



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

Practical Object-Oriented Design: An Agile Primer Using Ruby

Table of Contents

Cover

Half Title

Title Page

Copyright Page

Dedication

Contents

Introduction

Acknowledgments

About the Author

1 Object-Oriented Design

1.1 In Praise of Design

1.1.1 The Problem Design Solves

1.1.2 Why Change Is Hard

1.1.3 A Practical Definition of Design

1.2 The Tools of Design

1.2.1 Design Principles

1.2.2 Design Patterns

1.3 The Act of Design

1.3.1 How Design Fails

1.3.2 When to Design

1.3.3 Judging Design



Table of Contents

1.4 A Brief Introduction to Object-Oriented Programming

1.4.1 Procedural Languages

1.4.2 Object-Oriented Languages

1.5 Summary

2 Designing Classes with a Single Responsibility

2.1 Deciding What Belongs in a Class

2.1.1 Grouping Methods into Classes

2.1.2 Organizing Code to Allow for Easy Changes

2.2 Creating Classes That Have a Single Responsibility

2.2.1 An Example Application: Bicycles and Gears

2.2.2 Why Single Responsibility Matters

2.2.3 Determining If a Class Has a Single Responsibility

2.2.4 Determining When to Make Design Decisions

2.3 Writing Code That Embraces Change

2.3.1 Depend on Behavior, Not Data

2.3.2 Enforce Single Responsibility Everywhere

2.4 Finally, the Real Wheel

2.5 Summary

3 Managing Dependencies

3.1 Understanding Dependencies

3.1.1 Recognizing Dependencies

3.1.2 Coupling Between Objects (CBO)

3.1.3 Other Dependencies

3.2 Writing Loosely Coupled Code

3.2.1 Inject Dependencies

3.2.2 Isolate Dependencies

3.2.3 Remove Argument-Order Dependencies

3.3 Managing Dependency Direction

Table of Contents

3.3.1 Reversing Dependencies

3.3.2 Choosing Dependency Direction

3.4 Summary

4 Creating Flexible Interfaces

4.1 Understanding Interfaces

4.2 Defining Interfaces

4.2.1 Public Interfaces

4.2.2 Private Interfaces

4.2.3 Responsibilities, Dependencies, and Interfaces

4.3 Finding the Public Interface

4.3.1 An Example Application: Bicycle Touring Company

4.3.2 Constructing an Intention

4.3.3 Using Sequence Diagrams

4.3.4 Asking for What Instead of Telling How

4.3.5 Seeking Context Independence

4.3.6 Trusting Other Objects

4.3.7 Using Messages to Discover Objects

4.3.8 Creating a Message-Based Application

4.4 Writing Code That Puts Its Best (Inter)Face Forward

4.4.1 Create Explicit Interfaces

4.4.2 Honor the Public Interfaces of Others

4.4.3 Exercise Caution When Depending on Private Interfaces

4.4.4 Minimize Context

4.5 The Law of Demeter

4.5.1 Defining Demeter

4.5.2 Consequences of Violations

4.5.3 Avoiding Violations

4.5.4 Listening to Demeter

4.6 Summary

Table of Contents

5 Reducing Costs with Duck Typing

5.1 Understanding Duck Typing

5.1.1 Overlooking the Duck

5.1.2 Compounding the Problem

5.1.3 Finding the Duck

5.1.4 Consequences of Duck Typing

5.2 Writing Code That Relies on Ducks

5.2.1 Recognizing Hidden Ducks

5.2.2 Placing Trust in Your Ducks

5.2.3 Documenting Duck Types

5.2.4 Sharing Code between Ducks

5.2.5 Choosing Your Ducks Wisely

5.3 Conquering a Fear of Duck Typing

5.3.1 Subverting Duck Types with Static Typing

5.3.2 Static versus Dynamic Typing

5.3.3 Embracing Dynamic Typing

5.4 Summary

6 Acquiring Behavior through Inheritance

6.1 Understanding Classical Inheritance

6.2 Recognizing Where to Use Inheritance

6.2.1 Starting with a Concrete Class

6.2.2 Embedding Multiple Types

6.2.3 Finding the Embedded Types

6.2.4 Choosing Inheritance

6.2.5 Drawing Inheritance Relationships

6.3 Misapplying Inheritance

6.4 Finding the Abstraction

6.4.1 Creating an Abstract Superclass

Table of Contents

6.4.2 Promoting Abstract Behavior

6.4.3 Separating Abstract from Concrete

6.4.4 Using the Template Method Pattern

6.4.5 Implementing Every Template Method

6.5 Managing Coupling between Superclasses and Subclasses

6.5.1 Understanding Coupling

6.5.2 Decoupling Subclasses Using Hook Messages

6.6 Summary

7 Sharing Role Behavior with Modules

7.1 Understanding Roles

7.1.1 Finding Roles

7.1.2 Organizing Responsibilities

7.1.3 Removing Unnecessary Dependencies

7.1.4 Writing the Concrete Code

7.1.5 Extracting the Abstraction

7.1.6 Looking Up Methods

7.1.7 Inheriting Role Behavior

7.2 Writing Inheritable Code

7.2.1 Recognize the Antipatterns

7.2.2 Insist on the Abstraction

7.2.3 Honor the Contract

7.2.4 Use the Template Method Pattern

7.2.5 Preemptively Decouple Classes

7.2.6 Create Shallow Hierarchies

7.3 Summary

8 Combining Objects with Composition

8.1 Composing a Bicycle of Parts

8.1.1 Updating the Bicycle Class

Table of Contents

8.1.2 Creating a Parts Hierarchy

8.2 Composing the Parts Object

8.2.1 Creating a Part

8.2.2 Making the Parts Object More Like an Array

8.3 Manufacturing Parts

8.3.1 Creating the PartsFactory

8.3.2 Leveraging the PartsFactory

8.4 The Composed Bicycle

8.5 Deciding between Inheritance and Composition

8.5.1 Accepting the Consequences of Inheritance

8.5.2 Accepting the Consequences of Composition

8.5.3 Choosing Relationships

8.6 Summary

9 Designing Cost-Effective Tests

9.1 Intentional Testing

9.1.1 Knowing Your Intentions

9.1.2 Knowing What to Test

9.1.3 Knowing When to Test

9.1.4 Knowing How to Test

9.2 Testing Incoming Messages

9.2.1 Deleting Unused Interfaces

9.2.2 Proving the Public Interface

9.2.3 Isolating the Object under Test

9.2.4 Injecting Dependencies Using Classes

9.2.5 Injecting Dependencies as Roles

9.3 Testing Private Methods

9.3.1 Ignoring Private Methods during Tests

9.3.2 Removing Private Methods from the Class under Test

Table of Contents

9.3.3 Choosing to Test a Private Method

9.4 Testing Outgoing Messages

9.4.1 Ignoring Query Messages

9.4.2 Proving Command Messages

9.5 Testing Duck Types

9.5.1 Testing Roles

9.5.2 Using Role Tests to Validate Doubles

9.6 Testing Inherited Code

9.6.1 Specifying the Inherited Interface

9.6.2 Specifying Subclass Responsibilities

9.6.3 Testing Unique Behavior

9.7 Summary

Afterword

Index