Addison-Wesley Professional Ruby Series

PRACTICAL OBJECT-ORIENTED DESIGN IN RUBY

An Agile Primer

SANDI METZ

CHAPTER 1
Object-Oriented Design

The world is procedural. Time flows forward and events, one by one, pass by. Your morning procedure may be to get up, brush your teeth, make coffee, dress, and then get to work. These activities can be modeled using procedural software; because you know the order of events you can write code to do each thing and then quite deliberately string the things together, one after another.

The world is also object-oriented. The objects with which you interact might include a spouse and a cat, or an old car and a pile of bike parts in the garage, or your ticking heart and the exercise plan you use to keep it healthy. Each of these objects comes equipped with its own behavior and, while some of the interactions between them might be predictable, it is entirely possible for your spouse to unexpectedly step on the cat, causing a reaction that rapidly raises everyone's heart rate and gives you new appreciation for your exercise regimen.

In a world of objects, new arrangements of behavior emerge naturally. You don't have to explicitly write code for the `spouse_steps_on_cat` procedure, all you need is a spouse object that takes steps and a cat object that does not like being stepped on. Put these two objects into a room together and unanticipated combinations of behavior will appear.

This book is about designing object-oriented software, and it views the world as a series of spontaneous interactions between objects. Object-oriented design (OOD) requires that you shift from thinking of the world as a collection of predefined procedures to modeling the world as a series of messages that pass between objects. Failures of OOD might look like failures of coding technique but they are actually failures of

perspective. The first requirement for learning how to do object-oriented design is to immerse yourself in objects; once you acquire an object-oriented perspective the rest follows naturally.

This book guides you through the immersion process. This chapter starts with a general discussion of OOD. It argues the case for design and then proceeds to describe when to do it and how to judge it. The chapter ends with a brief overview of object-oriented programming that defines the terms used throughout the book.

In Praise of Design

Software gets built for a reason. The target application—whether a trivial game or a program to guide radiation therapy—is the entire point. If painful programming were the most cost-effective way to produce working software, programmers would be morally obligated to suffer stoically or to find other jobs.

Fortunately, you do not have to choose between pleasure and productivity. The programming techniques that make code a joy to write overlap with those that most efficiently produce software. The techniques of object-oriented design solve both the moral and the technical dilemmas of programming; following them produces cost-effective software using code that is also a pleasure to work on.

The Problem Design Solves

Imagine writing a new application. Imagine that this application comes equipped with a complete and correct set of requirements. And if you will, imagine one more thing: once written, this application need never change.

For this case, design does not matter. Like a circus performer spinning plates in a world without friction or gravity, you could program the application into motion and then stand back proudly and watch it run forever. No matter how wobbly, the plates of code would rotate on and on, teetering round and round but never quite falling.

As long as nothing changed.

Unfortunately, something *will* change. It always does. The customers didn't know what they wanted, they didn't say what they meant. You didn't understand their needs, you've learned how to do something better. Even applications that are perfect in every way are not stable. The application was a huge success, now everyone wants more. Change is unavoidable. It is ubiquitous, omnipresent, and inevitable.

Changing requirements are the programming equivalent of friction and gravity. They introduce forces that apply sudden and unexpected pressures that work against the best-laid plans. It is the need for change that makes design matter.

Applications that are easy to change are a pleasure to write and a joy to extend. They're flexible and adaptable. Applications that resist change are just the opposite; every change is expensive and each makes the next cost more. Few difficult-to-change applications are pleasant to work on. The worst of them gradually become personal horror films where you star as a hapless programmer, running madly from one spinning plate to the next, trying to stave off the sound of crashing crockery.

Why Change Is Hard

Object-oriented applications are made up of parts that interact to produce the behavior of the whole. The parts are *objects;* interactions are embodied in the *messages* that pass between them. Getting the right message to the correct target object requires that the sender of the message know things about the receiver. This knowledge creates dependencies between the two and these dependencies stand in the way of change.

Object-oriented design is about *managing dependencies*. It is a set of coding techniques that arrange dependencies such that objects can tolerate change. In the absence of design, unmanaged dependencies wreak havoc because objects know too much about one another. Changing one object forces change upon its collaborators, which in turn, forces change upon its collaborators, *ad infinitum*. A seemingly insignificant enhancement can cause damage that radiates outward in overlapping concentric circles, ultimately leaving no code untouched.

When objects know too much they have many expectations about the world in which they reside. They're picky, they need things to be "just so." These expectations constrain them. The objects resist being reused in different contexts; they are painful to test and susceptible to being duplicated.

In a small application, poor design is survivable. Even if everything is connected to everything else, if you can hold it all in your head at once you can still improve the application. The problem with poorly designed *small* applications is that if they are successful they grow up to be poorly designed *big* applications. They gradually become tar pits in which you fear to tread lest you sink without a trace. Changes that should be simple may cascade around the application, breaking code everywhere and requiring extensive rewriting. Tests are caught in the crossfire and begin to feel like a hindrance rather than a help.

A Practical Definition of Design

Every application is a collection of code; the code's arrangement is the *design*. Two isolated programmers, even when they share common ideas about design, can be relied upon to solve the same problem by arranging code in different ways. Design is not an assembly line where similarly trained workers construct identical widgets; it's a studio where like-minded artists sculpt custom applications. Design is thus an art, the art of arranging code.

Part of the difficulty of design is that every problem has two components. You must not only write code for the feature you plan to deliver today, you must also create code that is amenable to being changed later. For any period of time that extends past initial delivery of the beta, the cost of change will eventually eclipse the original cost of the application. Because design principles overlap and every problem involves a shifting timeframe, design challenges can have a bewildering number of possible solutions. Your job is one of synthesis; you must combine an overall understanding of your application's requirements with knowledge of the costs and benefits of design alternatives and then devise an arrangement of code that is cost effective in the present and will continue to be so in the future.

Taking the future into consideration might seem to introduce a need for psychic abilities normally considered outside the realm of programming. Not so. The future that design considers is not one in which you anticipate unknown requirements and preemptively choose one from among them to implement in the present. Programmers are not psychics. Designs that anticipate specific future requirements almost always end badly. Practical design does not anticipate what will happen to your application, it merely accepts that something will and that, in the present, you cannot know what. It doesn't guess the future; it preserves your options for accommodating the future. It doesn't choose; it leaves you room to move.

The purpose of design is to allow you to do design *later* and its primary goal is to reduce the cost of change.

The Tools of Design

Design is not the act of following a fixed set of rules, it's a journey along a branching path wherein earlier choices close off some options and open access to others. During design you wander through a maze of requirements where every juncture represents a decision point that has consequences for the future.

Just as a sculptor has chisels and files, an object-oriented designer has tools—principles and patterns.