur to search for deeper abstractions.

The practical effect of following these precepts is to loosen the coupling between objects. The code to which you would apply them generally already works so adherence might seem optional, and is certainly not free. Complying with these precepts will frequently increase the amount of code and add levels of indirection, at least in the short term. However, these added costs are overwhelmingly offset by the eventual savings accrued as a result of decoupling.

Any application that survives will change. The only thing of which you can be more confident is that you cannot predict where this change will occur. The certainty of change coupled with the uncertainty of that change's location means that your best programming strategy is to strive to loosen the coupling of all code everywhere from the moment of initial creation.

Therefore, these precepts don't attempt to guess the future; rather, they leverage against it. Instead of writing code that speculatively imagines a later need for one specific feature, they tell you to loosen the coupling of all code so that you can easily adapt to whatever future arrives.

Uncertainty about the future is not a license to guess; it's a directive to decouple. Your future will be brighter if you develop a programming aesthetic that drives you to do so.

9. Reaping the Benefits of Design

The current "99 Bottles" application satisfies all of the known requirements. The code looks very object-oriented, and you can articulate a persuasive justification for every decision that led to its current shape.

But the tests are a bit musty. They've gradually transformed from tests that tell a helpful story about the "99 Bottles" song into tests that misrepresent, if not outright lie, about the entire domain. Their only redeeming quality is that at least they fail if the code gets broken.

It's important to know if something breaks, but tests can do far more. They give you the opportunity to explain the domain to future readers. They expose design problems that make code hard to reuse. Well-designed code is easy to test; when testing is hard, the code's design needs attention.

Writing tests, even at this late date, will improve the code. As a final exercise, this chapter does just that.

9.1. Choosing Which Units to Test

Every class should have its own unit test, unless doing otherwise saves money. The allowed-to-skip-tests bar is high, but some code meets it. This first section explores such a case.

9.1.1. Contrasting Unit and Integration Tests

Back in Chapter 2, unit tests drove the Shameless Green implementation of "99 Bottles." Those initial tests continue to provide a safety net, and have assured the code's correctness through many refactorings.

Comforting as the tests are, over time they have gotten woefully out of date. In early chapters, `Bottles` was the only class. After `BottleNumber` was extracted, the unit tests for `Bottles` morphed into integration tests that covered both classes. This integration coverage expanded further as more classes were extracted. The original unit tests now cover `Bottles`, `BottleVerse`, and the entire `BottleNumber` hierarchy. They remain useful because they'll break if someone introduces an error, but they mislead readers about the intent and workings of the present code.

Improving the tests will accomplish two things. First, this will reduce costs. Tests that tell the right story make it easier for future readers to understand the code. Improving the story will save you money forever.

Next, updating the tests will lay bare the consequences of the code's design. It should be easy to create simple, intention-revealing tests. When it's not, the chief problem is often too much coupling. In such cases the solution is *not* to write complicated tests that overcome tight coupling, but rather to loosen the coupling so that you can write simple tests. The most cost-effective time to intervene in tightly coupled code is right now, before new requirements cause you to to start reusing these objects.

The good news is that writing tests will uncover every bit of overlooked tight coupling and immediately reward you for fixing it. Here's a reminder of the current tests:

Listing 9.1: Bottles Tests Reminder

```
describe('Bottles', () => {
  test('the first verse', () => {
    const expected =
      '99 bottles of beer on the wall, ' +
      '99 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '98 bottles of beer on the wall.\n';
    expect(new Bottles().verse(99)).toBe(expected);
  });

  test('another verse', () => {
    const expected =
      '3 bottles of beer on the wall, ' +
      '3 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '2 bottles of beer on the wall.\n';
    expect(new Bottles().verse(3)).toBe(expected);
  });

  test('verse 2', () => {
    const expected =
      '2 bottles of beer on the wall, ' +
      '2 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '1 bottle of beer on the wall.\n';
    expect(new Bottles().verse(2)).toBe(expected);
  });

  test('verse 1', () => {
    const expected =
      '1 bottle of beer on the wall, ' +
      '1 bottle of beer.\n' +
      'Take it down and pass it around, ' +
      'no more bottles of beer on the wall.\n';
    expect(new Bottles().verse(1)).toBe(expected);
  });

  test('verse 0', () => {
    const expected =
      'No more bottles of beer on the wall, ' +
      'no more bottles of beer.\n' +
      'Go to the store and buy some more, ' +
      '99 bottles of beer on the wall.\n';
    expect(new Bottles().verse(0)).toBe(expected);
  });

  test('a couple verses', () => {
    // ...
    expect(new Bottles().verses(99, 98)).toBe(expected);
  });

  test('a few verses', () => {
    // ...
    expect(new Bottles().verses(2, 0)).toBe(expected);
  });

  test('the whole song', () => {
    // ...
    expect(new Bottles().song()).toBe(expected);
```

```
60 |   });
61 |});
```

These tests made perfect sense when `Bottles` was the only class. As a first stab in an unknown domain they stand up pretty well, but now that you better understand the 99 Bottles problem you might be itching to do a few things differently.

It's worth taking a minute to remember why there's such a dearth of tests. The original `BottleNumber` class was extracted from and then used in `Bottles` by way of the *Extract Class* refactoring recipe. This recipe is known to work. As long as you make one-undo changes and confirm that the tests pass after every change, you can follow the recipe with confidence in the result.

Creating a new class by following a recipe instead of by doing TDD is perfectly allowable, but the result relies on the original tests to confirm that the extracted class remains error-free. Using the recipe consigns `BottleNumber`'s correctness to `Bottles`' tests. If `BottleNumber` is wrong, a `Bottles` test breaks.

Instead of updating the tests immediately after extracting `BottleNumber`, or after further extracting the `BottleNumber` subclasses, the sins against TDD were compounded by extracting yet another class, `BottleVerse`. `Bottles`' tests provided an increasingly broad safety net over an extended series of other objects.

`Bottles`' tests started out as unit tests but have become integration tests. Unit tests are meant to test the public API of a single class. The unit under test often requires a few collaborating objects in order to run, but these other objects are tangental and exist only so you can address the unit of interest. Unit tests ought not test collaborators.

Integration tests are intended to prove that groups of objects collaborate correctly; they show that an entire chain of behavior works. This is exactly what these `Bottles` tests do.

Your general approach to testing should be to create a unit test for every class, and to test every method in that class's public API. To do so here, you have to choose where to begin. In hopes that fixing the easy problems will clarify the hard ones, begin with the extremely simple `BottleNumber` subclasses. Not only are these classes small, but they are also leaf nodes on your object dependency graph[22] which suggests that they are minimally entangled with your overall domain. `BottleNumber1` is a reasonable starting point.

Despite the fact that `BottleNumber1` is tiny, testing it presents a conundrum. For example, here's its `pronoun` method:

```
pronoun() {
  return 'it';
}
```

The only reasonable test would look like:

```
test('pronoun', () => {
  expect(new BottleNumber1(1).pronoun()).toBe('it');
```

```
});
```

This is woefully circular. The test assertion exactly mirrors the method implementation. When the assertion duplicates the code, they both must change in lockstep or the test will fail.

Having to change 'it' in two places is a small thing, but this still feels like a waste of time and money. This test doesn't add much value, and imagining it should spur you to consider alternatives. Contrary to your original intentions, perhaps `pronoun` (and by extension `BottleNumber1` as a whole) shouldn't be tested at all.

Tests are most valuable when they exercise confirmable behavior, and you could argue that `BottleNumber1#pronoun` has none. While this code should very definitely be exercised during testing, the most cost-effective place to do so may be within some other unit's test.

Some other test is already covering this code; it gets executed during the `Bottles` tests. You can confirm this by arbitrarily changing `pronoun` and running the tests. For example, altering the code to return `oops` instead of `it`...

```
class BottleNumber1 extends BottleNumber {
  // ...
  pronoun() {
    return 'oops';
  }
  // ...
}
```

...causes these three tests to fail:

```
• Bottles › verse 1

  // ...

• Bottles › a few verses

  // ...

• Bottles › the whole song

  // ...
```

While it's comforting that tests fail if the code gets broken, the `Bottles` tests seem very far away from bottle number-ish concerns. Although you might be casting around for justification to omit unit tests for `BottleNumber1`, this code has to be executed as part of *some* test, and counting on the `Bottles` tests for coverage feels like a stretch. It's time to develop some principles to handle this situation.

9.1.2. Foregoing Tests

As previously mentioned, you should approach testing with the intent of creating unit tests for every class. After long and careful consideration you might decide to allow one object's unit tests to cover a collaborator as well, but this is the exception. You should plan to write a unit test for every class, and you are entitled to expect to see a unit test for every class in someone else's code.

The `Bottles` tests provide complete coverage for all the existing classes, but they're really integration tests masquerading as unit tests. Faux unit tests that reach out to cover distant collaborators become increasingly unhelpful to the next programmer. When a test involves many objects in combination, the code could break quite far from the origin of the problem. This makes it hard to determine the cause of an error.

Integration tests are important, but they serve a different purpose than unit tests. Integration tests are great at proving the correctness of the collaboration between groups of objects. They demonstrate the overall operation of all or a subset of your application. Because they cover a lot of ground, they're often slow.

In contrast, unit tests are for you, the programmer. They help you write down, communicate the expected behavior of, prevent regression in, and debug smaller units of code. When something goes wrong, it's the unit tests that provide an error message near the offending line of code. Since they narrow the set of potential code culprits behind any problem, they make debugging easier. They should stay out of your way when writing code, which means they should be fast.

It cannot be emphasized strongly enough that most classes deserve their own explicit unit tests. This should be your default point of view. But just as every rule is meant to be broken, occasionally the most useful thing to communicate about an object is that it is so small, simple, or invisible that testing it individually would raise costs rather than lower them. Every now and then it makes sense to test an object in conjunction with its enclosing unit. Doing so creates a test that leaks into the space between integration and unit, but if you think of the smaller objects as being private, you can be justified in calling the whole thing a unit test.

The classes in the `BottleNumber` hierarchy are examples of things so small, simple, and invisible that they might not deserve their own unit tests.

If you subscribe to the principle that applications should have 100% test coverage, you might need to reexamine your definition of that rule. Perhaps it means "100% of the code should be exercised during unit tests," rather than "100% of the public methods should have their own personal tests."

The tests previously imagined for `BottleNumber1#pronoun` are so tightly bound to implementation details that they may very well interfere with change. Tests should give you the freedom to improve code, not glue you to its current implementation. When they constrain rather than liberate, ask if they're worthwhile, and consider omitting them.

In the rare case where you decide to forego giving a class its own unit test, you must be able to defend this decision with a clearly articulated justification. In addition to size and complexity, visibility is also an important consideration. Visibility is determined by the *context* in which the class is known.

To illustrate, here's a reminder of the `BottleVerse` class, which references `BottleNumber` on line 3:

Listing 9.2: BottleVerse Reminder

```
1 class BottleVerse {
2   static lyrics(number) {
3     return new BottleVerse(BottleNumber.for(number)).lyrics();
4   }
5
6   constructor(bottleNumber) {
7     this.bottleNumber = bottleNumber;
8   }
9
10  lyrics() {
11    return (
12      capitalize(`${this.bottleNumber} of beer on the wall, `) +
13      `${this.bottleNumber} of beer.\n` +
14      `${this.bottleNumber.action()}, ` +
15      `${this.bottleNumber.successor()} of beer on the wall.\n`
16    );
17  }
18 }
```

The entire public API of `BottleVerse` consists of the `lyrics` static method on line 2, so all of `BottleVerse`'s instance methods can be thought of as private. Within this one public method, `BottleVerse` invokes the `BottleNumber` factory to get a player of the bottle number role.

Notice how profoundly `BottleVerse` depends on bottle numbers. Instances of `BottleVerse` are entirely reliant on a bottle number to produce a verse. It's hard to imagine any use of `BottleVerse` that doesn't require a concomitant bottle number. From `BottleVerse`'s point of view, it is inseparable from its bottle numbers. Bottle numbers exist in the context of `BottleVerse`.

Now flip your point of view and consider the relationship between these classes from the perspective of a bottle number. Things change dramatically. Individual bottle numbers aren't aware of `BottleVerse`. They don't know that they're part of a song. They are perfectly willing to be used in any situation. They are *independent* of context, which means that they could easily be reused in a new one.

Everything in the prior paragraph argues that bottle numbers are their own self-contained thing, which in turn argues that they should have their own tests. And yet. They are small, painfully simple, used nowhere other than in `BottleVerse`, and could be completely exercised by the soon-to-be-written `BottleVerse` tests.

The tipping point for deciding how to test is visibility. `BottleVerse` has assumed personal responsibility for supplying itself with `BottleNumbers`. `BottleNumbers` do not get created and injected from the outside, but instead, `BottleVerse` knows about them inside itself. The dependency between `BottleVerse` and `BottleNumber` is not visible to outside observers.

At this point, no other place in the application references the `BottleNumber` factory or any of the manufactured classes. It's certainly possible that someone might eventually do so, but no one does right now.

This wrapping of `BottleNumber` by `BottleVerse` means that if you stand in the space between the public objects of this app, you'll never see a reference to `BottleNumber`. It's so invisible to outside observers that it may as well not exist.

In summary, `BottleNumbers` are:

- small
- simple
- invisible from outside of `BottleVerse`
- used in no context other than `BottleVerse`

These factors combine to suggest that, at least for the moment, you should think about bottle numbers as an integral part of `BottleVerse`, and test them within `BottleVerse`'s unit test.

Most situations are more convoluted, which means that objects should generally have their own unit tests. Even this situation might someday change such that `BottleNumbers` should be unit tested independently. If these classes get more complicated, or begin to be used in other contexts, revisit this decision. When `BottleNumbers` need their own story, they should have their own tests.

One last point before moving on. The most critical and complicated part of bottle numbers is the factory, which you might want to unit test explicitly even if you defer to `BottleVerse` for testing the other bottle number behavior. When testing the factory it's important to test the factory's responsibilities, not the responsibilities of the objects the factory manufactures. The factory's job is to take a number and turn that number into an instance of the correct class. Thus the test might look like this:

Listing 9.3: Testing the Bottle Number Factory

```
describe('BottleNumber', () => {
  test('returns correct class for given number', () => {
    // 0,1,6 are special
    expect(BottleNumber.for(0).constructor).toBe(BottleNumber0)
    expect(BottleNumber.for(1).constructor).toBe(BottleNumber1)
    expect(BottleNumber.for(6).constructor).toBe(BottleNumber6)

    // Other numbers get the default
    expect(BottleNumber.for(3).constructor).toBe(BottleNumber)
    expect(BottleNumber.for(7).constructor).toBe(BottleNumber)
    expect(BottleNumber.for(43).constructor).toBe(BottleNumber)
  });
});
```

The above test puts many assertions in a single test, which violates a commonly adopted rule. However, as this is the most parsimonious expression of the factory's responsibilities, breaking that general rule is justified.

This section laid out some considerations for choosing to omit unit tests. It doesn't give license to write a bunch of monster integration-y tests and pass them off as unit tests, but instead acknowledges that some things are so interrelated that testing them independently will increase rather than reduce costs. The reason you're writing tests is *to save money*, and every potential test must be evaluated against that criteria.

Use good judgement. Be extremely biased towards creating a unit test for every method in every class's public API. But make sure that all tests justify their existence and eliminate those that don't.

While all *code* needs to be tested, some *tests* aren't worth writing.

9.2. Reorganizing Tests

Unit tests ought to tell an illuminating story. They should demonstrate and confirm the class's direct responsibilities, and do nothing else. You should strive to write the fastest tests possible, in the fewest number necessary, using the most intention-revealing expectations, and the least amount of code.

This next section examines the existing tests with an eye towards determining who is really responsible for the behavior, and thus who should own the tests.

With that, it's time to take up `BottleVerse`.

9.2.1. Gathering BottleVerse Tests

`BottleVerse` is both more complicated and more visible than the simple bottle number classes. Here's a reminder of its code:

Listing 9.4: BottleVerse Implements an Algorithm

```
class BottleVerse {
  static lyrics(number) {
    return new BottleVerse(BottleNumber.for(number)).lyrics();
  }

  constructor(bottleNumber) {
    this.bottleNumber = bottleNumber;
  }

  lyrics() {
    return (
      capitalize(`${this.bottleNumber} of beer on the wall, `) +
      `${this.bottleNumber} of beer.\n` +
      `${this.bottleNumber.action()}, ` +
      `${this.bottleNumber.successor()} of beer on the wall.\n`
    );
  }
}
```

There's a fair amount going on here. Relative to the bottle number classes, `BottleVerse` is definitely more involved. It's not quite as easy to tell what's happening, and it seems possible that a hasty change might break something. Tests would add safety and help tell the story of this code.

`BottleVerse` gets used when some outside entity chooses it for the verse template and injects it into `Bottles`, as shown on line 2 below:

Listing 9.5: BottleVerse Is Loosely Coupled to Bottles

```
1 class Bottles {
2   constructor(verseTemplate = BottleVerse) {
3     this.verseTemplate = verseTemplate;
4   }
5   // ...
6   verse(number) {
7     return this.verseTemplate.lyrics(number);
8   }
9 }
```

In the previous section, the connection between `BottleNumber` and `BottleVerse` was invisible from the outside. In the above code, the relationship between `Bottles` and its verse template is completely exposed. `Bottles` doesn't supply its own template; someone else picks the right object and passes it in.

This collaboration is public. Injecting the verse template dependency loosens the coupling between `Bottles` and `BottleVerse`, which drives these concrete classes apart. The two classes are independent.

Don't be confused by the fact that `verseTemplate` defaults to `BottleVerse` on line 2 above; this isn't a tight coupling between the classes. That default is an artifact of an earlier refactoring and is temporarily necessary because existing tests construct new `Bottles` without passing an argument. A later section of this chapter will update the `Bottles` tests and tend to this default.

This combination of code complexity and public visibility means that `BottleVerse` should get its own unit tests.

As always, start by creating a class to hold the tests:

Listing 9.6: Create Test for BottleVerse

```
1 describe('BottleVerse', () => {
2 });
```

Now you have to decide what to test. Since a unit test should contain at least one test for every method in the class's public API, you'll need to test the `BottleVerse` `lyrics` static method. It's fine to create multiple tests for a single method if doing so will better explain its behavior to future readers.

This `lyrics` method is responsible for taking a number and returning the lyrics for the associated verse. Keeping that in mind, peruse the existing `Bottles` tests:

Listing 9.7: Existing Bottles Tests

```
1  describe('Bottles', () => {
2    test('the first verse', () => {
3      const expected =
4        '99 bottles of beer on the wall, ' +
5        '99 bottles of beer.\n' +
6        'Take one down and pass it around, ' +
7        '98 bottles of beer on the wall.\n';
8      expect(new Bottles().verse(99)).toBe(expected);
9    });
10
11   test('another verse', () => {
```

```
12 |     // ...
13 |     expect(new Bottles().verse(3)).toBe(expected);
14 |   });
15 |
16 |   test('verse 2', () => {
17 |     // ...
18 |     expect(new Bottles().verse(2)).toBe(expected);
19 |   });
20 |
21 |   test('verse 1', () => {
22 |     // ...
23 |     expect(new Bottles().verse(1)).toBe(expected);
24 |   });
25 |
26 |   test('verse 0', () => {
27 |     // ...
28 |     expect(new Bottles().verse(0)).toBe(expected);
29 |   });
30 |
31 |   test('a couple verses', () => {
32 |     // ...
33 |     expect(new Bottles().verses(99, 98)).toBe(expected);
34 |   });
35 |
36 |   test('a few verses', () => {
37 |     // ...
38 |     expect(new Bottles().verses(2, 0)).toBe(expected);
39 |   });
40 |
41 |   test('the whole song', () => {
42 |     // ...
43 |     expect(new Bottles().song()).toBe(expected);
44 |   });
45 | });
```

The `'Bottles' block` already contains a number of tests for individual "99 Bottles" verses. For example, the test on line 2 above confirms that an input of `99` results in the lyrics for that verse. Similar single verse tests occur four more times above (notice the assertions on lines 13, 18, 23 and 28).

These tests were created during the initial TDD of `Bottles` back in Chapter 2, at which time they made perfect sense. Now that `BottleVerse` is responsible for creating individual verses for the "99 Bottles" song, `the 'BottleVerse' block` should take responsibility for proving that the lyrics are correct. This means that every test in `Bottles` that makes assertions about individual "99 Bottles" verses should move into `the 'BottleVerse' block`.

The best way to get started is to carefully move just one test. Since `'the first verse'` already exists, moving it will be accomplished with a refactoring. The test suite is currently passing and you don't want to accidentally introduce an error. The safest way to move code is by way of a now-familiar technique: copy the code, get it working in the new place, and then delete it from the old.

With that in mind, copy `'the first verse'` from `the 'Bottles' block` to `the 'BottleVerse' block`, as shown here:

Listing 9.8: Copy the First Test

```
describe('Bottles', () => {
  test('the first verse', () => {
    const expected =
      '99 bottles of beer on the wall, ' +
      '99 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '98 bottles of beer on the wall.\n';
    expect(new Bottles().verse(99)).toBe(expected);
  });
  // ...
});

describe('BottleVerse', () => {
  test('the first verse', () => {
    const expected =
      '99 bottles of beer on the wall, ' +
      '99 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '98 bottles of beer on the wall.\n';
    expect(new Bottles().verse(99)).toBe(expected);
  });
});
```

Change nothing about the copied test. Run the tests. Nine should pass.

Next, alter the new assertion to test `BottleVerse.lyrics` rather than `Bottles#verse`, as shown on line 8 below:

Listing 9.9: Update the Assertion

```
describe('BottleVerse', () => {
  test('the first verse', () => {
    const expected =
      '99 bottles of beer on the wall, ' +
      '99 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '98 bottles of beer on the wall.\n';
    expect(BottleVerse.lyrics(99)).toBe(expected);
  });
});
```

Running the tests will show that nine still pass.

Now that the new test in `the 'BottleVerse' block` has been shown to work, you can delete the old version from `the 'Bottles' block`. This leaves eight passing tests.

Having verified this process, go ahead and copy all the other individual verse tests from `the 'Bottles' block` to `the 'BottleVerse' block`, again without altering the copied code. After completing the copying, `the 'BottleVerse' block` will look like this:

Listing 9.10: Copy Remaining Tests for Individual Verses

```
describe('BottleVerse', () => {
  test('the first verse', () => {
    // ...
    expect(BottleVerse.lyrics(99)).toBe(expected);
  });

  test('another verse', () => {
    // ...
```

```
9 |    expect(new Bottles().verse(3)).toBe(expected);
10|  });
11|
12|  test('verse 2', () => {
13|    // ...
14|    expect(new Bottles().verse(2)).toBe(expected);
15|  });
16|
17|  test('verse 1', () => {
18|    // ...
19|    expect(new Bottles().verse(1)).toBe(expected);
20|  });
21|
22|  test('verse 0', () => {
23|    // ...
24|    expect(new Bottles().verse(0)).toBe(expected);
25|  });
26|});
```

At this point you will have 12 passing tests. After ensuring that you do, update the assertions in `the 'BottleVerse' block` to invoke `BottleVerse.lyrics` as shown on lines 9, 14, 19, and 24 below, perhaps using this as an opportunity to practice search/replace using regular expressions:

Listing 9.11: Update Test Assertions

```
1 |describe('BottleVerse', () => {
2 |  test('the first verse', () => {
3 |    // ...
4 |    expect(BottleVerse.lyrics(99)).toBe(expected);
5 |  });
6 |
7 |  test('another verse', () => {
8 |    // ...
9 |    expect(BottleVerse.lyrics(3)).toBe(expected);
10|  });
11|
12|  test('verse 2', () => {
13|    // ...
14|    expect(BottleVerse.lyrics(2)).toBe(expected);
15|  });
16|
17|  test('verse 1', () => {
18|    // ...
19|    expect(BottleVerse.lyrics(1)).toBe(expected);
20|  });
21|
22|  test('verse 0', () => {
23|    // ...
24|    expect(BottleVerse.lyrics(0)).toBe(expected);
25|  });
26|});
```

After the above update, 12 tests should still pass. Once you confirm that, you can delete the four obsolete tests from `the 'Bottles' block` with confidence that you haven't broken anything. This deletion returns you to eight total passing tests, and reduces `the 'Bottles' block` to just three tests.

`BottleVerse`'s tests now prove that it correctly produces each different flavor of "99 Bottles" verse. The lyrics tests are correct and complete, and are basically just an updated copy of the tests as they were initially created in Chapter 2. At that time these tests seemed good enough, but

now that you're addressing the test suite as a whole, you might be disappointed by the story these tests tell. They miss a golden opportunity to convey a deeper understanding of the domain to future readers.

Some of the `BottleVerse` tests need better names.

9.2.2. Revealing Intent

Writing the very first test for a new domain can be paralyzingly difficult. It's often much easier to write the tenth test, or the fifth, or even the second. The first test is important because without it you can't get started, but sometimes its ultimate purpose is to teach you that it's not quite right.

During the initial round of TDD back in Chapter 2, you had to make several moderately arbitrary decisions in order to write the first test. At that point you didn't thoroughly understand the domain of the song—this knowledge, after all, comes as a result of conceiving of and writing tests. The problem is circular. You can't write tests until you understand the problem, but you can't understand the problem without writing some tests. No wonder it's hard to get started.

Even so, here is where writing tests *first* really shines. It's far better to struggle with a test that you don't understand than to write code that you don't understand. Tests force you to clarify your intentions because they make explicit assertions. Code has no such pressure, and can be left a confusing mess forever.

So despite being a bit vague on the details, when given the task of writing code to produce 1) the entire "99 Bottles" song, 2) a range of verses, or 3) a single verse, you intrepidly decided to create a `Bottles` class with a public API of `song`, `verses` and `verse`. You then chose to follow the classic TDD inside-out style and begin testing at the `verse` method. At this point you wanted to write a test which took a number and returned the lyrics for the associated verse. In order to do that, you had to decide which number to use and what to name the test.

Names are important, and you could have invested time finding the perfect name for that first test. In this case, ignorance comes in handy. Since you knew it would be hard to pick a great name until you understood the domain better, and you also knew that writing tests would improve your understanding of the domain, you didn't spend a lot of time on this first name. Calling it `'the first verse'` allowed you to move on, and moving on gave you hope of someday knowing enough to do better.

As far as choosing the first number to test, `99` made sense because it appeared first in the song. When future programmers read your tests they will infer meaning from every decision. Starting with `99` is unsurprising. Any other number would cause them to wonder about the deep meaning behind your choice.

After getting that first test to pass, you realized that there were other variants of `verse` that should be tested. You noticed that the lyrics for verses in the range between 99 and 3 are identical except for the value of the number. The test for the top end of this range already existed, so you decided that next you should test the bottom, using `3` as input. You named this test `'another verse'`.

The `'the first verse'`/`'another verse'` tests exist to show that the lyrics of verses from `99` through `3` follow a similar rule, but nothing about their names gives any hint of this rule's existence. The bodies of the tests are fine—the problem is in their names. These vague names were perfectly acceptable when they were the best you could do, but now you know more and can do better. It's time to improve the story.

The names of these two tests should convey the following:

- this is a verse test
- a rule exists
- it applies to most verses
- it involves a range
- one test is for the upper bound
- the other test is for the lower bound

Names like `'verse general rule upper bound'` and `'verse general rule lower bound'` perfectly satisfy these constraints.

Now that you've done the hard part and unearthed better names, rename the first two `BottleVerse` tests as shown on lines 2 and 11 below:

Listing 9.12: Rename Verse Tests for 99 and 3

```
1 describe('BottleVerse', () => {
2   test('verse general rule upper bound', () => {
3     const expected =
4       '99 bottles of beer on the wall, ' +
5       '99 bottles of beer.\n' +
6       'Take one down and pass it around, ' +
7       '98 bottles of beer on the wall.\n';
8     expect(BottleVerse.lyrics(99)).toBe(expected);
9   });
10
11  test('verse general rule lower bound', () => {
12    const expected =
13      '3 bottles of beer on the wall, ' +
14      '3 bottles of beer.\n' +
15      'Take one down and pass it around, ' +
16      '2 bottles of beer on the wall.\n';
17    expect(BottleVerse.lyrics(3)).toBe(expected);
18  });
19  // ...
20 });
```

Here's an overall look at the resulting `'BottleVerse'` `block`:

Listing 9.13: Interim BottleVerse Tests

```
1 describe('BottleVerse', () => {
2   test('verse general rule upper bound', () => {
3     // ...
4     expect(BottleVerse.lyrics(99)).toBe(expected);
5   });
6
```

```
  test('verse general rule lower bound', () => {
    // ...
    expect(BottleVerse.lyrics(3)).toBe(expected);
  });

  test('verse 2', () => {
    // ...
    expect(BottleVerse.lyrics(2)).toBe(expected);
  });

  test('verse 1', () => {
    // ...
    expect(BottleVerse.lyrics(1)).toBe(expected);
  });

  test('verse 0', () => {
    // ...
    expect(BottleVerse.lyrics(0)).toBe(expected);
  });
});
```

The above names are consistent and symmetrical. They please the eye and convey more information about the song. But they're still not good enough.

The names clearly indicate that some verses follow a common rule and that others do not. While completely true, these suggestions continue to mislead the reader. The names strongly imply that every special verse has its own personal test. Now that you have fulfilled the six-pack requirement, verse 6 and verse 7 are also unique. Telling the best story requires that you give them their own tests.

It doesn't matter that verses 6 and 7 are also tested in the `song` test. The other special verses (2, 1, 0) are already being tested there too. Appearing in the song doesn't make these special verses less special. The point behind these tests is tell a story to future readers, and that story should be consistent and complete.

The expectations needed to test verse 7 and verse 6 can be plucked directly from the `song` test. Use these expectations to create two new verse tests whose names follow the current pattern, as shown below:

Listing 9.14: Test Special Verses 7 and 6

```
describe('BottleVerse', () => {
  test('verse general rule upper bound', () => {
    // ...
  });

  test('verse general rule lower bound', () => {
    // ...
  });

  test('verse 7', () => {
    const expected =
      '7 bottles of beer on the wall, ' +
      '7 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '1 six-pack of beer on the wall.\n';
    expect(BottleVerse.lyrics(7)).toBe(expected);
  });
```

```
18 |
19 |   test('verse 6', () => {
20 |     const expected =
21 |       '1 six-pack of beer on the wall, ' +
22 |       '1 six-pack of beer.\n' +
23 |       'Take one down and pass it around, ' +
24 |       '5 bottles of beer on the wall.\n';
25 |     expect(BottleVerse.lyrics(6)).toBe(expected);
26 |   });
27 |
28 |   test('verse 2', () => {
29 |     // ...
30 |   });
31 |
32 |   test('verse 1', () => {
33 |     // ...
34 |   });
35 |
36 |   test('verse 0', () => {
37 |     // ...
38 |   });
39 | });
```

At this point there should be 10 passing tests, half of which handle the special verses 7, 6, 2, 1 and 0.

Adding tests for verses 7 and 6 made the special cases consistent, but muddied the once crystal clear water of the general verse rule. There are now two special cases embedded within the range covered by that rule. If you worry that readers will be confused by this, spend a moment thinking about how you might rename these tests to mitigate that confusion.

The current names, however, are probably good enough. They've been significantly improved, and divulge information that was formerly only implicit. They are a tribute to the power of names.

This is a good time to declare `BottleVerse`'s tests complete and move on to `Bottles`.

9.3. Seeking Context Independence

The word "context" has been used a number of times under the assumption that its technical meaning would be clear based on, well, context. You've probably already developed an idea about its meaning, but a discussion of the term is in order.

An object's context is its surrounding environment, or the interrelated conditions under which it can exist. Objects that require large, convoluted contexts are picky about their surroundings; they know too much about the outside world. They require particular things to be true about their environment, which limits their use to very specific situations.

Objects are more usable when they know less. Objects that expect little of their surroundings can be more easily applied to novel situations, and so provide greater utility. The most useful are those that are entirely independent of context; they have no expectations about, and make no demands upon the external world. As far as context goes, ignorance truly is bliss.

This section explores these ideas by examining the `Bottles` class and its remaining tests with an eye towards reducing their context.

9.3.1. Examining Bottles' Responsibilities

Before looking at `Bottles`' tests, pause and refresh your memory of the class itself.

Listing 9.15: Reconsidering Bottles

```
1 class Bottles {
2   constructor(verseTemplate = BottleVerse) {
3     this.verseTemplate = verseTemplate;
4   }
5
6   song() {
7     return this.verses(99, 0);
8   }
9
10  verses(starting, ending) {
11    return downTo(starting, ending)
12      .map(i => this.verse(i))
13      .join('\n');
14  }
15
16  verse(number) {
17    return this.verseTemplate.lyrics(number);
18  }
19 }
```

The code above is the end result of iteratively extracting bits that needed to vary. The outcome is a class that has almost nothing to do with the "99 Bottles" song. It retains vestiges of its former context (the name `Bottles`, the default `BottleVerse`, the magic number `99`) but someone unfamiliar with the code's history would not recognize this class as that song.

`Bottles` is no longer the right name. This is not a `Bottles` at all; it's something completely different.

The name `Bottles` diminishes this class, making it appear to be less than it is. Remnants of the circumstances from which it came continue to bind it to a specific context, even though it is now more generally useful. The class has become more abstract but its very name will prevent programmers from recognizing its broader utility.

Take a fresh look at this code and describe what it does. Classes should be named after what they are: what is this?

You might be tempted to say it's a `Song`. That is correct, but you can do better. It's a particular kind of song, and is distinguished by the `downTo` on line 11 above. It's reasonable to assume that a song with numbered verses will count up. This song is a bit unexpected in that it starts at the top and counts down. This is a `CountdownSong`.

The next section renames `Bottles` to `CountdownSong`, but before moving on, sit back for a minute and think about how dissimilar a class named `CountdownSong` feels from a class named `Bottles`. `Bottles` is parochial and useful in one specific case. `CountdownSong` is generic and useful in many cases. For a class named `Bottles`, the `BottleVerse` default and the hard-coded

`99` seem entirely reasonable. In a class named `CountdownSong`, these relics of a prior context are just plain annoying, and you will be motivated to remove them. `Bottles` constricts possibilities; `CountdownSong` expands them.

Did you feel the world pivot under your feet? This is yet another testament to the power of names.

`Bottles` can be renamed to `CountdownSong` with a straightforward refactoring. As always, the process is to make one-undo changes and ensure the tests pass after every change.

Start by duplicating the entire `Bottles` class and changing the duplicate copy's name to `CountdownSong`:

Listing 9.16: Duplicate Bottles as CountdownSong

```
class CountdownSong {
  constructor(verseTemplate = BottleVerse) {
    this.verseTemplate = verseTemplate;
  }

  song() {
    return this.verses(99, 0);
  }

  verses(starting, ending) {
    return downTo(starting, ending)
      .map(i => this.verse(i))
      .join('\n');
  }

  verse(number) {
    return this.verseTemplate.lyrics(number);
  }
}

class Bottles {
  // ...
}
```

Now that `CountdownSong` exists, you can update the tests.

First, rename the test from `the 'Bottles' block` to `the 'CountdownSong' block` as shown on line 1 below:

Listing 9.17: Change Test Name

```
describe('CountdownSong', () => {
  test('a couple verses', () => {
    // ...
    expect(new Bottles().verses(99, 98)).toBe(expected);
  });

  test('a few verses', () => {
    // ...
    expect(new Bottles().verses(2, 0)).toBe(expected);
  });

  test('the whole song', () => {
    // ...
```

```
14 |     expect(new Bottles().song()).toBe(expected);
15 |   });
16 |});
```

Now go into `the 'CountdownSong' block` and update the places that mention `Bottles` to refer to `CountdownSong` (lines 4, 9, and 14 below):

Listing 9.18: Use CountdownSong in Tests

```
1 |describe('CountdownSong', () => {
2 |   test('a couple verses', () => {
3 |     // ...
4 |     expect(new CountdownSong().verses(99, 98)).toBe(expected);
5 |   });
6 |
7 |   test('a few verses', () => {
8 |     // ...
9 |     expect(new CountdownSong().verses(2, 0)).toBe(expected);
10|   });
11|
12|   test('the whole song', () => {
13|     // ...
14|     expect(new CountdownSong().song()).toBe(expected);
15|   });
16|});
```

At this point the old `Bottles` class is no longer used, so you can delete it. This should leave you with 10 passing tests.

This process for changing a class name involved a few small steps rather than one big one. It ordered the five necessary changes (the class's name, the test's name, and the test's three references to the old class name) so they could be made in series, all the while keeping the tests running green.

In a problem this small you'd likely have been fine making all of the changes at once, but you shouldn't. Build a habit of doing refactorings. When you insist on making changes under green, you never have far to go to debug whatever it is that you just broke. This makes everything easier. It's cheaper to always do refactorings than to sometimes debug big piles of altered code.

9.3.2. Purifying Tests With Fakes

Now that the `CountdownSong` exists, take a closer look at its tests (repeated below):

Listing 9.19: CountdownSong Tests Still Stuck to 99 Bottles Lyrics

```
1 |describe('CountdownSong', () => {
2 |   test('a couple verses', () => {
3 |     const expected =
4 |       '99 bottles of beer on the wall, ' +
5 |       '99 bottles of beer.\n' +
6 |       'Take one down and pass it around, ' +
7 |       '98 bottles of beer on the wall.\n' +
8 |       '\n' +
9 |       '98 bottles of beer on the wall, ' +
10|       '98 bottles of beer.\n' +
11|       'Take one down and pass it around, ' +
12|       '97 bottles of beer on the wall.\n';
13|     expect(new CountdownSong().verses(99, 98)).toBe(expected);
```

```
14 |   });
15 |
16 |   test('a few verses', () => {
17 |     const expected =
18 |       '2 bottles of beer on the wall, ' +
19 |       '2 bottles of beer.\n' +
20 |       'Take one down and pass it around, ' +
21 |       '1 bottle of beer on the wall.\n' +
22 |       '\n' +
23 |       '1 bottle of beer on the wall, ' +
24 |       '1 bottle of beer.\n' +
25 |       'Take it down and pass it around, ' +
26 |       'no more bottles of beer on the wall.\n' +
27 |       '\n' +
28 |       'No more bottles of beer on the wall, ' +
29 |       'no more bottles of beer.\n' +
30 |       'Go to the store and buy some more, ' +
31 |       '99 bottles of beer on the wall.\n';
32 |     expect(new CountdownSong().verses(2, 0)).toBe(expected);
33 |   });
34 |
35 |   test('the whole song', () => {
36 |     const expected =
37 |`99 bottles of beer on the wall, 99 bottles of beer.
38 |Take one down and pass it around, 98 bottles of beer on the wall.
39 |
40 |98 bottles of beer on the wall, 98 bottles of beer.
41 |Take one down and pass it around, 97 bottles of beer on the wall.
42 |
43 |97 bottles of beer on the wall, 97 bottles of beer.
44 |Take one down and pass it around, 96 bottles of beer on the wall.
45 |
46 |// ...
47 |
48 |No more bottles of beer on the wall, no more bottles of beer.
49 |Go to the store and buy some more, 99 bottles of beer on the wall.
50 |`;
51 |     expect(new CountdownSong().song()).toBe(expected);
52 |   });
53 |});
```

These tests are useful in that they prevent regressions, but they're an abysmal failure at telling the story of `CountdownSong`. They suffer from the problem you just fixed in the code; they still reflect the context from which they came. The expectations are misleading, and the 290 lines of code elided on line 46 drag your mind back to the "99 Bottles" context by sheer weight alone. These tests strongly imply that `CountdownSong` is about "99 Bottles."

It's a given that tests should prove that code works. But they have other purposes too, one of which is to explain the domain to the reader. The story told by these tests is not the story of `CountdownSong`. Tests should explain the essence of a class. The core responsibility of `CountdownSong` is to take a verse template and produce a song that counts down. These tests fail to convey those things.

If you study the tests long enough, you might eventually discern that the expectations contain verses that count down—but then again, you might not. Additionally, the fact that `CountdownSong` accepts an injected verse template is completely invisible. You cannot tell from reading the tests that the verse template can vary.

Consider the first expectation. The `'a couple verses'` test would be much more intention-revealing if instead of this:

```
const expected =
  '99 bottles of beer on the wall, ' +
  '99 bottles of beer.\n' +
  'Take one down and pass it around, ' +
  '98 bottles of beer on the wall.\n' +
  '\n' +
  '98 bottles of beer on the wall, ' +
  '98 bottles of beer.\n' +
  'Take one down and pass it around, ' +
  '97 bottles of beer on the wall.\n';
```

Its expectation read like this:

```
const expected =
  'This is verse 99.\n' +
  '\n' +
  'This is verse 98.\n' +
  '\n' +
  'This is verse 97.\n';
```

The second example above makes it crystal clear that the output is expected to contain verses that count down. This is a better test, and having it is as easy as imagining it. Don't be bound to the original context by your knowledge of existing verse templates; `CountdownSong`'s tests are not doomed to collaborate with `BottleVerse`. Formulate a wish for the story you want to tell and then make that wish come true.

Doing so simply requires creating a new player of the verse template role that fulfills this more intention-revealing expectation. Here's a new class that does just that:

Listing 9.20: VerseFake With Lyrics Implementation

```
class VerseFake {
  static lyrics(number) {
    return `This is verse ${number}.\n`;
  }
}
```

The `VerseFake` class above is perfect for your needs, though it must be acknowledged that it unrepentantly breaks several common programming rules.

First, Chapter 8 suggested that you put domain behavior on instances. This class violates that rule; its behavior is on the static side.

Next, there's an as-yet-unmentioned object-oriented programming rule that prohibits the use of pattern names in class names. The word "Fake" above refers to a testing pattern, so naming this class `VerseFake` violates that rule.

Fake things first. You're probably familiar with the idea of design patterns[23], which are named, re-usable solutions to common software problems. Pattern names act as shortcuts to big ideas and allow programmers to communicate with speed and precision. Pattern thinking has so influenced software design that most programmers are familiar with a number of patterns. For

example, you've likely heard of Decorator, Adapter, Enumerator, and so on, even if you're a bit fuzzy on the specifics of some of their definitions.

Since pattern names are so meaningful, it can be tempting to stick them in class names. For example, you might use the Decorator[24] pattern to enclose a `number` in a new class that adds additional responsibility. Initially, `NumberDecorator` might seem like a good name for the result. The problem with including the name of a pattern in the name of a class is that this permits you the feeling of having created a useful name without actually having done so. Pattern names don't generally reflect concepts in your application. Appending them to class names pollutes your domain with programmer-y words and circumvents the search for names that add semantic meaning. Class names that include patterns are a signal that you've given up too soon on the hard problem of naming.

Class names should reflect concepts in your domain, not the patterns used to create them. Compared to `BottleNumber`, the much-richer name you gave this class in Chapter 4, `NumberDecorator` is so abstract as to be meaningless. Future readers won't care that the class was created using Decoration but they'll be grateful to know that it's a bottle-ish kind of number.

The xUnit Test Patterns[25] book by Gerard Meszaros standardizes the pattern names of a set of objects that are used to simplify testing.[26]. `TestDouble` is his generic name for all of the patterns. Within `TestDouble` he further delineates the `Dummy`, `Stub`, `Spy`, `Mock`, `Fake`, and `Temporary Test Stub` patterns.

Meszaros defines `Fake` as a `TestDouble` that provides a lightweight implementation of a collaborator that is needed by the class you are actually unit testing. A `Fake` is a regular old object; no testing magic is involved. In this case the new `VerseFake` class is a real player of the verse template role; it's called a `Fake` because it's only used during testing. `BottleVerse` plays the role of verse template in production. `VerseFake` was created to play this role during `Bottles`' unit tests.

The upshot is that `Fake` is the name of a pattern, so `VerseFake` violates the don't-include-pattern-names-in-class-names rule.

Rules exist to save money, and the two rules that `VerseFake` breaks are primarily meant to save money in production code; they might not be so applicable in code created to simplify tests. For example, the purpose of `VerseFake` is to fake the role of verse template. In this case, `VerseFake` might be the most intention-revealing name possible. If you end up needing a number of different kinds of fakes, you might need additional qualifiers in their names (`SimpleVerseFake`, `ComplicatedVerseFake`) but the word "fake" still adds meaning in the domain of your tests.

Similarly, it's important that the shape of production code not interfere with your ability to change it. The put-domain-behavior-on-instances rule serves this goal. In tests, however, you're less concerned with preserving the fake's changeability and more interested in directly communicating its responsibilities. Putting the behavior in a static method simplifies the code in `VerseFake` at the expense of making it less adaptable. This is a trade-off you'll happily make in code used only by the tests.

To use `VerseFake`, first create an alternate `'a couple verses'` test, as shown on line 2 below:

Listing 9.21: Introduce Alternate Test

```
1 describe('CountdownSong', () => {
2   test('a couple verses', () => {
3   });
4
5   test('a couple verses', () => {
6     const expected =
7       '99 bottles of beer on the wall, ' +
8       '99 bottles of beer.\n' +
9       'Take one down and pass it around, ' +
10      '98 bottles of beer on the wall.\n' +
11      '\n' +
12      '98 bottles of beer on the wall, ' +
13      '98 bottles of beer.\n' +
14      'Take one down and pass it around, ' +
15      '97 bottles of beer on the wall.\n';
16    expect(new CountdownSong().verses(99, 98)).toBe(expected);
17  });
18  // ...
19 });
```

Next, fill this alternate test with an entirely new implementation that uses the fake, as shown here:

Listing 9.22: Parse Alternate Test

```
1 describe('CountdownSong', () => {
2   test('a couple verses', () => {
3     const expected =
4       'This is verse 99.\n' +
5       '\n' +
6       'This is verse 98.\n' +
7       '\n' +
8       'This is verse 97.\n';
9     expect(new CountdownSong(VerseFake).verses(99, 97)).toBe(expected);
10  });
11
12  test('a couple verses', () => {
13    const expected =
14      '99 bottles of beer on the wall, ' +
15      // ...
16    expect(new CountdownSong().verses(99, 98)).toBe(expected);
17  });
18  // ...
19 });
```

The alternate `'a couple verses'` test above injects `VerseFake` (line 9) and sets its expectations (line 3) based on the behavior of that verse template.

Running the tests will execute both tests, so at this point 11 should pass.

Now that the alternate test has been shown to work, delete the old one to cleanup obsolete code. You should again have 10 passing tests.

This reduces `the 'CountdownSong' block` to:

Listing 9.23: Retain Alternate Test

```
1 describe('CountdownSong', () => {
2   test('a couple verses', () => {
3     const expected =
4       'This is verse 99.\n' +
5       '\n' +
6       'This is verse 98.\n' +
7       '\n' +
8       'This is verse 97.\n';
9     expect(new CountdownSong(VerseFake).verses(99, 97)).toBe(expected);
10  });
11
12  test('a few verses', () => {
13    // ...
14  });
15
16  test('the whole song', () => {
17    // ...
18  });
19 });
```

Notice that `'a couple verses'` is more expressive now that it contains less noise. This new version also verifies three contiguous verses (the `99, 97` on line 9) rather than only two. `VerseFake`'s verses are so short that you can broaden the expectation to illustrate more clearly that the input is a range without overly complicating the test.

The `'a couple verses'` name is now out-of-date. Make a note that it needs to be updated, but tolerate it for the moment. Working on the other `CountdownSong` tests may provide more information about how to name this one.

Relative to the old, the body of this new test is both more intention-revealing *and* shorter. Using the fake converted it into a model of expressive concision. One might wish all tests to be so straightforward.

9.3.3. Purging Redundant Tests

Next, turn your attention to the other test that makes assertions about verses, `'a few verses'`, shown on line 12 below:

Listing 9.24: Comparing Verses Tests

```
1 describe('CountdownSong', () => {
2   test('a couple verses', () => {
3     const expected =
4       'This is verse 99.\n' +
5       '\n' +
6       'This is verse 98.\n' +
7       '\n' +
8       'This is verse 97.\n';
9     expect(new CountdownSong(VerseFake).verses(99, 97)).toBe(expected);
10  });
11
12  test('a few verses', () => {
13    const expected =
14      '2 bottles of beer on the wall, ' +
15      '2 bottles of beer.\n' +
16      'Take one down and pass it around, ' +
17      '1 bottle of beer on the wall.\n' +
```

```
      '\n' +
      '1 bottle of beer on the wall, ' +
      '1 bottle of beer.\n' +
      'Take it down and pass it around, ' +
      'no more bottles of beer on the wall.\n' +
      '\n' +
      'No more bottles of beer on the wall, ' +
      'no more bottles of beer.\n' +
      'Go to the store and buy some more, ' +
      '99 bottles of beer on the wall.\n';
    expect(new CountdownSong().verses(2, 0)).toBe(expected);
  });

  test('the whole song', () => {
    // ...
  });
});
```

Is `'a few verses'` necessary? If `'a couple verses'` and `'a few verses'` test different things, you should keep both. If they test same thing, you should delete one of them.

Even though the tests define different expectations, they are logically identical. Each test asserts that `CountdownSong` produces the correct list of verses for a given verse template and range. Since they test the same thing, you don't need both.

They have been redundant since they were initially created back in Chapter 2. It's so easy now to recognize that `'a few verses'` isn't needed that it's instructive to ask why this test ever seemed necessary.

Notice that the expectation in `'a few verses'` lists every verse of the initial "99 Bottles" implementation that did *not* follow the common rule. Put another way, verse 2, 1, and 0 were special, and each rightly had their own individual verse test. The fact that these single verses were special made it somehow seem reasonable to test them in combination. But this was never necessary. As long as you have tests to prove that every `verse` is correct, you don't need to test `verses` against more than one range. If `verses` works with one, it will work with all.

This point is more effectively made by imagining the logical outcome of adding other redundant `verses` tests. If you feel the need to test both 99-97 and 2-0, then why not 99-96? And 99-95? 40-28, etc? This way lies madness. These superfluous tests confuse the reader and increase maintenance costs without adding additional safety. One `verses` test is enough. Stop there.

Deleting `'a few verses'` condenses the tests to:

Listing 9.25: Testing the Verses Method Just Once

```
describe('CountdownSong', () => {
  test('a couple verses', () => {
    const expected =
      'This is verse 99.\n' +
      // ...
      'This is verse 97.\n';
    expect(new CountdownSong(VerseFake).verses(99, 97)).toBe(expected);
  });

  test('the whole song', () => {
    const expected =
```

```
`99 bottles of beer on the wall, 99 bottles of beer.
// ...
Go to the store and buy some more, 99 bottles of beer on the wall.
`;
    expect(new CountdownSong().song()).toBe(expected);
  });
});
```

The resulting `'CountdownSong' block` contains two tests. The entire suite now contains nine tests, all of which pass.

`CountdownSong`'s public API contains three methods, but `the 'CountdownSong' block` covers only two of them. The above example has no test for `verse`. Since `CountdownSong` offers this public method, `the 'CountdownSong' block` must test it.

`The 'CountdownSong' block` originally tested the entire API (`song`, `verses` and `verse`). However, the original `verse` tests were specific to "99 Bottles" and they recently moved to `the 'BottleVerse' block` where they belong. This move left a gap in `CountdownSong`'s coverage; it should have a test for `verse`.

Having gone through the exercise of creating and using `VerseFake`, it's exceedingly easy to construct an intention-revealing test for `verse` in `the 'CountdownSong' block`. Here's an example:

Listing 9.26: Test Entire API in CountdownSongTest

```
describe('CountdownSong', () => {
  test('verse', () => {
    const expected = 'This is verse 500.\n';
    expect(new CountdownSong(VerseFake).verse(500)).toBe(expected);
  });

  test('a couple verses', () => {
    // ...
  });

  test('the whole song', () => {
    // ...
  });
});
```

Notice that `CountdownSong` tests `verse` using `500` for the verse number. It intentionally conveys the idea that a number outside the 99-0 range is acceptable. This is visible evidence that `CountdownSong` is not constrained by the limits of the "99 Bottles" context from which it came.

There should again be 10 passing tests.

Now that you've chosen `'verse'` for the name of the test for `verse`, reevaluate the `'a couple verses'` name. Calling this test `'verse'` suggests a pattern, and following the pattern would make the other tests easier to understand. Your intention would be clearer if `'a couple verses'` was renamed to `'verses'`, as shown below:

Listing 9.27: Improve Verses Test Name

```
describe('CountdownSong', () => {
  test('verse', () => {
```

```
3 |     const expected = 'This is verse 500.\n';
4 |     expect(new CountdownSong(VerseFake).verse(500)).toBe(expected);
5 |   });
6 |
7 |   test('verses', () => {
8 |     const expected =
9 |       'This is verse 99.\n' +
10|       '\n' +
11|       'This is verse 98.\n' +
12|       '\n' +
13|       'This is verse 97.\n';
14|     expect(new CountdownSong(VerseFake).verses(99, 97)).toBe(expected);
15|   });
16|
17|   test('the whole song', () => {
18|     // ...
19|   });
20|});
```

Using `VerseFake` greatly simplified the `verse` and `verses` tests. The fake enhances the story of `CountdownSong` rather than distracting from it, as did the old expectations with their obsolete "99 Bottles" context. A glance at these two tests makes it clear that `CountdownSong` has nothing to do with "99 Bottles", but instead, it's about counting down.

The one remaining test is for `song`. It currently contains 305 lines of code, most of which are misleading.

9.3.4. Profiting from Loose Coupling

Having clarified the `verse` and `verses` tests, it's now time to address `'the whole song'`, shown in an abbreviated form below:

Listing 9.28: Revisit Test the Whole Song

```
1 |describe('CountdownSong', () => {
2 |  // ...
3 |  test('the whole song', () => {
4 |    const expected =
5 |`99 bottles of beer on the wall, 99 bottles of beer.
6 |Take one down and pass it around, 98 bottles of beer on the wall.
7 |
8 |98 bottles of beer on the wall, 98 bottles of beer.
9 |Take one down and pass it around, 97 bottles of beer on the wall.
10|
11|// ...
12|
13|1 bottle of beer on the wall, 1 bottle of beer.
14|Take it down and pass it around, no more bottles of beer on the wall.
15|
16|No more bottles of beer on the wall, no more bottles of beer.
17|Go to the store and buy some more, 99 bottles of beer on the wall.
18|`;
19|    expect(new CountdownSong().song()).toBe(expected);
20|  });
21|});
```

This test needs work. Counting the lines elided by the comment on line 11 above, it's 305 lines long, most of which involve a 301-line expectation. This wall of text makes it impossible to understand the story the test wants to tell.

You've already employed `VerseFake` to improve the other tests. Injecting it here would certainly help, but would not solve the wall-of-text problem. Even if you used the fake to simplify each individual verse listed in the expectation, the expectation would still be dauntingly long, listing 100 verses.

Have a look at `CountdownSong` and see if you can spot the source of this problem:

Listing 9.29: CountdownSong Reprise

```
1 class CountdownSong {
2   constructor(verseTemplate = BottleVerse) {
3     this.verseTemplate = verseTemplate;
4   }
5
6   song() {
7     return this.verses(99, 0);
8   }
9
10  verses(starting, ending) {
11    return downTo(starting, ending)
12      .map(i => this.verse(i))
13      .join('\n');
14  }
15
16  verse(number) {
17    return this.verseTemplate.lyrics(number);
18  }
19 }
```

The `99` on line 7 above condemns you to an expectation that lists 100 verses.

Early in this chapter three remnants of the original "99 Bottles" context were identified in this song-producing code. First, the class was named `Bottles`, second, the verse template defaulted to `BottleVerse`, and third, the class contained a hard-coded `99`.

Generalizing the class name to `CountdownSong` resolved the first problem but the other two vestiges remain. These echos of a now-obsolete context mislead readers and interfere with reuse. Since tests are the first reuse of your code, reusability problems are exposed by the tests. This is exactly what's behind that wall of text in `'the whole song'`. The awkward test is pointing out that all `CountdownSong`'s are required to have exactly 100 verses. This is cripplingly restrictive.

Fortunately, it's easy to gain independence from the old context; just alter `CountdownSong` to handle songs of any number of verses, within any range. Extract and then inject the `99` and the `0` to isolate the things you want to vary.

The first step is to name the concepts represented by these numbers. You might, for example, choose `max` and `min`.

Having done that, you can change `CountdownSong` to accept and save these arguments during `construction`, as shown below:

Listing 9.30: Inject and Set Max and Min

```
1 class CountdownSong {
2   constructor(verseTemplate = BottleVerse, max = 99, min = 0) {
```

```
3 |    this.verseTemplate = verseTemplate;
4 |    this.max = max;
5 |    this.min = min;
6 |  }
7 |  // ...
8 |}
```

Now the `song` method can refer to these abstract names rather than to the concrete values on line 9 below:

Listing 9.31: Use Max and Min in Song

```
1 |class CountdownSong {
2 |  constructor(verseTemplate = BottleVerse, max = 99, min = 0) {
3 |    this.verseTemplate = verseTemplate;
4 |    this.max = max;
5 |    this.min = min;
6 |  }
7 |
8 |  song() {
9 |    return this.verses(this.max, this.min);
10|  }
11|
12|  verses(starting, ending) {
13|    return downTo(starting, ending)
14|      .map(i => this.verse(i))
15|      .join('\n');
16|  }
17|
18|  verse(number) {
19|    return this.verseTemplate.lyrics(number);
20|  }
21|}
```

Thus the dependency is inverted. Where `CountdownSong` previously depended on the `99` and `0` concretions, it now depends on the `max` and `min` abstractions. Where it was once handcuffed by the context from which it came, it is now independent of that context. Paradoxically, now that it knows less, it can do more.

This completes the extraction of `max` and `min`. Before moving on to update the tests, there a few things to note about the code.

First, `max` and `min` are given defaults on line 2, and these defaults look suspiciously like values that make sense only in the context of the "99 Bottles" song. It would be better to pass more general defaults, but you can't do that yet because `'the whole song'` will fail unless max is `99`. For now just recognize that you'd like this default to be context-independent and be alert for an opportunity to make it so.

Next, notice that the song accepts pairs of numbers delineating a range of verses in two different places (lines 2 and 12). In one place, the pair is named `max` and `min`, and in the other, `upper` and `lower`. This is intentional. The idea that an entire countdown song has a fixed number of verses is embodied in `max` and `min`. This is about the song. In contrast, the idea that the `verses` method can produce a requested range of descending verses is reflected in `upper` and `lower`. This is about the subset of verses to be produced. These are two different concepts, and giving them different names clarifies the code.

Now that `max` and `min` exist, you can use them to simplify `'the whole song'`. Line 3 below shows a much shorter but arguably more useful song test:

Listing 9.32: Alternate Whole Song Test

```
describe('CountdownSong', () => {
  // ...
  test('the whole song', () => {
    const expected =
      'This is verse 47.\n' +
      '\n' +
      'This is verse 46.\n' +
      '\n' +
      'This is verse 45.\n' +
      '\n' +
      'This is verse 44.\n' +
      '\n' +
      'This is verse 43.\n';
    expect(new CountdownSong(VerseFake, 47, 43).song()).toBe(expected);
  });

  test('the whole song', () => {
    const expected =
`99 bottles of beer on the wall, 99 bottles of beer.
Take one down and pass it around, 98 bottles of beer on the wall.
// ...
No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99 bottles of beer on the wall.
`;
    expect(new CountdownSong().song()).toBe(expected);
  });
});
```

Now that this better test exists, you can delete the older version.

You're almost done with the `'CountdownSong'` block. The only remaining task is to rename `'the whole song'` so that it's consistent with the other names. Calling it `'song'` works perfectly well, and after making it so the final the `'CountdownSong'` block looks like this:

Listing 9.33: A Beautiful Test Suite

```
describe('CountdownSong', () => {
  test('verse', () => {
    const expected = 'This is verse 500.\n';
    expect(new CountdownSong(VerseFake).verse(500)).toBe(expected);
  });

  test('verses', () => {
    const expected =
      'This is verse 99.\n' +
      '\n' +
      'This is verse 98.\n' +
      '\n' +
      'This is verse 97.\n';
    expect(new CountdownSong(VerseFake).verses(99, 97)).toBe(expected);
  });

  test('song', () => {
    const expected =
      'This is verse 47.\n' +
      '\n' +
      'This is verse 46.\n' +
```

```
      '\n' +
      'This is verse 45.\n' +
      '\n' +
      'This is verse 44.\n' +
      '\n' +
      'This is verse 43.\n';
    expect(new CountdownSong(VerseFake, 47, 43).song()).toBe(expected);
  });
});
```

The above is a dramatically simplified but information-rich 30 lines of tests. Not only are the tests much improved, but your insistence on having high-value tests like these forced you to improve the code.

This point cannot be emphasized strongly enough. Tight coupling between objects made testing difficult. Instead of succumbing to those flaws and writing verbose, confusing tests, the code was altered to allow you to write better tests. The changes to the code loosened the coupling between objects by extracting and injecting dependencies. Now that `max`, `min` and `verseTemplate` are being injected, the code is more broadly useful, and the tests convey more information.

Insisting on simple tests improved both the tests and the code. It ought to be easy to reuse objects. Tests are a form of reuse. Making code easy to test therefore serves the greater purpose of making it easy to reuse. This pays off now and forevermore.

9.4. Communicating With the Future

The tests have been squarely focused on communicating the intent of the code to imagined readers. This commitment to thorough communication has guided the choice of where to locate tests, what to name them, and which collaborators to provide. As helpful and necessary as the previous changes were, there's still a bit more that can be done to improve the code.

This last section makes a few final changes to add more information and safety, and to reduce potential confusion. Think of this round as the last pass through the code before you check it all in and walk away.

9.4.1. Enriching Code with Signals

All current tests share a similar three-phase structure. Each test:

1. defines an expectation,
2. executes some code, and
3. asserts that the result matches the expectation

This pattern is known as Arrange-Act-Assert (AAA)[27]. Nothing requires that tests be formatted this way. They can obviously be made to work regardless of arrangement. This rigorous consistency isn't for the tests; it's for the humans.

Committing to a common shape lowers the cognitive load imposed upon future readers. When every test is structured in the same way, readers don't have to waste time determining if

spurious differences matter. The very shape of the code communicates "nothing odd to see here, la, la, la, all is boring and normal, move on along."

Choosing to arrange tests in a similar way is a matter of programming *style*. Your team should agree upon a common style. Lacking this, everyone's code will differ in meaningless ways, which raises the cost of reading code.

A pattern that suits all situations can sometimes feel a bit heavy for extremely simple cases. For example, `'verse'` deliberately declares the `expected` variable on line 3 below even though it could have merely hard-coded `'This is verse 500.\n'` on line 4.

Listing 9.34: Style Trumps Substance

```
1 describe('CountdownSong', () => {
2   test('verse', () => {
3     const expected = 'This is verse 500.\n';
4     expect(new CountdownSong(VerseFake).verse(500)).toBe(expected);
5   });
6   // ...
7 });
```

In the test above, the commitment to honoring a common style overrode the desire to shorten the listing. Following the style communicates to readers that there's nothing special or uncommon about this test, and conveying that information is more valuable than reducing this listing by one line.

By definition, future readers of your code know less than you know now. They are swimming in murky water, bumping into dimly perceived concepts, searching for clarity. The very shape of code can be a beacon of light, rich with meaning. You're always on the lookout for code arrangements that send *signals* to these hapless readers.

Style is one kind of signal, but there are others. For example, you may have wondered why `'song'` chose `47` and `43` for `max` and `min` (lines 14 below).

Listing 9.35: Signaling With Prime Numbers

```
 1 describe('CountdownSong', () => {
 2   // ...
 3   test('song', () => {
 4     const expected =
 5       'This is verse 47.\n' +
 6       '\n' +
 7       'This is verse 46.\n' +
 8       '\n' +
 9       'This is verse 45.\n' +
10       '\n' +
11       'This is verse 44.\n' +
12       '\n' +
13       'This is verse 43.\n';
14     expect(new CountdownSong(VerseFake, 47, 43).song()).toBe(expected);
15   });
16 });
```

Among the infinite universe of numbers, why choose these?

Future readers will wonder if there's meaning behind your choice of test data. They want very much to know whether the numbers shown on lines 14 above matter, or if other values would suffice. This problem is easy to solve in tests that take a string because in those cases you can communicate directly with the reader (for example `"any string goes here"`). Communicating that any number will do, however, requires an agreed-upon signal.

This test uses prime numbers to signal arbitrariness. Programmers familiar with the Prime Number Signal ™ will immediately recognize that the `47` and `43` do not matter; the numbers signal that they themselves are arbitrary examples.

Once you start thinking about sending signals, opportunities abound. It's sometimes possible to imbue actual language syntax with additional meaning. For example, the late Jim Weirich[28] took advantage of the fact that Ruby , his preferred language, allows code blocks to be delimited either with curly braces `{...}` or with `do...end`. Because most folks use these variants interchangeably, he felt free to co-opt them to send signals. Weirich used `do...end` to warn that the enclosed block had side-effects, and `{...}` to assure that it did not. Readers of his code remain grateful for these signals.

You might be wondering if it would be more straightforward to break down and write a comment rather than trying to send a signal. After all, a comment can be read by anyone, but someone unfamiliar with the signal will miss it. While true, this ignores a key quality of comments. Maintenance is hit-or-miss, and over time they often become invisible and outdated. As long as your team can agree upon a style guide, code-based signals are ultimately more reliable than comments.

Now that you're familiar with the Prime Number Signal, you might be wondering if every test for `CountdownSong` should use it. The short answer is that it would be perfectly fine to go back and change those other tests to use prime numbers. The longer answer is that it might not be necessary to do so.

The `500` used in `'verse'` is so ridiculously large that it already signals that any number, even a big one, will do. The `99/97` used in `'verses'` is a bit more difficult to defend, but readers will probably not be confused. If you are concerned that they will be, you should go alter those tests to use prime numbers right now.

Signals offer a cheap way to add valuable information to code. Look for opportunities to develop and use them. Even if your team consists only of you, the information imparted by a signal will be useful to your future self.

9.4.2. Verifying Roles

JavaScript is dynamically typed and doesn't have syntax to support interfaces. This presents both opportunities and challenges. On the positive side, these qualities make it supremely easy to create and use polymorphic duck types.[29] Lack of type declaration syntax makes JavaScript easy to write; the compilers' needs don't get in the way of creating direct, straightforward code. Additionally, JavaScript's dynamism gives you freedom to do amazing things, unchecked by the language itself, for better or worse.

Of course, the downside to not having a compiler checking types is that there's no compiler checking types. You may do anything without constraint, including sending messages to objects that don't understand them and suffering run-time failures. In dynamically typed languages, the challenge of ensuring that your duck types quack correctly rests solely upon you.

There are a number of ways to accomplish this. In order of lesser to greater ceremony, the options are as follows, with implications for `lyrics`:

1. Tell everyone to implement the correct API in their role players.
   Say "Hey, everyone should make their verse template objects respond to `lyrics(number)` *" at regular intervals.*
2. Programmatically verify that all players of a role respond to messages in that role's API.
   Test that each player responds to `lyrics`*.*
3. Programmatically verify that all players of a role respond to its API, including defining the correct number of parameters.
   Test that each player responds to `lyrics`*, and that* `lyrics` *has one parameter.*
4. Programmatically verify that all players of a role respond to its API, define the correct number of parameters, receive arguments of the correct types, and return the expected type.
   Test that each player responds to `lyrics(arg)`*, that* `arg` *is a number, and that the result is a string.*
5. Switch to a language with compiler guaranteed interfaces.

On very small teams, option 1 above may suffice. If not, escalate to option 2 and start programmatically confirming that role players meet their obligations. Most times option 2 will provide enough safety and you won't need to endure the additional complexity of implementing option 3. If you find that despite your best efforts you are seeing run-time failures after implementing option 3, the memo is clearly not getting out. Teams that have this problem might be better off switching to a statically-typed language.

To verify a role in Jest, start by creating a helper function:

Listing 9.36: Test to Verify Verse Role Player

```
const testPlaysVerseRole = rolePlayer => {
  test('plays verse role', () => {
    expect(rolePlayer).toHaveProperty('lyrics', expect.any(Function));
  });
};
```

You can add `'plays verse role'` to any test suite by invoking this helper. Notice that the function expects a `rolePlayer`, so tests that use this helper must provide one.

Verifying that `BottleVerse` plays the verse template role is as simple as adding this helper to its tests, like so:

Listing 9.37: Verify BottleVerse Plays Role

```
describe('BottleVerse', () => {
  testPlaysVerseRole(BottleVerse);
  // ...
});
```

Running the tests at this point executes `'plays verse role'` as part of `'BottleVerse' test`, so you should now have 11 passing tests.

Proving that `BottleVerse`, the real production object, plays the verse template role is definitely worthwhile, but now comes the good part. Because `VerseFake` is a real object *it can have its own tests*. It's scarily recursive to think about testing objects that were created for use only in tests, and you should probably not write tests that make assertions about `VerseFake`'s implementation details. However, `VerseFake` has one critically important responsibility; it must play the role of verse template. It's up to you to ensure that it does so.

Now that you have the role-verifying helper, this is surprisingly easy to accomplish. Create a simple `'VerseFake' test` that includes the `testPlaysVerseRole` helper. Here's that entire test:

Listing 9.38: Verify VerseFake Plays Role

```
describe('VerseFake', () => {
  testPlaysVerseRole(VerseFake);
});
```

You've now confirmed that `VerseFake` correctly plays the verse template role. This test prevents the unfortunate circumstance in which tests that collaborate with fakes sometimes continue to pass even after changes to production code have broken the application. If, for example, the API of the verse template role changes from `lyrics` to `sing`, a test will now fail unless you update every role player and the `testPlaysVerseRole` to reflect the new interface. This forces all players of a role, even your fakes, to stay in sync with the API.

You should now have 12 passing tests, and can be confident that every object that purports to play the verse role does so correctly.

Duck typing is powerful but can be perilous. While verbally agreeing on a role's API may be enough to keep very small teams out of trouble, as more people get involved, you'll need a more reliable way to communicate. Role tests are a great investment; they cost little but provide tremendous value. If you're using a dynamically-typed language and leaning on polymorphism and dependency injection, you should write them.

9.4.3. Obliterating Obsolete Context

The final task in this last section is to pass through the code one more time, locating and removing obsolete context. `CountdownSong` retains a few misleading vestiges of "99 Bottles"; it's finally time to remove them.

Consider the code below:

Listing 9.39: CountdownSong Default Reflect Obsolete Context

```
1 class CountdownSong {
2   constructor(verseTemplate = BottleVerse, max = 99, min = 0) {
3     this.verseTemplate = verseTemplate;
4     this.max = max;
5     this.min = min;
6   }
7   // ...
8 }
```

In line 2 above, the default values for `verseTemplate`, `max`, and `min` are artifacts of the "99 Bottles" context from which `CountdownSong` came. These defaults give the impression that the old context still matters. It doesn't, and the best way to make that clear is to change this code.

You now have to decide whether to provide defaults at all, and if so, what values to use. `CountdownSong` could conceivably supply defaults for `max` and `min`, but can't reasonably guess which verse template should be used.

Making outsiders supply a `verseTemplate` is as easy as removing the default, as shown on line 2 below:

Listing 9.40: Remove Verse Template Default

```
1 class CountdownSong {
2   constructor(verseTemplate, max = 99, min = 0) {
3     this.verseTemplate = verseTemplate;
4     this.max = max;
5     this.min = min;
6   }
7   // ...
8 }
```

This couldn't be done earlier because the old tests expected `CountdownSong` to set `verseTemplate` to `BottleVerse`. Now that all tests supply a verse template, they work seamlessly throughout this transition.

This leaves `max` and `min`. The lower number, `min`, feels almost as straightforward as `verseTemplate`. Songs probably count down to `0`; it's easy to defend a decision to default `min` to `0` in `CountdownSong`.

The `max` number is a bit trickier. It can be argued that, just as `CountdownSong` can't know what verse template you want, it can't know how many verses your song contains. Users of `CountdownSong` will likely always need to provide this value. Even so, `CountdownSong` might want to set a default in order to send a signal.

Defaulting `max` to a ridiculously large number signals that `CountdownSong` can handle very long songs. The following snippet does just that:

Listing 9.41: Reset the Max

```
1 constructor(verseTemplate, max = 999999, min = 0) {
2   // ...
3 }
```

Having made that final change, peruse the complete listings and glory in your accomplishments:

Listing 9.42: Final Code

```
class CountdownSong {
  constructor(verseTemplate, max = 999999, min = 0) {
    this.verseTemplate = verseTemplate;
    this.max = max;
    this.min = min;
  }

  song() {
    return this.verses(this.max, this.min);
  }

  verses(starting, ending) {
    return downTo(starting, ending)
      .map(i => this.verse(i))
      .join('\n');
  }

  verse(number) {
    return this.verseTemplate.lyrics(number);
  }
}

class BottleVerse {
  static lyrics(number) {
    return new BottleVerse(BottleNumber.for(number)).lyrics();
  }

  constructor(bottleNumber) {
    this.bottleNumber = bottleNumber;
  }

  lyrics() {
    return (
      capitalize(`${this.bottleNumber} of beer on the wall, `) +
      `${this.bottleNumber} of beer.\n` +
      `${this.bottleNumber.action()}, ` +
      `${this.bottleNumber.successor()} of beer on the wall.\n`
    );
  }
}

class BottleNumber {
  static for(number) {
    let bottleNumberClass;
    switch (number) {
      case 0:
        bottleNumberClass = BottleNumber0;
        break;
      case 1:
        bottleNumberClass = BottleNumber1;
        break;
      case 6:
        bottleNumberClass = BottleNumber6;
        break;
      default:
        bottleNumberClass = BottleNumber;
        break;
    }

    return new bottleNumberClass(number);
  }

  constructor(number) {
```

```
    this.number = number;
  }

  toString() {
    return `${this.quantity()} ${this.container()}`;
  }

  quantity() {
    return this.number.toString();
  }

  container() {
    return 'bottles';
  }

  action() {
    return `Take ${this.pronoun()} down and pass it around`;
  }

  pronoun() {
    return 'one';
  }

  successor() {
    return BottleNumber.for(this.number - 1);
  }
}

class BottleNumber0 extends BottleNumber {
  quantity() {
    return 'no more';
  }

  action() {
    return 'Go to the store and buy some more';
  }

  successor() {
    return BottleNumber.for(99);
  }
}

class BottleNumber1 extends BottleNumber {
  container() {
    return 'bottle';
  }

  pronoun() {
    return 'it';
  }
}

class BottleNumber6 extends BottleNumber {
  quantity() {
    return '1';
  }

  container() {
    return 'six-pack';
  }
}
```

Listing 9.43: Final Tests

```
class VerseFake {
  static lyrics(number) {
    return `This is verse ${number}.\n`;
  }
}

const testPlaysVerseRole = rolePlayer => {
  test('plays verse role', () => {
    expect(rolePlayer).toHaveProperty('lyrics', expect.any(Function));
  });
};

describe('CountdownSong', () => {
  test('verse', () => {
    const expected = 'This is verse 500.\n';
    expect(new CountdownSong(VerseFake).verse(500)).toBe(expected);
  });

  test('verses', () => {
    const expected =
      'This is verse 99.\n' +
      '\n' +
      'This is verse 98.\n' +
      '\n' +
      'This is verse 97.\n';
    expect(new CountdownSong(VerseFake).verses(99, 97)).toBe(expected);
  });

  test('song', () => {
    const expected =
      'This is verse 47.\n' +
      '\n' +
      'This is verse 46.\n' +
      '\n' +
      'This is verse 45.\n' +
      '\n' +
      'This is verse 44.\n' +
      '\n' +
      'This is verse 43.\n';
    expect(new CountdownSong(VerseFake, 47, 43).song()).toBe(expected);
  });
});

describe('BottleVerse', () => {
  testPlaysVerseRole(BottleVerse);

  test('verse general rule upper bound', () => {
    const expected =
      '99 bottles of beer on the wall, ' +
      '99 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '98 bottles of beer on the wall.\n';
    expect(BottleVerse.lyrics(99)).toBe(expected);
  });

  test('verse general rule lower bound', () => {
    const expected =
      '3 bottles of beer on the wall, ' +
      '3 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '2 bottles of beer on the wall.\n';
    expect(BottleVerse.lyrics(3)).toBe(expected);
  });

```

```
65  test('verse 7', () => {
66    const expected =
67      '7 bottles of beer on the wall, ' +
68      '7 bottles of beer.\n' +
69      'Take one down and pass it around, ' +
70      '1 six-pack of beer on the wall.\n';
71    expect(BottleVerse.lyrics(7)).toBe(expected);
72  });
73
74  test('verse 6', () => {
75    const expected =
76      '1 six-pack of beer on the wall, ' +
77      '1 six-pack of beer.\n' +
78      'Take one down and pass it around, ' +
79      '5 bottles of beer on the wall.\n';
80    expect(BottleVerse.lyrics(6)).toBe(expected);
81  });
82
83  test('verse 2', () => {
84    const expected =
85      '2 bottles of beer on the wall, ' +
86      '2 bottles of beer.\n' +
87      'Take one down and pass it around, ' +
88      '1 bottle of beer on the wall.\n';
89    expect(BottleVerse.lyrics(2)).toBe(expected);
90  });
91
92  test('verse 1', () => {
93    const expected =
94      '1 bottle of beer on the wall, ' +
95      '1 bottle of beer.\n' +
96      'Take it down and pass it around, ' +
97      'no more bottles of beer on the wall.\n';
98    expect(BottleVerse.lyrics(1)).toBe(expected);
99  });
100
101 test('verse 0', () => {
102   const expected =
103     'No more bottles of beer on the wall, ' +
104     'no more bottles of beer.\n' +
105     'Go to the store and buy some more, ' +
106     '99 bottles of beer on the wall.\n';
107   expect(BottleVerse.lyrics(0)).toBe(expected);
108 });
109});
110
111describe('VerseFake', () => {
112  testPlaysVerseRole(VerseFake);
113});
```

In the heat of coding, it can be tempting to skip this final, cleanup pass. But don't miss this last bit of scrubbing to make the code shine.

9.5. Summary

Tests help prevent errors in code, but to characterize them so simply is a disservice; they offer far more. Good OO is built upon small, interchangeable objects that interact via abstractions. The behavior of each individual object is often quite obvious, but the same cannot be said for the operation of the whole. Tests fill this breach.

Object-oriented applications rely on message sending. The key virtue of messages is that they add indirection. Messages allow the sender to ask for an abstraction and be confident that the receiver will use the appropriate concrete implementation to fulfill the request. Senders are responsible for knowing what they want, receivers, for knowing how to do it. Separating intention from implementation in this way allows you to introduce new variations without altering existing code; simply create a new object that responds to the original message with a different implementation.

When designed with the following features, object-oriented code can interact with new and unanticipated variants without having to change:

1. *Variants are isolated.*
   They're usually isolated in some kind of object, often a new class.
2. *Variant selection is isolated.*
   Selection happens in factories, which may be as simple as isolated conditionals that choose a class.
3. *Message senders and receivers are loosely coupled.*
   This is commonly accomplished by injecting dependencies.
4. *Variants are interchangeable.*
   Message senders treat injected objects as equivalent players of identical roles.

Initially, this reliance on abstractions and indirection increases the complexity of code. What OO promises in return is a reduction in the future cost of change. Highly concrete, tightly coupled code will resist tomorrow's change. Code that depends on loosely coupled abstractions will encourage it.

Because tests need to execute code, they supply early and direct information about inadequate design, and they provide impetus and inspiration for refinements. When tests are difficult to write, require lots of setup, or can't tell a satisfying story, something is wrong. Listen. Fixing problems now is not only cheaper than fixing them later, but will improve your code, clarify your tests, and make glad your work.

Afterword

Congratulations, you've made it. You are now, if not at the end of all things, at least at the end of *this* thing. Completing this book is an accomplishment, and you deserve to take a minute to revel in your success before moving on. Regardless of your mindset or experience level before you started, you're different now that you're done.

This book has two primary goals. The first relates to *process*, and the second, *perspective*.

The first goal is to supply concrete, repeatable techniques that you can employ to improve your own applications. These techniques were illustrated, not just with code, but also by chronicling the rationale behind every decision—there was no hand-waving around awkward corners. The detailed, specific explanations eventually accumulated into a number of general ideas, or *canons*, about how to write code.

Strive for simplicity. Don't abstract too soon. Focus on smells. Concentrate on difference. Take small steps. Follow the *Flocking Rules*. Refactor under green. Fix the easy problems first. Work horizontally. Seek stable landing points. Be disciplined. Don't chase the shiny thing.

In addition, deal with new requirements by first refactoring existing code to be open to them, and then writing new code to meet them. Achieving openness is usually the more challenging task, but can be sought in absolute safety if you have tests that act as a wall at your back.

You may need better tests. If so, writing them will save you money.

The canons are practical rules that guide the programming *process*. Adhering to them will lower your stress, speed up your work, and improve your code. If you commit to nothing more than to follow them, your reading time will have been well spent.

However, this book is not just about process. It has a second, more abstract, goal—it aspires to infect you with a certain perspective about object-oriented programming. This book wants you to fall in love with polymorphism.

When you write conditionals that supply behavior, and put those conditionals in classes whose names *other* classes know, your code depends upon concretions, and will break with every change.

However, when you disperse behavior into polymorphic objects, you can use factories to isolate both the names of the classes and the conditionals that choose them. Factories instantiate role-playing objects, which you can then inject as dependencies. When you inject smart dependencies, and trust them to behave correctly, your code depends upon abstract roles rather than on concrete classes. This loosens the coupling between objects and makes code open to change.

Trust is necessary, but the path to reaching it is circular. Acting in trust requires faith, and faith can only be earned by trustworthiness. Your objects must be trustworthy, and your code must

trust your objects. Failing at either obligation dooms you to conditionals.

The secret to programming happiness is to combine the canons with the infection, building applications from polymorphic, trustworthy objects, and changing them one step at a time.

That's officially the end, but before you go, one last request.

Hold high standards, but judge yourself gently. Perfection is just not that likely. Most times the requirements you're given aren't quite right, or are incompletely conveyed, or misunderstood, or about to change, ad nauseam. Circumstances conspire to make it hard to get everything exactly right, despite your best efforts.

Think of your code as a message in a bottle, written in haste for future readers. They'll always know more than you know right now. Your job is not to be perfect, but to write a generous and sympathetic story. Tell them a story they can understand, and they'll cherish you forever.

Thanks for reading, and we hope that you've enjoyed the book.

Appendix A: Initial Exercise

Getting the exercise

The code in this book is on GitHub. The simplest way to get the exercise is to clone the repository and check out the correct branch, as follows:

```
git clone --depth=1 --branch=2.0-appendix-b-exercise-10 https://github.com/sandimetz/99bottles_js.gi
t
```

The directory structure for the exercise should look like this:

```
├── lib
│   ├── bottles.js
│   └── helpers.js
└── test
    └── bottles.test.js
```

If you don't have git installed, create the expected directory structure, and then copy and paste the contents of the raw file on GitHub into `bottles.test.js`.

You can also copy and paste the contents of the helper file into `helpers.js`.

Finally, if you don't have an Internet connection, you can find the full code listing for the test suite below, in the Test Suite section.

Doing the exercise

To run the test suite, install the dependencies with `yarn install`, then run the tests with `npm test`.

The test suite contains one failing test, and many skipped tests. Your goal is to write code that passes all of the tests. Follow this protocol:

- run the tests and examine the failure
- write only enough code to pass the failing test
- unskip the next test (this simulates writing it yourself)

Repeat the above until no tests is skipped, and you've written code to pass each one.

Work on this task for 30 minutes. The vast majority of folks do not finish in 30 minutes, but it's useful, for later comparison purposes, to record how far you got. Even if you can't force yourself to stop at that point, take a break at 30 minutes and save your code.

Return to Preface.

Return to Chapter 1.

Test Suite

Listing A.1: Exercise

```
import { Bottles } from '../lib/bottles';

describe('Bottles', () => {
  test('the first verse', () => {
    const expected =
      '99 bottles of beer on the wall, ' +
      '99 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '98 bottles of beer on the wall.\n';
    expect(new Bottles().verse(99)).toBe(expected);
  });

  test.skip('another verse', () => {
    const expected =
      '3 bottles of beer on the wall, ' +
      '3 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '2 bottles of beer on the wall.\n';
    expect(new Bottles().verse(3)).toBe(expected);
  });

  test.skip('verse 2', () => {
    const expected =
      '2 bottles of beer on the wall, ' +
      '2 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '1 bottle of beer on the wall.\n';
    expect(new Bottles().verse(2)).toBe(expected);
  });

  test.skip('verse 1', () => {
    const expected =
      '1 bottle of beer on the wall, ' +
      '1 bottle of beer.\n' +
      'Take it down and pass it around, ' +
      'no more bottles of beer on the wall.\n';
    expect(new Bottles().verse(1)).toBe(expected);
  });

  test.skip('verse 0', () => {
    const expected =
      'No more bottles of beer on the wall, ' +
      'no more bottles of beer.\n' +
      'Go to the store and buy some more, ' +
      '99 bottles of beer on the wall.\n';
    expect(new Bottles().verse(0)).toBe(expected);
  });

  test.skip('a couple verses', () => {
    const expected =
      '99 bottles of beer on the wall, ' +
      '99 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '98 bottles of beer on the wall.\n' +
      '\n' +
      '98 bottles of beer on the wall, ' +
      '98 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '97 bottles of beer on the wall.\n';
    expect(new Bottles().verses(99, 98)).toBe(expected);
```

```
  });

  test.skip('a few verses', () => {
    const expected =
      '2 bottles of beer on the wall, ' +
      '2 bottles of beer.\n' +
      'Take one down and pass it around, ' +
      '1 bottle of beer on the wall.\n' +
      '\n' +
      '1 bottle of beer on the wall, ' +
      '1 bottle of beer.\n' +
      'Take it down and pass it around, ' +
      'no more bottles of beer on the wall.\n' +
      '\n' +
      'No more bottles of beer on the wall, ' +
      'no more bottles of beer.\n' +
      'Go to the store and buy some more, ' +
      '99 bottles of beer on the wall.\n';
    expect(new Bottles().verses(2, 0)).toBe(expected);
  });

  test.skip('the whole song', () => {
    const expected =
`99 bottles of beer on the wall, 99 bottles of beer.
Take one down and pass it around, 98 bottles of beer on the wall.

98 bottles of beer on the wall, 98 bottles of beer.
Take one down and pass it around, 97 bottles of beer on the wall.

97 bottles of beer on the wall, 97 bottles of beer.
Take one down and pass it around, 96 bottles of beer on the wall.

96 bottles of beer on the wall, 96 bottles of beer.
Take one down and pass it around, 95 bottles of beer on the wall.

95 bottles of beer on the wall, 95 bottles of beer.
Take one down and pass it around, 94 bottles of beer on the wall.

94 bottles of beer on the wall, 94 bottles of beer.
Take one down and pass it around, 93 bottles of beer on the wall.

93 bottles of beer on the wall, 93 bottles of beer.
Take one down and pass it around, 92 bottles of beer on the wall.

92 bottles of beer on the wall, 92 bottles of beer.
Take one down and pass it around, 91 bottles of beer on the wall.

91 bottles of beer on the wall, 91 bottles of beer.
Take one down and pass it around, 90 bottles of beer on the wall.

90 bottles of beer on the wall, 90 bottles of beer.
Take one down and pass it around, 89 bottles of beer on the wall.

89 bottles of beer on the wall, 89 bottles of beer.
Take one down and pass it around, 88 bottles of beer on the wall.

88 bottles of beer on the wall, 88 bottles of beer.
Take one down and pass it around, 87 bottles of beer on the wall.

87 bottles of beer on the wall, 87 bottles of beer.
Take one down and pass it around, 86 bottles of beer on the wall.

86 bottles of beer on the wall, 86 bottles of beer.
Take one down and pass it around, 85 bottles of beer on the wall.
```

```
85 bottles of beer on the wall, 85 bottles of beer.
Take one down and pass it around, 84 bottles of beer on the wall.

84 bottles of beer on the wall, 84 bottles of beer.
Take one down and pass it around, 83 bottles of beer on the wall.

83 bottles of beer on the wall, 83 bottles of beer.
Take one down and pass it around, 82 bottles of beer on the wall.

82 bottles of beer on the wall, 82 bottles of beer.
Take one down and pass it around, 81 bottles of beer on the wall.

81 bottles of beer on the wall, 81 bottles of beer.
Take one down and pass it around, 80 bottles of beer on the wall.

80 bottles of beer on the wall, 80 bottles of beer.
Take one down and pass it around, 79 bottles of beer on the wall.

79 bottles of beer on the wall, 79 bottles of beer.
Take one down and pass it around, 78 bottles of beer on the wall.

78 bottles of beer on the wall, 78 bottles of beer.
Take one down and pass it around, 77 bottles of beer on the wall.

77 bottles of beer on the wall, 77 bottles of beer.
Take one down and pass it around, 76 bottles of beer on the wall.

76 bottles of beer on the wall, 76 bottles of beer.
Take one down and pass it around, 75 bottles of beer on the wall.

75 bottles of beer on the wall, 75 bottles of beer.
Take one down and pass it around, 74 bottles of beer on the wall.

74 bottles of beer on the wall, 74 bottles of beer.
Take one down and pass it around, 73 bottles of beer on the wall.

73 bottles of beer on the wall, 73 bottles of beer.
Take one down and pass it around, 72 bottles of beer on the wall.

72 bottles of beer on the wall, 72 bottles of beer.
Take one down and pass it around, 71 bottles of beer on the wall.

71 bottles of beer on the wall, 71 bottles of beer.
Take one down and pass it around, 70 bottles of beer on the wall.

70 bottles of beer on the wall, 70 bottles of beer.
Take one down and pass it around, 69 bottles of beer on the wall.

69 bottles of beer on the wall, 69 bottles of beer.
Take one down and pass it around, 68 bottles of beer on the wall.

68 bottles of beer on the wall, 68 bottles of beer.
Take one down and pass it around, 67 bottles of beer on the wall.

67 bottles of beer on the wall, 67 bottles of beer.
Take one down and pass it around, 66 bottles of beer on the wall.

66 bottles of beer on the wall, 66 bottles of beer.
Take one down and pass it around, 65 bottles of beer on the wall.

65 bottles of beer on the wall, 65 bottles of beer.
Take one down and pass it around, 64 bottles of beer on the wall.

```

```
64 bottles of beer on the wall, 64 bottles of beer.
Take one down and pass it around, 63 bottles of beer on the wall.

63 bottles of beer on the wall, 63 bottles of beer.
Take one down and pass it around, 62 bottles of beer on the wall.

62 bottles of beer on the wall, 62 bottles of beer.
Take one down and pass it around, 61 bottles of beer on the wall.

61 bottles of beer on the wall, 61 bottles of beer.
Take one down and pass it around, 60 bottles of beer on the wall.

60 bottles of beer on the wall, 60 bottles of beer.
Take one down and pass it around, 59 bottles of beer on the wall.

59 bottles of beer on the wall, 59 bottles of beer.
Take one down and pass it around, 58 bottles of beer on the wall.

58 bottles of beer on the wall, 58 bottles of beer.
Take one down and pass it around, 57 bottles of beer on the wall.

57 bottles of beer on the wall, 57 bottles of beer.
Take one down and pass it around, 56 bottles of beer on the wall.

56 bottles of beer on the wall, 56 bottles of beer.
Take one down and pass it around, 55 bottles of beer on the wall.

55 bottles of beer on the wall, 55 bottles of beer.
Take one down and pass it around, 54 bottles of beer on the wall.

54 bottles of beer on the wall, 54 bottles of beer.
Take one down and pass it around, 53 bottles of beer on the wall.

53 bottles of beer on the wall, 53 bottles of beer.
Take one down and pass it around, 52 bottles of beer on the wall.

52 bottles of beer on the wall, 52 bottles of beer.
Take one down and pass it around, 51 bottles of beer on the wall.

51 bottles of beer on the wall, 51 bottles of beer.
Take one down and pass it around, 50 bottles of beer on the wall.

50 bottles of beer on the wall, 50 bottles of beer.
Take one down and pass it around, 49 bottles of beer on the wall.

49 bottles of beer on the wall, 49 bottles of beer.
Take one down and pass it around, 48 bottles of beer on the wall.

48 bottles of beer on the wall, 48 bottles of beer.
Take one down and pass it around, 47 bottles of beer on the wall.

47 bottles of beer on the wall, 47 bottles of beer.
Take one down and pass it around, 46 bottles of beer on the wall.

46 bottles of beer on the wall, 46 bottles of beer.
Take one down and pass it around, 45 bottles of beer on the wall.

45 bottles of beer on the wall, 45 bottles of beer.
Take one down and pass it around, 44 bottles of beer on the wall.

44 bottles of beer on the wall, 44 bottles of beer.
Take one down and pass it around, 43 bottles of beer on the wall.

43 bottles of beer on the wall, 43 bottles of beer.
```

```
Take one down and pass it around, 42 bottles of beer on the wall.

42 bottles of beer on the wall, 42 bottles of beer.
Take one down and pass it around, 41 bottles of beer on the wall.

41 bottles of beer on the wall, 41 bottles of beer.
Take one down and pass it around, 40 bottles of beer on the wall.

40 bottles of beer on the wall, 40 bottles of beer.
Take one down and pass it around, 39 bottles of beer on the wall.

39 bottles of beer on the wall, 39 bottles of beer.
Take one down and pass it around, 38 bottles of beer on the wall.

38 bottles of beer on the wall, 38 bottles of beer.
Take one down and pass it around, 37 bottles of beer on the wall.

37 bottles of beer on the wall, 37 bottles of beer.
Take one down and pass it around, 36 bottles of beer on the wall.

36 bottles of beer on the wall, 36 bottles of beer.
Take one down and pass it around, 35 bottles of beer on the wall.

35 bottles of beer on the wall, 35 bottles of beer.
Take one down and pass it around, 34 bottles of beer on the wall.

34 bottles of beer on the wall, 34 bottles of beer.
Take one down and pass it around, 33 bottles of beer on the wall.

33 bottles of beer on the wall, 33 bottles of beer.
Take one down and pass it around, 32 bottles of beer on the wall.

32 bottles of beer on the wall, 32 bottles of beer.
Take one down and pass it around, 31 bottles of beer on the wall.

31 bottles of beer on the wall, 31 bottles of beer.
Take one down and pass it around, 30 bottles of beer on the wall.

30 bottles of beer on the wall, 30 bottles of beer.
Take one down and pass it around, 29 bottles of beer on the wall.

29 bottles of beer on the wall, 29 bottles of beer.
Take one down and pass it around, 28 bottles of beer on the wall.

28 bottles of beer on the wall, 28 bottles of beer.
Take one down and pass it around, 27 bottles of beer on the wall.

27 bottles of beer on the wall, 27 bottles of beer.
Take one down and pass it around, 26 bottles of beer on the wall.

26 bottles of beer on the wall, 26 bottles of beer.
Take one down and pass it around, 25 bottles of beer on the wall.

25 bottles of beer on the wall, 25 bottles of beer.
Take one down and pass it around, 24 bottles of beer on the wall.

24 bottles of beer on the wall, 24 bottles of beer.
Take one down and pass it around, 23 bottles of beer on the wall.

23 bottles of beer on the wall, 23 bottles of beer.
Take one down and pass it around, 22 bottles of beer on the wall.

22 bottles of beer on the wall, 22 bottles of beer.
Take one down and pass it around, 21 bottles of beer on the wall.
```

```
21 bottles of beer on the wall, 21 bottles of beer.
Take one down and pass it around, 20 bottles of beer on the wall.

20 bottles of beer on the wall, 20 bottles of beer.
Take one down and pass it around, 19 bottles of beer on the wall.

19 bottles of beer on the wall, 19 bottles of beer.
Take one down and pass it around, 18 bottles of beer on the wall.

18 bottles of beer on the wall, 18 bottles of beer.
Take one down and pass it around, 17 bottles of beer on the wall.

17 bottles of beer on the wall, 17 bottles of beer.
Take one down and pass it around, 16 bottles of beer on the wall.

16 bottles of beer on the wall, 16 bottles of beer.
Take one down and pass it around, 15 bottles of beer on the wall.

15 bottles of beer on the wall, 15 bottles of beer.
Take one down and pass it around, 14 bottles of beer on the wall.

14 bottles of beer on the wall, 14 bottles of beer.
Take one down and pass it around, 13 bottles of beer on the wall.

13 bottles of beer on the wall, 13 bottles of beer.
Take one down and pass it around, 12 bottles of beer on the wall.

12 bottles of beer on the wall, 12 bottles of beer.
Take one down and pass it around, 11 bottles of beer on the wall.

11 bottles of beer on the wall, 11 bottles of beer.
Take one down and pass it around, 10 bottles of beer on the wall.

10 bottles of beer on the wall, 10 bottles of beer.
Take one down and pass it around, 9 bottles of beer on the wall.

9 bottles of beer on the wall, 9 bottles of beer.
Take one down and pass it around, 8 bottles of beer on the wall.

8 bottles of beer on the wall, 8 bottles of beer.
Take one down and pass it around, 7 bottles of beer on the wall.

7 bottles of beer on the wall, 7 bottles of beer.
Take one down and pass it around, 6 bottles of beer on the wall.

6 bottles of beer on the wall, 6 bottles of beer.
Take one down and pass it around, 5 bottles of beer on the wall.

5 bottles of beer on the wall, 5 bottles of beer.
Take one down and pass it around, 4 bottles of beer on the wall.

4 bottles of beer on the wall, 4 bottles of beer.
Take one down and pass it around, 3 bottles of beer on the wall.

3 bottles of beer on the wall, 3 bottles of beer.
Take one down and pass it around, 2 bottles of beer on the wall.

2 bottles of beer on the wall, 2 bottles of beer.
Take one down and pass it around, 1 bottle of beer on the wall.

1 bottle of beer on the wall, 1 bottle of beer.
Take it down and pass it around, no more bottles of beer on the wall.

```

```
No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99 bottles of beer on the wall.
`;
    expect(new Bottles().song()).toBe(expected);
  });
});
```

References

Abelson, Harold, and Sussman, Gerald, with Sussman Julie. *Structure and Interpretation of Computer Programs, Second Edition*. Cambridge, MA: The MIT Press, 1996.

Beck, Kent. *Test-driven development by example*. Boston: Addison-Wesley, 2002.

Beck, Kent. "Don't Cross the Beams: Avoiding Interference Between Horizontal and Vertical Refactorings" https://www.facebook.com/kentlbeck/notes. 15 September 2011. Web. 26 Feb 2017.

Fowler, Martin, and Kent Beck. *Refactoring Improving the Design of Existing Code*. Boston: Addison-Wesley, 1999.

Fowler, Martin, with Beck, Kent. *Refactoring Improving the Design of Existing Code, Second Edition*. Boston: Addison-Wesley, 2018.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vissides. *Design patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1994.

Kerievsky, Joshua. *Refactoring To Patterns*. Boston: Addison-Wesley, 2004.

Knuth, Donald. *"The Complexity of Songs."* Communications of the ACM, Volume 27 Issue 4. Association for Computing Machinery (ACM). April 1984. Web. 15 Feb 2019.

Lieberherr, K., Holland, I., Riel, A. *"Object-Oriented Programming: An Objective Sense of Style"*. OOPSLA '88 Proceedings. September, 1988. Web. 15 Feb 2019.

Meszaros, Gerard. xUnit Test Patterns. Boston: Addison-Wesley, 2007.

Acknowledgements

Sandi Metz

This book represents the distillation of innumerable discussions about object-oriented programming and design. As every programmer knows, these conversations aren't for everyone. Accordingly, I am deeply grateful to all those who graciously and enthusiastically participated in extended debates about these ideas.

My heartfelt thanks to Jim Gay, Avdi Grimm, Sara Mei, Katrina Owen, TJ Stankus, and Tom Stuart, all of whom have helped teach the Practical Object Oriented Design course from which this book arose. Their deep expertise and gentle objections challenged my certainties and refined my thoughts. Their influence is felt throughout this book.

I owe a special debt to all who've taken the POOD course. There's nothing like trying to explain something to make it obvious that you don't actually understand it. The explanations herein were vastly improved as a result of students insisting that they make sense. I am grateful for their tenacity.

Julia Trimmer's flair for editing is evident throughout the book. This book is a tribute to her unwavering commitment to readability, and her determination to teach me the usage difference between "which" and "that". I fear she succeeded in only the first of these, so I am grateful for both her past and her future efforts.

And finally, my thanks to Amy Germuth, who smiled and nodded when I swore that I would never write another book, and then watched patiently for years while I battled with this one. Having her in my corner made this possible.

Katrina Owen

Learning to refactor was fun and confusing and strenuous. Perhaps more strenuous for Thomas Drevon and my other colleagues at Bengler than for me. They not only put up with my endless search for better design, but encouraged me in it. I'm sorry for all the spurious abstractions that I saddled you with when I left.

The "99 Bottles of Beer" song is not an obvious choice of topic for a programming book. I don't think we'd have stumbled onto it if it hadn't been for the students at Turing School of Software and Design, whose struggles and trials led me to create Exercism. If you solved the "99 Bottles of Beer" problem on Exercism in the early days, you might be the direct inspiration for this book.

A special thanks goes to Jim Weirich who explored the design ideas present in the "99 Bottles of Beer" problem with us, and who generously shared all of his knowledge and insights.

Writing a book has been an adventure. Thanks to Mariana Lenetis, who would go drinking with me when everything became too much. The hot chocolate was amazing. Thanks also to John Ryan, who has compassionately listened to my rants, and who contributed perspective and

advice. And thanks to Sander who has miraculously taken care of the thousands of small and enormous things to keep our lives running smoothly.

TJ Stankus

In 2005 I found myself in a meeting room with developers from different departments within the vast Duke University system. In that group was a woman named Sandi, who I found myself nodding in agreement with every time she spoke. We bumped into each other again in 2006, at the first RailsConf. Since then, I've enjoyed a working relationship and friendship with Sandi that has been one of the most rewarding aspects of my career.

Around 2014, I was a student in a Practical Object-Oriented Design (POOD) course taught by Sandi and Katrina. It was (and still is) the best programming course I've ever taken. Later, when Sandi and Katrina invited me along to be a co-instructor, I couldn't have been more thrilled and appreciative.

When the first edition of the this book came out, we were able to more quickly cover material in the POOD course, and with the time that freed up, we experimented with additional requirements to the problem. These requirements ended up becoming new content for the second edition of the book you are now reading.

Thank you to the many students whose questions and feedback allowed us to refine the new content over the course of a couple years. There are only three names on the cover of this book, but it is the collective output of an extensive collaboration of engaged and inquisitive programmers. I could not be more grateful for the conversations and generous interactions I've had with each of you.

---

1. From the novel by Joseph Heller, a catch-22 is a paradoxical situation from which you cannot escape because of contradictory rules.
2. As per Cambridge Dictionary, "separated from an event by an amount of time or other events."
3. For those unfamiliar with the fairy tale, this is a reference to everything owned by the Little, Small, Wee Bear in Goldilocks (Goldenlocks) and the Three Bears.
4. This quote was historically thought to originate with Mark Twain but is now widely attributed to Charles Dudley Warner. Twain and Warner were neighbors and the former apparently heard it from the latter.
5. The ABC scores are the magnitude of the vector <A,B,C> as per wikipedia.
6. A red herring is something that misleads or distracts from a relevant or important issue.
7. A hair shirt, or cilice, is an undergarment made of animal hair, worn to induce discomfort as a sign of repentance or atonement.
8. An opportunity cost is the "cost" incurred by not enjoying the benefit associated with the best alternative choice.
9. See Kent Beck Don't Cross the Beams: Avoiding Interference Between Horizontal and Vertical Refactorings.
10. Thanks to Avdi Grimm for the suggestion of using rows and columns in an imaginary spreadsheet to help find names for underlying concepts.
11. Thanks to Tom Stuart for the suggestion that, when you're struggling to name a concept for which you have only a few examples, it can help to imagine other concrete things that might also fall into the same category.
12. "You'll never know less than you know right now" is a quote from Kent Beck.
13. Spidey (or spider) sense is a tingling feeling at the base of Marvel Comics superhero Spider-Man's skull that alerts him to danger.
14. A quote from 1 Corinthians 13:12 of the King James Version of the Christian Bible.
15. Merriam Webster defines bang on as "exactly correct or appropriate."
16. A fine kettle of fish is a muddle, or awkward state of affairs.

17. A google search for exceptions for flow control uncovers many articles, most of which roundly condemn the use of this technique.
18. Pseudocode is an informal high-level description of the operating principle of a computer program or other algorithm.
19. Forwarding is an OO technique whereby a message is directed to another object.
20. Delegation is like forwarding except that in the receiving object `self` refers to the original sender, not the current object.
21. Why Ruby Class Methods Resist Refactoring from the Code Climate Blog.
22. Leaf nodes on a dependency graph are the end of the line. Other objects depend on them, but they depend on no one. As such, they are often simpler than average, and usually represent concepts that are far from the center of your domain.
23. The idea of a Design Pattern was introduced by the architect Christopher Alexander. It was applied to software by the Gang of Four in their seminal book Design Patterns: Elements of Reusable Object-Oriented Software.
24. The decorator pattern allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class.
25. Martin Fowler's introduction to xUnitTest Patterns is a better explanation of why you should read this book than the official book blurb.
26. Any early version of Meszaros's section on Mock, Fakes, Stubs and Dummies is available on his xUnitPatterns website.
27. Arrange Act Assert is a pattern for arranging code in unit tests. It suggests that group your tests into 3 sections: 1) Arrange → inputs and preconditions, 2) Act → on the object or method you're testing, and 3) Assert → that you got what you expected.
28. We still miss you, Jim.
29. In duck typing an object's suitability for use is determined by the messages to which it responds rather than its type.