Including tips by top developers from
Google, Microsoft, Spotify, Amazon, and more

14 Habits of Highly Productive Developers

Zeno Rocha

Including tips by top developers from
Google, Microsoft, Spotify, Amazon, and more

14 Habits of Highly Productive Developers

Zeno Rocha

14 Habits of Highly Productive Developers

v1.0.0
Date: July 14th, 2020
Site: 14habits.com
Email: zeno@14habits.com
Illustrations by Briza Bueno

ISBN: 978-1-7352665-0-3

14 Habits of Highly Productive Developers

Part One: Principles

Hello World

“There is no elevator to success, you have to take the stairs.” — Zig Ziglar

I started developing software more than ten years ago. Since then, I’ve built many applications, created dozens of open source projects, and pushed thousands of commits. Besides that, I spoke in more than a hundred conferences, and had the opportunity to chat with a ton of developers along the way.

I was fortunate enough to be in contact with some of the best software engineers in the industry, but I also met a lot of programmers who are still doing the same thing for many years.

What separates one group from the other? What’s unique about people who work on the biggest companies in our industry? What’s special about individuals who create the most used applications in the world? How can some developers be so prolific at work and also outside their jobs?

These questions stayed on my head for a long time. I realized that I could buy the best mechanical keyboards, go to the most famous tech conferences in the world, and learn all the newest frameworks. Still, if I cultivated bad habits, it would be impossible to become a top developer. Because of that, I decided to reach out to the best developers I know and ask them tips on how to be more productive.

This book doesn’t offer a straight path or pre-defined formula for success. This book is the result of a quest. A quest to uncover what habits can be cultivated to help become a better software engineer.

Methodology

This is not a traditional book. You won’t find the same format or structure that a regular book has. In fact, this book was designed to be as simple and objective as possible. You can follow the order of chapters, or you can read them individually. Everything is standalone and doesn’t depend on previous knowledge.

At the end of each habit, you’ll find a section marked as **“Questions & Answers”**, where I interview senior developers and tech leads from various companies to understand how they got there. I went after tech giants such as Google, Amazon, Microsoft, and Adobe. Powerful startups such as GitHub,

Spotify, Elastic, Segment, GoDaddy, and Shopify. All the way to established organizations such as Citibank, BlackBerry, and The New York Times.
These people come from all over the world and have a pretty diverse background. From San Francisco to New York. From São Paulo to Montreal. From London to Stockholm. The idea is to present you not a one man's point of view, but a collection of insights on how to navigate your career.
You'll also find sections marked as **"TODO"**, where I encourage you to reflect on certain topics and take action with specific directions. I highly recommend taking a few minutes to dive into them since this will generate even more knowledge.
And finally, you may find some sections marked as **"Bonus"**. These are extra content that I prepared for you. They can be found outside the book and will complement what you're reading.

Why Habits?

If you've ever tried to lose weight, you know how frustrating that entire process is. You can exercise as hard as you can for three hours, but if you do that only once in a week, it will have zero effect on you. What truly generates results is when you go multiple times per week. Then suddenly, a few months later, you'll start noticing changes in your body.
Consistency matters, and that same concept applies to your professional career as well. Things take time, and intensity is not always the answer. The habits you decide to cultivate (or not cultivate) will determine your future life opportunities.
As described in the book *Atomic Habits*:
"Habits are the compound interest of self-improvement. The same way that money multiplies through compound interest, the effects of your habits multiply as you repeat them. They seem to make little difference on any given day and yet the impact they deliver over the months and years can be enormous. It is only when looking back two, five, or perhaps ten years later that the value of good habits and the cost of bad ones becomes strikingly apparent."
— James Clear
I often get emails from other software engineers asking what programming language they should learn. And while that's a very important question to ask, I feel like a more valuable question would be: "What habits do I need to cultivate in order to be effective in *any* programming language?"
That's why I decided to focus this book on habits instead of tactics.
Now let's dive in! Are you ready?

Part Two: Learning Habits

Habit 1: Look For The Signals

The oldest, shortest words – yes and no – are those which require the most thought." — Pythagoras

Every single day we're bombarded with tweets, newsletters, and videos telling us what we should do, what we should learn, what we should focus on. We are constantly faced with FOMO (the fear of missing out). What if we're wasting our time with the wrong programming language? What if the most productive framework is not the one that you're currently using?

For me personally, it all started with choosing the right Operating System (OS). At the time, I had a Windows machine, but everybody was telling me that Apple computers were better and I should switch. However, their prices were way out of my league, so getting one was not even a possibility. Fast forward a couple of years and one of my employers gave me the option to choose between Windows and MacOS. I went straight for the MacOS to see what the fuss was about and what was so incredible about it. After some time using MacOS, Ubuntu started to become very popular and everybody was telling me I should switch to Ubuntu. So I thought I'd give it a try and started using it. My point here is not to tell you which one you should choose, the point here is that **there's no such thing as the best tool**.

Instead, we should practice JOMO (the joy of missing out), which is mostly about being happy and content with what you already know. Give yourself some credit and look back on everything you learned so far. Of course, you shouldn't be complacent and stop studying new technologies. It would be best if you find some balance between practicing your existing skills and learning new ones.

People will try to convince you which is the best OS, the best programming language, the best framework. They will tell you about all the amazing things that *their* tool does and that *your* tool does not. The reality though is that every single tool is different and we are also different as users. What is best for you, may not be the best for me or for others.

Think of it as a radio station that you're trying to tune into. I know you probably don't use radios anymore, but stay with me. Imagine you turn the dial and it's just picking up static noise, but after a few frustrating seconds, it finally manages to pick up a signal and tune into a station. The signal is the meaningful information that you're actually interested in. The static noise is just the random,

unwanted variation that interferes with the signal. That's why self-awareness is so important, you need to be able to identify what is the signal and what is just noise.
It's crucial to understand that the noise will always be there. You don't need to necessarily abandon social media, unsubscribe from all newsletters, and stop watching YouTube videos. A digital detox can definitely help for a while; however, it's not a long term solution. What you need to do is to cherry-pick what is relevant to you at this point in your career.
Accept the fact that you simply can't learn everything. **Remember, desires are endless; needs are limited**. Accept the fact that newer is not always better. There are people working with ancient programming languages and still making a lot of money. Practice daily the subtle art of saying 'no'. No to that newest library. No to that fancier platform. Say more 'noes' so you can say 'yes' to what really matters to you.
Hear the noise, but only pay attention to the signals.

```
// TODO
Create a list of all technologies and tools that you would like to learn. Now label each of them with a different priority: "This Week", "Next Month", "Next Year". Whenever you feel like you're missing out on some new shiny trend, revisit this list and reorganize the priority.
```

 # Questions & Answers: How do you decide which technology to learn and invest time in?
Daniel Buchner (Microsoft):

"Earlier in my career, I remember paying close attention to every new framework that would land on the front page of *Hacker News*. I would read up on whatever unique approach they would take, and spend extra cycles tinkering with the framework or library to get a sense of it. This is perfectly fine if you have ample free time to explore and learn, but can often lead to fatigue, because the list of hot new things is never-ending.
These days, I try to focus on a few key things that drive my evaluations:

1. What are the absolute must-have technical requirements for whatever I am working on? This may include things like performance, desired UX, or interoperability with target systems.
2. How easy is it going to be for others to work on what I am building, and what will it look like for them to integrate it in whatever projects they are working on?

3. Is what I am doing aligned with the trajectory of the open web, standards, and specifications I believe will stick around over the long term?

The three points above tend to weed out many of the libraries, frameworks, and other utilities I come across. This has saved me time and allowed me to focus more on delivering what I need to, instead of getting caught up in dev-tourism."
Addy Osmani (Google):

"Accept that you can't learn everything, but you can learn enough to be effective. When it looks like an idea, framework or technology is gaining some traction, I'll invest an afternoon in trying it out myself to get a sense of two things: 'Does it improve my productivity?' and 'Does it improve the user-experience of the types of projects I usually build?' If the answer to either of these is yes, I'll consider spending time learning about the framework or technology in more depth.
Because our time is finite, there are plenty of technologies that I'll try, will find compelling enough to keep an eye on, but will make a trade-off about learning and investing in in favor of something else. I think that this is healthy. I originally tried (and punted) on React the first year it came out, but now use it regularly. I like a lot of the ideas in Svelte, but because I had already learned React, Preact and Vue, decided that it was a better investment of time to level up in completely different areas with that time, like the Web Animations API.
As I said, with time being finite, it's good to find a balance in what you choose to learn to keep yourself effective. When you pick just a few, high-impact choices you set yourself up to have enough time to get more depth in your technology of choice. This can be immensely useful when you're trying to building anything non-trivial in the real world."
Loiane Groner (Citibank):

"I can answer this question after asking myself a few more questions. What problem does the technology solve? We, developers, are paid to solve problems, we're not paid to necessarily write code. Writing code is one of the consequences of solving the client's problem with technology. So what problem is this new technology or framework trying to solve? How is it different from existing frameworks/technologies? Do I need to learn something new to be able to learn it? Can I take advantage of my existing skills and knowledge (does the framework use a language I already know)? What about infrastructure? Are there cloud providers that support the new technology, or can my existing infrastructure support it? These are only a few of the questions I would ask

myself before learning a new technology.
Besides the questions, you should always learn an overview of the technology, so you know what it's about. In my humble opinion, there is no such thing as too much knowledge. Maybe you have a scenario in the company that you work for or a personal project you are trying to put in practice that can use this new technology.
And at last, keep an eye on companies that are adopting the technology and putting projects in production. This is the most critical part. We all can code POCs (proof of concepts), but to really learn all scenarios and also what's going to be required to maintain a project, it's crucial to ship it to production. If a lot of companies are adopting the technology, maybe it's worth to deep dive and try to learn more advanced concepts about it."
Netto Farah (Segment):

"I would say this is something I still struggle with these days, but I think my experience and overall gut feeling have been good compasses that have helped guide me in scenarios like this.
I'll try and pattern match the new frameworks against previous experiences. React, for example, seems pretty similar to Flex in some regards, which was a well tried and tested idea. Next.js feels a bit like Rails as far as DX is concerned, so I also consider that to be good signal.
Some more drastic paradigm shifts, however, are a little more interesting and harder to judge from just a glance. For those, I tend to experiment with the technology a bit before investing too much time in them.
Another point to consider is how transferable the acquired knowledge can be. For example, learning Rust vs. Go: Two pretty distinct tools that are competing in the same problem space. Rust seems a lot cooler and sophisticated, but Go is very likely the language that's gonna help you land a good job. In that case, I would personally optimize for the one that can help me reach the next step of my career."

Habit 2: Focus On The Fundamentals

"You can practice shooting eight hours a day, but if your technique is wrong, then all you become is very good at shooting the wrong way. Get the fundamentals down and the level of everything you do will rise." — Michael Jordan

In March 2009, I was studying at university and I had a professor, Mariano Pimentel, who forced us to write HTML/CSS on the Notepad. Of all the apps

that you can choose to write code, Notepad was one of the worst. At that time, Dreamweaver was the *coolest kid on the block*. It had a friendly GUI, real-time syntax checking, and code hints. In contrast, in Notepad, there was no syntax highlighting, there was no autocomplete, there was no live preview, there was absolutely nothing to help you to be more productive.

That entire semester had a great impact on me. Even more than 10 years later, I still remember all the HTML tags and CSS properties, because I had to manually write them without any further help. If I had forgotten even just one character, the entire thing wouldn't work, so I needed to make sure that I completely understood what I was writing.

I'm not advocating that we all follow the same path and start abandoning our beloved code editors. The crucial idea is that learning the fundamentals will set you up for the future.

If you want to become a painter, it's essential that you learn some important fundamentals. Understanding *color theory* will help you evoke different meanings, compositions, and moods. Understanding *perspective* will help you with the placement of shadows and lighting. Understanding *form* will help you make objects appear three dimensional, giving them the illusion of volume. Understanding *anatomy* will help you draw characters with the right proportion and body structure. You don't necessarily need all this knowledge to do your first painting, but I'm sure that you cannot become a great professional painter without knowing these topics.

The same applies to you. If you decide to become a great developer, it's important to understand core concepts such as algorithms, logic, network, accessibility, security, and user experience. You don't necessarily need them to build your first app, but knowing them will help you build the next ten complex applications you will create in the future.

/* Bonus: The open source community built a roadmap of fundamentals for DevOps, Front-end, and Back-end developers. Go to 14habits.com/bonus/2 to grab it. */

```
// TODO
Spend some time researching what the fundamental concepts in your field are. Grab a piece of paper and divide it into two columns. On the left side, list all the knowledge that you already have. On the right side, list all the knowledge that you still need to acquire. Plan a dedicated time of your day to study those concepts.
```

 # Questions & Answers: How would you approach learning a new programming language today?

Lais Andrade (Google):

"I usually approach new languages as I'd approach learning any new framework: I try to get a basic understanding of the key concepts and search for examples of how others solve similar problems with it. This usually covers a lot of best practices, as you can see patterns on how certain structures are used, and from there, it's easy to find documentation about the reasoning behind those choices.
Learning a new language usually comes to me with the need to apply it right away on a project. I'm not a programming languages savvy, so I tend to get in contact with new ones as I need them in my day-to-day work. Having peer reviewers that are familiar with the language and the team's codebase is also very important to quickly learn new things while making progress.
And the most important thing I try to do is to always understand everything I write. Whenever I find myself confused between similar concepts, or whenever someone suggests a new feature I never saw before, I try to read the documentation and understand the theory behind them. I need to be confident about why that is the best choice possible."
Fabio Costa (GoDaddy, Ex-Facebook):

"I would learn a new programming language by picking one of my existing pet projects and rewriting it using this new programming language. In my opinion, this is the most efficient way to learn a new language, because I can focus 100% on just learning the language and nothing else. Picking a completely new project, for example, might require thinking about new features, documentation, and other things that would take my focus away from learning the language."
Michael Lancaster (BlackBerry):

"Because I don't have a formal degree in the computer science field, it was very important early on to find the most favorable way for me to learn new technologies while continuing to adapt them.
It all starts with ABC (Always Be Coding), as cheesy as it sounds, practice makes perfect. For that reason, when learning a new programming language, I hard commit to working on it at least 1 hour a day for about 30 days consecutive.
I go on YouTube and pick tutorials to build projects but I choose authors that go more in-depth on the fundamentals.
While following the tutorials, I always reference the official documentation, projects' unit tests, and popular open source projects to help frame my foundation and habits as close as possible to those already experienced in that particular language."

Habit 3: Teaching Equals Learning

“Power comes not from knowledge kept but from knowledge shared.” — Bill Gates

Last year, I gave my 100th presentation at a developer conference. Since 2011, I have had the opportunity to visit 16 countries and 71 cities to present. I’ve flown 712,575 miles, which is the equivalent of 28.6 times around the Earth or 3 trips to the Moon.

These look like impressive numbers. When you read them, it might seem like I have it all figured out. The truth is though, I don’t. I still feel a lot of fear before going on stage. *Every. Single. Time.*

I’m an introvert. When I was a kid, my mom used to say that I walked down the street looking at the floor, that’s because I was so shy I couldn’t look at people’s faces. At school, I never wanted to present anything in front of the class. I was extremely timid. I wish I could say this was all in the past, but it’s not. Even after giving more than a hundred presentations, I still feel insecure.

Then why do I keep putting myself in this position? Why submit a talk? Why record a video? Why write a blog post? The answer can be found in a simple phrase that my father used to say to me:

“If you really want to learn something, you have to teach it”.

The process of digesting content to someone is what makes you really understand it. The process of writing down how to do something is what makes you memorize it. The process of teaching is what makes you truly learn it.

The catch is you have to do it in public. There’s no way around it; for this to work, you need a human being listening on the other side. If you decide to do this, let me tell you something: It’s not going to be easy, and you’ll want to give up many times.

There are many ways to teach other people. You can start by sharing links on social media, writing blog posts, and recording screen casts. My favorite one is still giving talks.

According to many studies, most people’s number one fear is public speaking. Number two is death. That’s right, death is number two. You’ve probably seen and heard this phrase a thousand times. I bet it’s written in every public speaking training in the world.

When I look back at all those presentations I gave, I can identify a pattern between all of those experiences. I can even summarize it in five acts.

Act 1 – The invitation: It all starts with a random email. You’re living your normal life and then suddenly a notification arrives. It’s from an event organizer who is excited about the work you do. They ask you to travel to a different city — perhaps even a different country — to share the knowledge you have. You accept the invitation and move on with your life.

Act 2 – Reality settles in: A couple of weeks goes by and one day you take a look at your calendar and suddenly realize how fast the conference is approaching. You panic a little as you realize you haven't started to work on the slide deck yet. You know that you have to stop everything to start working on this, but there's always something more important to work on.
Act 3 – Freaking out: Suddenly, it's the week of the conference. You haven't been able to put your ideas on paper yet. You start to regret it. You ask yourself: *"Why did I even accept this invitation? What excuse can I give? I'm just a developer, I'm not a public speaker!"* Still, you made the commitment to go. Now the airplane tickets have already been bought, the hotel reservation has been made, they already announced your name, and you can't go back. So what do you do? Somehow you find the time, somehow you find the motivation to finally do this.
Act 4 – The moment of truth: It's the day of your talk. You travel to the venue and anxiety starts to take over. You're about to face hundreds (or maybe even thousands) of people. You are scared. You feel like a fraud. You know that there are much better people out there. But once again, you can't go back. So what do you do? You just face them and try to do your best.
Act 5 – Let's do it again: The presentation ends. Your adrenaline is extremely high. You are super relieved it's finally over. Some people reach out to you in the halls of the conference. They ask you about some ideas that you shared. Some questions don't really make sense. Other questions are extremely interesting. You feel amazing. You feel like you have made an impact on people's lives. A couple of months later, you submit a talk to give at another event and it all starts over.
This may seem crazy, but if you've ever given a presentation, I'm sure that you can identify the same behaviors and patterns. Yet still we submit ourselves to this ordeal. And maybe the reason why is because, at the end of the day, the person who gets the most out of these teaching experiences is not someone in the crowd, it's us!

```
// TODO
Find an event online and submit a presentation. Open a screen share software and record yourself doing something. Create a blog and share an article. Choose any topic that you want to learn and try to teach it instead.
```

 # Questions & Answers: Have you taught anything in public? How did that affect your life?
Fernando Tadashi (Adobe):

"I'm not in the habit of giving lectures to large audiences, but as I work in the

consulting area, I have an audience that always demands the maximum from my technical knowledge. Meetings and presentations with clients help me not only in interpersonal relationships but also in self-control over several situations, mainly those involving more significant stress.”

Loiane Groner (Citibank):

"It’s been a little more than ten years since I decided to give back to the community. I have presented talks, written a few books, published some videos on Youtube, and was also invited to teach a class for an extension course at the university I graduated from. It has been a fantastic experience and has taught me a lot.

One way of truly knowing a concept or technology is trying to create a practical project. Another very effective method is when you try to explain it to someone else. This allows you to realize the gaps you have, so you deep dive and study more about the topic you are trying to teach. In other words, teaching is a great way to learn about a specific topic.

Teaching and presenting talks publicly have transformed my career. I’ve had the opportunity to travel to new places, meet new people, make new friends, and talk to people I’ve never imagined I would have the chance to do. It’s important to note that you do not need to be an expert in a topic to present a talk; you can present about technologies that you are learning internally for your co-workers, at meetups, or even publish a video about it."

Addy Osmani (Google):

"Yes! Teaching others changes you for the better. It forces you to critically take a step back and ask, ‘Do I really understand this topic as well as I think I do? Could I explain it to a beginner in a minute?’. Sometimes when you need to explain something to someone else, it challenges you to simplify it down to the most core idea. This can be a great forcing function for improving your knowledge. It’s also a great skill to have in the tech industry because you’ll often convey ideas to folks with different roles or who have a different context than you have.

Teaching others also helps you question how well you want to understand a topic. You may have breadth in a field, but do you have depth on topics you’re interested in? Does it make sense to go deeper? Take front-end engineering - there’s a lot going on - new Web APIs, JavaScript, CSS, how browser engines work, how hardware works, how frameworks work etc. But your level of understanding of each topic likely varies. I may feel I understand JavaScript, but my depth in it is not going to match that of a low-level JavaScript engine

engineer. That’s totally OK. The topics I want to teach may not require me to do that deep, but going deeper than a surface level understanding is great because it helps you reason about the trade-offs even better.
I ultimately feel like a good measure of how well you’re able to teach a topic is, do you understand it well enough to be effective? If the topic is say, a tool - do you know when you shouldn’t use it? As you can tell, the biggest thing teaching has opened my eyes to is the value of questioning how well you know what you know and equipping you with the ability to decide if you want to know more."
Manuel De La Peña (Elastic):

"I’ve taught in public for around 15 years or so. Even before turning into a developer professionally speaking, I worked as a teacher in an academy. I never thought about this, but while writing these lines, I can see myself as more self-confident when talking in public nowadays thanks to this experience.
When giving a talk, you have to read, maybe code a few examples, prepare the slides to share with the audience, and condense all the information to make it understandable. During this process, you are learning about the topic you are going to present. Because giving a talk does not mean you are the most expert person in the world about something. For me, it means you want to share a piece of knowledge with others."

Part Three: Daily Habits

Habit 4: Be Boring

“Everything that needs to be said has already been said. But, since no one was listening, everything must be said again.” — Austin Kleon

In the software world, intensity is always praised. How many times have you heard someone say *“I spent the whole night programming”* and everybody around was impressed? We admire those who go the extra mile to get something done. I can’t count how many times I’ve slept at the office, but I always remember feeling like a superhero after doing so. On the other hand, how many times have you heard someone say *"I can’t handle this anymore, I’m burned-out"*? There’s a fine line between intensity and burnout. Many of us have already crossed this line and it’s very hard to get back.

Don’t get me wrong. I completely understand when people have to work extra hours in order to hit a very specific deadline or to resolve an emergency that is affecting users in production. However, when that kind of behavior becomes normal, you should stop and pay attention to what is truly happening. Intensity is a tool that you can use on special occasions, but if that’s your *modus operandi*, you won’t be able to sustain it for a long time. Just think about who you would rather have as a co-worker.

Who is a better designer? The one who is constantly trying to design solutions that are going to help the user? Or the one who only builds stuff when they feel like it?

Who is a better project manager? The one who is constantly doing follow-ups with the team and other stakeholders? Or the one who only talks to you when the client is pressuring?

Who is a better product owner? The one who plans, takes notes, and prioritizes tasks consistently? Or the one who only works when they are motivated?

Now, who is a better programmer? The one who is constantly trying to become a better version of themselves? Or the one who only cares about testing their code, improving the performance, and documenting their solution when they are obligated to do that?

Breaking the rules. Doing whatever you want. Those concepts are all admired by us. Deep down we all want to be rebels. Discipline. Consistency. Persistency. Those concepts are not sexy at all, but they are the key to playing infinite games.

In 1986, the philosopher James Carse introduced the concept of two types of

games in his book *Finite and Infinite Games: A Vision of Life as Play and Possibility*.

A finite game is defined as having known players, fixed rules, and agreed-upon objectives. For example, soccer is a finite game. Eleven people on one team play against eleven people on another team. You have to use your feet or head to score a goal. And after 90 minutes of play, the team that has scored more goals wins the game. Those are the players, rules, and objectives. You can't have one team playing with twelve players, you can't score with your hands (unless your name is Maradona), and you can't change the objective to be the team that scores fewer goals wins.

An infinite game, on the other hand, is defined as one where there are known *and* unknown players, the rules are changeable, and the objective is not to win, rather it's to keep playing. Because there are no winners and losers, what happens is that players drop out when they run out of the will or resources to play, which then leads to them being replaced by other players. For example, marriage is an infinite game. There are always at least two players involved, while the wider family and children also play a big part in that game. Boundaries exist and rules should be followed, but these can change over time and depend on who is playing. There's no way you can "win" the game of marriage, there's no finish line, and you can't declare yourself the #1 best wife/husband. The objective of marriage is only to keep it existing in a healthy and respectful way.

In 2019, Simon Sinek picked up on these concepts and wrote a book about them called *The Infinite Game*, in which he describes the challenges that emerge with these types of games.

When there's a finite player competing against another finite player, the system is stable. And when there's an infinite player competing against another infinite player, the system is also stable. Problems start to happen when there's a finite player competing against an infinite player because the finite player is playing to win and the infinite player is playing to stay in the game.

Programming is an infinite game, even though most of us act like it's a finite game. The players are known and unknown — there are always new programmers entering the job market. The rules are changeable — new paradigms emerge, new patterns are invented, new bugs are found. The objective is to keep playing — you can't win in programming, you can only keep evolving the software to be better, more scalable, and more useful every day.

Programmers who are playing the finite game are focused on their bonus at the end of the year or waiting for that freelance project to end. Whereas programmers who are playing the infinite game are less worried about intensity and are more focused on consistency. They don't know exactly when they will

see results. In fact, different people will show results at different times. Still, without question, they all know it will work.
An infinite game is compromised by small finite games - so you need to work hard. Let’s not romanticize that only discipline solves everything. Discipline.

```
// TODO
Are you playing a finite or infinite game? How much time do you spend going after short-term gains vs. investing time in long-term outcomes?
```

 # Questions & Answers: Think about the best programmer you’ve ever worked with. What did they do constantly that makes you think that way?
Luciano Sousa (Shopify):

"I think I learned a lot from people that always had a good schedule/agenda in their lives. They were always on time, starting and finishing their workday on the perfect hour every single day. All of these people showed me that I would only win if I could smartly organize my day and avoid procrastination.
In general, I think most of them had a positive impact on my life, encouraging me to always try to build a simple and straightforward day-to-day work pattern. Not too much on my plate, and not so little either."
Berg Brandt (Amazon, Ex-Yahoo):

"One of the things that I value the most in any professional is high integrity. High integrity manifests itself in many forms and best developers that I have worked with have demonstrated high-integrity in some way or the other. The most fundamental one is the focus on doing the right thing for the right reasons.
High-integrity developers are mindful of the outcome and why it is important. While they can sprint, if necessary, they consistently and relentlessly march towards the goal. They are the ones who will proactively call out if something is wrong or call you out if your plan or directions need adjustment. They are the ones who will take on a project that is not so glamorous just because it’s important to get it done. Last, but not least, they are the ones that will be beside you helping to find solutions when things go south."
Caio Gondim (New York Times):

"The best programmers I worked with are reliable. And you can’t have consistency without reliability. They know that it’s not worth it to work long hours through the night because the next day, you won’t be rested enough to think clearly. They know that it’s essential to have fun, eat well.
They don’t give estimations based on perfect conditions. Because they know it’s rare to get a full day without distractions - a full day of just programming.

They know that there are meetings, that things can explode in production, and that they will need to help others navigate the source code."
Blake Williams (GitHub):

"Thanks to my time consulting, I was able to work with a large number of teams in varying contexts over the years. I eventually noticed a pattern, the people I considered the best programmers were all extremely disciplined. The discipline these developers had seemed almost like a superpower early on in my career. They always had the time to write great code, add comprehensive test coverage, and explain the nitty-gritty details through detailed commit messages.
These disciplined developers don't work all through the night and stress about shipping the next feature. They invest their time in doing things well and communicating clearly. In my experience, that discipline helps others on the team feel less of a need to rush and enables them to invest time improving their existing work instead of immediately moving onto the next task. Ultimately these developers save themselves and their team time by slowing down and taking a pragmatic approach to problem-solving."

Habit 5: Do It For Your Future Self

"When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous." — *Martin Fowler*

I don't like to judge other people's code. I'm not just saying this to prevent people from judging my code either. I say this because even the best programmers in the world have written crappy code in the past. And that's not just because they were still learning and didn't know all the best practices. The reason why we all write crappy code from time to time is related to the circumstances around us on that particular week, day, or time.
Imagine trying to write code that is simple to understand, well documented, fast, and modular, while you have a lot of anxiety going on in your life. It's very hard to be efficient as a programmer when you're facing a lot of pressure from your boss and have crazy deadlines to meet. Still, many of us go through those experiences at least once in our working lives. For those of you who are still being treated like this every day, let me tell you something: there are better places to work out there.
Now let's assume that you are not in a toxic work environment. Let's pretend that you're working on a side project, and there's absolutely no external pressure for you to finish this software. You open your code editor and start programming. Ideas begin to flow, and everything seems to *click*. After hours of

coding, you look at all the files you wrote and feel proud of your work. Everything makes sense; it's easy to read, easy to maintain, easy to evolve.
Suddenly, there's a change of priority, and you abandon that idea. You get involved in other projects, you start using different languages, you try new frameworks. A year later, you decide to continue what you started, so you open that project again, and it's unrecognizable!
We've all been there, really. And the root cause is that you're writing code for your current self. Instead, you need to write for your future self. Your current self has all the context needed to understand that block of code, whereas your future self will be involved with other things and probably won't even remember what that function X means.
Don't try to be clever, don't try to code something to make you feel smarter, you don't need to show off all the new tricks you just learned. Just write readable code. Think about maintainability, and use meaningful names for your classes, functions, and variables. Next time you start developing, ask yourself this question: *"Will the future me understand the intention of this code?"*
/* Bonus: I created a list of my personal top 5 books about software architecture and best practices. Go to 14habits.com/bonus/5 to receive it. */

```
// TODO
Open a current project that you're working on. Is there any
refactoring you could do to make the life of your future self
easier?
```

 # Questions & Answers: What are the things you do today to help yourself in the future?
Daniel Buchner (Microsoft):

"Obviously everyone tries to build something that will make their current deliverable a success, but to help ensure the future of your work is more predictable, with better long-term outcomes, a critical skill to develop is the ability to look around corners. There are likely many paths you can take that will lead to success for your present deliverable, but choosing the one that sets up the next version for success, and the five after that, is more challenging.
When thinking about this, I tend to look at larger industry trends, the progression of where I believe technology will be in 3–5 years, and what emerging standards appear solid enough to bet on. You're not always going to find something actionable to incorporate for every project you undertake, but when you do, a small modification to your course early on in a project can significantly change where you end up over time."
Silvio Gustavo (Spotify):

"I try to have empathy with anyone that could touch that code in the future, including my future self. Is this code readable for anyone with any level of experience? With that thought in mind, I aim to:

1. Write simple code with meaningful variable/method/class names. Do not assume that other people (including yourself) will understand short names that don't mean much or some specific terms that can change over time.
2. Have good automated testing. This will give additional documentation about your code and also help anyone that needs to touch that code in the future.
3. Use the version control as documentation. When the project evolves, changes and bug-fixes are made all the time. In the future, when it becomes legacy code, no one will be able to understand the decisions and changes made at that time if they are not documented. The commit/pull request history is a good tool to explain the whys."

Lais Andrade (Google):

"One of the best things to keep the codebase future-proof is to have good standards for best practices, and for these to be followed by the team. This goes beyond styling; it goes all the way to which pre-existing tools to use for certain known problems. Reinventing the wheel is something most of us are easily compelled to do, but a quick search can reduce the amount of new code added, code which would require future maintenance. It also helps avoiding familiar mistakes that have already been solved by good libraries.
The second key point is very famous, but surprisingly difficult to put into practice: early optimization is the root of all evil. It's easy to write a one liner without comment that does exactly what we need, but it's very hard to debug such a line a few weeks, never mind months or years, down the road. Peer reviews are extremely useful to avoid such pitfalls: whenever you need too much context to understand exactly what an incoming line/method/component is doing, you should step back and ask the coders to add more intermediate variables with meaningful names, or to add a few comments explaining why it's done that way, or even to break it down into smaller and simpler components.
The third and last aspect is to always test everything you write. Unit testing, integration testing, performance/benchmark testing, all of it. Enforcing a simple policy of 'no new code without unit tests,' for example, can work wonders. You can definitely notice the difference between codebases that apply it and the ones

that don’t. If tests are taken as a fundamental aspect of the project and are given the right priority, then it’s easy for the team to spare some cycles working on better testing tools (the quality of test code is as important as the quality of the production one) and infrastructure (like continuous integration, etc.). No code should be added without unit tests, no bug fixed without a regression test. Such small steps will have a huge impact on the overall quality of the project."

Habit 6: Your 9-to-5 Is Not Enough

“Don’t expect to be motivated every day to get out there and make things happen. You won’t be. Don’t count on motivation. Count on discipline.” — Jocko Willink

In America, there’s a term called “9-to-5”, which means a traditional 8-hours per day, 40-hours per week, full-time job. If we extend that definition to a “9-to-5 programmer”, this would mean a person who has a tech job, but after their work is done, they leave work there and do not take it home with them.

During my career, I’ve met a lot of 9-to-5 programmers. Some of them were brilliant, focused, and knew exactly what they were doing. Others simply didn’t really like programming that much, this was just a way for them to earn money, and they had other hobbies that felt more important to them.

I don’t think there’s anything wrong with that. Each one of us has the freedom of using our own free time in whatever way we want. However, I truly believe that if you want to have a successful career in tech, you need to invest extra hours into your craft.

One could argue that we don’t expect policemen to go hunting criminals on their own time, or firefighters to put out extra fires. So why should we expect programmers to use their free time to learn something new? The answer lies in the nature of our work.

Our regular jobs are probably not enough to show our full potential. What usually happens is that we are bound to the technology constraints of the company we work at. We have a determined set of languages, frameworks, and code styling we have to follow. Sometimes these technologies change and we have the opportunity to learn the new ones at work, but if you’re just starting your career, you’ll probably need to expand your horizon with other technologies in your free time.

I’m not suggesting that people should lose hours of sleep to study a new technology or spend their entire weekend programming. I’m definitely not saying that you should sacrifice your other responsibilities as a husband/wife or

father/mother in the pursuit of writing better code. After all, at the end of your life, when you look back at everything you did, you don’t want your scoreboard to look like this:

- Software developer: 9/10
- Husband/wife: 2/10
- Father/mother: 4/10

What I’m trying to say is this:

- You can watch Netflix and still be productive.
- You can play your PlayStation and still work on an open source project.
- You can enjoy quality time with your husband/wife and still read a book.

Don’t let people think you can’t relax and get stuff done. You have more than enough time.

```
// TODO
Think about the skills you think could be improved. Can you plan some extra time to develop them? Even 10 minutes a day could make a difference.
```

 # Questions & Answers: How do you spend your free time? How do you balance work with your personal life?

Silvio Gustavo (Spotify):

"I used to spend a lot of my free time diving deeper in software development, especially on Android internals, best development practices and how to better architecture code. It was like a hobby for me.
At a certain point, even though I loved it, I could feel more stressed sometimes when combining that with my 9-5 job workload increase. I decided then to slow down and use my free time to do more of other things like gaming, learning new hobbies and exercising. I have a much better work-life balance nowadays and I’m still being able to learn new things."
Fernando Tadashi (Adobe):

“In the last few years, I have changed my learning habits because of the change I am making in my career. I have always liked the technical and programming area, and it is something that I am glad to know the details of how certain software works and the control you have over it. I still dedicate my time to

knowing which technologies are emerging and are still commonly used, and sometimes I go into this in more depth to learn something that I can use that could help me daily.”
Fabio Costa (GoDaddy, Ex-Facebook):

"Over the weekends, I tend to stay a lot more with my family and friends. This might generally involve a birthday party or just a get together at a friend’s house, or at our own home. During the week, it really depends. We put our kids to sleep very early, so after that I either:

- Stay with my wife, talking about multiple subjects and having something special to drink and/or eat.
- If I’m excited about a pet project, I’ll work on any open issues.
- If I’m excited about work I might work extra hours as well.

I don’t feel like the workload at my current job specifically requires me to work beyond regular hours. The truth is that, like most developers out there, I really love what I do, so when I’m excited about a project I might work extra hours, but that doesn’t affect my quality of life, because I wasn’t forced to do that."

Part Four: Career Habits

Habit 7: Master The Dark Side

"Named must your fear be, before banish it you can." — Yoda

In the beginning, programming was an extremely selective field. Only high-end universities had curriculums that covered software engineering topics. Only privileged families could afford such education for their children. Only hard-core nerds were able to code and get a job. In other words, the barrier to entry was very high.

Computers, the internet, and smartphones are not the future anymore, they are a normal part of our daily lives. From the west to the east, from the youngest to the oldest, everyone has some kind of device nowadays. Software is everywhere and that means programmers are everywhere too. According to a study by Evans Data Corporation (EDC), as of 2019, there are 26 million software developers worldwide. And guess what, this number is only continuing to grow every single minute.

Nowadays, anybody can sign up for a bootcamp and learn how to code in just a couple of weeks or months. You don't need a degree or certificate. There are thousands of books and courses online that can teach you how to become a developer. Implementing code isn't the hardest part anymore. Still, one thing that is extraordinarily rare are developers who know the business side of things.

If you think about it, developers are essentially translators. Our job is to translate user requirements into functionalities. In some companies, those user requirements are communicated to us with a high level of detail. There are even business analysts (BAs) who break down all the user stories and deliver to us an enormous list of acceptance criteria. In a perfect world, we would only sit in our chairs and write code, but that's not always the case. Getting information from a key stakeholder can be very tricky since we all have a different understanding of the system. It's easy to hide behind complex architectures and technical jargon, but breaking silos and communicating with those people can be a very nice differentiator.

For a very long time, I saw clients and salespeople as evil. These pushy, money-driven characters didn't care about the quality of my code and that always made me dislike them. That kind of attitude made me focus only on the code and forget about everything else. For me, programming was the *Light Side* and business was the *Dark Side*. However, over time I started to recognize their

value and began to see that knowing the business side could help me with my day-to-day activities.

1. **Save time:** Even if you spend dozens of hours planning before you start working on a particular feature, you will always uncover new implications and edge-cases that affect your implementation. The better you understand the business, the more capable you will be to solve those problems yourself. When you find that exception to the rule, you are more likely to already know the answer instead of having to schedule a meeting with the domain expert to find a solution.
2. **Prevent complex code:** We always want to build the most abstract, flexible, extendable, and reusable code possible. We want that so much that sometimes we develop an excessively complex application. More often than not, these implementations will never be extended or reused, generating a graveyard of code. There are certain things that will never be extended because that's simply not the focus of the application. When you have domain expertise, you are in a much better position to determine what part of the codebase needs more attention than others.
3. **Better prioritize:** If we have a deadline for next week, the implementation will be very different than if it's due for tomorrow. That's just the reality of work. With better business knowledge it's easier to prioritize the micro-decisions you need to make when coding. You can anticipate which part is more important to spend time on. You can put yourself in the user's shoes and feel their pain. Besides, when you know what the most critical business functionalities are, you'll be more likely to write high-quality code in that area, which will prevent future refactoring.

Every industry is different, so there's no recipe to follow when it comes to better knowing the business side. However, my recommendation is to start with the vocabulary. Pay attention to the specific words and terms used by the business folks. If you emulate the same terminology, the communication with them will be much more effective. Once you start reading and consuming more content about your industry, you will naturally develop more domain knowledge.
Remember, a person who knows how to code is powerful, a person who knows how to code *and* knows how business works is unstoppable.

```
// TODO
Elaborate a list of the most common terms used in your industry.
Start a conversation with your colleagues to understand their
areas. What does the sales funnel look like? What marketing niches
```

are being targeted? What are the most common questions for customer support? How is your product different from those of competitors?

 # Questions & Answers: Do you try to understand the business side of things before programming something?

Michael Lancaster (BlackBerry):

"It depends on the complexity of the work and the time because very often we can get the job done really well by understanding little to none of the business.
That being said, the more we know about the business, the more valuable we are to the team because that will allow us to use both our technical and business knowledge to make a greater contribution."

Caio Gondim (New York Times):

"Let me answer that question by telling you a story.
John works at a bank as a tech lead. His team is responsible for creating a new system used by almost all bank employees.
In a meeting with the users, they asked for a way to export all data that had previously been saved in Excel to the new system. However, the system has no feature to import directly, so it would be necessary to implement a program that reads all the entries in the Excel files and translates them into calls to the new system. John estimated that this would take one month.
Marissa gets allocated to the project. Despite the complexity of the task, she's able to complete it in three weeks (one week less than expected).
John is happy with the result and says: 'Finally, we will be able to import spreadsheet data into the new system. Our users were looking forward to it!'.
Marissa is surprised to know that this is what she worked on, so she says: 'Come on, John. We didn't have to do all that work. We could simply export from Excel to CSV and use the CSV import feature. Our users could have done this by themselves'.
The XY problem is when we ask how to implement our solution, when we should be talking about the problem itself. John had concluded that it was necessary to implement feature X, and asked his team how to implement such a feature. If he had talked about the problem that he was trying to solve, the team would answer that this feature is not necessary."

Luciano Sousa (Shopify):

"I always try to understand the business side of what I'm developing. That's the best way to make sure I'm delivering something useful to the client. It's a tough process, but I'm glad that most companies I've worked with always try to

promote this kind of interaction between R&D and customers. Without these interactions, I'm pretty sure neither the quality of what I'm doing nor the quality of the project overall would be that effective regarding the customer's expectations."

Habit 8: Side Projects

"Failure and invention are inseparable twins. To invent you have to experiment, and if you know in advance that it's going to work, it's not an experiment." — Scott Galloway

When we think about the most successful products today, we immediately think about those unicorn startups with a lot of financial support and an enormous team behind them. But did you know that Twitter, Craigslist, and Slack all started as side projects?

Some people start side projects to create a new product and eventually build a successful company; however, there are many other reasons why working on side projects can be helpful. And you don't need to have an entrepreneurial spirit or give up your full-time job to do it.

If you're just starting your career, side projects can be a tremendous opportunity to grow your portfolio, build a résumé, and showcase your skills. You can read many books on any particular subject, but there is nothing like *getting your hands dirty* and side projects can be that sandbox for you.

If you're already established in your career, side projects can still be very useful. They allow you to experience and learn new skills that your daily work might not offer. Also, there's no pressure, schedules, or specifications imposed by anyone, which means that you can use all your creativity and have fun doing it.

Okay, let's say that you want to create a side project. The next step is to decide what should you build? What ideas are worth pursuing?

This is a list of questions I ask myself before starting a new side project.

1. A side project means you will need to give up personal hours to work on the project, so the most important thing to ask yourself is: *"Do I really enjoy this subject, field, type of work?"*

If the answer is "no", then the chances are you won't have the energy to spend extra hours working on it.

2. It takes time for a side project to gain traction, so the next thing you should ask yourself is: *"Am I willing to spend at least 5 years working on this idea?"*

If the answer is "no", it's very likely that you'll lose motivation and will end up giving up this project before it takes off. However, if your goal is just to experiment, then you don't need to worry about this.

3. Having an idea is one thing, having the ability to execute that idea is completely different, so you should ask yourself: *"Can I execute this idea fully by myself?"*

If the answer is "no", you might consider learning a new skill or inviting a friend to fill the gaps.

4. Saying "yes" to one idea, means saying "no" to a bunch of other ideas, so ask yourself: *"Is this particular idea better than others I had in the past? Is there any other idea that could better use my time?"*

If the answer is "no", then repeat this cycle with the other idea.

5. Understanding who are you building this solution for is vital. If you don't know your audience, it's highly unlikely that you will understand their needs, so ask yourself: *"Do I personally experience this problem or am I solving it for someone else?"*

If the answer is "no", then considering reaching out to people who actually experience the pain that you're trying to solve. You might be surprised by what they have to say.

6. And the last question is: *"Why am I excited about this idea now?"*

It's crucial to understand your underlying motivation. Do you want to learn a new technology? Do you want to make more money? Do you need a distraction from other problems? Be honest with yourself.
More important than choosing your idea is choosing the scope of your project.

More often than not, many side projects won't see the light of the day.
/* Bonus: I listed 7 ideas for getting started on a side project, including real life examples from my own personal projects. Go to 14habits.com/bonus/8 to get it. */
`// TODO`
`For the next few days pay attention to the apps you use the most. Is there anything missing from them? Would you be able to create a better version of that? Have a dedicated notebook for all of your side project ideas and take it everywhere with you. When you're ready, pick one of those ideas and give it a try.`
 # Questions & Answers: Do you have a habit of building side projects? What has been the impact on your career?
Addy Osmani (Google):

"Side projects have always been a great creative outlet for solving small, fun problems in a low-stress way. I love the work I do as part of my day job, but sometimes it can be complex (also cool) and take multiple quarters to ship. Side projects on the other hand are often quick, can be thrown away or at best, might lead to a small open-source project you can share with others. Nowadays, there are whole sites dedicated to helping you get started on side projects, so it's never been a better time to try one.
The impact side projects have had on my career is substantial. They've given me opportunities to create libraries and tools that are now used by others. Through open-source, I've met many friends (including Zeno!), grown my knowledge through reviewing pull requests, and gained a deeper appreciation for engineering best practices that are different from those I typically use at work.
It's also totally cool to work on side projects that others have attempted before. Some of my side projects have just been me working in a private repository trying to re-implement an idea just for my own knowledge. As long as you're having fun, building cool shit and learn something out of the experience, it's worth it."
Michael Lancaster (BlackBerry):

"Building side projects has helped me in a variety of things, such as gaining better knowledge of technologies I was already using in my day-to-day work, gaining familiarity with technologies that otherwise I would not have had exposure to at work, and learning much more about myself and how to be productive.
Besides all that, when working on side projects, we can have a lot of fun and the chance to be very creative, which will force us to explore and work on other

areas that perhaps we wouldn’t have otherwise."
Blake Williams (GitHub):

"I love side projects. They allow me to express my creativity and invest in myself by encouraging me to learn new skills to accomplish the goals I’ve set. Over the years, side projects have pushed me to level up my CSS, learn React, become better at design, and so much more. Those skills have been instrumental in helping my career growth and have opened up doors for me that would have been otherwise closed.
In addition to helping my career, it also gives me a chance to create and give back to the community through open-source contributions. Open-source has led to me meeting and working with some really incredible people. Without side projects, I don’t think I’d be where I am today."

Habit 9: Mario or Sonic?

“Because pain is the universal constant of life, the opportunities to grow from that pain are constant in life. All that is required is that we don’t numb it, that we don’t look away. All that is required is that we engage it and find the value and meaning in it.” — *Mark Manson*

One of the best party games you can play is called *“Who would win in a fight.”* If you’re not familiar with the game, the basic idea is to elaborate on crazy scenarios and to discuss who is more powerful. For example: Who would win in a fight between Batman and Iron Man? Dumbledore and Gandalf? Darth Vader and Thanos? You get the idea.
There are two fictional characters that were very present in my childhood and that I always thought should have an epic battle. In the red corner, imagine Mario, the iconic plumber who can jump super high and always protects the Mushroom Kingdom from the tyrannical Bowser. In the blue corner, imagine Sonic the Hedgehog, who can run at incredible speeds, punch through robots like nothing, and who protects the world from the vile Dr. Eggman. Now, let’s imagine them as software engineers, and rather than them being in a fight, they are just living their lives and progressing through their careers.
Mario is always jumping from one place to the other. If he was a programmer, he would be one of those people who stay at their job for six months and then move to another. Mario is confident and strong, but most of the time he’s really just avoiding dangerous situations. You might think of that one friend who is very smart, but is for some reason afraid of going for that job interview.
On the other hand, Sonic is always willing to face the biggest challenges he can

encounter. If he was a programmer, he would be always looking for complex problems to solve and for the most cutting edge technologies to work with. Sonic is fast, he can learn new things, adapt to new challenges, and doesn't stay behind.
At the beginning of your career, it's ok to switch jobs frequently. You probably want to work in many different places in order to have new experiences. However, there's a side effect to this. The less time you spend on a project, the more superficial you'll be as a professional. In contrast, the more time you spend on a project, the more opportunities you'll have for long-term impact.
Both are amazing and, at the end of the day, they both have the potential to save the world. But if you had to choose between these two, don't be like Mario; be like Sonic.

```
// TODO
Think about the three things you love the most about your current job. Now think about the three things you hate the most about your current job. Is there anything you can do to turn those bad parts into good parts? Are these changes outside of your control or not?
```

 # Questions & Answers: What was the longest job you had? Why did you stay there for such a long time?
Berg Brandt (Amazon, Ex-Yahoo):

"I worked at Yahoo! for almost 10 years in 3 different countries - Brazil, Canada, and United States - in many different functions and departments. My first project was a redesign of the Yahoo! front pages for Latin America and Canada. That work opened up the opportunity to interact with many people in Canada, which eventually generated an offer for me to go work there in late 2007. From there, almost 2 years later, in September of 2009, I ended up coming to Santa Monica where I spent most of my tenure.
In Santa Monica, I had the opportunity to be one of the founders of a project that consisted of a publishing platform for hundreds Yahoo! media and entertainment sites in US and internationally. I learned and grew so much with that project. Given my expertise in the platform, I became a tech lead and later transitioned to manager. After a couple of year on that project, I was offered the opportunity to lead the redesign of Yahoo! Screen (Yahoo's video property) and transitioned to the Video department where I remained until I left in 2016.
Yahoo! has transformed my life profoundly and I am incredibly grateful for that. I stayed so long because there was so much to do, so much to learn, and my work was impactful. Marty Cagan, from the Silicon Valley Product Group, has a great article on 'Missionaries vs. Mercenaries'. I am very much a missionary. I believe you have to stay somewhere for a certain amount of time to make an

impact and to expose yourself to the opportunities that will take your career to the next level. I think of every job in terms of investment and return, and it's not just about money, it's about the return of the investment for your career as a whole."

Netto Farah (Segment):

"I've been at Segment for 3.5 years, and that's the longest I have been employed by a tech company. The main thing keeping me here is how excited I am about the company trajectory, followed by some really good autonomy that I was given (which came with its own set of responsibilities as well).

I don't think there's a magic number though or correct answer for this question. I tend to try to optimize for happiness, and consider myself very privileged to be able to switch jobs easily if I want to. Working 40~50 hours per week is a pretty significant time investment in your life, so it's very important that you're satisfied with your current job, regardless of the pay, or how far along you are on some career ladder."

Manuel De La Peña (Elastic):

"I worked at Liferay for almost 8 years. I joined in the summer of 2011, and left in the spring of 2019. I entered as a Java developer with some experience in the technology this company used, and I left as a software engineer with a totally different profile. I had the chance to work with many stacks and technologies, often not even related to the product. Cloud, virtualization, automation, programming languages, design patterns. I can say that half of my expertise comes from my time there.

However, the technology wasn't the reason why I stayed so long; I stayed there for almost 8 years because of the company values, which I still love and genuinely related to my personal interests. For example, working for your closer circles/communities, and giving back to them so that they can grow with you. Plus, the open source philosophy - contributing back to other OSS projects to help make them better. This sharing culture is important and highly motivating for me as a human being.

And finally, I also stayed there because of the people. They told me in the interview that 'we prefer hiring nice (good behavior) people that can learn the job over hiring smart people who don't behave well' (direct translation from Spanish). And I think this is a fair point. If you hire a rock star who produces a lot but cannot work as part of a team, nobody can touch their code, or things like that, but then what happens when that person leaves? And are the problems this person produces worth it compared to their productivity? The recruiting process

was great because they hired a lot of nice and, at the same time smart people, so I can honestly say that I met some amazing engineers there: very motivated, very skilled, and very nice people."

Part Five: Team Habits

Habit 10: Active Listening

"Wise men speak because they have something to say; Fools because they have to say something." — Plato

Imagine an entire company made up of only software engineers. Imagine all the decisions being made based on metrics. Imagine a place without any subjectiveness, where everything is solved with pure rational, analytical thought. This seems like a perfect environment; however, that's not how companies work, and that's not how human beings work.

A company is made up of a variety of professionals, like designers, project managers, salespersons, executives, and whatnot. Further, these people are of different ages, have different backgrounds, come from different cultures. Decisions are sometimes made with numbers and sometimes made with gut feelings. Problems are created, and solved, with or without you. And even if you're the founder of the company, there will be many things that you simply won't have control or visibility over.

A company is not a logo, it's not a building, it's just an agglomeration of people, and if you want to become effective there, you need to be able to communicate well, even if you're an introvert.

Conversations are typically bi-directional, but that doesn't mean you should speak all the time just for the sake of speaking. One misconception many people have is that we need to **listen to reply**, whereas in reality, we need to **listen to understand**. Understanding someone is far more important than replying to someone. Anyone can reply, few can understand.

Conversations can be informative, entertaining, and sometimes combative. When having a "higher ranking" in a tough conversation, it can be very easy to show a lack of patience, to be aggressive, and to show power and authority. In contrast, when having a "lower ranking" in difficult conversations, it can be very easy to stop paying attention, or to put the blame on someone else. Like it or not, we all have to deal with conflicts and these are the moments where you can either show your best side or your worst side.

Regardless of the conversation that you're having, practice the habit of active listening. Not just listening in order to reply, I mean **truly listening**, like truly trying to understand what point the other person is trying to convey.

If you're a front-end developer, how is your relationship with your team's

designer? What happens when they propose something that seems totally crazy and extremely difficult to implement? Do you get reactive and immediately try to shut down those ideas?
If you're a back-end developer, how is your relationship with your team's project manager? What happens if you miss a deadline and they are pressing to have something done? Do you get defensive and start complaining?
If you're a software architect, how is your relationship with your team's junior developers? What happens if they implement something completely different than what you imagined? Do you have empathy for their problems and challenges?
Remember the words of the great comedian Robin Williams:

"Everyone you meet is fighting a battle you know nothing about. Be kind. Always."

```
// TODO
At your next meeting, choose to practice your listening skills. Instead of being the first to say something, wait until everybody shared their ideas, and be the last one to speak. This will give everybody else the feeling of being heard, plus you have the benefit of listening to everyone's ideas before sharing your own opinions.
```

 # Questions & Answers: How do you communicate with other team members?

Manuel De La Peña (Elastic):

"Since I've been working completely remote for more than 3 years, and partially remote since 2011, I'm convinced that the only efficient manner of communicating in remote jobs is by providing as much context as possible. For a company that is 100% distributed (remote + asynchronous), there are not only engineers but also HR, sales, and consultants working remotely, so being effective when communicating is key.
Also, we are developers, so we are more used to communicating with code. For that reason, a pull request or an issue must provide the context for the reviewer to continue in their own time zone, which might be 9 hours difference from the developer who issued the pull request. Creating templates for issues or pull requests are often something required for me. Those templates must cover the items that can make the sender think carefully about what they are sending.
Finally, it's super important for me to talk about the code, never about the person, because you do not know what is happening on the other side of the screen. Maybe a kid needs too much care for some medical reason, or maybe the

developer simply had a bad day… who knows! So, for this reason, it's vital to be always polite and nice. This is something I try to follow as a mantra. And I feel super lucky for having found this in the majority of the teams I've worked!"
Fernando Tadashi (Adobe):

"I learned that communication is an art and needs to be practiced every day, as well as an opportunity for letting us speak; it is even more important to know how to listen to create empathy. My day-to-day life is filled with meetings, emails, and messages via chat tools, and I use the 'Zero Inbox' and 'Bullet Journal' technique to organize my activities with my contacts."
Berg Brandt (Amazon, Ex-Yahoo):

"When communicating with people as a leader, effective communication is not necessarily an efficient process. Efficiency is about doing more with less, efficacy is about doing it successfully. In order to convey a message successfully, you almost always have to repeat the message a couple of times, inculcate, which is, by definition, inefficient. Context also plays a large role. The less an audience shares a context, the more inefficient effective communication is.
Effective communication plays an immense role in team-building. As a leader, especially when assembling a team from scratch or adding new members, you need to make sure that you are always bridging the context gap in communication. People come from different backgrounds, bring different experiences and skills to the mix and even speak different languages natively (like me: Portuguese). That's why, when someone joins our team, I consider that a new team has formed instead of the existing team plus one. That's why we put a lot of effort on onboard to bridge the context gap and bring the new member up to speed.
All in all, communicating effectively takes a lot of effort, but is an essential skill for a leader (managers and anyone in a leadership position). Effective but inefficient communication is much better than the opposite and it is absolutely worth the effort. Of course, it is the job of a leader to bridge the gap between effectiveness and add efficiency over time."

Habit 11: Don't Underestimate

"It's only going to take 5 minutes." — Every Developer Ever

As developers, we are paid to solve problems. Every task is a completely new problem. Even if you've worked on the exact same task in the past, there are always nuances involved in each task. Maybe there's a different team member

you need to collaborate with, maybe the development process has changed, maybe the software architecture is not the same. Whatever the case may be, there's no such thing as solving the same problem twice.

Besides that, every task is associated with a business challenge. Some business challenges don't affect users as much and they can still use the software without too many problems. While other business challenges are extremely sensitive and they may be affecting users in production right now.

It doesn't matter how experienced you are, it doesn't matter what your job title is, it doesn't matter what company you work for, sooner or later someone will come at you and ask: *"How long is this task going to take?"* Like it or not, stakeholders need deadlines and you'll need to answer them something back.

If it's a trivial task, we'll probably just say it's going to take an hour, but three hours later we'll still be working on that. When we think it's a larger task, we might say it's going to take three hours, but then the whole day is suddenly gone and we're still finishing up some final details. Why does this happen? Why are we still so bad at software estimates?

There are many reasons for this, let's see if any of these sound familiar to you.

1. **We want to impress others:** Each of us wants to be recognized by our work. We want to feel important, we want to feel valuable, we want to feel superior. So when it comes time to give an estimate, we say it's going to take a short amount of time in order to impress others. That way, we hope they will see us differently, they will see us as someone who can finish things faster than others.
2. **We forget that's not all about coding:** When we are assigned a ticket, the first thing we want to do is go straight to the code. That's normal, developing is definitely the most exciting part of the job, but we must remember the entire software development lifecycle. We still need time to build, create unit tests, document the fix, merge conflicts, respond to pull request comments, attend daily meetings, etc.
3. **We don't focus on one thing:** We are often working on multiple projects at the same time, waiting for one person to finish their task, so we can move forward with our part. As a result, we're constantly being pulled into and out of context. There's a real cognitive cost associated with constantly having to refocus on your current task. In other words, it takes time to *"get into the zone."*
4. **We think everyone is the same:** If you're a team lead, it's possible that you might be giving estimates not only for you, but for the rest of your

team. It's common to think in terms of average time it would take to finish a task. However, we must understand that estimates are strictly individual. The time it might take for Jim to finish task X is different from the time it might take Dwight to finish the same task X.

5. **We can't handle the pressure:** Sometimes it's not about ourselves. External clients, project managers, product owners, or even the CTO might be facing an extremely high pressure from others and they might be transferring that same pressure to you. They need to get something done as soon as possible and it's hard to show them that it's not going to be that easy.

In our desire to impress others, in our inability to say "no" to others, in our lack of accurate planning for non-coding related activities, we end up underestimating tasks, missing deadlines, and losing trust. So what's the solution? How can we get better at estimates?

First, we must admit that it's impossible to be 100% accurate with estimates. Unexpected problems will always occur, hidden bugs will always show up, team members will leave the company. We must also understand that we'll never have the full picture in front of us. Business requirements won't be fully documented, acceptance criteria won't be always complete, and new information is going to appear in the middle of the development process.

Now you might be asking: *if there are so many variables to consider, and it's impossible to provide accurate estimates, why even try?* The answer is, because you will get closer and closer to it every time you do it.

In the book *Software Estimation: Demystifying the Black Art* by Steve McConnell, he recommends:

- Separating large tasks into smaller ones improves the accuracy of estimates. Decompose estimates into tasks that will require no more than 2 days of effort.
- Estimate in ranges: worst case, most likely case, best case for a task.
- For first-time development with a new language/tool compare with development using a familiar language/tool, allow for a 20% to 40% increase in effort.
- Document and communicate the assumptions embedded in your estimate.
- The best estimation techniques for small projects tend to be "bottom-up" based on estimates created by people who will actually do the work.

Regardless of the estimation techniques that you are going to use (T-shirt sizing, Story points with Fibonacci scale, or Poker planning), you should treat estimation discussions as problem-solving, not negotiation. **Recognize that both you and the project stakeholders are on the same side of the table.**
It is also important to know that experienced programmers are not necessarily experienced at providing estimates. A developer that's not involved in the estimation process will simply not get good at estimation. Also, if the actual time spent on a task is never measured and compared to the estimate, there is no feedback to learn from.

```
// TODO
Next time someone asks for an estimate, grab a Post-it and take a note of your answer. When you are ready to begin the task, start a clock and see how long it takes to complete. Once the task is done, take a note of how many hours it actually took to finish. Repeat the same thing for the next task.
```

 # Questions & Answers: How do you estimate tasks?
Blake Williams (GitHub):

"I think the best estimation is no estimation. Sometimes saying 'no' isn't an option though. When I can't say no, I do what I assume every other engineer does in that situation, guess. Guessing doesn't mean that you should scream YOLO and shout a random duration of time, but use your past experiences and best judgment to make the most educated guess you can. Once you have that estimate, multiply it by 2. I've seen plenty of people get upset when software was delivered late, but I've never seen anyone get angry at software being delivered early. If you live by 'under-promise and over-deliver' when it comes to software estimates, you'll be in good company."
Caio Gondim (New York Times):

"Imagine you bought a new mechanical keyboard online. The estimated time of arrival is in 4 days. Even though you wanted it for tomorrow, 4 days is ok. Maybe they don't have it in the main warehouse, or the product comes from overseas. So 4 days is not bad.
And then, in 2 days, you come back from work, enter your home and….boom! Your package is there. It was supposed to come in 2 more days. You were mentally prepared to wait longer. It feels so nice to have it before the ETA.
Now imagine this other situation - You buy the same mechanical keyboard, and the ETA is for tomorrow. You tell your friends and make plans to do an unboxing with them.
It's the end of the day, and your package didn't arrive. The online store says that

there was a delay, but it should come tomorrow. And it indeed arrives the next day. You made all these plans; you had your expectations super high; you're happy the package is here, but not as much.
In the end, the delivery time for both situations was the same: 2 days. However, in which one of them the client was happier? I sure think that the client in the former situation is way more satisfied.
We should do the same in estimating our tasks. Overestimate it and deliver before the deadline. That's a better way to manage expectations and prevent yourself from unknowns that might happen along the way."
Luciano Sousa (Shopify):

"Estimating a task is not something I'm happy to do, but as a developer, we are all asked about it all the time. When I have some knowledge about the task in question, I always try to compare it with other similar issues I've tackled in the past. Of course, this is not a 100% reliable process, and I always try to explain to my interlocutor that everything we want to 'estimate' won't always be trustworthy."

Habit 12: Specialist vs. Generalist

"The next Bill Gates will not build an operating system. The next Larry Page or Sergey Brin won't make a search engine. And the next Mark Zuckerberg won't create a social network. If you are copying these guys, you aren't learning from them." — Peter Thiel

When you start your career as a software engineer there's a lot of technical knowledge that you need to absorb in a relatively small period of time. As with any career, you'll likely start at the bottom as an intern or junior position. A couple of months later, as you become more familiar with how things work, you'll understand how to write code, and find the programming language is not a huge problem anymore.
Eventually, you'll transition to a mid-level position. This is the period when you'll feel the most comfortable. You'll start to try new fancy techniques, and it'll all seem to click. Some years later, you might be offered a senior position. At this level, you understand how things work at a deeper level, you know how to build an architecture that will scale, and you are able to guide new developers on the team.
Once you become a senior, a new dilemma arises: What path should you take next?

- Become a specialist, someone aware of all the details in a certain subject.
- Become a generalist, someone able to tackle a variety of different subjects.

This is a very common question that many developers ask themselves. Just like Neo, in the film *The Matrix*, it may seem like you only have two paths to choose. How do you know which path/pill is better? Do you take the red pill? Or the blue pill? Should you go broad? Or should you go deep?
The answer all depends on self-awareness. You need to know yourself, look inside, and consider what types of tasks most motivate you during your work day. Take a look at this list and see which you most identify with.

Specialist – Pros

- You can achieve very good compensation in that specific field
- You can typically gain a lot of technical authority in a particular subject
- You need to keep up to date in only one platform/language
- Companies are always searching for specialist people in that specific field

Specialist – Cons

- It can be hard to find a position if your specialization is too narrow
- If the chosen technology has a risk of becoming obsolete, it will be hard to start from the bottom again
- It’s possible that your knowledge may be too company-dependent, which will make it difficult to apply the same skills somewhere else
- If there’s an urgent need to use another technology, there’s a chance you will take longer to complete a certain task

Specialist – Work Opportunities

- Senior positions, especially at big companies
- Research projects at universities
- Freelancer in a specific field

Now let’s take a look on how these lists compare to the same lists for a generalist.

Generalist – Pros

- You are used to learning new technologies fast
- There's a wide variety of opportunities that you can pursue in different industries
- If you need a new job, you are flexible and easy to transition
- If you need to switch context to a different type of task, there's a chance you will find a solution faster than others

Generalist – Cons

- It's hard to stay updated in a lot of different languages and technologies
- It's possible that you'll take a long time to solve very complex and technology-specific problems
- It's difficult to reach a leadership position if you're changing fields constantly
- Even though it's easier to find a new position, it may be arduous to show that you're the best candidate

Generalist – Work Opportunities

- Startups/Early stage companies
- Consultancy opportunities
- Start your own business

Hopefully these lists will help guide your decision process.
The one thing I have noticed over the years is that leaders tend to be generalists. They can shift course more easily and they're more flexible. A good example is Elon Musk, who has built four companies in four separate fields (software, energy, transportation, and aerospace). That doesn't mean he didn't have to be a specialist in something first. It only means he was able to adapt the knowledge he has to a wide variety of situations.
At the end of the day, these are not permanent states. Sometimes in your career, you'll focus like crazy and go really deep into a certain topic, but at other times, you'll go wider and tackle problems you've never seen before.
My advice is to create a habit of asking yourself: *"How can I better help my team?"*
If that means learning something completely new, so be it. If that means not coding at all, so be it. When individual success is not a priority, you can go much further than you think.

```
// TODO
Examine your work day. Examine your work week. What are the parts that drive you the most? What type of work would you rather do? Examine your company. Examine your team. What are they struggling the most? Is there anything you can help with?
```

 # Questions & Answers: Do you consider yourself a specialist or generalist? Which one is better in your opinion?

Daniel Buchner (Microsoft):

"Most folks have heard this classic proposition before: ‘Should I specialize in a niche and become an expert, or strive to be a generalist who focuses on learning higher-level skills.’ I tend to disagree with this, and personally subscribe to a different approach: Learn how to learn in a way that allows you to mentally touch bottom on a new subject as fast as you can and become functionally capable in the target area as quickly as possible. This doesn’t mean you have to understand everything about a topic the way someone who has worked on it for years does, it means you need to understand the most critical points that drive decisions, and to see them the way a veteran sees them. In terms of applied skills, I try to ensure I can ‘hold my own’ if asked to produce something in the area of technology I’m learning about.

To do this, seek mentors who are willing to let you ask them a lot of questions, and get over the feeling that your questions are naive — hell, some of them probably are, but a good mentor will make you feel comfortable asking questions like that. Frequently ask for reviews, and when you get feedback, focus on soaking up the optimal patterns, not just learning the specific answer to an isolated problem. Do these things with the humble attitude of a student, and you can become the best of both a Generalist and Specialist in the area you are working in."

Silvio Gustavo (Spotify):

"I always wanted to become a mobile developer, so I started learning Android and got a chance to work with it. I studied a lot, followed the best people in the area and got deeply specialized in it. I think it was a good strategy and that made it possible for me to work as an Android Engineer in a big company, something that I always dreamed about. I believe more specialized engineers have more chances in certain companies since they have the deep knowledge to solve problems in some specific domains.

In the end, I believe that being a specialist is a good thing, but even more important is to have the ability to bring all that knowledge and experience when you need to learn and work with new technologies."

Lais Andrade (Google):

"I’m not so sure which one I’d say I am. I’ve done server-side development in the past, then moved on to work on an Android app, and currently I’m working on Android frameworks. The environment changed drastically between those projects, and so did the technologies and programming languages I used, but there was one thing in common, which was my focus point: the design and development of APIs. Even when I worked developing an app, I still focused on the lower level structural parts of the app: creating components, widgets, and libraries to be used by the entire team to develop and ship new features.
I focus on understanding what drives me in software engineering, and then I try to invest myself in that area so I can continue learning and improving my skills. It doesn’t necessarily mean to work specifically with the same technologies or languages. It means finding what makes me feel the best about my work, that interests and motivates me, and then using it to drive me to become what I admire the most about it."

Part Six: Life Habits

Habit 13: Control Your Variables

"The only thing you sometimes have control over is perspective. You don't have control over your situation. But you have a choice about how you view it." — Chris Pine

While I was writing this book, our generation was going through the biggest epidemic it has ever faced. COVID-19, or Coronavirus, is different from anything we have ever experienced in our lives. Thousands of people are dying, everything we see on the news is alarming, and the economy has reached its lowest numbers in many years.

I must admit, I was extremely anxious about that entire situation. Seeing people wearing masks everywhere, seeing the shelves empty on the supermarkets, seeing everybody working from home without any predictability of when things would go back to normal.

It took me a couple of days to realize that there's no reason for me to panic. Yes, there were certain things I could do as an individual to help prevent the virus from being spread even more, like washing my hands more often, avoiding public gatherings, and canceling air travel. However, I can't control the stock market, I can't control politics, and I definitely can't control the virus. The only thing I knew I could control is how I was going to react to this whole situation. And that's when I started to write this book.

Even though this is an unprecedented event in our lives, history is here to teach us lessons of what to do when those times happen. We should not forget that this is not the first time an epidemic hits the world. Let's take Isaac Newton, for example, the great physicist and mathematician who developed the principles of modern physics. He was 23 years old when the Great Plague of London happened. This was a major epidemic of the bubonic plague that occurred in England and killed about 100,000 people.

In 1665, Newton was just another college student at Trinity College, Cambridge. As described in Gale Christianson's *Isaac Newton* book, when the plague threatened Cambridge, Newton fled to his family farm. During that period, he formulated his theory of gravity, a new theory of light, and calculus. Newton returned to Cambridge in 1667 with his theories in hand. Within six months, he was invited to be a member of the academic fellowship. Two years later, he became a professor.

As software engineers, we're used to writing code, creating functions, and managing different variables. These variables can be made of different types, such as integers, strings, booleans, etc. In computing, we have the power to change these variables' values at any time. In life, there are trillions of variables that are constantly being created and changed by others. We have absolutely zero control over these variables, but still, they affect our lives. The way we respond to those changes is what determines our character.
Variables that you can control:

- Your thoughts
- Who your friends are
- What you eat and drink
- How you spend your money
- What you do with your time
- How you treat your body
- How much you appreciate the things you already have

Variables that you **cannot** control:

- The weather
- The economy
- The public health
- How people treat you
- What people think of you
- What people like or dislike
- What happened in the past

Stop wasting time on variables that are out of your control. Focus on the variables that you can change.

```
// TODO
What are the variables that you're worried about now? How many of them are out of our control?
```

 # Questions & Answers: How do you deal with outside events that affect your daily life?

Fabio Costa (GoDaddy, Ex-Facebook):

"I guess it's impossible not to mention the COVID-19 pandemic when answering this question. I felt sorry for all the people losing their jobs, but on the

positive side, I’m really grateful that in tech we are able to work remotely. Getting out of most social media (I’m mostly a light Twitter user these days) also helped me avoid that continuous anxiety that would sometimes make me lose focus on the things that matters. So, in summary, I guess I would say, stay positive and avoid social media.”
Netto Farah (Segment):

"I try to optimize for 2~3 hour blocks of intense and focused work, which is usually worth an entire day of procrastinating. I’ll put my phone away, turn off all notifications, put my headphones on, and try to crank out as much code/experiments/writing/reviews as I can. Afterwards, I try to be more relaxed and indulge myself with some things I enjoy:

- Taking my time to cook and enjoy my lunch
- Going to the gym in the middle of the day
- Going out on a run"

Loiane Groner (Citibank):

"All these things are external factors you cannot control. Sometimes, it can even draw all your energy and will not do you any good. Our time and energy are limited, so I always try to focus on things I can control. Of course, this is easier to say than to put in practice, and the best we can do in this case is trying to educate those closest to us.
“In those days I’m preoccupying myself with external factors, and I’m not feeling ok, I try to do something that brings me joy such as go for a walk in a park, be in contact with nature, binge-watch a TV show so I can clear my head or spend time with my loved ones. Once I’m feeling better, I focus again on what I have to do to achieve my goals.”

Habit 14: Stop Waiting

“We suffer more often in imagination than in reality.” — Seneca

One characteristic that is present in absolutely all human beings is the fact that we’re never satisfied with life. Just look around, from the poorest to the richest, there is always something to be accomplished or something to complain about.
“My job sucks, I need to work somewhere else…”
“This country is hopeless, I need to leave it asap…”

“I have an incredible idea, I need to put into practice…”
And there’s nothing wrong with that. Wishing for a better future for yourself is a basic tenet of being a human. That’s the reason why you’re still reading this book. The problem is what happens with these phrases.
“…but I’m afraid of changing.”
“…but I don’t know another language.”
“…but I don’t have time.”
The fact is, most people have the ability to recognize what bothers them, but only a few people have the courage and determination to face those challenges.
The only thing that prevents you from achieving something is yourself.
Do you want to change jobs? If so, what are the main companies that you would like to work for? How many times have you interviewed in each of them? What kind of skills do you need to learn in order to get that job? What books are you reading to fill that gap?
Do you want to live abroad? If so, how many days of the week are you investing to learn the language of that country? How much money per month are you saving to pay for moving costs? What type of visa do you need to live in that country? What documents do you need?
Do you want to invest in an idea? If so, how many hours are you dedicating to this project? Can you wake up 1 hour earlier? Can you sleep 30 minutes later? What about lunch time, can you make it shorter?
I get it, we all have responsibilities, we all have bills to pay, and we all have people who depend on us. Nothing in life is easy and we all know that very well.
But if for one second you stop being romantic and start being practical, you will notice that behind each excuse there is an alternative. Behind each goal, there is a series of tasks that could be done today.
You just finished reading this book. You can close it and continue with your life, or you can try to adopt these habits starting today. So what are you going to do next? What are you waiting for?

Part Seven: The End

Acknowledgments

“And now the end is near. And so I face the final curtain. My friend, I’ll say it clear. I’ll state my case of which I’m certain. I’ve lived a life that’s full. I’ve travelled each and every highway. And more, much more than this. I did it my way” — Frank Sinatra

I want to start by thanking my grandparents. You may not be here with us anymore, but your attitudes inspire me until today.

Papa, you raised four girls alone, you dedicated your life to your students, you did everything to improve your community. More than that, you’ve always been kind to everyone. You always treated people the same, no matter how rich or how poor, you always left a conversation with a smile on their faces. *Penha,* you lived lightly and happily. You used to host several people in your house and, regardless of who came to that door, you always made a huge party. Your genuineness and your laughter are still in my memory. *Carlos,* you were always different from anyone else. Your ambition was something out of the ordinary; your energy was inexhaustible; your desire to learn was tireless; you saw things that no one understood at that time. I wish you were here to read this. Thank you for serving as a role model for me.

If it weren’t for my parents’ dedication I definitely wouldn’t be here.

Mom, you abdicated your dreams so that I could live mine. You did all that was possible and impossible to give us the best education. You’re the reason I cry at silly movies. Your love has no limits, and I’m very proud to be your son. *Dad,* you showed me the meaning of hard work. There are people who teach with books, you taught me by example. There was no problem that you couldn’t solve, no matter the day or time. The only thing you cared about in life was your family. I hope one day I can be a father as good as you were. *Briza,* my sister, there’s no other way to define you - you are my other half. You had the patience to sit down and teach me everything you learned. In the puzzle of my life, you are a fundamental piece. You helped me, not only with the design of this book but also at every stage of my life. You are always present even if geographically separated. One day our parents will no longer be here, but you can be sure that I will always be here for you.

In the last few years, a person came into my life and turned the world upside down. That person is my wife today.

Carol, you make each of my days an adventure. You may not know it, but you are an example to me. You are always trying to become a better person; always facing your own monsters instead of sweeping them under the carpet. You are a fantastic person, and I'm very grateful that you chose me to share your life. Thank you for putting up with me and my crazy ideas.

Finally, I would like to thank all my colleagues who once gave me the opportunity to work with them. This book is a compilation of what I learned from you.

Thank you for everything.

About The Interviews

This book is a collection of valuable learnings not only from me but from experienced programmers from all over the world. I'd like to dedicate this space to introduce each one of them.

Addy Osmani *(Sunnyvale, United States)* - Addy is an Engineering Manager working on Google Chrome. He leads up a developer tools team focused on measuring user-experiences to help keep the web fast. Some of his team's projects include Lighthouse, PageSpeed Insights and the Chrome User Experience Report.

Berg Brandt *(Santa Monica, United States)* - Berg is a Head of Studios Applications at Amazon. He is a product & technology leader with over 20 years of experience delivering highly-engaging technology products and user experiences. Before Amazon, Berg worked at Yahoo! for more than 8 years.

Blake Williams *(Boston, United States)* - Blake is a Software Engineer at GitHub with a passion for problem solving, pragmatism, and process.

Caio Gondim *(New York, United States)* - Caio is a Senior Software Engineer at The New York Times. He previously worked on Booking.com on Amsterdam, Globo.com on Brazil, and CHRI in India. Now he works on the web platforms team on The New York Times. He has published several npm packages that were downloaded more than 32 million times.

Daniel Buchner *(Redmond, United States)* - Daniel leads open source and standards development for Decentralized Identity at Microsoft. He's passionate about developing apps and services that positively impact people's lives on a global scale. Previous to Microsoft, he ran the Developer Ecosystem product group at Mozilla.

Fabio Costa *(San Francisco, United States)* - Fabio is a Front-end Engineer passionate about building performant UIs. He's currently the Tech Lead at

GoDaddy's Website Builder, which allows users to have a strong online presence by streamlining the creation of a good looking website without any coding or design knowledge. He previously worked at the WebSpeed team at Facebook, making various aspects of facebook.com faster.

Fernando Tadashi *(São Paulo, Brazil)* - Fernando is a Technical Leader at Adobe. He is a multi-disciplinary software engineer who delivers engaging solutions across different areas such as architecture and development of enterprise systems with scalability in mind. He worked as a consultant since 2012, which helped him to be more resilient in crisis situations with customers. His most significant accomplishments were developing high-performance systems for government and private companies.

Lais Andrade *(London, United Kingdom)* - Lais is a Software Engineer at Google. After graduating from the Federal University of Pernambuco, she worked part-time for a couple of years in a small startup company, while also working on her masters on logic and theoretical computer science. After finishing it, she went back to the industry and started working at Liferay, where she had her first chance to collaborate with people from around the globe on many projects. Since then, she moved to London to join Google, where she has been working for the past couple of years.

Loiane Groner *(Tampa, United States)* - Loiane is a Business Analysis Senior Manager at Citibank. She has been working with software development for 14+ years. Loiane has not only authored books for Packt Publishing, but is also a Google Developer Expert, Microsoft MVP, Sencha MVP, Oracle Groundbreaker Ambassador and Java Champion. In her spare time, she publishes articles at her blog and video tutorials online.

Luciano Sousa *(Montreal, Canada)* - Luciano is a Software Developer at Shopify. He is a programmer since 2010 and escaped from Java in 2011 to find simplicity in Ruby. He is a proud Vim user since 2007 and values frugality when choosing his work tools. Passionate about traveling, he lives in Montréal, Canada, since 2018, where he continues to try to learn the "Quebécois".

Manuel de la Peña *(Toledo, Spain)* - Manuel is a Senior Software Engineer at Elastic. He works in Elastic's Observability team, more specifically in the Engineering Productivity team, where he's constantly improving the quality of the processes and products from the automation and testing side. Before joining Elastic, he worked at Liferay, improving development processes, adopting continuous integration, and continuous delivery, which allowed teams to move code from laptops to deployment as fast as possible. Manuel holds a Computer Science Degree and a Master's Degree in Research in Software Engineering and Computer Systems, from Spanish UNED.

Michael Lancaster *(Irvine, United States)* - Michael is a Senior Software Architect at BlackBerry. He has 10 years of experience in the software engineering space working at known PR/marketing companies to unicorn startups to well-established corporations. In his spare time, Michael enjoys building side projects and practicing Brazilian Jiu-Jitsu.
Netto Farah *(San Francisco, United States)* - Netto is a Principal Software Engineer at Segment focused on building tooling that empowers product engineers to build features faster and more reliably. Netto is a believer in solving complex organizational problems with design instead of processes. His current motto for engineering is “Don’t fine cyclists for biking in the middle of traffic. Paint bike lanes instead”.
Silvio Gustavo *(Stockholm, Sweden)* - Silvio is a Senior Software Engineer at Spotify. He started his career a bit after Apple and Google released their first smartphones. Soon he realized he wanted to become a mobile developer. He has deep experience in Android but has also worked in iOS and Cross-Platform projects. He is passionate about well-crafted code, software architecture and quality. Silvio holds a Computer Engineering degree and a Master’s degree in Computer Vision, both from the Federal University of Pernambuco.
Disclaimer: The opinions expressed on this book are their own and not the views of their employers.

About The Author

Zeno Rocha is a Brazilian creator and programmer. He currently lives in Los Angeles, California, where he’s the Chief Product Officer at Liferay Cloud. His lifelong appreciation for building software and sharing knowledge led him to speak in over 110 conferences worldwide. His passion for open source put him on the top 20 most active users on GitHub at age 22. Before moving to the US, Zeno developed multiple applications, mentored startups, and worked at major companies in Latin America, such as Globo and Petrobras.
You can find him online at zenorocha.com.

Bonus

Habit 2: Focus on the Fundamentals
The open source community built a roadmap of fundamentals for DevOps, Front-end, and Back-end developers.
Go to 14habits.com/bonus/2 to grab it.

Habit 5: Do It For Your Future Self
I created a list of my personal top 5 books about software architecture and best practices.
Go to 14habits.com/bonus/5 to receive it.
Habit 8: Side Projects
I listed 7 ideas for getting started on a side project, including real life examples from my own personal projects.
Go to 14habits.com/bonus/8 to get it.

Now What?

If you learned anything from this book, if this book inspired you somehow, please consider sharing it with your friends.
Feel free to send any questions, comments, or feedback via email: **zeno@14habits.com**.
Thank you so much for taking the time to read this!
P.S.: it would make my day if you could leave a review on Amazon (https://amzn.com/B08BF74RRG), it really helps other readers to find this book. You know how these algorithms work :)

Table of Contents