Junior to Senior

How to level up as a software engineer

Yuri Karabatov

Contents

1 Introduction

Thank you for buying this book. It means a lot to me. Read it, live it, get excited, and if you have any questions, feel free to email me at yuri@norikitech.com. I read every email.

1.1 Who this book is for

The earlier you are in your software engineering career, the more you will get out of this book. If you have just started a new job after a programming bootcamp, have recently completed your Computer Science degree, or even have been working in software for a year or two, you will learn some insights that usually come with several years of experience, or not at all if you don't care to pay attention. I have some strong words to say about "years of experience," but let's leave it for later.

There is much to learn in software development that is not about writing code. Being successful in commercial software development is often not about writing code at all. At the same time, the more junior your role is, the more you are judged by how well you write code and how many libraries you're familiar with. That is what programming bootcamps are all about: in your short time there, they teach you how to code, and not much else.

Don't get me wrong, programming skills are important. They can get you impressively far in the workplace and in your career, and there are certain domains where sheer programming ability (the time to deliver or the amount of practice in one knowledge domain) can make all the difference for the business. But most of us are not superstar programmers and even if we were, we rarely work alone. You will likely work on a team, and with other people: designers, testers, managers. If you're lucky you'll even work directly with customers.

This book is for you if you want to find out what it means to be an engineer in commercial software development, what *other* skills besides programming the companies are looking for, and how to advance your career in a systematic way by knowing how the game is played.

1.2 How I can help

I've been in the beginner's shoes, too, everyone has been. I have always loved to find out *why* and *how* people do the things they do, and for many years I've been learning both the technical skills and more importantly, what it means to be a professional engineer. I don't know it all, and I don't have thirty years of experience, but I have ten, and I've spent a lot of time seeking out and working on teams with true craftsmen that helped me learn and improve. Now I want to share what I've learned with you.

This book won't teach you the best libraries to use, or what size your functions should be but I'll recommend some books that will. These are the technicalities that you can learn at any time or find on the internet. I will show you something better: timeless skills that you can use *now* to launch your career and by applying them repeatedly, become a valued member of your team. I will show you how professionals think about their place in the company and the code they're writing, and why it matters.

You will learn three things in particular: how to navigate the workplace, how to navigate the field of software development and how to navigate your career. By knowing where you are, where you can go and how to go there, *you* will be in charge of your software engineering career.

You don't need to learn any of that on purpose to have a long and successful career in software development. There are lots of programmers in the world and if you work in a profession for many years, just by handling different situations you learn what is expected from a professional. It happens naturally but it takes *too damn long*. You don't have to spend ten years working for an insight to go off in your head, your brain finally matching the pattern that you have seen dozens of times. Glean the insight from a book. Start with this one, and read others.

The trouble with time travel—skipping the years of experience and learning from a book—is that when you're just beginning the journey, you don't understand *why* you should follow that one bit of advice. You think you don't need it or it doesn't apply to you at all. Give it time. A lot of the professional's behavior patterns are reinforced neural pathways firing in the brain, and thinking hard will not emulate them. You just don't have the experience (again, that pesky word!). I try to give you advice that you can use at any point in your career.

Experts often share advice that looks simple—but is too advanced for a beginner—in tight little books. A couple that spring to mind are Steven Pressfiend's "Turning Pro" and Alex Hillman's "The Tiny MBA." If you are not seasoned in a field (for Pressfield, creative work, and for Hillman, online business) the advice they give will likely do you no good. It's too high-level to be actionable, requires knowledge and failed attempts under your belt to understand *why* they say what they say. For professionals, it's a handy reference of what can go wrong that they can quickly skim and find inspiration and focus. As learning tools these books produce more questions than answers. That's why I'm aiming to give you some of the high-level vision that distinguishes the professional from the beginner, without losing the beginner along the way.

2 The point of reference

> “Would you tell me, please, which way I ought to go from here?”
> “That depends a good deal on where you want to get to,” said the Cat.
> “I don’t much care where–” said Alice.
> “Then it doesn’t matter which way you go,” said the Cat.
> “–so long as I get *somewhere*,” Alice added as an explanation.
> “Oh, you’re sure to do that,” said the Cat, “if you only walk long enough.”

To get anywhere, you need a map. To plot your way on the map, you need to know where you are. To make the trek, you need to know how to cross difficult terrain. I think the map metaphor works well for what we’re trying to learn. *The map* is the field of software development, *your location* is what you know (and what you can do), and *the terrain* is the workplace and all the people you have to talk to on your way to become a senior engineer.

The junior engineer James has a low-resolution paper map, with big chunks of it blank, edges marked with “Here be dragons.” The senior engineer Susan has a dedicated GPS unit with a color screen that also shows heights and points of interest. The junior wears sandals and a T-shirt. The senior has packed a rain jacket and trekking shoes for the rocky path ahead.

The question is: could the junior bring an umbrella? Yes, if he cared to look at the weather forecast. The thought never crossed his mind.

By knowing what’s available and what’s missing from your arsenal of knowledge, you get a superpower: you’ll know what doesn’t let you make better decisions, and then you will go out and learn it. Next time when it’s going to rain, you’ll stay dry.

2.1 “Developer” or “engineer”?

Some advice on the internet warns against calling yourself a developer, and only ever writing “software engineer” on your résumé. The developer is considered a lesser species than the lofty engineer. “A developer simply *writes code*,” they say, “but an engineer… *engineers*!” (hand waving commences). As if code was something you didn’t touch. After seeing some people’s code, I agree: there *is* code that you shouldn’t touch.

To be serious, this debate is completely irrelevant in the context of getting programming jobs. Nobody will discriminate against you if you put one on your résumé or the other. Some companies call their software engineers “software developers” while others have “engineers.” Don’t call yourself a “software ventriloquist” and you’ll be golden (though that’s one cool title).

This advice has a core of truth. As Patrick McKenzie brilliantly explains in his essay “Don’t Call Yourself a Programmer,” if we take a bird’s eye view of the company, developers are very expensive “resources” and in many companies it’s hard to directly link the value that the developers produce to the revenue of the business. Hence, the managers want to decrease these costs and aren’t interested in educating and retaining good engineers. From the business point of view, it makes sense: a business must generate profit to stay afloat, and one way to generate *more* profit is to cut costs. Patrick’s advice is to position yourself as someone who can increase the revenue of the business, rather than someone who “writes code.”

Understanding the business side of things is yet another level of the career game and involves another set of skills that are mostly orthogonal to those of a software engineer. If you want to know more, read what Patrick has to say and follow his pointers because in this book I only touch briefly on navigating your career, and I’m mostly talking about becoming a better software engineer—the foundation from which you can take your career where you see fit. We must learn to walk before we can fly.

2.2 Going meta

One thing we can learn from this is that there is almost always a deeper, more advanced, more high-level way of looking at everything you do. Math or physics is the same: people use a formula for decades or even centuries and then some genius comes along and explains that the formula is only a special case (this is called a generalization). In geometry, the square is a special case of the rectangle, the rectangle is a special case of the quadrilateral (a polygon with four edges and four vertices).

You can learn these new ways of seeing by seeking out experts in your field and learning what they know by talking to them and reading their work. The outcome is that it completely changes your behavior. You now know your previous destination to have been a mirage, with the actual place you want to arrive somewhere to the side, or behind a hill that you never knew existed. It can be discouraging because it devalues some of the knowledge that you've worked to earn but at the same time unlocks new paths where there have been none.

How are you supposed to follow their advice if it's so meta? There is no simple way. Treat what you find as a high-level map and fill in information as you learn. Come back to these books (blog posts, courses) again and again and they will reward you with new insights. Our brain is perpetually looking for new patterns and if you are stuck on a problem, by rereading an advanced book you are likely to notice new connections or a path to the solution that you haven't noticed before.

2.3 Typical descriptions

Each company defines its levels of software engineers differently (or doesn't have them at all), but we can find some common threads that I will use, so that we have shared context when we speak about being a "junior," "middle" or "senior" software engineer.

In my mind, we can place all levels on two axes: *responsibility* and *impact*, every level having more of each than the previous. Responsibility is how much the engineer stays in the confines of their role, and impact is what weight they have within the company, or what the consequences of their actions will mean to the company. Let me explain.

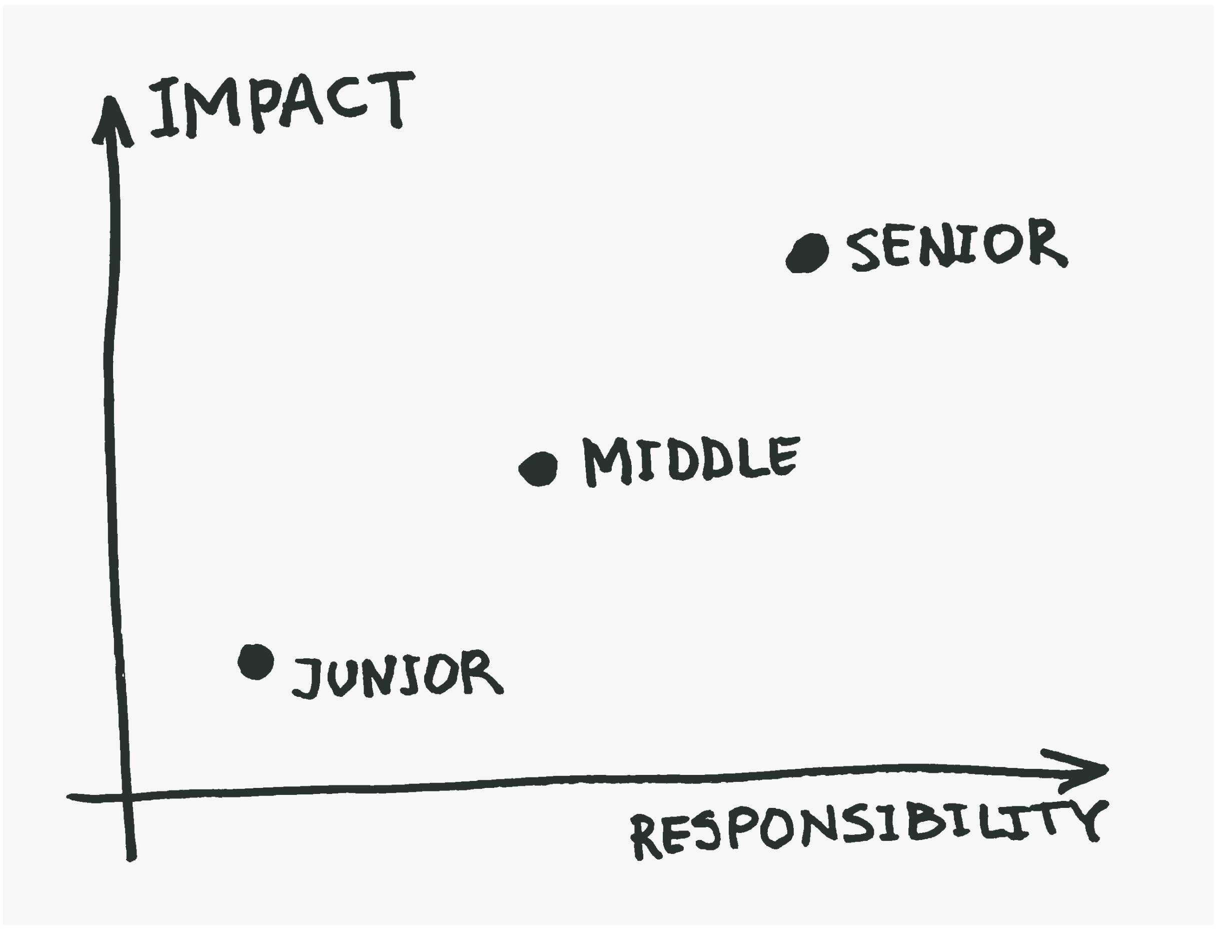

Fig. 1: Responsibility and impact.

The junior engineer James does not have much of either responsibility or impact. The job description of the junior is to learn, and he is under supervision from a more senior engineer. His *responsibility* is low because he does not step far from his role of *writing code*. He does not decide *what* problems to solve—they are handed to him by his project manager or a senior engineer, who is responsible for grading the problems' difficulty and assigning them to the team.

Often the junior engineer does not even decide *how* to solve a problem. His solutions are not ideal and after code review a senior engineer will tell him *how* to solve the problem, because the senior engineer is responsible for shipping the best solutions.

The junior engineer's *impact* is also low because he is handed low-priority "learning" problems that are not critical to the product or the company. If the problem were critical, it would more likely be solved by a senior engineer.

That paints a bleak picture, doesn't it? For a lot of us the experience would have been (or is) completely different: we are given complicated work that makes a difference to the company, and our opinion on product features is valued. Giving "stretch" projects is the best way to grow the engineers' skills while keeping them engaged, and that's how we get the knowledge to level up in most companies. But when we cut everything non-essential to the role, the junior engineer has low impact because the tasks he does are simple and can't break the product, and he has low responsibility, because he does little to decide the direction or performance of the product.

What about the middle software engineer (let's call her Mary)? Mary has significantly more responsibility and impact than James, the junior engineer. Her job description is to "do the work" and implement whatever is needed for the product. She is not (or lightly) supervised and works independently, thus she is responsible for any mistakes that she makes.

She has medium responsibility because aside from writing code, she ventures into other roles: helping with design and architecture decisions, and of course she decides how to solve the tasks that she is given. At the same time, she rarely (if ever) *makes* any but minor software architecture decisions, or what goes into the product. Her impact is much higher than the junior engineer's, because her implementation decisions are most often her own—reviewed, but rarely given by more senior engineers. That means she has the means to break the product, and if she does, she will also likely be the one responsible to fix it.

The senior engineer Susan has even more responsibility and impact, sometimes as much as the company's size allows. In smaller to medium-sized companies, there are few if any purely technical roles higher than the senior software engineer. Even if the senior engineer does not call the final shots on new technology or major architectural changes, she is most likely the one driving these changes.

Her responsibility reaches far from her technical role. As a technical expert, she can make design and technology decisions that impact the whole product or its major pieces, because she has the knowledge whether some design is feasible, if the platform supports what has been proposed by marketing, or if implementing something is possible at all, and at what cost. The senior engineer is responsible for these decisions, and if something doesn't work out, she is to bear the consequences. She is often also responsible for any work that is produced by the junior engineers—they aren't responsible if something doesn't work, instead she is.

What about her impact? Depending on the size of the company, the senior engineer's work and decisions influence the whole product or its major parts, and any mistakes can result in outages and financial loss. Often there is nobody else (besides the other senior engineers, if there are any) who is expert enough to review the senior engineer's code, or rather, what is implemented—the subject matter.

There are other technical roles that have even more responsibility and impact, and they mostly start to appear in companies with at least several dozens of software engineers. In smaller companies, the *role* itself might exist virtually: there will be no dedicated person, but, for example, the project's architecture would be decided by the team lead and a senior engineer.

Some other *mostly* technical roles that you are likely to encounter are the tech lead, the team lead, the principal engineer and the software architect. I will not talk much about them because their technical skills as a software developer are unlikely to be better, and are often worse than those of a senior engineer. I don't say that in contempt, this is simply the nature of these roles. The role of the senior software engineer is primarily to produce code, while the tech lead and the team lead's roles are more about managing the people and how the work is done. The principal engineer's and the software architect's roles are more about coordinating and directing technical effort while focusing on the business needs. Here's Yan Cui describing his experience as a principal engineer in his post "How to Break the 'Senior Engineer' Career Ceiling":

> …my personal output as an individual contributor was the lowest it has ever been. And that's OK, it's kinda the point—you need to optimize for your impact on the business. Sometimes that means sacrificing your personal output as an individual contributor.

I have been a nominal team lead in one of the companies I worked for, but I like to write code and think about mostly technical things, so early in my career I've made a conscious decision to avoid "breaking the career ceiling" and going into management or other roles that are technically more senior (but less technical, if you get my drift).

There are lots of industries in the world that need highly skilled software engineers, and there are lots of skills to master in the field of software development to keep your career interesting without going "higher" in the company hierarchy. By carefully choosing the companies and teams you work for, and in which industry, you can significantly raise the salary ceiling as well. If you augment your software development skills with some business savvy, becoming an independent contractor or creating software products on the internet can increase your earning potential even further—all while being primarily an engineer and not a manager.

2.4 Years of experience

Almost every job posting has the line "X years of experience" in the *Requirements* section (that usually comes right after "Bachelor's degree in Computer Science"—but we'll talk about that later). Usually it mentions a specific technology, such as "5 years of experience in Java and Spring," but sometimes you see a more general "3 to 5 years of software development experience." That may look disheartening if you've only been working for one and half years, but have already mastered your job and start to feel bored.

What do these companies really want when they ask for a certain amount of years of experience? That is simply an easy filter for the HR department to reject people who may be qualified (which means "can do the job"), but don't have the credentials (which means the recruiter will have to spend extra time to look at their other skills to see if they match the job description). What this filter represents is the *average* rate at which even the slowest learners accumulate the skills necessary to do the job. It is very much like the system at school: school is designed to accommodate the slowest learners, so that by the end of the school year, everyone has learned enough. But if you sort the children by *ability*, you will find that those who are "gifted" can learn much faster.

So, after two or three years in the workplace even the junior engineers who learn very slowly will get enough excess knowledge to qualify for a promotion and become a middle software engineer. Same with the middle engineers, just by doing the work (even without much interest) and just spending the time encountering various situations at work, after a few years they will gain enough knowledge and skills (at least in their main role) that they would be able to become a senior engineer. Of course, that heavily depends on the industry you're in and the nature of the work you're doing, but that is the average. I think that is soothing. You can know with certainty that even if you are not outstanding at your job, or not particularly interested, if you come to work every day and do your part, in five to seven years you can become a senior software engineer.

Your knowledge and skills will accumulate naturally because you will be handed harder tasks to do (at least, that's how it usually happens). As you work, you will discover that you have more aptitude for one type of work than for others, or become interested in another part of the business, and eventually little by little you'll take more responsibility. You'll be able to do more because you've mastered what you've been doing before.

Don't let the line "X years of experience required" prevent you from applying. Some people don't apply to jobs unless they believe they satisfy all of the requirements, which is usually impossible in software. The job posting frequently is a bag of facts describing the *ideal* candidate who simply doesn't exist.

On top of that, knowing *all* the technologies listed in the job posting is often not required as well. Again, it's almost impossible to find a person skilled in the exact stack the company is using, and it's much easier to teach a good candidate than to spend time looking for the ideal candidate. In the first couple of months nobody expects much performance from new hires anyway, so they can learn whatever they're missing while already working, and it's very common.

Later in the book, in the section about interviews, I will talk about how to step around this and other filters you might find in a job posting. They are designed for the lowest common denominator, the weakest candidates, so the applicants can be quickly rejected without even looking at them closely. If you are dedicated to improving and are excited about software development (and you must be since you are reading this book), you can learn much faster than that and get your required "years of experience" twice as fast.

Consider the direction of your learning and the Pareto principle (that moving 20% towards the goal gives you 80% of the total benefit). Compared to the average, you will likely know much better which direction you want to go in your career. That will inform *what* exactly you learn and how far you will go in a subject before only tracing the contours of what you don't yet know, to learn later if you need it. In my experience, it is far more important to know the "shape" of a subject and know its basic principles—

because there are many such nuggets of knowledge in software development that you'll want to know—than spend an inordinate amount of time learning something and never using it again. Are the school memories coming back? It's a little different at school, because children don't yet know what they will need (and let's be honest, us adults often don't know either), but at work you're free to learn as much or as little about a subject as you want.

2.5 Computer Science degree

Ah, the elephant in the room, a Computer Science degree. I'll come right out and say that I don't have a degree at all, even though I spent four years studying. I spent two years studying physics at one university and then two years studying English at another. Then I met my future wife, dropped out and moved to another city. C'est la vie.

Well, do you need it? The answer is: yes and no. It depends. The "CS degree" is actually three things, or "three kids in a trench coat": the credentials (the face), the knowledge (the torso) and the skills (the feet). Let's look at each of them in turn.

I will be talking primarily from the point of view of an application developer, be it web or mobile, and simple backend development, all of which are typical first roles for self-taught software engineers. I started with web development and later moved into mobile. Do I say these fields are "easy"? No. The skills and knowledge ceiling is quite high, especially when high performance is involved and is critical to the product. But application development is undeniably accessible: you can start with a text file and a web browser.

If we look at other fields, like finance or math-heavy industries, you will need more knowledge and skills even in the entry-level positions, which is why in those fields there are fewer self-taught software engineers. Read on and remember I'm talking about a field with a low-ish barrier of entry, where a lot of the knowledge a CS degree provides will be unused at the start.

2.5.1 Credentials

This is the only part for which you have to go to a university. You can get the knowledge and the skills on your own, but you can't get the credentials without attending the university. Now you don't even need to attend in person, there are online universities like the Open University and a few others. They are not free, but the price for a year is reasonable. Still, the curriculum is paced in such a way that you will have to spend almost or just as much time getting the degree as if you were attending in person. I don't think you can go lower than three years for a Bachelor's.

Having the credentials will help you open some doors, and by being a student, you can get early job offers and start a career right after graduation (Joel Spolsky makes a case in his post "Finding Great Developers"), not to mention have a great time and make some lifelong friends. But we are only talking about what the degree gets you.

The degree gives you much better chances at getting an entry-level software development role, since by the virtue of getting it, you are passing through the filter where a person with the same knowledge and skills, but no degree, would be rejected. Your degree tells your potential employer that you have at least some aptitude for programming and you know how to write code, that you can persevere and complete tasks that are assigned to you, on time, and that you probably know the bare minimum on how computers work because you've studied networks, computer architecture and other things.

If you have a CS degree and relevant work experience, you will be one of the first candidates the company looks at, and that can help you land much better jobs than if you didn't have one. So far it's been pretty obvious and logical.

Where it gets interesting is at or around the senior level. Your degree (you may have other credentials by this time: certifications, etc.) becomes much less important, because after five, seven, ten years the knowledge has eroded. I can vouch for that: I've been out of the university for thirteen years and I don't remember much at all. Of course, if you were using something, you'd remember it, but here we come to the next point.

The actual work experience at the senior level becomes much more important, because by this time the self-taught software engineer has *learned* whatever they've been missing and needed for work. So at the five, or seven, or ten-year mark, the person *with* a degree would only remember what they needed for work, and the person *without* a degree would have learned what they needed for work. In the end, the amount of general, non domain-specific “computer science” knowledge both of these people have is the same.

What starts to differentiate these people is the amount and quality of work experience they have accumulated, and the specific personal aptitudes they have discovered e.g. one likes to write documentation, and the other can deliver prototypes quickly—one of which may be significantly more important for the specific role they are applying to, than the other.

Another (huge) door that a Computer Science degree can open for you is getting work visas if you decide to work abroad. Software engineering is an international profession and besides having the skills, you only really need to know English to get hired anywhere in the world. Getting hired is one thing, but actually getting a work visa and moving is another. The requirements are different in different countries and for different nationals, but generally in order to get a “skilled professional” or “engineer” visa that allows you to work as a software engineer, you will need to have a computer science, mathematics or mechanical/electrical engineering degree.

For example, as a Russian national I need a technical degree to get the German “Blue Card” visa or to get a Japanese working visa. Even if I passed the interview with a foreign company and they wanted to hire me (based on my skills), I would be unable to get a work visa and actually move into the country, and the company would be unable to legally hire me (based on my credentials, or rather lack of them).

The visa issue is probably my biggest regret for not getting a Computer Science degree, because I probably could have. But at the time, when I was sixteen and was choosing which university to enter, I wanted to be a translator and study English, and I only studied physics for two years because that’s where I managed to enroll, before transferring to my preferred university and program. Anyway, my English degree, even if I managed to graduate, would not have helped me. Oh well.

I considered getting a Computer Science degree from Open University, but decided that spending four years purely to get around some red tape was not my cup of tea. It would be somewhat valuable for the knowledge, but I’m no stranger to learning by myself. The knowledge is available on the internet for free if you have the discipline to study. There are ways around working visa restrictions, and I've chosen to keep the four years and learn other things. In the end, if I exhaust other options, it's never too late to learn, and four years is not that much time.

On that personal note, let's move on to the next one. If we were to learn everything that is on the Computer Science degree curriculum, what would that be worth?

2.5.2 Knowledge

All the knowledge that you are supposed to get when studying for a Computer Science degree is available—for free—on the internet. And not just knowledge, but the curriculum and video course lectures from the best universities like Stanford and MIT. Commit, spend the time and you will learn the same things and in the same way, as if attending the same lectures. Isn't that cool?

The website "Teach Yourself Computer Science" offers several options you can choose from. There are also Open Source Society University, Awesome CS Courses, The Open Source Computer Science Degree and more.

The university teaching system is similar to that at school, but in a different sense to what I said earlier (catering to the slowest learners). The top universities are particular about filtering out slow students, so the amount and breadth of material you will need to master is enormous. That is the problem. For a Computer Science degree, you will study algorithms and data structures (in depth), networks (in depth), 3D graphics (also in depth, like Stanford CS348C) and then graded based on how well you've learned all of it. It's mostly due to the breadth and depth of material that the typical degree takes four years of study (if you don't go for the Master's). The typical coding bootcamp takes four months. It's about paring down the material you have to learn to the barest essential skills you will use at work.

If you already know what you are working with and are studying yourself, you can skip learning the parts of the degree that are irrelevant to what you are doing, like 3D graphics if you only intend to do backend programming. There are concepts you can learn from 3D graphics programming that you can use in backend programming, but let's not get into that now :) Everything is a learning experience for something else if you choose so. By skipping parts that are irrelevant you can cover the material

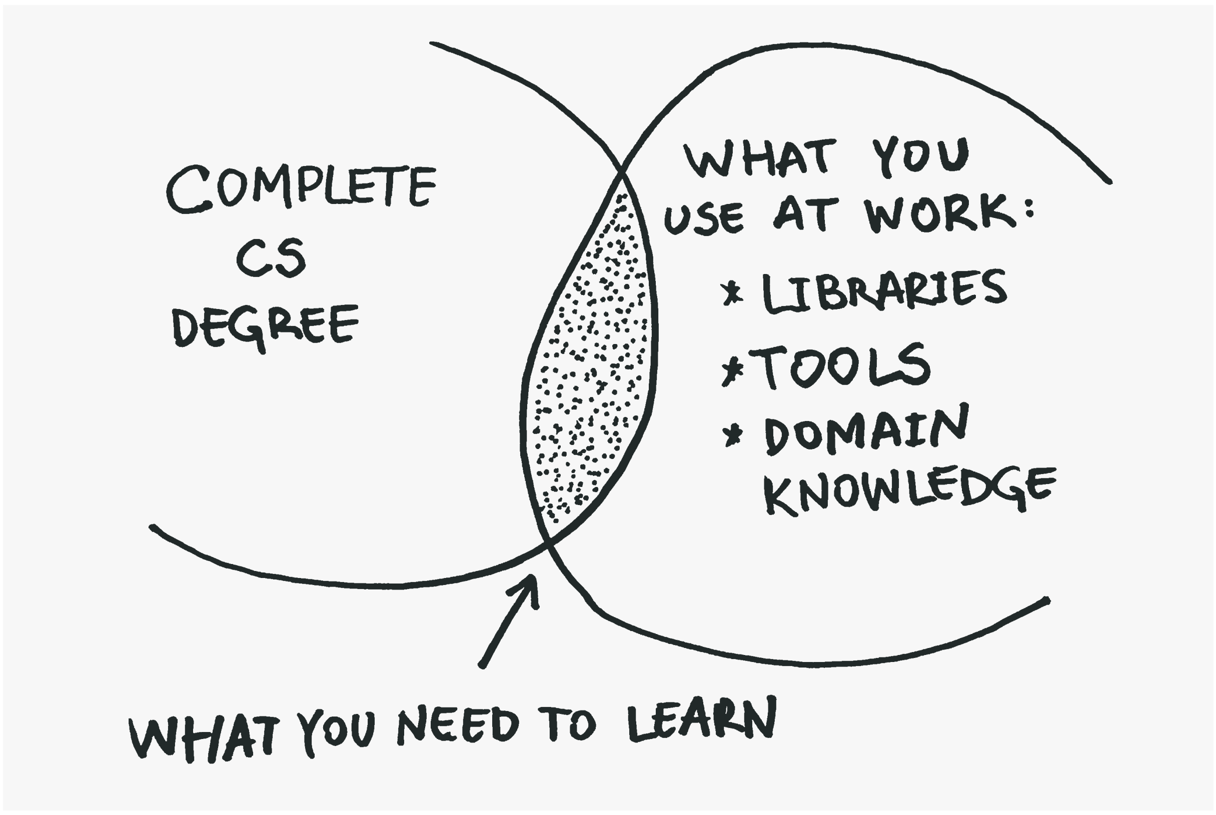

Fig. 2: The crossover between what you need at work and a complete CS degree.

that you *do* need in more detail and finish sooner. On top of that, if you were to self-learn, you wouldn't need to pass the final exam, and so you could avoid cramming and remembering all the details, and practice only the essentials while learning about—but not closely studying—the advanced details.

By following this method, you can learn everything that is relevant *to you and your work* from a general CS degree in a matter of months, rather than years. Would you be worse off than a fresh graduate who has actually spent the time cramming the deeper details? Maybe, but does it matter if you would never use them? If and when you do need the details, by being aware of their existence, you can go and learn them without spending the time up front.

The graduate, when she needs the details, will have to refresh them in her memory too, because we forget what we don't practice. In a few years, both you and the graduate will be in the same position, where you will be *aware* of the deeper details (because you've read about them and the graduate studied them), but both of you will have to spend the time learning them (in the graduate's case, again, because by that time they will likely be forgotten). After a few more years we end up in the situation that I've described in the section *Credentials* about the two senior engineers, where the amount of knowledge would be roughly the same for you and our imaginary graduate.

Please don't get me wrong, I certainly don't devalue getting a degree, and bear in mind I'm only talking about Bachelor's. If you look at a Master's degree or even further, that's a tremendous amount of knowledge that I wouldn't dream of, being self-taught, if only because I don't spend all my time learning and learning and learning.

How much of that knowledge you'll be able to apply at work also heavily depends on the industry. I just said you could learn most of what you'll need in a matter of months, but I was talking mostly about software developments jobs with a lower barrier to entry, like simple web or backend development (where deployment and maintenance is often performed by other people), where you don't need much domain knowledge and mainly need to be skilled with the tools.

If you wanted to start working in a more knowledge-demanding industry or part of the stack, such as anything high-performance or math-heavy, or both, like high-frequency financial trading, you would need multiple years of self-study, not months, to acquire all the knowledge—credentials notwithstanding—that is required even at the entry level in roles like these.

Is there anything specific that even the junior engineers should know, and can learn quickly? We'll talk about that in the section *Know your place*.

2.5.3 Skills

We know how the technical interview often goes: you are at the whiteboard, writing code to invert a binary tree. Is this the skill you learn by getting a Computer Science degree? No, it isn't. This "interview coding" is much closer to competitive programming, which is completely different to the skills you have as a CS graduate. *I* don't have a Computer Science degree, so I won't make an actual list of things that a graduate is supposed to know, and anyway, all the universities and countries are different, so the exercise would be futile.

We can make a generalization. Straight out of the university an average graduate, compared to someone who is self-taught, would have multiple hundred hours worth of programming (likely in multiple languages), have practice decomposing tasks and completing projects, and have some experience designing systems.

It is important in commercial software development that you know the language and relevant libraries as intimately as possible, can work under supervision and produce production-grade code that fits well with the existing codebase. Working under supervision and knowing the programming language are probably the two skills that have the most crossover with those of a CS graduate.

The set of libraries and tools a company works with is likely to be different to what has been taught at the university, and the graduate has little experience writing code that is considered "production-grade" or fits a certain style. The programming students do in the academic setting (short-term personal projects with little quality assurance, written once and forgotten forever) is very different from programming in a commercial setting (long-term projects continuously worked on by a group of people with strict quality and maintainability requirements).

In the end, both the graduate and the self-taught developer will have to learn most of their "work" skills at work, and while the skills that you get acquiring a CS degree will certainly give you a headstart, it is not *that* significant compared to someone self-taught, who has purposefully studied the language, the frameworks, libraries and tools they will be using at their job. Again, this is the result of directional learning. A CS degree gives a broader set of skills applicable to many industries, but if you know beforehand where you want to work and in what language, you can learn only that and be immediately productive in the workplace.

That is the main selling point of Lambda School who spend about a year teaching their students to be software developers, from zero. A year is much longer than the usual four-month coding bootcamp, and their program and reviews prove that it's enough time to get the knowledge and skills necessary to be successful even at top-tier real-world companies. Are they missing some skills the graduates of a traditional CS degree would have? Absolutely. But as I've said earlier, you can get those skills later when you actually need them, and have them fresh.

As long as you spend some time programming what you want to program at work, and practice good coding habits that you won't have to unlearn, you will not lose much in practical development skills compared to someone with a CS degree.

2.5.4 If you don't have a Computer Science degree

It's not the end of the world.

Companies need good software engineers who have the skills to do the work. An incredible amount of great software engineers do not have a CS degree. Some don't have any degree at all: *John Carmack* is a dropout (well, he's also a genius, but still). A Computer Science degree does not automatically make you a great developer. What makes you a great developer is that you care about the craft and what you make, you come to work to learn, and you know what you don't know (more on this in the section *Unknown unknowns*).

I often hear the argument "But you wouldn't want to be operated by a surgeon without a medical degree!" I wouldn't, but surgery is different from software engineering. A surgeon is a specialist, performing the same few operations over and over again according to well-known best practices, on people who are all extremely similar. It is the same with construction or building bridges: the physics of the process is well-known and there are established, standardized practices.

Yet after all the years of writing software, there are still no proven, universally accepted practices of constructing software projects, it's all ad-hoc. On top of that, the realm of software engineering is so broad and deep that it's impossible to apply the same practices everywhere. Software written by degree holders is not provably, objectively better than software written by self-taught developers, and the degree curriculum doesn't have any esoteric knowledge that a self-taught developer wouldn't be able to grasp when necessary.

I believe that a software engineer is much more like a writer than a surgeon. The world needs writing in a multitude of styles and languages, and many successful authors with great English do not have literary degrees. Could their writing be improved by getting one? Perhaps, but then they probably wouldn't have gotten the experience that has become the *subject* of their writing.

Now that we have consoled ourselves that a CS degree is not that important, how do you convince your potential employer, and what can you do about it?

There are specialized job boards like "No CS OK," and even Google has recently removed the requirement to have a CS degree (or any degree) in their job postings. In my experience, as long as you can show that you have the skills and can do the job, the fact that you don't have a CS degree doesn't matter. We will talk about applying to jobs later, but generally you will be the preferred candidate if your skills are up to scratch. In one of my past jobs, the engineers who decided if the candidate was worth being invited to an interview never even saw the candidate's résumé, only their test project, and as far as I know, the candidates were never picked based on their credentials over how they handled themselves in the interviews.

You can get most of the skills by simply programming and working on projects, and as for "whiteboard programming," that is a skill you'll want to train separately anyway—and there's a high chance you won't need it. There are no universal practices in software, and interviewing is all over the place too. Recently there has been a rise in the number of companies that don't make you study LeetCode for weeks, you just talk about what you've done and maybe write some code, similar to what you'll be working on, which is much less stressful.

As for the knowledge that comes with a CS degree, you can get a "fast-track" by studying the public material yourself. How exactly do you do that? We'll talk about it in the section *Exponential learning*.

2.6 The senior engineer mindset

The senior software engineer has skills and knowledge that far exceed that of a junior, but to me, the main difference they have is their mindset, or how they see their place in the company and in the team, because that directly informs what they do. By changing the beliefs that you use to navigate work and your career, you will gradually change your behavior, and I think, for the better. What you learn here will help you move up, just as it has let me. Some of this may be obvious or simple to you, but not all things need to be complicated.

There are many traits that we may assign to the senior engineer, and I could give you a list of ten, twenty thought patterns and nuggets of knowledge that someone who is more advanced in their career has, compared to someone who is more junior. Instead, I've chosen four, which I think are critical to how you see yourself.

Note that these are not *skills* or things that you do, but something more like a set of *criteria* or *values* that are integral to your behavior in the workplace. Every action you are about to do can be run and validated against this list. As cheesy as this sounds, these values are not set in stone and I don't preach following them at all times, not at all. It is simply a framework that you can keep in mind as you think about what you want to do at work and in your career.

2.6.1 You are not paid to write code

How many software engineers think about themselves: "I go to work to write code"? Do you?

Code is easy to see (it's right there on your screen), easy to count and get statistics about, so it's easy to fall into the habit of making code your KPI, and be content in building a solid green wall in GitHub Contributions. How much code we write, and how fast we can move those Jira tickets across the board, becomes our measure of success, because that is what lies within our control. We simply focus on what needle we can move, and that, in our mind, becomes the measure of our work.

The job you applied for had writing code as one of the primary requirements. During your job interview, your future colleagues tested how well you could write code. Your job description might say that your job is to write code. That means you are paid to write code, doesn't it? How come you are not?

There's a well-known saying "the best code is no code at all." Kelsey Hightower has a parody repo on GitHub and Jeff Atwood has a post urging to write as little code as possible. But it is a saying mostly about maintenance, the tactical level. Let's move up and consider what it means to be a software engineer working at a company. (If you've read Patrick McKenzie's blog post I've linked to earlier, you'll know what comes next.)

Why does a company hire software engineers? Certainly not to write code: if the company could buy or use a product that would allow having fewer engineers, it would, and companies do that all the time, especially smaller ones. They can use Stripe and not have someone write billing code for them, they can use Ghost to make their website and not hire someone to make one for them from scratch, and not have someone to configure hosting. They can use React Native and avoid hiring mobile engineers who can do only Android or iOS.

The goal of a business is to generate profit, and software engineers are *very expensive*. They generate a lot of value for the business, which means profit, but sometimes the connection is not direct and it's quite hard to see, so the top managers may always be on the lookout for cheaper alternatives to in-house software engineering.

Companies generate profit by providing services (or selling products) to customers, even if those customers are other companies. In other words, companies provide value to their clients. If no value was created, there would be no clients and no profit for the business. Software engineers who create the company's products or help provide services are among the people who are creating value for the company's clients. What is this ephemeral "value" anyway? It is simply something that helps customers solve their problems quicker and easier, than if they did that themselves—if they could do it on their own at all.

Ultimately, your first duty as a software engineer is to the customer and the product. Does it mean you have to take on the roles of the product manager, the designer, the user researcher? If we take a more cynical view, what does it mean if you make the company's product better—you'll get paid the same salary anyway? None of the additional profit you create for the company will end up in your pockets. What if you are working in a role very far from the customer, making some line-of-business (read "boring") application, that is only used by a division of your company? What can you do and what's the payoff?

What you get out of it is both tangible and intangible. The tangible stuff is what you put on your résumé: by how much you improved something, how you decreased the company's costs on running cloud services, how tools you made helped your coworkers move quicker. That's what gets you promoted, or given more stock options, or helps you overcome the competition for your next job, or all of that and more. Then there is the intangible. Thinking about how you can improve the way you work, what you're working on, and finally the life of people using what you've made gives additional meaning to what you do. Work is more fun when it's more than a stream of tasks that need to be done this sprint and the next.

I believe that you can add value (that is, make someone's life better) anywhere in your process. From your unique position in the trenches you can spot problems and opportunities and devise solutions that will help your coworkers, the company and its clients. As an application or web developer, it is somewhat easier to focus directly on the end product or the customers, but if working on the backend or the database, there are also many things you can do: you can speed up existing queries, suggest a new DB schema that will slash response time, introduce better deployment practices that would decrease downtime, and on and on.

Paraphrasing a reaction to Paul Graham's "How to Get Startup Ideas": "I wonder why people keep 'looking for an idea.' Just look around you, nothing is done. Start improving everything." It is the same in any company—nobody has the perfect process, the most efficient code, the best UX or the fastest load time. Everything can be improved when you start to pay attention.

When working on a feature or part of a process, I try to look at the big picture and see where it fits. You can notice amazing things this way. Think about how our junior engineer James and senior engineer Susan write code. For the junior, the feature he's working on is often a *tabula rasa*, he has no awareness how it fits within the product or the codebase. All the time, he gets tripped by the question, "What is the best way to do X?"

The senior engineer has the whole codebase and the whole product in her head. She looks at the big picture and notices patterns and can spot problems that nobody has thought about when designing the new feature. Designers sometimes think in distinct flows and separate screens, not how they work together in a big app or in what other contexts this particular screen is used, or how the new flow can disrupt the user's mental map of the app that they've had before. Are the designers and the product owners who have come up with the new feature in the first place bad at their job? No, of course not, they simply have a different view of the product than the engineers who are working on it.

By being in this unique position of the person implementing the product and the platform expert, standing right next to the customer, you can support and help your product people and designers, but in order to do that effectively, you'll have to refine your skills to support your instincts (as the singer Linda Ronstadt said). You don't need to become a product manager or a designer, but you need to be able to talk to them in their own words, and bring facts to the table that prove you are right. Here we come back to the responsibility and impact that you have in your role. By stepping outside and getting some skills of the other roles around you, you get more responsibility, and by pushing your ideas through and working on them you get more impact. After that we get to do the fun part: implement what you've suggested.

When are you supposed to do your work, if all you're thinking about is how to "add value" and get promoted? This all sounds very selfish. Indeed, but isn't our goal to make the best product possible and get the most profit for the company? It's a win-win situation. The company hired you not to write as much code as you could as quickly as possible, but to improve the product it's making to get more profit from its customers.

You find things to improve *by* doing your work, it's not one or the other. Thinking about what you do, how and why you do it is the mark of the senior engineer. The same principle lies at the root of *kaizen* (continuous improvement), made popular by the Toyota Production System. When they let every worker suggest improvements to the manufacturing process, the workers did, and now over 80% of their suggestions are implemented every year. In their 1973 report, Toyota said:

> The value to Toyota (and eventually to the public) of this well-functioning employee suggestion program is incalculable, both in terms of monetary savings efficiency and employee morale.

We are trying to do the same thing.

"Continuous improvement" and "adding value" sound a lot like corporate values that have been made up by a committee during a six-hour-long meeting. It doesn't mean that we can't have fun doing it. People work better and are more productive when they're having fun. At least for me, twirling ideas in my head is a fun thing to do. Infusing humanity and humor into your work *is* already improving it. It puts a smile on your coworkers' faces. When you find your work meaningful, you start caring more about it, and it shows. It's the same with the code you write: you want it to be elegant, efficient and fit well within your codebase, so that your colleagues' code could be better too.

Let's get back to the beginning of this section. Now we know that you've been hired because you *can* write code, not *to* write code. You are a specialist and an expert who happens to be able to program. Out of respect for yourself and your craft, start thinking about what and why you are programming. It just might happen that you will code yourself out of your job—instead of implementing the same stuff again and again, you'll notice a pattern and implement a system to be operated by other people who are not developers. You've just become a 10x developer by thinking about what you do and empowering other people in the process. Now you can sit back and look for more fun things to do.

A final note. If you feel stuck or feel the onset of burnout (How does it feel? Read in the section *Burnout*), buried under an endless stream of tickets to do, come back to this section. Take a deep breath. (Just kidding, it doesn't really help when you're stuck. But try breathing anyway.) Remember there's more meaning to your work than writing more and more code. And if you can't find any even if you try, maybe it's time to start looking for another job. Out of respect for yourself and your craft, no less.

2.6.2 Exponential learning

As we start to work in software engineering, we soon discover that we simply can't learn *everything* that tickles our fancy—there's too much. As we continue working, the mountain of Things We Will Never Learn™ only grows, along with the number of things we *need* to learn. Every new subject that you get into unlocks two more that are new, that you're excited about and eager to learn. At least this has been my experience. How about you? I like to know how things work, and why they ended up the way they are. That provides an endless stream of questions that sometimes require extensive research.

A simple "How do I write better code?" can lead you on a trek learning the history of programming languages, discovering best practices and whole books written on the topic, endless Hacker News threads and blog posts summarizing what people learned, comparisons of paradigms and approaches, and the latest research into code analysis and developer productivity. Every topic can be drilled down endlessly. Chasing down a bug, you might end up in the bowels of your system you never knew existed. Where to stop, and how much is enough?

Then there is the other side of the coin: what if you only learn what's in front of you? In "A Scandal in Bohemia," Holmes asks Watson how many steps lead up to their

Baker Street flat. Watson doesn't know because he's been using the staircase, not thinking about how it works. But is the number of steps on a staircase knowledge worth pursuing, even if you see no use for it?

If I run with the metaphor a little more, and liken the staircase to a library (the one that you use in code, not the one with books), then Watson already knows that there are straight and spiral staircases, that you put harder wood or stone on top of the steps to decrease wear, and all the other facts about the construction and use of a staircase. (A fact he probably doesn't know: in castles, staircases are often winded right so that it's easier for the defenders to fight down attackers by holding the handrails with their left hands, and the attackers' sides would be exposed when they try to fight back.) He wouldn't know the number of steps of the one he's using at home because it's irrelevant *to his use case*. As long as the steps are in order and it's not creaking, he does not see the reason to care. If a hypothetical user discovers a bug (one of the steps is creaking, or users sometimes fall over and break their necks), *then* Watson will see the need to examine that staircase slash library more closely, count the number of steps and check if either of them is creaking or making the users stumble. That's a practical approach and I admire people who have the restraint to *not* go off on tangents and learn everything in sight.

In both cases, we know that there is always *more* to learn about the thing in front of us. We have good reasons to believe that it will allow us to do our job better. But if we spend all day reading about this stuff, there won't be any time left to do the work. Thus, we must use several mental rules to give *shape* and *direction* to our search.

What does it have to do with being a senior software engineer, specifically? I think most people who reach the senior level (or are simply good engineers) are, first, curious enough about programming to have stayed in the field and gained the knowledge necessary to solve bigger problems, and second, have mental frameworks that allow them to effectively choose what to research and how much, so that they can use it in their work and not be overwhelmed. Quite often you are dropped into a new project running on a technology you don't know. Where do you start and how much do you learn before you can make calculated decisions?

Others may do it differently, but I'll tell you how *I* do it. If you go back to the beginning of this section, I'm definitely the first type, not the second: I can go down rabbit holes for *a long time* before feeling assured enough to make any judgements on a subject or make progress on a project, especially if the project is personal and I don't have time pressure to release it. Obviously, this is different at work where you have time constraints, so over time I learned to limit the amount of wandering I do when looking into new subjects to reasonable levels (reasonable for other people until they start screaming at me for doing nothing).

I use two components that I'm giving names to for the first time (just for you). They are *directive frameworks* and *tree shapes*. In the ocean of knowledge, tree shapes are my ring buoy and the directive frameworks are my oar. It won't help if a shark decides to have you for dinner, but it's something.

2.6.2.1 Directive frameworks

A *directive framework* is a set of facts and beliefs you know about a subject that is enough to allow you to make decisions that you consider informed and move forward (e.g. write code). Like a GPS navigator, it will tell you "Turn left here!" because to the right is a dead end. Like a compass, it may point to the true north, but there will be a deep gorge between you and your destination. That's fine, as we know, you can't know everything. The gorge is a signal that you need to learn something else to reach your destination. Don't forget to look under your feet.

When starting to learn a new topic or practicing new skills—writing code in a new programming language, learning a new framework—my first order is to build a mental directive framework for this new topic, so that I can take further steps by practicing and adjust it as I go. It can be tiny in the beginning and expanded later. It provides a frame (the purpose of a _frame_work!) on which you can attach additional rules and facts as you move.

Let's see how it works by trying to learn a programming language, for example a flavor of Lisp since it has a simple syntax. The first two *facts* that you learn are that every invocation is surrounded by parentheses, and that function calls follow a postfix notation, meaning that, inside the parentheses, you first put the function name and then the arguments. You learn that to add two numbers you should say `(+ 2 2)`.

Here it's important to look at the definition of a directive framework in more detail. It is "a set of facts and beliefs," because what's in your head is not necessarily true, it's only what you believe. You may even *know* something to be false, but for the purpose of the framework it does not matter as long it allows you to move forward.

This is a built-in unblocking mechanism, and very similar to writing a program. You may *believe* your program to be free of bugs and are expecting it to run flawlessly, but that is not necessarily true. Most programs have bugs, and that is fine. As long as you find them and fix them, you can move forward. If you wanted to produce a program that you knew for a fact had no bugs, you wouldn't be able to write anything.

It is much easier to fix *something* that is not working quite right than to build something that is absolutely perfect. It is the same with building a directive framework that *only* contains incontrovertibly true facts. In theory, it's possible, but it's slow and some facts turn out to be true under some circumstances and false under others.

The next important bit in our definition is about making "decisions that you consider informed." The framework should give you enough information so that you are *able* to make decisions based on the facts and beliefs contained in the framework. Otherwise it would not *be* directive—it wouldn't allow you to make decisions, meaning that it was incomplete and you needed to expand it first.

Coming back to our Lisp example, if you were asked to multiply two numbers, based on the facts you had in the framework you would reasonably write `(* 2 2)`. Since you've already seen the example with a `+`, the framework allows you to make an *educated guess*, to move forward and write some code that you expect to be correct.

Beliefs, compared to facts, often come from guidelines like "If you do X, then Y will not happen" which you can encounter in beginner texts and tutorials. You don't have enough knowledge yet to fact-check them, and even if you tried you probably wouldn't understand the explanation. They are good for a beginner, but generally stop working when you venture into more advanced territory. But often we carry these along and we never check some of them again. "If I use `new` and don't forget to `delete` I won't leak memory." "If I always use `weak self` then my iOS app will never crash."

When you're using your directive framework, pay attention to how you make decisions. Are you guided by facts or beliefs? Do you know the reasons behind the guideline? Guidelines are masking knowledge and facts behind a façade, and one day, when your framework has grown, you should look behind the curtain.

We learned that we need a directive framework in order to make decisions, but how do we know when it's time to stop growing it before moving on to something else? When learning a new programming language or a library, how deep do you have to go on your first pass? If you've been working with something for a couple of years, when is it time to learn more about how it works, and what exactly should you look at? "Peeking under the hood" is not very useful advice if you don't know where to peek.

If a directive framework is your compass, a *tree shape* is your map.

2.6.2.2 Tree shapes

A *tree shape* is a fractal mind map of a subject that contains both the big picture and the details. The big picture is the overall "shape" of a subject, the trunk and the leaf crown. Imagine a child drawing a tree: a straight line for the trunk and a circle for the leaves. It looks like a tree even from a distance. That's the big picture. The details are in the branches, twigs and leaves.

Ever noticed the subtle fanning of lines on the leaves? Sometimes we have to get down to that level of detail. The tree is, by its nature, fractal, or self-similar: every branch is also a (smaller) tree, with a trunk and its branches and leaves, and it follows the same rules. Start with the big picture, and trace the branches and leaves as you go.

How is this useful? When drawing a (generic) tree (in summer) you can't have leaves without branches, or branches and leaves without a trunk to put them on, or a trunk with branches and no leaves (it's summer, and we're not in a horror movie). It is the same with any subject you're learning. By starting with the tree *shape* we know that before looking at the small details, we must first have an overview and put all the major parts in place, build a structure so that our tree can support itself and look unmistakably like a tree from any distance. After we've done that, we can fill in the biggest branches, and when we're sure the tree won't fall over, *then* we can start looking at the twigs and the leaves and inspect the inside of the big hollow in the middle.

By knowing that the tree shape is *there* we can avoid missing critical but non-obvious information, or having gaping holes in our knowledge years after we've started working with a technology. We will talk more about this in the section *Unknown unknowns*.

While a directive framework is mostly about practice and things that you can do, a tree shape is more about general knowledge—because general knowledge makes up the

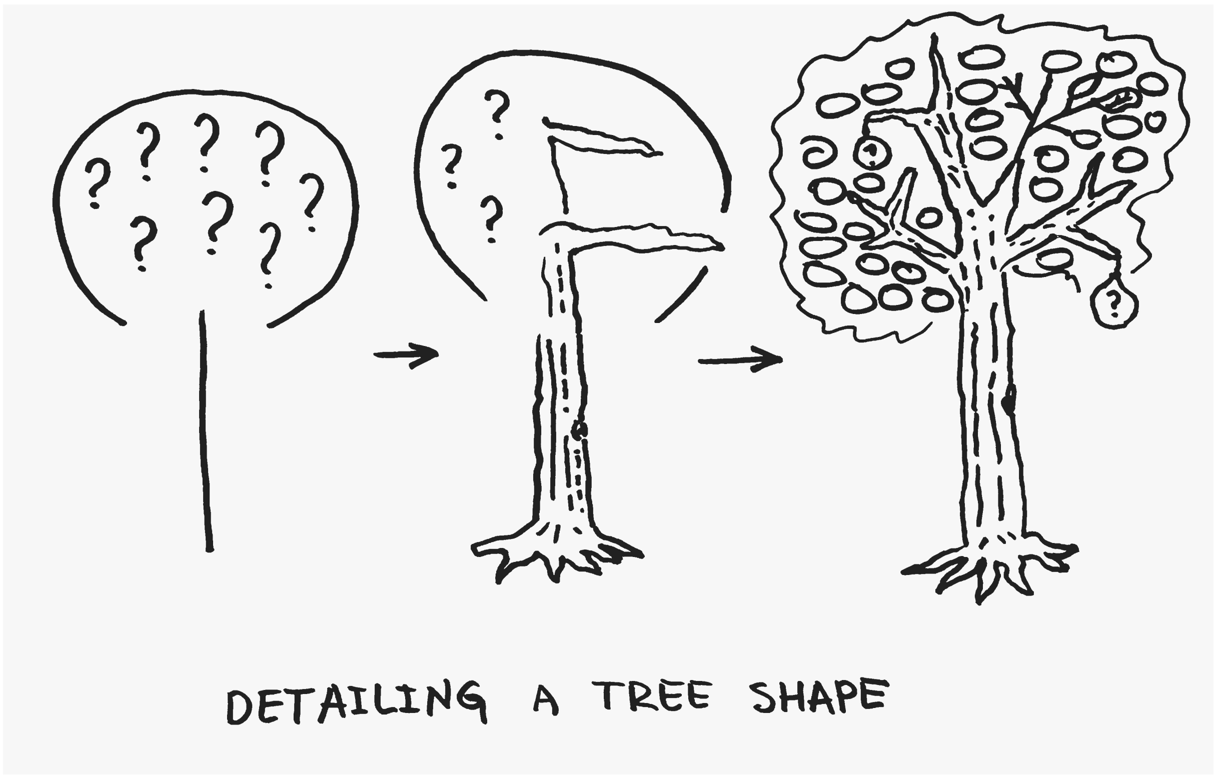

Fig. 3: Detailing a tree shape.

big pieces. The facts that you *discover* while filling in the details of the tree shape end up in your directive framework for that subject, helping you make better decisions.

Learning a new programming language fits well for both of these metaphors. This year I decided to make a cross-platform game in C++ (yeah) so I needed to learn the language and a few libraries I wanted to use. I know a bit of C and C# and I've been curious about C++ for a while, so it's not like I went in blind, but I didn't know much, and certainly couldn't write anything but the simplest program. Knowing that `std::vector` existed was the extent of my knowledge of the C++ standard library.

By virtue of the fractal nature of tree shapes, C++ was just another hastily scribbled twig on the branch "programming languages I barely know" in my head, but I could use everything I learned about learning other languages and add detail to it.

I usually start by reading a thick book to get my bearings and for some languages, there is *the* book that you start with, like K&R for C (even though it's outdated in style, it gives you the structure that you can fill later). The books branch had nothing to hold on to, so I read a few quick start guides first and reviews from experienced people to draw the basic tree shape. C++ has got some positive changes in recent years, and there are several standards that are adopted to various degrees in different industries. You can't use a newer standard than that which the compiler on your target platform supports. C++ is, in a lot of ways, about speed and the efficient use of memory, and is notorious for being complicated. The newer standards make it significantly less painful, but these new "facilities"—a frequent word in Bjarne Stroustrup's books—completely change how you use the language. I read how they differed and watched a couple of conference talks on what it now meant to write "modern" C++, so at least I knew what I needed to look for in a book. That meant some of the better books were outdated, and by reading them I would learn bad habits. That helped prune the list further and I started with Bjarne Stroustrup's "Programming: Principles and Practice Using C++" that was updated for the newer standards.

Reading a book might not be your cup of tea, and I don't blame you. I know many people start with writing simple programs, doing Advent of Code or Project Euler and looking up things they need, like how to use arrays and dictionaries, as they go along. That's fine if you want to write some code and poke at a language. I prefer to learn *how* to write code first, what is available to me in the standard library, and how to use it. I usually start writing code in a new language when I start working on a project, and there is a lot of simple code to write to learn the basics as I go.

Anyway, I usually read programming language books quickly and without doing any of the exercises—they are fun but I can play with the language later. I'm reading to fill the big holes in my drawing and create the initial tree shape. Code examples in the book show you how a programmer should approach problems in the language and some idiomatic ways to write code. Reading a book is interesting, but unless I start using the language soon after, I tend to forget most of the details after not much more than a week.

After finishing the book I researched what else I was missing for creating the types of projects I wanted, and it was relevant for any programming language you would learn: what IDE or editor to use, how you build projects, how to install and use libraries on macOS, Windows and Linux, what libraries were available and relevant to what I was trying to do, and if it was even possible to compile the project for all the platforms I wanted to target (mobile, desktop, and consoles like a stretch goal).

Starting with a project that you want to do gives you an easier time to fill out your tree shape since it will guide what parts of the tree you want to fill out first. It took a couple of weeks to read the first book while researching everything else, but at the end I was satisfied that I hadn't missed anything *obvious* since I had done a good job drawing my tree shape and filling in some of the detail for C++. I was ready to start coding.

The actual process of writing code is guided by the directive framework. Since I was not practicing the language before that, I was very slow. The number of facts and best practices I collected during my research phase was significant, but they weren't clear or automatic enough for me to write anything complicated. They were enough to give me ideas and the general approach I should use, and after starting to write code, I used the references I had gathered to look things up when I got stuck, and debugging the mistakes I made helped me commit the details to memory.

By using this approach, I didn't quickly "jump into coding," and of course I didn't write perfect code. You learn by doing, not reading a book. But what all the research gave me was a glimpse into the expert's head and how experienced practitioners used the language. I think you've heard the joke "You can write Java in any language"? That's what happens when you don't take the time to learn a new language for what it is, instead starting to use your existing directive framework for a language you already know in lieu of building up a new one. It doesn't mean that none of your knowledge of other programming languages is transferable—it is. You will see better how to apply it after you build a solid foundation by first filling out the tree shape. The foundation in place, you will be able to reach the advanced language concepts faster, because you'll build on the foundation for that language, not jerry-rigged bits and pieces on top of the tree shape for another language that you know.

2.6.3 You are not your role

Earlier in the book I've talked about how as you get more responsibility, you step more and more outside of the pure requirements for your role of the software engineer (that is, writing code). Well, this section is not about that at all.

When you have worked in software development for a few years, especially if you are self-taught, it is common to think about yourself as a "web developer," "iOS engineer," or a "Java programmer." As we get more skills and learn the platform we're working on, we start to appreciate its depth and the variety of complementary skills and knowledge that are relevant to this platform. We follow industry blogs and look at trends. We feel in control.

What if after a few years you discover that you're falling out of love with your main technology? The platform may have shifted in a direction you don't like. Your stable job has turned out to be *too* stable and you're bored of writing essentially the same code over and over. You discover that the only jobs available in your industry for your technology are in sweatshops. Whatever the reason, you simply want changes more drastic than updating your IDE's major version every year.

You look out to the job market and your heart sinks. Even in your industry, companies want 3 years experience with a set of libraries completely different than those you've been using at work. They want you to know a different programming language that has quietly gained popularity. In other industries, they want you to know multiple languages at a professional level *and* have hands-on experience with several acronyms. For some, "experience with X" means "long-time contributor to X." You look at your blog—the last post published three years ago, about a library that has been obsolete for two years. You think you will *never* find another job.

I may have painted it a bit too black, but this is close to the experience of someone coming into the software industry as well. The job requirements seem insane. For a junior, two years experience and a list of technologies that doesn't fit on your screen. It's just as daunting if you're a middle engineer in your first software role and want to switch industries.

The reality is that not all is lost. As we will talk later in the section *Reconnaissance*, the job postings are often describing ideal candidates who do not exist, that is, the requirements are a kitchen sink of every technology a company is using. Most often you neither need to have worked with a half of them, nor be an expert in the other half—you'll still be able to do your job, because you won't be productive in your first couple of months anyway and you can use that time to catch up on everything that you're missing. But the more soothing fact is that big chunks of your knowledge are *transferable* if you get it the right way.

Time for another gross oversimplification. Nowadays, a lot of the frontend work (both web and mobile) is taking some JSON, possibly putting it into a local database, and transforming it into a set of controls in the view hierarchy, a tree of nodes. The trend of the last few years is using unidirectional data flow and declarative UI frameworks. A lot of the respective backend work is querying a database and exposing REST endpoints that serve JSON for web and mobile.

Most of the time we work with all that through specialized libraries that make the work quicker and simpler, because nobody likes to write boilerplate. The junior engineers are expected to learn and use these libraries to do the tasks that they're assigned, and knowing the libraries is important, since that's where we do the bulk of our work as software engineers: we're using libraries for the abstractions they provide that let us accomplish the work we're doing.

If you only know how to use a library, you're in trouble. Please don't make a disservice to yourself and learn the concepts and the protocol behind the library you're using. By knowing *what* the library works over and *how*, you unlock the transferable knowledge that you can use when switching to another platform or even another industry. There is a deep learning experience in discovering how the tools that you're using work under the hood.

Let's look at something as simple as JSON. Imperfect but acceptable both for humans and machines, its complete spec fits on a short single page. Did you ever read the spec for it, or relied on how much JSON you had seen to decide if a particular example is correct or wrong? Every programming language has at least one JSON library, so you may have never felt the need to closely look at the format. It just works, the library does its thing and you get objects to work with in your language of choice. Do you know *how* JSON parsing libraries work?

What can we discover by looking at JSON in *learning mode*? Just by reading the spec, we see that JSON's syntax is described by grammar, and learning to read notation like this is helpful when you come across any other interchange formats like Google's Protocol Buffers. We learn about the Backus-Naur form (BNF) that is often used to write programming language grammars, and by practicing reading it, you can understand the grammar used to describe *your* favorite language. Instead of scouring the docs for a rarely-used language construct, you can check the grammar reference now and again and find what you're looking for quicker, because now you can read it.

Since JSON grammar is so simple, there are lots of libraries, and reading the source of some can be a gentle introduction about parsing in general, *why* exactly you need a library, how they differ, and tradeoffs between speed, memory and convenience. You might learn that it's possible to write a streaming JSON parser, that is, one that doesn't have to load the whole payload into memory. On mobile, where your app can be killed by the OS if you use too much memory, or in embedded development, where the amount of RAM is extremely limited, a streaming parser can have a great effect on the speed and responsiveness of your project. We can come across an article such as "An Intuition for Lisp Syntax" that shows how JSON can be used as a "code is data" structure. If you follow that trail, who knows where it might bring you?

Writing parsers, knowing how they work, reading BNF grammars, evaluating the sources of third-party projects are all *transferable skills* that you will use in any industry and in any role as a software engineer. We can learn much more than that if we simply become a little curious what lies behind the default library that the language we work in gives us.

This technique can be used for any library, tool or technology. Learning the concepts behind the tools you're using will not only help you potentially switch industries or technologies later, but will elevate your mental model to a new level. This is exactly the same as what I've talked about earlier—the improved mental model will change your behavior. For example, when debugging, knowing *how* a library works will help you instantly pinpoint the likely failure points and eliminate hours of poking around if you only knew how to *use* a library. You will be considered very smart indeed.

When the time comes to learn a completely new technology, the concepts that you've learned will help you understand the new domain much quicker because, for instance, you'll be able to map the API of a new library to the concepts you already know. Essentially, you'll already be able to manipulate the concepts, and the only thing to learn would be a bit of new syntax.

The longer you work in software engineering, the more you realize that transferable skills are what matters for your career, while specific technologies that you use are transient. It's unlikely that you will use the same libraries and the same programming language in the same way even five years from now, even if you're working at the same company. New, better libraries are developed, older ones are expanded, platforms move to different programming languages (like Android from Java to Kotlin, iOS from Objective-C to Swift).

Experienced professionals whose career spans decades are likely to have programmed in a dozen or more programming languages. As polyglots know, the more languages you learn, the easier it becomes—especially if we look at the history of programming languages. They are much closer to each other than human languages, and the ergonomic improvements from one language often end up in another not long after. People change technologies and tools they use all the time, and it's not such a big deal to learn something new, as long as you look under the surface and recognize some of the patterns underneath.

That brings us to the realization that *you are not your role*. You may be a "Java programmer" or a "web developer" now, but this role does not define you. There's much more that you know and can do in other fields and industries, given some time to learn. Do not despair that you will never be able to start programming something else. As we've seen earlier, even if you "stay in place" and don't make a conscious effort to expand your toolkit, the libraries and languages you use in five or ten years are likely to be different.

Everyone goes through this process, but if you start being more curious about your tools, you can speed it up (and have fun doing it). Knowing that you are not chained to what you do now is a liberating feeling. Experienced hiring managers also know that good engineers pick up new languages and libraries quickly, and if you can show that you can and have done that, you'll be able to find work in another domain or another industry entirely.

Now, let's not get too excited. There are still hard skills and knowledge that you should have if you want to start working in another industry. How do you breach that gap without taking a year or two off from work to learn the required skills?

My approach and advice for changing industries is what I call *transitional roles*. These are a series of successive roles with overlapping skills, where each gives your more skills on your way to the role you want. You know that Wikipedia game where they find the shortest distance from one article to another by following links in them?

Of course, some roles and industries are easier to break into than others, and for some there is no realistic way, so you'll have to take the time to learn the skills by yourself. But for some the transition can be quick. It's also much easier if your company encourages internal transfers. In some companies management understands that to retain good

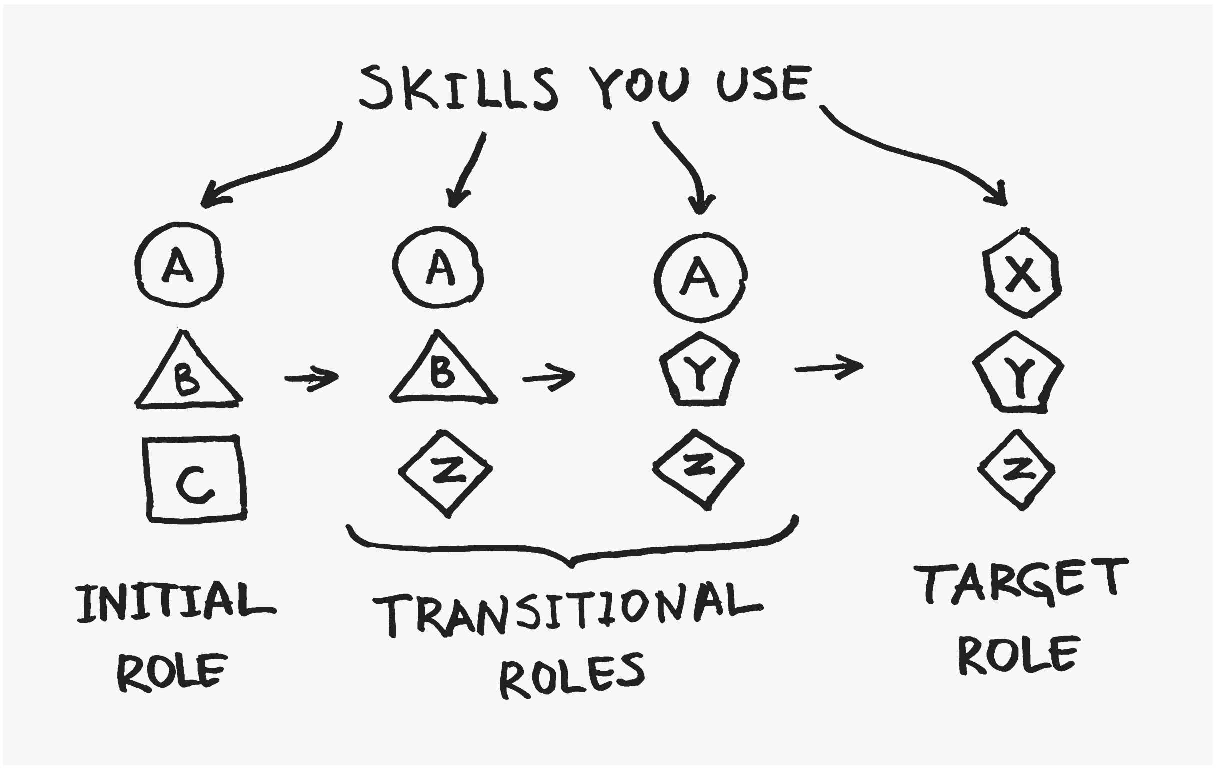

Fig. 4: Transitional roles.

engineers is more valuable as a long-term strategy even if they change roles, than to let them go and hire and train new ones.

Let’s say you work in iOS and decide to work in DevOps. First you could assume build and deployment automation responsibilities which share some skills and tools with the DevOps skill set. If your company was supportive, you could become an “intern” on the DevOps team and either help with simple tasks or gradually take over the infrastructure for mobile projects. After working like this for a while, you will be able to completely switch to perform DevOps duties. The transition would be harder if you needed to switch companies because showing the other company that you can quickly gain the necessary skills is harder—they haven’t seen what you’re capable of.

Another technique that will help you switch roles and companies quicker is to become an early adopter of some new technology, like a new programming language or a new major framework. Since the number of people having *any* experience with it will be smaller, you can do with less experience than for an established technology, and so stand out among other candidates. We will discuss this in the section *Becoming an* *early adopter*.

2.6.4 Development is a team sport

The junior software engineer may have noticed companies looking for “ninjas” or “rockstar developers.” At one point it became so widespread that it became a meme. Ninjas and rockstars work alone, other people are a hindrance. The senior software engineer knows that the overwhelming majority of modern software development happens on teams. There is a shared codebase with multiple engineers contributing, there are other teams that have their own projects and sometimes changes should be synced, there’s management and architects and designers. Like in construction, one person can sometimes do it and design and build a bridge all by themselves, but at the scale and the necessary speed of commercial development, nobody has time to wait for twenty years until one person does it. The company needs the bridge next year. (Now, let’s not start the discussion about “The Mythical Man-Month” and how more people is not always better, that is an exercise for the reader.)

The junior engineers think that their technical skills are what matters most when looking for a job, and they may be right, because juniors are usually not expected to have experience working in a team setting, besides not force-pushing to `master`—sorry, `main`—on a Friday night. But starting even from the middle level, as long as you’re sufficiently proficient technically, how well you work on a team defines your success as a software engineer. You may pass the interviews on technical skills alone, especially if you’re interviewing for a company looking for “rockstars,” but it’s unlikely you will stay at the company for long unless you can work effectively with other people.

What does this mean, exactly? What immediately comes to mind is your manager telling you to “take one for the team” and clock in overtime while taking all the credit and being promoted. That… totally happens, but we’ll discuss company culture later.

Bad freelancers are the perfect counterexample to being a good team player. How are they team players at all if they work alone? The client *is* their team. Software development doesn’t happen for its own sake in a vacuum, a freelancer (let’s call him Frank) is working *with* his client just as the software engineers are working *with* their manager, ultimately beholden to the company’s customers.

What is it like to work with a bad freelancer if you're a client? He doesn't comment on your spec or your references, he says he can do it. When you ask a question, he doesn't answer for three days and when he replies, it's as if he's read a different question, or doesn't give any specific information. He agrees to a deadline or a milestone, and if there are multiple, he misses every one of them, and only emails you after the fact, or after you remind him. When (and if!) he delivers the work, it's missing a critical feature, and is slow and buggy. You email him about it, and you never hear from him again. The other freelancer (who happens to be good—let's call her Gwen) says that's the worst code she's ever seen and rewrites it from scratch in half the time. In other words, working with a bad freelancer is the source of endless frustration for you, the client. Perhaps you've had coworkers like this? Imagine how frustrating it is to have someone like that on *your* team. What if that's you? `*le gasp*`

The "rockstar" engineer (they are always male, so let's call him Rick) is more dangerous because he's actually good. Silicon Valley startups keep looking for that 10x developer, and they do exist. In Joel Spolsky's post "Hitting the High Notes" there is some data supporting the fact that some engineers are much, much faster than others, and produce code that works great. In the startup world, where speed of execution trumps everything and technical debt will never need to be paid off unless the startup gains traction, hiring an engineer like that makes a lot of sense, even when they cannot work with other people.

But when the company grows, one person can't do it all, and the "rockstar" ends up working on a team, and that's where trouble begins. He considers himself the smartest person in the room (often, rightly so) and dismisses the other engineers' suggestions or approaches, demanding that they do as he says. He can be rude and condescending, but the team and the company has to keep him since he has such a big impact on the product. His code, even if it's performing well, can be hard to read, obscure and complicated (and he may produce more of it every day), a nightmare to maintain. Eventually he may end up working on some business-critical, but isolated components that nobody needs to ever touch again (since nobody wants to talk to him), before getting bored and leaving to bootstrap another startup. I'm sure Paul Graham has an essay praising this type of person, but I can't be bothered to look it up.

Companies that care about retaining and evolving their engineers and have long-term software projects—companies you want to work for—are specifically checking for this aspect in their interviews. That's why you're asked questions like "How did you handle conflict?" and "Tell me about a time when X." Even if a candidate was extremely skilled technically but unable to communicate in a team setting, it would likely be a hard pass. Nobody wants jerks on their team. You're a professional, so act like one.

I'll say it again: being a good team player is a prerequisite, just as your technical skills, to *be* on software engineering teams that value and nurture their engineers, places where you will grow and help others grow. Being able to learn effectively is a trait of good software engineers, so if you show yourself to be capable of learning quickly, the technical requirements can be significantly relaxed. But nobody expects that you unlearn being a jerk. There is no second chance to make the first impression, so having the qualities of a good team player will be what wins or loses you the job.

What are those qualities and what does it mean to be a "good team player"? First of all, it's acting like a decent human being. (You *are* one, right?) We have seen in the previous examples how you shouldn't act if you want to be considered a professional, and I will talk about it in the section *Considerate communication*—things that you can do at any stage of your career. If you start doing that, you will immediately improve your communication up a notch and other people will like dealing with you. For code, this will be the section *Code mimicry*, and this is also something you can start doing at any time.

As for behavior specific to a team, there is a project called "Contributor Covenant" that defines the code of conduct for open-source projects. The central pledge is "to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community," and the same applies to building an engineering team with the same qualities. Some examples of this positive behavior, quoted from Babylon Health's "Code Review Etiquette", but it's really about any team communication:

- Avoid selective ownership of code ("mine," "not mine," "yours"). Prefer referring to it as *our* code.
- Use "we" instead of "you," unless asking for someone's opinion ("What do you think ...?" but "Here we are doing ... What if we ...?").
- Do not expect your comments to be answered immediately. If you need an answer fast, talk directly with the person (the same applies to instant messaging).
- Accept that it's fine to have disagreements and it's hard to please everyone.

For me the most profound has been the shift from personal to shared ownership of code, and using "we" when referring to shared goals. A simple change, but consider going from the mental model of a product, assaulted by competing pull requests that sometimes change the parts you've been working on, to one where the product is owned by the whole team, and every code change is seen, first of all, as an improvement of the shared whole. Where in the former model engineers would dunk and even insult each other ("my" code versus "their" code), in the latter the engineers' intent is to treat their colleagues' code as if it were theirs, and to offer positive, constructive critique instead of ridicule.

We've got a nice fuzzy feeling in our stomachs, but if you want to improve as a software engineer, do you *want* to be on a positive team? Does all this positive communication and inclusiveness translate to better code and a more successful product? Besides clearly letting you focus on work and not petty workplace squabbles, a positive working environment, according to an article about employee empowerment in Forbes, strongly correlates with increased feelings of empowerment and engagement, which results in better decisions made, more creative work solutions and better results for the business, which, in turn, means that the product is better in the end.

This positive team environment is conducive to learning and improving yourself, since instead of being judged you'll be encouraged. Improving our skills and getting more knowledge is exactly what we want, isn't it? If we get stuck, which inevitably happens, more so as juniors since we don't know what we're doing, we can take from the team what we bring into it. We can directly ask for help, and this is the fastest way to learn things that we didn't know about the codebase, the project architecture and more. We only need to ask, and someone who understands the project better will explain how the part that we're struggling with works, and why. Immediately getting expert help is the dream.

Let's go back to the beginning. Where a junior software engineer might imagine commercial programming as a group of loner coders, finding flaws in each other's code, the senior engineer knows that a high-performing software development team is an environment focused on mutual improvement, where people are empowered to do their best work without hand-holding, don't hesitate to ask for help when stuck, and treat their product as a shared effort that everyone owns.

2.7 Know your place

The last thing to find out is to discover where *you* fall on the scale. The gap between a junior and a senior engineer is not only about pure domain and programming knowledge (the *what*), a lot of it is also the senior engineer's behavior when developing software (the *how*).

For a start, let's get the question of domain and general computer science knowledge out of the way. A hill I will die on is my strong belief that there is not one piece of knowledge that is critical for each and every one of the software engineers, at any level. The beauty of software engineering, and what makes it so accessible to people from so different educational backgrounds, is that you need very little hard knowledge to start "tinkering" and go from there. Undoubtedly, general knowledge is important as your projects become more serious, but as long as you're willing to learn and are aware of what you're missing (and know where to find the details when you need them), you'll be fine. The real liability is the software engineer who does not want to learn.

Seeing tables like Sijin Joseph's "Programmer Competency Matrix" exposes this domain-agnostic approach to measuring knowledge. A "Level 3" engineer is supposed to have knowledge of data structures like B-trees, red-black trees, tries and more. I can honestly say that after ten years of programming professionally, I don't know how to invert a binary tree, because in the industries that I've been working in it's simply not something you do. If I needed it, I'd look it up. I don't remember the SQL syntax because I haven't written any SQL for years, despite it being on the list of What Every Software Engineer Should Know™. Note that it doesn't mean that I don't know *anything* about SQL. I know enough to learn more when I need to. Instead, I'm intimately familiar with the platform I'm working on and the theory behind it. At the same time, a software engineer just starting to work on any project involving efficient search would need to know tries and red-black trees—because that's what the knowledge domain requires.

I'm not ashamed to admit that I would promptly fail a whiteboard interview for "real" software engineers because of the gaps in my knowledge. First, because I'm aware of them—I know what I don't know—and second, because I'm confident in the knowledge I *do* have and that it is both relevant and sufficient to produce effective code for the platform I'm on and the industry I'm in. If I wanted to change industries and programming languages, I would use the process I described in *Exponential learning* to identify what was required and fill the gaps in my knowledge.

The most comprehensive—and unreadable—attempt at collecting and presenting *all* available software development knowledge is the “Software Engineering Competency Model” (SWECOM) produced by the IEEE Computer Society. In its 168 pages you’ll find everything from theory to software lifecycle and even human-computer interaction skills, split into five levels: Technician, Entry Level, Practitioner, Tech Lead and Senior Software Engineer. It is high-level, more of an overview, but you can use the gap identification worksheets it provides to get a rough idea of where you are on the software development map, and use the terms as pointers to learn more and how the gaps apply to your particular industry (and if they apply at all).

I’ve said in the beginning that the difference between the junior and the senior software engineer was in the knowledge they had and in their behavior. Let’s try to expand our definition in a way that you can use for inspiration. The junior software engineer needs guidance. The senior software engineer has enough domain knowledge and skills to reliably perform any software development work a project may require (from planning to release and maintenance) by producing efficient code that fits well with the project. If necessary, the senior engineer could work independently and find out whatever was necessary to perform the task on her own, but by asking the team for help and helping others improve, she’s creating an open and honest environment that lets the team as a whole perform better. (That didn’t turn out quite as short as I imagined.)

In other words, the senior engineer is simply a great engineer that others can trust and rely on, and there is abundant research that we can draw on to see what makes a software engineer *great*, versus *ordinary* or *beginner*, and what skills the fresh graduates are missing. Two papers that I found the most useful were “What distinguishes great software engineers?” (2019, by Paul Luo Li, Amy J. Ko and Andrew Begel) and “Investigating the skill gap between graduating students and industry expectations” (2014, by Alex Radermacher, Gursimran Walia and Dean Knudson). Actually, there was a third one: “Towards a theory of software development expertise” (2018, by Sebastian Baltes and Stephan Diehl).

How does this all help us understand what the senior engineer knows and can do that the junior doesn’t? Let me make a list of characteristics—based on my own experience and the research above—that make a real difference, and how you can bridge the gap.

2.7.1 The senior engineer has sufficient knowledge to work independently

The main reason why the senior engineer is so reliable is that she has the ability to write the code for the whole project on her own. This entails several things at once, but the central one is that she has enough knowledge to start working on any part of the project without getting blocked, and if she does hit a snag, she can independently identify what's missing, find the information and get the task done.

We have touched on the general map of the knowledge that you may acquire as a software developer, but here come the specifics. What knowledge, exactly, does a senior engineer have that a junior doesn't in your industry and on your project? We can describe it as *the knowledge that is required to effectively solve all of the typical tasks that you can get in this role*.

You can make a detailed map of this "sufficient" knowledge, first of all, by asking the experts in your company. I'm sure they will be happy to describe in detail what exactly you need to know to successfully solve any task that you may work on. That will become your curriculum to learn and practice, and by getting these skills, you'll become self-reliant and will be able to work independently.

For your industry in general, making a map of this knowledge is more difficult because of the differences of the approaches of different companies, but you can still do a good job by closely reading *job postings* and *interview questions* for the senior roles in your industry. Job postings frequently contain all the tools and frameworks that the engineers work with. The interview questions (not the general software development ones, but those specific to the role) contain all the key concepts and advanced techniques that the company is looking for in their senior engineers. By making this map, you will discover what you need to know and be familiar with to be able to learn any other details that come up on your own.

Why do I keep saying "work independently" when I've made such a big fuss about being a good team player? These concepts are orthogonal and complement each other. If you are unable to work on your own (you get blocked often and cannot move forward without the team's help) then the team has to spend their time helping you, and you are unlikely to help anyone on the team with *their* problems. Conversely, if more people on the team are self-reliant and can work completely independently, the team spends time unblocking people less often, and you are more likely to help others by virtue of having more knowledge. This way the team as a whole can move quicker.

Having this knowledge is also a matter of professional pride. Other people in the company seek the expertise of the senior software engineer, and you want to be able to answer their questions and give advice. Saying "I don't know" is fine, as long as you add "but I can find out."

This expertise—the ability to know or find answers to other people's questions and to work on all implementation or debugging tasks that may come up in the course of a project—is what distinguishes the senior software engineer from the junior. Others know that they can depend on the senior software engineer to get the work done. By making the knowledge map and working towards acquiring it, you will come closer to being self-reliant and dependable.

2.7.2 The senior engineer writes good code

The crucial trait that gives the senior software engineers their title is their ability to write "good code." What specifically makes the code "good" is the subject of many discussions, and can be *extremely* different from company to company, even in the same industry. We will also talk about this later in the section *"Elegant" code*.

Code does not exist in a vacuum, and if a software engineer writes beautiful code that does *the wrong thing*, it doesn't matter how beautiful or performant it is. Code exists to solve a problem, and as we've seen earlier, sometimes the best thing one can do is not to write any code at all. If a senior software engineer writes code, then what it does first is it *solves the problem* (implements a feature, fixes a bug). Only then does it have merit as "good code," otherwise it's useless.

Even though everyone argues about specifics, research shows that expert-level programmers define three traits that make code good: it has clear structure, it is maintainable, and it is performant. The opposite is the textbook definition of "spaghetti" code: slow, tangled, and unmaintainable. Let's look at each of them in turn.

Clear structure of the code, like clear writing, reflects the clarity of thought. Code that is clearly structured is more likely to have a well thought-out (and correct!) design than code that is disorganized and hard to read. Structure doesn't only mean how neatly the files are organized in a directory, it's also the construction of interfaces and functions, clear variable names and consistent formatting. Well-structured and uniform code is much easier to read for humans (the compiler doesn't care), and that takes us to the next trait: maintainability.

The only constant in software is change. Maintainable code means code that is *expected* to be changed, and makes it easy for the next person to change it. Maintenance doesn't sound interesting, but this is simply an umbrella term for the continuous improvement of software. Maintenance is not only patching legacy code, it's also implementing new features in an actively developed codebase while running the project in production. Any software is being worked on while it is used. The software that doesn't get any changes is likely unused and dead.

The senior software engineer writes code that is expected to be changed with minimal effort. That is achieved (for example) by decoupling so that one object can be changed independently of another, using small pure functions so that you don't have to think about mutating global state, separation of concerns so you don't have to change surface UI code if you change a part of deeply buried network code, and a lot of other techniques and tricks that all make code easier to change. More often than not, code written using these techniques is also simpler because the context in which its components work is smaller and easier to reason about. When the next person—this could be yourself in six months when you don't remember anything about it—comes to change the code, they perform the task quickly and move on without untangling any spaghetti.

If we start the project with code that is easy to change, and every change we add is simple and also primed for change, then by induction after many such changes the project will still be maintainable. This is the core of the approach called "Simplicity-Oriented Design" embodied in the very successful open-source project ZeroMQ.

On the contrary, if we add code that is complicated and hard to change, after a while the project will become *unmaintainable*, that is, extremely hard to change without breaking something else. These complicated and hard-to-change additions are what we call technical debt, code that is known to be bad, and maintainable code is the opposite of it. I like the sound of *technical investment*, because unlike debt, you don't have to pay it back in the future, it only brings dividends shared equally by everyone on the team.

Performant code simply means code that only does the necessary work. In rare cases, when the performance is critical, we need to use magic tricks and hardware-specific hacks, but for common applications, writing plain, idiomatic code while using appropriate data structures is enough. Computers are fast, and you don't need to introduce hacks anticipating a bottleneck before it's proven to exist.

Benjamin Franklin said: "If I had more time, I would have written a shorter letter," and that is broadly true of the well-structured, maintainable and fast code of the senior software engineers. Rarely does code like this get written on the first attempt, and even if it is, it comes from experience and thinking through the problem. More often it is the result of invisible iteration. Kent Beck's quote "Make it work, make it right, make it fast" reflects this process well. The first version of the senior engineers' code, and the second, might just be barely functional. What makes them senior is they don't stop at that and produce the final version that has all of the traits that make it good code.

On top of that, in research two attributes of great software engineers trumped all others: paying attention to coding details (with examples being error handling, memory, performance, style) and being mentally capable of handling complexity (the ability to comprehend multiple interacting software components).

These two attributes are directly responsible for producing "good code" that the senior software engineer is praised for. The junior engineers' code is often inconsistent and missing critical details specifically because they did not pay attention to some details, while in the senior engineer's code, every line is accounted for and the implementation completely solves the task.

Again, it doesn't mean the senior engineer's attention span is longer or better than the junior's, but her attention is more focused on the final version of the code change. It is the same with handling complexity: the senior engineer often has all the software components of a project in a working mental model that she uses to anticipate the effect of her changes on the project as a whole, while the junior engineer can only think about a few components at a time.

I've read somewhere that as musicians learn their craft, it becomes easier for them to remember music because they start thinking in musical phrases instead of individual notes. It's similar to how a software engineer thinks about a large software project. The ability to consider multiple software components at a time is not exclusive to the senior engineers, it comes with practice, just as knowing where to pay attention in a code change so that it's as coherent as possible. We will discuss how you can deliberately improve in this area in the section *Code mimicry*.

2.7.3 The senior engineer knows the software development process

New graduates and junior software engineers have little to no experience with the process of commercial software development and when they start a job, they quickly learn how much they're missing. Writing code is the crucial skill of a software engineer, but in the scope of a software project and shipping a product, writing code takes only a small part.

Susan, our senior software engineer, is keenly aware of her place (and her impact) in the process of creating software and continuously getting the product ready for release. To put things into perspective, let's take a brief and very simplified look at how the software sausage is made, from the perspective of a software engineer.

Let's say the user researchers have identified a new customer need, and Engineering is tasked with solving it. Software engineers can be involved as early as this stage, because as experts in the platform and the inner workings of the product, they can help the product owner and designers decide which approach to take, and its tradeoffs. For example, a backend engineer would be able to say what data was available at which stage, and how quickly it could be retrieved. If some query was known to be inherently slow, the approach using it could be discarded early. Without any software engineering input, a "new feature" cooked up only by Marketing could be completely infeasible and would make the product, the user experience, server costs (or all of that at the same time) worse.

After the approach has been chosen, a new feature is fleshed out and split into tasks. Again, the expertise of the software engineers who are going to work on it can provide a better direction early and save a lot of unnecessary (and expensive) rewriting later.

Then the software engineers do their work implementing the new feature. Depending on how the team is using version control, it can be completely separate from the main project until it's complete, or added bit by bit in a way that is walled-off from the customers. Tests are written, and code is reviewed to be "production-grade," primarily meaning that it's efficient and all the possible errors have been handled.

The feature as a whole is deployed to a testing environment for, you guessed it, testing. Acceptance testing—which compares a feature against a set of pre-agreed criteria—is often performed by people *other* than the software engineers themselves, both because it can be time-consuming (there are a lot of permutations of configuration, etc.) and because the software engineers are myopic when verifying their own code. After several rounds of finding and fixing bugs, the feature is ready for release.

The release is usually the same code being marked in version control and deployed to yet another environment: production, the one used by customers, with real data and often higher load. Inevitably, some more bugs are found and categorized, from trivial to critical, sometimes resulting in the whole feature being rolled back, or disabled for a portion or all customers by a remote feature switch (a setting in the backend that controls whether users can see a particular feature in the product or not).

After the feature is released, the team switches to the next one, while fixing issues from previous releases. Depending on the number of developers and the size of the project, multiple features can be developed at the same time, all being in different stages of development, all in the same codebase and often touching the same code, so another team changing code your team relies on can mean new bugs in your feature :) On and on goes this process, and the codebase lives for years, or even decades.

This is my best approximation of the software development process in product companies, so your experience in your industry may be slightly (or vastly) different. There are many approaches to developing large projects, but the one I've described, I think, is the most common. This process is not without flaws, and I'm sure many companies have their own.

The process is not a ritual inherited from ancestors and revered (as some Scrum cargo cult meetings may feel), but something that should be tweaked or changed completely. Your duty as a software engineer is to the customer and the product, and the development process is just a tool, not a temple. Raise it and build it anew if your team feels the need to. For example, the team at Basecamp have done just that and named their process “Shape Up.”

The junior software engineer or the new graduate don’t have experience with any of that. They have only worked with what we may call “side projects”—software written from scratch by one person, often released once and never worked on again. Code quality was never an issue, and the extent of using version control was making some commits. No wonder they struggle when thrown in the middle of an established process in a company.

According to research, the skills found to be most lacking in new graduates were communication, knowledge of environments, version control, and testing. That confirms my own experience. Let’s look at each in turn and see how you can catch up.

Communication is the first and trickiest. The junior engineers don’t know how to talk about their work, and the work of others. In the section *Development is a team sport* we have seen how to improve communication within a team, but there is also the matter of writing good emails, and various documentation, and explaining your ideas during meetings, and talking to other departments in *their* language. We will look at this in more detail in the sections *Help your company teach you* and *Considerate communication*.

Knowledge of environments (testing, staging, pre-production, production) and how the code that you write is deployed, and in what configuration, is fundamental to the senior engineers’ ability to predict how their code will perform in the real world, and what kinds of errors may happen.

Just like a piece of software is never “finished,” a large project with many moving parts doesn’t exist as a single “version” that everyone is using. On the contrary, it is a sprawling web of versions that not only have different code, but their configuration and available data is also different in different environments, and on each of the engineers’ machines because they’re all working on different features or fixing issues. Understanding and predicting how a certain code change will perform takes the knowledge of how it propagates through this web.

This concept is, of course, closely tied to using version control. For the senior engineer, version control is two things: the first is the actual version control system that the company is using, and the second is the meta-level of code changes.

The version control system, for better or for worse, is most likely to be Git. You may

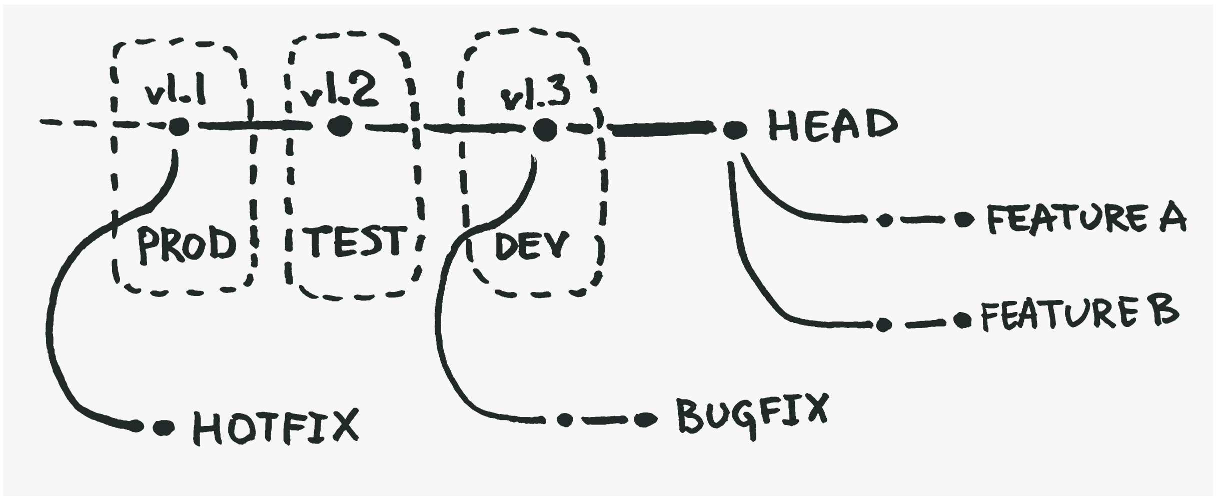

Fig. 5: Environments and product versions.

not like it, but it won (for now), and until something better comes along, we are going to use it. The best we can do about this situation is to learn the concepts and some inner workings of Git so that it's easier to understand and remember the commands, and what they are doing. The simplest way to do so is to read the free book "Pro Git" and use it for reference later. For the junior engineer, who has only worked on small projects in isolation, going from the occasional `git commit` to the commercial usage of Git can be daunting.

The senior engineers, especially if they are working on a release, will work on the level of code changes, rather than individual "in-progress" commits. Code changes can be sets of commits, or (squashed) individual commits, but in the end they are units of work shipped by software engineers to be integrated (merged) into the project. These code changes are the building blocks of the product, and a set of them, put together from the very first line, *is* the product.

In different environments, different sets of these changes coexist, and to assemble them, the software engineer needs to join them together, which often causes conflicts because they are touching the same code. They need to be marked and tallied to be included in the next release and the changelog. Afterwards, some may be rolled back, and the coexisting versions of the product with different sets of these changes may need to be reassembled in a different order or joined.

Working with changesets, the meta-level, can be done over any version control system, but in order to perform this work, the software engineer needs to be skilled with the particular system that the company is using. That's why you can often see on the job postings that version control experience is required, but with any version control system, because the meta-level works over any of them, and you can learn another tool to perform the same process when necessary.

2.7.4 The senior engineer enables others to make decisions efficiently

Since the amount of knowledge the senior engineer has about the project and the domain in general is high, she uses that knowledge by helping others answer questions and solve problems, often proactively, because she's on the lookout for inefficiency in the company's process. This is the opposite of passively waiting for someone else to tell you what to do.

To make a decision is to resolve ambiguity. Theodore Roosevelt said: "In any moment of decision … the worst thing you can do is nothing." A variation of this quote is popular in business, since the only certain thing in business is uncertainty. Perfect information to make a decision will never be available, there are too many factors and too many unknowns. This is also true even at the level of writing code. Junior software engineers can spend days or whole weeks trying to decide which implementation path to pick. Designers will make multiple prototypes of a feature, some of which could be eliminated early. If only a project expert was available who could help them make a decision quicker… The senior software engineer is that expert, lending her knowledge to those who need it.

By being proactive and helping other people answer their questions and make decisions quicker by literally being the expert around whom they can ask, and giving advice when she notices problems, the senior software engineer can save an enormous amount of the other specialists' and her fellow software engineers' potentially wasted time. Wasted time directly translates to slower product development and lost profit for the company.

The senior software engineer knows this even from her own work. Writing code is an exercise of balancing tradeoffs and constraints. Accommodating every constraint and external requirement involves making a decision. Making these decisions can have high costs for the project, since the senior engineer's impact on the product is high (that is, the consequences of her decisions can be dire). She is actively looking for anything that can de-risk these decisions by providing supporting and convincing information—an expert in the problem domain is the best source of this help and advice. It's like having your personal Google that knows the answers to all of your questions. Knowing that, the senior engineer seeks to offer her help to everyone around her who is in need of it or in denial.

2.7.5 The senior engineer is continuously improving

We have already seen in the section *You are not your role* that even if you don't make a particular effort at improving your repertoire of skills, with time your role will change naturally, most likely towards what you're better at. This is a low-resistance way of improving your skills by simply using them. But, by doing that, you will likely not move far in your software development career, or do it quickly. If that has never been your goal, that's fine. That's less stressful and rational, but I personally find it unexciting. Besides, we're here to learn how *great* software engineers approach their work and how we (the merely *ordinary*, I'm not pretending I'm smart) can try acting the same.

Someone said: "One engineer will have five years of experience, and the other will have one year of experience five times," describing how different people improve at work. That's a keen observation—it's also been around for decades, in different industries—and it's also true for other skills: playing the guitar, running, writing. After reaching a certain level of competence that is sufficient for work, we have to consciously apply effort in order to improve our skill further.

This is very evident in learning a foreign language, especially in immigrants. After a few years, most people learn enough to be competent and stop improving. You can hear their original accent a decade and more later, making the same mistakes, even though they are speaking their new language every day. They are not *trying* to improve and so they don't. Good for them. Who likes extra language lessons?

We know from the research of K. Anders Ericsson that experts become experts and reach the height of their performance by engaging in *deliberate practice*—a tight loop of practicing a particular component of a skill with the desire to improve, followed by feedback, changing how they perform the skill and practicing again. You can learn much more about it by reading the book that summarizes 30 years of his research, “Peak: Secrets from the New Science of Expertise” (co-authored with Robert Pool). The 10,000-hour rule, popularized by Malcom Gladwell in his book “Outliers,” was based on Ericsson’s research, but oversimplified it.

Well, most people have never heard of deliberate practice but still improve and become great software engineers, right? That’s true, partly because software development is a skill that is itself similar to deliberate practice. When programming, we have a mental image of what we want to achieve, we write some code, run it and get immediate feedback, fix any errors or add more code and run it again, and so on. This loop of trying to achieve a result with almost instantaneous feedback that we can immediately act upon *is* deliberate practice.

As our projects become more complicated and we get more responsibility, and as we get more competent, we engage in this loop much less often than early in our careers, and almost never do it on purpose. Invariably, our progress slows down. Also bear in mind that even the sheer volume of programming we do helps improve our skill, compared to how bad we’ve been in the beginning.

So, people are improving unconsciously. But remember how I’ve said about the average number of years (five or more, depending on the industry) it takes to become a senior engineer? If we consciously engage in deliberate practice, we can speed up this process and grow *more* than if we haven’t. Some people do it, and companies that care about their software engineers encourage it by giving them learning opportunities, more responsibility and stretch projects, specifically designed to allow the engineers to work on growing their skills.

Finally, working to improve your skills is more fun because it gives you a moving target, or rather, a series of targets that you keep hitting, which is good for motivation. And, what’s even better, you don’t have to spend your free time doing that, you can do it at work. Deliberate practice involves a skill you already have—programming—and you can do it while working, these are not separate exercises that you do during lunch, or at home.

As Patrick McKenzie says, you *can* work on cutting-edge technology and become a great engineer, all by working nine to five and being home at six. Even if you want to improve quicker, you don’t have to work on side projects or code in your free time if you don’t find it fun. In the long run, having an active life outside of work and relaxing will help you focus and learn better when you *are* at work.

3 What you can do now

Phew, that was a lot to process. The senior software engineers, especially if they're *great* engineers, are expected to know and do so much. But even knowing the destination, it's hard to imagine how to get there. What can you do right now that will set you on the path to leveling up?

You can see some advice in this section as making an impression of a better engineer, rather than actually getting better. Impressions and how they've felt are what people remember about you. You can feel that you are "deceiving" people or "wearing a mask" by changing what you would say otherwise, but you are not—you are simply being professional. Doing the work to make an impression does make you better because you do additional research, think a few steps ahead and read others' code to improve yours. Remember what we've talked about in the section *Development is a team sport*. Cultivating a positive team atmosphere takes work on everyone's part, and it's good for everyone in return.

The advice in this section will help software engineers of all levels to become better members of their team, while both improving their technical skills and making other people's work easier. I'm not telling you what exactly you should learn (well, maybe a little bit), but rather, how to give yourself more space and potential to grow, so that you can get to your destination quicker.

But first, you need to decide what you want.

3.1 Where do you want to go?

The biggest threat to your improvement as a software engineer is not knowing what you want. That doesn't mean you have to decide and commit to a 30-year career (just yet), but knowing your preferences and options for the short term (months) and the longer term (years) can do a lot to steer your efforts in the direction that *you* find meaningful.

This is all part of the navigation theme I've been bringing up. "Going with the flow" and just working at your job may feel less stressful, but often it's like drifting in the sea, you can go this way and then that way. It's in your best interest to at least check occasionally that the direction you're going is aligned with your interests.

What if you do not want to learn anything and just want a stable job? Maybe after reading the previous sections you've decided you're disinterested in software development in general and want to do something else? That's… fine, I guess. There is space in tech for everyone, and it's not your obligation to be excited about your work. I'll only say that being disinterested and unimpressed, even hostile, can be a sign that you're too tired. Chronic weariness is common in software development, and we will talk more about it in the section *Burnout*. If you have ever felt excitement about learning programming, this feeling can be brought back.

Software development in general is just another job (no matter how much some programmers consider themselves superior to other people), and it's not fun all the time. In fact, most of the time it's tedious, and if you don't like code and programming, work can be seriously boring. But some programming jobs are far more boring than others, and by following some of the advice that follows (yeah…) you can have a better time at work, or change your role or company to something that doesn't feel so tedious. Being on a good team can make all the difference.

With that said, I assume you *want* to improve, and before going further, you'll be well-served by trying to answer some questions that will help you establish direction. In my experience, having at least a vague goal that you're drifting towards is better than having none since it gives an anchor to your efforts and lets you decide what you do or don't do based on whether that moves you closer to your goal or not.

We'll use both positive (what you want) and negative (what you don't want) questions because some people find one type easier that the other.

- What are parts of your job that you like and don't like doing? We want to do more of what we like and possibly leave what we don't like to people who are better at it.

- For parts that you don't like (like "designing a new API" or "giving public talks"), do you not like them because you're bad at them and don't like the feeling of being inadequate? Imagine if you were good, would you feel the same? Whether you dislike the activity or your own inadequacy is an important distinction explained in detail by Josh Kaufman in his book "The First 20 Hours." In short, you can learn to be better and discover that you enjoy the activity—being good feels good.
- When writing code, what are the areas you're struggling with most often? These could be design decisions, being confused by library APIs, difficulty coming up with what to test, and so on.
- During code review, what kinds of comments do you get? Do you fail to include some details, is your code structure confusing, do you write too many comments or too few, does your style match the project? Code review, as long as it's constructive, is a great source of ideas for improvement because it's other people's assessment, not just something you think.
- Are you happy with the industry you're in and the product you're building? Do you know if the common ways of working in other industries are different from yours, and if your project follows good engineering practices? Are you empowered to change the process that your team is following? Note that having no process is a process too.
- Do you have a positive environment at least in your team? Are engineers encouraged to learn? Being on a toxic team, or led by a toxic manager can make you feel trapped and useless, but as we've talked in the section *You are not your role*, it's in your power to change your team or switch to another company.
- Are you satisfied with your level of income, your working conditions, is there a clear track and timeline for a promotion?

Given these examples, I'm sure you can come up with more things to consider that are important to you. The baseline is simply to become better at the work that you do and treat people around you well. Even by doing just that, you will improve faster than someone who doesn't have any goals at all, nothing to look forward to. Work can be more than a place where you come to spend the day.

3.2 Help your company teach you

Learning is at the core of being a great software engineer, as we've seen in the section *The senior engineer is continuously improving*. The job description of the junior software engineer is to learn and gradually take more responsibility and have more impact by working on bigger tasks. The company as a business benefits from educating its software engineers so that they can create a better product for the company's customers (and let's be honest, compensation often lags behind responsibility so it's also cheaper).

But the company is often inept at teaching, and the software engineers themselves are not doing a good job learning. The company, because unless there are people who are specifically working on improving teaching, the process goes downhill, and the software engineers, because, as we know, many software engineers think their job is to write code.

Here's a common scenario when the company hires James, our junior software engineer. He is given a couple of "simple" tasks and told to ask questions to his "onboarding buddy" Betty (a senior software engineer) if he has them. If he is lucky, there is some outdated project documentation that he can read. The senior engineer is a bit shy and has a lot of work of her own, so she also tells the junior engineer to ask questions when he has them.

The junior engineer spends the week trying to code a solution to the simple problem. On Friday he has a short meeting with the senior engineer to check in on his progress. Betty looks at his code which is not finished and has a lot of temporary hacks (because James is trying to make sense of the project) so she gives him a couple of pointers, and suggests to use the same solution like in that other module.

The junior engineer spends the next week trying to understand how the other module works and adjusts his code. Next Friday the senior engineer asks if he has considered this and that (non-obvious cases that are impossible to discover) and asks him to handle them. When next week the junior makes his first pull request, he gets enough code review comments to spend another week fixing them.

The juniors often don't know what they are supposed to learn, or to what end. They are given as much time as they want to work on their tasks, because they are "learning."

Fig. 6: Junior Software Engineer in an average small company.

They get stuck and can spend days looking here and there for a solution, and never asking other software engineers for help, because they think they're supposed to figure everything out themselves. The seniors are too busy working on their own tasks to pay enough attention to steer the juniors in the right direction and give them enough information to get unstuck. The senior engineers aren't *managers* after all, they aren't supposed to know this stuff, right? This sight is so typical that it's ended up as a meme of a dog walking itself.

Senior software engineers can also become myopic to the benefits of talking to other people and asking for help sooner. Earlier in my career, I have left companies over lack of learning, but looking back, I understand that I could have had more learning freedom in a familiar project and in a company that trusted me.

Let's see how we can reclaim time before it's lost, from the small things to the large.

If we had perfect knowledge, by definition we would be perfectly equipped to answer all the questions we may have and make any decision. We don't, and we are left to either get the knowledge ourselves, or ask others to give it to us. If we tried to get all of the necessary knowledge from others, they would feel they are making all the decisions for us, and we just take their (valuable) time. So we must have balance between looking for knowledge and asking for it. *How* we ask for help is also important since it shows our peers if we have done any work first.

Let's imagine software development as a series of tasks you should perform by writing some code. This is generally true for any level of software developer. As you begin working on a task, you start asking yourself questions because you need to make some implementation decisions. Debugging is a little different, but can also be reduced to a series of questions. Senior engineers tend to have higher-level questions ("Do I extract some functionality I need from another module for this new feature I'm writing?") because they can answer the simpler, lower-level questions that the junior engineers have ("Is there a function to do X already implemented in the codebase, or maybe I need to do Y instead?"). Junior engineers usually have more questions they have trouble answering, so they get stuck more often.

Getting stuck is the key concept here and your main indicator of when you need to seek help from somebody else. My rule of thumb is that if you don't make any progress for about an hour, you are stuck. "No progress" means that you haven't written any code because you are unsure what to do; you've written some code, it hasn't quite worked and you don't know why; you've been debugging a problem and don't understand it better than an hour ago. The common thread here is that you are unsure or don't know something, that is, you have a question without an answer. Often you don't know what the question *is*.

But noticing that you are stuck is not yet time to ask for help. First, we must try to bring some clarity by asking ourselves a few questions. Here are some that I ask myself:

- What exactly am I trying to do? If you've been "trying to make your code work," this can help you snap back to your actual problem and notice that you are in a dead end, the question you are trying to answer is unnecessary, and you can go back and try an alternative from two questions back.

- Can I solve a simpler problem first? I'm guilty of trying to satisfy all the requirements in my head at once before writing a single line of code, so I can sit and stare at the screen for an hour, thinking. Mathematicians, when trying to prove something, solve embarrassingly simple problems first before adding details, and it's a good approach to relax your requirements. Write a solution for a simpler problem first, and from that vantage point you will likely discover a possible solution that you haven't considered before.
- Is there an alternative approach I haven't considered? Maybe if it doesn't work, you're doing something too complicated. To answer this question, it helps to look at what pieces you have of the solution in isolation. When working on a problem, we write some code, change some, but rarely look at only the changes we've made, as if we were to make a pull request. Seeing only the new code you've written often helps you come up with something else, or see where you've gone on the wrong path.
- Is my problem already solved somewhere in the project? Well, you should ask yourself this *before* starting to write code, but it also works when you are stuck trying to solve a sub-problem of a problem. Here it helps to know what you are trying to do (the first question). Try to look around the codebase and see if someone has already come up with a solution.

That last one is also useful for code style questions, but we'll discuss code style in the section *Code mimicry*. Now I can say that if you *get stuck* on code style ("Should I write it this way or that way?"), just pick one that works and that you feel better about (for any definition of "better") and finish your task.

Now that we've tried to solve our problem ourselves, it's time to start thinking about asking for help. But our peers won't appreciate it if we message them saying "Hey, can you help me with something?"—they are working on their own tasks, and asking a question like this doesn't help them help you at all. When writing to someone at work, the best thing you can do is to *not make their job any harder*. We'll talk about this in detail in the section *Considerate communication*.

Briefly describe the problem you are solving (give links if available), the approach you've taken and the reasons for it, the specific problem you are stuck on and why you think it's not working. Finish with a question. Aim for no more than half a page of text in total (even that is stretching it). Your message can look like this:

> I'm working on the ticket T (link). I found a similar implementation in module M but I also need to update the user field F. The API X that I'm using doesn't provide this field. I tried injecting API Y but it needs an extra dependency Z that I don't have access to in this module. Is there another way to update the field F [the problem you're stuck on] or there's another API I can use [question your overall approach]?

Why do all this extra work when you can just come up to the person or Slack them and let *them* ask you questions? First, the act of describing your problem helps you clarify to yourself what you're missing, and you may understand in the process that your approach is wrong, or you don't know *why* you've made the decision to use this approach. Second, the goal of this exercise is to create *shared context* between you and the person whom you are asking for help, so that after this first message they can immediately help you, because they already know what you're trying to do, what you've tried, and why.

In reply to your previous message, they can say: "I've had the same problem when working on module K. Try using API W instead, it returns the field you need and has the same dependencies as X so you won't have to inject anything else." You thank them and both of you continue working. Instead of a 20-minute back-and-forth where they try to understand what you're doing and get increasingly frustrated, you spend a little time up front to compose a message that *helps them help you*, they spend a minute typing up their answer (because you've asked a specific question) and can go back to their own work. A minor distraction for them, and you've shown respect for their time. They will be happy to help you again later. For a little extra, tell the person that it's worked after you implement their advice.

Even doing just that, spending a reasonable time trying to solve a problem on your own, noticing that you're stuck, clarifying what exactly you're stuck on and respectfully asking for help by asking a good question, will let you learn quicker and teach you how to work more independently by trying to answer your own questions first.

Sometimes you're completely stuck and asking a question will start a discussion, which can naturally transition into a pair programming session, when another person (who is usually a senior software engineer, able to solve the problem you're having) helps you with your code.

Some people consider pair programming to be the best for learning, even when it's two similarly skilled engineers working together on a feature, exchanging ideas. I haven't got to do much of it, and I find it hard to talk and think at the same time, so it doesn't work too well for me. It's fine if something that works for many people doesn't really work for you, like learning from a book, or watching lectures on Coursera, or pair programming. Everyone is wired differently and the trick is to find what works better for *you* and do more of it.

But even I can vouch that pair programming is *especially* good when it's one person trying to help another with a problem they're having. For a junior engineer, it's like borrowing the knowledge of an expert and immediately having all the answers, and the reasons behind them. When working with a person who is more familiar with the project than you, you get to learn their mental map of the project, discover code that you haven't seen before, learn the concepts they use to think about the codebase, and you can ask them to explain how some parts of the project work that you don't understand.

If pair programming is practised at your company, don't hesitate to use the opportunity. Start the practice at your company if you don't have it. Pair programming is similar to the Ancient Greek practice of a mentor having an educated discussion with his pupils. It worked for them and it will work for you.

On the question of borrowing expert knowledge, an invaluable habit is to have people tell you exactly what you need to know before you spend any time trying to discover what you need to learn on your own. This is the most valuable when you are just starting on a project, and there are already people on it who are experts.

Usually you know what you are going to work on, so you can come to them and ask: "If I'm going to work on X, what are some essential things and concepts I need to know? Is there anything non-obvious?" They tell you and you take notes of everything they say, and ask to clarify if you don't understand something at all, and confirm if you understand correctly: "So to use X, you first have to pipe data through Y and filter with Z, is that right?" When they've finished, ask: "Who else should I talk with about that?"

You're new at the job and don't know who is responsible for what. Some knowledge is not in the codebase, but in the heads of other people, whom you don't know yet, but who can give you insights *why* something has been done the way it is and give you useful data for reference. Aim for 30 minutes to an hour of talking with each person, not more, otherwise you'll be overloaded with information. I can't believe I have to say it, but don't forget to thank them for their time.

In one of my past jobs, one the first things I did after I started was to sit down with another senior engineer and have him walk me through the concepts behind the part of the project's architecture I was supposed to work on. There was also no documented overview for it, so I took my notes and turned them into a guide on all the moving parts involved in creating a new user-facing feature (with short code samples). By writing it up in the form of documentation, I learned more myself and helped the other engineers who joined after me to have an overview they could refer to from the very beginning.

By asking existing experts, in hours you get instant expert knowledge (just like instant coffee) and a mental map of the essential concepts behind the project that would take you months to discover on your own (if ever). If the project is well-documented, a lot of the expertise and the right way to look at things are still locked in people's heads, and asking them to share this knowledge with you unlocks it.

Even if we are talking only about code, on a project that is larger than a few hundred thousand lines you are likely to never touch some major components, only use them (if that), and so never learn how they work. But the expert, who has probably written this code, will tell you how they think about it and what you need to know about its inner workings.

Does that all make you an expert? Of course not. What it does is allows you to start building the *tree shape* of the project in your head, and what knowledge and skills you will need to work on it. You still have to do the work of filling in the details.

After you've settled in, work becomes less challenging as you learn more. In my experience, this happens after three to six months after you start in a new role.This is a good moment to reassess your position and adjust your compass. What did you learn? Did it bring you closer to your goals? What are you still struggling with at work? Do you like what you are doing and the part of the project you are working on?

When you are not new at your role and are used to your usual work, opportunities for learning come in the form of doing *different* or *more challenging* work, usually called "stretch projects." These are fixed-length tasks that are more involved than what you usually do, but are not so difficult that you won't be able to do them. Such difficult but not impossible tasks "stretch" your skills and let you scout new territory, filling in more areas in your tree shapes.

This is usually done in coordination with your manager. Companies that care about helping their software engineers improve are proactive about this. Even if you're interested in a different role, this can be arranged as long as people are happy with how you perform. Doing quality work is a prerequisite for getting more responsibility. In any case, you should let your peers (and your manager) know that you *want* to improve and become a better software engineer, and what direction you want to focus.

The "stretch projects" don't necessarily have to be some huge tasks that affect the whole codebase (even if you are a senior engineer), but can be, for example, tasks from other teams working on the same project but in an area you're interested in. They will be harder for you and a learning experience because you're unfamiliar with that part of the codebase. It could also be an experiment completely outside the scope of the project, like trying out a new technology, doing performance tests on new approaches and reporting the results, working on qualitative test improvement and so on.

Again, you should think about going further only after you've begun to master your current role. If you are a junior software engineer and want to refactor a whole module and get rid of legacy code, you will likely be refused, because you have mastered neither the scope nor the necessary skills to perform this type of work. A more fitting project for junior engineers would be to let them participate in design discussions and take a new feature from design completely through implementation (with occasional help), thus proving that they can work independently and now have enough skills to work without hand-holding.

If you are worrying that you are going to be refused, such personal development is not encouraged at your company (why?) and your manager insists that you should focus on doing your work, either you are not ready (but you think you are—check yourself against other software engineers) or you can take it into your own hands and do learning forays while working on your regular tasks—by now you should roughly know what you need to learn. There is much to learn even when you're adding a button in a boring line-of-business application.

We have touched briefly on how to talk with other people, now let's look at it in more detail.

3.3 Considerate communication

The formula “don’t make their job any harder” that I mentioned in the previous section is the central principle of *considerate communication* and can be applied in a variety of contexts. The primary objective is to value other people’s time. The secondary objective is to answer other people’s questions before they have them. The result is that you are building a reputation of being dependable (you do what you say you will do, cultivating trust) and pleasant to work with (you don’t waste anybody’s time). People take you seriously and you become an authority (a trusted expert).

Being taken seriously doesn’t mean that you get to have no fun. A trusted expert isn’t someone sour-faced. As long as you have a mutual understanding with people and your style is appropriate for the context, you’ll be fine. Knowing what is appropriate and predicting the reaction of people you are writing or talking to is also a part of considerate communication.

In the same way, valuing people’s time does not mean that you have to value their time over yours. It is not a zero-sum game, where if someone wins, the other loses. It is acknowledging that the work other people are doing is important, just as yours is, and you are communicating in a way that lets them do their work by doing yours. Often it is simply a matter of efficiency.

Like we’ve seen in the previous section and the example of asking for help, putting your thoughts in order before asking questions helps both of you. You get help quicker, and the other person spends less time distracted from their work. On your side, you would spend the same amount of time if you asked for help first and then answered questions about what exactly you needed help with, but the other person would need to spend all this time extracting information about your problem, unable to help. Why would you willingly choose to *waste* people’s time at work if you knew a quicker way that would save time for both of you?

Perhaps, the simplest example of considerate communication is “No Hello,” a single-post website that asks to avoid saying “Hi!” in chat, waiting for the other person to reply, and only then typing your question.

> It’s as if you called someone on the phone and said “Hi!” and then put them on hold!

Instead, it suggests putting your greeting and the question you have in the same message. You can also ask for permission and clarify in the same message: “Hi! If I may ask a question: X? If you’re not the right person to answer this, could you please suggest whom I can talk to?” Treat your first message like an email, not requiring the other person to be present, and ask the question right away so they can start thinking about your question at once and not wait for you to type it up.

If you don’t ask the question and have to leave the chat, it can get completely ridiculous:

> [12:02] You: Hi!
> (You get a sudden call at 12:05.)
> [12:10] Coworker: Hello!
> (Coworker leaves for a meeting at 12:15.)
> [12:35] You: Sorry, I was in a meeting. Could you help me with X?
> [12:50] Coworker: Same, half an hour meeting. What do you need?
> [12:51] You: (finally asking your question)
> [12:52] Coworker: (answers your question)

Instead, if you ask the question right away, you will get a reply sooner, and without the unnecessary back-and-forth:

> [12:05] You: Hi! If you’re not busy, could you help me with X? (your question)
> (You manage to type your question before the sudden call at 12:05.)
> [12:10] Coworker: Hey! Sure. (answers your question)
> (Coworker leaves for meeting at 12:15.)
> (You get back at 12:35, your question is already answered.)
> [12:35] You: Thank you!

Only two messages instead of six to get your question answered, and nobody had to wait on each other, even with the disruptive calls. Again, I’m not advocating being robotic and measuring your “communication efficiency” by counting messages. Chat with your coworkers all you want! But even in this example, you get work out of the way sooner and, having gotten your answer, you can have a chat instead of waiting for the moment when both of you are available to ask your question.

This is less of a problem when both of you are at the office, but writing gets increasingly important every year with the rise of inter-company communication tools (that are not email) and remote work. Even if you and the person you want to talk to *are* in the same office on different floors, would you walk to their desk every time you wanted to talk to them? No, you would message them.

Expanding our previous example, imagine if you needed help from three people, and someone needed yours, and everyone waited on each other to ask questions. You would spend all day chatting instead of working. That is exactly what happens in a lot of companies that are forced to work remotely. You get less done, but feel more tired because you've been distracted by chat all day. Battle the distraction by giving people information they can act upon.

Thanks for coming to my TED talk!

Practicing and getting better at writing is at the heart of considerate communication. Practice writing by starting with thoughtful Slack responses, then well-researched emails, then documentation, then posts on the company engineering blog. I found myself writing this book.

All of that builds your credibility as a software engineer and, what is more important, through better writing you are making it easier for people to understand what you want to tell them. That people appreciate what you have written—because they have found it clear and useful—also helps curb impostor syndrome that many software engineers struggle with.

William Zinsser in his book "On Writing Well" (my favorite writing book that I highly recommend) said "Clear thinking becomes clear writing; one can't exist without the other." By noticing that you cannot write clearly about something (or people have to ask you additional questions), you discover what information you are lacking. The same principle is behind the advice to teach something to really understand it, because in order to give a clear explanation, you have to fill in the gaps in your knowledge as well.

We can get better at composing our messages by practicing for the second objective of considerate communication—answering other people's questions before they have them. This applies both to the information you want to convey and the questions you ask to others.

Let's go back to our "No Hello" example. Imagine that the question you asked was so generic and simple, that the answer was the first result when you typed it literally into Google. Your coworker might answer your question, but no matter how well-written your message, she would think: "Couldn't you google that first?"

Nobody likes to answer questions that are the first result on Google (unless you expressly need their unique summary and opinion). Your coworker would get the impression that you clearly value your time over hers, or rather, that you don't value her time at all (because people think in absolutes). If you do that more than once, how do you think her opinion of you will change?

If you think this example is too extreme, I assure you people do that all the time. Just open Stack Overflow, switch to the newest questions, and you'll see for yourself. I picked one, copied the title directly into Google and got an existing Stack Overflow question as the first result, that not only answered the question, but also gave additional information about the values (the question was about a Python function parameter), what sizes were best and why.

Practice improving your writing for other people by trying to answer your questions yourself first, and by trying to predict what questions other people may ask.

The previous situation with the overly simple question could have been avoided if we tried to type the question into Google first, and would have required minimal effort. Answering your own questions (that come up while you're working) means finding information, and that is a crucial skill in software engineering. I have explained earlier, that if you have a poor map of the territory around you and don't take the time to build tree shapes of the knowledge you might need, then, essentially, you will be lost and will be forced to ask for help. Conversely, if you were able to find answers to any of your questions (let's consider speed not important for now), then by definition you would have been an expert.

The quality of questions you ask directly depends on the quality of your knowledge about a subject. If someone asks simple questions that are easy to answer again and again, we can conclude that they don't really know much. In many cases, by looking for answers before you ask for help, you're able to solve your problem yourself, and this process of discovery is a much better teacher than if you've just been told the answer.

Predicting other people's questions in advance is a little more difficult, but the main trick to remember is to start with a notion that they do not know anything about the context of your work. If they're software engineers and your coworkers, then, of course, they know about the project and its inner workings, but don't assume they know anything about what you are trying to do. Our task when composing a message, then, is, first, to educate the person and second, to explain how what we are doing affects *their* work (because people are the heroes of their own stories).

In our thread of convoluted examples, imagine you are about to do some maintenance on a backend system people are using to run some of their work against. You drop a message into general chat:

> Heads up, there's system maintenance soon.

Now place yourself in the shoes of another software engineer. Do you remember that "the senior engineer enables others to make decisions efficiently"? The message that you have posted does not allow anyone to make any decisions, because it's lacking information. The first question that everyone has is "What does it have to do with me?" Here are some questions other people have that you can answer while composing your message *before* you send it:

- Exactly which system is being maintained and for how long?
- What is the environment, is it development or production?
- When exactly does maintenance start?
- Will the system be still online during maintenance?
- How long will the outage last?
- Will I lose any of the data that I've just loaded into the system?

To you the context may be obvious because you are the one doing it: "soon" means twenty minutes, "the system" is the one you are working on, "maintenance" is just a restart. But people who are reading the message don't have any of that context, so your task is to educate them, not as if, but because they know nothing. We can rewrite your message to answer all of those questions:

> Heads up, backend system B in the DEV environment will be restarted in 15 minutes (at 12:30) and will be unavailable for 10 minutes. It will be back up at 12:40 with the same data as just before restart. If you are working on X or Y you can get errors while the system is down (go get a coffee!). If you are loading something big, let me know and I will hold the restart for a couple of minutes.

If you send the first message, you will probably spend the same amount of time (but likely more) answering questions that people send you, as thinking about those questions and answering them in advance while writing the second message. Including all the extra information gives people the ability to make a decision whether they have to do anything about the restart and whether it will affect their work. And you won't have to spend more time chatting, answering the same questions that you could have answered while composing the message.

Predicting what other people want to know (and when) is also important when managing expectations. When you are working on a task that needs to go into the next release and has a deadline, it's not hard to predict that your manager will want regular updates, and will want to know as soon as possible if you are going to miss the deadline. (The question of programming tasks with deadlines is out of scope for this discussion.) But remember to answer their (unvoiced) question in your update! If you write "I'm working on it" on the day of the deadline, how does it help your manager *make any decisions*? It doesn't answer their question: "Will it be ready today, and if not, approximately when?"

My rule of thumb is that if a person has to ask me a question about the progress of my work, I have failed. This is especially relevant for remote work, because nobody can see you at your desk working. Even if you are the most dependable person and are never late with your tasks, people will still have a question in their head: "What if you're late *this time*?" Overcommunicate by default and send more updates than you think is necessary. Of course, don't do it so often that it interferes with your work.

For a task, some good points I use are 20%, 50%, 75% and 90% of its completion. If you expect to work on a task for a week, on Tuesday you say "no big problems," on Wednesday "about halfway done, should be finished by Friday," on Thursday "mostly working, final touches" and early on Friday "need to finish X and Y and I'll be done today." It doesn't take much time at all but brings peace of mind to everyone working with you because they know what you are doing. If you post too often, someone will tell you, don't worry about that.

What if you made a mistake, your estimate was wrong, you sent out an important email with incorrect information? Think about the situation in terms of making decisions. You've discovered a mistake, that means you know that some information is wrong. Other people still believe that this information is correct and will make decisions based on that information. The longer people believe the information is still correct, the more bad decisions they will make.

The best thing you can do in this situation is to let the other people know that this information is wrong, so that they can adjust their decisions. In other words, you should own up to your mistakes as soon as you discover them. By waiting, you are making other people's jobs harder because they will have to adjust more of their decisions. This is directly related to being considerate—thinking about other people.

Remember that work is not school. You are not given points for correct answers and blamed for wrong answers (well, maybe a little). Blaming people for their mistakes is counterproductive because it doesn't help solve the problem. You were hired as a software engineer to solve business problems, so making a mistake is just another problem to solve. By owning up to it early and suggesting (or implementing) a solution you can even be praised for it, because you have likely avoided a worse outcome. This happened to me when I discovered and fixed a nasty bug before release: I was congratulated, but I was also the one who had introduced the bug. It didn't matter because I was able to fix it in time and it didn't affect customers.

What if people *are* blamed and ostracised for mistakes in your company? This culture results in problems being ignored and swept under the rug. In relation to software development, it's creating technical debt and hoping the bug happens on somebody else's shift because then they'll have to fix it. The culture of blame shuns experimentation and learning, because that is risk. But you can't learn without making mistakes, failure is the best teacher. If you wait until you can write code without bugs, you won't be able to write a single line of code. I would start looking for another company.

On the question of company culture, considerate communication is also about cultivating a positive atmosphere in a team. When we compose our messages (or any other writing, like comments in code review), we must think not only about the way we convey information, but also emotions that other people will have about our writing.

In the section *Development is a team sport* we have already discussed positive team environments, how positive teams are more productive, and that you have to expend additional effort and watch your language to make your tone positive. I have also linked to Babylon Health's "Code Review Etiquette," but there are more good examples there, and I encourage you to read it.

Sparing people's feelings does not mean that you can't have firmly held opinions or will have to agree with everyone. Not at all. Being intentionally rude, insulting other people at work and degrading discussions into shouting matches is simply unprofessional. Practice being calm and *considerate* ;)

3.4 Unknown unknowns

The meaning of "unknown unknowns" is well-explained in the famous quote by the United States Secretary of Defense Donald Rumsfeld during a Pentagon news briefing in 2002:

> …there are known knowns; there are things we know that we know. We also know there are known unknowns; that is to say, there are things that we know we don't know. But there are also unknown unknowns—the ones we don't know we don't know.

Here I will use this term to mean knowledge in your domain of expertise that you are not aware about. This is closely related to creating tree shapes that we have discussed in the section *Exponential learning*. The *unknown unknowns* are knowledge that you have not yet put in your tree shape at all because you don't know that it exists.

Knowledge that you are aware about (or *known unknowns*) is something painted in broad strokes in your tree shape, not too detailed. Finally, *known knowns* are the detailed regions in your tree shapes, knowledge that you can use immediately.

The keen reader will notice there can also be *unknown knowns*, and those can be described as things that we know but choose to ignore. But we are trying to become better software engineers here, not build a psychological framework. If you're interested, the framework that has these four quadrants is called the "Johari Window."

It’s common that our junior engineer James will struggle with a problem for a couple of days and try different approaches, and then a senior engineer will have one look at the problem and say: “You should just use X.” That X, for the junior engineer, is an unknown unknown, because he has not been aware of it.

Now that we *know* (duh) about this, what can we do?

We must do our due diligence and broadly research the knowledge domain of our choice, so that even if our map is low on detail, parts of it are not completely blank. In other words, we must put special care into organizing our knowledge and filling in our tree shapes.

In more cases than you think, simply being aware of some bit of knowledge or the existence of an approach is sufficient, because if we’re aware of it and its qualities, we can go and research it and find *more* details about it, up to and including using that approach or that library (or whatever bit of knowledge) in our project. Otherwise, when facing a task, we are likely to try to apply the approaches that we already know, not looking for alternatives.

Perhaps this is my personal bias, but I think having an extensive mental library of patterns and bookmarks (how best to do things and where to find more information) is more important in software engineering than knowing few things very well and being good at execution. Is this the difference between a generalist and a specialist? Not really. You can both be a specialist in one or two domains and have the breadth of (not very deep) knowledge in others—sometimes called being a “T-shaped developer.” This is closer to the difference between being a “Java programmer” and not being interested in other fields (but being able to create generic backends very quickly) and being a “software engineer” and having enough transferable knowledge to be effective in any specific role. It is a matter of setting yourself up for success in a long career.

Luckily for us, information about technology (if it’s not too advanced and/or obscure) can be easily found on the internet, and when necessary, unlocked from the heads of experts. In the section *Tree shapes* I have shown how I’ve been learning C++, and here are some more ways to fill in the detail on your map:

- It’s good to start with several introductory articles and a couple of books, even by reading only the table of contents. The authors are usually experts who have taken the time to organize their knowledge, so it’s often a ready-made framework that you can use to research further.

- Look at `awesome`-style collections on GitHub to get not pointers, but actual links to things you can see and read. The better ones are also a good overview of all the things you can do with a technology because they'll link to different projects using it.
- I find recorded conference talks a great source of best practices and mental concepts that you can use to think about a particular technology. Getting knowledge from an expert, structured in a way that they think about the subject is the next best thing to having an expert answer your questions.
- Yes, you can have experts answer your questions too. Every technology has hubs, forums, chats, communities and more where experts hang out discussing the technology with other experts (because everyone needs help and inspiration sometimes). By finding them you can not only have your questions answered, but also get an overview of the latest developments and trends that can tell you where the technology is headed and what the current problems are.
- On the question of current problems, the easiest (and the most fun) way by far is looking for memes. Google "X memes" and you will find Twitter accounts, subreddits and other sources. Memes are an outlet for frustration and exaggerated exposure of problems with the technology. They aren't a source of truth, but they are generally true and can guide you to well-researched articles talking in detail about the same problems (which can often be a deal breaker), that you would not have found otherwise. I call it Meme-Driven Research™ and it's surprisingly effective.
- Books, articles, videos and talking with real people should give you enough information to start experimenting with the technology by yourself. Pay special attention to things that are similar to what you know, because their similarity on the surface (like a similar API to a library from another language) can hide an "unknown unknown," some concept of the technology that you miss and that can come back to bite you later. I especially like to adjust the way I think when stumbling on some early problems. Often I'm either doing something wrong (having a false hypothesis) or don't have the right mental model, and if it's this early, it's likely that many other people have the same problem, so it's easy to find articles that explain some key concepts. For example, you can have a lot of problems with Git just trying to use it through non-obvious commands. Dealing

> with it becomes much simpler when you have a mental model of what happens under the hood (it's not too complicated), and you can find the commands more easily when you know what you want to do conceptually.

Don't skimp on broad research and you'll be able to work more independently. Doing research like this is similar to answering questions about your code, only it's about concepts rather than specifics, so both of those are really one skill of effectively looking for information, and one reinforces the other.

Since we've started talking about code, let's see what you can do to get through code review easier (even if you're new) and get more familiar with the codebase quicker.

3.5 Code mimicry

Ah, the sight of a new codebase. New to you, of course. The gleaming front door and the cobwebs in the hall. A whiff of smoke down the corridor. Weirdly shaped windows. Rooms with "DO NOT ENTER" written on their doors. Hallways ending with brick walls.

Orienting in an unfamiliar codebase and blending in when contributing code are two skills that are interdependent on each other and will help you at any stage of your software development career—and will make an impression on your peers.

If you're a software engineer, being positive with your coworkers is important and is a sign of professionalism, but your main skill and your main job function is still writing code. We have already discussed what *good code* is in the section *Development is a team sport*. We know it should be readable and effective, and that any code you write should solve the task at hand. Another point that is not less important is that code exists in the context of the codebase and the project you're working on. Every project is unique, and every codebase has its own architecture, style and code practices that may be different from what you're used to.

You must have seen discussions on what case to use to name variables, how to name classes and methods, whether to use tabs or spaces, and more. On a higher level, developers discuss how to compose functions, the rules for creating library APIs, writing doc comments, and on and on. Software engineering is like shifting sand, and everything is up for discussion. You may have your own strong opinions on many of these questions. People often do, hence the existence of many "opinionated" projects that make certain choices the certain way, and they are not up for discussion in the context of that project any more.

In general, a sizable part of these discussions is irrelevant or orthogonal to the logical structure of the code. This is also known as "bikeshedding," or the "Parkinson's Law of Triviality"—meaning that a large amount of time will be spent discussing trivial technological details that don't have much weight in the context of the whole project. My advice is to stay away from these discussions.

Bruce Lee famously said "Be like water" and this will be our approach to a new codebase. Our job as software engineers is to solve problems for the company and the customers, most often by producing code that is maintainable by other people. Maintainable code means it's easy to read and uniform, and easy to change. To achieve this, it should follow *the principle of least surpise*, meaning that it will be structured and look the way people *expect*. The best way to make this impression is to have your code follow the style, formatting and structural conventions that already exist in your project, so that your newly written code looks as if it has been in the codebase all along.

The codebase you're working on can have formatting you don't really like, or have some weird architectural decisions, but if you try to impose your opinion by ignoring the existing conventions, you'll make a disservice to your coworkers. The worst thing for a project is to be inconsistent, because that brings mental overhead for everyone working on the code. I don't mean only the style and formatting, but also conceptual things like API design and what patterns you're using. Experienced programmers will read any syntax with little overhead since they are used to reading a lot of code, but when you have to switch gears to a different set of concepts in every file, it becomes taxing and will slow down the team.

The counter-argument is that a completely different set of concepts and data structures may fit a particular task or module better. It may, but it severely limits the opportunity for change, and the ability of different people to work on different parts of the project at will. As software developers, we work with abstractions wrapped in abstractions. By having a consistent set of abstractions at a lower level, we can start thinking of the project at a higher level of abstraction and see patterns that would have not been apparent had the codebase been a mish-mash of different approaches.

That doesn't mean the project has to be stuck with a particular unfortunate technical decision forever. This situation can and should be improved through refactoring (improving the code while preserving its functionality). Shims and wrappers could be written for legacy code, so that developers don't have to deal with them directly, or legacy parts could be replaced by new ones, if it's worth the effort.

It's critical that the whole team is aware of these decisions and the direction the project is taken, again to maintain consistency. If one part of the team refactors one legacy module one way, and another part of the team refactors another legacy module another way, both versions end up with "better," but different APIs. The project will become a disjointed mess, and it would have been more reasonable to keep the legacy, and for both teams to work together either to produce API-aligned refactorings, or to write *one* wrapper that would fit both legacy modules.

Thankfully, code style is becoming less of a problem—and less of a discussion topic—with the rise of automated code formatters and linters with rules that can be shared across the whole team. As long as the majority agrees on a set of rules, code formatting becomes a non-issue.

Some languages, like Go, have formatting built-in from the beginning and as a feature of the language, so *everyone* using the language is using the same style, which makes it easy to read code by other people in a variety of projects, since they all look the same. With formatting out of the way, you can focus on understanding the concepts and data structures (the meat of the project), rather than lament the poor taste of the authors.

Now that we understand *why* we want to mimic the existing code, let's see *how* we can do it.

In the section *Help your company teach you* I've already said about talking to experts. Ideally, the first step when you start working on a new codebase is to talk to existing experts and ask them to explain the top-level concepts and the decisions behind the project's architecture. Both of these are often missing from the documentation (if there is any). You will learn the specifics and the details of writing code later. The goal of this exercise is—you guessed it—to begin making a *tree shape* of the codebase, and we start by mapping out the larger moving parts first, before we move on to filling in details. Knowing the high-level concepts gives you an immediate advantage of the right way of thinking about the specifics.

The next best thing is to read documentation and look at the physical project structure (directories and files). Projects are usually organized according to their logical structure, so just by looking at the directories you can discover most of the moving parts and how they relate. Even if your first task in a new codebase is to work on a small feature, putting it into the context of the whole project will help you understand where it fits in the structure.

Before you write any code, read some of the features that are similar to what you'll be working on and pay attention, first, to the style and formatting (usually there's also a code style document you can read, or a set of rules for a code formatter or linter), and second, what practices and patterns are used when writing code.

Is the style more imperative or functional? What is the average length of methods? Is the code more focused on OOP or composition? Do people try to be terse or the code aims to be simpler and longer? Are there any comments? Most big programming languages used in commercial software development have a lot of flexibility in how you can write code, and some teams will use libraries or approaches that can completely change the "usual" way the language is written.

I hope you like to read code because we will do a lot of reading! Writing code that looks similar to existing code and uses the same patterns depends on you being closely acquainted with that existing code via reading a lot of it. To get the most of our time, we won't just read the codebase like a book. Armed with the high-level concepts that we have learned first, we start very close to where we'll be working within the project. I've just said to look at some of the similar features, but now we'll need to look even closer and read the code that you'll interact with.

Now we want to understand the local structure and usage. What are the data structures and classes that are used? Are there any utility functions or helpers? If you notice how utilities are used in code (and where they are implemented, so you can find more of them), you'll get a good idea of what people do often enough that they've needed to write utility functions. What are the dependencies and where does the data that you work with come from? Notice how simple tasks are accomplished line by line, because that's how we write most of our code, trying to accomplish a simple task in a few lines. Often reading just a few files is enough to see many common patterns that you will use when you begin writing your own code.

For contrast, have a look at some places in the codebase that are far from where you are going to work. How are they different? Presumably, they have been written by other people than what you've been reading before. Is the code similar to what you've seen? If it's very different, try to understand which of the styles is the older one and which is preferred. Either look at the file history in version control or ask someone on your team.

In one of my past jobs, my first task was to implement a feature that was very similar to an existing one, but after I submitted it for code review, I learned that even though the feature I had copied wasn't old or legacy, it used some older patterns and there were newer ones available that I should have used. If I had looked around the codebase a little more or have asked my teammates, I would have known about them.

Again, asking existing experts is one of the best things you can do when you have many implementation questions in your early days with the codebase. Junior software engineers often submit code for review that does roughly the right things, but they have reinvented the wheel by not using existing utility code. As long as you are asking for help in a way that respects other developers' time, you should do it more often to resolve the ambiguity about the code you're writing.

Try writing *some* form of the solution first and look at similar features again to see if there is anything that can help with what you're struggling with. Notice when you feel like you have to jump through hoops and the solution doesn't come naturally. For example, if you need to transform some data and you can do it in a way that is idiomatic to the programming language in general, and comes naturally, is short and clear, then there's no problem and even if there is another way to write it, you can leave it as is until code review. If, on the other hand, you notice that the code to do something comes out confusing, or you feel like there should be an easier way to do what you're trying to do, there probably is. You did the work by trying first, so you can ask: "I'm trying to transform data I get from X to do Y, and I have this code: `[your code]`. I can't find a simpler way, is this approach OK?" Not all code has to come easily, and someone probably will assure you that you're doing fine. Maybe there is a helper to do it, and you can simplify your code.

In larger projects, almost everything you're trying to do has already been done, and blending in is a matter of finding this code. I don't necessarily mean whole features (though that happens too), but rather the small building blocks that you are using to construct the solution to your problem, like the order of methods within classes, working with data within functions, what signatures and parameters names to prefer when writing your own functions, down to how to loop through arrays and do small data transformations.

All these small details contribute to the "feel" of your code and whether it fits with the project as a whole. At first you will spend more time looking for code that does something similar to yours, but that means reading more code, noticing more patterns and in the end, all of that is a learning experience helping you become more and more familiar with the codebase, and with time you'll find things quicker and quicker.

Finally, I'm not advocating that you have to produce code that perfectly matches the project on your first try. Write what and how you can. The goal of this section is to make you aware of a way to approach a codebase that will let you produce better code sooner. Better, because you'll be more familiar with the codebase and will use existing style and approaches instead of having to invent your own for problems that have already been solved. This ties back to treating the project as *shared code* that the whole team is responsible for. Of course, it's impossible to have the project's code look as if it's been written by one person (and there's no need to), but keeping the codebase consistent makes it more maintainable, which lets the team as a whole be more productive.

You will find that code review will be much smoother if you try to mimic existing code from the beginning. Instead of being littered with comments "we usually do X instead" and "restructure this as in Y" and "please rewrite to use existing helper Z"—all related to the minor details—you will be able to turn your reviewers' attention to the conceptual merit of your solution. When all the smaller details are in order, your code starts to feel like being written by someone who has been on the team for a long time (which is what we want).

4 Leveling up

In previous chapters, we have learned what it takes to *be* a senior software engineer, and here we will see how to *become* one, either by getting promoted at work, or by finding another job. Some hot takes in this one…

One thing before we begin. My position is that you can be a senior engineer without the *title* Senior Software Engineer. If you are doing senior-level work that lets you realize your potential, your team is positive, your company invests in learning and doesn't skimp on compensation, please be content and don't chase the title for its own sake. We are trying to improve as software engineers, and spending several years in an environment conducive to learning will let you improve much faster, and sets you up for further success in your career, compared to being Senior in a company where everything is on fire. This is not fine at all.

4.1 The real world

> Welcome to the real world.
> Morpheus to Neo

Did I just make a reference to corporations treating workers as living batteries to power the business, literally draining them of life? Absolutely not! `*huff*` Well, maybe a little.

What does it mean to work for a company in the real world? We already know that work is not like school. You do not move forward as much or as quickly if you simply show up and do the work that somebody else has laid out for you. Doing everything right and getting "good grades" does not mean that your company will automatically recognize your work and promote you.

My version is that for a company, we are a business function on its balance sheet, a "human resource," but at the same time a company is not sentient and *everything* that happens in a company is the result of people making decisions. We are suspended in the middle between these two notions.

From this definition we can make several conclusions:

- People who make decisions often think in balance sheet terms, not human terms. Laying off 5% workforce to keep the budget solvent, or hiring two more software engineers for team X so that it's able to deliver project Y on time.
- If we want something to happen, we must influence people who can make that decision.
- People who make decisions can be influenced on a human scale (because they are people and like or dislike other people) and on a balance sheet scale (because that's how they often think).
- To the company, you are one among many and the company cannot be loyal to you. The company will expunge you from its balance sheet without hesitation. Thus it's unnecessary to "prove" your loyalty to the company, because it won't show any to you. But:
- People can be loyal and look out for other people within the company.

This reads like a story of favoritism and office politics, but isn't that exactly how a lot of companies work? Clueless manager gets promoted because he's friends with the CEO—a balance sheet decision primarily from human-scale influence (a detriment to the company). A quiet software engineer who has been with the company for eight years is fired because his unit became unprofitable—a purely balance sheet decision and a lack of human-scale influence. Another software engineer from the same unit got transferred to another unit because she has been active on other company projects and made an argument with her manager that her experience was perfect for accelerating a project that another unit had just started—a balance sheet decision primarily from a balance sheet argument (a boon to the company).

Does it mean you *have* to cozy up to your manager to get promoted or simply not to be fired? The answer is our favorite answer in software engineering: it depends. Unfortunately, whether we like it or not, the situation I've described is the reality. It is foolish to refuse to accept it. Of course, there is a spectrum, and some companies are friendly and care about the wellbeing of their employees, while others are like snake nests (and both can coexist in a big enough company).

In the end, all companies are just a bunch of people working together, and large groups of people always involve a layer of personal attitudes and relationships. People aren't robots performing their "job function." So even if all we want is to write code, we will have to deal with people.

I hate office politics with a passion, not only because I prefer to work rather than bicker and badmouth, but also because people involved in politics make decisions that are less objective, with detrimental results to the company, the product and ultimately the customers. In tech, you can often *prove* that your argument is correct. You can't persuade the compiler that its opinion of your code is wrong. Politicking runs across that.

That is the primary reason I choose to stay at the regular engineering level. A good engineering manager will shield and protect his or her team and let them focus on their work. Still, no matter how sheltered we are (and how socially dysfunctional I sound), it's our job to get "noticed," to improve our skills and get promoted if that is what we want. While it may sound grim, this is actually a blessing: your career and what you do is not controlled or handed to you by other people (like at school). *You* are in control of your career and you can steer it the way you like it.

Fortunately, we can do it ethically, without the need to cozy up to anyone or to badmouth other people. We will talk more about it in the next section, but primarily this is about knowing your value to the company, talking and working with your superiors and peers, taking responsibility and following through on your promises. This is a professional and calm approach. It helps to be friendly, but you don't have to make friends if you don't want to. We will talk more about being closer to other people and "building a network" in the section *Interesting people*.

A common story is a manager promising a promotion or a salary increase for years and never following through. If you have taken more responsibility, improved your skills and had an agreement with your manager (best do that in writing—more on that later), then this is plain manipulation of your loyalty to the company. And we already know that a company can't be loyal to you. If the spiel "If you work here some more, you'll be promoted" repeats again and again, it's time to leave. Your work and potential is wasted working for a company that doesn't keep its promises.

I've read the other day on Twitter that it's a privilege to be able to find software engineering jobs at will, and it's disingenuous to advise leaving a stable job. I agree, especially if you don't have a network, or have not given much thought to career development. When you've spent more than six months looking for work, it's extremely demotivating (speaking from personal experience).

My hope is that by learning and practicing what this book has to offer, you will have an easier time looking for a job when you need it. Meanwhile, my advice, if you find yourself in an unfortunate situation at work, is to look for another job *while you are employed*. Unless it's very bad or you have at least six months of savings, you will be much more confident doing interviews and will have more success if you keep working.

Finally, you may find that when you ask for more (and deliver what you've promised on your end) you will see more respect, and even get unexpected help from your superiors. This happens because there aren't that many bad apples. Normal people *like* other people who are ambitious and follow through with their promises. When people trust and respect you, you are far more likely to get what you ask for. As we've learned earlier, you gain trust and respect by helping other people, and this is exactly the opposite of undermining their efforts for your own gain, and completely ethical. Hint: if you think if something is ethical or not, you are probably not a jerk (they use any tactic they can).

4.2 One company or job-hopping?

If you are a junior or a middle software engineer and you want to become a senior, is it better to stay and get promoted at one company, or work as a junior engineer in one company, then as a middle in another, and then get hired as a senior software engineer in the third one? The answer is, you guessed it, it depends.

Primarily it depends on what kind of company you're working at and what the promotion history is. Past behavior is often a good predictor of future behavior, and what has happened at the company a few years ago is likely to happen again. Look at the facts and try to find answers to some questions:

- What is the turnover of software engineers? Do they tend to stay or leave after a couple of years?
- Have the developers who have the Senior title now been hired as middle or junior software engineers? How many years did it take them to become senior engineers? What was the process?
- Does your company have a clear track for promotion, ideally with a timeline? Can you “jump” the timeline and get promoted quicker if you fulfill the requirements for a more senior role?
- Have there been any significant changes in hiring and promotions in the past year that make answers to the questions above irrelevant?

The answers to these questions will help you understand how promotions work in your company, and if they work at all. Maybe nobody knows how to train junior engineers, and they leave after a year of doing menial tasks (but we know better than that!). Maybe, like at larger companies, there is a set track that lets you increase your grade every three years (but not sooner).

In any case, the answers, if you manage to find them, will give you information to adjust your expectations, but your particular situation is unique. As long as you can prove you can do senior-level work, there is no reason for the company not to increase your pay and give you the Senior title. For the business as a whole, individual salaries are a drop in the bucket, and the title costs nothing at all, but to you both can be life-changing.

Getting a higher-level title sets a precedent that will have a big impact on your career as a software engineer. Getting promoted from being a junior software engineer becomes a strong signal for other companies to take you much more seriously, because that means you can now work with minimal supervision and create more value for them, while being a junior means the scope of tasks you can do is limited, and someone will need to keep an eye on you.

The jump is smaller with getting a Senior Software Engineer title (as I’ve said, some companies don’t have the senior titles at all), but is still a signal that you have become an expert and can steer whole projects on your own. Being Senior also means that you are extremely unlikely to start as a Junior Software Engineer again in any other industry, because the Senior title implies you have enough transferable skills, knowledge and the ability to learn new technologies that can be applied in your new role.

A major benefit of staying at the same company is that often you can choose what you work on and evolve your skillset, even to the point of completely transitioning to a different role. If you were working as a web developer, you would have a hard time finding a new job as a backend developer in another programming language, but you could try it out while working at the same company and later make an internal transition to another team.

This happens all the time in big companies, because as we've learned, retaining great software engineers is often more important than what specifically they're doing—they're great, so they do good work. So if you want to try out other roles than your own, or if you're not sure if you want to continue improving in your particular role, staying at the same company is a relatively safe way to get some skills in other roles.

A major downside of staying at the same company for more than five years, especially if you're not closely following industry developments and aren't proactive about self-improvement, is that you can find your skillset outdated. Five years is a lot of time in tech, and if you've been working on the same project for five years, doing roughly the same work, without upgrading it to newer technologies (often there's no need to do that), when you decide to look for work you'll find the job descriptions for your role have completely changed and you haven't worked with any of the new tools. You will have to learn everything from the beginning.

You can both reinvent yourself and become outdated at the same time. We know from the traits of the senior software engineers that they are continuously improving, so it's more likely that you end up reinventing yourself and staying up-to-date with the latest industry trends. Changing jobs every one or two years will expose you to a much larger variety of technologies and processes and will force you to learn more, especially if you choose to work in different industries. This cannot really be replicated if you keep working at the same company, unless the company is large enough and you have the freedom to choose what you work on.

Another major benefit of changing jobs more often (but not too often, aim to work somewhere for at least a year, or you'll find it harder to get hired), besides the variety of experience, is that you can dramatically increase how much you earn, much faster than if you were working at the same company (if it's not FAANG—but 99.9% of us don't work at FAANG). Even though, if you're good, you can get significant pay raises at one company, it's unlikely that you will persuade them to pay you two, three or five times as much as initially.

The salary ceiling for a senior software engineer is quite high if you consider the rise of remote work. If earlier you have been limited to several companies in your city, all paying roughly the same, now you can work almost anywhere in the world as a contractor. I was able to double how much I earned several times by improving my skills and carefully choosing the companies I worked for. You could also find a company that paid about the same but allowed you to work from anywhere and move to a lower-cost area, leaving you with more in the end (though you can also arrange this with the company you're already working at).

So, which is better? Here's my advice.

If you are a junior software engineer, I would suggest staying at the same company until you are promoted to a middle engineer, and perhaps a year after that. Going from junior to a middle engineer, in my experience, should not take you more than one or at most two years, even if you're new. Follow the advice in the section *Help your company teach you* and purposefully work towards becoming less dependent on external guidance. After you get promoted, stay for a year to really learn what the company has to teach you, learn to work on your own and try to figure out if you like what you're doing. After that you can re-examine the situation and see if what you want to do aligns with your work at the company.

Talk with your manager early and say: "I want to learn and get promoted to a middle engineer. What are the requirements, and how soon can I become a middle engineer if I fulfill them?" Notice that you are not asking to get promoted, you are willing to do the work so that you can get promoted based on your performance, and not because you've been with the company for some time. This is a business transaction: you perform your part of the deal (do the work) and the company performs its part (promotes you).

The important bit here is to get the requirements, the timeline and the promise of the promotion in writing to keep the company accountable. Essentially, it's a contract. Go to your manager every week and show how you have progressed towards the requirements (by showing the work you've done and what other engineers say). When the time comes for the final check-in, you will be in a position of power since you've kept your manager in the loop for six months or more, and you will most likely get your promotion after the next annual review.

Sounds like a fairy tale, and some of you may say that the manager will definitely manipulate you and feed you with empty promises. That is a possibility (and there are ways to deal with that, one of them is to get their promise in writing like I've suggested), but our goal is to improve our skills and get promoted, and get more responsibility, and the only way to do that is to get people who make these decisions on board, and that includes our manager. We don't make threats and it's a natural desire to improve our skills in the best possible way, so letting our manager know that we want to be a better member of the team is logical. Asking for a pay rise may be stressful for a lot of people, but we're discussing being more useful to the company and the company raising your grade to reflect your new skills, that's all. We are not asking for more money for doing the same work.

As a middle engineer, you have more options, and what you do depends on what you want. Revisit the section *Where do you want to go?* for a list of questions you can ask yourself. If you're happy working at a company and you're learning, then stay. Finding a good place to work depends a lot on luck, and the rise in pay will not make you happier if your job is stressful and doesn't let you improve. Unsurprisingly, you can use the same tactic of making an agreement with your manager on increasing your pay conditional on you fulfilling some requirements first, and doing market research.

This is not a negotiation book, just an overview of what you can do, but I'll say this. I find that you can decrease the stress of discussing your salary at work by doing thorough market research and agreeing to do the work first. If you follow my earlier advice and know what skills and knowledge the engineers in your role need, you can compare what you do at work to other job descriptions and salaries for similar roles in companies similar to yours. By doing this, you shift the discussion from yourself ("I want more money") to objective facts, and this is especially useful if you find you're underpaid. Knowing your value helps you prevent your company from manipulating you. In a similar vein, asking "What do I have to do to get a 5% pay raise?" and having a written agreement shows that you're willing to earn a higher salary by doing more difficult work first and proving that you're creating more value for the company, rather than simply asking for a raise.

Now that we started talking about market research, let's see what you can find out when you start looking for work.

4.3 Reconnaissance

So, you have decided that you are going to look around and try to find a new job, one that fits your goals better than your current one, or if you have been "let go." Here I'll describe my general approach to finding a job that I've learned from several sources and have used successfully over the years to work at companies and teams that I enjoyed. I'll say right away that the process is not fast, and if you need work right away, only some techniques will be useful to you. In return for your time, you get a higher success rate and a job you know you'll like (because you've done the research).

Reconnaissance in the title of this section means that finding work—or deciding you're good where you are—is all about research (also known as "googling a lot"). The central principle is instead of saying "I want to find a job as X developer" and starting to fire off the same résumé to a hundred companies, we will find what is already available, and if it's available to you. Just like when talking about your promotion, we shift the focus from you and what you want to facts and data, what jobs are already advertised and what real companies are looking for in candidates.

This shift is powerful because we are not simply presenting ourselves and looking for any takers, any company that would hire us. We learn which skills and knowledge actual companies are already looking for, and can adjust what we offer accordingly by adding skills that we're missing and not talking about those that are irrelevant to the role you're after.

This research into job postings is almost exactly the same as the research that I've suggested to find out about the knowledge that companies expect from different levels of software engineers. We find and read job postings for roles that are similar to yours, in companies that hire in your area, the area you want to move to, or remotely. What are we looking for? We need to find patterns and details that we can link to our own work experience: specific languages and technologies, tasks you are expected to do, education and certification requirements (if that's relevant to your industry). By doing this research, we learn what *is* relevant to your industry, because each is different. It can also answer almost any question that we have, and whether we need to know or learn something or not.

The next step is to take the list of companies that have roles closest to what you're after, and find them on LinkedIn (or Xing, or whatever is popular in your country). We do this to find the *people* working in these companies in roles similar to yours, and look at their profiles. By doing that, you compare them to your own experience and in their job history you can find answers to such questions as:

- How often do people get promoted?
- Did people get the same or better (or worse?) title when they changed companies?
- How long do the people stay in roles similar to yours? Do they leave or get promoted after that?
- What do people *do* while they're working in these roles? The description of a role in someone's profile can be very different from the job posting.
- What are the skills of people in the same role at other companies? Compare them to the job posting for that company and you will get an idea of what you actually need to perform the job versus what they list as necessary.

By finding the data about people and companies hiring for the roles you want, you can objectively compare your skills and knowledge to the trajectories of other people in the same role. You will know if you can apply to a higher level role and if not, what exactly you are missing. You will not need to guess and hope.

If you're a middle engineer and are wondering if it's worth applying for a senior role at company X, find the senior engineers on LinkedIn (or from the Team page on the company website, or on GitHub through the company's organization, or on Twitter) and look how they've got their title. If you find several people who have been middle engineers at other companies before they've been hired, with experience and skills similar to yours, there's your answer.

Does it sound like something a stalker would do? It may, but we are not after any personal details, and our goal is not chasing people and sending them creepy messages. We are collecting intelligence (remember the section title?) that will help us make career decisions by looking through publicly available information, nothing more.

Some companies make it much easier than others by having a public playbook with their ways of working (so you can know what team culture you're getting into), an engineering blog, their engineers speak at conferences and have their contact details available, and so on. Letting prospective candidates know how cool it is to work at a company is free advertising and attracts talent.

Why would you spend so much time and effort finding information about a company if in the end you may not even apply, or it doesn't have any open positions at the moment? The reason is that the more advanced you become and as you improve your skills and learn your preferences, the more you *choose* where you want to work and have requirements for the company just as the company has requirements for candidates. Disqualifying a company early through a little research (even though you could get a job there) saves you time to apply to other companies that do fit your requirements. Absolutely, this is a privilege, but it's a privilege you earn through hard work and improving your skills.

Let's look at it from a different angle. If you wanted a stable 9–5, would you apply to a company where software engineers work six months on average before leaving, and its Glassdoor rating is 2.2? They would gladly hire you. An hour of research will save you a month submitting your notice and looking for work *again*. If you are a junior software engineer, do you apply to a company that only needs headcount to charge a client more and doesn't care about personal development? You will only lose time doing menial tasks. Even if you don't have a choice and need work, knowing about the problems in advance can give you leverage during interviews.

4.3.1 Role + company

All this talk about companies is there because your next job will be you working in a specific *role* at a specific *company*. You'll find companies that have people in roles like the one you want by doing research, and by talking to people (more on that in the section *Interesting people*).

You know your preferences, and by researching each company in turn you will remove all that clearly don't match and those where you wouldn't enjoy working. Again, through your research you will know that your skills and qualifications match those of other people working in the same role that you want to apply to, and you are not clearly under- or overqualified.

Starting with several companies from the list gives direction and purpose to your job search, and knowing for which role you are going to apply, and at which company, gives you clarity to how exactly you should do it.

Instead of copy-pasting your résumé and sending it to ten companies that you don't care about, you will edit your résumé so that it's using the same technologies and phrases as in the job posting and highlighting that you've done similar work in the past (you know what's relevant from other people's profiles). You will write a cover letter saying why you're a great candidate and will mention the qualities that you know the company is looking for. Maybe you've found the hiring manager's blog where he's telling exactly what he wants to see in the cover letter and the résumé (this happened to me, even though I found it after I was hired).

This takes more work than copy-pasting, but the chances that your application will be noticed among hundreds of others and you'll be invited for an interview are astronomically higher with this approach. We don't need to be the most qualified or the best, we only need to be better than other candidates, and at least 80% don't care, so you immediately leave most of your "competition" behind.

Finding people who work at the company you want to apply to often lets you see if you know someone, or know someone who does. Yes, it is time for the dreaded "networking."

4.4 Interesting people

According to some statistics I've seen and some anecdata, between 60 and 80 percent of all job openings are filled through networking (which means, the person was referred by someone they knew at the company). Consider the implications when a job opening is published: three people known to someone at the company have a 60% chance of getting the job between them, while the other forty applicants share the remaining 40% chance. Do you want to have a 20% chance of getting hired or 1%?

And that is only for jobs that reach the open job market in the first place. Some, if not the majority, are never published on a job site, but instead offered to people the team already knows. Only when the connections are exhausted, the company draws up a job posting to find external candidates. In our (OK, maybe not "our," but definitely "my") favorite "Don't Call Yourself a Programmer," Patrick McKenzie says:

> Most jobs are never available publicly… Information about the position travels at approximately the speed of beer, sometimes lubricated by email. The decision-

> maker at a company knows he needs someone. He tells his friends and business contacts. One of them knows someone… Introductions are made, a meeting happens, and they achieve agreement in principle on the job offer.

With that in mind, how relevant do you think is generic advice on the internet about how you should format your résumé and how it will get you the job? It completely misses the point.

What does it mean for us? We have to remember that when we are cold-emailing a company, no matter how good a fit for the position we are, we have a lower chance of getting the offer than if someone vouches for us.

Why is it like this? It’s a people thing. Hiring someone is a big risk for the company, because no matter how long the interview process is and what you talk about with the candidate, they might not work out and the company loses money. While the candidate’s technical skills can be tested (up to a point), no amount of behavioral questions will help the company understand if the candidate is dependable or can be trusted.

There is not much trust you can develop over a few hours of interviews (and often just one hour). Because of that, a person familiar to someone in the hiring process already has the level of trust much higher that anyone unfamiliar can achieve during the interviews. As a result, if you are technically competent enough to do the job (not worse than other candidates) but have a higher level of trust than others (that is, the company believes you have the qualities necessary to do good work on a team and are dependable), you are likely to get the offer.

People kind of know this through the advice to “build your network,” but it doesn’t mean adding lots of people on LinkedIn (the phrase “I’d like to add you to my professional network on LinkedIn” became a meme), or asking random company employees for jobs, like many believe—partly because this advice misses the point just as the one about résumés.

Building a network is increasing the level of trust between you and other people (as technical as this sounds). The level of trust between you and a random stranger is very low, and there is an infinite number of strangers. The level of trust between you and a friend is very high, and in this day and age, we have few people we can call real-life (not

Facebook) friends. Between these extremes is a long spectrum, and in this spectrum lies our network, the people who know us.

Building a network, then, is akin to making friends, and that is what makes it scary for a lot of people. Not only because it's hard, but also because we don't want to share any personal details (other people might hurt us), or go out of our way to talk to other people. We often don't talk much to people at work either, nor do we want to.

I propose to dial the expected level of trust (and thus, the risk to ourselves) way down, because we aren't looking for friends, or even acquaintances, and focus on bonding with other people in public spaces over shared interests, ideally related to our work.

What does it mean, exactly? You're probably interested in something enough to seek out groups or communities online to talk to other people about it. Software developers, unsurprisingly, are often interested in programming, and we participate in programming language forums, library communities, contribute to open source, hang out on community Slacks, and so on. That's what it means to bond with other people over shared interests related to work, and that's what a lot of us are already doing.

If you are doing that, congratulations, you are already building your network just by talking with other people about something you're interested in. It's often fun and definitely non-threatening, because your goal is to learn and help out other people with their questions, and in the case of contributing to open-source, directly solving people's problems. People get to know you as a member of the community who's there to help and is non-abusive. You are building trust.

This is what I call *active* networking because it requires you to seek out people and communities, and talk to people. Still, it's relatively low-maintenance since all of that is online, doesn't require much time, and you don't have to reveal any details about yourself that you don't want. Still, these community relationships can even grow into friendships as you accidentally find people whose interests are closely aligned to yours.

The best thing about it is that you would probably still have done it even if you didn't want to "build a network," simply because you're interested in the subject. And our goal in this kind of communication is never exploiting people, but learning from them and helping them. It's also a wonderful way to de-stress and disconnect from work. Some of that community participation can be *at* work, like discussing a programming language issue on its official forums, and is directly related to your career.

Sometimes, though, we don’t have much energy outside of work to participate in anything else work-related. Don’t feel like you have to. I’m often very low on energy, so I usually avoid communities that are strictly technical, since I don’t want to “work” all day. But I still participate in others, which are about completely different interests, and chance has it that a lot of people in those communities are also software developers, so over the years we’ve got to know each other, and there are some job discussions now and again. Even by trying to avoid networking on purpose, I still did networking.

What about networking that is *passive*? This is about creating and publishing something interesting (and related to your work), so that other people come to talk with *you* because they find you interesting, without you having to lift a finger—well, besides doing the work. It’s writing a blog, or speaking at meetups and conferences. You can also do all of that at work. If you speak at a meetup or, even better, a conference, your talk is not only giving you credibility as an expert, but is also free advertising for your company, because people get interested in working at a company where you do cool stuff that you’ve talked about in your presentation.

This passive networking can be both less and more stressful for different people, in contrast to what I’ve called *active* networking. Less stressful, because it doesn’t involve talking (a lot) to other people or helping them with their problems, it’s simply you sharing what you’ve learned or done. And at the same time, more stressful, because public speaking and publishing your opinions for everyone to see *is* stressful even if you do it semi-regularly.

By doing both, not with the goal of “building a network” or finding people that can get you the job you want, but in order to learn and help others by sharing what you know, by being interested in what other people do and sharing interesting information for others, in a year, two years, three years you will find that you’ve actually built a network of people who know and trust you, and you’ll get opportunities that you have never expected.

4.5 The interview game

> I want to play a game.
> Jigsaw

Even more stressful than networking are the interviews. We don't change jobs often, and we're usually out of practice. We feel like we are weighed and judged. Our performance in the interviews often reflects our self-worth. Getting and, most importantly, doing well in the interviews is another skill we can learn and improve at.

Unfortunately (for us) interviews in tech often don't allow us to demonstrate our qualities and how well we will do in the actual job. Software development is very rarely about solving problems on the spot, or talking to strangers about how good you are, but that's exactly what the interviews are like. There are perpetual discussions about how hiring in tech is broken, and every week there's a new startup promising to fix it. Still, that is the reality and we have to accept it whether we like it or not. Dreaming about a perfect world won't get you your next job.

I find that thinking about the interview stage like a game helps me step away from the notion that my self-worth is at stake, and that if I fail it means I'm "bad." As any game, it has rules and optimal strategies, and you can't win every round. If you take the time to read experiences of people who have been rejected, you'll find that no matter how skilled the person is or how well they can perform the role, much of the outcome is due to chance. The best you can do is shrug, get some feedback and do better next time.

I know that getting an interview is not easy, and sometimes everything is hanging on whether you do well or not (for example, if you are interviewing at your dream company). We cannot control the outcome—our interviewer may have been grumpy, or our hiring manager forgot to make some notes, or any other reason that is not our performance and is completely out of our control.

What we *can* control is what we do before and during the interview, and that's what we should focus on. All athletes train to win, but in a competition, there is only one winner. The good thing about interviews is that you only have to be better than others, and most people make easy mistakes (that you will know to avoid), and don't prepare or practice (you will). By getting marginally better, you dramatically increase your chances.

We already know that through networking, you can partly or completely skip the interview round, giving you a massive advantage. Let's look at some other rules.

4.5.1 The rules

These rules are heavily based, among other things, on Andrew LaCivita's book "Interview Intervention." He is a master recruiter who has brilliantly summarized the whole process, from preparation to negotiation and first days on the job. I consider it essential reading. I've been collecting my sources and data for several years before I discovered LaCivita's work, and he has confirmed and expanded everything I have known.

- The only purpose of the résumé is to get you invited to an interview. After you've started interviewing, it becomes almost irrelevant, if only being a starting point for excursions into your past experience.
- By inviting you to an interview, the company is saying that you are qualified for the job.
- Your goal at an interview is not to get the job (that decision is made much later), but to get the next interview or an offer.
- The optimal strategy is to uncover and resolve any of the interviewer's concerns about you and leave a positive and favorable impression. The main concern is "Will you do well at the job if hired?"

Notice that this is general enough that it also applies to technical interviews, even purely whiteboard coding. If you clearly explain what you are doing and achieve the result the interviewer wants, you are making a positive and favorable impression, and resolve the concern about your approach to tasks at work under pressure.

There is much to say about interviewing, but I don't have the space or the time to put it all here. Again, I strongly suggest that you read "Interview Intervention" because it teaches much better than I ever could what to remember, how to answer common interview questions and the general approach, and more.

Let's unpack the rules a little bit.

Your résumé and your cover letter are what get you the first interview, but that's about it. They are used primarily as a filter and are not needed again. Remember that hiring managers can get hundreds of applications for a position (this is becoming even more relevant with remote work on the rise because anyone in the world can apply), and you have about ten seconds before your résumé is rejected as irrelevant.

Now consider what matters if you have ten seconds to persuade someone you are qualified for the position. Does your font or the size of the margins matter? No, as long as it's readable. And nobody has time to read your five-page document—if you have less than ten years of experience, you can fit everything on one page. We'll look at what to put on your résumé in the next section.

If you have been invited to an interview, that means you are qualified for the job. Congratulations. Why would the company invite (on purpose) someone who is not qualified? Let that boost your confidence. The company wants you to succeed and confirm its belief that you are competent.

When you are in an interview, think small. It is too early to talk about getting the job, compensation and how many days off you get—leave it for the day when you get the offer. The goal of the interview for you is to answer any questions the interviewer might have for you, and also to have your own questions (prepared in advance) which help you evaluate the company as it evaluates you. Remember that we choose the company as much as it chooses us. We have a set of requirements too, and if something is important to us, we can find that out early on in the process. Beyond that, our only goal is to make a good enough impression that we are invited to the next interview.

Ah, the positive and favorable impression. It sounds so simple: just come to the interview and "answer some questions." But that's how you bomb. The interview is a ritualistic dance where every question is a double question, and the interviewer is actively reading your non-verbal signals. That's why I'm saying it's a game, because often it makes no sense.

Nevertheless, by doing anything at all you're making an impression on the interviewer. We want this impression to be what we want, and for that, we must decode what the interviewer (let's call her Irma) wants to hear when she asks where we see ourselves in five years (everyone's favorite question). Depending on the position, the company and your résumé, it might mean anything from the concern if you can even *stay* at the company for more than a year (if you've changed five companies in the past five years) or to see if you understand how promotions work in the industry, or what skills you want to get (and if you can get them at that company).

In any case, behind every question is a concern about you that you must find and resolve—all while making sure the interviewer is paying attention. Remember that the interviewer is a person, and she might be tired, distracted, busy, any and all of that, and it's *your* job to make sure she doesn't zone out. If you ramble for ten minutes in reply to the question "Tell me about yourself," well… you're not getting invited to the next interview.

As you can see, we are trying to do multiple things at once: look and sound relaxed, positive and confident, give crisp, detailed answers, keep the interviewer's attention and guess what concerns they have about us—all of that while mentally watching how we perform and course-correcting.

It's like juggling burning torches while standing on a balance board, and guess what happens if you don't prepare and practice? You fall on your bum and your hair gets singed. At the same time, a lot of people wing it and sometimes don't prepare at all, some because they believe that they don't own any preparation time to the company, and the company should accept them how they are. I can get behind that. I have strong opinions too. You can choose what suits you, I'm only describing the optimal strategy to get ahead in a competitive environment. If we want to succeed, we will put in the work. And by doing that, we'll dramatically raise our chances by being prepared better than anyone else.

All of this work for the company, but what does the company do for us? At the interview stage, the company is in a stronger position, since there are many candidates and only one offer. Conversely, we are in a stronger position when we reach the offer stage, when the company likes us more than anyone else and *wants* us to come work there. When you have the offer, you have the most leverage and can negotiate, and that is the time when all your work pays off, but not before.

4.5.2 Front-load the work

What does it mean to be prepared, what do you do exactly, and why? To be prepared is to be ready to face any challenge we may come across during the interview process, so that we show our best side and prove to the company that we are the best and most capable choice for the position.

This preparation starts even before you apply, because you have started researching the company even earlier. To prove we are the best choice, first we must understand what the company wants, what problems it wants to solve by hiring a person for this position, what it's looking for in the perfect candidate, and how we can show that we will solve these problems.

Our application starts with our résumé. You already know that often you don't have much more than ten seconds before it's rejected. What we want to do is to edit our "full" résumé in such a way that it provides a clear narrative, directly related to the position we are applying to. Read the job posting carefully (and anything that tells you what the company looks for in candidates) and filter your work experience through that. Decide in advance what impression you want the reader to take away after reading your résumé. If possible, use words and phrases directly from the job posting.

If you worked at company X and did Y and Z (and a bunch of other things), if Y is in the job posting, put Y first. We want the reader to immediately notice that we have successfully done the same work in the past. Be specific and focus on what you have achieved (ideally, with numbers) rather than what you did. "Led and delivered project X 15% ahead of schedule" is better than "Worked on project X." The job posting and what the company is looking for in candidates will tell you exactly what you need to mention and highlight. Keep it short and try to fit everything on one page.

Our résumé is a sales document, not a list of everything that you've done in your past jobs. The hiring managers don't care what you've done, they only care if you can do what the position requires, and more. The reaction we're looking for is "Wow, I want this person to achieve the same numbers for my company!"

The purpose of a sales document is to persuade the reader that the advertised services are exactly what the reader needs to solve their problem. If you are able, discern specific problems and technical challenges that the company is trying to solve and fit them into your narrative. For example, if you notice the company has just published their SDK and the documentation is lacking (as usual), don't leave out the fact you've been in charge of the documentation effort at one of your past companies.

Compare your thoughtfully written one-page (OK, you can make it two, but no more) "sales letter" to a five-page "generic" résumé that lists in excruciating detail everything the person has done (with a list of buzzwords on top). Which do you think is more likely to attract attention? With the "sales" approach, you're ten times more likely to get invited to an interview.

Even if you do not want to spend the time to rewrite your résumé for each company you're applying to, at least adjust your "full" one so that it tells a story *you* want and leaves an impression of *your* choice, highlighting the direction you're moving or what you enjoy doing: "a front-end engineer with an eye for design," "a generalist who has led and shipped popular products," "a Java developer specializing in high-load services," and so on.

Do this once and it will pay back for years because people will have an impression you control instead of what they choose to pick and notice from your detailed résumé. If you ask: "Doesn't it limit my options?"—appealing to everyone rarely works, and you don't have to leave out details, just make some more prominent than others and phrase them differently.

Now, how do we prepare for the interview itself?

The technical one is often about putting in the hours. If it's a LeetCode-style interview, drill computer science problems, if it's more about the work, review what the engineer of your level should know about the platform and popular libraries, frameworks and approaches. You will know what kind of technical interview it is from your research about the company. Don't skimp on the technical preparation, even though I'm fussing about the soft skills all the time. Technical skills get you through the first round or two of the interview, and as we know, our goal is just to get to the next round.

The thing is, most software engineers' technical skills are fine, the problems we have are in the talking department. I'm making such a focus on soft skills because they are likely deficient, and by improving them, we increase our chance of success significantly as they're just as important as the technical skills in getting you hired. Having better soft skills also makes up for some deficiencies in our technical skills that would otherwise be a problem.

I'm an average programmer, but in a team setting, my communication skills make up for it because I talk a lot with stakeholders, present technical decisions, write howtos for non-technical people and extra documentation for the project, and so on. I let people on the team who have superior programming skills shine in their realm by taking more of the communication burden myself, and as a result, the team effort is amplified, a win-win situation for everyone and the company.

What I'm saying is, the "behavioral" part of the interview is usually more difficult for the software engineers. We prepare for it—you guessed it—by doing more research, but also by practicing. We can find the lists of popular questions online, and as I've said earlier, every open question like "Tell me about yourself" or "Why do you want to work at this company?" is an opportunity for us to dive below the surface, try to guess what the interviewer *actually* wants to know, and tell them in a way that leaves a positive and favorable impression about us.

Among other things, the interviewer will want to know how you handle conflict, problems, deadlines and other mishaps. You are unlikely to come up with a good answer on the spot, and what we can do here is to recall and collect some stories from our experience that illustrate how we behave in these kinds of situations. Write these stories down, with specific details if possible. You can do this once because we will reuse them for all interviews.

When getting ready to interview with a specific company, combine the common interview questions with your research about the company, and in a new document, actually *write out* potential answers to these questions. They don't have to be long and detailed, a few words or a couple of sentences is sufficient if it lets you remember what you want to say in response.

In your response to "Why do you want to work at this company?" you can outline your most relevant experience and how the direction you're moving aligns with the position. In the response to "Tell me about a time when X" reference one of the stories you're written down, the one that is most relevant to the role, or one that shows qualities that are more important than others. By doing that, we plan our arguments and pick the right words without time pressure, in a calm environment, where we can be the most effective in our thinking. When the question comes up at the interview, we smile because we know exactly what to say.

But we can do more, and very few people actually do that. We can practice as if we are at an interview, and recite our responses (looking at our reference document) while watching that we put emotion in our tone, don't stumble and speak like a person. You don't want to sound like a robot reading out a corporate press release—yet that is often what happens when we're under stress and are talking to a stranger who's judging our every word.

Try to say the response to every question aloud at least a few times. By doing that, we commit the complete responses into our subconscious memory, so that when we're on the spot, our brain will feed us just the right words (because we've already said them, more than once).

Remember our juggling analogy? Instead of trying—on the spot—to compose a coherent and smart response to an open-ended question that doesn't have a clear answer (but can turn the interview) while watching your body language at the same time, you can deliver the response that you've already practiced and free up some of your brain juice to watch over *how* you're talking, how the interviewer is reacting and what you're going to say next.

The research, preparation and practice also give you a large amount of confidence both in your fit for the role and your interview performance. Fear and nervousness come from uncertainty. By trying to persuade the company that you're a perfect candidate, you persuade yourself. By knowing that you can handle any question the interviewer asks and knowing exactly what you'll say, you decrease the uncertainty and will be much more relaxed (which both makes a better impression and helps you think).

We're not making a "fake" or untruthful impression of ourselves by practicing interview responses and working on our body language. On the contrary, I think we are making a *more accurate* impression, because the majority of software engineers are simply lacking the interviewing skills, and by preparation and practice we bring them up to par.

Finally, we would all prefer to have a casual chat with our future boss and coworkers, and then get a generous offer, and sometimes this happens (especially if you do networking). The unfortunate reality is that we often have to make cold applications to companies we'd love to fork for, and do interviews, competing with hundreds of other applicants (because there are many other people who'd love to work there). In this competitive environment, the interview is a business transaction and a ritual with its own set of rules and skills, and we must learn to play the game if we want to succeed and advance in our career.

5 Prevention and control

As software engineers, we have our professional ailments that can prevent us from being our best at work. I will look at three that I think are the most common and the most limiting, from the more severe that can completely stop you doing any work (burnout), to the more benign that can cost you opportunity (impostor syndrome) and time (procrastination).

Being white-collar workers, we are protected from grave physical injuries, but we can still cause ourselves permanent physical harm by neglecting the ergonomics of our workplace. You get neck and back pain, headaches, eye strain from sitting in an unnatural pose all day, looking at a screen that may be too bright or too dark, often with text so small that you have to squint. If you're typing or using your mouse a lot with a bent wrist, you can get what is called a "repetitive strain injury" (RSI), a near-constant pain whenever you use the computer. Sitting for long periods of time is not good for you, either, as regular chairs restrict the blood flow in your legs, not to say how people are destroying their back by slumping or perching.

Our software development career will likely last for decades, so please take the time to educate yourself about the basics of ergonomics. If you are working at a desk, when sitting straight, your eyes should be higher than the center of the screen. Put your display on a couple of thick books. If you have a laptop, get an external keyboard and put the laptop on a big box. Having your screen high instantly prevents you from slumping your shoulders and straining your neck by looking down. To prevent sliding down in your chair, get something to put your feet on. If you sit straight, your arms at the keyboard start to look like an L, not like a V, so your wrists are straight too.

I had bad headaches early in my career from eye strain, so I made it a habit to bump up the size of the fonts I used when writing code, and I never had eye strain again. If you feel even the slightest tightening around your eyes when you read code, that means your eyes are doing extra work.

And that's it—by following even these few simple rules you don't unnecessarily strain your body so you'll be less tired after work.

Now back to our regular programming.

5.1 Burnout

Earlier I've described burnout as "chronic weariness" but it's not really that. You don't feel particularly tired, or sleepy, or anything like it. It's hard to notice because it creeps in gradually, and then you're already in the red. I burned out (to various degrees) several times, and even knowing I was susceptible, I didn't notice for weeks.

What is it, exactly? Burnout is a state of mental exhaustion that results in three things: you are unable to work, your job feels meaningless and unsatisfactory, and you feel cynical toward your coworkers. Where once you were productive, enjoyed your job and felt that you and your colleagues were working towards a common and meaningful goal, you would become numb and negative. Nothing actually changes about your job or your coworkers, only your attitude.

I'm saying "unable to work" quite literally. In the later stages, when you try to write code even for the simplest tasks, nothing comes out. The feeling is surreal. In the beginning, you have some difficulty focusing on tasks. But we are determined and resilient, not afraid of setbacks, and we power through. This additional mental effort required to overcome resistance compounds and exacerbates our exhaustion, until the resistance is stronger than the additional effort we can muster. It feels like your mental capacity is limited, and it is, because you've drained all the reserves of your brain juice required for concentration.

Negativity towards coworkers and work becoming meaningless are the two sides of the same coin—the lack of resources to care (because you're exhausted). Where earlier you *cared* about the quality of your work and the product, *cared* about your fellow software engineers by pouring your energy into coding and communication, when you've burnt out, there is nothing left to pour and your brain starts considering work (that requires concentration and thinking) and talking to other people (that requires compassion) a drain on your internal, mental resources, and shuts both down.

There is not one reason that makes us burn out, there are several and most often they need to influence us for some time before there is a negative effect. Burnout is also a very gradual spectrum, not something you have or don't. Our mental resources are like a well—we drain some water to do our work, and then some water is replenished. If we drain the well faster that it's replenished, we have to also spend extra effort to lower the bucket deeper and deeper, until we hit bottom and the well runs dry.

Let's look at some of the reasons:

- You work too much. Either you are forced to by deadlines or unrealistic expectations, or you're passionate and committed to doing the best work you can. Both result in spending more time working. When companies switch to remote work, managers fear that people will slack off, and some do. But people who are excited about their work and motivated (and a lot of software engineers are, according to Stack Overflow surveys) start working *more*, not less. It's too easy to start checking email during breakfast, skip lunch or stay an extra hour fixing an elusive bug. "Easy" work becomes a replacement for evening pastime. Do that enough and it will begin taking its toll. Most people can only do four to six hours of engaging mental work (like programming) per day before their focus and the quality of code sharply declines. Eighty-hour weeks have been proven to be net negative for long-term productivity. You can only do so much extra work before you need to recharge.
- You don't sleep enough. We all like to stay up coding, don't we? Regular sleep deprivation has been shown to decrease productivity and mental acuity. We often come up with our best solutions while sleeping. Sleep on that tough programming problem, and you'll wake up with a solution. Sleep is also when our well of mental energy is replenished, and by skipping sleep often, you're taking out a line of credit that you won't be able to repay.
- You don't get feedback or recognition at work. As I've said earlier, we can get submerged in a neverending stream of Jira tickets. We need to rise to the surface to breathe, and that comes in the form of feedback and recognition of your work. Some of the toughest time in my career was working on a project and not knowing how it was doing in the hands of customers. Our management at the time didn't care as long as we added new features to the project. I cared about the

product, but objectively, it didn't matter if I did shoddy or great work—there was no feedback. When the hard work you do is not recognized, you feel like you're flushing your effort down the drain. It drops your motivation, and again, you have to spend extra mental effort to do the same work which you now perceive to be less meaningful.
- You have goals that are hard or impossible to reach. Impossible deadlines and crazy requirements are obvious, but even the best teams make the mistake of constantly trying to hit goals that are a little bit too ambitious. An extra couple of stories per sprint, a release date that's a little too soon, a misplanned feature that turned out to be harder than expected, and instead of hitting a milestone after milestone, the team gets stuck in a "death march," always crunching on a feature that's a little too late. This brings no satisfaction of a job well done, and every push takes extra effort which soon drains the team's resources. Productivity tanks and people burn out.

Notice that most of these reasons don't even need any extra source of stress at work, and happen to great teams too. Now imagine if there is also a toxic manager, malicious coworkers or a CEO who changes the company direction every two weeks. This all *compounds* the effect.

What can you do to prevent and control burnout?

- Watch out for its signs. Pick an online burnout questionnaire that you like and check yourself every month or so (put it on your calendar). Notice if you start having trouble doing work ("I just need to work harder") and feeling demotivated. For me one of the first symptoms is negativity towards coworkers, I start avoiding talking to people.
- Leave work at work. It's a bit easier when working at an office. Even though I had been working remotely, I was lucky to have a separate computer that I could turn off at the end of the day. If you only have one computer, you can create another account on it just for work (or a personal account if it's a work computer). Mentally disconnecting to stop thinking about work in the evening is a bit harder, and it doesn't help that we software engineers often don't have much of a life outside of work. The easiest way to disconnect is to do something else that is engaging right after you stop working. By doing that you conserve your mental energy for tomorrow. Working less might seem disingenuous, but trust me, burning out is a

bummer and you're far better served by being more alert and doing less work consistently every day.
- Work on decreasing your level of stress. I know it's hard, and that some sources of stress are out of your control (which is an additional source of stress). Still, you can do *something* and that may tip the balance so that you replenish more resources than you spend. You know the usual drill to begin with: exercise, eat well and get enough sleep.

If you ever find yourself deeply burnt out, the best remedy is complete rest for at least ten days, and more if you can afford it. "Rest" doesn't mean lying in bed all day, you're not sick, rather you are not allowed to do any work and even think about it if you can. Of course, if you like to stay in bed all day, go for it. It also doesn't necessarily mean going somewhere for a vacation. Vacation is a change of pace, but you're exhausted and need low-stress activities, whichever those might be. Goof around and do what you feel like doing. People often talk about "recharging their batteries," and that's exactly how it should feel like.

You will know that you're doing it right when you get a spark of excitement about going back to work (but don't think about work too much yet). You will miss your coworkers because you'll begin thinking about them as friends again. Your resentment towards sitting down to work will wane. If you've rested enough, you'll be excited and happy to be back. Remember this feeling, and how burnout felt "normal."

Routinely being on the edge of burnout at work is not normal. If there's so much stress that you have to consciously expend effort to manage it, consider changing your role if you have the option, even internally in the company. The field of software development is huge, and the types and modes of work are many. Often you can offload some work that is stressing you to other people on the team who are OK with doing it or even enjoy it. Also please remember that you can ask for help. You don't have to do it all alone.

5.2 Impostor syndrome

We know from the section *Exponential learning* that you can't know everything you "need" to know. That also means you can't learn to *do* everything. This perceived lack of knowledge and skills becomes the primary reason for what is called “impostor syndrome”—a capable person fearing they will be discovered and outed as a “fraud” and discounting their previous accomplishments as misjudged or lucky.

This fear and the resulting self-doubt cause stress, anxiety, even depression, and a large percent of people is susceptible to it, especially when faced with new tasks and challenges different from what they’ve done before. In software developers, impostor syndrome is fairly common because of the extraordinary demands in the breadth of skills and knowledge that people expect from themselves and others, and how quickly one’s confidence may be shattered by “simple” tasks.

We can feel we are just flailing and trying random things until something works, or have no idea how to approach a problem at all, before somehow solving it. We think that other people know what they’re doing, and only we are struggling. If we are praised for a solution, we can say: “The task was easy” or “It was obvious in that context”—ascribing our success to external factors, rather than our skills and experience that made the task “easy” and “obvious.”

Research has found that what helps is to learn that other people also have these feelings. In fact, *many* people have them, and you can find threads on Twitter and tech forums where accomplished, respected developers say they feel like impostors.

Probably, the key component of self-doubt is that you don’t know if you can successfully apply your past experience to the tasks that lie ahead. You are not sure you’re up to the task, and if what let you succeed in the past was luck or chance. But that is the nature of software development: every task is not like the other. We aren’t chopping logs. These feelings of self-doubt are normal, but we don’t want to let them block us from acting. Let’s see how you can try and curb self-doubt.

Consider your past experience. Chances are, you have overcome countless programming tasks before, working alone or with others. How you’ve done it, assuredly or by flailing, doesn’t really matter as long as you’ve done them—the proof is in the pudding. What exactly makes you think that you won’t be able to solve the next few tasks ahead of you, and *this time* your coworkers will discover you’re “fake”?

All the tasks you’ve faced before couldn’t have been easy, so you have overcome hard ones too. Let’s try to think constructively. Is it possible that you won’t be able to make *any* progress at all on these new tasks you’re facing? Even if they’re in a domain new to you, trust your experience. I know you can do it!

You have options: make some research and extract smaller subtasks, try to answer your own questions first and ask for help (as we know, asking for help is not declaring defeat), take some harder details away and try to solve a simpler problem first. In mathematical proofs, the first case to be proven is so trivial that it's obvious it is correct. By analyzing and disassembling the task, you are shifting your thoughts from feelings to logic, and by making a stair after stair you can ascend a mountain. When you see yourself making progress on what you have thought insurmountable, you'll trust yourself more.

Impostor syndrome can also prevent us from stepping on the path to growth—looking for and accepting more challenging work. If we think we aren't worthy even of what we have now, how can we allow ourselves to dream of rising higher to an even more precarious position? We think that we are less than what other people think of us. We don't even bother applying to jobs we want but think we won't be able to do.

We're entering the territory where I'm not comfortable giving advice (because I'm a fraud too, haha), so instead I'll tell you what *I* do. I've been working remotely for many years and most often I work alone (there's nobody around to help me with the task) and don't talk much to other people. I'm self-taught so I have gaps in my knowledge where a new graduate wouldn't. I also struggle with procrastination (the next section—my favorite) and self-doubt.

I tend to trust my gut feelings a lot and work on training my intuition, but *relying* on feelings to make decisions has often gotten me in trouble, and that is the same with the feeling of being an impostor and low self-esteem. When working alone, I can get stuck in the cycle of dreading the big problem that I can't solve instead of attacking it from different angles. When not talking to anyone, this often produces negative self-talk: "I can't do this," "It's too hard," "I don't really know anything."

Notice that all of that is pretty vague and usually relates to the problem as a whole, because I can't muster the strength to analyze it. I've learned to recognize this (most of the time) and instead switch to a constructive approach: "OK, I can't do the whole problem, but what *can* I do?" Thinking back to past experience also helps: I know I can tackle some serious problems because I've done it in the past.

I don't outright consider myself a fraud (disregard the lame joke above), but probably only because I have done extensive research on what knowledge and skills I can reasonably expect to have in which domains. I know I have gaps in my knowledge, and in my heart I know I'm not a "real" programmer because I can't pass a whiteboard interview, don't know much CS theory and I don't effortlessly think in abstractions. I'm not afraid that someone will "discover" this because fear comes from uncertainty: "What if someone asks me? What if they tell others?"

I freely admit my ignorance, but I'm also certain in the knowledge and skills that I *do* have and *actually* need (and have needed in the past) to do my job. If I'm expected to have knowledge I lack, I'll start to acquire it before I need it. I know that I can't know everything, but I can find and learn whatever I'm missing.

This confidence that I can tackle almost any problem (eventually… even slowly and with false starts) lets me treat myself as more capable than I actually am, because I can find and learn any missing knowledge and skills that I discover by analyzing the problem. I don't know if it helps you or not, but it looks like this overconfidence is my solution to self-doubt. We jumped from impostor syndrome straight to Dunning–Kruger :) I guess my saving grace is that I *know* I'm bad at some things, but I also know I can get better, and being bad is not permanent.

5.3 Procrastination

We all know what procrastination is: when you need to do something, you choose to do something else. Do you struggle with it? I know I do, a lot. Over the years I've learned some tricks, but I can only dream of achieving a decisive victory. It's devious and smart and never sleeps (just like me staying up at 5 in the morning reading Twitter).

I define it as your brain being smarter than you and saving precious glucose (the fuel your brain runs on) by telling you through procrastination that you don't have a good plan for what you need to do. You are not lazy or inept, your body simply tries to save some energy that it thinks you're about to spend in vain. Why does your subconscious come to this decision?

- The task is too big and when you try thinking about it, you get overwhelmed. We can only hold 5 to 9 things in our working memory at the same time (also known as "Miller's law"), and if our task has more moving parts than that, we literally

can't think about it. "Add authentication to website" is just four words, but it immediately spawns many questions and considerations, and we're overwhelmed.
- The task is too vague and has few details. "Write documentation for X" can sit in our todo list for months because it doesn't help us begin working. Who is the target audience? How detailed should it be? Do we have a place to store it? What is the tone of voice? Without answers to these questions, we can't start working.
- You don't know how to start. Do you do X first or Y? Then there's question Z that can completely change the implementation, and you have idea W that you haven't had time to explore. Sometimes we don't know where to start at all, especially with vague tasks.
- You don't know why this task is important. If we're indifferent to whether the task is done or not, why should we do it? "Ignore the problem, and it will go away."
- There is much time until the task is due (or it doesn't have a due date). Combined with the previous one, the task is doomed to be delayed until later.

As a result, you don't *want* to do the task right now, because it involves making some decisions (and making decisions is exhausting to our willpower), so you choose to do something else that doesn't involve making decisions. You check your work email and Slack, and after a few minutes you come back to the task again. You haven't made any decisions last time, and now that you've been distracted by email, you need to restore all the task context in your head again. Meh… better go check Twitter.

Remember how I mentioned junior software engineers left to their own devices? That is an environment ripe for getting distracted. The tasks are often a bit vague (even if they're simple) because they assume more knowledge of the project than the junior has. There is often no deadline. The junior doesn't really know where to start. If you're prone to procrastination, you can spend all week struggling to make progress on a task.

Almost the same can happen to senior engineers (exhibit A: me). I've learned to handle it, but still sometimes I can lose an hour or two before I notice I'm not really doing the task, but thinking about it while being distracted. Why would I let myself get distracted? Because flesh is weak, and the modern workplace is crazy.

The worst thing is that procrastination is not "doing nothing," it's avoiding doing a task *very hard* by getting distracted. Every minute you're trying to make a decision to go back to the task and make progress, but even if you do, you get distracted again. You are thinking both about what you're distracted with and about the task, and your brain is doing double time, while attempting to make some decisions.

It's exhausting, and if you're low on energy to begin with, systematically procrastinating can lead you directly to burnout. You start procrastinating because you've worked too much and don't have enough energy for decisive action (also known as "getting shit done"), and procrastination drains even more energy so you get even more exhausted, and feel bad because you haven't done much. It's a vicious cycle.

I'm not saying it happens to everyone, and these examples are pretty extreme, but it's useful to be aware that if you feel lazy, unmotivated and let your tasks slip, the issue is more likely the quality of your tasks, and you're its victim. When you know that the problem is not you, you can stop feeling guilty and do something.

I know these descriptions sound a lot like ADD: difficulty focusing, completing tasks and keeping up with assignments. What's the difference? I know I don't have any problems with concentration—I can read a book for ten hours straight without stopping, or when I actually get to work on something, I spend four hours coding. Unbroken concentration like this is a sign of "being in the zone," or *flow*, after Mihaly Csikszentmihalyi's work and the book with the same name. He defines flow as "a state in which people are so involved in an activity that nothing else seems to matter; the experience is so enjoyable that people will continue to do it even at great cost, for the sheer sake of doing it." That's what we are aiming for when programming.

I find that if I can get through the initial resistance when starting to work on a task and manage to start it in earnest, I get in the zone and work on it without issue. As I began writing this book, I knew that I would face strong resistance (because "write a book" is the vaguest todo item in the world), so I used every trick that I knew to help myself write every day:

- Make other people expect your work. Probably, the best thing I did was to offer the book for preorder with a set release date. I'm very serious about my promises to other people (Rick Astley starts playing). Just the thought that I've promised to do something by some day helps me sit down and do the work. If you only do something for yourself, you can easily make an excuse and avoid working, but you can't go back on your promise and make up an excuse for other people. At work, I know my teammates expect and rely on the tasks I do, so I try to never let them down. Reputation beats procrastination.

- Remove distractions, at least in the beginning. Often the bump that we need to cross to begin working on a big task is relatively small, but when there's a distraction available without friction, we choose that instead. I write on a very slow Linux computer that can't play music, has no internet, and my phone is on silent, face down, in another room. There is "fog" in my head when I sit down to write, and I sit there and stare at the screen for ten minutes. But the fog is me thinking about how to start and what I want to say in the section. I need those ten minutes of initial concentration to break through the resistance of the task. If I had a distraction, I would probably take it. What also helps is having a notebook that you keep in front of you when starting, so that you can write down what you want to do about the task first, like a small todo list. I find that after I start, I can stay in work mode until I'm done, even through small distractions.
- Have a deadline that feels a little too close. Aggressive deadlines only cause stress and burn you out because you have to spend extra energy to hit them, and for whose sake? The trick is to have a due date for the task that will make you feel like you have to start working on the task now. If a task at work is due tomorrow and takes a day to get through code review, I know I have to start working on it right now, or I will be late. When I began writing, I picked the deadline for the book a month and a half away. For a whole book, it's ridiculously short (books take six months or more to write), but if I had a year, I would never finish it. If I had a week, I wouldn't finish it too, it's simply not enough time to write fifty thousand words. But a little more than a month? It's a stretch, but doable. I'm not overexerting myself by writing for twelve hours every day (like I would have to if I had a week), but I know that with the amount of work I need to do, I don't have time to procrastinate, only to work. So here I am, writing.
- Work towards a goal for the day. I learned this from the book "Make Time" by Jake Knapp and John Zeratsky. Pick what you want to accomplish in about two–three hours and feel that you've "done good work." I often pick a task that can take the whole day, but you get the point. If you've done it, you feel like you've won the day, and everything else you do is a bonus. A large todo list can be discouraging, but being able to tick off a large, important task makes you feel incredibly productive, and you get a boost of energy that you can use to do more work. Writing this book is not the only thing I do, but if I've written what I've

> planned for the day, I feel in control and calm, because I’ve done the work that requires the most concentration.

Here we are at fifty thousand words, and I’ve been writing for more than a month every day, a sustained effort of unprecedented proportions for me when working on what is essentially a side project. If I didn’t have a table of contents that I could follow, dozens of people waiting for the result and a not-too-distant deadline, I would never have made progress so consistently and intentionally.

6 Growing as a developer

> The lyf so short, the craft so long to lerne.
> Geoffrey Chaucer

Getting the Senior Software Engineer title is good, but ultimately we aim to grow and become better software engineers. The title is there only to recognize our skills and impact in the company and comes after we've improved, not before.

Here I've collected some things that are optional but will serve you well even if you're fairly advanced, and will be an additional source of inspiration, like advice from software engineers many of us know.

I'll wrap the book with the section *Recommended reading*, the list of all the books and articles that have helped me and will help you on your journey to improvement.

6.1 Complementary knowledge

We know that there is always *more* to learn in software development, no matter how advanced you are. I couldn't have written The Book of All Software Engineering™ so in this book I've tried to distill the advice and knowledge that you'll find the most useful in the first years of your software engineering career. But there's *more* and I have more to say about things that are not necessary, but are the subject of questions and you're better off learning about them earlier rather than later.

Fun fact: we have already met SWECOM or "The Software Engineering Competency Model," but did you know there is also SWEBOK—"The Software Engineering Body of Knowledge"—which is a standard produced by IEEE with a 300-page "guide" that aims to describe *everything* in the field of software engineering? It's freely available and you are welcome to peruse it for your own pleasure, or if you have trouble sleeping.

6.1.1 “Elegant” code

You may have seen pieces of code referred to as “elegant” or someone’s code called “elegant.” People might have complimented *you* on an “elegant solution.” Software engineers seem to intuitively know when code is elegant and when it’s not, and great software engineers are praised for the elegant code they routinely produce when working on their tasks.

What does it mean for code to be elegant? Elegance is beauty that shows unusual effectiveness and simplicity. The natural, economical grace of animal movement is elegant. An elegant dress is simple, without frills, and fits the wearer perfectly. An elegant mathematical proof has few simple notions that fit perfectly together, and is easy to follow. Elegant code, then, is code that surprises us with its simplicity and effectiveness by being unusually easy to follow. We feel that elegant code is beautiful, because it logically builds upon itself in a way that we can immediately grasp, all at just the right level of abstraction, with every snag and bump removed. If code is writing, then elegant code is poetry.

I consider surprise to be an essential quality of elegant code. It makes a leap of logic that we haven’t expected, but in our mind it’s like a puzzle fitting together. We learn something new and appreciate the author’s effort and skill.

Albert Einstein said: “Everything should be made as simple as possible, but no simpler.” The final part directly relates to elegant code. It uses the level of abstraction that fits the task at hand. Simplicity alone is not enough. Brainfuck is simple, but it’s not elegant because code is impossible to follow and everything is tedious to do. Elegant code uses language constructs and approaches that are not “as simple as possible” to be more effective and expressive, as long as they’re used in a straightforward way, and not in the name of terseness. I think we can all agree that terse code is hard to read and follow. One-liners are cool, but we’re not at IOCCC—elegant code is maintainable first, clever second.

How do we learn to write code that is elegant, and why would we want to? If you’re trying to write simple, maintainable code, congratulations, you’re halfway there. Making code elegant is often thinking a little more about the problem and finding an even simpler way of solving it. You can learn to do that by tinkering with small problems like those found on programming websites, and by reading code written by other software engineers that you consider elegant. It's mostly a matter of training your intuition.

By trying to find a simpler solution, we gain a deeper understanding of the problem we're working on (and might notice some bugs in our code while doing that). By trying to write elegant code, we directly improve our development skills. All programming can be reduced to solving small problems at different levels of abstraction, and training yourself to solve small problems elegantly and efficiently is a transferable skill that will serve you well in your career.

6.1.2 Paradigms and programming languages

There is nothing wrong in knowing only one programming language. Popular languages like JavaScript and Python are "general-purpose" and "multi-paradigm" for a reason—you can do almost anything you want, in very different styles, either through language constructs or with the help of libraries. As you keep working in the industry, you will naturally pick up at least the basics of a few more languages that may or may not be similar to those you already know.

I suggest that you try out a few programming languages that are impractical in your work and are different from what you are usually using. You don't have to spend much time, a few days is enough to learn the core concepts and see if you want to continue (if it's useful).

Why would you do that? We write code by arranging abstractions in our head first, and only then committing them to our editor. The programming language you use heavily influences *how* you approach solving problems by favoring one style over others, and certain things being easier to use than others.

Learning a new programming language that is fundamentally different from the one you are using can result in a fundamental change in your thinking because it will alter the basic logical concepts that you tend to use. By learning more languages, you'll have a stronger mental framework for approaches to problems, and even bring concepts from one language to another if that makes solving problems easier. It can also *not* alter your thinking as we know from the popular phrase "You can write Java in any language." Please take the time to learn how to write your new language idiomatically.

For example, Apple's Swift is not a functional-first language, but it's flexible enough to be extended. Developers in the community who are familiar with purely functional languages (like Haskell) and their libraries have created libraries that bring some operators and functional programming approaches to Swift. These libraries gained some popularity and it has made the Swift ecosystem better and richer. Learning another programming language does the same for your mind.

Where do you start? I think it's useful to be familiar—which means, knowing how syntax works and being able to read and maybe edit the code a little—with just one programming language from several big "classes." This will give you the most benefit for the least amount of time and effort, and if you like the language, then you can explore that class more.

So, my advice is to learn:

- a C-style procedural language. You probably already know one since they're so widespread. JavaScript, Java, C or C++, C# are the most popular ones. You'll come across a lot of random code online written in a C-style language, so it's good to know at least one.
- a functional language. Haskell is your best choice here and perhaps Elixir. The functional approach to problems is very different from the "default" procedural style we're likely used to.
- a Lisp. (I hope you like parentheses.) Lisps are programming languages based on lists, and this choice gives them many interesting properties. Like functional languages, they will do interesting things to your brain and you will discover some new unexpected ways to think about problems. If my goal was learning, I would pick Racket or Common Lisp, primarily because there are some great books available for both.
- an assembly language. When running your program in a debugger, your crash is often in system code and you only see the *disassembly* because you don't have the source for the system libraries. Knowing a bit of assembler is useful to have an idea what's going on, and sometimes see if your code is compiled effectively (if you're working in a compiled language). I suggest starting with the assembly for the platform where you're running your code (likely x86 or ARM) to know the common registers, but you can have much more fun doing it on a GameBoy emulator (it's an ARM) or the NES (it's an 8-bit 6502 CPU). Recently there has been a resurgence of making games for old platforms directly in assembly, so you can

find many howto articles.

There are a few others that are worth looking into because they either have an interesting philosophy (and hence interesting design decisions) or can teach you another way of thinking about types, programming style and other things. These are (in no particular order) Python, OCaml, Go, Rust, Erlang, Forth, Lua, Ruby and SQL (because it's declarative *and* useful and I'm not going to suggest Prolog).

Again, by "learning" I don't mean being able to write code. I "know" JavaScript and Python enough to read and understand pieces of code when I come across them, but I wouldn't be able to write anything because I don't remember the details of the syntax—and once upon a time I've written software for a living in both JavaScript and Python. Don't think you have to practice a dozen languages (unless you want to), simply being able to read code is enough.

6.1.3 "Not invented here" syndrome

The term "not invented here", often abbreviated as NIH, is the tendency of corporations or institutions to resist ideas from outside. In software development, it's defaulting to in-house development and the belief that it's better and more suited to internal requirements than some third-party solution. Unsurprisingly, this meaning is negative. You don't want to reimplement every library that you need when there are existing ones that do exactly what you want. There are valid reasons for a corporation to do that, but we won't get into those.

What I want to propose is that sometimes, and more often than you think, reimplementing a part of a big third-party library that does only what you need the way you want it is actually a viable solution. We did that all the time in the companies I worked for, because in mobile, external dependencies can be costly (in various ways) and not customizable enough. Some are also small enough that you can write your own in a week.

Writing your own involves some maintenance costs and time up front, but in return you get much more flexibility and it's *your* code you understand and control. A third-party library can try to please everyone and deprioritize or ignore your main use case, but yours only has one customer: you. So, if it's you who is doing the implementation and the amount of work is reasonable, you'll learn a lot and your company will be better off using a library that's perfectly suited for the company's needs.

Out of the context of work, this is an excellent learning exercise because it lets you peek under the hood of libraries and functions that you have considered fixtures in your industry. You learn how something works *and* understand the existing library better. This is very similar to an exercise all students do at the university when learning programming: reimplementing functions and containers from the standard library in order to understand how they work and learn the language better.

Pick something you're interested in or want to learn more about, either at work or personally, and write an alternative implementation to an existing library or function. A lot of open-source libraries are born this way, they do the same thing but in a different way, or pick different tradeoffs than the library they've originally copied.

For example, I needed to have a particular text layout in my language-learning game, but the library I used didn't offer that. I went down a level and by using a font rendering library created an alternative text layout function that did exactly what I needed, and which I completely controlled. I found useful options in the rendering library (that were unavailable through the existing API) and learned a lot about font rendering in general and the many tradeoffs.

This practice of going down a level and making an alternative implementation always stretches your skills as a developer because you're in unfamiliar territory. At the same time, it is not "new" functionality, and if your implementation doesn't work out, you can always fall back to the existing third-party library. If it does work out, you can drop an external dependency and be happy. In any case, I guarantee that you'll learn something new. I think that's "not invented here" done right.

6.1.4 Becoming an early adopter

It's a running joke in software that when a new technology or programming language comes out, companies immediately publish job descriptions looking for people with 5 years experience in it. Nevertheless, there's a grain of truth here. "Years of experience" is another term for "expert," and these companies are looking for people who have already tried and tested the new tech and have *some* experience with it.

When a new technology becomes available that is likely to become popular, you can quickly jump a grade by evaluating and learning it, building something and becoming an expert. It's hard to become an expert *quickly* in mature, established technologies like the JVM, or become a C++ expert simply because of the amount of knowledge accumulated over the years. There are people around who have *decades* of good experience and will run circles around you for years while you're catching up.

But with a new technology or programming language, everyone starts on square zero (Lua programmers insist on calling it square one). By starting to learn it early, you are one of the frontrunners—an early adopter—and because there is so little information available, you can learn *everything* there is to know about the new tech. And when in a couple of years the technology goes mainstream and companies start looking for people with "5 years experience" (despite the tech being only 2 years old), you'll be there among the very few people who have real-world experience. The knowledge you have gained is scarce and thus valuable.

When Apple's Swift just came out in the summer of 2014, I saw the potential and immediately started learning it (like all iOS developers, I was writing Objective-C at the time). By the end of 2014, I wrote a commercial app for a client in Swift. Early in 2015 I was able to secure a job writing Swift full-time because I was one of the few people who had real-world experience and I was already an "expert" (being an expert is simply knowing more than other people). After a couple of years, *everyone* on iOS was writing Swift.

6.1.5 Mentorship

"Find a mentor!" is another piece of clueless advice for software developers who are just starting out that you can find on the internet. A mentor is touted to be a famous person who will be our friend and teacher for life. (Epic music starts playing.) Like Gandalf leading Frodo, our mentor will lead us on the path to enlightenment.

No, no, no and no.

Again, we aren't at school. A mentor is not a teacher who tells us what to do, so we don't even have to think. If we want to improve and grow, *we* must make decisions, make mistakes and learn from them. A mentor is someone who is further along on the path of growth that we have chosen, and I consider the role of a mentor not to tell us what to do, but to save us from making *expensive* mistakes that can set us back (because small mistakes teach us and help us move forward) and deliver a swift kick when we're stuck.

At first, this can be a purely professional relationship with one (or more) of our peers at work. More, because we aren't limited to one "mentor," there isn't a slot we fill and that's it. We can ask anyone for advice who is in the position to give it—a better software engineer than we are. Remember that it's us choosing our path. As we grow, these relationships will develop primarily because we'll be able to help our peers back. The trust we build with people who are helping us and whom we are helping can become a friendship.

Ideally, a mentor is someone who cares enough about us that they're willing to help. A friend first, and a teacher second. If you have some friends who are doing what you want to do, and are better at it than you, you can start asking them for advice. If they are good friends, they'll appreciate that you are seeking their opinion, and will appreciate it even more if you follow through.

When I was just starting to learn programming, I asked an acquaintance for advice now and again, and reported back my progress. He was a very experienced developer, specializing in the field I was trying to get into, and I looked up to him. When I made some progress, I was able to help him out too. Over time as we talked, our relationship grew into a friendship. Now I can't say that he is my mentor, we are simply talking and helping each other as much as we can (and exchanging memes).

What if you really want a famous software engineer to help you (if that tickles your fancy)? I remember a situation from some fantasy book I've read where a wise old wizard tells the young hero: "You're still too inexperienced, it's too boring for me to teach you." Consider why a well-known person, who is very busy, would spend their time on you—a stranger? Don't expect a reply to your email saying "Please be my mentor!"

Thankfully, a lot of great programmers who are leaders in their fields are active on Twitter or reply to emails. Experts like to share what they know. So, you *can* talk to them and if you're lucky, they will help you.

Remember what we've discussed earlier in the book about asking for help. When trying to get help from a busy person, you must be even more demanding to yourself:

- Don't ask a question about yourself. "What programming language should *I* learn?" The busy person will reasonably answer: "I don't know, you decide."
- Don't ask a question that you can answer yourself (even if it's difficult). The expert isn't there to do research for you because you're lazy.
- Don't ask a question that is ambiguous or has an answer that won't help you. "Do you think technology X or Y is better?" is too vague because it lacks details. If the expert says Y, what would you do? You'll try to use Y only to find out it doesn't work for your use case because you haven't cared to mention it in your question.
- Don't ask many questions at once. You're not doing an interview!

Instead, ask *one* very specific question that is directly related to the expertise and experience of the famous person and seeks *their* opinion that will make a difference (or what *they* would do). Give enough context and detail, and show you have done the work before writing. And keep it short, nobody has time to read five-page emails. For example, your message to a famous compiler developer (working on language X implemented over LLVM) might be:

> Hi! Your work on programming language X has inspired me to get into compilers. I've gone through the Dragon Book and LLVM Cookbook and implemented a C-like language with reference counting (link). I also made a small contribution to LLVM (link). What would you suggest I do next if eventually I want to work on a mature language with an LLVM backend? I consider starting to contribute to X. Thanks!

If they reply, thank them for the advice and promise that you'll report back after a few weeks (or months) after you've had time to follow their suggestions. Now the most important thing is to follow through and actually write back with your results (and what you've learned) and thank them again.

After some time, you are likely to have another good question that you can ask the same expert, so do it. Again, follow through. By doing that, you are showing that you are worth helping, because you do your homework and don't waste their time. I assure you they will appreciate that not only you asked a good question, but also followed their advice. Perhaps after a while that person will ask *you* for help with something, because you have *proven* that you do the work that you promise and they can trust you.

That's how you build a relationship based on trust and mutual help. I want to specifically empathize trust. Trust comes from honesty, and if we base the relationship on deceit, not only we don't deserve trust, we will hurt the other person. It's OK to want something from the expert (that's why we're writing), but please be respectful and ethical about it. Don't pretend to be their friend when you aren't, or claim to like their work when you don't.

6.2 Advice from notable engineers

What do some well-known software engineers say about *improving* as a software engineer and what does it mean for them to be a good software engineer?

This is certainly a non-exhaustive list, and it's not in a particular order. If you have a favorite quote that inspires you to become better, send it to me and I'll include it in the next edition of this book.

Let's see what they think and get some inspiration for our own work.

John Carmack (whose clever *Doom* and *Quake* code inspired thousands) said in a tweet:

> Write lots of code. Clone existing things as exercises. Learn deeply. Alternate trying yourself and reading literature. Be obsessive.

Larry Wall (the creator of Perl) wrote in the first "Programming Perl" book:

> The three chief virtues of a programmer are: Laziness, Impatience and Hubris.

Alan Perlis (the creator of Algol) said in his "Epigrams on Programming":

> You think you know when you learn, are more sure when you can write, even more when you can teach, but certain when you can program.

Quoting Joshua Bloch (Chief Java Architect at Google at the time) from the book "Coders at Work":

> …intelligence is not a scalar quantity; it's a vector quantity. And if you lack empathy or emotional intelligence, then you shouldn't be designing APIs or GUIs or languages. What we're doing is an aesthetic pursuit. It involves craftsmanship as well as mathematics and it involves people skills and prose skills—all of these things that we don't necessarily think of as engineering but without which I don't think you'll ever be a really good engineer.

Donald Knuth said in his 1974 lecture "Computer Programming as an Art":

> We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better.

Bjarne Stroustrup (the creator of C++) wrote in the introduction to "The C++ Programming Language":

> The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones. For this, an appreciation of programming and design techniques is far more important than an understanding of details; that understanding comes with time and practice.

Richard Stallman said in his 2011 lecture in Paris:

> To learn to be a good programmer, you'll need to recognize that certain ways of writing code, even if they make sense to you and they are correct, they're not good because other people will have trouble understanding them. Good code is clear code that others will have an easy time working on when they need to make further changes. … How do you learn to write good clear code? You do it by reading lots of code, and writing lots of code.

This selection may be (a little (ahem)) biased, but the common thread is that we become better at programming primarily by *doing* programming and striving for excellence and beauty. Building software out of abstractions is like building a sand castle, and we'll only have a brief moment to admire what we've built before it has to be changed. But we've learned, and next time we will do it even better.

6.3 Recommended reading

The amount of books and articles on programming and software development, even if we exclude the ones about specific programming languages, libraries and frameworks, is incredible. Software engineers, it seems, spend a lot of their time writing. But when do they code?

Anyway, I could have made a huge list of every book that I think you may find useful, but instead I've decided to put here only the books (and blog posts) that I have read and can recommend as practical, that is, likely to inspire a change in how you program and work. They aren't only about software development, but also learning, productivity (in a wide sense), interviews and writing—the complementary knowledge and skills we need to advance our careers.

What's *not* on this list are programming language books, because we all use different programming languages, and there isn't one that everyone should learn. At the same time, I find books about programming languages, especially written by their creators (like Kernighan and Ritchie's books on C, Bjarne Stroustrup's on C++ and so on), enjoyable to read since the authors not only try to teach you the idiomatic way to use a language, but also impart their philosophy of building good software, and that's always interesting to read.

First, let's get the two books out of the way that will get you up to speed on the discipline of programming, algorithms and data structures, or what a software engineer is "supposed" to know. If you're self-taught, these give you the minimum to close the biggest gaps in your knowledge. We're not aiming to replicate a Computer Science degree, but to revisit or learn the fundamentals. Again, you don't *need* to read these or do the exercises, but both are fun and accessible if you're diligent, and you can learn a lot even if you're more experienced.

- "Structure and Interpretation of Computer Programs" by Harold Abelson and Gerald Jay Sussman with Julie Sussman. Widely known simply as "SICP" and praised by many developers, this is an introductory Computer Science course at MIT turned into a book. It uses a dialect of Lisp for instruction (remember how useful it is to learn a language different from what you know?) and moves assuredly and swiftly through some advanced concepts. It is also free online on the MIT Press website.

- “The Algorithm Design Manual” by Steven S. Skiena. It covers everything that you’d expect, but compared to more academic texts like CLRS or Knuth, I find it much more readable and clear. There are short summaries, “war stories” with real-world examples of approaches (and mistakes), and a handy guide for picking a set of algorithms you need to solve your task. If you can’t buy the book, Skiena’s lectures are online for free on YouTube. The links are on his website.

In the tools section, we only have one about Git. Version control, and Git in particular, is probably the only tool useful in any industry and field you might be working in.

- “Pro Git” by Scott Chacon and Ben Straub. Don’t get discouraged by the “Pro” in the title, it starts off *very* gently but finally takes you deep into the workings of Git. I can honestly say that I’ve read only about a half, but I’ve learned enough to understand what happens under the hood. I use it occasionally for reference since it’s clear, unlike some `man` pages. The book is free online on the Git website.

The next four are all about the practice of constructing software and writing code that we can consider “beautiful,” as told by the experts in their own words. We know that discovering how experts think is one of the best shortcuts to becoming an expert yourself.

- “Code Complete: A Practical Handbook of Software Construction” by Steve McConnell. The book takes a complete circle through all aspects of software development, focusing specifically on construction and finishing with the qualities and techniques of a good programmer.
- “The Art of Readable Code” by Dustin Boswell and Trevor Foucher. While this one may not be well-known, the authors made an outstanding job of creating a book that’s both useful and entertaining. Experienced programmers may find the tips and techniques too obvious, but we all have to start somewhere, and the book clearly lays out exactly what makes code readable with many examples. It’s fairly short at only 200 pages.
- “Beautiful Code: Leading Programmers Explain How They Think” by Andy Oram and Greg Wilson. This book is a collection of essays by some big names in programming (Brian Kernighan, Douglas Crockford and Simon Peyton Jones among others) talking about their code and the decisions behind the architectures of

their projects. It's a mixed bag in terms of quality, so skip the ones you don't like. But overall, the book is unique in allowing us to peek at how accomplished, expert programmers think about the problems they solve with code.
- "Coders at Work: Reflections on the Craft of Programming" by Peter Seibel. This one is similar to "Beautiful Code," but the author interviews programmers. From the description: "Peter Seibel focuses on how his interviewees tackle the day–to–day work of programming, while revealing much more, like how they became great programmers, how they recognize programming talent in others, and what kinds of problems they find most interesting." The interviews are more about the ways of working and lessons learned, rather than explaining a piece of code.

The next two entries are about *finding* work rather than *doing* it.

- "Interview Intervention: Communication That Gets You Hired" by Andrew LaCivita. This is an essential book for learning how to ace interviews and work on your soft skills.
- Patrick McKenzie does a great job explaining the business side of the software development career in his blog post "Don't Call Yourself a Programmer, and Other Career Advice."

With so much we have to learn to improve, how do we find enough hours in the day to do our work, study and still have time to rest? The three books that follow each expose a different angle of the problem, and each lets you pick and choose the techniques that are most relevant to how you work.

- "Deep Work: Rules for Focused Success in a Distracted World" by Cal Newport. The author makes an argument that in the modern world and going forward, the skill to focus and work for extended periods of time without distraction will become more and more valuable. It's not a gimmick, but a practical system of techniques that lets you focus on the work that matters to you, day after day. If you do that, you'll discover you can achieve a lot more than you've thought possible, and to have more free time.
- "Make Time: How to Focus on What Matters Every Day" by Jake Knapp and John Zeratsky. My manager recommended this book to me after I complained I was too distracted by Slack. Now I recommend it to you. Compared to "Deep Work," it's less dogmatic and is a grab bag of techniques that you can fit around your lifestyle

to make progress on the project that you consider important, at or outside of work.

- “The First 20 Hours: How to Learn Anything… Fast” by Josh Kaufman. This book presents a set of techniques to jumpstart learning a new skill and get to “good enough” quicker—we do that a lot in software development. From this book I’ve also learned the concept of disliking something because you’re simply bad at it, and getting a taste once you’ve improved.

We have established that besides writing code, we’ll also write a lot of text: messages, emails, presentations, blog posts. The quality of your writing will tip the other people’s opinions in your favor if it’s good.

- “On Writing Well: The Classic Guide to Writing Nonfiction” by William Zinsser is my favorite book on writing that I must have read at least five times, and have often referred to for specific advice. I think you’ll love it.

For desert, a couple of blog posts by Joel Spolsky, the co-founder of Fog Creek and the former CEO of Stack Overflow.

- In “The Duct Tape Programmer” he describes a type of developer who is different from what you may be used to, and is more focused on shipping than Making Everything Right™. Software engineers often forget their role is not to write code and make it more and more byzantine.
- The post “Making wrong code look wrong” shows an interesting technique, but it also raises the important topic of code locality and that we are humans, and our cognitive abilities are severely limited, so we must strive to remove any extra complexity that may slip through our fingers and cause a bug because we haven’t noticed something.

And one more thing that I almost forgot.

- “The Pragmatic Programmer: From Journeyman to Master” by Andy Hunt and Dave Thomas. This book is very similar to “Junior to Senior” in the sense that it’s a “non-technical tech book” as someone called it in their review. Experienced programmers will know most of the material, but its popularity is proof to how useful it is for people with less experience.