1. Professionalism

"Oh laugh, Curtin, old boy. It's a great joke played on us by the Lord, or fate, or nature, whatever you prefer. But whoever or whatever played it certainly had a sense of humor! Ha!"

—*Howard*, The Treasure of the Sierra Madre

So, you want to be a professional software developer do you? You want to hold your head high and declare to the world: "I am a *professional!*" You want people to look at you with respect and treat you with deference. You want mothers pointing at you and telling their children to be like you. You want it all. Right?

Be Careful What You Ask For

Professionalism is a loaded term. Certainly it is a badge of honor and pride, but it is also a marker of responsibility and accountability. The two go hand in hand, of course. You can't take pride and honor in something that you can't be held accountable for.

It's a lot easier to be a nonprofessional. Nonprofessionals don't have to take responsibility for the job they do—they leave that to their employers.

If a nonprofessional makes an error, the employer cleans up the mess. But when a professional makes a mistake, *he* cleans up the mess.

What would happen if you allowed a bug to slip through a module, and it cost your company $10,000? The nonprofessional would shrug his shoulders, say "stuff happens," and start writing the next module. The professional would write the company a check for $10,000![1]

Yeah, it feels a little different when it's your own money, doesn't it? But that feeling is the feeling a professional has all the time. Indeed, that feeling is the essence of professionalism. Because, you see, professionalism is all about taking responsibility.

Taking Responsibility

You read the introduction, right? If not, go back and do so now; it sets the context for everything that follows in this book.

I learned about taking responsibility by suffering through the consequences of not taking it.

In 1979 I was working for a company named Teradyne. I was the "responsible engineer" for the software that controlled a mini- and microcomputer-based system that measured the quality of telephone lines. The central mini-computer was connected via 300-baud dedicated or dial-up phone lines to dozens of satellite micro-computers that controlled the measurement hardware. The code was all written in assembler.

Our customers were the service managers of major telephone companies. Each had the responsibility for 100,000 telephone lines or more. My system helped these service area managers find and repair malfunctions and problems in the telephone lines before their customers noticed them. This reduced the customer complaint rates that the public utility commissions measured and used to regulate the rates that the telephone companies could charge. In short, these systems were incredibly important.

Every night these systems ran through a "nightly routine" in which the central mini-computer told each of the satellite micro-computers to test every telephone line under their control. Each morning the central computer would pull back the list of faulty lines, along with their failing

characteristics. The service area managers would use this report to schedule repairmen to fix the faults before the customers could complain.

On one occasion I shipped a new release to several dozen customers. “Ship” is exactly the right word. I wrote the software onto tapes and shipped those tapes to the customers. The customers loaded the tapes and then rebooted the systems.

The new release fixed some minor defects and added a new feature that our customers had been demanding. We had told them we would provide that new feature by a certain date. I barely managed to overnight the tapes so that they’d arrive on the promised date.

Two days later I got a call from our field service manager, Tom. He told me that several customers had complained that the “nightly routine” had not completed, and that they had gotten no reports. My heart sank because in order to ship the software on time, I had neglected to test the routine. I had tested much of the other functionality of the system, but testing the routine took hours, and I needed to ship the software. None of the bug fixes were in the routine code, so I felt safe.

Losing a nightly report was a *big deal*. It meant that the repairmen had less to do and would be overbooked later. It meant that some customers might notice a fault and complain. Losing a night’s worth of data is enough to get a service area manager to call Tom and lambaste him.

I fired up our lab system, loaded the new software, and then started a routine. It took several hours but then it aborted. The routine failed. Had I run this test before I shipped, the service areas wouldn’t have lost data, and the service area managers wouldn’t be roasting Tom right now.

I phoned Tom to tell him that I could duplicate the problem. He told me that most of the other customers had called him with the same complaint. Then he asked me when I could fix it. I told him I didn’t know, but that I was working on it. In the meantime I told him that the customers should go back to the old software. He was angry at me saying that this was a double blow to the customers since they’d lost a whole night’s worth of data and couldn’t use the new feature they were promised.

The bug was hard to find, and testing took several hours. The first fix didn’t work. Nor did the second. It took me several tries, and therefore several days, to figure out what had gone wrong. The whole time, Tom was

calling me every few hours asking me when I'd have this fixed. He also made sure I knew about the earfuls he was getting from the service area managers, and just how embarrassing it was for him to tell them to put the old tapes back in.

In the end, I found the defect, shipped the new tapes, and everything went back to normal. Tom, who was not my boss, cooled down and we put the whole episode behind us. My boss came to me when it was over and said, "I bet you aren't going to do that again." I agreed.

Upon reflection I realized that shipping without testing the routine had been irresponsible. The reason I neglected the test was so I could say I had shipped on time. It was about me saving face. I had not been concerned about the customer, nor about my employer. I had only been concerned about my own reputation. I should have taken responsibility early and told Tom that the tests weren't complete and that I was not prepared to ship the software on time. That would have been hard, and Tom would have been upset. But no customers would have lost data, and no service managers would have called.

First, Do No Harm

So how do we take responsibility? There are some principles. Drawing from the Hippocratic oath may seem arrogant, but what better source is there? And, indeed, doesn't it make sense that the first responsibility, and first goal, of an aspiring professional is to use his or her powers for good?

What harm can a software developer do? From a purely software point of view, he or she can do harm to both the function and structure of the software. We'll explore how to avoid doing just that.

Do No Harm to Function

Clearly, we want our software to work. Indeed, most of us are programmers today because we got something to work once and we want that feeling again. But we aren't the only ones who want the software to work. Our customers and employers want it to work too. Indeed, they are paying us to create software that works just the way they want it to.

We harm the function of our software when we create bugs. Therefore, in order to be professional, we must not create bugs.

“But wait!” I hear you say. “That’s not reasonable. Software is too complex to create without bugs.”

Of course you are right. Software *is* too complex to create without bugs. Unfortunately that doesn’t let you off the hook. The human body is too complex to understand in it’s entirety, but doctors still take an oath to do no harm. If they don’t take themselves off a hook like *that,* how can we?

“Are you telling us we must be perfect?” Do I hear you object?

No, I’m telling you that you must be responsible for your imperfections. The fact that bugs will certainly occur in your code does not mean you aren’t responsible for them. The fact that the task to write perfect software is virtually impossible does not mean you aren’t responsible for the imperfection.

It is the lot of a professional to be accountable for errors even though errors are virtually certain. So, my aspiring professional, the first thing you must practice is apologizing. Apologies are necessary, but insufficient. You cannot simply keep making the same errors over and over. As you mature in your profession, your error rate should rapidly decrease towards the asymptote of zero. It won’t ever get to zero, but it is your responsibility to get as close as possible to it.

QA Should Find Nothing

Therefore, when you release your software you should expect QA to find no problems. It is unprofessional in the extreme to purposely send code that you know to be faulty to QA. And what code do you know to be faulty? Any code you aren’t *certain* about!

Some folks use QA as the bug catchers. They send them code that they haven’t thoroughly checked. They depend on QA to find the bugs and report them back to the developers. Indeed, some companies reward QA based on the number of bugs they find. The more bugs, the greater the reward.

Never mind that this is a desperately expensive behavior that damages the company and the software. Never mind that this behavior ruins schedules and undermines the confidence of the enterprise in the development team. Never mind that this behavior is just plain lazy and irresponsible. Releasing code to QA that you don’t know works is unprofessional. It violates the “do no harm” rule.

Will QA find bugs? Probably, so get ready to apologize—and then figure out why those bugs managed to escape your notice and do something to prevent it from happening again.

Every time QA, or worse a *user*, finds a problem, you should be surprised, chagrined, and determined to prevent it from happening again.

You Must *Know* It Works

How can you *know* your code works? That's easy. Test it. Test it again. Test it up. Test it down. Test it seven ways to Sunday!

Perhaps you are concerned that testing your code so much will take too much time. After all you've got schedules and deadlines to keep. If you spend all your time testing, you'll never get anything else written. Good point! So, automate your tests. Write unit tests that you can execute on a moment's notice, and run those tests as often as you can.

How much of the code should be tested with these automated unit tests? Do I really need to answer that question? All of it! All. Of. It.

Am I suggesting 100% test coverage? No, I'm not *suggesting* it. I'm *demanding* it. Every single line of code that you write should be tested. Period.

Isn't that unrealistic? Of course not. You only write code because you expect it to get executed. If you expect it to get executed, you ought to *know* that it works. The only way to know this is to test it.

I am the primary contributor and committer for an open source project called FITNESSE. As of this writing there are 60ksloc in FITNESSE. 26 of those 60 are written in 2000+ unit tests. Emma reports that the coverage of those 2000 tests is ~90%.

Why isn't my code coverage higher? Because Emma can't see all the lines of code that are being executed! I believe the coverage is much higher than that. Is the coverage 100%? No, 100% is an asymptote.

But isn't some code hard to test? Yes, but only because that code has been *designed* to be hard to test. The solution to that is to design your code to be *easy* to test. And the best way to do that is to write your tests first, before you write the code that passes them.

This is a discipline known as Test Driven Development (TDD), which we will say more about in a later chapter.

Automated QA

The entire QA procedure for FITNESSE is the execution of the unit and acceptance tests. If those tests pass, I ship. This means my QA procedure takes about three minutes, and I can execute it on a whim.

Now, it's true that nobody dies if there is a bug in FITNESSE. Nobody loses millions of dollars either. On the other hand, FITNESSE has many thousands of users, and a *very* small bug list.

Certainly some systems are so mission-critical that a short automated test is insufficient to determine readiness for deployment. On the other hand, you as a developer need a relatively quick and reliable mechanism to know that the code you have written works and does not interfere with the rest of the system. So, at the very least, your automated tests should tell you that the system is *very likely* to pass QA.

Do No Harm to Structure

The true professional knows that delivering function at the expense of structure is a fool's errand. It is the structure of your code that allows it to be flexible. If you compromise the structure, you compromise the future.

The fundamental assumption underlying all software projects is that software is easy to change. If you violate this assumption by creating inflexible structures, then you undercut the economic model that the entire industry is based on.

In short: *You must be able to make changes without exorbitant costs.*

Unfortunately, all too many projects become mired in a tar pit of poor structure. Tasks that used to take days begin to take weeks, and then months. Management, desperate to recapture lost momentum, hires more developers to speed things up. But these developers simply add to the morass, deepening the structural damage and raising the impediment.

Much has been written about the principles and patterns of software design that support structures that are flexible and maintainable.[2] Professional software developers commit these things to memory and strive to conform their software to them. But there's a trick to this that far too few software developers follow: *If you want your software to be flexible, you have to flex it!*

The only way to prove that your software is easy to change is to make easy changes to it. And when you find that the changes aren't as easy as you thought, you refine the design so that the next change is easier.

When do you make these easy changes? *All the time!* Every time you look at a module you make small, lightweight changes to it to improve its structure. Every time you read through the code you adjust the structure.

This philosophy is sometimes called *merciless refactoring*. I call it "the Boy Scout rule": Always check in a module cleaner than when you checked it out. Always make some random act of kindness to the code whenever you see it.

This is completely counter to the way most people think about software. They think that making a continuous series of changes to working software is *dangerous*. No! What is dangerous is allowing the software to remain static. If you aren't flexing it, then when you *do* need to change it, you'll find it rigid.

Why do most developers fear to make continuous changes to their code? They are afraid they'll break it! Why are they afraid they'll break it? Because they don't have tests.

It all comes back to the tests. If you have an automated suite of tests that covers virtually 100% of the code, and if that suite of tests can be executed quickly on a whim, then *you simply will not be afraid to change the code*. How do you prove you are not afraid to change the code? You change it all the time.

Professional developers are so certain of their code and tests that they are maddeningly casual about making random, opportunistic changes. They'll change the name of a class, on a whim. They'll notice a long-ish method while reading through a module and repartition it as a matter of course. They'll transform a switch statement into polymorphic deployment, or collapse an inheritance hierarchy into a chain-of-command. In short, they treat software the way a sculptor treats clay—they continuously shape and mold it.

Work Ethic

Your career is *your* responsibility. It is not your employer's responsibility to make sure you are marketable. It is not your employer's

responsibility to train you, or to send you to conferences, or to buy you books. These things are *your* responsibility. Woe to the software developer who entrusts his career to his employer.

Some employers are willing to buy you books and send you to training classes and conferences. That's fine, they are doing you a favor. But never fall into the trap of thinking that this is your employer's responsibility. If your employer doesn't do these things for you, you should find a way to do them yourself.

It is also not your employer's responsibility to give you the time you need to learn. Some employers may provide that time. Some employers may even demand that you take the time. But again, they are doing you a favor, and you should be appropriately appreciative. Such favors are not something you should expect.

You owe your employer a certain amount of time and effort. For the sake of argument, let's use the U.S. standard of 40 hours per week. These 40 hours should be spent on *your employer's* problems, not on *your* problems.

You should plan on working 60 hours per week. The first 40 are for your employer. The remaining 20 are for you. During this remaining 20 hours you should be reading, practicing, learning, and otherwise enhancing your career.

I can hear you thinking: "But what about my family? What about my life? Am I supposed to sacrifice them for my employer?"

I'm not talking about *all* your free time here. I'm talking about 20 extra hours per week. That's roughly three hours per day. If you use your lunch hour to read, listen to podcasts on your commute, and spend 90 minutes per day learning a new language, you'll have it all covered.

Do the math. In a week there are 168 hours. Give your employer 40, and your career another 20. That leaves 108. Another 56 for sleep leaves 52 for everything else.

Perhaps you don't want to make that kind of commitment. That's fine, but you should not then think of yourself as a professional. Professionals spend *time* caring for their profession.

Perhaps you think that work should stay at work and that you shouldn't bring it home. I agree! You should not be working for your employer during those 20 hours. Instead, you should be working on your career.

Sometimes these two are aligned with each other. Sometimes the work you do for your employer is greatly beneficial to your career. In that case, spending some of that 20 hours on it is reasonable. But remember, those 20 hours are for *you.* They are to be used to make yourself more valuable as a professional.

Perhaps you think this is a recipe for burnout. On the contrary, it is a recipe to *avoid* burnout. Presumably you became a software developer because you are passionate about software and your desire to be a professional is motivated by that passion. During that 20 hours you should be doing those things that *reinforce* that passion. Those 20 hours should be *fun!*

Know Your Field

Do you know what a Nassi-Schneiderman chart is? If not, why not? Do you know the difference between a Mealy and a Moore state machine? You should. Could you write a quicksort without looking it up? Do you know what the term "Transform Analysis" means? Could you perform a functional decomposition with Data Flow Diagrams? What does the term "Tramp Data" mean? Have you heard the term "Conascence"? What is a Parnas Table?

A wealth of ideas, disciplines, techniques, tools, and terminologies decorate the last fifty years of our field. How much of this do you know? If you want to be a professional, you should know a sizable chunk of it and constantly be increasing the size of that chunk.

Why should you know these things? After all, isn't our field progressing so rapidly that all these old ideas have become irrelevant? The first part of that query seems obvious on the surface. Certainly our field is progressing and at a ferocious pace. Interestingly enough, however, that progress is in many respects peripheral. It's true that we don't wait 24 hours for compile turnaround any more. It's true that we write systems that are gigabytes in size. It's true that we work in the midst of a globe-spanning network that provides instant access to information. On the other hand, we are writing

the same `if` and `while` statements that we were writing 50 years ago. Much has changed. Much has not.

The second part of the query is certainly not true. Very few ideas of the past 50 years have become irrelevant. Some have been sidelined, it's true. The notion of doing waterfall development has certainly fallen into disfavor. But that doesn't mean we shouldn't know what it is, and what its good and bad points are.

Overall, however, the vast majority of the hard-won ideas of the last 50 years are as valuable today as they were then. Perhaps they are even more valuable now.

Remember Santayana's curse: "Those who cannot remember the past are condemned to repeat it."

Here is a *minimal* list of the things that every software professional should be conversant with:

- Design patterns. You ought to be able to describe all 24 patterns in the GOF book and have a working knowledge of many of the patterns in the POSA books.
- Design principles. You should know the SOLID principles and have a good understanding of the component principles.
- Methods. You should understand XP, Scrum, Lean, Kanban, Waterfall, Structured Analysis, and Structured Design.
- Disciplines. You should practice TDD, Object-Oriented design, Structured Programming, Continuous Integration, and Pair Programming.
- Artifacts: You should know how to use: UML, DFDs, Structure Charts, Petri Nets, State Transition Diagrams and Tables, flow charts, and decision tables.

Continuous Learning

The frenetic rate of change in our industry means that software developers must continue to learn copious quantities just to keep up. Woe to the architects who stop coding—they will rapidly find themselves irrelevant. Woe to the programmers who stop learning new languages—they will watch as the industry passes them by. Woe to the developers who

fail to learn new disciplines and techniques—their peers will excel as they decline.

Would you visit a doctor who did not keep current with medical journals? Would you hire a tax lawyer who did not keep current with the tax laws and precedents? Why should employers hire developers who don't keep current?

Read books, articles, blogs, tweets. Go to conferences. Go to user groups. Participate in reading and study groups. Learn things that are outside your comfort zone. If you are a .NET programmer, learn Java. If you are a Java programmer, learn Ruby. If you are a C programmer, learn Lisp. If you want to really bend your brain, learn Prolog and Forth!

Practice

Professionals practice. True professionals work hard to keep their skills sharp and ready. It is not enough to simply do your daily job and call that practice. Doing your daily job is performance, not practice. Practice is when you specifically exercise your skills *outside* of the performance of your job for the sole purpose of refining and enhancing those skills.

What could it possibly mean for a software developer to practice? At first thought the concept seems absurd. But stop and think for a moment. Consider how musicians master their craft. It's not by performing. It's by practicing. And how do they practice? Among other things, they have special exercises that they perform. Scales and etudes and runs. They do these over and over to train their fingers and their mind, and to maintain mastery of their skill.

So what could software developers do to practice? There's a whole chapter in this book dedicated to different practice techniques, so I won't go into much detail here. One technique I use frequently is the repetition of simple exercises such as the `Bowling Game` or `Prime Factors`. I call these exercises *kata*. There are many such kata to choose from.

A kata usually comes in the form of a simple programming problem to solve, such as writing the function that calculates the prime factors of an integer. The point of doing the kata is not to figure out how to solve the problem; you know how to do that already. The point of the kata is to train your fingers and your brain.

I'll do a kata or two every day, often as part of settling in to work. I might do it in Java, or in Ruby, or in Clojure, or in some other language for which I want to maintain my skills. I'll use the kata to sharpen a particular skill, such as keeping my fingers used to hitting shortcut keys, or using certain refactorings.

Think of the kata as a 10-minute warm-up exercise in the morning and a 10-minute cool-down in the evening.

Collaboration

The second best way to learn is to collaborate with other people. Professional software developers make a special effort to program together, practice together, design and plan together. By doing so they learn a lot from each other, and they get more done faster with fewer errors.

This doesn't mean you have to spend 100% of your time working with others. Alone time is also very important. As much as I like to pair program with others, it makes me crazy if I can't get away by myself from time to time.

Mentoring

The best way to learn is to teach. Nothing will drive facts and values into your head faster and harder than having to communicate them to people you are responsible for. So the benefit of teaching is strongly in favor of the teacher.

By the same token, there is no better way to bring new people into an organization than to sit down with them and show them the ropes. Professionals take personal responsibility for mentoring juniors. They will not let a junior flail about unsupervised.

Know Your Domain

It is the responsibility of every software professional to understand the domain of the solutions they are programming. If you are writing an accounting system, you should know the accounting field. If you are writing a travel application, you should know the travel industry. You don't have to be a domain expert, but there is a reasonable amount of due diligence that you ought to engage in.

When starting a project in a new domain, read a book or two on the topic. Interview your customer and users about the foundation and basics of the domain. Spend some time with the experts, and try to understand their principles and values.

It is the worst kind of unprofessional behavior to simply code from a spec without understanding why that spec makes sense to the business. Rather, you should know enough about the domain to be able to recognize and challenge specification errors.

Identify with Your Employer/Customer

Your employer's problems are *your* problems. You need to understand what those problems are and work toward the best solutions. As you develop a system you need to put yourself in your employer's shoes and make sure that the features you are developing are really going to address your employer's needs.

It is easy for developers to identify with each other. It's easy to fall into an *us versus them* attitude with your employer. Professionals avoid this at all costs.

Humility

Programming is an act of creation. When we write code we are creating something out of nothing. We are boldly imposing order upon chaos. We are confidently commanding, in precise detail, the behaviors of a machine that could otherwise do incalculable damage. And so, programming is an act of supreme arrogance.

Professionals know they are arrogant and are not falsely humble. A professional knows his job and takes pride in his work. A professional is confident in his abilities, and takes bold and calculated risks based on that confidence. A professional is not timid.

However, a professional also knows that there will be times when he will fail, his risk calculations will be wrong, his abilities will fall short; he'll look in the mirror and see an arrogant fool smiling back at him.

So when a professional finds himself the butt of a joke, he'll be the first to laugh. He will never ridicule others, but will accept ridicule when it is deserved and laugh it off when it's not. He will not demean another for making a mistake, because he knows he may be the next to fail.

A professional understands his supreme arrogance, and that the fates will eventually notice and level their aim. When that aim connects, the best you can do is take Howard's advice: Laugh.

Bibliography

[PPP2001]: Robert C. Martin, *Principles, Patterns, and Practices of Agile Software Development*, Upper Saddle River, NJ: Prentice Hall, 2002.