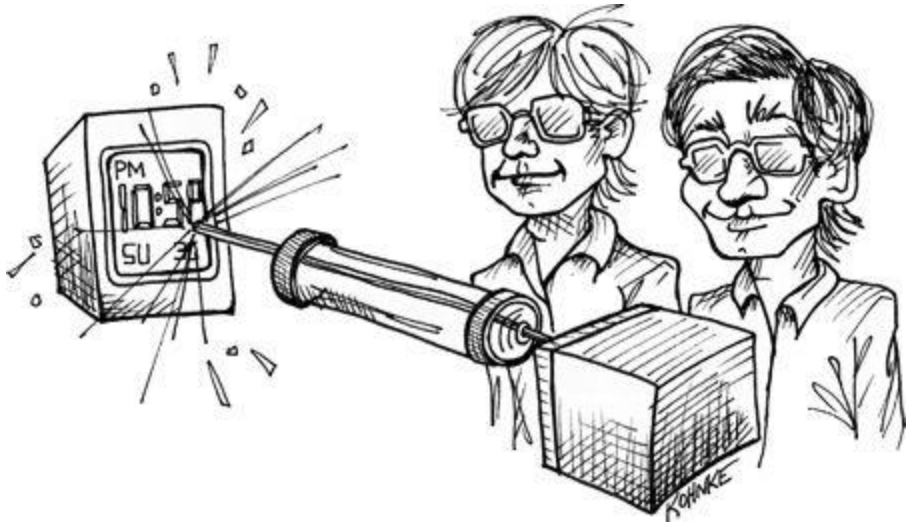
12. Collaboration



Most software is created by teams. Teams are most effective when the team members collaborate professionally. It is unprofessional to be a loner or a recluse on a team.

In 1974 I was 22. My marriage to my wonderful wife, Ann Marie, was barely six months old. The birth of my first child, Angela, was still a year away. And I worked at a division of Teradyne known as Chicago Laser Systems.

Working next to me was my high school buddy, Tim Conrad. Tim and I had worked quite a few miracles in our time. We built computers together in his basement. We built Jacob's ladders in mine. We taught each other how to program PDP-8s and how to wire up integrated circuits and transistors into functioning calculators.

We were programmers working on a system that used lasers to trim electronic components like resistors and capacitors to extremely high accuracy. For example, we trimmed the crystal for the first digital watch, the Motorola Pulsar.

The computer we programmed was the M365, Teradyne's PDP-8 clone. We wrote in assembly language, and our source files were kept on magnetic tape cartridges. Although we could edit on a screen, the process

was quite involved, so we used printed listings for most of our code reading and preliminary editing.

We had no facility at all for searching the code base. There was no way to find out all the places where a given function was called or a given constant was used. As you might imagine, this was quite a hindrance.

So one day Tim and I decided we would write a cross-reference generator. This program would read in our source tapes and print out a listing of every symbol, along with the file and line numbers where that symbol was used.

The initial program was pretty simple to write. It simply read in the source tape, parsed the assembler syntax, created a symbol table, and added references to the entries. It worked great, but it was horribly slow. It took over an hour to process our Master Operating Program (the MOP).

The reason it was so slow was that we were holding the growing symbol table in a single memory buffer. Whenever we found a new reference we inserted it into the buffer, moving the rest of the buffer down by a few bytes to make room.

Tim and I were not experts on data structures and algorithms. We'd never heard of hash tables or binary searches. We had no clue how to make an algorithm fast. We just knew that what we were doing was too slow.

So we tried one thing after another. We tried putting the references in linked lists. We tried leaving gaps in the array and only growing the buffer when the gaps filled. We tried creating linked lists of gaps. We tried all kinds of crazy ideas.

We stood at the whiteboard in our office and drew diagrams of our data structures and performed calculations to predict performance. We'd get to the office every day with another new idea. We collaborated like fiends.

Some of the things we tried increased performance. Some slowed it down. It was maddening. This was when I first discovered how hard it is to optimize software, and how nonintuitive the process is.

In the end we got the time down under 15 minutes, which was very close to how long it took simply to read the source tape. So we were satisfied.

Programmers versus People

We didn't become programmers because we like working with people. As a rule we find interpersonal relationships messy and unpredictable. We like the clean and predictable behavior of the machines that we program. We are happiest when we are alone in a room for hours deeply focussing on some really interesting problem.

OK, that's a huge overgeneralization and there are loads of exceptions. There are plenty of programmers who are good at working with people and enjoy the challenge. But the group average still tends in the direction I stated. We, programmers, enjoy the mild sensory deprivation and cocoonlike immersion of *focus*.

Programmers versus Employers

In the seventies and eighties, while working as a programmer at Teradyne, I learned to be *really* good at debugging. I loved the challenge and would throw myself at problems with vigor and enthusiasm. No bug could hide long from me!

When I solved a bug it was like winning a victory, or slaying the Jabberwock! I would go to my boss, Ken Finder, Vorpal blade in hand, and passionately describe to him how *interesting* the bug was. One day Ken finally erupted in frustration: "Bugs aren't interesting. Bugs just need to be fixed!"

I learned something that day. It's good to be passionate about what we do. But it's also good to keep your eye on the goals of the people who pay you.

The first responsibility of the professional programmer is to meet the needs of his or her employer. That means collaborating with your managers, business analysts, testers, and other team members to *deeply understand* the business goals. This doesn't mean you have to become a business wonk. It *does* mean that you need to understand why you are writing the code you are writing, and how the business that employs you will benefit from it.

The worst thing a professional programmer can do is to blissfully bury himself in a tomb of technology while the business crashes and burns around him. Your *job* is to keep the business afloat!

So, professional programmers take the time to understand the business. They talk to users about the software they are using. They talk to sales and marketing people about the problems and issues they have. They talk to their managers to understand the short- and long-term goals of the team.

In short, they pay attention to the ship they are sailing on.

The only time I was fired from a programming job was in 1976. I was working for Outboard Marine Corp. at the time. I was helping to write a factory automation system that used IBM System/7s to monitor dozens of aluminum die-cast machines on the shop floor.

Technically, this was a challenging and rewarding job. The architecture of the System/7 was fascinating, and the factory automation system itself was really interesting.

We also had a good team. The team lead, John, was competent and motivated. My two programming teammates were pleasant and helpful. We had a lab dedicated to our project, and we all worked in that lab. The business partner was engaged and in the lab with us. Our manager, Ralph, was competent, focused, and in charge.

Everything should have been great. The problem was me. I was enthusiastic enough about the project, and about the technology, but at the grand old age of 24 I simply could not bring myself to care about the business or about its internal political structure.

My first mistake was on my first day. I showed up without wearing a tie. I had worn one on my interview, and I had seen that everyone else wore ties, but I failed to make the connection. So on my first day, Ralph came to me and plainly said, “We wear ties here.”

I can’t tell you how much I resented that. It bothered me at a deep level. I wore the tie everyday, and I hated it. But why? I knew what I was getting into. I knew the conventions they had adopted. Why would I be so upset? Because I was a selfish, narcissistic little twerp.

I simply could not get to work on time. And I thought it didn’t matter. After all, I was doing “a good job.” And it was true, I was doing a very good job at writing my programs. I was easily the best technical programmer on the team. I could write code faster and better than the others. I could diagnose and solve problems quicker. I *knew* I was valuable. So times and dates didn’t matter much to me.

The decision to fire me was made one day when I failed to show on time for a milestone. Apparently John had told us all that he wanted a demo of working features next Monday. I'm sure I knew about this, but dates and times simply weren't important to me.

We were in active development. The system was not in production. There was no reason to leave the system running when no one was in the lab. I must have been the last one to leave that Friday, and apparently I left the system in a nonfunctioning state. The fact that Monday was important had simply not stuck in my brain.

I came in an hour late that Monday and saw everyone gathered glumly around a nonfunctioning system. John asked me, "Why isn't the system working today, Bob?" My answer: "I don't know." And I sat down to debug it. I was still clueless about the Monday demo, but I could tell by everyone else's body language that something was wrong. Then John came over and whispered in my ear, "What if Stenberg had decided to visit?" Then he walked away in disgust.

Stenberg was the VP in charge of automation. Nowadays we'd call him a CIO. The question held no meaning for me. "So what?" I thought. "The system isn't in production, what's the big deal?"

I got my first warning letter later that day. It told me I had to change my attitude immediately or "quick termination will be the result." I was horrified!

I took some time to analyze my behavior and began to realize what I had been doing wrong. I talked with John and Ralph about it. I determined to turn myself and my job around.

And I did! I stopped coming in late. I started paying attention to internal politics. I began to understand why John was worried about Stenberg. I began to see the bad situation I had put him in by not having that system running on Monday.

But it was too little, too late. The die was cast. I got a second warning letter a month later for a trivial error that I made. I should have realized at that point that the letters were a formality and that the decision to terminate me had already been made. But I was determined to rescue the situation. So I worked even harder.

The termination meeting came a few weeks later.

I went home that day to my pregnant 22-year-old wife and had to tell her that I'd been fired. That's not an experience I ever want to repeat.

Programmers versus Programmers

Programmers often have difficulty working closely with other programmers. This leads to some really terrible problems.

Owned Code

One of the worst symptoms of a dysfunctional team is when each programmer builds a wall around *his* code and refuses to let other programmers touch it. I have been to places where the programmers wouldn't even let other programmers *see* their code. This is a recipe for disaster.

I once consulted for a company that built high-end printers. These machines have many different components such as feeders, printers, stackers, staplers, cutters, and so on. The business valued each of these devices differently. Feeders were more important than stackers, and nothing was more important than the printer.

Each programmer worked on *his* device. One guy would write the code for the feeder, another guy would write the code for the stapler. Each of them kept their technology to themselves and prevented anyone else from touching their code. The political clout that these programmers wielded was directly related to how much the business valued the device. The programmer who worked on the printer was unassailable.

This was a disaster for the technology. As a consultant I was able to see that there was massive duplication in the code and that the interfaces between the modules were completely skewed. But no amount of argument on my part could convince the programmers (or the business) to change their ways. After all, their salary reviews were tied to the importance of the devices they maintained.

Collective Ownership

It is far better to break down all walls of code ownership and have the team own all the code. I prefer teams in which any team member can check out any module and make any changes they think are appropriate. I want the *team* to own the code, not the individuals.

Professional developers do not prevent others from working in the code. They do not build walls of ownership around code. Rather, they work with each other on as much of the system as they can. They learn from each other by working with each other on other parts of the system.

Pairing

Many programmers dislike the idea of pair-programming. I find this odd since most programmers *will* pair in emergencies. Why? Because it is clearly the most efficient way to solve the problem. It just goes back to the old adage: Two heads are better than one. But if pairing is the most efficient way to solve a problem in an emergency, why isn't it the most efficient way to solve a problem period?

I'm not going to quote studies at you, although there are some that could be quoted. I'm not going to tell you any anecdotes, although there are many I could tell. I'm not even going to tell you how much you should pair. All I'm going to tell you is that *professionals pair*. Why? Because for at least some problems it is the most efficient way to solve them. But that's not the only reason.

Professionals also pair because it is the best way to share knowledge with each other. Professionals don't create knowledge silos. Rather, they learn the different parts of the system and business by pairing with each other. They recognize that although all team members have a position to play, all team members should also be able play another position in a pinch.

Professionals pair because it is the best way to review code. No system should consist of code that hasn't been reviewed by other programmers. There are many ways to conduct code reviews; most of them are horrifically inefficient. The most efficient and effective way to review code is to collaborate in writing it.

Cerebellums

I rode the train into Chicago one morning in 2000 during the height of the dot com boom. As I stepped off the train onto the platform I was assaulted by a huge billboard hanging above the exit doors. The sign was for a well-known software firm that was recruiting programmers. It read: *Come rub cerebellums with the best*.

I was immediately struck by the rank stupidity of a sign like that. These poor clueless advertising people were trying to appeal to a highly technical, intelligent, and knowledgeable population of programmers. These are the kind of people who don't suffer stupidity particularly well. The advertisers were trying to evoke the image of knowledge sharing with other highly intelligent people. Unfortunately they referred to a part of the brain, the cerebellum, that deals with fine muscle control, not intelligence. So the very people they were trying to attract were sneering at such a silly error.

But something else intrigued me about that sign. It made me think of a group of people trying to rub cerebellums. Since the cerebellum is at the back of the brain, the best way to rub cerebellums is to face away from each other. I imagined a team of programmers in cubicles, sitting in corners with their backs to each other, staring at screens while wearing headphones. *That's* how you rub cerebellums. That's also not a team.

Professionals work *together*. You can't work together while you are sitting in corners wearing headphones. So I want you sitting around tables *facing* each other. I want you to be able to smell each other's fear. I want you to be able to overhear someone's frustrated mutterings. I want serendipitous communication, both verbal and body language. I want you communicating as a unit.

Perhaps you believe that you work better when you work alone. That may be true, but it doesn't mean that the *team* works better when you work alone. And, in fact, it's highly unlikely that you *do* work better when you work alone.

There are times when working alone is the right thing to do. There are times when you simply need to think long and hard about a problem. There are times when the task is so trivial that it would be a waste to have another person working with you. But, in general, it is best to collaborate closely with others and to pair with them a large fraction of the time.

Conclusion

Perhaps we didn't get into programming to work with people. Tough luck for us. Programming *is all about working with people*. We need to work with our business, and we need to work with each other.

I know, I know. Wouldn't it be great if they just shut us into a room with six massive screens, a T3 pipe, a parallel array of superfast processors, unlimited ram and disk, and a never-ending supply of diet cola and spicy corn chips? Alas, it is not to be. If we really want to spend our days programming, we are going to have to learn to talk to—people.[1]