CHAPTER TWENTY-TWO

Michael Collins on Research Teams

The area that Michael Collins works in interests us for two reasons. First, Michael spent much of his career working on research with teams in an academic setting. But second, that research has been oriented at solving concrete, real-word security problems for serious customers inside and outside government, and the work that he's done spans both academic and commercial areas. We wanted to hear what he had to say about that.

Andrew: You were working on a research project to try to detect intrusions into networks.

Michael: Mostly what we were trying to do was model normalcy. Most of what I do falls into the field of anomaly detection, which falls under the field of intrusion detection. Most anomaly detection is trying to build a model of normal behavior, so when you see that all of a sudden you're falling outside the domain of normal behavior, you get curious as to why that's happening.

A credit card example of this is that you've got normal spending habits. And if all of a sudden you start spending in Katmandu, that's when the credit card company calls up and asks, "Are you in Katmandu?" And the answer is no. That's anomaly detection, as done with credit cards. You do the same thing with network traffic.

Andrew: So your goal was to look at the data from routers, and just by looking at the gigabytes of daily data from router logs you can detect successful and unsuccessful attempts at intrusion?

Michael: That's the Holy Grail. But the first step was just to model what was going on. But to do that, you don't know the types of questions you're going to ask. And this is the cornerstone of research: you don't really know what you will need yet. Oftentimes, you build and rebuild your tools all the way through the process.

Now, the client at this point could conduct one query every five hours or so. It was something in the nature that they were literally conducting the query of the day. They'd put up the graph and say, "This is what we found today." We got that time down from five or six hours to about ten minutes on a dual-processor Pentium box (in 2001). We put out our initial report, which led to the question, "How are you able to do that many queries?" We came up with our explanation as to how we'd shrunk down the data and formalized the query process. And the response from the client was, "We want that."

Andrew: And suddenly you have a software project.

Michael: Exactly.

Andrew: And you need a team to build it.

Michael: Yes. Our original group was four people. Two of us were doing code, and neither of us were considered "programmers." Specifically, the one time I referred to myself as a programmer, my boss, Suresh, yelled at me for about five minutes and said, "You might do woodworking, but you're not a carpenter. You're a researcher. You write code to answer questions, you're not a developer."

Let me make a couple of points here. We'd actually already prepared the environment for tough work, for lack of a better term. So our file type headers, for example, already had a versioning system built into them. We'd been prepping for forward and backward compatibility.

We'd had enough experience with engineering environments to know that prototypes are something of a luxury, in the sense that the difference between your prototype and your production may sometimes be that you simply changed the label on it. So we were not

expecting the reaction we got, but we also treated the development of the system as a serious problem from square one.

Andrew: One thing I've seen when doing research projects and talking to people who've done research projects is that, like your boss Suresh said, researchers are not programmers. If you were talking to someone who's just starting their PhD research or starting a research project in a university environment, how would they apply the lessons you learned to their own project? What did you do differently that a more naive research team might not think to do?

Michael: I think one of the big things—and this is very true, especially of grad students—is that there's a tendency to build a lot of stuff out of spit and bailing wire without necessarily thinking about the solution ahead of time.

One of the big things that we did was that we tried to chop the work into discrete, tiny, well-defined "project-ettes." And one of the major reasons for doing this was to make sure that the code in project-ettes was robust. When you get to the architecture of SiLK* (the name of the system we were building), there's kind of a core library that manages reading files, I/O, a lot of this stuff. And then there are about forty applications that have been written at this point.

Research has a very high failure rate. So the ideal is that as long as we kept the development effort involved in these tiny projects, which we could then test, see if they were useful, see if they answered a question, and if they didn't, they went away. We expected to expend time, but at least this way we weren't expending huge amounts of time. The stuff that really mattered got folded into the central SiLK library. One of the key things was that because we spent a lot of time worrying about versioning, and making sure that the central library was robust, I think we saved ourselves from the headaches you usually see with research projects where you end up with this blob of software that keeps expanding all the time. I think that's because when you do research, you slap in things an awful lot. You're going to have an idea, you're going to try out that idea, and hope that idea becomes useful. We were actually fairly ruthless about cutting off things that didn't work and acknowledging when we had failures. We also spent a lot of time rebuilding and keeping the central core of the system small and robust.

Andrew: So it sounds like there's an architecture perspective, where basically you keep it to the scope, acknowledge that something didn't work. And when it doesn't work you get rid of it, you get it out of your solution so you don't end up with a lot of cruft over time that makes it harder and harder to maintain your code.

Michael: Right. The difference between research and feature creep can be really, really fine.

* SiLK, the System for Internet-Level Knowledge, is the collection of traffic analysis tools that Michael and the team were working on. It's been released as an open source project, and it can be downloaded from *http://tools.netsa.cert.org/silk/*.

One of the things we also did as time passed: let's say that tool X did something, but later tool Y did something that X did, but better. Then we would try to deprecate and remove tool X. Now, it turned out that since the system was being used live, that some people at the client would stay with tool X. But it no longer became a high-priority development task for us. One of the key things attached to this, because people were using the live system, was that there was a lot of documentation associated with it: there's training, there's manuals, there's meetings. And there kind of became a training course and training manual that defined how someone used the system. As we deprecated a tool, we would remove it from the main part of the training course and put it into the back section.

Andrew: So there's the tools and the architecture, and that's one area where you tried to optimize what you did towards building software. You changed the way you did your work in order to keep it more maintainable. What about the way you worked together? The people aspect of it? Did you feel like you had to do something differently from the way a lot of research projects were? Is it something you guys gave a lot of thought to? An environment that would evolve over time?

Michael: Well, that was an interesting problem, because we ran into some interesting skills stovepiping issues. You had a couple of people who were researchers, primarily from statistical or higher-level software engineering backgrounds, who really don't know code. And then you had people who were hired primarily as developers. But our ideal was to find people who sat in the middle: if you could do statistical analysis, and you could write C code to do numerical analysis, then that was what we were looking for on the whole. And part of the reason was because it's easier to justify a researcher than a coder in our line of work. So our goal eventually was to have everyone as a kind of semiautonomous coder/researcher, with a couple of people who basically were "guardians" of the architecture. It didn't really work out that way in the long run. I think the stovepiping ends up being inevitable, simply because people have specialties and interests and skill sets.

One of the problems we ran into early on was with a senior researcher who was explicitly *not* a coder. So if he ran into a problem and there was nobody around to write code for him, he had to basically sit around and twiddle his thumbs, and that was a running problem we had to figure out how to address. Eventually we ended up getting utility developers who would work with people like him and give them the code they need.

Andrew: So you've got this problem to solve, where you've got some of your team members—pure researchers, scientists, mathematicians, statistical analysts—who aren't coders. In a software company, what you'll see a lot of times is that you'll have a team working with business clients. But in this case, those non-programmers were really part and parcel of the team.

Michael: Yes. We ran into a different requirements extraction problem. The researchers served as the engine for new ideas. That's part of the reason we were looking for hybrid researcher-developers. These were people who'd have a problem, they'd go build a tool to solve the problem, and the tool would have to be viable in multiple situations. And we then had a couple of people who tried to figure how to take what had been built and plug it into the entire system. So you'd have these prototypes the researcher-developers would

build, and you'd have a couple of people in the back thinking about how this would go into the architecture. One person would think about how to plug things into libraries, and I'd think about where we were generally moving: here's our gaps in what we've done so far, here's a place where we can probably plug those gaps. We've got these tools, so how about we expand the functionality of these tools to do this additional stuff, and now we've got a coherent view of the problem.

The advantage was that as we did more and more of those things, the pure researchers would not be writing C code, but they might be doing scripting or something like that to plug the tools together. Then we'd have a solution—a slow one—and we could use something better here. So then we could devote some development effort into developing an optimized version of it.

Andrew: So since, at the core of this, this was a project to build software as much as it was a research project, the team members who normally might not have been coders were contributing something codewise in a way that hopefully connected some dots to help move on to the next interesting research question.

Michael: Right. The idea there was to reach a middle ground. First off, you'd never expect a pure researcher to contribute to the code base. But if we have a collection of tools, and a researcher could write a shell script to use the tools, that's not onerous to him and it's not destructive to us. It's something where he can go forward and come up with an initial answer to the question, and we can use that information to say, "Now it's time to build system X."

Again, we're getting to that idea that we don't know if what we're producing is going to be valued.

Andrew: How often did you find yourself going down false paths that required you to remove code from the software? Because when you pull features out or change code, you often end up introducing as many problems as you solve.

Michael: The way we avoided that was that SiLK had two layers. There's the architecture—the filesystem, the file storage, things of that nature—and then there were the tools around it. Research, from SiLK's perspective, consisted of either implementing or gluing together a set of tools to answer a question. As a rule, the research orbited the tools. If we came up with an idea, and we implemented it as a tool, no harm no foul to the central architecture.

Andrew: From an architecture perspective, you made things highly modular, to the point where you had different programs glued together with shell scripts, which is about as modular as possible. And from a team perspective, you tried to make sure that people were, from a technical perspective, as flexible as possible technically.

Michael: Yes. That said, we did end up with somebody who became the guardian of the architecture, and that's a technical job.

Andrew: Did you ever run into any conflicts between people on the team?

Michael: The research group was composed of strong-willed, largely autonomous people who were drawing in their own funding. Argument was the default state of affairs.

Basically, arguments fell into two categories. The first category of arguments are research arguments: I should try idea X, I should try idea Y, I should try idea Z. As a rule of thumb, you'd try out the idea, and if it turned out to be fine you went with it. If it didn't work out, you didn't. However, we were also producing things that did go to the client. We did all of the prototyping with our tools, and if our tools turned out to be useful we'd produce a module and a training course, and we'd teach people to use the tools. And one thing I'd do is elicit requirements from people who used the tools, because they usually had a better idea of what they were looking for than we did. We were using the tools for research; they were using them to actually find people!

So we had that class of arguments. And the other arguments we had were about things like the integrity of the code base, things of that nature. There was a spectacular, legendary argument that Suresh and I had, and this was more a vanity thing on his part. He'd been in jury duty for two weeks. He was coming out, and we were going to ship something, God damn it. Basically, I refused to let that happen until I had tested things. The reason for this was that at this point in time, we hadn't yet coalesced into having the guy who was the guardian of the code base. And one of the things I specifically was brought on to do was to be the system reliability guru. I was the guy who worked out all of the fault trees for how everything failed, and I wrote documents describing to the sys admins how to use these things so if the system fails in fashion X, here's how to handle that kind of recovery. So I wasn't letting anything out. It was a four-hour argument, and basically boiled down to me sticking to my guns. That was a major thing for me, because by that point I'd always been junior to him. This was the point where I had taken ownership of the project sufficiently that I wasn't going to let my name be damaged, because quite frankly he'd been in jury duty and was feeling salty.

The key part of that, really, is just that when you're an academic, reputation tends to be a big item. When you write a paper, you bet your name on the paper. And that was one of the key things about this: we had a culture that was generally dedicated to the idea that when we put something out, that was our reputation on the line. That was taught to everyone within the group. You're representing us, your work represents us, and when something goes wrong you have to own up to it, fix it, and be cautious about releasing things.

This was eventually streamlined to the point where we had a sort of jury system for moderating releases. We had screw-ups, which I handled internally as appealing to personal pride. One time there was a glitch in the system that resulted in two fields being swapped, and I ended up privately talking to the developer and saying, "Look, you made me look like a fool in front of the client. I took that bullet, but don't do that kind of thing again." And after that, he was extremely conscientious and diligent about making sure that never happened again.

I came from a background that was theoretical, very driven by the idea of what engineering design is. We were all trained with the idea that we don't have a monopoly on the truth. So we expect arguments to take place. We also placed a cultural emphasis on the idea that the arguments weren't personal. I tend to say that I expect the most productive environments are composed of people who respect each other but are personally neutral to each other, the reason being that they will provide unbiased judgments and they won't treat each other as fools. They're not interested in being nice to spare your feelings, and they're not interested in being nasty to hurt your feelings. As a group, you have the courage to argue with each other, and the objectivity to actually reach a consensus at the end.

One of our rules about arguing was that you have to reach some kind of consensus finally. Then we could go at each other full out. Most of the time the arguments we had were technical or experimental or something of that nature, so at the end of the argument we'd have to conduct a test to find out who's right. The thing is that most of the people we were dealing with were PhD-educated. This is part and parcel of the process: if you don't know if something's true or not, you have to test.