O'REILLY®

Fundamentals of Software Engineering

From Coder to Engineer

Free Chapter

Nathaniel Schutta
& Dan Vega

Fundamentals of Software Engineering

From Coder to Engineer

This excerpt contains Chapter 1. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Nathaniel Schutta and Dan Vega

O'REILLY®

Fundamentals of Software Engineering
by Nathaniel Schutta and Dan Vega

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Louise Corrigan
Development Editor: Rita Fernando
Production Editor: Aleeya Rahman
Copyeditor: Charles Roumeliotis
Proofreader: Sharon Wilkey
Indexer: Krsta Technology Solutions
Cover Designer: Susan Thompson
Cover Illustrator: José Marzan Jr.
Interior Designer: David Futato
Interior Illustrator: Kate Dullea

November 2025: First Edition

Revision History for the First Edition
2025-10-30: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098143237* for release details.

978-1-098-14323-7

[LSI]

Table of Contents

CHAPTER 1
Programmer to Engineer

> *Foundational skills, always tedious to learn, seem to be obsolete. And they might be, if there was a shortcut to being an expert. But the path to expertise requires a grounding in facts.*
>
> —Ethan Mollick, from *Co-Intelligence* (Portfolio, 2024)

Being a software engineer requires a vast array of skills across a variety of areas. Understanding what your customer is actually asking for. Translating those needs into maintainable code. Writing tests to ensure that the software does what you think it should do. Creating user interfaces that work. Architecture. Working with data. Getting code to production. If you want to grow your career as a software practitioner, you must focus on more than just writing code. You must embrace the entire craft of engineering software.

To get from programmer to engineer, you need to master the fundamental skills across the software development lifecycle (SDLC), work smart, acknowledge the things you don't know, and figure out how to close the gaps on those things. In this chapter, you'll get some tips and advice to get you started. You'll learn about the various paths to becoming an engineer as well as the key knowledge those approaches often omit. Ultimately, this chapter will give you hard-earned advice that will help you on your journey, smoothing the road from programmer to engineer.

But first, let's talk about what it means to be a software engineer.

An Engineer by Any Other Name

Software is filled with overloaded (often repurposed) terms, leaving ample room for ambiguity. Ask five software professionals to define a word, and you'll likely get at least five answers. Originally, "computers" referred to humans performing calculations. Now, you'll hear terms like programmer, coder, developer, and software engineer. Are they synonyms?

No, they're not. *Programmer* or *coder* often implies someone focused on the singular task of generating, well, code.[1] In general, they may be highly skilled in a specific language and/or framework but may not understand the full SDLC. While many begin their career as a programmer or coder, the path to promotions requires you to move beyond simply fixing bugs or implementing random features.

Developers typically have a better understanding of the larger picture of delivering software. They usually know a few languages and frameworks and have experience in multiple business domains. Often more seasoned than a programmer or coder, they've started to explore more of the SDLC and may serve as mentors or pairs for less experienced team members.

But moving into the realm of software engineering requires you bring *engineering discipline* to the entire SDLC. As a *software engineer*, it is a given that you are proficient in writing code, but you're expected to think about scalability, reliability, efficiency, and security. You've moved beyond the cursory understanding of algorithms and design. You know not only the rules but also when they should be bent and even when they must be broken. Software engineers are tasked with the most complex and critical systems in production today.

Boot camps and universities typically focus on the mechanical aspects of writing code, creating people well-versed in programming. They create developers or programmers. That is the starting point for a career in software. To excel, to get promoted, to work on the most interesting projects, you must move beyond merely writing code.

There is more to software than creating syntactically correct programs. The body of knowledge required to be a successful software engineer today extends beyond learning a programming language; you must also be well-versed in the full lifecycle of a software product. This book aims to be your guide on that journey, to show you the things you may not know you don't know.

Fundamentals Matter

Fundamentals matter. Professional athletes spend most of their time working on the things they learned when they first started playing their sport. Golfers focus on stance, grip, and alignment. Basketball players focus on layups, passing, and free throws. They don't spend nearly as much time on the things that you see on the highlight reels. While the fundamentals may not be as exciting, without a solid foundation, they'd never have reached the pinnacle of their profession.

1 Incidentally, these are the practitioners most likely to be replaced by agentic coding systems.

Like every profession, software also has essential underlying principles. Take time to master them. You will spend the majority of your career working with existing code; learning how to read code someone else wrote while quickly understanding a new codebase is vital to your success. Unfamiliar codebases can be very intimidating; however, you need to be comfortable with ambiguity. Even after many months, you won't understand every nuance of a codebase, and you don't need to comprehend everything before you make changes to the code.

While you probably know the basics of writing code, you may not be as familiar with how to write code that simplifies things for the person who comes after you. Technology doesn't stand still either; you must be able to learn new things. Many programmers obsess about the latest technology or the newest language feature while ignoring the evergreen soft skills that will help them with that next promotion. Fundamentals may not generate as much buzz as the newest fad, but they're the difference between career stagnation and opportunities with greater scope and responsibilities.

The Many Paths to Becoming a Software Engineer

There are a number of paths to becoming a software engineer. Some people get an undergraduate computer science degree, while others attend boot camps, and still others are self-taught. Many have degrees in related fields like electrical engineering and then migrate to writing code. Honestly, it doesn't matter how you learned to build software.

Some people may argue their path is "better," or "faster," or "cheaper." Every approach has pros and cons. Teaching yourself doesn't require you to pay tuition fees and, depending on your aptitude, could take a matter of weeks instead of years. The internet has no shortage of videos and tutorials; however it can be daunting to navigate and requires you to be highly self-motivated. Being self-taught can also involve a fair amount of trial and error as you work your way through material of differing quality as well.

Fundamentally, the goals of boot camps and degree programs are different. Undergraduate computer science programs are designed to prepare you to enter a graduate school computer science program. They focus on algorithms, language design, compiler theory, operating systems, and related topics. However, they don't necessarily prepare you for working day in and day out on real-world applications. The projects are small and rarely span more than a few months and are nearly always starting from scratch, free of the baggage of existing applications.

Boot camps cram very specific information in a short timeframe, often a matter of weeks. They tend to focus more on the language of the day, frameworks, etc. One could argue this approach is more practical, but you could also claim the knowledge is more transitory in nature as languages and frameworks' popularity ebbs and flows.

Boot camps are sometimes housed at universities and may even be taught by the same people who teach undergraduate computer science courses, so there may not even be a difference when it comes to the teacher!

Each of these approaches involves different time commitments as well. Boot camps are often in the neighborhood of 600 hours over several weeks, and you have near constant access to someone who can get you through a challenging spot. An associate's degree is typically 850 hours spread out over one or two years, with a bachelor's program clocking in at around 1,400 hours over four or five years.

Some claim an undergraduate degree puts you at an initial advantage, though it tends to even out over time. Of course, traditional computer science programs as you know them today are relatively new, but it wasn't that long ago that most programmers were mathematicians by training.

Early computer science degree programs were often very heavily weighted toward math, with some essentially being math degrees with a little bit of computer science thrown in. In fact, many of the professors had degrees in mathematics. To this day, some people still think math aptitude is a requirement to be a successful software engineer. It isn't. Coding is about communication, and most of your job is communicating with other developers, not the computer. Language aptitude is far more important than mathematical skill. Unsurprisingly, writing code is about language, not numbers, something you will read about in Chapter 2.

Ultimately, success in the software industry is about problem-solving, tinkering, and creativity. If you have the mindset, you have it. Period. It doesn't matter how you developed your skills, and you don't need to apologize for your path.

What You Were Taught Versus What You Need to Know

Regardless of your learning path, you were taught how to write code. You learned a language or two, you learned about `foo` and `bar` and other generic variable names. You learned a bit about debugging, efficient algorithm design, and other related topics. However, a vast number of important things aren't covered in typical course curricula or mentioned in self-led learning. Why? Usually, because of a lack of time, space, or the assumption that the knowledge is a given.[2]

What these courses and learning materials leave out can critically impact your long-term success. These missed subjects include how to work with others, how to read

2 Like how to use a debugger or familiarity with modern editors.

code, how to write *readable* code, and how to work effectively with legacy code. We'll cover these topics, and more, throughout the rest of the book.

Most developers learn on greenfield projects, those that begin with a blank slate and none of the technical debt of existing codebases. Given the time constraints of semesters and boot-camp curriculum, in most educational situations, you're starting from scratch to practice a specific skill or reinforce a concept. The projects are often very small and isolated, and you often code alone. While that works in a teaching environment, it isn't representative of the real world.

As a practicing software engineer, you will spend most of your time working with existing, legacy, or heritage applications and all the baggage of decisions made before your time.[3] And when you have the luxury of a new codebase, even that will accumulate technical debt in the months ahead. Software projects today consist of hundreds of thousands or millions of lines of code by teams of people across the globe. You will work with people constantly, be it with your teammates or your customers. Your code won't just exist in a repository for someone to grade; it will be in production, where real people will rely on it to function as designed, delivering business value. To put it mildly, your first exposure to the reality of software can be a shock.

In school, most projects are short—a few months, maybe less. However, in the real world, projects don't ever really end, though they might be abandoned someday, like art. Code spends most of its life in the maintenance phase, which is another way of saying software is full of products with ongoing investments as opposed to discrete projects that have a well-defined start and end. So long as a software product continues to deliver business value, there will be continued investment.

Embrace the Lazy Programmer Ethos

The lazy programmer ethos isn't about spending your day watching cat videos on your favorite social media site, and it's not a methodology to shirk your work. The *lazy programmer ethos* is a philosophy focused on efficiency. Many new software engineers instinctively rush to solutioning; they immediately start writing code without giving themselves a beat to consider the problem domain. Being strategically lazy gives you time to think, which ultimately helps you be more productive and create better solutions.

> *Take the time it takes so it takes less time.*
>
> —Pat Parelli, renowned horsemanship trainer (attributed)

3 Though it always pays to ask why something is the way it is, don't be surprised when your colleagues' guess is as good as yours.

Beware the brute-force approach. You may start there, but you should iterate on your design. It can be helpful to understand things like big O notation.[4] For example, you may have coded a bubble sort routine, but also learned it is not the most efficient sorting mechanism. As an engineer, you should consider the best case, worst case, and average case, choosing appropriately from there.

Odds are you're not, in fact, the first person to solve a particular problem. Spend some time searching for an existing solution or library. In many cases, 10 minutes of searching could save you days of work. If it feels like there should be an easier way, there probably is one. We'll talk more about this in of Chapter 3.

The Capitalization Assignment

Nate here. Early in my career, I was given a very simple task: capitalize the billing addresses in a database to ensure consistency for the post office when bills were printed and mailed. My tech lead walked me through exactly what to do and where to do it, showing me the routine to modify, as well as the function (written by someone long gone) I should call to capitalize the billing address. Exploring the capitalization function, it struck me as overly complicated. In my mind, it should've taken one argument: the string to capitalize. But this function didn't have one argument. It had more than a dozen. It would take me hours to understand this homegrown thing.

It seemed to me that this was the kind of problem that the programming language should solve for us. I did a little searching and, sure enough, the language had a built-in capitalization function, which I used. Fast-forward to the following week to our team meeting: as we're going around the room discussing what we're working on, I mentioned the capitalization assignment. One of my colleagues looked in my direction and said, "Oh, did you use Joe's capitalization function?" I said no.

Every head in that room turned. It was a record scratch moment. Even though I was the least experienced developer in that room, no one else knew the language had a baked-in capitalization function. Maybe the language feature was added after the previous developer wrote the function; I don't know. The point is, if you think there should be an easier way, take time to look for it. Even a few hours investigating options and alternative solutions could save you days or weeks of effort.

4 In a nutshell, big O notation describes how some algorithms are more efficient than others.

The Value of a Fresh Set of Eyes

> *The most dangerous phrase in the language is, "We've always done it this way."*
>
> —Grace Hopper, computer pioneer and naval officer

Although you may not have as many years and lines of code under your belt as your more senior colleagues, remember that you have a valuable trait as a new developer: a fresh set of eyes. New people—whether to the codebase, the team, or the software industry—often see challenges with an unbiased perspective, unencumbered by historical decisions that have outlived their usefulness. This fresh outlook can lead to innovative solutions, identify inefficiencies, or challenge long-standing assumptions. Newcomers can also bring enthusiasm to a flagging project, leading to renewed energy and engagement on the team.

That's How We've Always Done It

Nate here. Years ago, I helped a client with a performance issue regarding how an application processed widgets in an overnight batch job. When the company started, it had only a few customers with a few boxes of widgets, which were easily completed overnight. As my client became more successful, they had more customers, and those customers had more widgets. By the time I worked with them, their overnight batch run wasn't finishing overnight anymore.

I sat down with them and reviewed the architecture. After a couple of hours, I asked a simple question: "Does the processing of widget A have anything to do with the processing of widget B?" No, there is some aggregation done at the end, but the individual widgets are independent. I made a simple proposal: could you just process those widgets in parallel? Deploy a set of workers that grab the next widget, process them, and put the results in a queue to be aggregated.

There was a pregnant pause. Yes. Yes, that would work. While I certainly wanted my boss to think I had some special insight, I really didn't. I just wasn't bound by the most dangerous phrase you can utter in an organization: "That's how we've always done it." Just because we've always done it that way doesn't mean it's the right way, and it certainly doesn't mean it's the best way. The approach someone took years ago could have been an accident or just the most expedient option. Chances are, there's a better way. Don't be afraid to ask questions or to look for another approach.

Never underestimate the value of a different perspective and don't be afraid to ask why things are the way they are. If you encounter something that seems odd, out of place, or just doesn't make sense, ask a teammate to explain it. In some cases, an unconventional approach will have valid reasons, but more often than not, people have just copied what was there when they arrived. As long as your questions come

from a desire to learn and not as an attempt to embarrass developers past (and present), most software engineers will engage positively with the discussion.

Don't Solution Too Quickly

When you encounter a problem, resist the urge to jump to a solution too quickly. Instead, dedicate time to understand the root cause of the problem. Give yourself some space to think. We don't get enough opportunities to just think uninterrupted. Odds are your work calendar is full of meetings, and your corporate messaging tool interrupts you multiple times a day. Between a daily standup meeting, your product owner, your manager, your team lead, and your architect, you're probably giving a status update every couple of hours. Your to-do list isn't shrinking, and the testers found another raft of tricky bugs. Despite all of the pressure on your time, it is vital that you take time to think about solutions and not jump to a quick fix.

Although quick fixes resolve your immediate problems, they can cause more issues in the long run. Jumping to a stopgap solution often creates more challenges. To address this, there has been a movement to test for problems earlier in the development cycle—in other words, "shifting left" or earlier on the project timeline. Bugs found in production are by far the most expensive to fix. Customers have been impacted, which could cost your company money. Data might be corrupted, which may be difficult to reconcile. Fixing the underlying issue often spans multiple routines requiring complicated refactoring and likely coordination with other teams.

Shifting left isn't just about fixing problems; it's ultimately about moving everything of value earlier in the process. Pair programming moves code reviews from something that occurred days (or weeks) after the work was finished to a real-time activity. Making testing an ongoing proposition simplifies solving problems by identifying them closer to when they are created. Continuous integration eases the pain of conflicting code changes by merging early and often and, when paired with continuous delivery, ensures that any variability in environments is discovered before it impacts a customer. Building security into the entire development process reduces the likelihood of catastrophic vulnerabilities.

It is much cheaper to find those issues in testing. When bugs are found closer to when the code was written, the work is top of mind for the development team, saving them time deciphering the underlying code. Test datastores can be restored with minimal fuss. While the issue may be complex to solve, there may be less time pressure to fix it fast since no customers are impacted. Thus was born quality assurance (QA) testing and QA engineers. It turns out it's even cheaper if you find issues during development, which partially undergirds the drive to test-driven development (see Chapter 5).

But what if you never wrote the bug in the first place? What if it was solved in design?

> *Most of the big problems we have with software are problems of misconception. We do not have a good idea of what we are doing before we do it. And then, go, go, go, go and we do everything.*
>
> —Rich Hickey, Clojure Conj 2010 keynote

Problems of misconception are endemic in software, and misunderstandings are a common challenge in software development. Terms like "client" and "customer" might seem interchangeable, until your product owner says that approach will work for a client but not a customer. They are similar but distinct things, and the nuance matters. No amount of testing, no type system will ever find a problem of misconception. A technical solution alone cannot resolve these problems; they stem from a fundamental difference in understanding.[5]

Rather than work around a problem created by a misunderstanding, try to avoid it in the first place. How? Focus on solving problems. Do you know what problem your application solves? Do you understand the context and the constraints? What are the related problems? Take the time to think through the problem space.

Developers often tend toward overengineering a solution. While not always an issue, there are many ways to solve a problem. Never lose sight of the problem you're trying to solve and don't be too quick to jump to solutions. What is your customer asking for; what is their actual need? With the scars of experience, you never want to paint yourself into a corner, and it is easy to imagine everything you *might* need. But you also don't want to build a bunch of code you must throw away later. Or worse, keep around unused.

It can be challenging to achieve, but drive to the "why," or the root cause behind the customer's request (see "The Five Whys" sidebar). Ask questions. Talk to your customers; talk to your product owners. In many cases, a customer will ask for one thing but need something else because they can't see it. They are stuck with "That's how we've always done it." Sometimes your customers will come to you with a very specific solution—for example, "We need to be able to export this table into a spreadsheet." It is tempting to immediately build the feature, but a seasoned engineer asks what the underlying need is first. There is often a disconnect between what your customers ask for and what solves their business problem.

5 A glossary can be one of the most impactful, and invaluable, project artifacts. Consider creating one if your project doesn't have one.

The Five Whys

Originating as part of the Toyota Production System, the Five Whys is a problem-solving method that strives to discover the root cause of an issue by repeating the question "Why?" five times, each time directing the next "why" to address the previous answer to "Why?" Doing so allows you to dig beneath the surface-level symptom or issue to unearth the fundamental problem. For example:

- Problem: The robot stopped working.
  - — Why? The circuit overloaded, causing a fuse to blow.
  - — Why? There was insufficient lubrication on the bearings, so they locked up.
  - — Why? The oil pump did not circulate enough oil.
  - — Why? The pump intake is clogged with metal shavings.
  - — Why? There is no filter on the pump.

Let's walk through a software example. Nate here. Years ago, I ran into a web page where all the closing tags were on the following line. While the interpreter didn't care, it was visually jarring to see closing tags starting every line. I asked a colleague *why* someone had used this approach. He responded because of a bug in an old version of the web server we ran on. OK, *why* do we still support that version? Oh, we don't. OK, *why* hasn't that bug been fixed? Oh, it has. OK, then *why* am I still seeing this pattern? Well. it was probably an old web page. I told him it had been created a few days prior. At this point, we agreed we should inform developers that they should stop following this pattern, and we added a cleanup task to rework the pages that still used the outdated approach.

In many cases, the primary source of requirements will be a super user of the old system. While their knowledge can be invaluable, it is also colored by what they currently have.[6] In some cases, they won't understand the underlying *why* of a given feature and may assume it is the only way a given problem can be solved. In other words, they may (consciously or not) lead you to a copy of the old system in a new technology.

It is vital to have a broad perspective. Pick the right tool for the job to solve the problem the customer actually has. Many engineers are overly optimistic. Be realistic; be ruthlessly pragmatic. Scalability matters, however; not every application can be a third of internet traffic.

6 Which explains why so many applications seem to devolve to email and spreadsheets.

Prediction is very difficult, especially if it's about the future.

—Danish proverb

Apply the Golden Rule to Software

Odds are, early in your educational career, you learned about the Golden Rule, which says you should treat others the way you would like to be treated.[7] This principle is embraced across continents and dates back thousands of years as a basic tenet of how to live in civilized society. But it turns out, it isn't just a pithy ethical principle. You can (and should!) apply the same standard to your code.

Think about the developer who will follow in your footsteps. And again, that developer may in fact be you![8] What would *your* future self like to see in the codebase? What would make life better for the next developer? Code is ultimately a communication mechanism. Yes, it is a way to instruct the computer to perform some task, but that pales in importance to its ability to speak to other developers. Write code that is meant to be read. Optimize for the human, not the compiler.[9]

If you want to stand apart from the crowd, follow the Golden Rule. Write clean code so those who follow you can understand and maintain it. Update the documentation to save others from spinning their wheels following an outdated approach. Create clear diagrams that clarify instead of confuse. Make time to answer questions and help others. Put it all together, and you'll be in demand; people will want you on their teams because you make everyone around you better.

Wrapping Up

A career as a software engineer is about far more than staring at a screen and spewing out code. Being more than just a coder or programmer requires you to do more than just produce syntactically correct code; it demands a well-rounded skill set. It doesn't matter where or how you learned the trade, be it through a computer science undergraduate program, a boot camp, or via materials you found online. What matters is how you package those skills together.

What you were taught isn't all you need to know to ensure a successful career. This chapter shared advice on how to avoid some of the more common pitfalls, such as solutioning too quickly, overengineering solutions, and the dangers of brute-force approaches. The earlier you learn these lessons, the better.

---

7 Or more commonly, "Do unto others as you would have them do unto you."

8 Again, every developer has stared at some code wondering what fool wrote this only to slowly realize they actually were the fool that wrote said code.

9 With the exception of certain, very specialized programming examples. You'll know it when you encounter it.

Putting It into Practice

Finding mentors can be incredibly valuable early in your career. Look around your organization and politely ask a more senior software engineer if you could set up some time to chat over a cup of coffee, your treat. Talk to them about their experiences, what they've learned, what they wish they had learned earlier, what has made them more valuable on a team, what they think you should learn, and so on. You can also reach out to event speakers and authors. Many of them are happy to help those who seek them out.

The next time you're assigned a bug or a new feature, pause before immediately jumping to the code. Ensure that you understand the problem you're trying to solve. Spend 30 minutes researching. Is there a ready-made solution you could leverage? Try sketching out or modeling concepts in your problem domain along with possible solutions. Consider discussing your options with an experienced coworker.

Look through the table of contents of this book. What topic areas do you feel you're strongest at? What could you improve on? Pick a topic area you'd like to learn more about and make a plan to do so. Act on the "Putting It into Practice" sections and read the additional resources of those chapters.

Your framework, library, or cloud provider may abstract away certain things from your daily workload. For example, leveraging Spring Data will save you from writing countless SQL statements. While you absolutely should embrace the productivity of these coarser-grained abstractions, you should understand the layer of abstraction *beneath* the one you are using. Take time to pop the hood and look around underneath; the knowledge you gain will help you immensely, especially when something doesn't quite work as planned.

Additional Resources

- *The Pragmatic Programmer: Your Journey to Mastery*, 20th Anniversary Edition, by Andrew Hunt and David Thomas (Addison-Wesley Professional, 2019)
- *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition, by Fred Brooks (Addison-Wesley, 1995)
- *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma et al. (Addison-Wesley, 1994)
- *Practices of an Agile Developer* by Venkat Subramaniam and Andy Hunt (The Pragmatic Bookshelf, 2006)
- *The Productive Programmer* by Neal Ford (O'Reilly Media, 2008)
- *Software Engineering at Google* by Hyrum Wright et al. (O'Reilly Media, 2020)
- *The Staff Engineer's Path* by Tanya Reilly (O'Reilly Media, 2022)

- "No Silver Bullet: Essence and Accidents of Software Engineering" by Frederick P. Brooks Jr. (The University of North Carolina at Chapel Hill Department of Computer Science, September 1986)
- *Code Complete*, 2nd Edition, by Steve McConnell (Cisco Press, 2004)

About the Authors

Nathaniel T. Schutta is a software architect and Java Champion focused on cloud computing, developer happiness, and building usable applications. A proponent of polyglot programming, Nate has written multiple books and appeared in countless videos and many podcasts. He's also a seasoned speaker who regularly presents at worldwide conferences, meetups, universities, and user groups. In addition to his day job, Nate is an adjunct professor at the University of Minnesota, where he teaches students to embrace (and evaluate) technical change. Driven to rid the world of bad presentations, he coauthored the book *Presentation Patterns* with Neal Ford and Matthew McCullough, and he also published *Thinking Architecturally* and *Responsible Microservices*, which are available from O'Reilly.

Dan Vega, a Spring developer advocate at Broadcom and Java Champion, has over 20 years of software development experience. A passionate problem-solver, he actively shares knowledge as a blogger, YouTuber, course creator, and speaker, inspiring fellow developers through continuous learning.

Colophon

The animal on the cover of *Fundamentals of Software Engineering* is a gold-lined rabbitfish (*Siganus lineatus*). Found in the western Pacific Ocean, these colorful fish thrive in lagoons and coral reef communities.

Gold-lined rabbitfish have pale blue bodies with wavy orange lines that run from their heads to their forked tails. Their bodies are laterally compressed and have a maximum length of 17 inches; the average length of these fish is approximately 9 inches. They have sharp spines on their dorsal fins that produce venom, which is used for protection against predators.

Thanks to their unique colors, gold-lined rabbitfish are able to seamlessly blend into their coral habitats. These fish are herbivores and mostly feed on algae, which they scrape off from beach rocks or coral reefs; they also consume seaweed and seagrass. Gold-lined rabbitfish play an important role in coral reef communities, as they help control algae growth and serve as a food source for larger reef fish.

Gold-lined rabbitfish are not an endangered species and have been classified as Least Concern by the International Union for Conservation of Nature. However, they do face certain threats, including overfishing, habitat loss, climate change, and the aquarium trade. All animals on O'Reilly covers are important to the world.

The cover illustration is by José Marzan Jr., based on an antique line engraving from *Lydekker's Royal Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.