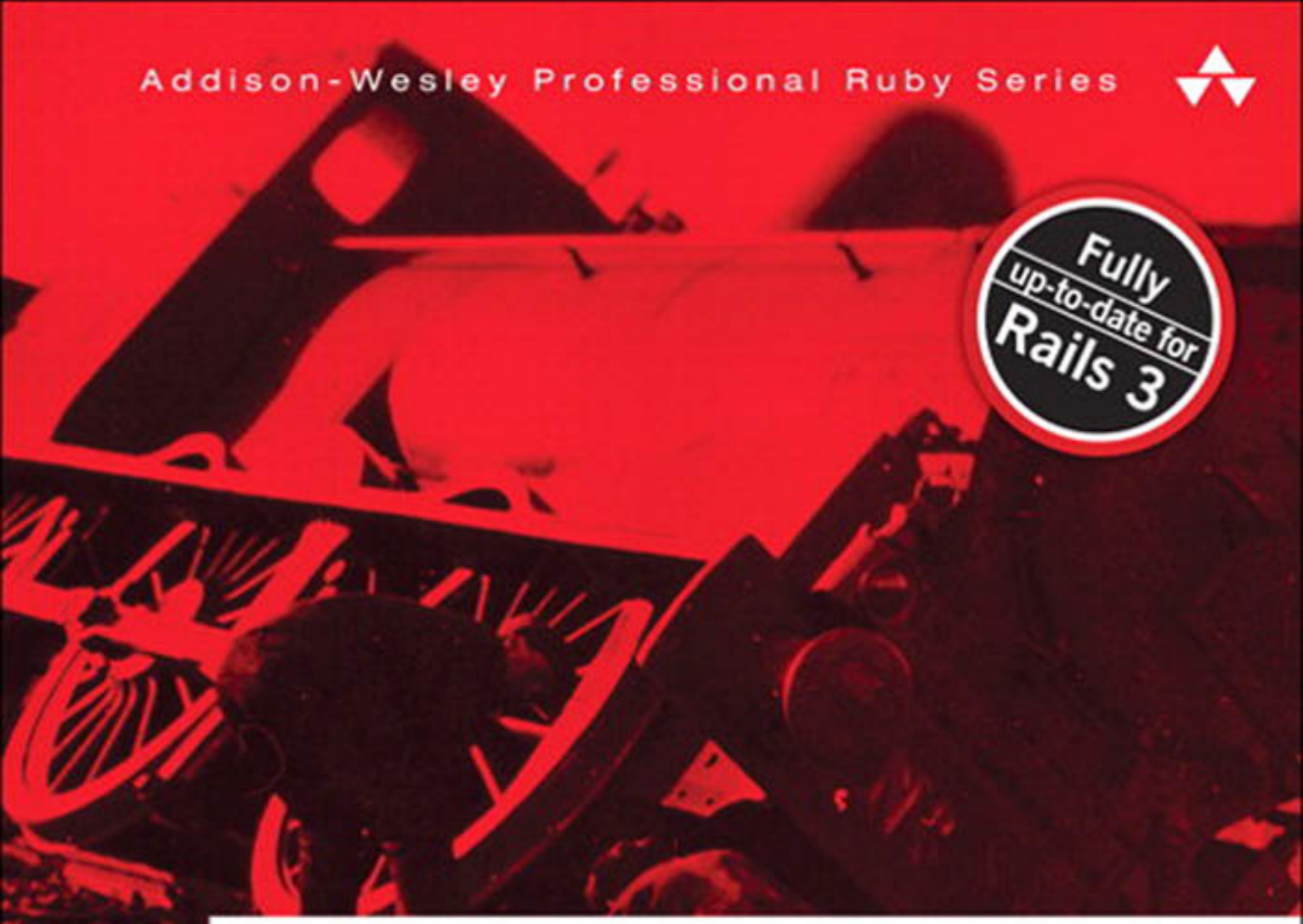
Addison-Wesley Professional Ruby Series



RAILS™ ANTIPATTERNS

Best Practice Ruby on Rails™ Refactoring

CHAD PYTEL ■ TAMMER SALEH

Library of Congress cataloging-in-publication data is on file with the Library of Congress

ISBN-13: 978-0-321-60481-1
ISBN-10: 0-321-60481-4
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, November 2010

Editor-in-Chief
Mark Taub

Acquisitions Editor
Debra Williams Cauley

Development Editor
Michael Thurston

Managing Editor
John Fuller

Project Editor
Elizabeth Ryan

Copy Editor
Kitty Wilson

Indexer
The CIP Group

Proofreader
Linda Begley

Technical Reviewers
Jennifer Lindner
Pat Allen
Joe Ferris
Stephen Caudill
Tim Pope
Robert Pitts
Jim "Big Tiger" Remsik
Lar Van Der Jagt

Publishing Coordinator
Kim Boedigheimer

Cover Designer
Chuti Prasertsith

Compositor
The CIP Group

Contents

Foreword

It's hard to believe that it will soon be three years since Zed Shaw published his infamous (and now retracted) rant "Rails Is a Ghetto." Even though Zed's over-the-top depiction of certain well-known people was wicked and pure social satire, the expression he coined has stuck like the proverbial thorn among certain higher echelons of the community. It's an especially piquant expression to use when we're called on to fix atrocious Rails projects. Occasionally, we'll even use the phrase with regard to our own messes. But most commonly, this expression is applied to code written by the *unwashed masses*. The rapid ascension of Rails as a mainstream technology has attracted droves of eager programmers from both outside and inside the wide sphere of web development. Unfortunately, Rails doesn't discriminate among newcomers. It offers deep pitfalls for bearded wise men of the object-oriented world and PHP script kiddies alike.

Frankly, I would have written this book myself eventually, because there's such a need for it in the marketplace. At Hashrocket, we do a lot of project rescue work. Oh, the agony! We've seen every AntiPattern detailed in this book rear its ugly face in real-life projects. Sometimes we see almost every AntiPattern in this book in a *single* project! My good friends and consultants extraordinaire Chad and Tammer have seen the same horrors. Only fellow consultants like these two could write this book properly because of the wide variety of coding challenges we face regularly. The solutions in this book cover a wide range of sticky situations that we know any professional Ruby developer will run into on a regular basis.

If you're new to Rails (and, based on the demographics, you probably are), then you're now holding one of the most valuable resources possible for getting past the chasm that separates an ordinary Rails developer from greatness. Congratulations and good luck making the leap.

—Obie Fernandez
Author of *The Rails 3 Way*
Series editor of the Addison-Wesley Professional Ruby Series
CEO and founder of Hashrocket

This page intentionally left blank

Introduction

As Rails consultants, we've seen a lot of Rails applications. The majority of the AntiPatterns described in this book are directly extracted from real-world applications. We hope that by formalizing their descriptions here, we can present you with the tools you'll need to identify these AntiPatterns in your own code, understand their causes, and be able to refactor yourself out of the broken patterns.

What Are AntiPatterns?

AntiPatterns are common approaches to recurring problems that ultimately prove to be ineffective.

The term *AntiPatterns* was coined in 1995 by Andrew Koenig, inspired by Gang of Four's book *Design Patterns*, which developed the concept of design patterns in the software field. The term was widely popularized three years later by the book *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (William Brown, Raphael Malveau, Skip McCormick, and Tom Mowbray). According to the authors of *AntiPatterns*, there must be at least two key elements present to formally distinguish an actual AntiPattern from a simple bad habit, bad practice, or bad idea:

- A repeated pattern of action, process, or structure that initially appears to be beneficial but ultimately produces more bad consequences than beneficial results
- A refactored solution that is clearly documented, proven in actual practice, and repeatable

What Is Refactoring?

Refactoring is the act of modifying an application's code not to change its functional behavior but instead to improve the quality of the application itself. These improvements

are intended to improve readability, reduce complexity, increase maintainability, and improve the extensibility (that is, possibility for future growth) of the system.

This book makes extensive reference to the process of refactoring in order to fix code that is exhibiting an AntiPattern. In an attempt to increase readability and understandability of the AntiPatterns and solutions in this book, we've left out the automated test suite that should accompany the code. We want to draw extra attention to the fact that your code should be well tested. When you have tests in place, some of the solutions we've presented will be much easier to implement with confidence. Without tests, some of the solutions might not even be possible. Unfortunately, many of the applications you encounter that exhibit these AntiPatterns will also be untested.

How to Read This Book

Each AntiPattern in this book outlines the mistakes we see in the wild and the negative effects they have on developer velocity, code clarity, maintenance, and other aspects of a successful Rails project. We follow each AntiPattern with one or more solutions that we have used in practice and that have been proven as proper fixes for the AntiPattern.

While you can read this book straight through from front to back, we've taken great pains to make each solution stand on its own. Therefore, this book is both a strong technical publication as well as a quick source of reference for Rails developers looking to hone their techniques in the trenches.

The following is a brief outline of what's covered in each chapter:

- **Chapter 1, "Models":** Because Rails encourages code to be pushed down the Model-View-Controller (MVC) stack to the Model layer, it's fitting that a chapter on models is the largest chapter in the book. Here, we focus on a variety of AntiPatterns that occur in Model layer code, from general object-oriented programming violations to complex SQL and excessive code duplication.
- **Chapter 2, "Domain Modeling":** Going beyond the nitty-gritty code at the Model layer in a Rails project, this chapter focuses on overall schema and database issues. This chapter covers issues such as normalization and serialization.
- **Chapter 3, "Views":** The Rails framework gives developers a large number of tools and conventions that make code in the Model and Controller layers consistent and maintainable. Unfortunately, the required flexibility in the View layer

prevents this sort of consistency. This chapter shows how to make use of the View layer tools Rails provides.

- **Chapter 4, "Controllers":** Since the integration of a RESTful paradigm in the Rails framework, the Controller layer has seen some significant improvements. This chapter goes through the AntiPatterns we've seen in Controller-layer-related Rails code.
- **Chapter 5, "Services":** Dealing with and exposing APIs requires tenacity. This chapter walks through all the common pitfalls we've seen, including timeouts, exceptions, backgrounding, response codes, and more.
- **Chapter 6, "Using Third-Party Code":** This short chapter reviews some of the AntiPatterns that can come from incorporating community plugins and gems into your applications.
- **Chapter 7, "Testing":** One of the strengths of Rails is the strong push toward test-driven development. Unfortunately, we've seen as many AntiPatterns inside test suites as in production code. This chapter outlines these AntiPatterns and how to address them.
- **Chapter 8, "Scaling and Deploying":** Developing a Rails application locally is a great experience, but there are many factors to consider once it's time to release an application to the world. This chapter will help you ensure that your applications are ready for prime time.
- **Chapter 9, "Databases":** This chapter outlines the common issues we've seen with migrations and validations.
- **Chapter 10, "Building for Failure":** Finally, the last chapter in the book gives guidance on general best practices for ensuring that an application degrades gracefully once it encounters the real world.

This page intentionally left blank

CHAPTER 9
Databases

With the Rails framework providing a simple ORM that abstracts many of the database details away from the developer, the database is an afterthought for many Rails developers. While the power of the framework has made this okay to a certain extent, there are important database and Rails-specific considerations that you shouldn't overlook.

AntiPattern: Messy Migrations

Ruby on Rails database migrations were an innovative solution to a real problem faced by developers: How to script changes to the database so that they could be reliably replicated by the rest of the team on their development machines as well as deployed to the production servers at the appropriate time. Before Rails and its baked-in solution, developers often wrote ad hoc database change scripts by hand, if they used them at all.

However, as with most other improvements, database migrations are not without pain points. Over time, a database migration can become a tangle of code that can be intimidating to work with rather than the joy it should be. By strictly keeping in mind the following solutions, you can overcome these obstacles and ensure that your migrations never become irreconcilably messy.

Solution: Never Modify the `up` Method on a Committed Migration

Database migrations enable you to reliably distribute database changes to other members of your team and to ensure that the proper changes are made on your server during deployment.

If you commit a new migration to your source code repository, unless there are irreversible bugs in the migration itself, you should follow the practice of never modifying that migration. A migration that has already been run on another team member's computer or the server will never automatically be run again. In order to run it again, a developer must go through an orchestrated dance of backing the migration down and then up again. It gets even worse if other migrations have since been committed, as that could potentially cause data loss.

Yes, if you're certain that a migration hasn't been run on the server, then it's possible to communicate to the rest of the team that you've changed a migration and have them re-migrate their database or make the required changes manually. However, that's not an effective use of their time, it creates headaches, and it's error prone. It's simply best to avoid the situation altogether and never modify the `up` method of a migration.

Of course, there will be times when you've accidentally committed a migration that has an irreversible bug in it that must be fixed. In such circumstances, you'll have no choice but to modify the migration to fix the bug. Ideally, the times when this happen are few and far between. In order to reduce the chances of this happening, you

should always be sure to run the migration and inspect the results to ensure accuracy *before* committing the migration to your source code repository. However, you shouldn't limit yourself to simply running the migration. Instead, you should run the migration and then run the down of the migration and rerun the up. Rails provides rake tasks for doing this:

```
rake db:migrate
rake db:migrate:redo
```

The `rake db:migrate:redo` command runs the `down` method on the last migration and then reruns the `up` method on that migration. This ensures that the entire migration runs in both directions and is repeatable, without error. Once you've run this and double-checked the results, you can commit your new migration to the repository with confidence.

Solution: Never Use External Code in a Migration

Database migrations are used to manage database change. When the structure of a database changes, very often the data in the database needs to change as well. When this happens, it's fairly common to want to use models inside the migration itself, as in the following example:

```
class AddJobsCountToUser < ActiveRecord::Migration
  def self.up
    add_column :users, :jobs_count, :integer, :default => 0
    Users.all.each do |user|
      user.jobs_count = user.jobs.size
      user.save
    end
  end

  def self.down
    remove_column :users, :jobs_count
  end
end
```

In this migration above, you're adding a counter cache column to the `users` table, and this column will store the number of jobs each user has posted. In this migration,

you're actually using the `User` model to find all users and update the column of each one. There are two problems with this approach.

First, this approach performs horribly. The code above loads all the users into memory and then for each user, one at a time, it finds out how many jobs each has and updates its `count` column.

Second, and more importantly, this migration does not run if the model is ever removed from the application, becomes unavailable, or changes in some way that makes the code in this migration no longer valid. The code in migrations is supposed to be able to be run to manage change in the database, in sequence, at any time. When external code is used in a migration, it ties the migration code to code that is not bound by these same rules and can result in an unrunnable migration.

Therefore, it's always best to use straight SQL whenever possible in your migrations. If you do so, you can rewrite the preceding migration as follows:

```
class AddJobsCountToUser < ActiveRecord::Migration
  def self.up
    add_column :users, :jobs_count, :integer, :default => 0
    update(<<-SQL)
      UPDATE users SET jobs_count = (
        SELECT count(*) FROM jobs
          WHERE jobs.user_id = users.id
      )
    SQL
  end

  def self.down
    remove_column :users, :jobs_count
  end
end
```

When this migration is rewritten using SQL directly, it has no external dependencies beyond the exact state of the database at the time the migration should be executed.

There may be cases in which you actually do need to use a model or other Ruby code in a migration. In such cases, the goal is to rely on no external code in your migration. Therefore, all code that's needed, including the model, should be defined inside the migration itself. For example, if you really want to use the `User` model in the preceding migration, you rewrite it like the following:

```
class AddJobsCountToUser < ActiveRecord::Migration
  class Job < ActiveRecord::Base
  end
  class User < ActiveRecord::Base
    has_many :jobs
  end

  def self.up
    add_column :users, :jobs_count, :integer, :default => 0
    User.reset_column_information
    Users.all.each do |user|
      user.jobs_count = user.jobs.size
      user.save
    end
  end

  def self.down
    remove_column :users, :jobs_count
  end
end
```

Since this migration defines both the `Job` and `User` models, it no longer depends on an external definition of those models being in place. It also defines the `has_many` relationship between them and therefore defines everything it needs to run successfully. In addition, note the call to `User.reset_column_information` in the `self.up` method. When models are defined, Active Record reads the current database schema. If your migration changes that schema, calling the `reset_column_information` method causes Active Record to re-inspect the columns in the database.

You can use this same technique if you must calculate the value of a column by using an algorithm defined in your application. You cannot rely on the definition of that algorithm to be the same or even be present when the migration is run. Therefore, the algorithm should be duplicated inside the migration itself.

Solution: Always Provide a `down` Method in Migrations

It's very important that a migration have a reliable `self.down` defined that actually reverses the migration. You never know when something is going to be rolled back. It's truly bad practice to not have this defined or to have it defined incorrectly.

Some migrations simply cannot be fully reversed. This is most often the case for migrations that change data in a destructive manner. If this is the case for a migration for which you're writing the down method, you should do the best reversal you can do. If you are in a situation where there is a migration that under no circumstances can ever be reversed safely, you should raise an `ActiveRecord::IrreversibleMigration` exception, as shown here:

```
def self.down
  raise ActiveRecord::IrreversibleMigration
end
```

Raising this exception causes migrations to be stopped when this `down` method is run. This ensures that the developer running the migrations understands that there is something irreversible that has been done and that cannot be undone without manual intervention.

Once you have the `down` method defined, you should run the migration in both directions to ensure proper functionality. As discussed earlier in this chapter, in the section "Solution: Never Modify the `up` Method on a Committed Migration," Rails provides rake tasks for doing this:

```
rake db:migrate
rake db:migrate:redo
```

The `rake db:migrate:redo` command runs the `down` method on the last migration and then reruns the `up` method on that migration.

AntiPattern: Wet Validations

Ruby on Rails generally treats a database as a dumb storage device, essentially working only with many of the common-denominator features found in all the databases it supports and eschewing additional database functionality such as foreign keys and constraints. But many Rails developers eventually realize that a database has this functionality built in, and they attempt to use it by trying to duplicate the validation and constraints from their models into the database. For example, the following `User` model has a number of validations:

```
class User < ActiveRecord::Base
  validates :account_id, :presence => true
  validates :first_name, :presence => true
  validates :last_name,  :presence => true

  validates :password, :presence     => true,
                       :confirmation => true,
                       :if           => :password_required?

  validates :email, :uniqueness => true,
                    :format     => { :with => %r{.+@.+\..+} },
                    :presence   => true

  belongs_to :account
end
```

You could attempt to create a database table to back this model that attempts to enforce the same validations at the database level, using database constraints. The (inadequate) migration to create that table might look something like this:

```
self.up
  create_table :users do |t|
    t.column :email,      :string, :null => false
    t.column :first_name, :string, :null => false
    t.column :last_name,  :string, :null => false
    t.column :password,   :string
    t.column :account_id, :integer
  end
  execute "ALTER TABLE users ADD UNIQUE (email)"
```

```
  execute "ALTER TABLE users ADD CONSTRAINT
user_constrained_by_account FOREIGN KEY (account_id) REFERENCES
accounts (id) ON DELETE CASCADE"
end

self.down
  execute "ALTER TABLE users DROP FOREIGN KEY
user_constrained_by_account"
  drop_table :users
end
```

However, there are several reasons this doesn't work in practice. For one thing, not all databases support all the constraints that Active Record supports. For example, in MySQL, it's possible to enforce the uniqueness constraints on email, but none of the other constraints are fully possible without the use of stored procedures and triggers. For example, in the migration earlier in this chapter, there is only a constraint on `NULL` values in the `first_name` column. A blank string would still be allowed to be inserted.

If you are on a database that supports these constraints, you are then left to maintain them all by hand, in duplicate—a process that is tedious and error prone.

Active Record does not handle violations of database constraints well. It does not automatically read the constraints in the database. And if something is out of sync and a constraint in the database is hit, this will result in an exception that is not handled gracefully at the library level. The result is a failure the user sees or one that the programmer must handle, which is impractical.

Solution: Eschew Constraints in the Database

It's simply best to not fight the opinion of Active Record that database constraints are declared in the model and that the database should simply be used as a datastore.

Despite all of the above, you may find yourself working with a DBA who insists that foreign key constraints or other constraints be stored in the database, or you yourself may simply believe in this principle. In such a case, it is strongly recommended that you not attempt to do this by hand and instead use a plugin that provides support for this. One such plugin is Foreigner (http://github.com/matthuhiggins/foreigner/), which provides support for managing foreign key constraints in migrations. Several other well-supported plugins provide support for additional constraints, most of which will be specific to your database server.

There's Always an Exception

In the example we've been looking at in this section, the exception is `NULL` constraints coupled with default database values. Active Record handles these constraints perfectly, with the defaults even being picked up and populated in your model automatically. Therefore, the recommended way to provide default values to your model attributes is by storing the default values in the database. For example, if you want to default a Boolean column to `true`, you can do so in the database:

```
add_column :users, :active, :boolean, :null => false, :default =>
true
```

This will result in the `active` attribute on the user model being set to `true` whenever a new user is created:

```
>> user = User.new
>> user.active?
=> true
```

You can use this swell behavior to your benefit to simplify code and make your objects more consistent. In most applications, setting all Booleans to allow `null` and to default to `false` is preferred. That way, your Booleans will really have only two possible values, true and false, not true, false, and nil.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y