Endorsements

In the brave new world of AI-generated codebases, this work is an essential companion book for every aspiring programmer. In his characteristic friendly and accessible writing style, Artie quickly introduces you to various ideas, techniques, and principles that are already the mainstay of elite, ultra-productive software engineers. With a great balance between pragmatic advice and theory, the book quickly equips you with the essential street-fighting moves needed to tackle even the most nausea-inducing code complexity horrors. If you want to go from a "coding monkey" to "system architect," then this book is an absolute must-read!

—Jesse Lu, PhD
Engineering Lead, SPINS Photonics Inc

This book is an excellent read, covering in detail often overlooked software development quality practices. It's an invaluable resource if you want to equip yourself with the tools that will help you write sustainable and maintainable code, while fostering a better programming culture.

—Laurynas Kavaliauskas
Senior Principal Architect, Zscaler

This book is filled with actionable guidance and helpful examples. The reframing of certain known ideas and concepts is also timely and instructive. I highly recommend this book to any software professional who wants to level up their knowledge and skills.

—Jeff Doolittle
Senior Software Architect, Trimble

Code Health Guardian offers solid theoretical advice aimed at experienced engineers, all while maintaining a practical focus. Reading this book can help establish or solidify engineering principles that are crucial for maintaining the health of your codebase. The book starts by highlighting that it is meant to be argued with, and the author certainly achieves that goal. Many of the author's views on software engineering mirror my own, while others challenged me to rethink my perspectives. Whether I agreed or not, arguing with the various statements from the book helped me refine my views on various aspects of software engineering.

—Sergey Tselovalnikov
Staff Software Engineer, Canva

Over 20 years in academia, I've observed that Software Engineering majors often focus on Algorithms, Data Structures, and Programming courses; yet becoming a well-rounded software engineer requires much more. Artie Shevchenko, a lecturer at ITMO University, brings a refreshing industry perspective to the classroom. Based on his university course, his book *Code Health Guardian* provides thoughtful, real-world guidance on writing maintainable and efficient code—skills that go far beyond the fundamentals of programming. It bridges the gap between theoretical learning and practical application, offering guidance on how to write software that stands the test of time. Highly recommended!

—Andrew Stankevich, PhD
Associate Professor and Dean, ITMO University
Competitive Programming Coach, ICPC Gold Medalist (2001)

Code Health Guardian

The Old-New Role of a Human Programmer
in the AI Era

FOREWORD

George Fairbanks once noted, “Functionality is localized. Quality is diffuse.” That is to say, you can ask where to find the authentication code, but you can’t ask where to find the security code: security has to be everywhere in the code. Similar logic applies to performance, portability, observability, and all sorts of other software qualities.

This book focuses on code health, a crucial software quality in software engineering, sometimes referred to as readability, or software maintainability. If we are writing short-term throwaway code for ourselves, we can cut corners. But professionally, most of our code is for consumption by other humans, as well as ourselves in the future. What I’ve found, unfortunately, is that for any *quality* attribute we care about, we need to be diligent to avoid sliding down the slippery slope.

As we enter the AI Era, these lessons are more pressing than ever. At present, with LLM-powered AI coding assistants, we’re accelerating the production of code, especially in the hands of junior engineers. I have seen no evidence, nor do I have any reason to believe, that this form of AI is going to be a net positive on code health. As Ted Chiang wrote in *The New Yorker*[1], AI is a lossy compression over the input corpus: we ought to generally expect that on a chunk-by-chunk basis, AI agents produce code at roughly the average quality of the training data. For foundational models, that’s likely to be the publicly available code on the Internet. While there is some good work being done on fine-tuning and localizing such models, it’s hard to envision an outcome where a lossy compression of the training data results in an average quality improvement.

I for one don’t feel reassured by the idea of accelerating our production of (best case) average quality code. Don’t get me wrong, I love having these tools available, and I think they offer substantial advantages for many developers. I rely on them myself, especially when I’m coding in a language that I’m not deeply familiar with. But unlike context-aware autocomplete tools in our IDE in the 2010s, these agents eagerly *try* to produce increasingly bigger suggestions. Without humans in the loop actively guarding for quality, I fear we are going to go off the rails.

In this book, Artie has assembled a truly excellent collection of practical advice for reducing those risks. In clear (but detailed) language, this book lays out the case for better human involvement in monitoring for code health, as well as pragmatic guidance for getting started. While it’s difficult (if not impossible) to teach *taste*, the lessons in this book are great at conveying practices that will on average lead to better software design and code health.

I am hopeful that this book will become an essential reference for software engineers, giving solid basic advice on day-to-day topics such as interface design, contracts, coupling, naming, code review, and documentation, just to name a few.

For the junior engineers out there: take this book as a chance to think about *how* you practice your craft, rather than only the functionality you can assemble.

For the mid-career engineers: use this book as a chance to revisit ideas that you saw years ago but didn’t understand the need for. It’s a great refresher, and especially potent if you tackle this at the same time as you look at adopting new AI tools.

---

[1] `https://www.newyorker.com/tech/annals-of-technology/chatgpt-is-a-blurry-jpeg-of-the-web`

And for the engineering leaders out there: a copy of this book is a one-time cost, probably on par with the monthly cost of AI developer tools for a single user for a month or two. Can you afford not to have a copy handy for each of your development teams? How much will you regret it in a few years when the AI-powered bump on coding speed is eaten away by the messes produced by this year's AI agents?

I'm flattered to have been cited by Artie several times through the text, but I'm just as excited for all of the ideas he has drawn from other thinkers and repackaged here. I hope you all enjoy it as much as I have.

Titus Winters
New York
October 2024

PREFACE

The subtitle of this book, "The Old–New Role of a Human Programmer in the AI Era," may make you believe this book is about how to collaborate with AI agents. But it's not nearly about that, for two very simple reasons. First, I don't believe it to be the crux of the problem. Basically, you can be among the first to master a certain tool or prompting technique, but if it truly makes the difference, everyone else will catch up in a couple of months anyway. This reminds me of my mom trying to teach my son English while visiting us in Australia (she is an English teacher in Russia). The thing is, in just a few months, he'll be able to teach *her* English, given that now he goes to school and has English-speaking friends. In the same vein, over the coming years, we'll surely learn mostly organically how to collaborate with AI agents.[2] The second reason is that we're not there yet, and anything we know today about guiding AI to do what we want is likely to become suboptimal or even obsolete with the release of the next breakthrough AI model. So instead, let's focus on what I believe is going to be at the very center of the old–new role of human software engineers in the upcoming AI era, that is, how to keep our codebases reasonably simple. To fully explain the reasoning behind this statement, let me start from afar.

In 2017, I met a guy from Google X, Google's secretive moonshot factory, who also held a PhD from Stanford. Although he never shared anything about his projects, he once briefly mentioned he was saving Google millions of dollars per month. To my surprise, he also habitually referred to himself as a coding monkey. At first, I criticized him for doing so, but very soon I found out that he wasn't alone—in fact, the question of whether software engineering is worthy of being called engineering is as old as programming itself. In his classic book *The Mythical Man-Month*, Fred Brooks wrote, "Parnas[3] reproves me for writing about software engineering at all. He contrasts the software discipline with electrical engineering and feels it is a presumption to call what we do engineering" [Brooks95]. To take it even further, recently, one of Canada's provinces made it *illegal* to call programming "software engineering"![4] Although at first it may sound ridiculous, it is in fact quite logical. Most of the code in production, both today and 30 years ago, is a weird mix of what we call best practices combined with authors' own ideas and accumulated experience. It's just a fact. And we must admit—it doesn't sound like a solid science-grounded engineering discipline.

Not only is there a huge gap between computer science and software engineering, but we're also not on a trajectory to narrow it [Barr18]. There is a simple explanation for that: problems from mainstream software engineering look neither interesting nor promising enough for those in academia. On the other side of the fence, we programmers don't see much value in scientific research as the vast majority of it is impractical, and any rigorous analysis of most real-world problems seems to be unfeasible. Thus, a growing gap between programming and computer science has been a rather stable status quo for decades.

---

[2] Similar to how many of us have been learning to interact with GitHub Copilot recently. `https://docs.github.com/en/copilot`

[3] David Lorge Parnas is a Canadian early pioneer of software engineering, who developed the concept of information hiding in modular programming, which is an important element of object-oriented programming today. `https://en.m.wikipedia.org/wiki/David_Parnas`

[4] `https://www.cbc.ca/news/canada/calgary/tech-companies-alberta-premier-software-engineer-title-1.6617742`

Yet today, AI is here to disrupt our industry and challenge this status quo. The rise of AI has already begun elevating the role of science across our field, but it will soon do so at scale, elevating the role of science even for typical software engineering jobs. It's highly likely AI agents will become the best coding monkeys on the market, but what will be much harder for AI to master is managing code complexity, or as I call it in this book, keeping code healthy. Indeed, that's the most intellectually challenging problem in software engineering, so the role of the Code Health Guardian currently seems to be reserved for humans. Yet, in that role, understanding the theory behind code complexity is almost as important as your practical experience. That's already the case today, and it will become even more so tomorrow.

Many years ago, a friend of mine asked me why I care so much about theory instead of focusing solely on practice. Being such a good student at the time, I responded with a counter question: "But then, how would you deal with any new situation you encounter if your practice is not rooted in theory?" I don't think he was able to answer that, and that's how our short conversation ended. However, that's not the full story. It was a genuine question from his side as he's a man of practice. In particular, driving a car was a part of his job at the time, and he had really mastered it, so that when driving, it seemed he could literally sense the boundaries of his car. But it was him who was driving when I got into the scariest car accident of my life, just a few months after that conversation. And he wasn't speeding or anything like that. It was a rainy night, and we were driving on a highway. I remember those 20 seconds as if it were yesterday. When the left wheels of our car hit a rut full of water, at first, it looked like everything was still under control. The car began to slide and sway, but it was a barely noticeable movement. Yet the amplitude increased with every next sway, and in just several seconds, we were sliding along the highway at full speed, spinning and crashing into the guardrail and other cars. As you may guess, the problem was that my friend released the gas pedal and tried to level the car with the steering wheel. Despite him being one of the most experienced drivers I've ever known, he made a mistake that could have cost us our lives. And this happened simply because he was too focused on practice and ignored theory. Lack of theoretical knowledge in our field may not have such drastic immediate consequences, but it definitely makes it much harder for long-running projects to survive and for us to properly do the job of a Code Health Guardian, that is to keep our codebases reasonably simple.

There are so many coding ideas floating around that it's almost impossible to navigate them. It's so easy to accept and mimic some experts' practices, or something from your own past experience, or even an established industry practice, just to realize months later that it turned out to be counterproductive. It happens all the time! Therefore, this book has three primary objectives. It aims to:

1. Explain the theory behind code complexity
2. Use that theory to evaluate a bunch of industry's best practices
3. Train you to apply it to any code complexity-related problem that you may face in the future

When discussing coding practices, this book is slightly more scientific than you may be used to. However, it is exactly this kind of approach that empowers us to get the insights that we wouldn't get otherwise.

It's important to find the right balance between theory and practice in everything we do, and I hope this book makes its modest contribution to closing the aforementioned gap between software engineering and computer science. But most of all, I hope it will make your code simpler, your software

less fragile, your coding less stressful, and ultimately help you qualify for the old–new role of human software engineers in the upcoming AI era—the role of a *Code Health Guardian*.

Sydney, Australia
September 2024

How to use this book

This book tells the story of a healthy codebase—a codebase that is reasonably simple. After the introductory chapter, the book is divided into two parts: Part I: "The Code" (Chapters 2–7) and Part II: "The Key Code-Related Processes" (Chapters 8–10).

Chapter 1: "Introduction" demonstrates how vital code health is for productive development and highlights the importance of always striving for the pragmatic balance between short- and long-term productivity. Next, Chapter 2 introduces a code complexity model that is used throughout the book. It provides just enough theory to understand the relative value of the principles and techniques discussed, so that you can prioritize them and maybe even focus on the most important ones on your first read.

The order of chapters 3–10 generally doesn't matter, so feel free to read them in an order that's most meaningful to you. Chapter 3: "Design by Contract" arguably covers the most fundamental software engineering technique. It focuses on tests and assertions, given that these contract-enforcement mechanisms are widely applicable to both strongly and loosely typed languages. Chapter 4: "Interface Simplicity" introduces the concept of interface depth; analyzes the trade-offs of small, deep, and shallow modules; and covers several interface simplification techniques. But most importantly, it sets the tone for the rest of the book, demonstrating the importance of the interface-first mindset. Chapter 5: "Independence Quest" offers a fresh perspective on the software design principles that you're likely very familiar with and don't expect to learn anything new about them. Among other things, it illustrates how the theory from Chapter 2: "The Code Complexity Model" helps evaluate the relative importance of different best practices.

There are several chapters that I expect to be the most influential, as, from my experience, not that many developers practice what is being discussed there, probably assuming it's unrealistic. Chapter 6: "Semi-functional Programming" and Chapter 8: "Trade-off Analysis and Design Docs" each in its own way challenge the industry's status quo and stimulate you to try to do things differently. Both of these chapters offer pragmatic approaches that give you 80% of value for 20% of effort. Chapter 6 specifically covers immutability practices that deserve much wider adoption. Part I ends with Chapter 7: "Naming: The Obscurity Champion." Besides the subject matter discussed there, this chapter is full of real-life stories, insights, and is just fun to read.

Chapter 8: "Trade-off Analysis and Design Docs" opens Part II: "The Key Code-Related Processes." Without the second part, this book would tell only half the story of a healthy codebase, as code health starts outside of code. In this book, I apply the trade-off analysis technique from Chapter 8 to two practical problems—choosing between deep, shallow, and small modules (see Chapter 4: "Interface Simplicity") and between the build and buy options (see Chapter 9: "Build vs. Buy"). Understanding these two applications would be a great practice for those who are new to the formal trade-off analysis techniques, but, most importantly, these two chapters make it crystal clear what I actually mean by formal trade-off analysis in Chapter 8.

Chapter 9: “Build vs. Buy” covers a topic that is likely to be of interest only to senior developers. However, if you’re skipping any chapter, it’s worth reading the “Summary” section, which every chapter of this book ends with.

Finally, Chapter 10: “Code Reviews” covers the central code-related process. Code reviews are where the story of a healthy codebase is being written, so getting them right is tremendously important. Learning from Google’s wisdom derived from millions of code reviews done by hundreds of thousands of developers over two decades looks like a smart move. I advise everyone to read that chapter.

Code examples in this book are written in Java. They’re optimized for brevity, just to illustrate a specific point, so they may not follow all the best practices and even ignore some of the recommendations from other chapters of the book. Please don’t consider the provided snippets to be production-ready code.

I’m convinced that books must be studied and argued with, not just be read. While it’s very individual which sentences or paragraphs to meditate on, I think there are some common breakpoints in this book that are worth stopping at for everyone. These are marked as

Please treat such breakpoints as gentle nudges to check whether you’re just consuming content or are truly studying and reflecting on what you’re reading. For the very same reason, this book has many footnotes—also, treat most of them as nudges to argue with the book and meditate on the content. Furthermore, numerous external links and references are provided. Treat them as optional further reading—feel free to ignore them unless you’re particularly interested in delving deeper into the topic.

Chapter 1

INTRODUCTION

When it comes to software complexity, I will let the legends set the stage for us. In the 20th-anniversary edition of his classic book *The Mythical Man-Month*, Fred Brooks says:

> *The distinctive concerns of software engineering are today exactly those set forth [twenty years ago]: … how to maintain intellectual control over complexity in large doses.* [Brooks95]

Furthermore, Tony Hoare, although speaking about software complexity somewhat categorically,[5] perfectly conveys the very crux of the problem in his 1980 Turing Award Lecture:

> *There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.* [Ashenhurst87]

Finally, the last quote comes from John Ousterhout, whose book *A Philosophy of Software Design* highly influenced this work:

> *The greatest limitation in writing software is our ability to understand the systems we are creating.* [Ousterhout18]

I have no doubt that these quotes resonate well with every experienced programmer. Too often does complexity win over teams, but it shouldn't be that way. At the very least, we want a draw!

Code complexity is one of the biggest—if not the biggest—challenges in software engineering. The codebase of any nontrivial product is inevitably somewhat complex, which is mainly due to the complex nature of business requirements: if users want a program to do 30 different things, there will be a complexity cost to pay, regardless of the language and technologies used. To make it work fast, there may be even more unavoidable[6] complexity to add, and the same is true for other non-functional requirements: security, reliability, privacy, and so forth. Complexity is often a necessary trade-off. That's why, pragmatically speaking, we aim at having a healthy codebase, not a simple one. Sometimes, I would still use a well-established "simple code" phrase in this book, but keep in mind that, strictly speaking, there's not much production code in the world that is truly simple in the full sense of that word.

That said, code is not responsible for *all* the complexity in software projects: it may be a complex UI, chaos in the database, burdensome development methodologies, organizational complexity, complex

[5] Section 1.2 probably explains why.

[6] Not necessarily *essential complexity*, as that term is commonly understood. It may be unavoidable in one technology stack and absolutely unnecessary in another.

tooling, and so forth. However, the codebase, if complex, typically dominates every other problem.[7] So, in this book, *code complexity* is implied whenever the term complexity is used without any clarification.

Summary

While there are many aspects of a software project that can make it complex, source code is the main complexity contributor, which typically dominates everything else. There are three interconnected productivity problems that a complex codebase introduces: developer exhaustion, bugs, and slow development. Code complexity is highly contagious and reinforces itself in multiple ways (see Figure 1.1). Furthermore, it typically grows incrementally as thousands of small complexities accumulate and interconnect over time, which makes it extremely difficult to reverse the damage. Unlike programming, software engineering requires a strategic approach to complexity. Normally, you should be somewhere in the middle on the *-term productivity spectrum (see Figure 1.3). That's what most long-running projects require from us—to constantly balance between short- and long-term productivity, with code health being one of our top priorities.

[7] It doesn't have to be execution code, which first comes to mind, but configurational code as well. Configuration can be complex in itself, or it can stimulate complexity growth indirectly (e.g., the way bad database schemas provoke data chaos).

Chapter 2

THE CODE COMPLEXITY MODEL

When somebody refers to a piece of code as being complex, what they typically mean is that it has poor readability. We often think of code complexity as if it were a single problem, and that's also how it was presented in the previous chapter. However, in practice, there are three distinct problems at play here: high cognitive load, change amplifications, and unknown unknowns. Being able to tell them apart is very useful as they infrequently go hand in hand and greatly differ in severity. If you're familiar with the code complexity model from *A Philosophy of Software Design* [Ousterhout18] by Dr. Ousterhout, then Section 2.1 will also look very familiar to you. But note that what Dr. Ousterhout calls "complexity *symptoms*," I call "complexity *problems.*"

In addition to the three complexity problems, this chapter also introduces the seven underlying causes of complexity. And that's where it differs substantially from Dr. Ousterhout's model, as he identifies only two complexity causes (i.e., obscurity and dependencies), saying that all the complexity stems from either of them. While I agree, "dependencies" as a complexity cause sounds a bit vague. So instead of obscurity and dependencies, this chapter introduces seven more specific causes:

- Obscurity
- Duplication
- Unfamiliarity
- Too many dependencies
- Unstable dependencies
- Untrue interfaces
- Complex interfaces

Chapter 2 thus sets the *foundation* for the rest of the book. It introduces the code complexity model that will be assumed and extensively used in the chapters that follow.

In addition to setting the foundation, Chapter 2 also provides a *book overview*. It outlines the story of a healthy codebase, providing references to different parts of the book; therefore, you can use it as another map[8] to navigate to the topics of most interest to you.

Summary

Code complexity is not a single problem. In fact, there are three problems: change amplification, cognitive load, and unknown unknowns. Change amplification is the least critical one as it typically doesn't lead to exhaustion or bugs—it just slows you down. Unknown unknowns is the worst one as it makes you rely on luck when making changes.

Healthy code is code that looks familiar, has little to no obscurity and duplication, and has a reasonable number of stable dependencies with true and simple interfaces. Same as complexity problems

[8] The "How to use this book" section provides the first such map.

themselves, not all the causes of complexity are equally bad. They should be prioritized based on the severity of the complexity problems they create.

Finally, somewhat counterintuitively, a healthy codebase may have quite a lot of complex code. That's because "isolating complexity in a place where it will never be seen is almost as good as eliminating that complexity entirely" [Ousterhout18].

The code complexity model introduced in this chapter will be assumed and extensively used in the chapters that follow. Yet note that the order of the following chapters generally doesn't matter, so feel free to read them in an order that's most meaningful to you.

Chapter 3

DESIGN BY CONTRACT

Modular design and encapsulation are the bread and butter of software engineering. However, like any powerful and flexible technique, they can be harmful. An extra module can increase code coupling instead of decreasing it, or the interface exposed may be too complex, incomplete, or even plainly misleading. In the next three chapters, we explore the pitfalls of modular design, starting with its most fundamental problem—building trust in modules' interfaces. The first part of this chapter introduces the concept of contracts, and the second one provides an overview of relevant documentation practices and contract enforcement mechanisms. To build trust in modules' interfaces, their contracts must be properly documented[9] and enforced.

The ultimate goal of this chapter is to make it crystal clear that Design by Contract is not one of the myriad optional techniques that developers may or may not adopt, depending on their school of thought, but rather an essential ingredient of any nontrivial healthy codebase.

Summary

Document and enforce contracts internally—that's all it takes to make interfaces trustworthy. If you practice these two rules consistently within your codebase, you'll notice that over time, it builds trust in your interfaces among developers and, as a result, dramatically improves the health of your code. Trustworthiness of interfaces is the first cornerstone of a healthy codebase.

[9] Which sometimes means no documentation at all; see Section 3.3.

Chapter 4

INTERFACE SIMPLICITY

The previous chapter demonstrated how important it is for interfaces to be trustworthy. Yet, in addition to being true, good interfaces must also be simple. That's their second property that is so essential that it deserves a separate chapter as well. Of course, interfaces are not the only code that needs to be true and simple, but as we'll see in Section 4.1, this is *especially* important for them.

Luckily, it's also much easier to have simple interfaces than to have simple code in general. Usually, saying "Our codebase is simple" is a bit of an overstatement, but having truly simple interfaces is absolutely realistic. So given the utmost importance and feasibility of interface simplicity, investing in it is a wise choice. You won't regret the time and effort spent on simplifying interfaces.

Summary

Simple interfaces are your top priority as a Code Health Guardian. "Simple" here means not only easy to read but also easy to use, which typically means deep. Aim at building all your modules deep, and try to keep them small as long as it makes them more readable and not dysfunctional. At the same time, feel free to write large modules if decomposing them into smaller ones is problematic.

In this chapter, we've discussed three particular interface simplification techniques, but these are not the only ones covered in this book. When discussing code health principles and techniques in the following chapters, we'll focus more on how they are applied to interfaces rather than implementations. That's the interface-first mindset in practice.

As a reminder, chapters 3–10 are mostly orthogonal to each other and greatly differ in complexity, so feel free to read them in an order that's most meaningful to you, or even skip a particular chapter.

Chapter 5

INDEPENDENCE QUEST

Understanding a function that makes 20 method calls across 10 different classes may be challenging. It's hard to reason about code with too many dependencies. Yes, there are trivial ones, such as `Math.max(x, y)` calls that can hardly complicate anything, but that's more of an exception. Most dependencies *do* count toward our brains' quota, whether these are local variables, parameters, fields, invariants, or function calls. So, what about making our code more independent? That's the idea explored in this chapter.

As a reminder, for the purposes of this book, a dependency exists when a given piece of code cannot be understood or modified in isolation, which is commonly referred to as coupling. In Chapter 2, when talking about the "too many dependencies" complexity cause, we compared our brain's working memory to a limited number of CPU registers. However, this analogy isn't flawless as it might give an impression that this complexity cause is relevant to individual statements only. While a single logical transition involving multiple facts can definitely be complex, the "too many dependencies" problem is relevant at every level: a *function* may have too many dependencies, or a *class*, or even a *microservice*, or some *high-level design*. In this chapter, we're going to explore the idea of having fewer of them—fewer dependencies, or less coupling, if you will.

This chapter opens with a section on knowledge duplication, which is basically coupling in its extreme form. Then we'll talk about code discoverability, using the right abstractions, and anemic domain models, wrapping it up with a tempting idea of introducing an extra level of indirection.

Summary

Given that dependencies are an indispensable part of any code, having less of them is tricky but doable. One of the least controversial and most important principles here is avoiding knowledge duplicates, which are essentially dependencies taken to the extreme. Don't violate the DRY principle without a good reason, and if violating it, use bidirectional pointing comments to mitigate the problem of unknown unknowns. To avoid *unintentional* duplicates, make sure your code is easily discoverable. Finding the right home for your code is an important part of that—build a rich domain model based on the right abstractions with a properly encapsulated behavior. Finally, avoid dysfunctional layers, that is, those that do not take any substantial responsibilities and mostly just delegate. Following these principles will help you have exactly as many dependencies as needed.

Chapter 6

SEMI-FUNCTIONAL PROGRAMMING

Perhaps to the disappointment of some and the relief of others, this chapter is not about functors, monads, Haskell, or Scala. Instead, it's about what I've learned as a Google software engineer writing in Java.[10] Even if you've never written a line of code in functional style, I bet you've heard about functional programming (FP) from your friends or colleagues. But do you remember exactly from whom? If it was one of your smartest friends, you're not alone. While the popularity of FP among the brightest minds stems from its academic roots, even more fundamentally, I believe it's a passion to understand code that attracts so many of them to FP. I know it may sound counterintuitive, because to many of us, even good functional code often looks unfamiliar and confusing. However, simplicity not only precedes complexity but also follows it. Although FP is more complex than mainstream imperative programming at first glance, it paves the way to more understandable code. Just bear with me for a moment, and I'll explain how.

When we say we understand something, we're often wrong. Even though as programmers we experience the veracity of this statement more often than other professionals, still, it may take us years to fully comprehend how true it is. We can spend many years believing that we understand the programs we're working on (or at least we're almost there),[11] only to wake up one morning and finally realize how fundamentally complicated programming really is [Barr18]. Yet, at this point, it may be easier to accept the status quo than to challenge it. But not for those brilliant folks—they are something different (maybe you even find them a bit odd). They are extremely curious, and you can say even crazy about understanding things.[12] And I think the reason why so many of them are attracted to FP is that, in the FP universe, they feel they can reason about their own creations and not be wrong, which is a great value proposition, especially in the context of big complex software systems. Don't get me wrong, FP doesn't make the Herculean task of understanding them simple, but it does make it more tractable.

Reading the above, you may be thinking I'm trying to convert you to FP, but that's not the case. I'm not one of those functional folks, and I wouldn't argue for a full-fledged FP. But at this point, hopefully, you're already thinking: "There should be something in functional programming. Is there any chance we can adopt any of its practices to simplify our own code?" I believe the answer to that question is, "Yes, we should strongly prefer immutability over mutability." It's mostly the lack of side effects and *immutability* that makes functional code easier to reason about. While many of the mainstream programming practices were originally developed in the context of FP languages,[13] unfortunately, arguably the greatest idea behind FP is still not quite popular. Immutability is what I truly discovered for myself only after joining Google, and I still wonder why people talk about it so little. It shouldn't be some sort of secret wisdom. So, let's take a closer look at what "strongly preferring immutability" means in practice by examining the three pillars of what I would call semi-functional programming.

---

[10] Moreover, when I joined, it was Java 6 (no streams support); only later on we upgraded to Java 8.

[11] Especially, if we don't make it easy for our users to report bugs.

[12] Yes, they're often not that product focused, but the only thing we care about in this context is their passion for understanding, because ultimately, it has a potential to serve the product (see Figure 1.1).

[13] For example, the type inference, lazily evaluated streams, and generics [Seibel09].

Summary

The number of moving parts is often a huge cognitive load issue by itself. But what makes the moving parts problem even more serious is obscurity around which potentially moving parts are actually moving, and in which scenarios. That's why sticking to the immutability principles of semi-functional programming greatly simplifies reasoning about code. In larger systems, you can even say it makes reliable reasoning possible. Here are the three principles of semi-functional programming:

1. Strongly prefer immutable objects and collections.
2. Treat function parameters as read-only.
3. Make all the mutability explicit.

Semi-functional programming is all about immutability by default. This is certainly so for objects and collections, and maybe even for the variables themselves as well. It's about functions that don't modify their parameters. However, at the same time, it's about being flexible enough to allow mutability where it's handy, but ensuring such mutability is explicit.

Those who love functional programming may argue that it's not functional enough to be called semi-functional. And those who strongly prefer the flexibility of imperative style may say that this approach is too restrictive and *too* functional. But that's okay—it's on the spectrum somewhere, in the spirit of pragmatic excellence.

Chapter 7

NAMING: THE OBSCURITY CHAMPION

You've probably already heard this old programming joke, but I can't stop myself from using it to open a chapter on naming: "There are two hard problems in computer science: cache invalidation, naming things, and off-by-one errors."[14] Although I'm not that sure about off-by-one errors, naming things most definitely deserves its place on the list!

Finding the right name can be truly hard, but sometimes, it's hard only because it is illusively simple. Let me start with a confession. This book grew out of my lectures at ITMO University, and as a part of the negotiation process to start that course, I sent a calendar invite to an associate professor titled "Chat about ITMO lectures." Guess what? While that title made total sense to me, for him, half of the meetings in his calendar could have been named the same! I'm still amused by the fact that he accepted my invite. That was a terrible mistake to make for someone applying to teach code simplicity, as good names are such an important component of it.

It was the same year that I was sitting in a room with ten other people, many of whom I didn't even know. I believe we were trying to simplify some sentence structure when I mentioned that "I work with words a lot on a daily basis." Then somebody, who turned out to be a professional linguist, got intrigued and asked me about my occupation. Up to that point, she was obviously assuming she was the only expert in this field among us. Actually, I was surprised as well—me articulating it felt like a revelation to myself, too. But it's so true! Software engineers should be sort of professional linguists. We should always be on a journey to learn how to name things better and how to make our code read more like a story.

Summary

Starting our conversation on naming with "Name different things differently" and finishing it with "Name same things the same" is symbolic. There is no rocket science in naming. But it is illusively simple. That may be the reason why I've never seen a self-development goal like "This quarter, I want to master naming things well." It just doesn't sound ambitious enough! Instead, there would be goals such as mastering a new framework, production monitoring, integration tests, infrastructure setup, and so on. However, in practice, having a not-that-measurable goal of mastering complexity management (and specifically mastering naming) would have a bigger impact on your overall productivity and your career as a Code Health Guardian. The good news is that you don't need to spend a lot of time expanding your vocabulary to be better at naming. Just being more conscious of and collaborative about it makes the biggest difference.

[14] `https://martinfowler.com/bliki/TwoHardThings.html`

Chapter 8

TRADE-OFF ANALYSIS AND DESIGN DOCS

This book tells the story of a healthy codebase. So far, we've talked about the code itself—complexity problems, their causes, and various code simplification techniques. However, it would be only half the story if we stop here, as code health starts outside of code. Even the cleanest code possible can still be confusing if the technologies used, the underlying approach, or the overall architecture are supposed to be fundamentally different.[15] That's why the rest of the book covers the *key code-related processes*: trade-off analysis, design docs, build-versus-buy decisions, and code review. Without them it's hardly ever possible for a nontrivial codebase to be truly healthy.

Although up-front design is no longer mainstream,[16] over the last decade, I've been fortunate to work for several companies that not only use design docs, but also view them as one of the key artifacts of the development process. The main thesis of this chapter is that they are right—for most projects, design docs are objectively the best way of doing trade-off analysis. Design docs are mostly about effective collaboration and better-informed decision-making, but their benefits don't stop there. In this chapter, we'll explore a wide range of their benefits and compare design docs with the alternative approaches.

Even if you're already familiar with design docs and practice writing them, I wouldn't recommend skipping this chapter—at the very least, skim through the headings and read the Summary section. I personally find it extremely helpful and motivating to remind myself of a full spectrum of benefits that design docs, if done properly, offer. Few of us are truly immune to the temptation to skip the proper trade-off analysis step and jump straight into the coding.[17,18] Reinforcing the best practices may be as valuable as learning them in the first place.

Summary

Design docs are the best way of doing trade-off analysis for most projects as they standardize the process and make it timely, explicit, collaborative, and centralized. In addition, they:

- Enable diagrams in code
- Have a team-building effect
- Serve as a website for a project
- Serve as an important artifact for the next performance review cycle

[15] For example, just recently, I wrote a piece of super confusing code in an attempt to get some data consistency guarantees without a huge performance hit out of the database that doesn't provide it out-of-the-box.
[16] The unpopularity of design docs today makes me even more eager to bring up this topic.
[17] And this is mostly because it's so easy to believe that our latest idea is definitely the best one.
[18] I'm not referring to true prototyping here. A true prototype is 100% complimentary to design docs [HT19, "Prototypes and Post-it notes"].

These reasons combined make starting any nontrivial project with a design doc a great default strategy. And whenever you lean toward not writing a design doc, consider writing a one-pager instead!

Although it's not strictly necessary for design docs to be a part of your team's culture to be beneficial to you, they shine when they are. So, if your team doesn't practice writing them, be a catalyst for positive change! Yet, keep in mind that design docs are just a tool to make you, your users, and your teammates more productive and happier. Not the other way around. It's easy to get overly excited and take it too far. That's why so many experienced folks repeat it twice that they mean one-pager when asking somebody to write a design doc.

Chapter 9

BUILD VS. BUY

If you're not new to software engineering, surely you know that only a few experiences match the satisfaction of deleting code, as less code in your codebase means less code to maintain. In terms of the formula of codebase complexity from Chapter 4 (see Eq. (1)), it's great when you can isolate complexity making the t factor small for a particular term, but removing that term entirely is even better. Sometimes, this can be achieved by eliminating unnecessary functionality; yet another option is to simply delegate that complexity and make it somebody else's problem.

This chapter is about reusing existing stuff: libraries, services, frameworks, components, and so on. It may be more relevant for senior folks as the high-stake build-versus-buy decisions often pop up to the top, becoming brain teasers for architects[19] and senior leadership. However, from time to time, all of us face the dilemma whether to implement something by ourselves or adopt an existing solution. In this chapter, we'll explore how to make better-informed build-versus-buy decisions.

Summary

In build-versus-buy dilemmas, default to adopting an existing solution, as reinventing the wheel is rarely a better option. It may be if the implementation is trivial or if existing solutions are too expensive, result in a vendor lock-in, or introduce security or reliability risks that are too high. Worth mentioning, we tend to underestimate the costs and risks associated with *our own* solutions.

Widely known mature modules have a broad range of benefits:

- They are usually better documented, supported, and have fewer bugs.
- They almost certainly won't suddenly cease to exist.
- They often guarantee backward-compatibility.
- They are more likely to have the features you will want in the future.

[19] Yes, it's great if they disappear as a strata (see Chapter 8), but I'm using the word "architect" here as a synonym for "more senior folks," to not repeat the phrase that was already used in the previous sentence.

However, if you're unlucky enough to ever want something that's not there, the module being widely known and stable only increases the risk of your feature requests being ignored. So prefer open-source modules as they provide you with more options to meet your future needs and even mitigate the vendor lock-in problem to some extent.

Finally, when facing a build-versus-buy dilemma, don't rely on general recommendations—do your own trade-off analysis to make a decision. The devil is in the details.

Chapter 10

CODE REVIEWS

It is extremely rare that the first draft of any code change is good enough. A healthy change is typically the result of multiple iterations of review and improvements. It's a step-by-step process in which each subsequent step provides the necessary insights for the following one. Yet, it's not about iterations only—a fresh perspective is another essential ingredient, and the easiest way to get it is to ask somebody else to take a look.[20]

In this final chapter, we'll discuss such collaborative code reviews. Yes, it is important for authors to review and iterate on their code themselves, but not getting any external opinion is hardly ever the best approach.[21] And this is not only so because of the reviewer's fresh perspective, but also because of their expertise.

In the coming years of AI dominance, it is our expertise that will empower us to outperform AI reviewers. A scientific approach, combined with our creativity, experience, and complex reasoning capabilities will likely help us remain the best candidates for the old-new role of a Code Health Guardian for the next few years.

Summary

Code reviews are the primary means of keeping your codebase healthy, but that's not their only goal: they're also one of the best opportunities to be on the same page with each other and effectively share knowledge and provide constructive feedback. In addition, they play a crucial role in shaping the trust climate within a team. Last but not least, it's important to deliver quickly and not get stuck in code reviews.

There are multiple practical tips and tricks to strike the perfect balance among these competing objectives:

1. Review every line.
2. Aim to review within one business day.
3. Aim to approve quickly.
4. Label nonsevere comments.
5. Don't be nitpicky or pedantic.
6. Cite your sources.

[20] If you want to get a fresh perspective by yourself, then taking a break and getting back to your code the next week might be the best idea.

[21] In the 1980s, Donald Knuth famously designed and implemented TeX all by himself (maintaining an active feedback loop with the TeX community). Yet it's likely that even he himself would be more productive with the help of the formal code reviews as we know them today.

7. Assume competence and good will.
8. Provide guidance.
9. Leave positive comments.
10. Choose your battles wisely.

There are six main rubrics to pay attention to in code reviews:

- **Test coverage:** Are all the necessary tests there?
- **Correctness:** Does it look correct to you?
- **Complexity:** Is it the simplest change possible?
- **Security:** Are there any security issues?
- **Documentation:** Is there any relevant documentation, and is it (still) accurate enough?
- **Consistency:** Does the code follow your style guide, and does the solution fit well into the overall architecture?

I believe code reviews will be at the very heart of the old-new role of human software engineers in the upcoming AI era—us still being Code Health Guardians even when most of the code will be written by AI agents.

EPILOGUE

Leo Tolstoy famously wrote, “All happy families are alike; each unhappy family is unhappy in its own way.” Over time, I’ve become increasingly convinced that this holds true for nontrivial codebases as well. Ultimately, it is this realization that led to the idea of writing this book—to publish the story of a healthy codebase. That was in 2022, half a year prior to the release of ChatGPT, which I believe made this topic even more relevant. If there’s one thing that’s challenging for AI, it’s complex reasoning, so managing code complexity (the most intellectually challenging problem in software engineering) seems to be our primary competitive advantage over the AI coding monkeys. At least for some time.

While living in Cupertino, I used to pass by tennis courts seeing the same coach playing with kids. That was more than five years ago, but I can still hear his voice in my head—it was just a single phrase with a strong Mexican accent repeating over and over again: “What you’re gonna do? What you’re gonna do? What you’re gonna do?” He was clearly focused on making his students think positively and proactively about the situations they find themselves in on the court. “Tell me what’s your next move!”—he wanted to prevent any situation from paralyzing them. Instead, what he demanded from them was to make a clear decision in a rapidly changing environment. Today, looking at how AI is disrupting our industry, we’re all at that decision-making moment, and if anything, we better not be paralyzed by it. As a first step, we need to clearly hear that call to positive thinking: “What you’re gonna do?”

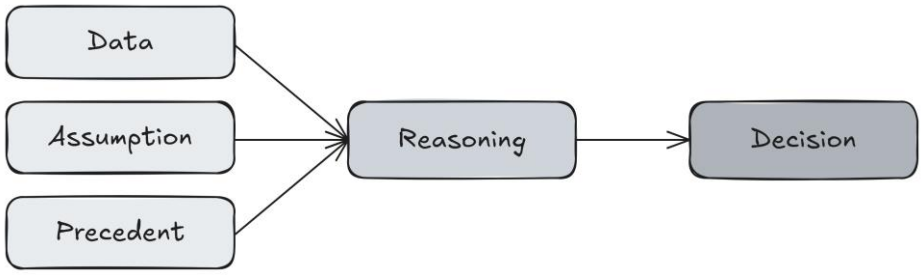

Figure X.1. The four components of real-life decision-making.

Yet, what makes this situation really challenging is the limited number of inputs we have to make our decisions. Google is known to be a data-driven company—internally, it sometimes looks like there is data behind every single decision being made. Yet, in reality, for most of them, there are four components involved (see Figure X.1). Here is how *Software Engineering at Google* puts it:

> *Being data-driven is a good start, but in reality, most decisions are based on a mix of data, assumption, precedent, and argument.* [WMW20]

It’s simply because you rarely have enough data to make a truly data-driven decision. Thinking of what we should do to prepare for the upcoming AI era, it’s even worse—it’s not only the lack of data, but it’s also the lack of any reasonable precedents in the past. People compare it to the invention of the wheel, electricity, computer, but even being a strong believer in the value of precedents in decision-

making,[22] I feel like all these analogies are flawed. When talking about AI and LLMs in particular, we're talking about a totally new category—a new type of breakthrough that we've never seen before. So if I'm right, assumptions and reasoning are all we have for making our decisions today on how to prepare for the upcoming AI era (see Figure X.2). That looks pretty bad, as it's basically the definition of speculation.

Figure X.2. The two components of decision-making left, when making decisions to prepare ourselves for the upcoming AI era.

How smart will AI get? How fast? If its complex reasoning capabilities tomorrow become better than ours, that would make this book totally redundant.[23] Yet, with an assumption that we'll continue to be smarter than AI for some time, we can still compete with it for the most intellectually challenging parts of our jobs. We, humans, have good chances of remaining the best Code Health Guardians on the market. Maybe for several more years, or who knows, maybe even for several more decades.

[22] I feel like precedents are generally undervalued, people tending to unproportionally focus on data, assumptions, and reasoning in decision-making. We could have learned more from other people's experience and mistakes.
[23] Well, except that AI could be trained on it, but hopefully not.