33 Things I Wish Somebody Would Have Told Me

A programmer's guide to quality code, great work relationships and respect.

Aaron Saray

Aaron Saray

http://aaronsaray.com

Hello. My name is Aaron Saray, and I.. am a Programmer.

I know - that just sounded like an admission of guilt. And, partially, it was. I'm guilty of a lot of mistakes, bugs, and failures.

But instead of feeling sorry for myself and burying my head in the sand, I paid close attention to the ups and downs of my programming career and decided to compile my lessons-learned in this online book.

My goal is simple: I want to make sure that other programmers *(like you?)* don't make the same mistakes as I did. If I can save you just one mistake, one problem, one sleepless night, then the years putting this book together have been worth it.

Are you ready to find out what the *33 Things I Wish Somebody Would Have Told Me* before I began my programming career?

Table of Contents

Introduction

When I was very little, I thought I wanted to be a mathematician. I can't remember exactly why, but I think it had something to do with multiplication tables. Or, perhaps, because of the timed tests where I solved one hundred addition problems in sixty seconds. My father told me a story from his childhood where he broke off the corner of a 6 sided die and then rolled it hundreds of times. He kept track of the result in a notebook. He was very interested in how changing the shape of the die affected the results. Maybe I inherited math skills from him *(one of the better traits, but, I'd love to give back the snoring and bad eyesight)*.

I was also very fond of Tinker Toys, Lincoln Logs, and Legos. I enjoyed building things, I was good at math. Perhaps I was on the path to being a structural architect?

During the 80's recession *(and afterward)*, our family was very poor. I had a few toys, but my prized possession was the hand-me-down Commodore 64/Vic20. This computer was pretty awesome. I mean, it was, if you had a hard disk, removable diskette system, printer, tape drive, or anything besides just the core machine. Mine, however, hooked into an old black-and-white TV and booted to a screen that simply said "Ready." I had no games, no permanent storage, and only a complicated manual describing the BASIC language. If I wanted to play Blackjack, I'd have to program it myself.

I did just that. When other 8-year-old children were still fantasizing about the newest MicroMachines and Teenage Mutant Ninja Turtle action figures they could collect, I was programming Blackjack, a pong-like program, and even "Go fish." And, I was living in supreme dread over thunderstorms. You know, thunderstorms - those great summer treats that knock out the power in rural Wisconsin fifty percent of the time. No power meant losing my program and having to start over. Oh, and I just hope that I had a chance to write in my college-ruled notebook all of the tens of thousands of lines of code to this latest game. If not, I'd be giving a new meaning to "rewriting" the software.

Yet, I still loved it. I loved everything about computers. I was able to use my building and architectural planning talents and my mathematics skills. *(Plus, nowadays, how many Mathematicians do you know? I know I'm crazy, but I kind of like options when I'm picking a job.)* To this day, I still love programming. Any language I can learn, any problem I can solve, I'm ready to take the challenge. I'm here to stay.

As my career has progressed, I've had a lot of chances to work with different teams, join unique companies, and mentor up-and-coming programmers. This diversity has driven home the simple fact: not everybody is like me. Not every programmer has the same talents, skills, and comprehension of processes and procedures. Everyone is different. Some things that seem common sense to me may not be to other programmers. And, of course, vice-versa.

These differences inspired me to write this book. The topics in this book are backed by years and years of research, and decades of experience - and mistakes. I'm incredibly proud of my journey so far as a programmer and I look forward to the upcoming decades that I have left in this field. I'm excited to see what new technologies my team and I will be leveraging even five years from now. Can you imagine the innovations that are to become common place in seven or ten years from now? While I'm saddened that some of the technologies I've learned are now irrelevant *(I'm looking at you, BASIC)*, I know that I'll continue learning new technologies to keep myself current.

But this book isn't about technologies. If your entire bookshelf is filled with various language and tool technical books, this book is not the competition. This book is about the core concepts of programming that are language agnostic. These are the things you learn from programming in any language, in any role, with any responsibility, project size, and budget. See, while the technology comes and goes, the base skills of being a programmer are the same. Great programmers can swap technologies while continuing to build the core skills of logic, architecture, design, planning, and communication.

And that's what this book is for. *33 Things I Wish Somebody Told Me* is a look back on my career, growing as a programmer, unrelated to each technology I worked with.

These are the things that I wish somebody told me. Perhaps if I would have known these things *(or read this book back then… Doc Brown, ready your flux capacitor)* I would have had an easier road and would be farther along.

Basically this is a book to help you escape those embarrassing **"you should've known"** or **"you wouldn't do it like that"** moments. You know exactly what I mean - when the boss or the senior programmer points to your code or software, everyone's staring at you, and your face suddenly flushes. Wow - is it hot in here?

This book covers both technical process concepts as well as some mental abilities that any successful programmer needs to have. No matter what your chosen language is or where you fall in the experience ladder of programming *(or even managing programmers)*, you can always get better. We can always get better.

I took a deep look at both my successes and failures to create the topics for this book. I am passionate in my quest to help mentor and educate other programmers to be successful. I'm hoping to help groom programmers to become great.

These are the *33 Things I Wish Somebody Told Me*.

The Voice of This Book

Some of my best friends are also my co-workers. I mean, I do have some non-nerd friends, but most of them are big geeks just like myself. They think about code even when they should be relaxing. They envision redesigns for our web applications while watching TV. We help each other out. We all are friends *(even though, I technically am the boss for some of them)*.

And, because you, dear reader, are probably also a big geek, I'm going to call you friend as well. We're friends now. I'll be talking to you directly. I'm not going to go abstract. You will hear "You should do this" and not "The programmer should do this." As you can probably tell already, this book's composition and writing style is a bit more relaxed. I meant it that way. I want to have a conversation with you. In fact, I hope to hear from you before, during, and after reading this book. Find me on aaronsaray.com and let's chat. Tell me your thoughts. I'd love to clarify anything - and even better, I'd love to hear your success stories from implementing some of the 33 things.

I'll tell you a secret about my bookshelf. I'm scared of it. I see some really great books that have some really good theory that I need to crack open. That's my next step in becoming a more skilled programmer. But, while the information is great, the books themselves are dry and boring. I know the programmers and computer scientists that wrote them are great people, but I just can't seem to read them cover to cover. After a few minutes, I get distracted. Look - there's a fly on the wall! Let's play tug-of-war, dog! What's new on Netflix. I just can't stay connected. Some publishers understand this problem. I won't name names, but one of them has put pictures of the writers on the front covers of their books to make the connection with the reader. I mean, now that I can see a black-and-white awkward photograph of the programmer, I'm now going to be much more connected and enthralled with every single page of useful, but somehow still boring, information, right? Sorry! *(If you do your research on these books, you'll see that I REALLY know first hand what I'm talking about.)*

In the following pages, there are some great concepts to learn with some technical backing, but that doesn't mean we have to have a snooze-fest. Since you've made the commitment to take time out of your busy schedule to read 33 things about becoming a great programmer, my return on your commitment is two-fold.

First, I'll try my hardest to educate with the 33 things. Trust me, these are huge reflections from myself. You're getting a part of my soul with this book - blood, sweat and tears all went into this. Ok, well not blood. Lots of sweat. Perhaps a tear or two, but that was allergies… yeah!

The second thing I guarantee is that I'll write in a voice that is conversational, story filled, and interesting. I'll embellish where there's a need, but cut out the extra BS when it's not needed. I use stories and my experiences extensively because that's how I learn. And, I find that a lot of programmers learn this way as well. In the same way that looking at code samples can be used to learn a new language, stories about a programmer's decisions and consequences are useful. I don't have a page-count in mind, and that's a good thing. That means everything in the final product is exactly only what I think needs to be here, written in my true voice. If you read this book, you now know how I talk on a daily basis - whether that's a positive or negative, I guess I don't know!

Organization of This Book

This book has two sections: one for programmers and one for managers of programmers. I thought it was important to include both because I've seen a lot of programmers be promoted to managers without having any resources that are aimed at the background where they came from.

The section for programmers focuses on a number of necessary skills. These range from building social connections and soft-skills, learning to become better technically at your craft, and communicate in ways that show confidence and garner appreciation.

The managers section is useful for both prior programmers and those with no programming experience. It aims to give information about leading groups of programmers, discovering programmer motivations, and how to interpret actions and events that happen on a development team.

I would suggest reading both sections, no matter which position your current role is. Both have useful tips and tricks, but with different aims.

Programmers

I’ve seen so many talented programmers get the short end of the stick. They get passed up for promotion, make a lot less money than they deserve and work harder than they need to. This makes me sad. But, instead of just ignoring the problem or hoping it gets better, I made this section for them.

The business of programming involves much more than just code and applications. There are ways to communicate, processes to develop, and habits to form. Once you can get a grasp on these ancillary concepts, it can literally make your coding life better, easier and more valuable. You might have great skill now, but this section is designed to help you transform into something even better. Instead of a programmer with great skills, you can be known as simply a great programmer. Skills and technologies come and go, but great programmers are worth their weight in gold.

If you’re a manager, you shouldn’t skip over this part. You may be able to benefit from the advice a programming veteran has for your team. By reading these tips, you should get a better insight into the struggles programmers can face as well as the ways you can use to help your team overcome them.

Are you ready to become a great programmer? Let’s find out some things I wish somebody would have told me!

#1. Learn From Everything

AND LEARN TO LOVE THE HATERS

I don’t know when it happened, but sometime in my life, I suddenly had this feeling that I knew everything. We all experience that sort of feeling here and there in our lives, I’m sure. Perhaps you’ve attained the best or highest level in a particular field, or you are the most talented person on your team at work. You might feel on top of the world, which is not bad. But the bad part is when you start thinking there isn’t more room for growth and that you can’t learn any more.

You can always learn. But, learning doesn’t always come in traditional vehicles. It seems that most people think of learning as academic laden time wasted. You’ll need to get this book, take this class, do these particular tests, etc. That’s not the case all the time. *(Don’t get me wrong, even though I have my student loan payments still looming, I do see some value in the time I spent in the academic world.)* But learning comes in all different forms and from many different places.

I’ve definitely seen this firsthand. My parents used to make great decisions. They taught me things while I grew up that really primed me to make positive decisions and continue to grow my career. Something changed, however, and they started making very poor decisions. Fast forward to now, and I think most would agree that the final product of their combined decisions has lead them into poverty, depression, and bouts with alcoholism. I’ve been very bitter watching this decline. I’m even more upset considering I have two sisters that still live with my parents.

One of my sisters called me a few months ago and wanted to talk. She lamented about the decisions that my parents had made. She told me about the things that they had told her that she could and should do with her future. She seemed to be of the mindset that mom and dad were a weight holding her down. My first reaction was to agree with her - how dare they make her feel this negatively!

Luckily, I quickly reframed my thoughts while she was telling me the most recent news in her life. I thought about all the positive things that my parents had taught me. I looked at the completely obvious mistakes that they had made in their lives up until now, and I decided I knew exactly what to say to my sister.

"You need to imitate the positive things they do and say, learn from their failures, and disregard the rest." I went on to explain what I meant to her for the rest of the conversation. I described various events that happened while I was growing up, and what I learned from my parents' reactions. I encouraged her to continue to do the best she could, follow her own path, show her parents respect, and never forget that there is something to learn from even the bad things they said and did.

I think this type of scenario is something that we can all relate to, whether or not you can relate directly to the sibling/parent dynamic. I highly doubt that there isn't at least one horrible boss that you've had in your career. You've probably had downright despicable coworkers as well. That doesn't mean you should block them out of your mind. Instead, listen to, but do not absorb what they have to tell you, and learn from it.

At one company I worked at, most of the programmers work remotely. One of my peers was telling me how he pulled in roughly double my salary. I asked him how that was possible. He told me his little secret: He actually worked full time at two jobs at the exact same time remotely. He somehow was able to balance both workloads, while giving 50% effort. He never admitting that he was doing this, and no one seemed to suspect a thing. I found this particularly disrespectful to his employers, and I found that I could never trust him again.

I did learn something from meeting this programmer, however. Now, as a manager, I make sure everyone on my team knows they can speak openly about other opportunities they have. I learned that programmers may want to take on more responsibility, freelance, or experiment with different technologies. I maintain this open dialog to make sure that everyone is comfortable and that I won't have people going behind my back to do things. These covert actions only damage our relationships and destroys productivity.

There are other ways to learn from people, too. I think of a few times I've given a final product to a customer and heard the worst words a programmer can hear: "I hate it." Are you kidding me?! I put in months of my time, my heart, my energy, and my creativity into this product and you quickly tear this all apart with three little words. Especially if you work directly with individual end users, I'm sure you've heard this before too.

Usually the first reaction is defense. You may want to retort that you've designed the application or program exactly to their specifications or argue that you weren't given enough direction. However, that's going to get you nowhere. Instead, let's take a look at this from another direction.

There are just a few, very basic human emotions that are the root of all the ways that we feel. *(Warning: I'm not a psychologist, so don't hunt me down and yell at me.)* For me, they boil down to levels of happiness and levels of passion. *(For those designery people in the house, think of them as brightness and saturation.)* Hate comes from passion. So, when someone tells you in an angered tone that they hate your work, it's just misdirected passion. I know it's hard to believe, but it's true.

So, your job is to figure out what is this customer's passion. It's not logical that they would absolutely hate something that you've created - most people are not psychopaths or that emotionally unstable. What's the deal? What is it that they're passionate about? This is what you may have missed. You didn't know what they were passionate about, and because of that, you either forgot to emphasize it or left it out entirely.

During some of my freelance web design career, I came across one particular customer that really helps illustrate this point. After the design had been completed, I met with the customer to show off my work and their new site. As I explained it to them, they looked very unhappy. Finally, when I had finished talking *(and I'm sure I talked longer, and then, increasingly more frantic than I should have because of the negative facial expression on my paycheck… I mean client's face)*, they said those fateful words… "I hate this design."

I knew this couldn’t be the case. I didn’t make horrible “hate-worthy” designs, so I delved in deeper. I told the client that I understood, but I just need to get a little bit more information out of them. I asked if they liked the color scheme and how I featured their logo. That was ok, said the client. Ok, do you like the pages and menu scheme I used? Yup, they said. Finally, I touched on the passion point.

“Do you like the contact page?”

“No! I hate how you didn’t ask for the phone number, and why do you have this google map so large on the page! We don’t need them coming here, we just need to talk to them. Remove that!” So I said I completely understood. If I removed that, restructured it to emphasize calling or emailing, should we move forward with the design? Yes, they emphatically responded.

I had found their passion.

In this case, the customer was vehemently opposed to having people come to their retail location with customer support issues. They had a whole call-center available for that. The phone number was particularly important because that’s how the relationship was built in their business. I had found out what was important to them, tweaked that messaging, and the whole design was now something they loved. I was able to look into their “hate” for the design, find their passion, and learn.

Besides serving this customer, I learned even more from this experience. When I designed future contact pages, I thought to ask the customer what kind of contact form they appreciate the most. This helped me feature the primary contact information in a way that helped convert customer complaints and sales opportunities the right way.

There is always something to learn from every interaction we have with clients, customers, and peers. Remember to always look at these interactions, whether they are positive or negative, as opportunities to learn. Extract the knowledge from the situation, reflect on what you could have done, what you did do, and what you’ll do next time. Learning from everything, not just technical documents and manuals, is what makes a good programmer great.

#2. Log Everything

LIES, DAMN LIES, AND LOGGING

People lie. I lie. Even you lie. So, what can you do about it?

Log everything. It's very simple. If something happens, log it.

Chapter done.

Users will lie to you. And the worst part is, they don't even mean to. When I used to do technical support for a local Internet Service Provider, I ran into some of the worst liars. Now, mind you, these were good, quality, hard working people, but they were liars. They didn't mean to be, but they were.

I can think of one particular example that makes me laugh still. I was working with an older lady and she was having problems getting connected to the Internet. I asked her what version of Windows she had. She said Windows ME. I started giving her tips to get to the control panel and look at her TCP/IP settings. Each time I gave her a new step to complete, she sounded somewhat confused. She sometimes would repeat different words than I had said. For example, when I said "Network and Internet Settings" she would say "Ok, opened Internet Options." Towards the end of this troubleshooting step, I was telling her to check on a specific Windows ME setting. She swore up and down that she couldn't see the option I was referring to. I asked her again what version of Windows she had. She rather angrily confirmed that she had Windows ME. I couldn't seem to solve her problem because the remaining step just couldn't get accomplished if she couldn't find the option I needed.

She finally became so angry that she put her daughter on the phone. I asked the daughter what she saw on the screen. She described the placement of all the icons and the labels. I asked what version of Windows it was and she said it was Windows

98. I said her mother had insisted it was Windows ME and that's why we were having problems solving the internet connection issue. She laughed and told me that her mom had just saw the newest computer commercial on TV earlier that day. Because she couldn't get on the Internet, she was thinking of buying a new computer. This new computer came with Windows ME installed. So, because her mom didn't know much about the different versions of Windows, she had just assumed from that point forward that she also had Windows ME on her older computer.

In this story, the mother lied to me. She didn't mean to and she didn't even know she did, but it really was a lie. *(Some would argue that a lie is purposeful, and this was just a mistake, but I disagree. In this case, this became more than a mistake and morphed into a lie because she insisted on a false truth without taking the steps to verify it.)*

When I create software, I'm very generous with my logging. I like to log everything. I log errors, debug messages, and even changes to records, objects, and data. I try to build historical paths so that I can audit the changes that the system has gone through. *(Now that data storage is much cheaper, how can you justify not storing this information?)* This helps immensely with tracking down what users do before, during, and after a problem occurs.

I've had a number of instances when support requests were put in and my logging helped track down the problem. I can think of another scenario where a customer service representative at one of the companies I worked at was lying to the programmers. There was a multi-step process that needed to occur in one particular order for both business and technical reasons. Other representatives were able to do the task accurately. Nothing bad happened with their customer records. However, when this particular individual did the task, she always had to submit a support request because "the software is way too buggy." We asked repeatedly if she did the process the right way. She swore she did.

Looking back into the log files, we were able to determine when she had logged into the website and what data she had changed. We were able to recreate her tasks on our test system. Then, we were able to correlate those steps with timestamps on other systems and found out why she was having the issue. She was repeatedly doing the tasks in a different order than what was required and lying about it. Once I presented this information, I closed the support issue as not relevant. I tried to give her the benefit of the doubt - she didn't realize she was lying. *(Turns out, a week later when I was doing follow up on the support request, I found out she was no longer working for the company. Hmmmm.)*

When we look at ourselves, we can't imagine we'd lie *(or make mistakes and let them spiral out of control into "lies")*. Yeah, normal users do that, but certainly not us. Software programmers are far more honest and accurate. I know that I would never lie.

Except, I did. I lied to my car mechanic. He was very gracious in how he caught me, making sure I knew of my mistake while not embarrassing me too much *(in public)*.

I was having brake problems. Whenever I slowed down, I heard a horrible grinding sound. It felt like the wheel might even fall off. I brought my car into the mechanic and did my best to describe the problem. He asked if it was all the time or just when I hit the brake hard. I said all the time, no matter what. He questioned if it was when I turned left, right, or all the time. I said it was all the time. He took the car and went to work. *(Maybe he knew right then I was lying. The loaner car that he gave me was… interesting at best. Purple is not my color.)*

Later, my mechanic called me with the bill and an explanation. He said it looked like I had problems with my right-front brakes and suspension. He said these would have been apparent with hard braking and when I turned to the right. He said when his team tested it, they found that right away. After thinking about it, I put it all together. I had lied! I was so angry about having car problems that the specifics had clouded in my head. The more I thought about it, I realized what had happened. Normally, I heard the problem when I was heading home from work. I had a stressful job at that time so I was particularly aggressive when driving home. Plus, my road was a right turn off of a very busy main road. I drove the road too fast, had to turn right into my subdivision and would usually brake very hard. If I would have told him this, he could have easily told me what the problem was. Instead, I had *(accidentally)* lied to him. I guess I’m not immune to this either.

Log everything. When in doubt, create a log of the problem. When something doesn’t match the workflow, log it. In fact, start creating software with logging first. If you really need to conserve disk space or have a heavy load where extra logging deeply affects performance, initiate multiple levels of logging, so you can adjust what amount of logging you need based on the situation. Another thing that I’ve implemented in some of my projects is various levels of logging targeted per user or per platform. This allows me to turn on extensive logging for only one user in the system. All in all, you’ll never find that you regret having created more logs than less. It not only helps cover yourself against false bugs, but it helps you track down existing ones faster, too!

#3. Programmers are Customer Service

BUT SOMEONE FORGOT THEIR HAIR NET AND RUBBER GLOVES

I used to work as a family-style banquet waiter. I would bring out large trays of food and serve from head table down the sides. Never would fail, by the time I got to the last grouping of people, they were irritated. Why did we have to wait this long for our food? As I listened to their complaints, I couldn't help thinking "you know, I'm pretty certain that you've eaten before… and that you've had at the very least one other meal today."

The restaurant customers seemed to have no respect for me. They treated me like garbage. I couldn't believe the attitude that some of them had with me. Just because I am serving you doesn't mean you should treat me as if I'm second rate or scum. I wondered how many of these people had ever been in a service industry before. I found out later that some of the customers were lawyers or financial planners. I couldn't believe they would treat a fellow customer service provider this way.

As I learned more about the various groups of people who came to my restaurant, I began to realize that a lot of them had 'family money' or had been put into their positions by default. They never had to start 'at the bottom' - that is, the place that they seemed to think I was at. These men and women had lost the wisdom of our elders that proclaimed that to truly serve another human being is a privilege. The act of one person choosing of their own free will to meet the needs of another is to be respected.

As my career goes on, I look back at these memories with a particular fondness. I recognize that those experiences had shaped me in ways I am still discovering. For example, if you are now currently in a customer service job that allows for tipping, you can tell others who have had similar jobs. At the restaurant, the patron far removed asks friends or uses a calculator to figure out exactly what that 18% mark should be *(or some do less!)*. Generally, you can recognize your fellow restaurant veterans by the quick way they calculate the tip: move the decimal one spot, multiply by 2, and make sure that's the least amount left on the table. That simple act of 20% tipping, or more in some cases, is usually a sign of someone who has been there before and understands the demands and the importance of good customer service.

As a software developer, you might not yet draw the correlation of these stories to your daily project work. After all, while the restaurant server is bringing food to the table and cleaning up, you are actually creating something out of nothing. Your thought, your design, your artistry creates a piece of software that wasn't previously available. If you're on the cutting edge, you may be even creating software and algorithms that have literally never existed before in human history. You label yourself as a product producer.

You are wrong.

All programmers are in customer service. We are the epitome of customer service. Let me describe what I mean. A good portion of software is meant to simplify or automate other processes or systems that already exist. Your customer is asking you to refine their business process or vehicle. Think of it as taking your car to the mechanic to get a tune-up. Your mechanic isn't going to give you a brand new car, he's just going to make sure this one works better, faster, and more reliable *(and call you on your lies like mine did!)*. This is what you are doing as a software developer. The purchaser of your software already has a business, you are just creating something to propel it further and faster.

Depending on your industry, however, you may be making a brand new solution to a problem by creating a new piece of software and defining the process in which it can work. However, I still insist that you are customer service. The root of this statement again is that you're solving a problem. If it weren't for people who had a problem, you wouldn't be creating a solution. Once again, you are serving your customers with a service solution. If they didn't need this service, you'd not be in business.

There are many examples of software that were made without a clear solution to a problem. These entrepreneurs or programmers lost sight of the main goal: provide a customer service solution. They may have made an amazing product, but since it wasn't serving a customer need, it lost steam after its initial release. Before you know it, the non-customer-service-centric business failed.

When you look at this book in its entirety, you'll actually realize these are just chapters on increasing your level of customer service. In the same way that we're likely to appreciate a boutique with knowledgeable salespeople or generously tip our favorite coffee house, the same goes for programming. Your job is to go the extra mile. Not only should you be solving the problem that client has asked for, but you should also be looking for ways to continue to serve them better.

Expanded customer service comes in two forms, both profitable. The first is the most clear to understand: the upsell. While you're solving the problem for the client, evaluate the rest of their processes that your customer has made visible to you. Consider if you can make improvements to help save the customer even more time and money. You may want to offer them that service as an extra *(a veritable "would you like fries with that?")*.

Here's the kicker: don't offer to just fix the problem, but take the guesswork out of it for them. Describe the problem you can solve, state your solution, and give them the benefits of implementing the solution. For example, you may say that you noticed that their representatives could spend up to 20% of their day using an old, outdated software interface. You could update this interface to make it easier to use, therefore shaving an estimated 50% off of the time each representative spends accomplishing a task. This reduces the amount of time spent to 10% of the day, realizing a 10% savings and productivity boost over their entire staff. How can they say no to that? That's an upsell, securing more work for you. Better than that, you've solved another problem for them. That's customer service.

The second form of expanded customer support in the programming world is simply exceeding expectations. I've seen both right and wrong examples of this. First, let's start with the wrong example. A colleague of mine was creating a new website for a non-profit organization. He really felt passionate about their mission, so he not only created the website, but built a donation interface using PayPal too. All this for free! The client didn't realize this new addition at first. Days later, they did - and it was very bad. They had not yet finalized the paperwork creating their non-profit organization and tax-deductible donation status, but had received donations to their PayPal account already. The interface had falsely told donors that they could claim a deduction on their taxes for this donation. The client was obviously irritated that they now had to contact those donors and let them know about their current situation. The donations were not tax-deductible yet. This programmer had tried to exceed expectations, but he had done it in the wrong way. Needless to say, he has not been contracted to do any more work for them.

There is a right way to exceed expectations, however. A friend of mine was working with a client who had a rather small budget. Because of this, they were generally fine with fixes and updates being applied in a queue mixed with other higher paying clients. The client understood that sometimes their fixes would take a few days, even if they appeared to be urgent. One Friday night, however, the client discovered a bug on the website. He contacted my friend via email to let him know about it. By now, the relationship was established, expectations were set, and the client knew it may take a few days to get the fix applied *(especially since it was the weekend)*. However, my friend happened to have some time available, so he fixed the problem that night, sent a follow-up email, and invoiced the client.

As usual for this particular client, the invoice was paid immediately. The owner responded to the follow-up email with nothing but the highest praise for the work that was done on a Friday night. My friend had truly exceeded expectations and made the client very happy. It doesn't end there, though. A few days later, my friend received a letter in the mail from the client thanking him again for his quick work. Included in the letter was a $250 gift card to a local home improvement store.

This particular outcome is a rather unique one. The only other similar scenario that comes to mind was of a colleague who was once presented with a disposable cooler full of fresh steaks. Regardless of if you see an immediate material return, the main point is to exceed customer expectations through great customer service no matter what. This will help make the difference between choosing you in the future to do the job or trusting you to run a huge project and going a different route. What's even better is the indirect outcome from this great service: when you increase your rates or ask for a raise, it's much easier to stick with you than to attempt to find someone else who may not have the level of service you provide.

I think its incredibly important to approach every programming task as a new foray into the realm of customer service. Remember, if these customers didn't have these service needs, we wouldn't have jobs.

#4. Real Talent is Making Things Simple

NO, PLEASE. TALK TO ME LIKE I'M A FIRST-GRADER

I remember when I used to have to time the start of a television movie just right so I could fit the entire broadcast onto a VHS tape. I knew that for longer movies, I'd have to either lower the quality of the tape recording or make sure to pause the recording during commercials. I had to make sure I bought the right VHS tape, connect the coaxial antenna cable to the VCR correctly, and be vigilant about making sure that the tape didn't become jammed.

Apparently, I was lucky. I was told that watching movies and recording them used to be much harder than this. Fast forward today and recording a television show is as simple as finding it in your listing *(present or future)*, and clicking record on your DVR. It has plenty of space to record the broadcast, even in HD. You don't even need to be awake or around. This is pretty simple.

If you look closely, the trend in all technology is directly related to making things simpler. The interfaces become simpler *(it's not that difficult to use an iPad - 2 year olds are doing it)* and parts of our lives become simple or automated. Marketing is poised around the concept of simplicity. Rarely do you hear something like, “On your Android mobile phone, you can easily install any software package by searching for it in Google Play Store. Or, you can find the proper .apk and upload it. You can also connect using adb and push packages to your phone over usb. You have so many choices!” Instead, marketing says, “the iPhone syncs all your apps from the app store - you never have to worry about choosing or installing anything new again.” Everyone seems to value simplicity.

Let’s move beyond the concept of programming and software. Success is measured by simplicity. When someone is selling a wonder-cure or a get rich quick scheme, it’s all about how simple it is. *(No one wants to buy a product that requires 120 steps to get payback. Give me 3 steps or less, or it’s not worth it - not even for a million dollars.)*. Speaking of rich, have you ever noticed how refined and simple the truly rich person’s belongings or clothes are? Compare the $10 speckled, multi-colored button up shirt from Walmart *(a budget-conscious choice)*, to the flashy designer t-shirt *(someone looking to demonstrate the illusion of wealth and success)*, to the simple cotton woven dress shirt with quality stitching and no logos *(look at how real millionaires dress)*. The plain pin-stripe suit that gushes luxury by simply having equally aligned vertical striping. Simple, elegant, successful.

Real talent is making things simpler. Great programmers create simple solutions. Let’s break this down into code and user interface.

The best code is the simplest. Very elegant solutions often come in simple, small packages. When creating code for your project, continually concentrate and focus on making sure that the code is simple. If you start extending and mangling the code in such a way that only someone with a deep history of the project can work on it, it’s no longer simple. It’s no good. If this is the path you’re following, it’s immediately time to refactor. Even when you’re creating it the first time, keep an eye on the simplicity of your code. How can you do this in an easier manner?

Within the concept of simple code comes another warning: don’t try to be too clever. Actually, don’t be clever at all. Whenever I hear about a programmer coming up with a really clever solution, it’s usually way too complex. A great programmer is excited about the solution, the technology implemented, and the solution they came up with. However, they should also be proud about how simple the solution is with the right code, not how clever the solution is by twisting and turning that square peg of code into that round hole.

Along the lines of being too clever, short and simple code should not be created just for the sake of creating simple code. Have care not to sacrifice the readability, art, and flow of your code while trying to make it simple. I've seen this interpreted wrong a few times: "Oh, to make it simple, I named this instantiated class 'x'." Looking back, no one knows what 'x' means. Yes, it is simple, but it doesn't mean it's good.

The simplest and most elegant code won't guarantee success of the project if the users can't seem to use it. The second area to concentrate on simplicity involves the user interface.

There are a lot of competing thoughts on what makes a great user interface. Some experts argue that the best user interface is one that anyone could use intuitively. Others suggest that in some situations, the interface must be customized to the workflow. Custom workflow interfaces may not be the most user-friendly, but they do guarantee speed within a custom job. I'm not going to pick a solution or solve this problem for you. This book is too short for that. I tend to lean towards the intuitive use spectrum, so that's how I'll frame the rest of my explanation.

A good rule to follow regarding simplicity in user interface is to test it with some non-stakeholders of the project. Remember, you're so close to the project that you won't be able to address the interface as good as someone fresh, new, and not as invested. With this, of course, comes the requirement that you must be open to the suggestions from the fresh pair of eyes. Test the interface with many people and see what resonates. Did this really work? Could something be simpler? Do they have to ask you questions to use it or can they figure it out right away? Are there any unneeded requirements to be overly precise when using the interface? *("Oh no! Please don't wiggle the mouse! You'll destroy everything!" I hate to be the bearer of bad news, but that means something's wrong with your design.)*

Creating simple user interface is important. *(For those programmers who have designers and UI/UX people to work on this with them, consider yourself lucky!)* The best suggestion I can have is to listen to the project requirements and then determine the need of any specified interface suggestions. Try to determine if the suggestion is mandatory or if its just the stakeholder's vision for solving the problem based on their limited experience. You'll be surprised: sometimes even your outside view can greatly simplify the interface that the project needs. When you aren't solely focused on the daily business, you can have that fresh view. Be ready to make the case to support your decisions, however.

A really great, talented programmer makes things simple. He or she can create artistic, simplistic, elegant code solutions with intuitive and simple user interfaces. Whenever possible, recognize that the simpler you can make something, the more it will be accepted and the better it will perform.

#5. Have Pride in Your Work

FOR THE TENTH AND FINAL TIME, YES, YOU ARE REALLY, REALLY GOOD AT DOUBLE CLICKING

I've said it many, many times. Show me a programmer who was never behind on a project and I'll show you a liar. There's always going to be times when a task takes longer than expected, something comes up, or there just isn't the possibility of completing the task in the original bounds. There are many ways to solve this problem.

When you're behind on a project, you could talk with the client or stakeholders and request an extension. You may be able to explain what happened, give them an insight into the new difficulties, or work out a feature compromise. Maybe that'll work.

Another way to catch up on a project is to cut corners. Instead of writing unit tests, skip them. Do not create any documentation. If you're not as familiar with the proper way of doing something, just fall back on old, less accurate methods. Maybe that will catch up the project.

Perhaps you'll find a programming genie, and your wishes will be granted. Back on time!

This chapter isn't about how to catch up on a late project or program. I'm not going to tell you what is the best way to stay on task. This chapter is about something else I want to stress: pride in your work.

Whenever you can, resist the urge to cut corners. Remember, this code *(and art)* you're creating is a reflection of you. Imagine if your mom or dad understood what you were doing *(perhaps they do, and then you're incredibly lucky)*, would they be proud of it? Would you be willing to show your programming mentor this work? Would you like hundreds of thousands of fellow programmers to download your code and use it? If you feel squeamish about any of these questions, you may be cutting corners.

Way back in elementary school, I learned the value of having pride in your work. I don't remember the specific teacher or class, but I remember the story. The class had finished homework and turned it in at the beginning of class. The teacher looked through all of our work, graded it, and handed it back to each of us. I remember not having the best grasp of the material, but I did the best I could. After the homework was handed back, many students looked visibly upset.

The teacher stood in front of class and said, "with the exception of Aaron and [some other student], I don't think any of you have any pride in your work. I gave you the grades you did based on both the answers you gave and the quality of your work. If you'd like, you can redo this assignment for a higher grade." After class, I compared notes with other friends. While I had more errors than many of them, I had one of the highest grades of my peers. I had been rewarded for trying my hardest and taking pride in my effort. I opted not to redo my homework, but I did study more to try to learn the answers to the questions I had got wrong.

As primarily an open source programmer, it is very important for me to have pride in my work. Many people download my projects, review the source code, and implement it in their projects or at their company. I need to make sure I do the best I can. I've been on the receiving side of poor work many times and it's not fun. Whether it's undocumented code, no help or FAQ, or poor logic and structure, programmers who didn't have pride in their work just seemed to have lower quality projects.

Even if you're not an open source programmer, you may run into future colleagues that have had the joy to work on your code. You will have gained more respect, even if you had bugs in your code, by having created very accurate, ordered, precise code that you are proud of. Hitting deadlines is important, but pride in your work is paramount. When you create quality work you're proud of, you'll see the ramifications everywhere. Peers will know that you are someone who can produce good code and are happy to work with you. Managers and clients will know that the product they get from you is of the highest quality and caliber.

Have pride in your work. Your work is a reflection of you. And while it may not seem like it at the time, this investment always comes back to reward you - or haunt you - depending on which path you choose.

#6. Solve the Right Problem

SOME GREAT PROGRAMMERS DON'T EVEN NEED TO CODE

Having programmers solve problems can be expensive. Telling them the wrong problem to solve can be even more expensive.

I have a great example of this. At one place I worked, the project manager gave me a detailed requirements document. Well, that is strange! Rarely were the requirements this detailed. I was pretty excited to have this set of requirements, though. Perhaps with less guesswork, my job will be easier! As I paged through the document, however, I became a little bit concerned. The amount of work here was staggering.

This project was requesting that I make changes to every single area of our system that interacted with a specific account number format. It required that each input, each validation, each sanity check and every service call be modified. This was quite a lot of work. I read through the document and estimated the amount of work like a good robot programmer. I gave the project manager back my quote. He was not happy.

To solve this problem, I estimated it would take 2 programmers roughly 5 months to solve the problem. At our billable rate of $125 an hour for the 1680 combined hours, that was a bill of $210,000. As one might expect, the project managers and stakeholders asked if there was any way that I could shave any time off of that. That's too much, they said. I asked for the requirements document back.

By this time, I decided to start billing for my quoting and analyzing. It had taken me hours to give the first quote, which was rejected. I racked up another 6 hours *(at $125)* requoting and analyzing everything. Then it hit me: they had me solving the wrong problem.

I asked for more detail about the account number problem. I had to go much deeper into the business to find out what the real problem was. It had to do with the way that certain accounts were created in the system. The chance of this particular scenario happening was as frequent as about 1 in 100. I suggested a safeguard against this particular process from happening. Instead of a programming patching the problem downstream, I suggested a process change to stop the flow at the origin. The only other necessity was a few checks and followup to make sure the new process was being applied properly.

The stakeholders were amazed at my solution. Then, they took the time to figure out how much the process change would cost. They estimated they'd have to train the 15 data entry and administrative resources for about 2 hours. This cost was about $30 an hour. Then, they'd be able to do a few checks here and there to make sure the new process was being followed *(considered a negligible administrative cost)*. The total cost of this change was 30 hours at $30 an hour, plus my 6 hour estimate: $1,650. Not only was this hundreds of thousands of dollars cheaper, the total cost in dollars was so close to my total of hours estimate, it was funny. Basically, our team would have to work for a combined wage of $1 an hour to solve the problem within the same fiscal constraints.

In this case, I looked at what the problem I was being asked to solve was, and decided that any programming solution wasn't the best thing to do. I reframed the project to focus on a different problem with a different solution.

Programming is not just about problem solving, it's about deciding what problems to solve. In this case, the solution came to me after a second look and a period of analysis. However, an even better programmer may have been able to catch that even sooner. Figure out what problem actually needs to be solved, and solve that.

Earlier in this this book, I related our programming jobs to customer service. Even with startups, we are customer service. We are serving the needs of someone. Another way to look at this is that we're solving someone's problems. In fact, when we're not successful, it's almost guaranteed that we're solving the wrong problems. A great piece of software trying to solve a problem that isn't needed or doesn't exist only will be a great piece of software forever. It will never see full potential use. It'll be a great piece of software, but never successful.

Another way I look at this concept in programming is related to optimization. So many times we find ourselves trying to eke out the last little bit of performance from our software. Sometimes, it can be easy to go down a random rabbit hole and do something called premature optimization. That is, we invest in optimizations for problems that may not exist yet. We're solving the wrong problem. Instead, look for the low hanging fruit and attack that. Solve the problem that needs to be solved, that is right in front of your face, and which has a solution that makes an impact.

Want to be a great programmer? Just don't solve problems - solve the right problems.

#7. Design for Your Users

STOP BEING SELFISH. THERE'S A WHOLE LOT MORE OF THEM AND ONLY ONE OF YOU.

It is very rare that a programmer will find themselves in a completely siloed world where they never make any decisions about how the user interacts with the software. While you might not be the UI/UX creator, designer of the application, or architect of the process, there are still times when you do make some choices that affect the user interface. So with that in mind, I have something very important to tell you.

Stop designing like a programmer.

Seriously, stop it.

Half of the readers will know exactly what I'm saying. You've seen it in your own work. You've worked with a great designer or layout specialist and they've made tweaks to your design that just made sense. That "Ooooh" moment where you saw your work polished. That's what I'm talking about. Pay attention to that.

The other half of readers have no idea what I'm talking about. You are the ones I want to reach out to now.

If you are a programmer, and you're designing interfaces, chances are you're doing it wrong. What could be worse is if you don't at least recognize this.

When I first started in web development, I thought that I could be all things to everyone. I could program the backend while simultaneously making a great layout, logo, and sales-based design. I pumped out a lot of designs that I was proud of. I saw no problems with them.

A full time designer colleague looked at my portfolio and said, “You really are a programmer, aren’t you?” I asked him what he meant. He explained that I really focused on lining things up in rows. What’s wrong with that, I argued. He smiled and went on, “your colors are all very monochromatic. There is no eye flow and concentration. You seem to use attention grabbing things just to fill up space.” I still was pretty incredulous. I didn’t think I had done that bad. I could use the site perfectly. He finally tamed my temper and indifference with his last sentence:

“You’re not designing for your client. You’re designing for yourself.”

This one experience really kicked off something for me. I suddenly had the desire to do design only when I absolutely had to. Whenever possible, I would find people who were skilled in design to offload that work to instead. I did, however, pay incredible attention to the final output. I set up some basic guidelines to follow for both myself *(when I had to)* and when I reviewed my partners’ work.

First, don’t design like a programmer. Programmers think very logically. One form element goes under the next, under the next, under the next. Each step should have its own screen. Eyes go directly from left to right, one line at a time. You start and complete a process, never abandoning it in the middle. All things that programmers do and think, but all design principles that we shouldn’t employ.

Designing like a programmer is not sometimes bad - it’s always bad! I think of some great tools that I use for programming that were created by a programmer, but also designed by a programmer. The core functionality does what I need it to do, but I never really feel happy about using the tool. I look at other products that were obviously developed by a team of designers and programmers, and somehow they just seem to function better. They make me feel more efficient and happier when I use them. They might even have less features, but I like them more. The difference here is that while the programmer knew what needed to be accomplished, he did not understand usage patterns of people. It was designed with logic, not with usable sense.

Second, consider the anthropology of your user segment. Most products are not used by everyone of every demographic. You may wish that was our target market, but you should have a general idea of who our audience actually is. Design for them. Architect for them.

It's important to get the demographic information from the sales or marketing team as soon as you can. This is how you learn to target your product. I'm not saying demographic lock-in *(like forcing the font way too large for an older subset of users)*, but I'm saying demographic concentration. Learn who your users are, design for the most of them. Don't design for yourself. You are one user and hopefully less than 1% of your market! That's crazy to cater to such a small segment.

Learn how your users generally use your product. You might have an idea of the demographics of your users, but what is special about them? For example, users under 20 would rather send messages via Facebook than via email. Ages 20 to 30 seem to use text messages for anything from casual conversation to important business information. Those above 30 still consider email authoritative. The point is, knowing how the users would want to use the product is just as important as learning who they actually are.

Stop designing like a programmer. To have a successful product, design for your users. The end result is something that you'll love to work on and use yourself while your users find it intuitive and valuable as well.

#8. When In Trouble, Break Up

PROGRAMMING AND NOT RELATIONSHIP ADVICE

If you think way, way back to learning your alphabet, you probably learned it from a to z immediately, in one go. You didn't say 'a b c d e…' over and over until you learned 'f g… h i…' No, you knew it immediately from start to finish.

No? Then why do programmers force themselves to do everything in complete packages? When expected to add a new system or interface to the software, most programmers will try to build the entire thing all at once. I would argue against this philosophy of start to finish. Instead, try something different. There is value in breaking things up.

One of the programmers on my team had to work with our software's authentication system to integrate an LDAP solution. He was rather new to our custom authentication model and to LDAP. He began the project by making a new interface for our authentication system and then applying LDAP settings to it. He was stuck almost immediately. His lack of understanding of both our authentication system in addition to his recent introduction to the LDAP protocol combined to make even the simplest of steps overly complex.

I told him that he was trying to do too much. Instead, let's break it down into pieces. First, spend some time learning the authentication system. Do not create the LDAP process yet. Just authenticate correctly, incorrectly, try to break it, etc. Watch how the system responds. Use debugging features to track through the code to see how it works. Learn the current system. That's the first step.

The next logical thought for most programmers is predictable. "Ok, after that, then I'll create my own interface in this system and add the LDAP support." But, that's wrong, I cautioned him. After you understand how our system works, create a new, very simple application. Create your own very simple, non abstract, hard-coded LDAP routine to authenticate. Use this to learn how the protocol works. Authenticate, fail, try to break it. Repeat the same processes as you used to learn our authentication model.

Only after both of these steps were complete, I told him, then combine the pieces. By now, he would have known how our authentication system works and how LDAP works. The combination then is just purely adaptive programming. He doesn't need to learn multiple things at once. Instead, he can focus on one thing at a time now: integrating two known technologies.

Sometimes we can be in a rush to implement new technology. Just copy and paste some code into the application and we're good to go. However, more often than not, this introduces more bugs and more false-starts than learning each part separately.

Another example I work with often has to do with complex HTML and CSS layout issues. We've created a very complex site with a lot of features. The final piece of the puzzle is being added, but the dimensions and placement of this feature are just not working right. No matter what, the web designer can't seem to get it placed properly.

As you can guess, there are two approaches here. First, keep pounding your head against the wall and try every conceivable combination possible. You're bound to *(accidentally)* come across the solution. Who needs to hit the deadline! Timeliness be damned! Or, the second approach is to break it up.

Create the feature on a new, completely blank page. Get rid of any scaffolding CSS and base markup. Validate that you can do the complex placement without any other layout affecting the feature. Once that is complete, slowly add on pieces of the existing page. First, add on the scaffolding, but no content or major layout. Then, add in the page layout. Then, add in the content. Along this path, at some point, the properly formatted and positioned original feature in question will skew. This process allowed the designer to narrow down the factors that were affecting the feature. It is much easier to fix the problem now.

In programming communities, there is much discussion about proper design methodologies and patterns. Some argue about the conciseness of classes or methods. Others will point to the necessity to use procedural programming over object oriented programming based on the task to be completed. But, if you look at the core of all these discussions, it's a focus of making something simpler, cleaner, more efficient, and independent. Programmers are constantly looking for ways to reduce spaghetti code and make components instead of dependencies. You never hear someone argue for the fact of combining all the code into a completely cyclical mass of indistinguishable source. Everyone wants to make it simple and modular, based on their best guess of how to do this. Programmers are continually trying to break code up into manageable pieces.

Since we already focus on making individual components of the simplest pieces as a programming architecture methodology, why not use this same idea elsewhere? Whether you're creating new software or troubleshooting existing problems, break it up. Smaller pieces are always easier to solve.

#9. Just Write More Code

YOUR FINGERS WILL BULK UP, BUT IN SOME CULTURES, THAT'S SEXY

Really smart people should be successful. But, more often than we'd like to believe, they aren't. They suffer from a mechanism that is meant to ensure their success, but ends up hampering them. This is analysis paralysis. *(Say that three times fast! But not out loud… don't be weird.)*

I've met tons of people in my life that have just great amounts of potential. However, they talk themselves out of being the best person they can be. A smart friend of mine would have made a great coffee house owner, but he let his brains get the best of him. Instead of striking out and doing it, he decided not to. He became a programmer instead. That's fine, I myself am a programmer. However, that's not what he wanted. It's not what he needed to do to feel successful. Instead of going full force into his idea, he analyzed it. He thought about it. He found all the negatives and got paralyzed by the thought of all the work. He never pulled the trigger. If he does finally get over this analysis paralysis, I'll be the first one ordering his espresso! *(I am, after all, a programmer - gimme caffeine!!)*

This same scenario happens with really great programming engineers. They realize the value and need for planning, so they sit down and plan out the project. They incessantly document, analyze, plan, structure, architect, and compare various parts of the project. They took that planning and documenting tome from their programming classes and stretched it far beyond what is necessary.

If you look around at your colleagues, you can identify this programmer. He has a great idea, but never seems to get the task done. He will miss deadlines, never complete the project, and always seem rather upset about his work. Too much time was spent planning and not enough time coding. The solution to this type of problem for programmers is very simple.

Just write more code.

Now am I talking about writing poor code just to have something down in the editor? YES! Am I implying that you should write a solution even if it's not the best one? YES! Before you crucify me for being a horrible influence and despicable author, let me explain why.

By just writing more code, you will actually stimulate all parts of your brain to keep thinking. You engage both left and right brains because you're requiring both art and logic to be applied. And, don't underestimate the brainpower you really do have. Even while you're writing this sub-par code, you're thinking about the problem in a different way. While you're writing code, you're actually solving the problem in your head in possibly a completely different way.

Don't get paralyzed by thinking of the best answer. Just come up with an answer. So, what do you do with this code when you're done? Refactor or throw it away. Remember, it's just bits; 1's and 0's. You can simply delete the file and try again. But, the exercise kept your momentum going forward. Instead of analyzing and talking yourself out of things, you tried your best idea at the time. Maybe it was wrong, but now you can look at it and know that for sure. Very few people can think entirely in abstract *(if we could, we'd all probably have better, more reusable code bases)*. If you recognize this, then you'll have more success by thinking and acting in the linear. Pump out code, solve the problem the very first way you think of. Then, refine.

Write more code. It's going to be wrong, but that's ok.

Now, one caveat: don't write this potentially "bad" code and move on. A common argument I hear from programmers is, "If I write bad code, the project will look done, the project manager will accept the work, and I'll have to abandon it." Another argument is, "I know once it's working, I'll not want to go back and work through it again. I want to go on to the next innovation." While these are two seemingly different arguments, the root answer is the same. Just because code is written, doesn't mean the problem is solved and the project is done. Write more code, and then try again and write it well.

I'm not suggesting that you should do anything underhanded or dirty, but perhaps you'll have to modify your workflow. Work in a separate sandbox and only contribute the pieces that are up to the standards that you want. Have a little self control and pride in your work when thinking of moving on to innovation. Remember, someone else will see this. Just because you innovate, doesn't mean you're great. *(As for innovators, I think about all those blog writers who write "introduction to blahblah technology" articles. They're on the forefront finding out all the newest stuff and teaching people the very basics. However, they're rarely as respected as those who apply a significant amount of time and energy towards mastery of a technology.)*

Write more code, write early, write often, and refine.

This advice is especially useful for beginner and novice programmers. Face it, you just won't have enough experience to come up with the most elegant and accurate solutions. Time breeds design maturity. So, while you're less experienced, just write more code. Look over it and see if it's good enough. If not, refactor and refine. You may find yourself spending a little time planning, a medium amount of time writing, and a lot of time refactoring. This is fine. This is what you need to do at this point in your career.

Those very experienced programmers may follow this rule, to some extent. However, they have the skill and experience to spend most of their time planning. Then, they can execute and create the most elegant, accurate code and programming solutions. Very little time is spent refactoring. This level of mastery and the ability to do it right the first time is because they have done this before. They have written a lot of code.

The paths of writing more and planning more are even slopes. As you progress through your career and gain experience, you'll notice that these slopes intersect. Your path goes from one to the next. Finally, you’re starting to plan more, and write just the right amount of code.

I can’t think of a time that I’ve ever heard of anyone saying, “Wow, I wish I wouldn’t have written so much code in my life.” There are really no negatives about writing a lot of code. And, since we’re in an electronic media, its pretty easy to manage. So, write more code. Engage the brain in many ways of solving the problem. Refactor as needed. Please, just write more code.

#10. Don’t Underestimate Analogies

A GOOD ANALOGY IS LIKE A NICE, CHILLED BOTTLE OF BOURBON…

There is this weird, crippling fear that non-computer users have. I’m not sure how it started out or where it comes from. All I know is that it’s real. And, believe me, you’ve seen it. That frustrating feeling you have when explaining something technical to a person while they seem to slink away. That’s it.

It’s that deer in the headlights shut down that happens, that unattached gaze, or the immediate denial of being able to do what you ask. It’s quite an irrational response to the thought of having to use a computer or understand how it works. I hate it.

But, I learned how to get past it. The key is an analogy.

Analogies are used all the time when we’re trying to explain things to each other. If you think back to your classroom days, I’m sure the teachers and professors used these to help explain concepts to you. Analogies offer a way of relating something that you already know about to something that you don’t. And, as you have probably now guessed is a common theme in this book, I’m saying that programmers have forgotten something we once knew before we became programmers. A great way to learn is through analogies. It’s also a great way to reduce fear.

One of the best examples I can think of involves ditching support for Internet Explorer 6. This is a real email that I sent the project stakeholder.

If we take a look at the market share of IE6 on your website, it's small but still measurable. I can't say no one comes to your site with IE6, but I can say it's under 5%. Now, I recognize that 5% of visitors is still a large number. We don't want them to be unhappy - and so that's probably the reason you're asking for IE6 support. Unfortunately, I can't provide that compatibility without considerable additional cost. IE6 contains a flawed model of displaying pages, so it requires much more work than the standards based website I made for you. Let me explain it a different way.

I'm 99% certain that your car runs on unleaded fuel. Most cars do. However, there is a very small subset of older cars that may run on leaded fuel. Perhaps they are hobby cars, or they've been grandfathered in after our emissions and safety laws. Either way, a very small portion of drivers need leaded fuel.

If you look at gas stations, you'll find various versions of unleaded fuel as well as diesel. You won't find any leaded fuel, though. Why not? There is clearly is a market for it, right?

Unfortunately, the market for leaded fuel is very small. However, the amount of investment required to store and distribute leaded fuel is at least as much, if not more, than unleaded gasoline. This means that the gas station owner would have to sacrifice some of the space and inventory to satisfy a very small market segment. The gas station potentially could even lose money when customers find out that he doesn't have enough of the fast-moving, highly sought after unleaded fuel. So, he elects to not distribute leaded fuel. He may know a few people who are very passionate about their needs of leaded fuel, but he needs to make the fiscally sound decision.

In this instance, IE6 is like the leaded fuel I've mentioned. There is a very small subset of people who need IE6 support, but you will have to make a choice on how to provide this support. Either your costs increase dramatically (I don't believe

> the ROI on this investment is worth it - and yes I know I'm talking myself out of money as your programmer!) or you'll have to lower your features and requirements for the other 98% of your users (which may make it harder to create conversions into paying members).
>
> In the end, the choice is yours - I just wanted to make sure you were aware of the type of decision this was. I didn't want to get you bogged down in the technical details, that's our job, not yours!

As you can see, it may have gotten a bit wordy and long. However, the core message here is that I was able to explain the Internet Explorer 6 support requirements using an analogy that the stakeholder understood. In the end, we didn't have to supply Internet Explorer 6 "enhancements" - whew!

There's no way to shield "regular people" completely from the technical aspects of some of the work you may be doing for them. However, making sure you communicate in a way that is simple enough goes a long way. I like to also relate this idea to a doctors visit. My best friend is a new doctor. When she starts rambling about her most recent medical knowledge, I feel like I don't know anything at all in life. I know she's really smart, but I just kind of tune out.

When I actually need medical attention, my own doctor is experienced enough to give me the information in a way I can relate to. Only after I understand the basic information does he go into the details I want to know. This is like the doctor saying, "You have too high of blood pressure, you eat too much salt." That's pretty easy to understand. An alternative could be where the doctor described the condition as a dangerously high systolic pressure, usually from eating more than twice the recommended sodium content. While that still is a relatively simple explanation, wasn't the first one easier to understand?

Don't estimate the power of analogies when communicating technical information to non-technical people. After all, most of us were taught in part by other analogies during our youth. So why do we suddenly forget to use them when we are working within our own realm?

#11. Give Proper Visual Cues

<BLINK> IS MY FAVORITE HTML TAG!

All good applications validate and error-check user information. The time will come when you are required to alert the user that the input they gave does not pass the sanity check. Usually this done via some visual notice or cue.

It is important to understand that there is a difference between what is programmatically wrong with the data, that thing that has been flagged by the programming logic, and what the average user will understand as the flawed data entry. Humans interpret invalid data differently than a computer.

For example, when two pieces of information do not match equally, a logical program deems both pieces of data as inaccurate. Both pieces either need to be identical, making the relationship true, or the entire data and relationship is false.

That is not how humans interpret the information. Instead, humans logically approach the data individually in a linear fashion. If the data is marked incorrect, it's human nature to look at the first *(or last, depending on the personality… I'm talking to you glass-half-empty people!)* data piece and determine if there is an error. However, what's different than a logical computer program is the tendency to not consider the data wrong when in context of the whole, but rather individually wrong. The user generally tends not to look for the relationship error, but looks for the concrete individual data error.

A good example of this is when there are two corresponding input forms on an interface. Perhaps this could be a password and confirm password field. Or even better, I have an example that recently happened with one of the programmers on my team.

He was creating a calendar application that allowed the user to enter a new event. The event had a title, a description, a start date and an end date. One of the validation steps was to make sure that the end date was either equal to or after the start date.

When the programmer wrote the validation, he attached the logic to both form fields. The code said to take both form fields and compare the values. If the first was after the second, mark them as error. The submitted interface marked both field as an error and alerted the user that the end date must be after the start date.

Unfortunately for the programmer, I rejected this code. I said that the error for the user wasn't in both fields, it was only one error. Marking both fields technically means there are two errors. Instead, alert the user one time to say that the start date must be before the end date. The code came back with the start date being marked with an error saying the start date must be before the end date. I rejected this code again.

I understand why the programmer decided why that placement was where the error should be. He was almost right, too! However, it wasn't good enough yet. Why?

If you look at the way that people think about event dates, there is a greater chance that they'll know the start date rather than the end date. When you're entering in data, the farther down on the interface, the more you're likely to make a mistake. Generally, the end date is after the start date on the interface. So, combine the fact that most people remember when their event starts more accurately than when it ends and the location of the field on the interface, the field to be marked with an error should be the end date. That is where the error must be shown.

To drive it home, I asked the programmer "can you imagine a time when you're filling out this form and you forget to fill in the end date? How about a scenario when you filled in the end date properly, but you forgot to add a start date?" He said the first scenario was much more likely. He admitted it seemed much more clear now.

Visual cues should be given with a thought to their context. A good programmer may remember to haphazardly place the cues. A great programmer pays attention to context and logical user interface.

#12. Find Someone Smarter

STEPHEN HAWKING NEED NOT APPLY

For years, I struggled with the decision of whether to frame my certifications and hang them in my office. I didn't want to be "that guy" who always seemed to brag about himself. But, on the other hand, I was very proud of all the things I had learned. I had been tested, and I had passed. I finally decided to hang them up in my office. I had so many, I ended up having to display them on a shelf as well. If you enter my office now, it really looks like I know what I'm doing. That, or that I'm one of the most conceited managers in the world. Oh, I hope not!

When I work with my team, I do code reviews and architectural decisions. Each programmer is learning, growing, and becoming truly excellent at their job. However, I am still the leader of the team, helping make the tough decisions, dissecting solutions and making sure our level of quality code remains high. When a programmer gets stuck, they generally will come into my office and sit and talk over the problem with me. Most of the time, I can help lead them to the solution that I think will be best. It doesn't hurt that we program mainly in PHP and I'm the author of a PHP Design Pattern book.

In my local area, I also started a PHP Users Group. I see various levels of programmers come and go in our group. They take turns learning from each other and teaching each other. Generally, I don't pick up much from the presentations, but I enjoy meeting with the people. If I volunteer to give a talk, it tends to be pretty tough for most of the people to follow. Some get lost in the details. Only one or two other programmers seem to feel comfortable asking questions or challenging my solutions.

I pretty much know everything.

And, if I really thought that, I'd be the biggest jackass - and you should put down this book immediately. But, that's not the case.

I can honestly say that I've had those thoughts go through my head a few times, though. I know a lot, there isn't anyone who can challenge my knowledge, I don't need to try anymore.

But, don't worry, I've been brought back down to Earth - a few times!

While in my somewhat smug attitude years ago, I went to a PHP conference in California. There I met people who created extensions for the PHP language, PECL programmers, and others who specialized in tuning the hardware and software underneath the platform that was running the PHP applications. A lump flew up into my throat as I realized how misguided I had become.

Compared to these guys, I knew nothing.

I could have wallowed in my own pity, or at the very least, stuck my head in the sand (ostrich style)and went back home, secure in my bubble. But, I put this next practice into effect instead.

Always find someone smarter than you! That's what I've done many times, and I'm so happy I did.

There always is someone smarter than you. For example, if you do PHP like I do, there are people who program C modules for the core. If you are a jQuery rockstar, other programmers specialize in just plain Javascript. If you are a great Mono or Python programmer and can create amazing Linux desktop applications, there are others who work on and submit code for the Linux kernel. There always is someone smarter than you, and that's ok.

What’s not ok is ignoring that fact. It’s not ok to pretend like these people don’t exist. You should make contact with smarter people and learn from them.

This concept really leads into forms of mentorship. Many years ago, I read an article in a business magazine *(Remember magazines? Or should I say, have you been to your doctor’s office lately?)* that talked about successful entrepreneurs. They focused primarily on Tech companies, so I kept reading. One of the most important things that was mentioned as a game-changer for successful companies was the leadership team having high-quality mentors. The article ended by stating that of all things, the most important thing for an aspiring person in the technology field was to find a mentor. That’s what kicked off my search.

Throughout the years, I’ve found many smarter people. When I was beginning to learn PHP, I had a great mentor who helped me learn more and challenged me every step. Later, when I got into business, I had a financial mentor who taught me about balance sheets and their importance to a strong organization. I have had mentors who helped me understand how to communicate technology to non-tech stakeholders. I’ve even had mentors that helped me understand the importance of various managerial decisions I’ve made. I’m truly indebted to these mentors as they’ve done two things for me: they’ve pointed me towards a path of success and they helped remind me that there are lots of people smarter than me.

Do you have a mentor? If so, when’s the last time you thanked them? Remember, there is no rule, requirement, or law that people have to help each other. Mentors happen to see value and potential in those who they build relationships with, and generally give guidance and direction out of the kindness of their own heart *(there’s actually a tiny bit more to this, but more on that later)*. It’s surprising how much a quick thank you can be appreciated by your mentor. I strongly encourage you to thank them the next time you can. Nothing too spectacular - a quick note, a nice card, or even a Tweet publicly appreciating them is great. In fact, if you have a mentor and you haven’t thanked them, please go do that now! *(I’ll wait. Nope. Don’t just put it on your todo list, get up, and do it now.)*

If you don't have a mentor, it's time to start thinking about one. Who are the people that you know that are smarter than you? And when I say 'smarter,' I don't use just one definition of the word. This person may be more intelligent, or they may have more experience, or just a different insight on the world. Perhaps they've been there before. Perhaps they've even failed where you might in the future. These are the "smarter than you" people that you should be looking for as your mentor.

If you don't know of anyone in your immediate group of friends or colleagues, you should look elsewhere. There are a number of online organizations that support mentor relationships. Take the time to email your heroes in your field and ask for their guidance in finding a mentor. *(Fun tip: if you ask someone for help finding a mentor, they may suggest some good resources for you. They might even volunteer themselves. For some reason, my experience is if you directly ask them, they may not be as open. Not sure why this happens, but it has to me. When I asked for help finding mentors, that's when I opened the floodgates.)* Start looking. I'm sure you can find at least one.

Don't get discouraged if it takes a while. All good things require effort. Also, don't get sad if your first mentorship relationship doesn't work out. I've went through a number of mentors in my career so far. Some work out for just a few instances or a short amount of time. Others are longer relationships. *(As cheesy as it sounds, I think of mentorships just like the quote on friendship: friends come into your life for a reason, a season, or a lifetime.)* Just because the relationship ends, doesn't mean everything was a failure. Any value you got out of the relationship is a measurement of success.

Don't forget that the mentor relationship isn't always defined by that name. Sometimes this person is just someone who you happen to work with, volunteer with, or know just in passing. This doesn't have to be a formal situation. Sometimes mentors are found in your current relationships, you just didn't frame it like that. The important point is that you really need to develop a relationship with "someone smarter than you" to continue to be successful.

On the flip side of finding a mentor is the act of being a mentor. As I mentioned above, mentors will invest time in the relationship mostly out of the kindness of their heart.

Perhaps they're excited to help develop a younger person's career, love encouraging potential, or just want to pay-it-forward for opportunities that they had. However, there are hidden benefits, too.

There are many opportunities to learn something from being a mentor. When you're a mentor, you get exposed to a number of other scenarios that you might not have run into otherwise. Someone is bringing their perspective and challenges to you and asking you to help.

The mentor gets to learn from this by analyzing different perspectives. Perhaps you've only thought of a problem a certain way, but you get to see it in a different light. Another value is the requirement to solidify and analyze the knowledge that you already "know." This means sometimes there is an underlying idea or concept that you make decisions based on, but when you need to explain it to someone else, you must think about it and package it in a new way. This extra thought forces you to analyze what you know to be true, evaluate it, and possibly expand on it. Then, those ideas get to be challenged by the person you've explained it to. This really is a value to the mentor. By continuing to give, they actually get more clarity and understanding of the things that made them successful. I've also seen the act of mentorship really reignite passions in those mentors who may have let certain parts of themselves go dormant.

The best way to learn something really well is to teach it. That's why I view a mentor as a teacher. Mentors are teaching the things they know to someone else. That forces them to learn it better.

One of the challenges I put myself through is finding a speaking opportunity with a group of people who may have a need to know about some new technology that I barely know about myself. I volunteer to go speak there and introduce the topic. This forces me to learn about the particular topic or technology. Not only do I need to know the very basics of it, I need to concentrate on those points to clearly communicate them to others who have not learned it, and be ready to answer questions about it. It really forces me to learn the material well.

To me, this concept is shared with those who are mentors. When you need to demonstrate and teach values to someone else, you really need to know and understand them yourself very thoroughly. You must take time to explain both the what and the why.

So, no matter what, please go find someone who is smarter than you. If you have that person already, thank them. They're helping you more than you realize. If you don't have one yet, go find one. Finally, don't forget that you can also be that mentor. No matter where you are in your career, there is probably someone who is taking those same steps you did before, and can benefit from your experience and knowledge.

#13. Sometimes, Just Be Great

OR, SUCCUMB TO THE MANAGER MOTTO: “MO’ MONEY, MO’ PROBLEMS.”

A few days ago, I was interviewing web designers for a new position I need to fill. I try not to ask the cheesy, “Where do you see yourself in 5 years” type question. I do want to know the answer to this, though. So I’ve come up with a different way to ask it.

The particular candidate we were interviewing was a very junior designer. He had many years to go before he would be someone I would consider to be in the position of Senior Web Designer. And many, many more before I thought he was experienced enough to run his own team. So, I asked him the newly phrased question: “What is it that you really want to be the path of your career?”

He answered in what might have been a textbook answer. “I want to put my time in to become a great designer. In a few years, I’d love to manage a team of designers.” This, however, is not the answer I was looking for. It wasn’t because I thought he would be some competition for my job, but because I didn’t believe him. I wasn’t sure if he wanted to be a manager at some time. Perhaps he just thought that was the best answer or the right answer for an interview. I asked him that point blank: “Do you really want to be a manager in a few years, or are you saying you’d like to contribute to the project in a meaningful way?” He went on to explain what he thought he would like to accomplish in his career. Long story short, he just wanted to just be highly skilled, important and respected. He didn’t particularly have any desire to be a manager.

I’m glad I was able to determine what this candidate really wanted out of his career. I sometimes fear that the standard career advice forces people to think that they must be managers or business owners. Not everyone is cut out for that. Sometimes, it’s just ok to be great.

Let me explain further - if you’re a follower of my blog, you might recognize this.

What is the measurement of a good career? Promotion? Raises? Running a project / leading a team? Usually all of this. However, the most traditional measurement of success in a career is moving upward. Time to run a department, time to lead a team, etc. However, I submit a very radical idea: sometimes just be great, don’t lead.

What does this mean? Well so often, we see great engineers get promoted to be our ‘boss.’ However, when this happens, suddenly the department starts to slip, success is lost, and this engineer isn’t as happy as he used to be. Sometimes it’s ok to just be great at your job. Know that being the number one resource, the go-to for your skill, is good enough. If your goal is seniority, demand it for your position. If your desire is more money, demand it. Sometimes we think that getting to a leadership position is the only way to gain. I would suggest to really look at this before you try to achieve it.

When you become a leader or a ‘manager,’ your responsibilities will change. Remember that exciting feeling when you solved a problem? You programmed a solution and it worked the first time!? This is something you won’t experience as often as a leader. Instead, your goals will be bottom line, juggling numbers, and dealing with people. How do you keep your department inspired and on track?

Let me break it down very frankly. There are two things to consider. First, if you want more money, ask for it. Keep doing what you’re so good at, demonstrate and document your value, and ask for it. Don’t take a promotion if that’s truly not what you want. A promotion generally means not necessarily more responsibility but different responsibility. Do you want this? Be honest with yourself. If you really don’t *(maybe you don’t like being in front of people, you like taking responsibility for yourself only instead of a team you had little hand in with the end result, etc)*, don’t do it. There are other opportunities and different jobs. There is always a different company that has the same needs as you’re filling right now, but they’re bigger and more scaled. That will translate into more money.

Second, are you willing to give up the thrill of the chase of programming and trade that in for the successful balancing of numbers and managing of people? Be honest with yourself. When you enter a room, do people congregate to you – or do you wait to be introduced? A manager(leader) needs to have these people skills. In addition, you need to be able to take responsibility for all the negatives for your team, but disseminate all the accolades. Can you do this? If not, just continue to be great, don't lead.

So, again my idea: sometimes just be great – don't strive to lead. You may find that you cause more harm than good, and you become unhappy. Instead, take pride in being the best you can be at that skill you've already mastered. Just be great.

The point here is to remind great programmers that they can be just that: great. You're not less of a programmer, you're not less of a success, just because you don't want to be a manager. Those who run a truly successful company - the kind that you should be working at anyway - will value your honesty and skills. They'll love that "you just want to be great."

As a manager now, I can absolutely attest that this is the truth. I always encourage my team members to become better, more efficient and learn more skills. If they truly want to learn more about management or product development, I am the first to support them. But, I do not want to push them into leadership positions if they don't want it. A good manager shows great programmers that he appreciates the programmer just the way they are.

Oh, and here's a secret too: bad managers want to keep employees who have aspirations on their own team and to themselves for selfish reasons. Good managers want to keep star employees because they make the manager's job easier but they won't stop creating opportunities for them even if it means losing them in the long run. It's fine to be good. It's ok if you want to be a manager after you've reached your goals in programming. It's great just to be great.

#14. Catch Your Breath

RELAX. CHILL OUT. UNCLENCH THOSE FISTS. MAKE YOUR BOTTLE OF IBUPROFEN LAST LONGER!

Sometimes I'm afraid to start out with cliche names. "Catch Your Breath" is often used for so many things, you probably have no idea what this chapter will be about. Should I be taking the time to relax on the beach when I get home from work? Should I be planning time to play catch-up at intervals in my project? Maybe should I stop talking to the client so much and just listen to them?

All of these things. Catch your breath, in every situation.

Because software development and technologies change at a fever pitch, programmers are almost always rushed to follow. The newest technology comes out, and you must know that software, immediately! *(It's even worse if you have a boss or a client who has been 'sold' on this new technology by an outside consultant or 3rd party and now believes you must have this feature in your product.*) It can also be a challenge to take a break from the project. It has to be done 3 days ago, so we need to work 24/7 on it. And if you're like me, you've been in meetings with stakeholders that can't seem to articulate exactly what they want. It can be really easy to try to finish their thoughts and sentences for them so you can get out of the meeting and back to work!

How about you just take a breath?

Let me address the three scenarios now in a little bit more detail.

First of all, innovators need to innovate. That's how we get new software, products and techniques. This is necessary and awesome. However, that's not the path for most of us. Our innovation comes from business process support and growth, not just using and creating new technology for technology's sake.

Most of us are allowed to take a breath. It is just as important to have a good base with whatever technology you're using as well as keeping up with the newest technology.

I used to be at odds with college degrees. I didn't think they were really necessary in our field. This was further compounded by a colleague of mine coming to work with me on my team. He had a computer science degree, but he knew nothing and was nearly useless on our team when he started. I had to teach him even the basics of developing and deploying our software. Those 4 years of college, what worth were they? Not until maybe a year and a half later did he finally tell me he was starting to see the importance of his education related to what we were doing. He finally got to a point where he was doing a little bit more advanced programming using proper architecture and design patterns. He mentioned that some of these things he was doing now were what was taught to him theoretically in college. Only after all this time did he finally have some context and appreciation for the base that he was taught.

I think we both reached the same conclusion. There was indeed value in his degree in computer science. He had the base, he just didn't have any exposure to the practical technology side. I think we were able to finally discover this realization because we used the same technology for the past 1 and a half years. He was able to specialize more in the actual technology and programming and wasn't distracted by needing to learn the next greatest technology. He had the time to catch his breath and therefore relate his theoretical knowledge to his daily programming. He had time to reflect on his tasks instead of always pushing forward with something new.

Even though we focused on the same technology for years, our products didn't suffer. We were able to continually produce high quality, sustainable, fully featured products for the business without introducing new technology. There was plenty of room in our current toolset to exceed expectations.

Because I forced the team and my new programmer to take a breath, we drew a strong link to the value of the computer science degree. Now, this programmer learns new technology, but has a very large wealth of base theory to integrate into each new technology. He is on his way to becoming an expert in this field. Since we took that breath, and didn't run to the newest, next technologies *(spending time learning new things, reading tutorials, etc)*, we were able to produce better, higher quality products for the company. I am certain that we wouldn't have been able to reach the level of mastery in programming and stability in our programs without this breath we took.

The next concept related to catching your breath is the strategic break from work. I've touched on this before when referring to 'do something different.' However, this is slightly different. Instead of doing something different, just don't do anything. Breathe.

When we look at our ancestors, many people focus on the food and meals *(in dieting books)* or the community structures *(for psychology and sociology study)*. I would point to a third part: Look at how they spent their day. From what we can guess, our ancestors spent a lot of time hunting or gathering food. Then, they made their own clothes, tools, housing, etc. However, when they were finished with this, what did they do? Did they play XBox or continue hunting for fun? Naw. They made fires, shared food, told stories and just relaxed in their community.

That's the key. They relaxed. They had a hard life; they needed to really be on their game when the time was right. It's sad that we've lost touch with this. Now, instead of having that time to relax, rejuvenate, and catch your breath, we measure success by our ability to work more. As we get older both our bodies and our ability to do certain tasks deteriorate. Some blame this on our society values. Others blame it on our poor diet. I think both can be true. But, we're also forgetting to take a breath. Relax, take a breath. Work will be there. Taking the time to catch this breath will make sure that you're more efficient anyway. And who doesn't want to get the same done with less effort!? Don't forget the lessons of our ancestors.

Finally, when working with a client or stakeholder, take a breath. Listen to them explain what they want. Don't try to finish their descriptions and thoughts for them. This almost always works out poorly. Instead, relax, take a breath, and listen to them. Let me give you a great example from one of my colleagues.

This friend was doing a freelance website for a small business owner. The business owner had a smaller budget which would get them a quality, small, simple website. In the end, that's all they wanted. However, they took a while explaining what they wanted.

At some point, the client said a phrase that really derailed the requirements gathering meeting: "I want something unique on my website, that none of my competitors have…" At this point, my friend broke in talking about the most recent technology he had learned about. The business owner seemed confused. In the end, probably out of frustration and confusion, the client agreed that the new technology was a necessity in his website.

When all the planning and quoting was done, the freelancer presented a project that was roughly 1.5 times the entire budget the client had proposed. A majority of this cost was associated with the new technology that needed to be applied.

As you might guess, the freelancer didn't win the contract. A competitor did win the contract, though. I have a good feeling that they listened better to the client thoroughly. The end result was a website that my friend could have done definitely within the budget of the client. We both agreed that the suggested feature to make the website new and unique using new technology was the reason why he lost the contract. He should have just caught his breath and listened to the client.

Here's the real funny part. We discovered what the "something unique" was after a couple months. In the business owner's industry, the colors that most signify a successful vendor were green and white. The unique thing about the new website was the addition of purple as a featured color. Purple was the unique thing and the only thing the client wanted.

Seriously, take a breath.

Part of being a great programmer is understanding the importance of catching your breath. Whether it’s letting the client explain their needs, mastering technology instead of incessantly acquiring new technology, or relaxing - just a little bit - after work, these things will help good programmers be great.

#15. Test Everything

9 OUT OF 7 USERS AGREE, BUGS SUCK

One very cold, very icy night, I was driving a bit too fast. Ok, I was driving my VW Rabbit way too fast. Suddenly, a dog jumped out in front of me. Since I'm a dog lover, I hit my brakes as hard as possible. I slid out of control and hit two fences and bounced off of a F–250 truck. Side and front airbags deployed and I survived with just minor bumps and bruises. One of the reasons why I got a VW was because I trusted their safety record. Even in an accident, this car performed as expected. More on this later…

As a software developer, it is very important to test everything. Test your code, test all the scenarios that could happen, and test it in every environment that it can possibly be used. Just because something "should" be fine doesn't mean it will. Test it. Test it again. Finally, when it's done, have someone else test it.

I've never met a programmer who didn't have bugs in their software. I've also never met one that didn't acknowledge the fact that they could have tested better to have less bugs. It's common sense, yet we seem to forget about it. Test, test, test! *(Or, my favorite excuse that I hear now that I'm a manager: "I didn't have enough time." Um, a buggy finished program is the same as a non-finished program to me. Sorry buddy!)*

Testing is incredibly important. *(Don't believe me yet? I suggest looking into Test Driven Development. This whole discipline and methodology is built around the importance of testing first.)* It's horrible that we forget it or put it on the back burner because the standard non-programming person doesn't understand bugs. Why in the world would the piece of software not work as defined? They're not programmers, so they don't understand the complexities required of programmers and creating something from nothing. They just don't understand that bugs are bound to happen. So, the less bugs, the better. Even one bug is bad when it comes to the users.

Remember my crashed VW? Can you imagine if there was a bug and the airbags didn't deploy? Well, I mean, the car always stayed on the road and it never randomly accelerated *(here's looking at you, Toyota)*. But when I had an accident and it crashed, a non-standard event, the airbags didn't deploy because of a programming bug. The programmers could defend that I shouldn't crash my car, so the bug is moot. However, luckily, VW tests their cars. They test, test, test. I trust their car and their process.

In software, trust is important. Bugs erode trust. This trust is broken easily when something doesn't go as planned. And, remember, because normal users don't understand programming, they can't comprehend the difference between different priorities of bugs. That's why its important to not have any *(or more realistically, very, very few)*. That only comes through testing.

Buggy products destroy perception even in the non-technology world. "Bug" perception is not restricted to software, only. Imagine the first time you visited a restaurant what you would feel if you were served cold potatoes and a burnt steak. *(That is, unless you asked for a well-done steak. And then, friend, you don't understand meat. Matter of fact, give me your steak, I've got a nice piece of beef jerky I can trade you.)* You'd not only be very unhappy with your current experience, but you'd probably not trust the restaurant to have a replacement meal ready for you. Even if you got a better value and a higher quality replacement meal, it just wouldn't taste the same. You've lost trust in that restaurant to meet your needs.

If you asked the kitchen staff, they might tell you what happened. They hired a new cook and he wasn't familiar with the stove and warming equipment. Because no one tested his prowess combined with the new environment, he made a 'buggy' dinner. Now, they might have lost you as a customer permanently. As a programmer, you need to test to make sure there are no bugs in your product as not to lose your customers.

After a particularly long project, one developer I was working with seemed a little stressed. He was working on the last feature for this release and was ready to push it out to production the next day. I had to sign off on it before we celebrated the finish of this product and release cycle. When I was running through the feature, I noticed that he had inadvertently messed up another feature. I told him about it and he said he just found that too. He said it’s not that big of a deal, hardly anyone uses that feature, and he’d fix it on the next 2 week cycle release. I said that wasn’t good enough, we couldn’t do that. We couldn’t knowingly introduce a new bug in an existing feature just to get our new software out the door. He didn’t seem to understand why that was a problem.

I reframed the situation by using the example of online banking. Imagine that you were transferring funds using an online banking system provided by your bank. You meant to transfer $400 from checking to savings. Instead, there was a bug in the software and you transferred $4000. This created an overdraft. When you notified the bank, they admitted there was a bug, they fixed it, and refunded the overdraft fee. Now, I asked the programmer, how likely would you be to use the software again? He said probably not that likely. I asked why not? They admitted there was a bug, they fixed it, and they even took care of the fee that they had imposed by accident. He still wasn’t convinced: “I just wouldn’t trust it - I mean it’s an online banking website, they shouldn’t have bugs.”

As he said that, I saw his face change. He suddenly realized what I had been saying. Any bug erodes trust. Even if you take care of it, you still cause distrust in your customers. The last sentence I said in our conversation before he went back to work to fix the issue was a mantra that I repeat a lot in the teams I work in. “Release the quality you’d expect.”

It’s important to test everything. Remember, any bug affects the customers in a very negative way. We need the trust in our software to continue to serve our customers properly.

#16. Seek Out Feedback From Peers

IT'S THE MOST VALUABLE FREE THING OUT THERE

There are two types of programmers: great programmers who are willing to learn, and jerks. And, I'm guessing you know both kinds.

The biggest difference between these two is ego and willingness to submit to peer review. Think about it again: that jerk programmer you know, how many times has he asked a fellow programmer for help? As you realize you can count the number of times on no fingers, you may start to really grasp the difference. The non-jerk great programmer? He solicits feedback from his peers.

Not all of us started out as talented, knowledgeable, and experienced programmers. *(I mean, I did. But your mileage may vary.)* Perhaps you're still in the beginner phase, learning new things every day. Perhaps you've accomplished a lot and feel rather comfortable and confident in your skillset. Either way, though, you can still benefit from asking for feedback from your peers.

Asking for feedback and submitting yourself for constructive criticism allows you to grow in a number of ways. The first and potentially most important is that whole "second set of eyes" paradigm. Whenever you get a team of individuals, or at least a partner, on a task, there are bound to be different viewpoints and unique perspectives. The colleagues you've asked for feedback will notice bugs that you haven't caught. They might also help by giving you suggestions based on their own mistakes. In fact, when you humble yourself and ask for advice and feedback, others are more than willing to reveal their own mistakes for the sake of coaching.

Don't believe me? Imagine someone asking you to teach them to swing a baseball bat. If you learned this skill by yourself, you probably missed the ball many times. But, finally, you learned how to knock that ball out of the park. Now, imagine a child asking you for help when they begin to learn to play baseball. You can pretty much guarantee you'd be overwhelmed with a desire to teach them all the things you didn't know. "This is how you swing the bat. Keep your eye on the ball." All this knowledge is based on the mistakes you made. This is the same in any industry, especially programming. If you ask your peers for feedback, they can not help but give you feedback that will save you from further mistakes.

When you ask peers for feedback, you're saying something to their subconscious as well. You're elevating them to a temporary authority. You're saying "during this task, I imagine that you know things that I don't know." Whether or not you really believe that about the programmer, this is the underlying idea and mentality. When asking for help, you're looking to someone who has greater resources than you. In this case, you're subconsciously telling someone they have greater programming prowess than you do. This isn't a bad thing for you. And there's even more benefit. It not only boosts the self esteem of your colleague, but it puts them in a position where they actually serve your project by identifying issues and depositing their knowledge, for free!

The final way that asking for peer review creates impact is quite minimal. But, it's something you should realize. For some reason, computers have created a number of interesting neurosis in our population *(remember "computer panic?")*. There is a concept called "programmer envy." The act of being a programmer actually requires a lot of underlying competition and desire for success. If you don't want to be a great programmer, you may not have these goals. But if you do, you want to be the best, and you want to succeed, you have this desire to become better than your peers.

Because of this competition, programmers develop a subconscious understanding of different levels and roles based on your experience. If you don't know what you're doing, you're on the bottom rung. You don't know anything. In a sense, it's almost like the military: "come on maggot, you don't know anything. I've put in the time, you haven't. Step up and learn!" It's incredibly easy for the more senior programmers to imagine the less experienced as less worthwhile humans. Newbie, you don't know what you're doing!

Want to flip things around? Ask for feedback. Right away. Humbly acknowledge that you're on your way, learning, and you need their feedback. Not only have you fed the ego of the programmer by putting them on an authority pedestal, you've demonstrated that you don't know everything. You're more than willing to learn. As a peer, but with different experience, this is how you demonstrate your worth, your ability to be a team player, and your willingness to learn. This whole idea helps change the thought of "this guy sucks" to "this new programmer is willing to learn."

Every once in a while, I'll issue a challenge to some of the web developers on my team. It comes in the package of "solve this simple task." We all put our code on github gists or pastebin and share it with each other. In this way, we're asking each other for feedback, all at an equal level. The first time I did it, there was some rumblings of "that way sucks" when reviewing colleagues work. But as time has went on, I've actually noticed that each programmer will give positive, constructive feedback. If they think a piece of code can be better, they'll suggest updates. Others will say things like, "you know, I've never thought of doing it that way, that's cool." The end result is that I have a group of programmers that are actively asking for feedback, leveling out the playing field, getting rid of extraneous ego, becoming more productive and creating better quality code.

This task can be done by any programmer in many ways. You can either ask your team to facilitate peer review, ask your senior programmer or boss for custom peer review of code or post your public code on open source websites and ask for feedback. Trust me, this makes you seem like an honest, worthwhile person while enhancing the self esteem of others that may help you - and ending up with a better product from a great programmer.

#17. Disagree In the Form of a Question

I'LL TAKE USELESS SUBROUTINES FOR 400, ALEX

Earlier in my career, I decided to study body language. I learned about some of the cultures of various states, countries, nationalities, how to detect someone who may be weaving a tale of deceit, and how your body gives away the fact that you're attracted to the opposite sex. *(It's interesting, certain people are known to perspire when they have a little bit of sexual attraction. In the United States, we are taught to be embarrassed of this. Please wear tons of antiperspirant! No one wants a sweaty dude or chick. However, studies have shown that one of the main ways to release attraction hormones is through the sweat. Believe it or not, you might actually be attracted to a person more if they have a certain amount of sweat.)*

During my study, I saw one particular demonstration that really drove home how important body language is. *(I still somehow find that crossing my arms is incredibly comfortable even though it looks like I'm annoyed during meetings. Perhaps this sign of "I don't care" would be less comfortable if I lost the shelf of a belly towards my front? Anyone want to weigh in? OOH Nice pun!)* This demonstration was based around personal space.

The first thing that was done was a test of face-to-face personal space during a conversation. Two subjects face each other and have a conversation. While the conversation progressed, the antagonist moved forward inch by inch. When he was finally too close for comfort, the other person would say "STOP" and the distance was measured. In the example I saw, about 16 inches was as close as the one person wanted to be to the other.

The next demonstration was the same basic conversation with the same actors. Instead, they both leaned against a wall instead of facing each other. As the conversation went on, again the first person inched closer to the other. The final "STOP" was issued at a distance of less than 3 inches from shoulder to shoulder.

That amazing demonstration was meant to show how body positioning and posture have a great effect on the way someone is open for conversation. Those same lessons can be translated into the world of communication in software programming.

If you're like me, you've had at least one *(just one??)* example where you're being told to do a task or program something in a manner that seems insanely ridiculous. However, it may be the stakeholder, boss, or client's request. They're proud of the solution they came up with and you must do it exactly to their specifications.

And you know what this pride also brings? Defensiveness. Yup, tell them you disagree with the solution they've proposed, and you're dead in the water. No matter if you were trying your best to get the greatest return for the investment, you are now the enemy. Why won't the damn programmer just do what I tell him to do?

You may want to tell them they're wrong. It's easy to insist that you're right. After all, you know best. This is what you do. All they do is sell coffee tables, but you've created over 300 inventory tracking systems. You are absolutely right, they are wrong. You're going to continue telling them that until they believe it!

Don't you think that might be a bit confrontational? Like having a conversation face to face? Try aligning with them instead. Could you try forming your critiques in the form of a question? Do you think it would hurt? I think all you want is for them to be successful, right?

Two paragraphs ago, I meant to ignite your passion. If you've been programming for even a little while, your blood boiled when I painted the picture of the demanding, hapless client. After reading the last paragraph, however, doesn't it all make sense?

Didn’t the idea I proposed just feel right? Could you disagree with any question I posed?

Yes, you already trust me - you’re reading this book, so you’re going to try to believe what I tell you. But, even if you weren’t already sold, my goal was to sell you on my point of asking questions. Did it work? If you disagreed with anything I said, didn’t you feel like that would be so crazy and wrong?

When you’re faced with a particularly tough truth or fact, try phrasing it in the form of a question. Instead of, “I don’t think the next button should be at the top right,” try, “do you think there would be any harm in positioning the button at the bottom right?” For the mastery students, “do you think that customers might misinterpret that button at the top right as the cancel or close button? What if we tried putting it at the bottom right? That would still keep the flow consistent, right?”

This type of communication is also tested in the world of marketing and sales. When you get a potential client in the form of responding to YES-based questions, the last sales based question seems like a no-brainer. Do you lose productivity during the day? Do you value productivity software? Do you like saving both time and money? Would you like to purchase this software today - at a 20% discount? Yes and YES! Give it to me!

A corollary to this approach is the multiple choice question. If you find that the decision maker has… well… made decisions poorly, try something new. Instead of giving your own answer in the form of a question, just blow them away with a thoroughly researched response containing multiple choices. This is a tactic that should be used sparingly, though. You don’t want to get your project too off the path. Just throw it in here and there, especially when the real right answer might be too extreme for them to buy into yet. Give them incremental choices in the form of a question with multiple choices for answers.

When you know the right solution is not the one you’re being asked to program, try the dispute in the form of a question. Note how positioning yourself as the choice-giver instead of the negative denier can help you achieve what you need while adding even more value to the project.

#18. Guarantee Long-Term Quality Using Two Development Paths

OR FALL OVER AND BREAK YOUR NECK, WHICHEVER IS BETTER

Having lived in Wisconsin my entire life, I have a great appreciation for our natural resources and attractions. We have a great lake ecosystem, lovely rivers, and distinct seasons *(if you've lived here or been here for a little while, you understand the sarcasm of my appreciation of distinct seasons I bet)*. One thing that we don't have, however, are the serene and lovely beaches of the coastal states. *(We have some beaches, but come on. They're not the same. Plus, no starfish.)* I've heard great stories about going to the beaches, enjoying the cabanas, and swimming out into the ocean to cool off. Those things are just not the standard fare in Wisconsin.

I think one of the most interesting things about the beaches are the sand castles. I have always been interested in how they were made and how builders became inspired. I used to play in the gravel pits by my house and pretend I was building nice sandy roads. *(Ok, so I'm not doing a great job of selling Wisconsin here, am I?!)* I attempted to build pea gravel castles, but somehow it just never worked. Then, on TV, you'd see these amazing intricate sand castles. I really wanted to be involved with that. I didn't want to move, though.

I did the next best thing: I researched sand castle building. I have seen pictures of some really detailed castles. One that particularly caught my eye was a three story sandcastle between some palm trees.

I wondered how anyone was able to build a sand structure so tall. I figured that maybe because it was next to a palm tree they could climb the tree and continue to build while hanging from wires and ropes. They would need their tools on very long poles to reach all sides of the castle, though. Another thought I had involved some really tall step ladders. Perhaps a 3 story a-frame ladder would be used to raise the sculptor to the proper height. This seemed dangerous, though. Also, wouldn't the ladder start to sink into the sand? By reading more, I finally found out how they built these great castles!

But first, let's talk about how programming projects can fail. In software creation, there are two different paths to take: innovation and stabilization. The requirements are gathered and the software is created. This is the innovation part. New features are created and the software is now the newest and innovative version of itself. This is where it gets difficult.

When there is momentum in a project, it's very easy to remain solely in the innovation path. That is to say, once new features are created, build on top of them to make even more features. Innovate, create, expand: that's the software battle cry! There is a second, equally as important path to take now, though: stabilization.

Stabilization comes in many forms. A recognizable example of this for programmers would be bug fixes. After software is released to the consumer, bugs invariably pop up. Part of the stabilization path is to fix these bugs. Other forms of stabilization are testing different markets and customer types, running A/B tests on interface designs, and evaluating the effectiveness of the newly created features.

But this stuff is boring! Let's move forward with new features! This is where the failure begins. You're adding more and more features to an unstable base where no bugs were fixed and no features were tested. Before you know it, this previously innovative software is now a slow, feature dense *(see how I didn't say feature-rich)* product that slowly creeps along. Now, trust-eroding bugs appear, the software stops performing, and the business either flops or the initial product is forced to be rewritten. *(Trust me, as much as you'd like to do it, almost always fully rewriting software is a bad idea.)*

This is where the competitive sandcastle builders kick our butt as programmers. They've implemented a perfect balance of the two paths and have both amazing and durable products in the end. How do they do it?

When a multi-level sandcastle is being built, the bottom level is created first. Basic layout, shapes, and architecture is created. It is far from a refined product. When the very basic is finished, sandcastle builders use wood to shore up all the sides of the current level, creating a stable platform for the next level. The process is repeated for the second level. Generally, the sculptor never has to use a ladder, long stick, or any sort of ropes or pulleys off of a nearby palm tree. Once at the top, more detail is applied. The artist stands on this nice stable platform built of sand and wood to create a very polished top level. Once the highest part is complete, each lower support is removed and the same amount of detail is applied to that level. Finally, when the builder is on the ground level, the details are finished and the sandcastle is now complete.

This same process should be implemented when we develop software. Create the initial features and stabilize them. Then, build on top of that and stabilize. When this iteration of the product has all of the features that are planned, continue back down the stable stack tweaking and refining the architecture and features. At the very bottom of the project, then, you'll have a fully featured, amazing, refined and stable product that had little risk of bugs the entire time. Or, you can tie yourself to a palm tree and see what happens.

It's your duty to make sure that both paths are followed during the product development cycle. This can be easier said than done, though. It can be difficult when you have a client or a boss that keeps pushing you to do a new innovation without investing in the stabilization portion. In the short term, sticking to your guns can be hard and may not make you a lot of friends. What makes you the better programmer is how you demonstrate your commitment to quality over the long term. By forcing a period of stabilization, you will be saving the product owner from headaches in the future. It is your responsibility as a professional to make sure you do what is best for your customer, even when they might not realize what that is. Remember, if they knew how to do this and what was best, they wouldn't be asking you to do it.

I need to temper this with a bit of realism and associated business impact, however. There are times when the customer or boss may know and fully understand the risks involved with not stabilizing the product. As the programmer, you need to investigate and dig in to find out if they really do understand the risks involved with pushing forward. Make sure that it is a deep understanding and not just a surface level comprehension of "something bad could happen." Work with them to define what "bad" means and what the cost and repercussions are.

In business, some success is had by taking risks. There may be times when the cost and the potential reward is greater to the software owner than the risk involved. This is something you must do your best to understand as well. Please make sure that your client or boss has all of the possibilities and relevant information in the case that they ignore your strong plea to continue on the stabilization path.

Telling programmers to ignore the requests of a boss or client can be a very dicey bit of advice to give. Let me just rephrase what I'm telling you to do. The successful project is one that is balanced with innovation and stabilization. You should do all that is in your power to make sure both paths are balanced.

Sometimes programmers can get carried away - and you just smiled because you've done this yourself. You may want to just keep innovating and applying the newest features. I'm cautioning you against this, do not be the driver of all innovation. Make sure you balance projects in which you have control between both paths. In projects where you are not the primary decision maker, you still want to make every effort to make sure that both paths are followed. In this case, your primary objective must be to provide all relevant information to the decision maker regarding the best practices to follow. Let them know in a non-emotional, very specific way what the quantifiable risks are for ignoring the stabilization path.

In the end, once you start applying both paths to your projects, you'll have a large track record of successes that simply can't be ignored.

#19. Social Capital Is Just as Important As Skill

DOES COMPTIA OFFER CERTIFICATIONS IN “PEOPLE LIKE ME?”

We all work on teams. This is an easy concept for those who are actually in a programming team at an organization or company. It’s a bit more difficult for the lone-wolf, the freelancer. But your team is just composed differently. Your team is made up of other freelancers who have similar skills to you. Sometimes these are your competitors; sometimes they can augment your projects with different skills. Point being, we all have team members.

Within those teams, leaders and followers emerge. Generous team members and leeches are discovered. Programmers who light up a room or make those in an email conversation smile and those that suck the life out of everyone. I think it’s pretty obvious which you should aim to be.

As you make inroads to become one of these positive members of the team, you develop something I call Social Capital. Similar to the value of money, it is not measured in dollars and cents. This represents the favors that you can call in. This is the ability to speak and have people listen to you. This capital is gained from your social circles, your network, and your team.

I’ve met some really good programmers. They are super talented at what they do. But, beyond that, they can’t save the world. No one I know is a specialist at everything. None of us have infinite time. These really good programmers are still struggling to become great and successful, missing deadlines and not getting the respect they deserve. Why?

Because while they have mastered their skill, they hadn't banked Social Capital. Or even worse, they don't see the value in this capital. That's what this chapter is about.

A significant amount of your time is spent learning the skills in your particular programming arena. What separates the good from the great programmers is the amount of time spent on developing their social capital. Great programmers invest in Social Capital. These are the people who contribute to open source software, that organize non-profit conferences, and are the first ones to answer questions on collective sites like Stackoverflow. They've mastered the concept of Social Capital.

On a cursory glance, however, it looks more like they're just generous or altruistic. But this isn't the case. They're banking up this capital because they've learned the lesson and see the value. And, you should be too.

What's the use of having such a bank? What do you spend it on? Do you really need favors?

I can think of two, somewhat related reasons why this capital matters: the too-big contract and the too-evil boss.

The too-big contract is that gravy ship that you can't turn away. It's just out of reach because you don't have enough time or enough skill and resources to meet all of the requirements. You know that if you can find a way to solve this one big problem, the rest is going to be exactly in your wheelhouse and you will have a great client for the foreseeable future.

Now, you have already probably figured out that you can outsource the tasks that you don't know. If you've done this before, you know that outsourcing can be a pain. How do you know the quality of the people you will work with? Are their rates going to be affordable? Do you want to base all of your success on a complete stranger?

This is where you cash in that Social Capital. Know another freelancer? Cash in that favor and have them join your team for just this task. Normally, they might be too busy or have higher rates than you can afford. However, since you've developed this relationship with them, they may be more willing to work for you at a rate that you can budget for this project. And, as an added bonus, because they owe you (in a sense), you know that they will want to do their best work for you. It's still business, they're still making good money, but all this extra work you put into developing your social capital with them has guaranteed a partner to get you through this one tough patch.

I know this works. I have one particular designer that I work with in this manner. He has a freelance business where he's more than busy. When he gets stuck or has a question about HTML5, Javascript, or setting up new servers, I'll help him. He always offers to pay me, but I decline. I'm happy to build this relationship with him, helping him, banking up my social capital. And, in turn, every once in awhile when I want to create a new startup venture which I'm bootstrapping *(see: I have no money)*, he's the first one I will call to design the product. Even more demonstrative of this relationship, once he hears about my projects, he's the first one to offer to step in even before I ask. I don't offer to pay him, he doesn't ask. But, if I were working for a paying client, I'd pay him a fair rate. *(He shouldn't have to do stuff for me for free - I don't expect work for free from my colleagues.)* Since I value his quality, his vision, his attention to detail so much, when my project has money I'll pay for his time without question. And I know that I'm guaranteed his best work because of all the social capital I've built up with him. *(If you're honest, when's the last time you did your best work for say… a family member? Yeah, probably not. You did good work, but not your best. Either that, or I'm just a horrible person and you are much better than I!)*

I try not to drop the "I'm a boss" bomb in this book, but I need to now. I manage a group of programmers and designers. This leads me into my second reason why Social Capital is important.

I'm going to share a secret. My team might hate me if they read this *(or when I force them to read this book and tell me how awesome I am.)* One thing some managers do, myself included, is to assign a task that we know is too complex, too long, or too technically advanced for a programmer to complete. I do it here and there. My expectation is that the programmer or designer will step up their game to solve this task. Then, when we reflect on this “impossible” task that they’ve created, they are now the proud architect. And, as a result, they’ve grown, learned, and become a better programmer.

There’s no feasible way for these programmers or designers to do the task that I assigned them on their own in the timeframe I’ve allowed. It’s nearly if not completely impossible. However, since they work on a team, I know that they can lean on each other to work out the tough tasks. Now, mind you, each one of the team members has their set of tasks with a deadline they should not miss. Because of this, if they help another team member, they put their own projects in danger of going late.

But, they always do help each other.

And for the most part, all projects get on time and on budget. You might say this is just an example of teamwork, and I would agree to some extent. But, it’s also a cyclical demonstration of the advantage of building the social capital among the team members. One programmer might help another three or four times over the next few projects, but knows that they can cash in that favor next time their own timeline is tight.

From a management point of view, I try to do the same thing. I have a full time job managing the team. However, from time to time, I’ll invite team members out for a meal or a drink on my own dime. Or, I’ll join them in programming some of the project. I hope to help them advance their timetable or catch them up if they’re behind. Because I do these things, I’ve develop a relationship with my team that is returned both in quality work and loyalty. I’ve invested in social capital and I see it returned in so many ways.

As a programmer, it can be easy to fall into the trap of just being the best at your skillset. Accolades and fame fall from the sky as you become a technology leader. However, the next best thing is always on the horizon, someone is always smarter.

Great programmers invest in their trade while simultaneously investing in social capital. Go the extra mile, give, lift up other programmers, and the results will be returned to you in multiples you may never expect.

#20. Step Up to Be A Senior Programmer

SHOW THESE YOUNGINS THAT PROGRAMMING ONLY WENT DOWNHILL AFTER PUNCH CARDS

Perhaps I’m the only person who pictures a balding, old man with a white beard when I hear Senior Programmer. I even picture him getting a discount at his local diner. Maybe I’m just weird. But, then, what is a senior programmer?

The senior programmer on a team is a very valuable position. This status could be reflected with a higher wage, a nicer office or better access to resources (and that’s great!) It might also be shown by the boss seeming to favor this position above all others. Who gets the nicest computer? Senior programmer. Who is the one who gets to meet the CEO? Senior programmer.

But, dear Senior Programmer, how do you follow up on your part of the deal? What are the requirements for this title, the things that you must do to cement this valuable position?

I recently promoted a programmer to the status of Senior Programmer. During our meeting, we were talking about his concerns about this new set of responsibilities. He gave the normal concerns but the one that really stuck out to me was his last statement.

“Well, this isn’t going to be easy.”

No kidding! No one promised it would be easy. The promotion wasn’t meant to be a gift. It wasn’t meant to make your job easier. It was meant to plug a hole in my team. I needed that Senior Programmer. So, I'll ask again, how does the Senior Programmer provide value? What hole is he there to plug?

The Senior Programmer is no longer a task-only programmer. A normal programmer is put on a project or a task, and that's what they complete. They will analyze all of the requirements for this task, provide the best estimate, and create the best solution possible. A senior programmer does this, and then some. He looks beyond the here and now. He thinks before this task, during the task, and after the task about all the ramifications of this new software or project. How will this task affect other features? What can we do to make sure this has the most impact and reward? What are the things we have to consider after this project is complete? The Senior Programmer is the master and keeper of these thoughts.

Since the Senior Programmer is no longer a task programmer, they must push themselves to serve as the valuable linchpin in the team. Some teams organize themselves in a way where a certain set of programmers are the authority when it comes to a certain project. Each team member has some code or application that they are the go-to resource for. Other teams are organized in a round-robin sort of project management. Each time a new feature comes up for a product, a different programmer is assigned. The rotation continues on each product, project and feature. However, if you pay close attention, you'll still notice that each project or feature tends to have a lead programmer or champion. Generally, this is the programmer who initially created the feature. This is the person that the other team members lean on when they can't immediately fix the problem.

Senior programmer, you are no longer allowed to do this. You must now know the intimate details of all of your projects. If you don't know something, take the time to learn it. This position may require a bit more work initially and some research on your point. Suck it up and do it. This is your task and your responsibility.

Consequently, you need to demonstrate this value by making sure that all on your team know that you're available to help them. Regular and junior programmers should be looking to you, no matter what project, to help them. If you don't know, you'll find out for them. This way, you'll learn it and be able to help them out too. Chances are, because of your already large knowledge base and ability to troubleshoot, you'll be able to ask other programmers who know the answer and get to the solution faster anyway! The senior programmer is this connection to the team.

The Senior Programmer position is actually a lot more about giving than you may realize. I've met programmers that imagine the senior position as something that allows them to get a private office, close the door, and just nerd out all day long on complex algorithms. Not true. The senior programmer is now the glue of the team. They not only code, they serve the other team members by collectively taking the lead in solving problems.

Another way the Senior Programmer serves is through the connection they have with the boss or manager. Managers, no matter how good-intentioned, normally don't have enough time to develop an intimate relationship with each programmer on the team. They should focus on their senior programmers, though. Your job as the senior programmer is to listen to your team, and then leverage your position with the boss. Learn to distill the team needs into something that the boss can easily hear and understand. Give of your time to go to bat for your team. The Senior Programmer is someone that can solve all of these problems for the team.

There is a difference between a manager and a leader. Great managers are also leaders. But, even if you have a great manager, leadership flows down. It manifests on many levels. Whereas all the programmers have a boss *(the manager)*, the Senior Programmer generally tends to act as the most connected, main "leader" on the ground and in the trenches with the team. All of this giving and service is a reflection of a great leader. The senior programmer becomes the glue that holds the whole team together while simultaneously leading them through the journey.

From the other side, the manager depends on the Senior Programmer for a lot of this connections. A manager might also lean on the Senior Programmer as the one that comes to meetings as an analyst. Instead of having to pick what programmer would be best for each meeting and project, the boss can just lean back and schedule the Senior Programmer.

As you see, there are different requirements and responsibilities with being a Senior Programmer. It's not all about just being recognized as a better coder. It's also about developing a team, leadership, and being a connection for all those around you. The Senior Programmer label is almost akin to a Team Leader label.

For those who are great programmers and want to continue to program, perhaps the Senior Programmer position isn't something that you should aim for. Just like when you're a programmer and a management position opens up, you might not always want to go for it. But, if you find value in programming and see how a greater connection and more leadership in your team could propel the team farther forward, then this position is for you.

#21. Write Out Your Goals

GOAL #1, FINISH THIS BOOK. GOAL #2, PROFIT!

In almost every leadership class I've ever taken and every book that I've scanned through *(because come-on, after a while they all seem to rehash the same things, right?)*, I've always seen one particular piece of advice that I've never really taken to heart.

Physically write down your goals.

Hogwash. Why do I need to write these down? Afterwards, the experts tell you to frequently go back and look at your goals. Some suggest putting your goals on a notepad that is always within your sight. *(How embarrassing - have a colleague in your office and they now see your life plan on your bulletin board.)* I thought this advice was probably one of the most stupid things I've ever heard. Plus, as a programmer, I don't write things anyway. I keep all the important stuff in my head - or I search Google.

One day I woke up with a great idea for a new project. I could see the screens in my head. I knew exactly what I wanted to accomplish. But wait… what did the screen look like again? I freaked out because everything was already fading from my mind. So, I grabbed my moleskin notebook and sat down and sketched out the images in my head. Then, I made a bullet point list of the project's goals. Finally, before I began the project that day, I put together another list of the tools and libraries I'd need to implement, what domain name to register, and what email service to get activated. *(I hate when stories just end like this with no follow up on the cool idea. I was going to do that to you, too, because the end isn't that exciting. But, here you go: I implemented the project. It wasn't as amazing as I thought. I ended up closing it down. But, I was able to sell the domain name afterward for $1,000, so I guess that's kind of a success.)*

This experience made me really think about the advice pounded into my head about physically writing down goals. Here I had this great idea when I woke up and then it started to fade. The idea was fresh and I wanted to implement it, but I was slowly losing my vision of it. So, I took pen to paper and sketched out my ideas. I created my bullet point list of tasks so I wouldn't miss any.

You wouldn't start out a programming project without at least making a list of your tasks and goals, right? If you're a newer programmer, you just guffawed at this silly statement. Just let me code! But those who have more experience absent-mindedly agreed to this statement in that 'duh' way. Of course I'd write down a few tasks; there's no other way to begin.

So, in our profession, we create lists. We plan things out because we know we may forget things. Even in the moment, when this project is the most important thing in the world, we forget things. *(That's one of the reasons why programmers create bugs! I bet there is not a single programmers out there thinking "yeah I'd like to have some bugs." Everyone hates bugs. And one of the reasons we have them is we forget things during the programming process.)* To combat this forgetfulness, we write down tasks. Or, if you're completely crazy like me, you not only write stuff down in a nice, orderly notebook, but you have a thousand little sticky notes on your desk too *(including amazing scribbles that made sense when you drew them)*. Oh Sticky notes! Because sometimes, paging through your bookmarked moleskin page takes just a tad too long compared to the immediately available yellow pad you can scribble on and then stick to your desk top. *(Note to self: need to clean off some of this 5 year old 3M glue residue.)*

If this is all true, why is it so weird that I'd write my life goals out in the same way?

I decided to do this. I broke the list out into a few topics: professional goals, family goals, retirement goals, health and spirituality goals, and finally charity goals.

I took some time one day to write out all my goals. I looked long term.

This was a tough day.

I had such a hard time writing down what I wanted. I really didn't have any idea what I wanted in my life. Somehow I had convinced myself I was on the right path, but what path was it?

It was a really hard day.

I cried about 5 times throughout the day. I realized how far away I was away from some goals. Other times, just coming face to face with the deepest needs in my life caused me to lose control.

It was one of the hardest days I can remember.

It was the most valuable day I've ever spent.

At the end of the day, I had my list of goals. I followed all of the rules from all the books when it came to writing them down and organizing my thoughts. I wrote them in first person as present conditions. I didn't say "I want," instead I said "I have." I put these all in a document in Google Docs.

Some of these goals are private. I will accomplish them, but I don't feel like telling the world about them. I don't mind sharing a few of my goals publicly, though.

Some of my publicly available goals:

> I have started, organized, and made self-sustaining 5 community or non-profit groups.
>
> I bike, kayak, or walk a few miles at least twice a week. I've not missed one week.

> I have found a faith-based community where I participate and actively learn and grow.
>
> I own a company that supports 50 families through employment opportunities.

As you can see, some of these goals are pretty simple. Others are lofty. Either way, they're my goals. I wrote them down.

The next step was setting a reminder to check them every month. At the beginning of each month, I make sure I read through each one of my goals.

But, for those who are afraid of being pigeonholed, here's one bit of good news. Every month, I review each goal for two things. First, how far along am I? *(I've created one community group that is self-sustaining. I'm working on my second one as I write this.)* And, second, do they still make sense?

It's not wrong to change your goals in your list. Remember, this is a guide to being successful, it's not the rules. So, if it's not reflecting where your life is going, don't give up and throw out all your goals. Just change them.

That's not a free pass to remove goals that are tough. But, be honest with yourself. If your goal was to live on the coast of California by the time you're 30, but you've married and you can't imagine a place better than being surrounded by your family and your spouse's family in northern Minnesota, change the goal. Find a way to get down to the core of your desire about moving to California was, and change the goal to reflect that. Perhaps you wanted to be involved with fresh food or technology that is rampant in California. Well, Minnesota has that too. So, change your goal from moving to California to becoming involved with your local technology and community sponsored agriculture movement.

Now that I have this list, I'm surprised how many things I've already completed. I've actually accomplished a lot! It's a great way to track your progress.

It also helps me make decisions. When faced with a tough decision, I look at my goals list again. I make the decision that takes me closest to one of my goals. Then, while I don't know if my short-term decision was right, I know I at least made the right decision for my longer term path.

Ok, this sounds all fine and dandy, but how does this relate to programming?

If you want to be a great programmer, understand that programming is your career. A good career is just like life. You can have a plan for it. You can have a goal sheet for it. You are the master of your life, why are you not the master of your career?

If you develop your goal sheet for your career, it will help you make better decisions at your job. Perhaps your goal is to be a development manager. Then, you know what types of positions you should apply for and what size companies to work for. Perhaps you have a goal of learning 5 programming languages in 5 years. You know you might have to do self-study to accomplish this. Or, if your company is working you too hard and will only stay with one technology, ever, perhaps its time to look for a new position.

Surfing technology blogs and getting certifications is a great thing to do. It sure uses up your time and makes you feel like you're getting somewhere. This is the same thing as building a widget by hand. But, those who are really successful develop an automated assembly line to develop that widget. Stop making widgets by aimlessly 'getting better' with technology, and build your career into an assembly line of efficiency by writing out your goals.

#22. What to Look For In Code Review

LOOK BEYOND THE TYPOS

Code review is a very important part of the programmer toolbox. When I say code review, I mean from both tech leads and peers. The process of code review can be different depending on your team and the technologies you employ. The general premise is, however, the same: you submit a section of code for another programmer to review. This process helps create better, higher quality code in the entire project. If you're familiar with my blog and follow me on Twitter, you might recognize the rest of this chapter.

A few days ago (ok, a bunch of days), someone asked me on Twitter what I look for when I do a code review. After thinking for a while, I've distilled the list of things I look for down into something I can describe. Now, mind you, I don't set out with my checkbox list or a manual, I just look at the code and "feel" it. Yes, that sounds crazy. I understand. But, subconsciously, I think I'm doing the following things when I do code reviews.

First of all, I'm looking for bugs. I'm examining the quality of the code. In my open source career, you'd be surprised by the number of submissions to revision control I've seen with PHP or Javascript errors included *(free - no charge!)*. I always suggest using a lint tool before committing your changes *(or uploading them via FTP, or whatever you happen to use to deploy code)*.

Quality also comes from sticking to a set of coding standards. In my projects, we have a coding standard that we use *(it's a combination of PEAR and Zend Framework – but either of those really works)*. If the code strays from this too much, I will bring it up during the review. Please don't confuse my suggestion of sticking to standards as architecture review. I'll cover that later.

Next, I focus on code architecture. Of course, it depends on the level of programmer who created the code you're looking at, but there should always be some semblance of an architectural pattern throughout the code in question. In cases where things look haphazard *(you know, like "and me!" or "I'm an add on!")*, I'm going to note that.

It's also important to pay attention to directions from a lead architect and proper design patterns. Sometimes programmers will come and chat with me about a solution. I try to guide them to the solution themselves. However, if they still need additional help, I might tell them the solution that I'd like to see implemented. When doing a code review, I'll follow up on this direction and compare it to what I mentioned. If it is different, part of my review is asking for the details on why the programmer did what he did. This open dialog will sometimes increase my understanding of where the idea came from. It may open up more learning opportunities for the programmer and myself. In rare cases, if the programmer did not follow my directions and has no legitimate reason for doing it, that would be another thing I'd bring up as part of my review.

I'll also make sure the code appears to be "how we do it here." That is to say, if the team has decided on a macro structure for features, a sort of company or team design, the code better be implemented with that in mind. That isn't to say that I suggest staying in a less-stable version of an architecture, but instead I strive to ensure consistency. To state it another way, if we're working at a mountain bike shop, we might bring in road bikes or hybrid bikes. But, we wouldn't bring in a car.

Don't forget to spend some time reviewing code legibility. We're typing, so why is this an issue? Code legibility actually has to do with the complexity or obscurity of the code. Sometimes, I'll notice new programmers will create too complex code because of a lack of familiarity with the language being used. For example, PHP has a huge list of array manipulation functions. In my code review, I'll see programmers unfamiliar with these functions recreating the wheel by building their own sorting algorithms. This is the perfect place to step in and suggest a better solution based on the language and technology in use.

I also look at the complexity of a code block and how it is organized. Often times programmers try to put way too much logic inside of a single method. Basically, this is unintended procedural encapsulation. I look to make sure that a method is doing one, or at most two, logical functions. Additional logical blocks should be refactored into their own methods for readability, clarity, and testing. *(I know I might have just sparked a heated debate about how much logic methods should include and how to properly build a code base, but that wasn't my intent. I have a set of my personal beliefs, and I aim to implement and measure them during code review.)* In cases where code just can't be broken out or the logic is quite complex, I'll look for comments. Most code, at a very basic level, can be understood. However, with a complex variable variable or polymorphic interface, this becomes more confusing. I suggest comments near code that is not clearly understood if you're unfamiliar with the project or codebase. I want to make sure a new programmer would be able to jump right in.

So, these are the things that I look for in a code-review. I have noticed that I may want to start integrating more positive feedback in my reviews as well. Basically, if you take a lot of the opposites of what I mentioned, they'd make great positive statements, too, right?

Also remember that the goal of code reviews is not to catch programmers making mistakes. Instead, we should all be creating teachable moments while keeping quality and consistent code. That's why I often say that "I'll bring it up" instead of "I'll yell at the programmer" when referring to infractions in my review process. I would encourage you to frame things nicely and positively as well. As a great programmer, when you do code review, you hope for the best code to make your job really easy. But in cases where it is not, positively enforce quality.

#23. Do Something Different

ARE YOU SURPRISED? YOU SAID YOU WERE UP FOR ANYTHING, RIGHT?

When I was very young, my best friend had a dad that was just larger than life to me. Well, first of all, he was over 6 foot and seemed to be a giant man. But, he also was into computers and played guitar. I looked up to him for all three of these reasons!

At one of my first “computer jobs” I remember talking to a coworker about his band. He wanted to release his own CD. I decided to create a website for his music.

When I do public speaking about the topics in 33 Things, I usually like to ask “who plays a musical instrument?” Without fail, more than two thirds of the audience raises their hands proudly. *(After the first three times, I remember thinking that I must have stumbled upon something here…)*

Looking back, I realized that every single technology job I worked at had a larger ratio of musicians than the rest of the population. I personally have been playing guitar since I was 14. I even created, wrote, and produced my own solo album.

Perhaps to be a great technical programmer, you have to play guitar!

Or saxophone?

Or piano? Some instrument?

No, that’s wrong. However, to be great, you need to remember to do something different. Do something besides program. I know this can be difficult, especially when you’re passionate about your project and your work. I remember many years working full time and then coming home and continuing on the project work until the wee hours of the morning. I seemed to have amazing output, but I was working sometimes up to 20 hours a day.

Then I hit a roadblock. When I compared my output to my peers, I was blowing them away. However, they seemed happier, and were involved in multiple activities, sports, and even family time. I was doing none of that. I then compared myself to some geniuses in my particular technology field and realized my output was nothing compared to them. Wow! How can this be? I only spent 4 to 5 hours a day non-programming!

That's when I realized that I could become a better and more efficient programmer by doing something different. That doesn't mean that I was quitting programming, no sir! It meant that I was going to spend my time programming, but also dedicate some time to other tasks. Maybe all these musicians knew something I didn't. *(Sudden confession time: though I had been playing guitar since I was 14, the more serious I took my programming career, the less I played. I found that I'd play less than once every few months before my realization.)*

I started practicing guitar every night after work. I found myself working about 12 hours on programming, 2 hours of playing music, and the rest relaxing and sleeping. My output was even slightly higher! I continued down this path until I came up with a mix that is perfect for me. Now, I spend more time with friends and family, create amazing programming projects, plunk away on my guitar here and there, and still educate myself in business and management.

Now, I'm not saying you can never program when you get home from your programming job. If you must for short periods to keep on a deadline, fine. But, evaluate your deadline afterward and make sure you don't get stuck doing that again. If you really are passionate about programming, make sure to do a completely different discipline after your main work. For example, when I was at a job a few years ago, I would architect very modular and enterprise level PHP and MySQL. I worked with message queues, object oriented domain designed classes and services, and tweaked as much performance out of my database as possible. Sometimes, when I really wanted to do a little nerdery when I got home, I would experiment with the development releases of Chrome and Firefox to test out their support for the new HTML5 spec. I technically was still programming, but I was doing a completely different discipline. I was engaging a different part of my brain.

The goal of doing something different is to engage all parts of your brain. The Right Brain is used for artistic things *(music, painting, designing)* whereas the Left Brain is used for more logical tasks *(programming, performance measurement, arranging a sound structure)*. As you do something different, you engage both parts of your brain. While I'm not a certified "brain-expert doctor guy," I can tell you for certain there is a benefit to doing this. At the very least you will have the same output for a shorter amount of time. Chances are, though, that you'll be more imaginative and creative. You may even start to see the art in your code!

Do something different. Pick up a musical instrument. Acoustic guitars can be purchased very inexpensively at rummage sales, craigslist or even your local music shop. *(My first one was a $5 P.O.S. brand that would untune itself after an hour.)* You might even mention your desire to learn to other programmers you know - I'm sure at least one of them has an extra guitar they could lend you to check it out. You can take lessons, try to figure it out yourself, or watch countless YouTube videos to learn your new instrument. This is just another challenge like a new programming language! Oh, and bonus! You already type, so you have significantly more control of your fingers than other beginners will have.

Do something different. Start creating some sketches or art. Almost all of us drew interesting little pictures when we were younger. *(You probably know when I say interesting I mean not museum-worthy, but who cares!?)* When did that desire go away? When was it suddenly not ok to create art for the sake of making art? Next time you want to mock up a new interface for a project, complete the task as you normally would. Later, take home this idea and try to make it even more detailed with pencil and paper. Or, you can explore other creativity by installing things like Photoshop and edit some stock photography. There are many Photoshop tutorials online that teach basic photo editing to advanced creation of art from scratch. Are you more extreme? Your local department store probably has some inexpensive canvas and artistry paints near the crafts or kids section. You don't have to be a kid to use these. Grab a new set and start painting. See what happens.

Do something different. Read a book. You remember those, right? I don't mean read some interesting blog entries about being frugal or traveling the country. I mean, grab a fiction book or a biography and commit to reading it. *(I found that while I like the idea of Kindle or iPad reading, I still get real books. Something about being purely disconnected while enjoying the book still speaks to me.)*

A mentor of mine once told me that I should start reading famous biographies. I protested saying that I really didn't like history. I remember him laughing and saying "You, and every other 20 year old." He went on to say that he figured out a long time ago why young people never seemed to like history. It wasn't history itself, it was their lack of understanding of the point of studying history. They just finished school, so they were purely trained and intent on knowing the facts. And, he said, history facts, dates, and names can be boring. He said the study of history was something different. "History isn't so much the what, it's the why." That has stuck with me for years! That last sentence made all the difference to me. I started reading historical works, not to know what year George Washington became General or when Susan B Anthony was drawn to the temperance movement, but to understand why they chose these paths for their lives.

There are three books that I've read in the last few years that have greatly shaped my personal and professional life. I made the commitment to do something different, and I've been rewarded.

"Losing My Virginity" by Richard Branson made a great impact on me. I'll try not to give away all of the secrets of this books or ruin his stories, but I can say a few things. Branson had both success and failure in his career. He tells the story of his various wins and losses, and the decisions he made. Now, I would consider him pretty successful. What I learned from this book: "Don't be afraid to take a gamble." I relate this to the programming world in a few ways. First, while staying in the bounds of what the client or stakeholder needs, you can still take some risks and gambles. Don’t be afraid to make small decisions on their behalf. Of course, don’t get crazy, but don’t be afraid to gamble either. The second way I relate it to programming has to do with new technology. Most of the people that are super successful in our industry took the gamble. They saw a new trend or technology and went at it full-force. These are the people we all look up to, they gambled and won. It may be easy to sit back and go where you’re needed, but if you want your own record label, mobile phone service, and airline, take a gamble!

The next book that I really enjoyed was “Decision Points” by George W. Bush. I don’t think many would argue with one simple fact: during his presidency, there was a lot going on. Depending on your political beliefs, he either became one of the most respected or most criticized presidents in recent U.S. history. I remember being very interested in following the news stories during his presidency. I was interested in seeing how the war was going, how the economy was doing, and his response to the United States’ largest terrorist attack. In his book, Bush described a little bit more about each of the main decisions in his presidency. He explained information that the general public might not have gathered from the media reports. He admitted to his mistakes, but also defined the reasons for his other, less popular decisions. Regardless of your opinion of his decisions or him personally, the book is a great read. The biggest thing I took away from this was some clarification in some of the decisions he made that I didn’t agree with. While I may or may not have changed my opinion of those decisions after reading the book, he did explain and point out a lot of other things that I didn’t know about. I learned that there is always more to a decision than meets the eye. I feel like this is also a very good point for programmers to grasp. I’ve heard the phrase so many times: “oh that’ll be easy” or “I can get that done really quick.” Inevitably, the project is late or that “real quick” turns into 5 to 10 times longer than expected. There is always more to every decision than meets the eye! Take the time to analyze all parts before making the decision.

The last book I’ll mention is “Confessions of a Wall Street Analyst” by Dan Reingold. This book gives an insider look at the analysis that was done for large companies like

WorldCom. Reingold explained some of the tricks that companies were using to inflate their value. He also pointed out places where analysts, including himself, missed the boat. The main thing I learned from this book was simple: You don't always know what you thought you did. Programmers will get in trouble if they assume too much. I can think of a few times where I gave quotes and commitments based on a very large assumption. I never disclosed that I was assuming so much and therefore I failed to meet expectations. Programmers should make every effort to make sure they are aware of all aspects of the task or project.

Are you looking to take your skills to the next level? Do you want to create even more output and possibly work less? Then, do something different!

Managers

There are tons of great books out there that are written by people with more training and more experience than I have regarding proper management. I've read some of them and they helped me formulate myself into the manager I am today. This section is not a replacement for them.

Instead, this section is aimed at managers with a technical background. I'm going to communicate some of the most important things I learned about managing a group of programmers and interweave that with my experiences as a programmer.

Just because you're a programmer now doesn't mean you should skip out on this section. I would encourage you to read this as well. You never know how these things will apply to your work now and in the future.

If you were promoted from programmer to manager, this is for you. If you're just new to managing a programming team, this is for you. If you've read every management book under the sun, this is still for you. You may see parallels in what you've read, but the added spin of being from a development background is the extra qualification that will really put you over the top as a great manager.

#24. Make Face to Face Work

THIS IS WHY YOUR PARENTS SPENT ALL THAT MONEY ON BRACES

In a world where instant message, text messages, and email rule, the written letter languishes as a decrepit communication tool. Who writes letters anyway? *(I've had the same 40-pack of forever stamps since… well, since forever stamps were first released. I have 3 left. I think that should last me at least two more years.)* But, you'd be surprised at the response you get when you send a written letter. I'm not talking about the letter that Grandma sends on your birthday with the world's crispest $10 bill. I'm talking about a real, live letter, from one colleague to another.

When you get this letter, you barely know what to do. Is this junk mail? It looks like it has come from someone you know, so probably not. You slowly open the envelope with… your fingers *(who needs envelope openers anymore?)* You get a papercut and proceed to bleed all over the enclosed letter *(well maybe I have worse luck opening letters than you do)*. You read it, hold it in your hand, and somehow seem to smile. Someone wrote you a letter.

The simple act of writing a letter made a difference. It was unique. And if the sender asked you to do something for them, it was effective. Somehow, you just did it. You bowed to their demands. They did, after all, send you a letter. Through the mail. It had a stamp. They paid to send this to you.

These feelings you get from a letter are studied in detail by direct marketing mailing response firms. They know everything there is about this tool and they use it effectively. These companies know there is a difference between sending an email to a mass list versus directly mailing you a letter. *(What's the difference? Why does this work? That's the topic for a whole other book.)*

But, there is something important to notice here. Even with all the other ways to communicate now, a letter triggers a response in a unique way. It has a certain effect. We've been using letters since we settled our country. *(Actually much, much longer.)* This was the only way to communicate. It kept relationships strong, is the stuff of love movies, and was the final note on business negotiations for decades. Letters reigned supreme for way longer than any of the readers of this book were alive.

Recently, we moved into the world of electronic communication. The wave of the future! Let's replace the tried and true letter with a low cost, super high available communication method. It works, and it works well. However, I would be one of those that would argue that just because something works better, you shouldn't completely abandon the older 'technology.' I mean, we have classic car shows and antique shops. The letter is an antique. And, as my story has demonstrated, even more valuable at times.

The letter draws parallels to other real time communication. In modern business, you now have so many choices for real time communication. You can text, you can instant message, you can call on the phone *(who does that anymore?*), or you can join a chat room. *(Want to know how to really prank me? Join my team as a programmer and master the intricacies of smoke signals for real time communication.)*

And then you have the last technique anyone seems to want to do these days: face to face. Face to face is almost today's postal letter it seems. Why would I get up and walk over to the office of another person if I could just send them an email?

Something special happens when we communicate face to face, though. We develop a relationship and communicate non-verbally. This actually helps us to be even more efficient.

How!? How can it be more efficient to walk over and talk to someone than to send them an email? Let's look at relationship and non-verbal communication.

When you build a solid relationship with someone, they are more intune to your communication style. Because of that, you have to explain less. They just ‘get you.’ Granted, you can develop relationships through text alone; many dating websites have proven this. But, we still meet people in person to take it to the next level. Face to face has something to offer: the efficiencies and rewards of a relationship.

The second is the non-verbal communication. True, if you’re programming and speaking simply in code, you might not fully see how the the statement that 94% of our communication is done non-verbally applies to you. I get it. But, as a manager, chances are you’re speaking less in code, so you don’t get to use this excuse. And, since you are a manager, you like numbers now. *(I know you have at least one spreadsheet in your inbox.)* Let’s break this down into efficiency numbers.

You have something you must communicate. It is an important thing that the team must do. You have to communicate the urgency and importance of this particular message. By the numbers, only 6% of the message is communicated verbally. That means to get 100% of the message across, without face to face interaction, you must invest nearly 17 times more into the conversation using text. If you could speak face to face, the numbers say that you could get this done 17 times faster, or more efficiently.

Of course, that was kind of a fun and silly exercise. But, let me put it into an example that might make it easier to understand. You have something important and urgent that you must communicate to your team. If you sent an electronic message, you could communicate this topic. However, it can be difficult to tell how urgent the need is and how totally passionate you are about this thing you’re communicating. This means that you’ll have to decorate the message with more description indicating the importance.

The message becomes longer. Those with short attention spans stop reading and just scan your message. All this effort you made to make the point stronger is actually lost and the message is diluted. Now, imagine doing this face to face. Normally you might walk into your team’s area and communicate information out loud and informally when it’s not too important. Today, you ask everyone to turn around and stop working. You take a seat in the middle of the group, sit up straight and slightly lean forward. You communicate with a calm yet stern tone what you need the team to know. There is no doubt that this message is different and the team should listen. You really care, it's important and it's urgent.

Face to face communication is the key. And as a manager, you need to understand this. Programmers might be more willing to IM, email or even text and shy away from face to face communication. You'll need to force them. But that's ok. In the end, it's worth it.

The way I've implemented this in my own team is by having a face to face meeting with each of my employees at the beginning of the week. This time face to face helps me build a relationship. Because I meet with each person individually, sometimes I have to repeat myself. But, I invest in this repetition because I know that my message has hit home more thoroughly through the relationship I've built. I've more efficiently said my piece face to face compared to generating some super long email talking about each project for this week that the programmers may likely ignore till lunch *(or later!)*.

Another thing I do goes hand in hand with the 'open door policy' you hear many managers boasting about. They may keep the door open, but when you walk in, you feel like you're interrupting them. Well, you are. But, that's their job. If you continue to compose an email, making the team member wait, you're being a jerk. I bet this person doesn't really come into your office that much anymore because of this. Remember, as a manager, your primary job is the management of your team. The rest is less important. Stop being a jerk with an open door.

When programmers walk into my office, I do two things. First, I immediately stop what I'm doing. I greet them and turn all of my attention to them. And second, I physically turn my body towards them. I want them to know that we're face to face. I want to show that I'm valuing what they have to say. This allows them to ask their question or communicate their concern in the most efficient manner. You know what it's like to go to your own boss with a concern? It can be hard. It can be intimidating. By removing barriers, stopping what you're doing, and giving them the 100% communication offered by a fully committed face-to-face conversation, you're reducing the amount of communication required to get their points across. They will appreciate this for both the reason of having to communicate less and the respect that you're showing them by making them the center of your world for this short time.

Face to face communication can be hard when you're working with people working remotely. I understand that, but in this age, there's really no excuse. Webcams are either built into laptops or for sale under $30. Free services like Google Hangout or Skype exist. Take advantage of these. While you're not physically face to face, you can see them and they can see you. *(Plus, if you're working with remote people, this might encourage them to put pants on just this one time during the week!)*

There's no excuse not to have meetings with remote workers 'face to face' over video conference. It might seem weird at first, but before you know it, it becomes second nature. One word of warning, though. If it is at all feasible or possible, do not rely on video conference if you can physically meet in person. Only use video when you absolutely need to.

I think face to face communication is one of the basic human needs. And dear manager, your programmers, deep down are human. Give them the gift of being able to speak to you face to face while you enjoy the efficiencies and benefits of it. To be the best manager you can possibly be, just show your face.

#25. Learn to Do What They Do

IF YOU WALK A MILE IN THEIR SHOES, YOU'RE MORE LIKELY TO REMOVE THE STONE

If I had a chance to take a poll of all the programmers in the software companies, I bet I'd find out that their number one gripe is that managers have no idea what they do. The runner up would probably be a substitution of "manager" with "Project Manager." After that, various other gripes about those involved in the project not understanding the complexities or technologies involved.

If you're a manager who has previously been a programmer, I've got a tip for you. Don't forget to do what you used to do. Take some time out of your day, week or month to make sure you keep the set of skills you have current. This will benefit you in two ways. If you ever get downsized, you can still jump back into the programmer role. *(Or become a freelancer.)* Either way, you'll be able to earn a wage. My experience has shown that a manager is a hard position to find and fill *(maybe because it seems that they are rarely let go?)* The second benefit of staying current with your skill is that your team will notice that you still care. You're a manager, sure, so your work responsibilities are different. But, you're not forgetting where you came from.

It's even more important to take extra steps if you've never been a programmer. You may never be able to program like those on your team, and that's perfectly fine. That's probably even a good thing. It helps you have a different perspective. But, in order to do your work properly, you need to have two things: an understanding of the assigned tasks and the respect of those who you lead.

Understanding new projects and the tasks required to complete them is easy. Really, it is. Just take some time and sit with your programmers and ask them what they're doing. Tell them you're not there to try to slow them down *(if you can work extra time* *into their schedule for the inevitable delay you cause, even better!)*. You're not there to judge them or watch over their shoulder in a negative way. You just want to get a deeper understanding of what their day requires.

I think about the show "Undercover Boss" and how valuable those days undercover can be for the leaders and CEOs. They get to learn firsthand what it's like to do the job. But, here's where it's different. The undercover boss *(or yourself in our example)* has the authority and ability to correct some of the wrongs that the employee may not be able to themselves. If you have to do an annoying task yourself, you're much more likely to want to fix that process for the staff. If you never had to do it yourself, it's pretty easy to just ignore the complaining or assume it isn't as bad as it's described.

Another way to stay in tune with your staff is to take a class. You can go to a free or cheap online class. You may even want to enroll in a technical school or online college for a short amount of time. Get educated about what your team is doing on a daily basis. This will not only help you understand the tasks better, but it will also garner respect from your team. They know you want to keep up with them, not just bellow at them to do it better, cheaper, and faster.

Speaking of respect, that's a bonus added to your toolbox if you learn to do what your employees do. They may want to mock you *(to your face or otherwise)* for being so poor at the skill, but deep down they will respect you for taking the time to learn about their tasks. That's the best way to be a leader is to show that you're willing to do whatever they do. Never assign someone a task that you're not willing to at least try to do. If you fail, that's fine. The goal isn't to be a great, well oiled successful programmer. That's why you have your team. Your job is different so this failure is ok. Remember, though, one key responsibility of your job is to understand your team's job.

Everything your team does is important. If you have designers, learn about the design process. Take some classes on design theory and color choices. Learn to do a basic "hello world" program in each of the languages your team uses. Learn to compile and deploy the software. Join your tech support and technical writers as they attempt to explain the features. Write a chapter of the manual yourself. Submit it to the team to “peer-review.” Learn what it’s like to do what “they” do and you will do your own job better. You will be on your road to becoming the best manager you can be.

#26. If You are Seducing a Developer, Follow-Through Is Key

SOMETIMES IT TAKES MORE THAN ONE DATE TO KNOW IF THIS IS “THE ONE”

I’ve had my share of struggles finding developers to fill positions on my team. But, every one I have is the exact one I needed. I took the time to put out a job opening, follow up, do research and seduce a great programmer. I don’t take the attitude that they should be lucky that they’re getting a job. I realize this is a two way street. I need something from them, so I seduce them. I rambled about this on my blog after I had a particularly silly conversation with a recruiter once.

I remember a scene from “A Night at the Roxbury” where the less-than-slick brothers take some girls back to their room and have a problem closing the deal. One of them keeps saying pickup lines while he has the girl sitting on his bed. She basically says ok, we’re past this. He just can’t seem to close the deal no matter how much she wants him to.

That reminds me of what happens sometimes when I watch recruiters and other business people when it comes to seducing developers to projects. They pitch a great idea, get a little bit of interest, and then stop there. They don’t close the deal. They offer things but never follow through.

I’ve been hearing for the longest time that it’s very hard to recruit for PHP developers in our area. I’ve seen offers from recruiters for these jobs to those who aren’t looking. Even if they pique a little interest, the recruiters never follow through. Let me give a fun example.

> Recruiter emails: here's a great job for you if you're looking.
>
> Developer: No Thanks.
>
> {{ crickets }}

Hey – that's not follow through! Why did the developer say no? Because he's not looking? Or is it because the job doesn't suit him? Recruiter, you lost out on gathering some more information and possibly getting this programmer as a hire. Here's another example.

> Business owner: I'd love to buy you lunch or a drink and just talk – see if you can give me some advice about some upcoming projects.
>
> Developer: Cool, let me know.
>
> {{ same damn crickets }}

In both scenarios, the initiator probably wants the developer for their team. But, there is no follow through. However, there is hope!

Recently I've been noticing the recruiters follow through a little with a response saying "Thanks." Not huge, but it's making progress. But, you need more follow up than this.

Dear business owner, manager or recruiter, let me tell you a truth that no programmer wants to admit. Computer "people" are not the same as you. Every group of people have their own traits and some generalities seem to hold true for them. In the programmer world, these can be at odds with bubbly owners and recruiters. *(Ok fine, some of us can't stop talking and want to be your best friend, like a little puppy. But, that's not the norm.)* Now, I don't want to offend with generalities, I'm only working from my own experience. But, I've noticed a higher percentage of nervousness, a greater fear of the unknown, and a high propensity to do anything to avoid situations with failure and social risk in programmers.

So here's my advice: Follow through. Let's do the scenarios again.

> Recruiter emails: here's a great job for you if you're looking.
>
> Developer: No Thanks.
>
> Recruiter: Alright! Thanks so much for taking the time to get back to me. Really appreciate it. I know not every opportunity is a great fit at this time, but do keep me in mind if you're looking for any new challenges. I've got a few positions that offer unique sets of rewards for the right individual.

Oh, and dear recruiter, if you're local, build some bonus points. Continue with this:

> Recruiter: Actually, if you're a coffee addict like me, could I buy you some coffee at Starbucks next week? I could use some insight from people like you to find out how I might find other qualified candidates.

Programmers are people too. And, people like feeling important and wanted. The fact that you're following through really displays this. Plus, through this time you invest, you get to know the person, and determine why they may have turned down your original offer. Maybe they really are looking, but the job wasn't a fit. Now you have a chance to develop a profile on this candidate to help them out in the future. Let's look at the other example.

> Business owner: I'd love to buy you lunch or a drink and just talk – see if you can give me some advice about some upcoming projects.
>
> Developer: Cool, let me know.

Let’s take a quick pause and try to predict what each is thinking. My guess is that the business owner is thinking, “well I’m pretty busy so when the developer has some time, he’ll email me.” The developer is thinking, “That seems cool, and I know business owners are busy, but why isn’t he getting back to me?”

Here’s how the follow up should go:

> Business owner: Excellent! Next Thursday looks pretty open for me. Would you like to go to the local Chinese restaurant at noon? If there is a better time, let me know!

These little bits make a difference. No matter if you’re a recruiter, owner, or a manager, you’ll need someone at some point. If you’re trying to seduce someone to a job, follow through. People sometimes need that extra little push.

#27. Motivation Isn't Always About Money

THAT DOESN'T MEAN SHINY THINGS AREN'T DISTRACTING, THOUGH

When talking with a member of my team the other day about a wage adjustment I gave him, what he said surprised me. "Yeah, the raise isn't really all that important to me." I was taken aback. I thought everyone was driven by money! *(At least, that's what you'd believe nowadays from all the special reports on television about Wall Street and corporate greed. Money, money, money!)* I asked more about what was important to my new favorite programmer. What motivated him? "I have things to learn and ways to grow. I get those chances here."

What a simple answer. The honesty of this answer amazed me. I know I'm doing something right if he's willing to tell me that money isn't important. But, I still didn't really believe it. Not fully.

Then, I began to think about others that are on this and previous teams of mine. There are some that are motivated by money. You can almost always pick these out of the crowd once you know what to look for. The money hungry are always trying to scheme a different way to get more money. They measure success by the size of the paycheck and by the quality of the car they drive to work. These are the people who ask about raises if the review has been a week late. These programmers ask about the wage during the interview. They clearly are driven and motivated by money. *(Somehow, I imagine them all to have really flashy shirts, jewelry and slicked back hair. Oh, and shiny shoes.)*

Others are motivated by convenience. That is, they've been doing the same job over and over for many months, years, or decades. They enjoy knowing what they're going to do when they go to work. In other words, they don't like surprises. It fulfills them to do the task put out in front of them. Want an extreme example? Check out the documentary Jiro Dreams of Sushi. This sushi master has been doing the same tasks for decades. He loves it and he wouldn't have it any other way. The simple task of doing his job, yet always improving, is what drives him. It motivates him. He is absurdly, insanely happy doing the same thing over and over.

In the same vein of convenience, perhaps the job is close by the programmer's house. Or it's on the way so he can drop of his significant other on the way to work. Or any number of other conveniences and comforts you may not know about.

Others enjoy the challenge. Yeah, they could make a lot of money sitting in a cubicle sporting the newest Hugo Boss suit, but they'd rather work here. The challenges presented are what drive them. They have managers and project leaders that can throw tasks at them that seem impossible. But, this programmer will find a way. They love these challenges. They love their drive home because it gives them even more time to come up with the best, most elegant solution to an otherwise unsolvable problem. *(Once the challenge goes away, no amount of money or promotion can keep this person. They're moving on to the next challenge at the next company.)*

Returning to my original example, others are on your team to learn. I see this definitely in some of the younger programmers on my team. They have learned enough now that they could leave my team and perhaps join a team in a company that could pay them more money. But, they have their eyes set on the horizon and can see even more learning opportunities possible. They know their job is secure. They can find value in learning from and working with those who have many more years of experience. The learning opportunity is what drives them. Here, they are getting paid to learn.

As a manager, it's your job to pay attention to each employee and determine what their motivation is. And, news flash, their motivation most likely will not be the same as yours. You need to cater to what each individual needs.

My current team of programmers is very highly motivated. For this book, I really tried to pinpoint what I do to help motivate them, and I couldn't really come up with an exact reason. Then, a trusted peer told me I said something once that really resonated with him. He said that if I say that same phrase to my own team, no wonder they're motivated to stay and work with me. This is what I said:

"When you work for me, I will make it my number one priority - I will go above and beyond - to make sure that I make opportunities for you. If you want to learn more, I will find the budget for it. If you need time off, we'll make it work. If you want to program a particularly tough task, I'll assign it to you. In return, I know you'll work hard, be honest and loyal. If there is ever a time when I can no longer make opportunities for you, I'll be the first to tell you. And, I'll look forward to working with you to find the next place that you can work and be successful beyond your wildest dreams."

I barely remembered saying that, but I did. And when my peer mentioned it to me as something that was particularly important to him *(something he implemented in his own team)*, I realized it was the key to my team's motivation. Some want money. So I go to bat for them. 3% raise this year? How can I get this programmer 5%? Some want to learn. How big can I make my training budget? Some want a challenge. How do I challenge this programmer yet keep us on task to solve the business problems?

I used to think that the easy answer was to throw money at the problem. You want the best programmers, throw money at them. This isn't the case. It's all about finding out what the true motivation of each programmer is and making sure you fit that need. *(As a side note, some of the best, most highest paid programmers I've met don't care about the money. They simply raise their rates because of the law of cost and supply/demand. They need to be careful with their time, and high rates guarantee that.)*

#28. Programmers Are Like Rockstars

LUCKILY, WITH LESS INSTANCES OF LEATHER PANTS, THOUGH

Creating product, building revenue and profit, and reducing waste are all things that are highly prized and studied in the manufacturing world. We've been producing things for such a long time that we now know how to get the best performance out of each one of our tasks. We know there are different types of assembly lines that operate, different performance metrics that can be used to measure output, and all kinds of ways to get more of the same product out in less time.

Manufacturing is not easy. But, in general, we know how to do it now. General managers of production plants can sit down and develop spreadsheets and scenarios to measure output and productivity. If your shop makes 100 widgets today, and 105 tomorrow, you've created a measurable gain. It's pretty much a linear style of creation which comes with a linear style of management.

Now, think about some famous artists. They create paintings or sculptures with amazing style and tremendous beauty. When they're done, they're done. No one really knows when the next piece of art will be available. That's fine. When the artist finally releases a piece, you can be sure that it captures the essence of their message and embodies their soul. This is now displayed proudly or perhaps put up for sale. It can be days, weeks, months, years - who knows - before we can find a way to measure the return on this piece of art.

And somehow, this is ok. In fact, this whole idea is welcome *(well, tell that to the starving artist)*. Artists can be eccentric, they don't need a schedule, and they don't need a predictable turn around time. There's no real management here. No one is measuring them.

Now, let's jump between the two extremes to talk about rockstars. A musician goes and creates a brilliant piece of art. They put everything they have into the words. They toil along until every note is perfect. Then, they release the art and it's done. It's available to be appreciated.

But what does a rockstar do next? They go on tour. They have to take this beautiful piece of art they created and duplicate it flawlessly many times. They're attached to it, they love it, so it's not that hard the first few times. As time goes on, though, to them it becomes drab and old. They may be so bored with playing it that they talk bad about the song even.

But don't you dare tell them it's bad. You can't agree with them. Want to see someone prickle up real fast? Agree with that musician that his song sucks. Who are you!? You can't write a song this good. This is a part of me. Don't say that!

But even though the song is old and boring, they get up again and play the song. You know when you go to the concert, they're going to play the song you know.
They'll (try) not forget any words and aim to play it perfectly. You can measure the success of this rockstar by how many seats they sell each night. If you raise the prices, and people still buy the tickets, and the rockstar is more successful. As the artist gets better and more famous, you definitely can raise the prices because fans will pay.

When it comes to programming, I've seen examples using both of the extremes. You have the outsource shop that puts a ton of programmers in a room. They measure the contracts they get, the lines of code they deploy, and the number of projects completed. It doesn't matter if the solution is good or elegant. It only matters if the solution solves the problem. And, if you can get more solutions finished faster, that's more measurable product and more success. The manufacturing business of programming is really easy to understand because we have a handle on the manufacturing business of goods.

Then, far too often I see the artist mentality perpetrated into programming. As almost a knee-jerk reaction from the manufacturing view of programming, managers jump to the extreme opposite side. Or, at least they're being convinced by the programmer. *(Drat! You've been caught!)* The programmer knows what he does is an art. It's done when it's done. I don't need to do any estimates. I can't tell you if it's going to work right. I need to refactor and make this better. The solution must be the most elegant and beautiful solution ever. The diva artist mentality of programming is live and well!

But a great programmer is somewhere in between these two extremes, like a rockstar. *(Oh no - that doesn't mean I like all the job ads that say "rockstar programmer needed" - if I never hear that term again, I would be one of the happiest people in the universe.)* There is an art that is programming. It's not just as simple as making a widget in the same way over and over. Some of the problems sound like the same thing, but the solution is always affected by many other dependencies. But, the point is, the solutions are art. This can't be measured as simply as one might expect.

But, in the same way, the programmer can get to a point where they can produce the same product *(or close to it)* in a consistent manner. It just takes work. It takes practice, but it can be done. It's a combination of artistry in a set of code or a library used repeatedly in new projects.

When managing programmers, it's important to think of the rockstar mentality. No, I don't mean give them extreme riders, but give them everything they ask for, within reason. Let the art flow. You will notice how protective and proud they get of the art they create. And, then, it shifts into the concert stage. Once that art is created, let them duplicate it over and over. Just don't forget, the next album is set to release next year. *(Or in programmer terms, after some time, you'll need to refactor.)* When you realize how to keep programmers out of the manufacturing management plan, but keep them scheduled on something that can be duplicated, then you'll really have those rock star programmers.

#29. Sometimes, Just Ask Why

ALL MOMMIES AND DADDIES ARE PROBABLY CRINGING NOW, THOUGH

What happened to “why?” What happened to make people so afraid of asking this question? Perhaps it’s when all of the kids start asking “why” about everything. Why does daddy have to go to work? Why do we need money? Why do we need to buy food?

Or perhaps it’s been the poorly trained hoard of bosses and superiors who have asked the question “why” which somehow was almost drowned out by their arrogant incompetence. When the purposely ignorant ask “why” it can be quite annoying.

It might even be the menacing “why” asked by the scary vice president at your company. You know, the question is phrased as an innocent request for information but which is dripping with malice and anger. “When you hear ‘why’ from the VP, you best run!”

But, “why” doesn’t have to be so bad. I think the greatest inventors asked “why.” So many “why” questions have lead to so many awesome discoveries. There is a very good side to this question.

I think “why” should be asked more often. But it requires the right framing. When used correctly, “why” can be one of the most important questions you can ask programmers on your team.

Before I get into further detail about how “why?” can help, I need to make something incredibly clear. When you ask a programmer “why,” you are not being disrespectful. You are trying to learn. So, don’t act like they did something wrong. This destroys the confidence in your desire to learn and can turn programmers defensive. Once you can completely grasp the fact that this is not meant to be a ‘check-up’ on the programmer, you can start to communicate this to them as well. When you ask “why?” you’re not meaning to say that you disrespect the decision that the programmer made.

I’ve heard managers say that if they knew how to do all the programming, then why would they need the programmers. When you ask “why,” we’re not talking about knowing every detail about the actual code. As a manager, you should have an intimate knowledge of the decisions that are made and the solutions that are applied. You may not have a complete understanding of the technology or its implementation, but you need to understand why it was implemented. That returns us to the core question and point here. You are asking “why” because it is your job to understand why. You don’t necessarily need to know how.

You are simply trying to get a better, broad understanding. This will help you make better decisions as well. You are learning from the programmer, not questioning their skill or integrity.

When you ask the programmer “why,” you get two benefits: the chance to make the programmer (and quite possibly yourself) think through the task thoroughly and the ability to provide feedback.

When you ask someone “why,” a unique thing happens in the brain. Instead of having this vague reason floating around in the brain, it has to be solidified and put into spoken word. This is something in itself. There are times when we make decisions without thinking about it. I bet in the last few seconds, you adjusted your arm, perhaps supported your head. You made the decision to move your arm to do this. Why? Was your head too tired? What if that cost you five cents every time you made a move with your arm. Would you think through each movement then?

The point is, when you ask the question, you get the person to put the time into describing and owning the decision. As a consequence to your question, the programmer must now own all decisions they’ve made and actions they’ve done.

There is no simple “just because” anymore. *(“Just because” causes business to lose tons and tons of money a year. Don’t believe me? Take a whole day and write down all the tasks you do. Then, ask yourself why. For fun, count the number of times you answer “I just always do it that way” or “just because.”)*

In addition to getting to hear the programmer’s rationale behind their decisions, you get to rehash the project in your mind as well. You might hear some crazy reasons from the programmer for what they’re doing. But, perhaps that’s because what you asked for is a little bit “crazy.” Or a more accurate example, if the programmer is giving you some really wild reasons, perhaps this can demonstrate that the benefit of the project compared to the work involved doesn’t add up.

Asking “why” is an organic opening for feedback. Sometimes it can be hard to find the time to remind your staff that they’re doing a good job. And, let’s face it, sometimes it’s easier to ignore a problem if you can just shut your office door and not have to work with them directly. This “why” time allows you to give them both positive and negative feedback.

Try it. Just a few times. You’ll see what I mean.

When you hear something that seems like it was a great idea, say so! Other times, when listening to a complex explanation, take the time to mentor and tutor the programmer along. Perhaps their solution is complex and doesn’t necessarily take into account the day-to-day business concerns. Take this time to teach them a bit more about the business, the part that you dwell in, so they can make better decisions in the long run. Of course, if they’re mistaken, this is the perfect time to point it out and help develop and discover the correct solution.

But there’s one more important thing you must do when asking the question of “why.”

Don’t interrupt. Ever. Resist the urge. Don’t do it. Wait. Just wait. You’ll have your turn.

When you first start out asking the “why” questions, it might already be stressful. But remember, nothing turns up the heat and ignites a defense more than interrupting the conversation. Let the programmer finish. Even if you know, or think you know how the story is going to end, wait, and let it end. Allow them to finish their thought. Make sure you can determine the difference between someone taking a pause for a breath, waiting for their thoughts to form, or just being nervous. Listen, listen, listen. *(I’m sure there are whole chapters or books in the management arena about the importance of listening. Do it!)*

Don’t interrupt during the answers to your “why” questions. This just causes stress, makes things tense, and kind of demonstrates your lack of respect for the programmer. Remember, by asking “why,” the investment the programmer is putting into an answer is actually really doing you a favor. They know the solution and are responsible for the end product. By the strictest sense, you don’t need to know the reasons for everything. But it sure helps you become a better boss and manage projects more effectively. Don’t interrupt.

When executed properly, “why” can be one of your most valuable questions. Show your programmers you care about their thoughts and decision process by asking about it. Take the time to give feedback and show respect. Ask “why” with a smile - and mean it. Something as simple as that will make you more informed and make you a better manager.

#30. Great Programmers Don't Always Know It

OR, HOW AND WHY I ALMOST RUINED MY LIFE

I don't remember anything from when I was in elementary school. I'm pretty certain classes were going on. I didn't pay attention. For whatever reason, I could read well. When other kids were struggling reading out loud to the group, I was always the kid staring at the ceiling, and then whispering the word they were stuck on so we could continue. I read the entire book right away, and then just daydreamed and really examined that ceiling. I also built lots of little army boats out of my pencil case, pencils, and erasers.

In middle school, I did basically the exact same thing. But, I also got a little bit more enterprising. And by enterprising, I mean business not like your corner store but more like the mob. Middle school was the beginning of new lockers with combinations. For some reason, a lot of my classmates could not open their new combination lockers. To me, it was super easy. They'd ask me to come and do it for them, giving me the combination for their locker. I would help them, walk away, and write down the locker position and the combination. *(See, our school said they changed the locker combinations each year. Instead, they just changed the numbers on the lockers. I figured out this money-saving scheme.)* Later on in the year, I was promoted to hall monitor during study hall. When I got really bored, I'd grab some of those locker combinations and open up some random student's locker. This was the beginning of my weight problem, too. Perhaps it wasn't helped by my ability to take twinkies and other cakes from the kid's bag lunch inside their newly pilfered locker. *(In hindsight, I know some of the kids packed their own lunch. I wonder what they thought when they remember packing their lunch with a Hostess treat, and then at lunch it was gone.)*

As time went on, I became an even worse kid. I started stealing CDs from lockers. Don't worry, I was finally caught, suspended, and had to do some restitution. But not before I tried every method of getting out of trouble, including putting the stolen CDs into another kid's locker to hide my crime.

In high school, I got really bored. I started doing drugs, running away from home, and basically causing all kinds of chaos. On a field trip, I circumvented the long line from our bus at Burger King by walking across the street to go to Taco Bell. That got me suspended as well. If you met me in my sophomore year, I think you would agree that I was a bad kid going down a horrible road.

But I wasn't. I think I gave away the real problem in the descriptions of my capers. I wasn't bad, I was bored. I had no way to channel my energy. In fact, I was probably actually really smart. Because I wasn't challenged, I turned into a very negative version of myself.

This same thing happens with some great programmers. But, it just manifests in a very different way. And there is one core reason for this.

Sometimes great programmers don't know they are great.

It's that simple. Or, they are never recognized as great. So, they act out. They may not really even know they're "acting out." But there are two ways that they turn bad. These are random tasks and negative reports.

Sometimes really great programmers will run off and do some task that is completely devoid of any resemblance to the project they're working on. If they are supposed to be making a website for an insurance company, they might also create the game of pacman in javascript. This clearly is not a requirement for this website!

The first reaction you might have as a manager is anger. How dare they defy you? Or, even worse, how dare they steal from the company! They could be finishing this project and moving on to the next one. Instead, they wasted billable, paid time making a useless game. What a bad programmer!

Rarely that's true. Only in very unusual cases is he or she a bad programmer and a bad person. Your programmer is just better than you realize. They are not being challenged. So, they have to do something that challenges them. *(Because I didn't feel challenged in high school, I regularly got high to introduce the challenge of keeping a straight face and not attracting attention to my reckless, stupid, illegal activity.)* The worst part of this scenario is that the programmer may not be self aware of the reasons why they did what they did. They might just say "I'm bored" if you ask them. But, that's a clue. Don't let it slip away.

The truth is that this programmer is better than he knows and you recognize. And, because they're better, you're not challenging them enough. So, they're acting out. Instead of punishing the programmer next time something like this happens, try to find out the root cause of this behavior. Find out if they were really truly being disobedient or if it was an absentminded task outside of the project scope without much thought. Chances are, they're not bad. They are actually really, really good. Your job is to find ways to harness this, keep them on track, and challenge them.

The other way that really good programmers manifest is through negative reports about everyone and everything. Initially, they might actually have an idea that they're good at their job. However, society makes it so abundantly clear that anyone who acknowledges their own greatness is conceited. "Be modest." "Don't tell anyone you're great. That would be bad." Before you know it, the programmer has allowed everyone else to beat them down and they become negative. Look at how awesome they are, but they can't tell anyone. Look at all the poor quality people who get promoted instead of them.

In other cases, negativity might come from the programmer never realizing that they're great. They might not understand what true level they're at. If they're at a high level, but everyone else is at an acceptable level in your team, this might be ok with you. But, from that programmer’s vantage point, he is at an acceptable level, and everyone else is below them, at a poor level.

Bring on the negativity.

You can recognize this phenomenon by listening intently to the reports the programmer gives you. Pick out any times where they brag about their own work (or toot their own horn, if you will). If your programmer is suffering from public negativity from internal greatness, you’ll hear none of the self-worth statements and prideful acknowledgements. Listening intently at this step is important. The lack of self-promotion is what separates the great programmers from the truly negative, dispassionate people that you might have to cull from your team. Listen for self-aggrandizing. If there is none, continue with the troubleshooting! Otherwise, thin the herd.

Next, listen for particular negative reports about fellow teammates or projects. When talking about other programmers, look for phrases that sound like “is not good enough,” “is too slow,” or “misses things a lot.” You might have to get clarification as to what specific scenarios the programmer is referring to. But, this can be another sign that the programmer doesn’t realize how good he really is. He looks at all the less talented programmers and reports them as inferior. Your job as a manager is to be able to determine the difference between a real report of useless programmers and a misguided comparison to the programmer’s own skills. If you can’t, you might actually believe that your team of good programmers is useless. If you get rid of them, you’ll have just one great programmer. He or she will burn out. Oh, and when you’re hiring replacements you might actually get the real low tier of programmers to replace your previously acceptable level.

I like to think about some of the cars I had growing up and use them to illustrate my point. I had a 1978 Chevy Impala. It had a large engine but it was a tank. It took a little while to get up to highway speeds. Nothing too insane, but you could tell it was working on it. Later on, I got a 2001 Monte Carlo. This car kicked butt. I could clearly say the acceleration was much better than the Impala. Now, I have a 2009 WRX STI. It has a 0–60 of 4.7 seconds. This really kills the Monte Carlo. The Monte Carlo was slow in comparison (7.5 seconds). If someone asked me what I'd rather have, I'd like to have my STI, not that slow Monte Carlo. But, if you remember, the Monte Carlo totally kicked butt compared to the 13.6 second 0–60 time of the Impala. However, if you could only choose between the Impala or Monte Carlo, you'd pick the latter every time. It's all about your frame of reference and comparison. Your team might be full of Monte Carlo cars. One of the cars is the STI, though. Don't let the fact that you have a fast car make you get rid of the other ones. You never know, when that STI finally is retired, you might end up with a replacement Impala!

Sometimes really good programmers just don't realize it. Watch them and recognize that perceived disobedience or consistently negative reports are signs of this condition. If you determine that the programmer is better than you or they realize, the fix is simple. Challenge them more. Also, don't back down from telling them what's going on and describing what they're doing. Confront them on the negativity. Explain that they've worked hard and are a great addition to the team. Teach them to understand every programmer is on their own journey. Help them attain a leadership position in the team perhaps. Or, find ways to give them the most challenging parts of the projects. Any of these steps will help you. Just don't ignore the great programmer and think they're just acting out.

#31. Always Be Perfect

IN OUR INDUSTRY, THERE IS NO AIRBRUSH

Throughout my career, I’ve had some pretty great bosses. Sadly, I’ve also had some poor ones. In either case, I still remember looking up to them. These were the men and women who made the choices that affected my work life. They had great power. They had my respect.

And then, they each invariably made a mistake. I was appalled. How could my manager have made such a huge oversight? Or an even worse reaction: “why is my boss being so dumb?” I wanted them to be perfect. I needed them to be perfect. Who wants to be lead by someone who make mistakes just like you do?

This feeling is something you need to remember when you become a manager. You’re no longer the programmer who can have bugs and forget things. You’re expected to be above this. You’re expected to be perfect. Your team will now expect that you’re impossibly perfect. Keep this in mind when you make your decisions.

Now is the time to turn on spell-check. A manager can’t have spelling mistakes in his email. Don’t you dare jumble the letters of an acronym.

Now is the time to look up words you use but you aren’t completely certain of their meanings. Don’t fully know what the acronym SaaS means? Look it up before you use it again. Do you have a habit of using the word “indubitably” but don’t know what it means? You just think it sounds cool? Time to look it up.

Now is the time to read the books on personality style and management training. That split second after you became a manager, you’re now expected to know all of these things. Of course this is impossible, but that doesn’t mean you shouldn’t try. It’s now your new set of responsibilities. Take the time to educate yourself on people and processes.

We all know that we can never be perfect but we need to keep this unreachable expectation in mind. Make all decisions and choices as if your whole world on display. Your team definitely is watching your every move now.

Having a team that expects you to be perfect isn't such a bad thing. This accountability can be leveraged into a good thing. Acknowledge that you're not perfect, but share that it is your goal to be the highest quality and caliber that you can be. Remember, your team is counting on you to reach these heights, so use that to motivate yourself. You'd be surprised, when no one expects someone to be good, it is easy to be subpar. When others are expecting, encouraging, and challenging you, even us normal folks develop insatiable desires to reach greatness. We have a support system now. We have people rooting for us. Being great isn't something that is reserved for those crazy people with the insane internal drive anymore. You can do it now, too.

When you become the manager, you are expected to be perfect. You never will be. Acknowledge this. But, take every care to reach a higher level, learn more, and become more accurate. People are watching you. Be an example of what you'd expect of your manager, and what you'd like your team to emulate.

#32. How to Deal With the Idiots Upstairs

THE BIGGER THE DINOSAUR, THE SMALLER THE BRAIN?

I struggled with where to put this topic. I didn't know if it made more sense to include in the manager section or the programmer section. In the end, I decided it belonged here. A manager's reaction to a challenge situation can have a deep, rippling effect on the team. I think it's more important to address your reaction as a manager, but this concept really applies for any position on the team.

Sometimes, a completely insane requirement comes across your desk. It may be for a new programming project or a fresh HR initiative. I've had many of these questionable requests. *(Oops, I just switched back into manager speak. I meant insane requests.)* You stare at it and just get incredibly confused. What is the point of this? Or even worse, I think this is going to hurt the company!

The first thing to do is to make your case. Don't sit back and let things flow downhill. Remember, you were there once. Don't make the same mistake that some employees do. As a manager, remember the company is a conglomeration of all of the work and decisions each employee of the company does. If you think it doesn't affect you, you're wrong. You are just as responsible for any decision as anyone else in the company. You should always be trying your best to make sure the company succeeds. Sometimes, that means taking a stand against silly or dangerous projects.

Put together your thoughts and make your case. If you can sleep on it, do that. Try to remove the emotion. Assemble a counterpoint and proposal that includes the following parts:

First, demonstrate that you understand the requirement. Explain that you understand each point. If you can't do this, you shouldn't be disagreeing anyway! You could be missing something that makes your total argument a waste of time or invalid.

Second, suggest or predict what the negative ramifications could be of this project. Depending on your communication style, don't forget to prioritize these in order of impact. Mixing minimal impact and epicly dangerous, high impact topics with one another causes each topic to blend together. Instead, start with the worst case or the best case *(glass half full vs half empty?)* and work your way to the other side of the spectrum. Remember to keep your predictions based on actual experience and fact, not on your feelings. Speaking of feelings, repeatedly take time to review your work so far to make sure you're not communicating with any emotional statements.

Finally, give suggestions. You must come up with some alternatives otherwise you're just one of the many complainers that fill up cubicles each day. You know the type. Don't be like the annoying ones that come to you complaining about a situation but don't have a solution or suggestion to alleviate the problem. You're just supposed to know. Do yourself the favor of not passing this inconsiderate method of communication upward.

Once you have a chance to communicate your case, you have two outcomes. Either the submitter will apply your changes and move the project forward, or you'll lose. If you're lucky enough to get a new proposal implementing all of your changes, great! Move forward and get it done. But, do not brag about it. Don't tell your team about the horrible proposal that was submitted. Resist the urge to tell them about how you saved their asses. Instead, bring the proposal forward like any other. This is a project that we've been given, let's do it. The difference here to recognize is while you're a team player, fighting for your staff, you are not peers. You shouldn't share with them these wins. The people who need to know (your boss), already know. If you really need to share this, find a friend. Don't share with your employees like this. All this does is make you sound like a blowhard and diminish the respect and authority of those above you. That's not your job.

The other outcome is that you might lose. You might lose hard. That's ok. Just eat it. Don't complain. Don't go and grumble to your team about how bad the proposal is or how you fought all these points. The more you complain to others, the less professional you look.

Your job is to be a representative of the company and its decision makers. Take this seriously. If you complain downward, you're only bringing yourself to a lower level, removing any shred of respect.

If you really must answer questions from your team about the project, be polite in the way you handle it. I'm not suggesting you lie, just frame the response in a way that may not add your personal support to the proposal, but backs it with your position. "This isn't necessarily the way I'd personally do this, however, I think we can make this work." "The decision has been made to do this set of tasks. I think if we put our best foot forward, we'll still be successful. If it turns out bad, we can use this as an experience to help teach others why they might want to do it a different way." It's all about framing it the proper way.

One last little note about complaining and sharing. One of the best tips I've ever heard was one sentence: "Share downward, complain upward." The point of this is simple. When you have information (that you can share), share it with those on your team. Give them more information, it empowers them. It allows them to make decisions. However, when you have complaints, complain upward. Find your boss, shut the door, and make your case. Do not go to your peers or to your team and complain. While it may make you feel better in the very short term, that's not the path to being a successful manager.

#33. Get QA Involved Sooner

UNLESS YOU LIKE BROKEN SOFTWARE, THAT IS

If you have a QA member or team in your programming group, you're lucky. I know you may initially find that hard to believe. Doesn't all the drama seem to come out of the programmer / QA relationship? At least in a few teams I've been on, this sadly has been the case. It's too bad, too. They're great members of the team. They're needed.

And, if you don't have a QA team, let me just take a quick second to tell you to get one! If you can't afford one in your budget, then take one of your programmers from a different project and have them act as QA temporarily. Of course, make sure to rotate this for each project so that one programmer doesn't have to do QA all of the time. Using another programmer for QA is the least-worst "best" case compared to not having QA at all.

I could ramble on about the importance of QA forever. Let's move forward with the assumption that you now have QA staff available. Now, lets correct the next mistake that I've seen over and over. It's time to stop putting QA at the end of the programming schedule.

Programmers will insist, kicking and screaming, that projects can't be tested until they have completely built the application. Sorry, but this is wrong. This objection happens for one of two reasons. First, it could be that the programmer hasn't been disciplined to program features in a linear fashion. They may complete portions, get bored, and move on to another feature. This leaves most of the project completely broken until the last few minutes when everything comes together. How in the world could anyone test that?

The other reason for this objection is because the programmer can't figure out or imagine how someone could or would want to test a project that wasn't completely finished. They might question when the newly created user authentication system should be tested. Should testing wait until the project is complete? What if he chooses to refactor something and break the user authentication. Won't that make that whole testing scenario invalid? Don't let this stop you from getting QA in earlier. Remember, this is just the programmer not understanding the concepts of QA. That's fine, as much as programmers would like to think it, we can't be masters of everything.

The answers to these concerns and questions could take up a whole testing book so I won't be diving into them now. That's a topic for a testing based book. But suffice to say, with proper training and information, the arguments against getting QA sooner are void.

Since we've squashed these two arguments, there is no other reason to put QA at the end of the programming task. QA should be involved sooner. QA should be involved immediately.

When the project charter or definition has been finished, you would generally involve your programmers. Now, also involve QA. While the programmers are reviewing, estimating, and creating a task list, the QA resource will be doing the same thing. Instead of figuring out what classes to program and what languages are required to complete the task, QA will be working on developing use cases for the business as well as identifying potential areas to test/target for bugs. They should probably create their own test plan.

Now, as soon as any feature is even remotely available to test, assign QA. QA should run through the wireframes as soon as possible or test the first draft of written code if no wireframes are available. Some teams function by a paradigm that requires the programmer to envision the solution for every request. Whatever the programmer thinks of is what will make up the final product. QA should just be plugged in to validate that the solution the programmer made doesn't have any major bugs compared to his interpretation of the solution.

But, that's not the right approach. Instead, involve QA right away to validate that the programmer is making assumptions and creating things that make sense when compared to the documentation and project request. As individual features come out, QA tests them. This allows them to catch the bugs sooner and to perhaps deter a scenario where the rest of the application is built on a faulty assumption.

Architecting your testing in this manner may require a test cases to be ran repeatedly at different parts of the project. If there is a large refactor, tests should be ran again. But, this is ok. The more testing, the better the product will be. Remember, the QA tester position is different than a programmer. The idea of repetition destroys a programmer's world. A good QA resource understands the importance of their testing and comprehends that repetition is just part of the task they need to do. *(Hopefully QA has some automation tools at their disposal, though!)*

In the beginning of this chapter, I mentioned my experiences with drama between programmers and QA staff. This really happens. Think about it. As a programmer, you just put your heart and soul into some code. QA comes by and says "nope, doesn't work" and sends it back. How dare they not understand and appreciate the work you've put in. You were even up late last night programming the feature they just rejected. At least acknowledge that most of it works! Or that it looks cool! Don't be so cold or callous. *(Ok, so maybe you're starting to see some of my diva programmer sneak out, but people do think like this!)* This piece of art *(if you will)* isn't up to par according to QA. This can cause anger and anxiety.

It's even worse if you measure your programmer and QA performance on bugs created and bugs reported. You've immediately put them head-to-head in a battle to protect themselves and build their reputation, not to create better software.

Now, let's flip it around. Imagine this scenario: as a QA person, you test the signup process. You find three bugs and they're fixed. Next, you move on to the forgot password section. You find one bug and that gets fixed. You are told that some of the code has been refactored, so you have to go back and test signup again. Ok, understood, so you go back and test again just to be sure. The same three bugs exist again! Oh, and now there's a fourth one. What is wrong with the programmers? Don't they get it? Don't they use their software? You already reported these bugs, so why did their new programming allow these regressions? This type of scenario can be the cause of some of the ill will between the two teams too.

Once again, I restate the solution to all of these problems. Bringing everyone together sooner. Keep the communication open. Don't just put QA at the end making it a battle royale before the software is released.

One interesting phenomenon I've noticed with some QA resources is the ownership they develop of the software increases dramatically the sooner they get involved. This is always a good thing. The more people who feel ownership in the project, the better the outcome of this project and the higher quality product released. I've personally seen QA resources defend the programmers and talk great about the product even in the face of a large bug report. Why? Because those QA people were brought in sooner, as project members, not towards the end as custodians.

Want success with your QA team? Bring them in sooner. Make them get involved as soon as the programmers do. Keep lines of communication open. You'll see greater ownership in the project and less animosity between the teams and those with potentially opposite responsibilities.

Endnotes

Thanks so much for reading this book. I’m truly honored to have shared these 33 things with you. As I hope you can tell, I’m very passionate about making sure that everyone in our industry can enjoy a level of success greater than they have already. If you have any questions, please do not hesitate to reach out to me at aaronsaray.com. I wish you the best while you travel along your path to becoming a great programmer or manager.

Special Thanks and Acknowledgements

Equally important are both friends and work colleagues for each provided the insight, the motivation and the encouragement to complete this book.

I thank my friends for the encouragement, insight, and sometimes the competition that they fostered in me. Thank you Crystal Cichon, Eric Lightbody, Frank Cichon, Mark Skowron, Joel Clermont, and Jeremy Dee.

I thank my work colleagues both past and present, who provided insight whether they knew it or not. Thank you David Lundgren, Mark Hillebert, Jenny Bennett, David Hoover, Keith Alberts, Clare Zajicek, Chad Stovern, Heather Dorsey, Jake Ewerdt, Billy Gilbert, James Rodenkirch, Brandon Danielson, and Andy Karnopp.

Finally, I thank the managers, bosses, and owners who I've interacted with. Some demonstrated great examples of leadership while others showed me the need to develop my skills further to be successful. Thank you Jared Alfson, Jason Keup, Mark Neumann, and Joe Luedtke.