Code Responsibly

Ruby Edition

99 Bottles of OOP

A Practical Guide to Object-Oriented Design

Sandi Metz & Katrina Owen

99 Bottles of OOP

Sandi Metz · Katrina Owen – 0.3, 2016-10-06 | beta-2

Table of Contents

Colophon
Dedication
Preface
    What This Book Is About
    Who Should Read This Book
    Before You Read This Book
    How to read this book
    Code Examples
    Errata
    About the Authors
Introduction
1. Rediscovering Simplicity
    1.1. Simplifying Code
        1.1.1. Incomprehensibly Concise
            Consistency
            Duplication
            Names
        1.1.2. Speculatively General
        1.1.3. Concretely Abstract
        1.1.4. Shameless Green

1.2. Judging Code
1.2.1. Evaluating Code Based on Opinion
1.2.2. Evaluating Code Based on Facts
Source Lines of Code
Cyclomatic Complexity
Assignments, Branches and Conditions (ABC) Metric
1.2.3. Comparing Solutions
1.3. Summary
2. Test Driving Shameless Green
2.1. Understanding Testing
2.2. Writing the First Test
2.3. Removing Duplication
2.4. Understanding Transformations
2.5. Tolerating Duplication
2.6. Hewing to the Plan
2.7. Exposing Responsibilities
2.8. Choosing Names
2.9. Revealing Intentions
2.10. Writing Cost-Effective Tests
2.11. Avoiding the Echo-Chamber
2.12. Considering Options
2.13. Summary
3. Unearthing Concepts
3.1. Listening to Change
3.2. Starting With the Open/Closed Principle
3.3. Recognizing Code Smells
3.4. Identifying the Best Point of Attack

3.5. Refactoring Systematically
3.6. Following the Flocking Rules
3.7. Converging on Abstractions
3.7.1. Focusing on Difference
3.7.2. Simplifying Hard Problems
3.7.3. Naming Concepts
3.7.4. Making Methodical Transformations
3.7.5. Refactoring Gradually
3.8. Summary
4. Practicing Horizontal Refactoring
4.1. Replacing Difference With Sameness
4.2. Equivocating About Names
4.3. Deriving Names From Responsibilities
4.4. Choosing Meaningful Defaults
4.5. Seeking Stable Landing Points
4.6. Obeying the Liskov Substitution Principle
4.7. Taking Bigger Steps
4.8. Discovering Deeper Abstractions
4.9. Depending on Abstractions
4.10. Summary
5. Separating Responsibilities
5.1. Selecting the Target Code Smell
5.1.1. Identifying Patterns in Code
5.1.2. Spotting Common Qualities
5.1.3. Enumerating Flocked Method Commonalities
5.1.4. Insisting Upon Messages
5.2. Extracting Classes

5.2.1. Modeling Abstractions
5.2.2. Naming Classes
5.2.3. Extracting BottleNumber
5.2.4. Removing Arguments
5.2.5. Trusting the Process
5.3. Appreciating Immutability
5.4. Assuming *Fast Enough*
5.5. Creating BottleNumbers
5.6. Recognizing Liskov Violations
5.7. Summary
6. Replacing Conditionals with Objects
Appendix A: Prerequisites
A.1. Ruby
A.2. Minitest
Appendix B: Initial Exercise
B.1. Getting the exercise
B.2. Doing the exercise
B.3. Test Suite
Acknowledgements

Colophon

Version: 0.3

Version Date: 2016-10-06

Version Notes: beta-2

ISBN-10:1-944823-00-X

ISBN-13:978-1-944823-00-9

Published By: Potato Canyon Software, LLC

1st Edition

Cover Design and Art by Lindsey Morris.

Created using Asciidoctor.

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.

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 and Katrina Owen. These ideas have been battle-tested by hundreds of students, and refined by a series of deeply thoughtful co-instructors, beginning with Katrina. While neither Katrina nor Sandi have the hubris to claim perfect understanding, both have learned a great deal about Object-Oriented Design from teaching this song, and have come to feel that it's time to buck it up and write it down.

Therefore, this book. 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 is not yet available. This chapter performs a miracle which not only removes all conditionals, but also allows you to finally implement the new six-pack requirement without altering any existing code.

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 Ruby, have no

fear. The syntax of the language is so straightforward that you'll have no trouble understanding what follows. The ideas in this book are not about Ruby, they're about object-oriented programming and design.

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 examples are written in Ruby, and the exercises rely on Minitest. The code is available in the 99bottles repository on GitHub, which contains a branch for each chapter.

Errata

A current list of errata is located at www.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.

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, 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 and exposes the sausage 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 to 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 a refreshing 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 (i.e. being willingly 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
  def song
    verses(99, 0)
  end

  def verses(hi, lo)
    hi.downto(lo).map {|n| verse(n) }.join("\n")
  end

  def verse(n)
    "#{n == 0 ? 'No more' : n} bottle#{'s' if n != 1}" +
    " of beer on the wall, " +
    "#{n == 0 ? 'no more' : n} bottle#{'s' if n != 1} of
beer.\n" +
    "#{n > 0  ? "Take #{n > 1 ? 'one' : 'it'} down and pass
it around"
              : "Go to the store and buy some more"}, " +
    "#{n-1 < 0 ? 99 : n-1 == 0 ? 'no more' : n-1} bottle#
{'s' if n-1 != 1}"+
    " of beer on the wall.\n"
  end
end
```

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 11:

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

Other statements are made conditional by adding a trailing `if`. Line 11 again provides an example:

```
's' if n != 1
```

Finally, there's the ternary within a ternary on line 16, 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 11 and

13:

```
's' if n != 1
```

But later, on line 16, the pluralization logic is subtly different. Suddenly it's not `n` that matters, but `n-1`:

```
's' if n-1 != 1
```

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 implicit receiver `self`.

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 `self`, 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 efforts 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

```
class Bottles
  NoMore = lambda do |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"
  end

  LastOne = lambda do |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"
  end

  Penultimate = lambda do |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"
  end

```

```
  Default = lambda do |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"
  end

  def song
    verses(99, 0)
  end

  def verses(finish, start)
    (finish).downto(start).map {|verse_number|
      verse(verse_number) }.join("\n")
  end

  def verse(number)
    verse_for(number).text
  end

  def verse_for(number)
    case number
    when 0 then Verse.new(number, &NoMore)
    when 1 then Verse.new(number, &LastOne)
    when 2 then Verse.new(number, &Penultimate)
    else        Verse.new(number, &Default)
    end
  end
end

class Verse
  attr_reader :number
  def initialize(number, &lyrics)
    @number = number
    @lyrics = lyrics
  end

  def text
    @lyrics.call self
  end
end
```

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 lambdas (lines 2, 9, 16, and 23) and saves them as constants (`NoMore`, `LastOne`, `Penultimate`, and `Default`). Notice that each lambda 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 `verse_for` and sends `text` to the result (line 40). This is the line of code that returns the correct string for a verse of the song.

Things get more interesting in `verse_for`, but before pondering that method, look ahead to the `Verse` class on line 53. `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 initialize. The `text` method is more complicated; it sends `call` to `lyrics`, passing `self` as an argument.

If you now return to `verse_for` and examine lines 45-48, 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 lambdas. The `verse_for` 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 lambda that corresponds to that number.

To summarize, sending `verse(number)` to an instance of `Bottles`

invokes `verse_for(number)`, which uses the value of `number` to select the correct lambda on which to create and return an instance of `Verse`. The `verse` method then sends `text` to the returned `Verse`, which in turn sends `call` to the lambda, passing `self` as an argument. This invokes the lambda, which may or may not actually use the argument that was passed. Regardless, executing the lambda 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 2, 9, 16 and 23 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` lambda 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 2-28), and second, it chooses the appropriate template for a specific verse number and renders that verse's lyrics (lines 39-63).

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, and 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 lambdas) 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 lambda to hold a template, 2) creating a new object to hold the lambda, and 3) invoking the lambda with `self` as an argument, the code could merely have put each of the four templates into a method and then used the case statement on lines 45-48 to invoke the correct one. The lambdas aren't needed, nor is the `Verse` class, 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 a lambda" variant come about? At this remove, 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.[2]

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

  def song
    verses(99, 0)
  end

  def verses(bottles_at_start, bottles_at_end)
    bottles_at_start.downto(bottles_at_end).map do |bottles|
      verse(bottles)
    end.join("\n")
  end

  def verse(bottles)
    Round.new(bottles).to_s
  end
```

```ruby
end

class Round
  attr_reader :bottles
  def initialize(bottles)
    @bottles = bottles
  end

  def to_s
    challenge + response
  end

  def challenge
    bottles_of_beer.capitalize + " " + on_wall + ", " +
    bottles_of_beer + ".\n"
  end

  def response
    go_to_the_store_or_take_one_down + ", " +
    bottles_of_beer + " " + on_wall + ".\n"
  end

  def bottles_of_beer
    "#{anglicized_bottle_count} #{pluralized_bottle_form} of
#{beer}"
  end

  def beer
    "beer"
  end

  def on_wall
    "on the wall"
  end

  def pluralized_bottle_form
    last_beer? ? "bottle" : "bottles"
  end

  def anglicized_bottle_count
    all_out? ? "no more" : bottles.to_s
```

```
56    end
57
58    def go_to_the_store_or_take_one_down
59      if all_out?
60        @bottles = 99
61        buy_new_beer
62      else
63        lyrics = drink_beer
64        @bottles -= 1
65        lyrics
66      end
67    end
68
69    def buy_new_beer
70      "Go to the store and buy some more"
71    end
72
73    def drink_beer
74      "Take #{it_or_one} down and pass it around"
75    end
76
77    def it_or_one
78      last_beer? ? "it" : "one"
79    end
80
81    def all_out?
82      bottles.zero?
83    end
84
85    def last_beer?
86      bottles == 1
87    end
88  end
```

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* save you money. 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. When you invoke a method instead of implementing behavior, you add a level of indirection. This indirection makes the details of what's happening harder to understand, but DRY promises that in return, your code will be easier to change.

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 42. 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 43 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:

```
def beer
  "Kool-Aid"
end
```

Or ponder some of the other method names:

```
def bottles_of_beer
def buy_new_beer
def drink_beer
def last_beer?
```

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:

```
def beverage
  "beer"
```

```
end
```

to:

```
def beverage
 "Kool-Aid"
end
```

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 that are named at the wrong level of abstraction. It's not just the names that are wrong, but the methods themselves. Many methods in this code represent the wrong abstractions.

The problem 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. *Concretely Abstract*'s heart is in the right place but 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

  def song
    verses(99, 0)
  end

  def verses(starting, ending)
    starting.downto(ending).map {|i| verse(i)}.join("\n")
  end

```

```
11    def verse(number)
12      case number
13      when 0
14        "No more bottles of beer on the wall, " +
15        "no more bottles of beer.\n" +
16        "Go to the store and buy some more, " +
17        "99 bottles of beer on the wall.\n"
18      when 1
19        "1 bottle of beer on the wall, " +
20        "1 bottle of beer.\n" +
21        "Take it down and pass it around, " +
22        "no more bottles of beer on the wall.\n"
23      when 2
24        "2 bottles of beer on the wall, " +
25        "2 bottles of beer.\n" +
26        "Take one down and pass it around, " +
27        "1 bottle of beer on the wall.\n"
28      else
29        "#{number} bottles of beer on the wall, " +
30        "#{number} bottles of beer.\n" +
31        "Take one down and pass it around, " +
32        "#{number-1} bottles of beer on the wall.\n"
33      end
34    end
35
36  end
```

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 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 that 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",[3] 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. Robert "Uncle Bob" Martin opens his book *Clean Code* by asking a number of luminaries for a definition of clean code. Their thoughtful answers 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 ...

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

— *Grady Booch*
author of [Object Oriented Analysis and Design with Applications]

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 does not predict application development time nor does it measure programmer productivity. Its desire to identify code that is *difficult to test or maintain* aims it 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, i.e., it counts 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.

The most popular tool for generating ABC scores for Ruby code is Ryan Davis's Flog. Flog is more ABC-*ish* than strictly ABC. Davis has specifically tuned it to reflect his considered opinion about what makes for good Ruby code. If you're interested in the ways in which Flog differs from classic ABC, you can find out by simply browsing the source code, but you don't have to delve into the gory details to benefit from running this metric against your own code.

Flog 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 Flog says otherwise, you should think again.

Every example in this book will eventually be run through Flog and the relative scores will be compared and discussed. Although Flog scores aren't everything, they are very definitely a useful something.

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 examine some scores for the code examples shown in this chapter.

The following table shows each solution's total lines of code

(SLOC), total Flog score, and worst scoring "bit."

Table 1.1: Flog Scores

Solution	SLOC	Flog Total	Flog Worst Bit	
Listing 1.1: Incomprehensibly Concise	19	42.5	#verse	36.2
Listing 1.2: Speculatively General	63	50.6	*lambdas*	26.5
Listing 1.3: Concretely Abstract	92	61.9	#challenge	14.4
Listing 1.4: Shameless Green	34	25.6	#verse	19.3

In most cases, the worst scoring bit is a method, but in the case of Listing 1.2: Speculatively General, the worst score is earned by the group of lambdas that are defined as constants.

The following chart makes the numbers easier to compare. Although SLOC is not related to Flog score, the values are in similar ranges so it's convenient to display everything on the same chart.

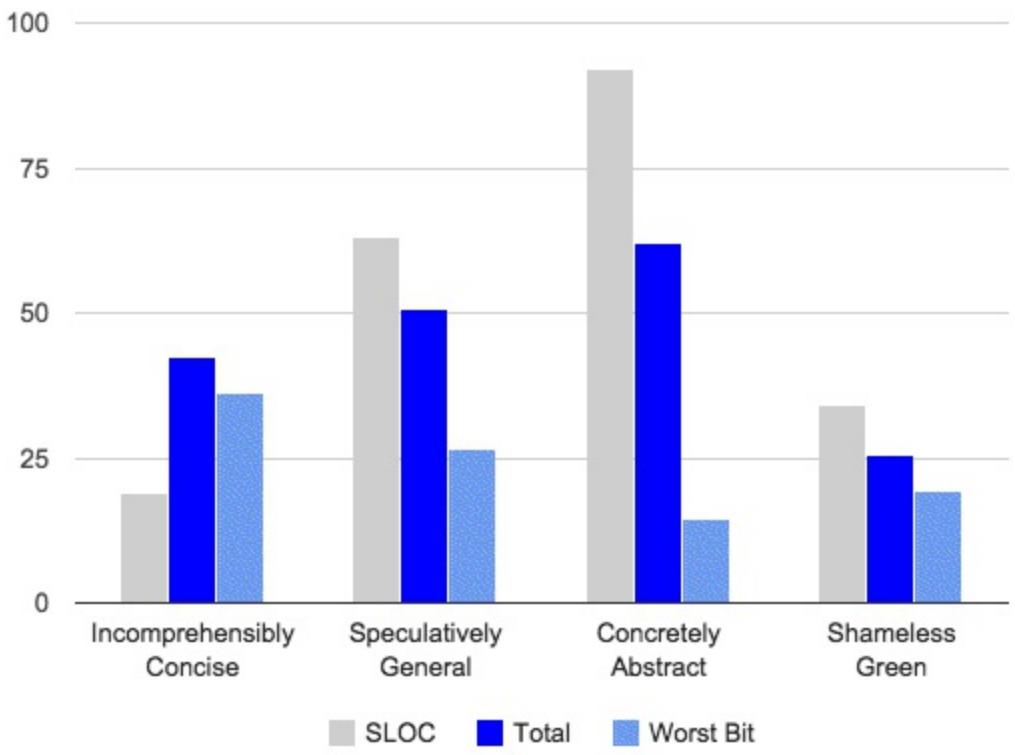

Figure 1.1: Flog Score Chart

This graph exposes a number of interesting patterns.

First, it is unsurprising that solutions with more lines of code tend to Flog to higher scores, i.e., that total Flog score generally rises in tandem with SLOC. *Shameless Green* is the notable exception—it is second lowest in SLOC but lowest in total Flog score by a considerable margin.

Next, *Concretely Abstract* scores at the extreme of every dimension. It contains the most code, Flogs to the highest total, and has the best, if you will, "Worst Bit" Flog score. The total Flog score is reasonable in light of the total lines of code, and the low Worst Bit score indicates that the methods are small and focused.

These metrics suggest that *Concretely Abstract* contains good code, but as you may recall, it does not. Metrics clearly don't

tell the whole story. The problem here is that although the code is nicely arranged, it contains names that are at the wrong level of abstraction. These names make *Concretely Abstract* expensive to change despite its orderly arrangement. The metrics overstate the quality of *Concretely Abstract* because they approve of the code structure, but they are unable to recognize the poor names.

Speculatively General is the second longest solution, and has the second highest Flog and Worst Bit scores. Its length and complexity reflect an attempt to arrange the code such that certain imagined changes will be easy, i.e. to guess the future. These guesses are unlikely to pay off and relative to the other solutions, *Speculatively General* is both longer and more complex than necessary.

The final two examples, *Incomprehensibly Concise* and *Shameless Green*, are similar in that most of their complexity is contained in a single method. In each case, the score of their worst bit is very near to their total score. This reflects the fact that both are basically procedures and that neither has attempted to identify abstractions.

Despite this similarity, if you compare their SLOC scores to their total Flog scores, you'll see that they are also very different. *Incomprehensibly Concise* has a high Flog score 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.

Overall, *Shameless Green* has the lowest total Flog score, the second lowest SLOC, and the second lowest Worst Bit score. 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 which 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 well be code that's *not* 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 become 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 Minitest:

Listing 2.1: Verse 99 Test

```
class BottlesTest < Minitest::Test
  def test_the_first_verse
    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"
    assert_equal expected, Bottles.new.verse(99)
```

```
8     end
9   end
```

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 3-6 above define the expected result and are thus part of the setup. Setup continues on line 7, where a new bottle is created via `Bottles.new`. Line 7 also sends `verse(99)`, which is the action, and then verifies the result with `assert_equal`.

Running that test produces this error:

```
1) Error:
BottlesTest#test_the_first_verse:
NameError: uninitialized constant BottlesTest::Bottles
  test/bottles_test.rb:16:in `test_the_first_verse'
```

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
end
```

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:

```
1) Error:
BottlesTest#test_the_first_verse:
NoMethodError: undefined method `verse' for #
<Bottles:0x007fde360741f0>
  test/bottles_test.rb:16:in `test_the_first_verse'
```

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

```
class Bottles
  def verse
  end
end
```

Running the test now produces this error:

```
1) Error:
BottlesTest#test_the_first_verse:
ArgumentError: wrong number of arguments (1 for 0)
  /Users/skm/Projects/books/99bottles/lib/bottles.rb:6:in
`verse'
  test/bottles_test.rb:16:in `test_the_first_verse'
```

The `verse` method requires an argument. Notice that nothing about this message requires you to add code to the `verse` method, so merely adding an argument will suffice to change

the error.

Ruby programmers by convention use `_` for the name of an unused argument. This argument is unused, at least at this moment, so `_` is a reasonable name for now.

```
class Bottles
  def verse(_)
  end
end
```

Running the test again produces the following error:

```
1) Failure:
BottlesTest#test_the_first_verse [test/bottles_test.rb:16]:
--- expected
+++ actual
@@ -1,3 +1 @@
-"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.
-"
+nil
```

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

Minitest shows the difference between expected and actual output by prefixing the expected with '-' and the actual with '+'. Therefore, you can interpret the above failure as indicating that Minitest 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 `nil`.

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

```
1 class Bottles
2   def verse(_)
3     "99 bottles of beer on the wall, " +
4     "99 bottles of beer.\n" +
5     "Take one down and pass it around, " +
6     "98 bottles of beer on the wall.\n"
7   end
8 end
```

The API says that `verse` takes an argument, but you can make this first test pass without actually using it. Therefore, the argument continues to be named `_` in line 2 above.

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 are all written, the code is incomplete.

2.3. Removing Duplication

Now that the first test passes, you must decide what best 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

```
1  def test_another_verse
2    expected = "3 bottles of beer on the wall, " +
3      "3 bottles of beer.\n" +
4      "Take one down and pass it around, " +
5      "2 bottles of beer on the wall.\n"
6    assert_equal expected, Bottles.new.verse(3)
7  end
```

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 these tests by adding a conditional that checks the value of `number` and returns the correct string:

Listing 2.4: Conditional

```
  def verse(number)
    if number == 99
      "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
      "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"
    end
  end
```

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

```
1   def verse(number)
2     if number == 99
3       n = 99
4     else
5       n = 3
6     end
7
8     "#{n} bottles of beer on the wall, " +
9     "#{n} bottles of beer.\n" +
10    "Take one down and pass it around, " +
11    "#{n - 1} bottles of beer on the wall.\n"
12  end
13 end
```

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 2-6) contains the conditional, and the second (lines 8-11) contains a template that *could* correctly generate many verses. Lines 2-6 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 8-11 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

```
  def verse(number)
    "#{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"
  end
```

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*.[4] 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, Flog score and Cyclomatic Complexity for the variants.

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

Solution	SLOC	Flog Total	Cyclomatic Complexity
Listing 2.4: Conditional	15	9.2	2
Listing 2.5: Sparse Conditional	14	7.4	2
Listing 2.6: Interpolation	7	5.1	1

As you can see, as the examples progress, they get shorter, Flog to lower scores, and decrease in Cyclomatic Complexity. These metrics indicate that each subsequent example is better than the previous, and that the general solution, Listing 2.6: Interpolation, is best of all. You must, of course, take metrics with a grain of salt, but here they cast a clear vote for Listing 2.6: Interpolation.

2.4. Understanding Transformations

The progression shown in the metrics above likely maps nicely to your intuitive sense of correctness. Intuition, however, is merely an unconscious prodding to follow an unarticulated rule, and it turns out that Robert Martin has dragged a set of these unconscious rules into the bright light of day.

In the Transformation Priority Premise, Martin defines *Transformations* as "simple operations that change the behavior of code." Not only does he describe a set of transformations that move code from more specific to more generic, he arranges these transformations in "priority" order, from simpler to more complex. He asserts that when a problem can be solved with any one of several transformations, the transformation with the highest priority is simplest and therefore best.

In the examples above, Listing 2.6: Interpolation transforms the code by interpolating a variable into a string. In Martin's terminology, this transformation is an example of *constant→scalar* ("replacing a constant with a variable or an argument"), which is fourth in priority on his list. Listing 2.4: Conditional and Listing 2.5: Sparse Conditional both transform code by adding a conditional where none previously existed. Martin calls this *unconditional→if* ("splitting the execution path") and places it sixth in priority.

Because lower numbered transformation have priority over higher numbered ones, the Transformation Priority Premise also casts a vote for Listing 2.6: Interpolation as the simpler solution. Interpolating a variable into a string is simpler than adding a new conditional.

Metrics, the Transformation Priority Premise, and common

sense converge on Listing 2.6: Interpolation as the best solution. This solution generalizes the code, which creates an abstraction. The abstraction expresses a concept, a truth if you will, about the 99 Bottles domain, i.e. that verses 99 through 3 are alike in that their numbers change in a common and predictable way.

Notice that this generalization is not one bit speculative. You haven't overreached, or made guesses about the uncertain future. Here, in this moment, there are 97 examples which follow the same straightforward rule. There's plenty of evidence to support replacing duplication with an abstraction, and doing so here simplifies the code.

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

2.5. 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 "*nn* bottles" on the wall, i.e. 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   def test_verse_2
2     expected = "2 bottles of beer on the wall, " +
3       "2 bottles of beer.\n" +
```

```
      "Take one down and pass it around, " +
      "1 bottle of beer on the wall.\n"
    assert_equal expected, Bottles.new.verse(2)
  end
```

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

```
  def verse(number)
    if number == 2
      "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
      "#{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"
    end
  end
```

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

Listing 2.9: Interpolated Conditional

```
  def verse(number)
    "#{number} bottles of beer on the wall, " +
    "#{number} bottles of beer.\n" +
    "Take one down and pass it around, " +
    "#{number-1} bottle#{'s' unless (number-1) == 1} of beer
" +
      "on the wall.\n"
  end
```

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 both by metrics and the "Transformation Priority Premise," and those things 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	Flog Total	Cyclomatic Complexity
Listing 2.8: Stark Conditional	15	10.8	2
Listing 2.9: Interpolated	9	10.9	saikuro *reports* 1

Conditional			

The table above shows that Listing 2.9: Interpolated Conditional is noticeably shorter and only fractionally more complex than the alternative. Also, in the previous section, the test of verse 3 presented an apparently identical problem, and in that case, the interpolated version *was* the best solution. In that case, it was simpler to interpolate `number` into the string than to wrap the string in a new conditional.

The metrics *and* recent history seem to be casting votes for Listing 2.9: Interpolated Conditional, but in this case they are both misleading.

There are several problems here. First, one of the metrics shown above is just plain wrong. Saikuro reports that Listing 2.9: Interpolated Conditional has a Cyclomatic Complexity of 1. However, line 5 of that example states:

```
"#{number-1} bottle#{'s' unless (number-1) == 1} of beer "
```

The `unless` keyword defines a conditional which uses the value of `number` to determine whether to append the letter "s" to "bottle." Saikuro failed to notice this keyword and so reported the Cyclomatic Complexity to be 1, which is incorrect. Both examples actually have a Cyclomatic Complexity of 2.

Therefore, the Cyclomatic Complexity scores are identical and the Flog scores virtually so. 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#{"s" unless (number-1) == 1} 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:

```
def verse
  #...
  "#{number-1} #{pluralize(number)} of beer "
  #...
end

def pluralize(number)
  "bottle#{'s' unless (number-1) == 1}"
end
```

Unfortunately, the code above just confuses the issue. The concept of pluralization is a red herring.[5] 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 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 isolating 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, i.e. 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 use the same transformation (*unconditional→if*) and have nearly identical Flog 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.

2.6. 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 it 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

```
  def test_verse_1
    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"
    assert_equal expected, Bottles.new.verse(1)
  end
```

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 `case` statement:

Listing 2.11: Verse 1 Code

```
  def verse(number)
    case number
    when 1
      "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"
    when 2
      "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
      "#{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"
    end
  end
```

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 `case`. 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 `case`?

```
if number == 1
  # something
elsif number == 2
  # something else
```

```
else
  # default
end
```

```
case number
when 1
  # something
when 2
  # something else
else
  # default
end
```

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

In contrast, use of `case` implies that every condition checks for equality against an explicit value. While it's true that the `when` clause supports more complicated operations, the form above is most common and is the one your readers will expect. Readers of `case` statements expect conditions to be fundamentally the same.

In the 99 Bottles case above, the conditions *are* fundamentally the same. Switching from `if` to `case` 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

```
  def test_verse_0
    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"
    assert_equal expected, Bottles.new.verse(0)
  end
```

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

```
  def verse(number)
    case number
    when 0
      "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"
    when 1
      "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"
    when 2
      "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
      "#{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"
    end
  end
```

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 case 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.

2.7. 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, i.e. 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

```
  def test_a_couple_verses
    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"
    assert_equal expected, Bottles.new.verses(99, 98)
  end
```

Here's one possible way to pass that test:

Listing 2.15: Verses 99 98 Literal

```
  def verses(_, _)
    "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" +
```

```
6       "\n" +
7       "98 bottles of beer on the wall, " +
8       "98 bottles of beer.\n" +
9       "Take one down and pass it around, " +
10      "97 bottles of beer on the wall.\n"
11    end
```

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 case 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 `else` clause of the `case` statement of `verse` (repeated below).

Listing 2.16: Verse Case Statement Else Branch

```
  def verse(number)
    case number
      # ...
    else
      "#{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"
    end
  end
```

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

```
  def verses(_, _)
    verse(99) + "\n" + verse(98)
  end
```

The story this code tells is that `verses` are made up of `verse` s (sorry), 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 starting to ending number, 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[6] 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

```
1   def test_a_few_verses
2     expected = "2 bottles of beer on the wall, " +
3       "2 bottles of beer.\n" +
4       "Take one down and pass it around, " +
5       "1 bottle of beer on the wall.\n" +
6       "\n" +
7       "1 bottle of beer on the wall, " +
8       "1 bottle of beer.\n" +
9       "Take it down and pass it around, " +
10      "no more bottles of beer on the wall.\n" +
11      "\n" +
12      "No more bottles of beer on the wall, " +
13      "no more bottles of beer.\n" +
14      "Go to the store and buy some more, " +
15      "99 bottles of beer on the wall.\n"
16    assert_equal expected, Bottles.new.verses(2, 0)
17  end
```

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

```
  def verses(starting, ending)
    if starting == 99
      verse(99) + "\n" + verse(98)
    else
      verse(2) + "\n" + verse(1) + "\n" + verse(0)
    end
  end
```

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

Listing 2.20: Verses Within a Range

```
  def verses(starting, ending)
    starting.downto(ending).collect {|i| verse(i)}.join("\n")
  end
```

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 (i.e. you know all of the possible verse ranges), 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.

2.8. Choosing Names

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

- `verse(n)`
- `verses(starting, ending)    # 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

```
1   def song
2     verses(99, 0)
3   end
```

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.

2.9. 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
  def song
    verses(99, 0)
  end
```

```
5
6     def verses(starting, ending)
7       starting.downto(ending).collect {|i|
verse(i)}.join("\n")
8     end
9
10    def verse(number)
11      case number
12      when 0
13        "No more bottles of beer on the wall, " +
14        "no more bottles of beer.\n" +
15        "Go to the store and buy some more, " +
16        "99 bottles of beer on the wall.\n"
17      when 1
18        "1 bottle of beer on the wall, " +
19        "1 bottle of beer.\n" +
20        "Take it down and pass it around, " +
21        "no more bottles of beer on the wall.\n"
22      when 2
23        "2 bottles of beer on the wall, " +
24        "2 bottles of beer.\n" +
25        "Take one down and pass it around, " +
26        "1 bottle of beer on the wall.\n"
27      else
28        "#{number} bottles of beer on the wall, " +
29        "#{number} bottles of beer.\n" +
30        "Take one down and pass it around, " +
31        "#{number-1} bottles of beer on the wall.\n"
32      end
33    end
34  end
```

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.

2.10. 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 sheepishly apologize for not doing so. Even those who don't test seem to believe they ought to.

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

```
class BottlesTest < Minitest::Test
  def test_the_first_verse
    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"
    assert_equal expected, Bottles.new.verse(99)
  end

  def test_another_verse
    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"
    assert_equal expected, Bottles.new.verse(3)
  end

  def test_verse_2
    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"
    assert_equal expected, Bottles.new.verse(2)
  end

  def test_verse_1
    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"
    assert_equal expected, Bottles.new.verse(1)
  end

  def test_verse_0
    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"
    assert_equal expected, Bottles.new.verse(0)
  end

  def test_a_couple_verses
    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"
    assert_equal expected, Bottles.new.verses(99, 98)
  end

  def test_a_few_verses
    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"
    assert_equal expected, Bottles.new.verses(2, 0)
```

```
71    end
```

2.11. 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

```
1   def test_the_whole_song
2     bottles = Bottles.new
3     assert_equal bottles.verses(99, 0), bottles.song
4   end
```

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 of tests that 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
  def song
    verses(99, 0)
  end

  def verses(starting, ending)
    if starting == 99 && ending == 98
      "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"
    elsif starting == 2
      verse(2) + "\n" + verse(1) + "\n" + verse(0)
    else
      "ok"
    end
  end

  def verse(number)
    case number
    when 0
      "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"
    when 1
      "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"
    when 2
      "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"
    when 3
      "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"
    else
      "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"
    end
```

```
52    end
53  end
```

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

```
1   def test_the_whole_song
2     bottles = Bottles.new
3     expected = 99.downto(0).collect {|i|
4       bottles.verse(i)
5     }.join("\n")
6     assert_equal expected, bottles.song
7   end
```

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

```
99.downto(0).collect {|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

```
  def test_the_whole_song
    expected = <<-SONG
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.
```

```
10  Take one down and pass it around, 96 bottles of beer on the
wall.
11
12  # ...
13
14  4 bottles of beer on the wall, 4 bottles of beer.
15  Take one down and pass it around, 3 bottles of beer on the
wall.
16
17  3 bottles of beer on the wall, 3 bottles of beer.
18  Take one down and pass it around, 2 bottles of beer on the
wall.
19
20  2 bottles of beer on the wall, 2 bottles of beer.
21  Take one down and pass it around, 1 bottle of beer on the
wall.
22
23  1 bottle of beer on the wall, 1 bottle of beer.
24  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      SONG
29      assert_equal expected, Bottles.new.song
30    end
```

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.

2.12. 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 then 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
  def song
    verses(99, 0)
  end

  def verses(starting, ending)
    starting.downto(ending).collect {|i|
verse(i)}.join("\n")
  end

  def verse(number)
    case number
    when 0
      "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"
    when 1
      "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"
    when 2
      "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
      "#{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"
    end
  end
end
```

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

2.13. 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 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

```
1  class Bottles
2    def song
3      verses(99, 0)
4    end
5
6    def verses(starting, ending)
```

```
    starting.downto(ending).collect {|i|
verse(i)}.join("\n")
  end

  def verse(number)
    case number
    when 0
      "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"
    when 1
      "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"
    when 2
      "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
      "#{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"
    end
  end
end
```

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 nn 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 case 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

```
  def verse(number)
    case number
    when 0
      "No more bottles of beer on the wall, " +
      # ...
    when 1
      "1 bottle of beer on the wall, " +
      # ...
    when 2
      "2 bottles of beer on the wall, " +
      # ...
    when 6
      "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"
    when 7
      "7 bottles of beer on the wall, " +
      "7 bottles of beer.\n" +
      "Take one down and pass it around, " +
```

```
21        "1 six-pack of beer on the wall.\n"
22      else
23        "#{number} bottles of beer on the wall, " +
24        # ...
25      end
26    end
27  end
```

The `verse` case 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 then 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.

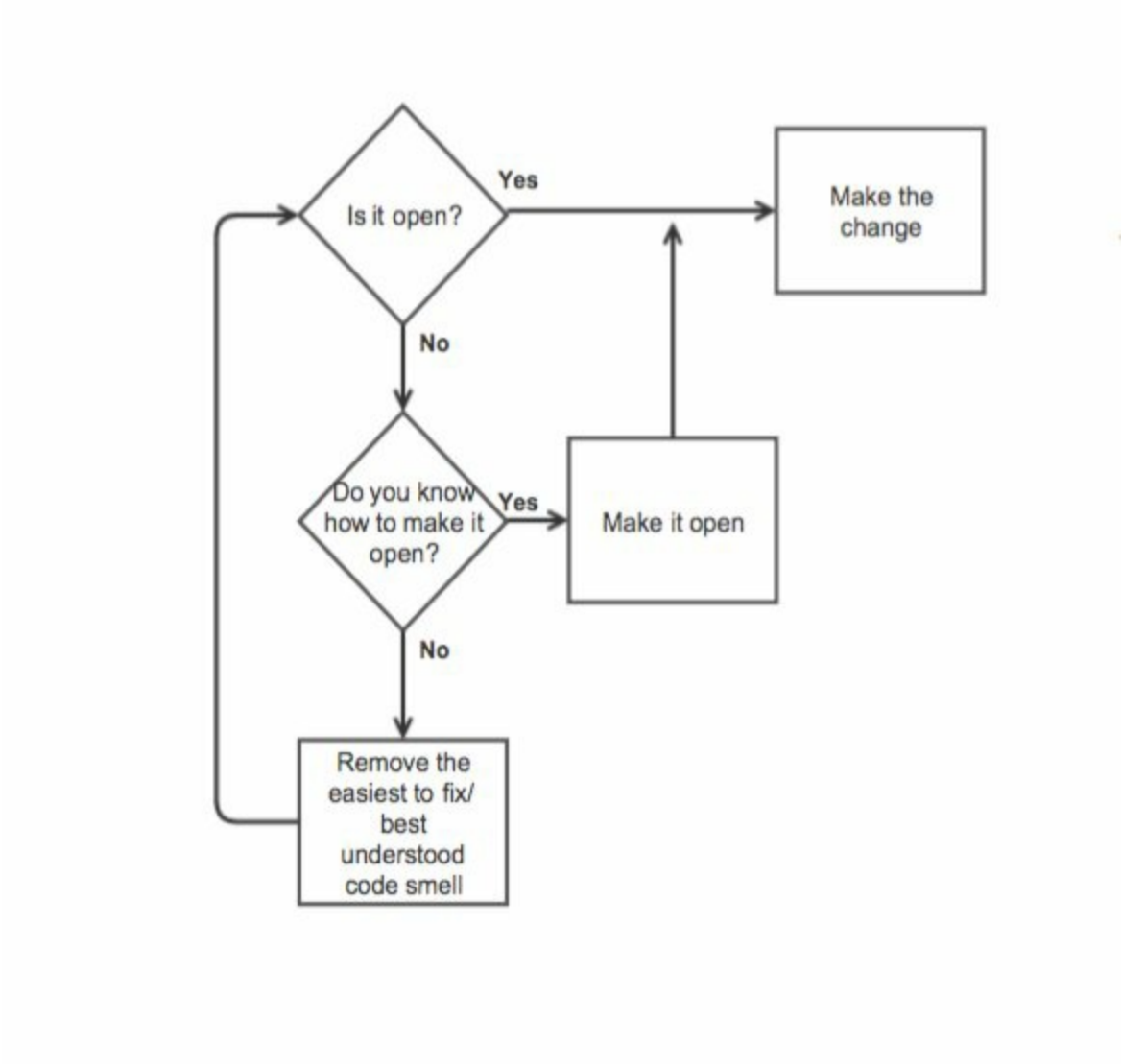

Figure 3.1: Open Closed Flowchart

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 it 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. Also, if you prefer your examples in Ruby, you may be interested in Jay Fields' version of the book.

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

```
  def verse(number)
    case number
    when 0
      "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"
    when 1
      "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"
    when 2
      "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
      "#{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"
    end
  end
```

This method contains a `case` 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 `case` statement of the `verse` method.

The `case` 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 `case` 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` `case` 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 `case` statement:

Listing 3.4: Verse Method Conditional

```
    case number
    when 0
      "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"
    when 1
      "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"
    when 2
      "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
      "#{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"
    end
```

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 `else` 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 else 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.[7] 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 `else` 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 Else Case

```
    when 2
      "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
      "#{number} bottles of beer on the wall, " +
      "#{number} bottles of beer.\n" +
```

```
 9         "Take one down and pass it around, " +
10         "#{number-1} bottles of beer on the wall.\n"
11       end
```

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. Not only that, but many times the most difficult bit of a larger problem is the riskiest, and requires the most thought. It's no wonder that many programmers gravitate towards starting a problem at its most confusing part. 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 rightwards, 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

```
    when 2
      "#{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"
    else
      "#{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"
    end
```

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

```
    when 2
      # ...
      "#{number-1} bottle of beer on the wall.\n"
    else
      # ...
      "#{number-1} bottles of beer on the wall.\n"
    end
```

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 and to do so using language of the domain.

One way to identify the category is to imagine the concrete examples as rows and columns in a spreadsheet.[8] 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; 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. [9] For example, if the song were about wine, the wine might come in a carafe. Juice sometimes comes in small boxes. Soft drinks 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

```
  def container(number)
    if number == 1
      "bottle"
    else
      "bottles"
    end
  end
```

This method must be invoked from both branches of the `verse`

`case` statement. Here the code:

```
"#{number-1} #{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 with a 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 `else` 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

```
  def container
  end
```

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

```
    when 2
      # ...
      "#{number-1} bottle of beer on the wall.\n"
    else
      # ...
      "#{number-1} bottles of beer on the wall.\n"
    end
```

The current `container` method returns `nil`. It will eventually be called from two places. The `2` case wants the return to be "bottle," and the `else` 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

```
1   def container
2     "bottles"
3   end
```

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 `else` branch to send the message in place of the word "bottles," as on line 8 below:

Listing 3.12: Sparse Container Used in Else Branch

```
1   def verse(number)
2     # ...
3     when 2
4       # ...
5       "#{number-1} bottle of beer on the wall.\n"
6     else
7       # ...
8       "#{number-1} #{container} of beer on the wall.\n"
9     end
10  end
11
12  def container
13    "bottles"
14  end
```

So far, so good, but consider the next step. To be usable in both

the `2` and `else` 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.

Just as `container` doesn't currently *take* an argument (line 12 above), its invoker doesn't currently *send* one (line 8 above). Now you face a conundrum. The goal is to make changes on one line at a time, but this situation seems to require that you change both the sender and the receiver simultaneously.

To illustrate the problem, consider what happens if you make either of these changes without the other. You could add the argument to the method definition first, like so:

```
def container(number)
```

In this case, the message send fails because it doesn't yet send the argument:

```
ArgumentError: wrong number of arguments (0 for 1)
```

If you reverse the order of the changes, and send the argument first, as so:

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

Then the opposite failure occurs, i.e. an argument is passed where none is expected.

```
ArgumentError: wrong number of arguments (1 for 0)
```

This problem, needing to add a required argument, arises

regularly in the real world. But instead of one sender and one receiver, as in this case, real applications might have 10 or 100 or 1000 senders. It might be impossible to fix everything at once, so it's handy to know the technique for working around this problem in an incremental manner.

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, gradually switching over a small number of pieces at a time. This type of refactoring can be done alongside other development work so as not to affect the release schedule.

Many of your colleagues, including your customers, will applaud this strategy and your willingness to keep the code releasable. But when you are prevented from fixing all of the senders at once, you must do something to allow some to pass the new argument while others remain unchanged. The trick here, as you may already have guessed, is to begin by adding an *optional* argument that supplies its own default, as shown below:

Listing 3.13: Container With Defaulted Argument

```
1   def container(number=:FIXME)
2     "bottles"
3   end
```

The above code takes an argument named `number`, which it defaults to the symbol `:FIXME`. You may have expected the default to be `nil`, or at the very least, a numeric value, but in this case it makes sense to set it to something that's usefully wrong. This default is a temporary shim whose purpose is to enable a step-by-step refactoring. Once the refactor is

complete, the default *should* be removed. Setting it to a value like :FIXME will help you remember to do this clean-up.

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," i.e. change the receiver, or
- alter the `else` branch to add the `number` argument to `container` message, i.e. 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 `else` 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

```
1  def container(number=:FIXME)
2    if number == 1
3      "bottle"
4    else
5      "bottles"
6    end
7  end
```

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:

```
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 `else` branch of the `case` statement in `verse`), and that as yet no argument is being passed. This means that the `number` argument in `container` gets set to `:FIXME`, 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 8 below:

Listing 3.15: Passing an Argument to Container

```
  def verse(number)
    # ...
    when 2
      # ...
      "#{number-1} bottle of beer on the wall.\n"
    else
      # ...
      "#{number-1} #{container(number-1)} of beer on the
wall.\n"
    end
  end

  def container(number=:FIXME)
    if number == 1
      "bottle"
    else
      "bottles"
    end
  end
```

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 `:FIXME` 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 below:

Listing 3.16: 2 and Else Cases Identical, Number Default Exists

```
  def verse(number)
    # ...
    when 2
```

```
4         "#{number} bottles of beer on the wall, " +
5         "#{number} bottles of beer.\n" +
6         "Take one down and pass it around, " +
7         "#{number-1} #{container(number-1)} of beer on the
wall.\n"
8       else
9         "#{number} bottles of beer on the wall, " +
10        "#{number} bottles of beer.\n" +
11        "Take one down and pass it around, " +
12        "#{number-1} #{container(number-1)} of beer on the
wall.\n"
13      end
14    end
15
16    def container(number=:FIXME)
17      if number == 1
18        "bottle"
19      else
20        "bottles"
21      end
22    end
```

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

Two tasks remain to complete this entire horizontal refactoring. First, as all senders of `container` now pass `number`, the `:FIXME` default has served its purpose and can be removed. Next, the `2` case is now obsolete, and so it also can be deleted. The following example shows the resulting, complete code:

Listing 3.17: 2 Subsumed Into Else Case, Number Default Removed

```
1   def verse(number)
2     case number
```

```
    when 0
      "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"
    when 1
      "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"
    else
      "#{number} bottles of beer on the wall, " +
      "#{number} bottles of beer.\n" +
      "Take one down and pass it around, " +
      "#{number-1} #{container(number-1)} of beer on the
wall.\n"
    end
  end

  def container(number)
    if number == 1
      "bottle"
    else
      "bottles"
    end
  end
end
```

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

1. identified `verse 2` and `else` 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` `else` case to send `container` in place of "bottles"
10. changed `container` to take `number` argument with default `:FIXME`
11. added conditional logic to `container` to return "bottle" or "bottles" based on `number`
12. changed `verse` `else` case to pass `number-1` to `container`
13. changed `verse` `2` case to send `container(number-1)` in place of "bottle"
14. deleted `verse` `2` case
15. deleted `container` `:FIXME` `number` argument default

Of these 15 steps, 12 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 peskier 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 `else` 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 `else` 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 Case Statement

```
1   def verse(number)
2     case number
3     when 0
4       "No more bottles of beer on the wall, " +
5       "no more bottles of beer.\n" +
6       "Go to the store and buy some more, " +
7       "99 bottles of beer on the wall.\n"
8     when 1
9       "1 bottle of beer on the wall, " +
10      "1 bottle of beer.\n" +
11      "Take it down and pass it around, " +
```

```
12          "no more bottles of beer on the wall.\n"
13        else
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} #{container(number-1)} of beer on the
wall.\n"
18        end
19      end
```

The `1` case differs from the `else` 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 `else` 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 `else` case than each other.

Of these three verse templates, the `1` and `else` 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 Else, 1st Phrases Differ

```
1       when 1
2         "1 bottle of beer on the wall, " +
3         # ...
4       else
5         "#{number} bottles of beer on the wall, " +
6         # ...
7       end
```

Just as with the `2` and `else` 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 Else, 1st Phrases in Progress

```
    when 1
      "#{number} bottle of beer on the wall, " +
      # ...
    else
      "#{number} bottles of beer on the wall, " +
      # ...
    end
```

A similar change was made in the previous chapter, where the hard-coded `"2"` was replaced by `#{number}` when combining the `2` and `else` 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 Else, 1st Phrases Identical

```
    when 1
```

```
      "#{number} #{container(number)} of beer on the wall, "
+
      # ...
    else
      "#{number} #{container(number)} of beer on the wall, "
+
      # ...
    end
```

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 Else, 2nd Phrases Differ

```
    when 1
      "#{number} #{container(number)} of beer on the wall, "
+
      "1 bottle of beer.\n" +
      # ...
    else
      "#{number} #{container(number)} of beer on the wall, "
+
      "#{number} bottles of beer.\n" +
      # ...
    end
```

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

Listing 4.6: 1 and Else, 2nd Phrases Identical

```
    when 1
      "#{number} #{container(number)} of beer on the wall, "
+
      "#{number} #{container(number)} of beer.\n" +
      # ...
    else
```

```
6         "#{number} #{container(number)} of beer on the wall, "
+
7         "#{number} #{container(number)} of beer.\n" +
8         # ...
9       end
```

After the above changes, the first two phrases of the `1` and `else` 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 Else, 3rd Phrases Differ

```
1       when 1
2         # ...
3         "Take it down and pass it around, " +
4         # ...
5       else
6         # ...
7         "Take one down and pass it around, " +
8         # ...
9       end
```

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 `it_or_one` 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 `thing_drunk`.

Although `pronoun` *does* feel a bit too general, `thing_drunk` 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, [10] 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

```
  def pronoun
  end
```

Alter `pronoun` to return "one," i.e. the value from the `else` branch.

Listing 4.9: Sparse Pronoun Method

```
  def pronoun
    "one"
  end
```

Alter the `else` branch to send `pronoun` in place of "one":

Listing 4.10: Send Pronoun in Else Branch

```
    when 1
      # ...
      "Take it down and pass it around, " +
      # ...
    else
      # ...
      "Take #{pronoun} down and pass it around, " +
      # ...
    end
```

Add a defaulted argument to `pronoun`.

Listing 4.11: Pronoun With Defaulted Argument

```
  def pronoun(number=:FIXME)
    "one"
  end
```

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

Listing 4.12: Pronoun With Conditional

```
  def pronoun(number=:FIXME)
    if number == 1
      "it"
    else
      "one"
    end
  end
```

Alter the `else` case to pass the `number` argument to `pronoun` (line 9).

Listing 4.13: Passing an Argument to Pronoun

```
 def verse(number)
   # ...
   when 1
     # ...
     "Take it down and pass it around, " +
     # ...
   else
     # ...
     "Take #{pronoun(number)} down and pass it around, " +
     # ...
   end
 end
```

Alter the `1` case to send `pronoun(number)` in place of "it" (line 5).

Listing 4.14: 1 and Else Cases Send Pronoun

```
 def verse(number)
   # ...
   when 1
     # ...
     "Take #{pronoun(number)} down and pass it around, " +
```

```
      # ...
    else
      # ...
      "Take #{pronoun(number)} down and pass it around, " +
      # ...
    end
  end
```

Alter the `pronoun` method to remove the `:FIXME` default:

Listing 4.15: Final Pronoun Method

```
def pronoun(number)
  if number == 1
    "it"
  else
    "one"
  end
end
```

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. The `container` abstraction replaced the "bottle" and "bottles" strings, and the `pronoun` abstraction replaced "it" and "one."

This completes the addition of the `pronoun` method, and makes phrase three of the `1` and `else` 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 `else` cases:

Listing 4.16: 1 and Else, 4th Phrases Differ

```
      when 1
        # ...
        "no more bottles of beer on the wall.\n"
      else
        # ...
        "#{number-1} #{container(number-1)} of beer on the
wall.\n"
      end
```

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} #{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 `else` case.

If `"bottles"` is actually the same as `container(number-1)`, then that part (at least logically), is resolved. This leaves:

```
"no more"
```

which 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, i.e. `99` becomes `"99"`, `50`, `"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. 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 setting an argument to a default. In each case, the symbol `:FIXME` was used as that default. `:FIXME` is handy because the name itself reminds you of its temporary nature, and acts as a reminder to remove it at the end of the refactoring. Helpful as `:FIXME` 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 just 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 4.17: Initial Quantity Method

```
1    def quantity
2    end
```

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 `else` branch first. But in this case, the `else` branch contains interpolated code that references `number`. Therefore, you can't copy the `else` 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 4.18: Quantity Method First Return

```
  def quantity
    "no more"
  end
```

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

Listing 4.19: Quantity Message First Send

```
    when 1
      # ...
      "#{quantity} bottles of beer on the wall.\n"
    else
```

Add the normal `:FIXME` default to the `number` argument in `quantity`:

Listing 4.20: Number Argument Defaulted to FIXME

```
  def quantity(number=:FIXME)
    "no more"
  end
```

If you're concerned about the `:FIXME` default above, your Spidey-sense[11] 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 `else` 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 4.21: Quantity Message With Conditional

```
  def quantity(number=:FIXME)
    if number == 0
      "no more"
    else
      number
    end
  end
```

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, FIXME bottles of beer on the
wall.
```

Have a look at the `case` statement below. Examine line 3 and try to explain what went wrong.

Listing 4.22: Using the Number Default From the 1 Case

```
1       when 1
2         # ...
3         "#{quantity} bottles of beer on the wall.\n"
4       else
5         # ...
6         "#{number-1} #{container(number-1)} of beer on the
wall.\n"
```

This failure occurs because line 3 above calls `quantity` without passing an argument. Upon invocation, the `quantity` method sets `number` to `:FIXME`, which sends execution to the false branch of its conditional. The false branch obediently returns `number`, which unfortunately still contains `:FIXME`. This result then gets interpolated back into the verse. Thus, `"FIXME bottles of beer"`.

The reason the `:FIXME` 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 4.23: Number Argument Defaulted to FIXME Reprise

```
  def quantity(number=:FIXME)
    "no more"
  end
```

An obviously wrong and temporary value like `:FIXME` can be a handy default, but you can only use this technique if you begin these refactorings by returning the difference from the `else` branch. While it's perfectly acceptable to begin by returning "no more" (the non-else difference), doing so means that you have to think more carefully about the default. So, use a default like `:FIXME` thoughtfully.

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

Listing 4.24: Number Argument Defaults to 0

```
  def quantity(number=0)
    "no more"
  end
```

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

Listing 4.25: Default Takes the True Branch

```
  def quantity(number=0)
    if number == 0
      "no more"
    else
      number
    end
  end
```

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 5 below has been changed to pass `number-1` to `quantity`:

Listing 4.26: 1 Case Passes an Argument

```
1    def verse(number)
2      # ...
3      when 1
4        # ...
5        "#{quantity(number-1)} bottles of beer on the wall.\n"
6      else
7        # ...
8        "#{number-1} #{container(number-1)} of beer on the
wall.\n"
9      end
10   end
```

The tests still pass. The next step is to use `quantity` in the `else` case, as shown on line 8 below:

Listing 4.27: Else Case Sends Quantity

```
1    def verse(number)
2      # ...
3      when 1
4        # ...
5        "#{quantity(number-1)} bottles of beer on the wall.\n"
6      else
7        # ...
8        "#{quantity(number-1)} #{container(number-1)} of beer
on the wall.\n"
9      end
10   end
```

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

Listing 4.28: Quantity Method

```
1    def quantity(number)
2      if number == 0
3        "no more"
4      else
5        number
6      end
7    end
```

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

Listing 4.29: 1 and Else Cases More Alike

```
1    def verse(number)
2        # ...
3      when 1
4        "#{quantity(number-1)} bottles of beer on the wall.\n"
5      else
6        "#{quantity(number-1)} #{container(number-1)} of beer
on the wall.\n"
7      end
8    end
```

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

Listing 4.30: 1 and Else Cases Identical

```
1     def verse(number)
2         # ...
3       when 1
4         "#{number} #{container(number)} of beer on the wall, "
+
```

```
5         "#{number} #{container(number)} of beer.\n" +
6         "Take #{pronoun(number)} down and pass it around, " +
7         "#{quantity(number-1)} #{container(number-1)} of beer
on the wall.\n"
8       else
9         "#{number} #{container(number)} of beer on the wall, "
+
10        "#{number} #{container(number)} of beer.\n" +
11        "Take #{pronoun(number)} down and pass it around, " +
12        "#{quantity(number-1)} #{container(number-1)} of beer
on the wall.\n"
13      end
14    end
```

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 `else` branch of the case statement, but the `quantity` method differs in that it initially returns the value from the `1`, or non-`else` 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 `else` branch is implemented first, `:FIXME` can always be used for the default. This not only saves you from having to

figure out the right value, it also serves as a reminder to remove this temporary default later. If the non-`else` 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-`else` branch first places a slightly greater burden on you. You have to use a specific, real, value for the default, and then must 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 4.31: Three Abstracted Concepts

```
  def quantity(number)
    if number == 0
      "no more"
    else
      number
    end
  end

  def pronoun(number)
    if number == 1
      "it"
    else
      "one"
    end
  end

  def container(number)
    if number == 1
      "bottle"
    else
      "bottles"
    end
  end
```

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*--they're 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 case statement. One case (the `1` case) has

been removed, leaving the `0` and `else` cases still to be resolved. Here's a reminder of the current state of the code:

Listing 4.32: 0 and Else Cases Differ

```
1   def verse(number)
2     case number
3     when 0
4       "No more bottles of beer on the wall, " +
5       "no more bottles of beer.\n" +
6       "Go to the store and buy some more, " +
7       "99 bottles of beer on the wall.\n"
8     else
9       "#{number} #{container(number)} of beer on the wall, "
+
10      "#{number} #{container(number)} of beer.\n" +
11      "Take #{pronoun(number)} down and pass it around, " +
12      "#{quantity(number-1)} #{container(number-1)} of beer
on the wall.\n"
13    end
14  end
```

Begin this next refactoring by focusing on lines 4 and 9 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 9 in the `else` case. This method can be substituted on line 4 for the word `"bottles"`, and needs no further consideration.

The remaining difference is at the very beginning of lines 4 and 9, 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 4 to send `"#{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 `case` statement. Replace `"No more"` with `"#{quantity(number)}"`, and capitalize the result, as on line 4 below:

Listing 4.33: Quantity Capitalized in 0 Case

```
1   def verse(number)
2     case number
3     when 0
4       "#{quantity(number).capitalize} bottles of beer on the
wall, " +
5       # ...
6     else
7       "#{number} #{container(number)} of beer on the wall, "
```

```
+
 8         # ...
 9       end
10     end
```

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, it now seems reasonable to make a similar one in the `else` case, shown on line 7 below:

Listing 4.34: Quantity Capitalized in Else Case

```
 1     def verse(number)
 2       case number
 3       when 0
 4         "#{quantity(number).capitalize} bottles of beer on the
wall, " +
 5         # ...
 6       else
 7         "#{quantity(number).capitalize} #{container(number)}
of beer on the wall, " +
 8         # ...
 9       end
10     end
```

Despite seeming reasonable, the change makes the tests fail with:

```
NoMethodError: undefined method `capitalize' for 99:Fixnum
```

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 4.35: Quantity Method Reprise

```
  def quantity(number)
    if number == 0
      "no more"
    else
      number
    end
  end
```

The most recent change invokes `quantity` with an 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 indeed an instance of `Fixnum`. `String` understands `capitalize`, but `Fixnum` does not; 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 sending `capitalize`. Line 7 below inserts `to_s` into the method chain:

Listing 4.36: Else Branch Converts Result

```
  def verse(number)
    case number
    when 0
      "#{quantity(number).capitalize} bottles of beer on the
wall, " +
      # ...
    else
      "#{quantity(number).to_s.capitalize} #
{container(number)} of beer on the wall, " +
      # ...
    end
```

```
10    end
```

The above change fixes the failing test, but introduces a new difference between the phrases. To remove this difference, you must also insert `to_s` into the `0` case, as on line 4 below:

Listing 4.37: Both Branches Convert Result

```
1     def verse(number)
2       case number
3       when 0
4         "#{quantity(number).to_s.capitalize} bottles of beer
on the wall, " +
5         # ...
6       else
7         "#{quantity(number).to_s.capitalize} #
{container(number)} of beer on the wall, " +
8         # ...
9       end
10    end
```

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 understand `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 understands `capitalize`. The `verse` method knows that instances of `String` *do* understand `capitalize`. It knows that any object can be converted to a string by sending `to_s`. Therefore, it knows that it can convert any object into something that understands `capitalize` by sending it the `to_s` 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 Ruby objects understand `to_s`, 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 `to_s`, but adds the overhead of sending `to_s` 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 4.38: Quantity Obeys Liskov

```
1  def quantity(number)
2    if number == 0
3      "no more"
4    else
5      number.to_s
6    end
7  end
```

Now that `quantity` always returns a "capitalizable," you can pretend that the `to_s` dependency never existed in verse, which returns the code to the state shown here:

Listing 4.39: Verse Trusts Quantity

```
1  def verse(number)
2    case number
3    when 0
4      "#{quantity(number).capitalize} bottles of beer on the
```

```
wall, " +
5         # ...
6       else
7         "#{quantity(number).capitalize} #{container(number)} of
beer on the wall, " +
8         # ...
9     end
```

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 4.40: 0 Case Sends Container

```
1     def verse(number)
2       case number
3       when 0
4         "#{quantity(number).capitalize} #{container(number)}
of beer on the wall, " +
5         # ...
6       else
7         "#{quantity(number).capitalize} #{container(number)}
of beer on the wall, " +
8         # ...
9       end
10    end
```

After that change, the first phrases of the `0` and `else` 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 `else` cases are identical, so it's time to examine the second. It's repeated below:

Listing 4.41: 0 and Else, 2nd Phrases Differ

```
  def verse(number)
    case number
    when 0
      # ...
      "no more bottles of beer.\n" +
      # ...
    else
      # ...
      "#{number} #{container(number)} of beer.\n" +
      # ...
    end
  end
```

The above differences reflect the `quantity` and `container` concepts, which have long since been identified. Resolve them by changing the code as follows:

Listing 4.42: 2nd Phrases Send Quantity and Container

```
  def verse(number)
    case number
    when 0
      # ...
      "#{quantity(number)} #{container(number)} of beer.\n"
+
      # ...
```

```
7       else
8         # ...
9         "#{quantity(number)} #{container(number)} of beer.\n"
+
10      end
11    end
```

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 4.43: Phrases 1 and 2 Are Identical

```
1     def verse(number)
2       case number
3       when 0
4         "#{quantity(number).capitalize} #{container(number)}
of beer on the wall, " +
5         "#{quantity(number)} #{container(number)} of beer.\n"
+
6         "Go to the store and buy some more, " +
7         "99 bottles of beer on the wall.\n"
8       else
9         "#{quantity(number).capitalize} #{container(number)}
of beer on the wall, " +
10        "#{quantity(number)} #{container(number)} of beer.\n"
+
11        "Take #{pronoun(number)} down and pass it around, " +
12        "#{quantity(number-1)} #{container(number-1)} of beer
on the wall.\n"
13      end
14    end
```

To locate the next difference, it can again be helpful to scan the verse from the end. Both variants end with "of beer on the wall." On line 7, phrase 4 of case `0` begins with `"99"` followed by `"bottles"`. These seem to go with `quantity` and `container` on line 12. Ignore this fourth phrase for now and turn your thoughts to phrase 3, isolated below:

Listing 4.44: 0 and Else, 3rd Phrases Differ

```
  def verse(number)
    case number
    when 0
      # ...
      "Go to the store and buy some more, " +
      # ...
    else
      # ...
      "Take #{pronoun(number)} down and pass it around, " +
      # ...
    end
  end
```

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 `else` 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 phrase 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 case statement. Once this becomes apparent, it makes sense to begin extracting the method in a single step, as shown below:

Listing 4.45: Leap Into Action

```
  def action(number)
    if number == 0
      "Go to the store and buy some more"
    else
      "Take #{pronoun(number)} down and pass it around"
    end
  end
```

This new `action` method contains a conditional that reflects the case statement from whence it came. Just as the original `case` 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 `else` branches of the `case` 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 4.46: 3rd Phrases Send Action

```
  def verse(number)
    case number
    when 0
      # ...
      "#{action(number)}, " +
      # ...
    else
      # ...
      "#{action(number)}, " +
      # ...
    end
```

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 case statement turn into a single conditional. Now that you know how to make this change using small steps, and have seen this pattern, 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 case 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 `else` cases, on lines 7 and 12 below. Here's the current state of the code:

Listing 4.47: Phrases 1, 2, and 3 Are Identical

```
 1    def verse(number)
 2      case number
 3      when 0
 4        "#{quantity(number).capitalize} #{container(number)}
of beer on the wall, " +
 5        "#{quantity(number)} #{container(number)} of beer.\n"
+
 6        "#{action(number)}, " +
 7        "99 bottles of beer on the wall.\n"
 8      else
 9        "#{quantity(number).capitalize} #{container(number)}
of beer on the wall, " +
10        "#{quantity(number)} #{container(number)} of beer.\n"
+
11        "#{action(number)}, " +
12        "#{quantity(number-1)} #{container(number-1)} of beer
on the wall.\n"
```

```
13      end
```

The trailing `"of beer on the wall"` in the lines above is a sameness, and the word `"bottles"` in line 7 is an example of the `container` abstraction, which is already used in this place in line 12. If you ignore these for now, the remaining difference is that:

```
"99"
```

seems to be set against:

```
"#{quantity(number-1)}"
```

This may lead you to conclude that `"99"` is a third example of the `quantity` abstraction. If so, then you should alter `quantity` to sometimes return `"99"`. The resulting method would look like this:

Listing 4.48: Quantity Overreaches to Handle 99

```
1  def quantity(number)
2    case number
3    when -1
4      "99"
5    when 0
6      "no more"
7    else
8      number.to_s
9    end
10 end
```

If you made the alteration shown above, and then replaced `"99"` with `"#{container(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 4.49: Consistent Abstractions

```
  def action(number)
    if number == 0
      "Go to the store and buy some more"
    else
      "Take #{pronoun(number)} down and pass it around"
    end
  end

  def quantity(number)
    if number == 0
      "no more"
    else
      number.to_s
    end
  end

  def pronoun(number)
    if number == 1
      "it"
    else
      "one"
    end
  end

  def container(number)
    if number == 1
      "bottle"
    else
      "bottles"
    end
```

```
31    end
```

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 4.50: Song and Verses Reprise

```
1  def song
2    verses(99, 0)
3  end
4
```

```
5     def verses(starting, ending)
6       starting.downto(ending).collect {|i| verse(i)}.join("\n")
7     end
```

Line 2 above encodes the knowledge that the overall song starts on verse 99 and counts down to 0. Line 6 decrements 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, repeated below:

Listing 4.51: Case 0 Handles Restart

```
1     def verse(number)
2       case number
3       when 0
4         # ...
5         "99 bottles of beer on the wall.\n"
6       else
7         # ...
8         "#{quantity(number-1)} #{container(number-1)} of beer
on the wall.\n"
9       end
10    end
```

Line 5 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.[12]

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 5 above look more like line 8 (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 4.52: 99 Is a Quantity

```
  def verse(number)
    case number
    when 0
      # ...
      "#{quantity(99)} bottles of beer on the wall.\n"
    else
      # ...
      "#{quantity(number-1)} #{container(number-1)} of beer
on the wall.\n"
    end
  end
```

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 4.53: Case 0 Sends Container

```
  def verse(number)
    case number
    when 0
      # ...
      "#{quantity(99)} #{container(number-1)} of beer on the
wall.\n"
    else
      # ...
      "#{quantity(number-1)} #{container(number-1)} of beer
on the wall.\n"
    end
  end
```

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 `5`, `4`, when `1`, `0`, and when `0`, `99`. It's where

the song determines the *next verse to be sung*. "Next" is a keyword in Ruby, so it can't be used as the name, but a concept like this already exists for many Ruby objects, and it makes sense to leverage Ruby's existing name.

Several classes in the Ruby standard library define a "successor," using the unfortunately named `succ` method. Here are a few examples:

```
"a".succ # => "b"
9.succ   # => 10
```

Most verses are succeeded by the next lower verse, with the exception of verse 0, which is followed by verse 99. "Successor" is a good name for this concept.

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 `else` branch difference. Here's that code:

Listing 4.54: Successor Handles Default

```
1   def successor(number)
2     number - 1
3   end
```

The code in `successor` refers to `number`, so the argument must be defined from the first.

Now that `successor` exists, use it in the `else` branch in place of `number-1` (line 8 below):

Listing 4.55: Else Case Sends Successor

```
1    def verse(number)
2      case number
3      when 0
4        # ...
5        "#{quantity(99)} #{container(number-1)} of beer on the
wall.\n"
6      else
7        # ...
8        "#{quantity(successor(number))} #{container(number-1)}
of beer on the wall.\n"
9      end
10   end
```

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 4.56: Successor Handles Both Cases

```
1    def successor(number)
2      if number == 0
3        99
4      else
5        number - 1
6      end
7    end
```

Now that the conditional exists, the `99` can be replaced by a send of `successor`, as shown on line 5 below:

Listing 4.57: Both Cases Send Successor

```
1    def verse(number)
2      case number
3      when 0
4        # ...
5        "#{quantity(successor(number))} #{container(number-1)}
of beer on the wall.\n"
```

```
6       else
7         # ...
8         "#{quantity(successor(number))} #{container(number-1)}
of beer on the wall.\n"
9       end
10    end
```

After this change, the `0` and `else` cases are identical.

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 `"#{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. Therefore, you should now defer to that abstraction, and replace all occurrences of `number-1` with `successor(number)`.

That final change results in this code:

Listing 4.58: Deferring to Successor

```
  def verse(number)
    case number
    when 0
      # ...
      "#{quantity(successor(number))} #
{container(successor(number))} of beer on the wall.\n"
    else
      # ...
      "#{quantity(successor(number))} #
{container(successor(number))} of beer on the wall.\n"
    end
  end
```

Here's the whole `verse` method, showing the `0` and `else` cases to be identical:

Listing 4.59: Identical 0 and Else Cases

```
  def verse(number)
    case number
    when 0
```

```
      "#{quantity(number).capitalize} #{container(number)}
of beer on the wall, " +
      "#{quantity(number)} #{container(number)} of beer.\n"
+
      "#{action(number)}, " +
      "#{quantity(successor(number))} #
{container(successor(number))} of beer on the wall.\n"
    else
      "#{quantity(number).capitalize} #{container(number)}
of beer on the wall, " +
      "#{quantity(number)} #{container(number)} of beer.\n"
+
      "#{action(number)}, " +
      "#{quantity(successor(number))} #
{container(successor(number))} of beer on the wall.\n"
    end
  end
```

One last refactoring trick proves that this common template works for all cases. Copy the template and insert it below the case statement, as follows:

Listing 4.60: Using the Same Template for Every Verse

```
  def verse(number)
    case number
    when 0
      # ...
    else
      # ...
    end
    "#{quantity(number).capitalize} #{container(number)} of
beer on the wall, " +
    "#{quantity(number)} #{container(number)} of beer.\n" +
    "#{action(number)}, " +
    "#{quantity(successor(number))} #
{container(successor(number))} of beer on the wall.\n"
  end
```

Ruby methods return the result of the last bit of evaluated code, so 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, and so you can safely delete the entire `case` statement.

Here's a complete listing of the resulting code:

Listing 4.61: Final Listing

```
class Bottles
  def song
    verses(99, 0)
  end

  def verses(starting, ending)
    starting.downto(ending).collect {|i|
verse(i)}.join("\n")
  end

  def verse(number)
    "#{quantity(number).capitalize} #{container(number)} of
beer on the wall, " +
    "#{quantity(number)} #{container(number)} of beer.\n" +
    "#{action(number)}, " +
    "#{quantity(successor(number))} #
{container(successor(number))} of beer on the wall.\n"
  end

  def quantity(number)
    if number == 0
      "no more"
    else
      number.to_s
    end
  end

  def container(number)
    if number == 1
```

```
27        "bottle"
28      else
29        "bottles"
30      end
31    end
32
33    def action(number)
34      if number == 0
35        "Go to the store and buy some more"
36      else
37        "Take #{pronoun(number)} down and pass it around"
38      end
39    end
40
41    def pronoun(number)
42      if number == 1
43        "it"
44      else
45        "one"
46      end
47    end
48
49    def successor(number)
50      if number == 0
51        99
52      else
53        number - 1
54      end
55    end
56  end
```

This completes the current refactoring. The `verse` `case` 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 Flog thinks it's worse.

From Flog's point of view, all you've accomplished is turn one conditional into many, while simultaneously adding 55% more code.

However, be of good cheer. Despite the Flog 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):

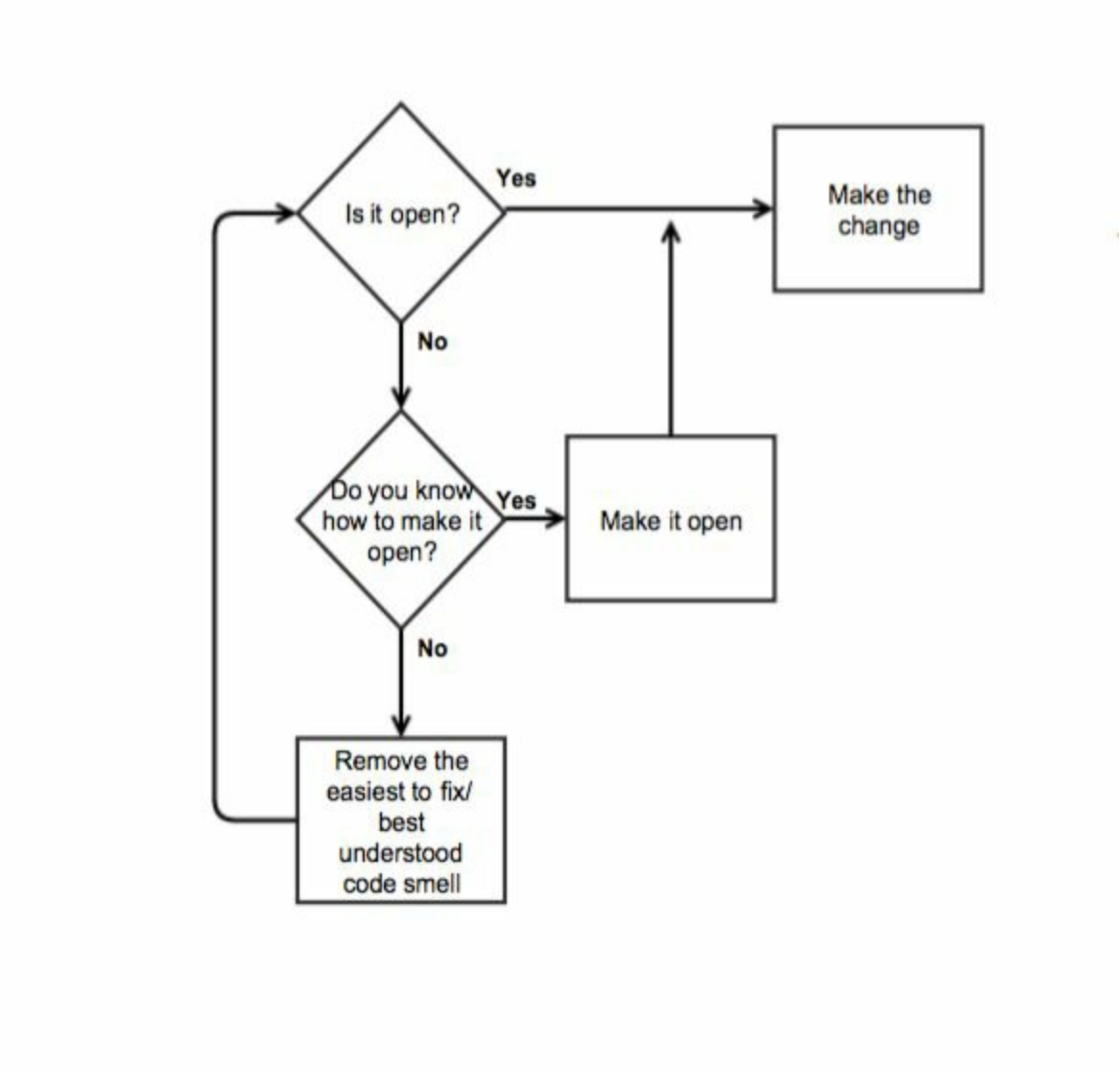

Figure 5.1: Open Closed Flowchart

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 case 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

  def song
    verses(99, 0)
  end

  def verses(starting, ending)
    starting.downto(ending).collect {|i|
verse(i)}.join("\n")
  end

  def verse(number)
    "#{quantity(number).capitalize} #{container(number)} " +
      "of beer on the wall, " +
    "#{quantity(number)} #{container(number)} of beer.\n" +
    "#{action(number)}, " +
    "#{quantity(successor(number))} #
{container(successor(number))} " +
      "of beer on the wall.\n"
  end

  def container(number)
    if number == 1
      "bottle"
    else
      "bottles"
    end
  end

  def quantity(number)
    if number == 0
      "no more"
    else
      number.to_s
    end
  end

  def action(number)
    if number == 0
```

```
38          "Go to the store and buy some more"
39        else
40          "Take #{pronoun(number)} down and pass it around"
41        end
42      end
43
44      def pronoun(number)
45        if number == 1
46          "it"
47        else
48          "one"
49        end
50      end
51
52      def successor(number)
53        if number == 0
54          99
55        else
56          number - 1
57        end
58      end
59    end
```

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 to add the private keyword to this class, where would it go?
5. If you were going to break this class into two pieces, where's the dividing line?

For those methods created by the Flocking Rules (`container`, `quantity`, `action`, `pronoun` and `successor`, hereafter referred to as the "flocked five"):

6. Do the tests in the conditionals have anything in common?
7. How many branches do the conditionals have?
8. Do the methods contain any code *other* than the conditional?
9. 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 five 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
def container(number)
  if number == 1
   "bottle"
  else
   "bottles"
  end
end

#guard clause
def quantity(number)
  return "no more" if number == 0
  number.to_s
end

#ternary expression
def pronoun(number)
  number == 1 ? "it" : "one"
end
```

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

```
def verse(number)
def container(number)
def quantity(number)
def action(number)
def pronoun(number)
def 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

```
  def song
    verses(99, 0)
  end
```

When `song` sends `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

```
1   def verses(starting, ending)
2     starting.downto(ending).collect {|i| verse(i)}.join("\n")
3   end
```

The `starting` and `ending` arguments are verse numbers. The `verses` method iterates between them, so `i`, the argument yielded to the block, 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`.

```
def 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`:

```
def 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

```
1    def verse(number)
2      "#{quantity(number).capitalize} #{container(number)} " +
3        "of beer on the wall, " +
4      "#{quantity(number)} #{container(number)} of beer.\n" +
5      "#{action(number)}, " +
6      "#{quantity(successor(number))} #
{container(successor(number))} " +
7        "of beer on the wall.\n"
8    end
```

Notice that line 2 above invokes `container` with `number`, while line 6 invokes `container` with `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 case 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 to add the private keyword, where would it go?

After `verse` and before the flocked five methods.

> Question 5: If you were going to break this class into two pieces, where's the dividing line?

Same as above, i.e. 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 six through nine, which apply only to the flocked five methods.

> Question 6: Do the tests in the conditionals have anything in common?

Here's a summary of the conditionals:

Listing 5.7: Flocked Five Conditional Tests

```
  def container(number)
    if number == 1
      # ...
  end

  def quantity(number)
    if number == 0
      # ...
  end

  def action(number)
    if number == 0
      # ...
  end

  def pronoun(number)
    if number == 1
      # ...
```

```
19    end
20
21    def successor(number)
22      if number == 0
23        # ...
24    end
```

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 tests 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 7: How many branches do the conditionals have?

Each conditional contains two branches. This may or not have meaning, but it's certainly a visible quality of the code and thus worth noting.

> Question 8: 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 9: Does 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 nine 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:

```
def container(number)
  if number == 1
   "bottle"
  else
   "bottles"
  end
end
```

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:

```
def container(smarter_number)
  smarter_number.container
end
```

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 classes like `String`, `Fixnum`, `Integer` (`Fixnum`'s superclass), `Array`, and `Hash` are examples of "primitives." *Primitive Obsession* is when you use one of these data classes 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
  # ...
end

class BottleNumber
end
```

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
  # ...
  def container(number)
    if number == 1
      "bottle"
    else
      "bottles"
    end
  end

  def quantity(number)
    # ...
  end

  def action(number)
    # ...
  end

  def pronoun(number)
    # ...
  end

  def successor(number)
    # ...
  end
end

class BottleNumber
  def container(number)
    if number == 1
      "bottle"
    else
```

```
      "bottles"
    end
  end

  def quantity(number)
    if number == 0
      "no more"
    else
      number.to_s
    end
  end

  def action(number)
    if number == 0
      "Go to the store and buy some more"
    else
      "Take #{pronoun(number)} down and pass it around"
    end
  end

  def pronoun(number)
    if number == 1
      "it"
    else
      "one"
    end
  end

  def successor(number)
    if number == 0
      99
    else
      number - 1
    end
  end
end
```

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 fields, and then methods, of interest. In contrast, the example above starts with Fowlers 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 an `attr_reader` for `:number`, and an `initialize` method to set the variable. Here's the code:

Listing 5.10: BottleNumber Holding Onto Number

```
1 class BottleNumber
2   attr_reader :number
3   def initialize(number)
```

```
4       @number = number
5     end
6     # ...
7   end
```

On line 2 above, `attr_reader` is a class method. Invoking it with the symbol `:number` effectively defines a new instance method on `BottleNumber` that acts like this:

```
def number
  @number
end
```

Because of the `attr_reader`, `BottleNumber` responds to the `number` message by returning the value held in the `@number` instance variable. This variable is *set* within the `initialize` method on line 4 above. That `initialize` method gets invoked when `new` is sent to `BottleNumber`.

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 `container` method of `Bottles` to invoke the container method of `BottleNumber`:

Listing 5.11: Parse and Execute a Bit of New Code

```
1  class Bottles
2    # ...
3    def container(number)
4      BottleNumber.new(number).container(number)
5      if number == 1
6        "bottle"
7      else
8        "bottles"
```

```
 9       end
10     end
11     # ...
12   end
```

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.

```
BottleNumber.new(number).container(number)
```

In the above code, both `new` and `container` require the `number` argument, so it must passed twice. You may find this annoyingly redundant. In the newly-created `BottleNumber` class, the `container` method could easily make do without an argument. It can get the right number by simply sending the `number` message to itself. Instead of the code above, you'd prefer:

```
BottleNumber.new(number).container
```

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

`container` message within the `Bottle` class. The easiest way to accomplish this is to move line 4 to the bottom of the method, like so:

Listing 5.12: Parse, Execute and Use Result

```
1  class Bottles
2    # ...
3    def container(number)
4      if number == 1
5        "bottle"
6      else
7        "bottles"
8      end
9      BottleNumber.new(number).container(number)
10   end
11   # ...
12 end
```

The tests pass, so now you can delete the old implementation from `container` (lines 4-8 above). This leaves the following code:

Listing 5.13: Resulting Container Method

```
1  class Bottles
2    # ...
3    def container(number)
4      BottleNumber.new(number).container(number)
5    end
6    # ...
7  end
```

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
  # ...
  def container(number)
    BottleNumber.new(number).container(number)
  end

  def quantity(number)
    BottleNumber.new(number).quantity(number)
  end

  def action(number)
    BottleNumber.new(number).action(number)
  end

  def pronoun(number)
    BottleNumber.new(number).pronoun(number)
  end

  def successor(number)
    BottleNumber.new(number).successor(number)
  end
end
```

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 `container` methods:

Listing 5.15: Redundant Arguments

```
1  class Bottles
2    # ...
3    def container(number)
4      BottleNumber.new(number).container(number)
5    end
6    # ...
7  end
8
9  class BottleNumber
10   attr_reader :number
11   def initialize(number)
12     @number = number
13   end
14
15   def container(number)
16     if number == 1
17       "bottle"
18     else
19       "bottles"
20     end
21   # ...
22 end
```

Line 4 above gets a new `BottleNumber` and asks for its `container`. Doing so requires two references to `number`. The `initialize` method (invoked by `new` and defined on line 11) and the `container` method (line 15) 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 every message, it's easy to make this change. For example, you could fix the `container` method by changing line 4 above to remove the parameter being passed to `container`, while simultaneously deleting the argument from line 15. If you make both of these changes at once, and then save and run the tests, the tests will pass.

Keep in mind that is a multi-line change. Some problems are so simple that it's easiest 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. Whether arguments are being added or removed, the trick is the same; you must change the method definition to temporarily set the argument to a default.

There are a several ways to accomplish this. The following

technique is the most direct, but requires a short refresher on Ruby syntax.

Consider `container`, repeated again below. This method takes a `number` *argument*. Remember, however, that the `BottleNumber` class itself responds to the `number` *message*. Now answer this question: On line 4 below, does `number` refer to the argument, or to the message?

Listing 5.16: BottleNumber Container Redoux

```
1  class BottleNumber
2    # ...
3    def container(number)
4      if number == 1
5        "bottle"
6      else
7        "bottles"
8      end
9    end
10   # ...
11 end
```

Ruby is perfectly happy to allow the same name to be used for different things, and to infer which you mean based on context. In the code above, the programmer clearly intends for `number` on line 4 to refer to the `number` argument from line 3, and that's exactly what Ruby does. The `number` on line 4 is interpreted as a reference to the method's argument rather than as a send of the `number` message.

Armed with this knowledge, you can guess that removing the argument from the method definition would cause Ruby to interpret line 4 as a send of the `number` message. This is your goal, but unfortunately, the `Bottles` class is still *sending* `container(number)`, so this change breaks the tests.

The trick to working your way forward under green, while making only one-line changes, is to alter the name of the argument to something *other* than `number`, and simultaneously give it a default. Line 3 below contains that change:

Listing 5.17: Renamed Argument

```
1  class BottleNumber
2    # ...
3    def container(delete_me=nil)
4      if number == 1
5        "bottle"
6      else
7        "bottles"
8      end
9    end
10   # ...
11 end
```

Above, the `number` argument for `container` has been renamed to `delete_me` and assigned a default of `nil`. That change turns the `number` reference on line 4 into a message send, which allows this method to depend upon a message sent to itself rather than an argument passed by someone else.

Now that the argument is optional, turn your attention to senders of `container`. In this application there's only the one in `Bottles`, shown here:

Listing 5.18: Forward With Redundant Arguments

```
1  class Bottles
2    # ...
3    def container(number)
4      BottleNumber.new(number).container(number)
5    end
6    # ...
7  end
```

Removing the `number` parameter from the `container` message invocation on line 4 results in this code:

Listing 5.19: Forward Without Redundancy

```
1 class Bottles
2   # ...
3   def container(number)
4     BottleNumber.new(number).container
5   end
6   # ...
7 end
```

Once you have located and removed the parameter from all of its senders, the `container` method definition no longer needs to take an argument. You can now return to `BottleNumber` and remove the `delete_me` argument and default, as on line 3 below:

Listing 5.20: BottleNumber Container Method Without Argument

```
1  class BottleNumber
2    # ...
3    def container
4      if number == 1
5        "bottle"
6      else
7        "bottles"
8      end
9    end
10   # ...
11 end
```

Here's a recap of the steps for removing an argument using one-line changes.

1. Alter the method definition to change the argument name and provide a default.

   Start by changing the existing argument name to anything other than what it currently is. Using `delete_me` will help you remember to delete the argument when you've updated all of the senders. The value of the default does not matter, so it's common to use `nil`. In the example above:

   ```
   def container(number)
   ```

   became:

   ```
   def container(delete_me=nil)
   ```

2. Change every sender of the message to remove the parameter. In the example:

   ```
   BottleNumber.new(number).container(number)
   ```

   became:

   ```
   BottleNumber.new(number).container
   ```

3. Finally, delete the argument from the method definition. So, finally:

   ```
   def container(delete_me=nil)
   ```

   became:

   ```
   def container
   ```

As you can see, despite the length of the explanation, the technique is simple, and involves only three steps. Having

practiced on `container`, 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 `quantity` and `action` work as expected, but that when you change `pronoun`, the tests begin 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 in many circumstances, for many people, in many situations. 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 `container`, `quantity`, and `action` but causes the tests to fail when applied to `pronoun`.

Specifically, if you go to the `pronoun` definition in `BottleNumber`:

```
class BottleNumber
  # ...
  def pronoun(number)
```

and change `number` to `delete_me`, and supply a default:

```
class BottleNumber
  # ...
  def pronoun(delete_me=nil)
```

Then go to the `pronoun` method in `Bottles`:

```
class Bottles
  # ...
  def pronoun(number)
    BottleNumber.new(number).pronoun(number)
```

and remove the parameter from the forward of `pronoun` to `BottleNumber`:

```
class Bottles
  # ...
  def pronoun(number)
    BottleNumber.new(number).pronoun
```

Finally, you return to the `pronoun` method definition in `BottleNumber` and delete the entire argument:

```
class BottleNumber
  # ...
  def pronoun
```

Then the tests begin to fail with:

```
ArgumentError: wrong number of arguments (given 1, expected 0)
```

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 change the argument name, and provide a default.
2. Change every sender of the message to remove the

parameter.

3. Delete the argument 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; i.e. 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
  # ...
  def action(number)
    if number == 0
      "Go to the store and buy some more"
    else
      "Take #{pronoun(number)} down and pass it around"
    end
  end
```

It turns out that `pronoun` is invoked only from the `action` method of `BottleNumber`, where the message is sent to `self`. 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
  # ...
  def action(number)
    if number == 0
      "Go to the store and buy some more"
    else
```

```
      "Take #{pronoun} down and pass it around"
    end
  end
```

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

  def song
    verses(99, 0)
  end

  def verses(starting, ending)
    starting.downto(ending).collect {|i|
verse(i)}.join("\n")
  end

```

```ruby
  def verse(number)
    "#{quantity(number).capitalize} #{container(number)} " +
      "of beer on the wall, " +
    "#{quantity(number)} #{container(number)} of beer.\n" +
    "#{action(number)}, " +
    "#{quantity(successor(number))} #
{container(successor(number))} " +
      "of beer on the wall.\n"
  end

  def container(number)
    BottleNumber.new(number).container
  end

  def quantity(number)
    BottleNumber.new(number).quantity
  end

  def action(number)
    BottleNumber.new(number).action
  end

  def successor(number)
    BottleNumber.new(number).successor
  end
end

class BottleNumber
  attr_reader :number
  def initialize(number)
    @number = number
  end

  def container
    if number == 1
      "bottle"
    else
      "bottles"
    end
  end

```

```
51    def quantity
52      if number == 0
53        "no more"
54      else
55        number.to_s
56      end
57    end
58
59    def action
60      if number == 0
61        "Go to the store and buy some more"
62      else
63        "Take #{pronoun} down and pass it around"
64      end
65    end
66
67    def pronoun
68      if number == 1
69        "it"
70      else
71        "one"
72      end
73    end
74
75    def successor
76      if number == 0
77        99
78      else
79        number - 1
80      end
81    end
82  end
```

The completes the extraction of the `BottleNumber` class. Despite its many conditionals, the code 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 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, i.e. 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` variable 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 `BottleNumber.new(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 an 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 `BottleNumber` s. 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
  # ...
  def verse(number)
    "#{quantity(number).capitalize} #{container(number)} " +
      "of beer on the wall, " +
    "#{quantity(number)} #{container(number)} of beer.\n" +
    "#{action(number)}, " +
    "#{quantity(successor(number))} #
{container(successor(number))} " +
      "of beer on the wall.\n"
  end

  def container(number)
```

```
13      BottleNumber.new(number).container
14    end
15
16    def quantity(number)
17      BottleNumber.new(number).quantity
18    end
19
20    def action(number)
21      BottleNumber.new(number).action
22    end
23
24    def successor(number)
25      BottleNumber.new(number).successor
26    end
27  end
```

In the code above, a new instance of `BottleNumber` is created each time `container`, `quantity`, `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 `container`, `quantity`, `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 other 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.

The fourth phrase of verse 99 sends `quantity` and `container` once, and `successor` twice, creating four instances of `BottleNumber` on `number` 98. Caching the first instance would save three further object creations within this verse. Additionally, the cached copy could also be used within the first three phrases of the *following* verse, saving five more object creations, for a total of eight altogether. Over the course of the song, this would 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 avoid creating 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 `container`, `quantity`, `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    def verse(number)
4      bottle_number = BottleNumber.new(number)
5
6      "#{quantity(number).capitalize} #{container(number)} " +
7        "of beer on the wall, " +
8      "#{quantity(number)} #{container(number)} of beer.\n" +
9      "#{action(number)}, " +
10     "#{quantity(successor(number))} #
```

```
{container(successor(number))} " +
      "of beer on the wall.\n"
  end
  # ...
end
```

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 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 self. The next example begins the transition with the simplest change possible. Line 4 below asks this new object for its `action`:

Listing 5.24: Asking the Cached Object for Its Action

```
1  def verse(number)
2    bottle_number = BottleNumber.new(number)
3    # ...
4    "#{bottle_number.action}, " +
5    # ...
6  end
```

In the code above, `action(number)` has been replaced by `bottle_number.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

```
1   def verse(number)
2     bottle_number = BottleNumber.new(number)
3
4     "#{bottle_number.quantity.capitalize} #
{bottle_number.container} " +
5       "of beer on the wall, " +
6     "#{bottle_number.quantity} #{bottle_number.container} of
beer.\n" +
7     "#{bottle_number.action}, " +
8     # ...
9   end
```

In lines 4 and 5 of the code above, `quantity` and `container` are now sent directly to `bottle_number`. 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 `self`. 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

```
1   def verse(number)
2     bottle_number = BottleNumber.new(number)
3     # ...
4     "#{quantity(successor(number))} #
{container(successor(number))} " +
5       "of beer on the wall.\n"
6   end
```

The plan is to change phrase 4 to send messages to instances of `BottleNumber` rather than to `self`. Previously, when making a similar change to phrase 1 and 2,

```
quantity(number)
```

was replaced with

```
bottle_number.quantity
```

On line 4 above, phrase 4 also invokes `quantity`, but it passes a different argument than does phrase 1:

```
quantity(successor(number))
```

The `quantity` method above is passed `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 [13] if you expect the `successor` of a `BottleNumber` to be another `BottleNumber`. If this were true, you could replace:

```
quantity(successor(number))
```

with:

```
bottle_number.successor.quantity
```

Unfortunately, as is, this code doesn't work. If you make the above change and run the tests, you'll see:

```
NoMethodError: undefined method `quantity' for 99:Fixnum
```

The problem is that `successor` still returns a number, when logically it should now return the succeeding `BottleNumber`. `BottleNumber` s know `quantity`, but `Fixnum` s 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` case statement into the new `successor` method. The case 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 line 3 below, the following example bravely does just that:

Listing 5.27: Caching the Successor

```
 1    def verse(number)
 2      bottle_number = BottleNumber.new(number)
 3      next_bottle_number =
BottleNumber.new(bottle_number.successor)
 4
 5      "#{bottle_number.quantity.capitalize} #
{bottle_number.container} " +
 6        "of beer on the wall, " +
 7      "#{bottle_number.quantity} #{bottle_number.container} of
beer.\n" +
 8      "#{bottle_number.action}, " +
 9      "#{next_bottle_number.quantity} #
{next_bottle_number.container} " +
10        "of beer on the wall.\n"
11    end
```

Line 3 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 `next_bottle_number` exists, line 9 can ask it for its `quantity` and `container`.

After that change, the `verse` method contains two distinct parts. Lines 5-10 above define a template which queries

instances of `BottleNumber` for details. Lines 2 and 3 create new instances of `BottleNumber`. Line 2 seems reasonable, but line 3 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
3    def song
4      verses(99, 0)
5    end
6
7    def verses(starting, ending)
8      starting.downto(ending).collect {|i|
verse(i)}.join("\n")
9    end
10
11   def verse(number)
12     bottle_number = BottleNumber.new(number)
13     next_bottle_number =
BottleNumber.new(bottle_number.successor)
14
15     "#{bottle_number.quantity.capitalize} #
{bottle_number.container} " +
16       "of beer on the wall, " +
17     "#{bottle_number.quantity} #{bottle_number.container} of
beer.\n" +
18     "#{bottle_number.action}, " +
19     "#{next_bottle_number.quantity} #
{next_bottle_number.container} " +
20       "of beer on the wall.\n"
```

```ruby
  end
end

class BottleNumber
  attr_reader :number
  def initialize(number)
    @number = number
  end

  def container
    if number == 1
      "bottle"
    else
      "bottles"
    end
  end

  def quantity
    if number == 0
      "no more"
    else
      number.to_s
    end
  end

  def action
    if number == 0
      "Go to the store and buy some more"
    else
      "Take #{pronoun} down and pass it around"
    end
  end

  def pronoun
    if number == 1
      "it"
    else
      "one"
    end
  end

```

```
62    def successor
63      if number == 0
64        99
65      else
66        number - 1
67      end
68    end
69  end
```

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 Flog score, for example, has gone up again. From Flog's 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. Replacing Conditionals with Objects

This chapter is pending.

This ultimate chapter transforms the code to be open to the six-pack requirement. Along the way it resolves a few more code smells, delves into monkey-patching and metaprogramming, and utilizes inheritance *and* composition.

The code ends up being so simple and expressive that the six-pack requirement is met by adding just a few lines of new code. It's a wonder to behold.

Appendix A: Prerequisites

A.1. Ruby

The code is compatible with any Ruby version starting at 1.9. Check which version of Ruby you have with the following command:

```
ruby --version
```

If you don't have Ruby 1.9 or higher installed follow the instructions on ruby-lang.org to install it.

A.2. Minitest

The code examples include a Minitest test suite. To check which versions of Minitest you have, use the `gem list` command.

```
gem list minitest
```

If Minitest is not installed, or if none of the versions listed are in the 5.x series, install it with `gem install`.

```
gem install minitest --version "~> 5.4"
```

Appendix B: Initial Exercise

B.1. 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=exercise
https://github.com/sandimetz/99bottles.git
```

The directory structure for the exercise should look like this:

```
├── lib
│   └── bottles.rb
└── test
    └── bottles_test.rb
```

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.rb`.

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.

B.2. Doing the exercise

The test suite and exercise are written in Ruby. If you're unfamiliar with the language, ruby-lang.org has installation instructions, a gentle tutorial (Ruby in Twenty Minutes), and further references.

To run the test suite, invoke Ruby with the path to the test file.

```
ruby test/bottles_test.rb
```

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.

B.3. Test Suite

```
class BottlesTest < Minitest::Test
  def test_the_first_verse
    expected = <<-VERSE
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.
```

```
VERSE
    assert_equal expected, ::Bottles.new.verse(99)
  end

  def test_another_verse
    skip
    expected = <<-VERSE
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.
VERSE
    assert_equal expected, ::Bottles.new.verse(89)
  end

  def test_verse_2
    skip
    expected = <<-VERSE
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.
    VERSE
    assert_equal expected, ::Bottles.new.verse(2)
  end

  def test_verse_1
    skip
    expected = <<-VERSE
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.
    VERSE
    assert_equal expected, ::Bottles.new.verse(1)
  end

  def test_verse_0
    skip
    expected = <<-VERSE
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.
```

```
    VERSE
    assert_equal expected, ::Bottles.new.verse(0)
  end

  def test_a_couple_verses
    skip
    expected = <<-VERSES
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.
VERSES
    assert_equal expected, ::Bottles.new.verses(99, 98)
  end

  def test_a_few_verses
    skip
    expected = <<-VERSES
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.
VERSES
    assert_equal expected, ::Bottles.new.verses(2, 0)
  end

  def test_the_whole_song
    skip
    expected = <<-SONG
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.
```

```
138
139 78 bottles of beer on the wall, 78 bottles of beer.
140 Take one down and pass it around, 77 bottles of beer on the
wall.
141
142 77 bottles of beer on the wall, 77 bottles of beer.
143 Take one down and pass it around, 76 bottles of beer on the
wall.
144
145 76 bottles of beer on the wall, 76 bottles of beer.
146 Take one down and pass it around, 75 bottles of beer on the
wall.
147
148 75 bottles of beer on the wall, 75 bottles of beer.
149 Take one down and pass it around, 74 bottles of beer on the
wall.
150
151 74 bottles of beer on the wall, 74 bottles of beer.
152 Take one down and pass it around, 73 bottles of beer on the
wall.
153
154 73 bottles of beer on the wall, 73 bottles of beer.
155 Take one down and pass it around, 72 bottles of beer on the
wall.
156
157 72 bottles of beer on the wall, 72 bottles of beer.
158 Take one down and pass it around, 71 bottles of beer on the
wall.
159
160 71 bottles of beer on the wall, 71 bottles of beer.
161 Take one down and pass it around, 70 bottles of beer on the
wall.
162
163 70 bottles of beer on the wall, 70 bottles of beer.
164 Take one down and pass it around, 69 bottles of beer on the
wall.
165
166 69 bottles of beer on the wall, 69 bottles of beer.
167 Take one down and pass it around, 68 bottles of beer on the
wall.
168
```

```
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.
```

```
261
262 37 bottles of beer on the wall, 37 bottles of beer.
263 Take one down and pass it around, 36 bottles of beer on the
wall.
264
265 36 bottles of beer on the wall, 36 bottles of beer.
266 Take one down and pass it around, 35 bottles of beer on the
wall.
267
268 35 bottles of beer on the wall, 35 bottles of beer.
269 Take one down and pass it around, 34 bottles of beer on the
wall.
270
271 34 bottles of beer on the wall, 34 bottles of beer.
272 Take one down and pass it around, 33 bottles of beer on the
wall.
273
274 33 bottles of beer on the wall, 33 bottles of beer.
275 Take one down and pass it around, 32 bottles of beer on the
wall.
276
277 32 bottles of beer on the wall, 32 bottles of beer.
278 Take one down and pass it around, 31 bottles of beer on the
wall.
279
280 31 bottles of beer on the wall, 31 bottles of beer.
281 Take one down and pass it around, 30 bottles of beer on the
wall.
282
283 30 bottles of beer on the wall, 30 bottles of beer.
284 Take one down and pass it around, 29 bottles of beer on the
wall.
285
286 29 bottles of beer on the wall, 29 bottles of beer.
287 Take one down and pass it around, 28 bottles of beer on the
wall.
288
289 28 bottles of beer on the wall, 28 bottles of beer.
290 Take one down and pass it around, 27 bottles of beer on the
wall.
291
```

```
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.
354
355  6 bottles of beer on the wall, 6 bottles of beer.
356  Take one down and pass it around, 5 bottles of beer on the
wall.
357
358  5 bottles of beer on the wall, 5 bottles of beer.
359  Take one down and pass it around, 4 bottles of beer on the
wall.
360
361  4 bottles of beer on the wall, 4 bottles of beer.
362  Take one down and pass it around, 3 bottles of beer on the
wall.
363
364  3 bottles of beer on the wall, 3 bottles of beer.
365  Take one down and pass it around, 2 bottles of beer on the
wall.
366
367  2 bottles of beer on the wall, 2 bottles of beer.
368  Take one down and pass it around, 1 bottle of beer on the
wall.
369
370  1 bottle of beer on the wall, 1 bottle of beer.
371  Take it down and pass it around, no more bottles of beer on
the wall.
372
373  No more bottles of beer on the wall, no more bottles of
beer.
374  Go to the store and buy some more, 99 bottles of beer on
the wall.
375      SONG
376      assert_equal expected, ::Bottles.new.song
377    end
378  end
```

Acknowledgements

We're grateful to everyone, and will tell you all about it very soon.

---

1. From the novel by Joseph Heller, a catch-22 is a paradoxical situation from which you cannot escape because of contradictory rules.
2. 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
3. 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.
4. A quote from Robert Martin's Transformation Priority Premise blog post.
5. A red herring is something that misleads or distracts from a relevant or important issue.
6. A hair shirt, or cilice, is an undergarment made of animal hair, worn to induce discomfort as a sign of repentance or atonement.
7. See Kent Beck Don't Cross the Beams: Avoiding Interference Between Horizontal and Vertical Refactorings
8. Thanks to Avdi Grimm for the suggestion of using rows and columns in an imaginary spreadsheet to help find names for underlying concepts.
9. 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.
10. "You'll never know less than you know right now" is a quote from Kent Beck.
11. Spidey (or spider) sense is a tingling feeling at the base of Marvel Comics superhero Spider-Man 's skull that alerts him to danger.
12. A quote from 1 Corinthians 13:12 of the King James Version of the Christian Bible.
13. Merriam Webster defines *bang on* as "exactly correct or appropriate"

0.3
Last updated 2016-10-06 11:48:16 EDT