What does a software engineer do?

Posted on March 3, 2010 by Owen Pellegrin

It is not uncommon for me to find myself in a social situation where someone asks me what I do. When I state, "I'm a software engineer," the conversation dries up pretty quickly. My guess? People ask what you do in an attempt to find something they might have in common with you. Many of the people I meet aren't software engineers, and it's a field that definitely has an aura of mystery around it. I might as well say I'm a mime for the reactions I get. People are taught that it's hard or the domain of geeks. It *is* the domain of geeks, but most of us know well enough not to start babbling about software design to people who aren't interested. Talk to us about movies, or TV shows, or explain what you do at your job. Often, software engineers are curious and excited to learn about the intricacies of other fields. Knowledge, particularly uncommon or obscure knowledge, can be a very powerful tool for software engineers.

Anyway, that doesn't help explain what a software engineer does. "Software engineer" is roughly interchangeable with "software developer" or "programmer" in my opinion; I might use any of these terms accidentally but they all mean the same thing to me (this will cause some friction with other software engineers, but it's a good simplification.) What I do is the same thing that any other engineer does, and it can be described thus:

I solve problems for my client, and ensure that my solution meets all of the constraints in which the client requires the solution to operate.

That's a beautiful definition to me. It's indecipherable to non-engineers. Let's talk about what it means.

Solving Problems

This is pretty self-evident. Programs are created to solve problems. Microsoft Word exists because people want to write letters, reports, books, and a myriad of other document types. Microsoft's engineers solved this problem by developing Microsoft Word. The specifics of the problems I solve aren't important. Software engineers satisfy whatever needs the client expresses.

Good software engineers have a special talent for describing problems in very discrete steps. Think about giving directions to someone. You could tell them, "It's 10 miles away at a heading of 210 degrees", but that won't help them. What they want is, "Follow this road until you reach a 4-way stop sign. Take a left onto Oak Street. Follow that for 5 miles, if you see a McDonald's you've gone too far…" So, too, must software engineers find a way to express a problem in very simple terms. Computers aren't as smart as you may believe; you can't tell a computer to fix a cup of coffee. You have to tell it to walk to this drawer, open the drawer, remove a filter, then close the drawer, and so on. Good software engineers can break complicated problems down to these tiny steps, then get the computer to follow the steps.

It's not all simple. Here's a variation of a famous problem in Computer Science:

A salesman needs to visit 10 cities on a sales trip. He has to start from Boulder, Colorado and end the trip there as well. Given a timetable of flights to and from each city over the next few months, determine the flights the salesman should take to spend the least amount of time on his sales trip.

This is called the Traveling Salesman Problem. For more than 30 years, we have tried to find a way to break this into simple, efficient steps that will yield the best solution in the shortest amount of time. For 30 years, the answer has been, "You must try every combination of flights and compare the time it takes." This can take a long time. If there's 10 cities and 8 candidate flights between each city, there's more than a billion possible solutions (my calculation may be wrong, it's quite late for statistical analysis! If it is wrong, it is too small). Add an 11th city and there's 8 billion solutions. In

the real world, the trip would likely involve more than 20 cities and hundreds of flights between each city. It can quickly approach a point where if the computer tests 1,000 paths per second it will take so long to finish the calculation that the salesman will have died aeons ago and the sun has fizzled out. So software written for this task tries a few thousand trips and picks the best one out of that. It's not the *best* solution, but at least it's a solution. Part of a software engineer's job is recognizing hard problems like this and explaining to the client why they cannot be solved perfectly. The software engineer that finds a way to solve any of these problems in a practical amount of time will become very famous in the community. In fact, there's a list of 7 of these problems known as the "Millennium Problems" that carry a $1 million bounty. It's been 10 years since the bounty was posted, and none have been claimed. Not all are related to software engineering directly, but a solution to many of them would turn modern mathematics inside out and thus have a dramatic effect on software engineering.

Meeting Constraints

This is one where I could ramble on for hours, and this post is already longer than I want. The client never wants any old solution to a problem. Usually the client needs it to be fast, small, accurate, bulletproof, or any number of other things. Would Microsoft Word be so popular if it used 3 Terabytes of storage (as of today that's multiple hard drives)? Would Youtube be a success if videos took 9 hours to download? Would Facebook be popular if every page took 15 minutes to load? What if MSN messenger took so much memory it was the only program you could run? Now you have a concept of the kinds of constraints software engineers have to respect. It's not enough to just solve a problem, I have to make sure that the solution satisfies whatever constraints the customer wants.

Those are the externally visible constraints; the user sees them. There's also constraints the user won't ever see, but still matter to the software engineer. These have to do with how the code is written. These constraints include complexity, quality, communication, effort, budget, and time. These constraints are also very related to each other, and satisfying one usually means sacrificing another. For example, code complexity makes it harder for developers to understand what they are doing. Complicated code decreases quality, makes communication more difficult, increases effort, and can increase budget and time. However, *fixing* complicated code takes a lot of effort. When a project is over budget or running late, quality and complexity usually suffer. But when you increase complexity and decrease quality, you tend to have more bugs and spend more time fixing them. We call this a *tension* between the constraints because when you work on one, it has effects on another as if they were connected by a rope. I suppose you could call it a tradeoff as well; that fits. Good software engineers understand dozens of these constraints and strive to reach the best balance among them while staying within time and budget.

That, in a nutshell, is what a software engineer does. Software engineers solve problems, but ensure that the solutions are satisfactory. It's not magic, it's a rigid scientific approach to describing solutions to problems. In the next post, I'm going to talk about some quirks software engineers tend to share and some myths surrounding software engineers.