6. Practicing

All professionals practice their art by engaging in skill-sharpening exercises. Musicians rehearse scales. Football players run through tires. Doctors practice sutures and surgical techniques. Lawyers practice arguments. Soldiers rehearse missions. When performance matters, professionals practice. This chapter is all about the ways in which programmers can practice their art.

Some Background on Practicing

Practicing is not a new concept in software development, but we didn't recognize it as practicing until just after the turn of the millennium. Perhaps the first formal instance of a practice program was printed on page 6 of [K&R-C].

```
main()
{
  printf("hello, world\n");
}
```

Who among us has not written that program in one form or another? We use it as a way to prove a new environment or a new language. Writing and

executing that program is proof that we can write and execute *any* program.

When I was much younger, one of the first programs I would write on a new computer was `SQINT`, the squares of integers. I wrote it in assembler, BASIC, FORTRAN, COBOL, and a zillion other languages. Again, it was a way to prove that I could make the computer do what I wanted it to do.

In the early ’80s personal computers first started to show up in department stores. Whenever I passed one, like a VIC-20 or a Commodore-64, or a TRS-80, I would write a little program that printed an infinite stream of ‘\’ and ‘/’ characters on the screen. The patterns this program produced were pleasing to the eye and looked far more complex than the little program that generated them.

Although these little programs were certainly practice programs, programmers in general did not *practice*. Frankly, the thought never occurred to us. We were too busy writing code to think about practicing our skills. And besides, what would have been the point? During those years programming did not require quick reactions or nimble fingers. We did not use screen editors until the late ’70s. We spent much of our time waiting for compiles or debugging long, horrid stretches of code. We had not yet invented the short-cycles of TDD, so we did not require the fine-tuning that practice could bring.

Twenty-Two Zeros

But things have changed since the early days of programming. Some things have changed a *lot*. Other things haven’t changed much at all.

One of the first machines I ever wrote programs for was a PDP-8/I. This machine had a 1.5-microsecond cycle time. It had 4,096 12-bit words in core memory. It was the size of a refrigerator and consumed a significant amount of electrical power. It had a disk drive that could store 32K of 12-bit words, and we talked to it with a 10-character-per-second teletype. We thought this was a *powerful* machine, and we used it to work miracles.

I just bought a new Macbook Pro laptop. It has a 2.8GHz dual core processor, 8GB of RAM, a 512GB SSD, and a 17-inch 1920 × 1200 LED screen. I carry it in my backpack. It sits on my lap. It consumes less than 85 watts.

My laptop is eight thousand times faster, has two million times more memory, has sixteen million times more offline storage, requires 1% of the power, takes up 1% of the space, and costs one twenty-fifth of the price of the PDP-8/I. Let's do the math:

$$8{,}000 \times 2{,}000{,}000 \times 16{,}000{,}000 \times 100 \times 100 \times 25 = 6.4 \times 10^{22}$$

This number is *large*. We're talking about *22 orders of magnitude!* That's how many angstroms there are between here and Alpha Centauri. That's how many electrons there are in a silver dollar. That's the mass of the Earth in units of Michael Moore. This is a big, big, number. And it's sitting in my lap, and probably yours too!

And what am I doing with this increase in power of 22 factors of ten? I'm doing pretty much what I was doing with that PDP-8/I. I'm writing *if* statements, *while* loops, and *assignments*.

Oh, I've got better tools to write those statements with. And I have better languages to write those statements with. But the nature of the statements hasn't changed in all that time. Code in 2010 would be recognizable to a programmer from the 1960s. The clay that we manipulate has not changed much in those four decades.

Turnaround Time

But the *way* we work has changed dramatically. In the '60s I could wait a day or two to see the results of a compile. In the late '70s a 50,000-line program might take 45 minutes to compile. Even in the '90s, long build times were the norm.

Programmers today don't wait for compiles.[1] Programmers today have such immense power under their fingers that they can spin around the red-green-refactor loop in seconds.

For example, I work on a 64,000-line Java project named FITNESSE. A full build, including *all* unit and integration tests, executes in less than 4 minutes. If those tests pass, I'm ready to ship the product. *So the whole QA process, from source code to deployment, requires less than 4 minutes.* Compiles take almost no measurable time at all. Partial tests require *seconds*. So I can literally spin around the compile/test loop *ten times per minute*!

It's not always wise to go that fast. Often it is better to slow down and just *think*.[2] But there are other times when spinning around that loop as fast as possible is *highly* productive.

Doing anything quickly requires practice. Spinning around the code/test loop quickly requires you to make very quick decisions. Making decisions quickly means being able to recognize a vast number of situations and problems and simply *know* what to do to address them.

Consider two martial artists in combat. Each must recognize what the other is attempting and respond appropriately within milliseconds. In a combat situation you don't have the luxury of freezing time, studying the positions, and deliberating on the appropriate response. In a combat situation you simply have to *react*. Indeed, it is your *body* that reacts while your mind is working on a higher-level strategy.

When you are spinning around the code/test loop several times per minute, it is your *body* that knows what keys to hit. A primal part of your mind recognizes the situation and reacts within milliseconds with the appropriate solution while your mind is free to focus on the higher-level problem.

In both the martial arts case and the programming case, speed depends on *practice*. And in both cases the practice is similar. We choose a repertoire of problem/solution pairs and execute them over and over again until we know them cold.

Consider a guitarist like Carlos Santana. The music in his head simply comes out his fingers. He does not focus on finger positions or picking technique. His mind is free to plan out higher-level melodies and harmonies while his body translates those plans into lower-level finger motions.

But to gain that kind of ease of play requires *practice*. Musicians practice scales and études and riffs over and over until they know them cold.

The Coding Dojo

Since 2001 I have been performing a TDD demonstration that I call *The Bowling Game*.[3] It's a lovely little exercise that takes about thirty minutes.

It experiences conflict in the design, builds to a climax, and ends with a surprise. I wrote a whole chapter on this example in [PPP2003].

Over the years I performed this demonstration hundreds, perhaps thousands, of times. I got *very* good at it! I could do it in my sleep. I minimized the keystrokes, tuned the variable names, and tweaked the algorithm structure until it was just right. Although I didn't know it at the time, this was my first kata.

In 2005 I attended the XP2005 Conference in Sheffield, England. I attended a session with the name *Coding Dojo* led by Laurent Bossavit and Emmanuel Gaillot. They had everyone open their laptops and code along with them as they used TDD to write Conway's *Game of Life*. They called it a "Kata" and credited "Pragmatic" Dave Thomas[4] with the original idea.[5]

Since then many programmers have adopted a martial arts metaphor for their practice sessions. The name Coding Dojo[6] seems to have stuck. Sometimes a group of programmers will meet and practice together just like martial artists do. At other times, programmers will practice solo, again as martial artists do.

About a year ago I was teaching a group of developers in Omaha. At lunch they invited me to join their Coding Dojo. I watched as twenty developers opened their laptops and, keystroke by keystroke, followed along with the leader who was doing *The Bowling Game* Kata.

There are several kinds of activities that take place in a dojo. Here are a few:

Kata

In martial arts, a *kata* is a precise set of choreographed movements that simulates one side of a combat. The goal, which is asymptotically approached, is perfection. The artist strives to teach his body to make each movement perfectly and to assemble those movements into fluid enactment. Well-executed kata are beautiful to watch.

Beautiful though they are, the purpose of learning a kata is not to perform it on stage. The purpose is to train your mind and body how to react in a particular combat situation. The goal is to make the perfected

movements automatic and instinctive so that they are there when you need them.

A programming kata is a precise set of choreographed keystrokes and mouse movements that simulates the solving of some programming problem. You aren't actually solving the problem because you already know the solution. Rather, you are practicing the movements and decisions involved in solving the problem.

The asymptote of perfection is once again the goal. You repeat the exercise over and over again to train your brain and fingers how to move and react. As you practice you may discover subtle improvements and efficiencies either in your motions or in the solution itself.

Practicing a suite of katas is a good way to learn hot keys and navigation idioms. It is also a good way to learn disciplines such as TDD and CI. But most importantly, it is a good way to drive common problem/solution pairs into your subconscious, so that you simply know how to solve them when facing them in real programming.

Like any martial artist, a programmer should know several different kata and practice them regularly so that they don't fade away from memory. Many kata are recorded at http://katas.softwarecraftsmanship.org. Others can be found at http://codekata.pragprog.com. Some of my favorites are:

- ***The Bowling Game:***http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata
- ***Prime Factors:***http://butunclebob.com/ArticleS.UncleBob.ThePrimeFactorsKata
- ***Word Wrap:***http://thecleancoder.blogspot.com/2010/10/craftsman-62-dark-path.html

For a real challenge, try learning a kata so well that you can set it to music. Doing this well is *hard*.[7]

Wasa

When I studied jujitsu, much of our time in the dojo was spent in pairs practicing our *wasa*. Wasa is very much like a two-man kata. The routines are precisely memorized and played back. One partner plays the role of the aggressor, and the other partner is the defender. The motions are repeated over and over again as the practitioners swap roles.

Programmers can practice in a similar fashion using a game known as *ping-pong*.[8] The two partners choose a kata, or a simple problem. One programmer writes a unit test, and then the other must make it pass. Then they reverse roles.

If the partners choose a standard kata, then the outcome is known and the programmers are practicing and critiquing each other's keyboarding and mousing techniques, and how well they've memorized the kata. On the other hand, if the partners choose a new problem to solve, then the game can get a bit more interesting. The programmer writing a test has an inordinate amount of control over how the problem will be solved. He also has a significant amount of power to set constraints. For example, if the programmers choose to implement a sort algorithm, the test writer can easily put constraints on speed and memory space that will challenge his partner. This can make the game quite competative . . . and fun.

Randori

Randori is free-form combat. In our jujitsu dojo, we would set up a variety of combat scenarios and then enact them. Sometimes one person was told to defend, while each of the rest of us would attack him in sequence. Sometimes we would set two or more attackers against a single defender (usually the sensei, who almost always won). Sometimes we'd do two on two, and so forth.

Simulated combat does not map well to programming; however, there is a game that is played at many coding dojos called randori. It is very much like two-man wasa in which the partners are solving a problem. However, it is played with many people and the rules have a twist. With the screen projected on the wall, one person writes a test and then sits down. The next person makes the test pass and then writes the next test. This can be done in sequence around the table, or people can simply line up as they feel so moved. In either case these exercises can be a *lot* of fun.

It is remarkable how much you can learn from these sessions. You can gain an immense insight into the way other people solve problems. These insights can only serve to broaden your own approach and improve your skill.

Broadening Your Experience

Professional programmers often suffer from a lack of diversity in the kinds of problems that they solve. Employers often enforce a single language, platform, and domain in which their programmers must work. Without a broadening influence, this can lead to a very unhealthy narrowing of your resume and your mindset. It is not uncommon for such programmers to find themselves unprepared for the changes that periodically sweep the industry.

Open Source

One way to stay ahead of the curve is to do what lawyers and doctors do: Take on some pro-bono work by contributing to an open-source project. There are lots of them out there, and there is probably no better way to increase your repertoire of skills than to actually work on something that someone else cares about.

So if you are a Java programmer, contribute to a Rails project. If you write a lot of C++ for your employer, find a Python project and contribute to it.

Practice Ethics

Professional programmers practice on their own time. It is not your employer's job to help you keep your skills sharp for you. It is not your employer's job to help you keep your resume tuned. Patients do not pay doctors to practice sutures. Football fans do not (usually) pay to see players run through tires. Concert-goers do not pay to hear musicians play scales. And employers of programmers don't have to pay you for your practice time.

Since your practice time is your own time, you don't have to use the same languages or platforms that you use with your employer. Pick any language you like and keep your polyglot skills sharp. If you work in a .NET shop, practice a little Java or Ruby at lunch, or at home.

Conclusion

In one way or another, *all* professionals practice. They do this because they care about doing the best job they possibly can. What's more, they practice on their own time because they realize that it is their responsibility—and not their employer's—to keep their skills sharp. Practicing is what you do when you *aren't* getting paid. You do it so that you *will* be paid, and paid *well*.

Bibliography

[K&R-C]: Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Upper Saddle River, NJ: Prentice Hall, 1975.

[PPP2003]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003.