CHAPTER TWENTY

Alex Martelli on Development at Google

It seems like everyone we know wants to work at Google. We hear about all sorts of great perks—from developers being able to spend some of their time working on their own projects, to massage therapists on staff to make sure everyone's nice and limber. But Google takes its software development very seriously, and Alex Martelli, originally hired by Google as an "Über Tech Lead," has been working on the serious side of software development there. We wanted to hear what he had to say about building teams at Google.

Andrew: How did you come to know what you know about how software works?

Alex: I started out as an electrical engineer designing computer chips for Texas Instruments and then IBM Research. I found myself drifting from doing the hardware, the chips, to doing the system, to doing the microprogramming, because the research operation was just building a few prototypes at a time—to get the chance for the prototypes to actually be used by our target audience of scientists and such, we basically had to make it easy for them to get to there with, say, FORTRAN or APL. They definitely weren't going to be happy doing microcode. So eventually, after a few years of drifting, I had to realize I wasn't designing hardware anymore. I lost track of exactly what was happening in hardware design and for chips and technologies, and resigned myself to being a software person instead.

Shortly after that, I joined a start-up where everybody basically had to do everything. Given that I was probably the most experienced developer in the group, I was by default tagged as a managerial type who was supposed to organize the team and make it work. It didn't really work that well the first time. Fortunately, I did manage to avoid killing the company and actually found somebody who had a little bit more of a clue about management, and I went back to doing mostly software development. Then, learning from what I'd done wrong the first time, I tried my hand at management again, and gradually got more comfortable and more skillful at balancing the purely technical contribution with organization and management and leadership points.

In 2004, I went to Google and interviewed for them and just at the end of the year they sent me an offer to join them with a job description too cool to refuse: "Über Tech Lead." It basically means what I'd been doing for the last few years in my career, with a lot of technical aspects and a lot of management and leadership points as well. The "Über" part has to do with the fact that I've never been responsible for a single software development team at Google, but rather each team has its own tech lead. But I was responsible to oversee and integrate many teams in a certain area of the business.

Andrew: Let's talk a little bit about that first attempt at management you were just talking about. What happened?

Alex: I think the main thing that interfered with my performance at the time was very related to a deeply seated trait of my character, which is an abiding faith in human nature. I use the word *faith* advisedly because it's something you just have to believe. From observing the world, you cannot really come to the conclusion that human beings are fundamentally good—experimental evidence doesn't appear to support that very well. I guess I just have it in me.

So, for example, we had this guy who had been brought on board because he worked with another of the founders on a previous venture. But the other founder wasn't actually a software developer, so they didn't realize that he was bringing in somebody who really wasn't up to the extreme pressure that you experience in a start-up. He probably did well in a large company, where he could take limited responsibilities and execute them decently without making waves. But to make a start-up survive, you just need a com-

pletely different level of intensity, of involvement, and skill. One thing he definitely had learned from his previous stints at large companies was to make very credible excuses.

Andrew: That's hilarious, but surely a bit of an exaggeration?

Alex: I don't remember him ever coming forward with "The cat ate my software," but we kept falling for it. So eventually I was facing the near—not rebellion exactly, but the rest of the team were not stupid. I wasn't stupid, either, just too optimistic and too trusting. And they were realizing that this guy was basically dragging them all down. His productivity was negative. He was actually able to work intensely. He wasn't a shirker, but he'd work intensely for three days and make seven days of work for somebody else who had to fix all the bugs he had introduced. I kept giving him second chance, third chance, fourth chance. And in that, in retrospect, I make myself sound silly. I assure you that with the amount of things to be done all the time in that kind of setting, it's only in retrospect that I recognize the patterns.

Eventually, I had to face the guy and do what is really the hardest thing to do for a manager—and one of the most crucial things to make a team work well. I had to say, "Even though you have been with us from day one, it just isn't working." That was the very first time in my life I had to terminate somebody, or heartily invite him to look for other opportunities, and I didn't expect how heart-rending it would be. But in the end that was absolutely indispensable to let the team survive. It's like one bad apple in a bushel can spoil everything. If management is too kind-hearted and, particularly, if the way the team is formed is based on those acquaintances and friendships, it's even harder to say these very unkind, but true and necessary words to somebody you count as a personal friend.

The fact that he just can't hack software well enough doesn't mean he's a bad person. And I'm sure part of why his excuses were so credible to me, beyond my incredible optimism, was that he probably deluded himself that they were the real reason he wasn't delivering. In the end, having to communicate that message worked. I'm sure people who are particularly skilled can do it without even breaking the friendship, but unfortunately, I'm not as good at that. It's like terminating a personal relationship that isn't working, but you've given it chances and chances and chances. It's heartbreaking, because when the relationship has become a part of yourself, you don't want to hurt this person. But it just isn't working, so what are you going to do? So that was probably the harshest lesson I ever had to learn.

Jenny: Do you feel like before you did that, there were problems in team dynamics because of this one particular person and his excuses?

Alex: Yes, exactly. Everybody was very much invested in the success of the whole enterprise because it's a start-up. You're getting pocket-money-level salaries, and your big payday is supposed to be when the firm triumphs and takes the market by storm. So you don't have just a professional involvement in the firm's overall success, your whole life has been poured into that. So keeping somebody in your team who is actually doing damage to your chance of success is really pointless. And it's difficult to communicate because, hey, isn't that the manager's job to realize that? And the answer is yes. I was not doing my

job—not all of my job. I had more work to do, between managing and technical contributions, than I could deal with, but this part I obviously didn't do well and it was a very important one.

The rest of the team was fine among themselves. When I stepped back to individual contributor and found another pretty good guy to hand the management reins to, things really soared. First, because I was now able to use 100% of my time and energy to actually develop software, and second because this guy was obviously a better, more experienced manager than I was at the time.

Andrew: What would you tell someone on a software team who sees that his manager is having that particular problem? Is there something that one of your team members could have told you to help you realize how bad the situation was getting, so you would deal with it sooner?

Alex: There are many possibilities, and believe me, I've done a lot of introspection about it. I'm pretty sure I wasn't deliberately deceiving myself. There is a strong human temptation to just make believe it's not happening, and maybe it will solve itself on its own. It's just like someone who doesn't want to go to the dentist because he strongly suspects that he has something wrong with his teeth, but he's really afraid of how painful it will be to fix, so he keeps not going. It's wrong, but it's very human. We don't really like to face harsh realities that may require painful action and consequences.

So take the guy you mentioned hypothetically—the one who recognizes, "OK, we've got this six-person team that would be absolutely great if it was five people because that person is just doing damage instead of giving positive contributions and the manager is totally, totally blind to that." It's particularly important if it's a small firm or start-up so that the success of the whole team is paramount to everybody. And that includes the manager.

How does he tell the manager? It's going to be almost as hard as it is for the manager to tell the underperforming person, particularly if it's not a one-on-one thing. If he's realized that there's this one person who just isn't pulling their weight, probably some other teammate is just as aware of that. If it's two or three people going to the manager and basically showing their bad perception here, it's going to be more effective. The manager obviously needs to be hit with a two-by-four, and if you can get two or three teammates together to supply the bad news, do so.

Incidentally, this is something where you really want to be face to face. I just can't imagine myself doing it in email or on the phone or instant messaging, because when you give bad news, you want to leave open the communication channel for emotions that body language and face-to-face communication provide. At the same time, it's absolutely crucial that you will be able to offer emotional reassurance. "You're doing a bad job doesn't mean you're a bad person. It means you've gone awry, and should do something different." That is something that is really much better communicated face to face, like most bad news.

Jenny: So how much do you think the success or failure of that team relied on your role as a manager and how much of that do you think came from the team dynamic itself?

Alex: Well, it relied on the success of the team's manager because, basically, that's the same thing as success of the team. But it didn't have to be me. Indeed, I stepped back to be a senior individual contributor and somebody else took the management reins, and things went much better. That was about 20 years ago, but I don't think human nature has changed very much. Technology, incredibly. Human nature, not so much.

There are technological and methodological aspects that would have made things much clearer—like, for example, if 20 years ago we'd had a properly structured agile team organization with daily standup meetings. One thing about agile, which is a very big plus for actually working, but a potentially big risk and why so many people are afraid of adopting it, is that it makes truth emerge much sooner than the traditional channels of communication.

If everybody is standing around for five minutes in the morning and basically telling each other what have I accomplished yesterday, what I'll accomplish today, what stands in my way that maybe somebody could help with, you can't keep hiding bad performance day after day after day for very long. It becomes pretty evident. That is just one example of how agile structured methods provide so much better communication, and therefore, for better or for worse, make it extremely difficult to hide things from yourself or anybody else. The classic example is how agile estimating—burn-down charts and so on—gives a very clear picture of progress. It makes it painfully obvious to everybody: "Hey folks, this is our velocity. This is the amount of work we're actually able to get done. If we insist on having all the features that we originally accepted, this thing is going to be two months late." That happens to a lot of software projects, and is normally the kind of thing that people don't want to admit to themselves. They may start doing overtime and then maybe even burn themselves out, but in the end they just delay the inevitable. And if it appears far too late to do anything about it, it can be a disaster.

With the high visibility of agile methods, the painful reality will have to be faced much sooner. The earlier you face the reality, the less painful—although still painful—it is to fix it. You might have to drop some features and slide a couple weeks on the delivery time if you realize things early enough. But if you realize them way past deadline, then it's too late to do anything as relatively painless as that.

Andrew: Do you think there's a shortcoming to the agile approach?

Alex: The shortcoming is that you can't hide, and that is very scary for many developers and managers. Eventually the project will probably be a disaster, but in most particularly big corporate situations it may be that two weeks before you are scheduled to deliver and will not make it, there is a big reorg and the project disappears, and you don't get a bad mark on your record. That's like winning the lottery in a sense, but people keep hoping for that to avoid damage to their careers. I don't really think it's a minus from the point of view of the team or of the corporation or whoever wants the software delivered and wants to use it, but it's definitely scary for the manager and for the developers on the team

because there's nowhere they can hide. It makes things so clear, so obvious to everybody involved.

Another thing, which I already mentioned about giving bad news, is that in my modest experience I don't know how to make agile work in a distributed team. I know people who claim otherwise and I really admire them, but in my experience, if the team is not collocated, all the extra amounts of communications going on are just missing. Not that I have any better solution—if you have to work with a totally dispersed team, I don't have any magic bullets to make that work. But I suspect that the usual Scrum experience just doesn't stretch that well to it.

Andrew: I think it's interesting that you brought up two really important sorts of things that go on in a team: transparency, or not being able to hide, and being able to have that constant flow of communication.

Jenny: I think pretty much all development methodologies, agile or otherwise, focus on transparency, on trying to make sure that people understand who is accountable for what and how they progress on that. You're saying that things like burn-down rates and projects will actually make that information much clearer for other people. I'd like for you to drill down into that a little bit more and tell me why you think those specific practices might make it clearer than, say, I don't know, earned value management, or other tools that have been used in all sorts of projects—software and otherwise—for decades.

Andrew: You know, you're right. I never thought of it that way, but burn down—a staple of agile projects—isn't really all that different from earned value. Is it all that much more effective than stodgy 50-year-old project management stuff that they used to build, say, a nuclear reactor or the space shuttle?

Alex: So the reason I think burn-down charts work particularly well is that they form an easily understood, shared language among any member of the project. And let's never forget the external stakeholders, the people who are rooting for the right software to be delivered on time and on budget. They may be internal customers if you are doing internal enterprise development, or they may be product managers, or marketing people if it's supposed to be sold in the marketplace; whoever plays the role of the customer for the specification discovery process.

There's the problem of quantifying exactly what it is that we need to do and what would just be nice, what we don't have to do, and how much will it cost to actually get those tasks done and deliverable to the user; all of these issues are absolutely crucial to the team. (I'm including the customer in the team, as is the best agile tradition.) And these issues are very effectively communicated by the burn-down charts. You don't call something "finished except it still needs QA." If it still needs QA, or anything else, then it is not finished. Finished means finished. Finished means something you can ship tomorrow.

Any code that has been checked into the trunk, approved, accepted, whatever your detailed procedure for calling a feature finished, should be ready to ship today. That's crucial. So the meaning of the word *finished* doesn't stretch in agile as much as it does in other methodologies. And the costs that come with estimating and planning what do we do first

and what do we do next are just as transparent. There is no magic wand that tells you those things for free. There isn't in any software development project. It's always a substantial part of the project's workload to determine exactly what is being done, and to change it in mid-flight and keep responding to the exact needs of the customer or of the stakeholder.

Andrew: A lot of what you're saying would probably come as a big surprise to someone who thinks that "agile" is synonymous with "we don't plan our projects."

Alex: With agile development and burn-down charts, all of those factors that could otherwise end up stretching the projected delivery dates well beyond what the more rigid, more traditional methodology shows are fully accounted for as part of the software development process—which is exactly how they should be, because really, they are at the beginning or at a crucial cyclical time in the process of developing software. This doesn't really have all that much to do with the team aspect, but it has a lot to do with the development that the team is supposed to be doing. So I guess these are basically two faces of the same coin.

If you count the stakeholders or product managers or customers as part of the team, then I guess you can say it has to be the team, but I haven't seen that done except in agile development shops.

Andrew: I find it very interesting that you drew a straight line from burn-down charts to talking with the customer. Burn-down charts are all about answering questions about the effort we've put in and the work we've done. I like that you used the phrase "right software," and talked about figuring out how you know when the project is finished, if there's still work to be done, specifically calling QA and testing in particular as something that people keep doing after they consider the project "done." I like that a lot, because if you crack open any traditional quality textbook—and I'm not even talking about software quality, I'm just talking about engineering quality in general, going back 50 years, going back to Deming and Juran—you'll see those are practically the textbook definitions of quality. Yet you brought them up in the context of agile development.

Jenny: Building the right software, conforming to requirements, fitness to use—we might not use those traditional quality terms today in an agile shop, but I think it's really interesting that they're still behind the ideas you've been talking about. It's almost drawing a straight connection to the team adopting this practice, having this meeting every day where they look at these charts, having a task board up there, where you actually see traditional quality practices showing through. I think a lot of software people wouldn't make that connection.

Andrew: And it all has a big impact on the team—how they work together, their morale. There are other traditional quality ideas that are clearly part of agile, too, right? Embracing change is an important part of an agile process. But responding to changes, and making sure your stakeholders have a channel for making changes, that's also an

important part of traditional project management. And doesn't that have an impact on the team's morale?

Alex: I believe that in this particular case, the periodic enrichment nature of most agile processes actually helps. If you do, say, a monthly iteration for some very large project, or bi-weekly for something possibly smaller and moving faster, you do fully expect things to change at the end of the iteration, because you devote some time at the end of the iteration to do a retrospective: "OK, what did we do in this last iteration? What went wrong? What went right? Is there some new thing we learned that we should apply in the future either to avoid mistakes or to take advantage of the best practice?" That means that change is systematically going to happen, not at random and unpredictable times, but on a schedule. There is a huge advantage in being time-driven. By time-driven I mean meeting every Thursday or Wednesday or something like that, as opposed to "OK, whenever we're done, we're done."

Andrew: Does that help you keep your project's schedule under control?

Alex: It helps you pull into a stride, a rhythm, where you know that this is the deadline and everything that is not tested, committed, accepted by the deadline is not in this iteration. Even if a work is still in flight, you know that nevertheless, this is when you do retrospective introspection. Why did we end up with three tasks in flight where we've already done a lot of work on them and we have nothing to show for that? Well, they were harder than we expected, maybe, or my darn computer kept breaking down. So there are solutions that are completely different. One is to redo the estimate of the remaining tasks: maybe through much more layering we'll have many more, smaller tasks, and so we are not going to be blindsided by one of them proving to be much bigger than we had originally thought.

So then at the beginning of the next iteration, it is absolutely crucial that upper managers or other stakeholders are there, because this is their occasion to change things. This is when they get the priorities right for this iteration, which may require dropping or postponing some features and tasks, moving others ahead, injecting completely new ones, saying that something that sounded like a great idea at the time is unfortunately not going to help, so it shouldn't be part of the shipment—and so on, and so forth.

Again, this is not random change that happens out of the blue, and strikes you as a team member by surprise. It's planned. It's forecast. You don't know exactly what will change, but you do know that something will change more likely than not, because the underlying business reality has changed.

Andrew: It sounds to me just from what you're saying that a big part of making sure that your team actually is successful and builds good software is not just setting the

expectations of, say, the managers and the customers, but actually managing the expectations of the team themselves.

Alex: That's a very good observation. It's a two-way street. The engineers and developers in general and the team leader must be talking regularly to the people who are hopefully burningly expecting the software to be delivered.

Jenny: A couple of things kind of ran through my mind when you were talking about expectations, and meeting the expectations of the customer. I've worked personally on a few agile projects that have gone kind of crazily awry because of just those two factors. I've had customers before who, while they were involved in the entire process as agile dictates, decided to change their mind drastically during various iterations—to the point where we had to rewrite the entire premise of the thing that we were doing, to the extent that maybe we shouldn't have built the software in the first place until they had worked some of those out.

That's one thing that I would bring up. Another thing that's happened to me working in an agile environment is I've had developers who kind of refused to rein in their creativity, and have felt the need to add a lot of gold plating, features that the stakeholder never asked for and didn't need, but which the programmers thought were a great idea. Sometimes that resulted in better software, so people didn't say anything negative about it, but it almost always led to serious quality problems.

Alex: Yes. Engineers, particularly those who are young and brilliant but with not yet enough scars on their back to have fought many battles, do tend to overengineer. I totally agree with that. So what I've found works well for that is a practice that I already found well established at Google when I got in, and I absolutely love. I tried to make it happen in the past, but it's so hard to actually make happen in the real world unless it's already in the culture: mandatory code reviews. I would give credit for this to employee number one, Craig Silverstein. No piece of software gets into the code base unless it's been examined and approved by somebody else on the team. That's not the kind of structured code review that people had in mind in the '70s, with hours and hours of preparation and projection and a whiteboard, that sort of thing.

It's really a lightweight process, and it's perfectly suitable to happen in email or with lightweight web-based tools. The point is that if you can keep an eye on all the change sets and all the code reviews firing back and forth, whether you're directly involved or not, you'll never get blindsided by some change when it hits the code base. Now, I'm talking about changes done internally by the team itself—such as the gold-plating aspects that you were mentioning—as opposed to changes requested by the customers, which is a very different kind of issue.

Andrew: I could definitely see how that sort of code review could help the second problem Jenny talked about, the engineering, gold-plating problem. What about the first thing she mentioned, which I've definitely seen, too. I definitely remember what she was talking about, when her team just got destroyed by people on the business side who were asking for things they never should have been asking for in the first place. Jenny's team faced a terrible problem on that project. The customer completely and unexpectedly

changed direction, and they really did have to practically rebuild the entire code base from scratch. I'm sure that she's not the only person who has run into that problem. Is there a good agile approach to that particular problem?

Alex: It boils down to going to the two big objectives of a development team, which are do the right software and do the software right. Much of what we've been talking about is do the software right. Given that the software has to do this and that, make it happen. Nice, solid, well-performing, bug-free, and so on.

But maybe even more important is to do the right software: the software that has the feature set that will really make the company succeed. I realize that if you have never been on the business side of things, it may seem that those changes, those totally absurdly drastic changes you're being asked for, are capricious and arbitrary. But they do make sense, unless you're working for a seriously dysfunctional company—in which case I would suggest printing your resume on the good laser printer.

If it's not a dysfunctional company, put yourself in the business guy's shoes. The world is changing under them and it's all they can do to stay afloat.

There is one aspect in most schools of agile programming, that I don't entirely subscribe to: that generalities are shunned. Some generality costs you, but some generality is actually a savings of time. So making—though it may be a silly example—a routine that computes the remainder of a division by three, a very specific task, is probably going to be more costly than doing a routine that computes a remainder by anything and passing it three as an argument.

So if you have a good nose, a designer's nose and, to a lesser extent, an implementer's nose, for where generality comes free or even makes your work cheaper, then you as part of embracing change and being prepared for change build classes and modules and packages and routines to be configurable for situations that are a little bit different.

The point is that if the total amount of your work is more than starting from scratch, then this is a sad reality that just has to be faced. If there is reusable code in there, then it can actually be worth reusing. If the business-side people are being totally arbitrary and capricious and wasting the firm's money and your time and their own time and so on, then it's laser printer time. But more likely than not, they are doing the best they can in an extremely challenging, difficult, and changing world.

Andrew: All this discussion about agile reminds me of something that happened a few years ago. Jenny and I were brought in to do a talk at a prominent company about improving software development. At the end of the talk, we got into a discussion with the audience about agile development, which I think it was a little bit newer at the time. Then a developer stands up. He clearly had been there for a while. He was a big guy with a big, gray beard—he really looked like a very stereotypical developer—and other people in

the room clearly deferred to him. So he gets up and… Jenny, do you remember exactly what he said?

Jenny: He said, "Agile means that you don't write anything down, you just start coding immediately."

Alex: Right; I have a pretty famous colleague who would basically be applauding this guy's theories. He's a pretty well-known blogger, Steve Yegge. He works over in the Seattle area. Let's just say that we have our own dissents on the technical plane.

Search for him and "agile" in the blogosphere and you'll find a million posts of his essentially condemning the whole idea, and explaining how the only right way to do software is basically what that developer was advocating. Just start hacking, and things will start to happen on their own. Don't just accept my summary of his theories; go read his own posts. They are well written indeed! And to be perfectly honest, to play devil's advocate, there are examples in which this has worked. My favorite one is—do you know the software company called Autodesk?

Andrew: Sure.

Alex: Have you ever wondered, what does their business have to do with desks?

So the story is they wanted to do an office automation application. Before Microsoft Office or other integrated suites, they wanted to do the automatic desk. And a couple people, very senior engineers in the start-up who actually didn't really like the idea very much, on their own spare time hacked on a computer-aided design application instead.

Andrew: And they built AutoCAD just like that?

Alex: Yes. They thought it would be more fun than doing invoice filing and all that stuff that the businesspeople said there were huge markets for. So the "AutoDesk" application never actually happened, but AutoCAD took the world by storm. The company didn't change their name, but despite their name, Autodesk, they are still flourishing in the computer-aided design business after 20 years or more. There's a great book, by the way, by one of the founders, explaining all these things and the history of Autodesk. So that would be the counterexample, where the guys on the team actually doing things on the proper channels were, as it turned out in the end, not really producing anything marketable for the company. And the rebels, who basically were off on a ledge doing something completely different and "wasting" company resources to make something happen that has nothing to do with the company's business model … actually saved the company and made it a huge success.

But that's basically a story of people winning $137 million with a lottery ticket. It happens, but I wouldn't bank on it.

Andrew: Is that different from a classic skunkworks project?

Alex: In my mind, a skunkworks project is something that upper management is actually supporting. The team is hidden from the rest of the company, particularly middle manage-

ment. Middle management's main role at a large company is to stop any change from ever happening (or at least it seems that way if you ever work with them). Good engineers in the trenches and, funny enough, some of the top management, may actually have vision, and so they may make something happen by hiding in a corner. This project, as far as I can understand from the history that Autodesk has published, was actually the group of engineers deciding entirely on their own.

Another example would be Microsoft Internet Explorer 1, the first edition. Some engineers had this idea that this new Internet thing was all the rage, and that Microsoft should have a browser … and the long chain of middle management totally killed that stupid idea. Nobody will ever make a penny with browsers. So the guys hid in a corner and starting writing a browser anyway. Then suddenly the CEO at the time, Bill Gates, issued a memo explaining that the Internet was everything, including the company's future, and that everything must change on a dime. And those guys had more or less working software ready the next day.

Again, that may be something you'd call a skunkworks project, but again, it's not something I would bank on. If you are working in a company that is dysfunctional, then maybe that's the only way to survive. But you know that there are other companies. If everybody in the company is pulling in the same direction and we're all interested in getting the customers good software and making money that way, there should be no need to hide and go do your own thing instead of pulling the cart together with everybody else.

Jenny: In an environment like that, do you think that a more structured, more traditional agile approach would work a lot better?

Alex: I think it probably would, because the idea of change is baked in, in a way it isn't in otherwise good processes—like the Rational Unified Process, for example. The continual iterations and the concept of timing them, fixing dates, and so on, are optional in RUP—it's something you can do, but it's not the core of the process. The core of the process is drawing a huge stack of diagrams that are illegible to anybody outside of the "priesthood" of your experts. I think that many teams love to produce very tall stacks of paper, because they basically can use them as walls to hide themselves from the rest of the organization, which might otherwise demand accountability and transparency. OK, I'm being a bit snide here, but you know what I mean.

The idea of building an enormous amount of specification before you start coding—eek! That is the kind of dysfunctional behavior to which the programmer you were quoting was basically overreacting in the other way. "Everybody is doing far too much rigid specs, so let's not do any at all." Well, there's a feasible middle way, you know?