CHAPTER TWENTY-SIX

Scott Ambler on Getting Past Obstacles

Scott Ambler helped lead the software world through two revolutions: first in the move to object-oriented analysis and design in the 1990s, and then in the adoption of agile practices in the decade that followed. He's spent a lot of his career focused on making enterprisewide changes, and doing that successfully means having to overcome obstacle after obstacle. We wanted to hear how he managed to make change work on that scale, especially when people around him weren't sure that it would work.

Jenny: We're interested in talking to you about your experience, and how you came to know about how teams work together.

Scott: Right now I'm the practice leader for agile development for IBM Software Group. For the most part, I help customers become more agile and more effective at what they're doing. I also work with IBM itself. A lot of that is helping people to work together more effectively, often in teams. I've written a few books, and done a lot of work in the agile community. I'm the guy behind Agile Modeling, the Agile Data Method, and the Enterprise Unified Process.

As far as my background in teaming, it's a lot of hard-earned experience, I guess. Software is developed in teams, for the most part. As a result, you either learn the easy or the hard way—and I guess for me it was the hard way—about how to work in teams.

Jenny: Getting companies to adopt the kinds of practices and techniques you talk about means convincing a lot of people to change the way they do their jobs—and the way they think about those jobs. How do you go about making those kinds of large changes?

Scott: I think a lot of it is just trying to get stuff done. Software's hard, software's complex, and you need to work with people. I think the first obstacle you have to overcome is yourself, in recognizing that you can't do it all. You're going to need help, and you're going to need to learn from others.

As far as obstacles go, a big one is being able to observe when the team is not working well, because it's not always obvious. If people are shoving each other, then something bad *might* be happening. But I worked on one team where an observer from the outside, if they were to look at us, would think we were about to kill each other. But for whatever reason, the way we worked together best was to argue. We could argue something out, and we'd be shouting sometimes—you'd think we were about to kill each other. What was happening was that we were working together really effectively. Some of it was just because we'd grown to respect each other. And even though we'd never really talked about it, we knew that our process was to talk about it and argue things through. Invariably, nobody had it right going into the discussion, but through the argument we came to a much better conclusion to whatever we were working on.

My point is that you can't always tell when a team was dysfunctional. Because in that case, an outsider would have thought we were dysfunctional, but we were actually phenomenally effective. If there's negative shouting and screaming, that's a problem. But sometimes you can be on a team where everyone's trying, but nobody's communicating, and nobody's reaching their goals. It might not be explicit, but there's a slippery slope that occurs where everybody can work really hard for a long time, but in total you're not accomplishing much. The easy analogy is that during the day you've got people digging a hole, and at night other people are filling it up. Everybody's working really hard digging and filling holes, but in the end nothing of value is actually occurring.

That can happen on dysfunctional teams, and it's not always obvious that it's happening, particularly if the team is large and distributed. When you haven't bonded and you're not close, it's difficult to observe that.

Andrew: So the way that the team communicates can be an obstacle, not just to getting the work done if the communication is problematic, but even to recognizing whether or not a team is effective. And it sounds like personality conflicts can be a real problem. If the team has the right stuff technically, but the people don't mesh for whatever reason, it can keep work from getting done. I'm guessing that someone who was very introverted and avoided conflicts whenever possible would have had trouble joining your team, right? Even if they were perfect, technically, to fill a slot that's open on your team, I have a feeling that they'd be lost. But sometimes you end up in a situation where there's a personality mismatch. What would you do about it?

Scott: So say there was an introvert that ended up getting caught in the crossfire, or however you want to describe it. Someone would have to notice. The introvert would have to step up and say, "You know what? I really don't like this." But that would be really hard for them. So say that's not happening. Then somebody else would have to notice, which is easier said than done. In that environment, our manager would have noticed.

Andrew: Do you think it's the manager's job to notice?

Scott: I would hope so. A good manager or team lead should be responsible for looking out for everybody on the team, which is easy to say but hard to do in some situations. Whoever's in the team lead role or the manager role should be looking out and asking themselves, "How is this person doing? How are they fitting in? How can I help them? What problems are they running into?"

And they need to have techniques to help them notice these things, and status reports might not do it. The introvert probably wouldn't write in a status report, "I hate working with these guys. I'm not getting things done, and I'm not fitting in." The team lead needs to observe that.

Say that someone observes that this person is running into trouble and not fitting in well. I would hope that person would bring it up amongst the team, or point it out to the extroverts who are shouting at each other: "Hey, this is working for you, but Sally over here is really having a rough go of it. Can you try to bring her into the conversation, and maybe calm things down and try to find ways to work with her?" Everyone else on the team would have to find different ways to work with Sally, and Sally would have to maybe step up a bit. We'd all have to take the opportunity to learn; which, to me, is the mark of a professional. You should always be trying to improve how you're doing and how you work. In particular, you've got to get good at collaboration, working together with others and working on a team.

For technical, this can be a challenge. This industry rewards people for their technical skills, and many of us are really geared for being technical, but we often shortchange the softer "people skills," or whatever you want to call them. As a result, we don't focus on them as much. And those are important things, at least in software development. People

are a primary factor of success on a software team. The way you act with each other matters a lot more than whatever cool new technology you're working with today. This is something we don't appreciate as much.

Andrew: I think a lot of people would be a little bit surprised sometimes to hear that, since you've spent so much of your career talking about architecture, development, and practices: object-oriented development, UML and software modeling, agile development, planning, and process in general. But when asked about the biggest problems that face teams, the ones you bring up are people-based, how people work together. Were you surprised to figure that out? Or is that something you knew from the beginning?

Scott: Yes, I was surprised, but I figured that out fairly early, I guess through observation and experience. I figured out pretty early that there's more to it than just technology. I didn't actually fall into this people stuff right away. I guess my first step was to realize that we need to look past technology to process and practices. When I went back to school for my master's degree, I started focusing on Computer Supported Collaborative Works (CSCW), which is a fancy academic term for groupware. I guess that's what gave me an appreciation for software people issues, because that's what CSCW is all about. In this case, how do we use technology to get people to work together more effectively? That's how I became more aware of these software issues.

Jenny: You've also spent a lot of your career talking and writing about practices. Have you ever found that the practices themselves that people are using can help you get past those obstacles to team building that you were talking about? Or, on the flip side, can those practices themselves be an obstacle?

Scott: In non-solo development, techniques like pair programming—or from Agile Modeling, modeling with others—you can start to get that recognition that you have to work together on a regular basis. A lot of people in the agile community practice what they call "promiscuous pairing," where not only do you work in pairs; you swap pairs on a regular basis. When you do that, not only does the quality of your work improve, you learn a lot from others from working with them. By swapping pairs on a regular basis, that really forces the issue, and the team has to learn these softer communication skills to survive in this environment.

But like you pointed out, that in and of itself can be a challenge, and not everybody can work this way. This is one of the challenges that the agile community faces. In a way, the software skills that we're talking about and trying to improve upon are a barrier to entry to some people. Some people really do just want to focus on the technology, and really do want to work by themselves. And that's OK. That, I think, is something that an organization or team needs to recognize: that not everybody's the same, and some people just don't want to work on teams. There are some jobs where it is solo work. I think that's an important observation.

Andrew: Throughout doing the research for this book, we've talked to many people who have told us about agile practices and agile in general. And everyone seems to have a slightly different definition of what agile actually means. Can you tell us a little bit about

how you got started with agile? What kind of project did it solve? And also, what kinds of problems did it cause? Because nothing's a silver bullet. We'd love to hear your thoughts on that, and on what agile really means to you.

Scott: Definitely. On the Agile Modeling site (*http://www.agilemodeling.com/*) I've got my definition. I've actually got it on the screen here—let me read it to you: "Disciplined agile software development is an iterative and incremental (evolutionary) approach to software development which is performed in a highly collaborative manner by self-organizing teams within an effective governance framework with 'just enough' ceremony that produces high-quality software in a cost-effective and timely manner which meets the changing needs of its stakeholders."

Now, there's some "motherhood and apple pie" stuff in there. But I think those are all important features, and a lot of it is teamwork-based. So, for example, all of the talk about working in a highly collaborative manner and on self-organizing teams, that's more obvious from a team perspective. But one of the things that a lot of agilists don't talk about, which I think is unfortunate, is the governance framework aspect to it.

No team works in a vacuum. There's an overall organization and environment that the team is working in. And the work of that team needs to be governed, and governance in some ways can be a good thing. Unfortunately, it's not always done right, which is why some software professionals cringe when you use the term *governance*. But you do want to make it lean, and you do want to make it as effective as possible.

Working closely with your stakeholders is absolutely critical, and that's a teaming issue. In the agile world, we talk about the "whole team" concept. That may or may not be a practice, depending on your point of view. People should have enough skills on the team to get the job done, and you're going to work together to do whatever that job is. And that's a very good thing. But what's often missed is that someone needs to keep an eye on that whole team, and make sure that whole team is actually staying on course and whatever it is they're trying to achieve actually makes sense in the overall organization. That's something I think isn't discussed enough in the agile community.

Andrew: Those two things—governance and self-organizing teams—sound like opposites to me.

Scott: They're really not. Everybody is governed, and this is something that often gets missed. "Self-organizing" doesn't mean that you're out of control and doing your own thing. It means that the team members themselves decide how to meet their goals. But the goals themselves, the resources they use, the time frame they have to do it, those are governed by the organization. Somebody's paying the bills. Just the financial issue alone is an issue, because if the organization doesn't like what your team is doing, they can pull the plug. Now, that's a harsh governance mechanism, but it's fairly realistic. How did the team start up? If it's a software development project, there must be some kind of goals you're trying to achieve. There has to be an initial vision. And setting that vision, the mission—that's a form of governance as well. The rights and responsibilities of the team, the report-

ing chain: that's governance. For example, the junior programmer might not have the same decision rights as the senior technical lead. That's a form of governance as well.

So governance is given short shrift, I think, in many situations. It's always happening, but we don't recognize it. And it's a shame if we ignore it, because one of the things I see in the agile community is that we're so paranoid about bureaucracy and wasted time that things like governance will instantly be beaten up. "We don't need any governance; it's a waste of time!" Well, no. Bad governance can be an obstacle to leadership. Let's try to be effective at governance, because the value of governance is to help us make good decisions and go in the right direction. Let's keep the baby and throw out the bathwater.

Andrew: So a team can be governed by the vision, by the goal of the project, as well as by the expertise and limitations of the team and the company. Now, I've personally seen plenty of software projects go off the rails because the project governed the vision, and not the other way around. I guess that's one way you can describe scope creep.

Scott: Yes. The vision should evolve over time—and rightly so, because the situation evolves, too. But what happens, for whatever reason, is that long before the project actually starts, somebody has this idea that you need to do X and achieve whatever goals, for whatever business reason. There's always this long list of opportunities that an organization has. But there's only so much funding, so they have to choose what they want to do, and hopefully do it effectively. The reason for that project really does direct the effort, because you really want to make sure that you're achieving those goals. And if the goals are changing, you want to make sure they're changing the right way, and you want to manage that evolution. You should be doing a reality check every so often, which doesn't happen as often as it should.

I ran a survey through Dr. Dobb's a year or two ago about how people define success, because we don't have many numbers on this. It's my philosophy that if a project is in trouble, someone should actually say, "Hey, this project is in trouble!" Somebody has to make a decision about how to get this team out of trouble. If nobody knows how to get the team out of trouble, we need to cut bait now, and stop throwing good money after bad. Now, that's my philosophy, but I didn't know how many other people thought this way. So we asked. And what we found was that something like only 41% or 42% of respondents said that in their organization, if a project is in trouble, it's considered a success to cancel that project—which is a real shame, because if I recognize a project is in trouble, I want to get it out of trouble as soon as I possibly can. I would consider stopping a project as early as possible. That's obviously not a good idea, but I'd rather waste half a million dollars and learn the hard way than waste three or four million dollars and learn the hard way.

The only way that you can actually make that work is to monitor what's going on, and ask these sorts of questions. Is this team still succeeding? Does this project still make sense? Because teams can become blind to that. A team can be phenomenally successful in whatever the scope of the project is. But the environment could have changed, and they may

not have recognized that. Whatever reasons there were to start that project may have changed, and we need to stop it.

So, for example, in the fall of 2008, we had the financial crash. For a lot of financial institutions, their business environment changed dramatically from September to October. I'm sure that some of them put some projects on the shelf, because the business environment changed so radically that some projects no longer made sense. It's no fault of the development team at all. Still, why throw good money after bad?

Andrew: The idea of aligning the project to the company, of having somebody looking out to make sure the project we're building is the right project to build, reminds me of a problem that I suspect a lot of agile teams face. But I've never heard anyone talk about it, and maybe you can help shed some light on it. One of the bread-and-butter practices that a lot of agile teams follow—and, I think, a good one!—is to bring a business representative into the team. Sometimes this is part-time, sometimes it's even full-time. I can see that embedding a stakeholder or business person in the team itself can do really good things for communication and information sharing, and for making sure the scope doesn't creep in the wrong direction. Now, to be honest, I've never really seen this done all that well, because I've rarely come across stakeholders who feel that they have that kind of time, for better or worse.

But for a team that does manage to embed a stakeholder into the team, I worry that they may run into a problem where they also start to be affected by that very same blindness that you just mentioned. If you bring the stakeholders from the business side into the team, and everybody suffers from the same blindness, then who's looking out for the project's goals to make sure they're still aligned with what the company needs?

Scott: There are a couple of things there. Sometimes the outsiders go native. It doesn't matter who you're bringing into the team. If you work closely with an existing team enough, you'll start relating to that team, start relating a little bit less to where you came from, which in this case is the business side of things. And this can be a problem.

One of the risks of the "whole team" concept is groupthink. The basic concept of groupthink is that if a group of people work together long enough—and this can be several months—they start to think the same way, and start to become blind to the same issues. Psychologists sometimes talk about this in relation to risky decision making. For example, you might not be willing to bet $100 on a horse race. But if you're out with your friends, and one of them bets $100 on a horse, and then another bets $100 on a horse, then betting $100 starts to seem less risky to you. So the entire group together would together make what they perceive as a very risky decision, but because you're in a group, suddenly that risk has gone down. And that sort of thing happens as well.

So there are these interesting team dynamics that you have to watch out for. Nothing's perfect, right? And this is something that governance efforts have to watch out for. The challenge with governance—and this is the reason I think that people are bitter about IT governance—is that a lot of organizations facing a problem like this will go into "command and control" mode. The people doing governance will think that they're managing. They'll try to direct the project, and they'll get too actively involved, or they'll place too

much of a burden on a team. They'll start asking the team to produce regular status reports or attend the monthly control board meeting, or they'll have these milestone meetings every couple of months to ask the team to justify what they're doing. That throws a lot of extra unnecessary burden on the team. Good governance should be about enablement and about motivation. It shouldn't be about command and control, and it shouldn't be a burden. Obviously, there will be a little overhead, but it shouldn't be too much. If it's too much, the governance effort becomes detrimental, and that's a very serious challenge.

Jenny: So how do you approach that challenge? What's the first thing you do to make sure that you're not forcing a team to have to put up with detrimental governance?

Scott: You want to automate as much as possible. Per Kroll and I wrote a white paper on this about a year and a half ago called "Lean Development Governance"; the philosophies there were to automate as much as possible, to make it as easy as possible to report accurate metrics, and for the person doing governance to understand that different teams are in different situations. The people doing governance need to be flexible, and understand that a team of five people will work differently from a team of 50. A team building a data warehouse works differently from a team building a website. Your team building a website will work differently from my team building a website. We're different people, and we're different teams, even if we're building similar things with the same technology. And as a result, those teams need to be managed in different ways.

A very common mistake that governance people make is that they try to inflict the same process and the same governance structure on different teams, and it just doesn't work because different teams need to be governed in different ways. The goals might be the same, but the way that you reach those goals will be different. The way I like to say it is that you should be aiming for repeatable results, not repeatable processes. But that can be a challenge for the more rigid and bureaucratic among us.

Jenny: That's interesting to me, because when you talk about engineering a lot of automated solutions to deal with this stuff, that can actually get to be as heavyweight as writing a lot of documents. Creating a lot of test frameworks and putting automation scripts in place, and putting all the practices in place that you need to automate your quality activities upfront, and the engineering effort needed for that, seems like it can be very heavy.

Scott: That's a very good observation. If you're toolsmithing all of your own tools, it is quite heavy. This, I think, is another challenge to the agile community. Many agile teams rely on open source software, and for very good reason. But there are limits to that, and one of the limits is one of integration, and particularly one of governance and accurate metrics reporting. You need to go beyond some of the agile rhetoric.

For example, there's an example called Jazz, and if you go to *http://jazz.net* you can download demo copies of it. But in Rational Team Concert, which sits on top of Jazz, we automate all of this. All of the stuff that agile teams are hand-jamming—their defect trend reports and their burn-down charts, whatever it is that they may or may not be reporting

on—that they're either doing manually, all of that gets generated automatically in real time. So there's no real curtain there. But that's because it's already implemented.

If you had to implement it all yourself, it'd be a phenomenal amount of work. So there's limits to some of the things we see the mainstream agile community doing right now. There are some challenges there.

No development team in a bank or an insurance company would think of developing their own compiler. That's something you buy or download free of charge. So now you're starting to see better-integrated tools that are providing the information you need to govern effectively. But you definitely don't want to be hand-jamming all this and implementing it yourself, because that's a huge burden.

Andrew: We've been talking a lot about obstacles to building software, but we'd really love to end on an upbeat note. Can you tell us about a great team that you've worked on?

Scott: One of the best teams I was ever on wasn't a software team. It was at my karate dojo. I trained in karate for about ten years until an injury sidelined me, which is unfortunate, but that's the way it goes. There are some very interesting philosophies that a lot of teams can benefit from. One of the philosophies or rules in North American karate is the concept of belts: you go through a white belt, then a yellow belt, and so on, all the way up to black belt. And one of the rules was that somebody who's a lower belt, somebody who's not as experienced, can always go up to someone with a higher-ranking belt and ask them for help. That person is responsible for helping them to the best of their ability. Now, part of that help might be to say, "I don't know how to describe this to you, but this person over here can help us, so let's work on this and get better at it."

The willingness to ask for help is critical. But the willingness to give help is critical, too, and one of the principles of martial arts is that you learn more through helping and teaching than you do by just trying to work things through by yourself. This is something that many people can relate to.

If it's just orange belts in the room at the time you can still work together on things, and help each other achieve what you need to achieve. That willingness to work with each other that I learned in karate, I try to apply on software teams. It's interesting to me in the agile community of doing coding katas and running coding dojos. I think a fair number of people are bringing martial arts ideas into software development. The martial arts have been around for a long time, and they've figured out how to teach people. Because it's pretty much all voluntary: as an adult, you go to a martial arts class because you want to learn, or get more fit, whatever your goal is. But you're there because you want to get better.

Andrew: It's really interesting to me that you say that. I've been studying another Japanese martial art, aikido, for about 10 years. And one thing I really like about aikido training is that everybody always trains together. And teaching more junior people is considered an important part of your training, especially as you get more senior. One of the things that I've found over and over again—I didn't expect to find this, but I did—is

that I actually learn more from teaching other people than I do from being taught by other people.

That's another thing that I think translates well to the programming and software world. For example, when Jenny and I wrote Head First C#, *our book on teaching people to program, I learned a lot by figuring out how to explain some of the concepts to somebody new. I mean, I definitely understood, say, core principles of object-oriented development going into the project. But I really feel that figuring out how to explain to a new C# programmer why they care about encapsulation, or the difference between interfaces and abstract classes, in a way that they'll actually understand and connect with, brought me to a whole new level of understanding.*

And I found that this all translates directly to my job. I found that as a manager, especially, when part of my job is to train people on my team and help them not just do their jobs but develop professionally, I'd learn from them, often unintentionally but sometimes intentionally. Sometimes someone who's only been programming for a few years has some really good ideas that I've never heard before. So I definitely relate to what you're talking about.

Scott: It's interesting, especially if you look at some of the pair programming research into different combinations. If a novice pairs with an expert, for example, what they found is that both people benefit. Obviously, the novice will pick up a lot from working with the expert. But the expert learns from answering those questions. And the question might be something straightforward: "Why are you doing that?" Well, it's because … wait, why *am* I doing that? It forces you to think through some of your practices, which is an opportunity to improve. "This really doesn't make a lot of sense, and maybe I can do it better."

Andrew: Wow. What you just said is almost exactly the same as something Jenny and I wrote about in our first book. Actually, I've got it right here—it's from the section that's about pair programming teams that have a junior member and a senior member: "Often, a junior team member will ask a seemingly 'naIve' question about the code that turns out to identify a serious problem. This is especially common with problems that the senior member has been living with for so long that she no longer notices them. Sometimes the extent of a code problem only becomes clear when it is explained to somebody else." That sounds exactly like what you were just talking about.

And that begs the question for me: why is it so damn hard to get programmers to do it? Of all the practices, agile or otherwise, that I've had my own teams work with and talked to other people about, pair programming is the one practice that I've had an almost impossible time getting teams to adopt. It's even harder to get them to do that than to start doing automated unit tests and test-driven development.

Jenny: I think there's sort of this intuitive notion that having two people on the same thing is just inherently wasteful, and people just don't want to do it.

Scott: There's a lot of that. Also, people just aren't comfortable with it. There's something to be said for being at your desk by yourself, doing your own thing. It's interesting; pair programming's tiring. You do it for five or six hours, and you're exhausted, because you're actually working.

One of the things I do, and it's a bit harsh, is to really force the issue. My technique is that I'll bring the idea up with the team: here's what it's all about, here's how you do it. But it's hard. We'll talk it through, and talk about why it's hard. And what I'll get the team to agree to is to try it out for a month. We'll swap pairs on a regular basis—every day, you should work with someone new, not whoever you worked with yesterday. We're going to talk it out. We're not going to tolerate solo programming for that entire month. And at the end of the month, then we'll make the decision about whether or not we want to keep doing it. And what I've found is that by forcing the issue, and by really keeping people's noses to the grindstone, is that by the end of the month very few people want to go back to solo programming. But it takes awhile. It's a "no pain, no gain" kind of deal—I'm sure there's other rhetoric, but sometimes you've just got to suck it up and do it. And pair programming is one of those things where you just have to force the issue for a while. Because it *is* uncomfortable at first. It feels strange, and for many people it's outside their comfort zone. So you've just got to do it. What I've found is that on the teams that choose to do it, very few—maybe 5%—of the people go back to solo programming. But it takes a month.

Andrew: Do you think that's because it's hard for someone to put himself in the mindset of someone who's actually doing it if he's never done it before?

Scott: I think so. Pair programming is initially a hard thing, but there are a lot of benefits to it. There's an intrinsic benefit that's very hard to observe directly, and that's the problem. It's one of those things where it's easy to say, "Well, there are two people working at the same desk, so they're half as productive." So it's easy to knock if you've never done it. But once you've experienced it, it's pretty good.

Andrew: Do you think that's something that might be a general rule for getting teams to accept changes? That once they've tried it, they won't want to go back, no matter how much whining there was at the beginning?

Scott: The general conclusion I'd draw is that if there's something where there's so much discussion out there and so much evidence that it works, then it's worth trying. I'm not sure if you're at the point of your life where eating a lot of bran is a good thing—eventually, you'll get there, believe me! It's not something you want to do to begin with. But after awhile, you think, "Eh, that's pretty good for me." So eventually you'll need to just tough it out and do it.