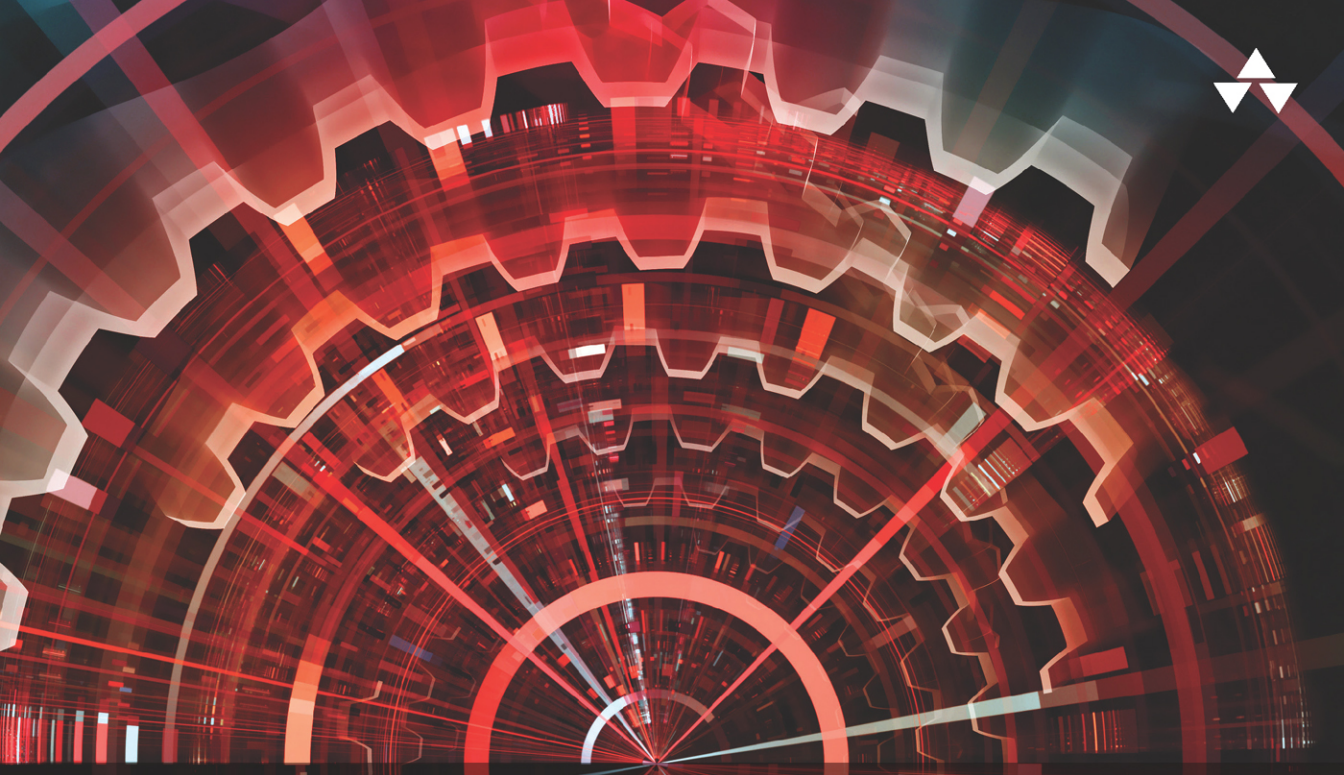

PRACTICAL OBJECT-ORIENTED DESIGN

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

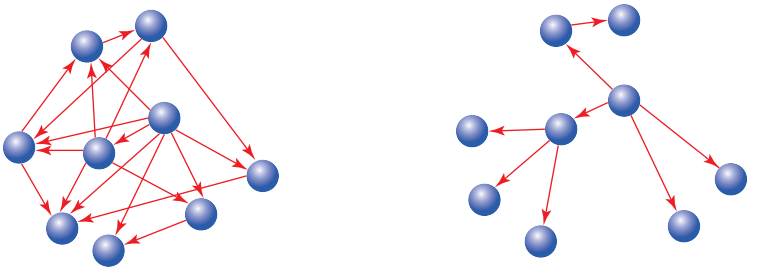

Figure 4.1 Communication patterns

In the second application, the messages have a clearly defined pattern. Here the objects communicate in specific and well-defined ways. If these messages left trails, the trails would accumulate to create a set of islands with occasional bridges between them.

Both applications, for better or worse, are characterized by the patterns of their messages.

The objects in the first application are difficult to reuse. Each one exposes too much of itself and knows too much about its neighbors. This excess knowledge results in objects that are finely, explicitly, and disastrously tuned to do only the things that they do right now. No object stands alone; to reuse any you need all, to change one thing you must change everything.

The second application is composed of pluggable, component-like objects. Each reveals as little about itself, and knows as little about others, as possible.

The design issue in the first application is not necessarily a failure of dependency injection or single responsibility. Those techniques, while necessary, are not enough to prevent the construction of an application whose design causes you pain. The roots of this new problem lie not in what each class does but in what it *reveals*. In the first application, each class reveals all. Every method in any class is fair game to be invoked by any other object.

Experience tells you that all the methods in a class are not the same; some are more general or more likely to change than others. The first application takes no notice of this. It allows all methods of any object, regardless of their granularity, to be invoked by others.

In the second application, the message patterns are visibly constrained. This application has some agreement, some bargain, about which messages may pass between its objects. Each object has a clearly defined set of methods that it expects others to use.

These exposed methods comprise the class's *public interface*.

The word interface can refer to a number of different concepts. Here the term is used to refer to the kind of interface that is *within* a class. Classes implement methods; some of those methods are intended to be used by others, and these methods make up its public interface.

An alternative kind of interface is one that spans across classes and that is independent of any single class. Used in this sense, the word interface represents a set of messages where the messages themselves define the interface. Many different classes may, as part of their whole, implement the methods that the interface requires. It's almost as if the interface defines a virtual class; that is, any class that implements the required methods can act like the *interface* kind of thing.

The remainder of this chapter will address the first kind of interface, that is, methods within a class and how and what to expose to others. Chapter 5, "Reducing Costs with Duck Typing," explores the second kind of interface, the one that represents a concept that is broader than a class and is defined by a set of messages.

4.2 Defining Interfaces

Imagine a restaurant kitchen. Customers order food off a menu. These orders come into the kitchen through a little window (the one with the bell beside it, "order up!") and the food eventually comes out. To a naïve imagination it may seem as if the kitchen is filled with magical plates of food that are waiting, pining to be ordered, but in reality the kitchen is full of people, food, and frenzied activity, and each order triggers a new construction and assembly process.

The kitchen does many things but does not, thankfully, expose them all to its customers. It has a public interface that customers are expected to use; the menu. Within the kitchen many things happen, many other messages get passed, but these messages are *private* and thus invisible to customers. Even though they may have ordered it, customers are not welcome to come in and stir the soup.

This distinction between public and private exists because it is the most effective way to do business. If customers directed the cooking, they would have to be re-educated whenever the kitchen ran low on an ingredient and needed to make a substitution. Using a menu avoids this problem by letting each customer ask for what they want without knowing anything about *how* the kitchen makes it.

Each of your classes is like a kitchen. The class exists to fulfill a single responsibility but implements many methods. These methods vary in scale and granularity and range from broad, general methods that expose the main responsibility of the class to tiny utility methods that are only meant to be used internally. Some of these methods represent the menu for your class and should be public; others deal with internal implementation details and are private.

4.2.1 Public Interfaces

The methods that make up the public interface of your class comprise the face it presents to the world. They:

- Reveal its primary responsibility.
- Are expected to be invoked by others.
- Will not change on a whim.
- Are safe for others to depend on.
- Are thoroughly documented in the tests.

4.2.2 Private Interfaces

All other methods in the class are part of its private interface. They:

- Handle implementation details.
- Are not expected to be sent by other objects.
- Can change for any reason whatsoever.
- Are unsafe for others to depend on.
- May not even be referenced in the tests.

4.2.3 Responsibilities, Dependencies, and Interfaces

Chapter 2, "Designing Classes with a Single Responsibility," was about creating classes that have a single responsibility—a single purpose. If you think of a class as having a single purpose, then the things it does (its more specific responsibilities) are what allow it to fulfill that purpose. There is a correspondence between the statements you might make about these more specific responsibilities and the classes' public methods. Indeed, public methods should read like a description of responsibilities. The public interface is a contract that articulates the responsibilities of your class.

Chapter 3, "Managing Dependencies," was about dependencies, and its take-home message was that a class should depend only on classes that change less often than it does. Now that you are dividing every class into a public part and a private part, this idea of depending on less changeable things also applies to the methods *within* a class.

The public parts of a class are the stable parts; the private parts are the changeable parts. When you mark methods as public or private, you tell users of your class upon which methods they may safely depend. When your classes use the public

methods of others, you trust those methods to be stable. When you decide to depend on the private methods of others, you understand that you are relying on something that is inherently unstable and are thus increasing the risk of being affected by a distant and unrelated change.

4.3 Finding the Public Interface

Finding and defining public interfaces is an art. It presents a design challenge because there are no cut-and-dried rules. There are many ways to create "good enough" interfaces and the costs of a "not good enough" interface may not be obvious for a while, making it difficult to learn from mistakes.

The design goal, as always, is to retain maximum future flexibility while writing only enough code to meet today's requirements. Good public interfaces reduce the cost of unanticipated change; bad public interfaces raise it.

This section introduces a new application to illustrate a number of rules of thumb about interfaces and a new tool aid to in their discovery.

4.3.1 An Example Application: Bicycle Touring Company

Meet FastFeet, Inc., a bicycle touring company. FastFeet offers both road and mountain bike trips. FastFeet runs its business using a paper system. It currently has no automation at all.

Each trip offered by FastFeet follows a specific route and may occur several times during the year. Each has limitations on the number of customers who may go and requires a specific number of guides who double as mechanics.

Each route is rated according to its aerobic difficulty. Mountain bike trips have an additional rating that reflects technical difficulty. Customers have an aerobic fitness level and a mountain bike technical skill level to determine if a trip is right for them.

Customers may rent bicycles or they may choose to bring their own. FastFeet has a few bicycles available for customer rental, and it also shares in a pool of bicycle rentals with local bike shops. Rental bicycles come in various sizes and are suitable for either road or mountain bike trips.

Consider the following simple requirement, which will be referred to later as a *use case:* A customer, in order to choose a trip, would like to see a list of available trips of appropriate difficulty, on a specific date, where rental bicycles are available.

4.3.2 Constructing an Intention

Getting started with the first bit of code in a brand new application is intimidating. When you are adding code to an existing code base, you are usually extending an existing design. Here, however, you must put pen to paper (figuratively) and make

decisions that will determine the patterns of this application forever. The design that gets extended later is the one that you are establishing now.

You know that you should not dive in and start writing code. You may believe that you should start writing tests, but that belief doesn't make it easy to do. Many novice designers have serious difficulty imagining the first test. Writing that test requires that you have an idea about what you want to test, one that you may not yet have.

The reason that test-first gurus can easily start writing tests is that they have so much design experience. At this stage, they have already constructed a mental map of possibilities for objects and interactions in this application. They are not attached to any specific idea and plan to use tests to discover alternatives, but they know so much about design that they have already formed an intention about the application. It is this intention that allows them to specify the first test.

Whether you are conscious of them or not, you have already formed some intentions of your own. The description of FastFeet's business has likely given you ideas about potential classes in this application. You probably expect to have Customer, Trip, Route, Bike, and Mechanic classes.

These classes spring to mind because they represent nouns in the application that have both data and behavior. Call them *domain objects*. They are obvious because they are persistent; they stand for big, visible real-world things that will end up with a representation in your database.

Domain objects are easy to find, but they are not at the design center of your application. Instead, they are a trap for the unwary. If you fixate on domain objects, you will tend to coerce behavior into them. Design experts *notice* domain objects without concentrating on them; they focus not on these objects but on the messages that pass between them. These messages are guides that lead you to discover other objects, ones that are just as necessary but far less obvious.

Before you sit at the keyboard and start typing, you should form an intention about the objects *and* the messages needed to satisfy this use case. You would be best served if you had a simple, inexpensive, communication-enhancing way to explore design that did not require you to write code.

Fortunately, some very smart people have thought about this issue at great length and have devised an effective mechanism for doing just that.

4.3.3 Using Sequence Diagrams

There is a perfect, low-cost way to experiment with objects and messages: *sequence diagrams*.

Sequence diagrams are defined in the Unified Modeling Language (UML) and are one of many diagrams that UML supports. Figure 4.2 shows a sampling of some diagrams.

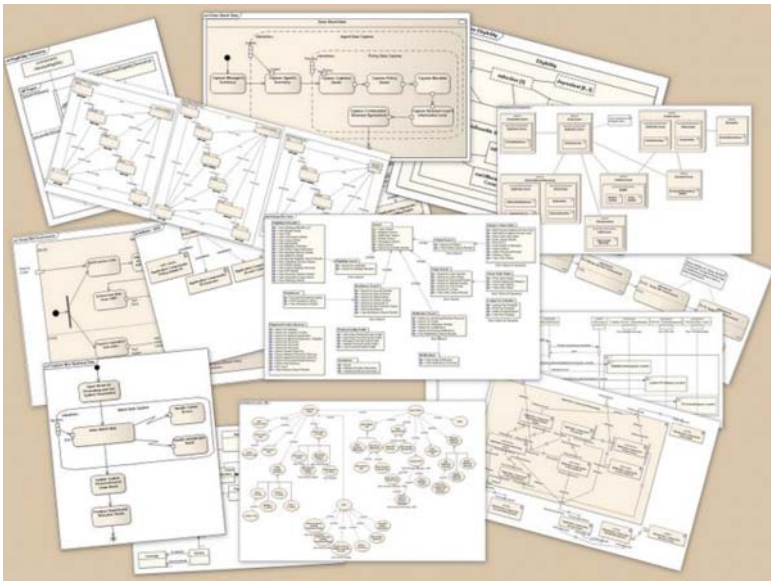

Figure 4.2 Sample UML diagrams

If you have joyfully embraced UML, you already know the value of sequence diagrams. If you are unfamiliar with UML and find the graphic alarming, fear not. This book is not turning into a UML guide. Lightweight, agile design does not require the creation and maintenance of piles of artifacts. However, the creators of UML put a great deal of thought into how to communicate object-oriented design, and you can leverage their efforts. There are UML diagrams that provide excellent, transient ways to explore and communicate design possibilities. Use them; you do not need to reinvent this wheel.

Sequence diagrams are quite handy. They provide a simple way to experiment with different object arrangements and message-passing schemes. They bring clarity to your thoughts and provide a vehicle to collaborate and communicate with others. Think of them as a lightweight way to acquire an intention about an interaction. Draw them on a whiteboard, alter them as needed, and erase them when they've served their purpose.

Figure 4.3 shows a simple sequence diagram. This diagram represents an attempt to implement the use case above. It shows Moe, a `Customer`, and the `Trip` class, where Moe sends the `suitable_trips` message to `Trip` and gets back a response.

Figure 4.3 illustrates the two main parts of a sequence diagram. As you can see, they show two things: objects and the *messages* passing between them. The following paragraphs describe the parts of this diagram, but please note that the UML police will not arrest you if you vary from the official style. Do what works for you.