IV

Craftmanship

The following is an abridged and edited excerpt from my book Clean Craftsmanship.

The profession of software began, inauspiciously, in the summer of 1935 when Alan Turing began work on his paper. His goal was to resolve a mathematical dilemma that had perplexed mathematicians for a decade or more: *the Entscheidungsproblem*. The decision problem.

In that goal, he was successful; but he had no idea, at the time, that his paper would spawn a globe-spanning industry upon which we would all depend and that now forms the lifeblood of our entire civilization.

In 1945, Turing wrote code for the Automatic Computing Engine (ACE). He wrote this code in binary machine language, using base 32 numbers. Fewer than a dozen people had ever written code like this before, so Turing

had to invent and implement concepts like subroutines, stacks, and floating-point numbers.

“A Great Number”

After several months of inventing the basics and using them to solve mathematical problems, he wrote a report that stated the following conclusions.

“We shall need a great number of mathematicians of ability, because there will probably be a good deal of work of this kind to be done.”

“A great number.” How did he know? In reality, he had no idea just how prescient that statement was; he could not have envisioned the great number we have now.

But what was that other thing he said? “Mathematicians of ability.” Do you consider yourself to be a mathematician of ability? Turing did.

In the same report, he went on to write:

“One of our difficulties will be the maintenance of an appropriate discipline, so that we do not lose track of what we are doing.”

Discipline! Alan Turing looked forward through eight decades and saw that our problem would be discipline. And with that, he laid the first stone in the framework of software professionalism. He said that we should be mathematicians of ability who maintain an appropriate discipline.

Is that who we are? Is that who you are?

Eight Decades

One person’s lifetime. That’s how old our profession is as of this writing. And what has happened in those fourscore years?

In 1945, there were a handful of computers in the world, and an equally small number of programmers. In computer science terms, this is on the

order of one: O(1). These numbers grew rapidly in those first years. But let's use this as our origin.

1945. Computers: O(1). Programmers O(1).

In the decade that followed, the reliability, consistency, and power usage of vacuum tubes improved dramatically. This made it possible for larger and more powerful computers to be built.

By 1960, IBM had sold 140 of its 700 series computers. These were huge, expensive behemoths that could only be afforded by the military, the government, and very large corporations. They were, by our standards, slow, impoverished, and fragile.

It was in the '50s that Grace Hopper invented the concept of a higher-level language and coined the term *compiler*. By 1960, her work led to COBOL.

In 1952, John Backus submitted the FORTRAN specification. This was followed rapidly by the development of ALGOL. By 1958, John McCarthy had developed LISP. So the proliferation of the language zoo had begun.

In those days, there were no operating systems, no frameworks, no function libraries. If something executed on your computer, it was because you wrote it. So, in those days, it took a staff of a dozen or more programmers just to keep one computer running.

By 1960, 15 years after Turing, there were O(100) computers in the world. The number of programmers was an order of magnitude greater: O(1000).

Who were these programmers? They were people like Grace Hopper, Edsger Dijkstra, John von Neumann, John Backus, and Jean Jennings. They were scientists and mathematicians and engineers. Most were people who already had careers and already understood the businesses and disciplines they were employed by. Many, if not most, were in their 30s, 40s, and 50s.

The 1960s was the decade of the transistor. Bit by bit these small, simple, inexpensive, and reliable devices replaced the vacuum tube. And the effect on computers was a game changer.

By 1965, IBM had produced over 10,000 transistor-based 1401 computers. They rented for about $2,500 per month, putting them within the reach of thousands of medium-sized businesses.

These machines were programmed in Assembler, FORTRAN, COBOL, and RPG. And all those companies who rented those machines needed staffs of programmers to write their applications.

IBM wasn't the only company making computers at the time; so we'll just say that by 1965, there were O(1E4) computers in the world. And if each computer needed ten programmers to keep it running, there must have been O(1E5) programmers.

Twenty years after Turing, there must have been several hundred thousand programmers in the world. Where did these programmers come from? There weren't enough mathematicians, scientists, and engineers to cover the need. And there weren't any computer science graduates coming out of the universities, because there weren't any computer science degree programs—anywhere.

So companies drew from the best and brightest of their accountants, clerks, planners—anyone with some proven technical aptitude. And they found lots.

And, again, these were people who were already professionals in another field. They were in their 30s, 40s, and 50s. They already understood deadlines and commitments, what to leave in, what to leave out.[1] And, although these people were not mathematicians per se, they were disciplined professionals. Turing would likely have approved.

---

1. Apologies to Bob Seger.

But the crank kept on turning. By 1966, IBM was producing 1,000 System/360s every month. These computers were popping up everywhere. They were immensely powerful for the day. The model 30 could address 64K bytes of memory and execute 35,000 instructions per second.

It was during this period, in the mid-’60s, when Ole-Johan Dahl and Kristen Nygaard invented SIMULA 67, the first object-oriented language.

It was also during this period that Edsger Dijkstra invented structured programming.

And it was also during this time that Ken Thompson and Dennis Ritchie invented C and Unix.

And still the crank turned. In the early ’70s, the integrated circuit came into regular use. These little chips could hold dozens, hundreds, even thousands of transistors. They allowed electronic circuits to be massively miniaturized.

And so the minicomputer was born.

In the late ’60s and into the ’70s, Digital Equipment Corporation sold 50,000 PDP8 systems and hundreds of thousands of PDP11 systems.

And they weren’t alone! The minicomputer market exploded. By the mid-’70s, there were dozens and dozens of companies selling minicomputers.

And so by 1975, 30 years after Turing, there were O(1E6) computers in the world. And how many programmers were there? The ratio was starting to change. The number of computers was approaching the number of programmers. So, by 1975, there were O(1E6) programmers.

Where did these millions of programmers come from? Who were they?

They were me. Me and my buddies. Me and my cohort of young, energetic, geeky boys.

Tens of thousands of new EE and CS grads—we were all young. We were all smart. We, in the United States, were all concerned about the draft. And we were almost all male.

Oh, it’s not that women were leaving the field in any number—yet. That didn’t start until the mid-’80s. No, it’s just that many more boys (and we were boys) were entering the field.

In my first job as a programmer, in 1969, there were a couple of dozen programmers. They were all in their 30s or 40s, and one-third to one-half were women.

Ten years later, I was working at a company with about fifty programmers, and perhaps three were women.

So, 30 years after Turing, the demographics of programming had shifted dramatically toward very young men. Hundreds of thousands of twenty-something males.

We were typically *not* what Turing would have described as disciplined mathematicians.

But businesses had to have programmers. The demand was through the roof. And what very young men lack in discipline, they make up for with energy.

And we were also cheap. Despite the high starting salaries of programmers today, back then companies could pick up programmers pretty inexpensively. My starting salary in 1969 was $7,200 per year.

And this has been the trend ever since. Young men and women have been pouring out of computer science programs every year; and industry seems to have an insatiable appetite for them.

In the 30 years between 1945 and 1975, the number of programmers grew by at least a factor of a million. In the 40 years since then, that growth rate has slowed a bit, but it is still very high.

How many programmers do you think are in the world this year, 2025? If you include the VBA programmers, I think the number must be O(1E8). There must be hundreds of millions of programmers in the world today.

This is clearly exponential growth. Exponential growth curves have a doubling rate. You can do the math. Hey, Albert, what's the doubling rate if we go from 1 to 1E8 in 80 years?

The log to the base 2 of one hundred million is approximately 27—divide that into 80 and you get about 2.96. So perhaps the number of

programmers doubled roughly every three-ish years.

Actually, as we saw earlier, the rate was higher in the first decades and has slowed down a bit now. My guess is that the doubling rate is about five years. Every five years the number of programmers in the world doubles.

The implications of that hypothesis are staggering. If the number of programmers in the world doubles every five years, it means that half the programmers in the world have less than five years of experience; and this will always be true so long as that doubling rate continues. This leaves the programming industry in the precarious position of—perpetual inexperience.

Nerds and Saviors

Perpetual inexperience. Oh, don't worry, this doesn't mean that *you* are perpetually inexperienced. It just means that once you gain five years of experience, the number of programmers will have doubled. By the time you gain ten years of experience, the number of programmers will have quadrupled.

Perhaps you think AI will slow this rate. Don't be deluded. Every new software technology that was supposed to slow that rate has only increased it. Pick a new software technology. FORTRAN, C, Unix, OO, C++, Java, C#, Ruby, Functional … Not one of them decreased the demand for new programmers. If anything, the demand accelerated because each of those technologies made the field accessible to larger and larger groups of people —mostly young.

People look at the number of young people in programming and conclude that it's a young person's profession. They ask, "Where are all the old people?"

We're all still here! We haven't gone anywhere. There just weren't that many of us 50 years ago.

The problem is that there aren't enough of us old guys to teach the new programmers coming in. For every programmer with 30 years of experience, there are ~63 programmers who need to learn something from her (or him); 32 of whom are brand new.

So, perpetual inexperience, with insufficient mentors to correct the problem. And so the same old mistakes get repeated over, and over, and over again.

Notoriety

But something else has happened in the last 80 years. Programmers have gained something that I am sure Alan Turing never anticipated: notoriety.

Back in the ’50s and ’60s, nobody knew what a programmer was. There weren’t enough of us to have a social impact. Programmers did not live next door to very many people.

That started to change in the ’70s. By then, fathers were advising their sons (and sometimes their daughters) to get degrees in computer science. There were enough programmers in the world so that everybody knew somebody who knew one. And the image of the nerdy, Twinkie-eating geek was born.

Few people had seen a computer; but virtually everyone had heard about them. Computers showed up in TV shows, like *Star Trek*, and in movies, like *2001: A Space Odyssey* and *Colossus: The Forbin Project*. All too often in those shows, the computers themselves were cast as villains.

Notice, however, that in each of these cases, the programmer is not a significant character. Society didn’t know what to make of programmers back then. They were shadowy, hidden, and somehow insignificant compared to the machines themselves.

I have fond memories of one television commercial from this era. A wife and her husband, a nerdy little guy with glasses, a pocket protector, and a calculator, were comparing prices at a grocery store. Mrs. Olsen described him as “a computer genius” and proceeded to school the wife and husband alike on the benefits of a particular brand of coffee.

The computer programmer in that commercial was naive, bookish, and inconsequential. Someone smart, but with no wisdom or common sense. Not someone you’d invite to parties.

Indeed, computer programmers were seen as the kind of people who got beaten up a lot at school.

By 1983, personal computers started to appear; and it was clear that teenagers were interested in them for lots of reasons. By this time, a rather large number of people knew at least one computer programmer. We were considered professionals; but still mysterious.

That year, the movie *War Games* depicted a young Matthew Broderick as a computer-savvy teenager and hacker. He hacks into WOPR, the US weapons control system, thinking it's a video game and starts the countdown to thermonuclear war. At the end of the movie, he saves the world by convincing the computer that the only winning move is not to play.

The computer and the programmer had switched roles. Now the computer was the childlike, naive character and the programmer the conduit, if not the source, of wisdom.

We saw something similar in 1986 in the movie *Short Circuit* in which the computerized robot known as #5 is childlike and innocent but learns wisdom with the help of its bumbling creator and his girlfriend.

By 1993, things had changed dramatically. In the film *Jurassic Park*, the programmer was the villain and the computer was not a character at all. It was just a tool.

Society was beginning to understand who we were and the role we played. We had graduated from nerd to teacher to villain in just 20 years.

But the vision changed again. In the 1999 film *The Matrix*, the main characters were both programmers and saviors. Indeed, their godlike powers came from their ability to read and understand "the code."

Our roles were changing fast. Villain to savior in just a few years. Society at large was beginning to understand the power we have for both good and evil.

Role Models and Villains

Fifteen years later, in 2014, I visited the Mojang office in Stockholm to give lectures on clean code and test-driven development (TDD).

Mojang, in case you didn't know, is the company that produced the game *Minecraft*.

Afterward, since the weather was nice, the Mojang programmers and I sat outside at a beer garden, chatting.

All of a sudden a young boy, perhaps 12, ran up to the fence and called out to one of the programmers: "Are you Jeb?"

He was referring to Jens Bergensten, one of the lead programmers at Mojang.

The lad asked Jens for his autograph and peppered him with questions. He had eyes for no one else.

Programmers have become role models and idols for our children. They dream of growing up to be like Jeb, or Dinnerbone, or Notch.

Programmers, real-life programmers, are heroes.

But where there are heroes, there are also villains.

In October of 2015, Michael Horn, the CEO of Volkswagen North America, testified before the Congress of the United States regarding the software in their cars that was cheating the EPA testing devices.

When asked why the company did this, he blamed programmers. He said, "This was a couple of software engineers who put this in for whatever reasons."

Of course he was lying about the "whatever reasons." He knew perfectly well what the reasons were, and so did the Volkswagen company at large. His feeble attempt to shift blame onto the programmers was pretty transparent.

On the other hand, he was exactly right. It *was* some programmers who wrote that lying, cheating code.

And those programmers, whoever they were, gave us all a bad name. If we had a true professional organization, their recognition as programmers

would, and should, be revoked. They betrayed us all. They besmirched the honor of our profession.

And so we've graduated. It's taken 80 years. But we've gone from nothing, to nerds, to role models, to heroes and villains in that time.

Society has begun—just begun—to understand who we are and the threats and promises that we represent. (For more on that, see the section "Why Should We Be Clean?" in Chapter 1, "Clean Code.")

We Rule the World

But society doesn't understand everything yet. Indeed, neither do we. As I told you in the introduction, you and I, we programmers; we rule the world.

That may seem like an exaggerated statement. But consider. There are more computers in the world right now than there are people. And the computers that outnumber us perform a myriad of essential tasks for us.

They keep track of our reminders. They manage our calendars. They deliver our Facebook messages and keep our photo albums. They connect our phone calls and deliver our text messages.

They control the engines in our cars, as well as the brakes, the accelerator, and sometimes even the steering wheels.

We can't cook without them. We can't wash clothes without them. They keep our houses warm in winter. They sweep and mop our floors. They entertain us when we're bored. They keep track of our bank records and our credit cards. They help us pay our bills.

In fact, most people in the western world interact with some software system every waking minute of every day. Some even continue interacting while they sleep.

The point is that *nothing* happens in our society without software. No product gets bought or sold. No law gets enacted or enforced. No car drives. No Amazon products get delivered. No phone connects. No power comes out of outlets. No food gets delivered to stores. No water come out

of faucets. No gas gets piped to furnaces. Without software monitoring and coordinating it all.

And *WE* write that software. And that makes *us* the rulers of the world.

Oh, other people think they make the rules—but then they hand those rules to us, and *WE* write the rules that execute in the machines that monitor and coordinate every aspect of our lives.

Society does not quite understand this yet. Not quite. Not yet. But the day is coming soon when our society will understand it all too well.

We, programmers, don't quite understand this yet either. Not really. But again, the day is coming when it will be savagely driven home to us.

Catastrophes

We've seen plenty of software catastrophes over the years. Some have been pretty spectacular.

Between 2018 and 2019, two 737 MAX aircraft dove headlong into the ground at near the speed of sound, killing 346 people. The reason: a software weakness exposed by a hardware failure.

In 2016, we lost the Schiaparelli Mars lander and rover due to a software issue that caused the lander to believe it had already landed, when it was actually nearly 4 kilometers above the surface.

In 1999, we lost the Mars Climate Orbiter due to a ground-based software error that transmitted data to the orbiter using English units (pound-seconds) rather than metric units (newton-seconds). This error caused the orbiter to descend too far into the Martian atmosphere, where it was torn to pieces.

In 1996, the Ariane 5 launch vehicle and payload were destroyed 37 seconds after launch due to an integer overflow exception when a 64-bit floating-point number underwent an unchecked conversion to a 16-bit integer. The exception crashed the onboard computers, and the vehicle self-destructed.

Should we talk about the Therac-25 radiation therapy machine that, due to a race condition, killed three people and injured three others by blasting them with a high-powered electron beam?

Or maybe we should talk about Knight Capital, who lost $460 million in 45 minutes because they reused a flag that activated dead code inadvertently left in the system.

Or perhaps we should talk about the Toyota stack overflow bug that caused cars to accelerate out of control—killing perhaps as many as 89 people.

Or maybe we should talk about Healthcare.gov and the software failure that nearly overturned a new and controversial US healthcare law.

These disasters have cost billions of dollars and many lives. And they were caused by programmers.

We, programmers, through the code that we write, are killing people.

Now I know you didn't get into this business in order to kill people. Probably you became a programmer because you wrote an infinite loop that printed your name once, and you experienced that joyous feeling of power.

But facts are facts. We are now in a position in our society where our actions can destroy fortunes, livelihoods, and lives.

One day, probably not to long from now, some poor programmer is going to do something just a little dumb, and tens of thousands of people will die.

This isn't wild speculation—it's just a matter of time.

And when this happens, the politicians of the world will demand an accounting. They will demand that we show them how we will prevent this in the future.

And if we show up without a statement of ethics; if we show up without any standards, or defined disciplines; if we show up and whine about how our bosses set unreasonable schedules and deadlines …

Then we will be found GUILTY.

The Oath

And so, to begin the definition of our ethics as software developers, I offer the following oath.

In order to defend and preserve the honor of the profession of computer programmers, I promise that, to the best of my ability and judgment:

1. I will not produce harmful code.
2. The code that I produce will always be my best work. I will not knowingly allow code that is defective either in behavior or in structure to accumulate.
3. I will produce, with each release, a quick, sure, and repeatable demonstration that every element of the code works as it should.
4. I will make frequent, small releases so that I do not impede the progress of others.
5. I will fearlessly and relentlessly improve my creations at every opportunity. I will never degrade them.
6. I will do all that I can to keep the productivity of myself and others as high as possible. I will do nothing that decreases that productivity.
7. I will continuously ensure that others can cover for me and that I can cover for them.
8. I will produce estimates that are honest both in magnitude and precision. I will not make promises without reasonable certainty.
9. I will respect my fellow programmers for their ethics, standards, disciplines, and skill. No other attribute or characteristic will be a factor in my regard for my fellow programmers.
10. I will never stop learning and improving my craft.