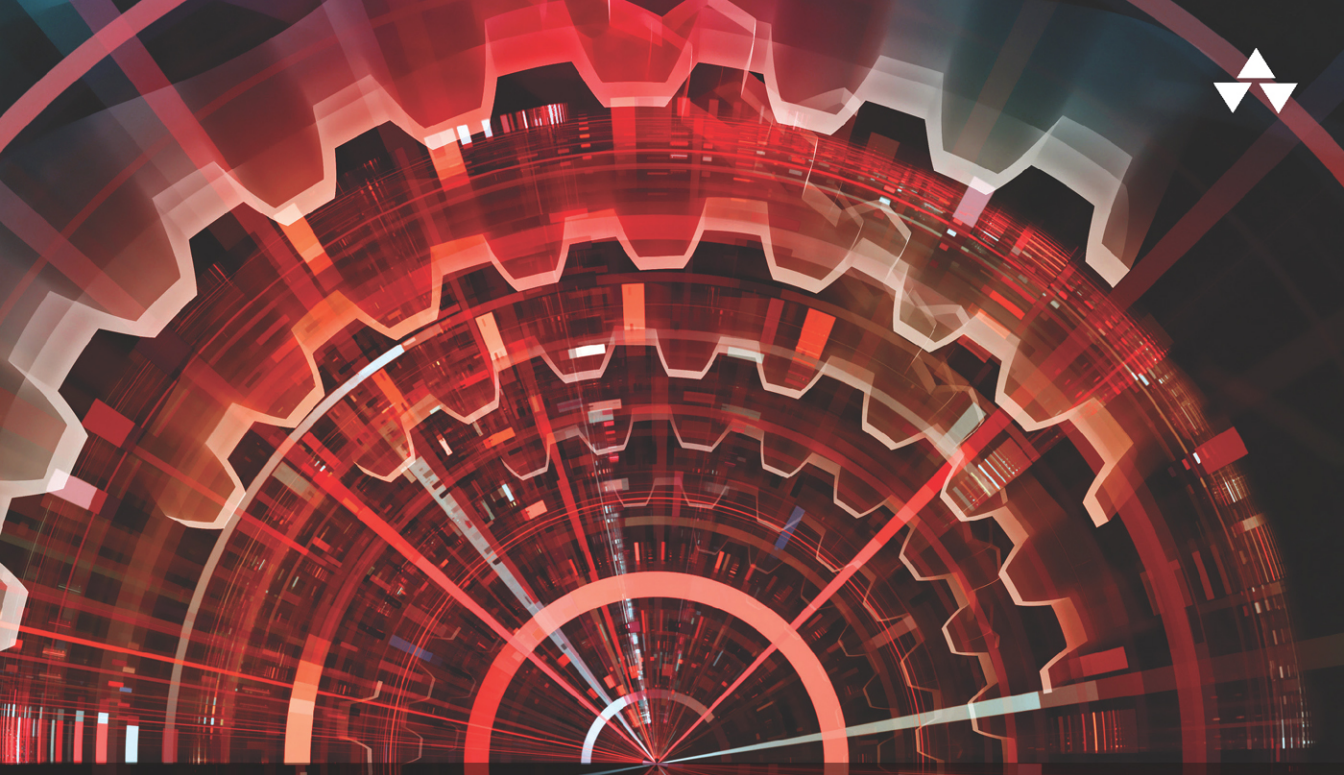

PRACTICAL OBJECT-ORIENTED DESIGN

An Agile Primer Using Ruby

Second Edition

SANDI METZ

Praise for the first edition of *Practical Object-Oriented Design in Ruby*

"Meticulously pragmatic and exquisitely articulate, Practical Object Oriented Design in Ruby makes otherwise elusive knowledge available to an audience which desperately needs it. The prescriptions are appropriate both as rules for novices and as guidelines for experienced professionals."

—Katrina Owen, Creator, Exercism

"I do believe this will be the most important Ruby book of 2012. Not only is the book 100% on-point, Sandi has an easy writing style with lots of great analogies that drive every point home."

—Avdi Grimm, author of *Exceptional Ruby and Objects on Rails*

"While Ruby is an object-oriented language, little time is spent in the documentation on what OO truly means or how it should direct the way we build programs. Here Metz brings it to the fore, covering most of the key principles of OO development and design in an engaging, easy-to-understand manner. This is a must for any respectable Ruby bookshelf."

—Peter Cooper, editor, *Ruby Weekly*

"So good, I couldn't put it down! This is a must-read for anyone wanting to do object-oriented programming in any language, not to mention it has completely changed the way I approach testing."

—Charles Max Wood, Ruby Rogues Podcast co-host and CEO of Devchat.tv

"Distilling scary OO design practices with clear-cut examples and explanations makes this a book for novices and experts alike. It is well worth the study by anyone interested in OO design being done right and 'light.' I thoroughly enjoyed this book."

—Manuel Pais, DevOps and Continuous Delivery Consultant, Independent

"If you call yourself a Ruby programmer, you should read this book. It's jam-packed with great nuggets of practical advice and coding techniques that you can start applying immediately in your projects."

—Ylan Segal, San Diego Ruby User Group

"This is the best OO book I've ever read. It's short, sweet, but potent. It slowly moves from simple techniques to more advanced, each example improving on the last. The ideas it presents are useful not just in Ruby but in static languages like C# too. Highly recommended!"

—Kevin Berridge, software engineering manager,
Pointe Blank Solutions, and organizer, Burning River Developers Meetup

"This is the best programming book I've read in ages. Sandi talks about basic principles, but these are things we're probably still doing wrong and she shows us why and how. The book has the perfect mix of code, diagrams, and words. I can't recommend it enough and if you're serious about being a better programmer, you'll read it and agree.

—Derick Hitchcock, software engineer, Cisco

"Metz's take on the subject is rooted strongly in theory, but the explanation always stays grounded in real world concerns, which helped me to internalize it. The book is clear and concise, yet achieves a tone that is more friendly than terse."

—Alex Strasheim, network administrator, Ensemble Travel Group

"Whether you're just getting started in your software development career, or you've been coding for years (like I have), it's likely that you'll learn a lot from Ms. Metz's book. She does a fantastic job of explaining the whys of well-designed software along with the hows."

—Gabe Hollombe, software craftsman, avantbard.com

Practical Object-Oriented Design

Second Edition

This page intentionally left blank

Practical Object-Oriented Design

An Agile Primer Using Ruby

Second Edition

Sandi Metz

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town • Dubai
London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2018939833

ISBN-13: 978-0-13-445647-8
ISBN-10: 0-13-445647-5

ScoutAutomatedPrintCode

For Amy, who read everything first

This page intentionally left blank

Contents

This page intentionally left blank

Introduction

We want to do our best work, and we want the work we do to have meaning. And, all else being equal, we prefer to enjoy ourselves along the way.

Those of us whose work is to write software are incredibly lucky. Building software is a guiltless pleasure because we get to use our creative energy to get things done. We have arranged our lives to have it both ways; we can enjoy the pure act of writing code in sure knowledge that the code we write has use. We produce things that matter. We are modern craftspeople, building structures that make up present-day reality, and no less than bricklayers or bridge builders, we take justifiable pride in our accomplishments.

This all programmers share, from the most enthusiastic newbie to the apparently jaded elder, whether working at the lightest weight Internet startup or the most staid, long-entrenched enterprise. We want to do our best work. We want our work to have meaning. We want to have fun along the way.

And so it's especially troubling when software goes awry. Bad software impedes our purpose and interferes with our happiness. Where once we felt productive, now we feel thwarted. Where once fast, now slow. Where once peaceful, now frustrated.

This frustration occurs when it costs too much to get things done. Our internal calculators are always running, comparing total amount accomplished to overall effort expended. When the cost of doing work exceeds its value, our efforts feel wasted. If programming gives joy it is because it allows us to be useful, when it becomes painful it is a sign that we believe we could, and should, be doing more. Our pleasure follows in the footsteps of work.

This book is about designing object-oriented software. It is not an academic tome, it is a programmer's story about how to write code. It teaches how to arrange software so as to be productive today and to remain so next month and next year. It shows how to write applications that can succeed in the present and still adapt to the future. It allows you to raise your productivity and reduce your costs for the entire lifetime of your applications.

This book believes in your desire to do good work and gives you the tools you need to best be of use. It is completely practical and as such is, at its core, a book about how to write code that brings you joy.

Who Might Find This Book Useful?

This book assumes that you have at least tried to write object-oriented software. It is not necessary that you feel you succeeded, just that you made the attempt in any object-oriented (OO) language. Chapter 1, "Object-Oriented Design," contains a brief overview of object-oriented programming (OOP), but its goal is to define common terms, not to teach programming.

If you want to learn OO design (OOD) but have not yet done any object-oriented programming, at least take a tutorial before reading this book. OOD solves problems; suffering from those problems is very nearly a prerequisite for comprehending these solutions. Experienced programmers may be able to skip this step, but most readers will be happier if they write some OO code before starting this book.

This book uses Ruby to teach OOD but you do not need to know Ruby to understand the concepts herein. There are many code examples but all are quite straightforward. If you have programmed in any OO language you will find Ruby easy to understand.

If you come from a statically typed OO language like Java or C++ you have the background necessary to benefit from reading this book. The fact that Ruby is dynamically typed simplifies the syntax of the examples and distills the design ideas to their essence, but every concept in this book can be directly translated to a statically typed OO language.

How to Read This Book

Chapter 1 contains a general overview of the whys, whens, and wherefores of OO design, followed by a brief overview of object-oriented programming. This chapter stands alone. You can read it first, last, or, frankly, skip it entirely, although if you are currently stuck with an application that suffers from lack of design, you may find it a comforting tale.

If you have experience writing object-oriented applications and want to jump right in, you can safely start with Chapter 2. If you do so and then stumble upon an unfamiliar term, come back and browse the "Introduction to Object-Oriented Programming" section of Chapter 1, which introduces and defines common OO terms used throughout the book.

Chapters 2 through 9 progressively explain object-oriented design. Chapter 2, "Designing Classes with a Single Responsibility," covers how to decide what belongs in a single class. Chapter 3, "Managing Dependencies," illustrates how objects get entangled with one another and shows how to keep them apart. These two chapters are focused on objects rather than messages.

In Chapter 4, "Creating Flexible Interfaces," the emphasis begins to shift away from object-centric toward message-centric design. Chapter 4 is about defining interfaces and is concerned with how objects talk to one another. Chapter 5, "Reducing Costs with Duck Typing," is about duck typing and introduces the idea that objects of different classes may play common roles. Chapter 6, "Acquiring Behavior through Inheritance," teaches the techniques of classical inheritance, which are then used in Chapter 7, "Sharing Role Behavior with Modules," to create duck typed roles. Chapter 8, "Combining Objects with Composition," explains the technique of building objects via composition and provides guidelines for choosing among composition, inheritance, and duck-typed role sharing. Chapter 9, "Designing Cost-Effective Tests," concentrates on the design of tests, which it illustrates using code from earlier chapters of the book.

Each of these chapters builds on the concepts of the last. They are full of code and best read in order.

How to Use This Book

This book will mean different things to readers of different backgrounds. Those already familiar with OOD will find things to think about, possibly encounter some new points of view, and probably disagree with a few of the suggestions. Because there is no final authority on OOD, challenges to the principles (and to this author) will improve the understanding of all. In the end, you must be the arbiter of your own designs; it is up to you to question, to experiment, and to choose.

While this book should be of interest to many levels of reader, it is written with the particular goal of being accessible to novices. If you are one of those novices, this part of the introduction is especially for you. Know this: Object-oriented design is not black magic. It is simply things you don't yet know. The fact that you've read this far indicates you care about design; this desire to learn is the only prerequisite for benefiting from this book.

Chapters 2 through 9 explain OOD principles and provide very explicit programming rules; these rules will mean different things to novices than they mean to experts. If you are a novice, start out by following these rules in blind faith if necessary. This early obedience will stave off disaster until you can gain enough experience to make

your own decisions. By the time the rules start to chafe, you'll have enough experience to make up rules of your own, and your career as a designer will have begun.

Software Versions Used in This Book

The examples in this book were written using Ruby 2.4 and tested with Minitest 5.10.3. Source code for the examples can be found at https://github.com/skmetz/poodr2.

Acknowledgments

It is a wonder this book exists; the fact that it does is due to the efforts and encouragement of many people.

Throughout the long process of writing, Lori Evans and TJ Stankus provided early feedback on every chapter. They live in Durham, NC, and thus could not escape me, but this fact does nothing to lessen my appreciation for their help.

Midway through the book, after it became impossible to deny that its writing would take approximately twice as long as originally estimated, Mike Dalessio and Gregory Brown read drafts and gave invaluable feedback and support. Their encouragement and enthusiasm kept the project alive during dark days.

A number of reviewers cast their keen eyes on the entire book, acting as gracious stand-ins for you, the gentle reader. As the first edition neared completion, Steve Klabnik, Desi McAdam, and Seth Wax gave it careful readings. The second edition was meticulously scoured by Will Sommers and Tory Peterschild. Their impressions and suggestions caused changes that will benefit all who follow.

Late drafts were given careful, thorough readings by Katrina Owen, Avdi Grimm, and Rebecca Wirfs-Brock, and the book is much improved by their kind and thoughtful feedback. Before they pitched in, Katrina, Avdi, and Rebecca were strangers to me; I am grateful for their involvement and humbled by their generosity. If you find this book useful, thank them when you next see them.

I am also grateful for the Gotham Ruby Group and for everyone who expressed their appreciation for the design talks I gave at GoRuCo 2009 and 2011. The folks at GoRuCo took a chance on an unknown and gave me a forum in which to express these ideas; this book started there. Ian McFarland and Brian Ford watched those talks and their immediate and ongoing enthusiasm for this project was both infectious and convincing.

The process of writing was greatly aided by Michael Thurston of Pearson, who was like an ocean liner of calmness and organization chugging through the chaotic sea of my opposing rogue writing waves. You can, I expect, see the problem he faced. He insisted, with endless patience and grace, that the writing be arranged in a readable structure. I believe his efforts have paid off and hope you will agree.

My thanks also to Debra Williams Cauley, my editor at Pearson/Addison-Wesley, who overheard an ill-timed hallway rant in 2006 at the first Ruby on Rails conference in Chicago and launched the campaign that eventually resulted in this book. Despite my best efforts, she would not take no for an answer. She cleverly moved from one argument to the next until she finally found the one that convinced; this accurately reflects her persistence and dedication.

I owe a debt to the entire object-oriented design community. I did not make up the ideas in this book, I am merely a translator, and I stand on the shoulders of giants. It goes without saying that while all credit for these ideas belongs to others—failures of translation are mine alone.

And finally, this book owes its existence to my partner Amy Germuth. Before this project started I could not imagine writing a book; her view of the world as a place where people did such things made doing so seem possible. The book in your hands is a tribute to her boundless patience and endless support.

Thank you, each and every one.

About the Author

Sandi Metz, author of *Practical Object-Oriented Design in Ruby* and *99 Bottles of OOP*, believes in simple code and straightforward explanations. She prefers working software, practical solutions, and lengthy bicycle trips (not necessarily in that order), and writes, consults, speaks, and teaches about object-oriented design.

This page intentionally left blank

CHAPTER 3
Managing Dependencies

Object-oriented programming languages contend that they are efficient and effective because of the way they model reality. Objects reflect qualities of a real-world problem and the interactions between those objects provide solutions. These interactions are inescapable. A single object cannot know everything, so inevitably it will have to talk to another object.

If you could peer into a busy application and watch the messages as they pass, the traffic might seem overwhelming. There's a lot going on. However, if you stand back and take a global view, a pattern becomes obvious. Each message is initiated by an object to invoke some bit of behavior. All of the behavior is dispersed among the objects. Therefore, for any desired behavior, an object either knows it personally, inherits it, or knows another object who knows it.

The previous chapter concerned itself with the first of these, that is, behaviors that a class should personally implement. The second, inheriting behavior, will be covered in Chapter 6, "Acquiring Behavior through Inheritance." This chapter is about the third, getting access to behavior when that behavior is implemented in *other* objects.

Because well-designed objects have a single responsibility, their very nature requires that they collaborate to accomplish complex tasks. This collaboration is powerful and perilous. To collaborate, an object must know something about others. *Knowing* creates a dependency. If not managed carefully, these dependencies will strangle your application.

3.1 Understanding Dependencies

An object depends on another object if, when one object changes, the other might be forced to change in turn.

Here's a modified version of the `Gear` class, where `Gear` is initialized with four familiar arguments. The `gear_inches` method uses two of them, `rim` and `tire`, to create a new instance of `Wheel`. `Wheel` has not changed since you last saw it in Chapter 2, "Designing Classes with a Single Responsibility."

Listing 3.1

```
class Gear
  attr_reader :chainring, :cog, :rim, :tire
  def initialize(chainring, cog, rim, tire)
    @chainring = chainring
    @cog       = cog
    @rim       = rim
    @tire      = tire
  end

  def gear_inches
    ratio * Wheel.new(rim, tire).diameter
  end

  def ratio
    chainring / cog.to_f
  end
# ...
end

class Wheel
  attr_reader :rim, :tire
  def initialize(rim, tire)
    @rim  = rim
    @tire = tire
  end

  def diameter
    rim + (tire * 2)
  end
# ...
end

puts Gear.new(52, 11, 26, 1.5).gear_inches
# => 137.0909090909091
```

Examine the preceding code and make a list of the situations in which `Gear` would be forced to change because of a change to `Wheel`. This code seems innocent, but it's sneakily complex. `Gear` has at least four dependencies on `Wheel`, enumerated as follows. Most of the dependencies are unnecessary; they are a side effect of the coding style. `Gear` does not need them to do its job. Their very existence weakens `Gear` and makes it harder to change.

3.1.1 Recognizing Dependencies

An object has a dependency when it knows:

- The name of another class. `Gear` expects a class named `Wheel` to exist.
- The name of a message that it intends to send to someone other than `self`. `Gear` expects a `Wheel` instance to respond to `diameter`.
- The arguments that a message requires. `Gear` knows that `Wheel.new` requires a `rim` and a `tire`.
- The order of those arguments. `Gear` knows that `Wheel` takes *positional* arguments and that the first should be `rim`, the second, `tire`.

Each of these dependencies creates a chance that `Gear` will be forced to change because of a change to `Wheel`. Some degree of dependency between these two classes is inevitable; after all, they *must* collaborate, but most of the dependencies listed above are unnecessary. These unnecessary dependencies make the code less *reasonable.* Because they increase the chance that `Gear` will be forced to change, these dependencies turn minor code tweaks into major undertakings where small changes cascade through the application, forcing many changes.

Your design challenge is to manage dependencies so that each class has the fewest possible; a class should know just enough to do its job and not one thing more.

3.1.2 Coupling Between Objects (CBO)

These dependencies *couple* `Gear` to `Wheel`. Alternatively, you could say that each coupling *creates* a dependency. The more `Gear` knows about `Wheel`, the more tightly coupled they are. The more tightly coupled two objects are, the more they behave like a single entity.

If you make a change to `Wheel`, you may find it necessary to make a change to `Gear`. If you want to reuse `Gear`, `Wheel` comes along for the ride. When you test `Gear`, you'll be testing `Wheel` too.

Figure 3.1 illustrates the problem. In this case, `Gear` depends on `Wheel` and four other objects, coupling `Gear` to five different things. When the underlying code was

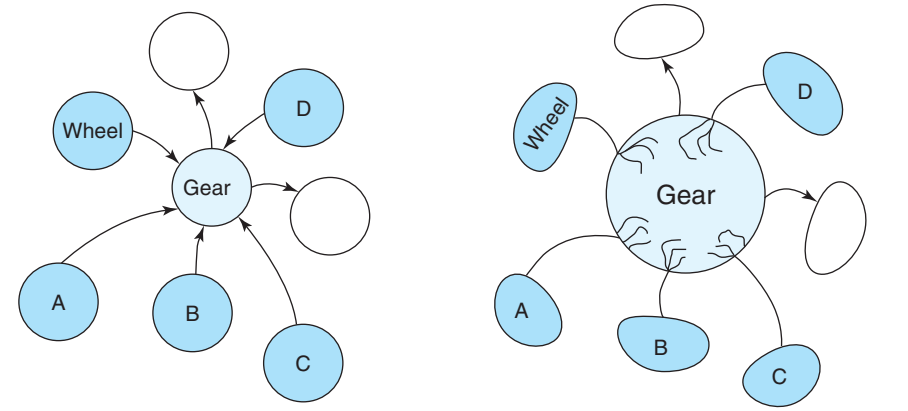

Figure 3.1 Dependencies entangle objects with one another

first written, everything worked fine. The problem lies dormant until you attempt to use `Gear` in another context or to change one of the classes upon which `Gear` depends. When that day comes, the cold hard truth is revealed; despite appearances, `Gear` is not an independent entity. Each of its dependencies is a place where another object is stuck to it. The dependencies cause these objects to act like a single thing. They move in lockstep; they change together.

When two (or three or more) objects are so tightly coupled that they behave as a unit, it's impossible to reuse just one. Changes to one object force changes to all. Left unchecked, unmanaged dependencies cause an entire application to become an entangled mess. A day will come when it's easier to rewrite everything than to change anything.

3.1.3 Other Dependencies

The remainder of this chapter examines the four kinds of dependencies listed previously and suggests techniques for avoiding the problems they create. However, before going forward, it's worth mentioning a few other common dependency-related issues that will be covered in other chapters.

One especially destructive kind of dependency occurs where an object knows another who knows another who knows something; that is, where many messages are chained together to reach behavior that lives in a distant object. This is the "knowing the name of a message you plan to send to someone other than *self*" dependency, only magnified. Message chaining creates a dependency between the original object and every object and message along the way to its ultimate target. These additional couplings greatly increase the chance that the first object will be forced to change because a change to *any* of the intermediate objects might affect it.

This case, a Law of Demeter violation, gets its own special treatment in Chapter 4, "Creating Flexible Interfaces."

Another entire class of dependencies is that of tests on code. In the world outside of this book, tests come first. They drive design. However, they refer to code and thus depend on code. The natural tendency of "new-to-testing" programmers is to write tests that are too tightly coupled to code. This tight coupling leads to incredible frustration; the tests break every time the code is refactored, even when the fundamental behavior of the code does not change. Tests begin to seem costly relative to their value. Test-to-code over-coupling has the same consequence as code-to-code over-coupling. These couplings are dependencies that cause changes to the code to cascade into the tests, forcing them to change in turn.

The design of tests is examined in Chapter 9, "Designing Cost-Effective Tests."

Despite these cautionary words, your application is not doomed to drown in unnecessary dependencies. As long as you recognize them, avoidance is quite simple. The first step to this brighter future is to understand dependencies in more detail; therefore, it's time to look at some code.

3.2 Writing Loosely Coupled Code

Every dependency is like a little dot of glue that causes your class to stick to the things it touches. A few dots are necessary, but apply too much glue, and your application will harden into a solid block. Reducing dependencies means recognizing and removing the ones you don't need.

The following examples illustrate coding techniques that reduce dependencies by decoupling code.

3.2.1 Inject Dependencies

Referring to another class by its name creates a major sticky spot. In the version of `Gear` we've been discussing (repeated in Listing 3.2), the `gear_inches` method contains an explicit reference to class `Wheel`.

Listing 3.2

```
1 class Gear
2   attr_reader :chainring, :cog, :rim, :tire
3   def initialize(chainring, cog, rim, tire)
4     @chainring = chainring
5     @cog       = cog
6     @rim       = rim
7     @tire      = tire
8   end
```

```
 9
10   def gear_inches
11     ratio * Wheel.new(rim, tire).diameter
12   end
13   # ...
14 end
15
16 puts Gear.new(52, 11, 26, 1.5).gear_inches
17 # => 137.0909090909091
```

The immediate, obvious consequence of this reference is that if the name of the `Wheel` class changes, `Gear`'s `gear_inches` method must also change.

On the face of it, this dependency seems innocuous. After all, if a `Gear` needs to talk to a `Wheel`, something, somewhere, must create a new instance of the `Wheel` class. If `Gear` itself knows the name of the `Wheel` class, the code in `Gear` must be altered if `Wheel`'s name changes.

In truth, dealing with the name change is a relatively minor issue. You likely have a tool that allows you to do a global find/replace within a project. If `Wheel`'s name changes to `Wheely`, finding and fixing all of the references isn't that hard. However, the fact that line 11 above must change if the name of the `Wheel` class changes is the least of the problems with this code. A deeper problem exists that is far less visible but significantly more destructive.

When `Gear` hard-codes a reference to `Wheel` deep inside its `gear_inches` method, it is explicitly declaring that it is only willing to calculate gear inches for instances of `Wheel`. `Gear` refuses to collaborate with any other kind of object, even if that object has a diameter and uses gears.

If your application expands to include objects such as disks or cylinders and you need to know the gear inches of gears which use them, *you cannot*. Despite the fact that disks and cylinders naturally have a diameter, you can never calculate their gear inches because `Gear` is stuck to `Wheel`.

The code above exposes an unjustified attachment to type. It is not the class of the object that's important, it's the *message* you plan to send to it. `Gear` needs access to an object that can respond to `diameter`; a duck type, if you will (see Chapter 5, "Reducing Costs with Duck Typing"). `Gear` does not care and should not know about the class of that object. It is not necessary for `Gear` to know about the existence of the `Wheel` class in order to calculate `gear_inches`. It doesn't need to know that `Wheel` expects to be initialized with a `rim` and then a `tire`; it just needs an object that knows `diameter`.

Hanging these unnecessary dependencies on `Gear` simultaneously reduces `Gear`'s reusability and increases its susceptibility to being forced to change unnecessarily. `Gear` becomes less useful when it knows too much about *other* objects; if it knew less, it could do more.

Instead of being glued to `Wheel`, this next version of `Gear` expects to be initialized with an object that can respond to `diameter`:

Listing 3.3

```
class Gear
  attr_reader :chainring, :cog, :wheel
  def initialize(chainring, cog, wheel)
    @chainring = chainring
    @cog       = cog
    @wheel     = wheel
  end

  def gear_inches
    ratio * wheel.diameter
  end
  # ...
end

# Gear expects a 'Duck' that knows 'diameter'
puts Gear.new(52, 11, Wheel.new(26, 1.5)).gear_inches
# => 137.0909090909091
```

`Gear` now uses the `@wheel` variable to hold, and the `wheel` method to access, this object, but don't be fooled: `Gear` doesn't know or care that the object might be an instance of class `Wheel`. `Gear` only knows that it holds an object that responds to `diameter`.

This change is so small it is almost invisible, but coding in this style has huge benefits. Moving the creation of the new `Wheel` instance outside of `Gear` decouples the two classes. `Gear` can now collaborate with any object that implements `diameter`. As an extra bonus, this benefit was free. Not only is the resulting `Gear` class smaller than the original, but the decoupling was achieved by simply rearranging existing code.

This technique is known as *dependency injection*. Despite its daunting reputation, dependency injection truly is this simple. `Gear` previously had explicit dependencies on the `Wheel` class and on the type and order of its initialization arguments, but through injection these dependencies have been reduced to a single dependency on the `diameter` method. `Gear` is now smarter because it knows less.

Using dependency injection to shape code relies on your ability to recognize that the responsibility for knowing the name of a class and the responsibility for knowing the name of a message to send to that class may belong in different objects. Just because `Gear` needs to send `diameter` somewhere does not mean that `Gear` should know about `Wheel`.

This leaves the question of where the responsibility for knowing about the actual `Wheel` class lies; the example above conveniently sidesteps this issue, but it is examined in more detail later in this chapter. For now, it's enough to understand that this knowledge does *not* belong in `Gear`.

3.2.2 Isolate Dependencies

It's best to break all unnecessary dependencies but, unfortunately, while this is always *technically* possible, it may not be *actually* possible. When working on an existing application, you may find yourself under severe constraints about how much you can actually change. If prevented from achieving perfection, your goals should switch to improving the overall situation by leaving the code better than you found it.

Therefore, if you cannot remove unnecessary dependencies, you should isolate them within your class. In Chapter 2, you isolated extraneous responsibilities so that they would be easy to recognize and remove when the right impetus came; here you should isolate unnecessary dependencies so that they are easy to spot and reduce when circumstances permit.

Think of every dependency as an alien bacterium that's trying to infect your class. Give your class a vigorous immune system; quarantine each dependency. Dependencies are foreign invaders that represent vulnerabilities, and they should be concise, explicit, and isolated.

Isolate Instance Creation

If you are so constrained that you cannot change the code to inject a `Wheel` into a `Gear`, you should isolate the creation of a new `Wheel` inside the `Gear` class. The intent is to explicitly expose the dependency while reducing its reach into your class.

The next two examples illustrate this idea.

In the first, creation of the new instance of `Wheel` has been moved from `Gear`'s `gear_inches` method to `Gear`'s initialization method. This cleans up the `gear_inches` method and publicly exposes the dependency in the `initialize` method. Notice that this technique unconditionally creates a new `Wheel` each time a new `Gear` is created.

Listing 3.4

```
1 class Gear
2   attr_reader :chainring, :cog, :wheel
3   def initialize(chainring, cog, rim, tire)
4     @chainring = chainring
5     @cog       = cog
6     @wheel     = Wheel.new(rim, tire)
```

```
  end

  def gear_inches
    ratio * wheel.diameter
  end
  # ...
  end
end

puts Gear.new(52, 11, 26, 1.5).gear_inches
# => 137.0909090909091
```

The next alternative isolates creation of a new `Wheel` in its own explicitly defined `wheel` method. This new method lazily creates a new instance of `Wheel`, using Ruby's `||=` operator. In this case, creation of a new instance of `Wheel` is deferred until `gear_inches` invokes the new `wheel` method.

Listing 3.5

```
class Gear
  attr_reader :chainring, :cog, :rim, :tire
  def initialize(chainring, cog, rim, tire)
    @chainring = chainring
    @cog       = cog
    @rim       = rim
    @tire      = tire
  end

  def gear_inches
    ratio * wheel.diameter
  end

  def wheel
    @wheel ||= Wheel.new(rim, tire)
  end
  # ...
end

puts Gear.new(52, 11, 26, 1.5).gear_inches
# => 137.0909090909091
```

In both of these examples, `Gear` still knows far too much; it still takes `rim` and `tire` as initialization arguments, and it still creates its own new instance of `Wheel`. `Gear` is still stuck to `Wheel`; it can calculate the gear inches of no other kind of object.

However, an improvement *has* been made. These coding styles reduce the number of dependencies in `gear_inches` while publicly exposing `Gear`'s dependency on `Wheel`. They reveal dependencies instead of concealing them, lowering the barriers to reuse and making the code easier to refactor when circumstances allow. This change makes the code more agile; it can more easily adapt to the unknown future.

The way you manage dependencies on external class names has profound effects on your application. If you are mindful of dependencies and develop a habit of routinely injecting them, your classes will naturally be loosely coupled. If you ignore this issue and let the class references fall where they may, your application will be more like a big woven mat than a set of independent objects. An application whose classes are sprinkled with entangled and obscure class name references is unwieldy and inflexible, while one whose class name dependencies are concise, explicit, and isolated can easily adapt to new requirements.

Isolate Vulnerable External Messages

Now that you've isolated references to external class names, it's time to turn your attention to external *messages*, that is, messages that are "sent to someone other than `self`." For example, the `gear_inches` method below sends `ratio` and `wheel` to `self` but sends `diameter` to `wheel`:

Listing 3.6

```
def gear_inches
  ratio * wheel.diameter
end
```

This is a simple method and it contains `Gear`'s only reference to `wheel.diameter`. In this case, the code is fine, but the situation could be more complex. Imagine that calculating `gear_inches` required far more math and that the method looked something like this:

Listing 3.7

```
def gear_inches
  #... a few lines of scary math
  foo = some_intermediate_result * wheel.diameter
  #... more lines of scary math
end
```

Now `wheel.diameter` is embedded deeply inside a complex method. This complex method depends on `Gear` responding to `wheel` and on `wheel` responding to `diameter`. Embedding this external dependency inside the `gear_inches` method is unnecessary and increases its vulnerability.

Any time you change *anything*, you stand the chance of breaking it; `gear_inches` is now a complex method, and that makes it both more likely to need changing and more susceptible to being damaged when it does. You can reduce your chance of being forced to make a change to `gear_inches` by removing the external dependency and encapsulating it in a method of its own, as in this next example:

Listing 3.8

```
def gear_inches
  #... a few lines of scary math
  foo = some_intermediate_result * diameter
  #... more lines of scary math
end

def diameter
  wheel.diameter
end
```

The new `diameter` method is exactly the method that you would have written if you had many references to `wheel.diameter` sprinkled throughout `Gear` and you wanted to DRY them out. The difference here is one of timing; it would normally be defensible to defer creation of the `diameter` method until you had a need to DRY out code; however, in this case, the method is created preemptively to remove the dependency from `gear_inches`.

In the original code, `gear_inches` knew that `wheel` had a `diameter`. This knowledge is a dangerous dependency that couples `gear_inches` to an external object and one of *its* methods. After this change, `gear_inches` is more abstract. `Gear` now isolates `wheel.diameter` in a separate method, and `gear_inches` can depend on a message sent to `self`.

If `Wheel` changes the name or signature of *its* implementation of `diameter`, the side effects to `Gear` will be confined to this one simple wrapping method.

This technique becomes necessary when a class contains embedded references to a *message* that is likely to change. Isolating the reference provides some insurance against being affected by that change. Although not every external method is a candidate for this preemptive isolation, it's worth examining your code, looking for and wrapping the most vulnerable dependencies.

An alternative way to eliminate these side effects is to avoid the problem from the very beginning by reversing the direction of the dependency. This idea will be addressed soon, but first there's one more coding technique to cover.

3.2.3 Remove Argument-Order Dependencies

When you send a message that requires arguments, you, as the sender, cannot avoid having knowledge of those arguments. This dependency is unavoidable. However, passing arguments often involves a second, more subtle dependency. Many method signatures not only require arguments, but they also require that those arguments be passed in a specific, fixed order.

In the following example, `Gear`'s `initialize` method takes three arguments: `chainring`, `cog`, and `wheel`. It provides no defaults; each of these arguments is required. In lines 11–14, when a new instance of Gear is created, the three arguments must be passed and they must be passed *in the correct order*.

Listing 3.9

```
1  class Gear
2    attr_reader :chainring, :cog, :wheel
3    def initialize(chainring, cog, wheel)
4      @chainring = chainring
5      @cog       = cog
6      @wheel     = wheel
7    end
8    # ...
9  end
10
11 puts Gear.new(
12       52,
13       11,
14       Wheel.new(26, 1.5)).gear_inches
15 # => 137.0909090909091
```

Senders of `new` depend on the order of the arguments as they are specified in `Gear`'s `initialize` method. If that order changes, all the senders will be forced to change.

Unfortunately, it's quite common to tinker with initialization arguments. Especially early on, when the design is not quite nailed down, you may go through several cycles of adding and removing arguments and defaults. If you use positional arguments, each of these cycles may force changes to many dependents. Even worse, you may find yourself avoiding making changes to the arguments, even when your design calls for them, because you can't bear to change all the dependents yet again.

Use Keyword Arguments

There's a simple way to avoid depending on positional arguments. If you have control over `Gear`'s `initialize` method, change the code to take *keyword* arguments.

The following example illustrates this technique:

Listing 3.10

```
1 class Gear
2   attr_reader :chainring, :cog, :wheel
3   def initialize(chainring:, cog:, wheel:)
4     @chainring = chainring
5     @cog       = cog
6     @wheel     = wheel
7   end
8   # ...
9 end
```

The arguments on line 3 now end in `:`, which denotes that they are keyword arguments. Keyword arguments are referenced just like positional arguments, so lines 4–6 have not changed.

You can pass keyword arguments as a hash, as shown in the following example:

Listing 3.11

```
1 puts Gear.new(
2       :cog       => 11,
3       :chainring => 52,
4       :wheel     => Wheel.new(26, 1.5)).gear_inches
5 # => 137.0909090909091
```

You can also use the explicit keyword syntax:

Listing 3.12

```
1 puts Gear.new(
2       wheel:     Wheel.new(26, 1.5),
3       chainring: 52,
4       cog:       11).gear_inches
5 # => 137.0909090909091
```

Keyword arguments offer several advantages. As you likely noticed in two examples above, keyword arguments may be passed in any order. Additionally, `Gear` is now free to add or remove initialization arguments and defaults, secure in the knowledge that no change will have side effects in other code.

This technique adds verbosity. In many situations verbosity is a detriment, but in this case, it has value. The verbosity exists at the intersection between the needs of the present and the uncertainty of the future. Using positional arguments requires less code today, but you pay for this decrease in volume of code with an increase in the risk that changes will cascade into dependents later.

When `Gear` switched to keyword arguments, it lost its dependency on argument order but it gained a dependency on the names of the keywords. This change is healthy. The new dependency is more stable than the old, and thus this code faces less risk of being forced to change.

Using keyword arguments requires the sender *and* the receiver of a message to state the keyword names. This results in explicit documentation at both ends of the message. Future maintainers will be grateful for this information.

Keyword arguments are so flexible that the general rule is that you should *prefer* them. While it's certainly true that some argument lists are so stable, and so obvious, that keywords are overkill (for example, what would `Point` take but an `x` and a `y`?), your bias should be toward declaring arguments using keywords. You can always fall back to positional arguments if that technique better suits your specific problem.

Also, it is perfectly acceptable for some classes in your application to take positional arguments and others to take keyword arguments. This is especially true in long-lived applications, where much of the code predates the introduction of keywords. In these cases, as you change or add code, consider using keyword arguments. However, there's no need to proactively retrofit the entire application. Over time, as you touch code, introduce keyword arguments if doing so will add clarity and enable subsequent refactorings.

The remainder of this book uses both types, to supply a flavor of the consequences.

Explicitly Define Defaults

So far, keyword arguments look very similar to hashes. One advantage they have over hashes, however, is that they allow you to set defaults right in the argument list, just like positional arguments. Line 3 below supplies defaults for `chainring` and `cog`.

Listing 3.13

```
1 class Gear
2   attr_reader :chainring, :cog, :wheel
3   def initialize(chainring: 40, cog: 18, wheel:)
4     @chainring = chainring
5     @cog       = cog
6     @wheel     = wheel
7   end
```

```
  # ...
end

puts Gear.new(wheel: Wheel.new(26, 1.5)).chainring
# => 40
```

Notice that the syntax for adding defaults to keyword arguments is a bit different than that of positional arguments. Keywords omit the = operator and state the default directly after the trailing :. Adding a default renders the keyword argument optional.

The above syntax is great for supplying simple defaults to optional arguments, but some situations may benefit from a bit more sophistication. For example, line 3 below sets a more complex default by sending a message.

Listing 3.14

```
1  class Gear
2    attr_reader :chainring, :cog, :wheel
3    def initialize(chainring: default_chainring, cog: 18,
     wheel:)
4      @chainring = chainring
5      @cog       = cog
6      @wheel     = wheel
7    end
8
9    def default_chainring
10     (100/2) - 10        # silly code, useful example
11   end
12   # ...
13 end
14
15 puts Gear.new(wheel: Wheel.new(26, 1.5)).chainring
16 # => 40
17
18 puts Gear.new(chainring: 52, wheel: Wheel.new(26, 1.5)).
   chainring
19 # => 52
```

The key to understanding the above code is to recognize that `initialize` executes in the new instance of `Gear`. It is therefore entirely appropriate for `initialize` to send messages to `self`. It's best to embed simple defaults right in the parameter list, but if getting the default requires running a bit of code, don't hesitate to send a message.

Isolate Multiparameter Initialization

So far, all of the examples of removing argument-order dependencies have been for situations where *you* control the signature of the method that needs to change. You will not always have this luxury; sometimes you will be forced to depend on a method that requires positional arguments where you do not own and thus cannot change the method itself.

Imagine that `Gear` is part of a framework and that its initialization method requires positional arguments. Imagine also that your code has many places where you must create a new instance of `Gear`. `Gear`'s `initialize` method is *external* to your application; it is part of an interface over which you have no control.

As dire as this situation appears, you are not doomed to accept the dependencies. Just as you would DRY out repetitive code inside of a class, DRY out the creation of new `Gear` instances by creating a single method to wrap the external interface. The classes in your application should depend on code that you own; use a wrapping method to isolate external dependencies.

In this example, the `SomeFramework::Gear` class is not owned by your application; it is part of an external framework. Its initialization method requires positional arguments. The `GearWrapper` module was created to avoid having multiple dependencies on the order of those arguments. `GearWrapper` isolates all knowledge of the external interface in one place and, equally important, it provides an improved interface for your application.

As you can see in line 22, `GearWrapper` allows your application to create a new instance of `Gear` using keyword arguments.

Listing 3.15

```
# When Gear is part of an external interface
module SomeFramework
  class Gear
    attr_reader :chainring, :cog, :wheel
    def initialize(chainring, cog, wheel)
      @chainring = chainring
      @cog       = cog
      @wheel     = wheel
    end
    # ...
  end
end

# wrap the interface to protect yourself from changes
module GearWrapper
  def self.gear(chainring:, cog:, wheel:)
```

```
17      SomeFramework::Gear.new(chainring, cog, wheel)
18    end
19  end
20
21  # Now you can create a new Gear using keyword arguments
22  puts GearWrapper.gear(
23    chainring: 52,
24    cog:       11,
25    wheel:     Wheel.new(26, 1.5)).gear_inches
26  # => 137.0909090909091
```

There are two things to note about `GearWrapper`. First, it is a Ruby module instead of a class (line 15). `GearWrapper` is responsible for creating new instances of `SomeFramework::Gear`. Using a module here lets you define a separate and distinct object to which you can send the `gear` message (line 22) while simultaneously conveying the idea that you don't expect to have instances of `GearWrapper`. You may already have experience with including modules into classes; in the example above, `GearWrapper` is not meant to be included in another class, it's meant to directly respond to the `gear` message.

The other interesting thing about `GearWrapper` is that its sole purpose is to create instances of some other class. Object-oriented designers have a word for objects like this; they call them *factories*. In some circles, the term factory has acquired a negative connotation, but the term as used here is devoid of baggage. An object whose purpose is to create other objects is a factory; the word factory implies nothing more, and use of it is the most expedient way to communicate this idea.

The above technique for replacing positional arguments with keywords is perfect for cases where you are forced to depend on external interfaces that you cannot change. Do not allow these kinds of external dependencies to permeate your code; protect yourself by wrapping each in a method that is owned by your own application.

3.3 Managing Dependency Direction

Dependencies always have a direction; earlier in this chapter it was suggested that one way to manage them is to reverse that direction. This section delves more deeply into how to decide on the direction of dependencies.

3.3.1 Reversing Dependencies

Every example used thus far shows `Gear` depending on `Wheel` or `diameter`, but the code could easily have been written with the direction of the dependencies reversed.

`Wheel` could instead depend on `Gear` or `ratio`. The following example illustrates one possible form of the reversal. Here `Wheel` has been changed to depend on `Gear` and `gear_inches`. `Gear` is still responsible for the actual calculation, but it expects a `diameter` argument to be passed in by the caller (line 8).

Listing 3.16

```
1  class Gear
2    attr_reader :chainring, :cog
3    def initialize(chainring:, cog:)
4      @chainring = chainring
5      @cog       = cog
6    end
7
8    def gear_inches(diameter)
9      ratio * diameter
10   end
11
12   def ratio
13     chainring / cog.to_f
14   end
15   # ...
16 end
17
18 class Wheel
19   attr_reader :rim, :tire, :gear
20   def initialize(rim:, tire:, chainring:, cog:)
21     @rim  = rim
22     @tire = tire
23     @gear = Gear.new(chainring: chainring, cog: cog)
24   end
25
26   def diameter
27     rim + (tire * 2)
28   end
29
30   def gear_inches
31     gear.gear_inches(diameter)
32   end
33   # ...
34 end
35
36 puts Wheel.new(
```

```
        rim:       26,
        tire:      1.5,
        chainring: 52,
        cog:       11).gear_inches
# => 137.0909090909091
```

This reversal of dependencies does no apparent harm. Calculating `gear_inches` still requires collaboration between `Gear` and `Wheel` and the result of the calculation is unaffected by the reversal. One could infer that the direction of the dependency does not matter, that it makes no difference whether `Gear` depends on `Wheel` or vice versa.

Indeed, in an application that never changed, your choice would not matter. However, your application *will* change, and it's in that dynamic future where this present decision has repercussions. The choices you make about the direction of dependencies have far-reaching consequences that manifest themselves for the life of your application. If you get this right, your application will be pleasant to work on and easy to maintain. If you get it wrong, then the dependencies will gradually take over and the application will become harder and harder to change.

3.3.2 Choosing Dependency Direction

Pretend for a moment that your classes are people. If you were to give them advice about how to behave, you would tell them to *depend on things that change less often than you do*.

This short statement belies the sophistication of the idea, which is based on three simple truths about code:

- Some classes are more likely than others to have changes in requirements.
- Concrete classes are more likely to change than abstract classes.
- Changing a class that has many dependents will result in widespread consequences.

There are ways in which these truths intersect, but each is a separate and distinct notion.

Understanding Likelihood of Change

The idea that some classes are more likely to change than others applies not only to the code that you write for your own application but also to the code that you use but did *not* write. The Ruby base classes and the other framework code that you rely on both have their own inherent likelihood of change.

You are fortunate in that Ruby base classes change a great deal less often than your own code. This makes it perfectly reasonable to depend on the * method, as `gear_inches` quietly does, or to expect that Ruby classes `String` and `Array` will continue to work as they always have. Ruby base classes always change less often than your own classes, and you can continue to depend on them without another thought.

Framework classes are another story; only you can assess how mature your frameworks are. In general, any framework you use will be more stable than the code you write, but it's certainly possible to choose a framework that is undergoing such rapid development that its code changes more often than yours.

Regardless of its origin, every class used in your application can be ranked along a scale of how likely it is to undergo a change relative to all other classes. This ranking is one key piece of information to consider when choosing the direction of dependencies.

Recognizing Concretions and Abstractions

The second idea concerns itself with the concreteness and abstractness of code. The term *abstract* is used here just as Merriam-Webster defines it, as "disassociated from any specific instance," and, as so many things in Ruby, represents an idea about code as opposed to a specific technical restriction.

This concept was illustrated earlier in the chapter during the section on injecting dependencies. There, when `Gear` depended on `Wheel` and on `Wheel.new` and on `Wheel.new(rim, tire)`, it depended on extremely concrete code. After the code was altered to inject a `Wheel` into `Gear`, `Gear` suddenly began to depend on something far more abstract, that is, the fact that it had access to an object that could respond to the `diameter` message.

Your familiarity with Ruby may lead you to take this transition for granted, but consider for a moment what would have been required to accomplish this same trick in a statically typed language. Because statically typed languages have compilers that act like unit tests for types, you would not be able to inject just any random object into `Gear`. Instead you would have to declare an *interface*, define `diameter` as part of that interface, include the interface in the `Wheel` class, and tell `Gear` that the class you are injecting is a *kind of* that interface.

Rubists are justifiably grateful to avoid these gyrations, but languages that force you to be explicit about this transition do offer a benefit. They make it inescapably and explicitly clear that you are defining an abstract interface. It is impossible to create an abstraction unknowingly or by accident; in statically typed languages, defining an interface is *always* intentional.

In Ruby, when you inject `Wheel` into `Gear` such that `Gear` then depends on a *Duck* who responds to `diameter`, you are, however casually, defining an interface. This interface is an abstraction of the idea that a certain category of things will have a diameter. The abstraction was harvested from a concrete class; the idea is now "disassociated from any specific instance."

The wonderful thing about abstractions is that they represent common, stable qualities. They are less likely to change than are the concrete classes from which they were extracted. Depending on an abstraction is always safer than depending on a concretion because by its very nature, the abstraction is more stable. Ruby does not make you explicitly declare the abstraction in order to define the interface, but for design purposes, you can behave as if your virtual interface is as real as a class. Indeed, in the rest of this discussion, the term *class* stands for both *class* and this kind of *interface*. These interfaces can have dependents and so must be taken into account during design.

Avoiding Dependent-Laden Classes

The final idea, the notion that having dependent-laden objects has many consequences, also bears deeper examination. The consequences of changing a dependent-laden class are quite obvious—not so apparent are the consequences of even *having* a dependent-laden class. A class that, if changed, will cause changes to ripple through the application will be under enormous pressure to *never* change. Ever. Under any circumstances whatsoever. Your application may be permanently handicapped by your reluctance to pay the price required to make a change to this class.

Finding the Dependencies That Matter

Imagine each of these truths as a continuum along which all application code falls. Classes vary in their likelihood of change, their level of abstraction, and their number of dependents. Each quality matters, but the interesting design decisions occur at the place where *likelihood of change* intersects with *number of dependents*. Some of the possible combinations are healthy for your application; others are deadly.

Figure 3.2 summarizes the possibilities.

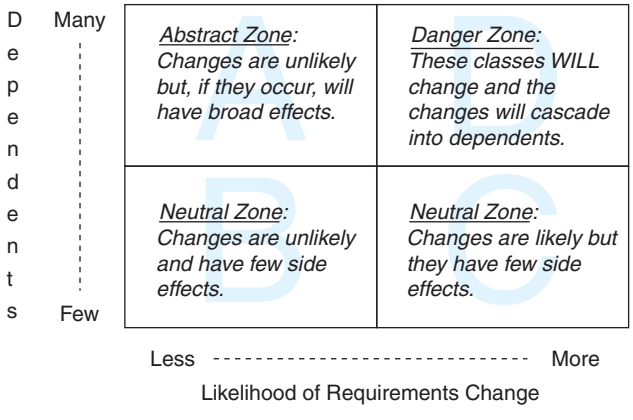

Figure 3.2 Likelihood of change versus number of dependents

The likelihood of requirements change is represented on the horizontal axis. The number of dependents is on the vertical. The grid is divided into four zones, labeled A through D. If you evaluate all of the classes in a well-designed application and place them on this grid, they will cluster in Zones A, B, and C.

Classes that have little likelihood of change but contain many dependents fall into Zone A. This zone usually contains abstract classes or interfaces. In a thoughtfully designed application, this arrangement is inevitable; dependencies cluster around abstractions because abstractions are less likely to change.

Notice that classes do not become abstract because they are in Zone A; instead they wind up here precisely because they are *already* abstract. Their abstract nature makes them more stable and allows them to safely acquire many dependents. While residence in Zone A does not guarantee that a class is abstract, it certainly suggests that it ought to be.

Skipping Zone B for a moment, Zone C is the opposite of Zone A. Zone C contains code that is quite likely to change but has few dependents. These classes tend to be more concrete, which makes them more likely to change, but this doesn't matter because few other classes depend on them.

Zone B classes are of the least concern during design because they are almost neutral in their potential future effects. They rarely change and have few dependents.

Zones A, B, and C are legitimate places for code; Zone D, however, is aptly named the Danger Zone. A class ends up in Zone D when it is guaranteed to change *and* has many dependents. Changes to Zone D classes are costly; simple requests become coding nightmares as the effects of every change cascade through each dependent. If you have a very specific *concrete* class that has many dependents and you believe it resides in Zone A, that is, you believe it is unlikely to change, think again. When a concrete class has many dependents, your alarm bells should be ringing. That class might actually be an occupant of Zone D.

Zone D classes represent a danger to the future health of the application. These are the classes that make an application painful to change. When a simple change has cascading effects that force many other changes, a Zone D class is at the root of the problem. When a change breaks some far away and seemingly unrelated bit of code, the design flaw originated here.

As depressing as this is, there is actually a way to make things worse. You can guarantee that any application will gradually become unmaintainable by making its Zone D classes *more* likely to change than their dependents. This maximizes the consequences of every change.

Fortunately, understanding this fundamental issue allows you to take preemptive action to avoid the problem.

Depend on things that change less often than you do is a heuristic that stands in for all the ideas in this section. The zones are a useful way to organize your thoughts, but in the fog of development, it may not be obvious which classes go where. Very

often you are exploring your way to a design, and at any given moment the future is unclear. Following this simple rule of thumb at every opportunity will cause your application to evolve a healthy design.

3.4 Summary

Dependency management is core to creating future-proof applications. Injecting dependencies creates loosely coupled objects that can be reused in novel ways. Isolating dependencies allows objects to quickly adapt to unexpected changes. Depending on abstractions decreases the likelihood of facing these changes.

The key to managing dependencies is to control their direction. The road to maintenance nirvana is paved with classes that depend on things that change less often than they do.

This page intentionally left blank

Index

Credits

Chapter 1, “a way to produce cheaper and higher quality software”: Laing, Victor & Coleman, Charles. (2001). Principal Components of Orthogonal Object-Oriented Metrics (323-08-14).

Chapter 1, “simple and elegant solutions to specific problems in object-oriented software design”, “make your own designs more flexible, modular, reusable and understandable”: Gamma, E., et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley.

Chapter 2, “A class has responsibilities that fulfill its purpose”: Rebecca Wirfs-Brock; Brian Wilkerson’s idea, Responsibility-Driven Design (RDD).

Chapter 2, “a convenient way to bundle a number of attributes together, using accessor methods, without having to write an explicit class.”: http://ruby-doc.org/core/classes/Struct.html

Chapter 3, abstract: “disassociated from any specific instance”: Merriam-Webster.

Chapter 8, “Inheritance is specialization”: Bertrand Meyer, Touch of Class: Learning to Program Well with Objects and Contracts.

Chapter 8, “Use composition when…sum of its parts” Booch, G., et al. (2007). Object-Oriented Analysis and Design with Applications. Upper Saddle River: Addison-Wesley.

Chapter 8, “Inheritance is best…small amounts of new code”: Gamma, E., et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley.

Chapter 9, “Refactoring is the process…improves the internal structure”: Fowler, M., et al. (1999). Refactoring: Improving the Design of Existing Code. Reading, MA: Addison-Wesley.

Cover: Wheels of Progress series. Backdrop of gears and fractal radial elements on the subject of science, technology and education. Agsandrew/Shutterstock.