Praise for *Practical Object-Oriented Design in Ruby*

“This is great stuff! Your descriptions are so vibrant and vivid that I’m rediscovering the truth buried in OO principles that are otherwise so internalized that I forget to explore them. Your thoughts on design and knowing the future are especially eloquent.”

—Ian McFarland, President, New Context, Inc.

“As a self-taught programmer, this was an extremely helpful dive into some OOP concepts that I could definitely stand to become better acquainted with! And, I’m not alone: there’s a sign posted at work that reads, “WWSMD?—What Would Sandi Metz Do”?

—Jonathan Mukai, Pivotal in NYC

“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, developer, Bengler

“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, video and audio show host, TeachMeToCode.com

“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, editor, InfoQ.com

“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
“The book is just perfect! The elegance of Ruby shines but it also works as an A to Z of object-oriented programming in general.”

—Emil Rondahl, C# & .NET consultant

“This is an exceptional Ruby book, in which Metz offers a practical look at writing maintainable, clean, idiomatic code in Ruby. Absolutely fantastic, recommended for my Ruby hacker friends.”

—Zachary “Zee” Spencer, freelancer & coach

“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, senior developer, SciMed Solutions

“I predict this will become a classic. I have an uncomfortable familiarity with programming literature, and this book is on a completely different level. I am astonished when I find a book that offers new insights and ideas, and even more surprised when it can do so, not just once, but throughout the pages. This book is excellently written, well-organized, with lucid explanations of technical programming concepts.”

—Han S. Kang, software engineer and member of the LA Rubyists

“You should read this book if you write software for a living. The future developers who inherit your code will thank you.”

—Jose Fernandez, senior software engineer at New Relic

“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

“This is an amazing book about just how to do object-oriented thinking when you’re programming in Ruby. Although there are some chapters that are more Ruby-specific, this book could be a great resource for developers in any language. All in all, I can’t recommend this book enough.”

—James Hwang, thriceprime.com

“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

“In short, this is in my top five programming books I’ve ever read. I believe that in twenty years this will be considered one of the definitive works on object-oriented programming. I plan to re-read it at least once a year to keep my skills from falling into atrophy. If you’re a relatively new, intermediate, or even somewhat advanced OO developer in any language, purchasing this book is the best way I know to level up your OO design skills.”

—Brandon Hays, freelance software developer
PRACTICAL OBJECT-ORIENTED DESIGN IN RUBY
The Addison-Wesley Professional Ruby Series provides readers with practical, people-oriented, and in-depth information about applying the Ruby platform to create dynamic technology solutions. The series is based on the premise that the need for expert reference books, written by experienced practitioners, will never be satisfied solely by blogs and the Internet.

Visit informit.com/ruby for a complete list of available products.
For Amy, who read everything first
This page intentionally left blank
Contents

Foreword xv
Introduction xvii
Acknowledgments xxi
About the Author xxiii

1 Object-Oriented Design 1
In Praise of Design 2
The Problem Design Solves 2
Why Change Is Hard 3
A Practical Definition of Design 4
The Tools of Design 4
Design Principles 5
Design Patterns 6
The Act of Design 7
How Design Fails 7
When to Design 8
Judging Design 10
A Brief Introduction to Object-Oriented Programming 11
Procedural Languages 12
Object-Oriented Languages 12
Summary 14

2 Designing Classes with a Single Responsibility 15
Deciding What Belongs in a Class 16
Grouping Methods into Classes 16
Organizing Code to Allow for Easy Changes 16
6 Acquiring Behavior Through Inheritance 105
   Understanding Classical Inheritance 105
   Recognizing Where to Use Inheritance 106
      Starting with a Concrete Class 106
      Embedding Multiple Types 109
      Finding the Embedded Types 111
      Choosing Inheritance 112
      Drawing Inheritance Relationships 114
   Misapplying Inheritance 114
   Finding the Abstraction 116
      Creating an Abstract Superclass 117
      Promoting Abstract Behavior 120
      Separating Abstract from Concrete 123
      Using the Template Method Pattern 125
      Implementing Every Template Method 127
   Managing Coupling Between Superclasses and Subclasses 129
      Understanding Coupling 129
      Decoupling Subclasses Using Hook Messages 134
   Summary 139

7 Sharing Role Behavior with Modules 141
   Understanding Roles 142
   Finding Roles 142
      Organizing Responsibilities 143
      Removing Unnecessary Dependencies 145
   Writing the Concrete Code 147
   Extracting the Abstraction 150
   Looking Up Methods 154
   Inheriting Role Behavior 158
   Writing Inheritable Code 158
      Recognize the Antipatterns 158
      Insist on the Abstraction 159
      Honor the Contract 159
Use the Template Method Pattern 160
Preemptively Decouple Classes 161
Create Shallow Hierarchies 161

Summary 162

8 Combining Objects with Composition 163
Composing a Bicycle of Parts 164
  Updating the Bicycle Class 164
  Creating a Parts Hierarchy 165
Composing the Parts Object 168
  Creating a Part 169
  Making the Parts Object More Like an Array 172
Manufacturing Parts 176
  Creating the PartsFactory 177
  Leveraging the PartsFactory 178
The Composed Bicycle 180
Deciding Between Inheritance and Composition 184
  Accepting the Consequences of Inheritance 184
  Accepting the Consequences of Composition 187
  Choosing Relationships 188
Summary 190

9 Designing Cost-Effective Tests 191
Intentional Testing 192
  Knowing Your Intentions 193
  Knowing What to Test 194
  Knowing When to Test 197
  Knowing How to Test 198
Testing Incoming Messages 200
  Deleting Unused Interfaces 202
  Proving the Public Interface 203
  Isolating the Object Under Test 205
  Injecting Dependencies Using Classes 207
  Injecting Dependencies as Roles 208
One of the core truisms of software development is that as your code grows and requirements for the system that you are building change, additional logic will be added that is not yet present in the current system. In almost all cases, maintainability over the life of the code is more important than optimizing its present state.

The promise of using object-oriented (OO) design is that your code will be easier to maintain and evolve than otherwise. If you are new to programming, how do you unlock these secrets to maintainability using OO? The fact is that many of us have never had holistic training in writing clean object-oriented code, instead picking up our techniques through osmosis from colleagues and a myriad of older books and online sources. Or if we were given a primer in OO during school, it was done in languages such as Java or C++. (The lucky ones were taught using Smalltalk!)

Sandi Metz’s *Practical Object-Oriented Design in Ruby* covers all of the basics of OO using the Ruby language, meaning that it’s ready to usher countless Ruby and Rails newcomers to the next steps in their professional development as mature programmers.

Ruby itself, like Smalltalk, is a completely object-oriented (OO) language. Everything in it, even primitive data constructs such as strings and numbers, is represented by objects with behavior. When you write your own applications in Ruby, you do so by coding your own objects, each encapsulating some state and defining its own behavior. If you don’t already have OO experience, it can feel daunting to know how to start the process. This book guides you every step of the way, from the most basic questions of what to put in a class, through basic concepts such as the Single Responsibility Principle, all the way through to making tradeoffs between inheritance and composition, and figuring out how to test objects in isolation.

The best part, though, is Sandi’s voice. She’s got a ton of experience and is one of the nicest members of the community you’ll ever meet, and I think she did a great job
getting that feeling across in her writing. I’ve known Sandi for several years now, and I wondered if her manuscript would live up to the pleasure of actually getting to know Sandi in real life. I’m glad to say that it does, in spades, which is why I’m glad to welcome her as our newest author to the Professional Ruby Series.

—Obie Fernandez, Series Editor
Addison Wesley Professional Ruby Series
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 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. OO design 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, Object-Oriented Design, 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 towards 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.
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.

As it neared completion, Steve Klabnik, Desi McAdam, and Seth Wax reviewed the book and thus acted as gracious stand-ins for you, the gentle reader. 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 Education, 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 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.
Sandi Metz has 30 years of experience working on projects that survived to grow and change. She writes code every day as a software architect at Duke University, where her team solves real problems for customers who have large object-oriented applications that have been evolving for 15 or more years. She's committed to getting useful software out the door in extremely practical ways. Practical Object-Oriented Design in Ruby is the distillation of many years of whiteboard drawings and the logical culmination of a lifetime of conversations about OO design. Sandi has spoken at Ruby Nation and several times at Gotham Ruby User’s Conference and lives in Durham, NC.
This page intentionally left blank
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 know about others. Knowing creates a dependency. If not managed carefully, these dependencies will strangle your application.
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 you saw it in Chapter 2, Designing Classes with a Single Responsibility.

```ruby
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

Gear.new(52, 11, 26, 1.5).gear_inches
```
Examine the code above 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.

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 the first argument to Wheel.new 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.

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, \texttt{Gear} depends on \texttt{Wheel} and four other objects, coupling \texttt{Gear} to five different things. When the underlying code was first written everything worked fine. The problem lies dormant until you attempt to use \texttt{Gear} in another context or to change one of the classes upon which \texttt{Gear} depends. When that day comes the cold hard truth is revealed; despite appearances, \texttt{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.

**Other Dependencies**

The remainder of this chapter examines the four kinds of dependencies listed above 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 \texttt{self}” dependency, only magnified. Message chaining creates a dependency between the original object and
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.

Inject Dependencies

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

```ruby
1 class Gear
2   attr_reader :chainring, :cog, :rim, :tire
3 def initialize(chainring, cog, rim, tire)
4   @chainring = chainring
```
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 static types. 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:

```ruby
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’
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 one additional line of code was written; the decoupling was achieved by rearranging existing code.

This technique is known as dependency injection. Despite its fearsome 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.

Isolate Dependencies

It’s best to break all unnecessary dependences 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, Designing Classes with a Single Responsibility, you isolated extraneous responsibilities so that they would be easy to recognize and remove when the right impetus came; here you should isolate unnecessary dependences 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.
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.

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:

```ruby
1 def gear_inches
2   ratio * wheel.diameter
3 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:

```ruby
1 def gear_inches
2   #... a few lines of scary math
3   foo = some_intermediate_result * wheel.diameter
4   #... more lines of scary math
5 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:

```
1  def gear_inches
2     #... a few lines of scary math
3     foo = some_intermediate_result * diameter
4     #... more lines of scary math
5   end
6
7  def diameter
8      wheel.diameter
9  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.
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.

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

Gear.new(52, 11, Wheel.new(26, 1.5)).gear_inches
```

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 fixed-order 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 Hashes for Initialization Arguments

There’s a simple way to avoid depending on fixed-order arguments. If you have control over the Gear initialize method, change the code to take a hash of options instead of a fixed list of parameters.
The next example shows a simple version of this technique. The initialize method now takes just one argument, args, a hash that contains all of the inputs. The method has been changed to extract its arguments from this hash. The hash itself is created in lines 11–14.

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

The above technique has several advantages. The first and most obvious is that it removes every dependency on argument order. 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 fixed-order 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 the code in line 11 changed to use a hash, it lost its dependency on argument order but it gained a dependency on the names of the keys in the argument hash. 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. Additionally, and perhaps unexpectedly, the hash provides one new, secondary benefit: The key names in the hash furnish explicit documentation about the arguments. This is a byproduct of using a hash but the fact that it is unintentional makes it no less useful. Future maintainers of this code will be grateful for the information.

The benefits you achieve by using this technique vary, as always, based on your personal situation. If you are working on a method whose parameter list is lengthy
and wildly unstable, in a framework that is intended to be used by others, it will likely lower overall costs if you specify arguments in a hash. However, if you are writing a method for your own use that divides two numbers, it’s far simpler and perhaps ultimately cheaper to merely pass the arguments and accept the dependency on order. Between these two extremes lies a common case, that of the method that requires a few very stable arguments and optionally permits a number of less stable ones. In this case, the most cost-effective strategy may be to use both techniques; that is, to take a few fixed-order arguments, followed by an options hash.

Explicitly Define Defaults

There are many techniques for adding defaults. Simple non-boolean defaults can be specified using Ruby’s `||` method, as in this next example:

```ruby
# specifying defaults using ||
def initialize(args)
  @chainring = args[:chainring] || 40
  @cog = args[:cog]       || 18
  @wheel = args[:wheel]
end
```

This is a common technique but one you should use with caution; there are situations in which it might not do what you want. The `||` method acts as an `or` condition; it first evaluates the left-hand expression and then, if the expression returns `false` or `nil`, proceeds to evaluate and return the result of the right-hand expression. The use of `||` above therefore, relies on the fact that the `[]` method of `Hash` returns `nil` for missing keys.

In the case where `args` contains a `:boolean_thing` key that defaults to `true`, use of `||` in this way makes it impossible for the caller to ever explicitly set the final variable to `false` or `nil`. For example, the following expression sets `@bool` to `true` when `:boolean_thing` is missing and also when it is present but set to `false` or `nil`:

```ruby
@bool = args[:boolean_thing] || true
```

This quality of `||` means that if you take boolean values as arguments, or take arguments where you need to distinguish between `false` and `nil`, it’s better to use the `fetch` method to set defaults. The `fetch` method expects the key you’re fetching to be in the hash and supplies several options for explicitly handling missing keys. Its advantage over `||` is that it does not automatically return `nil` when it fails to find your key.
In the example below, line 3 uses `fetch` to set `@chainring` to the default, 40, only if the `:chainring` key is not in the `args` hash. Setting the defaults in this way means that callers can actually cause `@chainring` to get set to `false` or `nil`, something that is not possible when using the `||` technique.

```ruby
# specifying defaults using fetch
def initialize(args)
  @chainring = args.fetch(:chainring, 40)
  @cog = args.fetch(:cog, 18)
  @wheel = args[:wheel]
end
```

You can also completely remove the defaults from `initialize` and isolate them inside of a separate wrapping method. The `defaults` method below defines a second hash that is merged into the options hash during initialization. In this case, `merge` has the same effect as `fetch`; the defaults will get merged only if their keys are not in the hash.

```ruby
# specifying defaults by merging a defaults hash
def initialize(args)
  args = defaults.merge(args)
  @chainring = args[:chainring]
  # ...
end

def defaults
  { :chainring => 40, :cog => 18 }
end
```

This isolation technique is perfectly reasonable for the case above but it’s especially useful when the defaults are more complicated. If your defaults are more than simple numbers or strings, implement a `defaults` method.

**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 fixed-order 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 fixed-order 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 external 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 fixed-order 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 importantly, it provides an improved interface for your application.

As you can see in line 24, GearWrapper allows your application to create a new instance of Gear using an options hash.

```ruby
# 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(args)
    SomeFramework::Gear.new(args[:chainring],
                            args[:cog],
                            args[:wheel])
  end
end
```
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.

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

```ruby
23 # Now you can create a new Gear using an arguments hash.
24 GearWrapper.gear(
25   :chainring => 52,
26   :cog => 11,
27   :wheel => Wheel.new(26, 1.5)).gear_inches
```

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 24) 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 substituting an options hash for a list of fixed-order arguments 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.
Gear is still responsible for the actual calculation but it expects a `diameter` argument to be passed in by the caller (line 8).

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

  def gear_inches(diameter)
    ratio * diameter
  end

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

```ruby
class Wheel
  attr_reader :rim, :tire, :gear
  def initialize(rim, tire, chainring, cog)
    @rim = rim
    @tire = tire
    @gear = Gear.new(chainring, cog)
  end

  def diameter
    rim + (tire * 2)
  end

  def gear_inches
    gear.gear_inches(diameter)
  end
end
```

```ruby
Wheel.new(26, 1.5, 52, 11).gear_inches
```

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.

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 begin 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.

Rubyists 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 painfully, 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](image_url)
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.

**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
<table>
<thead>
<tr>
<th>Abstract</th>
<th>Behaves-like-a relationships, 189</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Behavior acquired through inheritance, 105–39</td>
</tr>
<tr>
<td></td>
<td>confirming, 233–36</td>
</tr>
<tr>
<td></td>
<td>data structures, hiding, 26–29</td>
</tr>
<tr>
<td></td>
<td>depending on, instead of data, 24–29</td>
</tr>
<tr>
<td></td>
<td>instance variables, hiding, 24–26</td>
</tr>
<tr>
<td></td>
<td>set of, 19</td>
</tr>
<tr>
<td></td>
<td>subclass, 233–39</td>
</tr>
<tr>
<td></td>
<td>superclass, 234–39</td>
</tr>
<tr>
<td></td>
<td>testing, 236–39</td>
</tr>
<tr>
<td></td>
<td>Behavior Driven Development (BDD), 199, 213</td>
</tr>
<tr>
<td></td>
<td>Big Up Front Design (BUFD), 8–9</td>
</tr>
<tr>
<td></td>
<td>Booch, Grady, 188</td>
</tr>
<tr>
<td></td>
<td>Break-even point, 11</td>
</tr>
<tr>
<td></td>
<td>Bugs, finding, 193</td>
</tr>
<tr>
<td>Category</td>
<td>Case statements that switch on class, 96–98</td>
</tr>
<tr>
<td></td>
<td>kind_of? and is_a?, 97</td>
</tr>
<tr>
<td></td>
<td>responds_to?, 97</td>
</tr>
<tr>
<td>Class under test, removing private methods from, 214</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td></td>
</tr>
<tr>
<td></td>
<td>concrete, writing, 147–50</td>
</tr>
<tr>
<td></td>
<td>dependency injection to shape, 41–42</td>
</tr>
<tr>
<td></td>
<td>depending on behavior instead of data, 24–29</td>
</tr>
<tr>
<td></td>
<td>embracing change, writing, 24–33</td>
</tr>
<tr>
<td></td>
<td>organizing to allow for changes, 16–17</td>
</tr>
<tr>
<td></td>
<td>dependent-laden, avoiding, 55</td>
</tr>
<tr>
<td></td>
<td>grouping methods into, 16</td>
</tr>
<tr>
<td></td>
<td>references to (See Loosely coupled code, writing)</td>
</tr>
<tr>
<td></td>
<td>responsibilities isolated in, 31–33</td>
</tr>
<tr>
<td></td>
<td>Ruby based vs. framework, 53–54</td>
</tr>
<tr>
<td></td>
<td>type and category used in, 111</td>
</tr>
<tr>
<td></td>
<td>virtual, 61</td>
</tr>
<tr>
<td>Class-based OO languages, 12–13</td>
<td></td>
</tr>
<tr>
<td>Class class, 14</td>
<td></td>
</tr>
<tr>
<td>Classical inheritance, 105–6</td>
<td></td>
</tr>
<tr>
<td>Class of an object ambiguity about, 94–95</td>
<td></td>
</tr>
<tr>
<td>checking, 97, 111, 146</td>
<td></td>
</tr>
<tr>
<td>Class under test, removing private methods from, 214</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>See also Inheritable code, writing</td>
</tr>
<tr>
<td></td>
<td>Inherited code, testing</td>
</tr>
<tr>
<td></td>
<td>concrete, writing, 147–50</td>
</tr>
<tr>
<td></td>
<td>dependency injection to shape, 41–42</td>
</tr>
<tr>
<td></td>
<td>depending on behavior instead of data, 24–29</td>
</tr>
<tr>
<td></td>
<td>embracing change, writing, 24–33</td>
</tr>
</tbody>
</table>
Index

initialization, 121
loosely coupled, writing, 39–51
open–closed, 185
organizing to allow for changes, 16–17
putting its best (inter)face forward, writing, 76–79
relying on duck typing, writing, 95–100
single responsibility, enforcing, 29–33
truths about, 53
Code arrangement technique, 184
Cohesion, 22
Command messages, 197, 216–18
Compile/make cycle, 102, 103, 104
Compiler, 54, 101, 103–4, 118
Composition
aggregation and, 183–84
benefits of, 187
of bicycle, 180–84
of bicycle of parts, 164–68
consequences of, accepting, 187–88
costs of, 187–88
for has-a relationships, 183, 190
inheritance and, deciding between, 184–90
manufacturing parts, 176–80
objects combined with, 163–90
of parts object, 168–76
summary, 190
use of term, 183–84
Concrete class, 106–9, 209
Concretions
abstractions separated from, 123–25
inheritance and, 106–9
recognizing, 54–55
writing, 147–50
Context
independence, seeking, 71–73
minimizing, 79
Contract, honoring, 159–60
Costs
of composition, 187–88
of duck typing, 85–104
of inheritance, 185–86
of testing, 191–240
Coupling
decoupling classes in writing inheritable code, 161
decoupling subclasses using hook messages, 134–38
between superclasses and subclasses, managing, 129–38
understanding, 129–34
Coupling between objects (CBO), 37–38
C++, 102
Data
depending on behavior instead of, 24–29
instance variables, hiding, 24–26
structures, hiding, 26–29
types, 12, 13
Decoupling
classes in writing inheritable code, 161
subclasses using hook messages, 134–38
Defaults, explicitly defining, 48–49
Delegation, 82, 183
Demeter. See Law of Demeter (LoD)
Demotion failure, 123
Dependencies
argument-order, removing, 46–51
coupling between objects, 37–38
direction of (See Dependency direction)
finding, 55–57
injecting (See Dependency injection)
interfaces and, 62–63
isolating, 42–45
loosely coupled code, writing, 39–51
managing, 55–57
objects speaking for themselves, 147
other, 38–39
recognizing, 37
removing unnecessary, 145–47
reversing, 51–53
scheduling duck type, discovering, 146–47
summary, 57
understanding, 36–39
Dependency direction
abstractions, recognizing, 54–55
change in, likelihood of (See Likelihood of change)
choosing, 53–57
concretions, recognizing, 54–55
dependent-laden classes, avoiding, 55
finding, 55–57
managing, 51–57
reversing, 51–53
Dependency injection
failure of, 60
in loosely coupled code, 39–42
as roles, 208–13
to shape code, 41–42
using classes, 207–8
Dependency Inversion Principle, 5
Dependent-laden classes, avoiding, 55
Design
act of, 7–11
definition of, 4
failure in, 7–8
judging, 10–11
patterns, 6–7
principles, 5–6
problems solved by, 2–3
tools, 4–7
when to design, 8–10
Design decisions
deferring, 193
when to make, 22–23
Design flaws, exposing, 194
Design patterns, 6–7
Design Patterns: Elements of Reusable Object-Oriented Software (Gamma, Helm, Johnson, and Vlissides), 6, 188
Design principles, 5–6
Design tools, 4–7
Documentation
doctor types, 98
of roles, testing used in, 212–13
supplying, in testing, 193
Domain objects, 64, 83, 96, 199
Doubles, role tests to validate, 224–29
DRY (Don’t Repeat Yourself), 5, 24, 27, 28, 45, 50, 196
Duck types, 219–29
defining, 85
documenting, 98
finding, 90–94
hidden, recognizing, 96–98
overlooking, 87
sharing between, 99
testing roles, 219–24
trust in, placing, 98
using role tests to validate doubles, 224–29

Duck typing
for behaves-like-a relationships, 189
case statements that switch on class, 96–98
choosing ducks wisely, 99–100
code that relies on, writing, 95–100
consequences of, 94–95
costs reduced with, 85–104
dynamic typing and, 100–104
fear of, conquering, 100–104
problem, compounding, 87–90
static typing and, 100–102
summary, 104
understanding, 85–95

Dynamic typing
embracing, 102–4
static typing vs., 100–102

Embedded types of inheritance
finding, 111–12
multiple, 109–11
Explicit interfaces, creating, 76–78
External messages, isolating, 44–45
Extra responsibilities
extracted from methods, 29–31
isolated in classes, 31–33

Factories, 51
File data type, 12
Fixed-order arguments, 46–51
Fixnum class, 13, 14
Fowler, Martin, 191
Framework class, vs. Ruby based class, 53–54

Gamma, Erich, 6, 188
Gang of Four (Gof), 6, 188
Gear inches, 20–21

has-a relationships, 183, 190
vs. is-a relationships, 188–89
Hashes used for initialization arguments, 46–48
Helm, Richard, 6, 188

Hidden ducks
finding, 90–94
recognizing, 96–98
 Highly cohesive class, 22
Hook messages, 134–38
Hunt, Andy, 5

Incoming messages, testing,
200–213
injecting dependencies as roles,
208–13
injecting dependencies using classes, 207–8
interfaces, deleting unused,
202–3
isolating object under test, 205–7
proving public interface, 203–4
Inheritable code, writing, 158–62
abstraction, insisting on, 159
antipatterns, recognizing, 158–59
classes, preemptively decouple, 161
contract, honoring, 159–60
shallow hierarchies, creating, 161–62
template method pattern, using, 160

Inheritance
abstract class, finding, 116–29
behavior acquired through, 105–39
benefits of, 184–85
choosing, 112–14
classical, 105–6
composition and, deciding between, 184–90
concretions and, 106–9
consequences of, accepting, 184–86
costs of, 185–86
embedded types of, 109–12
family tree image of, 112
implying, 117
for is-a relationships, 188–89
misapplying, 114–16
multiple, 112
problem solved by, 112
recognizing where to use, 106–14
relationships, drawing, 114
rules of, 117
single, 112
summary, 139

Inherited code, testing, 229–39
behavior, testing unique, 236–39
inherited interface, specifying, 229–32
subclass responsibilities, specifying, 233–36
Inherited interface, specifying, 229–32
Inheriting role behavior, 158
Initialization arguments, 41–43
hashes used for, 46–48
in isolation of instance creation, 42–43
Initialization code, 121
Injection of dependencies. See Dependency injection
Instance variables, hiding, 24–26
Intention, constructing, 64–65
Intentional testing, 192–200
Interface
inherited, specifying, 229–32
Interfaces. See also Private interfaces; Public interfaces
code putting its best (inter)face forward, writing, 76–79
defining, 61–63
deleting unused, 202–3
dependencies and, 62–63
explicit, 76–78
flexible, 59–83
Law of Demeter and, 80–83
responsibilities and, 62–63
summary, 83
understanding, 59–61
Interface Segregation Principle, 5
is_a?, 97
is-a relationships, 188–89
vs. has-a relationships, 188–89
Isolation
of dependencies, 42–45
of external messages, 44–45
of instance creation, 42–43, 42–44
of multiparameter initialization, 49–51
of object under test, 205–7
of responsibilities in classes, 31–33

Java, 102, 118
JavaScript, 106
Johnson, Ralph, 6, 188
Keywords, 77–78
kind_of?, 97

Law of Demeter (LoD), 5, 80–83
defining Demeter, 80
Demeter project, 5, 80
listening to Demeter, 82–83
violations, 80–82
Likelihood of change, 53–57
in embedded references to
messages, 45
vs. number of dependents, 55–57
understanding, 53–54
Liskov, Barbara, 160
Liskov Substitution Principle (LSP),
5, 160, 230–31, 237, 239
Loosely coupled code, writing,
39–51
inject dependencies, 39–42
isolate dependencies, 42–45
remove argument-order
dependencies, 46–51
Managing dependencies, 3
Message, 15
Message chaining, 38–39, 80–83
Messages. See also Incoming
messages, testing, 76
automatic message delegation,
105–6
command, proving, 216–18
delegating, 82
external, isolating, 44–45
incoming, testing, 200–213
likely to change, embedded
references to, 45
message forwarding via classical
inheritance, 112
objects discovered by, 74–76
query, ignoring, 215–16
testing outgoing, 215–18
Metaprogramming, 102–3
Methods
extra responsibilities extracted from,
29–31
grouping into classes, 16
wrapper, 24–25, 82
Methods, looking up, 154–58
gross oversimplification, 154–55
more accurate explanation, 155–56
very nearly complete explanation,
156–58
Metrics, 5, 10–11
Meyer, Bertrand, 188
MiniTest, 200
Modules
definition of, 143
role behavior shared with,
141–62
Monkey patching, 100
Multiparameter initialization,
isolating, 49–51
Multiple inheritance, 112
NASA Goddard Space Flight
Center applications, 6
Nil, 48–49, 113
NilClass, 113
Object class
ambiguity about, 94–95
checking, 97, 111, 146
Object-Oriented Analysis and Design
(Booch), 188
Object-oriented design (OOD),
1–14. See also Design
dependencies managed by, 3
masters of, 188
overview of, 1
Object-oriented languages, 12–14
Object-oriented programming,
11–14
object-oriented languages in,
12–14
overview of, 11
procedural languages in, 12
Objects. See also Parts object
combined with composition,
163–90
domain, 64, 83, 96, 199
messages used to discover, 74–76
speaking for themselves, 147
trusting other, 73–74
Object under test, 200, 202, 205–7
Open–closed code, 185
Open-Closed Principle, 5, 185
Overridden methods, 115
Parts object
composition of, 168–76
creating, 169–72
creating PartsFactory, 177–78
hierarchy, creating, 165–68
leveraging PartsFactory, 178–80
making more like array, 172–76
manufacturing, 176–80
Polymorphism, 95
Private interfaces
defining, 61, 62
depending on, caution in, 79
Private keyword, 77–78
Private methods, testing, 213–15
choosing, 214–15
ignoring, 213–14
removing from class under test,
214
Programming languages
statically or dynamically typed,
100–104
syntax in, 118
type used in, 85–86
Promotion failure, 122–23
Protected keyword, 77–78
Public interfaces
context independence, seeking,
71–73
defining, 61, 62
eample application: bicycle
touring company, 63–64
finding, 63–76
intention, constructing, 64–65
message-based application,
creating, 76
messages used to discover objects,
74–76
of others, honoring, 78–79
proving, 203–4
sequence diagrams, using, 65–69
trusting other objects, 73–74
“what” vs. “how,” importance of,
69–71
Public keyword, 77–78
Query messages, 196, 197, 215–16
Refactoring
barriers to, reducing, 215
definition of, 191
in extracting extra responsibilities
from methods, 29–31
rule for, 123
strategies, deciding between,
122–23
testing roles and, 220–21, 226
in writing changeable code, 191–92
Refactoring: Improving the Design of Existing Code (Fowler), 191
Relationships, 188–90
aggregation and, 183–84
use composition for has-a relationships, 190
use duck types for behaves-like-a relationships, 189
use inheritance for is-a relationships, 188–89
responds_to?, 97
Responsibilities, organizing, 143–45
Responsibility-Driven Design (RDD), 22
Reversing dependency direction, 51–53
Roles
concrete code, writing, 147–50
finding, 142–43
inheritable code, writing, 158–62
injecting dependencies as, 208–13
role behavior shared with modules, 141–62
summary, 162
testing, in duck typing, 219–24
testing to document, 212–13
tests to validate doubles, 224–29
understanding, 142–58
Ruby based class vs. framework class, 53–54
Runtime type errors, 101, 103–4
Sequence diagrams, using, 65–69
Shallow hierarchies in writing inheritable code, 161–62
Single inheritance, 112
Single responsibility, classes with benefits of, 31
code embracing change, writing, 24–33
creating, 17–23
design decisions, when to make, 22–23
designing, 15–34
determining, 22
enforcing, 29–33
example application: bicycles and gears, 17–21
extra responsibilities and, 29–33
importance of, 21
real wheel, 33–34
summary, 34
Single Responsibility Principle, 5
designing classes with, 15–34
SOLID design principles, 5, 160
Source code repository, 59
Source lines of code (SLOC), 10–11
Specializations, 117
Spike a problem, 198
Static typing
duck types and, subverting with, 100–101
vs. dynamic typing, 100–102
String class, 13–14
String data type, 12, 13
String objects, 13–14
Subclass behavior
confirming, 233–34
testing, 236–37
Subclasses
decoupling using hook messages, 134–38
superclasses and, coupling between, 129–38
Superclass behavior
confirming, 234–36
testing, 237–39
Superclasses
creating, 117–20
subclasses and, coupling between, 129–38
Syntax, 118
Technical debt, 11, 79
Template method pattern
implementing every, 127–29
using, 125–27
in writing inheritable code, 160
Test Driven Development (TDD), 199, 213
Testing
abstractions, supporting, 194
bugs, finding, 193
cost-effective, designing, 191–240
creating test doubles, 210–11
development decisions, deferring, 193
design flaws, exposing, 194
documentation, supplying, 193
duck types, 219–29
incoming messages, 200–213
inherited code, 229–39
intentional testing, 192–200
knowing how to test, 198–200
knowing what to test, 194–97
knowing when to test, 197–98
knowing your intentions, 193–94
outgoing messages, 215–18
private methods, 213–15
summary, 240
to document roles, 212–13
Testing outgoing messages, 215–18
command messages, proving, 216–18
query messages, ignoring, 215–16
Thomas, Dave, 5
Touch of Class: Learning to Program Well with Objects and Contracts (Meyer), 188
Train wreck, 80, 82, 83
TRUE code, 17
Types, 85. See also Duck typing
across-class, 86
static vs. dynamic, 100–102
within-class, 63
Type used in class, 111
Unified Modeling Language (UML), 65–66, 114
Use case, 64, 65, 66, 67, 69, 74
Variables, defining, 12
Virtual class, 61
Vlissides, Jon, 6, 188
“What” vs. “how,” importance of, 69–71
Wilkerson, Brian, 22
Wirfs-Brock, Rebecca, 22
Within-class types, 63
Wrapper method, 24–25, 82