# The Mediocre Programmer

Craig Maloney

# The Mediocre Programmer

CRAIG MALONEY

Published in the United States by
Craig Maloney
http://themediocreprogrammer.com



Cover Artist: David Revoy
Editor: Sharon Maloney

# Introduction

## The Mediocre Programmer?

Let's face it: we don't want to be mediocre programmers. We want to be `[great|amazing|superlative]` programmers! We want to be the programmers they call whenever they're in a bind, and we want to be the programmers that rush into unfamiliar code bases and produce perfect code in a matter of minutes. Code that would sit in the Louvre as a work of art, studied by generations of programmers for its intrinsic beauty and exceptional functionality.

Why would we want to be mediocre programmers? Isn't mediocre the opposite of great? Shouldn't we strive to be great programmers instead?

Sure, we should strive to be great programmers in the long term, but to become great programmers we have to pass through the mediocre programmer stage first.

Mediocre programmers know they're not great programmers (yet). Mediocre programmers see the distance between where they are and the greatness they want in their programming careers. They see the

work that goes into a being great programmers and believe that if they do the work they too will become great programmers.

But they also see their own faults and failings. They see their browser history littered with online-searches for basic syntax and concepts. They see their email archives of questions they've asked other programmers. They wince at their code from several months ago and wonder if they'll ever get to be great programmers with all of these mistakes and missteps. They see the gap between them and great programmers, and it feels like the gap widens every step of the way.

The mediocre programmer wonders if it's even worth it. They wonder if they should do something else with their lives other than computer programming. Maybe they're not as good as they thought they were, or maybe they lack that special talent that great programmers have. Maybe they feel they learned the wrong things early on in their journeys, or maybe they think they should have started sooner.

They see others being wildly successful and wonder if they were absent the day the great programmer genes were handed out.

The truth is we're all mediocre programmers in some way. We all still ask questions and have to look up syntax and concepts in our day-to-day programming. Computers continue to evolve and programmers keep adding complexity to everyday programming tasks. It takes a lot of mental bandwidth to keep all of those concepts fresh in our mind.

## Why this book?

This book is about helping you on your journey as a mediocre programmer. Together we'll uncover some of common misconceptions we have

about programming, failure, and growth. We'll understand that the act of programming and development is something we undertake every day. Every day we can improve in small ways. It's these small changes that transform us from being mediocre programmers into better programmers.

There are plenty of books on how to become a better programmer out there. They tend to have checklists and other advice that the author deems important enough for you to do in order to become a better programmer. They tend to focus on specific improvements like choosing a better editor, writing better test cases, or drinking lots of water. Those books have lots of useful advice, but they read like a laundry list of things that you must do all at once in order to succeed. This book will try not to saddle you with more work (you likely have enough as it is). Rather, we'll discuss what it feels like to be a programmer. We'll talk about the emotions that show up while we're programming; the feelings of frustration, guilt, anger, and inadequacy. We'll cover the struggles in learning new things and keeping your skills current. We'll talk about those times when you feel like giving up and walking away from computing and whether those feelings come from a place of love or a worry that you're not keeping up.

This book is a personal journey for both of us. It's a memoir of my time as a programmer and my feelings along the way. I've thought many times about giving up and finding a different career path but doing anything other than being a computer programmer scares me even more. Does that mean I'm stuck in a perverse Ouroboros of self-pity and self-doubt? Hardly. It means that I need to dig deeper to understand why I chose the path of being a programmer and realize

that it took a lot to get here and it's going to take a lot more to get where I want to be. It's a commitment to seeing things as they are now and moving forward from wherever I'm standing.

## Disclaimer

I am not a professional doctor or a therapist. I'm not qualified to give you medical advice. I'm a programmer. All of the information in this book is given from the perspective of a struggling programmer and should not be taken as medical advice. If you need help from a medical professional please seek out that help (There is an entire chapter about seeking help from others near the end of this book).

Let's begin our journey by figuring out where we are and remembering what lead us to this place.

# Chapter 1

# Journey of The Mediocre Programmer

## How we got here

You have your own unique story of how you got here as a programmer. You might have found out about programming as a curious child who wanted to see what the computer could do. Or you could have arrived as an adult who heard about these things called computers that you could program. Whatever the case, you had a journey to get to this point, and you learned a certain amount to get here. You spent your free time learning how to code, or you were fortunate to be able to work on programming as part of your job. You went to school to learn more about programming or you took training classes. You bought books or read articles online to learn more about programming. Whatever path you took you began your journey as a programmer.

And now you feel stuck.

You look around and wonder if you'll ever know everything that you should know. You read an article on a site and your interest is piqued. An online friend mentions this neat thing that they've found and expects that you will learn more about it. Your colleague found something that might solve an issue you're having on a project and now you have one more thing to learn.

New topics and technologies seem to emerge almost weekly. These "things" creep into our programming discussions and in our work. Perhaps you're finding these new things appearing in job postings requiring a minimum of 3+ years of experience, and you're wondering how could anyone have that level of experience. Perhaps you chose to ignore these things for a while and now they've become a driving factor in your work. It's as if someone flipped a bit and now you're unworthy of being called a programmer unless you were an early adopter of these things.

Each of these experiences causes you to feel as though you are incomplete without learning these new things. They show us that no matter what our current experience is there are still gaps in our knowledge that must be filled. As you look to the horizon you can see the gaps that creep up in between where you are and where you think you should be.

## The Gap

I've chosen the word "gap" to describe the difference between where you are now and where you think you should be for good reason. A gap is something imposed by others, whether by force or by neglect. If a gap appears in your garden fence it means the fence is weaker at

protecting the garden. A gap can also be something that requires our attention. "Mind the gap" is a phrase coined in the late 1960s by the London Underground to warn folks of the space between the platform and the train cars. If people aren't careful around that gap it could lead to an unsafe situation.

The gap in this case is the distance between our current abilities and where we think we should be. Sometimes the gaps are self-imposed because of our desires to improve ourselves, but more often the gap is externally imposed.

One of the biggest creators of gaps in our programming career is change. As programmers we are fully aware of the cycle of change in programming culture. We're constantly having changes thrown at us: changing technology, changing priorities at work, or even changing our strategies to try to keep up with the demands made of us.

Change can also come from within our community. Our community of programmers and users could move on to new positions and new technologies. We may no longer get the support we need to do our jobs and be faced with the prospect that we too need to update our skills or be left behind in an abandoned community.

Change can lead to stress. Stress is prevalent in programmer circles because things are often changing. What worked on Friday afternoon can be broken on Monday morning because of a change to a library that we were using. Our development environment could break because of an upgrade that didn't apply properly. Production might need security fixes, and those fixes mean we have to redo a major piece of our software because it no longer works. There are plenty of ways in which we are kept in a cycle of change.

Not all change is bad. The software that we use could have very good reasons for changing. New features might be introduced that require new ways of thinking about your code. Security fixes may mean that our systems are more resilient to outside attacks, but require different ways of using that system. New and better optimizations may lead to our code running faster but needing a few tweaks to take advantage of those speed benefits. A refactor of an API can lead to cleaner and more concise ways of interacting with another system, but introduce changes that are not backward-compatible with current code.

Change can be positive, but change requires time and effort in order to adapt to it. The gap can only be closed if we have the resources and time to work on it. If we're struggling with our current workload and someone changes how something works we now have to budget our time to adapt to that change. If our mental-muscle-memory knew how something worked and is now faced with a significant change it requires us to re-train that mental-muscle-memory to match how it now works. And if you're already feeling like you don't understand the systems and environments that you're working on then adding additional change can leave you feeling stranded among the newly-formed gaps in your knowledge.

## Closing The Gap

I'd love to tell you that there's a way to close the gap; a way to say you've learned it all and can feel confident that you have mastered the totality of programming.

Sadly, I haven't found a way to close all of the gaps.

You can keep learning everything there is to know about whatever

topic you've chosen to learn. You can take every course available. You can attend every talk and colloquium, read papers about the subject, and even do your own research, and you can still feel like you haven't closed the gap.

So if there's no way to completely close the gap what can you do?

There are three options available to you:

The first option is not to try. Realize there will always be more to learn. Why bother? It's easier to tell yourself you'll never be able to close the gaps in your knowledge. The best option, you tell yourself, is to stick with what you know and ride that out as long as you can.

The second option is to try to do everything at once. You grab every book, blog post, paper, video, or what-have-you to try to learn the topic. Next, you realize that you only have a finite amount of time to learn the topic, and that you can't use all of that material at once. You look over your progress and despair that your learning is not progressing as quickly as you would like. You blame the materials for your lack of progress and look for something else that will help you better learn the topic. Frustration sets in as you blame yourself for not starting earlier to close this gap.

The third option is the more measured approach. You start small with small tasks and work your way toward the goal. Rather than look at the gap as something to be closed, you realize that you can't know the totality of whatever topic you're learning. You enjoy the knowledge you receive in the pursuit of learning the topic. You keep a steady pace toward learning as much as you can. Instead of lamenting that you didn't start sooner you're grateful that you started at all.

Of these three options the first and third are the ones where you'll

find the most contentment. The first option (not trying) allows you to sit with the knowledge you have. But there's a downside to just staying in place. Our industry is constantly changing and technology continues to move. What used to be the norm becomes legacy and what was just around the corner becomes the thing that is in demand.

One of the most useful skills of a programmer is the ability to adapt to new technology. As our technological environment changes, our ability to adapt to those changes allows us to continue on as programmers. Faster machines, different technologies, different devices, different requirements; each of these brings us exciting challenges if we recognize them. But they also take time to learn and create gaps in our knowledge. Relying on our previous knowledge to carry us through these changes isn't going to be enough. We're challenged to adapt to the new surroundings.

The second option (trying to cram and becoming frustrated) is the least optimal path. Trying to learn by grabbing every available resource and jamming it into our brains is a path to frustration, fatigue, and burnout. Many developers try this because they feel the need to adapt to the new environment, but it's difficult to make sweeping changes all at once. It's like trying to evolve wings because you're late for an appointment: you'll be frustrated with your inability to grow wings and still be late for your appointment. The second option also measures your progress by how much further you feel you need to go. It discounts the progress you've made, and creates an endless cycle of running toward a moving finish line.

Of the three options it's the third option that makes the most sense. Taking a measured approach in closing the gaps of our knowledge al-

lows us more joy in our learning process. By breaking down each of the steps on our journey we give ourselves little wins along the way. Instead of expecting a grand transformation we allow ourselves gradual changes and mutations to adapt to our environment.

With this measured and gradual approach we also gain the wisdom that we don't have to close all of the gaps. We can allow ourselves to keep learning in the areas that we need to and gradually build up our skills.

We also can realize that closing the gap is an illusion. As long as we're alive there will always be gaps; new things are created every day. We can choose how far we want to go to become more expert in whatever topic we chose to learn. We can still strive to learn as much as we can, but we do so with a sense of joy in the learning process, not from some perverse need to try to collect the totality of computing knowledge into our heads.

## The Journey is the Reward

The secret to moving forward in our journey as programmers and developers is realizing that each step we take along that path is valuable and worthy of our attention. Just because we haven't learned the latest technology in a fortnight doesn't mean we should give up trying. Nor does learning a technology only to watch it become overshadowed by something else mean that our learning time was wasted. Every challenge we face, every technology we learn, and every hour we spend coding is another step on our journey to becoming better programmers. Each mistake and wrong turn introduces us to opportunities to learn from those mistakes and grow as programmers and developers. There

is no perfect path toward becoming better at this. Even if there were I'm sure we could point to any point in our past and say that it was less than perfect. Carving the perfect path from where we are to where we wish to be is not possible. Worse, it is an illusion that keeps us from moving forward in our daily practice of programming and developing. It may seem trite to say "the journey is the reward" but every day we work as a programmer and developer is one day closer to shoring up those gaps in our knowledge and becoming more content with how our skills are growing.

# Chapter 2

# Traveling Companions

## Famous programmers

Throughout the history of computing there have been folks who have demonstrated amazing coding abilities. They exist in the pantheon of great computer programmers: Ada Lovelace (the first computer programmer), Dennis Ritchie (creator of the C programming language), Rear Admiral Grace Murray Hopper (creator of COBOL and credited with finding the first documented computer "bug"), and so on. We also have developers in our own communities that have a certain celebrity status, whether they're the folks who wrote the language we currently use, the folks who maintain the operating system we use, or someone who rose to prominence in our chosen community. It can be intimidating when we compare ourselves with these luminaries. After all, whatever we're currently working on might not measure up to whatever they have done. Worse, we may be working on something similar and feel like whatever we're working on will never measure up to what

these folks have accomplished. We may even be friends with programmers who seem to figure out things much quicker and cleaner than we can and marvel at how they seem to have this body of knowledge at their fingertips that we can't possibly understand.

## Backstage vs. performance

One of the best pieces of advice I have received about comparing ourselves to others is realizing that the comparison is between our backstage versus their performance. The metaphor draws from the theater, where performers know every thing that isn't right about their own theater-group's performance and unfairly compare it with another theater-group's finished performance. This metaphor is useful to us because we tend to compare all of the things that we know (the long hours of unproductive coding, the struggles with learning, and so forth) with the finished product of someone else. We don't see their struggle in getting things to work, or their countless hours making crappy prototypes and unfinished projects before making the thing we admire. Allow yourself to have a messy back-stage and do plenty of rehearsals and understand that it takes effort and practice to put on a good performance.

## The lure of the post-mortem

There's a tradition in some programming projects (especially in game development projects where there is a clear end to the project when the product ships) of doing a post-mortem on the project. What the post-mortem does is allow the developers of a project to state what

went right and what didn't go right. The better ones tend to be frank accounts of the successes and failures of a project.

The post-mortem can be a fascinating look into the development of a project. I've found myself reading a lot of these looking for insights into the development process. But there's a subtle trap in the post-mortem: they're a recollection of events from the vantage point of a successful (or unsuccessful) project. They're a recollection of someone who worked on a project that was successful enough for you to spend time reading about that project's ups and downs. They're written from a perspective where the success of the project is a foregone conclusion (or they're written about projects that were noteworthy in how they failed or didn't meet the expectations of those involved). It can lead to the belief that what you're working on is not as important as the things that other people are working on. But we don't know the importance of our project in real-time. Even the folks in the post-mortem didn't know if their project would work or be successful as they were working on it. Our projects may never see the light of day, or they might be something that changes the world. We can't know the value of what we're working on while we're working on it (though we can have a sense of whether or not we *feel* our work is important).

A post-mortem also has the benefit of hindsight. Decisions that were clear and definite at the time might not make much sense when viewed with data obtained later in the project's lifespan. There's also an issue with "selective memory" where something might not be remembered with the same clarity, or may be conflated with other events. Confident statements like "We knew this one thing wouldn't have worked" might actually have been "We weren't sure

if this would work so we tried several things. They all didn't work.". Consider anyone writing about their past as an unreliable narrator. True, they may be the best and most knowledgeable narrator we have, but they do not have an objective perspective on whatever they were creating. They have their own biases and reasons for the stories they present in a post-mortem. Treat a post-mortem like you would treat an auto-biography of a famous person: a primary source with an agenda to show the subject in the best way possible.

There's nothing wrong with reading a post-mortem about a project — we can learn a great deal about how a project is run (or shouldn't be run) and what pitfalls to be aware of if we go down a similar path, but understand that you're reading one account (whether by one person or one team of people). They have the perspective of someone deep in the conflict. You're looking at their recollections of tactics, not the overall strategy that brought that them to the place.

## Ranking programmers

There are many metrics which folks use to rank programmers. You've likely seen these metrics manifest themselves in different ways: competition sites, numbers of commits to projects, productivity measurements, time to turn-around code, and good ol' fashioned gut feelings. We do it to ourselves and others. We compare our work against the work of our peers and folks that we admire, but that can lead us to make comparisons that aren't objective or based on all of the data. I can compare myself against folks who do low-level programming and find that I don't measure up in that realm. Never mind that I haven't done a whole lot of low-level programming; the comparison is valid. Or, I

can compare myself against folks who were mentored by programmers whose names are legendary in the field. I will find gaps between my knowledge and their knowledge, because I didn't have access to those mentors (or worse: I didn't take advantage of the mentors I could have accessed. Whoops!). Comparisons like these are not helpful and lead us into punishing ourselves for not being someone else. Our assessment of our projects and history give us the conclusion that we're not that other person, nor could we ever be that other person.

The major problem with ranking programmers (or ranking anything for that matter) is that ranking systems are based on one set of criteria. There is no real standard for ranking programmers. Sites that rank programmers based on numbers of problems solved or difficulty of problems solved have only determined that there are a set of programmers who really enjoy solving these types of problems. They've also collected a set of programmers who will spend the time and effort to solve these problems and will be competitive while solving them. It tells us little about the programmer's abilities outside of that domain.

There are also other metrics to rank programmers. One classic metric is reviewing how many lines of code a programmer used in order to solve a problem (this is sometimes referred to as "code golf", where the fewer number of lines of code the better the solution). We can argue how "clean" the solution is (clean being another nebulous term). We can determine the "Big O notation", a notation used to describe the performance or complexity of the algorithms that a programmer used in their code. We can stress test the code to determine how well the code adapts to various circumstances. We can count the number of cycles a particular piece of code takes in order to run and benchmark it against

similar code. Very little of this tells us anything about a particular programmer. What it does tell us is that the programmer has experience that lead them to that particular solution. It tells us that the programmer has seen these sorts of problems before and cared deeply enough about the problem to think hard enough about how to make a better solution. We learn that the programmer devoted time and energy to practice these sorts of problems. What it doesn't show us is an overall measurement of the programmer's skills or abilities. It's similar to the apocryphal tale of a brilliant professor. This professor was an absolute genius in his field and was one of the go-to people for answers about his subject, but despite his brilliance he was unable to understand how to change a tire on a car. Does that mean the professor was not as brilliant as folks claimed him to be? Hardly. It means the professor spent more time thinking about his profession than he spent thinking about changing tires. The same is true for programmers. If a programmer spends most of their time solving a particular set of problems they will eventually become skilled at those sorts of problems. But if that programmer struggles with a different sort of problem it doesn't discount their overall skills; it just points out areas they might want to work on.

## Measuring programmer output

There's also a tendency to measure programmer productivity by how many contributions the programmer can make to a project. Under certain version control systems these are called "commits". They list out a set of changes that the programmer wishes to make to the code. In an era of social coding sites like Github and Gitlab we can easily review what other programmers are committing. Since we can measure the

number of commits, we can use this measurement to feel that we're not generating the same number and frequency of commits as other programmers. And unlike measurements of old (lines of code in particular, which measures how many lines of code a programmer adds to a program) we can review the quality of their commits to a project. It can be daunting to see a lot of quality work done by our peers. It can also be source of frustration and feelings of inadequacy. "Why can't I be as productive or contribute as this other person?" we ask ourselves.

Even more frustrating is when others use these metrics to judge productivity and code contributions. We may find ourselves being criticized for our output (or lack thereof).

Commits and lines of code are the most visible measurement of coding productivity, but they don't show much about the actual practice of programming. We can't measure the amount of time thinking about the problem just by looking at a commit. We don't see the mounds of reference material the programmer used in order to figure out a solution, and we certainly don't know if this commit is the result of one afternoon of work or many days of work (unless they commit more often). We might even find out that this person is acting as the focal point of an organization and is folding the work of multiple folks into their commits.

Measuring ourselves on the quantity of others output is easy and seductive but it isn't useful for figuring out how to improve ourselves in relation to the other programmer (other than "generate more commits"). That way of thinking can lead us to believing that we're not spending enough time doing "actual programming" and lead to overwork, stress, and burn-out.

## Traveling Companions

There are times when it is useful to compare ourselves with other programmers. Sometimes we can learn about new technologies or new methodologies by looking at the work of others, but it's easy to fall into the trap of thinking that because we're not at the level of other programmers we're somehow inferior to them. Rather than looking at other programmers as competition we should look at them as companions. We're all in this profession working to make our respective projects better. With Free and Open Source Software we have a unique opportunity to see how other folks do their work in public. We can learn from the code of others much in the same way that scientists can look at the papers of other scientists to see what worked (and can improve the validity of the paper with replication and repetition). No coder is completely isolated from the work of others. (It's a rare programmer that has coded their entire programming environment from scratch without the work of others). We all learn from each other, but rather than be intimidated by the work of others we can instead take it apart and learn from it. If we're lucky we can take the opportunity to ask them how the code works and why they chose to write the code in that way.

There's value in asking questions of our fellow programmers. We tend to overlook asking questions for fear that we're going to ask something obvious or ask a question that will make us feel inadequate for asking. Asking questions is very useful when we don't understand what is going on with an idea or a particular piece of code. There are programmers out there who don't mind answering questions, and my hope is that you find them. Granted there are some programmers who are very busy and might not have the time or inclination to answer ques-

tions, but if we are truly stuck and have exhausted all other avenues perhaps we can ask questions of them that don't require much of their time and effort. They may even be grateful for the question because it gives them insights into a perspective they might not otherwise have. When we ask questions we initiate a sharing of ideas in both directions.

There is an art to asking questions and it can be frustrating when folks don't answer our questions or come back with other questions and suggestions that are less than helpful. This manifests itself in exchanges where person A asks: "I'd like to know how to do X" and persons B and C respond "I would do Y instead". It's frustrating when folks won't answer our questions directly. It's also easy to get embroiled in exchanges with folks about the merits of doing Y where Y was suggested by someone else that has nothing to do with the original question about X. But if we re-frame the experience as "this person is trying to help me; perhaps there is something in this recommendation that might be helpful" then we can have a better conversation. Perhaps what we're asking isn't the best way to do something and pausing to listen may help us better understand why they suggested what they did.

Pulling our egos out of the question allows us to be more open to the answers we receive. When people don't understand our question it becomes easy to take it personally and frame it as "they're not understanding me" or "they're not listening to me". Pulling ourselves out of the question allows us to accept the answer provided "as-is" and gives us the ability to change the question as needed to get better answers.

Of course there are folks who won't respond with your best interests at heart and are only interested in imposing their own world-view upon you. Instead of answering your question they question why you're do-

ing that at all and suggest that you should be using their methodology instead. It can take a lot of energy to engage with these folks to tell them "no, I really, really intended to learn more about X." I wish I had good answers for how to handle these folks. There are plenty of them that feel that whatever they're doing is the only right path and those that stray from their chosen path are anathema to their world. My best suggestion is to thank them for their time and ask someone else for help. Perhaps they may be useful in the future when you have questions about whatever is part of their agenda, but for now be as kind as possible and wish them well on their programming journey. Technology spaces have a lot of folks who have been working with computers for a long time and have formed strong opinions about their tools and technologies. My hope is that you can find the ones that are also kind and willing to share what they know and not badger you with their strongly held beliefs. Over time you too will form your own beliefs on what works and what doesn't work and pass that knowledge on to others. Recognizing folks who are there to help educate and those who are there to proselytize is part of our growth process.

If we look at other programmers as our traveling companions on this journey; as peers in our coding practice, then we can realize that we're all in this together. Even someone with many more years of experience than you have is your peer. You have knowledge and experience they won't have, and they have experience and knowledge you don't have. If we strip away the barriers of perceived rank and meritocracy we can better engage with and learn from each other.

The journey to becoming a better programmer is long and hard. We need the best companions we can find to help us along the way. We

need more than just the technically-skilled companions; we also need companions we can talk to when the day is done. We need companions we can sit with around the proverbial campfire where we can laugh and commiserate about our struggles together.

# Chapter 3

# The mistakes along the way

## Whoops!

It's bound to happen: something you thought was a good idea didn't work the way you planned, and now you realize you've made a terrible error. Sometimes it's something that could have been easily avoided (committing code that was meant for debugging, for instance). Sometimes it's a cascade of errors, each building on the efforts of the previous error. There's the mistake of neglecting the side-effects of a module when it's used in a way that wasn't intended, or it's the realization that you've designed a small, tightly coupled module only to learn that your module will be part of a larger piece of software and your code isn't designed to make a smooth transition. Whoops!

The mistakes that really frighten me though are the ones that I did not expect; the ones where the unintended consequences run rampant throughout the system like a chain-reaction. Those mistakes keep me up at night.

Programmers make mistakes. The nature of our jobs require us to be aware of what is going on in multiple sections of code. We lose track of the state of our program and committed code. We try to pepper our code with comments and reminders of what's going on in a section of code but comments become stale and add to our distraction. We rush and rely on muscle memory to pick up the slack. We deny ourselves areas where we can adequately test code because we feel we need to rush to get things done quickly.

We panic, and when we panic we make mistakes.

## Avoiding mistakes

Let's be clear: there's no way to avoid or eliminate mistakes. Software is too complex to be completely bug-free. What we can do, though, is create places where we can tease out as many bugs from the code as possible before we set it in front of others. We can better understand our code and what it's doing when we have the ability to debug and test our code in a safe environment. We can see how it will behave under certain circumstances. Creating a model of the target system allows us to test our code against miniaturized versions of the target system's reality and see how it behaves under those conditions.

We put a lot of emphasis on avoiding mistakes, both in programming and in programming culture. There are horror stories of how small bugs in a program caused enormous pain for those involved. The moral of these stories is that simple mistakes can be costly and we need to be doubly careful about avoiding mistakes. These anecdotes are told in the hopes that they'll somehow scare developers into being more cautious, but they can have the opposite effect. They can make programmers

paranoid about making any mistakes at all, and when we operate in a fear-based mode we begin to panic. Telling programmers to not make any mistakes is similar to telling someone to not be afraid: they become more afraid of being afraid.

The best (and perhaps only) way we learn is by making mistakes. Learning by making mistakes is an effective way to allow us to be curious and see what caused the program to fail. When we deprive ourselves of the freedom to make mistakes we deprive ourselves of the learning opportunities in making those mistakes. That doesn't mean we have to make every mistake that other developers have made before us (that would be a lot of mistakes). Nor does it mean that we need to introduce chaos into our development process in order to learn better. What it means is that we need to make our own mistakes in our own way in order to keep learning and figuring out where the gaps in our understanding exist.

## Making a model

We need environments where programmers can safely learn from their mistakes. We need spaces where programmers can feel good and confident about trying new things. We need places where developers can try out their ideas and not have those changes ripple out to other unrelated systems. This is the best way that developers can learn and be brave in their learning process.

These environments must model the target systems, and they must be as close as is practical to those target systems. That doesn't mean you need to make exact copies of expensive production environments, but you do need to create models of production environments that

test most of the pieces with which your code will come in contact. Having models that mirror production systems means that when you move your code to production you'll introduce fewer changes with unintended consequences. Your changes will have already existed in a production-like environment. You can take comfort in knowing that the changes you enact in these models will be the same changes that will appear on the target system.

Ideally you'll need to have an environment like this on a machine that you control. This means that you're not competing with other programmers in your organization who are also being brave with their changes. You'll also want to ensure that your environment is kept up-to-date with their changes (and any production changes) so your development model matches what's on the target system and what will be on the target system. A good model is one that is kept current with what it is modeling. It's the same as a map of a city: it's best when it matches the area its modeling and is kept current with changes that occur in that city. A good map of the city might tell you about the recent construction happening on your route. A useless map doesn't even show your route because it wasn't built when the map was created. If our model of production is constantly falling behind what's in production we will spend more time rectifying the changes that we're making with the changes between our model and production.

This also means having an environment that you can rebuild quickly and replicate as needed. Having a model that becomes its own separate reality becomes one more system to maintain. This model should be something that you can delete and rebuild at will in order to remove any previous experiments. It's best to think of it as an ephemeral copy

of your target environment that has limited use and can be tossed when no longer necessary. It should be quick to replicate this environment so there's little friction in creating new environments to play in. That can mean scripting the building process for these environments. How you decide to do this is up to you but keep in mind that you want something that's as simple as you can make it and requires as little thought as you can manage to replicate it.

Again, it doesn't have to be perfect - it's only a model, but it does need to be close enough so your code will behave in a similar fashion between the model and the target environment.

## Time machines

There are plenty of folks who will tell you the benefits of revision control (and many folks who will show you the exact steps for how to set up a revision control system). Revision control systems such as `git`, `svn`, `cvs`, and the like have helped programmers coordinate releases and keep a log of what work was added to their project. Having a good revision control system allows you to create areas where you can test code without having to merge these tests into production code. Good revision control lets you to create a space (or "branch" in `git` parlance) based on existing code that you can use to experiment and develop. It also allows you to commit in that space and diverge as much as you want or need to in order to fully explore the changes you're making. What's most important though is that good revision control will also allow you to abandon that space if you need to – you're not forced to add those changes back to your production code. This allows you to see if something might work and abandon those changes if they don't.

Good revision control affords programmers the ability to branch off from any point in time and explore what happened in the code base. In a sense they're time machines and infinite universes, allowing you to play "what if?" scenarios with your code and move back and forward through time in your code. This is vital for your learning because you can feel secure in testing and trying things and rewinding those changes (or deleting them entirely) without affecting the work of others.

Learning how your revision control system works will give you freedom to make mistakes. Many of these systems can seem complex at first, but with continued practice and patience you'll understand what the revision control system is doing and what its capabilities are. You'll be able to judge how many risks you can take with your code and be more confident with the risks you take.

Revision control can also play a role in seeing the development of the code of other folks. You can get a window into their development process and see what certain features look like as they are added. This can help you learn about an unfamiliar code base and show you the direction they took in order to make the code the way that it is. It can give you a window into the history of a project and what went into making it happen. Revision control can be a time machine into the history of a project and can help you understand that programming is a process. Not all projects come fully-formed from programmer minds.

## Learning from failure

Sometimes we fail. Sometimes the code we write isn't up to the realities of the system it's implemented on. We push code that does something unexpected and systems break as a result. We can lose track of where

we are in our code and make changes that conflict with other changes which then causes us to spend time undoing those changes. All of these cases cause discomfort, whether to us, the folks we support, or the folks we work with.

I'm not going to lie: failure sucks. It makes us feel like we're less of a person because we failed. We feel inadequate and wonder how others think of us. Do they think less of us? Have we damaged our relationship with those who use whatever we've programmed? Have we let our team down? All of these questions stem from two desires: the desire to do our best and the desire to do no harm to others. We want others to think well of us and our skills. Failure runs counter to those desires and amplifies whatever feelings of inadequacy we might have. Those feelings can include wondering if we should be programming at all or wondering if our talents should be used elsewhere. We wonder if we should just give up.

We don't usually think of failure as part of the learning process. Failure is often seen as the end-point of the journey. In school a failing grade is viewed as a condemnation. We don't view it as "I need to practice this some more"; instead we feel that we have caused shame and discomfort to ourselves and our loved ones. We do ourselves a grave disservice if we don't realize that failure is a natural part of the learning process and that it's OK to fail. Not everything we do will be perfect. Mistakes will creep into the best code we write. We will slip up and deploy to the wrong system. Our mistakes will cause discomfort to others. Accepting this gives us the freedom to realize that despite our best efforts we will not be perfect. Instead of viewing failure as a limitation we can use it as part of our growth process.

When we realize we are going to make mistakes we can change our approach in how and where we make them. I mentioned before about creating models of our environments. What better way to allow us to make mistakes than in an environment where those mistakes can be contained and rolled back? Creating models allows us to practice and test our assumptions in environments that nobody else has to see. It's akin to a practice space for musicians where they can run through their material without the need to perform it right the first time. They can work out the troublesome parts and make mistakes until they are confident in their performance.

Mistakes are how we learn what works and what doesn't work. They are an integral part of our learning process. We tend to remember the lessons of what didn't work better than the ones that did work. Mistakes help us shore up where we lack knowledge and help us understand the gaps we've yet to close.

Mistakes also act as a reminder to pause for a moment and not get too wound up in the urgency of things. My own mistakes tend to crop up when I'm rushing to meet a deadline (whether real or self-imposed). My worst mistakes happen when I'm tired and rushed, when I'm practically flailing at the keyboard trying to get something (anything!) working. When I allow myself to pause for a moment, reflect on what I'm trying to do, and feel the uncertainty in the moment I can take steps to recalibrate and refocus in the moment. I give myself the freedom to course-correct and understand that I'm not doing my best and need to do something different. It might be something small like giving my brain a bit of rest or something large like revisiting the assumptions I made about what I'm doing. Taking the pause lets me determine if I

want to continue doing what I'm doing and understand if that's the best path.

## Journaling our mistakes

There's value in not making the same mistakes twice, but when we do repeat the same mistake can still be useful. Knowing that we've repeated the same failure is useful because it gives us a pattern we can understand. Those patterns show us that doing this particular thing leads to a repeatable failing result. We can then determine what caused the mistake and plan for how to mitigate it. This is part of the learning process, as long as we don't fall into a spiral of self-recrimination when we realize that we've made the same mistake again.

One trick that I should use more often is journaling. Keeping a journal of what happened and how we fixed it is one way to explain to someone else (often ourselves) about what happened. Explaining what happened allows us to become a teacher to ourselves and others. It reinforces our learning process. Writing down what happened in a way that others can understand allows us to arrange the thoughts in our head in a way that is clear and understandable. When we articulate our own thoughts about what happened and codify them, we start to understand our own thoughts and can shake loose other ideas about how to fix this and other problems. We give ourselves the pause we need to fully understand what happened and how best to move forward. We become our own sounding-board for ideas on how best to proceed.

This isn't about keeping a record for posterity so we can look back at a list of failures and beat ourselves up about the past (if you're anything like me that happens automatically). It's a way to teach ourselves and

maximize the learning process. It's about giving ourselves the freedom to be the instructor to our future selves so we can be more aware when a mistake is about to happen and understand how to correct for it. This allows us to focus on the moment just long enough to understand what happened, what we did to correct it, and how we can best proceed from here. It also helps us to locate where our gaps are and the "next actions" that we'll need to take in order to fill in those gaps.

We'll talk more about journaling in later chapters but I fully recommend a journal habit if for no other reason than it gives you a willing apprentice to teach, even if that apprentice is only yourself.

# Chapter 4

# The inns we stayed at

## Fellow travelers

Whenever we think of programmers we tend to think of someone sitting in front of a computer entering code; the glow of the monitor reflecting off of their face. Usually the programmer is alone (though there are methodologies that utilize more than one programmer at a time, "pair-programming" for instance). During those coding sessions there isn't a lot of contact with other programmers and it can feel isolating being in the company of yourself. Granted this can be a good feeling (there are times when I really enjoy being alone at the computer, fully engaged and focused), but there are other times when we need to feel like we're not alone. This is especially true when we're learning and pushing ourselves into uncomfortable territory. Finding others in similar situations can help us with our learning process. Others can help us by fielding our questions and reviewing our progress. Finding a good community that is supportive in our learning is essential on our programming jour-

ney. When we have a good community we have a place where we can learn and help others learn. We can grow in the community and find support.

A good community is one that strengthens us and those around us. It nurtures us and provides us shelter. It is a safe place where we don't have to keep our guard up from attacks on ourselves and others. It holds people accountable to each other. We can trust the members of the community and feel that trust is reciprocated. Good communities exist without competition and ego, where members can express themselves openly and accept others as they are.

## Finding a good community

There are a lot of good communities out there that are willing to help you become a better programmer, but how do you find them?

That's a tricky question.

Most programming languages have some form of community centered around them. Some have mailing lists or other communication channels that you can join and participate in. Unfortunately, most popular languages have spaces that are difficult to follow, especially when you're trying to learn. I know I have joined the main channel for a popular language only to be overrun with multiple conversations happening at once. Mailing lists designed to support beginners can have major amounts of traffic, and that traffic can be overwhelming when you're trying to understand the basics of the language while trying to keep up with the deluge of mail in your inbox. Reading the archives of the mailing list or chat can help determine if you're ready for that level of traffic and if the conversations on the list are the types of conver-

sations you enjoy. Remember: this is to help you along your journey. Throwing yourself into a crowded room only to be inundated by the amount of conversations and cacophony will only make you feel more isolated and unwelcome.

Some programming languages have local user groups. Those can seem intimidating at first, especially if the group has been around for a long while. I know I was intimidated before I went to my first user group for fear of what I might find inside. What I found was a group of folks who were interested in the topics that I was interested in. I've made lasting friendships through users groups and I encourage you to see if they might work for you.

If you're at a loss for finding the right group (perhaps you're in an area where you feel you're the only person who shares your interests) you might consider starting your own or branch off of an existing group. My friend Rick and I started a local branch of a group called Coffee House Coders where coders meet once a week for a few hours to sit and code. All we did was post the times and places that we were meeting and told folks to just show up with a laptop to code. We've met some amazing folks along the way, and we've kept the group going for many years. Starting a group is an act of courage. There have been many evenings where I've sat alone in a coffee shop waiting to see if others would show up. That's fine. People get busy and interests fade over time. What is important is creating the space for ourselves and others to feel welcome. For us that meant finding a local coffee shop that was open late at night and had ample space for setting down a laptop. It also helps to find a place that has electrical power so folks can charge their batteries if necessary.

There are many ways to be creative with starting a community. The advent of online tools allows you to build communities with folks across the globe. Bringing these folks together to talk and discuss ideas and offer help is amazing when it happens. Sometimes it can be as simple as creating a chat room around a common interest. Explore what's out there, and if it doesn't meet your needs feel free to create your own.

## The difficulty in finding a good community

I recognize that not everyone can join or build a community. Online spaces have a reputation for not being welcoming places for folks, and in-person group meetings can use up whatever mental resources you have. It took me a long time to find the courage to go to my first in-person meeting as I'd had a bad experience with someone I worked with that I thought would be at these meetings. (I'm not sure if that person ever went to those meetings). But I'm grateful that I did eventually attend my first meetings. Attending these meetings led me to friendships, opportunities, and other "traveling companions" for my journey. It led me to switch to one of my favorite programming languages (Python) and led me to several jobs. It also helped me to feel like I wasn't alone with my interests and introduced me to others I could rely on. It gave me a feeling of belonging.

Getting over the initial hurdle is hard. Our fear of rejection and our fear of making ourselves vulnerable to strangers can wear us down. Overcoming that fear takes a lot of our mental energy and can sap us of the desire to be part of yet-another-community. I can't say that it will be easy, but I can point to some of the benefits it had in my life. I hope you can find those benefits as well.

An alternative to in-person communities are online communities. Online communities can be a great way to find others. They gather folks from many different locations and bring them to a common area. Part of the reason I made my jump into meeting folks in person was because of the good experiences I had with these folks on IRC (Internet Relay Chat). I enjoyed the company of these folks through our online interactions and felt comfortable meeting them in person.

The low barrier to entry for many online communities can allow us to see what the community is about. What are their priorities? Are they kind to folks who are starting out? Do they have a pattern of helping folks like us or do they tend to hurt folks like us? Do they have members who nurture their fellow members or are they cutting each other down?

I'm not aware of a good strategy for determining if a community is helpful or hurtful. It takes some effort to follow a community and get a sense of who they are. It is emotionally draining to put ourselves in situations where we are vulnerable in order to see if others will be gentle with us. Communities are made up of people and people are fickle and irrational creatures. What might be an amazing community for one person may be a toxic environment for another. While I don't have a strategy I do have some ideas on key elements that make up a community.

## Things to look for in a good community

There are a number of things that I would look for in a community. While this is not a definitive list of everything that makes up a good community it lists some guidelines for what I think is important:

- **Code of conduct**: Good communities have guidelines for things that the community will accept, tolerate, and abhor. It should be visible to all members of the community, and each member of the community must be accountable to those guidelines. It must also be enforced. If you note situations where the code of conduct is selectively enforced against certain members you should be wary of staying within that community.
- **Moderators**: There needs to be someone (or a group) in the community that can diffuse situations and meter out meaningful punishments when folks get out of hand. Moderators should be even-handed and as consistent as possible with their decisions. They should demonstrate that they are also following the same code of conduct by their actions in the community. A good moderator should be visible but not overbearing. You should feel welcomed by the presence of a moderator and feel free to engage the moderators if you have questions about the community.
- **Spaces for questions and guidelines for good questions**: There should be a place for folks to ask questions related to the topic of the community. People should feel safe in asking on-topic questions, and the community needs to be clear on what it considers an on-topic or good question. Is the space OK for beginner questions? If not, could such a space be made? What sorts of questions would the community be happy to answer and what sorts of questions would upset the community? These need to be clearly defined so beginners can have a sense of what the community will welcome and what it will not tolerate.
- **Joy**: Do the people in the community seem pleased to discuss

things? What is the tone of the conversations? Are folks interacting in a positive way with each other or are they resorting to insults and name-calling? Are questions welcomed or are they discouraged or ignored? If there's no joy in being in the community then the likelihood of folks sticking with it will be lowered.

- **Compassion and empathy**: Does the community allow for people to make mistakes? When something goes wrong does the community try to help? Does the community remember what it was like to be beginners and act with compassion, or do they expect everyone to have more experience before participating?
- **Kindness**: This is the most important factor — does the community behave in a kind manner to others or do they split off into factions and try to cut each other down. Do they view new folks as friends or as outsiders that must prove themselves? This relates to compassion and empathy above, but we tend to see acts of kindness before we see compassion and empathy. Kindness manifests itself when community members are OK with folks not getting everything right away and act with gentleness rather than taking a stern approach. They let folks know that they too have had trouble and suggest ways to work together to smooth things out for the next folks who might experience this same trouble. They act in a way that does not put their ego first, and instead behave as though they have been given a gift that is best shared with others.

We'll talk more about kindness in upcoming chapters.

These are just a sample of what I find in good communities. Feel

free to add to this list as your experience grows and let me know so I can update this list for future readers.

# Chapter 5

# A day's journey

## Riding until dawn

As programmers we are always trying to find new ways to be productive. Tweaks to text editors, compilation tweaks, scripts and automation; the list goes on for how programmers want to maximize their productive coding time. We also spend time tweaking the rest of our lives with the belief that we should always be doing something related to coding. Any moment we're not coding is a moment where our projects get behind. And getting behind with our coding can lead to other problems: missed deadlines, other companies getting their program to market before us, or other instances where we miss an opportunity. We're constantly worrying that we're not doing enough to succeed.

We've heard stories of developers waking up at their computers to the strange sound of beeping because they fell asleep at the keyboard and the keyboard's auto repeat can't handle any more input with their faces resting on the keys. Isn't that how developers should work?

There's a tendency to believe that because we work with machines that are tireless and ready for more work that we need to adapt ourselves to these machines. We feel the urge to always be "on" and ready to give the machine more work. Idleness is regarded as a waste. We try to become like the machine; tireless and always ready for more work.

There's a problem with always being "on." When we feel like we always have to be "on" we never let ourselves feel like we can be "off." We don't allow ourselves any periods of idleness and rest. This creates a pattern where we deny ourselves the moments to sit and reflect on what we're doing. We force ourselves to keep moving; keep programming no matter the personal cost. Our brains don't get the ability to rest, relax, and recharge. Our minds are too busy and exhausted to process what we've learned and sweep that knowledge into long-term storage. When we get exhausted we start to worry that we're not doing enough. This doesn't motivate us; instead it creates a vicious feedback loop of fear and panic. We spend our day worrying that we're not doing enough while our minds cry out "enough!" from exhaustion. This feedback loop of fear and exhaustion can spiral us into a vortex of burnout, depression, and a desire to leave programming for good.

There's a delicate balance that we need to strike between our desires of being on all the time and our needs for relaxation and reflection. Our desire for invincible and indefatigable development needs to be tempered with the reality that our bodies and minds have finite resources per day that must be allocated appropriately. Think of this as power-management for a complex machine that the manufacturer (currently) doesn't allow you to swap out the battery when it is spent. Being aware of what processes are running, how much energy is being

used, and how much energy is left is vital for ensuring you can still be functional later on in the day. That's the level of awareness we need to have about ourselves.

How do we balance these feelings of wanting to be on all the time while allowing ourselves to relax and reflect on what we're doing? How do we pay attention to the needs of this "programming machine"?

## Lights out

First, we need to acknowledge that we can't be on all the time. We may know this intuitively and think "yes, of course" but knowing is not the same as doing. We need to have periods where we are not programming and not thinking about programming. We should have moments where we can turn off the programmer part of our being. These periods of not-programming are vital to our well-being and give us chances to explore the wider world and let our minds rest in-between programming sessions.

This can be tricky if we feel like we're falling behind in our learning. When are we supposed to learn all of the new things happening daily? When are we supposed to catch up on all of that technical debt we've been accruing over the years? When will we have time to learn the ins-and-outs of technologies that aren't part of our day-to-day work but are still interesting to us?

These feelings that we have (that there's more to do, and that we need to spend every waking moment doing it lest we fall behind) aren't helped when we compare ourselves to other programmers who appear super productive. These are the programmers who think of a clever idea in the morning and have a working prototype in the afternoon

(while still handling normal work routine). When we compare ourselves against these programmers we wonder if they ever take time away from the computer.

We can acknowledge that we have feelings of wanting to push ourselves to keep learning and doing. We can notice our feelings when we think "just one more line of code before bed" or convince ourselves "I can read a few more articles or pages or [insert favorite way to consume more information here]". We can pause and notice where these feelings and thoughts come from and understand why we're still pushing ourselves beyond exhaustion.

These feelings usually stem from a sense of inadequacy. We feel like we're not measuring up to the ideals we have; whether these ideals are ones we've created or ones that are externally driven. These ideals come from analyzing other programmers (colleagues or folks we admire), and measuring our progress against their work. They also come from our own mythical ideas of what makes a perfect programmer.

What we need to realize is that those ideas of what makes good and perfect programmers are fantasies. They're a composite of what we think a good and perfect programmer should be. They don't exist in the real world. True, we may see programmers out there that seem to wake up with a keyboard attached to their hands, spend the entire day coding, and go to sleep with dreams of more code formulating in their heads. But we need to realize that we're only seeing one side of their lives. We're not seeing the whole picture of who they are. We need to focus on our own bodies and minds and realize when we are tired and need rest. We can't make ourselves into other people; we need to work with who and what we are.

Our bodies require down-time in order to be most effective. We need moments where we can step away from the keyboard and allow ourselves to wind down and relax. Our minds are not designed for constant work, especially at the levels that computer programming requires. The sooner we realize we should step back and take breaks throughout the day to recharge ourselves the happier (and more productive) we will be.

## Taking a break

Taking a break is more than just flipping over to another application on our computer. My tendency while taking a break is to start checking my email or open up one of my various chat programs to catch up on what happened since I last opened it (usually since the last time I took a break). This really isn't taking a break as it is trying to multi-task at my desk. Real breaks involve getting up from the computer. It doesn't have to be a large break; taking a break can be as simple as moving away from your work-space into another room or area. You need to move away from your computer to get a "Context Switch", where your mind can feel like it isn't in the same place as it was earlier. Context Switching lets your mind completely switch out and flush out the context of the area you're in. It allows your mind to focus on new context and new input.

This can be tricky in an office where the underlying expectation is that one must be at their work space in order to be productive. And there are only so many "bio breaks" (breaks that are related to matters of human biology, also known as using the restroom) someone can take

in such situations. How can you give yourself the context switch your mind needs in such situations?

You might be able to achieve the same sort of Context Switch by looking away from the computer display for a few moments. It's a good idea to look away from the screen every now-and-again to give your eyes a rest. Giving your mind a rest while you give your eyes a rest can give you the incentive to do both.

Changing your sitting / standing arrangement can also be a good Context Switch where you allow yourself a change in your physical workspace. It can be as simple as just standing up and stretching from time to time, or as complex as raising or lowering your standing desk. Telling yourself that there are two contexts around your desk: sitting and standing at the desk, may be enough to give yourself the Context Switch and rest that your mind needs.

If your workplace has a culture that allows you to step away from your desk and move around then that would be a great Context Switch. Adding a physical component (as much as you are able) to your Context Switch can help your mind to relax and recharge.

You'll have to experiment with a few of these to determine what works. At the bare minimum you'll want your mind to feel as though it doesn't have to be on all the time. You want your mind to have cool-down periods between coding sessions so it can flush the remnants of that session from your mental "cache" and into longer-term storage. Then when you get back to your coding session you'll be more likely to remember what was going on.

You may also find when you go away from the computer for a while that you'll forget what you were previously doing. That's OK. What I

would recommend is keeping a journal or log of what you were thinking in as much detail as you need. Either write it on a physical piece of paper or use a text file to keep these notes so you'll have enough clues to allow you to pick up where you left off.

## Productive thinking

Next, we need to realize that productivity is not a constant. There are days where we will find ourselves generating remarkable levels of code and code quality and days where we'll be lucky if we can string together a coherent string of words for a code comment. We have varying levels of energy and mental focus available to us per day. It's up to us to be mindful of these levels and understand what our productivity might look like for the day.

Understanding these swings of productivity can allow us to better gauge whether or not the day will allow us to generate the code that needs to be generated, but there's a level below that I think is important.

We put a lot of emphasis in our day on completion and hitting deadlines. This emphasis can cause us to create strong attachments to completion and deadlines. Sometimes this is warranted because of external factors (the "critical-path" of the project require us to get this done by a certain date and time). But many of our deadlines are internal deadlines that we've set for ourselves. We set a goal that we will be this productive by the end of the day. The unstated condition of this internal productivity deadline is that we'll feel guilty and ashamed if we miss the goal. We'll feel like we're not measuring up to our expectations and wonder if we're worthy of the task at hand. We'll feel like our day has been wasted and wonder if we're capable of doing anything at all.

It's better for us to remove deadlines wherever possible. We won't be able to get rid of the external ones where folks are waiting on our contributions (though it may be possible to renegotiate those if they're not hard deadlines) but we can let go of the desire to meet arbitrary productivity levels and arbitrary deadlines.

Arbitrary goals may work for some tasks. Some examples of this are game programming contests that only run for a week which makes teams focus on the critical pieces of the design and implementation of their game in order to release it in the allotted time. These can be a fun exercise for focusing your efforts, but they also incur a lot of stress and pressure before the contest's deadline. If you continually feel guilty and unworthy because you can't seem to meet the goals you set for yourself then you should reconsider whether it's useful to use them at all.

One trick that has helped me is creating small spaces of concentrated focus. That trick is described in the next section.

## Containers

We should replace soft deadlines (deadlines that aren't externally imposed on us) with a commitment to work on a particular project for a given length of time. One trick I've found useful is the idea of a "timed focus container". When I do a timed focus container I start by choosing what will be focused on during the container. Once the task is chosen I set a timer at my work-space and then focus on that task with my full attention for the remainder of the time on the timer. I've had the best luck with using 10 minutes but a session as small as 5 minutes or as large as 30 minutes can be useful. The work selected at the beginning of the container is the only thing I work on, and I do my best to

make sure there are no interruptions (whether internal or external) until the container is complete. When the work is done I wrap up the task with whatever I've completed, note whatever the next actions are for that task on my next actions list, and then take a quick break (usually around 5 minutes) before starting the next container. The next container can be a continuation of the same task, or I can select another task, but the idea is simple: I only focus on the task in front of me for the allotted time. When my mind tries to wander or I get the temptation to "just check this one thing" I pause for a moment and determine if it is indeed important. Most of the time it isn't important and I can make a quick note to check on it whenever I finish the container.

We can use these containers to overcome our desires to multitask. We only focus on one thing at a time. We can also use containers to just let the session go where it wants to take us. When we start the container we don't start off with trying to finish a particular task; instead we see where the session takes us. There is no judgment of the quality of the work in the container, just the expectation that we will work for the duration of the container. There's no expectation for what work we will accomplish, just that we will work on it until the container is finished. If we complete the task before the container ends then that's awesome! We can then figure out what the task for the next container will be. If the container ends and we're still in the middle of a task we can then write down where we left off and what steps we took in order to get there. We can then work on something else, or we can take a quick break and then come back to the work with a focus container.

The underlying concept of the timed focus container is to let ourselves agree to work within the confines of the container without judg-

ment either for the work done or the progress made. When the work is done we close out the container by reflecting on what we did and where we need to go. We give ourselves permission to not worry about our progress in the moment, but we do allow ourselves moments where we can review our progress and note how far our journey has progressed. We allow ourselves the freedom to just work in the moment without fear of judgment, reprisal, or self-recrimination. The container is a gift of uninterrupted work that we give ourselves (or at least as uninterrupted as we can manage). We make this the best gift we can give by closing out other programs, turning off notifications, and giving this task the full attention it deserves.

I invite you to incorporate this practice of doing focused containers every day. I think they are an excellent way to give ourselves permission to focus on one thing at a time without the need or worry for what will get accomplished during that container. It allows us to focus on one thing at a time and do it to the best of our abilities. The limitation of working on one thing at a time without thinking about the other bits of work that we have to do can be liberating, and I hope that working with these containers will give you a sense of what fully-focused work can feel like.

This whole book was created and edited using focus containers. I took about 10 minutes per container to write the initial draft, and later I used 10 minute containers to edit the book. Sometimes they bled over into 15 or 20 minute containers but that was because I was so engaged with the material that I didn't want to stop. This was in sharp contrast with how I've normally approached tasks. Usually I need to get over the initial hurdle of allocating a half-hour or more to the task. This means

that I need to feel like I have enough control over my schedule in order to clear out that space. Since I don't tend to feel like I have that level of control over my schedule I tend to procrastinate on the task. With a focus container I think to myself "I can just take 10 minutes to work on this" which is just enough time for my mind to not feel like it should be doing something else. With each container I gradually saw the progress of this book unfold. That then fed back into my desire to keep working on this book, which helped lower the mental friction to keep doing the containers to work on the book. It created a positive feedback-loop where I looked forward to the next time I could do the container and work on the book.

## Distractions

Life is full of distractions. So many things want our attention, and many of these distractions are outside of our control. Someone enters our work-space and needs our attention at that moment. An email thread that we thought was settled becomes a heated discussion and our attention is drawn to it. Something happens at home and now our minds are split between worrying about our work tasks and worrying about what's happening at home. Whatever the cause may be, there are times when our attention isn't where we want it to be and we feel pulled in every direction at once.

This is when the containers are most helpful. If something interrupts the container we can determine if it's more important than the work we're doing. If we determine that it is more important than what we're currently doing we can stop the container with the understanding that we'll return to the work once we've handled the interruption.

If the interruption is not more important then we can agree (both with whomever is interrupting us, or with ourselves) that our focus needs to be here with the work until the container ends. We'll be able to give that disruption our full attention once the container ends. We won't try to split our attention between the work and the interruption, rather we'll give each of them our full attention at the appropriate time.

This method creates a simple delineation between our work and the rest of the world, but just because it's simple doesn't mean it's easy. Keeping the delineation between our work and the outside world can be challenging, especially if the culture you're in is about immediate results.

I don't have good answers if the culture you're in demands your attention at all times. The best I can offer is a containerized approach that gives you at least some periods of undisturbed concentration. If you feel on-guard all the time because something might happen at any moment then you're going to remain less effective than if you can shut the world off for a bit. I'd also challenge you to examine whether that perception is really true; are you constantly being ambushed by interruptions? Testing that theory may be in order. Keep a log (whether it's a sheet of paper, text file, spreadsheet, or database is up to you) of when you did a focus container and if that container was interrupted or not. If you find that you are getting interrupted more often than not then you need to determine what is causing the interruption and assess if it's something that you can control. There are many ways to handle and minimize workplace distractions that I won't go into here, but being aware of the distractions and determining where they are coming from will be key to figuring out how to mitigate them in the future.

Also be aware of the self-imposed distractions you've added to your life. Do you need immediate notification that someone liked something you shared? Is the funny anecdote you just remembered important enough to warrant switching out of your current context so you can post it to your friends and colleagues? Do you really need something to pop up in your field of view to let you know that your music player changed a track? Are you willing to sacrifice your attention and flow throughout the day because a program detected a change in your environment, regardless of the importance of that change?

We add these distractions into our lives because we worry that we might miss something important. Programs also come configured with most of their notifications turned on so a user can be reminded of the status of the program at all times. Perhaps it's useful, but for me they are very distracting. In my career I've sat at the desks of many other folks and have cringed at the number of notifications they receive in the short period I was there (a span of ten minutes or less). I've seen folks interrupt their current line of thinking because a notification for a message unrelated to the current task distracted them. What happened to the original thought? They had to mentally switch back to it and remember where they left off, usually at great mental effort.

I challenge you to turn off as many notifications as you can and get a taste of what your experience is like without them. That may be as simple as closing out an application when you're done with it, or may be as complex as changing the settings so an application doesn't notify you when new messages arrive. You'll need to play with this and see what works best for your needs and concentration. A good rule of thumb is "what does this thing track that is important enough for me

to drop my important work and focus on this thing?." If you can scale your notifications so that only the most time-critical notifications reach you at the appropriate time then you'll be better able to relax and focus into your work. You won't have to parse the notifications to determine if what you're seeing is important or not.

One of the reasons I've heard for folks keeping their notifications on is that they feel they might receive something that requires an immediate response. We've created cultures where we feel a need to respond to messages the moment we receive them. I'd argue that most of the messages you receive during the day don't require the attention you're giving them, and certainly not the level of attention that warrants interrupting what you're doing to view and respond to them. You may be better served by scheduling several periods of the day where you do nothing but check and respond to your messages. Schedule these as infrequently as you can. Some folks recommend two or three times a day, but even setting a limit where you check your messages once an hour can make a vast improvement over how many times you're already checking your messages. You'll need to judge how often you check your messages based on your needs and your work culture. Also consider the person to whom you're responding. Does it make sense to give this person a quick, semi-thought-out response, or does this message require more time to simmer in your mind before you respond? Giving yourself time to think about a your response may give you additional insights into a problem that might not be readily apparent in the moment. This could mean the difference between one well thought out response versus a deluge of half-thought-out back-and-forth brainstorming via your messaging application. Responding to everything as

it's received is very stressful and requires huge amounts of attention that could be better used in your programming work.

It may seem challenging and foreign to live without notifications and without the need to respond to every message and notification, but our attention is finite and limited. Maintaining focus throughout the day is challenging and stressful. If we can limit the number of distractions we receive throughout the day we then give ourselves the freedom to not have to work as hard to keep our attention attuned to our programming tasks. We get to say "not right now" to our distractions and handle them at a more appropriate time.

# Chapter 6

# The map is not the territory

## The changing landscape of programming

The one constant in the field of programming is that it is always in flux. Programming languages come into prominence and then fade away over time. What once was a given is now considered obsolete (or even "considered harmful", as many essays will point out).

When I graduated from college we learned Pascal, Modula2 and Ada. Unfortunately those languages were starting to decline in popularity in favor of C. When I started my first "professional" programming position Perl was the language of choice (partially because Perl could be easily transformed into the ubiquitous CGI scripts of the era, and was considered superior to scripting tools like awk and traditional shell scripts). As of this writing I'm using Python as my main development language, and I foresee that I'll have to look into other languages to expand my programming career.

Programming requires flexibility. It's difficult to learn only one way

of doing things and have that remain relevant for over 20 years. Think back to what was current technology 20 years ago and you'll no doubt notice that things are quite different now. If you would like a fun exercise, search for articles describing the state-of-the-art technology from 20 years ago and notice how much of it you recognize.

## Learning to learn

Learning specific methodologies and technologies is not a good long-term strategy for programmers. We're better served by learning how to learn, and more importantly, how we ourselves learn. That sounds simple: once we've cracked how to learn effectively then we'll be effective programmers. Unfortunately there isn't a foolproof way to learn that works for all people. Different folks learn in different ways. All of us have learning styles that work better when certain things are emphasized. Some learn better in a classroom while others learn best with self-directed study (books, video recordings, etc.). Some can read a book and be perfectly fine with understanding the material, while others may need more visual approaches. If you have the luxury of trying several different methodologies for learning I'd encourage you to use as many as you can to figure out what works best for you. Understanding what works for you will be key to helping you progress and grow.

I've found that some simple principles work best for me. The first is repetition. I learn better when I do something daily, over and over again, in small chunks. The second is having a small goal that I can achieve. So for me having a daily practice time on a project where I can work toward an end goal works best. When I was learning Python I enrolled in PyWeek, a one week game programming sprint where the theme is

announced near the beginning and all programming happens during the week. For that entire week I made time to complete my game, and by the end of the week I'd learned more about Pygame (the library I used for my game) and Python than I had in the weeks leading up to PyWeek. Doing a one-week "game jam" (as they're currently called) is a bit extreme, but it gave me a clear goal (a completed, working game) and a time-frame to accomplish it (one week). Over the years I've learned more about Python with various projects (both professionally and for myself) that had daily practice and clear end goals.

You'll need to experiment to see what works best for you. The underlying principle is that your learning process should be something that you can use for any language or concept in programming. It should also offer the least amount of resistance to your learning. Your ability to learn and adapt will be vital to your experience as a programmer, so understanding your learning process and what works best for you will help you in this process.

At the very least, set aside 10 minutes per day as a container (see previous chapter) for focused reading and learning. There is a lot to learn in programming and creating a habit of learning will help you keep up. Remember, though, to keep your learning contained in small chunks. A lot of information can overwhelm you into thinking that you can't possibly learn it all. You're right – you can't learn it all in one sitting. If someone told you to drink one of the Great Lakes in one sitting you'd be hard pressed to complete the task (note: please don't attempt this!). If, however, you filled a glass of water several times a day from one of the Great Lakes and drank it (10 minutes at a time) you'd start to make an appreciable dent in the reduction of that lake

over your life-time. (Sure, it might not look like much on the outside, but that's the junction where reality and metaphors break down).

Each day you have an opportunity to learn more about the realm of computers and computer programming. Taking a small part of every day to learn a little bit more will help you on your journey.

## How to choose what to learn

There are many opportunities to learn, whether it be via books, tutorials, videos, or computer-based training. There are also a myriad of different topics to learn. How do you decide which one is most important to learn? How do you manage what you're learning? How do you keep from getting overwhelmed with the options available?

This brings us back to focusing on one thing at a time and understanding how you learn best. This feedback will help you decide what to learn next. One approach is to think about the things that you're most passionate about right now; what excites you at this moment? If there's something that you're eager to learn then start there. If you have multiple things that are exciting or interesting to you then write them down on a list and notice if you are more drawn to one of the topics than the others. If you're still having trouble deciding from this list then pick one at random (roll some dice or create a random-number generator to select one — that could be a project).

If you have trouble thinking of something to learn and are struggling to come up with one item that is exciting to you then give yourself permission to browse and find what is out there. Observe the conversations of other programmers and find out what they are discussing. Head to a programmer meeting to follow the discussions

of what they're talking about. Or, if you're really stuck, browse some job listings to find out what employers are searching for and notice if that sparks some interest.

This isn't about picking the most useful thing or the most important thing, though your current situation may add some urgency to certain topics over others, it is about figuring out what has your attention and where to place your focus. Don't be concerned with making the perfect choice that will get you your next job or bolster your career. This exercise is about making a choice to learn something interesting and sticking with it long enough to learn more about it.

Once you have chosen what you want to learn then it's time to focus on learning it. If you have a preferred methodology (books, videos, tutorials, classes, etc.) then spend some time (no more than an hour or so) researching what resources are available. Some topics have beginner-friendly resources available that list things that the community believes are helpful for programmers just getting started, while others may require asking the community where to start. Often something as simple as a tutorial can be a good way to get started with this exercise.

If you can find some resources in a short amount of time that's great! Start your learning process with those resources. Don't worry if they're the right resources or worry that they might lead you down the wrong path, just get started with them; you'll come back and evaluate them later. For now we're more interested in just getting started.

One trap that I'm guilty of falling into is trying to find the best resources for learning a topic. I'll spend hours looking for the right book, the right videos, the right courses; whatever it is, I want to find the best materials available. I want to reduce the amount of false-starts

while learning a topic. This seems like a noble pursuit (after all, why wouldn't you want the best materials available?). It's also a trap and can lead you into spending more time thinking about how you're learning rather than actually learning. Worse, if the material starts to confuse you (which is highly likely when you are learning something new) you'll spend your learning-time wondering if you made the right decision picking this material. You'll wonder if you chose the right material and continue searching for the best material (perhaps those good and great reviews really didn't know what they were talking about after all). This diminishes your ability to learn the topic because you're more focused on discerning the quality of instruction and not spending time on the actual instruction.

After a few days of practice sessions give yourself the opportunity to check in and see how you're learning. Are you feeling engaged or are you not enjoying this? If you're not feeling engaged (the material is loosely organized, the instructor is confusing, the examples don't work, this material assumes you're already familiar with another topic, etc.) then give yourself permission to search for better material or a different topic that interests you more. Even if your learning-experience wasn't great you'll have a better idea of what to look for when choosing something new. You'll have a sense of where your gaps are in the topic and will have a better feel for what you're looking for in learning materials.

If you're finding that the topic you're trying to learn is no longer interesting to you then give yourself a few moments to reflect on why that is. Is it a difficult topic? Do you feel ready for the topic? Are you currently overwhelmed with other projects and are feeling tired when you approach this topic? Sometimes we think we're ready to learn a

topic, only to realize that there's something else we need to know before we can fully understand the topic. It's OK to find additional resources and focus on those before we tackle this topic. Just be aware of your struggles and your internal dialog. Be honest with yourself about why you want to move to something different. See yourself in the difficulty and notice if you're wanting to run because it is difficult or if you are truly unprepared for or uninterested in this topic. See if you can engage more with the difficulty and notice when you start to feel overwhelmed by it. Give yourself permission to stick with the difficulty as long as you can and notice your feelings and urges as you practice with it.

Treat your learning as an iterative process, with regular check-in periods to note your progress. Think about how you feel when you're learning. Are you excited and engaged or do you feel tired and withdrawn? Do you procrastinate when you think about this topic? When you focus on your learning does your mind wander? Note these feelings as they occur during your focus sessions and reflect on them when you think about your overall learning process. Later you can reflect on those feelings and see the patterns in your learning process. If you feel tired while learning you may want to try adjusting when you do your learning session. You may need more sleep or need to find other materials that are more stimulating. If you feel overwhelmed perhaps you need to start with something more basic before tackling this difficult project. If you're confused perhaps there is someone you can ask questions to gain clarity. These answers may not be apparent while you're in the moment (you may be too busy feeling frustrated to understand where that frustration is coming from), but with practice you'll be better equipped to notice your feelings. When you notice these feelings

you can use them to learn how your mind works and understand what it needs in order to keep engaged with your learning.

## Resistance and The Container

Any time we learn new things we put ourselves into a vulnerable and uncomfortable place. We take the things we are familiar with and try to apply them as we push into new territory. We become uncertain of the outcome; will it be successful or will it be a failure? Will this topic be too difficult for us to grasp? Will it help us or hurt us? Will we choose the wrong thing to learn and will that cost us opportunities in the long run?

Discomfort and uncertainty are certainly a part of learning, but instead of thinking of them as something to be avoided we should instead think of them as beacons. A beacon gives us direction and illumination when we're in uncertain territory. When we feel uncertain about what we're doing that feeling means we're pushing into new territory. Instead of trying to avoid it or wishing for comfort, we can instead relish that we're in uncertain territory and feel those brief twinges of fear and doubt. We can say "I'm about to learn something new. I'm frightened, and don't know where this will lead, but that's OK. I'm willing to see where this goes and enjoy the journey."

We've been conditioned to think of the unknown as something to be feared. These emotions have served us well. They've kept us from venturing too far out of our comfort zone and exploring the unknown. When you're living in forests and caves the unknown can house all sorts of dangers. It makes sense not to provoke those dangers by showing up on their doorstep. But programming is not the same as venturing into a

dark forest or peeking into a damp cave; programming hardly warrants the amount of fear we give it. Instead we need to realize that we're not in any mortal danger. Our fears are merely letting us know that we're venturing into the uncharted territories of ignorance. It's up to us to let our fears know that this is OK and that by exploring these realms we will only find understanding.

Steven Pressfield in *The War of Art* nicknamed these feelings "Resistance". He considers Resistance as a sort of mythological being who lives in each of us to thwart creative acts. As the work progresses Resistance ratchets up the pressure to stop by introducing the feelings of fear and anxiety that we mentioned above. I think of Resistance as something that also happens whenever we are learning, especially if we're learning tools that help us in our creative pursuits. Pressfield limited his definition to creative folks who were working to complete creative work (books, paintings, games, etc.), but I'm expanding his definition to the learning process itself. In our case Resistance shows up when we're learning the tools to help us be more creative. Resistance is what tells us we're not good enough to learn these things, or we're unworthy of the benefits they'll bring us. It tries to keep us safe in what we already know.

This is why the "focus container" is so important: it gives us small doses of discomfort and difficulty in manageable chunks. We can guide ourselves through small amounts of daily discomfort and keep learning through our discomfort. It helps us work through our tendency to avoid and hide from difficult situations. If we focus on one thing at a time we can keep ourselves from the distracting thoughts about whether or not this is the thing we should be working on. Whatever we're working

on in this moment is exactly what we should be working on. Whatever learning material is in front of us is what we should be learning. We can be secure in knowing that everything we are doing for the duration of this container is exactly as it should be. When we finish the container we can reassess how it went and what challenges lay ahead.

## Mapping out longer-term goals

As you progress through the learning process you'll start to see that a lot of what we call programming is interconnected. Languages borrow heavily from each other and ideas that seem new and innovative have their roots in concepts dating back to the genesis of computing. Rather than dissuading us it should encourage us to open the doors of programming by learning simple, transferable concepts. The question is, which ones?

The simplest answer is "all of them", but that's hardly satisfactory or possible. A less cheeky answer would be "enough of them to start seeing the patterns emerge" but that sounds more like a truism than something we can use to start making our longer term goals for learning.

Rather than give specific advice on which concepts will serve you best in your pursuit of becoming a better programmer I'm going to suggest a technique that might help you map out what could help you.

Programming languages will mention the concepts they borrow from. Whenever you're learning and you see mention of one of these other concepts make a note of it and keep focusing on what you're learning now. When you've completed your learning for the day review the list of other concepts and do some searching to see what else shows up. If there are other things that show up then write them down

on your list. These concepts might not make sense at the moment but having that list available and referring to it might help you make connections about programming that you might not otherwise notice.

When I was learning JavaScript I noticed that someone mentioned that JavaScript borrowed from languages like Scheme. Scheme is a functional language based on Lisp and was created as a teaching language for functional programming and recursion. So I took a brief detour into learning Scheme, partly because it was more interesting to me than JavaScript. Call it "creative procrastination", if you're being charitable. What I learned while learning Scheme piqued my interest into other functional languages and functional programming. This in turn helped me understand some of the functional programming paradigms that were becoming popular in Python (list comprehensions, lambdas, etc.). By taking a brief detour in my learning of JavaScript I learned more about a whole family of languages and now I feel like I understand JavaScript and Python with more clarity than when I started.

I'm not suggesting that everyone take the "creative procrastination" steps like I have (I'm still in the process of learning JavaScript as of this writing), but it does help to make notes of the concepts you encounter and dig further.

This is one way to map out learning goals (notice the other connections that show up as you are learning and be curious about how they fit together), but you may need a different approach. Perhaps you're under pressure to learn something to remain marketable or acquire some skill for your job that needs to be learned quickly. How do you map out those goals?

The pressure to learn quickly can make any task seem insurmount-

able, especially if you don't know how best to proceed. You may be tempted to rush through this process and hope you retain the knowledge you've learned. This approach doesn't lead to understanding, it leads to stress and burnout. The approach I'm outlining is designed to help you learn how to learn. The best way to learn something quickly is to understand how other concepts fit together with what you're learning. This is great when you have experience with a lot of different languages and concepts, but for those who don't have much experience yet it will feel like you're trying to shove an elephant through a small funnel. This is where practicing learning every day will help you. It will help you break apart larger learning goals into smaller chunks and will help you recognize the fear and discomfort for what they truly are: acknowledgment that you're expanding your skills into new territory.

Longer-term goals are just goals that have been broken down into shorter-term goals. Focus on the short-term goals and allow yourself to course-correct and follow a few connections as needed.

## Failure and learning

One thing that we are afraid of while learning is failure. We worry that we won't learn the topic quickly or completely. We pick up material that starts off simply but later on becomes very complex, and we struggle to keep up. We try typing example code into our editors and find ourselves needing help to get them to work. We fail to grasp the material and wonder if we'll ever learn what we're trying to learn.

Failure is a part of learning. If you knew the material you wouldn't be learning.

One of the reasons for practicing learning using containers is be-

cause we give ourselves those brief moments of failure and repetition. Repetition is how we get better at whatever we are learning. Failure allows us to course-correct our learning so we can determine how best to approach this the next time we make an attempt.

We often feel that failure is something to be avoided, but while we're learning it is unavoidable. Our learning process requires us to fail in order to get better at what we're learning. That's the whole point of learning: reworking our brains so that they can finally understand the concepts we are trying to learn.

Part of learning is having the right mindset for learning. Instead of feeling like you're constantly failing and struggling to keep up you may want to approach it with a different perspective. Instead of thinking "I can't do this. It's too hard.", approach it with a more curious "This is all new to me. This is why I'm practicing learning this." Giving yourself a more positive mindset will help keep you from giving up when you struggle with the material.

## Dead ends and changing topography

Sometimes we'll find ourselves learning something that's a dead end. We look at our progress and see no real improvement. We don't find the topic as engaging or as exciting as we'd imagined. We realize that what we're learning is an evolutionary dead-end in the realm of programming. What then?

Part of our learning process is understanding that our expectations of how something will turn out can be completely different from how things actually do turn out. We envision all sorts of rewards and platitudes that never come. Does that mean we're at a dead end? I don't

think so. It might be that what we expected we'd be doing with our newfound knowledge isn't panning out. We might find our expectations for how quickly we'd learn the topic aren't being met. We may also expect that our career will be bolstered by learning this topic, yet the job market hasn't recognized our new-found skills with job offers or more money.

Our engagement is related to our expectations. Programming demands a certain amount of fun and reward, and if we're not finding the experience fun or rewarding then we're unlikely to want to continue learning that topic. Our minds begin wanting something else to engage us, and we start craving anything other than to continue with this learning process. After all, shouldn't we be enjoying this? If there's no engagement and enjoyment then the learning becomes drudgery. We become distracted more easily while trying to learn and our minds drift away rather than focusing on our learning experience.

There is also the problem of learning things that are evolutionary dead ends. The world of computing is littered with the remains of technologies and methodologies that are either no longer relevant or are considered "out-of-fashion." What once was cutting edge is now considered moribund, and the community around that technology or methodology scoots on to new technologies and methodologies and leaves their previous work as a technological ghost town. When we mention that we're learning these things we get curious looks from developers: "Why would you learn that? We've moved on to this other thing." It's as if we've heard about a party and arrived in time to see the clean-up crew picking up the litter and breaking down the tables

and chairs. We feel like we've missed out on the good parts and wonder if it's even worth trying to keep up and find the next thing.

All of these can pose their own problems for learning, but it's up to us to take a more critical look at why we started this whole process of learning. What did we bring into this?

In each of these cases we brought our expectations of how the learning would progress. We brought the expectation that it would always be fun, engaging, and relevant. Sometimes our learning expectations do pan out, but when they don't we get discouraged and disappointed.

Rather than being upset at how our expectations of learning this technology or methodology aren't being met we can take a more mindful approach. We can see ourselves in our moments of learning and notice if we're trying to bring more than our focused attention into the learning container. We can realize that learning is about changing ourselves and change is not always fun, engaging, or pleasant. We can put aside our expectations and concentrate on the learning itself.

That doesn't mean we shouldn't acknowledge our feelings. We should certainly acknowledge the feelings of boredom, anxiety, disillusionment, and so on, but we should also be mindful of where those feelings originate. Are we truly bored or is this our mind trying to tell us to stop so we can do something more fun? Are we not engaged with this material because we don't find it relevant or are we giving in to our distractions? Is this really a dead-end in our learning or are we just feeling stuck? Notice when the feeling comes up and be curious about what prompted the feeling. Note when you get the feeling and where you feel it most in your body. Stay with the feeling for a few seconds and keep noticing it. Then, continue your work. While you work

keep noticing all of the feelings you're having and repeat the process of staying and noticing your feelings. When you're finished you can reflect more on those feelings and make an honest determination of what those feelings are indicating. Through this process you can clarify what is causing those feelings and notice if they are just resistance to learning new material or a desire to run to distractions or something more familiar.

If, however, you realize that you're really not enjoying learning this topic, if you feel you're spending more time convincing yourself to learn rather than actually learning, then you'll need to have an honest discussion with yourself about why you're learning this topic at all. Is this topic still relevant to you or has the topic become irrelevant? Are you learning this out of an obligation to yourself or others, and is that obligation still present? Are you trying to learn whatever it is because you're worried you'll be left behind, personally or professionally? Think about what brought you to start learning this topic and determine if the situation has changed. If someone came up to you and asked you if you would like to use this topic in the next few days would you consider it?

You'll need to reconsider your true motivations for learning this topic and see if they still match what you want to do with your programming profession. You will also need to be honest with yourself about why you're learning this topic and why it is important to you. There are plenty of things to learn that are great career paths, but if you have no interest in the topic, or are just learning it "to get hired" you're going to have a more difficult time learning the topic than if you had a genuine interest in it. You'll also need to determine if this is just resistance to learning. Your challenge will be to sort out your true

feelings about this topic and tease out whether you've genuinely lost interest or are just struggling.

There have been many things in my career that I have tried to learn, but there have been many more that I haven't learned. Part of the reason I haven't learned them is because the computing landscape changed as I was learning them. At school I learned the Pascal language. I got reasonably good at it but over time my Pascal skills have faded. Right now there's very little need for proficient Pascal programmers so continuing to develop my Pascal skills would be purely for my own enjoyment. I find other computing topics more enjoyable so my Pascal skills lie dormant. Should Pascal arise from its moribund state I can revisit the decision to reinvigorate my Pascal knowledge, but for now I'm content that I've made the right call. At one point in my career the Java language came to prominence. I spent many sessions learning Java until I realized I didn't enjoy the language. It felt too cumbersome to me and the directions it took weren't ones that I cared to pursue. So after some reflection I stopped learning Java. Was this all wasted time? Hardly. During my sessions I learned more about Object Oriented Programming and how objects fit together. I learned more about recursion while trying to solve a problem for one of my projects. These skills transcend Java, so when I started learning Python I was able to transfer my knowledge on how objects worked from Java to Python. I used that knowledge to understand what Python was doing and how it was different from Java. Should the need arise I can revisit my decision to stop learning Java and see if it interests me again.

It's OK to give up on learning something. It's up to you to determine what you want to learn and for how long. We are complex beings and

our interests morph and change. We also exist in a complex industry of changing whims and technologies. What was interesting and necessary at the beginning of the year might become uninteresting or unnecessary at the end of the year. We shouldn't feel beholden to learning something just because others are learning it or because the job market seems to require it. Give yourself permission to listen to your own desires. If they match up with what a fickle industry wants then great! Go learn with abandon. But if they don't match up and you find yourself spending weeks trying to stir up enough motivation to learn the topic then you're doing yourself and your craft a disservice. Let this topic sit dormant for a bit and give yourself something else to learn. There is little point in making yourself miserable to please others.

If you feel the urge to revisit this topic at a later point then let yourself come back to it. You should also allow yourself to come back to this topic without the baggage and expectations of your previous attempts. Saying "I already tried this once, so we'll see if this works this time" sets your mind to expect that you will give up again. Give yourself permission to approach this topic as though you're experiencing this topic fresh, with no expectations of how it will turn out. Be gentle with yourself and experience this topic again from your current perspective.

## Approach with curiosity

As beginners we engaged the computer with curiosity and enthusiasm. We didn't know what to expect and had no idea how long it would take. We just learned as much as we could and took everything at face-value. As we continued to learn we traded our curiosity for certainty, and our enthusiasm for expectations. The excitement we got from learning be-

came the drudgery of feeling that we must always be learning. We can re-capture that beginner's spirit by looking at each opportunity to learn as a new experience. We can let go of our expectations of how our learning will progress and instead approach each learning session with curiosity for what we will learn during the session. We can re-kindle the spark that we had when we were beginners with infinite possibilities. That spark will sustain us through the periods of uncertainty.

We can learn to love learning again. With each focus container we can approach our learning fresh, with no preconceived notions of how it will end, and be curious for what we'll find when we dig deeper into what we're learning. Each learning session brings us one step closer on our journey to close up the gaps. There so much to explore in our field. I hope you always find something new and exciting to help you on your journey.

# Chapter 7

# The struggle within

## The emotions of programming

There's a stereotype of a programmer sitting emotionless in front of the computer. The stereotypical programmer sits, quietly entering lines of code as though they were transcribing them from memory. If you're a programmer or have been around programmers you know that the stereotype should be that of a frustrated composer. Sure, we sit in front of our computers in long periods of silence and concentration, but we're far from emotionless. We bask in the glories of code that works perfectly the first time. We glower at code that misbehaves. We go from cheering ourselves in victory to cursing the machine and threatening it with clenched fists. We clench our teeth when bugs rear their misbehaving heads. We swing from emotion to emotion: exuberance, joy, fear, anger, resentment, sadness, loneliness, guilt, and shame.

No wonder we're exhausted by the end of the day.

Programming is a taxing process. Not only do we need to keep a

mental model of the software we're working on, but we also keep a mental model of how the software should behave. We create a story of how this software will work and paint a picture of how we will feel when everything works as we envisioned. We create an emotional attachment to the software. Our emotional state can mirror what we feel about what we're creating; excited, bored, or stuck. Keeping a positive attitude about software that isn't measuring up to our expectations is exhausting. Couple that with our own insecurities, fears, and doubts and we begin to see why programmers tend to burn out — it's a combination of the stress of the job and our emotional reaction to that stress.

## Emotional drains

There are several factors that can cause us emotional highs and lows while programming. These are some that I've noticed, both in my own programming and while talking to others about their programming.

### .1 Purpose and utility

If we clearly see where and how this code will become useful we can get a sense of drive and purpose — we're working toward something that will benefit folks! We know that people are depending on us so we do our best to make the code work regardless of the pitfalls that await us. We tap into the emotional highs of self-worth and purpose to help carry us through to completion.

The opposite is true, of course — if we don't see the purpose then our work will seem useless and in vain. We'll struggle to meet deadlines and feel a sense of worthlessness in our pursuits. Sometimes it's a project

that isn't aligned with our own purposes and goals. It can be a poorly managed project that we're being forced to work on because of external pressures. We might find ourselves forced to meet arbitrary deadlines that we never agreed to meet. We can become frustrated if we don't understand the ultimate goal of whatever project we're working on.

## .2 Engagement vs. boredom

We've already experienced several layers of engagement with our programming. These are the projects that don't feel like a chore while we work on them. We feel like we're learning something each step of the way. The outside world disappears while we work in this cocoon of focus. We lose track of time and feel both disoriented and refreshed when the work is completed.

Unfortunately we're probably more experienced with the opposite of engagement: boredom. The code base doesn't engage we at all. The topic we're learning or working on is just re-hashing something we already know. It's a chore to get started. Everything else in the world feels way more interesting and the minutes drag along throughout the whole process.

## .3 Awake vs. tired

Sleep is a major contributor for how we perceive the world. Getting enough sleep allows us to feel refreshed, awake, and inspired. We need to have the energy reserves to take on whatever challenges befall us. When we don't get enough quality sleep we become irritable and less-open to engagement. We conserve our resources as best we can lest we

use them up too quickly. We look to stimulants (caffeine, distractions, and the like) to keep us engaged throughout the day.

### .4 Mental state

I'm using "mental state" in a broad sense to cover any of our existing feelings and current mental well-being. These can range from temporary feelings of unhappiness and melancholy to complex and serious topics like clinical depression and Post Traumatic Stress Disorder (PTSD). Our minds are complex machines that do their best to adapt to the situations and environments presented to them. At times this adaptation can clash with our desires to be productive and the struggle between our mental state and our desires can cause further emotional drain, discomfort, and despair.

There are more things that can affect our emotions but these are the ones that I'd like to focus on as they cover a broad spectrum of what we bring to the tasks of learning and programming.

## Awareness of our Emotional State

Being aware of our emotional state (what we're feeling right now) gives us our current emotional location. We can map out where we are and understand what our mind is telling us. Giving ourselves a few moments to truly notice what emotional state our mind is in will help us to move forward.

Note that we're not trying to change our emotional state. We're not trying to force ourselves to be something that we aren't. If we're truly unhappy with where we are or what we're doing it's more helpful

to understand what's causing our unhappiness rather than try to paper over and prevent those emotions. Seeing our emotions clearly allows us to recognize what is causing them. Being present with these emotions allows us to better understand our mental state and what we're capable of in the moment.

You can do this in the context of mindfulness meditation but even sitting at your desk and thinking "for one to two minutes I'm just going to sit here and explore my emotional state" should suffice. Noticing our emotions, understanding what they are, and digging in to find out what is causing them can help us understand what we're feeling.

You might already know what is causing these emotions and emotional state and be afraid of exploring them. Some emotions may overwhelm us and make us feel in ways that we don't want to feel. This is especially true for emotions related to anxiety and PTSD. Do as much of exploration and introspection as you are able, and be gentle with yourself. Remember, you're not trying to change the emotions, only to notice them. You may find that your gentle prodding of these emotions can lead you to better understand them. Be as brave as you can with these emotions and if they start to overwhelm you then pull back and let the residue of those feelings subside before continuing.

## Our story

Each of us has a story we tell ourselves. These stories shape our perception of the world. We tell ourselves stories of how the day will be and how we will engage with the day. We create a world through our stories in which we are the central protagonist of our story. We tell stories such as "the work I'm about to do will be amazing" or "I'm going to

work through this problem quickly and will have an awesome solution when I'm done". That's if we're being positive with ourselves. When we're being negative with ourselves our stories weave a tale about how we're not good enough at what we're doing and will likely fail in the attempt. Those stories create a complex tale of struggle, pain, and misery where everything wrong with the world is the direct result of our actions.

Our emotions help inform the type of story we tell. If we're feeling amazing we tell ourselves that what lies ahead will also be amazing. If we're feeling down and defeated our story reflects our defeated tone.

The truth is that our story is just that — a story. Our stories are not a guarantee of how the day will progress. We can tell ourselves a story that today will be amazing and watch in horror as each interaction causes our day to be anything but amazing. Conversely, our story could be that today will be terrible and we won't accomplish anything, but instead we experience a decent and productive day. The story only accentuates what we're experiencing; it can't predict what we will experience.

Rather than being attached to these grand stories we can focus more on the things that we love about the present moment. Instead of a story that you're going to have an amazing day you could focus on the aspects of your project that appeal to you and hope that you can work on them soon. Instead of filling your day with stories of dread and doom you can focus on the little victories that happen along the way. Even something as simple as "my computer booted without crashing" can be a victory. One of those little victories could be setting an intention to remain focused and curious for the next 10 minutes (the focus

container from previous chapters) and celebrating when you make it through that intention. You can get more little victories as you keep working with that intention throughout the day. Our little victories won't all be perfect (perhaps your computer is being extra stubborn today), but we can use them to re-calibrate our day for the next 10 minutes and keep using them to re-calibrate throughout the day as each container of focus becomes another little victory.

Giving ourselves the ability to focus more on the present and the very next steps we're about to take gives us a mindful way to check in with ourselves and our progress. We can focus on the positive aspects of what we're doing instead of worrying about how reality is diverging from our stories. We can course-correct throughout the day and keep trending towards a more positive and productive day rather than fretting about how distant we are from our ideal day.

This will take practice. We're accustomed to letting our stories drive our day, but over time we'll be able to break our day into smaller chunks where we can be more mindful of the stories we tell ourselves.

## Awareness in action

Let's pretend for a moment that it's a typical day for us. Today we're feeling anxious. We've just received a bug report and it's related to something we've been working on. The bug report states that the code that we committed to the project earlier this year isn't working and likely never worked the way we thought it worked. As we read the bug report our anxiety levels increase. Our inner monologue kicks in and we start telling ourselves that we aren't nearly as good as we thought. We're not perfect. We suck. We didn't get enough sleep the night before

so our emotions are in a state of heightened awareness. Our mind races and flashes back to images of the other times when we've failed. As we keep reading our sense of dread kicks in. Our internal monologue becomes a frenzied chatter: "What will they think of me? What do they think of me now? Am I going to lose my job over this?"

Before we've even finished reading the bug report we've created a story. The story begins with our own anxiety for what will happen during the day. Then the worst happens: we get something that confirms our fears. The story then presents us with a montage of our past failures and adds this latest bug report as a capstone to the montage. Our story then ratchets up the pressure by raising the stakes of the importance of this bug report: not only do we have to fix whatever broke, but now we have to fix our reputation and start a job search. As the story progresses in our minds we wonder if we'll ever work as a programmer again, and feel that our career as a programmer is over.

The story we created is a terrible story, but I'm sure you can relate to the factors that generate it. It's a story that draws from the deep pools of our own feelings of inadequacy and insecurity. It's fueled by fear: fear that you'll ruin your reputation, fear they won't trust you, and fear that you'll fail.

Fear is one of the most powerful emotions we have, but it's not the only one. Reading that bug report may also elicit other emotions like grief (we thought that code was good and now that thought is gone.), uncertainty (how will we fix the problem?), and anger (how could we have deluded ourselves into thinking this worked?). We may also have other feelings: sadness, loneliness, and abandonment. Our sense of self

worth may also be affected, and we could feel disconnected from those whom we serve and the folks we work with.

Being aware of these feelings can help us parse the story we told ourselves and how it didn't match reality. These feelings and the story we told ourselves can give us feedback on how we are perceiving our world and the work we're doing. Pausing for a moment to acknowledge our feelings and understand where they are coming from give us an understanding of what our emotions are trying to tell us.

You can relax now. The bug report in this book isn't real, but take a moment to recognize the feelings that you felt when you read the above section and notice where your mind went. That's the kind of awareness we're seeking to have.

## Finding our feelings

Our feelings manifest themselves in our bodies in many different ways. Fear can be a knot in our stomachs or tension in our chests. Anger can make our jaws clench or make our heads feel hotter than normal. Sadness can feel like a weight upon our shoulders, or make us feel tired. When we notice these feelings we can pause for a moment and just sit with our feelings while we keep noticing them.

Think of this exercise as though you are scanning your body for the source of the feelings you're having. Notice where your mind is drawn: tightness in your chest, tightness in your stomach, a clenched jaw, or whatever you may feel. Notice the sensation of that feeling. You can dig deeper and try to find the underlying causes of the feeling but for now just notice that it exists. Sit for a few moments more and be curious about how it feels. Notice any other attributes about that feeling: color,

texture, intensity, or any other attributes you're experiencing. Let the feeling exist — be kind and gentle with it. Allow it to exist without judgment. Give it space. Above all, don't try to fight the feeling or wish that it would end; just notice it. Eventually the feeling may subside, but for now just acknowledge that you have this feeling and you're going to be curious about it.

Some feelings and emotions are more painful or traumatic than others. Give them space and let yourself be curious about them for as long as you are able. If you notice your mind starting to panic or feeling overwhelmed by these feelings then you may stop noticing them before they overtake you. Remind yourself that these are emotions and those emotions are a part of you. You and your emotions work together to help you. You're both on the same team.

This exercise isn't about dwelling on or punishing yourself with your feelings. If the act of noticing these feelings causes you to experience physical or emotional pain then you may need help from a professional or a support group to help guide you in understanding these feelings and where they come from. A professional or a support group can help you have those feelings without having those feelings overwhelm you. There is no shame in finding others to help you along your journey.

## Emotional Triage

One of our learned behaviors with our feelings is to run away from them or try to suppress them. We do our best to avoid feelings that make us unhappy or uncomfortable. We also try to hold back our positive feelings lest we show too much exuberance. This can lead us to be confused or conflicted about what we're feeling and why we're feeling

that way. By sitting with our feelings and emotions and understanding where they're coming from we can get a clearer idea of what our mind is thinking and the story we're telling ourselves.

Think of this practice as emotional triage. Hopefully you've never had to go to a hospital emergency room, but if you have you'll see a whole array of medical professionals who are trained to diagnose what just walked through the door and determine the severity of the problem. When we recognize and reflect on our emotions we too are diagnosing what emotions we're having and the severity of those emotions. We take these moments when we're experiencing these emotions to determine what the emotions are and what triggered them. As we review our emotions we are gentle with them and recognize them for what they are. A good medical professional doesn't impose their own desires on the patient; they simply accept the patient for who they are, diagnose what the patient is experiencing, and act accordingly. When we recognize our emotions for what they are and determine where they are coming from we can better understand what we're facing.

The more we do this practice the better we'll become at recognizing our emotions and why we're having them. We'll be better able to notice what we're feeling and understand why we're feeling that way. When we feel anxious we can recognize that it might be because we're exploring an area of programming that we don't fully understand. We can feel that anxiety for a bit (don't try to chase it away) and then think about what we're currently working on and how we can explore those areas that are new to us. We can then mentally note those areas or write them down (preferably in a journal) so that when we complete what we're doing we can review the areas that caused us anxiety.

With this practice we can turn our emotions from something that drives us into something that guides us. We can use our emotions as tools to better calibrate our internal stories. We can re-frame our stories about how we're unworthy of being called programmers and instead give ourselves the intention that we're going to spend the next 10 minutes exploring this area of our work and finding where the gaps are. We can set an intention to be curious about where this next 10 minutes will lead us. As we continue to explore these topics we'll notice our emotions and use those emotions to let us know where we feel we need to improve and adapt. This will allow us to change our plans as needed and address those areas we feel are lacking or need improvement. This cycle continues in each practice container, with our emotions acting as a barometer for our comfort level with this topic, and helping us draft a road-map for how best to proceed. We transform our discomfort and anxiety from things that hinder our progress into indicators of where we feel we need to focus our attention.

## Burnout

One thing our emotional triage can help us diagnose is the feeling of being burned out. Burnout is a collection of emotions coupled with emotional and physical exhaustion. Burnout can be something as simple as being bored or overworked, but it can also be the sign of something more serious. It can lead to physical or mental complications if we're not careful. We can work ourselves into serious levels of exhaustion and delude ourselves into believing it's part of the cost of being a programmer.

Burnout manifests itself in different ways. For some it may be the

feeling of dread while working on a project. They feel like they are ineffectual in making any changes. For others burnout can be feeling exhausted. They feel as though they're on a treadmill that will not stop. Worse, they wanted that treadmill to stop a long time ago. Burnout can also manifest in feelings of being creatively drained, where imagining a different future is difficult and things that were once inspiring or interesting no longer generate that spark.

Burnout is tricky to self-diagnose because it is a collection of seemingly unrelated emotions. Our feelings of boredom, fear, exhaustion, and anxiety can all have different root causes, but when we combine those feelings with an unrelenting work schedule and loss of control we amplify those feelings. Left unchecked we can lead ourselves into trying to numb out those feelings. We'll find ourselves not wanting to program anymore, and resent ourselves for ever getting into programming in the first place. We can cause ourselves more undue suffering by just "powering through it" which can lead us to compound and complicate our emotional state.

There are some things we can do to understand and help alleviate burnout:

- Realize that we're burned out, or about to burn out. Acknowledging that we're about to burn out is key to not experiencing the burnout. That seems simple enough but we tend to ignore the symptoms when we're nearing the throes of burnout. If we can recognize that we're about to burn out then we can take measures to avoid it. And if we realize that we're burned out we can take measures to be kind to ourselves and help ourselves out of this burned-out state.

- Examine our emotions. Sit for a while and notice what emotions come into view. Are we feeling stress, fear, anxiety, nervousness, or anger? Notice what feelings emerge and recognize these feelings. Examine where these feelings are coming from and what might be triggering these emotions.
- Re-negotiate our commitments. Many times burnout is the result of over commitment, whether to ourselves or others. We always have too much to do, and despite our best efforts we will always acquire new obligations. Perhaps the plans we made were too aggressive, or something changed in the world that disrupted our plans. Whatever the reasons we may need to re-evaluate what is expected of us and what we are capable of doing. If we see that we've created an intractable situation for ourselves we need to figure out how to remove some of these obligations or re-negotiate them.
- Give our "drive" a rest. Unlike our mechanical counterparts we need downtime and rest. We can't work a straight eight or more hours without at least some moments where we aren't working. Programming demands a lot of mental bandwidth and pushing ourselves to exhaustion can lead to emotional instability, stress, and burnout.
- Examine if this is truly how we want to live our lives. We need to determine if what we're doing is really what we want to be doing. If we're not happy with what we're doing then every moment we continue doing it can compound our feelings of unhappiness. If we feel nothing but dread for our current situation then we may need to renegotiate our commitments. That can be something as

simple as agreeing not to learn something right now, or can be as complex as taking on different work or changing careers.

By understanding that we're headed toward burnout (or are burned-out already) we can take measures to course-correct so we can approach our programming practice with joy and enthusiasm. Sometimes taking a step back and re-evaluating what we're doing can help us not sit in the constant loops of frustration, anger, and guilt. Changing our story to better fit reality can keep us from trying to match an impossible dream.

I mentioned before about re-negotiating commitments. We often get ourselves into situations where we have way more to do than is physically possible, even under the best of circumstances. This may be in part because we've said "yes" to too many things, or because we're being swamped with work commitments, such as a large high priority project, or several smaller projects that need urgent attention. The best way to renegotiate your work load is to review your work load and notice which tasks feel "urgent" and which ones feel "important". "Urgent" tasks are tasks that feel like they need to be done immediately. They might not be "important" tasks, but they have a sense of urgency to them. "Important" tasks are tasks that will benefit yourself or others. These are tasks that have significant value when completed, both monetarily and significance. Take out a sheet of paper or open up a text document and create two categories: "urgent" and "important". List out the tasks you need to complete and categorize them under "urgent" or "important". Next mark the due date (as best you can) of each of these tasks. If you have more than three urgent and important items and they're all due the same week then it's likely you're overworked

and will need to renegotiate those commitments. You may feel that you are capable of doing all of these things but if you're already feeling stressed, tired, and burned out then you'll only compound those feeling by trying to meet the deadlines. If you can, find out if you can move some of these deadlines to the next week, or check with your customers to find out if these are really as urgent and important as you think they are. If they are urgent or important then find out if your management can assist you with other resources, or if they can intervene to renegotiate these deadlines and priorities. If you're truly stuck (management won't budge and the customers won't renegotiate the commitments) then you have some decisions to make about how important their priorities are versus your own capabilities. There's the temptation to say that your management and your customer's priorities are more important than your own priorities (they facilitate your income, which contributes to making your lifestyle possible), but your own health and well-being should have more weight in your decision than their priorities and deadlines. Perhaps you can negotiate some down-time after this period so you can rest, relax, and regain your strength and mental acuity before being plunged into a similar situation.

Learning to say "no" is an important skill as a programmer. Too often we regard ourselves as super-beings that can do anything, in part because the computers we work on seem like they can do anything. Unfortunately, we have finite physical and emotional resources, so learning to pick and choose the projects that are most important to us (depending on our own internal criteria) will help sustain us as we progress through our programming careers. If we say "yes" to everything that someone pitches to us then we'll have less time to work on things that

really matter to us. We'll be at the mercy of folks whose priorities and desires do not match our own. The most effective way to burn-out is to spend all of your energy working on projects that don't match your own priorities and desires.

You will experience periods of burnout in your programming career. Things will come at you that will overwhelm your ability to cope with them. You will find yourself stuck in loops wondering if this is really what you should be doing. Understanding what you're feeling and acknowledging your feelings are valid is the first step to changing your trajectory from burnout and stress. Programming shouldn't be drudgery (no work of value should be drudgery). There should be something in your programming day that keeps you motivated and helps you grow your skills. Adding bits of learning, joy, and wonder, along with periods of downtime, will help sustain you through the emotional turbulence that awaits. And recognizing when you're burning out and renegotiating your agreements with yourself and others can help reinvigorate your desires to keep programming.

## Reaching out for help

I want to emphasize that it's OK to ask others for help. I've struggled with asking for help. Part of my reluctance in asking for help was instilled in me whenever I would ask a question and get the dreaded "you should know that already" response. Other times I believed that by asking for help I would somehow diminish my reputation. I'd be exposed as a fraud and an impostor. Folks would wonder why they ever trusted me in the first place. But when I actually asked for help the responses I received weren't "why don't you know this?"; they were "why didn't

you ask for help sooner?". Sure, there were occasions where someone would be surprised I didn't know something, or I would receive some criticism for my ignorance, but I've found that the benefits of asking the question outweighed any negative effects.

Asking for help isn't just limited to asking technical questions; there are many more ways that we might need help. We may need to ask our colleagues to help us during a difficult time in our lives. We may need the help of our management when we're struggling personally and professionally. We may even need a whole different set of support staff to help us along (doctors, therapists, etc.). Involving other people with our struggle can be daunting (even overwhelming) but getting help early can help prevent the more serious forms of burnout and stress.

The most common reason for our reluctance in asking for help is our desire for comfort. Asking for help means placing ourselves into a state of vulnerability and hoping the people we're asking to help us will treat us with kindness, respect, and dignity. This vulnerability can be amplified if we don't know the person we're asking for help, or if the person is a medical professional. But putting ourselves in these vulnerable situations is necessary, especially if the problems or situations we're facing are out of our control or experience. If we're close to burning out (or are suffering through burnout) we may need the help of a doctor or therapist to uncover better ways to cope with what we're experiencing. If our job is causing stress and strain we may want to talk with others in our community to find out if others are also experiencing these feelings. Even the simple act of commiseration with our peers can help us realize that we're not alone in facing these issues, and may help us find better ways of managing our workload and stress. They may also help

us recognize abnormal or abusive situations that we're facing. Sometimes we don't realize when our jobs or relationships have turned from caring and nurturing to ones that bring us more harm than anything positive.

"There's no shame in asking for help" is an overused phrase, but asking for help is not a shameful act. We need the help of others. Even someone saying "I'm sorry you're dealing with that" can be a connection with someone else who sympathizes what we're going through. Finding others who are willing to listen, empathize, and commiserate can be the difference between feeling part of a community and feeling like we've been abandoned in our profession.

We also need to recognize when our support systems aren't supporting us. If we find that talking with someone else is not helping us resolve the issue we may need to find other means of help. We may realize that we need additional support.

Realizing the need for additional support can be difficult, but once you have come to that realization I'd encourage you to act and get additional help. This requires self-awareness and honesty with how you are feeling. Only you know your situation and if you're being honest with yourself. If you're not being honest with yourself then only you can realize this and can take the initiative to seek out the help that you need. Nobody else knows your inner-workings better than you.

Asking for and receiving help is a skill, and like any skill it needs practice. When we're young we have simple means of asking for help (crying, pointing, etc.). These skills are baked into us as part of our survival mechanisms, but as we grow our world becomes more complex. Our methods for asking for help need to mature as we mature. This is

not something that comes naturally to any of us. We will struggle to ask for help, and we will resist when we're receiving help from others. Repetition and careful practice will help us improve our skills in asking for help. Improving these skills will help us to overcome the obstacles we face throughout our day. That improvement will help us to become not only better programmers but also better at handling the challenges that life gives us.

## Giving up

Programmers don't like to think about giving up. How many times have we asked others to be patient while we try to fix something that isn't working? ("Just a few more minutes, please. Honest!") We work on machines that seem to have limitless possibilities. As programmers we feel compelled to explore those possibilities, but sometimes we don't want to do that exploration. Sometimes we look at the list of things we should be learning and wonder if it's worth the effort. We look at job postings for our set of skills and find nothing but lists of meaningless work. New programmers ask us what it's like to be a programmer and we consider if we should warn them about the dangers of choosing a career that have led us to being unhappy and unfulfilled. The joy that sustained us while learning the craft disappears and we struggle with the fear that we will never cultivate that feeling again.

Programming isn't for everyone. There are times when I've wondered if I should continue working as a programmer. I'm frustrated that I can't possibly learn everything that I want to know. I worry if what I'm learning will still be relevant by the time I'm finished. I'm anxious that I won't be able to compete in a job market where every-

one else seems like they had a head start. I struggle looking through job positions that offer work that I don't think will be relevant six months from now, let alone 10 to 100 years from now. I feel like the computing future I was promised has been corrupted and all of us are stuck in a world where computers are little more than levers for companies to pry open the wallets of their customers.

It's easy to become fatalistic about the practice of programming, but I've realized that there's more to computing and programming than what the job market has to offer.

Part of the joy of programming is curiosity. If we can nurture our curiosity while programming then we have so many avenues to explore. There are always other ideas and other topics to discover, such as game development, esoteric languages, or other programming paradigms. The job market uses a fraction of the programming ideas that are out there waiting to be explored. There are also many emulators and retro-computers available with good documentation and vibrant communities. One area that has intrigued me is learning about how older computers work. Older computers are simple and can be understood with patience and the right mindset. These machines are well-understood, and most of these older programs were put together by one programmer. They make an excellent space for learning not only how older machines worked, but many of the concepts that still permeate our modern machines.

What happens when we realize that there is no joy left for us in programming? What do we do when the thought of programming computers no longer excites us? How do we keep going when even the thought of trying something new fills us with dread? What then?

If we don't find any joy in programming then we need to understand why we are feeling this way. Maybe we're tired after a rough project that sapped out all of the fun and excitement of programming for us. Perhaps the communities we've joined in our area and online are hostile and unwelcoming. Maybe we thought programming would be more fun but every time we start we wish we were doing *something* / *anything* else instead.

Programming is at its best when you really want to do it. It isn't for everyone. If you're stuck in a situation where you don't want to program anymore then your best course of action is to quit being a programmer. There's no shame in quitting programming — many programmers have lost the spark of joy and the desire to keep programming, and have moved into other fields. It's OK to leave the field of computer programming and do something else.

Programming is only one facet of our lives. True, it may be a big facet of our lives, and it may feel scary to give up something that we've worked so hard to accomplish. But if we realize that we're just going through the motions and are no longer experiencing any joy in programming then it's time to think about what else we can be doing with our lives outside of programming. We're granted a limited amount of time to live our lives. Doing something we don't enjoy robs us of a meaningful life.

Giving up should not be a negative experience. There is no shame in taking time away from being a programmer. Plenty of programmers have given themselves a "sabbatical" from programming to explore other interests and recharge themselves. Breaking loops of negative experiences in programming can help us identify what we want

out of programming and a programming career. It can help us find and confirm our innermost feelings about programming and see if we're still meant to keep pursuing this path.

There are several fears that can keep us from making this break with programming. The first fear goes by the fancy name of the "sunk cost fallacy". The sunk cost fallacy is the belief that the time and effort we spent learning and programming is an investment, and that investment will be wasted if we quit. Thus, in order to preserve our investment we must keep programming. The problem with this fallacy is that it assumes we have not already received the benefit from that time and effort. I'd argue that learning any sort of programming is not a wasted skill. Programming can be applied to many facets of our lives, such as simplifying tasks into manageable steps, applying structured thinking, and understanding basic Boolean logic. Many other fields have also adopted computers so having computer skills can be helpful for yourself or others. The knowledge you have will not go to waste.

The second fear is the fear that we'll somehow let down our fellow programmers and others in our organization if we stop programming. This one is tricky. It's tricky because it includes others in our decision-making process. We might be in an organization that has a substantial load of tasks to complete, and our decision to quit will mean these tasks won't be completed the way we wish those tasks to be completed. It's not hard to imagine our absence causing harm to the entire organization and resulting in its eventual collapse. Is this scenario true? It's up to us to tease out whether our absence will truly let everyone in our organization down. This puts us in a situation where our fear leaves us feeling "stuck". We feel "stuck" because our fear has created a situation

where we're choosing between our own well being or the well being of others. This is a false dichotomy. Our absence might be the catalyst for someone else to pick up our tasks and work on them, and possibly complete them more effectively than we can in our current state. We need to determine if we are truly irreplaceable or could someone else take our place? The answer might be "I am irreplaceable, but I need to leave this situation or I will cause harm to both myself and others if this continues". It's up to us to review if we are helping ourselves and the organizations we serve, or if we are harming them and ourselves by deluding ourselves that this is working.

The third fear deals with our own personal concept of identity and the memory of our community. If we decide to stop being a programmer will that somehow erase a part of our identity? Will our community stop identifying us as a programmer and will we lose contact with folks that have become friends and colleagues? Again, this fear is tricky to overcome because programming may be a large part of our identity. Letting go of programming can lead to feeling like we are stripping away a piece of ourselves and our identity. There's also the fear that folks will stop calling us for jobs or other programming projects if we decide to take a temporary break. If the break is temporary will people remember our programming skills when we decide to return?

Each of these fears and worries are valid, but they may not be the truth. We can be afraid that we wasted our time as a programmer, but the truth is any learning isn't wasted effort. We can worry how others perceive us or how the organizations we were a part of move on without us, but the truth is we can't control their perceptions and actions. What we can control is our participation in each of these communities

and our perception of the time end effort invested. We can determine if a hard break from programming would be better than gradually easing ourselves out of our commitments. We can clarify to others what our current status as a programmer is and determine if this status is temporary or permanent. The most important thing is not letting others persuade us into doing something that we don't want to do or is harmful to us. If we need to stop programming because we are emotionally drained and burned out then we need to make it clear to others that we will be doing a disservice to them and ourselves if we continue.

Mature communities will understand the need to take a break and stop programming. They will understand that your mental and emotional well-being is more important than their need for you to continue, and they will be able to piece together what needs to be done and heal from your absence. It is normal and natural for folks to move on from organizations and pursue other priorities.

What's important to remember is that it's OK to turn off that portion of your being and stop being a programmer. Whether or not you make that a permanent change is up to you and your desires. Feeling emotionally drained, uninspired, and burned out is counterproductive to your programming practice — programming is hard enough. Taking a break from programming to explore other interests is natural and doesn't mean you're less of a programmer for wanting to do something different to recharge yourself. If you find that you're happiest when you're not programming then pursue whatever else has your attention with wild abandon. If you decide to return to programming after being away for a bit then you can return and pick up your learning practice. Remember: our lives take many different turns and paths. The best

path for you is the one you make yourself, regardless of where that might lead.

# Chapter 8

# Epilogue

It's a cliché for an author to say the book they wrote is the one they wished they had read when they were confronted with the topics presented in the book. Perhaps it's a cliché because it's true: this book contains the advice that would have helped me when I started this journey. Too often I've wondered if I've measured up to whatever metrics I created to represent the ideal programmer. Many times I saw the success of my peers and wondered if I was defective as a programmer or deficient in my learning. What was I getting wrong that others were getting right?

I've come to realize that every programmer's journey is unique. Your journey is going to wend and wind in directions that are different than the directions that my journey took. You'll have experiences that I can't share, and I've had experiences that will be difficult for you to replicate unless you have a time machine. Neither of our experiences is more or less valid than each others', nor are they more or less valid than the experiences of other programmers. These are our experiences.

The areas of our knowledge are only the areas that we've explored so far. There will always be gaps, but we can define those gaps as the areas that we haven't explored yet. There's always more to explore, and that exploration is the fun part of the journey.

No traveler can be at all places at all times. They must travel to each destination as quickly or as slowly as their transport allows, and stay there for as long as they can before traveling to their next destination. They travel with whatever companions they can find in whatever communities they can build. They build relationships and trust with themselves and others. They use their strengths to help others, and explore and improve their weaknesses. Each day they press onward. Like the traveler we too must choose our destinations and our companions. We can find those who, like us, are traveling down the same road and help us on our journey. We can exchange stories about our successes and failures, and experience each day as another link in our journey.

I continue my journey each day and I hope that as a programmer you continue your journey each day for as long as you are able. We might not be on the same exact roads together, but we have the same goal: doing the best we can in each moment.

I wish you well on your journeys, and hope to hear the tales of your travels when we meet again.

# Chapter 9

# Gratitude

This book would not exist without the folks who have accompanied me on my journey, both as instructors and as colleagues. My thanks and appreciation to all of my instructors in my formative years for giving me their best efforts to teach me programming in its various forms. I am indebted to all of my colleagues and programming friends over the years who shared their knowledge with me and trusted me enough to help them along the way. I am also blessed to have many communities that help sustain me, including the Michigan!/usr/group, PyOhio, Coffee House Coders, and the Ubuntu Michigan Loco. Also to those who don't fit in these neat categories; know that if we have spent any time discussing programming or other matters that our discussions are deeply appreciated.

I also am grateful for the work of Leo Babauta of Zen Habits which provided me the ideas of mindfulness and focus containers. They have been transformative in my own work, as this book demonstrates. I com-

mitted to spending at least 10 minutes each morning writing each section, and the results are the work you see before you.

Thank you to those who helped me directly with this project. Thank you to my mom, Sharon Maloney, for help in my editing of this book. Any mist steaks what remain are an responsibilities of the author. Thank you to Beau Sheldon for reviewing the chapter on mental health and for helping me to better understand and highlight areas where folks struggle. Thank you to my friend, David Revoy, for his amazing cover art and for his inspiration throughout the project. Thank you to Esteban Manchado Velázquez for adding CSS and cleaning up the HTML version of the text. Thank you to the beta readers for your valuable comments and feedback in the Framagit Repository, including (in alphabetical order by handle or first name): Brendan Kidwell, D. Joe Anderson, David Revoy, Eric Hallam, Jer Lance, Matthew Piccinato, Matthew Balch, Midgard, Nicholas Guarracino, RJ Quiralta, Valvin, and Wilhelm Fitzpatrick. Thank you to Paco Esteban, Shreyas Ragavan, TheOtherNick, and Victorhck for editing fixes.

My deepest gratitude goes to my wife JoDee and my parents for their support and belief in me. Words cannot express the love and thanks I have for you.

# Appendix A

# My journey

My journey as a programmer started when I was in elementary school. I became interested in computers after reading about them in the World Book Encyclopedia and hoped to work with them some day. What I didn't realize was that those encyclopedias were out-of-date and only showed the larger, more expensive mainframe and mini-computers of the 1960s and not the more modern microcomputers that were introduced in the late 1970s. When I realized that an Apple ][ was a microcomputer and that it was designed for the home market I began my quest to get a computer of my own (AKA: I started dropping not-so-subtle hints to my parents that I wanted a computer). I scoured magazines like Popular Computing and Byte Magazine looking for the right computer; from the Commodore VIC-20 and Sinclair ZX-80 to the Radio Shack TRS-80 Model III (Even the Rockwell AIM-65 or Heathkit H89 would have worked. I wasn't picky back then.) My dad took me to computer stores and I marveled at the variety of machines that were there (and likely made a few sales-people nervous as I poked and prod-

ded the new and rather expensive machines). Finally my dad picked up an Atari 400 computer with tape drive, and I began learning BASIC programming in earnest. Around the same time my school opened a "computer lab" with three Commodore PET 4032 machines (complete with floppy disk drives), and I found myself spending every moment I could with those machines. In high school I took two programming courses, one in BASIC and the other in Pascal (which was my first exposure to procedural languages, and the basic concepts of computer science). In college I majored in Computer Science with a Bachelor of Science and did my best to keep up with all of the things that they tried to teach me. Unfortunately, I wasn't a great student (especially in mathematics). I struggled with and later dropped my compilers class, and felt like I was falling behind where other students succeeded. Most of our classes used Pascal, which I was becoming more familiar with, but there were a few classes that used COBOL, Ada, SNOBOL, C, and assembly language. I graduated with modest scores and returned home.

Throughout my career I've straddled the divide between system administration and programming. Linux was similar to the SunOS machines that I admired in college, so I transitioned to using that as my primary OS around 1995. My first jobs tasked me with maintaining various sorts of computers: desktop PCs, UNIX-based machines, and backing up the occasional VAX machine. It wasn't until one of my positions needed a website that I added more programming to my resume. Programming websites in the 1990s was where I really started to learn and understand Perl, SQL, relational databases, and HTML. The web was so new in the 1990s that all of the folks on our projects were learning at the same time. I leveraged my Perl knowledge into several other jobs

and projects doing web-based programming. Perl in the 1990s was a language where the basics were simple to learn but the language could handle really complex ideas and data structures. Perl and CGI made it easy to get something onto a web page that had some interactivity. Where Perl becomes complex is the syntax for things like regular expressions, and the tendency for Perl programmers to value code that does multiple actions on the same line. The Perl community also values code that is clever, which lead me to wonder on several occasions if I was clever enough to be a Perl programmer.

One of the companies I worked at decided to migrate a Perl system over to a Java-based environment. They looked at the skills of the existing development team and decided they needed to outsource the project to another company. This was a common trend in the early 2000s for reasons that are outside of the scope of this book. This gave me my first taste as a team leader. I know a lot of programmers migrate over into managerial roles but at the time I didn't feel I had fully explored my programming potential. I sat down on several occasions and tried to learn Java but it never clicked for me. Java web development always felt more cumbersome than Perl CGI scripts that I created. It also didn't help that we were shipping `.war` files into a Tomcat system, which seemed like they were comprised of a lot of configuration metadata and very little code. This is what I was referring to when I spoke about being OK with giving up on learning something — sometimes what we try to learn is more of a chore. Having something that is a chore isn't going to provide a good learning experience.

It was around this time that I began learning Python on my own using Pygame. I took the plunge by entering my first Pyweek compe-

tition. Pyweek is a week-long competition where folks build an entire game from scratch in one week. It was a challenge, but was also one of the most rewarding experiences I've had in programming. I built a game called "Busy Busy Bugs" that was playable during a time when I really didn't know what I was doing. In many ways I was learning to swim by throwing myself into the proverbial deep end. I wasn't in any danger, but the desire to get something done at the end of a week drove me in ways I didn't think was possible.

As the technical lead position continued I found myself wanting to do something else. I interviewed at several places and was hired at Sourceforge as a member of the Systems Operations Group. Sourceforge was an amazing experience for me. I dreamed of working for an Open Source company, and few Open Source companies were as well-regarded as Sourceforge and Slashdot were in the Open Source community, but my insecurities and "impostor syndrome" kicked in. Would I be able to cut it? Would they hire me only to realize they'd made a mistake and I wasn't as good as I claimed to be? I was friends with and had gone to school with several of the people who worked at Sourceforge / Slashdot, so I wondered about their motivations for hiring me. Was my getting hired an elaborate prank, or was I hired because several folks in the company knew me? All of these thoughts ran through my head while I worked there. It didn't help that my position was primarily system administration at a level that I was inexperienced at. There was also a programming component to the position, but I constantly felt like I was in way over my head. I lived in constant fear that I was going to be found out and that the job that I wanted would no longer be available to me. Granted I learned a lot and had very supportive

and kind management at Sourceforge but I lived in dread whenever my manager would check in on me because I feared that the conversation would only highlight that I wasn't really supposed to be there.

I'd love to say that it had a happy ending and that my fears were unfounded, but sadly I was let go from that position due to budgetary constraints. I'm grateful for the opportunities I had there, the friends I made, and the experiences I had but I'd be lying if the layoff didn't come with a mixture of sadness and relief. Sadness that I might never have a cool job like that again, and relief that I could put away those impostor syndrome feelings for the time being. In many ways I grew from the experience of working at Sourceforge and learned a lot about myself and my capabilities while working there, but it was also the position where I felt my impostor syndrome at its fullest.

Later on I was hired for a full-time position doing Python programming. This position allowed me to strengthen my craft as a Python developer. I created many interesting projects there and kept learning more about Python. It helped me to recuperate and broaden my skills. The folks at this position were very supportive and kind. There were times where it got stressful (all jobs seem have stress), but overall it was a positive experience.

Sadly I was also laid off from that position (money sucks, y'know?).

I started my job search in earnest and went to a bunch of interviews. While everyone in the interviews seemed impressed with my skills and my career I fell into one of two categories: either I wasn't a good fit for the position, or I didn't have enough skills in areas they were looking for. I found myself taking timed-coding-tests that felt like they were testing whether I got stressed easily rather than any coding skills

I might have. I sat in coding sessions with shadowy figures that barked out commands and requirements while I tried my best to follow them. I did math puzzles and logic problems (which is horrifying if you're not good at either math or logic problems). Promising leads turned into rejection letters (assuming I heard back at all), and desperation set in as I faced the real prospect that I would have to give up programming if I wanted to make a living. Visions of heading back to the beginnings of my career filled me with dread. It seemed like nobody wanted to take a chance on me anymore, and I couldn't compete with so many new programmers who hadn't made the mistakes that I had in my career.

I registered themediocreprogrammer.com during this period. If I was going to be a fuck-up then I might as well own it.

Fortunately, I've had a community of friends and fellow programmers to help support me. My current position is contracting with one of these friends to help him with programming tasks. That position came about from showing up to PyOhio (a programming conference) every year. Throughout my struggles I've been fortunate to have a community there to help me. This is why I think communities are so great — they give us a network of folks that we might not otherwise have, and help us through our triumphs and our struggles.

I'm a collection of all of these experiences. They all make me who I am. Sometimes I wonder if I should have taken a different path or done something different, but that's a futile exercise. The best I can do with these experiences is learn from them and move on. Each day I work to improve and better myself. Each day I make new mistakes, but that's all part of the learning process.

My journey continues.