Sustainable Web Development

WITH RUBY ON RAILS

by **David Bryant Copeland**

Practical Tips for Building Web Applications that Last

Sustainable Web Development with Ruby on Rails

Practical Tips for Building Web Applications that Last

David Bryant Copeland

For more information, visit https://sustainable-rails.com

Contents

Contents

Beta Information and Changelog

This is a beta release of this book. That means that there are certainly mistakes and errors, but more importantly, the book's not yet completed!

By reading it now, you can help me make the book better by providing feedback.

The best way to do that is to use the form at `bit.ly/sus-rails-feedback`[1].

As I update the book, you'll receive an email to download an updated copy. This chapter will contain a changelog of what was updated since the previous version.

Aug 2, 2020

- Final chapter: "Sustainability As You Grow" on page **??**

There may be another beta release, but next steps are to do some editing, technical review, typesetting and then my book-that-I-thought-would-take-2-months will be done!

July 19, 2020

- New chapter: "API Endpoints" on page **??** about making JSON APIs inside your Rails app.

July 11, 2020

- New chapter: "Operations" on page **??** that discusses concerns related to operating your Rails app in production.

June 22, 2020

- New chapter: "Other Boundary Classes" on page **??**, which covers mailers and rake tasks.

[1] http://bit.ly/sus-rails-feedback

- Other New chapter: “Sustainable Process and Workflows” on page **??**, which covers a grab bag of processes and techniques for sustainable development outside the code in your Rails app.
- Updated end-to-end example to run all tests and fix them. If you were running `bin/ci` after that and saw test failures. . . sorry.
- Changed the use of < for dates in the chapter “Jobs” to use `after?` which then revealed a bug in the code around date parsing which is now fixed. Seriously, don’t use < for date comparisons.
- Corrected mistake around helpers - only `ApplicationHelper` is included by default.

June 12, 2020

- New chapter: “Authentication and Authorization” on page **??**.
- New chapter: “Jobs” on page **??**.
- Added Redis to the Docker setup on GitHub[2].
- Changes to test support code. I realized it should be in a module in `test/support` and not just dumped into the base test case classes.
- Various typos.

May 25th, 2020

- New chapter: “Controllers” on page **??**.
- Revised the End-to-End Example chapter on page **??**. It now shows the entire set of tests and implementation rather than hand-waiving. I’m sorry it made the chapter so long, but I think it was worth it. I also changed the implementation a bit after sitting with it for a week.
- New cover design! Let me know what you think!

May 18th, 2020

- New chapter showing and end-to-end example on page **??**. This brings together a lot of the techniques discussed in the book up to this point.
- A few more typos fixed.

May 11th, 2020

- New chapter: “Models, Part 2” on page **??**.
- Changed the guidance on Turbolinks. Turns out that it *does* show loading progress and thus won’t make your app look broken. That said, the default of 500ms before showing progress is too short, so the section on Turbolinks on page **??** has been updated.

[2] https://github.com/davetron5000/sustainable-rails-docker

- Complicatified the invocation of `yarn audit` to handle information vulnerabilities we don't care about.
- Attempted to clarify the naming conventions around presenters.
- *Tons* of type and wording fixes courtesy Sean Miller. Thank you!!!!

May 4th, 2020

- New Chapter: "Business Logic Code Should Be Thought of as a Seam" on page **??**.
- Reversed the changelog so the newest stuff is up first.
- Note that the Imperial Senate will no longer be of any concern to us. I have just received word that the Emperor has dissolved the council permanently. The last remnants of the Old Republic have been swept away. What's that? Oh, don't worry. Fear will keep the local systems in line.

April 27th, 2020

- New Chapter: "The Database" on page **??**.
- Clarified use of `:dependent` in modeling relationships.

April 15, 2020

- New Chapter: "Models, Part 1" on page **??**
- Fixed sidebar formatting for Epub and Kindle
- Fixed cross-reference links

April 8, 2020

- Removed syntax highlighting from Epub so that the code shows on on Apple iBooks.
- Update to Ruby 2.7.1

April 7, 2020: Initial Beta Release

- Introduction and chapters on the Rails view.

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", because 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's 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. The ability to write software is one of the most in-demand skills, 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 has referential integrity.

The features of the software are merely one input into what software gets built. They are a significant one, to be sure, but 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 teams hiring plans, that work can be easily justified (see the sidebar "Understanding Growth At Stitch Fix" on page 8 for a specific example of this).

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.

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 just 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?

1.4 Assumptions

This book is pretty 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 you find yourself in writing software 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.

So the assumption here is that that has 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 things 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 page 11 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 in the way developers think. 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.

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.

Roll quality into 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 that 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.

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 decide the cost/benefit analysis. A comparison we'll make 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.

It is carrying costs that most greatly affect sustainability. 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 things.

So 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 ten years I have been a product engineer, working for companies building one or more products designed to last. I was a rank and file engineer at times, a manager on occasion, and most recently, an architect (meaning I was responsible for technical strategy, but I assure you I wrote a *lot* of code).

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 over 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.

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.

So 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. Hopefully, I can at least clarify the trade-offs and how to think about them, so if you aren't in the same exact situation as me, you can still get value from my experience.

1.7 Up Next

Hopefully this chapter has 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.

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.

This chapter is going to walk you through the way I think about it. 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 can have a more serious impact than bugs in isolated code.

From there, we'll then be able to speak as objectively as possible about the ramifications of putting business logic in Active Records versus putting it somewhere else.

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

2.1 Business Logic is What 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 figure "System Diagram to Understand Fan-in" on page 18. 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.

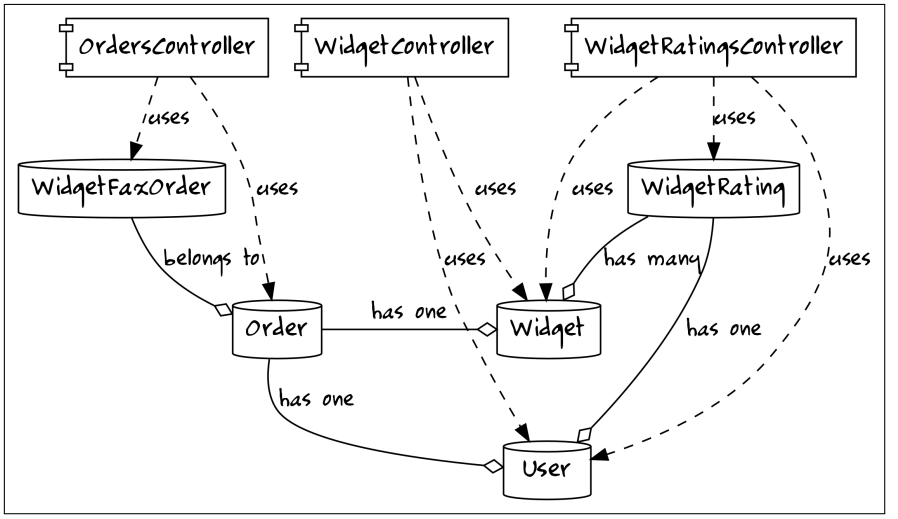

Figure 2.1: System Diagram to Understand Fan-in

Consider a bug in `WidgetFaxOrder`. The figure "Bug Effects of a Low Fan-in Module" on page 19 outlines the affected 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`.

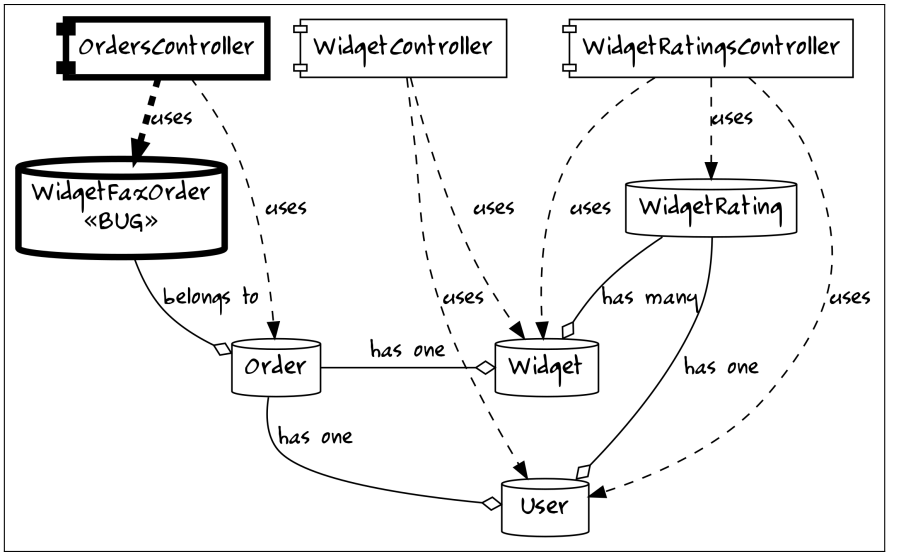

Figure 2.2: Bug Effects of a Low Fan-in Module

Now consider if instead `Widget` has a bug. The figure "Bug Effects of a High Fan-in Module" on page 20 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, and 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.

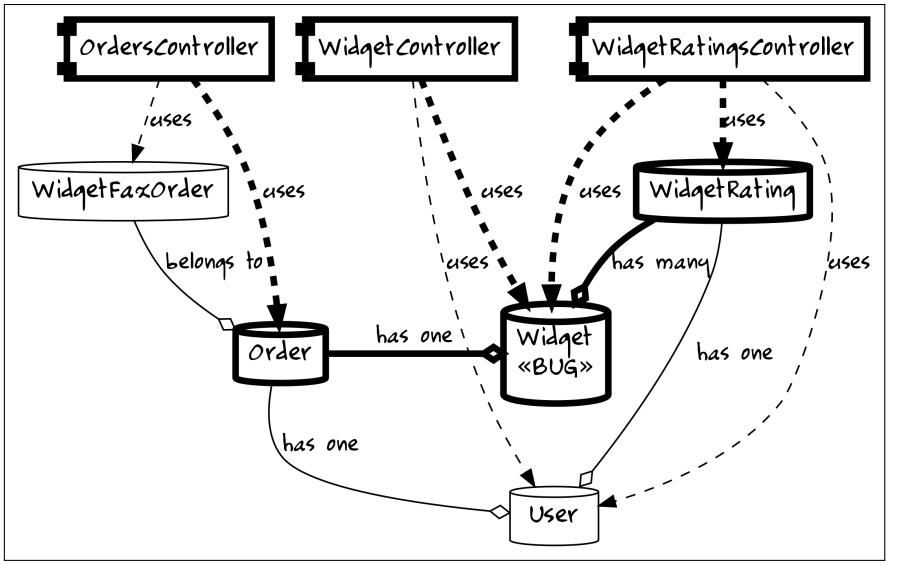

Figure 2.3: Bug Effects of a High Fan-in Module

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 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 page 21. 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.

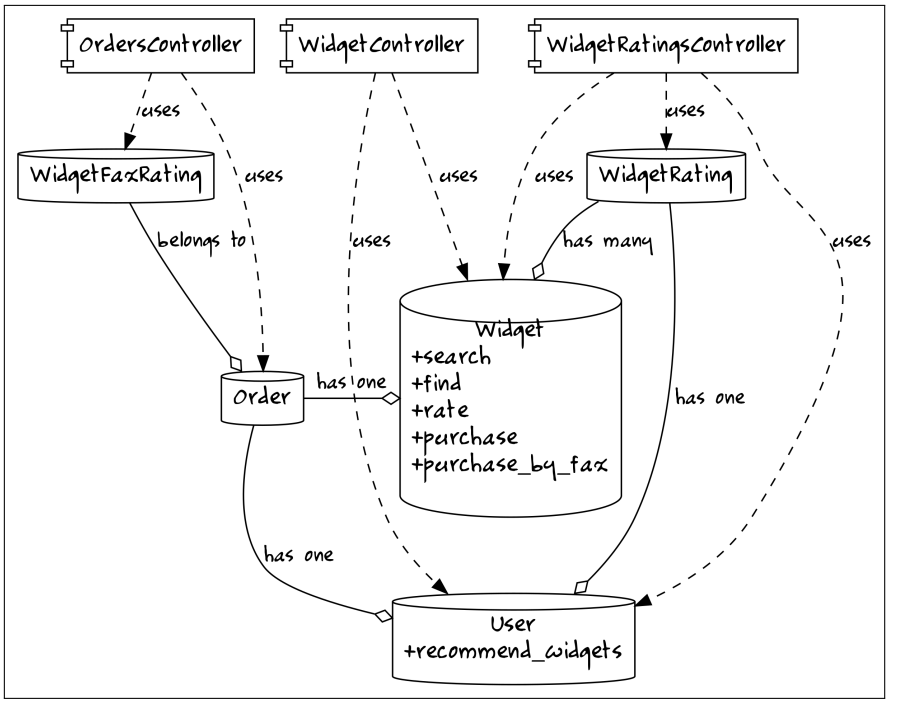

Figure 2.4: System with Logic on Active Records

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 "System with Business Logic Separated" on page 21.

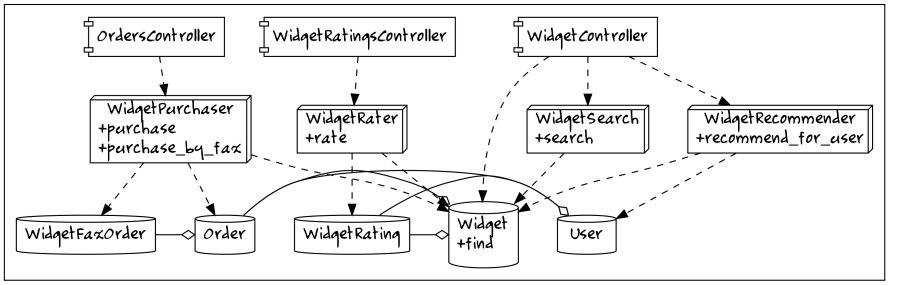

Figure 2.5: System with Business Logic Separated

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.

Let's see an example.

2.4 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 "Class Design of Feature" on page 23 shows the classes and files that would be involved in this feature. `WidgetEditingService` is probably sticking out to you.

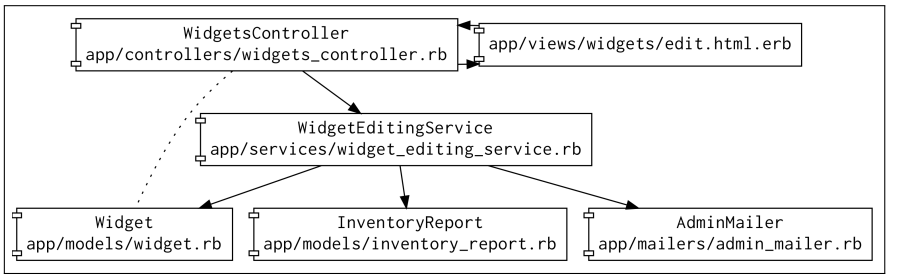

Figure 2.6: Class Design of Feature

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 whoever should be notified
    end

    widget
  end
end
```

The code in the other classes would be relatively vanilla Rails stuff. `WidgetsController` looks how you'd expect:

```
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
    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)
    @wiget = 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 "Design with a Background Job Added" on page 25, shows the minimal change we'd make. The highlighted classes are all that needs to change.

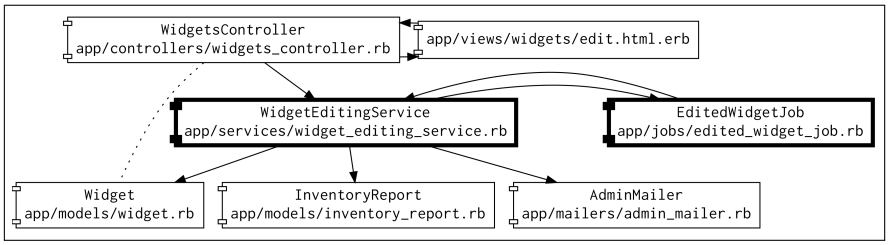

Figure 2.7: Design with a Background Job Added

We might imagine that `WidgetEditingService` is now made up of two methods, one that's called from the controller and now queues a background 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.

2.5 Up Next

I hope I've given you some 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.