The Software Craftsman

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

The Software Craftsman

Professionalism, Pragmatism, Pride

Sandro Mancuso

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Mancuso, Sandro.
The software craftsman : professionalism, pragmatism, pride / Sandro Mancuso.
pages cm
Includes index.
ISBN 978-0-13-405250-2 (pbk. : alk. paper)—ISBN 0-13-405250-1 (pbk. : alk. paper)
1. Computer software—Development. 2. Software architecture. 3. Quality of products. I. Title.
QA76.76.D47M3614 2015
005.3—dc23

2014040470

ISBN-13: 978-0-13-405250-2
ISBN-10: 0-13-405250-1

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, December 2014

This book is dedicated to my parents, Luiz Carlos and Marisa Mancuso,
for all the sacrifices they've made so that I could have better opportunities in life.
It was a very long and difficult road I shall never forget.

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

In 1973 Roberta Flack sang "Killing Me Softly." You've no doubt heard it in elevators or on your grandmother's radio station. It's a soft, lilting ballad about a woman who goes to a concert and hears a young man sing a song that she so strongly identifies with that she ponders whether the young man had found and read her letters. She even professes her belief that he feels her pain when he strums his guitar, and is singing the story of her whole life.

The book you are holding feels that way to me. Sandro Mancuso's career has been very different from mine. He is a bit younger than I. He and I have lived our lives and worked our careers on different continents and in different cultures. We share neither nationality nor ethnicity. I've met him only a few times, and each time it was never for more than a few minutes. In short, about the only thing we have in common is that we are both programmers. But that, it seems, is enough.

Within the pages you are holding you will find a fascinating alternation between autobiographical anecdotes that chronicle the author's vast experience and authoritative recommendations based on those experiences. If you are a programmer, you will feel these stories and recommendations resonate within you. You will say to yourself, as I did, "Been there. Done that." And you may hear the strains of that song echoing in your mind as he strums your pain.

And strum your pain he will, because this book is about pain. It is about the pain that you and I and, indeed, all programmers experience. It is the pain of feeling constantly constrained to do a poor job. It is the pain of feeling trapped in an *un-profession*. It is the pain of *wanting to do better* and not knowing how.

This book also contains the antidote for that pain. Indeed, I believe it contains the *cure*. You see, this book is all about software professionalism. Not just the professionalism of the programmer, but also the professionalism of the whole software organization. This is a book about Software Craftsmanship.

In these pages the author lays out a plan, a strategy, a set of attitudes, and a suite of principles that every programmer, programming team, and software organization can use to haul themselves out of the mire of mediocrity, to make themselves more professional, more productive, and more proud of the work they do.

The scope of this book is incredibly wide. Topics range from design patterns, pair programming, and Test-Driven Development to how to conduct and evaluate interviews, how to respond to tight deadlines, how to write job descriptions, and how to relate to coworkers and managers.

In short, this book is an encyclopedia on the behavior, attributes, and structure of an organization striving to grow in professionalism and adhere to the principles of Software Craftsmanship.

If you are the type of programmer, team lead, or manager who craves to be able to go home after a long day of work, look in the mirror, and say, "Damn, I did a good job today!" then this is the book for you.

—Robert C. Martin
August 2014

Preface

Back in the mid-1990s, two years after I started my professional career, a large international company in São Paulo, Brazil, announced that they would be hiring 60 developers in one go. Their selection process would take a few weeks and was divided into four phases: a three-hour-long technical test; two weeks of training in their proprietary technology followed by a test; a full day of group dynamics; and a round of final interviews. They announced it in a major newspaper and around 900 developers applied. I was working for a small software house at the time, a place where I was very happy, but I felt that I was ready for something bigger. Since the first phase would be on a Saturday, I decided to apply. Fewer than 300 developers passed to the next phase, and I was one of them. I was happy and confident but worried as well. I had to resign from my current job in order to move forward with the selection process because I would need to take too many days off for the remaining phases. Back then, my financial situation was not great and I could not count on any monetary support from my family. It was quite hard to resign from a job I liked in order to pursue a dream job that I had no idea I would get. Also, I had no idea how I would pay my bills if I didn't get that new job. But I had to do it. I had to try. That was the type of company I wanted to work for. That's what I wanted for my career.

I was 21 years old, and although I was young, I already had quite a few years of coding experience—I started coding when I was 11 and started my professional

career when I was 19. The problem is that the mix of youth and a bit of experience can easily lead to arrogance. And by no means was I an exception. Think of an arrogant, young developer you know. I could beat anyone hands down. I used to think I was awesome, better than any other person who studied with me at university, and better than the vast majority of the developers who worked with me in my previous jobs.

After going through all of the four phases, the international company announced that they could not find 60 developers at the level they were expecting. They hired just 32, and I was one of them. I was over the moon and more cocky than ever. In my first week there, I was placed in one of the teams responsible for delivering one of the business modules of the system. For the first few weeks, while speaking to developers from different business modules, I heard them talking about this *other* team, a team that was supposedly the best team in the company. They were the "architecture" team, responsible for the core of our system and for providing all infrastructure code used by the business teams.

The architecture team was led by this incredible guy who, besides doing all the *management* work, was also a fantastic developer. He was a busy man, but he would always find some time to code, check in code against his name, and review the code written by his team. I heard their team was always working on interesting things and that their code was really good. That's exactly what I was looking for. I wanted to be working with the best.

After a few long weeks, I decided to speak to the *manager* of the architecture team, this guy that I heard so much about. I didn't really know what to say or what to expect. I was only certain of one thing: I had nothing to lose. In the worst-case scenario he would say that he was not interested to have me on his team. One day I saw him alone in the coffee area. I was shaking. I approached him and introduced myself. "Hi, I'm Sandro." He looked at me and shook my hand with a smile. "I'm Namur. Nice to meet you." He was calm and relaxed. "I want to work for you," I nervously said after a few awkward seconds. He was a bit surprised but apparently took it in a very positive way. We then started talking about the selection process, why I applied, what I was expecting from the job. He also asked me if I had pet projects, technologies I was interested in, if I wrote code outside working hours, and some other random things that I don't

remember. After around 30 minutes of conversation he asked me when I could start. I was shocked. I was not expecting that at all. I was expecting to schedule a meeting, have a formal interview, and so on. It took me a long time to realize that he spent the entire conversation measuring my passion for software development. He was analyzing whether I cared about doing things right. He was not worried about my current technical knowledge. "I'll speak to my manager and hopefully it will be as soon as possible," I said. After a few weeks I was sitting among my new teammates.

My first day was a Monday. In the morning, Namur came to talk to me and assign me a task. He explained one part of the application and what I had to do, and he said he would sit with me again on Friday to check what I had done. I was thrilled. That was my chance to shine. I had to show him why I deserved to be there. I stayed in the office until almost midnight, slept a few hours, arrived very early on Tuesday, and around two o'clock in the afternoon, I was done. I had finished my first assignment in less than half of the time I had been given. I was feeling great. Well, I always knew how good I was, but being able to do that in that team, in a totally unknown code base, was a huge achievement.

I rushed to Namur's office and with excitement I said, "It's done. I finished it. And it is *working*." He stopped typing and turned to me. "Making things work is the minimum I expect from someone who is paid for it," he calmly said. "When you say that something is finished, it is implied that it works." That was like a big slap in my face. My smile faded away a little bit but I thought, maybe it is just his way of saying things. Maybe he is having a bad day. No, he definitely did not mean to be rude. "Sit here and let's see what you've done," he continued. I sat there and watched him typing in the command line, checking in the code from source control, and opening my single *.pas* file containing all my code. He opened the file using this horrible black and green editor on the command line. That was the first time I had seen vi. We were using Delphi back then and Delphi was very famous for its amazingly powerful integrated development environment (IDE). For me, seeing someone opening Delphi files on vi (a Unix text editor) was very alien. "Come close so we can have a look at it together," he said. I had written about 200 lines of code. He positioned the cursor on the first line and started looking at the code line by line. Every five lines he would stop and say things like, "Do you know what happens when we allocate and deallocate

memory? Can you see here? You are allocating memory in one method and deallocating it in another method. This is a potential risk of memory leak. Have you heard about temporal coupling? Can you see this block of lines here? If you thought a little bit harder, you could reduce these eight lines to two. Do you know what happens when you have a `try/catch` block this big? What about the names of this variable and method? What do they mean? Have you ever thought that some of your colleagues, when needing to change this code, may not have the same amount of information and context as you have now? How would you feel if you knew nothing about this part of the code but you had to maintain it? What about this hard-coded bit here? Have you ever thought that if we wanted to change where it points to, we would need to open it, change it, recompile it, and redeploy the entire application? Why do you have this piece of code duplicated all over the place? Wow! This is a big method. Do you know how much we would need to keep in our heads if every single method were that big? What about making them smaller and naming them according to their behavior?" He went on and on.

At some point he stopped and started staring at a few lines of code. He spent a few minutes there, occasionally moving the cursor one page up and down again. Back in the 1990s, a developer would be considered a senior developer if she could write code that no one else could understand. "Wow! She must be really good. I have no idea what her code does." And I made sure to have some cryptic code in there, trying to show how clever I was. At some point he figured out what the code was doing. I was expecting a compliment, at last. "Do you know how disrespectful this is?" he said calmly. "We work on a very large system, with many teams and developers working on the same code base. Can you imagine how hard it would be to understand the code if everyone decided to show off how smart he or she is? Imagine thousands, if not millions, of lines written like that." And that was a second slap in my face.

It was just 200 lines of code, and I could not answer any of his questions or find a good response to the points he had raised. Line by line he looked at the code, criticized it, and explained to me how it could be better. Once we reached the end of the file, I was ashamed and extremely annoyed. He was still very calm, as if we had been looking at some random code written by an absent stranger. "Have you understood everything I said? Do you agree with all the *suggestions*?"

Without saying a word, I just nodded. "Do you feel you could write this code in a better way now?" Without looking at him, I just nodded. "Do you feel you can apply the things we discussed as we move forward?" Once again, I just nodded. He then pressed a few keys and deleted the entire file with all my code in it. "Excellent. Since you still have three days left, do it again."

I was shocked. I didn't know how to react to that. I stood up and slowly walked to the door without saying a word. "Sandro," he called me when I reached the door. I stopped and looked back at him. "*How it is done is as important as getting it done.*" And with this, he turned back to his computer and started typing again in that horrible green editor.

I was frustrated. In fact, I was furious. I left his office, went straight downstairs, and then outside the building. Who the hell does he think he is speaking to me like that? What a bastard. I can't work for a guy like that. That's it. I'm done with this company. I'm resigning. After a few cigarettes, feeling a little bit calmer, I started reflecting on what had happened. Namur had spent over one hour going through my code and explaining to me how it could be improved. He listened to me on the few occasions I expressed my view and calmly showed me that either I was wrong or there were better ways. I realized that for the first time since I wrote my first line of code, I had found someone who took the time to show me how to write good code. I had found someone who, besides being far better and more experienced than I was, really cared about helping others become better. I had found someone who cared about producing great, quality software. I had found someone who took the time to teach me. And more than that, I had found my first mentor.

After a few more cigarettes, I pulled myself together and went back inside a different person. That day I learned I was not as good as I thought I was. I learned how to be humble. I learned that I had a lot more to learn. I learned that just getting things done was not enough, especially when you are working in a team. I learned to be respectful to my peers and clients, not leaving crap code behind. I learned that a great professional cares about his or her work.

For two and a half years I worked with my mentor and some of the best developers I have ever met. That experience shaped me not just as a professional, but

also as a person. Although we never used the term, I realized more than a decade later that that was my first encounter with Software Craftsmanship. I learned a lot from all those guys. Technically speaking, it was a great experience, but that was not the most important thing. What was really important to learn was the attitude that my boss and all the other developers had toward their career. The last words he said during our first code review session changed me forever. Ten years later, I founded the London Software Craftsmanship Community (LSCC) (http://www.meetup.com/london-software-craftsmanship) and Namur's words, "*How it is done is as important as getting it done*," were some of the first words I put on the website. Later I also had LSCC t-shirts printed with those words on them. Those words did not just make me a better professional, they also made me a better person.

About This Book

After decades and many different methodologies, software projects are still failing. Although there are many reasons why they fail, there are a few things that cannot be ignored: managers see software development as a production line; companies do not know how to manage software projects and how to hire good developers; and many developers still behave like unskilled, unmotivated workers, providing very poor service to their employers and clients. With the advent of Agile methodologies, the software industry took a big step forward, but the percentage of failing software projects is still incredibly high. Why are all these projects still failing? Why are we so bad at delivering successful projects? What is missing?

Although the term "Software Craftsmanship" has been around for over a decade, it was not until recently that it emerged as a viable solution for many of the problems faced by the software industry today.

Software Craftsmanship proposes a very different mindset for developers and companies. Although Software Craftsmanship is not a methodology, it strongly recommends the adoption of certain technical practices and disciplines, mostly the ones defined by Extreme Programming. With a great synergy with Agile and Lean principles, Software Craftsmanship promises to take our industry to the

next level. Professionalism, technical excellence, and customer satisfaction are the main focus of Software Craftsmanship. One of its main focuses is changing the perceptions that software developers are like workers on a production line and that software projects can be run as if running a factory.

How can we become better developers? How can we deliver better software projects? With real stories and practical advice for developers and companies, this book is relevant to all software developers and every professional directly involved in a software project.

This page intentionally left blank

Acknowledgments

My career has been a great journey, from a small town in Brazil to the biggest city in Europe. Having worked for many different companies and on both sides of the Atlantic, I met a lot of tremendous people who, in one way or another, helped to shape the person and the professional I am today.

First I would like to thank Professor Maria Cecilia Capelache. When I told her I would need to quit the university for financial reasons, she immediately offered me a job, which helped me to continue my studies. Even today I feel she gave me the job more to help me than because she really needed another developer on her team. She taught me the difference between all the academic things we learn at university and the real world of software development. If I had quit university I probably wouldn't be writing this book today.

Later I met Luiz Fernando Ferreira, one of the owners of a small software house, who gave me my second job. Although I was their first employee, I was never treated as such. From day one I felt I was working with an old friend. For his friendship, which has continued into today, his transparency during the good and bad times, and amazing support when the time came for me to move on, I am forever grateful.

Moving to a big international company and working under my first mentor changed my life. I'll be eternally thankful to Eduardo Namur for making me understand that how it is done is as important as getting it done, for making each one of us feel we were a big family, for instilling in me to be the best I could be, and for exposing me to the principles of Software Craftsmanship before the term had even been coined.

Alexandre Ehrenberger, a former manager who became a great friend and inspiration, was my biggest supporter when I decided to move to London. He had lived in Canada for six years and helped me understand what I needed to do in order to achieve my dream of moving to London. For his friendship, advice, encouragement, support, and for showing me that with a lot of hard work dreams can come true, I'm very thankful.

I knew that if I wanted to move to London I had to do two things. One was to learn Java, which I solved by studying hard and finding a job where I could work with it. The other one was to improve my English. I would like to thank Ana Maria Netuzzi, a lovely lady, for all the years of private English lessons, for being a friend, and more often than not, for being my psychologist, listening to all my personal problems and giving me valuable life advice.

Once in the UK and after a short period working for a small software house, I joined a startup. This startup had many talented developers and for the second time in my career I felt I was working with people who really cared about what they were doing. They helped me to step up my game, but most important, they were my first friends in the UK. They not only made me a better developer but also made me feel accepted in the UK. To Chris Webb, Greg Cawthorn, David Parry, Russell Webb, Simon Kirk, and James Kavanagh, thank you.

Joining Valtech was a big step forward in my career. It was there that I had my first contact with Agile and Extreme Programming (XP), and it is by far the place where I have learned the most. Although I could not name all the amazing people I met there, I feel the need to name a few. First I would like to thank Akbar Zamir for being a great mentor and a friend. The most important thing he taught me was to open my mind to new things. He made me see how much I did not know. Akbar convinced me to try Test-Driven Development. He also

introduced me to Domain-Driven Design (DDD) and taught me a lot about Agile, XP, diplomacy, pragmatism, and professionalism. Thanks, Akbar. I learned a lot from you. There were a few other people in Valtech who had a huge influence in my career. They are David Draper, Kevin Harkin, Andrew Rendell, and James Bowman. Thank you for teaching and helping me so much.

UBS was my first contact with really huge enterprise systems and a place that forced me to reevaluate all my beliefs when it comes to software development. There I had the pleasure to work with a great team. They really helped me see things differently. One of the best things about UBS was the opportunity, once again, to work with Mashooq Badar (ex-Valtech and today my partner at Codurance), one of the best developers I've ever met and also one of my best friends. Mash was responsible for bringing me to UBS. Mash also brought in Balint Pato to one of our projects, and a strong bond was formed. I would like to thank Mash and Balint for all of our discussions, enthusiasm, passion, friendship, professionalism, and for teaching me so much. You guys are awesome and true craftsmen. Portia Tung was another great person I had the pleasure to meet and work with at UBS. She is probably the best Agile coach I've met. Her contagious passion in making our working environment *a place where we belong*, and her willingness to share her deep knowledge on how to make large organizations more agile, made my experience at UBS even better. Thanks, Portia, for your friendship and for teaching me how to bring people together. To Robert Taylor and Alexander "The Machine" Kikhtenko (a.k.a. Sasha), thank you for being amazing software craftsmen. It was an absolute pleasure to work with you guys.

David Green. Where do I start? He is a true craftsman, a friend, and one of the most talented developers I've ever met. I still remember all the awesome evenings we spent in the pub discussing our projects, craftsmanship, and how to make software development better. Dude, I learned a lot from you. David and I cofounded the London Software Craftsmanship Community (LSCC) and without him LSCC would probably not exist today.

In October 2013, I started Codurance, a consultancy company based on Software Craftsmanship principles and values, with Mashooq Badar. Codurance was a massive step forward in our careers. For the first time we have the opportunity to run a business the way we think a business should be run. This experience

would not be so great without Samir Talwar. Samir is not just the first craftsman we hired; he also is a friend and an extremely talented craftsman. Codurance and LSCC would never be the same without him.

Last but not least, I would like to thank all the passionate developers from the LSCC. I would never be able to learn so much in such a short period of time if not for your willingness to sacrifice your own personal time to share your knowledge with others. A big thanks to Gonçalo Silva, Samir Talwar, Tom Brand, Tom Westmacott, Emanuele Blanco, Carlos Fernandez Garcia, and Chris Jeffery for their amazing job organizing LSCC. LSCC would never be such a cool community without you.

In relation to this book, I would like to thank Micah Martin and Tyler Jennings for their contributions related to the history of Software Craftsmanship. Thanks, Kevlin Henney, for the great tips on how to better structure the book and for convincing me to throw away the original Chapter 1 and write a brand-new one. Thanks, Gianfranco Alongi and Mani Sarkar, for proofreading parts of the book. A special thanks to Samir Talwar and Andrew Parker for their amazing job finding hundreds of grammar mistakes and typos, and for their valuable suggestions on how to make the book better. There were also many other people who contributed to this book, sending me suggestions and typos. Thank you all.

About the Author

Sandro Mancuso has coded since a very young age but only started his professional career in 1996. He has worked for startups, software houses, product companies, international consultancy companies, and investment banks. In October 2013, Sandro cofounded Codurance, a consultancy company based on Software Craftsmanship principles and values.

During his career, Sandro has worked on various projects, with different languages and technologies and across many different industries. Sandro has a lot of experience bringing the Software Craftsmanship ideology and Extreme Programming practices to organizations of all sizes. Sandro is internationally renowned for his work in spreading Software Craftsmanship principles and is a renowned speaker at many conferences around the world. His professional aspiration is to raise the bar of the software industry by helping developers become better at—and care more about—their craft through sharing his knowledge, skills, and experiences.

Sandro's involvement with Software Craftsmanship started in 2010, when he founded the London Software Craftsmanship Community (LSCC), which has become the largest and most active Software Craftsmanship community in the world, with more than 2,000 craftsmen. For the past four years he has inspired and helped developers to start and organize many other Software Craftsmanship communities in Europe, the United States, and other parts of the world.

This page intentionally left blank

This page intentionally left blank

4 The Software Craftsmanship Attitude

If we think that a piece of code we wrote some time in the past is still good enough today, it means we didn't learn anything since.

For us, software craftsmen, caring about and being proud of the work we do is a given. Constantly finding ways to become better professionals is a lifetime commitment. Satisfying and helping our customers to achieve what they want to achieve, in the most efficient way, is our main goal. Sharing our knowledge with less-experienced software craftsmen is our moral obligation.

A few years ago, I was speaking to a colleague of mine. We had joined the company at roughly the same time, hired at the same position, and worked in a project together for almost one year. Since we were working for a consultancy company, after some time we went separate ways to work on different projects and for different clients. It was only after a few years that we had the opportunity to work together again. I then asked him how things were going with him and he said, "I don't really like this company. This company sucks." I was surprised by what he said since I had been really enjoying my time in that company. So then I asked him why he was saying that. "In all these years, they never bought me a book. Never sent me on a training course, and never gave me a project using modern technologies. They never gave me a promotion either," he said. "I haven't learned anything new for quite a long time," he complained.

I was not quite sure what to think or say about his comments. I had had two promotions during that period, worked on quite a few good projects, and learned many new things. "Who owns your career?" I suddenly asked him after a few seconds of awkward silence. He didn't quite understand my question and asked me to repeat it. "Who is in charge of your career and your professional future?" I asked him again. Even after a few years since we had this conversation, I still remember the puzzled look in his eyes.

In this chapter I discuss how we can own our careers, keep ourselves up to date, practice, and discover the things we didn't know. I also talk about how to create time for all these things.

Who Owns Your Career?

What if the company we work for does not buy us any books? What if the company never sent us to any training course or conferences? Would that be the only way we could learn anything new? Does it really mean the company is bad?

Imagine that you need a plumber to do some work in your house. Imagine that you need a lawyer to solve any legal issues, or a doctor when you are sick, or a dentist when you have an aching tooth. You normally go to these professionals when you have a problem, so now imagine them turning back to you and saying, "Could you buy me a book? Can you send me on a training course?" What would *you* think about that? To make things even worse, imagine that you, for some really bizarre reason, decide to buy them a book or send them on a training course, and once they acquired the knowledge you gave them, they come back and charge you for their services. How does it sound to you?

These professionals need to invest in their own careers so they can do a good job, satisfy their clients, and hopefully be referred to other clients. They use their own money and time to keep themselves current. Those that don't do it end up losing clients, receive fewer referrals, and will slowly be forced out of business.

On the other hand, factory workers, for example, rely on training. Factories need to train their employees to use new machines so they can do their mechanical and repetitive work well. However, factory workers have no say in what

machines the factory should buy or how they are going to do the work. They are just told what to do.

Clients pay professionals with the expectation of receiving a good service. They pay professionals to solve their problems in the best way possible. Clients don't pay professionals to learn. Clients pay professionals for their knowledge and skills. Professionals are expected to provide solutions, viable alternatives, and ideas to their clients. Professionals are expected to help their clients to achieve whatever they want to achieve in the best way possible, and that is how they build their reputation.

We all want to be treated and respected as software professionals but before we achieve that we need to start behaving like professionals. That means that we should use our own time and money to get better at what we do. We should own our own careers and be in control of what we learn and when we learn. We should be in a position that we can help our clients and employers to achieve their goals. Developers who rely only on their companies to provide them knowledge are not professional software developers. They are just factory workers in disguise.

"So companies should not be investing in their own people?" you may be asking. No, that is not what I meant. Companies should invest in their people but software professionals should not see that as their obligation. That should be seen as a bonus, a win-win situation. Companies that provide time to developers to get better at what they do are much smarter and can become far more efficient. Passionate developers will always favor these companies when choosing who they work for.

Our industry moves, possibly, faster than any other industry. Languages, frameworks, practices, and processes are all constantly evolving. Keeping up to date and constantly getting better at what you do is key for a successful career as a software craftsman.

EMPLOYER/EMPLOYEE RELATIONSHIP

In creative work, the employer/employee model is the wrong model. On one hand, we have a contractual model that states how people should be paid and that also states the legal and behavioral obligations that need to be respected by

employers and employees. On the other hand, we have the type of relationship that professionals have with their clients. The old top-down, command and control style of management became very popular during the Industrial Revolution and arguably still has its merits when the majority of the workforce is doing manual or repetitive work. However, it does not work well for creative workers. This style of management can be extremely damaging to the morale of software professionals, making the company far more inefficient. Companies that are still using this style of management normally struggle to hire talented professionals and are slowly losing the ones they still have. But, of course, this is just one side of the story. Keeping our heads down, working hard from 9 to 5, and doing only what we are told to do is not professional either. That is what factory workers do. If developers want to be treated as professionals, they should start acting as professionals. A software professional must, among other things, get involved and contribute to the business, provide options and solutions, and offer the best service when it comes to the implementation, technologies, and quality of what we produce.

The relationship between software craftsmen and their clients should be seen as a productive business partnership, regardless of which contractual model we may have. Being a permanent employee, a contractor, a consultant, or a developer working for an outsourcing company should not affect at all this relationship or the attitude we have toward our clients.

KEEPING OURSELVES UP TO DATE

> We live in a world where there is more and more information, and less and less meaning.
> —Jean Baudrillard

Different people have different learning patterns and preferences, and by no means do I feel that I could describe all the possible ways a person could learn. However, the following is a small list of things we can do to keep ourselves up to date.

BOOKS, MANY BOOKS

Having our own library, physical or electronic, is essential. We are very lucky to be in an industry where so much information is produced. However, there are many different types of books, and choosing which books to read can be a very difficult task.

- *Technology-specific books* are very valuable but they *expire.* They are essential for the immediate need, when we want to learn a framework, language, or any other software we need to use. They are great at giving us a deep understanding of how things work and the knowledge acquired can usually be used immediately. They are also great when we are planning our next career steps. They can give us details of how to use the technologies that may be between our current job and our desired job. However, many of the technology-specific books get old extremely quickly. When a new version of the technology they cover is released, or a different way of doing things becomes more popular, they will not add as much value as before. Examples would be books about Java, Hibernate, Node.js, or Clojure.
- *Conceptual books* are the books that give us the foundation to advance in our careers. They are the books where we get introduced to new concepts, paradigms, and practices. The knowledge we acquire through this type of book cannot always be applied immediately; it may take a significant amount of time to digest the information and become proficient. Quite often a technology or language may be used to explain some technical concepts but usually the knowledge we get can be applied broadly. Books covering topics like Test-Driven Development, Domain-Driven Design, object-oriented design, functional programming, or modeling different types of NoSQL databases, just to mention a few examples, would fit in this category. Learning new concepts, paradigms, and practices is far harder than learning a specific technology, and it may take years until we get comfortable with them. However, conceptual books are the books that give us the foundation to learn specific technologies much quicker.
- *Behavioral books* are the books that make us more efficient when working in teams and better professionals in general. They help us learn how to deal with people, clients, deadlines, team members, and so on. Knowing some programming languages, frameworks, and practices is not enough if we want to be good professionals. We also need to learn how to deal with everything else that is not related to code but is also part of a software project or organization. Books in this category will cover the more human and professional side of software development, including topics like Agile methodologies, Software Craftsmanship, Lean software development, psychology, philosophy, and management.
- *Revolutionary books* (some call them classics) are the ones that change the way we work or even some of our personal values. They propose a different set of

values and principles, quite often initially rejected or ignored by the majority of professionals. Bit by bit, they end up making their way into the mainstream. They are books that every software developer is expected to have read and are constantly mentioned in technical conversations. Very rarely does a technology-specific book become a classic. Normally the revolutionary books are conceptual, behavioral, or a combination. Books in this category define or have a great influence on the direction and evolution of our industry. A few examples would be *The Pragmatic Programmer, The Mythical Man-Month, Design Patterns (GoF), Test-Driven Development: By Example, Extreme Programming Explained: Embrace Change, The Clean Coder, Software Craftsmanship,* and *Refactoring.* It may take many years to master the content in the books in this category.

Books give us a deeper understanding of a technology or subject. Favor conceptual and behavioral books for your career progression, starting with the revolutionary ones. Read technology-specific books for your short- and medium-term plans.

Reading itself also has a learning curve. There are many different ways to read books, and understanding them can make a big difference in how fast we can read them and how much we can learn. This topic is beyond the scope of this book but I recommend you research speed-reading techniques.

BLOGS

Blogs are extremely popular and a great way to keep ourselves up to date. Quite a few very good developers I know and respect just read blogs. They have almost abandoned books. Blogs tend to fit well in the Software Craftsmanship and Agile models because they contain real experiences, personal findings, opinions, successes, and failures in short snippets. Reading blogs from more experienced professionals and subject matter experts is a good, quick, and free way for us to learn from many different great professionals at the same time. There are also great apps like Instapaper and Evernote, just to mention a few, that we can use to keep track of blogs.

Blogs can be dangerous for the uninformed though. The vast majority of blogs are written without much research or deep thought. Some blog posts are just a

brain dump of half-baked ideas, rants, or random thoughts. Some developers use their blogs to keep track of their own professional progression. Some report their own experiences in real projects but that does not necessarily mean they were able to solve their problems well or even to identify their real problems. And that is OK. That is exactly why blogs are great. As long as we understand that we need to read blogs with a pinch of salt, they are fantastic.

But do not think that just experienced professionals should write blogs. All software developers should have their own blogs, regardless of how much experience they have. We should all share our experiences and findings and help to create a great community of professionals. Sometimes we may think that we are not good enough or do not have much to say. We may think that we don't have an original idea and no one will read our blog anyway. First of all, we should treat our blog as a record of our own learning and progression—a history of our thoughts, ideas, and views of the world over our careers. We should not worry too much about what other people will think about it. We should first write it for ourselves. Even if developers more experienced than us have written about the subject many times before, it is worth writing whatever we are currently learning anyway. Every year there are thousands of new developers joining our industry and they will need to learn many of the things we are learning now. Maybe for them, our blogs will be very useful since we will be writing them from the perspective of a beginner. Do not worry about being judged by more senior developers because that is not going to happen. Whenever we Google for something and the first link we click leads to something we already know, we just jump to the next link. All developers should appreciate the effort that other developers make to write and share their views with the rest of the world, for free.

Technical Websites

Technical websites are also good in order to keep ourselves up to date with what's going on in the market. There are many websites that work as a digital magazine, announcing new trends and techniques. Some of these websites have technical writers writing for them every day. Some of them just aggregate the best blogs or provide a big discussion forum.

Know Who to Follow

In every profession, there is a group of people that contribute massively to move that profession forward. We have many of these people, some on the technology-specific side and others on the more generic, conceptual, and behavioral side. They are all important, so know who these people are. That helps us to filter the information we have online or in physical books. For example, if we work with Java or Ruby, we should know who publishes the best material on it. We should know who is helping the language to move forward or defining better ways to use it. We should know who the people reshaping our industry are. When we hear terms like Agile and Software Craftsmanship, we should try to discover who the people behind these ideas are. Look at their history. See what they've published. Try to understand where they got their inspiration or what they are basing their ideas on. We may discover that many of the things we talk about today go back a few decades.

Social Media

Learn how to use Twitter. Used wisely, Twitter can be a great tool for information gathering. Following the people described above and other fellow developers can be one of the best ways to keep ourselves up to date. Whenever someone publishes a blog post, they will tweet the link. Whenever someone finds some interesting material, they will also tweet the links. Sometimes we can follow quite a few interesting online conversations and that may be a good opportunity to join in.

Practice, Practice, Practice

How it is done is as important as getting it done. If we want to be good at writing quality code, we need to practice how to write quality code. There is no other way. When practicing, the focus should be on the techniques we are using and not in solving the problem. Solving problems is what we are paid for and what we do at work. However, when practicing, we should focus on *how* we are solving the problem. For example, imagine you are trying to learn or get better at Test-Driven Development. This is a hard discipline to master. It is counterintuitive at first, and that is why we need to practice.

Do you remember when you were learning how to drive? I remember that every time I was driving up a hill I was praying for the traffic light not to go red. I used to think to myself, "Damn, I'll need to stop, put it in first gear, release the clutch slowly, and be careful that the car does not stall or roll backward." I also remember the first time I had some friends in the car and they were asking me, "Can you switch the radio on?" and I, nervously glancing between all the mirrors, firmly gripping the steering wheel, would tell them "no." That happened because when I was learning how to drive, I was worried about the car. I was worried about the gear shifting. I was worried about the mirrors, other cars, driving straight and staying in the lane, not driving into the other lane when turning, and so on. I could not divert my attention to switch the radio on. Now, after a few years, imagine how you drive. We do not even remember we are in a car. We do not think about the mechanics of driving a car anymore. We just focus on where we are going, what we are going to do when we arrive there; we listen to music, sing along, have conversations while driving. Driving a car became second nature and the car is now almost an extension of our own bodies.

Technical practices and new technologies are the same thing. The more we practice, the more comfortable we become, to the point where we don't think about them anymore. We just focus on what we want to do with them.

When practicing, we need to focus on writing the best code we could possibly write. If it takes us minutes or even a few hours, to write a single test but it is the best test we could have written, that is OK. We should not worry if we take a long time to name a variable, method, or class. As long as we tried our best to find the most appropriate name for it, we should feel great. We are practicing and when we do it, we should strive for perfect practice. With this approach, when facing the demands of a real software project, we can concentrate on finding a good solution to the problem and not on how to write tests or which commands to use.

Katas

Katas are simple coding exercises. They have a simple problem domain that can be understood in a few minutes and are complicated enough not to be solved too quickly. They normally take from a few minutes to a few hours to be solved

and are a perfect way to try new techniques and approaches. An example would be a kata where we need to calculate the score of a bowling game or implement tic-tac-toe. We can use these katas to practice things we think we are not very good at, like Test-Driven Development, another language, or a framework.

Katas have been criticized by certain people in the Agile community, though. Some people say that it is plain stupid to do the same thing over and over again. Well, to a certain extent there is some degree of truth to it. Usually this is said because the term comes from martial arts and that is how it is done in karate, for example. We do the same movements over and over again. That was probably the original intent when we started using the term *kata* for a coding exercise. However, solving katas with our current toolkit (the techniques and tools that we are comfortable with) does not make a lot of sense.

When doing katas, the idea is that we stretch ourselves, using practices, techniques, and technologies that we are not very comfortable with, with the intent of getting better at them. After practicing these things quite a few times, you may feel ready or at least fairly comfortable with doing that in a professional environment. Think about musicians who practice for hours, days, and months before their live performances. That is exactly what we are trying to achieve here.

Katas can be very beneficial when the same kata is done over and over again but using a completely different approach or technique each time. That allows us to experiment and compare. The correct word to define this practice, according to martial arts, is *keiko.* However, as developers all over the world have already adopted the word *kata,* we should keep using it instead of introducing another term.

We can find a good source of coding katas at codingkata.org, codekata.pragprog.com, and kata.softwarecraftsmanship.org.

PET PROJECTS

Pet projects are for me, by far, the best way to learn and practice. A pet project is a *real* project but without pressure. There are no deadlines, they do not need to make money, you control the requirements, and, most important, you use the

technologies and methodologies you want, whenever you want, wherever you want. You are the boss. In real projects, a professional software developer would first understand the problem domain, then make technical decisions and write code. We would speak to stakeholders, product owners, users, business, marketing, and whoever else is involved and could contribute to the business idea. These conversations, in a healthy Agile environment, should happen frequently throughout the project. According to what the customer wants to achieve and the scope of the project, we would then choose the most suitable technologies to develop the project. Pet projects are exactly the opposite. First we decide what we want to learn—practices, disciplines, methodologies, and technologies—and then we find a problem to solve. It is much easier to understand why and how we can or cannot use certain technologies if we have a pet project to try them out. Pet projects allow us to play, experiment, discover, and learn in a very safe environment, giving us the experience we need to apply what we have learned to real projects.

Another important thing about pet projects is that we can experience several aspects of a real project. For example, we need to come up with an idea. Once we have an idea, we need to start refining the features and preparing a backlog. Preparing a backlog means thinking about priorities, making rough estimations, writing stories, and splitting tasks. We also need to think about tests, how we are going to deploy the application, version control, continuous integration, usability, the user interface, infrastructure code, design, and databases. As soon as we start using the application ourselves, we start changing our minds about the features (business) and also the technology choices. And that is exactly what happens in a real project. We do not need to have all that in place, but we can if we want to. We can use pet projects to learn any aspect of a real project. We can even try to write a business plan for it if we want to learn something about that. Remember, we are the bosses and we do whatever we want, as long as we are learning something. That's the whole point.

Above all, pet projects are meant to be fun. A common problem that developers have with pet projects is finding a good idea. The good news is that you do not need one. You are not starting a new business. I always advise developers to choose a subject that they are very passionate about. For example, if you like traveling, try to create a website about travel. If you like running, create an

application that can capture your progress, display graphics, and so forth. If you feel you should be more organized with your own tasks, create a mobile app where you can track what you need to do. It does not matter if there are thousands of other applications that do the same. There are always different things you would like them to do. The advantage of choosing a topic you are passionate about is that you will never run out of ideas for features or improvements. Besides that, you will always want to work on it since you usually will want to use it as well. All these things will help you to be quite close to a real project, and practicing them will have an enormous impact on your professional career.

A common question is if we should transform our pet projects into a real business. The answer is, "It depends." I normally would say *no* to that since if you want to have a business, writing code and learning new technologies should not be your number one priority. I would recommend that you find some good material and get yourself familiar with Lean startup concepts before writing a single line of code. The transition from a pet project to a real business can be extremely painful and full of disappointments. Believe me, I've tried it myself. Many of us get very attached to our own applications and code base, which can blind us to what the market really wants. If we feel we should transform our little Frankenstein pet project into a business, focus on the business first and be ready to throw whatever percentage of what you have written away if that is what the business demands. Detach yourself from the code, clean it up, and leave just the bare minimum to satisfy the business needs.

Open Source

Contributing to open source projects is also a great way to practice. There are thousands of them out there. Find a project that is related to what you want to learn or know more about, and download the source code. Start running and reading the tests, if any exist. Inspect, debug, and play with the code. If you want to contribute, start small: add some documentation, write some tests, check the list of bugs to be fixed or features to be implemented, pick a simple task, and give it a try. You can also propose and implement a new small feature to start with.

Open source projects tend to be very different from pet projects because they are usually, with few exceptions, very specific in scope. For example, it may be an

object-relational mapper (ORM) framework, a library to make web service calls, transaction management, a social network integrator, and so on. Although all these things are extremely important, necessary, and useful, open source projects tend to be just one of the libraries your professional applications will use, so do not forget to look at the whole picture.

Another great thing about open source projects is to see great developers in action. Looking at how they code and solve problems can be a great way to learn how to code well. Besides all that, it is also a great way to raise your public profile.

PAIR PROGRAMMING

Pair programming is more a social activity than a technical practice. It enhances the team spirit and brings developers together. Many developers are afraid to try it or think they will feel uncomfortable when pairing with others. Many years ago, I used to think like that but I realized that, in reality, I was afraid to expose my own limitations. If that is how you feel, the best advice is to get over it. There is a limit to what we can learn on our own. Of course we can learn whatever we want because we are all very smart people, but the problem is the amount of time that it can take. In addition, we will always have a naive and biased opinion when we are doing things on our own.

When pair programming, we can learn how to use a new language or parts of the application to which we had no previous exposure, a technical practice like Test-Driven Development, a few keyboard shortcuts, or even a completely new way to solve problems. Pairing can lead to very interesting discussions. It can help us to validate our own ideas or have them challenged, forcing us to rethink why we do things the way we do. It also can be a very humbling experience. Usually, we think of ourselves as very good developers, and we like to think that all the other developers are bad. Other developers write crap code, not us. When pairing, we get immediate feedback on our code and ideas. Our pair *validates* whatever we type immediately. Whenever our pair does not understand what we are doing, or does not agree with a variable name, the use of an application programming interface (API), or a design decision, we have an opportunity to step back and reevaluate the decision. Instead of thinking that the other person is stupid (which is rarely the case), we should think that we are

probably not as good as we think we are. A good developer is a developer who can write code that any other developer can understand. When our pairs don't agree or don't understand what we are doing, we should take this as an opportunity to have a good discussion. Use it to learn something new and open your mind to different approaches. If someone is questioning what we've just done, maybe it is because it's not good enough and there is a better way of doing it. We should take opportunities like that to share what we know, making everyone around us better. When teaching, we are forced to structure our thoughts, making us really understand our quite often half-baked ideas so we can make someone else understand them.

Pairing with someone from our team or a friend is great, but pairing with someone that we barely know can be even better. Usually team members and friends, after some time working and pairing together, develop a common understanding and style of coding. When pairing with people we have never paired with before, we end up potentially exposing ourselves to very different ways to solve and think about problems. The best way to find different pairing partners is attending meetings organized by our local user groups or technical communities. There are also an increasing number of developers willing to set up remote pair-programming sessions in whatever we want to work with. There are plenty of tools out there that can make a remote pairing session very smooth.

We need to keep our minds open to new ideas when pairing. Sometimes we learn, sometimes we teach, and sometimes we do both.

Socialize

> Not only individuals and interactions, but also a community of professionals.

The idea that software developers are introverted nerds is totally outdated. Finding other developers whom we can bounce ideas off of, pair-program, and network is almost essential for a successful career. A great and easy way to do that is to join your local user groups and technical communities and to participate in their events. They normally promote free talks, coding activities,

and social events. Another great aspect of being part of a community is the feeling that we are not alone. User groups and technical communities tend to be extremely open and welcoming. We find developers from many different backgrounds, working for completely different industries, with different technologies, languages, and processes. Some developers are more experienced than others but there is one thing they all share: passion. The great thing about passionate developers is that they are constantly learning and are very happy to share what they know.

Being a member of a local user group or technical community is a fantastic way to learn and share ideas.

Deliberate Discovery

> I'm the smartest man in Athens because I know that
> I know nothing.
> —Socrates

The biggest mistake that software professionals can make is not accepting that they don't know what they don't know. Not knowing what we don't know is also called *second-level ignorance*. Accepting that we have a lot to learn is a sign of maturity and one of the first steps toward mastery.

The vast majority of us have a natural tendency to be positive and optimistic. An example of that is how bad we are at estimating tasks. Once tasks are completed, if we compare the amount of time they took to our original estimations, we will see that the majority of them took longer than we expected. We need to accept that there is a massive chance that things will not go according to plan, which means there will be unforeseen and unpredictable problems. Unfortunately, we have absolutely no idea when, where, or how. The consequence of us ignoring this fact is that we will be caught by surprise and will not be able to handle the problems as well as we could if we knew about them upfront.

There is not a magical way to completely solve this problem but we can try to minimize it. One way of doing this is to constantly expose ourselves to situations

where we can learn something new about the context we are in. This is very important mainly in the early days of a project or before building a major set of new features—when we are most ignorant about what we need to do. Spending time trying to identify and minimize our ignorance across all the axes we can think of can be time extremely well spent.

Ignorance is a constant. Imagine we could start our latest project from scratch again. Same people, same requirements, same organizational constraints, same everything, but the difference this time is that we would start with all the knowledge we already have. How long do you think it would take? Now, stop and think about it. When asking this question, usually the answers average between one half and one quarter of the original time and that's where my own answer would be as well. Ignorance is the single greatest impediment to throughput, meaning that if we reduce the level of our ignorance as fast as we can, we can be far more productive and efficient.

We should always try to create opportunities where we can learn something we don't know. "But if I don't know what I don't know, how can I create opportunities to learn that?" you may ask. Speak to random colleagues and ask them how they keep up with the progress in our industry. Go to technical community and user group events. Show your code to other people. Ask for help even when you think you don't need it. Try to figure out which aspects of your current project you and your team have not explored yet, then start discussions about it or even write a proof of concept. Aiming to remove the ignorance constraint should always be your priority in order to deliver projects more efficiently and grow as professionals.

Work-Life Balance

So far I have been saying how important it is to look after our careers and that we should dedicate loads of time outside working hours to practice and learn. However, we all have family, friends, and other interests in life. I have a wife and two kids, whom I love to bits and want to spend a lot of quality time with. It is never easy to balance work and personal life because we spend more than 50 percent of our waking hours at work, including all the time commuting to and from it.

The most common thing I hear every time I talk about investing time in our own careers outside working hours is, "I don't have time." And if you say or agree with that, you are probably right. That's what you decided to believe and that, in turn, became your reality. But the truth is, we all have time. We are just not very good at optimizing it. Maybe we prefer to spend our time with something else that may or may not be as important as our careers.

Stop reading now and think about what you did yesterday, from the time you woke up to when you went to bed. What did you do the day before yesterday? Seriously. Stop and really think about that. Now think about how much of it was waste? How much of it was productive? By productive we can mean many things, from learning something new to spending time with our loved ones. If you took some time to rest, this is also important, especially after a busy day or a busy week. Our bodies need to recharge and we should cater to that as well. Some people genuinely do not have a lot of time. I once met a German speaker at a conference who told me he had a wife and five kids. Talking about learning and practicing outside working hours, he told me that his only alternative was to be very smart in how he uses the very limited amount of time he has. Despite his tough situation, he managed to present at a conference.

Creating Time

Quite often, lack of time is used as an excuse for our own laziness. People are different. Some live in large cities, some in small villages, some have family, and some live on their own. Some people have hobbies. Some love going to the gym. Some are young and love going out with friends; others are older and prefer to stay at home. Some are morning people; some go to bed very late and hate to wake up early. By no means do the following tips apply to everyone; they are just a collection of things that we can do to create time and use it to invest in our careers.

I realized that I used to waste a lot of time in front of the TV, aimlessly browsing the Internet, checking all the uninteresting things my friends publish on social networks, playing computer games, or watching sports. I decided to cut down on the number of hours spent on these things. That doesn't mean I don't do them anymore. It just means that I do them in moderation. Although they are great ways to relax and switch off, we do not need to do them every night, all night.

Your local coffee shop is your friend. Find a coffee shop near your workplace with a good Internet connection. If there isn't one, find one along the way. Even if your company has a coffee area, I would avoid that since we can be tempted to do work during this time or may be interrupted by colleagues. Take one day a week and try to get to a coffee shop one or two hours before you start working. Use this time to write code, read a technical book, blogs, or whatever you think you need to do to learn and move your career forward.

Lunchtime is another great opportunity we have to practice and learn something new. Once or twice a week just grab a sandwich and your laptop, and go somewhere quiet. It is amazing what we can do in such a small period of time when we are focused.

Check if there is a user group or tech community in your city and join them. Usually user groups meet regularly; many operate on a monthly schedule. Make a commitment to go to the meetings at least once or twice a month. Meetings tend to last between one to three hours so I'm sure the vast majority of us can plan for that. The advantage of attending user group meetings is that we usually can learn a huge amount in a very short period of time, either from a presentation or from speaking or coding with other developers. There is a limit to what we can learn on our own. Although we all feel we can learn anything, sometimes we don't know where to start. It's also worth remembering that it can be a lot faster to learn something from someone with more experience than trying to learn on our own.

Go to bed 30 minutes earlier than normal and use this time to read a book, look at blogs, or watch technical screencasts before falling asleep. This is something that works really well for me. Every single night I try to read at least a few pages, regardless of the time I go to bed.

Buy yourself a Kindle, iPad, or another e-book reader, and carry it with you at all times. Use it every time you have some dead time, like commuting or waiting for your dentist, doctor, or hairdresser appointment.

I believe every developer has a good smartphone today but if you do not have one, buy one now. Use Twitter or any other information aggregator when you

have a break or dead time. This is a great way to quickly read something or keep up with the latest news and trends. Being able to go to a single place and get all the information you want is key when you want to optimize time.

Focus: The Pomodoro Technique

In order to use our time outside working hours wisely, it is extremely important that we focus. A good technique is to decide beforehand what we want to do before we actually use this time slot. It is like deciding the agenda before scheduling a meeting. It does not need to be strict, but we need at least to have a good idea of what we want to achieve. Once this is done, we must ensure we can focus and get it done. One way we can achieve this is by using the Pomodoro technique. There are five basic steps to implement this technique:

1. Decide on the task to be done.
2. Set the Pomodoro (timer) to 25 minutes.
3. Work on the task until the timer rings.
4. Take a short break (normally 5 minutes).
5. Every four "Pomodoros," take a longer break (15–30 minutes).

During a Pomodoro (25 minutes), we focus on the task and nothing else. The breaks between Pomodoros are for a quick rest, coffee, checking emails, making a phone call, checking your Twitter, or whatever you feel like. We should do whatever we can to finish the Pomodoro with no interruptions but in case it needs to be interrupted (there is an important call we need to take or we really need to speak to someone), then the Pomodoro must be terminated and not paused. A new one should be created when we are ready to work on the task again. There are many Pomodoro tools available out there. Some are very sophisticated where you can keep track of all the tasks you completed, interrupted Pomodoros, and many other statistics. I, personally, prefer the simple ones but feel free to use one that suits you better.

Balance

Whatever you do, a sustainable pace is key. Keeping a healthy work-life balance is tough but not impossible. If you are the type of person that says, "I don't want to

touch a computer outside work," you probably should think again about your career choice; maybe software development is not for you. For the majority of us, software development—besides being our job—is also a hobby, which makes it relatively easy to find enough spare time to practice and better ourselves.

Keeping our professional life healthy is essential for a healthy family life. An unhealthy professional life, where we are constantly worried about not being paid well, or that we may not find another well-paid and interesting job if we are made redundant, may seriously damage our personal lives. Being at the top of our game, with good connections and with skills that are in demand in the market, puts us in a good position not to worry too much about our professional lives and give our family and friends the attention they very much deserve.

Summary

Owning our careers is hard. We need to put a lot of effort into getting to a position where we can say, "I feel confident I can find a good and well-paid job whenever I want." Determination and passion are essential for a successful career as a software craftsman. However, without focus, much of our efforts are wasted. We need to learn how to keep ourselves up to date and how to practice. We need to learn how to use our time well. The day we stop learning and practicing is the day we start losing control of our careers. The more knowledge and skills we have, the easier it is to own our careers—that means we are able to choose where, when, and for whom to work, and how much to charge for our services.

Time should never be used as an excuse for not doing certain things. Ever. We all have time. In fact, we all have exactly the same amount of time. The difference is how we choose to spend our time.

Index

P

S

T