Chapter 3. Writing Code

> Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
>
> Martin Fowler, British software developer, author, and international public speaker on software development

Writing code is, without a doubt, a very important part of software engineering. And while the act of coding is widely taught, the nuances of writing *good* code aren't as evenly distributed. Just because you *can* write code to solve a problem doesn't mean you *should* write code to solve a problem![1] With the advent of artificial intelligence and agentic coding tools, the job of a software engineer is evolving to one with less hands-on development work and more bug fixing, correcting, and reviewing code generated by an eager AI tool. However, to be good at reviewing code, you must be good at writing code. It may seem counterintuitive, but writing good, clean code is an invaluable skill in the world of AI.

For better or worse, developers often have strong opinions on what constitutes good code or bad code, but metrics and tooling can give you insights and guidance. Tests are some of the best documentation money can buy. Code reviews, done well, can ensure that your team doesn't rely on error-prone forms or overly clever code. Ultimately, code should be written to be read by humans.

Despite how it is often taught, programming is first and foremost a communication activity—and not just between the coder and the compiler. Don't forget, the computer understands *any* code (at least if it's syntactically correct), but that doesn't mean a human will follow what you're trying to accomplish. The best software engineers focus on writing code that can be understood by the humans reading it. This means that your code needs to be concise and well-organized. The code should clearly communicate its intent so a human can figure out what the code is supposed to do. If it's easy to read, it'll be easy to maintain, which is crucial for the long-term success of a software project.

The first thing you learn when you set out down the path of software development is how to code. Probably with "Hello World!" What is not always taught, whether in an undergrad course, boot camp, or self-led tutorial, is what makes code good, how to write good code, and when to write code to solve a problem. This chapter aims to fill that gap.

Don't Reinvent the Wheel

Before you start pounding out line after line of (undoubtedly excellent) code, take some time to see if the problem at hand has already been solved. Developers often write too much code, solving problems that others have already worked out. Before cranking out some fresh code, look around. Is there a library you could leverage? That isn't a license to add things ad hoc; don't forget to analyze those libraries' dependencies. Be sure to follow your organization's policies and procedures surrounding third-party libraries.

Double-check your primary programming language. Is there a language feature you could use? Languages evolve (see "The Capitalization Assignment"); five minutes of searching could save you hours of effort. What about your application frameworks? If you think there should be a better way, there just might be. Ask yourself, are you the first person in the universe who has ever written code like this? Odds are you are not (see "Embrace the Lazy Programmer Ethos").

Before you invest time in writing a new piece of code, it can help to ask yourself a series of questions. Christopher M. Judd teaches the following decision tree in his boot camps (modify for your language and frameworks):

- Is this being done anywhere in the current codebase?
- Does the JDK already do it?
- Are there any Spring Framework or Spring Boot projects that solve this problem?
- Does a solution exist in Google Guava?
- Does a solution exist in Apache Commons?
- Are there any other libraries in the project already that solve this problem?
- Are there any open source libraries that solve this problem?

If you answered no to all the questions, you're in the clear to write the code, making sure to use tests as you go along! If you answered yes to any of the questions, congratulations, you just saved yourself a lot of time; leverage what's available!

That said, yesterday's best practices are often tomorrow's anti-patterns, and just because something was the right thing to do five or ten years ago doesn't mean it is still the right thing to do *today*. Never be afraid to ask why, to challenge the status quo. You may find yourself mindlessly copying patterns or approaches you find or becoming complacent with code generated from AI without fully understanding *why*. Software moves pretty fast; if you don't stop and look around once in a while and deliberately adopt an open perspective, you could miss some really important things (see "The Value of a Fresh Set of Eyes"). Languages, frameworks, technologies, and techniques are constantly evolving; keeping up is par for the course as a software engineer. Ask probing questions and keep a weather eye on the horizon.

What Is Good Code?

All that aside, there won't always be a library or language feature to solve your problem. You will, of course, write code! And you may ask yourself, well, what is good code? To paraphrase Potter Stewart, good code can be hard to define, but you know it when you see it. Admittedly, good versus bad code can be a very subjective concept. After a while, you start to develop a sense for it, and you may even say code has a smell to it. For example, overly long methods are generally a bad thing, but you still need to examine the code; there may not be a simpler approach. Few developers will ever admit to writing bad code, but anyone who's ever opened an editor is guilty of some less-than-stellar software.

Metrics can provide insight into your codebase. For example, cyclomatic complexity can tell you the number of paths within your source code, with more paths indicating more complex code that would likely benefit from refactoring.

Many languages have source code analyzers like PMD, SonarQube, and JSHint, which can help prevent certain types of bugs and bad coding practices from infesting the source code. Tools like SonarQube and CodeScene can provide invaluable insight into your codebase. Odds are your organization has something. If you're not sure, ask around. If it doesn't, why not spearhead the effort to bring one into your project?

Adding Metrics to Existing Projects

In a perfect world, your project would leverage analysis tools like those mentioned from the very start of your project. Doing so allows your code to "start clean, stay clean" (assuming your colleagues fix the detected problems). However, you won't always have that luxury, and you'll have to add a linter or a source code analyzer to a project that has been in development for months or years.

It is tempting to add a tool like PMD and turn on all the rules! Do not do that. First of all, some of the rules contradict themselves; if one passes, another will fail. Second, turning on too many rules will typically result in an unmanageable number of warnings, which is counterproductive. It can demoralize a team. With hundreds or thousands of warnings, it is hard to notice when someone checks in code that adds a few more, and spending a few hours of effort to fix a couple of dozen warnings moves the graph an imperceptible couple of pixels.

Instead, have a discussion with your team and pick a few rules that you can all agree on. Finding that subset may prove challenging! Turn those rules on. That should result in a manageable number of warnings. As a team, work to fix them. Once you have done so, turn on a couple of more rules. Rinse and repeat. After a few months, you'll have a rich set of rules running against your project as well as cleaner code.

Metrics should never be mindlessly followed; some complexity just can't be avoided! You must apply common sense to any rule of thumb because, in some instances, applying the common rule may actually make things worse. However, you can take advantage of the Hawthorne effect, which says people modify their behavior when they're observed.[2] You can absolutely use that to your advantage on a project! For example, if you want more of the code to be covered by tests, prominently display the code coverage stats.

Increasing Code Coverage Through Metrics

Nate here again. Many years ago, I joined a project that had been churning out code for a few months. I was pleasantly surprised there were, in fact, tests, but I was disappointed that all but a small handful were skipped by the build because they "kept failing." Not to be deterred, I ran a code coverage report showing how many lines of code were executed after running all of the tests. Unsurprisingly, it was a single-digit number. But I posted it, and I talked about it. A lot. During most standups and retrospectives. Whenever the number ticked up, I heaped praise on my awesome teammates. When someone added tests to a particularly tricky part of the code, I had them discuss how they did it.

Slowly but surely, our code coverage number went up, and the code became less brittle. It took several months, but by the time I moved on to the next project, the code was closing in on 70% coverage. By no means were we satisfied, but we

were in a far better place. And it all started by running—and then promoting—a simple code coverage report.

Metrics can be abused or misused (see "Metrics Can Mislead"). For example, many organizations have tried to evaluate technical staff by lines of code written or deleted or modified. Use these metrics wisely. Consider external coupling or how many classes are dependent on a given class as another example. In general, if you see a high degree of coupling, you would consider that an opportunity for refactoring. What if that class is actually a facade that is essentially hiding a set of classes? You must contextualize any metric.

Just because you can measure something doesn't mean it will provide meaningful insights. Try to link metrics to project goals and focus on the direction your metrics trend. Are you getting better or worse over time? Favor short time horizons and be ready and willing to adjust and adapt.

Adding new tools to your pipeline isn't your only choice. Many IDEs will provide code-quality metrics, and there are any number of plug-ins you could add to your personal toolchain that will help you write better code. To shorten the feedback loop, many of the code analysis tools can be wired into your editor, notifying you of violations as you type instead of waiting for a build break. Your team may have a preferred configuration. Ask a colleague if you aren't sure.

Less Is More

> The goal of software design is to create chunks or slices that fit into a human mind. The software keeps growing, but the human mind maxes out, so we have to keep chunking and slicing differently if we want to keep making changes.
>
> Kent Beck

With the size of some codebases, you might think developers are paid by the character. While some problems genuinely require millions of lines to solve, in most cases you should favor smaller codebases. The less code, the less there is to load into people's brains. Many projects reach the size where it is no longer possible for one developer to understand *all* of the code, which is one of the forces that has given rise to microservices and functions as a service.

The typical big balls of mud often have dictionary-sized Getting Started guides and build processes that are measured by weeks and revolutions of the moon. It can take developers new to the project weeks or months to get productive within the code. The smaller the codebase, the less time it takes for a new developer to get their head wrapped around the code and the faster they can start contributing; see Chapter 6 for more details. The following are things to consider when trying to get your code length under control.

The Zeroth Law of Computer Science

Many of the practices software engineers espouse in an effort to tame code boil down to the zeroth law of computer science: high cohesion, low coupling. *Cohesion* is a measure of how things relate to one another. High cohesion means essentially that like things are together. The notification function that also contains print logic would be an example of low cohesion. *Coupling* refers to the amount of interdependence between modules or routines. Code with tight coupling can be difficult to modify as changes to one part of the code unexpectedly affect other,

seemingly unrelated parts of the system. Changing the notification service shouldn't break the print module.

High cohesion and low coupling tend to result in code that is more readable and simpler to maintain and evolve. Many patterns are ways of achieving high cohesion and low coupling, often at different levels of abstraction. At their best, arguably, microservices are high cohesion, low coupling applied to services.

Beware Boilerplate Code

Boilerplate code, even if it is generated by an editor or a framework, should be avoided. While you may not have to *write* it, you will still carry it around for the lifetime of the project. Classes should be short, a few pages or less.[3] Programming languages will impact your definition of *short*, as some languages are more verbose than others. In general, if you have to scroll, your class might be too long.

Favor Composition over Inheritance

Many languages allow classes to inherit from other classes, and while this feature can be very powerful and there are certainly *is a* relationships (a cat *is a* mammal) in software, it tends to be overused. Some developers use inheritance as a reuse mechanism. Reuse is a byproduct, not a rationale!

Let's look at a concrete example. Say your domain involves cars and trucks. You might create a vehicle superclass that cars and trucks both descend from that includes a combustion engine—since all cars and trucks have a combustion engine, defining it on the superclass ensures that all cars and trucks also have a combustion engine. For example, see this Ruby pseudocode:

```
class Vehicle {
    Engine engine
    Integer num_wheels
    Integer num_doors
    Function brake {}
    Function accelerate {}
 }

class Truck < Vehicle {
    Number tow_capacity
}

class ElectricVehicle < Vehicle {
    Number range
    # wait… EVs don't have engines…
}
```

But along come electric vehicles with nary a combustion engine to be found. An EV is-a vehicle, but not like those *other* vehicles. Composition is more flexible and should be favored over inheritance. That isn't to say you should *never* use inheritance, just that you should prefer composition.

Favor Short Methods

Write short methods, as in single-digit lines of code. Like a Linux or Unix command-line tool, methods should do one thing and only one thing, and they should do it well, working in concert to accomplish larger goals. Any method name that includes conjunctions (and, or, but) is a sign the method is doing too much. Method names should be clear and concise and avoid being clever. If you

are having a hard time naming a method, it might be doing too much. Try breaking it apart and see what happens. Be descriptive. Remove logic duplication, even in small amounts. Simplify, then simplify some more.

Overall, aim for smaller, more manageable codebases to improve developer understanding and productivity.

Write Code to Be Read

Think about the developer who will follow in your footsteps, the one who will need to read and maintain your code. Remember to apply the Golden Rule to your code (see "Apply the Golden Rule to Software"). How could you write the code so that you make life better for the next developer? What do you need to communicate to the next person?

There are any number of guidelines you can apply when it comes to the "correct" length of a function, from reuse to picking an often arbitrary number of lines. Martin Fowler offers sage advice: mind the separation between intent and implementation. In other words, how long does it take for you to understand what a function is doing? You should be able to read the name and understand immediately what the code does without investigating the method body itself. This principle will often lead to functions with only a few or even just a single line of code; make the *intention* clear. The class name should give you context, and the method names help you understand what the class does.

Code can be written like a newspaper article with an "inverted pyramid." Articles start with the lede then move to key facts and then on to deeper background. You, as the reader, can stop at whatever level of detail you wish—maybe just the first couple of paragraphs, maybe all the way to the end. As they say in publishing, don't bury the lede.

Code should follow the same model. The class name is almost like the headline of an article. From there you should be able to skim the method signatures to get a general understanding. If you want to explore a function or a call to another class, you are free to do so, but you should still understand the gist of the code.

Naming Things Is Hard

Every developer has stared at their editor struggling to name a variable, method, or class. And while this struggle can indicate an insufficient understanding of the problem or some overly complex code, there is a reason `foo`, `bar`, and `foobar` are such common occurrences. Coming up with meaningful names can be time-consuming. Don't rush! Again, take the time it takes so it takes less time. It is worth the effort to come up with good names. Don't be afraid to reach out to a teammate for their input. Sometimes just explaining what you're working on will be enough to inspire the perfect moniker.

Your IDE can help! Many developers work backward from the declaration, allowing the editor to make suggestions. In other words, start with new `Arrays.asList({"Red", "Green", "Blue"});`. Your IDE can use a postfix completion to suggest a variable name. Agentic coding tools can also provide some inspiration.

Don't hesitate to refactor poorly named code. Modern editors have powerful tools that make renaming a straightforward endeavor. Just make sure you aren't constantly renaming core domain concepts, because that usually indicates a problem of misconception.

It may also help to play the gibberish game. Often the first word you use to define a concept isn't the best option, but it may shape your thinking about the problem domain. When you are first working through the domain, make up a word! After you've done some additional analysis, go ahead and replace the gibberish with real words. You'll likely have come up with something that's very different, and clearer, than your first reaction. The next time you're really stuck on what to call something, throw in some nonsense words and circle back.

The Problem with Code Comments

Every programming language has *some* facility for a developer to speak, not to the compiler or runtime, but to their fellow developers via a code comment. And while it may seem like one should liberally comment their code, arguably doing so is a code smell.[4] Code comments violate the DRY principle, aka Don't Repeat Yourself. While you will most often see DRY violated with code that is copied and pasted or littered with logic duplication, comments can also be problematic. In most cases, you wrote the code and then you turned around and wrote about the code.

Code comments can be a maintenance headache as well. You modify the code, but will you take the time to update the comments too? Often the comments begin diverging from the code, adding another layer of sediment to the developers who encounter the code months or years later. Code should be written to be readable. Your time is better spent making the code simpler to read than in documenting what you did. Expressive languages definitely aid in achieving readable code, and you may want to avoid the more nuanced "magic" features of your language of choice. Good method and variable names are better than comments; if you think you need to write a comment, try renaming the method or variable. Doing so will nearly always obviate the need for a comment.

Arguably, the least useful code comment is the code change blocks that are often the first few hundred (or thousand) lines in a file. And while it can be an interesting waypoint along your archeological journey, the repetition of tracking numbers, dates, names, and paragraphs about the change adds noise to the process. Let your source code management tool do its job.

That isn't to say code should *never* have comments.[5] If you are doing something that isn't obvious, and you couldn't find a way to simplify the code, comments explaining the situation can be helpful. These comments should focus on *why* you did something instead of *what* you did. Providing context around a choice of one algorithm over another or explaining a particularly complex bit of logic can be helpful as well. That said, if code needs to be explained, rewrite it.

Comments can serve as reminders to our future selves, or as a warning that a hack works but you don't (yet) understand why it does. Some developers leave comments as warnings to future developers, as in the following example:

```
// Dear maintainer:
//
// Once you are done trying to
// "optimize" this routine,
// and have realized what a terrible
// mistake that was,
// please increment the following
// counter as a warning
// to the next person:
//
// total_hours_wasted_here = 42
```

Are these comments a good thing? It depends, and many of the examples you will hear of are likely apocryphal. That said, don't shy away from leaving behind advice for future you.

Tests as Documentation

If comments aren't an appropriate way to document code, what should you do? Write tests. Tests, especially those written in more fluent styles, are executable specifications that evolve along with the production code.[6] Documentation, whether code comments, READMEs, or specifications, tends to diverge from the code as soon as it's written. Tests written while you write code allow you to refactor freely and increase your confidence in the quality of your application. They also act as signposts for the developers who follow you. Testing is explored in more depth in Chapter 5.

Adding tests to existing code allows you to capture what you're learning about how the code works and unlocks your knowledge such that other developers can benefit from your work. As you rename a method or variable and prune some dead code, you're actively leaving the code better than you found it.

Some developers insist they need to write copious comments for those who consume their services. While these comments are less smelly than those mentioned earlier, they aren't your only option. Again, tests make for a more resilient documentation mechanism. Utilizing consumer-driven contracts allows you to convey what your service does while also giving you the confidence to iterate as necessary. As long as you haven't violated the contract, you can evolve your code freed from the worry that you might inadvertently break a downstream system. Consumers gain confidence in your services, as they have a set of tests they can execute that simulates the expected behavior of your code. They can modify their code without fear of introducing a new defect.

Consumer-driven contracts are a vital part of reliable and resilient software. Many languages and frameworks have projects you can (and should!) leverage with your applications. From Spring Cloud Contract to Pact versions for nearly every platform, you have options.

Avoid Clever Code

Software is hard, and the domains you work in are complex. But not all complexity is equal. In his widely cited No Silver Bullet essay, and *essential*Fred Brooks makes the distinction between accidental and essential complexity. In short, *essential complexity* is inherent in software, from the nuance of the business rules, to communicating with your team, to the ever-changing nature of a codebase. There is nothing you can do to remove this complexity from software; it comes with the paycheck. On the other hand, *accidental complexity* are ways developers make things harder than they have to be, from noisy technology to heavyweight tooling. When in doubt, keep it simple.

Software is not immune from the proverbial snake oil. Many companies attempt to sell products or processes that will revolutionize software delivery. It pays to be skeptical. There are opportunities for improvement of course, but the scientific method reminds us that extraordinary claims require extraordinary evidence. You should be vigilant about removing accidental complexity wherever possible. See Chapter 6 for a more detailed discussion.

Languages and frameworks often have error-prone forms. For example, take a look at the following Java code. Can you spot the problem?[7] In Java, brackets are *technically* optional on a single statement `if` block. Now, some developers might argue in favor of omitting the brackets, as it is less verbose and the (essentially) empty lines are a waste of vertical space. But what happens when another developer adds a second statement? Will they remember to add the brackets, or will they be seduced by the code indent?

```
if (condition)
  doFoo();
  doBar();
```

Avoid error-prone forms in your toolchain of choice. Just because you clearly understand it does not guarantee that the information is widely distributed across your team. Don't be afraid to update (or establish) coding standards to cover these cases. There are a number of static analysis tools you can add to your deployment pipeline to keep you and your team from inadvertently introducing these types of problems. Take advantage of them.

Code Reviews

Code reviews can vary from asking a colleague for feedback on a method all the way to hours-long walk-throughs with several developers. Regardless of the specific implementation details, code reviews are an excellent way to learn, share experience, and socialize knowledge. More eyes on code is a good thing and part of the reason some organizations use pair programming.

Whether formal or not, certain practices can improve your code review. First and foremost, don't be snarky. Avoid sarcasm. Asking for feedback can be very stressful for people, and many people take criticism personally. How you share your comments is critical. Be empathetic to your teammate. While it may be tempting to use a code review as an opportunity to drop some esoteric bit of trivia on your team, the goal is to improve the code, not exhibit your technical expertise.

> The only way to make something great is to recognize that it might not be great yet. Your goal is to find the best solution, not to measure your personal self-worth by it.
>
> Jonas Downey, software designer, developer, and writer

Focus your attention on the most important things. While style points matter, your effort is better spent lower down on the Code Review Pyramid. You can (and should) automate formatting and style-related issues;[8] let a computer handle those. Your time and effort should be spent on the things computers can't detect for you. Are method and variable names clear and concise? Is the code readable? Is there duplication? Does the code have the proper logging, tracing, and metrics? Are interfaces consistent with the rest of the code? Did the developer use any error-prone forms? Is the model correct? Did they choose the wrong abstraction?

Avoid the Checkbox Code Review

In some instances, code reviews are little more than a checkbox in the source code management system. Some organizations require all code to be reviewed before it is merged into the mainline. That goal may be admirable, but more often than not, it results in little more than one developer asking another to "review" the code, which usually means checking the box for their compatriot. Even though it comes

from a good intention, such reviews defeat the purpose. Working in small batches makes reviews simpler and more effective.

Some organizations use pull requests (PRs) as a way of ensuring code quality. While they can be more comfortable than spending the afternoon on a video call arguing over code, PRs aren't always conducive to building team cohesion and trust. Some developers use PRs as an opportunity to nitpick, start a turf battle, or reignite a previously closed issue. PRs are also vulnerable to the LFTM (looks fine to me) response, which may be acceptable in some cases but falls victim to the previously mentioned checkbox review.

Text-based comments also lack the tone and body language of face-to-face conversations. You may not *intend* a comment to come across in an acidic or biting way, but it may be taken as such by the person on the other end of the request. Don't be surprised if some of your teammates, especially those with less experience, dread a PR. Once again, practice empathy. Ask yourself how you'd like to be treated and act accordingly. Senior staff should lead by example.

It Is Hard to Be Criticized

Developers often invest a lot of themselves into their work, so sometimes it is hard not to take feedback personally. Code reviews are not an opportunity to embarrass someone because they didn't know about some new language feature or didn't immediately see a simpler way to solve a problem. No one is perfect; everyone makes mistakes. Code reviews are about building better applications and are about the code, not the coder. Don't get personal in a code review. Be humble and ask helpful questions. Critiques are more digestible when they are sandwiched by compliments, so be sure to point out the *good* things too.

Share your experiences. Personal stories carry immense weight and diffuse people's natural resistance to change. Offer assistance with things you've encountered on previous projects. Be careful with blanket proclamations. Make sure you have all the details before you pronounce something won't work, as you may be missing a key bit of context. Is there some background you don't have? Perhaps there are constraints you aren't aware of. Stick to the code.

> Every single one of us is doing the absolute best we can given our state of consciousness.
>
> Deepak Chopra, Indian-American author and alternative medicine advocate

If something in someone's code concerns you, don't be afraid to talk to the developer directly. People can be very defensive, especially in group situations. A quick one-on-one discussion could be the answer. Don't ambush a teammate; no one wins in those interactions. Remember, reviews are a chance to learn, an opportunity to teach.

Fostering Trust

If you shouldn't do checkboxes or LFTM reviews, what should you do? Regardless of your code review process, don't lose sight of its purpose. You should be sharing experiences, learning, and growing as a team while avoiding problematic practices like confusing idioms or deprecated approaches. Encourage your teammates to ask questions and provide constructive feedback. Code reviews should foster collective code ownership and foster trust among the team.

Promoting a bug of the week or taking the time to share something you ran into can be incredibly powerful.

A regular meeting where people are encouraged to talk about an interesting defect they solved is an invaluable learning technique. You might assume everyone knows what to do when they encounter a particular issue, but discussing those situations with your teammates spreads knowledge. The simple act of taking time during Friday's standup to discuss something interesting people experienced during the week can pay dividends.

It should go without saying, but treat your teammates with respect. Be kind, do what's right, do what works. Don't be afraid to take a moment to review your approach and ask if there might be a better way. Whether you're following an Agile development methodology or not, you should adapt and adjust on a regular basis. If something isn't working, change it!

Learning New Languages

If you want to get better at writing code, you need to, well, write code. But you can accelerate those skills by learning new programming languages. Think about learning a foreign language. For months, even years, you will translate that language into your native tongue. Eventually, you will think, even dream, in your new language. But it takes time. Programming languages are no different. Well, they have very demanding grammar rules and far fewer keywords.

The key is immersion. If you really want to learn French, moving to Paris will accelerate your progress. Picking up a new programming language is much the same. Build an app in the new language that solves a problem that's plagued you at work or at home. Follow the community on social media. Listen to the related podcasts or watch the videos from the latest conference focused on the language. Better yet, attend the local user group or go to the next event about the topic.

Developers tend to get very attached to their first language. Be careful here. Programming languages are just tools. Just as a hammer is a better tool for pounding in a nail than a screwdriver, some languages are better fits for certain problems than others. For example, many embedded systems are written in C since it is highly portable and reliable and can be heavily optimized for specific platforms. Make it a habit to look at, explore, and learn other languages. Some influential people suggest learning a new language every year or two.

Programming Is Fundamentally About Communication

Before computer science departments started springing up at universities around the world, programming often lived in the math department, and many assumed mathematical aptitude was a prerequisite for success in software. In some instances, universities still use math exams to recruit students! Despite this assumption, research shows that *language* aptitude is a far better predictor of how quickly someone picks up a new programming language than skill in mathematics. While certain domains may be very math heavy, the art of programming isn't.

Programming is first and foremost a communication activity—and not between the coder and the compiler. Don't forget, the computer understands *any* code (at least if it's syntactically correct), but that doesn't mean a human will follow what

you're trying to accomplish. The best software engineers focus on the person reading the code. Good writers (of any kind) always keep the audience front of mind.

Learning a new language takes time. How do you justify the investment it takes learning something you might not use daily at work? Learning a new language will change how you code even if you don't get to use the new tool day in and day out. When you seek out a new language challenge, try to pick something that is different from what you use at work. If you're an experienced Java developer, look beyond other C-like languages such as C# (not that there's anything wrong with learning C#, mind you) toward different paradigms. Consider instead a dynamic language like Ruby or a functional language like Haskell. Trust us, even just a cursory examination of a language outside your normal neighborhood will fundamentally alter your approach to programming. You may come to appreciate your regular language more, or you may find yourself writing code in a different way. Take time to learn new things.

Learning new languages gets easier over time. The more languages you know, the more you have to compare to. Think back to the first language you learned—you were starting at zero. By the third, fourth, fifth language, you start to see how one idiom is just like another one from Java, or how some structure was borrowed from Ruby.

Early in your career, you should focus your attention on going as deep as you can on the languages and frameworks you use daily. However, don't neglect exploring other options. Doing so will not only invigorate you but also make you a better developer.

Wrapping Up

Developers write code; it is part and parcel of the job. Avoiding error-prone forms and overly clever code can be the difference between a codebase that's a pleasure to work on and one that developers avoid like the plague. From code reviews to analysis tools, there are many ways to help you write better code. Favor writing tests over copious comments. When critiquing code, be empathetic. Never forget, code should be written to be read by humans; adhering to that principle goes a long way toward ensuring you write code others will stamp with the elusive "good" label!

Writing good code is an ongoing learning process. Be open to learning new skills and improving the ones you've already developed. Staying up-to-date on best practices in software engineering is a key to your long-term success in the field.

Putting It into Practice

There are no shortcuts; if you want to improve as a developer, you will need to write code! As the old joke goes: A pedestrian on 57th Street sees a musician getting out of a cab and asks, "How do you get to Carnegie Hall?" Without pause, the artist replies wearily, "Practice." If you want to be a better developer, you need to practice.

Consider adding some effortful study to your routine. Periodically, say every other month or so, block out a couple of hours to tackle a code kata. In martial arts, katas are a series of blocks, kicks, and punches that students study and repeat countless times. Code katas bring this idea to software, providing simple problems

that give you a chance to practice your craft and to work on things you may not encounter in your day-to-day coding life.

You can leverage code katas in any number of ways. You could pick one and solve it in two or three programming languages. You could pair with a friend or colleague to work through a kata. There are multiple ways to solve a given kata; challenge yourself to solve the same kata in two or three ways. Perform a code review on a kata you solved a few months (or years!) ago—what would you do differently today?

Once again, you can leverage the universe of open source software. Block out a couple of hours to contribute to an open source library you use (or want to use). If you're not sure where to start, check out the trending repositories on GitHub. Contributing to open source is an excellent learning laboratory, and it isn't nearly as hard to get started as you may think.[9]

You can still learn a tremendous amount about writing good code without contributing. Pick a project and spend a couple of hours reading through the code. What do you like? What don't you like? What would you do differently? Run a source code analyzer against the code: what does that tell you about the project? If you see any glaring issues, don't be afraid to raise them or contribute a PR!

Keeping up with change is a core component to a successful career in software; make it a habit to refresh your knowledge on your core languages and frameworks. You may not organically encounter new features in your daily work, so take a few hours once or twice a year to see what has been added to your toolkit. Most technologies have advocates or champions, so follow or subscribe to keep abreast of changes. Not every shiny new thing will work for your applications, at least today, but it is simpler to digest a small handful of updates periodically than to try to learn about dozens or hundreds of new things every few years.[10]

Last but certainly not least, consider volunteering your coding talent to a local charity or nonprofit. Many deserving organizations are constantly looking for help from the technical community. Volunteering can give you a chance to practice your craft, maybe using a language or framework you're trying to learn, while also helping a cause you care about.

Additional Resources

- "Portrait of a Noob" by Steve Yegge
- Discussion thread on favoring composition over inheritance
- *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition, by Frederick P. Brooks (O'Reilly, 1995)
- "An Appropriate Use of Metrics" by Patrick Kua
- "Simple Made Easy" by Rich Hickey

[1] AI doesn't currently, and likely never will, distinguish between whether it can write code to solve a problem and whether it should.

[2] Like on the freeway when people notice the state trooper in the median and everyone slows down.

[3] Though with modern monitor sizes, you may want to stick to a single page.

[4] You can often tell the experience level of a developer by their use (or avoidance) of code comments.

[5] Dogmatism, whether in software or life, is rarely the right path. Favor pragmatism.

[6] Libraries like Spring REST Docs can also help documentation stay in sync with the code.

[7] One of your authors, who shall remain nameless in this instance, spent the better part of two weeks debugging that code block.

[8] Also known as “looks good to me” reviews, they aren’t a good use of a person’s time.

[9] Many projects have lists of bugs marked as “for first-time contributors,” but don’t be afraid to reach out to the contributors; most will happily help you get up and running.

[10] Something that also applies to *upgrades* to said technologies…