Sustainable Web Development
with **Ruby on Rails**

by **David Bryant Copeland**

Sustainable Web Development with Ruby on Rails

Practical Tips for Building Web Applications that Last

David Bryant Copeland

Contents

Acknowledgements

If there were no such thing as Rails, this book would be, well, pretty strange. So I must acknowledge and deeply thank DHH and the Rails core team for building and maintaining such a wonderful framework for all of us to use.

I have to thank my wife, Amy, who gave me the space and encouragement to work on this. During a global pandemic. When both of us were briefly out of work. And we realized our aging parents require more care than we thought. And when we got two kittens named Carlos Rosario and Zoni. And when we bought a freaking car. And when I joined a pre-seed startup. It's been quite a time.

I also want to thank the technical reviewers, Noel Rappin, Chris Gibson, Zach Campbell, Lisa Sheridan, Raul Murciano, Geoff The, and Sean Miller. Also special thanks to Brigham Johnson for identifying an embarrassing number of typos.

Changes from Previous Versions

This book is intended to be somewhat timeless, and able to be used as a reference. Much of what's in here hasn't changed and I wouldn't expect it to. That said, some things have changed, and this section captures them.

Jan 1, 2025 - This Version

This is updated for Rails 8 and Ruby 3.4.

- General Changes
  - Examples testing and working with Rails 8 and Ruby 3.4.
  - Use Valkey instead of Redis, since Valkey is open source.
  - Re-ordered this section to show recent changes first.
  - `OpenStruct` requires `require "ostruct"` now.
- Chapter 4
  - Brakeman is included with Rails.
- Chapter 9
  - Added additional caveats around functional CSS with respect to custom elements and managing re-use.
  - Removed Tachyons in favor of very basic CSS needed for the app. Since Sprockets is no longer included in Rails and since Tachyons cannot easily be overridden without additional tooling, it seemed easier to re-focus the CSS section on the concept of a design system and style guide. I did consider replacing Tachyons with TailwindCSS, since that is currently popular, but it brought in a ton of complexity that would require a lot of explanation not relevant to the overall focus of the chapter.
  - Included a new subsection emphasizing just how difficult CSS is to manage and to not underestimate it.
- Chapter 10
  - Changed use of `setProgressBarDelay` to `progressBarDelay` (from `Turbo.config.drive`), per deprecation warnings in the JavaScript console.

- Chapter 11
  - Changed the vanilla JS section to use HTML Custom Elements, which should be the preferred approach when not using a framework.
- Chapter 14
  - Expanded on `TIMESTAMPTZ` to explain the care needed in changing the default type for an existing app.
- Chapter 19
  - Changed installation of Foreman to happen inside `bin/setup` instead of from the `Gemfile`, per advice from Foreman's docs.
  - Section with thoughts on how to think about Solid Queue.
- Chapter 20
  - Changed recommendations and thoughts around Action Cable and Active Storage.
- Chapter 24
  - Added section on how to decide if you should use the included `Dockerfile` and/or Kamal.

Dec 4, 2023

This is a more substantial update that previous updates. Chapter numbers refer to the PDF or printed book's numbering. e-book numbering continues to be a byzantine nightmare.

- General Changes
  - Updated for Rails 7.1.
  - Updated for Ruby 3.2
  - Added explicit language in each section about where to find the sample code for that section.
  - New cover
- Chapter 1
  - Update my experience, given the passage of time.
- Chapter 4
  - Remove mention of Spring and Listen, since they aren't included and haven't been in a few versions.
  - Remove mention of having to add the rexml gem, since selenium-webdriver brings it in.
  - Change `bin/run` to `bin/dev`, since this matches what Rails does (sometimes).

  - Remove mention of having to `bundle update` Thor.
  - Added help flags to the various `bin/` scripts.

- Chapter 5
  - Added a new section that references "Patterns of Enterprise Application Architecture", since this where the active record pattern originated.

- Chapter 7
  - Recommend the use of View Components
  - Recommend strict locals for partials

- Chapter 8
  - Clarify that helpers *can* be made to be modular, and discuss configuring Rails to either treat them that way or to not generate falsely-modular helpers.
  - More strongly discourage presenter-like libraries, and remove a lot of content around managing them.
  - In place of presenters, discuss how using Active Model or View Components can manage complexity instead of gobs of helpers.

- Chapter 9
  - Clear warning about Tailwind's lack of built-in design system and what you should consider if adopting it.

- Chapter 11
  - Qualify the recommendation for Hotwire given that 37 Signals have made it clear they will change it however they like whenever they like.

- Chapter 15
  - Reference "Patterns of Enterprise Application Architecture" and its definition of a *service layer*, which is what this chapter describes.
  - Make it clear that the term "Service Objects" is not a service layer and is actually just another name for the command pattern (and that you should not use this pattern).

- Chapter 16
  - Replace use of `before_validation` callback with the new `normalizes` macro
  - Make a stronger case for not using callbacks by clarifying exactly what they do and are for.

- Chapter 17
  - Replace the re-usable partial with a View Component in the example.

- Chapter 23
  - Show code to monkey-patch Thor to make it useful for Rails generators.
  - Discourage the use of app templates in favor of template repositories.
- Chapter 24
  - Use `CurrentAttributes` to store information for the log instead of thread local storage.
  - Discuss the need to revisit security practices, along with an anecdote from a previous job.
- Appendix A
  - Re-work Docker stuff based on updated learnings and code.
  - Explainer on getting your own shell aliases or software into the dev container.

March 15, 2022

- Updated for Rails 7
- Removal of all NodeJS-related stuff, including removal and re-thinking of the value of unit-testing JavaScript.
- Softened language around using React by default given Hotwire's existence.
- Changed guidance around nested routes to account for content-heavy marketing pages.
- Clarified the use of controller instance variables for managing UI state.
- Links to gems extracted from code based on the book.

January, 21, 2021

- No need to disable Ajax form submissions by default, since Rails 6.1 changed the default behavior.
- Use of `add_check_constraint` and `add_index` instead of SQL wrapped in `reversible`.
- Fixed color issues with sidebars on some e-readers

December, 12, 2020

- Updated for Rails 6.1 to remove deprecated method of setting errors on Active Records

PART

I

introduction

1

Why This Book Exists

Rails can scale. But what does that actually mean? And how do we do it? This book is the answer to both of these questions, but instead of using "scalable", which many developers equate with "fast performance", I'm using the word "sustainable". This is really what we want out of our software: the ability to sustain that software over time.

Rails itself is an important component in sustainable web development, since it provides common solutions to common problems and has reached a significant level of maturity. But it's not the complete picture.

Rails has a lot of features and we may not need them all. Or, we may need to take some care in how we use them. Rails also leaves gaps in your application's architecture that you'll have to fill (which makes sense, since Rails can't possibly provide *everything* your app will need).

This book will help you navigate all of that.

Before we begin, I want to be clear about what *sustainability* means and why it's important. I also want to state the assumptions I'm making in writing this, because there is no such thing as universal advice—there are only recommendations that apply in a given context.

1.1 What is Sustainability?

The literal interpretation of sustainable web development is web development that can be sustained. As silly as that definition is, I find it an illuminating restatement.

To *sustain* the development of our software is to ensure that it can continue to meet its needs. A sustainable web app can easily suffer new requirements, increased demand for its resources, and an increasing (or changing) team of developers to maintain it.

A system that is hard to change is hard to sustain. A system that can't avail itself of the resources it needs to function is hard to sustain. A system that only *some* developers can work on is hard to sustain.

Thus, a sustainable application is one in which changes we make tomorrow are as easy as changes are today, for whatever the application might need to do and whoever might be tasked with working on it.

So this defines *sustainability*, but why is it important?

1.2 Why Care About Sustainability?

Most software exists to meet some need, and if that need will persist over time, so must the software. *Needs* are subjective and vague, while software must be objective and specific. Thus, building software is often a matter of continued refinement as the needs are slowly clarified. And, of course, needs have a habit of changing along the way.

Software is expensive, mostly owing to the expertise required to build and maintain it. People who can write software find their skills to be in high demand, garnering some of the highest wages in the world, even at entry levels. It stands to reason that if a piece of software requires more effort to enhance and maintain over time, it will cost more and more and deliver less and less.

In an economic sense, sustainable software minimizes the cost of the software over time. But there is a human cost to working on software. Working on sustainable software is, well, more enjoyable. They say employees quit managers, but I've known developers that quit codebases. Working on unsustainable software just plain sucks, and I think there's value in having a job that doesn't suck. . . at least not all of the time.

Of course, it's one thing to care about sustainability in the abstract, but how does that translate into action?

1.3 How to Value Sustainability

Sustainability is like an investment. It necessarily won't pay off in the short term and, if the investment isn't sound, it won't ever pay off. So it's really important to understand the value of sustainability to your given situation and to have access to as much information as possible to know exactly how to invest in it.

Predicting the future is dangerous for programmers. It can lead to over-engineering, which makes certain classes of changes more difficult in the future. To combat this urge, developers often look to the tenets of agile software development, which have many cute aphorisms that boil down to "don't build software that you don't know you need".

If you are a hired consultant, this is excellent advice. It gives you a framework to be successful and manage change when you are in a situation where you have very little access to information. The strategy of "build for only what you 100% know you need" works great to get software shipped with confidence, but it doesn't necessarily lead to a sustainable outcome.

For example, no business person is going to ask you to write log statements so you can understand your code in production. No product owner is going

to ask you to create a design system to facilitate building user interfaces more quickly. And no one is going to require that your database have referential integrity.

The features of the software are merely one input into what software gets built. They are a significant one just not the only one. To make better technical decisions, you need access to more information than simply what someone wants the software to do.

Do you know what economic or behavioral output the software exists to produce? In other words, how does the software make money for the people paying you to write it? What improvements to the business is it expected to make? What is the medium or long-term plan for the business? Does it need to grow significantly? Will there need to be increased traffic? Will there be an influx of engineers? Will they be very senior, very junior, or a mix? When will they be hired and when will they start?

The more information you can get access to the better, because all of this feeds into your technical decision-making and can tell you just how sustainable your app needs to be. If there will be an influx of less experienced developers, you might make different decisions than if the team is only hiring one or two experienced specialists.

Armed with this sort of information, you can make technical decisions as part of an overall *strategy*. For example, you may want to spend several days setting up a more sustainable development environment. By pointing to the company's growth projections and your team's hiring plans, that work can be easily justified (see the sidebar "Understanding Growth At Stitch Fix" on the next page for a specific example of this).

If you don't have the information about the business, the team, or anything other than what some user wants the software to do, you aren't set up to do sustainable development. But it doesn't mean you shouldn't ask anyway.

People who don't have experience writing software won't necessarily intuit that such information is relevant, so they might not be forthcoming. But you'd be surprised just how much information you can get from someone by asking.

Whatever the answers are, you can use this as part of an overall technical strategy, of which sustainability is a part. As you read this book, I'll talk about the considerations around the various recommendations and techniques. They might not all apply to your situation, but many of them will.

Which brings us to the set of assumptions that this book is based on. In other words, what *is* the situation in which sustainability is important and in which this book's recommendations apply?

Understanding Growth At Stitch Fix

During my first few months at Stitch Fix, I was asked to help improve the operations of our warehouse. There were many different processes and we had a good sense of which ones to start automating. At the time, there was only one application—called HELLBLAZER—and it served up `stitchfix.com`.

If I hadn't been told anything else, the simplest thing to do would've been to make a `/warehouse` route in HELLBLAZER and slowly add features for the associates there. But I *had* been told something else.

Like almost everyone at the company, the engineering team was told—very transparently—what the growth plans for the business were. It needed to grow in a certain way or the business would fail. It was easy to extrapolate from there what that would mean for the size of the engineering team, and for the significance of the warehouse's efficiency. It was clear that a single codebase everyone worked in would be a nightmare, and migrating away from it later would be difficult and expensive.

So, we created a new application that shared HELLBLAZER's database. It would've certainly been faster to add code to HELLBLAZER directly, but we knew doing so would burn us long-term. As the company grew, the developers working on warehouse software were fairly isolated since they worked in a totally different codebase. We replicated this pattern and, after six years of growth, it was clearly the right decision, even accounting for problems that happen when you share a database between apps.

We never could've known that without a full understanding of the company's growth plans, and long-term vision for the problems we were there to solve.

1.4 Assumptions

This book is prescriptive, but each prescription comes with an explanation, and *all* of the book's recommendations are based on some key assumptions that I would like to state explicitly. If your situation differs wildly from the one described below, you might not get that much out of this book. My hope—and belief—is that the assumptions below are common, and that the situation of writing software that you find yourself in is similar to situations I have faced. Thus, this book will help you.

In case it's not, I want to state my assumptions up front, right here in this free chapter.

1.4.1 The Software Has a Clear Purpose

This might seem like nonsense, but there are times when we don't exactly know what the software is solving for, yet need to write some software to explore the problem space.

Perhaps some venture capitalist has given us some money, but we don't yet know the exact market for our solution. Maybe we're prototyping a potentially complex UI to do user testing. In these cases we need to be nimble and try to figure out what the software should do.

The assumption in this book is that that has already happened. We know generally what problem we are solving, and we aren't going to have to pivot from selling shoes to providing AI-powered podiatrist back-office enterprise software.

1.4.2 The Software Needs To Exist For Years

This book is about how to sustain development over a longer period of time than a few months, so a big assumption is that the software actually *needs* to exist that long!

A lot of software falls into this category. If you are automating a business process, building a customer experience, or integrating some back-end systems, it's likely that software will continue to be needed for quite a while.

1.4.3 The Software Will Evolve

Sometimes we write code that solves a problem and that problem doesn't change, so the software is stable. That's not an assumption I am making here. Instead, I'm assuming that the software will be subject to changes big and small over the years it will exist.

I believe this is more common than not. Software is notoriously hard to get right the first time, so it's common to change it iteratively over a long period to arrive at optimal functionality. Software that exists for years also tends to need to change to keep up with the world around it.

1.4.4 The Team Will Change

The average tenure of a software engineer at any given company is pretty low, so I'm assuming that the software will outlive the team, and that the group of people charged with the software's maintenance and enhancement will change over time. I'm also assuming the experience levels and skill-sets will change over time as well.

1.4.5 You Value Sustainability, Consistency, and Quality

Values are fundamental beliefs that drive actions. While the other assumptions might hold for you, if you don't actually value sustainability, consistency, and quality, this book isn't going to help you.

Sustainability

If you don't value sustainability as I've defined it, you likely didn't pick up this book or have stopped reading by now. You're here because you think sustainability is important, thus you *value* it.

Consistency

Valuing consistency is hugely important as well. Consistency means that designs, systems, processes, components (etc.), should not be arbitrarily different. Same problems should have same solutions, and there should not be many ways to do something. It also means being explicit that personal preferences are not critical inputs to decision-making.

A team that values consistency is a sustainable team and will produce sustainable software. When code is consistent, it can be confidently abstracted into shared libraries. When processes are consistent, they can be confidently automated to make everyone more productive.

When architecture and design are consistent, knowledge can be transferred, and the team, the systems, and even the business itself can survive potentially radical change (see the sidebar "Our Uneventful Migration to AWS" on the next page for how Stitch Fix capitalized on consistency to migrate from Heroku to AWS with no downtime or outages).

Quality

Quality is a vague notion, but it's important to both understand it and to value it. In a sense, valuing quality means doing things right the first time. But "doing things right" doesn't mean over-engineering, gold-plating, or doing something fancy that's not called for.

Valuing quality is to acknowledge the reality that we aren't going to be able to go back and clean things up after they have been shipped. There is this fantasy developers engage in that they can simply "acquire technical debt" and someday "pay it down".

I have never seen this happen, at least not in the way developers think it might. It is extremely difficult to make a business case to modify working software simply to make it "higher quality". Usually, there must be some catastrophic failure to get the resources to clean up a previously-made mess. It's simpler and easier to manage a process by which messes don't get made as a matter of course.

Quality should be part of the everyday process. Doing this consistently will result in predictable output, which is what managers really want to see. On the occasion when a date must be hit, cut scope, not corners. Only the developers know what scope to cut in order to get meaningfully faster delivery, but this requires having as much information about the business strategy as possible.

When you value sustainability, consistency, and quality, you will be unlikely to find yourself in a situation where you must undo a technical decision you made at the cost of shipping more features. Business people may want software delivered as fast as possible, but they *really* don't want to go an extended period without any features so that the engineering team can "pay down" technical debt.

We know what sustainability is, how to value it, what assumptions I'm making going in, and the values that drive the tactics and strategy for the rest of the book. But there are two concepts I want to discuss that allow us to attempt to quantify just how sustainable our decisions are: opportunity costs and carrying costs.

Our Uneventful Migration to AWS

For several years, Stitch Fix used the platform-as-a-service Heroku. We were consistent in how we used it, as well as in how our applications were designed. We used one type of relational database, one type of cache, one type of CDN, etc.

In our run-up to going public, we needed to migrate to AWS, which is *very* different from Heroku. We had a team of initially two people and eventually three to do the migration for the 100+ person engineering team. We didn't want downtime, outages, or radical changes in the developer experience.

Because everything was so consistent, the migration team was able to quickly build a deployment pipeline and command-line tool to provide a Heroku-like experience to the developers. Over several months we migrated one app and one database at a time. Developers barely noticed, and our users and customers had no idea.

The project lead was so confident in the approach and the team that he kept his scheduled camping trip to an isolated mountain in Colorado, unreachable by the rest of the team as they moved `stitchfix.com` from Heroku to AWS to complete the migration. Consistency was a big part of making this a non-event.

1.5 Opportunity and Carrying Costs

An *opportunity cost* is basically a one-time cost to produce something. By committing to work, you necessarily cut off other avenues of opportunity. This cost can be a useful lens to compare two different approaches when trying to perform a cost/benefit analysis. An opportunity cost that we'll take in a few chapters is writing robust scripts for setting up our app, running it, and running its tests. It has a higher opportunity cost than simply writing documentation about how to do those things.

But sometimes an investment is worth making. The way to know if that's true is to talk about the *carrying cost*. A carrying cost is a cost you have to

pay all the time every time. If it's difficult to run your app in development, reading the documentation about how to do so and running all the various commands is a cost you pay frequently.

Carrying costs affect sustainability more than anything. Each line of code is a carrying cost. Each new feature has a carrying cost. Each thing we have to remember to do is a carrying cost. This is the true value provided by Rails: it reduces the carrying costs of a lot of pretty common patterns when building a web app.

To sustainably write software requires carefully balancing your carrying costs, and strategically incurring opportunity costs that can reduce, or at least maintain, your carrying costs.

If there are two concepts most useful to engineers, it is these two.

The last bit of information I want to share is about me. This book amounts to my advice based on my experience, and you need to know about that, because, let's face it, the field of computer programming is pretty far away from science, and most of the advice we get is nicely-formatted survivorship bias.

1.6 Why should you trust me?

Software engineering is notoriously hard to study and most of what exists about how to write software is anecdotal evidence or experience reports. This book is no different, but I do believe that if you are facing problems similar to those I have faced, there is value in here.

So I want to outline what my experience is that has led to me recommend what I do in this book.

The most important thing to know about me is that I'm not a software consultant, nor have I been in a very long time. For the past fifteen years I have been a product engineer (or part of a project engineering team), working for companies building one or more products designed to last. I was a rank and file engineer at times, a manager on occasion, an architect responsible for technical strategy and, most recently, Chief Technology Officer (CTO) at a venture-backed startup. I've written a lot of code and set a lot of technical and product strategy.

What this means is that the experience upon which this book is based comes from actually building software meant to be sustained. I have actually done—and seen the long-term results of doing—pretty much everything in this book. I've been responsible for sustainable software several times during my career.

- I spent four years at an energy startup that sold enterprise software. I saw the product evolve from almost nothing to a successful company with many clients and over 100 engineers. While the software was

Java-based, much of what I learned about sustainability applies to the Rails world as well.
- I spent the next year and half at an e-commerce company that had reached what would be the peak of its success. I joined a team of almost 200 engineers, many of whom were working in a huge Rails monolith that contained thousands of lines of code, all done "The Rails Way". The team had experienced massive growth and this growth was not managed. The primary application we all worked in was wholly unsustainable and had a massive carrying cost simply existing.
- I then spent the next six and half years at Stitch Fix, where I was the third engineer and helped set the technical direction for the team. By the time I left, the team was 200 engineers, collectively managing a microservices-based architecture of over 50 Rails applications, many of which I contributed to. At that time I was responsible for the overall technical strategy for the team and was able to observe which decisions we made in 2013 ended up being good (or bad) by 2019.
- I was CTO of a healthcare startup, having written literally the first line of code, navigating the tumultuous world of finding product/market fit, becoming HIPAA[1]-compliant, and trying to never be a bottleneck for what the company needed to do.

What I don't have much experience with is working on short-term greenfield projects, or being dropped into a mess to help clean it up (so-called "Rails Rescue" projects). There's nothing wrong with this kind of experience, but that's not what this book is about.

What follows is what I tried to take away from the experience above, from the great decisions my colleagues and I made, to the unfortunate ones as well (I pushed hard for both Coffeescript and Angular 1 and we see how those turned out).

But, as they say, your mileage may vary, "it depends", and everything is a trade-off. I will do my best to clarify the trade-offs.

Up Next

This chapter should've given you a sense of what you're in for and whether or not this book is for you. I hope it is!

So, let's move on. Because this book is about Ruby on Rails, I want to give an overview of the application architecture Rails provides by default, and how those pieces relate to each other. From that basis, we can then deep dive into each part of Rails and learn how to use it sustainably.

[1]HIPAA is the Health Insurance Portability and Accountability Act, a curious law in the United States related to how healthcare information is managed. Like all compliance-related frameworks, it thwarts sustainability, but it's a fact of life in the U.S.

2

Business Logic (Does Not Go in Active Records)

Much of this book contains strategies and tactics for managing each part of Rails in a sustainable way. But there is one part of every app that Rails doesn't have a clear answer for: the *business logic*.

Business logic is the term I'm going to use to refer to the core logic of your app that is specific to whatever your app needs to do. If your app needs to send an email every time someone buys a product, but only if that product ships to Vermont, unless it ships from Kansas in which case you send a text message... this is business logic.

The biggest question Rails developers often ask is: where does the code for this sort of logic go? Rails doesn't have an explicit answer. There is no `ActiveBusinessLogic::Base` class to inherit from nor is there a `bin/rails generate business-logic` command to invoke.

This chapter outlines a simple strategy to answer this question: do not put business logic in Active Records. Instead, put each bit of logic in its own class, and put all those classes somewhere inside `app/` like `app/services` or `app/businesslogic`.

The reasons don't have to do with moral purity or adherence to some object-oriented design principles. They instead relate directly to sustainability by minimizing the impact of bugs found in business logic. That said, Martin Fowler—who popularized the active record pattern upon which Active Record is based—does not recommend putting all business logic in active records, either.

We'll learn that business logic code is both more complex and less stable than other parts of the codebase. We'll then talk about *fan-in* which is a rough measure of the inter-relations between modules in our system. We'll bring those concepts together to understand how bugs in code used broadly in the app—such as Active Records—can have a more serious impact than bugs in isolated code.

So, let's jump in. What's so special about business logic?

2.1 Business Logic Makes Your App Special. . . and Complex

Rails is optimized for so-called *CRUD*, which stands for "Create, Read, Update, and Delete". In particular, this refers to the database: we create database records, read them back out, update them, and sometimes delete them.

Of course, not every operation our app needs to perform can be thought of as manipulating a database table's contents. Even when an operation requires making changes to multiple database tables, there is often other logic that has to happen, such as conditional updates, data formatting and manipulation, or API calls to third parties.

This logic can often be complex, because it must bring together all sorts of operations and conditions to achieve the result that the domain requires it to achieve.

This sort of complexity is called *necessary complexity* (or *essential* complexity) because it can't be avoided. Our app has to meet certain requirements, even if they are highly complex. Managing this complexity is one of the toughest things to do as an app grows.

2.1.1 Business Logic is a Magnet for Complexity

While our code has to implement the necessary complexity, it can often be even more complex due to our decisions about how the logic gets implemented. For example, we may choose to manage user accounts in another application and make API calls to it. We didn't *have* to do that, and our domain doesn't require it, but it might be just the way we ended up building it. This kind of complexity is called *accidental* or *unnecessary* complexity.

We can never avoid *all* accidental complexity, but the distinction to necessary complexity is important, because we do have at least limited control over accidental complexity. The better we manage that, the better able we are to manage the code to implement the necessarily complex logic of our app's domain.

What this means is that the code for our business logic is going to be more complex than other code in our app. It tends to be a magnet for complexity, because it usually contains the necessarily complex details of the domain as well as whatever accidentally complexity that goes along with it.

To make matters worse, business logic also tends to change frequently.

2.1.2 Business Logic Experiences Churn

It's uncommon for us to build an app and then be done with it. At best, the way we build apps tends to be iterative, where we refine the implementation using feedback cycles to narrow in on the best implementation. Software

is notoriously hard to specify, so this feedback cycle tends to work the best. And that means changes, usually in the business logic. Changes are often called *churn*, and areas of the app that require frequent changes have *high churn*.

Churn doesn't necessarily stop after we deliver the first version of the app. We might continue to refine it, as we learn more about the intricacies of the problem domain, or the world around might change, requiring the app to keep up.

This means that the part of our app that is special to our domain has high complexity and high churn. *That* means it's a haven for bugs.

North Carolina State University researcher Nachiappan Nagappan, along with Microsoft employee Richard Ball demonstrated this relationship in their paper "Use of Relative Code Churn Measures to Predict System Defect Density"[1], in which they concluded:

> Increase in relative code churn measures is accompanied by an increase in system defect density [number of bugs per line of code]

Hold this thought for a moment while we learn about another concept in software engineering called *fan-in*.

2.2 Bugs in Commonly-Used Classes Have Wide Effects

Let's talk about the inter-dependence of pieces of code. Some methods are called in only one place in the application, while others are called in multiple places.

Consider a controller method. In most Rails apps, there is only one way a controller method gets called: when an HTTP request is issued to a specific resource with a specific method. For example, we might issue an HTTP GET to the URL `/widgets`. That will invoke the `index` method of the `WidgetsController`.

Now consider the method `find` on `User`. *This* method gets called in *many* more places. In applications that have authentication, it's possible that `User.find` is called on almost every request.

Thus, if there's a problem with `User.find`, most of the app could be affected. On the other hand, a problem in the `index` method of `WidgetsController` will only affect a small part of the app.

We can also look at this concept at the class level. Suppose `User` instances are part of most pieces of code, but we have another model called `WidgetFaxOrder` that is used in only a few places. Again, it stands to

[1] https://www.st.cs.uni-saarland.de/edu/recommendation-systems/papers/ICSE05Churn.pdf

reason that bugs in `User` will have wider effects compared to bugs in `WidgetFaxOrder`.

While there are certain other confounding factors (perhaps `WidgetFaxOrder` is responsible for most of our revenue), this lens of class dependencies is a useful one.

The concepts here are called *fan-out* and *fan-in*. Fan-out is the degree to which one method or class calls into other methods or classes. Fan-in is what I just described above and is the inverse: the degree to which a method or class is *called* by others.

What this means is that bugs in classes or methods with a high fan-in—classes used widely throughout the system—can have a much broader impact on the overall system than bugs in classes with a low fan-in.

Consider the system diagrammed in the figure below. We can see that `WidgetFaxOrder` has a low fan-in, while `Widget` has a high one. `WidgetFaxOrder` has only one incoming "uses" arrow pointing to it. `Widget` has two incoming "uses" arrows, but is also related via Active Record to two other classes.

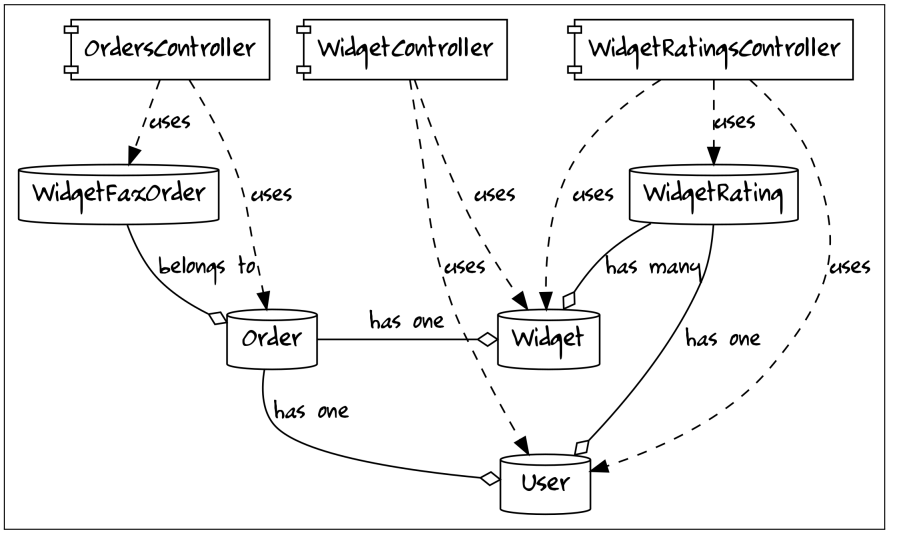

Figure 2.1: System Diagram to Understand Fan-in

Consider a bug in `WidgetFaxOrder`. The figure "Bug Effects of a Low Fan-in Module" on the next page outlines the effected components. This shows that because `WidgetFaxOrder` has a bug, it's possible that `OrdersController` is also buggy, since it relies on `WidgetFaxOrder`. The diagram also shows that it's highly unlikely that any of the rest of the system is affected, because those parts don't call into `WidgetFaxOrder` or any class that does. Thus, we are seeing a worst case scenario for a bug in `WidgetFaxOrder`.

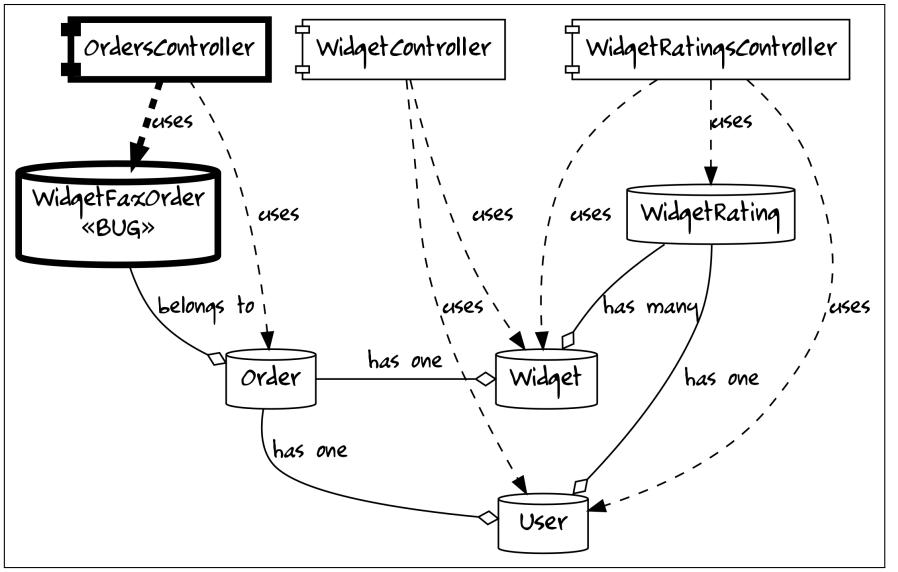

Figure 2.2: Bug Effects of a Low Fan-in Module

Now consider if instead `Widget` has a bug. The figure "Bug Effects of a High Fan-in Module" on the next page shows how a broken `Widget` class could have serious effects throughout the system in the worst case. Because it's used directly by two controllers and possibly indirectly by another through the Active Record relations, the potential for the Widget class to cause a broad problem is much higher than for `WidgetFaxOrder`.

It might seem like you could gain a better understanding of this problem by looking at the method level, but in an even moderately complex system, this is hard to do. The system diagrammed here is vastly simplified.

What this tells me is that the classes that are the most central to the app have the highest potential to cause serious problems. Thus it is important to make sure those classes are working well to prevent these problems.

A great way to do that is to minimize the complexity of those classes as well as to minimize their churn. Do you see where I'm going?

2.3 Business Logic in Active Records Puts Churn and Complexity in Critical Classes

We know that the code that implements business logic is among the most complex code in the app. We know that it's going to have high churn. We know that these two factors mean that business logic code is more likely to have bugs. And we also know that bugs in classes widely used throughout the app can cause more serious systemic problems.

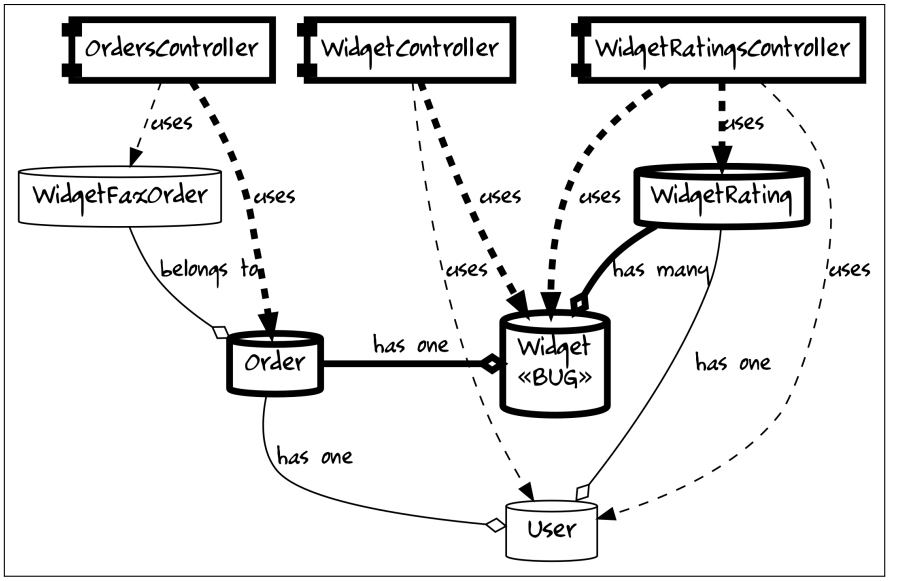

Figure 2.3: Bug Effects of a High Fan-in Module

So why would we put the code most likely to have bugs in the classes most widely used in the system? Wouldn't it be extremely wise to keep the complexity and churn on high fan-in classes—classes used in many places—as low as possible?

If the classes most commonly used throughout the system were very stable, and not complex, we minimize the chances of system-wide bugs caused by one class. If we place the most complex and unstable logic in isolated classes, we minimize the damage that can be done when those classes have bugs, which they surely will.

Let's revise the system diagram to show business logic functions on the Active Records. This will allow us to compare two systems: one in which we place all business logic on the Active Records themselves, and another where that logic is placed on isolated classes.

Suppose that the app shown in the diagram has these features:

- Purchase a widget
- Purchase a widget by fax
- Search for a widget
- Show a widget
- Rate a widget
- Suggest a widget rated similar to another widget you rated highly

I've added method names to the Active Records where these might go in the figure "System with Logic on Active Records" on the next page. You might

put these methods on different classes or name them differently, but this should look pretty reasonable for an architecture that places business logic on the Active Records.

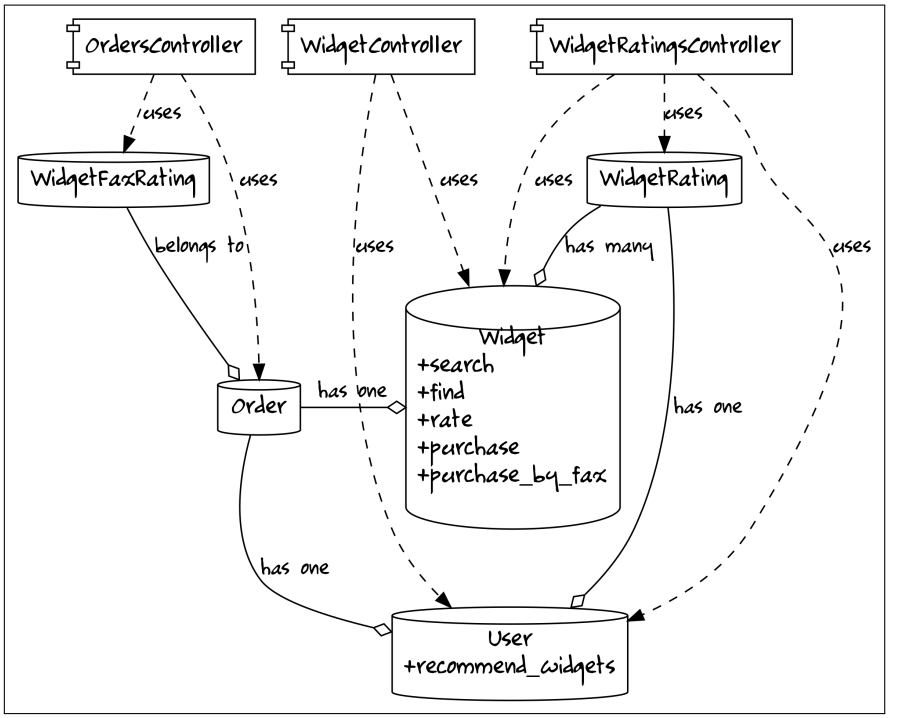

Figure 2.4: System with Logic on Active Records

Now consider an alternative. Suppose that each bit of business logic had its own class apart from the Active Records. These classes accept Active Records as arguments and use the Active Records for database access, but they have all the logic themselves. They form a *service layer* between the controllers and the database. We can see this in the figure below.

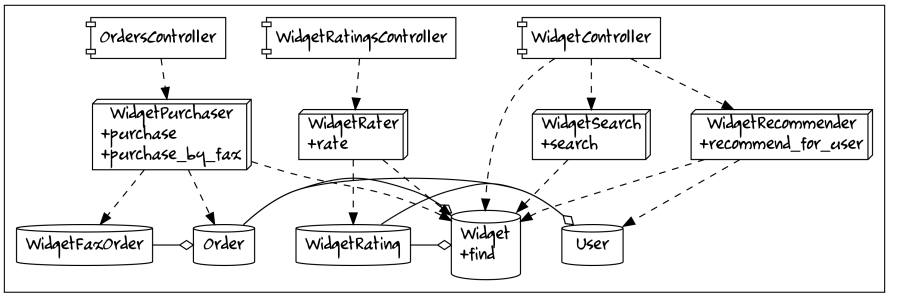

Figure 2.5: System with Business Logic Separated

Granted, there are more classes, so this diagram has more paths and seems more complex, but look at the fan-in of our newly-introduced service layer (the classes in 3-D boxes). All of them have low fan-in. This means that a bug in those classes is likely to be contained. And because those classes are the ones with the business logic—by definition the code likely to contain the most bugs—the effect of those bugs is minimized.

And *this* is why you should not put business logic in your Active Records. There's no escaping a system in which a small number of Active Records are central to the functionality of the app. But we can minimize the damage that can be caused by making those Active Records stable and simple. And to do that, we simply don't put logic on them at all.

There are some nice knock-on effects of this technique as well. The business logic tends to be in isolated classes that embody a domain concept. In our hypothetical system above, one could imagine that `WidgetPurchaser` encapsulates all the logic about purchasing a widget, while `WidgetRecommender` holds the logic about how we recommend widgets.

Both use `Widget` and `User` classes, which don't represent any particular domain concept beyond the attributes we wish to store in the database. And, as the app grows in size and features, as we get more and more domain concepts which require code, the `Widget` and `User` classes won't grow proportionally. Neither will `WidgetRecommender` nor `WidgetPurchaser`. Instead, we'll have new classes to represent those concepts.

In the end, you'll have a system where churn is isolated to a small number of classes, depended-upon by a few number of classes. This makes changes safer, more reliable, and easier to do. That's sustainable.

But don't take my word for it. Martin Fowler, the person who coined and first described the active record pattern that was inspiration for this part of Rails encourages this as well, when your application is complex.

2.4 Active Records Were Never Intended to Hold All the Business Logic

You may think that since Rails includes an implementation of the active record *pattern*, and that pattern is loosely defined as an object that adds domain logic to database data, we should follow the pattern the Rails Way and put our logic on our Active Records.

Let's set aside that this is an appeal to authority and let's also set aside that 99% of Active Record's documentation and 100% of its API are about database access. Is this actually what Martin Fowler, the author of *Patterns of Enterprise Application Architecture*, intended? No.

Early in the book, Fowler talks about business logic:

> Many designers, including me, like to divide "business logic" into two kinds: "domain logic," having to do purely with the problem domain (such as strategies for calculating revenue recognition on a contract), and "application logic," having to do with application responsibilities... sometimes referred to as "workflow logic".

Later, when talking about the active record pattern, he is clear that the logic you'd couple to your database schema is *domain* logic only:

> Each Active Record is responsible for saving and loading to the database and also for any domain logic that acts on the data.

"Domain logic that acts on the data" is certainly a subset of your application's business logic. For one, it doesn't include application logic, as defined by Fowler. Secondly, it doesn't include domain logic that doesn't "act on data". Fowler goes on to clarify this point:

> Active Record is a good choice for domain logic that isn't too complex, such as creates, reads, updates, and deletes. Derivations and validations based on a single record work well in this structure... If your business logic is complex, you'll soon want to use your object's direct relationships, collections, inheritance, and so forth. These don't map easily onto Active Record, and adding them piecemeal gets very messy.

I have never worked on an application that was so simple it could keep all of its logic in the Active Records. But I have definitely worked on applications where application logic and database-agnostic domain logic were crammed into the Active Records. It was not sustainable.

I mention this to really underscore that it's not just me telling you not to put all your business logic in Active Records. The guy that came up with it also doesn't think you should do that.

OK, let's see an example of some code that doesn't put business logic in the Active Records.

2.5 Example Design of a Feature

Suppose we are building a feature to edit widgets. Here is a rough outline of the requirements around how it should work:

1. A user views a form where they can edit a widget's metadata.
2. The user submits the form with a validation error.
3. The form is re-rendered showing their errors.
4. The user corrects the error and submits the edit again.
5. The system then updates the database.

6. When the widget is updated, two things have to happen:
    1. Depending on the widget's manufacturer, we need to notify an admin to approve of the changes
    2. If the widget is of a particular type, we must update an inventory table used for reporting.
7. The user sees a result screen.
8. Eventually, an email is sent to the right person.

This is not an uncommon amount of complexity. We will have to write a bit of code to make this work, and it's necessarily going to be in several places. A controller will need to receive the HTTP request, a view will need to render the form, a model must help with validation, a mailer will need to be created for the emails we'll send and somewhere in there we have a bit of our own logic.

The figure below shows the classes and files that would be involved in this feature. `WidgetEditingService` is probably sticking out to you.

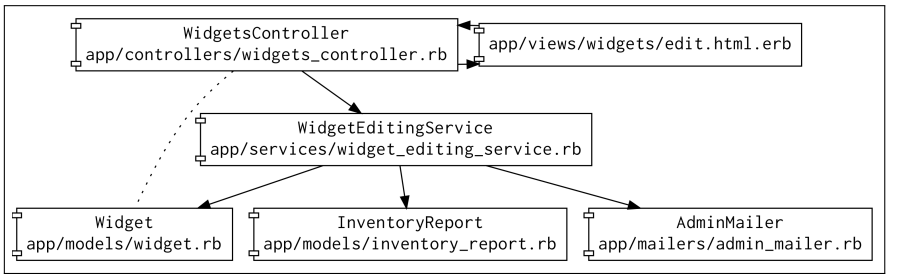

Figure 2.6: Class Design of Feature

Here's what that class might look like:

```
class WidgetEditingService
  def edit_widget(widget, widget_params)
    widget.update(widget_params)

    if widget.valid?
      # create the InventoryReport
      # check the manufacturer to see who to notify
      # trigger the AdminMailer to notify the right person
    end

    widget
  end
end
```

The code in the other classes would be more or less idiomatic Rails code you are used to.

Here's `WidgetsController`:

```
class WidgetsController < ApplicationController
  def edit
    @widget = Widget.find(params[:id])
  end

  def update
    widget  = Widget.find(params[:id])
    @widget = WidgetEditingService.new.edit_widget(
                widget, widget_params
              )
    if @widget.valid?
      redirect_to widgets_path
    else
      render :edit, status: :unprocessable_entity
    end
  end

private
  def widget_params
    params.require(:widget).permit(:name, :status, :type)
  end
end
```

`Widget` will have a few validations:

```
class Widget < ApplicationRecord
  validates :name, presence: true
end
```

`InventoryReport` is almost nothing:

```
class InventoryReport < ApplicationRecord
end
```

`AdminMailer` has methods that just render mail:

```
class AdminMailer < ApplicationMailer
  def edited_widget(widget)
    @widget = widget
  end

  def edited_widget_for_supervisor(widget)
    @widget = widget
  end
end
```

Note that just about everything about editing a widget is in `WidgetEditingService` (which also means that the test of this class will almost totally specify the business process in one place). `widget_params` and the validations in `Widget` *do* constitute a form of business logic, but to co-locate those in `WidgetEditingService` would be giving up a *lot*. There's a huge benefit to using strong parameters and Rails' validations. So we do!

Let's see how this survives a somewhat radical change. Suppose that the logic around choosing who to notify and updating the inventory record are becoming too slow, and we decide to execute that logic in a background job—the user editing the widget doesn't really care about this part anyway.

The figure below shows the minimal change we'd make. The highlighted classes are all that needs to change.

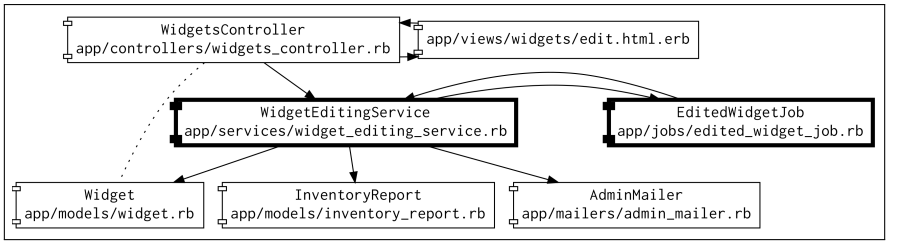

Figure 2.7: Design with a Background Job Added

We might imagine that `WidgetEditingService` is now made up of two methods, one that's called from the controller and now queues a background job and a new, second method that the background job will call that contains the logic we are backgrounding.

```
class WidgetEditingService
  def edit_widget(widget, widget_params)
    widget.update(widget_params)
```

```
    if widget.valid?
      EditedWidgetJob.perform_later(widget.id)
    end

    widget
  end

  def post_widget_edit(widget)
    # create the InventoryReport
    # check the manufacturer to see who to notify
    # trigger the AdminMailer to notify whoever
    # should be notified
  end
end
```

The `EditedWidgetJob` is just a way to run code in the background:

```
class EditedWidgetJob < ApplicationJob
  def perform(widget_id)
    widget = Widget.find(widget_id)
    WidgetEditingService.new.post_widget_edit(widget)
  end
end
```

As you can see, we're putting only the code in the background job that *has* to be there. The background job is given an ID and must trigger logic. And that's all it's doing.

I'm not going to claim this is beautiful code. I'm not going to claim this adheres to object-oriented design principles... whatever those are. I'm also not going to claim this is how DHH would do it.

What I will claim is that this approach allows you to get a *ton* of value out of Rails, while also allowing you to consolidate and organize your business logic however you like. And this will keep that logic from getting intertwined with HTTP requests, email, databases, and anything else that's provided by Rails. And *this* will help greatly with sustainability.

Do note that the "service layer" a) can be called something else, and b) can be designed any way you like yet still reap these benefits. While I would encourage you to write boring procedural code as I have done (and I'll make the case for it in "Business Logic Class Design" on page **??**), you can use any design you like.

Up Next

This will be helpful context about what's to come. Even when isolating business logic in standalone classes, there's still gonna be a fair bit of code elsewhere in the app. A lot of it ends up where we're about to head: the view. And the first view of your app that anyone ever sees is the URL, so we'll begin our deep-dive into Rails with routes.

3

The Database

For most apps, the data in its database is more important than the app itself. If a cosmic entity swooped in and removed your app's source code from all of existence, you could likely recreate it, since you'd still have the underlying data it exists to manage. If that same entity instead removed your *data*... this would be an extinction-level event for your app.

What this thought experiment tells me is that the way data is managed and stored requires a bit more care and rigor than is typically applied to code. This "care and rigor" amounts to spending more time modeling the data and using everything available in your database to keep the data correct, precise, and consistent.

This contradicts Rails' overly simplistic view of the database. By only following Rails' defaults, and designing your database when you write migrations, you will eventually have inconsistent or incorrect data, and likely a fair bit of unnecessary complexity in your code to deal with it.

That said, there are some realities about using a database we have to account for:

- Databases provide much simpler types and validations than our code.
- Large or high-traffic databases can be hard to change.
- Databases are often consumed by more than just your Rails app.

To navigate this, we'll talk about the logical model of the data—the one the users talk about and understand—as distinct from the physical model—what tables, columns, indexes, and constraints are actually in the database. With regard to the physical model, we'll break that down into two distinct steps for development. We'll learn how to decide what database structures you want first, and then how to write a proper Rails migration to create them.

First, let's define logical and physical models.

3.1 Logical and Physical Data Models

When you run `bin/rails g migration` to create a database migration, you are manipulating the *physical* data model: the actual schema in the

database. The *logical* model is the data model as understood by users and other interested parties. For simple domains, these models are often very similar, but it's important to understand the differences.

The logical model is a tool to get alignment between the developers who build the app and the users or other stakeholders who know what problems the app needs to solve. Users won't usually care about physical elements such as indexes, join tables, or reference data lookup tables when discussing how the app should behave.

The logical model is in the language of the users, at the level of abstraction they understand. This is often insufficient for properly managing the data, but you can't make a database without an understanding of the domain.

For example, a user will think that a widget has a status, or a manufacturer has an address. This doesn't mean that the widget *table* must have a status *column* or that the manufacturer *table* has columns for each part of an address. You may not want to (or be able to) model it that way in the database.

See the figure "Example Logical and Physical Models" on the next page for an example of a logical and physical model for a hypothetical widget and manufacturer relationship.

It stands to reason, then, that you should create a logical model to build alignment before you start thinking about the physical model.

3.2 Create a Logical Model to Build Consensus

The logical model is a tool to build consensus with the developers who must write the software and anyone else that understands what the software must do or what problems it must solve. The logical model is where you can identify requirements for the data to be stored without worrying (yet) about how to store it.

I recommend that the developers either lead this process or have final approval, since this model is input into their work. While non-developers can do a good job of drafting logical models, there are often some fine details they miss that a developer will need to know in order to move forward.

I don't want you to think of the logical model as some grandiose document created by a formalized process. Often a single spreadsheet is sufficient. No matter how you do it, I highly recommend writing it down and being explicit. It's usually sufficient to capture:

- The names of all entities or "things" to be managed
- For each attribute of those entities:
  - The name of it
  - What type of data it is

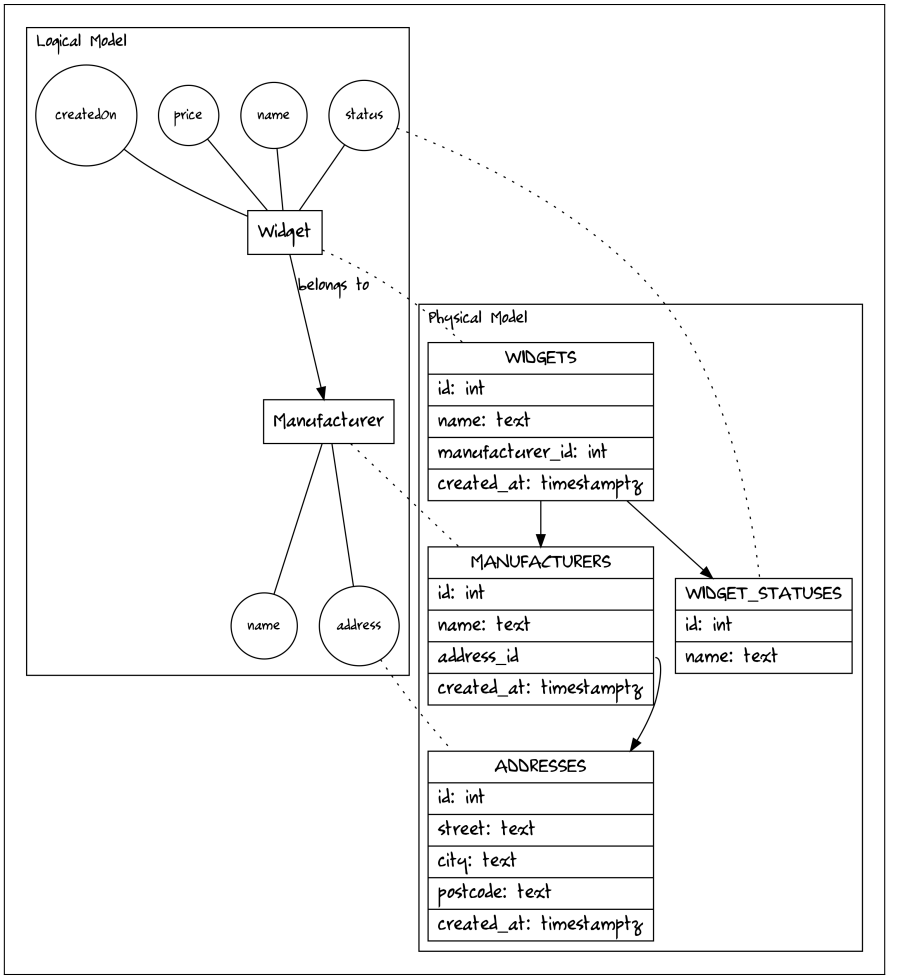

Figure 3.1: Example Logical and Physical Models

  - Is it a required value?
  - What other requirements are there, such as allowed values, uniqueness, etc.

- For each entity, what uniquely identifies it? Can two entities have the exact same values for all attributes and, if so, what does that mean?

For example:

Table 3.1: Example logical model as a spreadsheet

Entity	Attribute	Type	Req?	Other Requirements
Widget	name	String	Y	Unique to manufacturer
Widget	status	String	Y	"Fresh", "Approved", or "Archived"
Widget	price	Money	Y	Not negative, must be less than $10,000
Widget	created	Date	Y	
Manufacturer	name	String	Y	Unique
Manufacturer	address	Address	Y	Street and Zip Code are sufficient

However you draft this logical model, make sure you have a good sense of the allowed values for each attribute. If the user uses attribute types like "Address", define a new entity called "Address" and identify its requirements. For more general types like "String" or "Date", try to get clarity on what values are allowed. There are a lot of strings in the world and probably not all of them are a valid widget status.

As to the uniqueness questions, getting these right can greatly reduce confusion in the actual data. Often there are several sets of values that represent uniqueness. For example, the widget ID we've discussed previously sounds like a unique value. But you also may want widget *names* to be unique. It's fine to have multiple unique identifiers for entities, but it's important to understand all of them.

The less familiar you are with the domain, or the newer it is, the more time you should spend exploring it before you start coding. Mistakes in data modeling are difficult to undo later and can create large carrying costs in navigating the problems created by insufficient modeling.

You don't have to know everything, but even knowing how data *might* be used is useful. While you don't need to handle "someday, maybe" types of requirements, thinking ahead can help.

Knowing how stable certain requirements are can help you properly translate them to the physical model. Stable requirements can be enforced in the database; unstable requirements might need to be enforced in code so they can be more easily changed.

Once you have alignment, you can build the physical model, which you should do in two steps: plan it, then create it.

3.3 Planning the Physical Model to Enforce Correctness

> This section's code is in the folder `14-03/` of the sample code.

Translating the logical model to the physical model requires making several design decisions, especially as the app becomes more complex and needs to manage more types of data.

This should be done in two discrete steps. This section discusses the first, which is to plan exactly how you are going to store the data in the database. The next section discusses how to write a Rails migration to implement this plan.

Whereas the logical model was for building alignment and discovering business rules, the physical model is for accurately managing data that conforms to those rules. This means that correctness, precision, and accuracy are paramount.

The design decisions you'll make amount to how and where you will enforce the correctness of the data. Your database is an incredibly powerful tool to do this, and it's where most of your constraints around correctness should go.

3.3.1 The Database Should Be Designed for Correctness

Rails' view of the database is that it's more or less a dumb store and Rails—via validations and other mechanisms—will keep the data correct. This is unrealistic, even in simple circumstances. Active Record provides a public API to bypass validations, and the reality of most systems is that Things That Aren't Rails will be accessing the database directly.

For example, it's common to connect business and financial reporting systems directly to the app's database. It's often much more economical and flexible to allow business users to query the data however they like than to get developers to build custom views for them. Tools like Looker[1] or Heroku Dataclips[2] provide ways of turning SQL into reports. Common data warehousing techniques usually involve dumping the entire operational database into another system where it can be analyzed.

If these systems have to deal with incorrect or ambiguous data, in the best case, they will be complex and difficult to maintain. More realistically, the reports will simply be wrong. If, on the other hand, these systems can rely on the data in the database being correct and unambiguous, the reports are more valuable and can lead to better decisions.

For simple to moderate requirements, you can use the database to absolutely ensure the data is correct and precise. For complex requirements, you may

[1] https://looker.com
[2] https://devcenter.heroku.com/articles/dataclips

need to use code in addition to the database. Unstable requirements benefit from being implemented in code, because the database will become harder to change as time goes on. Stable or critical requirements, however, benefit greatly from being enforced in the database.

No matter what, we're going to use database-specific features. That requires using a SQL schema instead of a Ruby-based one.

3.3.2 Use a SQL Schema

It's rare to create an app that must connect to many different types of databases. It's also rare to migrate from one database type to another. Thus, we should not be shy about using database-specific features whenever it helps us meet our users' needs. Rails' API for managing the database doesn't provide access to all of these features, however.

This matters because Rails uses a schema file to maintain the test database, as well as to initialize a development database in a fresh environment. We need that schema to match production, so we cannot use `db/schema.rb`, and instead must use SQL.

Fortunately, this is a one-line configuration change in `config/application.rb`

```
# config/application.rb

        # per-controller helpers
        g.helper false
      end
→
→     # We want to be able to use any feature of our database,
→     # and the SQL format makes that possible
→     config.active_record.schema_format = :sql
    end
  end
```

Note that we added a comment as to why we made this change. It's important that all deviations from Rails' defaults are understood by current and future team members. Comments are an easy way to make that happen. Git commit messages are not.

We should also delete `db/schema.rb`, since that will no longer be used. Rails will store the SQL schema in `db/structure.sql`.

```
> rm db/schema.rb
```

I recommend this change for all database types, because it costs nothing and provides a lot of benefit.

For Postgres specifically, we need to make another change, which is to use `TIMESTAMP WITH TIME ZONE` for timestamps.

3.3.3 Use `TIMESTAMP WITH TIME ZONE` For Timestamps

The SQL standard provides for the `TIMESTAMP` fields to store... timestamps. A timestamp is a number of milliseconds since a reference timestamp, usually midnight on January 1, 1970 in UTC.

The `TIMESTAMP` data type does not store a time zone, however. Most databases store timestamps in UTC and provide an automatic translation based on... well, it's complicated.

By default, the computer your database is running on is configured with a system time zone. This can be hard to inspect or control. The connection to the database itself can override this. The code that makes a connection to the database can override this as well. Rails can override this. Your code can override Rails.

This means that your timestamps will be translated using a reference time zone that might not be obvious. And if the wrong reference is used when reading those timestamps out, the reader can interpret the timestamp differently. Even though Rails defaults to using UTC, some other process might be configured differently. This is extremely confusing.

Postgres provides the data type `TIMESTAMPTZ` (also known as `TIMESTAMP WITH TIME ZONE`) that avoids this problem. It stores the reference time zone with the timestamp so it's impossible to misinterpret the value. Postgres expert Dave Wheeler wrote a blog post[3] that can provide you more details.

We can make Rails use this type by default. The class `PostgreSQLAdapter` (which is in the `ActiveRecord::ConnectionAdapters` namespace) has an attribute named `datetime_type` that allows overriding the default SQL type used whenever a migration has a `datetime` in it.

We can set this to `:timestamptz` and all of our migrations will use `TIMESTAMPTZ` instead of `TIMESTAMP`. This can be done anywhere as long it loads when Rails does. Best place to do that is in `config/initializers/postgres.rb`:

```
# config/initializers/postgres.rb

require "active_record/connection_adapters/postgresql_adapter"
```

[3] https://justatheory.com/2012/04/postgres-use-timestamptz/

```
ActiveRecord::ConnectionAdapters::PostgreSQLAdapter.datetime_type =
  :timestamptz
```

Now, when we write code like `t.timestamps` or `t.datetime`, Rails will use `TIMESTAMP WITH TIME ZONE` and all of our timestamps will be stored without ambiguity or implicit dependence on the system time zone. See the sidebar "Changing Timestamps in an Existing Database" on page 40 if you want to use this in an existing app.

Changing Timestamps in an Existing Database

The change we discussed to `datetime_type` sets Rails' behavior when the type `:datetime` is used in a migration. If your app has existing migrations, the interpretation of those migrations would change from their original intent.

While you are unlikely to re-apply migrations in production, you want to make sure that your migrations, `db/structure.sql`, and your databases—production, test, and development—all agree.

One way to ensure this is to change any existing migration using `:datetime` to use `"timestamp"`. Rails always allows you to specify the precise type in a migration. This means you can change the meaning of `:datetime` safely:

1. Modify all migrations using `:datetime` to use `"timestamp"`.
2. Modify all use of `t.timestamps` to create both `created_at` and `updated_at` explicitly as having the type `"timestamp"`.
3. Re-run migrations from scratch. There should be no change in your `db/structure.sql`.
4. Create the initializer described above.
5. Again re-run migrations from scratch. There should again be no change in your `db/structure.sql`.
6. You can now resume the use of `:datetime` and `t.timestamps` in new migrations, safely knowing Rails will use `TIMESTAMPTZ`.

To actually change your database columns from `timestamp` to `timestamptz` is out of scope of this book, but should be done with care—and the consultation of the Postgres documentation.

With this base, we can start planning the physical model.

3.3.4 Planning the Physical Model

A formal way to model a database is called *normalization*, and it's a dense topic full of equations, confusing terms, and mathematical proofs. Instead, I'm going to outline a simpler approach that might lack the precision of theoretical computer science, but is hopefully more approachable.

Here's how to go about it:

1. Create a table for each entity in the logical model.
2. Add columns to associate related models using foreign keys.
3. For each attribute, decide how you will enforce its requirements and create the needed columns, constraints, and associated tables.
4. Create indexes to enforce all uniqueness constraints.
5. Create indexes for any queries you plan to run.

To do this, it's immensely helpful if you understand SQL. In addition to knowing how to model your data, knowing SQL allows you to understand the runtime performance of your app, which will further help you with data modeling. Of all the programming languages you will ever learn, SQL is likely to remain useful for your entire career. Execute Program[4] has a course that will help.

Outside of learning SQL, the hardest part of the planning process is step 3: deciding how to enforce the requirements of each attribute.

You will bring together some or all of the following techniques:

- Choosing the right column type
- Using database constraints
- Creating lookup tables
- Writing code in your app

Let's dive into each one of these.

Choosing the Right Column Type

Each column in the database must have a type, but databases have few types to choose from. Usually there are strings, dates, timestamps, numbers, and booleans. That said, familiarize yourself with the types of *your* database. Unless you are writing code that has to work against *any* SQL database (which is rare), you should not be bound by Rails' least-common-denominator set of types.

The type you choose should allow you to store the exact values you need. It should also make it difficult or impossible to store incorrect values. Here are some tips for each of the common types.

Strings In the olden days, choosing the size of your string mattered. Today, this is not universally true. Consult your database's documentation and use the largest size type you can. For example, in Postgres, you can use a `TEXT` field, since it carries no performance or memory burden over `VARCHAR`. It's important to get this right because changing column types later when you need bigger strings can be difficult.

[4] https://www.executeprogram.com/courses/sql/lessons/basic-tables

Rational Numbers Avoid `FLOAT` if possible. Databases store `FLOAT` values using the IEE 754[5] format, which *does not store precise values*. Either convert the rational to a base unit (for example, store money in cents as an integer), or use the `DECIMAL` type, which *does* store precise values. Note that neither type can store all rational numbers. One-third, for example, cannot be stored in either type. To store precise fractional values might require storing the numerator and denominator separately.

Booleans Use the `boolean` type. Do not store, for example, `"y"` or `"n"` as a string. There's no benefit to doing this and it's confusing. And yes, people do this and I don't understand why.

Dates Remember that a date is not a timestamp. A date is a day of the month in a certain year. There is no time component. The `DATE` datatype can store this, and allow date arithmetic on it. Don't store a timestamp set at midnight on the date in question. Time zones and daylight savings time will wreak havoc upon you, I promise.

Timestamps As opposed to a date, a timestamp is a precise moment in time, usually a number of milliseconds since a reference timestamp. As discussed above, use `TIMESTAMP WITH TIME ZONE` if using Postgres. If you aren't using Postgres, be *very explicit* in setting the reference timezone in all your systems. Do not rely on the operating system to provide this value. Also, *do not* store timestamps as numbers of seconds or milliseconds. The `TIMESTAMP WITH TIME ZONE` and `TIMESTAMP` types are there for a reason.

Enumerated Types Many databases allow you to create custom enumerated types, which are a set of allowed values for a text-based field. If the set of allowed values is stable and unlikely to change, an `ENUM` can be a good choice to enforce correctness. If the values might change, a lookup table might work better (we'll talk about that below).

No matter what other techniques you use, you will always need to choose the appropriate column type. Next, decide how to use database constraints.

Using Database Constraints

All SQL databases provide the ability to prevent `NULL` values. In a Rails migration, this is what `null: false` is doing. This tells the database to prevent `NULL` values from being inserted. Any required value should have this set, and most of your values should be required.

Many databases provide additional constraint mechanisms, usually called *check constraints*. Check constraints are extremely powerful for enforcing correctness. For example, a widget's price must be positive and less than or equal to $10,000. With a check constraint this could be enforced:

[5] https://en.wikipedia.org/wiki/IEEE_754

```
ALTER TABLE
  widgets
ADD CONSTRAINT
  price_positive_and_not_too_big
CHECK (
  price_cents >  0 AND
  price_cents <= 1000000
)
```

If you try to insert a widget with a price of -$100 or $300,000, the database will refuse. Thus, you can be absolutely sure the price is valid. Check constraints can do all sorts of things. If you want all widget names to be lowercase, you can do that, too:

```
CHECK (
  lower(name) = name
)
```

Modifying these constraints becomes more difficult as the database gets larger, because these sorts of changes can create locks on the table that prevent access or modification or both. This can create downtime for your app. There are strategies to deal with this that are beyond the scope of this book, but the strong migrations gem[6] is a great place to start with understanding them. Note, however, that it's entirely likely that you will *never* reach the size of database where this would be a problem.

Here are the guidelines I find most useful:

- Any stable requirement should be implemented as a check constraint.
- Any critical requirement should be implemented as a check constraint.
- Unstable requirements on tables expected to grow might be better implemented in code, so you can change them frequently, but it still might be better to use a check constraint and wait for the table to actually get large enough to be a problem.

The next technique for enforcing correctness is the use of lookup tables.

[6] https://github.com/ankane/strong_migrations

Using Lookup Tables

When a column's value should be one value from a static list of possible values, an `ENUM` can work as we discussed above. If the possible values are likely to change, or if users are modifying those values, *or* if you need additional metadata to go along with the values, an `ENUM` won't work. In these cases, you need a lookup table.

In the data model above on page 35, you can see an example of this for the widget's status. Suppose we had three widgets in the database, two of which have the status "Fresh" and the other "Approved". Here's how that would look in the database using a lookup table:

Table 3.2: Example `widgets` table referencing a lookup table

id	name	widget_status_id
10	Stembolt	1
11	Thrombic Modulator	1
12	Tachyon Generator	2

Table 3.3: Example `widget_statuses` lookup table

id	name
1	Fresh
2	Approved
3	Archived

Note a key difference between the physical and logical model. The logical model simply states that a widget has a status attribute. To enforce correctness and deal with a potentially unstable list of possible values, we are modeling it with a new table. In our code, a widget will `belong_to` a status (which will `has_many` widgets).

When using lookup tables, you must create a *foreign key constraint*. This tells the database that the value for `widget_status_id` *must* match an `id` in the referenced `widget_statuses` table. This prevents widgets from having invalid or unknown statuses, since `widget_statuses` contains all known valid statuses.

A lookup table also allows modeling metadata on the referenced value. For example, if only "Approved" widgets can be sold, we might model that with a `boolean` column on the `widget_statuses` table:

Table 3.4: Example `widget_statuses` lookup table with metadata

id	name	allows_sale
1	Fresh	false
2	Approved	true
3	Archived	false

The last tool available to enforce correctness is your app.

Enforcing Correctness in App Code

Some requirements are too difficult to enforce at the database layer, either because of necessary complexity or because of a lack of stability. In these cases, your app can enforce correctness by refusing to write data that violates the requirements.

Rails validations are quite powerful at doing this, and this is the mechanism you should use if you must validate correctness in code. Just be aware that Active Record's public API allows circumventing the validations. Anything your database can possibly store, you can put into it using Active Record, no matter what validations you have created.

That said, some requirements are so complex that using validations becomes quite difficult and you'll need to write a bunch of code to prevent bad data from getting written.

For example, if only supervisors can change a widget's status to "Approved" for manufacturers created before July 10, 1998, except for the manufacturer "Cyberdyne Systems", this is going to be a convoluted and hard-to-understand validation. It would be simpler as code (and relatively straightforward to implement if you've followed the previous guidance and avoided putting business logic in your Active Records).

Once you have decided how you are going to model everything, it's time to make your migrations.

3.4 Creating Correct Migrations

> This section's code is in the folder `14-04/` of the sample code.

Writing migrations is how we programmatically modify the database to conform to the physical schema we want to use. Because Rails' API for doing this is not SQL, it's important that we take some time to make sure the migrations we write result in the schema we need. Rails' API is powerful and will save us time and make the work easier, but it lacks a few useful defaults.

In the previous chapter, we created models so we could talk about some model basics. Rather than edit those models and the schema it created, let's start over (you can't do this in real life, but it'll make this chapter simpler if we do).

If we delete the migrations and fixtures created by `bin/rails g model` and re-run `bin/setup`, we should be good to go.

```
> rm db/migrate/* test/fixtures/*.* && bin/setup
«lots of output»
```

The figure "Example Logical and Physical Models" on page 35 outlines what we're going to do, but to re-iterate:

- A Widget has a name, price, status, and manufacturer, all of which are required.
- A Manufacturer has a name and an address, both of which are required.
- An address is a street and a zip code (both required).
- Widget names must be unique within a manufacturer.
- Manufacturer names must be unique globally.
- We'll use lookup tables for addresses and widget statuses.
- We'll use a database constraint to enforce a price's lower-bound, but code for the upper-bound.

It's important that changes that logically relate to each other go in a single migration file. Some databases, including Postgres, run migrations in a transaction, which allows us to achieve an all-or-nothing result. Either our entire change is applied successfully, or none of it is.

While we still want to end up with one migration, I find it easier to built it iteratively. Write some of the migration, apply it and check it, then rollback and continue until everything is correct.

The figure "Authoring Migrations" on the next page outlines this basic process:

1. Create your migration file.
2. Add some code to it.
3. Apply the migrations and check the database to see if it had the desired effect.
4. If anything is wrong, or you aren't yet done, roll back the changes.
5. Repeat until you have correctly modeled the physical changes.

This allows you to take each change step-by-step, but still end up with only one migration file that makes the cohesive change you're making. In our case, we want a single migration that creates the needed tables.

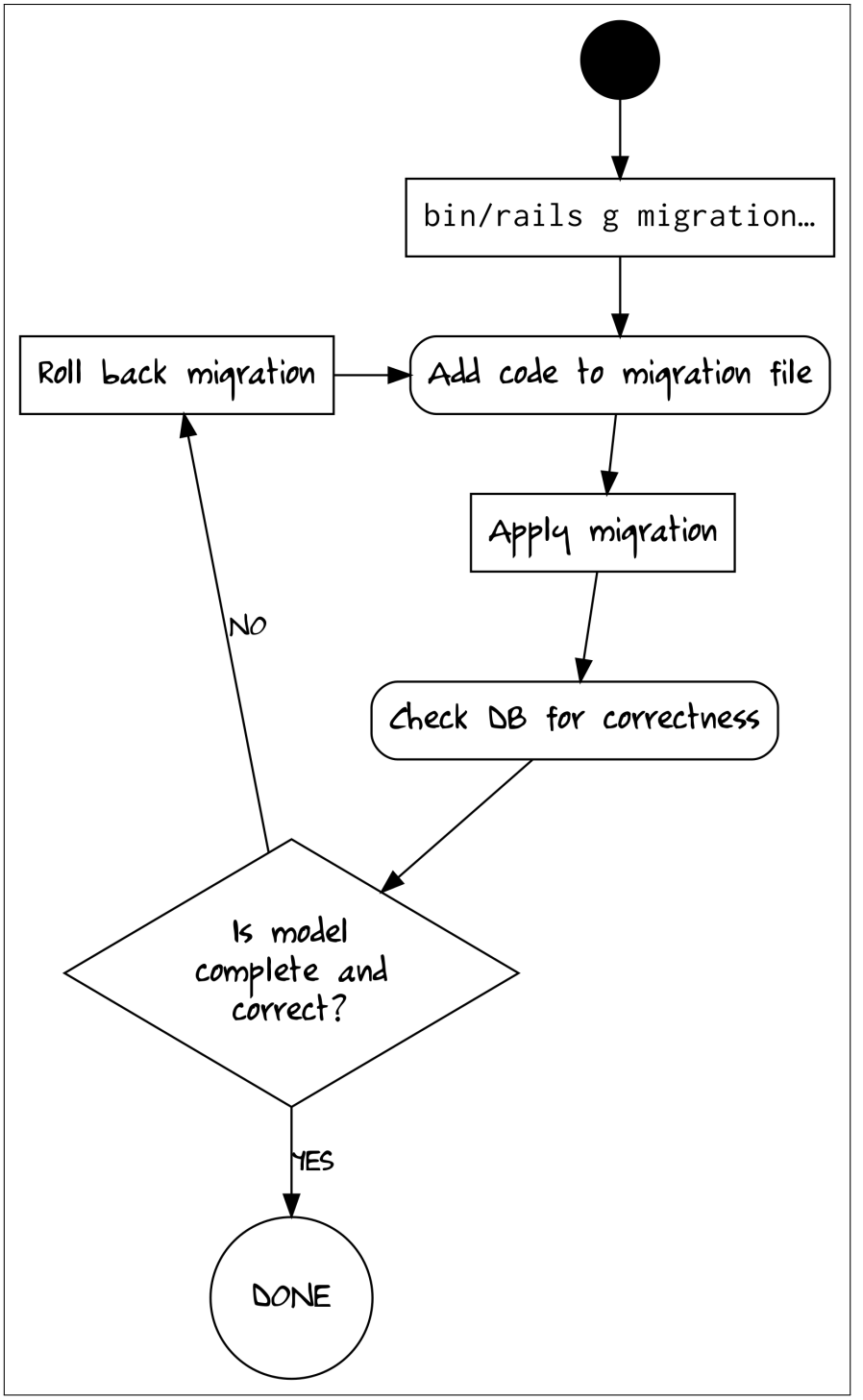

Figure 3.2: Authoring Migrations

3.4.1 Creating the Migration File and Helper Scripts

Before we create the migration file, we need three scripts to help this process. I find that `bin/rails db:migrate` and `bin/rails db:rollback` don't consistently modify both the development and test schema. This can result in a test schema that is not the same as what's described in the migration file, which can cause some confusing test behavior. Rather than document this problem, let's make two scripts to handle applying migrations and rolling them back.

Here's the script to migrate all databases (note again the duplicated checks for `-h` and friends):

```
# bin/db-migrate

#!/bin/sh

set -e

if [ "${1}" = -h ]     || \
   [ "${1}" = --help ] || \
   [ "${1}" = help ]; then
  echo "Usage: ${0}"
  echo
  echo "Applies outstanding migrations to dev and test databases"
  exit
else
  if [ ! -z "${1}" ]; then
    echo "Unknown argument: '${1}'"
    exit 1
  fi
fi

echo "[ bin/db-migrate ] migrating development schema"
bin/rails db:migrate

echo "[ bin/db-migrate ] migrating test schema"
bin/rails db:migrate RAILS_ENV=test
```

Here's the one we'll use to roll back all databases:

```
# bin/db-rollback
```

```
#!/bin/sh

set -e

if [ "${1}" = -h ]     || \
   [ "${1}" = --help ] || \
   [ "${1}" = help ]; then
  echo "Usage: ${0}"
  echo
  echo "Rolls back the current migration from dev and test databases"
  exit
else
  if [ ! -z "${1}" ]; then
    echo "Unknown argument: '${1}'"
    exit 1
  fi
fi

echo "[ bin/db-rollback ] rolling back development schema"
bin/rails db:rollback

echo "[ bin/db-rollback ] rolling back test schema"
bin/rails db:rollback RAILS_ENV=test
```

Let's also make a script called `bin/psql` that connects to our development database. I realize that `bin/rails dbconsole` does this, but a) it requires us to type a password each time, and b) it's incredibly slow to start up because it must load Rails first, only to delegate to the `psql` command-line client.

```
# bin/psql

#!/bin/sh

set -e

if [ "${1}" = -h ]     || \
   [ "${1}" = --help ] || \
   [ "${1}" = help ]; then
  echo "Usage: ${0}"
  echo
  echo "Uses psql to connect to dev database directly"
  exit
else
  if [ ! -z "${1}" ]; then
```

```
    echo "Unknown argument: '${1}'"
    exit 1
  fi
fi

echo "[ bin/psql ] Connecting to widgets_development"
PGPASSWORD=postgres psql -U postgres \
                         -h db \
                         -p 5432 \
                         widgets_development
```

Note that because we have consolidated all dev-environment configuration, we can safely rely on the database connection information to be consistent for all developers, and thus hard-code it into this script.

We'll need to make them executable:

```
> chmod +x bin/db-migrate bin/db-rollback bin/psql
```

It's also a good idea to add these to `bin/setup help`. I'll leave that as an exercise for the reader.

Now, let's create our migration file:

```
> bin/rails g migration make_widget_and_manufacturers
      invoke  active_record
      create    db/migrate/20250107152916_make_widget_and_ma. . .
```

For the sake of repeatability when writing this book, I'm going to rename the migration file to a name that's not based on the current date and time. You don't need to do this.

```
> mv db/migrate/*make_widget_and_manufacturers.rb  \
  db/migrate/20210101000000_make_widget_and_manufacturers.rb
```

With that set up, we can now iteratively put code in this file to generate the correct schema we want.

3.4.2 Iteratively Writing Migration Code to Create the Correct Schema

We'll need to work a bit backward. We can't create `widgets` first, because it must reference `widget_statuses` and `manufacturers`. `manufacturers` must reference `addresses`. So, we'll start with `widget_statuses`.

By default, Rails creates nullable fields. We don't want that. Fields with required values should not allow null. We'll use `null: false` for these fields (even for nullable fields I like to use `null: true` to make it clear that I've thought through the nullability).

I also like to document tables and columns using `comment:`. This puts the comments in the database itself to be viewed later. Even for something that seems obvious, I will write a comment because I've learned that things are never as obvious as they might seem.

```
# db/migrate/20210101000000_make_widget_and_manufacturers.rb

  class MakeWidgetAndManufacturers < ActiveRecord::Migration[8. . .
    def change
→     create_table :widget_statuses,
→       comment: "List of definitive widget statuses" do |t|
→
→       t.text :name, null: false,
→         comment: "The name of the status"
→       t.timestamps null: false
→     end
→
→     add_index :widget_statuses, :name, unique: true,
→       comment: "No two widget statuses should have the same name"
    end
  end
```

Note that I've created a unique index on the `:name` field. Although database indexes are mostly for allowing fast querying of certain fields, they are also the mechanism by which databases enforce uniqueness. Thus, to prevent having more than one status with the same name, we create this index, specifying `index: { unique: true }`.

This will create a case-sensitive constraint, meaning the statuses `"Fresh"` and `"fresh"` are both allowed in the table at the same time. Currently, the developers control the contents of this table, so a unique index is fine—we won't create a duplicate status in a different letter case. If the contents of this field were user-editable, I might create a case-insensitive constraint instead. Sean Huber wrote a short blog post[7] about how you could do this if you are interested.

Next, let's create the `addresses` table. Our user's documentation said "street and zip is fine", so we'll create the table with just those two fields for now.

[7] http://shuber.io/case-insensitive-unique-constraints-in-postgres/

```
# db/migrate/20210101000000_make_widget_and_manufacturers.rb


      add_index :widget_statuses, :name, unique: true,
        comment: "No two widget statuses should have the same n. . .
→     create_table :addresses,
→       comment: "Addresses for manufacturers" do |t|
→
→       t.text :street, null: false,
→          comment: "Street part of the address"
→       t.text :zip, null: false,
→         comment: "Postal or zip code of this address"
→
→       t.timestamps null: false
→     end
→
    end
  end
```

Again, liberal use of `comment:` will help future team members. At this point, I like to run the migrations to make sure everything's working before proceeding.

```
> bin/db-migrate
[ bin/db-migrate ] migrating development schema
== 20210101000000 MakeWidgetAndManufacturers: migrating ====. . .
-- create_table(:widget_statuses, {comment: "List of definit. . .
   -> 0.0069s
-- add_index(:widget_statuses, :name, {unique: true, comment. . .
   -> 0.0019s
-- create_table(:addresses, {comment: "Addresses for manufac. . .
   -> 0.0032s
== 20210101000000 MakeWidgetAndManufacturers: migrated (0.01. . .

[ bin/db-migrate ] migrating test schema
== 20210101000000 MakeWidgetAndManufacturers: migrating ====. . .
-- create_table(:widget_statuses, {comment: "List of definit. . .
   -> 0.0043s
-- add_index(:widget_statuses, :name, {unique: true, comment. . .
   -> 0.0015s
-- create_table(:addresses, {comment: "Addresses for manufac. . .
   -> 0.0040s
== 20210101000000 MakeWidgetAndManufacturers: migrated (0.00. . .
```

I also like to connect to the database and describe the tables to see if it looks correct. It may seem silly, but looking at the same information in a different way can often uncover mistakes.

With Postgres, you can use the `bin/psql` script we made and type `\d+ widget_statuses` or `\d+ addresses` to display the schema. If anything looks wrong—including a spelling error in a comment—use `bin/db-rollback`, fix it, and move on.

Of course, we aren't done yet, so we'll `bin/db-rollback` anyway.

```
> bin/db-rollback
[ bin/db-rollback ] rolling back development schema
== 20210101000000 MakeWidgetAndManufacturers: reverting ====. . .
-- drop_table(:addresses, {comment: "Addresses for manufactu. . .
   -> 0.0017s
-- remove_index(:widget_statuses, :name, {unique: true, comm. . .
   -> 0.0015s
-- drop_table(:widget_statuses, {comment: "List of definitiv. . .
   -> 0.0010s
== 20210101000000 MakeWidgetAndManufacturers: reverted (0.00. . .

[ bin/db-rollback ] rolling back test schema
== 20210101000000 MakeWidgetAndManufacturers: reverting ====. . .
-- drop_table(:addresses, {comment: "Addresses for manufactu. . .
   -> 0.0011s
-- remove_index(:widget_statuses, :name, {unique: true, comm. . .
   -> 0.0016s
-- drop_table(:widget_statuses, {comment: "List of definitiv. . .
   -> 0.0004s
== 20210101000000 MakeWidgetAndManufacturers: reverted (0.00. . .
```

Because `widgets` must refer to `manufacturers`, we need to make `manufacturers` next. We'll use `references` to create the foreign key from `manufacturers` to `addresses`. Rails' default is to skip creating a foreign key constraint. This is not a good default, because there's no benefit to skipping foreign key constraints.

We'll use `foreign_key: true` to make sure the constraint gets created. We cannot have manufacturers referencing non-existent addresses. We'll also add an index to the reference because we'll definitely be navigating these foreign keys and an index will ensure that navigation performs well.

```
# db/migrate/20210101000000_make_widget_and_manufacturers.rb

        t.timestamps null: false
```

```
      end

→     create_table :manufacturers,
→       comment: "Makers of the widgets we sell" do |t|
→
→      t.text :name, null: false,
→        comment: "Name of this manufacturer"
→
→      t.references :address, null: false,
→         index: true,
→         foreign_key: true,
→         comment: "The address of this manufacturer"
→
→      t.timestamps null: false
→     end
→
→     add_index :manufacturers, :name, unique: true
→
    end
  end
```

And now, finally, we can make the `widgets` table:

```
# db/migrate/20210101000000_make_widget_and_manufacturers.rb


      add_index :manufacturers, :name, unique: true

→     create_table :widgets,
→       comment: "The stuff we sell" do |t|
→
→      t.text :name, null: false,
→        comment: "Name of this widget"
→
→      t.integer :price_cents, null: false,
→        comment: "Price of this widget in cents"
→
→      t.references :widget_status, null: false,
→         index: true,
→         foreign_key: true,
→         comment: "The current status of this widget"
→
→      t.references :manufacturer, null: false,
→         index: true,
```

```
→         foreign_key: true,
→         comment: "The maker of this widget"
→
→       t.timestamps null: false
→     end
→
    end
  end
```

We have only two steps left. We must enforce the uniqueness of widget names amongst manufacturers, and enforce the widget's price allowed values. We'll tackle the uniqueness requirement next.

To enforce the widget name/manufacturer uniqueness requirement, we can create our own index on both fields using `add_index`:

```
# db/migrate/20210101000000_make_widget_and_manufacturers.rb

       t.timestamps null: false
     end

→    add_index :widgets, [ :name, :manufacturer_id ],
→      unique: true,
→      comment: "No manufacturer can have two widgets with " +
→               "the same name"
→
    end
  end
```

This allows many widgets to have the same name, as long as they don't also have the same manufacturer.

To create the constraint on price, we can use the `add_check_constraint` method. Prior to Rails 6.1, you needed to use `reversible` and `execute` to put raw SQL in your migration. No longer!

We'll add this to the migration file:

```
# db/migrate/20210101000000_make_widget_and_manufacturers.rb

        comment: "No manufacturer can have two widgets with " +. . .
                 "the same name"
```

```
→     add_check_constraint(
→       :widgets,
→       "price_cents > 0",
→       name: "price_must_be_positive"
→     )
→
    end
  end
```

If you don't know SQL or it's still new to you, this syntax for what goes into the second argument of `add_check_constraint` can seem daunting and hard to derive. Your database's documentation is a great place to start and you *can* piece it together from that. A little bit of trial-and-error also helps, and since you can easily apply and rollback your migration, a combination of reading docs and trying things out will allow you to arrive at the right syntax. That's how I did it!

Also note that we used the optional `:name` parameter to give the constraint a name. Like adding comments to our tables and columns, giving constraints a descriptive name can be useful. If the constraint is violated, the name will appear in the error message and it can be helpful to use that to start figuring out what might have gone wrong.

Lastly, you'll notice that we didn't need to use any raw SQL, but we are still using a SQL-based schema. A SQL-based schema is always a better option from the start, because they you don't have to remember to change it later if you *do* need to use SQL in your migrations.

Let's apply it:

```
> bin/db-migrate
«lots of output»
```

We aren't *quite* done, because we have not modeled the upper-limit on price. We planned to do that in code, so we need to make sure all of our model classes are created and correct, following the guidelines we learned about in "Active Record is for Database Access" on page **??**.

First up is `WidgetStatus`. Since there is a to-many relationship with widgets, we'll use `has_many :widgets`. Note that this file will not already exist and you must create it.

```
# app/models/widget_status.rb

class WidgetStatus < ApplicationRecord
```

```
  has_many :widgets
end
```

Next is `Address`. It has a to-many relationship with manufacturers, since multiple manufacturers can exist at the same address. Also note that this file won't already exist.

```
# app/models/address.rb

class Address < ApplicationRecord
  has_many :manufacturers
end
```

We'll add the other end of the relationship to `Manufacturer`:

```
# app/models/manufacturer.rb

  class Manufacturer < ApplicationRecord
    has_many :widgets
→   belongs_to :address
  end
```

Finally we'll model `Widget`. Because we did not model the price's upper-end in the database, we should add it to the code now as a validation. Even though we have no use-case that would trigger this validation, since it's part of the logical data model that we couldn't model in the database, we have to put it here.

Note that we *aren't* putting any other validations in these models. The database will enforce correctness and prevent bad data from being written. We only need redundant checks if there's a specific reason. We'll discuss this more in "Validations Don't Provide Data Integrity" on page **??**.

```
# app/models/widget.rb

        last_two: id_as_string[-2..-1]
      }
    end
→   belongs_to :widget_status
```

```
→   validates :price_cents,
→     numericality: { less_than_or_equal_to: 10_000_00 }
  end
```

If you aren't used to database constraints, it might feel like we've put business logic in our database. In a way, we have, and we really should consider testing some of it. The check constraint, in particular, seems hard to be confident in without a test.

Let's see what a test looks like for our database constraints.

3.5 Writing Tests for Database Constraints

> This section's code is in the folder `14-05/` of the sample code.

Like all tests, tests for the correctness of the data model have a carrying cost. I don't see a lot of value in testing `null: false`, or `unique: true`, because these tend to be easy to get right. Check constraints are more like real code and thus easier to mess up. I usually write tests for them.

Let's write a test for the constraint around the widget's price. We'll need two tests: one that successfully sets the widget's price to a correct value, and another that fails in an attempt to set it to a negative value.

Because this is testing the database and not the code in `app/models`, our tests will use `update_column`, which skips validations and callbacks, writing directly to the database. If we used `update!` instead, and we later added validations to the `Widget` class, our test would fail to write the database at all. Using `update_column` ensures we are testing the database itself.

To do that, we'll set up a valid widget in the `setup` method, which requires a widget status and a manufacturer (which requires an address).

```
# test/models/widget_test.rb

require "test_helper"

class WidgetTest < ActiveSupport::TestCase
  setup do
    widget_status = WidgetStatus.create!(name: "fresh")
    manufacturer = Manufacturer.create!(
      name: "Cyberdyne Systems",
      address: Address.create!(
        street: "742 Evergreen Terrace",
        zip: "90210"
```

```
      )
    )
    @widget = Widget.create!(
        name: "Stembolt",
        manufacturer: manufacturer,
        widget_status: widget_status,
        price_cents: 10_00
      )
  end
  test "valid prices do not trigger the DB constraint" do
    assert_nothing_raised do
      @widget.update_column(
        :price_cents, 45_00
      )
    end
  end
  test "negative prices do trigger the DB constraint" do
    ex = assert_raises do
      @widget.update_column(
        :price_cents, -45_00
      )
    end
    assert_match(/price_must_be_positive/i,ex.message)
  end
end
```

Note the way we are checking that we violated the constraint. We check that the message in the assertion references the constraint name we used in the migration: `price_must_be_positive`. This means our test should hopefully *only* pass if we violated that constraint, but fail if we get some other exception.

Now, let's run the test.

```
> bin/rails test test/models/widget_test.rb
Running 2 tests in a single process (parallelization thresho. . .
Run options: --seed 46939

# Running:

..

Finished in 0.139710s, 14.3154 runs/s, 28.6308 assertions/s.
2 runs, 4 assertions, 0 failures, 0 errors, 0 skips
```

This should pass. While we could write a test for the validation, I find those sorts of tests less valuable since the code is straightforward with no real logic.

Up Next

Data modeling is not easy and it can take a lot of experience to get comfortable with it. Hopefully, I've stressed how important it is to create your database in a way that favors correctness and precision at the database layer, as well as some helpful techniques to get there.

In the chapter after next, we'll finish talking about models, but to do that, we need to revisit business logic. While our database schema implements some of our business rules, most of the logic that makes our app special will be in code, so let's talk about that next.