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

- Editor: Lana Todorovic
- Proofreader: Cheryl Lenser
- Indexer: Cheryl Lenser
- Cover images: Glebstock/Adobe Stock, Fotomay/Shutterstock

ISBN 978-1-7637717-0-3

For my beloved wife Juliana

Soli Deo Gloria

CONTENTS

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.

---

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

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.

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

---

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

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.

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

---

[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`

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.

The best way to communicate with me about this book is to send an email to the following address:

`code-health-guardian@googlegroups.com`

If you'd like to see what others are saying and participate in discussions, you can join the `code-health-guardian` Google Group. If for some reason that Google Group disappears in the future, you can find updated instructions on how to communicate about the book at `http://codehealthguardian.com`. On the same website you can also find information on how to report typos.

Acknowledgments

First of all, I carry a heavy debt of gratitude to my wife, Juliana, and my children, Daniel, Kathy, and Eva. Juliana, thank you, dear, not only for enduring my three-year-long obsession with writing this book but, perhaps even more importantly, for supporting my decision to leave Google and return home to Russia to focus on the startup and university lectures. Your willingness to embrace my ideas and ideals, even if it meant leaving the comfort of life in the United States, is something I deeply appreciate. When we got married, you promised to follow me wherever I go, and you've kept that promise. I'm truly thankful and proud of you.

There are three key events in my life that made this book possible. The first one is John Ousterhout's book presentation at the Sunnyvale Google office in 2018. I still believe *A Philosophy of Software Design* [Ousterhout18] is one of the best books for developers, and together with *The Pragmatic Programmer* [HT19], these two books inspired my university classes, which ultimately led to the creation of this book. Dr. Ousterhout, thank you for your work, for the `software-design-book` Google Group, for responding to all my emails over the

course of the last five years, and for your best wishes for the *Code Health Guardian* book project.

The second key event is when I started the Good Code Reviewer classes internally at Google. I believe it's teaching those classes that gave me the necessary experience and courage to later teach at the university. Code review classes (as well as the last chapter of this book) were mostly based on Google Engineering Practices recently made public.[5] Although your name isn't explicitly mentioned there, Max Kanat-Alexander, I know you're the author of most of it. It's your wisdom that ultimately made those classes possible, and I want to express my gratitude for your support in that initiative.

Finally, the most important event is starting the Productive Software Engineering classes at ITMO University. Andrew Stankevich, thank you for being open to something new and giving me a chance (there are details in Chapter 7). This book was born out of those classes, with me iterating on their content for four years now. But of course, it's not just me iterating—I want to express endless thanks to my students for their feedback and counter-arguments. Truly, there's a reason why ITMO has more ACM Programming World Cup[6] titles than any other university in the world. Engaging in discussions with you was often a challenge, but I'm grateful for taking it on.

Titus Winters, your support and advice at the early stages of writing the book were invaluable. I know it ended up being a slightly different book than originally discussed, but that's what time and the release of ChatGPT brought with them. I'm especially thankful that you agreed to write a foreword to this book!

Leo Osang, thank you for a gentle nudge in early 2024 to finish this work, which was collecting dust on the shelf for most of 2023.

---

[5] `https://google.github.io/eng-practices/`

[6] ACM ICPC (International Collegiate Programming Contest). `https://icpc.global/`

Jesse Lu, I believe you were the earliest collaborator on the content when I was drafting the “Semi-functional Programming” chapter. Thank you for having so much patience with a first-time author.

This book has been tremendously improved by the suggestions and assistance of many reviewers. I would like to thank Adam Barr, Amy Wang, Ankit Ashri, Alex Shestakov, Melissa Ermishina, Tin Pavlinic, Erika Shefer, Sergey Kazakov, Viktar Sharepa, Aleh Boitsau, Egor Nazarov, Taras Skazhenik, Denis Vorkozhokov, Igor Podtsepko, Ilia Zaitsev, Rahim Nakimov, Nikita Kozhukharov, Laurynas Kavaliauskas, Alex Gemberg, and Kim Heusel, who reviewed the book at different stages, provided feedback, and made suggestions that helped make this book better.

Lana Todorovic, it truly was an amazing experience editing the book with you. Your meticulous work in fixing my English and making it easily readable was invaluable. I appreciate your dedication, attention to detail, and especially your humbleness.

Cheryl Lenser, thank you for your incredible work as both indexer and proofreader. Your sharp eye, experience, and precision ensured every detail was perfect, and your contributions have truly improved this book.

There was one traditional publisher that I shared the book proposal with, and that was O’Reilly. I want to thank you Louise Corrigan for your time—even though you were hesitant about publishing it, our relatively short interaction has definitely helped me with the publishing process.

Finally, I want to express my gratitude to my manager and teammates at Canva for having my back during the rollout of the ambitious project we’re working on right now. I recognize that taking a couple of months off during the final stages of working on this book was less than ideal. Justin Velluppillai, thank you for all your support along the way and for approving my countless leave requests!

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,[7] 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]

[7] Section 1.2 probably explains why.

Pause and Think

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[8] 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 tooling, and so forth. However, the codebase, if complex, typically dominates every other problem.[9] So, in this book, *code complexity* is implied whenever the term complexity is used without any clarification.

---

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

[9] 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).

1.1. Productivity implications of complexity

It is not an overstatement to say that a complex codebase poses an existential threat to any long-term project. The first reason for that is that it "sucks the life out of developers," as per Ray Ozzie, former CTO of Microsoft [Martin08]. That by itself is so serious that we could have stopped here and proceeded with Chapter 2. I know I may be biased, as I'm a software engineer myself, but even if you're not a developer and don't care much about their wellbeing, there are still good reasons for you to care about code health. Apart from exhaustion and increased turnover, there are at least two other major productivity problems that code complexity brings with it:

1. **Complexity makes the product buggy.** You can never be sure that a complicated piece of code is bug free. Bugs tend to hide for long periods of time, sometimes even compensating for each other, just the way two erroneous minuses in math do so. But even if there are no bugs in complicated code yet, it's not for long—just give it enough time and iterations. To make things worse, when fixing a bug in complicated code, it's super easy to introduce another one, which is the first vicious productivity regression cycle in Figure 1.1.
2. **Complexity slows you down.** Obviously, in a complicated codebase, it takes more time than it should to accomplish things, which is often an even bigger problem for a product (and product managers) than bugs. And unfortunately, "adding manpower to a late project just makes it later."[10] [Brooks95]

So, here are three main productivity problems coming from complex code: developers are exhausted, product is buggy, and development speed is low.

[10] Depending on how complicated your code is, you may be able to improve the development speed short-term by adding the right developers with the right experience. However, growing the team beyond a single-digit number of programmers in the long-term would also accelerate the growth of complexity, which will ultimately slow you down, no matter how much you increase the project's budget.

Once accumulated, these problems quickly shift any project into maintenance mode.

As you may have already noticed, these three problems are interconnected (adding more arrows to Figure 1.1):

- Bug fixing slows you down.
- "Quick fixes" introduce more complexity.
- Exhaustion doesn't help with the development speed and incident management (and the quality of the product overall).

Pause and Think

Last but not least, chaos in code is contagious. Not only are people afraid of refactoring the code they don't understand, but also having a broken window causes other windows to be broken soon, as it is perfectly depicted in the "Software Entropy" chapter of *The Pragmatic Programmer* [HT19]. The feeling that nobody cares about a little bit of additional complexity is appealing and contagious. That's yet another way how code complexity multiplies itself, introducing the final vicious cycle to our productivity regression diagram (see Figure 1.1). I think it already looks bad enough for us to get scared, wrap up this section, and move on.

1.2. Complexity: The silent killer of software projects

The productivity implications of a complicated codebase are scary, but what makes it such a dangerous enemy is that *complexity may look pretty innocent until it's too late*. Ignoring complexity is like slowly accelerating on a wet road—you're going increasingly faster, and everything seems to be okay and even fun until you start losing traction. Way too often, teams cope with growing complexity for a long time, barely noticing how complicated it already is.

Progressively, we just get used to the fact that we need to spend a bit more time exploring the codebase to accomplish a task at hand,[11] until one day we wake up and realize that our codebase is super confusing. What makes this scenario likely is the fact that software engineers are smart—we can digest quite a lot of complexity before even noticing it (see Figure 1.2). But by then, it may be too late—productivity problems start growing too fast when one is operating near the limits of their cognitive abilities.

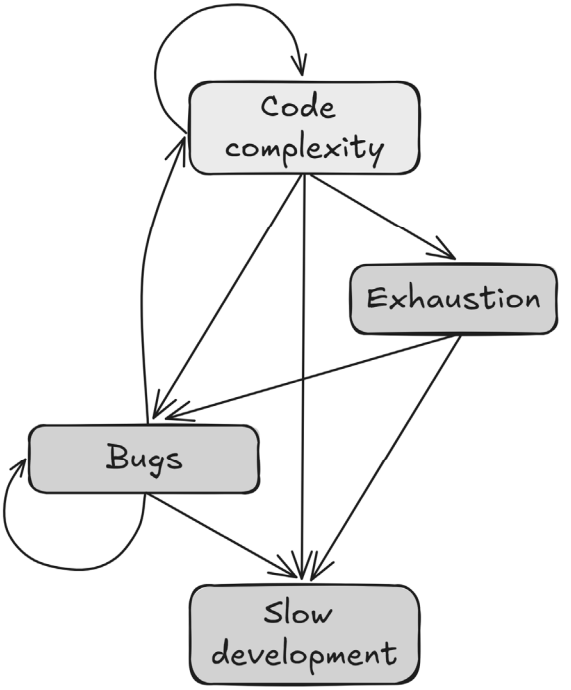

Figure 1.1. Productivity regression in complicated codebases.

[11] And not break something else.

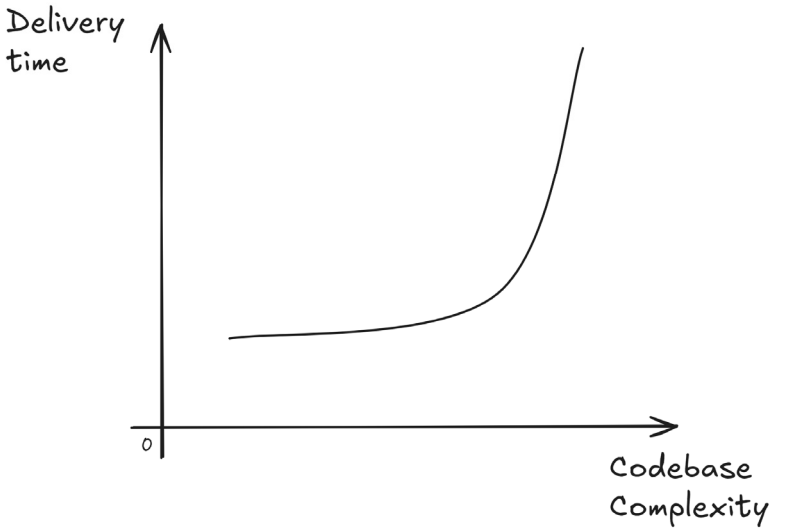

Figure 1.2. Growing complexity can be manageable and even unnoticeable for quite some time.

Finally, the worst thing about the codebase complexity is that there is almost no way back. Complexity grows incrementally, making it extremely hard to reverse the damage. At some point, you may decide to win it back—to refactor your code into something reasonably simple, but after multiple weeks of dedicated refactoring efforts, there would likely be very little progress just because it's impossible to pinpoint the origin of all that complexity. And there is no single origin. It's thousands of minor complexities that accumulated and interconnected over time. We had a team in Google Maps that was moving fast until they couldn't, so they had to take a break and spend over a year just refactoring their code. This is not something every company can even afford.

You can think about accumulating complexity as driving on a highway section without any exits. You move fast, accumulating complexity without even

noticing the speed you're driving at. Yet, deciding to pull over and back up to take the previous exit would be risky, and you'll definitely be surprised how far you've gone. Refactoring code to make it simpler, very similar to backing up on a highway, is slow and painful, and starting from scratch may be the only alternative.[12]

Avoid putting yourself in a situation where there are no good solutions. It's important to remember that code complexity often goes unnoticed and is extremely hard to undo. Be constantly killing complexity, or it will be killing you down the road—both your mood and your project.

1.3. *-term productivity spectrum

Moving fast, ignoring the complexity issues, is programming *tactically*. Even though it may be okay when prototyping or working on a tiny project,[13] continuous tactical development for a typical project inevitably leads to a dead end. Even short-term, it's usually not the best choice to cut corners for the sake of fast delivery. In the words of Max Kanat-Alexander:

> *Some people believe that writing simple code takes more time than quickly writing something that "does the job." There is no data of which I am aware that validates this idea. Serious software development nearly always has long timelines—weeks or months at the shortest. When you add complexity into your program, you're slowing yourself down tomorrow.* [Kanat-Alexander12]

In contrast, programming *strategically* is about maintaining a healthy codebase along the way. There is a widespread notion of a 10x-developer,[14] but there is

[12] Starting from scratch forces you to learn all the subtleties of the domain all over again, which can be slow and painful, too.
[13] That's a thin-ice situation, though, as big projects often start small.
[14] A developer that is supposedly as productive as 10 typical developers combined.

also a common misconception about what makes them 10x more productive than others. Realistically, you can hardly imagine one professional well-motivated programmer doing an order of magnitude more than another professional well-motivated programmer. Instead, what makes the difference is that the 10x-developer empowers others so that nine additional developers' worth of work gets done. And that's exactly what strategic programming is—maintaining a healthy codebase to enable others to work with that code productively.

This challenges another common misconception that only feature work is product focused, while the same cannot be said for the busy work of keeping the codebase healthy. In fact, both types of work, if done properly, are product focused. As an engineer, there is not much you can do to find the product–market fit or develop the right marketing strategy, but there are two things *only you* can do to help your company succeed and make your users happy:

- **Short-term productivity:** Deliver high-quality software right now.
- **Long-term productivity:** Write healthy code to deliver high-quality software a year from now.

Figure 1.3. *-term productivity spectrum.

Every well-motivated software engineer falls somewhere on the *-term productivity spectrum (see Figure 1.3). Tactical programmers, instead of finding the right balance, focus on short-term productivity, and theorists and idealists focus on the long-term. It's not to say that you should never do that. Sometimes you may deliberately choose to focus on short-term goals for weeks or even months, for users' sake. And sometimes you may choose to focus solely

on long-term goals, again for users' sake. But that should definitely not be your default mode as a developer. Strategic programmers, programmers who truly care about the product, constantly balance between short- and long-term productivity. They care about their users being happy both today and in a year from now.

1.4. Embracing simplicity: The hallmark of software engineering

When I was a student, one of our university teachers began his lectures this way: "You may think you already know how to write software. But you don't. Some of you know how to solve extremely complex coding problems and write tons of working code in five hours,[15] but that doesn't make you a software engineer." I still remember the silence in the auditorium after such an intro. It was my third university year. Some of the students in the room were ACM ICPC[16] champions already, and most of us had been coding for five to ten years at that point. Some even had a little bit of experience in big tech or an offer at hand. But he was right—we were not software engineers yet. As Brian Randell perfectly defined software engineering, it is "the multi-person development of multi-version programs" [Parnas11]. While programming happens in a simple, purely technological world, software engineering happens in a world with two additional dimensions: time and people. To better feel the weight of this change, just imagine a creature that is used to living in a 1D space. It may be an expert in living on a line, but if in the blink of an eye it's transferred into the 3D world we're all familiar with, it would be pretty helpless, right? It's not to say that all of its previous knowledge and skills would be in vain, but they would definitely have a very limited applicability in the world with the two additional dimensions. That's programmers on their first encounter with software engineering.

[15] Referring to ACM ICPC (International Collegiate Programming Contest) format.
[16] `https://icpc.global/`

> While programming happens in a simple, purely technological world, software engineering happens in a world with two additional dimensions: time and people.

One of the biggest differences between programming and software engineering are drastically different consequences of code complexity. That's where the fun definition of software engineering famous at Google is coming from: "It's programming if 'clever'[17] is a compliment, but it's software engineering if 'clever' is an accusation."[18] Simplicity plays such a huge role in the world of software engineering precisely because of the two new dimensions: time and people. An average program needs to be supported and improved over a long period of time, often by many different people. What looks understandable for an author today may confuse them six months from now. And what about others? To make things worse, what if the author leaves the company? A piece of complexity that may be okay and even cool in the world of programming is likely to be destructive and unacceptable in the world of software engineering.

Thus, you can say that attitude toward complexity distinguishes software engineers from mere programmers. It is possible to be a senior programmer and a junior software engineer at the same time. As Martin Fowler puts it, "Any fool can write code that a computer can understand. Good programmers write code that humans can understand" [Fowler99]. By the way, Fowler says "good programmers," not "good software engineers," and henceforward in this book, I'll be using these terms interchangeably as well. Just keep in mind that, strictly speaking, we won't discuss programming in this book—it's 100% about software engineering.

---

[17] Referring to "clever" complicated solutions.

[18] Titus Winters, *Software Engineering at Google: Lessons Learned from Programming Over Time* [WMW20].

> A piece of complexity that may be okay and even cool in the world of programming is likely to be destructive and unacceptable in the world of software engineering.

If a program's lifetime is several days, and there are just a couple of you working on it, you're doing programming, and you may be totally fine with a purely tactical approach. But any long-running project by definition belongs to the software engineering space and requires a more strategic approach of balancing between short- and long-term productivity, with code health definitely being one of your top priorities. That's what it means to be a software engineer, not merely a programmer.

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.