A Student's Guide to CS

A mini book on computer science and software engineering and the things that make both fields beyond (and including) the code. In this narrative guide, Ethan, a senior student in CS, explains his experience in computer science and software engineering and what makes great people, great communicators, and great software engineers in the field.

The full book can be downloaded as a PDF here.

Version: December 2025 Beta

Foreword

Hi, my name is Ethan Richards, and at the time of writing (late 2024), I'm a senior in Computer Science at the Colorado School of Mines. I'm working on finishing up my undergraduate degree (graduating May 2025), but I am also pursuing my masters and will graduate with that degree in December 2025.

I'll make this foreword brief and answer the questions that you're probably asking: *What gives me the qualifications to write a book about computer science and software engineering? What is even the point of this book?*

I think I'm a fairly well respected figure, at least in the CS@Mines circles. My college career has had a special emphasis on CS education; I helped design our introductory CS course, CSCI128, helped with our Systems Programming course, CSCI210, and got my research published at SIGCSE 2024. My masters project is designing a UI/UX course for the department. I've been a lead TA for 128 since its inception. I was also president of Mines ACM for a year and I had various involvement with other roles for about 3 years. I also helped organize the Mines High School Programming Competition for two years.

Outside of academia, I started programming when I was in middle school, fascinated by how I could use Java to modify Minecraft. I have had one internship (and another for my dad's company, if you

want to count that), with another lined up for Summer 2025.

Of course, all of this does not make me an expert by any means, and I still have a lot to learn and experience (and I hope that's the most you ever hear me brag). The purpose of this text is to share that helped me along my journey in CS so far and what helped me succeed up to this point in industry and academia.

This is also a great reflective process for me now that I'm wrapping up college; I want to reflect on the great people and lessons that I learned. I'll talk about my understanding of things in the realm of computer science and software engineering, understanding the people in the field, and how we can all succeed both individually and together. Enjoy!

The full book can be downloaded as a PDF here.

What is computer science?

Computer science, as I learned it in my CS1 course, is "the study of algorithms." A Google dictionary definition says "the study of the principles and use of computers." Wikipedia says "Computer science is the study of computation, information, and automation."

While these aren't necessarily incorrect definitions, I think they miss out on part of the story. Computer science, to me, is the blend of computational theory, practical software and hardware engineering, and problem solving. Computer science is as much of an art as it is a science; the science comes from the concrete theory, mathematics, etc., whereas the art comes from the people and the problem solving. To me, computer science is all about being handed any problem and being able to solve it or put a strong effort towards it.

There's no precise formula, mathematical proof, or scary AI replacement for problem solving (thank goodness, or I would be out of a job) nor is there one for working with real people and real teams. Nor is there one for making good decisions [1]. Real people and real teams do these things.

What is software engineering?

A fan-favorite programming site, Geeks4Geeks, says: "Software Engineering is the process of designing, developing, testing, and maintaining software." This looks suspiciously close to the Wikipedia definition, but I think we're all in agreement here.

Software engineering isn't *just* sitting and writing code for eye-straining amounts of time; it's about designing things that solve problems and serve real people's needs. It's about testing to ensure safety in critical operations, and designing the software with longevity in mind.

What's the difference?

Once upon a time, I thought there was no difference. I was quickly corrected. Software engineering is a subset of computer science, but as mentioned, it's not **all** of computer science. A lot of professors and important people in higher education will quickly argue that a degree in computer science is far better than an online bootcamp or course focused on *just* coding. If you go out into industry, you will see and hear about many people who dropped out or had online training and are doing just fine. I have nothing against online courses and have done a few myself, but I can see the academic argument too. Understanding the full picture of computer science will help you be a better software engineer, which is the main job outcome from a computer science degree [2].

What *isn't* software engineering?

I might be too much of an idealist, but I don't think software engineering is about the paycheck. It shouldn't be. I'm always pretty grossed out and disappointed by the people looking at levels.fyi comparisons of which company is going to give them $900k a year and work them into the ground; or the relatively newfound ghost software engineers, who work multiple remote jobs and get paid more for doing less [3].

Yes, the paycheck is nice. Yes, it can positively change your quality of life. But if you're doing it only for that reason, I think you will be left disappointed and frustrated.

Software engineering **is** getting frustrated at a problem but working diligently for hours, days, weeks, months, and years to fix it. It's hours of debugging, only for the sweet relief of a fix.

I'm not saying I enjoy being frustrated for hours and days on end, but I recognize the necessity of that process, and seeing the fix or the final product makes it all worth it for me. If you're not on board with that contract, it's going to be a rough ride.

---

[1] The pessimists among us finished this thought with: ...yet. Yes, AI+ML exists, and it's good. But we needn't worry about that day until it arrives, *if it arrives at all*.

[2] Not saying you have to go into software engineering. A CS degree opens up many other great opportunities. Additionally, even if you come from an online bootcamp or course, knowing how to code is the main thing you need to learn. As long as you keep an open mind, you're going to do great!

[3] This is a relatively new and disputed theory from Stanford, but I am aware of people who have already done this at the internship level in college.

The building blocks

Remember when you learned your ABCs as a kid? When you learned how to write and how to read? Okay, probably not, but there are similar foundations for working in CS. Before or while you learn a lot of other things, you should have these basics down. They will *really* help in the long run!

Your Computer

A computer scientist needs a computer? What? It's crazy, but I know a ton of computer science majors who are either really frustrated with their current computer (whether laptop or desktop) situation. I don't want to come across as insensitive to different financial situations as laptops are *not* cheap, but once you feel financially comfortable, it wouldn't be a bad idea to set yourself up for success. After all, you're going to spend a lot, like, a *lot* of time with your computer; you might as well enjoy it!

I am always encouraging Mac over Windows, especially because you get that builtin Unix terminal so you don't have to deal with PowerShell and other general Windows shenanigans.

Regardless of which operating system floats your boat, you don't need to worry about hardware too much. It's not all about RAM - coding takes minimal amounts [1] of memory. Space is cheap. If you really have money to burn, it's nice, but not necessary. Maybe invest in a nice monitor for your eyes or chair for your back.

Typing + Keybinds

It sounds silly, but people often underestimate how typing can help productivity. Learning to type at a reasonable pace (~60 WPM is a good, average starter) is a great idea when being a computer

scientist. Your keyboard is where you're going to work, like, all of the time.

You don't have to have a fancy $200 keyboard with decked out keycaps or even a duck keyboard [2] in order to type fast. Just get quick with typing on whatever keyboard you have and it will translate between any other keyboards you use.

Keybinds and shortcuts are also important. The Visual Studio Code or IDE-specific ones. Or you can even figure out those vim shortcuts. Or you can just make your own. Either way, they make you so much quicker if you just invest a bit of time to learn. They are nice to quickly move through many files you're editing - and you might even impress a few people at work!

If I really tried, I could probably not use a mouse for a few hours while coding. I know people who intentionally do that (this is not farfetched for computer scientists whatsoever), and do it well.

Git + The Terminal

I've helped undergraduate & graduate students, professionals at jobs, and maybe even a couple of professors with Git before. I can't stress enough how vital this skill is to have. You can learn any programming language and people will be patient with you, but if you roll up to a job not knowing Git either, I think that might be frustrating to your coworkers.

I won't get into the specifics of Git and GitHub, but I will talk about some good practices later. If you're looking for a tutorial, there are probably millions online. My friend hosted a great workshop on it a few years ago too.

On a similar note, the terminal is great to know and use. Know your file system and know how to use Git over the command line. While GitHub Desktop does the same thing, it has less capabilities for the complex stuff, and hides the details from you. Using the terminal tracks with what has been mentioned so far too. The terminal is great for quick coding if you're into vim, and you can even run some sort of window manager.

More than that, the terminal is where you will run code. That should be enough of a reason in itself! You are bound to see binaries, applications and other repositories that you might need to run. The terminal is a vital building block to know.

The internet is your oyster

The internet is there to help as your right-hand man of many answers. Whether it's StackOverflow, Google, ChatGPT, GitHub, Geeks4Geeks, language documentation or otherwise, if you can admit you need help and go and find some, it goes a long way. There is a high chance that *at least one* other person on the internet has had the same problem as you. In the uncommon scenario when that's not true, don't be afraid to ask for help.

When I was first learning to code, I felt bad or dishonest looking things up as it felt like cheating and I knew it would harm my learning. There's a balance to be found here. Give whatever problem you're working on an honest attempt *first*, and then if you're still stuck after trying a couple of things and (hopefully) asking a coworker or friend, hop on over to the internet.

It's okay to be resourceful!

---

[1] Assuming you're not doing some memory intensive work with, for instance, Docker containers or game development.

[2] I have a keyboard with about a hundred ducks over all of the keycaps. Yes, really.

A great computer scientist

What makes a computer scientist "good" or "bad" is pretty subjective, and my goal here is not to say anything bad about people. Instead, I want to say what makes computer scientists *great* - beyond just average. What are the traits of the people making startups, doing fascinating research, or simply even those people who are steadfast and enjoyable to work with. The following is a (not comprehensive) guideline that hopefully answers some of these questions.

This section is here to say why we study these different areas and how we can strive to have more a more holistic understanding of the field. We're not putting anybody down with this. In fact, I encourage you to think of the parts you're better at. I was never good at the theory.

The Practical

This is the code part. Can you write code? Okay, that sounds simple, but in software engineering terms, this is obviously really important. People don't want to hire people who can't actually code, which is why coding interviews (unfortunately [1]) exist.

I've spoken to senior engineers and directors from big companies, and they seem to agree. What elevates a developer (especially in terms of junior/senior roles) is their ability to notice patterns within code and make refactors that make the project better. Better could mean saving people time working on it, helping its longevity, or even improving its architecture.

More abstractly, can you take any problem and implement it? This could look like a backend server, or this could look like a beautiful website. Either way, you're that person who doesn't back down from a problem and quite simply, implements it and implements it well! You factor in maintainability, readability, and clarity to all of your code and designs.

The Theory

The theory part: hear me out, even though I'll hopefully never have to whip up a mathematical proof on the job site, I think the theory allows you to do three things: communicate, teach, and leverage power.

Communicating: This is the most important part in my opinion. My research advisor (who I would definitely classify as a *great* computer scientist) was great at this; he would pull out seemingly random analogies from formal language theory, the theory of computation, operating systems, or other broad CS-things. This helped us develop our research so much. Often times, abstract CS ideas could relate well to each other. (e.g. we used formal language theory grammars to help think about how we define a taxonomy for concepts in a curriculum.)

Teaching: Even if you're never going to touch a proof for the rest of your life, knowing more about systems and things as a whole isn't going to do you any harm. I think it will do a lot of good; on the same lines of communicating, it will help you communicate and teach more complex ideas to others.

The power: The AI-boom wouldn't have happened without the theory and the math. Doing cooler and more complex things often needs a slightly deeper understanding than just implementation details of a programming language; this can also be a great reason to know the theory.

The Person

I saved the best for last: the person could be the most important of all. Even if you know the practical stuff and the theory, you could be a pain to work with. Being a great communicator, friend, and coworker can go a long, long way. I don't want to stereotype a group I'm definitely apart of, but often, CS people can be thought of as "awkward" or not really personable. The good news: it just takes practice! The more interesting things you do aside from sitting at your computer staring at lines of code, the more you can share with others and the more personable you become [2].

So, be a human. Nobody expects you to be a workaholic (assuming you work at a company that respects your health) and just be a code-producing machine. The overall point I'm making: talk with other people! A great computer scientist knows when to switch off and talk about their life and ask people about theirs.

I'll have an entire section on mindset and perspectives a little later, but I think it's also worth mentioning that good computer scientists in my eyes are open-minded and perseverant. How else are we going to keep up with the ever-growing list of programming languages, frameworks, and many changes in the field?

---

[1] I say unfortunately because I think coding interviews aren't always the solution here, and are usually overkill for making sure people can code in a basic manner. Some more thoughts on this are in the *Interviews* section.

[2] A gross oversimplification, but my point is to encourage people in CS to think about other people and things outside of your work!

Planning

This might seem like the most obvious section of this entire book. But I think you would be surprised at just how *little* people actually plan what they're doing. I think these habits often come from academia: for the most part, you get an assignment, you get an idea of how to code it, and

you just go ahead and code it. Or, even if you want to plan, time constraints can make it feel like it's not worth your time.

Part of the issue which I noticed in industry is Agile. Agile has its strengths, but I feel like a good old fashioned planning session or design document is lost with Agile; it's all about that next ticket and how many points it's worth [1] without considering the bigger picture [2]. Time spent planning is time well spent, even when churning out features and tickets is the priority.

Drawing it out

Whenever I'm faced with a problem, I give it an honest attempt and then immediately *after* I draw it out or write out what I'm trying to do if it doesn't work. Yes, I am guilty of not planning up front either.

I've noticed this quality in a lot of high-functioning professors, PhD students, engineers, and more. They're *always* drawing things out and trying to understand things in their head. You don't have to be a good artist to do this. The best sketches come in the form of scratchy, messy, incoherent writings on whatever napkin or paper is nearby. Or, if your school or company has a ton of whiteboards nearby, use them!

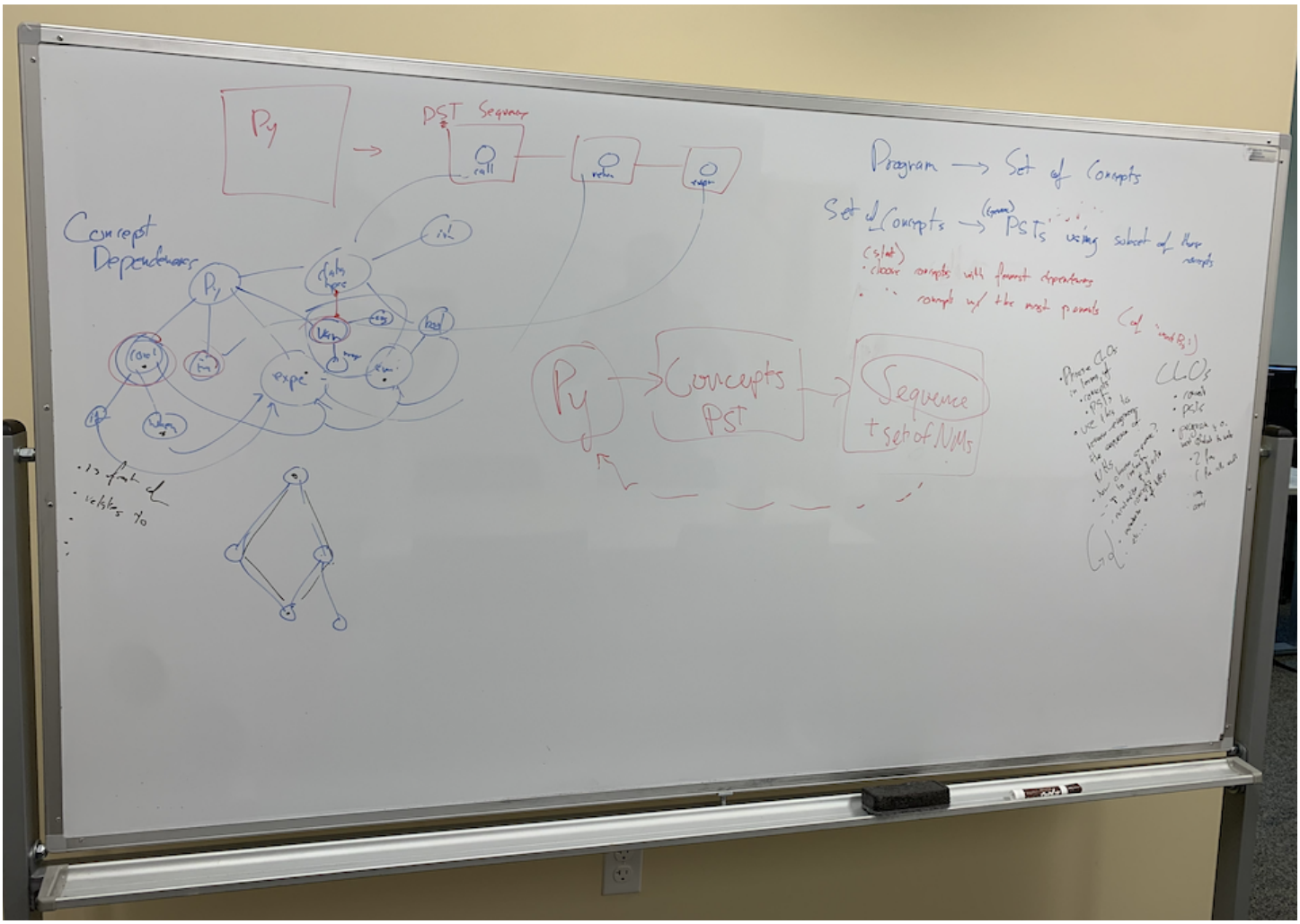
Writing things out can be in the moment. It can be ahead of a big project. It can be a reflective process after-the-fact. Regardless, it helps to get out of our heads once in awhile.

As an aside, I certainly hope you're drawing it out if you're in the realm of frontend/UI/UX design. Open up Figma or just go "old fashioned" with some pen and paper. You should plan for how things are actually going to be used and test it a lot.

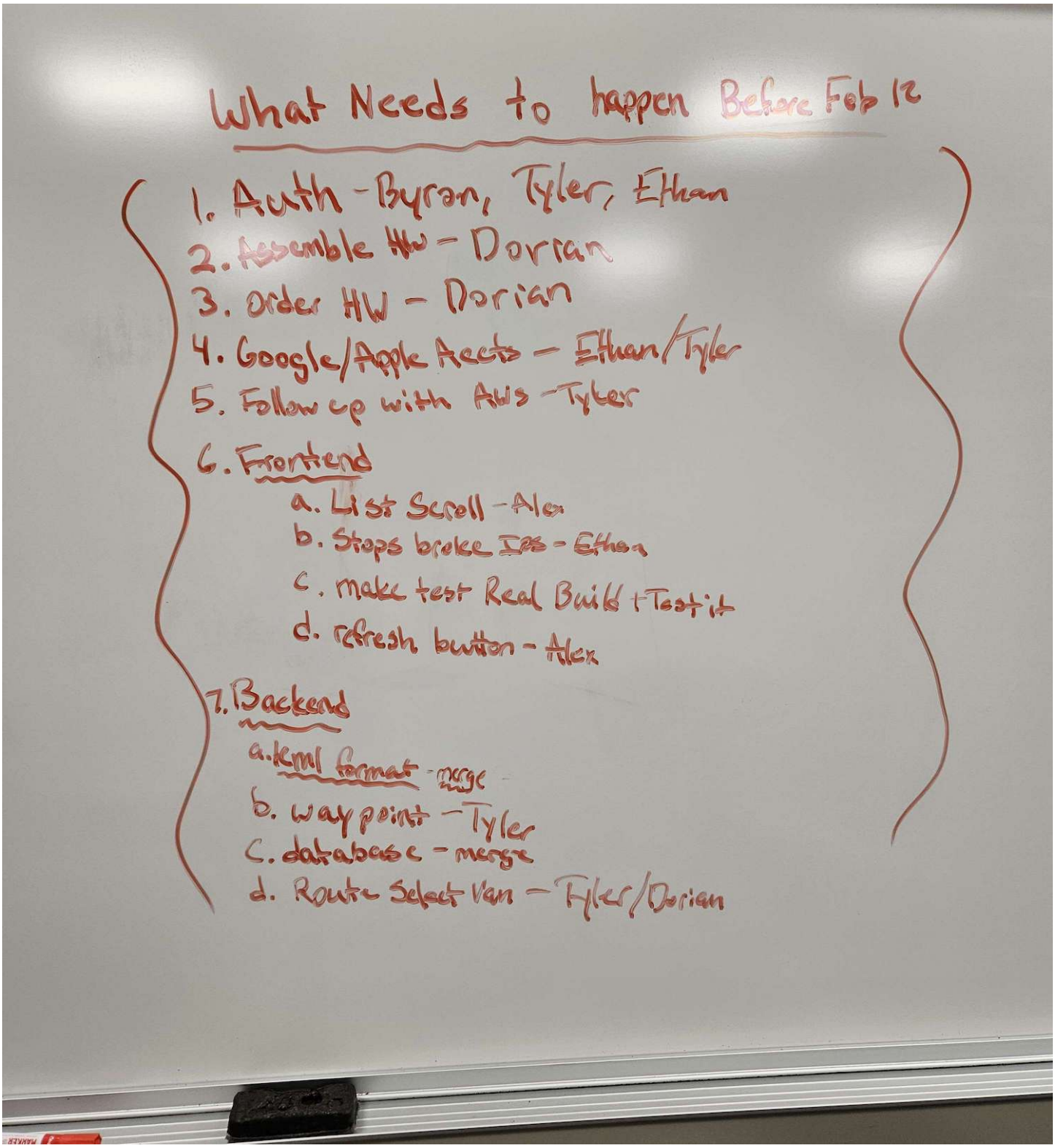
Commenting

Often times, when I plan: *the code writes itself*. I find people's use of GitHub Copilot pretty funny, as normally people don't comment code that well. But when you're trying to tell the AI what you're going for, you tend to explain yourself a little better and more clearly until it gets the right code that you're looking for. It's the same idea: if you outline your code, you (or AI) can write it pretty quickly. I'll have an entire section on commenting coming right up.

Gallery



A preliminary plan I worked on with my research advisor on the whiteboard.



A whiteboard brainstorm of upcoming tasks for the OreCart project.

---

[1] Don't get me started on pointing.

[2] Yes, the more senior engineers are considering the bigger picture, but I feel like us lowly junior-engineer-bug-fixers are often not involved in that when it can actually be really important if not useful to learn.

Comments

Recently I was in a coding interview with a “big tech” company. I had stumbled through the actual coding part and found a decent recursive solution and was happy with myself, and then I got an unexpected question from the interviewer: “What do you think about commenting? What is your policy on commenting code”

I made up some rushed answer about how I like to comment and try to comment everything, but also there’s a balance in commenting and if the code is self explanatory it’s not necessary. Almost immediately after, he spoke for 2 minutes on how commenting is the worst (I nodded along sadly) and this is something they would watch out for and work on at the company. He explained that you should be writing the good code that is meaningful and doesn’t need comments to begin with.

Further, at one of my internships, my boss was actually adamant that we shouldn’t be commenting code (to my coworkers’ delight!) and **definitely** not writing any documentation; again with the idea that the code itself should suffice [1].

However, school harps on you repeatedly to comment your code. From your first CS class to your software engineering course, you’re expected to follow “good practices” and comment.

What’s the deal with commenting?

It’s a balance

I greatly respect my old boss and that guy in the interview, but I would politely disagree with them. Like all things in life, it’s a balance!

Yes, I think the popular argument that the code should suffice itself **is true**. Don’t go around trying to one-line things into oblivion for the sake of “efficiency.” If it doesn’t make any sense to look at at-a-glance, then you’ve just wasted everybody’s time with regard to literally anyone and everyone who looks at that code in the future.

But I would much rather write a comment and have to remove it then write important business logic that makes no sense and have to rework it or try to understand it. Inevitably, if you’re doing

complex and useful things, I would imagine you'll have some code that needs commenting or documenting.

Like in the last section, if there are architectural/planning decisions, they definitely should be commented. Or, another scary idea, we could *write documentation* and put it there. 😱

Your job will inevitably have some "requirements" one way or another, but ultimately I think as long as you can communicate your design and thought process whether it's through commented or uncommented code, you can't go wrong. It's just the necessity to be *willing* to communicate.

---

[1] In that internship, I actually stubbornly wrote some code that contained comments. I wonder if they deleted it yet.

Complexity

Complexity can mean two things: time complexity, or literally complexity in a system [1]. I'll refer to both as such below for clarity. I had originally planned to just write about complexity, but I figured time complexity wouldn't hurt to mention here too.

Complexity

Senior engineers such as those from Google or my previous internships seem to all agree that complex systems are kind of the devil. (I'll speak about a talk from Google engineers in the next section.)

This has also been a coming-of-age realization in my growth as an engineer. I used to want to make the absolute *best* systems made for longevity that were optimized for everything in advance and covered every imaginable use case.

It's not a problem until it's a problem. Often, these projects went nowhere because we got lost in the sauce of complexity; trying to cover every base and every little thing slows people down.

We're getting a little more into the business side with this chat, but that's not a bad thing. You can and should make decisions that prevent the scope of the project from quickly growing out of reach, because it inevitably will if you don't make some sacrifices.

Complexity, whether by adding another framework, library, more API endpoints, more features, or whatever it might be, leads to more that sits in you and your team's headspace. It will probably come up when you least expect it or want it to, and it will be inconvenient to fix it.

This isn't to say good systems can't have complex features, but rather to pick your battles. It would be better to take some time to think upfront and make a good decision rather than having to spend hours working around it or rewriting it in the future. Rewriting things takes time and money, which are two things we would rather save.

Time Complexity

One thing that I never really expected from my CS degree was a talk about complexity. Everyone reading this probably has at least heard of it, but if not, in an oversimplified manner: time complexity describes how quickly a program will run *in the worst case*. For the purposes of this writing, I'll make it quick, as I think other points are more important but this is still worth mentioning:

Time complexity-optimized solutions are nice, especially as they're the focus of many academia classes, like *Algorithms*, *Theory of Computation*, *Programming Challenges*, and probably more.

Often, however, I think people in and coming out of academia worry about it too much or too little; they're always horrified of doing a triple-nested for loop for the O(n^3) complexity. Or, they might not even think twice about the data structure they're using for an operation.

I think there's a balance: write the code first and see if you can optimize later. Or, write it optimized the first time if you can (I wouldn't think of that immediately, and I know others may feel the same). This is also why code reviews exist. In an ideal world, your boss or friendly neighborhood senior engineer is going to hopefully catch things if they're a big issue.

It's also important to consider the use case; one time I had written a quadruple-nested for loop in a research project, and to my surprise, my advisor's reaction was that he didn't really care. Yes, it was a rather poor complexity, but the dataset we were working on was small enough to where the difference would be negligible.

TLDR: Time complexity is good to know and have in the back of your mind, but know your use case.

---

[1] Sorry, space complexity, I didn't forget about you, just not super relevant here. :)

Good Code

When I first started to code all of those years ago, I was inspired by this video about a "prodigy" programmer, Santiago Gonzalez, who went to Mines when most people around his age would've been starting high school [1].

In this video, both Santiago and Mines professors speak about the necessity to not just writing code but to write *good code*. And while I agree with them, I didn't truly understand what good code was for years [2].

The naive understanding is that shorter code equals better code, which is definitely not true and likely just presented as a simplistic measure in that video. However, I often see introductory computer science students assume or equate that in their own minds, so I think this is important to debunk.

Short code *can* be better, but if you're gunning for a one-line solution that is just going to be rewritten by someone else in the future, you should rethink it.

This isn't to say that we can't be smart, e.g.:

```
lst = [1, 2, 3]
count = 0
for el in lst:
    count += el
print(count)
```

versus:

```
lst = [1, 2, 3]
print(sum(lst)) # better!
```

The second case is better because we're not reinventing the wheel and it's semantically meaningful and obvious what the code is doing.

Writing good code is about communicating well. If other people can understand it, you're doing the right things. This doesn't just mean what I've said above; it's everything that comes with it:

comments, variable names, maintainable code, etc. See the *Good Code* and *Design Principles* sections for more.

Good code is also **readable**. I learned to read code before I learned to write it, thanks to an online course that was more about copy-pasting [3] than truly writing. But I learned the importance of understanding everything in the code and recognizing patterns throughout. I can assure you that you will deal with many different codebases of varying quality and organization throughout your career. Reading and understanding the code quickly will always take some time, but you will get better and better at it with practice.

Good Decisions

Writing good code is good, but when you think on a larger scale of a good system, there are other considerations. In September 2024, two Google engineers, Paul Christopher and Rob Warner, came and gave a tech talk to Mines ACM [4], had a really memorable quote:

> The things that are expensive to change are the things that are hardest and most important to get right. Building good systems means making good decisions, not writing good code.

I love this quote. For context, Paul often says in his talks that he spends a lot, if not the majority of his time (as a senior engineer) deleting and refactoring code. Thus, making good decisions is important. The more code you add, the more room for error there is. The worse your design is, the harder everything will be to maintain in the long run [5].

If you make good decisions, your systems and your code will reflect it. You and your fellow engineers hopefully won't hate working on it. If you make good decisions, your systems will scale in both capability and maintainability over time.

Good code just has to make money

There's one alternative perspective I want to share. One of my old bosses used to say that as long as the product is making money, we don't necessarily need to worry that much about the code.

I'm too much of an idealist to fully agree with him, but I see his point. Especially if you're a startup or small company, you can't waste time forever trying to improve and clean your codebase. You're

out there on the frontlines trying to push out the next feature ASAP so that you can grow and stay afloat.

I disagree with this point mainly because if you're just rushing features out, eventually your code is going to catch up with you and be a pain to work in. There's a balance to be found here: can we write good code and make good decisions the first time so we don't get into these situations anyways? :)

I don't think there's a right or wrong answer here, it depends on the case. Just weigh your options: does fixing *X* and *Y* in the codebase save us more time and money in the long run than it would to push a few more features out of the door?

---

[1] This is actually how I found out about Mines, and from about 7th grade, I had dreamed of going and following his footsteps. It's funny to look back on.

[2] For years, whenever I had asked for help in communities online, I was more preoccupied making sure my code was "beautiful code" than actually receiving any help. Not to say good/beautiful code isn't a good thing, but when you're learning, you have other priorities. Hopefully where you learn from teaches both.

[3] Not to say it wasn't a good course. It was a Minecraft modding course for 8-12 year olds. I think there's little chance that you can explain the intricacies of Java to an 8 year old, so I see why they taught it the way they did.

[4] I love these guys - I think I've seen every Paul Christopher talk since Fall 2021, and he never disappoints. Rob also has come along and delivered really insightful advice.

[5] ...and especially in the context of Google, you wouldn't want to end up in the Google Graveyard!

Attitude

Let me preface this chapter by say that can totally take or leave my advice. As with this whole book, I'm just here to say what has worked for me and made me enjoy my teams more.

Attitude is important in CS. Sometimes I think that this is the crucial thing that differentiates a computer scientist from other people, even if they know how to code.

Since I started coding at a young age, I grew up with friends who would always be worried about what they were going to do after high school, and they would always ask me for inspiration. I would simply say computer science, because I knew at that point and had a dream. Those friends would often give coding an honest effort and be able to code, but never really enjoyed it. It's not to say those friends couldn't have learned the mindset and the field - some of these friends I know are now studying neuroscience and psychology, and I'm sure they're going to go great places in their careers. But whether or not they wanted to commit their life to the ups and downs of coding is a different question, and one they probably answered well.

When I'm asked for advice by high schoolers or undecided majors on going into CS, I tell them it can be great, but I make it abundantly clear what they're getting into. I'll tell them how they need the right attitude, mindset, and a *lot* of patience.

Empathy

I used to have the line "empathy over genius" plastered all over my website. While this may have been a little bit extreme, the point I was trying to make was that if you can understand and work with people, the genius relating to the code isn't the important part.

Empathy is defined as: *the ability to understand and share the feelings of another.* In this context, I am using empathy by definition but also adding ideas of soft skills and compassion.

This is the hardest thing to learn for computer scientists (or anyone, for that matter). We're taught and wired to systemize rather than empathize. We solve problems, and problems with people are usually much harder, so we would rather solve those problems with machines.

A lot of people say you can't learn soft skills and that many come innately. Whether or not this is true, I truly believe you can learn a lot of soft skills with a lot of practice. The results may not be immediate, but you will get better over many years.

Start by asking someone how they're doing, and truly listen. Notice how their face changes in relation to new information. Did you say something funny? Did you say something hurtful? Notice these things and reflect on them.

I'm sure there are plenty of other guides and books and people in the world who could help with this. But just stay curious about yourself and others, and aim to care about others. It goes a long way.

Remember where you came from

So many computer scientists I know are so humble and modest that I questioned even writing this section. But you're bound to get a few big egos every now and then. Whether it's because they invented some great technology or startup or maybe they make three figures more than you, these people are sometimes not great to work with. Sometimes they are, it depends.

So, here's my two cents on being humble: Try to stay grounded. Remember when you could barely write a line of code, and remember those people who took you under their wing and showed you the ropes. I think this is especially important so that we can bring new people and new ideas into the field. It rarely ever hurts to be humble.

You're probably better than a lot of people in some ways. And a lot worse in others. And equivalent in others. That's just life. You might have a lot to say about one thing, but on another topic, you might just take the backseat and let the expert explain.

Perseverance

I probably don't need a chapter on this either, as this is a universal experience at this point throughout many professions. But beyond just perseverance, **perseverance with patience** seems to be the key while coding.

Yes, your code will break. Yes, you will be angry, frustrated, and/or disappointed. But if you can be patient and recognize that it will get better if you put your mind to it, you will succeed. If you can believe this, you can also inspire your team to work through the hard things and be patient as you develop something great over time.

Growth Mindset

> *Growth mindset*, simply stated, is a learner's belief that their intelligence can expand and develop. In contrast, *fixed mindset* is a learner's belief that their intelligence is a fixed, immutable trait. - Stanford/Carol Dweck

I think growth mindset is also something you've probably heard of by now and is universal across all professions. But if not, read up! Your intelligence is **not** fixed. Your beliefs aren't fixed either. You can grow and change, and that's the beauty of life.

There will be things you don't know. Always. Forever. And you have to believe that you have the ability to learn it, even when it seems like rocket science. Start small, put in the time, and you'll get there.

Perhaps a more cynical view of growth: adapt before you're left behind! Or, even better, keep an open mind toward new people, new teams, and new technologies. CS as a field always moves forward quickly, whether it's a groundbreaking discovery like generative AI or a new JavaScript framework. A growth mindset will help you drive forward into the unknown.

Failure

For the first two to three years at Mines, I felt like I was always failing. Whether it was literally failing a class [1], failing myself when I didn't understand things in CS, failing to make friends, failing to stay healthy or otherwise.

This may come as a surprise since I often portray a calm and collected persona, but for those three years, there wasn't a semester where I didn't think about dropping out or transferring schools. But I kept going.

The OreCart Story

My greatest failure (but perhaps one of the greatest learning experiences) came via the OreCart. The OreCart is a free public shuttle service provided by Mines and the City of Golden. At the time of writing, it has three routes that usually go around Golden 6 days a week.

In the late summer of 2023, Mines and the City of Golden came to us via Innov8x and the McNeil Center (lovely people!) to develop an app that would track the OreCarts so that people could know when they were coming and plan accordingly. This was an incredible opportunity for any students interested. We thought it was the most real app that you could ever make in college, in a class or club. It was a mini startup.

We had a core team of great, great, engineers. Since I was the ACM president at the time, we had the full support of our club and recruited more students to help out. The team developed the app for an entire year. One sentence doesn't do that justice, but that's what happened. We had weekly meetings where we were building real software and real, custom hardware solutions. Dorian developed and 3D-printed an entirely custom hardware unit that would go in the buses. Alex linked into that with a tracking system within our UI that he had mostly made. Tyler and I led the team and would step in for any big issues.

However, it wasn't all sunshine and rainbows forever. We had two major leadership changes in the Mines administration (one per semester) which brought with it conflicting priorities and timelines. Being the busy students we were, we didn't get as much done as we could have or should have. I think both parties can take the blame.

Coming into the summer of 2024, we had all gotten internships, some thanks to the OreCart project itself. We were working on improved hardware and better real-time OreCart tracking in the app interface, since our current solution wasn't quick enough to be usable. We had budgeted the 3 months of summer to finish the project.

One morning, Tyler and I got an unfortunate email from Mines administration: they wanted the app done immediately. I asked them: *how soon is immediately?* I don't remember the exact phrasing, but it was literally as soon as possible. Tomorrow. Next week. In the next two weeks. The City of Golden had just received a massive amount of federal funding that they would use as leverage to say: *get it done, people want this*.

With all of us working internships, this wasn't going to happen. We had a plan, and it was up in arms. We stopped all development anyways because we were unsure of the future of the project. When Mines got back to us, they had the City of Golden behind them. It seemed a foregone conclusion that we were getting slated. Tyler and I battled for weeks to keep the project: we were trained by investors and entrepreneurs to deliver a presentation proposing that our app would be the solution [2].

Perhaps the slide that undid us was asking for ~$100k [3]: this was a generous estimate toward hiring full time engineers, which is what it would've taken to finish the app. They got back to us, telling us that we would've beat Downtowner, but Transit was so much cheaper and easy to integrate. So, the end. Tyler and I broke the news to the team, especially Dorian and Alex, who had put their hearts into this for a year.

How to get back up

This is always the hardest part. After so many failures in a row, it just gets exhausting. This sounds cliché, but you have to get back up, even when you feel lower than you ever have before. That's when you find out your limits and go beyond them.

Talk to people you trust. That's what friends are for; picking each other up when you fail or feel down! I guarantee there is someone, whether it's family, friend, coworker, etc, they will care.

Failure doesn't define you, it builds you. It doesn't say anything about who you are or who you're always bound to be; it's just an opportunity for you to learn and change. Every time you learn and change, you widen your perspectives and horizons.

In the case of the OreCart project, it was an unforgettable college experience. We helped many people (including myself) get internships as a result of it. We learned how to *actually* engineer things, more than school or clubs could ever teach us. We even learned the cutthroat nature of startups and battling for business.

I'm not bitter. As a joke sometimes, I will be, but ultimately, it's in the past. It hurt a lot at the time but knowing that we had done everything we could is enough for me now. Knowing that I learned and changed so much from this project for the better is enough.

Failure brings new opportunities too. My old "cofounder" Tyler and I are still working on new startups today - and we can go in with more confidence and our heads held high.

I know it's tough to fail. And I know repeated failures hurt worse as they add up. But they teach us a lesson and make us stronger. Things won't always be the way they are.

> The light-soaked days are coming. - John Green

---

[1] I failed Calculus II my first semester! Being good at math equating to being a good computer scientist is an old and outdated myth. On a related note, failing a class is much more common that it seems, there's just a lot of built up and unnecessary shame around it.

[2] I never signed an NDA, so I feel no guilt in saying we were up against Transit and Downtowner. Having used Transit before, it's actually a great app. I assume Downtowner is too. We stood no

chance.

[3] I wrote a slide saying that we needed about $100k if the city and Mines wanted the app done as soon as possible, considering we all were working. I take full responsibility for the slide and it's nobody else's fault but mine. I had no idea what I was getting myself into.

Feedback

I took a class called Tech Startups from the founder of Piazza, Pooja Sankar. In addition to her teaching *Difficult Conversations*, she spoke a lot about feedback (which I believe was inspired by the book [1]).

If you're like me, you're probably wondering what all the fuss is about feedback. It's such a common word with perhaps even negative connotations; e.g. you try to give feedback to someone and they don't listen, or annoying survey on the internet are pestering you to give feedback about some service.

The type of feedback we're talking about here is centered around personal growth and true, undivided attention which leads to *real* listening. This is the type of feedback that helps people *change* and *grow*. Feedback can make a meaningful difference in running a team or in any sort of organization. Feedback could even be having a tough conversation with someone you love but are not quite getting along with.

Feedback is a gift

One of the few files I saved from that class was Pooja's lecture on feedback. I'll add my own commentary to a few of the points, but I give full credit to her for this premise.

> **Feedback is a gift**. It is an opportunity to connect with the other person and to build exceptional relationships. - Pooja Sankar *[emphasis added]*

This is to say that viewing other people's constructive criticism as an attack on yourself is unproductive and missing the point. View feedback and constructive criticism as a chance for both parties to get to know each other more and both improve in certain ways. It's a gift!

Giving feedback well looks like (citing heavily from Pooja [2]):

- Leading with your intentions: be honest and open so the other person can be too.
- Make it a conversation, rather than rant from one person
- Focus on what actually happened, avoid speculating on intent
- Check in with the other person and *listen*: how are they interpreting you, and what do they need to move forward.

Receiving feedback well looks like:

- Being kind and courteous, e.g. thanking the other person for sharing feedback & their time.
- Fight the urge to be or appear defensive: you're both being open and candid with each other.
- Ask questions to develop a **genuine** understanding; be curious about *their* story and *their* understanding.
- Repeat back what you heard and interpreted - it may be different from what they intended!

This was such a big deal in our class that I think we spent multiple weeks on it. We spent at least one week and had multiple scheduled feedback sessions with the class and our own group.

I am so confident in this approach that I have used it twice before in my own teams. In one team I was on, there was a falling out between two members. My boss had handled it fairly well, but I thought I could do even better and try to create a more friendly environment for them to work in. I invited them both to sit down with me in an empty classroom, and I acted as a moderator upholding the rules above. I quite literally opened up the slide deck on feedback and gave them an abridged version of the lecture beforehand. I'm not sure how much they both truly listened to each other, but I know they at least listened more than they would have, had they done it over text or even in-person one on one. They went on to work more respectfully with each other and accept their differences.

Another time in my side projects, I was working with someone I found quite difficult [3]. We would get so riled up and angry at each other; there was no hope of ever resolving things. We couldn't really even keep it to a respectful discussion about work - personality and other nasty claims got involved (not saying I am free of guilt, I contributed greatly to the problem). What I realized was that I wasn't *truly* listening. I was just trying to give feedback to someone who wasn't open to it. I didn't understand their needs and spaces in which they would be comfortable to discuss. It hurt, and it hurt both of us for a prolonged period of time. But after opening up some of those wounds and

being curious about each other's intentions and true feelings, we were able to heal and work together in a much more efficient and friendly way.

---

[1] *Difficult Conversations: How to Discuss What Matters Most* by Stone, Patton, Heen [Amazon]

[2] Pooja's slides aren't public anywhere, but these points were bulleted on her slides. With my knowledge from her, the class, and *Difficult Conversations*, I've added some points and fleshed out the explanations.

[3] See my section on *Difficult people*.

Getting things done

Another stellar element of the tech startups class I've been mentioning was reading the book Getting Things Done (GTD) by David Allen. This tells you how to organize your life, both personal and professional, and how to build tools for productivity that will last [1].

Throughout the class and our reading of that book, I was gratified to learn that a lot of my systems for productivity are exactly what he suggests. I'm not saying this to be full of myself; if you read it, you will hopefully find some of your organizational tools formally defined there too.

My guide to productivity, at a high level, is:

- Know (and learn) yourself
- Trust your systems
- Minimize distractions
- Continuation

That's it! The rest are details that you'll learn.

Know (and learn) yourself

You know what works for you, or you have at least part of an idea. You know when you should go to sleep and get up. You know when you need to eat and how grumpy you'll be if you don't. You know what systems you might use, and you probably know deep down whether they're effective or not.

Some people have an innate clock. I usually have just some sort of instinctive idea of what time it is and how long a task is going to take me or took me. If that's not you (or even if that *is* you), try Pooja's method. Pooja showed us one of her calendars from when she was starting Piazza; it was chaos. She had flights, meetings, and probably like 3 things going on at any and every given hour of the day. One of her most important recommendations was to mark things on your calendar, and then if they take longer, extend the time you took on your calendar event. Then, look back on how much time something took you to inform how much time you're going to spend on your next task.

This is what I mean by **learning yourself**. If you can increase your self-awareness of your productivity, you can figure out what's going well and what you need to change. This will help you a lot!

Systems

Systems are those structures and tools which you **trust** to organize **stuff**. Trust is crucial here: if you don't trust that, for instance, your notes app is going to save, how do you expect to ever be productive? Or, if you don't trust that your calendar is an accurate reflection of your time, why use it at all? **Stuff** is, well, stuff. Maybe it's an assignment you need to do. Maybe it's an appointment. Maybe it's a meetup with a friend. Whatever it is, you'll need a place to put and remember it so that you do it.

Before I continue, here are some decent apps and methods which you can use:

- Your phone or computer's notes app (this is what I use: it's simple and free!)
- Your phone or computer's reminders app
- Your phone or computer's calendar app, or Google calendars [2]
- The increasingly popular Obsidian
- Good old pen and paper (surprisingly effective and **not** outdated as we may think!)

Build up structures that help you. I like to split up my short term TODOs, long term projects, and those projects which "would be nice" but need to stay out of sight and out of mind for me to get things done.

- My short term TODOs are in my notes. I organize them vaguely by priority, e.g. the highest priority things go to the bottom so I can delete them easily.

- My long term projects are sometimes in my notes, and mostly float towards the top so I see them first while my short term TODOs are near the bottom.
- My “would be nice” things are in a “Project Ideas” document in the cloud. These are things that I want to get around to eventually, e.g. over a weekend or break. I’ll open this up during those times, but usually not when I’m at school or working (or I’ll open it to put stuff in for later).

Projects are hard to get done, but that’s what makes them worth it in the first place. You’re investing time and effort over an extended period of time to make something great. Often times, I don’t put my long term projects in my notes and instead in my calendar as an hour block with an alert to take a swing at it. I don’t expect to get it done, but I’m setting myself up to spend a bit of time every day on things.

An old professor and friend used to say: it’s the **aggregate of marginal gains** [3]. If you can just put in a small amount of time each day to “chip away” at a larger project, it will eventually become much bigger and much less scary to deal with.

One other system beyond a TODO list like I’ve said above is a **calendar**. Again, do what works for you, but I think to some degree if you’re trying toward higher levels of productivity, you need a way to manage your time.

Finally, as my first point explained: figure out what works for you. Not all of this advice is going to work, but hopefully it sets you on the right path. And if you find your systems aren’t working, change something! Drop a class. Tell your boss something isn’t working. Reorganize your life priorities. Often times we think we have less control over our lives than we actually do.

Distractions

Distractions are normal in life. They will always be there, however much we try to shove them away.

However, one method I find helpful to manage my distractions is to group them. If I’m already distracted by something (e.g. a friend or family member distracts me from coding), I’ll group some other tasks with it. That might be a great time to get up and stretch and to talk with them and give them my full attention. And then, if I can’t get into the zone again afterwards, I’ll see what other tasks on my notes list look inviting.

Sometimes I’ll even save tasks for these moments, because I know I’ll see these moments again. For instance, if I need to prep some food, go get groceries, and take the trash out, I’ll do all of those

(not necessarily in that order, lol) in an hour or two and then get back to work and be able to fully focus.

Don't try to optimize your life fully free of distractions. I've done it. And it sucks. You need friends, family, and people in your life. Just know how to work *with* distractions, not against them.

Continuation

The final thing that's important in organization and productivity is continuation. **It's not enough to do it once**. I never quite understand the people who clean out their room once every few years. Keep on top of it! The less distractions you let build up and the less that's in the back of your mind of what's around you, the less you have to worry about. It's the same with your organization and productivity. Don't let it slip, build up, and then you're stressed because you have a million other things to do but you also need to clean whatever it is; do a little bit when you can. It adds up.

Planning your days and your tasks is time well spent. Take 1-3 hours every weekend to reorganize your priorities [4], or spend 5 minutes every night before you go to sleep reorganizing. At the end of a day, I find it productive to spend those five minutes and see what I can knock off my TODO list from the day or add things that I'm thinking about. Just make sure you don't turn your planning into procrastination.

Often times this is unavoidable in life, but it feels like we're always reacting to things. People will drift in and out of your life at unexpected times, and so will work-related things. In my eyes, the less you can react to things, the more you get done. Once you get things done, rest assured that *something* is going to come up; and you'll have the freedom and bandwidth to deal with it.

Buyer beware

Despite all that I've said here, don't enter the **productivity trap**. The productivity trap is the idea that you have to be productive 100% of the time; you don't *have* to, and in most cases, you *literally can't be* [5]. Sure, if you're pushing toward something great you can make sacrifices, but otherwise, say no (see the next section).

Similar to this is the **self-improvement trap**: you're always buying things and just waiting for that one thing that's going to change and fix your whole life (no thanks to our modern consumerist culture). In reality, buying things probably won't make you that much happier or fix your whole life.

The self-improvement trap implies that you're not valuing yourself already and not believing in your ability; believe in yourself and how far you've made it, and then make healthy improvements on top of that.

---

[1] This book is pretty outdated by now. I think we read the 2001 version, or maybe the 2015 one. Either way, there are some dated references. The good news is that most of the systems David Allen mentions can be easily applied to the modern day and the "digital world."

[2] ...which can sync with your regular calendars, by the way. Your work calendars which are probably on Outlook, which can also do this. Get everything into one place, it helps.

[3] I think this is originally from Atomic Habits, which is a great book, but it looks like author James Clear also posted an explanation here.

[4] I think GTD suggested entire days, but do what works for you.

[5] I should take my own advice sometimes. :)

Work-life balance

One time I had lunch with a few executives from a fairly large tech company [1]. At one point, one of the executives had asked me something along the lines of: "So imagine we're on the brink of a discovery/innovation and you need to work late. Are you willing to work overtime without any pay if you're close?"

Looking back, this may have been a slight red flag, but in the moment I had responded with some noncommittal answer leaning towards "no," which if I read the room correctly, may have been the wrong answer. Those guys seemed to be looking for students who would give it their all and work long hours, which isn't bad if that's what you want. But I think the pursuit of "great things," especially in a startup environment, can start to blur the lines with your health and life, and therein lies our issues with work-life balance [2].

With the world being so technology centric, it's so easy to just write one more line of code on your laptop when you're at home, or do more coding, more thinking, more engineering. If you're getting Teams or Slack notifications on your phone, you'll probably keep on thinking about your work too. This "blurring of the lines" is, in my opinion, why so many people struggle to make compromises between their work and life, which causes stress and burnout.

One thing I realized that adults and older people (at least older than the college age) do well is set boundaries. They're not afraid to 'be judged' and say no to going to a party and go sleep or take care of themselves instead. They're not afraid to say 'no thanks, I don't want to stay late today.' If you're a college-age person reading this, I think that's a great skill to learn.

Building boundaries means setting up things in your life like exercise time, time to eat, and time for friends, that are non-negotiables as you work. That way, you've got the foundations to be healthy while still working.

Burnout

I'm by no means a health expert. I'm not even sure if I've officially had burnout before, but there have been two times in school where I think I've felt it [3]: I felt numb. I listen to music at least a few hours of every day, and no music would sound good anymore; it almost irritated me. Interactions with people seemed less interesting and meaningful, and I didn't even feel stressed, because I just felt so numb to everything.

Regardless of whether or not that was burnout or not, and regardless of the fact that burnout may manifest differently in people: life is a lot better when you don't feel incredibly numb or hopeless.

So, you just need to draw a line. Build some boundaries. Find ways to make your life *yours*: play sports, play an instrument, see friends, see loved ones, et cetera, **outside** of work. When you're working, you can push if you're excited about something, but if you find yourself pushing over a prolonged period of time (> 1 week), maybe reconsider the arrangement.

The entire next section speaks about how to say no and build boundaries in your life. This is also key in avoiding burnout and maintaining a healthy work-life balance. And, for the record: I don't like the expression "maintaining a healthy work-life balance." It seems like a love like ghosts [4] situation: everybody talks but nobody knows. Or, everybody talks but seems to disregard their own advice and do their own thing.

The best thing you can do is to **find what works for you**; stress is a normal part of life, but if you become incapacitated or depressed over it, you need to change something, and hopefully quickly. The good news is that there are *so many* options for help - do a quick Google search or talk to someone you trust.

[1] At Woody's, obviously.

[2] I've been reading Build by Tony Fadell (who co-created the iPod and iPhone) and he also has some good thoughts on this. Try and fail and succeed as you do, but don't hurt yourself, literally or metaphorically, working long hours in your twenties.

[3] For more information on burnout, Mayo Clinic seems like a good place to start on the topic. Please do your own research and **get help** if you're struggling.

[4] I love Lord Huron, and I think the metaphor holds up OK here despite being a little cynical.

Saying no

The best advice I ever got from my research advisor (or honestly anyone) was not how to code Python better, communicate with people, write scientific papers, do research, or anything like that. It was a small snippet of advice geared at managing your life: saying no.

His analogy goes as follows: work is like a gas. If you just try to run around waving it toward some direction, or trying to contain it from spreading, it's going to do nothing. More of it is going to keep finding you, too, if you don't set boundaries. You have to **build walls** in your life to stop it from spreading [1].

This advice is worth spending an entire section on because it's genuinely life-changing.

Stressculture

I always thought I needed to prove myself to everybody. Part of it was personal, but part of it was because I felt threatened or afraid of the culture. I never thought I would succeed unless I worked harder than absolutely everybody all of the time. Unfortunately, these sentiments seem pretty common in today's age, and almost exemplify the American dream of becoming rich and wealthy if you just work harder than everybody always.

Enter a term that I like to call *stressculture*: A culture where performance outweighs the good of your own health and emphasizes greatness at the cost of immense stress.

In the Spring 2024 semester, I stressed myself out to a near breaking point.

I was taking 6 classes (18 credits): algorithms, tech startups, HCI, development of 210, macOS app development, and a humanities class. I was still a lead TA for 128, and I was managing the (now-defunct, but then startup) OreCart project, and I was also the Mines ACM President and HSPC Chair, and additionally, for a little while during the start of spring, I was preparing to present my research at SIGCSE. I also played intramural soccer a little later in the spring [2].

If you want that in credit-hour terms, I'd probably say it was around 27. If you want that in real human being terms, it sucked. I stopped eating a lot, and what I did eat was crap. I stopped exercising, because there was seemingly no time to. I stopped meditating. I slept less, because I needed to get up and get to the library ASAP.

The only times I ever *stopped* and did nothing were on my walks to and from school since I lived so close. Even then, I would use those times to "get ahead" in the areas I had inevitably been neglecting and think over things.

The old analogy holds true: **something has to give**. I was quite simply doing too much for things not to give; giving my attention to one thing meant seriously downgrading how effective I was at another thing.

I cheated, sort of. On the walks, I would think about what I needed to do for Mines ACM and HSPC. At Mines ACM, I'd think about OreCart since all of our engineers were there. At home, after my mountains of work, I would try to catch up on the development of 210 and my 128 TA tasks.

There was no time for family or friends whatsoever. I made two new friends that I enjoyed that semester, and made an effort to see them, which was incredibly stressful too, as I knew I was hurting every other aspect of my life.

Looking back, it was miserable. I'm impressed I got through especially with the compounding effects of not treating myself well, which would just make it harder for myself. The only things that got me through were reading and journaling and sheer, foolish determination.

I remember the point at which I wanted to change was when I was speaking to my roommate. He would ask about ACM and OreCart, and I would try to explain, only to have long, awkward, 15-second pauses where I would look around and try to find my words because my head was so jam-packed with thoughts. My memory got much worse, and I developed a stutter because there was so much going on in my head and it was moving quicker than my mouth could.

It wasn't worth it. I finished out the semester, but to many people's surprise, I immediately resigned as Mines ACM President + HSPC Chair (despite what I think was a community that I built that enjoyed my leadership) because I just *couldn't* cope. OreCart would unfortunately fall through over the summer, which was incredibly sad, but not bad for my health. I tried to promise myself I would never do 18 credits again, but I did again in the fall; however, I coped much better having much less on my plate.

Not so unusual

Unfortunately, this is the saddening reality for a lot of people. I'm in a privileged position where I'm economically stable, but I know people in these positions who aren't quite the same and need to work on top of school to survive. This culture also encourages coping in the wrong ways, with alcohol or drugs, which I can proudly say I never did, but I again know people who cope that way.

I recommend everyone I know to do less, probably to an annoying extent. People come to me in similar situations and I tell them to say no to things - we often say yes to everything because 1) we're nice people, but 2) we see opportunity in things. If you want to say yes to something, sleep on it. Ideally give it a week to decide. See how willing you are to say yes after waiting awhile.

It's hard to do less, especially in the middle of a school term, but there is really always a safe way out: whether it's resigning from a project, withdrawing from a class, or simply taking a break.

However, the easier way to do this is to set boundaries (or with our analogy, build walls). As an adult (barely), I've noticed that older people and adults are great at this, and this goes unspoken for quite some time. Whenever you're staring down your next semester or work project or new big idea, ask yourself what you're willing to sacrifice, what things you could leave behind, and how you're going to change from doing everything all of the time.

Think about the big things: what do I hope to accomplish over the next few months? How does this relate to where I truly want to be in life? How does this relate to what I truly want to do?

Equally importantly, **think about the small things** too. It astounds me how many people ignore, for instance, time to eat or sleep in their schedule. They just don't care or pretend it doesn't matter. **It does!** These are the foundations that will allow you to do things to begin with. You don't have to schedule time for friends, but think about having open time where you can spontaneously do things for you *or* go and see friends.

> Nothing changes if nothing changes. If you always do what you always did, you'll always get what you always got. No matter what it is. [3]

A fun story

One day I went down to my friend's cubicle in the CS PhD offices. He had a sticky note up in the center of his desk which read: "Say no!" – he works for the professor who I used to, so I asked him if the professor had given him the same advice. He said: "No, Ethan, you told me that."

---

[1] Maybe this is where the analogy breaks down, but imagine we need to build some super pressurized walls that won't leak out the gas.

[2] I won't talk about it too much in this section, but this is an absolute context-switching and attention residue nightmare.

[3] Miscellaneous and aggregated quotes, partially from a silly little motivational YouTube video I watched, but genuinely life changing advice.

Self care

...is something that is way too often neglected by folks in our profession. The amount of times I've heard things along the lines of "I am mentally unwell" or morbid jokes about mental health at Mines is astounding. I think people push themselves so hard to reach a certain GPA and appease others, make a startup to make a living, whatever it is; to the point where they forget to care for themselves.

Self care can look like many different things: perhaps it's sleeping in a half an hour (or sleeping at all!), getting a good meal, going for a walk, or anything that helps **you** feel good [1]. I am by no means a health expert, but I feel like this is important to bring up because it often gets neglected or brushed aside.

Caring for others

Definitely care about yourself and treat yourself with kindness, but once you have stable foundations, it's great to care about others too.

It is **never** a bad idea to check in with somebody. I don't think there's been one time where I've checked in with a friend and they haven't been at least kind or respectful, if not grateful about it.

Through college, I think there were at least three instances where I had friends dealing with serious mental health issues [2], and I felt so unprepared. But the fact that you can simply be there for someone will help them more than you can ever imagine (which I have been told by people I've helped). Show **compassion** to them: imagine what you would need if you were in their situation. Be present and attentive with them.

Mindfulness

The greatest act of self care I do on a regular basis is mindfulness meditations. Hear me out! I'm not trying to become a monk or sound "new-agey" nor do I sit in rooms humming like meditation is portrayed in the movies.

Mindfulness is all about noticing your thoughts and feelings **in the present moment** and questioning and observing the world thoughtfully. I'm not sure I could teach it to you in a couple paragraphs. Check out Healthy Minds below for what I think is the best program to learn.

Mindfulness is rooted in neuroplasticity, the brain's ability to change over time. You can re-wire your brain to notice when you're getting carried away or when you're acting on impulses and urges when you don't want to be. It's like your brain is a muscle that you can train to be more mindful of the world around you.

Mindfulness meditation changed my life. I don't worry about the future or past as much as I used to, I feel like I'm a better listener, and I feel like I can be so much more compassionate and understanding of others (and much, much more). As an aside, I don't want to make it sound like some magic pill that will cure you of all ailments. I also feel like doing it with that motive degrades your experience as it's more about how you *actually* feel. I just view the benefits as a positive side effect from mindful introspection.

I believe so much in meditation and mindfulness that I would like to offer up a few options that you could literally start in the next five minutes [3]:

- Healthy Minds - free, out of a lab in Madison, WI called the Center for Healthy Minds, with some practices inspired by Tibetan Buddhism [4]. I've been using this app for about two years and I think it's the best one; it also talks about the science behind the meditations they're teaching you, which is great for us systematic engineers.
- Smiling Mind - the first meditation app I used. It's out of Australia, but as long as you don't mind an Australian accent (I actually found it *helped* me listen and focus in guided meditation) it's great!
- Calm - paid, but I've heard great things about it.
- Headspace - paid, but I think the most popular option nowadays along with Calm. One of my friends really enjoys this one, especially for sleep.

I'm really passionate about meditation, mindfulness, and the science behind it, as it has brought me so much calm and clarity in my life. Feel free to bring it up with me, I love to talk about it. It takes just a few minutes a day and has really positive effects if you stick with it. I hope you'll give it a try!

---

[1] We just had a section on saying no - and for the record, I would classify saying no to things and building boundaries as self care!

[2] If you're feeling mentally unwell or are in a crisis, call 911 and **talk to a trusted person if you feel up to it**.

[3] These are all great options that I have either personally used or know people who do. I receive no payment from any of these. I am just spreading the word.

[4] They don't "force it" upon you or anything, nor am I advertising this in any endorsement of religion. It's just interesting to hear what actual trained monks do and how even they deal with anxiety in their lives.

Diversity in CS

Wait! Don't skip this chapter if you're one of those people living in the stone ages. Diversity is important, it benefits everyone, and you can help.

Brotopia [Amazon] by Emily Chang is a great book, though it almost reads like a report, on all of the unfortunate things that happen in Silicon Valley, particularly to women in CS. I always wanted to work in Silicon Valley, but now I'm thinking twice about working there (assuming it's not at an established company like Apple/Google). *Brotopia* has damning chapters on the founders and the cultures that thrive in the valley. Spoiler: from the beginning (perhaps it has gotten better since), it's been white males.

As a white male in CS, I'm not helping the ratios nor the stereotypes, but as I'll harp on in the next section, we can help out by **simply listening**. Hear **everyone's** experiences and ask them how *you* can support *them*.

I know multiple folks who identify as LGBTQ+ but wouldn't want to be treated any differently or thought of differently if they came out to their friends or coworkers. It's understandable, and it's scary to come out, nonetheless in a field like this and in a society like this.

When people say bad things, call them out or tell somebody with the authority to stop it. Definitely don't encourage them or give them a laugh out of pity. I've been in awkward situations where someone makes a "joke" with someone in the room who would be offended by it. It's not to say you can't make jokes, but you can't joke about things that are going to make people feel uncomfortable or unsafe.

Even if you "don't believe in diversity" (which is wrong), you should recognize that **diversity makes great teams and great products**. Quite simply, more people with more experiences means that your team has more perspectives, which in turn means your product isn't as blindsided or biased to just one experience; your product becomes more universal.

The bottom line is: **stay open-minded and listen**. It will serve us all well.

Women in CS

I can tell you nothing more important in this chapter than **simply listening**, but I will attempt to elaborate. The following are some assorted points I've learned from women in CS over the years, again by truly *listening* and empathizing with the other person.

For far too long, women in CS have been harassed and at both spoken and unspoken disadvantages in the field. Demographics make things especially hard, when most women in CS will be surrounded by a male majority most of the time.

I really began to understand the experience women have in CS after reading the book Brotopia, which I would wholeheartedly recommend for **every** man in CS (women, too, although you may unfortunately be familiar with what's discussed).

Before I begin: I hope none of this comes across as patronizing nor am I saying I am free of all mistakes. I am merely trying to be an ally and spread the word to male audiences.

Sweat the small stuff: I've spoken to quite a few ex-Silicon Valley women and students in CS (at least, as far as that goes with Mines demographics) and they all seem to agree: **the small things** are what drains you down over the days, weeks, months, and years.

For instance, it's getting asked out for the *fourth time* [1], or hit on in academic or work settings where you're just trying to focus (and where that would be inappropriate, obviously), which is endlessly tiring.

In *Brotopia*, Emily Chang describes real Silicon Valley scenarios which sound like they're straight from The Office (e.g. hitting on people, making inappropriate jokes, etc). After reading that, I texted a friend in CS, and said "no way this happens!" while describing the chapters. She said that it was normal. Every time I read something, I was more and more enlightened to every little thing women in CS must go through on a regular basis. I kept on texting my friend. She was continuously unsurprised.

Sexual harassment and discrimination is real, call it out: Again, especially in Silicon Valley (if *Brotopia* is to be believed, which it should be), this is rampant. Most people don't report these types of things, either out of fear, unwillingness, or even the fact that they're used to it.

One of my now-graduated friends said her research advisor wouldn't trust her with tasks and would give her menial tasks if anything, while favoring the other men on the team. I told her to report it, and she said it would be over soon enough so she didn't.

Definitely don't be the person harassing or discriminating, but beyond that, help to make a more safe space for women in CS. Imagine if you were barred from a job or research opportunity merely because of your gender (or any other quality about you).

I think the people who do these awful things are more hidden nowadays and usually test the waters with you, saying borderline rude things first to see if you're "with it." Unrelated to gender, I had a former coworker drop a few "jokes" that were borderline fascist and homophobic. I told him to stop

and didn't associate myself with him anymore (I *should have* additionally said something to HR, but thought the shunning would be fine).

Call people out if they're saying bad things. You don't want to work for or with those people.

Actually listen: I've been in group projects with a male-majority where the female in the group gets completely ignored. People didn't even acknowledge her. My best strength was telling group members "*no*, listen to her." One time we had spent upwards of 12 hours banging our head on desks over a project and *she* held the last few answers but the group didn't believe her. I told them to listen, and we finally and swiftly finished once the group actually listened.

Intentions are pretty clear: In line with women constantly being asked out and hit on in these environments, women can see your intentions pretty quickly. And, the sad thing is, often times having a partner doesn't even stop men in CS. One of my friends says she has referenced her partner of *many* years to a guy who was hitting on her, and he keeps on following her around and dropping hints that she looks nice. **Don't be that guy** [2].

Don't stare: I can't believe I'm giving this advice to a primarily adult audience, but don't stare. You know what I mean. I once even had a friend say she was considering dressing more like guys. That's *heartbreaking*.

---

[1] I have genuinely heard of women who have been asked out *at least* four times a term. Do you understand how exhausting that must be?

[2] Not to say you can't form genuine connections, but there's a line between normal and creepy/unwanted approaches.

Difficult people

Bad news: There will **always** be difficult people to deal with, or difficult situations with those people. In some respects, I'm probably a difficult person to work with.

However, difficult people create a great opportunity for us to uncover our own weaknesses and form better connections with people in the long run. Difficult people are arguably even a good thing on a team to provide a different perspective (provided they're not saying anything offensive or just being unpleasant to everyone).

Be empathetic, as hard as it may be. If someone is being difficult to you personally, chances are that there's something going on in there life or some external factors pressuring them to say unhelpful things. Or maybe they truly, genuinely aren't understanding your work or you as a person. Your job is to understand how and why they feel this way, and how you can both coexist together in the future.

In their shoes

Remember the classic kindergarten proverb: **put yourself in someone else's shoes**. Try to see things from someone else's perspective.

Along these lines, I mentioned in a previous section that I do mindfulness meditations a lot. One insightful idea from a guided meditation I listened to is reflecting on the following questions when you just *can't* understand the other person's perspective:

- What am I missing?
- Can you believe that this person wants happiness & to be free of suffering just as much as you do?

Often, I find that these two questions alone can lead to a helpful resolution. Because, if you feel so strongly about one thing that you *can't* see any other perspective, **chances are that you're missing something**.

This isn't to say you can't or won't be annoyed by people on a daily basis. Maybe somebody cuts you off in traffic - that's irritating. But flip the script before you honk at them: maybe they're late for work. Maybe they're nervous merging into the lane because they just started driving. Maybe they've got a kid in the back who they're also trying to keep an eye on. I'm not justifying poor driving, but I am justifying the need for patience and respect with others. It will go a long way.

Working to a resolution

Sitting down with a difficult person to work out your differences, is, well, difficult. But it's better to not let things bubble up and to talk, especially if it's a professional relationship or if it's someone close to you on a team. I'll keep on referencing this book, but *Difficult Conversations* is a great book which provides advice on how to deal with these difficult conversations.

My advice is similar to that which I gave in the Feedback section: **Be honest, truly listen, and think about moving forward**. Most people won't want to deal with these things, but as mentioned, you don't want to leave it on the backburner and just let it get worse.

Communication

Communication is all about getting the point across effectively and understandably. What makes communication effective and understandable? It's clear, concise, and moves toward a result.

Communication is as much about listening as it is speaking. If you can't ever hear the other person talking and genuinely process what they're telling you, nobody is making any progress.

Communication comes in different shapes and sizes: a hallway chat, a presentation, a meeting, an email, a text, a book, and so on.

Both in verbal and written communication, you should **hear yourself speak**. Can you actually hear and understand what you're saying as you're saying it, or are you just going on and on? If the latter, the person listening to you will probably think the same thing.

Good communication is communication that your target audience understands via the medium you chose. Whether it's a text, email, or actually speaking out loud, if you can get the point across effectively, you did good.

Know your audience

Knowing your audience is vital to any sort of communication. Especially in the engineering sense, does your audience literally understand what you're saying? Or are they missing the prerequisite knowledge to understand?

Another aspect is professionalism and formality. A bit of good old fashioned professionalism doesn't hurt, and is totally necessary if you're working with customers, clients or stakeholders. Formality also: if you're working with someone superior to you, you can show them some respect. But if they're informal with you, you can be with them too. Meet them where they're at and both parties will hear each other a little better.

Cut to the chase

This is something I've worked on in my communication for years. I always feel like there is *always* more to say, and anything left unsaid is a loss. But this creates one-sided conversation where it's me speaking *at* somebody, not with them.

Less is more, and **silence is okay**. People process very differently, both in style and in speed. If you're neck deep into an explanation and you're getting blank stares, slow down. See if you can boil your explanation down to just a few key ideas, and then if your audience wants more details, they will ask for it.

It's fine to be silent for a few seconds too. This gives more time for you to think about what you're going to say, and for your audience to process what you're saying.

Emails

One example of communication we see abundantly in our lives is, of course, email. (Also texts, but people never put as much stock into those.)

People often freak out about writing emails or put way too much stock into them (or both). I definitely still do this sometimes. The key is to limit how much you're looking at that email draft. There will always be something more you can fix or change. And once you send the email off, you'll notice something that was maybe slightly wrong anyways. So there's no need to deliberate.

Give yourself 3 attempts: One to write the email just by "winging it." Next, you should review it for any structural or grammar issues. Finally, review it one more time to see if there's anything you could remove to be more concise and clear. Maybe review it for grammar issues one last time after that. And then convince yourself that it's good before you drain more time tweaking it into oblivion.

Fundamentally, emails are sent back and forth because **someone needs something**. They need you to do something for them. They need to do something for you. They need your attention. Whatever it is, both parties want to get down to business.

If you send over **four** emails back and forth, it's probably time to refocus because something is getting lost. Change the subject line. Start a new thread. Schedule a call or a meeting in person. Or

even send them a text or Teams message. Either way, don't make a giant thread that keeps on growing with more people.

Again, know your audience. If it's a CEO, department head, or someone who you respect, maybe you can spend a bit more time editing. Your email should be more formal as well. If it's a coworker, you can be less formal because they probably just want you to cut to the chase. I think it's also a bit silly if you're super formal over email but then casual in person. Emails should genuinely reflect you and what you're trying to communicate.

Leadership

I've learned a lot leading Mines ACM, HSPC, the OreCart project, and many other side projects and group projects. Again, this is just what I've noticed. You'll learn what works for you, and I'll learn and change my advice as my career goes on. I think a good leader has a **vision**, **cares**, takes **initiative**, and **delegates** as needed. I would also like to note that a good leader can be a good team player; you don't always need to be at the helm. See the next section for more info and see below for my thoughts on leadership.

Vision

One of the most important parts of being a leader is also one of the hardest parts: pushing your vision. I feel like this is especially difficult because you're usually executing on someone else's vision (unless you're in a high-up position or making a startup). If that's the case, you should work to align your goals with your superiors' vision.

If you're pushing your own vision, you probably think about it a lot. Even outside of work. It can still be scary and hard to drive this on your own, but keep trying. Explain yourself to people, and let them ask questions. At some point, your vision may even morph as per other people's vision, or to include their visions. That's okay! Say yes if you want, but don't be afraid to say no.

My vision for Mines ACM was to 1) grow the club and 2) build a community where people build soft skills. When I inherited the club, we had just recently added more officer positions and were coming off of the back of the pandemic, so it was a tall order. Over the year I was president, I cared about people. I tried to develop our students not just as engineers, but as respectable people who could do great things in CS. I listened to new ideas from the great team around me, such as hosting a

club mixer, where many great connections were made. I rebuilt a community based on our shared interest. While it wasn't perfect, I felt like I had people's respect. People still talk to me and ask about ACM. They know I am (or was) the ACM guy. In my mind, I succeeded at my vision.

You're not going to please everyone, but make sure they still respect you. Early on in my presidency, I made some changes that were instantly controversial. People thought I was too soft or indecisive, or just didn't like my choices. One scenario that comes to mind is registration: around each registration window for the next semester, we get an influx of people asking which professors are good and which are "bad" and what classes to take in our public Discord chat.

Often, this made for unpleasant viewing. One or two professors would always be scapegoated and made out to be the devil and have unnecessary personal attacks toward them. Unbeknownst to those people, one or two of those very professors were actually present in that chat. I knew I had to do something. I would give warnings and even delete messages with unfriendly comments, which was met with uproar. People wanted to speak freely, and I also wanted them to be honest, but baseless attacks on personality weren't something I would build my community on. People were mad for a little while. But they got over it, and they respected me more for it, and it built a better community aligned with my vision.

Care

A good leader cares. Tony Fadell once said that as a CEO, your job is to care. You have to care about all of your teams and people (to the best of your ability), not just one. If your team is big enough, it may have sub-teams, and you may not simply have enough time in the day to check in with everybody. But check in with each team or each project's leader.

In today's age, **attention is the best gift you can give somebody**. If someone is on your team, I would hope that you value their ideas and are willing to hear them out. Give them your undivided attention. While there will always be difficult people to work with, I hope you genuinely care about your team too, and can at least respect the work they do toward your mutual vision.

Initiative

People can learn to lead, but I think this is what makes a natural leader. Sometimes I just get an itch to organize people and things if they're unorganized. An old family saying I've heard a lot goes: *someone has to be the boss, and it might as well be me*.

When the time comes, you step up to the plate, roll your sleeves up, and get working. Whatever that may entail. This is how a lot of people get into leadership positions anyways: they see a gap and they fill it.

See the gap and fill it.

Delegate

Despite all of what I just said about vision and care and initiative, you can't do everything. Delegate it. An effective leader delegates tasks and picks the battles which they want to fight.

At ACM, I knew I couldn't do everything, so I had to pick some initiatives that I wanted to personally sponsor and let others go. HSPC was a big initiative for me, so I put a lot of time into it. My good colleague Tyler and my would-be successor Megan went onto run a very successful, first of its kind hackathon called BlasterHacks. I supported it, but I wasn't involved at all in its administration. I just didn't have the bandwidth, but it supported our community and my vision, so I let them have at it.

When I was training Megan for her presidency, I couldn't stress this enough. Pick your battles: you'll always have the opportunity to do more, but you just can't. You don't have the time or energy. Delegate the things you can. We hired officers who were excited to do a job and willing to listen to authority [1].

Other thoughts

Leadership styles are different. That's why I'm keeping it vague. I've seen equally effective "hands off" and "hands on" leadership. Personally, I'm a more "hands on" leader who prefers plenty of communication. But there's no right or wrong, as long as your people respect you and like you and you can all get things done.

Finally, in the "vision" section, I spoke about the idea of a superior. Again, unless you're some big C-suite exec, you're going to have a superior. But don't be fooled: *titles were only ever just titles*. They're useful for describing the job you should be doing (most of the time) but they're unhelpful when they put someone on a pedestal.

One time at one of my internships we had a corporate dinner which felt corporate-y, but my mentor and I went to talk with one of the VPs of our division, and he said the same thing. Titles aren't so

important, talk to the human. He told us how despite being a more senior executive, he had to stop school to raise his daughter as a single dad and he's going back to finish his degree now, mid-career.

That was one of the best moments from the entire internship: seeing that the big scary VP was just another human. You should respect your leadership, but you should also know that they're human just like you.

---

[1] While on the subject: My vice president, Umberto, was genuinely one of the most stellar communicators and leaders I ever met. He let me delegate a lot of things onto him, and he would communicate and delegate within our team. He *just understood* how I worked, and I told him how eternally grateful I was for that. Honestly, I think this is pretty rare, but if you can get someone like that to work with you, leadership becomes so much more fun.

Working in teams

Working in teams is a lifelong skill that I don't think anyone is perfect at. But there are many things that can help you get better. And, often times, especially in software engineering, we idolize that genius founder (Mark Zuckerberg, Bill Gates, etc.) who seemingly singlehandedly changes the world. But, when it comes down to it, humans are better working together: we can achieve more and do more.

Know when to step up (and down)

Not everyone needs to manage. I feel like often times, the progression at work or even in sports teams as you get more senior is that you *must* be a leader. Perhaps this is effective if you're seriously lacking a leader, but the point is, not everyone wants to be a leader.

In Tony Fadell's *Build*, he calls these types of people individual contributors (ICs). He notes that, once you become a manager, you're doing less of what did you successful. As such, it's probably going to be scary! In the software engineering sense, you're not coding as much anymore. If you're the type who want to code for the rest of your career, there are paths forward that allow for that too.

As someone more leadership-inclined, I love good team players and seriously respect them. If I give them a task, they do it. And it's not just like they're following me blindly: I love working with these people because they are willing to roll their sleeves up and work *together* even if they're not leading the vision. They're helping out toward the vision.

A leader is a team player. They're apart of a team, so why wouldn't they be? But even more than that, if you're a leader, you should know when to step down and listen to your team players. Often times they'll be the expert and know more than you about something, and you can treasure that and use that to the entire team's advantage.

Know when to step up to the plate, and know when to step aside to let someone else shine.

Know your team, know your people

Get to know your team well! Care about them and ask them what they're up to at work and outside of it. Most people will really enjoy that type of thing, and if they don't – respect that boundary too.

Know things about your people. Maybe they focus better in the mornings; don't disturb them then (unless it's urgent), talk to them a little later. This helps everyone stay productive, and it builds a community and a culture within your teams.

I've worked with some people who process one thing at a time but *really* process it and deliver good ideas about it. With them, I make sure I'm not overwhelming them with information and instead we focus on one thing and get it done. I've also worked with people who don't mind jumping around from thing to thing. Regardless, figure out what works for you and your people.

Communicate, communicate, communicate

Communicate your feelings toward others. How do you feel about the work you're doing as it relates to you? How do you feel about how the people around you are working an acting? If it's bad, maybe tell your boss or tell people directly.

Communicate your vision: even if you're not a leader on the team, you probably feel some sort of loyalty toward the vision of your project. (see *Leadership* section for how leaders might act.) Everyone who is working on a project has their own understanding of it and has valuable ideas for it.

Communicate your work: you're probably going to be required to do this, but find ways to communicate your work. Whether it's in a code review, 1:1 meeting with your boss, or to friends and family you talk to. If you can communicate what you're doing and why it's important to you, that should excite you (or perhaps clarify to you why it doesn't excite you).

Integrating new people

Integrating new people into teams always brings with it struggles: for both the team and the new people involved. I think sometimes not enough emphasis is put on integrating new people, which can make them lose interest in your team and your project.

I've seen this in my experiences managing both officers and members of Mines ACM, and in our OreCart app. But also just at a level of being integrated into teams myself. If you're a human being, at some point you were integrated into a team or group. What made you stay and/or what made you leave? What made you feel valued? These are important things to reflect on when you have your own teams.

The best way I saw at ACM to integrate new people into a community [1] was to simply **go and talk with people** and **break down the twos**. Most people would come to the club with someone they knew, and then awkwardly just sit in the back eating pizza and not interacting with anyone but themselves.

I saw this as an opportunity: I would go up to them and talk to them. I'd ask what's going on in their classes and why they were interested in ACM. Or, if that wasn't the right vibe, I would act silly. I'm not saying this in a manipulative way, but typically, cracking some awful jokes makes you seem a lot more human to others. There's obviously a balance so that you don't lose all of your credibility, but if you can make the newcomer smile, they'll remember you. And if you can **remember their name**, they'll make an effort to get to know you and remember yours too [2].

In the OreCart app development, we were always looking for more hands on deck [3]. Finding interested people was hard enough, but then when we got them into our meetings, it was really difficult to get them truly involved in the product and the development process. I'm not saying I was perfect or successful by any means. I've just learned from my mistakes.

One mistake I made was assigning busywork. It's pretty easy for someone to sniff out that they're doing irrelevant things. We thought it was necessary to get them in with the codebase, but

ultimately their interest would fizzle out and we wouldn't bother because we had bigger fish to fry. If I could do it again, I would get them involved closer to the heart of the product and have someone mentor them more closely.

Another mistake was not communicating well. I think our "core team" communicated very well and I'm proud of that, but our newcomers were an afterthought. None of us had enough time to mentor [4] and so we couldn't give the newcomers the time of day and the support that they needed. *Invest*, literally and metaphorically, into your newcomers. They're the future of the product and your teams.

Even if your team doesn't communicate well, **be the glue** that sticks everything together. People can come to you and you can redirect what they have to say to the right person that needs to hear it. Often, I think this is my best skill on a team: it's balancing two (or more) fires and holding everything together, like that scene in Captain America: Civil War [5]. It's the ability to clarify things between everybody and make everyone around you more productive and successful.

---

[1] Building a community is a whole different skill set: it takes a **lot** of work and is a fragile organism; one step wrong or one person removed and the community starts to break down.

[2] People who know me closely know that I'm pretty good at names. I've had professors, TAs, and friends alike ask me for people's names so that they can go talk to them. I'm not even sure why, it just comes naturally to me. You don't have to be the same, but find a way that works for you. **Trust me**, it pays off. Some of my best relationships have come from simply just remembering a name and remembering a thing or two about the person.

[3] Not enough help was probably the reason the project failed in the end, but then again, it's hard to convince people to work for free, and we had no funding.

[4] See chapter: *Saying no*

[5] Just when you thought it couldn't get any worse, I compare myself to a superhero with superhuman strength. Definitely not my intention, but it's just an analogy: you're Captain America, and you've got one person (the helicopter) and another person (the platform) to hang onto.

Mentoring

Mentoring makes the world go ‘round. I think most people in CS have had a great mentor at some point, and I owe a lot of what I know to my mentors and friends [1]. Even with AI that can help you solve most of your problems, I still think it’s priceless to have a human being that you can talk to about your code problems (or your life problems for that matter). Here’s some thoughts from both sides of the coin:

Being the mentor

Don’t do it when you’re stressed. Just don’t. I’ve made this mistake and my mentors have made this mistake. I’m neck deep into coding, debugging some frustrating problem, and I accidentally snap back at someone innocently asking a question to me. Your mentee will be horrified and more unwilling to ask you questions in the future. It’s *totally* valid to say: “Hey _____, I’m a little stressed right now and don’t have the bandwidth to answer. Can I get back to you in a few hours or tomorrow?”

Try not to give the answer straight away. The answer might be obvious to you, but like any good teacher, if you can lead them to it, that’s better. If you lead them to the answer, they’ll learn more and grow their own problem solving skills much better.

Sometimes the answer is subjective or impartial. You can still give your advice (or not), but you should do your due diligence and let them know if you’re not telling them the full picture so that they can make their own decisions.

If you don’t know the answer: that’s okay! I’ve had mentors tell me that they don’t know or don’t have the time to debug and to trust myself. Sure enough, I solved it in the end.

Being the mentee

This feels a little weird to have a section on being the mentee, but having been there many times, there are some things that make the process go more smoothly. And having a mentor is such a valuable experience that you wouldn’t want to let it slip.

First, be willing to learn. A mentor isn't going to want to help you if you're not listening to them. Most teachers and professors probably feel the same, but they're obligated to help you. A mentor isn't [2]. They have jobs or things to do and have busy lives [3]. But if you're willing to learn and they're willing to help, they *will* find the time for you (hopefully for altruistic reasons rather than for something in return [4]).

Next, don't be afraid to ask questions. I learned the hard way that asking "can I ask a question" can be deeply frustrating depending on the mentor (although, in some cases, it's respectful as you're trying to see if the mentor is busy).

I personally would just go ahead and ask the question: you'll probably get some quick response and the mentor will continue to think about it and give you progressively better advice. However, the mentor is **not there to solve your problems**. They're there to help *you* to solve the problem, and then maybe push you over the finish line if you can't.

Read your mentor as well. There *will* be times when they're in a bad mood and they will think they should still help you and then both parties will somehow get frustrated by the end of it. Leave them be if they're giving you those signs, and try again another time.

Finally, thank them. They took time out of their day to help you. They (probably) did it for free. Maybe even ask them what's going on in their life and show that you really care and value them.

---

[1] Gabe, Rob, Shane, Sumner, LinZi, Colin, Jake, thank you!

[2] I've had students in office hours who quite simply do not listen. I could tell them the answer directly and they would rather ignore me and write the code themselves. In these cases, it's best to meet them where they're at and try to support how they are trying to learn.

[3] I think the best mentors often have the busiest lives, unfortunately.

[4] I like helping people. I don't have time to help everyone, but I get a lot of value out of it. Some of the advice in this book is the same I've given to my mentees. I never ask for anything in return. Sometimes you also help people because you see a lot of you in them. I do this, and I know my mentors do this too.

Tenants

There are a couple of other (un)spoken tenants of computer science that I think everyone in CS should know. We'll start off with a few acronyms [1] and then move onto some big hitters.

DRY: Do Not Repeat Yourself. This one is self-explanatory, but let's say you're a 12 year old coding things for Minecraft, and you wanted to repeat an action 250 times. Will you copy the same line 250 times or use a for loop? Hopefully the latter. [2]

But even at a higher level, if you're repeating yourself: make a function. Make a class. Make some abstract classes and interfaces.

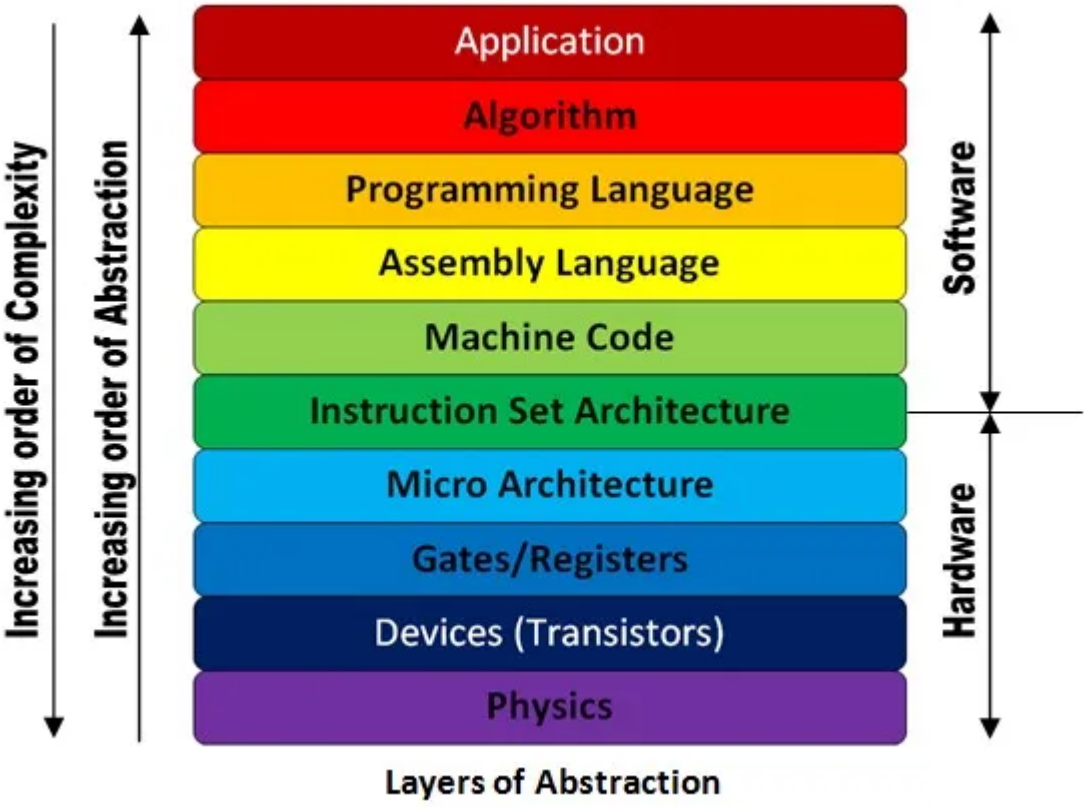
If you're repeating yourself, there's almost always a better way.

YAGNI: You Ain't/Aren't Gonna Need It. If you don't need it, don't use it. If it's not a problem yet, it's not a problem. Don't bloat code for no reason, or even features in your product.

TDD: Test Driven Development. The enemy of many programmers and students alike. Jokes aside, spend some time testing. I know it may be boring, but view it as you're just making the system more robust and building confidence in parts of the system so if you change them later, you can verify nothing breaks.

Abstraction

Abstraction is one of the most useful ideas in computer science. Abstraction is the idea that we can "abstract" details away so that we can focus on the bigger picture. I always like this picture [3] that we've shown forever in the intro to CS course:



For instance, it's useful to understand that computers are made of lots of transistors. And those transistors have a lot of physics things going on too [4].

Do I need to understand every element of how a transistor works in order to be able to program? Definitely not. And I definitely don't. Do I need to understand gates and registers to write an algorithm? No. But it is useful to know how things work, and then work your way up and disregard or postpone the details of things that don't matter as much on the "level" that you're at.

The more you see abstraction (which arguably is not limited to the realm of computer science or engineering), the better you can communicate complex things. If you understand the big picture, you can explain as much, and then when you need to get into the details, you've got that locked away in your memory too, ready to go.

The wheel

Don't reinvent the wheel, unless the wheels are flat or square [5]. Now, maybe there's a case for reinventing the wheel while you're learning. Software engineering students at Mines have forever been succumbed to making the board game Clue, but it's good practice.

When you get out into industry, you probably won't be reinventing the wheel, but this expression goes beyond that. If there's a library/framework/API/whatever out there that does what you need, **use it!** Don't fall into the trap of remaking everything yourself because you think you'll do it better (I do this all of the time, and I shouldn't).

Now, if the wheels are flat - maybe that library did good at one point but isn't headed in the right direction, or maybe it's not updated - knock yourself out, make it better. Maybe the wheels are square: they're serving a purpose, but they're not *really* working.

Overthinking & perfectionism

Don't overthink it. This is just good life advice, nonetheless CS advice. I think a lot of us are heavy thinkers and think about worst case behavior (which is good practice for finding edge cases), and we're always overthinking things.

If you're a perfectionist like me, it's also hard to let things go. I feel like I can always do a better job, but at some point you've got to finish the assignment or ship the product.

Do a good job and a passable job, and then maybe you'll get a second chance to get it perfect in the future. Codebases change and technologies change so quickly that your code might be obsolete; code was only ever just code, you can always write more of it. No need to freak out about it.

If you're unable to stop overthinking or being a perfectionist about things, that's okay! It shows how much you care about whatever it is you're doing. But maybe talk to someone else to get their perspective on things and move forward from there.

Symptoms and causes

Treat the cause, not the symptom. This is a classic software engineering saying. Just like with an illness, if you treat the symptom, it might get better temporarily, but the illness will probably come back.

To be fair, sometimes you *can't* treat the cause. Maybe someone made an architectural decision that prevents it and would cost literally millions of dollars and months of time to change. That's okay: you have to work around it. Hopefully you or your senior engineers are looking out for these

patterns and can weigh the tradeoff of rewriting a codebase for the 5th time versus just sticking with it.

But if you can treat the cause, do it. And do it early before it develops throughout your code. If you have a bad API or even a poorly designed function, and it gets used everywhere, you now need to convince the users to change and you yourself need to do a ton of refactoring [6].

---

[1] Sometimes I feel like the art of computer science is being able to guess the acronym of whatever obscure thing you're using. I considered looking through all of my college notes and making a book of every acronym I've ever seen in CS, and then realized that I value my sanity. However, if someone wants to do this, let me know.

[2] Based on a true, incredibly embarrassing story. Don't ask.

[3] Credit to Ishan Bhanuka on Medium

[4] I got a pretty bad grade in physics, that's about as clear as I can be. :)

[5] I think Sumner or someone else at ACM originally gave this quote.

[6] Hence why we have the aforementioned Refactoring Guru.

Design Patterns

Design patterns might seem like a weird thing to put in a book with so much soft skill and people-based advice. I agree! But if you're just starting out in computer science, I think it's helpful to know that these things exist and are out there (and are made to help you and your team!).

Refactoring Guru is a wonderful resource with both free and paid resources that can help you learn more about design patterns (and refactoring, which is another great skill to have).

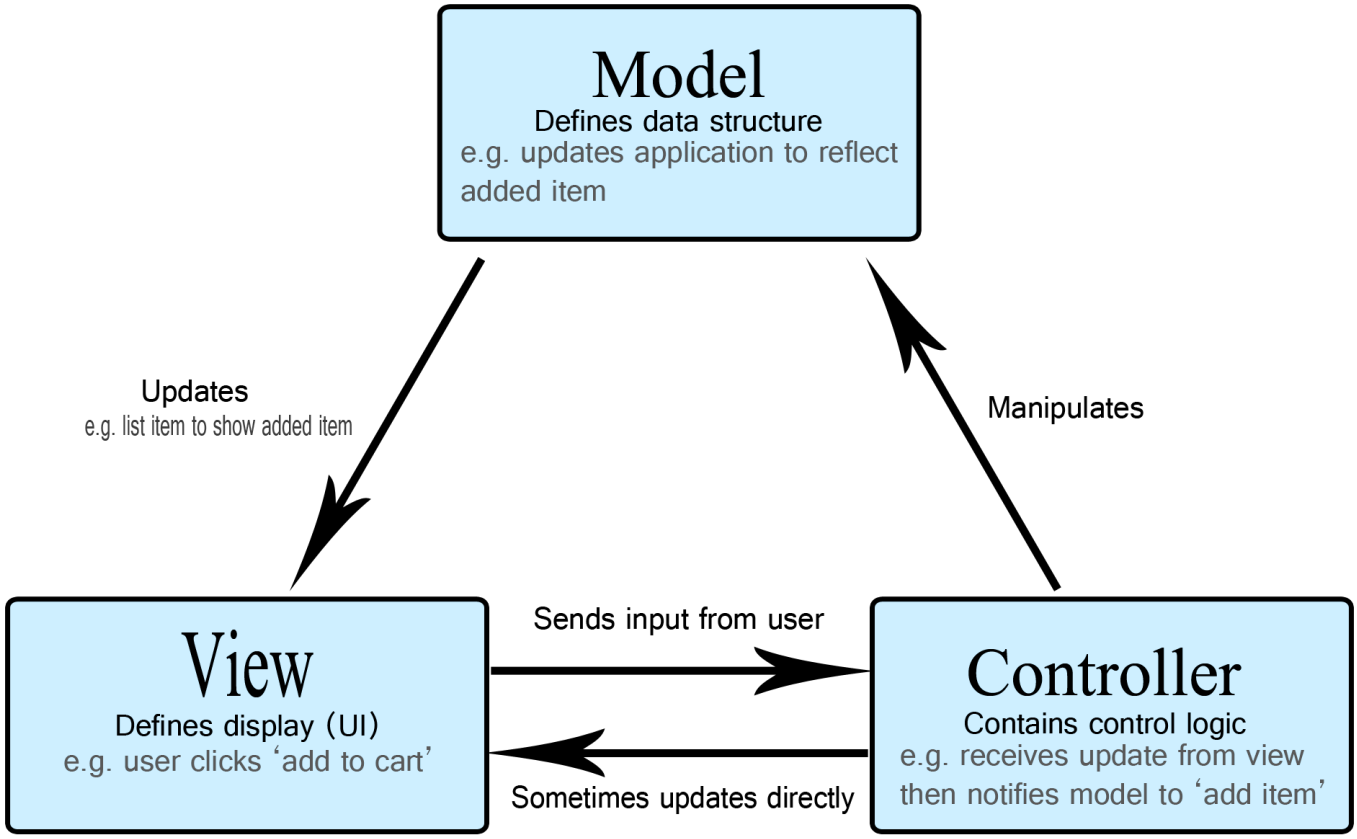
Design patterns are great because they can improve our ability to maintain code, explain it, and understand it. It might be a little more advanced, so don't worry about learning them at the introductory level, but as you get more intermediate and advanced in the field, give them a try.

Most languages also have books like Effective Java which explain patterns, paradigms, and other useful things that help you get to that next level of understanding [1].

MVC Architecture

Model-View-Controller (MVC) is an architectural sort of design pattern. The **Model** manages the data and logic; it may (and probably does) hook up with the database. The **View** manages the "frontend" logic; whatever is displayed to the user. The **Controller** lives between the two, routing the model data to the view.

MDN has a great article on this and provides this image:



There are other patterns such as MVVM (Model-View-ViewModel), and others. Altogether, these are great things to consider when you're first designing a website or service.

And again, I'm just trying to give a five second overview. Do some more reading on it! :)

Builder Pattern

One design pattern that might be useful as an example is the Builder Pattern. A friend and I often joke that this is the best design pattern and we should even do nested builders (which he *has* done). Either way, it's normally one of the first ones people learn.

The Builder pattern is great if you need to create objects quickly. You leverage returning the object itself until it's constructed (in which case you call `build()`) and chain calls together to make an object.

Here's an old Java example from my Minecraft days [2]; this code is meant to construct an item programmatically that could be given to the player:

```
public class ItemBuilder {

    private ItemStack itemStack;
    private ItemMeta itemMeta;
    private Material material;
    private int amount;

    public ItemBuilder(Material material, int amount) {
        this.material = material;
        this.amount = amount;

        this.itemStack = new ItemStack(material, amount);
        this.itemMeta = itemStack.getItemMeta();
    }

    public ItemBuilder setDisplayName(String name) {
        itemMeta.setDisplayName(name);
        return this;
    }

/* ... other methods, excluded for brevity ... */

    public ItemStack build() {
        itemStack.setItemMeta(itemMeta);
        return itemStack;
    }
}

// new class calling it:
ItemStack item = new ItemBuilder(Material.DIAMOND_SWORD, 1).setDisplayName("Andúril'
```

Instead of something like:

```
ItemStack item = new ItemStack(Material.DIAMOND_SWORD);
ItemMeta meta = item.getItemMeta();
meta.setDisplayName("Andúril");
```

...while it's only three lines, this gets old *quickly*. Especially with things I omitted, like item enchantments or lore (text shown when you hover over the item).

Boom! Intro to design patterns. There's more to learn, so check out the links to Refactoring Guru and Effective Java above. Design patterns are (mostly) language agnostic too, so don't worry if you're not into Java.

---

[1] I asked for this book for my birthday one time. My parents were slightly confused and perhaps a twinge disappointed. :)

[2] This code is over 5 years old, and the APIs have changed *massively* since then. I'm including this example because it should be pretty clear. This is *not* how you do this anymore.

API Design

For beginners: REST and CRUD are great acronyms describing how we design APIs and web operations. API design is one skill I want to get better at, but here are a few things I've noticed anyways:

Design as if you're the one using it: Often we design APIs with the idea that we need to handle every edge case and provide *all* of the options to users.

Remember **YAGNI**: you ain't gonna need it. If you don't need it, don't add it. Even if hardly anyone ever uses it, if you have one person using it, they're going to be up in arms once you delete or change it.

Keep the useful parts of your APIs and frameworks in, and if you're short on inspiration - imagine how you as a developer would *want* to call your own API.

Beware backwards compatibility: As mentioned, people will probably be up in arms once you change your APIs. This is also why we get lovely URLs chained together with /v1/ and /v2/ and what

not, so that we can maintain backwards compatibility.

It's okay to maintain backwards compatibility, and necessary if that's what you're doing, but eventually you should get your users up-to-date if possible [1].

Document it well: There's no excuse not to document APIs well with new things like OpenAPI or even GPT. Heck, with the rise of Cloudflare Workers, APIs as a whole are changing.

But a bit of documentation never hurt. This goes for everything in your project or product.

---

[1] You should keep everything you use up-to-date, but obviously this is a money and time drain and can't always be done. Make an effort though.

Gitiquette

What do you get when you combine "git" and "etiquette"? Gitiquette! Don't worry, my former boss and coworkers were unimpressed by that joke too.

However, I think it is worth it to mention a couple of conventions in the realm of Git and GitHub that help everyone out. They're not required, and they are subjective, but I think they can help you communicate your code.

Commits

Commit frequently: This was a constant battle with a coworker. He would do the classic entire user story in one commit and push. Maybe in some respects this is okay, especially if you're in the very early stages of a project, but once you're making large and intricate changes, it helps to commit frequently.

The idea here is that you're able to rollback to any state, should something go wrong. For instance, let's say we have just one commit: `implement entire product page`

versus the following commits (in order from oldest to most recent): `implement product lists` `implement product detail view` `implement cart functionality` `implement buy page` `reimplement buy page`

This is an oversimplified example, but it holds up. What if we broke something in the `reimplement buy page` commit and we somehow can't figure it out? Let's revert to the last commit: `implement buy page`; this way, we've got solid foundations to work from and can revert should things go horribly wrong. Whereas with the first commit, we now have to sift through the weeds with everything and just work with what we have.

This is also great because once your branch of frequent commits gets merged in, you get a more linear and understandable history in the commit log.

Meaningful commit messages: This is my silly and meaningless pet peeve. I think commit messages are pretty easy and usually people mess them up [1].

At least in the traditional sense (follow your organization's guide if it's different), commit messages use this "command" tone, as if you're commanding someone to do the changes you did. For instance, you would say `Add sign out button to navbar` instead of `added a signout button to the navbar` or `I added a signout button to the navigation`.

Your commits shouldn't be a narrative, and should be concise and as short as possible. If you follow the last guideline of committing frequently, the commit names should "write themselves" - what you did is exactly what the commit name should be.

One friend and I found out that you can put emojis (and most any unicode) in commit messages, but *it doesn't mean you should!* I've seen some repositories where they might categorize features and bug fixes with certain emojis, but that seems like the only time it should happen.

Issues & Pull Requests

Taking an issue or a ticket is like raising a child: you have to see it through and help it grow and put the work in. You shouldn't abandon it. Work hard at it and do the best job you can with the issue; these are the small moments that add up to good software over time.

Give a brief overview of what happened: In the description of a pull request, I like to type out some bullet points of what I did. It helps to keep me on track but the person reviewing my code can also look the description at a glance and find what they're looking for in the code.

Don't delete early. I am so often hitting "resolve conversation" early, which is a bad idea. Get verbal or textual confirmation that your reviewer understands why you're resolving the

conversation, or else they will be looking for where their conversation went *and* still needing to talk to you about it.

Follow through with your issues. **Talk to your stakeholders** or boss or whoever it is to truly understand what the ticket or issue is about. Don't just work in an isolated environment, see the full picture of where your fix or feature is in the entire system.

---

[1] Please don't look at the commit messages for this repository. Do as I say, not as I do. ;)

Accessibility

It's shocking how little people know about accessibility. Most people who know seem to know someone with a disability or have one themselves. I learned a lot about this in Human-Computer Interaction, which is a great introduction to overall design thinking but also had emphasis on accessible design.

You should want to accommodate for accessibility without any external motivators, but a lot of these things are required by law now anyways.

For instance, have you heard of the following things?

- A skip to main content link
- Screen readers
- The WCAG guidelines
- Heading level accessibility
- and more

..if you said no, you probably wouldn't consider them when building up your website either [1]. And you definitely **should** consider those things.

The argument here, if any, is that there's never enough time or money to do it. Especially in a startup, you're working on getting a product out of the door and in consumers' hands as soon as possible. I won't tell you how to run your startup or company, but I will encourage you to think eventually (and hopefully sooner rather than later) about how accessible your products are. Designing for a wider audience of people should be appealing.

I'm not pretending to be some expert on accessibility and I hope to learn much more and share and teach much more as I advance in my career. I just hope I've convinced you to care (or even learn just a bit) about accessibility too. I hope not, but some day you or I may be affected by something, and I would hope that the software and hardware we use can support our needs.

---

[1] If you *have* heard of these things, great! Spread the word.

Users

There is an abundance of knowledge as well as classes at universities you could take on this, but let me throw out some ideas.

Testing methods are a great idea when you want to see how users *actually* use your product (which you should want, and you should want to do **early** instead of once the product is done). Here's a good article on introductory usability testing and another guide on research methods.

Cognitive walkthroughs are another great tool for looking at the usability of your design or your system.

Even if your job title doesn't include the words "user interface" (UI) or "user experience" (UX), you should still be thinking of your end user.

Who are they? (e.g. make some personas) What do they like and dislike? What situations do they get into when using your product? What are they trying to achieve? Your product solves a problem; is the user able to solve their problem with your product?

These are all great questions to answer and to **test**. You can't think through all of these; at some point, you need to see the reality of how your design is being used.

We spoke about it a bit already, but it's worth mentioning that part of designing for your users is designing for *everyone* - see *Accessibility* section.

Society

Designing for society and thinking about morality is important in any profession; especially engineering. If you went to Mines, classes like Nature & Human Values (NHV), however much hated

they are by the public, should prep you for these sort of questions. But I urge you to always be thinking about the ethics.

I know a lot of people who work for defense contractors and I really like and respect them as people. I also know people who have left those scenarios, and I respect them too. It's important, especially in those situations, to be thinking about the full picture. Are you making a product that's going to kill people and be used for mass destruction? Some of my friends view it as *defense* for the security of our country. Some left because they felt it was *offense*. I see both perspectives. The point I want to make here is that you should be thinking about it and making your own decisions.

If you make some sort of AI or algorithm, what implications does it have on people's jobs and livelihoods? Do you see the full picture of what you're designing, or are you mindlessly working on a tiny part of the system?

Catalyst, a novel relating to Star Wars: Rogue One is actually a pretty good example of this. The writer, James Luceno, accurately portrays how scientists and engineers on the Death Star project weren't aware of the bigger picture of what they were making. They just thought it was an important, tiny piece of science. But it was actually being used to make a weapon that could remove entire planets and commit genocide on insanely large amounts of people. Although this is fiction, there's some realism in this. It's analogous to The Nazi engineers: reflections on technological ethics in hell by Eric Katz, which was taught in NHV, and spoke about how those engineers probably *did* know what they were doing and still did it anyways.

The Social Dilemma is another great documentary involving engineering ethics, where former executives of big social media companies talk about their invention. They mention that **you and your attention are the products being sold**. The way these companies make money is by selling your data to other companies and selling products to you tailored to your exact interests (because they're tracking your exact interests).

While this is an unfortunate reality of the business model of the internet today, you should consider the effect that this has on people. Are you writing software that is going to learn people's preferences and then sell that data? Is this software going to cause addiction and more societal problems than the good it does? All questions to be asking.

Just like in the accessibility section, you should want to design for society without any external motivators, but it's the law now with things like the General Data Protection Regulation (GDPR). Even if you don't live in the EU, chances are you'll have customers in the EU, and be legally bound

to laws regarding their data. Even so, many states in the US as well as many countries are beginning to write laws like this.

Be aware. Don't be ignorant to the effects your design and your engineering has on the world.

Jobs

People panic *a lot* about jobs, especially with the ongoing AI hype. Perhaps the panic is warranted, but if you believe in yourself and *your degree* (assuming you're pursuing one), you will be just fine.

Just know that you may not always get the heavy-hitter companies first: Google, Amazon, Apple, etc. They might take a bit more experience and a bit more networking, **but that's okay!**

The job search

Please, for the love, do not search up internships or jobs based on salary. The money will come - you work in a lucrative field already. Search for companies that excite you to work at and align with your values. Realistically, you may not always get these jobs, but you can work toward them and you should start with them first.

> Happiness is in the doing, not the result. [1]

Your happiness should be in the every day things: what you're coding at your job, the overall big picture of what you're working on, and the people you're working with. Not your salary or the names on your resume. Find a company with a culture that suits you. Glassdoor is a good website for seeing what people generally think of working at the company you're looking at.

One exception: **with internships**, go for quantity over quality. In a competitive and arguably oversaturated market, experience anywhere is better than no experience at all. Apply around and see what you can get, and then evaluate your options from there. In my internship search for Summer '24, I applied to around **140 companies**, didn't hear back from about **50 companies**, got rejected from over **80 companies**, and got **two offers**, one of which was partially from networking [2].

Network, network, network. Companies want to hire good people, and if you know good people, they will hopefully know you and recommend you [3]. This trumps everything else. If you network, it's an immediate "in" at the company which gets you past a lot of the initial interview shenanigans or through them even if you do poorly.

I know a lot of young people in similar situations to me think networking is cheating or morally wrong; e.g. they want to get a job on their own merits. I can see that perspective, but ultimately you should take what you can get. A lot of people network [4]; it's normal! Assuming you then get hired, you've already got a friend there who is looking out for you, which is good!

Decisions

A decision is hard because you don't have all of the information to make it. Deciding to move somewhere for a job is super tough. Thinking if the company culture is right for your personality is super tough. Figuring out if the company aligns with your career and personal ambitions is super tough. But you're not locked in forever.

The 40-70 rule states that "you need between 40 and 70 percent of the total information to make a decision." This is a great way of summing up my mantra above: decisions are hard, and often you don't have enough time to get all of the information, so do your best to get most of it. At least 40%, so you don't cut yourself short, and at most 70%, because you probably don't have time to get more and it might make the decision harder.

Trust your instincts and intuition: if it doesn't work out, it doesn't work out, and that's fine! There are plenty more fish in the sea. If it *does* work out, great!

---

[1] *Ikigai: The Japanese Secret to a Long and Happy Life* by Héctor García and Francesc Mirales [Amazon]

[2] I don't give these numbers to be depressing. In fact, a lot of friends I send this to actually seem reassured by my plight. So I'm sharing it here too. :)

[3] ..and get a sweet bonus if you're hired; it's a win-win-win for you, the people who you networked with, and the company.

[4] Not to say that something is OK because everyone does it; however, networking is pretty standard.

Interviews

Interviews are a two-way street. In an ideal world, it doesn't matter too much about how you perform in the interview. Interviews for the interviewers to see if you would fit in their teams and cultures both skill and personality wise; you also should be doing the same.

I've spoken with a lot of HR people and tech directors alike throughout hiring processes,and they all seem to agree: your attitude is more important than how good you are at coding. It's how you show the process of how you solve the problem, react to adversity, and show your personality in general.

You might get asked things like:

- What is a time when you made a mistake and how did you fix it?
- Let's say you're behind on a project. How do you rectify the situation?
- Name one time you were challenged and how you overcame the challenge.

I always hate these questions. It's hard to think on the spot (maybe I just need to prep more.. guilty!) and for instance, I rarely find myself behind on projects because I try to keep ahead [1]. But the point is: just try to answer these honestly. You should get to the point (STAR method), but I often find these questions as opportunities to get a bit of personality across. Sprinkle in something you found funny or interesting about the project you were challenged on. Tell them what technology you learned from a mistake.

Once you make it through the sea of unappealing questions, it's your time to shine.

You could ask things like:

- What's the most challenging thing you're working on? Do you find the problems you're working on interesting?
- How do you like your managers? Do they notice your hard work?
- What's your favorite and least favorite part about the work you do?

I like these questions because they put you in a closer position to what it would actually be like at the job. They help you imagine yourself in that exact role, which is good for you to find out if you want to be there; and it shows good interest to the interviewer.

I would also note that some of these questions could have negative responses. We're not trying to goad bad answers out of people (and *rarely, if ever* will you hear any interviewer say *anything* bad about the company), but you do get honest answers. I've heard things like "I don't like working with technology X or edge case Y," which is really insightful to know going into the job. ...and if you don't get honest answers, or answers feel forced and practiced in advance, do you really want to work there?

Also, there is a slightly different set of questions I would ask for a non-technical or human resources type interview, but still valuable:

- What's the culture like?
- What's the room for career growth like at _____?
- What do successful people in this role look like?

These questions show you're genuinely interested in working at the company and are trying to imagine what your life would be like there – which are things you *should* be genuinely interested in.

All in all, you could read a hundred interview books and blog posts, watch a hundred interview prep YouTube videos, do LeetCode problems until you're insane [2], and it will *maybe probably somewhat mostly* prepare you, right?

But when it comes down to it, **just be yourself**. Get your personality across, and **interview the company** as much as they're interviewing you.

---

[1] One time I was super vague and sort of made stuff up as an answer to this question. The big tech senior director interviewing me fully called me out and was like "yeah, it sounds like you didn't really actually have that experience." Lesson learned: Be honest! And on a related note, I ended up not choosing that job for a few reasons, but one factor was that I didn't feel super respected from him nor did I feel a lot of interest.

[2] In the rare event that a company is reading this: Please stop! LeetCode, HackerRank, CodeSignals, are such a joke. When am I ever going to implement Tetris (true story - there's a similar question) at my job? When will I ever implement an RB-tree traversal? I think HackerRank and CodeSignal are beginning to do this, but measure me on what I actually will be doing - writing APIs, database queries, designing frontends, etc. There's a way to show applicants actually know how to code without gamifying it.

At the job

Great, you got the job. The fun doesn't end there though: you can still be proactive and drive your career and life in the direction you want to. Here are some thoughts on the matter in no particular order:

Schedule regular meetings with your boss, assuming they don't already schedule meetings with you. This was one of the greatest strengths of my old boss: he genuinely wanted to know how I was doing often and would truly listen and try to change things if I felt I wasn't doing important work or moving where I wanted to in the job. This is also a great chance to get to know your boss more depending on how extroverted they are.

Be nice to people: okay, stick with me here. You should be nice all of the time, but every once and awhile, it doesn't hurt to be really nice. Do something nice for a coworker you like (or dislike); buy breakfast burritos for a morning meeting, or pay for someone's lunch [1]. I'm not saying I was or am stellar at this, but I made an effort at least a few times. One coworker and good friend made consistently thoughtful efforts, and everyone in the office came to like him and he quickly got more involved in office events [2].

Get to know people: I sound like a broken record, but this is really important too. Get to know people and their motivations: they're working toward something the same as you, whether it's a promotion or simply something for a loved one. Remembering things about people goes a long way, especially in the never ending churn of standups where your team might try to talk about something other than work.

Keep learning: everything we've spoken about applies. Have a growth mindset; you don't need to be flawless all of the time, you can and will make mistakes. Learn from them, and keep learning! You don't know where and when some of the things you'll learn will pop up again; sometimes they won't show up at all, and sometimes they will be the key to your next big project.

---

[1] One of my coworkers was so stubborn about this and would not stop trying to pay me back, so I hid a $20 bill in his notebook that he found months later after the internship had ended. I win! *(If you're reading this, please don't pay me back.)*

[2] Don't do it for the purpose of gaining favors or fame; people can tell when it's fake. I'm looking at you, Dwight Schrute. I'm just saying to be nice and you will probably enjoy your job and the people around you more, and the people around you will enjoy you more.

Parting Thoughts

If I had to boil down my advice to just a short sentence or two, I would say to be open minded and self reflective. And truly, *genuinely*, listen to people and care about them.

I started programming on November 23rd, 2015 (I know because I had bought an online course that day). I'm coming up on ten years of coding, and I am by no means an expert at everything. I'm definitely pretty good at some things, but I'm still learning a lot of things too. I'm proud of where I came from, just making fun things for my friends and I in Minecraft, to following my dreams of going to Mines and studying computer science.

I have no idea where my career is going to take me. And, honestly, I don't know how much I'll code, because I like leadership and management. But I'm open to it all, and I'm confident that I can work with people. I'm going to fail and I'm going to make mistakes, but that's all part of the ride.

I hope you learned a thing or two reading this. Check out my website if you want to get in touch. If there's anything you disagree with in this book or would like me to add, open a GitHub issue or get in touch somehow and I'd be open to hearing your thoughts!

Thanks for listening.

Ethan