14. Mentoring, Apprenticeship, and Craftsmanship

I have been consistently disappointed by the quality of CS graduates. It's not that the graduates aren't bright or talented, it's just that they haven't been taught what programming is really all about.

Degrees of Failure

I once interviewed a young woman who was working on her master's degree in computer science for a major university. She was applying for a summer intern position. I asked her to write some code with me, and she said "I don't really write code."

Please read the previous paragraph again, and then skip over this one to the next.

I asked her what programming courses she had taken in pursuit of her master's degree. She said that she hadn't taken any.

Maybe you'd like to start at the beginning of the chapter just to be sure you haven't fallen into some alternate universe or have just awakened from a bad dream.

At this point you might well be asking yourself how a student in a CS master's program can avoid a programming course. I wondered the same

thing at the time. I'm still wondering today.

Of course, that's the most extreme of a series of disappointments I've had while interviewing graduates. Not all CS graduates are disappointing—far from it! However, I've noticed that those who aren't have something in common: Nearly all of them *taught themselves to program* before they entered university and continued to teach themselves despite university.

Now don't get me wrong. I think it is possible to get an excellent education at a university. It's just that I also think it's possible to wiggle yourself through the system and come out with a diploma, and not much else.

And there's another problem. Even the best CS degree programs do not typically prepare the young graduate for what they will find in industry. This is not an indictment of the degree programs so much as it is the reality of nearly all disciplines. What you learn in school and what you find on the job are often very different things.

Mentoring

How do we learn how to program? Let me tell you my story about being mentored.

Digi-Comp I, My First Computer

In 1964 my mother gave me a little plastic computer for my twelfth birthday. It was called a Digi-Comp I.[1] It had three plastic flip-flops and six plastic *and*-gates. You could connect the outputs of the flip-flops to the inputs of the *and*-gates. You could also connect the output of the *and*-gates to the inputs of the flip-flops. In short, this allowed you to create a three-bit finite state machine.

The kit came with a manual that gave you several programs to run. You programmed the machine by pushing little tubes (short segments of soda straws) onto little pegs protruding from the flip flops. The manual told you exactly where to put each tube, but not what the tubes *did*. I found this very frustrating!

I stared at the machine for hours and determined how it worked at the lowest level; but I could not, for the life of me, figure out how to make it do what I wanted it to do. The last page in the manual told me to send in a

dollar and they would send back a manual telling me how to program the machine.[2]

I sent in my dollar and waited with the impatience of a twelve year old. The day the manual arrived I devoured it. It was a simple treatise on boolean algebra covering basic factoring of boolean equations, associative and distributive laws, and DeMorgan's theorem. The manual showed how to express a problem in terms of a sequence of boolean equations. It also described how to reduce those equations to fit into 6 *and*-gates.

I conceived of my first program. I still remember the name: Mr. Patternson's Computerized Gate. I wrote the equations, reduced them, and mapped them to the tubes and pegs of the machine. *And it worked!*

Writing those three words just now sent chills down my spine. The same chills that coursed down that twelve year old nearly half a century ago. I was hooked. My life would never be the same.

Do you remember the moment your first program worked? Did it change your life or set you on a course you could not turn away from?

I did not figure it all out for myself. I was *mentored*. Some very kind and very adept people (to whom I owe a huge debt of gratitude) took the time to write a treatise on boolean algebra that was accessible to a twelve year old. They connected the mathematical theory to the pragmatics of the little plastic computer and empowered me to make that computer do what I wanted it to do.

I just pulled down my copy of that fateful manual. I keep it in a zip-lock bag. Nevertheless, the years have taken their toll by yellowing the pages and making them brittle. Still, the power of the words shines out of them. The elegance of their description of boolean algebra consumed three sparse pages. Their step-by-step walk-through of the equations for each of the original programs is still compelling. It was a work of mastery. It was a work that changed at least one young man's life. Yet I doubt I'll never know the names of the authors.

The ECP-18 in High School

At the age of fifteen, as a freshman in high school, I liked hanging out in the math department. (Go figure!) One day they wheeled in a machine

the size of a table saw. It was an educational computer made for high schools, called the ECP-18. Our school was getting a two-week demo.

I stood in the background as the teachers and technicians talked. This machine had a 15-bit word (*what's a word?*) and a 1024-word drum memory. (I knew what drum memory was by then, but only in concept.)

When they powered it up, it made a whining sound reminiscent of a jet aircraft taking off. I guessed that was the drum spinning up. Once up to speed, it was relatively quiet.

The machine was *lovely*. It was essentially an office desk with a marvelous control panel protruding from the top like the bridge of a battleship. The control panel was adorned with rows of lights that were also push-buttons. Sitting at that desk was like sitting in Captain Kirk's chair.

As I watched the technicians push those buttons, I noted that they lit up when pushed, and that you could push them again to turn them off. I also noted that there were other buttons they were pushing; buttons with names like *deposit* and *run*.

The buttons in each row were grouped into five clusters of three. My Digi-Comp was also three bits, so I could read an octal digit when expressed in binary. It was not a big leap to realize that these were just five octal digits.

As the technicians pushed the buttons I could hear them mutter to themselves. They would push 1, 5, 2, 0, 4, in the *memory buffer* row while saying to themselves, "store in 204." They would push 1, 0, 2, 1, 3 and mutter, "load 213 into the *accumulator*." There was a row of buttons named *accumulator!*

Ten minutes of that and it was pretty clear to my fifteen-year-old mind that the 15 meant *store* and the 10 meant *load*, that the accumulator was what was being stored or loaded, and that the other numbers were the numbers of one of the 1024 words on the drum. (So *that's* what a word is!)

Bit by bit (no pun intended) my eager mind latched on to more and more instruction codes and concepts. By the time the technicians left, I knew the basics of how that machine worked.

That afternoon, during a study hall, I crept into the math lab and started fiddling with the computer. I had learned long ago that it is better to ask forgiveness than permission! I toggled in a little program that would multiply the accumulator by two and add one. I toggled a 5 into the accumulator, ran the program, and saw 13_8 in the accumulator! It had worked!

I toggled in several other simple programs like that and they all worked as planned. I was master of the universe!

Days later I realized how stupid, and lucky, I had been. I found an instruction sheet laying around in the math lab. It showed all the different instructions and op-codes, including many I had not learned by watching the technicians. I was gratified that I had interpreted those that I knew correctly and thrilled by the others. However, one of the new instructions was HLT. It just so happened that the *halt* instruction was a word of all zeros. And it just so happened that I had put a word of all zeros at the end of each of my programs so that I could load it into the accumulator to clear it. The concept of a halt simply had not occurred to me. I just figured the program would stop when it was done!

I remember at one point sitting in the math lab watching one of the teachers struggle to get a program working. He was trying to type two numbers in decimal on the attached teletype, and then print out the sum. Anyone who has tried to write a program like this in machine language on a mini-computer knows that it is far from trivial. You have to read in the characters, convert them to digits, then to binary, add them, convert back to decimal and encode back into characters. And, believe me, it's a *lot* worse when you are entering the program in binary through the front panel!

I watched as he put a halt into his program and then ran it until it stopped. (Oh! That's a good idea!) This primitive breakpoint allowed him to examine the contents of the registers to see what his program had done. I remember him muttering, "Wow, that was fast!" Boy, do I have news for him!

I had no idea what his algorithm was. That kind of programming was still magic to me. And he never spoke to me while I watched over his shoulder. Indeed, *nobody* talked to me about this computer. I think they

considered me a nuisance to be ignored, fluttering around the math lab like a moth. Suffice it to say that neither the student nor the teachers had developed a high degree of social skill.

In the end he got his program working. It was amazing to watch. He'd slowly type in the two numbers because, despite his earlier protestation, that computer was *not* fast (think of reading consecutive words from a spinning drum in 1967). When he hit return after the second number, the computer blinked ferociously for a bit and then started to print the result. It took about one second per digit. It printed all but the last digit, blinked even more ferociously for five seconds, and then printed the final digit and halted.

Why that pause before the last digit? I never found out. But it made me realize that the approach to a problem can have a profound effect on the user. Even though the program produced the correct answer, there was *still* something wrong with it.

This was mentoring. Certainly it was not the kind of mentoring I could have hoped for. It would have been nice if one of those teachers had taken me under his wing and worked with me. But it didn't matter, because I was *observing* them and learning at a furious pace.

Unconventional Mentoring

I told you those two stories because they describe two very different kinds of mentoring, neither of which are the kind that the word usually implies. In the first case I learned from the authors of a very well-written manual. In the second case I learned by observing people who were actively trying to ignore me. In both cases the knowledge gained was profound and foundational.

Of course, I had other kinds of mentors too. There was the kindly neighbor who worked at Teletype who brought me home a box of 30 telephone relays to play with. Let me tell you, give a lad some relays and a electric train transformer and he can conquer the world!

There was the kindly neighbor who was a ham operator who showed me how to use a multimeter (which I promptly broke). There was the office supply store owner who allowed me to come in and "play" with his very expensive programmable calculator. There was the Digital Equipment

Corporation sales office that allowed me to come in and "play" with their PDP-8 and PDP-10.

Then there was big Jim Carlin, a BAL programmer who saved me from being fired from my first programming job by helping me debug a Cobol program that was way beyond my depth. He taught me how to read core dumps, and how to format my code with appropriate blank lines, rows of stars, and comments. He gave me my first push towards craftsmanship. I'm sorry I could not return the favor when the boss's displeasure fell on him a year later.

But, frankly, that's about it. There just weren't that many senior programmers in the early seventies. Everywhere else I worked, I *was* senior. There was nobody to help me figure out what true professional programming was. There was no role model who taught me how to behave or what to value. Those things I had to learn for myself, and it was by no means easy.

Hard Knocks

As I told you before, I did, in fact, get fired from that factory automation job in 1976. Although I was technically very competent, I had not learned to pay attention to the business or the business goals. Dates and deadlines meant nothing to me. I forgot about a big Monday morning demo, left the system broken on Friday, and showed up late on Monday with everyone staring angrily at me.

My boss sent me a letter warning me that I had to make changes immediately or be fired. This was a significant wake-up call for me. I reevaluated my life and career and started to make some significant changes in my behavior—some of which you have been reading about in this book. But it was too little, too late. The momentum was all in the wrong direction and small things that wouldn't have mattered before became significant. So, though I gave it a hardy try, they eventually escorted me out of the building.

Needless to say, it's not fun to bring that kind of news home to a pregnant wife and a two-year old daughter. But I picked myself up and took some powerful life lessons to my next job—which I held for fifteen years and which formed the true foundation of my current career.

In the end, I survived and prospered. But there has to be a better way. It would have been far better for me if I'd had a true mentor, someone to teach me the in's and out's. Someone I could have observed while I helped him with small tasks, and who would review and guide my early work. Someone to act as a role model and teach me appropriate values and reflexes. A sensei. A master. A mentor.

Apprenticeship

What do doctors do? Do you think hospitals hire medical graduates and throw them into operating rooms to do heart surgery on their first day on the job? Of course not.

The medical profession has developed a discipline of intense mentoring ensconced in ritual and lubricated with tradition. The medical profession oversees the universities and makes sure the graduates have the best education. That education involves roughly *equal amounts* of classroom study and clinical activity in hospitals working with professionals.

Upon graduation, and before they can be licensed, the newly minted doctors are required to spend a year in supervised practice and training called internship.

This is intense on-the-job training. The intern is surrounded by role models and teachers.

Once internship has been completed each of the medical specialties requires three to five more years of further supervised practice and training known as residency. The resident gains confidence by taking on ever greater responsibilities while still being surrounded by, and supervised by, senior doctors.

Many specialties require yet another one to three years of fellowship in which the student continues specialized training and supervised practice.

And *then* they are eligible to take their exams and become board certified.

This description of the medical profession was somewhat idealized, and probably wildly inaccurate. But the fact remains that when the stakes are high, we do not send graduates into a room, throw meat in occasionally, and expect good things to come out. So why do we do this in software?

It's true that there are relatively few deaths caused by software bugs. But there *are* significant monetary losses. Companies lose huge amounts of money due to the inadequate training of their software developers.

Somehow the software development industry has gotten the idea that programmers are programmers, and that once you graduate you can code. Indeed, it is not at all uncommon for companies to hire kids right out of school, form them into "teams," and ask them to build the most critical systems. It's insane!

Painters don't do this. Plumbers don't. Electricians don't. Hell, I don't even think short-order cooks behave this way! It seems to me that companies who hire CS graduates ought to invest more in their training than McDonalds invests in their servers.

Let's not kid ourselves that this doesn't matter. There's a lot at stake. Our civilization runs on software. It is software that moves and manipulates the information that pervades our daily life. Software controls our automobile engines, transmissions, and brakes. It maintains our bank balances, sends us our bills, and accepts our payments. Software washes our clothes and tells us the time. It puts pictures on the TV, sends our text messages, makes our phone calls, and entertains us when we are bored. It's everywhere.

Given that we entrust software developers with all aspects of our lives, from the minutia to the momentous, I suggest that a reasonable period of training and supervised practice is not inappropriate.

Software Apprenticeship

So how *should* the software profession induct young graduates into the ranks of professionalism? What steps should they follow? What challenges should they meet? What goals should they achieve? Let's work it backwards.

Masters

These are programmers who have taken the lead on more than one significant software project. Typically they will have 10+ years of experience and will have worked on several different kinds of systems, languages, and operating systems. They know how to lead and coordinate multiple teams, are proficient designers and architects, and can code

circles around everyone else without breaking a sweat. They have been offered management positions, but have either turned them down, have fled back after accepting them, or have integrated them with their primarily technical role. They maintain that technical role by reading, studying, practicing, doing, and *teaching*. It is to a master that the company will assign technical responsibility for a project. Think, "Scotty."

Journeymen

These are programmers who are trained, competent, and energetic. During this period of their career they will learn to work well in a team and to become team leaders. They are knowledgeable about current technology but typically lack experience with many diverse systems. They tend to know one language, one system, one platform; but they are learning more. Experience levels vary widely among their ranks, but the average is about five years. On the far side of that average we have burgeoning masters; on the near side we have recent apprentices.

Journeymen are supervised by masters, or other more senior journeymen. Young journeymen are seldom allowed autonomy. Their work is closely supervised. Their code is scrutinized. As they gain in experience, autonomy grows. Supervision becomes less direct and more nuanced. Eventually it transitions into peer review.

Apprentices/Interns

Graduates start their careers as apprentices. Apprentices have no autonomy. They are very closely supervised by journeymen. At first they take no tasks at all, they simply provide assistance to the journeymen. This should be a time of very intense pair-programming. This is when disciplines are learned and reinforced. This is when the foundation of values is created.

Journeymen are the teachers. They make sure that the apprentices know design principles, design patterns, disciplines, and rituals. Journeymen teach TDD, refactoring, estimation, and so forth. They assign reading, exercises, and practices to the apprentices; they review their progress.

Apprenticeship ought to last a year. By that time, if the journeymen are willing to accept the apprentice into their ranks, they will make a recommendation to the masters. The masters should examine the

apprentice both by interview and by reviewing their accomplishments. If the masters agree, then the apprentice becomes a journeyman.

The Reality

Again, all of this is idealized and hypothetical. However, if you change the names and squint at the words you'll realize that it's not all that different from the way we *expect* things to work now. Graduates are supervised by young team-leads, who are supervised by project-leads, and so on. The problem is that, in most cases, this supervision *is not technical!* In most companies there is no technical supervision at all. Programmers get raises and eventual promotions because, well, that's just what you do with programmers.

The difference between what we do today and my idealized program of apprenticeship is the focus on technical teaching, training, supervision, and review.

The difference is the very notion that professional values and technical acumen must be taught, nurtured, nourished, coddled, and encultured. What's missing from our current sterile approach is the responsibility of the elders to teach the young.

Craftsmanship

So now we are in a position to define this word: *craftsmanship*. Just what is it? To understand, let's look at the word *craftsman*. This word brings to mind skill and quality. It evokes experience and competence. A craftsman is someone who works quickly, but without rushing, who provides reasonable estimates and meets commitments. A craftsman knows when to say no, but tries hard to say yes. A craftsman is a professional.

Craftsmanship is the *mindset* held by craftsmen. Craftsmanship is a meme that contains values, disciplines, techniques, attitudes, and answers.

But how do cratftsmen adopt this meme? How do they attain this mindset?

The craftsmanship meme is handed from one person to another. It is taught by elders to the young. It is exchanged between peers. It is observed and relearned, as elders observe the young. Craftsmanship is a contagion, a

kind of mental virus. You catch it by observing others and allowing the meme to take hold.

Convincing People

You can't convince people to be craftsmen. You can't convince them to accept the craftsmanship meme. Arguments are ineffective. Data is inconsequential. Case studies mean nothing. The acceptance of a meme is not so much a rational decision as an emotional one. This is a very *human* thing.

So how do you get people to adopt the craftsmanship meme? Remember that a meme is contagious, but only if it can be observed. So you make the meme *observable*. You act as a role model. You become a craftsman first, and let your craftsmanship show. Then just let the meme do the rest of the work.

Conclusion

School can teach the theory of computer programming. But school does not, and cannot teach the discipline, practice, and skill of being a craftsman. Those things are acquired through years of personal tutelage and mentoring. It is time for those of us in the software industry to face the fact that guiding the next batch of software developers to maturity will fall to us, not to the universities. It's time for us to adopt a program of apprenticeship, internship, and long-term guidance.