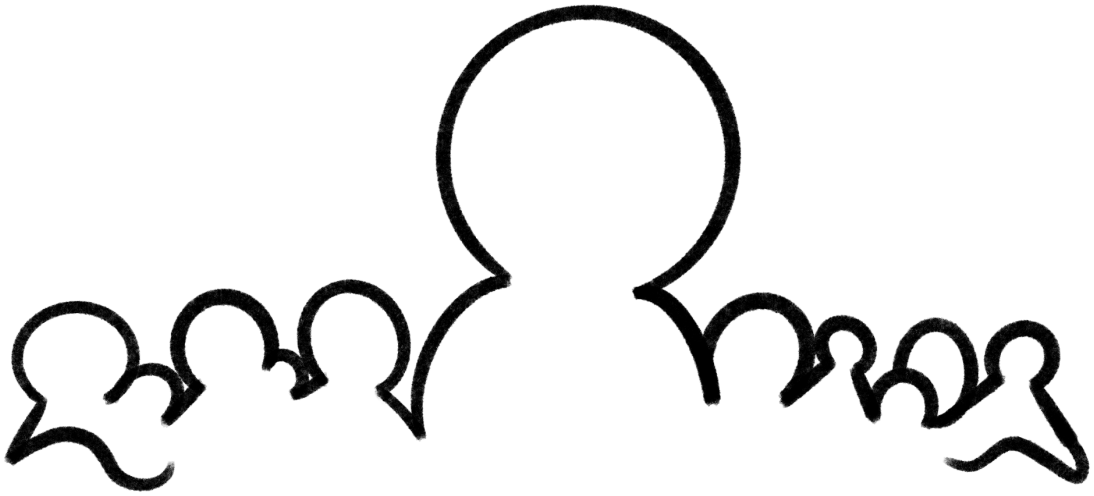
On Becoming the Person Your Team Needs You to Be

Standing out and leading in tech and beyond

by Joe Zobkiw



ISBN: 9798444388570

Prologue

This is a book designed to be read in no particular order. It contains many common sense (and some not so common sense) methods, tips, and personal learnings to consider and use when writing code and going about your day to day work and career in the tech industry. In fact, many of these ideas can be used in other businesses as well. The book also discusses leadership and interactions between teammates and others in your organization.

The thinking in writing this book was to make it more of a meditation on the art of architecting your work cleanly and elegantly - using lessons learned from over 35 years in the software industry. It has morphed to include thoughts not only on that, but also working in a way that makes you a good steward for your current and future team as well as yourself. My hope is that it will give you inspiration and ideas to build and maintain a positive work environment that is helpful and inclusive for everyone you work with while helping you avoid some of the common stressors that come with the territory.

It is programming language agnostic and has no code examples whatsoever. Some terms will be used interchangeably such as “methods” and “functions”, “properties” and “variables”, etc. The idea is more to grasp the concept being presented than any specifics about a particular language or tool.

Some of the ideas presented will be repeated in subtle ways, stated differently, explored from a slightly different angle. This is on purpose. Not everyone learns the same way and some items simply need to be reinforced to show their importance and ultimately become second nature.

I hope you get out of these words at least as much as I've put into them, if not more. Thank you for reading and caring enough to improve your working relationships. Once you've read this book, please share it with your team as well. If everyone is on the same page, your work life will become that much more productive and positive.

Happy (insert today's name here)!

Introduction

I started my career like many do, about as green as the summer grass.

I was a self-taught programmer. There were no code-camps. I more or less fell into software development because it was a creative endeavor that I enjoyed. I became the “expert” in my circle because no one else did it and people were realizing that they needed programs written to help them with various tasks.

Before I knew it, I was interviewing for a job light-years over my head and very much out of my comfort zone. I was hired because the very much genius-level programmer who was the interviewer took a liking to me and saw parallels in our journeys. Both of us went to college for music and were self-taught programmers. Within weeks I was writing code for a living. I owe that guy my career because of an interpersonal connection and the fact that he took a chance on me and my mullet.

As I sat in my office, staring at C, Pascal, and assembler, I quickly realized that the programs I was writing and giving away for free, or charging minimal shareware fees, were basically hacks. I was playing the role of designer, developer, tester, marketer, and sales. But now, I was writing commercial software, to be sold in a box, on a store shelf! Although I like to think I made it through that experience pretty well, I’m sure there was a lot of hand-holding along the way.

This first job, and my subsequent positions, taught me that software development is hard. It also made me realize there are always people more experienced than me that can be asked for help. Eventually I became one of those more experienced people and I made sure to return the favor as new

members joined my teams. One important thing to never forget, however, is there will always be someone you can learn from - regardless of their experience level. If you think you can't learn from anyone, you're simply wrong.

As I rose through the ranks of the various companies I worked for, I always kept this in mind. I also realized that it was important to treat everyone properly, regardless of their position above or below me in the hierarchy. Say hello to the CEO. Ask how their day is going. Say hello to the cleaning staff. Ask how their day is going. Spend quality time with the new team member who is quite possibly doubting their abilities, while trying to prove themselves, while trying to see where they fit into the team, while trying to deal with whatever else is going on in their lives.

This book is as much about technical things as it is people things. By making good decisions along the way, and cultivating relationships, you can ease the potential stress on your current teammates and those down the road. One line of code that you spend an extra few minutes on today might circumvent a bug that would have happened 6 months from now and take a week to find and resolve. You know the stress you can feel when there is a big problem and the business is depending on you to fix it. Imagine if a few of those bugs never happened. You have the ability to make that the case.

Read this book with an open mind. Read into the ideas and thoughts presented here. Ruminate on them. Tweak them to fit your situation. Ultimately, code like you would like others to code and be like you would like others to be.

I wish you the best of luck on your journey.

Foreward

I was hoping someone famous would write this, but alas, it ended up being you…feel free to write something here…or tweet about it. :-)

About the Author

Joe Zobkiw has been working in the software industry since the mid 1980s. Yes, he's probably old enough to be your Dad. That's OK. There is a slight chance that your Dad taught you a few things.

Back in the day, he wrote a highly regarded book on low-level code fragments and code resources for PowerPC. He followed that with a book on Mac OS X advanced development techniques. In those circles he is famous af.

Since that time, he has worked as a software developer writing code for macOS, iOS, the web, and other platforms. Eventually, he found himself drifting into technical and people management. Today he straddles the line of a technical and people manager who still likes to code and pass on what he has learned to anyone who will listen.

Credits

Author: Joe Zobkiw

Art: Grace Zobkiw

Reviewers: Halsey Burgund, Jess Bowers

Dedication

To everyone I've ever worked with who (knowingly or not) helped me formulate the thoughts here. Life is constant learning and I appreciate you.

Table of Contents

If you got this far, you don't need no stinking table of contents. Besides, this book is meant to be read in any order you wish.

You will see some captions at the top of each page that generally categorize the thought expressed on the page. Sometimes these can go more ways than one, but they reflect the first thing that came to my mind as I was writing the thought.

<code> - thoughts about coding and typing things in

(process) - thoughts about processes and working well together

[self] - thoughts about taking care of or managing yourself

{people} - thoughts about working with others or helping your team

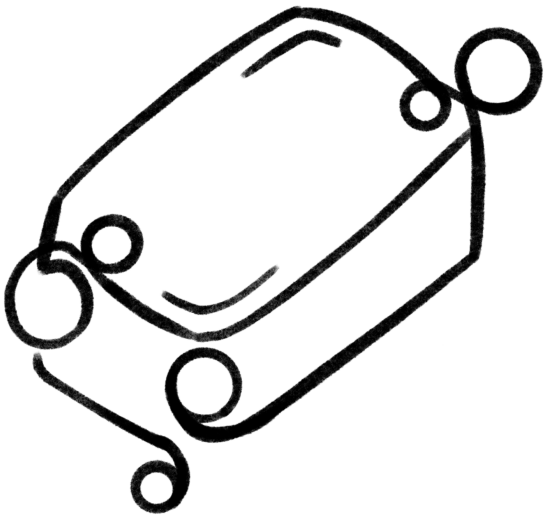
So flip to a random page or read in order - it's up to you. Choose your own adventure and enjoy!

<code>

Code cleanly from the start

At the moment you begin thinking about your project, promise yourself and your team that you will code cleanly. Do not cut corners. Think about the code you are writing as you write it and think if there is a better, safer way, as the lines emerge. Ensure your method signatures make sense. Name things what they are. Don't try to get fancy and tricky. If something represents a name, name it `name`. Comment your code unless it is absolutely obvious what it does - especially if it makes assumptions about data, state, or other things happening elsewhere in the program or outside the program. What may be obvious to you now may not be obvious to someone else, or your future self, later.

`#cleancode #rulenumberone`



<code>

Test your code

I can not tell you how many times I've reviewed code that either didn't run or (believe it or not) didn't even compile. And I'm not talking about after other commits have occurred. I'm talking about on the branch that was submitted for review. This is simply unacceptable.

When you write code, test all the paths it can take. Step through your code in the debugger and look at the values of each variable as they pass. This can often uncover subtle bugs that you might not think about if you just type it in and assume it works. You know what happens when you assume.

When reviewing code, be sure to check all of the `else` statements. Many developers ignore this code path even though they wrote it. They assume returning `nil` or `false` will "just work" but there are plenty of times where it simply won't.

Would *you* rather catch these problems or have your customers do it for you?

`#testyourcode`

[self]

What's your uptime?

The uptime of your computer can be impressive. If you're not familiar with uptime, open a terminal window and type `uptime`. 42 days since last restart? Nice! Your computer is working hard! However, your *personal* uptime is a little more sensitive than your computer.

Your eyes need their rest. Stop staring at the screen every now and then and look out a window or across the room. Let your eyes relax and focus on something at a distance instead of that screen smack-dab in front of your face.

Your mind needs its rest. Take a break and go for a walk. Let your mind be clear or (if you must) gently think about what you want to accomplish the rest of the day (or tomorrow) while you stroll, taking in some fresh air and sunshine.

Your body needs sleep. Listen to what your body tells you. Some people work better in the morning, some late at night, but I've never known anyone to work better 24 hours a day / 7 days a week. Make sure to get regular and sufficient rest.

Caffeine is wonderful but it's not a substitute for taking care of yourself.

`#uptime`

<code>

Avoid third-party code

Every once in awhile there will be a third-party framework or library that you need to use to get your work done. This is unavoidable and expected. However, you should make sure that they are well-supported, have a strong community around them, are popular, and well-respected before you even *think* about using them.

Too many times someone will write a useful bit of code that might save you hours or days - but what happens when that person has other responsibilities in life and the code they've provided becomes neglected? You may find that with a new OS update their code no longer works. Other users post issues about it (or you're the first one to discover and debug the problem if you're really unlucky!) But what if the original developer can not fix it for any of a number of reasons? Guess who fixes it? Either you, or no one.

All third party code adds risk to your project - be prepared to assume that risk, even at the most inopportune time - like release day.

`#futuretechdebt`

Name things what they are

I can't stress enough how important it is to name things what they are. If you have a variable that represents a field in a backend database named `total_cart_price` then why would you name it anything else in code? Naming it `total_cart` doesn't tell you the data type, a dollar amount. Naming it `cart_price` doesn't tell you that it represents the total. Naming it `tcp` is just asking for confusion down the road. Transmission Control Protocol? What is this doing here?

Imagine if there is another database field named `sub_cart_price` - things could get even more confusing unless you explicitly name your local variables the same as those on the remote system (or at least the same as those in the data you are reading from the remote system.) Yes, I realize you can't *always* name things the *same* but you can usually get the alphanumeric portion pretty darn close. Do your best.

In this case, `total_cart_price` or `totalCartPrice` are really the only sane choices. Don't abbreviate. Don't add words. Don't delete words. Name things what they are or what they represent so it is excruciatingly clear to anyone reviewing the code. And remember, if the database field changes in any way, update the code to reflect the reality of what it represents.

`#whatsinaname`

<code>

Every line of code you write is potential future technical debt

Are you using an out of date API that isn't fully deprecated yet but there is a newer replacement that is the better choice? Why are you using the out of date one then? Because it's easier? Because you've done it before? Because the new API would require learning something new or maybe more refactoring or typing? You get paid to type. That's what you do. You're a glorified typist. Take the time. Do the typing. Someday that deprecation will be complete and you'll be forced to anyway. Why procrastinate the inevitable?

`#futuretechdebt #dontprocrastinate`

Step through your code before committing

If you ever submit code that doesn't build or crashes immediately with expected (or unexpected) input you will quickly become the most despised member of your team. The extra few minutes that you could have spent checking that your code worked, before being committed, could have saved other developers the hassle of finding your bugs for you. Not to mention the turn-around time that was wasted. Take and hold responsibility for your work - don't push it off on other people when they least expect it and likely have deadlines of their own. Be your own best reviewer.

#takeresponsibility

(process)

Have someone review your code

Make sure to have someone other than yourself review your code. In many cases it may not matter if this person is more or less experienced than you. I mean, I wouldn't have someone who just learned to code last month be the only one to review something critical, but they can certainly be a part of the process so they can learn.

You're not infallible. This is a truth. You may be the most senior developer on your team, but even in that scenario, the team as a whole is smarter than you alone. Use the resources at your disposal to help make the project better.

In the event you work alone - a one-person-band as they say - find someone, a friend maybe, to give you a code review from time to time. This will keep you honest and also improve your product. For example, I'm one person writing this book, but I will have multiple friends and colleagues read it before you do to make sure it makes sense and doesn't contain any typos. Oh wait…you're reading it now? Oops!

`#reviewssave`

Design using native components

When given the choice, if a solid, native component exists then you should use it over any third-party choice. Native components are going to give you the best chance at future compatibility in most cases. A third party choice merely means you now have to depend on someone else for your app to work in the future. Usually this will be a developer you don't control or have any influence over other than begging for bug fixes via bug reports because your team doesn't have the bandwidth to deal with problems that arise. Is that puffy button really worth the extra effort when the native one not only looks good but is what users expect?

#choosewisely

Use the features provided by the OS before searching for alternatives

Don't reinvent the wheel. Whatever OS you are designing and building for likely comes with tons of features. Many of which are well-tested and stable. Always (at least) start with those things as much as possible. If you run up against limitations then by all means find solutions elsewhere or write your own custom implementation. The odds are you will need to do that at some point here or there but the majority of the time, the built-in capabilities will more than suffice, and make your code more stable as a result. Focus first on the functionality of your application, make sure it does what it is supposed to do. You can always "enhance" it later.

#focusfirst

<code>

Don’t copy and paste example code

We all know everyone searches for code and solutions on the web. This is fine. This is how we learn. You don’t have to write every bit of code from scratch. However, you should at least type it in manually as opposed to copy and pasting swaths of code, assuming it does what the top-rated comment says it does.

The best way I’ve learned to help grok found code like this is to simply type it in yourself. I don’t care how long it is, you’re still saving time from writing it from scratch, assuming it actually works. Look at the code in your browser while you type it in to your editor. Understand what each line is doing as if you’re writing it from scratch. You may find a bug. You may want to rename variables it uses. You may want to adjust the code stylistically. Regardless, by putting it under your fingers you will gain a deeper understanding of the code and its purpose and it just might save you a bug being found 3 months from now.

`#typedontpaste`

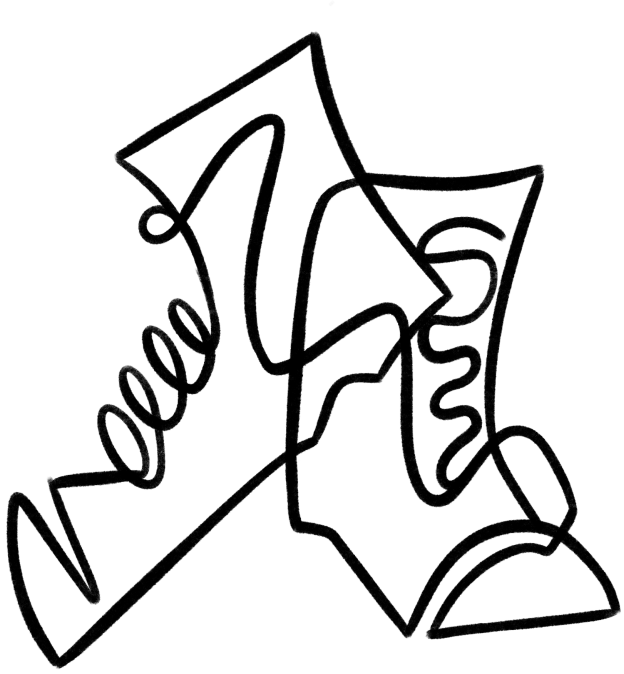
[self]

Put yourself in your coworkers shoes when looking at the code you just wrote

Review your code as if someone else wrote it before you submit it for review by a team member. Is it making assumptions? Is it going to break something? Do all the `else` statements work like you expect? Does it recover from bogus input gracefully?

This is a theme in this book that you may be noticing, taking personal responsibility for your actions before thrusting them on your team and ultimately your customers. Tiny tweaks to your personal process can ultimately make your project that much more solid, stable, and less likely to contain bugs. Imagine if your entire team coded defensively in this way? How much better would things be?

`#personalresponsibility`



(process)

Use source control

Source control is a must. Even if you are only writing a small shell script or simple program with just a few files. Even if you are a single developer working alone on your project. Knowing what changed, when, and under what circumstances, can be invaluable when tracking down the inevitable future problems. In the event you have (or build) a larger team around your project you will need a process in place to manage all of your source code. Source control is the foundation of any process you will build for this purpose.

I've worked on teams where it was a free-for-all, people throwing caution to the wind and committing code to primary branches, sometimes stomping on other changes. It's a nightmare. Use source control properly. Use pull requests. Develop a process for code reviews. Stick with it. Don't make exceptions unless in *extreme* circumstances and there is no other way out. Make sure to define the word *extreme* in a way that makes sure you don't make exceptions every day.

If you are not familiar with source control and pull requests, it may seem daunting to adopt this type of workflow. Trust me, take the daunt for a few days and get used to it. You do not want to live in the alternate world.

If you are already using a process like this you may think to yourself "How could anyone NOT be using source control?" They're out there…look for the people with tears in their eyes.

`#commit`

(process)

Review your internal processes regularly

No matter how smooth your internal processes seem to be working, you should take a good, hard look at them at least once a year, sometimes more often depending on the process. There may not be room for improvement today, but it is very possible that with changes to your business environment, a slight tweak (or complete overhaul) to a process may be needed to keep things running smoothly. Don't be afraid of making things better - sometimes those things that aren't broke can still use a little oil.

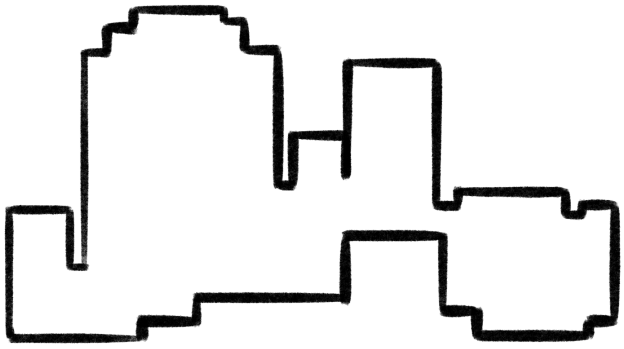
#maintainanceavoidsheadaches

Problems can be big (architecture)

When you are starting a new project, one of the things you should be thinking about is the overall architecture that you are about to implement. Some people like to just start coding, however, it's easy to find yourself a few weeks in with things just not working properly together as you build more and more individual pieces.

Take the time up front to think about what you're about to create. Sleep on it. Take walks. Map it out in your head for a week or month or more! Before you ever commit pixels to phosphor, make sure it all makes sense in your head (or on paper) and that you have left no major unknowns to be figured out later - they may come back to bite you.

`#thinkfirsttypelater`



<code>

Problems can be small (a single line of code)

The simplest line of code can be the devil.

It sure does seem easy enough to understand, but called at the wrong time or in the wrong order, due to a subtle interaction between surrounding code and code elsewhere in your application, and this simple line could take days to track down and then figure out *why* it is causing the problems it is.

Be aware as you write each line, what impact it may have elsewhere, what expectations it has, and what it assumes about the code around it.

`#youcanneverbetoocareful`

{people}

Don’t hire someone who you wouldn’t be willing to pay with your own money

Whenever I’m interviewing, I keep this in mind. My money seems more valuable to me than the money my employer has. This is because it’s mine. I might not even want to spend $5 on a cup of coffee unless I know it’s going to be really good. Imagine if you had to pay the person you are interviewing out of your own pocket.

If you don’t leave the interview very excited about the candidate, keep looking for someone else. You should want to (not necessarily be able to) give them your own money - they should be *that* much of a good fit for your team and organization.

`#worthit`

{people}

Everyone has their own sh*t

Just like you have things going on outside of work (if you don't, then you need more help than this book can offer) so do your team members, direct reports, managers, etc. Sometimes those things are not fun. Someone may be dealing with a serious medical condition of their own or in their family. Someone may be going through a breakup. Someone may have other family issues that are taking time and energy to handle.

Sometimes the stress from these types of issues can leak into workplace performance. Although it is a fine line about how much you can legally ask someone, be aware, that if someone is having trouble, they may have other factors at home that are influencing them.

You can always gently guide them to resources that your company provides to help if they need it. Give them the opportunity to manage the balance between their personal concerns and work deadlines. They may not know where to turn and your simpler gesture of guidance may mean all the difference.

#behelpful

{people}

Hire people that know how to figure things out

Once you get your first job, the rest is all figuring things out. You absolutely will not know everything you need to know for your entire career from an 8 week JavaScript boot camp or 4 years in college. If you think you do, you're confused.

It's all about taking the information in front of you, making educated guesses towards a solution, and then figuring out how to do it. You will *always* be searching for answers and learning new things as a result. You want people on your team that have this skill as well. Believe it or not, many people just guess, all day every day, and end up wasting a lot of their own time as well as yours. Find people who know how to find the proper solution without the guesswork.

`#hireendlesslearners`

[self]

Technology is only as compassionate as the people who created it. Make your work more forgiving and less stringent when you can.

If not currently, someday, you may be creating something that makes a huge difference in someone's life. Think about it. All you're doing is typing things in, but you know what to type and how it all works together to create a thing that can literally change the way people live their lives. It might be a game to help someone relax. It might be a tool to help them track their health. It might be an app to help them be more productive in life. You have the ability to take your attitude, your kindness, your compassion, and vision, and send it out to the world to enjoy and gain from. You're not just typing things in, if you want, you have the power to make a difference.

#followyourcalling

(process)

If someone is repeatedly asking about a particular thing, fix your process

I noticed once that someone was consistently asking about the latest client as they were onboarded to a product. What this means, is that the information this person was seeking was not readily available.

Think of their experience, they see a new client is using the software but they have no idea who they are or where they came from, so they feel a sense of frustration and then have to go ask in a chat room"who is this person?"

How could this be resolved? Pretty easily! Communication!

Set up a process so that when a new client is onboarded, some sort of client profile page is generated and displayed in a customer management system or even simply in a shared Google doc. This way, employees learn that if they see a customer they don't recognize, all they have to do is go there and find out everything they need to know about them. How they made first contact with the company, who they are, where they are, what industry they work in, their contact information, etc. Whatever is relevant can be added here.

This type of simple process not only saves frustration but also time since others who know the answer don't have to answer the question ever again. The information is always freely available to anyone who might need it.

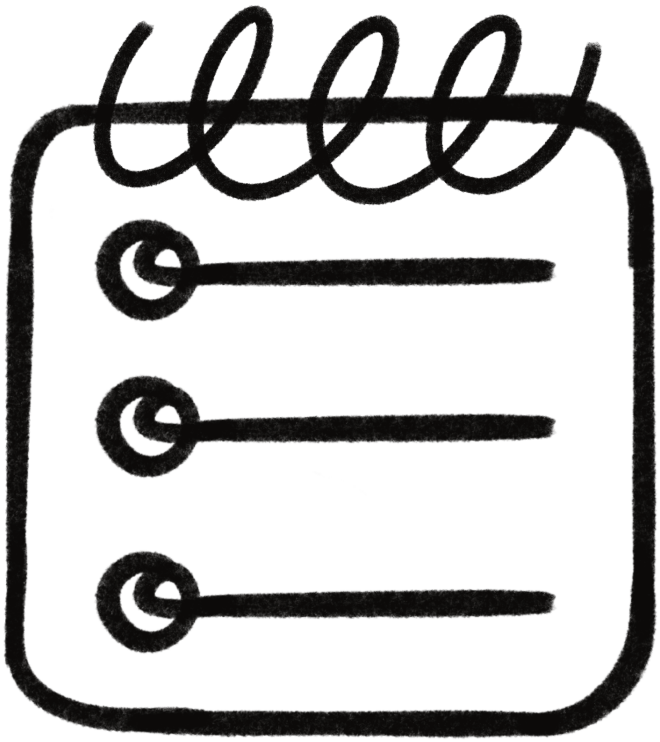
#iterateyourprocesses

[self]

Remind yourself about things immediately

If something comes up that you have to tend to at a later date, add a reminder immediately so you can get it out of your head. Let your technology work for you. Anyone who knows me knows I'm constantly talking into my watch things like "Remind me to review the client email tonight."

#technologyisyourfriend



{people}

When hiring, ask for and call references

Interviewing is hard, for the interviewer.

Some interviewees simply fail the interview. We don't need to be concerned with those at the moment. However, the ones that do well in the interview still may be complete idiots or worse, jerks, once you start working with them. This type of stuff can be hard to suss out before hiring which is why most companies have a 90 day probationary period.

One thing you can do to help with this problem, and hopefully avoid making the wrong hire, is to call the references provided by the applicant. If they have no references, or it's their grandmother, that might be a red flag. During the interview, ask them about someone who would *not* give them a good review. What would this person say? Why would they say it? Watch their body language as they answer this question.

There's only room enough for one idiot or jerk on your team and the odds are you already have that person.

`#hirecarefully`

{people}

Tell me something now that I'll find out in 6 months

This was my goto question when I was in the dating world - oh the answers I received. I realized that it also applies well to the work environment. During an interview, this is a great question to ask of a person. See how they handle it. They may tell you something that could help you make your decision one way or another.

"I got fired from my last job for stealing" = don't hire

"My biggest weakness is I'm too kind" = don't hire

"I enjoy being a team motivator once I get to know everyone" = maybe give them a chance

#tricknotrickquestions

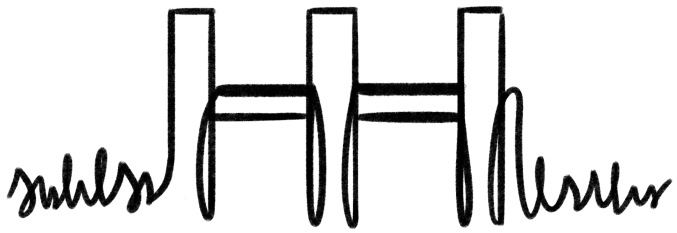
[self]

The grass is not greener at another company

No matter how much you want to think it's true, the same problems occur at every other company that occur at the company you are currently with. Sometimes more, sometimes less. The bottom line is you will always have adversity to overcome regardless of where you work, who you work with, or who you work for.

Try to accept the positive and manage the negative. Unless you feel threatened or abused (in which case you should get out of the situation immediately) you may be better off where you are - so think carefully before jumping ship. Sometimes you'll land on a cushy life boat, sometimes you'll land in a school of sharks.

#deadgrassisntgreen



[self]

Question authority (and marketing)

You may not be ready for this in the very beginning of your career or after you first join a company, but look for the ability to express yourself in this regard. Everyone you work with is a person just like you. They are mere mortals. Some may have knowledge you don't, but that doesn't always mean they are correct.

If you are presented with a project or task and you really think it is simply not a good idea, express that. Don't be an ass. Express it professionally and calmly. But express it. Bring up your concerns, you could be missing a piece of the picture - or - the person who presented the idea may be missing something you bring to light. Either way, someone will learn something from the interaction.

All conversations can be constructive and for the betterment of the project. Have the conversations.

`#questionbutdontbeanass`

<code>

Don’t change code that works

I wish I knew why some developers felt the need to mess with a good thing. But alas, I do not.

Here’s the thing…unless you have a good reason to change something that works, don’t mess with it. The odds are you’re behind on something else anyway, focus on that. Now, sometimes, certainly, you will need to refactor some older code in order to speed it up, or fix new compiler warnings or errors, but tread lightly. Remember, one line of code in the 10 you are changing may have external dependencies that were not documented. It may expect a certain set of circumstances to be true or false. Maybe that line is before that other line for a reason, don’t swap them just because you don’t like the style.

Make sure you *fully understand* each line of code *before* you change it and certainly understand it *after* you change it.

`#treadlightly`

Don’t reformat code

Everyone codes differently. If your team has guidelines that they follow for code style, follow them. If you are working on code, unless that person has left the company or the code is extremely hard to make sense of, try to resist reformatting it just to have them reformat it back when they work on it again. Sometimes I will reformat code temporarily while I work through it but then not commit the reformatting unless it’s really necessary.

In general, if you’re managing a team, put some coding standards in place so it’s hard to tell who wrote it just by looking at the style. This helps makes your codebase more cohesive and easier to manage for all developers on the team.

#codeneatly

(process)

Follow the formatting criteria that your team (or your language) uses

I've always found it's best to follow the formatting that the language documentation suggests. No one can argue with this.

Secondarily, if your team has coding standards, follow them instead. Everyone can argue about them so it makes them fun.

Reserve time at least once a week to "discuss" tabs vs spaces. Everyone knows one is better than the other.

`#followtherules`

{people}

Don’t forget to say thank you

Simple things that you learned as a child go a long way in the business world and even further when dealing with creative types.

Remember, programming is creative. Even if it’s “just” a bug fix - developers pour their heart and soul into figuring out what is wrong, what the best way to fix it is, and making sure it works. When you see someone do a good job, thank them, publicly if possible - in a chat room or meeting. Let them know that you noticed their work and you appreciate it.

It will make you *look* good but more importantly make the recipient *feel* great and other developers want to do better. It’s a win win win.

`#benice`

(process)

Don’t risk your codebase because someone “needs to get this merged”

Sometimes members of your team (or another team) will want to “slip something in” to the codebase to be included in a build that is happening in an hour. No matter how nice and accommodating you like to be, don’t give in to this type of pressure.

If you have a series of steps that you follow before merging code into the codebase then follow those steps, ignoring the hour deadline. If the code passes muster then merge it in but **do not** rush. Not only might the code cause problems if you miss something by rushing but it will also teach the person who requested the faster response to expect it from you regularly.

You can always say “Hey, I don’t make the rules.” - even if you do.

Before I hear about it on social media, yes, there are always exceptions, but you have to make that call. Exceptions == risk. Don’t complain when you’re woken up in the middle of the night because the code you rushed in is causing problems. The person who requested it is likely sleeping soundly.

`#sleepbetter`

<code>

Don’t take the easy way out

Find the *right* solution, not the fastest or easiest!

Your future self, and the future selves of others, will thank you!

`#doitright`

(process)

Make good commit comments

When submitting code to be reviewed and (hopefully) merged, please be sure to include very clear comments explaining what the submission is all about. You should start by linking the work request/task that triggered the work to be done. This gives the reviewer the back story and an idea of what was *supposed* to be implemented. The comment in the pull request itself should be very clear about what was done and if anything was left out (or added) and why. The goal is to give the reviewer an idea of your thought process and what you *think* you implemented based on the original ask.

With the original ask, your description of your changes, and the actual code you changed, the reviewer can validate if all was completed as expected. In the event it was not, they can make the clearest comments possible when sending it back to you for adjustments.

`#clarity`

(process)

Document everything

I would argue that every single line of code should be documented. Now, you may immediately think I'm insane. Notwithstanding, what I say is my truth. Having said that, note that sometimes, a clearly written line of code can function as the only necessary documentation. However, if a line (or lines) of code has any complexity, or is not obvious to a developer with less experience than you, or has external dependencies on state, there should be a comment. Explain what is happening. Why is it doing what it is doing and why is it doing it in the way it is doing it?

The same holds true when it comes to application architecture. Once an application is written, looking at a folder of files you might just see the trees - this class, that class, another bunch of utility functions. But what about the forest? What about the path through the forest? What about all the little objects…er…animals…that live in the forest and interact with one another? These can easily be invisible to someone looking at a clump of trees. They need to have somewhere to turn to be told that there is a forest and all of these things within the forest that exist and depend on one another for survival.

`#documenttheentiredamnforest`

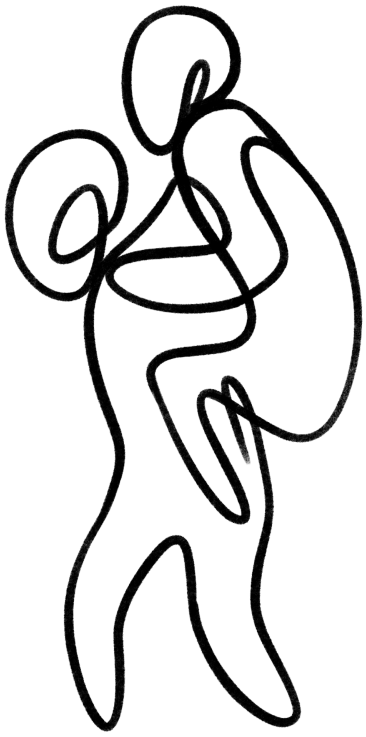
(process)

Don’t piggyback changes to another change

Many times while you are implementing a feature or fixing a bug, you see something (many things) that also need a quick tweak. Maybe there is a spelling error in the code that is unrelated to what you are doing but nonetheless you want to fix it. Fixing it is completely fine, but just don’t piggyback it along with the changes you are making.

Instead, include the unrelated change in a separate commit that can be reverted later without impacting your work. If your company has a process for creating change requests, follow the process. With today’s modern tools, shifting gears like this is usually pretty easy and quick and it will make things cleaner in your codebase.

`#keeplikewithlike`



(process)

Update the documentation

As you are perusing code, maybe looking for a bug, maybe simply trying to learn more about what that developer who is leaving did for the last 4 years, keep an eye on the comments and other documentation.

If you notice anything that is no longer accurate - fix it immediately.

Too many times, a developer will change code but not update the comment above it that explains what it does…er…did. The only thing worse than no comment is an out of date comment. Imagine reading a comment that explains some process very well, but then looking at the code and swearing it can't be doing what it says it should be. This is what happens when comments are out of date: you annoy your other co-workers or your future self.

`#lovethyfutureself`

(process)

Test *around* the bug

When you find a bug and it "gets fixed" it will (should) come back to you for validation. Don't trust the fix. Sure, test the bug as it was reported to make sure the test case passes with the fix, but go further. Test *around* the bug.

Test things that occur before and after the code executes that caused the bug. Test as many cases as you can possibly imagine *near* the location of the bug. This can mean many different things depending on the code, and the bug, but do your best to *invalidate* the fix.

If you run out of ideas and it still works, then we'll give this point to the developer - for now.

`#testaroundthebug`

{people}

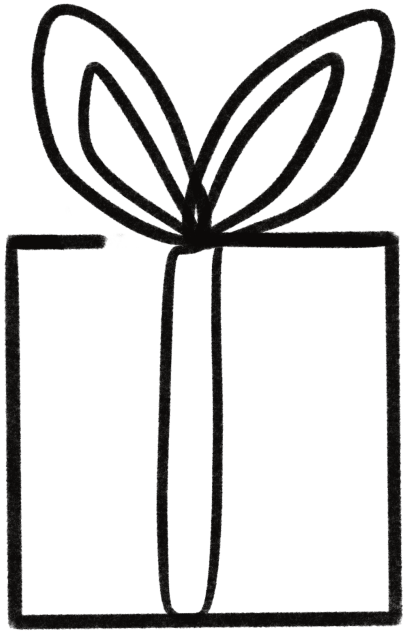
Inexperience can be a gift

There are times when the new developer, the one who just got hired and this is their first job, may be the key to a problem you, the experienced developer, are wrestling with.

There are plenty of times where your knowledge and deep understanding of a problem may be your Achilles heel. Sometimes it takes fresh eyes and new questions and *inexperience* to make you see things in a different light.

Explain the problem to the person who is not you and then answer their questions. Hear them out. I've seen this technique work just as much if not more than talking to a rubber duck and the noob gets to learn a valuable lesson along the way.

`#helpthenoob`



{people}

Offer to help someone

Even if you know they likely will not need your help - offer to help someone. Let them know that in the event they need you, you are there. Everyone needs assistance now and then and it's just nice to know someone is there if you need them. Other people on your team may be stressed out, due to work or personal concerns, but if they know someone has their back, that can help them not lose a nights sleep thinking about the next day's workload.

There is no I in TEAM, although there is ME if you jumble the letters around. Hmmm…offer to help someone anyway!

#thinkofothers

[self]

Be careful with buzzwords

Buzzwords aren't that helpful. You may need to use them here or there when talking to someone who only knows the buzzwords, but use that as an opportunity to explain what they really mean. What is the actual technology that is being discussed? What is it good for? How can it better (or make worse) your project? Be a teacher and expand other's understanding beyond the buzzwords.

If you don't know what a buzzword means, go find out. Your team depends on you.

`#gobeyondthebuzzwords`

[self]

Don’t be short-sighted

Think long-term when creating a solution. Is something a little harder to do but will last a year or more longer than an easier way? It may be worth making the extra effort to avoid having to revisit it too soon.

A quick fix or hack may work today but is the inherent risk worth it? You might be able to only spend a negligible amount of additional time to make a much more stable implementation. Sleep on it. Think about it.

`#looklong`

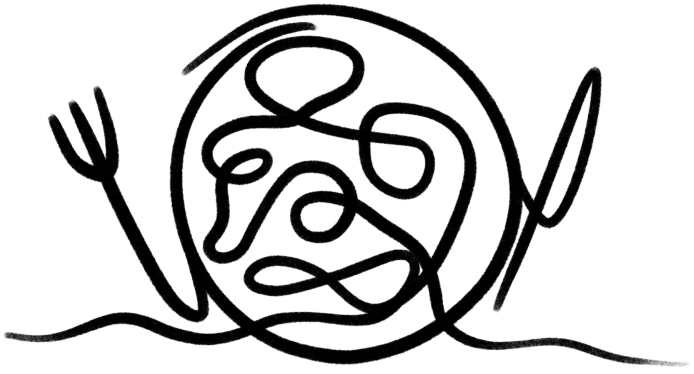
<code>

Spaghetti code is fine if it is uncooked!

Spaghetti code is not fun. If you have inherited some (because you would never write any, you're not that much of a horrible person), take care with it. Ultimately, the best thing to do is get rid of it - refactor it - rewrite it - clean it up. But please be careful.

Spaghetti code can also be described as "teetering on the edge of compatibility." What this means is that changing one tiny thing could potentially cause a collapse of the entire application. Imagine removing the bottom card in a house of cards. If the developer was of the type to have written spaghetti code in the first place, the odds are the rest of their code (and architecture) is a disaster waiting to happen.

`#maytheforcebewithyou`



<code>

Check your else statements

Especially if they are failure statements - make sure they fail properly without crashing when they are simply trying to show an alert with a failure message. It is of utmost importance to check each and every `else` statement in your code. So many times, these statements may configure an error structure and return it to be shown to the user. However, there are times when the code returning the error might be on a thread that is not the main thread, and that can spell a crash when trying to display user interface to the user. This is but one example of crashes I've seen in `else` statements. Check them carefully and make sure you cause each one to execute.

`#elsecrash`

(process)

Do your due diligence

If you're the developer, test your code! If you're the reviewer, test *their* code! Don't assume your own code or someone else's will work. It is everyone's job on the team to make sure the team stays solid. The best way to do this is to make sure the team only outputs quality deliverables. Push others on your team to be better than they are. Everyone, always, has room for improvement.

`#worktogether`

(process)

Don’t upgrade tools right before you are about to do an important build

Developers love complaining about their tools. They also love grabbing the latest version of them the moment they are released. They also love complaining about the latest version the moment they grab them.

Having said that, there are more ominous potential issues lurking in new tools. Will it work with your entire build process or did something subtle (or not so subtle) change that will cause failure? If your company is anything like ones I’ve worked for, releases are done on a specific day at a specific time because the marketing department has push notifications scheduled to go out. There is no time for problems right before a release. Therefore, hold off a day or two before upgrading your tools. Alternatively, install the new version separately from the old. You never know what monsters lay in wait for you.

`#thinkahead`

(process)

Don’t abbreviate stuff in a non-standard way

There are many tried and true mechanisms, processes, workflows, etc. in every industry. For example: KHz stands for kilohertz. Imagine if someone either learned it wrong or just decided on their own to call it HzdK for Hertz de Kilo. A contrived and cute example? Maybe. Maybe not. They see no reason why you shouldn’t understand what this means. But you bang your head against the wall trying to figure it out.

Call things what they are. Don’t make up your own abbreviations. Standards exist to allow us all to communicate succinctly and accurately. Don’t deviate from what is considered the norm on your team or in your industry.

`#hertzdekilo`

[self]

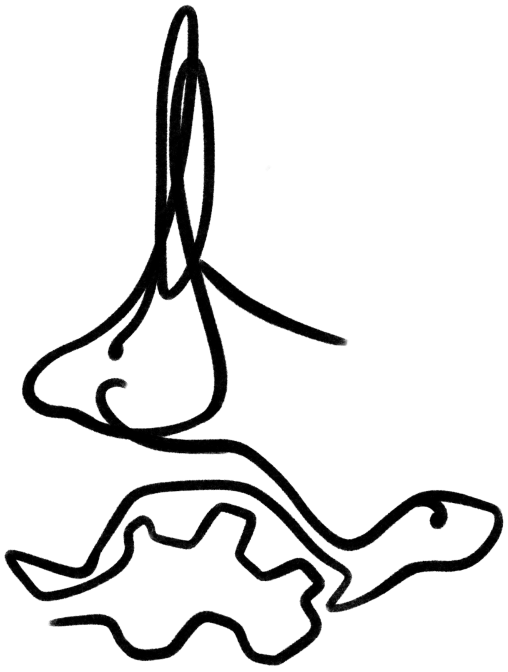
Do your best, take your time

Especially when writing what will be the first draft of a piece of code, you really need to simply make it work, as elegantly as possible, but it doesn't have to be perfect. You should expect to spend time iterating, cleaning up, and optimizing.

So, as you set out on those first steps of your 10 mile code hike, remember this. Enjoy the journey and don't try to plan things out for the happy path too far ahead, because the odds are by the time you get there, things may be different. Planning for your perfect sunny day hike will leave you unprepared when it begins to rain. So, plan for the rain, but think sunny thoughts.

As long as you do your best, and take your time, you will finish with a quality piece of code.

`#codethoughtfully`



[self]

Take comments to heart and think about them the next time you're implementing something

If you're not having your code reviewed before it is merged, you are doing it wrong. Assuming you are doing it right, you will receive a number of comments on your code and may need to make tweaks before it can be committed. Some of these comments will be simple fixes for silly mistakes. Others may delve deeper into style, architecture, or your use of certain aspects of the underlying code in the project. These comments are meant to make the entire project better, so read them carefully, ruminate on them, and take them into account next time you are writing code…which will be in about 10 minutes because we need to get that other bug fixed ASAP!

`#learnfromfeedback`

<code>

Don’t use double negatives in code

Don’t do something equals `false` is much harder to comprehend than do something equals `true`. Think about it. If you are *not* supposed to *not* open the door then you are actually *supposed* to *open* the door. So why not just say open the door equals `true` instead of don’t open the door equals `false`?

`notEnabled` should not be the name of your variable. `enabled` or `isEnabled` keeps things more straightforward. `notEnabled = true` is the same as `enabled = false`. Which do you think is easier to instantly make sense of?

It drives other developers crazy to try to figure out what you think is a clever way of coding something when in reality it’s not.

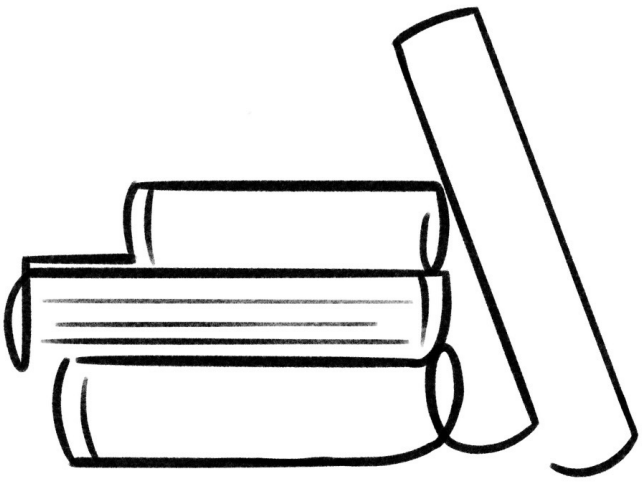
`#dontnotdotherightthing`

[self]

Learn something new each day

It might be about your company's API or another piece of code you aren't familiar with. It could be about a new tool that might make your life easier - or someone else's. Maybe it's reaching out to someone in another group within your company that you've worked with and want to get a better understanding about how they do something on their team or in their code. Every lit bit counts and eventually add up to more than the sum of their parts. As you learn about disparate things, connections are made. You learn to think about things differently. You learn to think outside the box. This ultimately makes you a better resource for your team. What are you going to learn today?

`#learnrandomthings`



Don’t ever assume there will never be more choices in an else statement

If you have two choices, include them all, explicitly. Don’t just leave a dangling `else` to pick up what you will assume to always be the *only* second choice. What happens if a third is added? Your code will likely not handle that case correctly, yet the dangling `else` may not cause a failure condition, meaning it might take awhile for this bug to rear its ugly head.

`#beexplicit`

(process)

Don't push code without testing it

You would be amazed how many times I've seen this. If you change something in the login code, try to actually login (with known good credentials and known bad credentials) before you commit your changes and submit a pull request for review. When your code fails because of a silly mistake you will be forced to buy coffee for everyone on the team.

If this sounds similar to another entry in this book, it's because it bears repeating.

`#sayitagain`

[self]

If you create open source code, it is a commitment & you need to keep it updated, or find someone who will

Open source is great, but, many times it's created and maintained by a single, solitary human. If that single, solitary human is you and you work on your pet project regularly and maintain it, responding to bugs, etc. it may grow to become somewhat popular and maybe even give you an income stream as a consultant helping your users make the most of it.

However, if that single, solitary human is you and you meet another single, solitary human and maybe even decide to have yourself a little, single, solitary human that makes three single, solitary humans. Whenever this happens, all single, solitary humans in the equation end up spending less time on things that they spent time on before, when they were the *only* single, solitary human.

The end result could be that your open source project becomes stagnant. But you know what isn't stagnant? Other software that people created that depends on yours. As the other software continues to be updated, someday, it could be a year from now or 3 years from now, there will be an issue that requires your software to be fixed. Mind you, you've long abandoned it, lost interest, and simply don't have the time or energy for it anymore.

So, the point here is, remember that open source (or any code you write) is a commitment. Try your best to keep up with it, and when you're done with it, find someone else who can. The community depends on you.

`#ifyouleavemenow`

<code>

All code has side effects

You may not live long enough to see them, but eventually the simplest of code that you write will have an unintended consequence that will cause 3 or 7 other people to have to wake up in the middle of the night and spend 3 or 7 hours or days figuring out what is going on.

This is why we plan, then write code carefully, understand every line, and achieve 100% test coverage of said code ourselves before submitting it for review and possible merge into the codebase.

`#dontbeasideeffect`

<code>

Check for nil (and other out of range conditions)

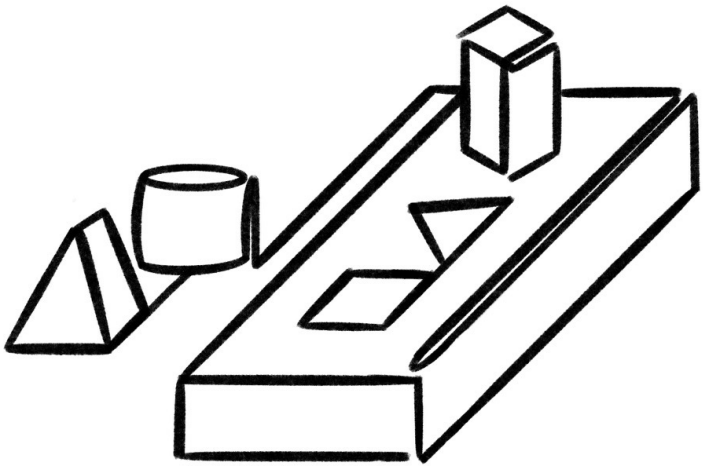
I know you don't think it's ever going to happen on that line of code, but do it anyway, trust me.

```
#nil
```

If you redesign the UI don't try to force fit it into your code, redesign the underlying code as well

Having said that, you may not need to completely *redesign* the underlying code, but you will likely need to at least update it. If a button in the UI changes from OK to SAVE but the code references a variable named `okButton` you should probably rename it to be called `saveButton` now. A simple change like this can save you seconds or minutes down the road when you're searching for the code that references the button in the UI. Following strict (but very obvious) naming conventions like this will help your team make more sense of the pieces that come together to make your app what it is.

`#namethingswhattheyare`



(process)

If you're the developer who updates a new SDK - test it! Don't leave it to QA to fend for themselves!

I can't stress enough how important it is to test your code, especially when you're updating something as significant as an entire third-party SDK.

Read the release notes. Even consider the changes for parts of the SDK that you aren't using. There is always a chance that the SDK itself has been refactored internally. This means that code within it that may have been simple before is now more complex, and using other (or new) parts of the SDK that were never executed previously.

Now, you would hope the SDK developers tested it fully but they can't know all the possible environments it will be thrust into. It's very possible they don't have a 5 year old smartphone to test on, but your software needs to run there.

Remember, you testing a little more thoroughly may help save a round-trip to QA, which could mean the difference between making your release date and not. Take the time now, lest the code come back to you for repairs and interrupt the next thing on your task list.

Using third-party SDKs are a necessary evil - but tread carefully.

`#takethetime`

[self]

How fast is your train moving?

If your train is moving too fast you risk running off the tracks. Slow down. Take your time. Look at the scenery and do the trip right. Don't miss the details. That's why you're taking the trip in the first place.

`#slowcode`

{people}

Ask questions even if you (think you) know the answers because it makes the person answering think differently

When reviewing code, if something looks OK but could be done better, or something just doesn't seem logical, even if you know what is wrong, ask the question to the original author, as opposed to tell them the answer. Allowing them to explain it will force them to think about it in a way other than how they convinced themselves it was the right choice. This will not only help them realize there may be an issue but also help them think through a solution. Let *them* connect the dots instead of doing the thinking for them.

`#helpthemteachthemselves`

<code>

If you need a comment to explain some code then it's probably too difficult - especially if the code should be simple

Of course you need comments on code that may not be obvious as to what it is doing. There will certainly be code in your project like this. However, if even writing the comment is complex, maybe think a little harder about the solution you've decided on. Is it possible to break it down? To simplify? It may not be, but it may be. Sleep on it.

`#codesimply`

[self]

Take frequent breaks

Rest your mind. Rest your eyes. Rest your fingers.

In fact, take one right now.

`#rest`

<code>

Make something reliable from the beginning

Especially when starting a project, many times we are so excited that we want to get it all done quickly. This can mean we rush through parts, or cut corners. Working in this way can easily cause us to make mistakes, some of which we won't realize we made until our code crashes on a user's machine.

Take your time, type slowly, re-read the lines you just wrote before you move on. Make sure they are solid and dependable. Add a comment if needed.

Build reliability into your way of working.

`#codereliably`

<code>

Don’t write gnarly code

It sucks for the next person and your future self.

Always strive to write clean, concise, clear code.

`#itsnotacontest`

Don’t “mark" something just to make it yours

Work with your team - you’re all in this together - so work that way. No one needs to own any particular piece of your project unless it is some crazy complex piece and there is only one person who understands it. However, in that case, someone else should start learning that piece and become an apprentice. What happens when the original maintainer gets hit by a bus?

#lookbothways

[self]

Hard now or hard later is the same hard so might as well make it now

Don't cut a corner now and promise to make it better later. You will never make it better later. It will stagnate for months or years and then when it causes problems again someone else will have to figure it out and deep dive into the mess you left in the first place. If you don't believe me, search your current codebase for `TODO` and see what you find.

You are doing nothing more than creating technical debt every time you cut a corner and postpone the inevitable. Do it right today so you (or someone else) doesn't have to undo it and do it again tomorrow.

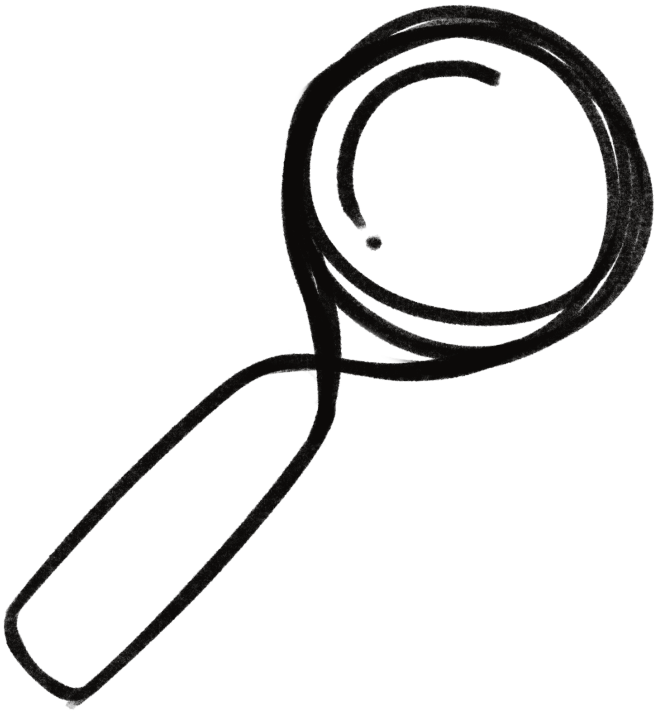
`#dontprocrastinate`

[self]

Read, research, learn

Don't ask questions about something that you chose not to learn yourself. Well, sure, ask a question or two, but you should be able to notice in yourself when you actually need to take the time to dig in deeper. Carve out a few hours, or a day, or a weekend, and really figure out what you need to know. Once you have a basic knowledge, then by all means have a meaningful conversation with the resident expert on the topic. You'll gain much more from it having some base knowledge.

#alwaysbelearning



{people}

Tell people they are doing a good job publicly (or a bad one, privately)

It's important for people to receive praise. I make it a point to call out positive effort in a public forum. Not to embarrass the person but to make sure the entire team knows that someone is noticing good work and that it is worthy of being noted. It can also help give a guide to the entire team as to the level of effort that is expected and appreciated.

Sharing praise can be as simple as saying, in your daily standup, something to the effect "…and a huge thank you to Jenny for really digging into that bug, resolving it, and documenting the heck out of it! Great work!"

Now, in the event that Jenny spent 4 weeks trying to resolve a problem that was much lower priority than many other things she should have been working on, that calls for a private conversation. If you are the leader of the team, or Jenny's people manager, bring this up privately - provide guidance - offer assistance if needed - suggest alternative approaches - one of which might be to focus on the important work now and come back to this during downtime. Use it as a teaching opportunity to help her more easily see when a problem is getting away from her and impacting the work that actually needs to be done.

#managingwithcompassion

Question everything, even if the code looks good, play the devil's advocate

The worst part about code reviews is that it is really easy to poke holes in other peoples work. Sadly, humans are great at really unwrapping people and finding their weaknesses. This is bad when humans are on the receiving end but it's good when code is on the receiving end. Make sure to address the code and not the person.

Take a look at every single line. Is the line using a parameter passed into the function? What if the value of the parameter is way out of range? Will the code crash? If it handles the error, how does it handle it? Sometimes developers don't really test their error conditions and how they bubble up to the caller. A developer might think "if I just `return nil` then all will be well." This can not be guaranteed until it is tested - thoroughly.

It's really easy to make code crash by giving it unexpected data. Code should be resilient enough to be able to handle this except in very few cases. One might be where the called function is very tightly optimized and it expects as part of its contract with the caller that the data passed in will be vetted and clean. This is perfectly acceptable - as long as it is documented as such. Many low-level operating system functions work this way to keep things moving swiftly!

`#commentonthecode`

(process)

Don’t put pain on someone else just to make your pain less - work together for the best solution

It’s easy, when designing an architecture or system, to push the hard work on some other piece of code that is not your problem. Letting Tom do that work instead of you because his code has access to the data anyway so why should my code deal with it? It completely depends on the situation but ultimately you should make sure that work is done in the code that makes the most sense to do it, regardless of if it means work for you or something else.

Now, the decision of where it “makes most sense” may be up for debate - but numbers don’t lie. If speed is the most important concern, then it might be obvious where to put the code. If memory is the most important concern, that might provide a different answer. Work with your team to determine the pros and cons of each possible solution and then go with what’s best for the project, not just for you.

`#maketherightchoice`

(process)

Don’t implement something “just because an ego said so”

So many times people who make decisions don’t know what they are talking about. Change this UI so it does that instead. Why? Ask yourself Is it really better? Is it solving a problem? Or is it just the project manager or business representative trying to mark their corner of the app?

There is no room for ego in product development. If the idea has merit and is good for the product and the customer, then by all means, move forward. But if we are changing the entire design of something just because someone “doesn’t like it” with no other reason - that’s a tough sell.

There are always exceptions, but push back on the ones you know are just plain wrong.

`#eventheceocanbewrong`

<code>

Keep it simple - don’t add fancy logic

This is pretty self-explanatory but the key with writing code is simplicity, readability, and stability. Not necessary in that order. However, the point remains, keep it simple. Sure, you can optimize later but it’s absolutely OK to use a simplistic or brute-force method to get things going. If the code is stable and acceptable in the speed and memory usage departments, that might be the end of the work. However, if not, you can take a second pass after it functions and figure out how to best optimize it for performance, etc. But even at this stage, don’t get fancy where you don’t need to. Keep it simple. Simple wins the race.

`#simplicity`

<code>

Check version numbers before merging third-party code to make sure you have the latest

Sometimes we need to use third-party code. Sometimes we've been working on integrating it for awhile and the version we have locally is no longer the latest version. Before you merge any third-party code, take one last look to make sure there isn't an update with critical bug fixes (or worse, a major update) available instead. The last thing you want is to be committing code that may already be out of date or contain known issues.

`#keepuptodate`

[self]

Just because you don’t know about something doesn’t mean it doesn’t exist

There will be people who know more than you. Learn from them.

#mostseniordoesnotmeansmartest

{people}

Learn how to communicate with different people (and their different styles) in your organization

I was in an interview once where the interviewer commented that I wasn't being concise enough. Now, I was in an interview, so I was trying to answer questions in a very detailed manner. However, once this was mentioned, which I had never heard before from anyone, I was a bit taken aback. I wasn't offended but I was certainly slightly embarrassed at the fact that what was perceived as a fault was pointed out so blatantly.

After talking to a confidant about the scenario, he mentioned, he would just take that as a note as to how to communicate with that particular person. It didn't mean that I was somehow flawed in my communication style. It had worked fine on multiple teams for decades and through an entire career. But this particular individual needed to be communicated with in a very optimized and concise manner. Their time was very important to them and was used as a measuring stick.

I've certainly had people on teams with egos or anxieties that I had to change my style for here and there, and you will as well. It's ok to adjust your communication style for the audience. Remember, the way you talk to your friends may not be the way you talk to your grandmother.

#communicatewithstyle

<code>

Comment when needed

Comments can be super helpful in code, but they can also be unnecessary. You likely do not need a comment on a line of code such as `x++` - it's probably pretty obvious what is happening there.

However, if you are writing code that depends on the state of another thing, or works asynchronously and waits for a certain condition to be met, this is where comments become a requirement. Although you may have thought through the entire process in-depth and have been writing it all week long so you understand it inside and out, you know who doesn't? Any developer who looks at this code 6 months from now - maybe even your(future)self.

If some functionality or expectation or intent is hidden, write a comment to explain it! Be as detailed as possible. Make references to other files or functions where the dependency lives. As a corollary, make comments there as well. Make sure someone editing that code down the road knows that there are other things that depend on it working a certain way and provide a reference to those things.

`#readthecomments`

Don’t over-comment

If your code needs a comment other than for hidden intent, etc. then you are probably coding it wrong. Let me explain. As much as possible, the code should be the comment. If you follow other concepts discussed in this book, such as naming things what they are, then the code becomes so easy to understand that comments aren’t really needed, except in those cases where dependencies on external pieces need to be notated.

Why wouldn’t you want to comment? Comments get out of a date and they themselves become technical (or documentational) debt when they are so. So please be sure to keep them up to date whenever you come across one that could use a freshening up.

`#commentwithcare`

(process)

Don’t think of the solution before you think through the problem

Just because you really really want to use a clever construct to solve a problem, it may not be the right choice. Don’t settle on a solution before you really think through the problem. Another, more simplistic, way of implementing it might make more sense. You can always save your fancy coding for something that it is more suitable for.

Another way to look at this is to get back to basics when trying to solve a problem. Think of the brute force method first. How would you solve this like an engineer who is new to the job? Can it be solved that way? Once you work through this exercise, look for places to optimize that brute force method. If the code is solid and functional then that may be all you need to do. There may be no need for something more complex.

Every line of code should be no more and no less than what is absolutely necessary.

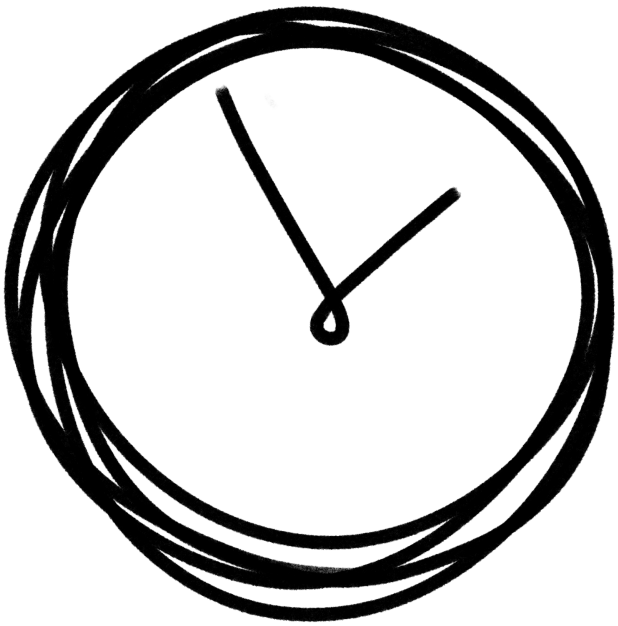
`#codesimply`

[self]

Ask yourself if the time you are saving now is worth three other people's time later?

If something saves you 10 minutes is it going to be worth the 45 minutes it takes three other developers to undo or decipher what you've done? Don't cut corners. Don't be convoluted. Do it right. Make it clear. Even if it takes you longer.

`#yourtimenowissomeoneelsestimelater`



[self]

Don’t procrastinate, especially if someone is waiting on your work

Unless you are working on a team of one, there may be someone waiting for something you are responsible for. Therefore, make sure your priorities are straight to avoid as much residual delay as possible. If you put something off that another member of your team needs before they can start their work, you are not only delaying yourself but also them, not to mention anyone downstream from them.

Don’t be the log blocking the river.

`#dontbethelog`

[self]

Don’t over-schedule yourself

Schedule yourself for no more than 75% of your available time, not 125%. This goes for teams too. People need breathing room to relax and also have time to handle the inevitable, unexpected issues. If you feel like you’re cheating the system, think about this: You can’t do your best if you’re overloaded and stressed. Ooh, that rhymes! You will actually be more positive and productive working a strong 75% than a rundown and weak 125%. The remaining time will easily fill itself in anyway with people pinging you for advice or assistance.

Take care of yourself first so you can take care of others later.

`#putyouroxygenmaskonfirstbeforehelpingyourchild`

(process)

Everything is an emergency - not really

Certainly there are emergencies, however, not everything is. There are times when something impacts a person or customer and to them it may be an emergency. However, you need to evaluate the entire problem and the impact across the board before determining how much of an *actual* emergency it is and prioritize from there.

This can be difficult when the CEO walks in your office and sees a problem that they think needs to be repaired immediately. The best thing to do in this case is at least look into it to evaluate the severity and see if it impacts others as well. But if it ends up that the CEO just needs a software update to get the fix, or was doing something incorrectly, be gentle in your approach and help them through their issue. “Emergency” resolved.

`#triage`

{people}

Be proactive - don't make people come looking for you

A lot of this boils down to communication. If you communicate what you're doing, what you did, how you did it, or when you're going to do it, then most of the time people won't be wondering and trying to track you down.

Communicate!

Be clear about your intentions to "fix that bug" or "check into that thing you promised Mike you would help him with." Don't use nebulous terms such as "I'll get to it" or "Let me see how things look"…instead, be specific. "I will look at this tomorrow" or "I don't have time this week but I will look at it first thing Monday morning."

Once you've committed, set a reminder for yourself, and then follow up once you've looked into the thing. If it looks like it will take 3 weeks to fix, let the stakeholder know that *early*, like Monday afternoon…don't let them wait (and wonder) for three weeks. Even though you know you're working on it, they don't.

Communicate!

`#communicationiskey`

<code>

Quality now equates to less problems later

Take your time. Code carefully. Test thoroughly. Build with quality first.

`#simpleasthat`

<code>

Rule of threes - anything more than 3 is probably too many

I've always loved the number 3. It's my favorite number and I tend to favor it when a number is part of something I'm doing. Over the years, I've found that more than 3 choices is simply overwhelming for most people. When giving choices to team mates or customers or whoever, limit them. If there are more than 3 then see if you can distill them down to 3.

Famous threes that make sense include:

- Small, medium, large
- Regular, mid-grade, premium

Famous groupings that are confusing:

- Short, tall, grande, venti
- Medium, large, extra-large, jumbo

#3

If something is hard for the top developers on your team to understand maybe it's time to rethink what you're doing

Code, processes, and architecture don't have to be complex. So many things can be distilled down to core components that talk simply to one another with a little bit of effort. If you have trouble keeping an entire system in your head and understanding all of the intricacies, it's simply too complex. Try to simplify it.

There may be a case where you simply can't do this for part of your work, that's ok. In that case, document the hell out of it! Your future self and the person who takes over for you after you get fired for making something so complex in the first place will be very happy you did!

#documentforthefuture

(process)

Distill it down

When planning, designing, or considering either low-level code or user experience, distill it down to the bare necessities. Feature creep will inevitably come later, but in the beginning, consider the absolute minimum needed to complete a task, get a point across, or solve a problem.

`#simplerisbetter`

[self]

Pick your work from the stream and avoid being stressed about workload

Although it seems harsh and negative to say, you will never catch up. The stream is endless. There will always be more work to be done. How long does inbox zero last? Sometimes a few seconds. Maybe a few minutes. Maybe a day if you achieved it on a New Years Eve when you should have been having fun.

Think about this, and don't stress out, if a few things pass you by in the stream while you're working on something else. Don't overload yourself. You can only do so much and that is exactly what you are doing. Work within your means.

#endless

[self]

Coffee (or tea) first, email second

Take your time ramping up in the morning. Ease your mind into your work. Create and follow a morning ritual to help you be the best you can be once you're ready to begin your workday.

#takecareofyourself

[self]

Developers should only work on one thing at a time

You can only think of one thing at a time. You can only focus on one thing at a time. You can only type one thing at a time. Don't try to be everything to everyone. Humans are single-threaded. It's OK to focus and complete something and let the other things wait.

Take it from someone who has experienced this. Trying to constantly switch focus throughout the day can not only be extremely hard but also make you feel completely wiped out and like you accomplished nothing at the end of the day. Essentially, this is because you end up not finishing anything. You get a bunch of little bits of things done but nothing is complete. Whereas, if you focused on one item at a time, you might have finished 3 things by the end of the day instead of doing 5% of each of 8 things.

Focus and finish instead of feeling frazzled.

`#focusandfinish`

{people}

Compare saying no to a project manager vs overworking your best developers

Project Managers can wait. They'll get over waiting another few weeks for their "urgent" request. If you constantly say **yes** you will overwork your developers who will *not* get over it and will burn out and leave. Then the project managers will *have* to wait. Do the right thing. Say **no** a few times and let your developers have a breather. Don't try to overstuff the machine, it will jam.

`#respectyourteam`

{people}

If a business person says it should take less than two minutes they should be fired.

There's not much more to say here.

Other versions of this include "What's that like a weekend worth of work?"

But seriously, educate these folks that say things like this. Make sure they realize what they are saying and how it is disrespectful and negligent. If they spread the thought that something that really will take a month should only take a weekend, that makes it harder for the development team down the road when they are working with other people within the organization.

We don't ask Todd in finance how long it's going to take him to finish his spreadsheet do we?

#educate

<code>

The code is the true documentation - bugs and all!

I don't care what the code is *supposed* to do. It does what the computer says it does. You may know how to code pretty well but the computer is the ultimate maker of the rules. Learn to speak its language.

`#thecomputermakestherules`

<code>

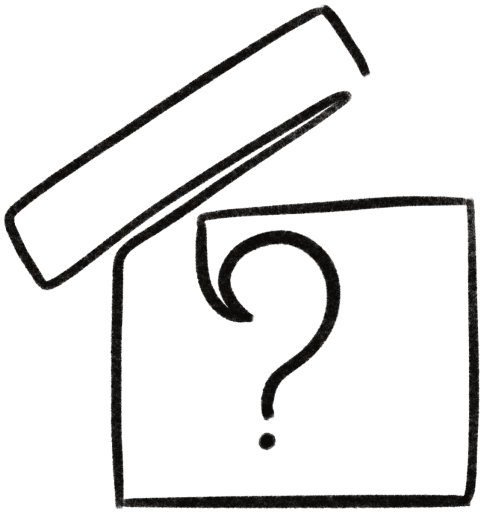
Don’t guess

Seriously. Don’t. Figure out the right way to do something from the start.

When I was learning C I had no idea what a handle or pointer really was. I would sit there and type `*x` or `*x->` or `*x.` until something compiled properly and (seemingly) worked. I had no idea how to properly dereference because I didn’t understand the relationship between handles and pointers and the underlying data! However, once I learned, my life became much easier and my code much more likely to do what I wanted - and not have bugs. The time it took to learn was .001% of the time I wasted guessing.

The code is always right even when it's wrong.

`#dontguess`



{people}

If you only hire cheap developers, your software will end up that way

I would add to this, if you only hire developers interested in more and more money, you will also get software that is not worth their salt.

Hire people who care, are passionate, and have an interest in your product. Pay them a fair wage. Nothing too little and certainly not more than they are actually worth - these ego-programmers can be shown the door. Let someone else hire them.

Today's market is a little crazy but try to stay realistic when hiring and rewarding your team. Be fair but don't be taken advantage of.

`#hirecarefully`

(process)

Don't over-complicate your comments with formatting that no one else can understand

```
1;;;re:crash[98][110];-branch:fixer;
```

What does this mean? I don't know either.

Let's guess. 1 could be a flag or an ID of an issue. Not sure why there are multiple ;;; in a row, likely missing data between them. Regarding a crash. Maybe the numbers in brackets are line numbers? I'm guessing the branch name is "fixer." The last thing I want to do is learn this format and come up against it every time I need to get work done.

Write in clear, proper English (or the language your team agrees on) but not in a format that takes IBM's Watson to decipher. Although this may mean something to you or you might have a script that generates or reads it, it is likely useless to other folks on the team - aka: the humans you work with.

You should change to conform to the team, the team should not change to conform to you.

```
#thereisnoiinteam
```

Don't hide parts of your UI in the simulator

I was working on a project where users were reporting an issue on their devices that I could not reproduce at all in the simulator. The reason? Because that feature required a device and the controls for it were completely hidden when running in the simulator.

This initially caused confusion as we tried to reproduce the problem and didn't have an actual device to test on. After awhile of trying different things, digging into code, changing feature flags, etc. we finally figured out why we weren't seeing the problem.

Then, we had to track down a device that supported the feature - sometimes hard to do when you are supporting dozens if not hundreds of potential device configurations - even moreso with remote workers. "Hey, can you FedEx me that iPhone 11 Pro? I have a bug I'm working on and I need that specific device."

Imagine how much easier this could have been if the feature was implemented fully in the simulator as well - even if it didn't 100% work, we could have at least seen some semblance of it to help guide us.

Now, certainly, there are always work-arounds. You could go change the code around the bug to make it appear in the simulator, and then hope the bug is reproducible. It could have been that even if the code was in the simulator from the beginning the bug may only appear on devices. But the point is this, don't add limitations just because you can. Make it easy for your team to be successful when they need it most.

#dontaddfalselimitations

(process)

Don’t just test with data that you expect. Test with data that is “impossible” and “should never happen”

Test like you hate yourself - like you want to make your life miserable.

This is how you will find the bugs.

Always toss data at your program that is completely impossible and should never happen. Strings that are too long and have strange characters in them or numbers that are out of range in the positive or negative.

Remember, SQL injection bugs were never supposed to happen either.

`#codedefensivelyandtestmoreso`

{people}

Ask what others think

You know the developer sitting in the corner that not a lot of people understand - either personally or technically? They probably know a thing or two. Build a relationship. Pull that person into the fold. Not only are you reaching out to another human, potentially in need, but you are gaining a possibly excellent feedback mechanism for your code and ideas.

You may not have a developer like this on your team, and that's fine. In this case, ask someone else, either more senior or more junior, about something you're putting together. Lift a junior up to learn from you and you just might learn from them. Take advantage of a senior and their wealth of knowledge from years in the industry.

`#interact`

{people}

Be a lifeguard

If you see an individual or a team flailing, help them out. If you don't know how to help them, discuss it with your other team members to see what you can do together.

It's usually pretty easy to see someone stressing over a feature or a bug fix. Ask them how it's going. Take them out for a coffee and talk through the problem away from the computer. It's amazing how abstracting a problem away from the screen can shed new light. Not to mention, simply talking it out instead of only talking to yourself, inside your head, can do wonders for breakthroughs.

I once had a coworker that would simply request my presence in the room because our back and forth would help him solve the problem. I had almost a 100% success rate of fixing his bugs for him just by sitting across the table. Sometimes I didn't even have to utter a word.

In the event of a team needing help, you can usually tell. If you depend on a team for something and they aren't delivering or delivering shoddy code, talk to them about it. See what you can do to help them. Maybe meet them half way. Maybe help them test. There are endless possibilities of what could be going on in their world that is adding to their inability to be productive.

`#strongcurrentsswimwithcaution`

[self]

Don’t rush

If you feel yourself stressing out about what you’re working on, take a step back. Trying to meet a deadline when you are not completely clear on the direction you are going or how something is supposed to work can be detrimental. Taking some time to reframe the idea, talk it out with someone else, take a walk and think it through away from the screen, or even sleep on it - maybe for a weekend - can make all the difference in the world.

Many people feel guilty taking time away from the computer to think through problems, but you’ve undoubtedly heard about people solving problems in the shower. Being away from the screen - in a different physical place - that’s really the point there. Just get away from the screen, let your eyes focus on something further away.

The corollary to this is rushing, trying to change what you’ve done already to fit some new simplified hack that is not the proper way to solve the problem, but will make the deadline. That is a recipe for disaster later in one way or another. You are creating technical debt that will come due when you least expect it because you cut corners now.

Take your time. Step back. Do it right.

`#takecalculatedsteps`

If you're typecasting something you are probably doing it wrong

There are always exceptions, but why are you typecasting that thing? Usually typecasting equates to shoving something in a place where it doesn't fit or doesn't belong. Really think about what you're doing and why you're doing it. Is there another way to make this work in a more expected way?

If you *do* have to typecast, be sure to document the sender and the receiver so future developers know what is expected and what *could* be passed in as a parameter in *some* cases. There is nothing like changing code because you expect *this* but end up crashing because you received *that*.

`#beclearandspecific`

{people}

If you realize someone on your team is a liability then let them know and let them go

Many of us are trying to simplify our lives and lower our stress levels. Having someone on your team that is a liability, that is, they do not play by the rules, they make extra work for other people, they bring bad energy to the team, etc. can be a detriment to the *entire* team, the product, and the company.

You can certainly work with the person to see if they can improve but they may not have it in them to or they may simply not even be enjoying their job. You'll know when it's time to part ways. Be kind about it if you can, but get it done. Everyone will be better off, including the liability.

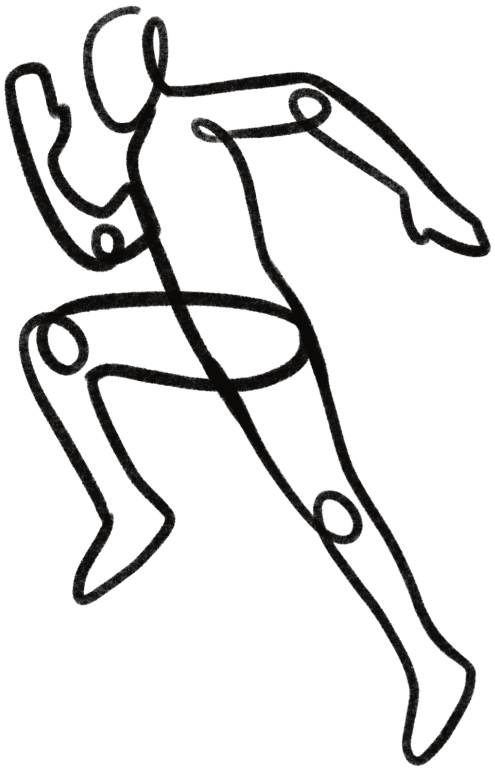
#behonest

Don’t be late for meetings

You’re an adult. You know how calendars work. You know how notifications work.

If you’re going to be late for a meeting, let the others know. There is nothing worse than waiting for 10 minutes for someone to arrive who happens to be pivotal to the agenda.

#respect



Always think about the order of operations and optimization

For example, if you sort your clothes as you put them in the dirty piles then you don't have to sort them when you're actually doing the laundry.

I'm constantly thinking of the order of operations of everything I do all day long. Is it faster or easier for me to take the trash out the back door and *then* walk out front to get the mail? Or should I get the mail and then come back in and handle the trash. If it is muddy in the yard this may influence my decision. If there is a lot of trash this may also influence the decision.

The same is true for coding. There are so many variables to consider. So many potential states of being. Especially when making round trips on the network, you want to be sure to optimize not only the calls but also the work that happens before and after them, and maybe even during them.

Now, don't get caught up in the pre-optimization game. You don't have to optimize from the get-go. Many times it's easier to brute-force a solution and then look for optimizations later, once you know it works. But no matter how you end up doing it - whatever works for you - be sure to consider these points as you code each line.

`#think`

{people}

Give someone a chance

My uncle bought me my first color computer. Out of the blue he offered to spend close to $3000 at the time while I was in college. This improved my programming immensely. I now had the ability to write programs that made use of color and I also had a much faster computer and larger screen. It was pretty amazing and it was all because someone came along and decided to give me a boost when I didn't even know I needed it.

My first *real* job out of college was because someone gave me a chance. A college-trained musician turned software engineer hired a college-trained musician embarking on a software engineering career. In fact, two people gave me a chance. One who I knew from college mentioned my name to a company he happened to be talking to at a trade show. They then reached out to talk to me. We met at the trade show and chatted in the corner of a busy room for maybe an hour and the next thing I knew, I was reporting for work as a software engineer.

Give someone a chance, it just might completely change their life.

`#thankyouuncledavid #thankyouralph #thankyoumashski`

(process)

Be proactive

Instead of making people search for you for answers - be proactive.

As you move on in your career you start to learn what kind of information you need from people and how much pain it can cause you when they aren't reachable. This type of experience is exactly what lead me to start being proactive. I am now a *huge* proponent of proactive communication.

Before you leave for the day, let the team know about anything, no matter how small, that might impact them before you return to work. Let's say you're working on a branch and had to make a change to a development API endpoint on a shared test server. Even though you think no one will want to use the test server while you sleep, you never know what might happen.

If you don't say anything, and just assume no one will try to use the server, someone may try to use it and as it continually fails for them (because the API endpoint changed and is incompatible with their branch), they are getting more and more behind and frustrated. The next day you come in to see your code changes on the API endpoint undone - which you now then have to change back. Imagine if you just warned everyone that the test server was something you had tweaked and would be finishing with in the morning? That would have saved a lot of time, wondering, confusion, annoyance, and effort.

`#proactivecommunication`

Think of the future

When coding a feature - maybe even one that has been very strictly defined - always look for faults in that definition, especially ones that will cause problems down the road. You may be able to make simple or subtle changes behind the scenes to make things more flexible and future-proof.

Now, this can be a slippery slope, so don't go overboard and over-engineer just because you can. There should always be a balance that is maintained between what is required and what you implement.

`#futureproof`

(process)

Be organized and prepared

Make lists. Think through your day before it begins. Plan ahead. Think through potential contingencies and how you will react. Practice.

Your friendly, neighborhood SWAT team doesn't just do what they do without endless practice in a variety of circumstances. When you never know what will be thrown at you, you need to be prepared and practiced for as many potential situations as possible.

In the (likely) event that you do encounter something you haven't considered, you then need to be able to think on your feet. Make quick decisions that will have as little negative impact as possible. All of this takes practice and knowledge. The more you know, the more you can imagine outcomes that you haven't experienced yet. In tech, this can mean the difference in your e-commerce site being down for 10 minutes or 10 hours.

`#beprepared`

Everything is negotiable

Regardless of what it might be - how to implement a feature, how to present a concept to customers, inter-team relationships, your salary- *everything* is negotiable.

Just because someone presents something a certain way, if that way causes friction between teams or people or is bad user experience - negotiate. You have the power to bring up your own opinions and ideas about a thing, no matter what it is, as long as you do it in a respectful and open way. It's very possible that the original idea-person may not be aware of the issues you will bring up. So do it! Bring them up - communicate. Make sure that the decisions being made by and for your team are aligned to provide the best outcome possible.

`#dowhatsright`

Do it now

Instead of planning to do something in the future, do it now. Someone recently defined a process and had suggestions for future enhancements that actually made sense to implement right away, while the process *itself* was being implemented. I find this to be a much more optimized approach over creating a simpler process that then has to change later. Take the hit now and in a few weeks you won't even feel it anymore.

This reminds me of when I wanted to learn to play the fretless electric bass. I was concerned that a fretless neck would be too hard to navigate so I asked the luthier if I should have lines put on. This would help me more easily guide my hands to the places they needed to be to sound the right pitch. The luthier told me: think about where you want to be in a few years and have *that* instrument made. In other words, if you want to play without lines eventually, then don't play with lines now. Train your brain for the ultimate destination, not using a crutch in the meantime.

#whywait

Conclusion

Thank you for reading!

If you enjoyed this book, please reach out. I would love to hear from you.

email: zobskewed@gmail.com

Twitter: @zobskewed

Web: https://zobskewed.com

Until next time…

On becoming the person your team needs you to be

Standing out and leading in tech and beyond

by Joe Zobkiw

"I am not a professional coder myself, but work with them often and found this easy-to-digest collection of technical (and not so technical!) insights to allow me to communicate and work much more effectively with the creative coders I engage." - Halsey Burgund, Emmy-Award Winning Interactive Director

"I can't believe this hasn't existed before" - Jane Doe

"WTF?" - Joe Doe

"Just an old grumpy C programmer trying to stay relevant" - Twitter

A Glorified Typist Publication