CHAPTER NINE

Grady Booch on Creating Team Cultures

Few people in the software industry are as widely recognized for their work as Grady Booch. His work with Rational and IBM helped guide the industry in both object-oriented development and process improvement with the Unified Modeling Language (UML) and the Rational Unified Process (RUP). Lately, he's done a lot of work with distributed teams and platforms. We started our conversation with him there.

Andrew: Tell us a little bit about what you did in the past that taught you about how teams work.

Grady: Wow. Great question. You know, from the very beginning of my professional career, I was involved with some distributed teams. The very first project I was engaged with was when I was actually still in the Air Force. My first assignment was at Vandenberg Air Force Base, where I was first a project engineer for a telemetry system and then a project manager for a ranged safety system. In each of those cases, we were dealing with a really distributed system. This was back in the late '70s, early '80s. It was really ahead of its time from a commercial perspective, because we were dealing with the fusion of sensor data from radar sites around the world. So that in itself led me to some experiences with how one deals with the architecture of distributed systems. And additionally, the people who were building these systems, most of them were collocated in the Vandenberg Air Force Base area, but we had groups that were scattered about because of the distributed nature of the system itself. Indeed, on the ranged safety display system, which was the one I was a project manager for, we eventually coordinated with groups at the Kennedy Center, so we were across the country in that case. So very, very early on, I was used to development across geographic time zones.

Then, when Rational began, my focus was mostly on architectural issues. I helped most of our customers deal with the transition from waterfall methods to more agile methods and also from monolithic architectures to looser architectures. It really was about ten, fifteen years ago that I began to see a sea change among the customers I was working with. Many of their projects no longer fit in the same building or the same company. And you saw, for economic reasons, outsourcing or near-shoring to break up the team. Also, they were building systems assistants, which meant that their teams were not just the one team on a particular part of the system, but they were dealing with teams across the country or the world. Even more so, as I began to work with multinational corporations, you were dealing with systems that involve different companies as well. So the natural progression of both the technology—the kinds of systems we were building—and the economics had led to natural pressures which meant organizations are distributing both in time and in space. And it's those experiences that really shaped my worldview.

Andrew: So what's the hardest thing about working with a distributed team: different locations, different personalities, or different technologies?

Grady: It's interesting you point that out, because there are some technical issues that are still wickedly hard, which are the problems of coordinating management across time zones. But I think the really difficult ones have nothing to do with technology but have everything to do with social issues. I'd put those in two categories: one is the issue of trust, and the other is cultural differences.

On the issue of trust, if I have a group that is geographically distributed and I may not be working with them face to face, it's difficult to calibrate their abilities and how they're going to react or how I'm going to work with them. It's very different than when I'm working in the same room with them and can see them on a day-to-day basis. This is

called the "water cooler problem" because, when you have a distributed situation like that, there's no opportunity for serendipitous connections, and those kinds of very loose, opportunistic connections are very important in building trust.

On a cultural perspective, if I'm dealing with systems across countries, then the culture of developing on the West Coast is subtly different from the East Coast view, which is very different than that in India, in China, and in Japan, and so there are subtle things that go on that add friction to the development process.

So I'd say that those two, the issues of trust and cultural dislocation, are the most difficult to overcome.

Jenny: What does it look like when trust doesn't work on a team that's distributed, as opposed to one that's working in the same location?

Grady: Well, you know, it smells much the same if the team doesn't trust one another, whether they're collocated or not. If they're collocated and they don't trust one another, then politics get very, very nasty because they're immediate. If they're distributed, you get more of this passive-aggressive behavior, where you sort of ignore the other group and just get things done and eventually certain groups become polarized and eventually disenfranchised.

And you *can* really smell that lack of trust. I mean, I've smelled it in teams that were collocated, and you just go in and you can feel the tension and hate among one another. And nothing gets done very well. There was a great article, an interview with Andy Hertzfeld recently about his experience with the Mac. He observed, and I'm paraphrasing, that conventional processes lead to conventional products. That's not necessarily a bad thing, because there are many conventional products to build, but if you're in a space that's doing some tremendous innovation, and you don't have a culture of trust, then that innovation is going to just suffer, absolutely.

So how does one build that kind of thing? Well, I'm going to mention some exotic ideas, some exotic to simple ideas. The simple ideas are: occasionally, you do need to get people together. Having, in a distributed team, some way that I can get people together at least once really helps, or in a periodic way, moving people back and forth. On the more exotic side, there's been some interesting work in the virtual worlds in IBM, where people have built Outward Bound-type programs inside the virtual world, so that teams working together, although distributed, can work on a common task that has nothing to do with software development, but they learn about one another that way.

It's those kinds of things that can build that opportunity for trust. But it doesn't really deal with the water cooler problem, because with the water cooler problem, you need to create environments where people can serendipitously get together. This is why, on video conferences and phone calls, it's really not a bad thing to get the chatter before and after: "Hey, how's your day, how's your kid, how's your dog?" These things really help humanize the whole process—and ultimately, software development is a human experience. And

insofar as you can bring that human experience to the table, then trust can find its own way.

Andrew: What about dealing with developers who just want to bury their noses and really would prefer if nobody talked to them ever and they could just be left alone to code? It sounds like you'd have to approach the whole trust issue differently with different kinds of people.

Grady: Absolutely. And you do need code warriors like that, who just totally go heads-down. But you have to be careful about them: you need to channel them, you need to interest them, and you need to keep them excited about what they're doing. In fact, the best thing you can do in those circumstances is to get out of their way. But you can't do it in a totally hands-off fashion, because they'll go off and build things that are absolutely fascinating but useless to the business. And so there is this balance one has to have between respecting the work style of individuals, but also the needs of the business itself. And any good manager is going to recognize those differences among individuals.

Andrew: What if you're on a team where you're a developer and you're not sure how to work with someone like that?

Grady: In those circumstances, you find places for common agreement. If I'm working with someone who is just a heads-down, code-warrior kind of guy, antisocial perhaps even to a degree, I have to find a common ground upon which we can work. That might be some wicked piece of code I'll try to engage him on. Also, as a developer in those spaces, I'll often try to draw them out beyond where they're heads-down about and say "Hey, have you looked at this in the real world?" Because such people are a delight to have, but if they're going to be an enduring jewel for your organization, you have to make sure you feed them constantly, too. And that kind of feeding helps build trust.

Andrew: So a lot of this is sort of "care and feeding of the software team" fundamentals.

Grady: There are days when I swear I need to be Dr. Phil. It is more of a sociological issue than it is a technical issue—not that there aren't technical issues, but the sociological issues as you start moving to teams of size become more and more important.

Andrew: Jenny and I have both been in the position of trying to grow teams that hit that wall of three or four or five people—trying to get past that barrier and build teams of 10, 20, 30, 50, 80. How have you handled it? Has it always worked out well for you?

Grady: Oh, not always. I couldn't give you a percentage, but some teams just simply don't fit together—it's true not only in technology but in any kind of organization. Some teams gel, some teams don't. And the key is to provide the mechanisms, and those teams will find their way. If you have too heavy of a hand in terms of process, then there's no room whatsoever for people to self-organize. But if you make it too loose, there's no organization and no degree of predictability. All of us have to realize that some people really, really do like structure. This is what attracts people to large organizations and makes them shy away from start-ups. At the same time, the mentality you need in a start-up, where you have no legacy strangling you, is a very, very different kind of mentality than if you're

going to maintain that. And the people who often build things initially often get really bored quickly and really don't like maintaining the same code for years and years and years, and yet the business goes on beyond that.

This brings up another problem in the space, which is one that I call tribal memory. Every system has an architecture: most of them are accidental, only a few of them are intentional. And what I mean by this is that the architecture that we end up with is really borne from the tens of thousands of small decisions made on a daily basis. You wake up one day and you say, "This is what I have." Not to single Google out anymore, but they're in the same situation. They're saying, "We had no legacy when we started the company, and now we've got this marvelous system and we've dominated the marketplace," and they have made certain architectural decisions for their cloud that are very different than Amazon's cloud, that are very different from IBM's cloud, and yet they've made those investments so their architecture is what it is. And now as people move on, you have this interesting issue of that architecture, those details, are rarely left in the documentation, there often is no documentation, and though the code is the truth, it's not the whole truth, because it doesn't capture things like rationale and trade-off, nor does it capture things that are difficult to discern at the individual lines of code; they're the patterns that transcend the lines of code itself. That kind of stuff is kept in tribal memory. And so you'll find these tribal elders within an organization who've been around for a while, and they understand that rationale and those cross-cutting concerns. The challenge is, as the organization continues to grow, these people move on, and that IP is often walked out the door, literally.

So the challenge is, in addition to the problems of temporal and geographic distribution, that you also have to deal with intellectual property leakage, because having it kept in tribal memory is very, very expensive—well, actually, it's very inexpensive, but it's very expensive to extract it, and it's very, very expensive when it leaves.

Andrew: Is it feasible to extract it? Is it feasible to actually take these busy people and say, "Hey, let's write down the rationale for this piece of software?"

Grady: Absolutely. I have a circumstance where I was working with a state office on the East Coast where they said, "We have these curious demographics: we've got these really young folks right out of college, we've got these older folks, and there's nobody in the middle, because they've heard the siren song of Manhattan and gone off to Wall Street." And now the older people are retiring and dying—not necessarily in that order—so my recommendation was to do oral histories. Get video cameras. Interview them and find out where the skeletons lie. Even something as light touch as that really helps.

Jenny: I want to bring it back to what you said about rationale in the code and how people are making their decisions within the team. What do you think of criticism within a team and how do you handle that in a distributed team? Also, a lot of what process is

about is helping people to understand how they fit in and be transparent with one another. How do you think that works with a distributed team?

Grady: That's a great question, and I think this is a case where you really need to have the mechanisms for the team to find its own way. I really liked the phrase you used there, and I'll rephrase it by saying something to the effect that the ideal developing environment is one that produces a frictionless environment, meaning that it allows you to collect information with minimal intervention by the developers themselves, insofar as I can instrument my change management systems to do things for me, insofar as I can instrument my tests to do daily builds and things like that; this is the kind of stuff that needs to be shown in the tools, because it allows tasks to be automated to a degree that gets out of the way of the individual developers themselves. And yet, as you point out, there will be points of tension, and there will be times when one has to make some hard decisions. In a team that has no visible leader, this can often be very, very painful because egos run high and you see a lot of tension. But ultimately, in most healthy teams, you do find some centers of gravity and knowledge. They can be self-appointed, because of their experience. A good example of this is Linus Torvalds and the whole Linux project. He has risen as the intellectual leader in that space because he began it, but it's not just Linus. There are people that also drive elements of the kernel itself, and they have, through their experience, earned the right to do so and earned the respect, too. And the same thing happens in any gelled team.

As you grow larger and larger, these kinds of roles have to be institutionalized. This is why you see people getting the role of "senior designer" or "architect" or, in larger systems, you see someone who's an architect for a whole group, or for the company as a whole. In fact, there are these growing patterns of where the center of control lies as you move toward less rapid development, where there's a tremendous amount of innovation but you're also throwing away lots of code, to the point where the code begins to accumulate, becomes a capital investment, and becomes part of something of the culture itself that you can't just throw away anymore. And you have to institutionalize some of that decision-making process.

Jenny: You've been talking about the people on the team, but what about the team as a whole? What do you think are the top characteristics of a well-run team?

Grady: Well, I'll give you two that I think are the most important to me. And they're the things I can really smell when I go into a group. The top characteristic is that they're having fun. You can really tell, when you go into a group, if they're enjoying each other and enjoying what they're doing. That's a clear measure that this is a team that's working well.

And the second is that they're wonderfully reactive, but at the same time, they're producing code that meets the needs of the business. This is the difficult balance that I spoke of earlier. You can have small teams that are just running totally unconstrained and having a hell of a lot of fun just burning up other people's money and not doing anything useful, but a group that does understand its business's needs and communicates and does what it

does well in building code—that's the other side. They're building good code for the business.

So those are the two things that I think are the most important. And everything else, frankly, in my mind, is just a minor measure below that. There are lots of detailed measures, but those are the big ones.

Andrew: That's a tall order. A lot of people have probably been on teams that have been fun to work on; not that many could say that those teams were really productive.

Jenny: They tend to cancel each other out.

Andrew: Yeah—how do you keep that from happening?

Grady: The magic is hard to create, and it's something that comes and goes. Xerox PARC, in its time, was such a magical place, although they kind of failed on the equation of trying to make money about it. I think Apple is in a place, right now, where many of their developers are having fun and they're just going gangbusters in the marketplace. Google, in its earlier days, was a place with more emphasis on fun, and they made more money than God, so you didn't notice the inefficiencies—now it's going to become more challenging. IBM's a company that emphasizes the business to the extent of fun, but you still find lots of pockets of fun where people are gelling and loving what they do.

So I don't know how you do it. You're asking me, in a sense, what are the rules that produce this kind of emergent behavior. And if I knew the answer to that, I'd own a small country.

Andrew: There won't be rules, but there are things you can do to nudge your team or your culture in the right direction.

Grady: Sure. And I'll mention two of them here.

One is to make certain that they get out a bit. In other words, you find groups that are heads-down in the code itself, and that's wonderful, because you need people that are skilled in the tools themselves. But ultimately, we're building code to fill some need, and we need to make certain that the team is not insulated, not totally isolated. I think that's an important one because it gives them a context and rationale for what they're doing.

The second thing you have to do, especially as a project manager, is shield them from the rest of the world. There's a lot of really crappy politics that goes on with some companies. And for a team to obsess over the daily machinations of what goes on in the political structure—they're just going to be worried to death. And so a good project manager has to be a shield between the developers and the politics of an organization.

And in some cases, as I said earlier, the best thing a manager can do is to make certain that you get everybody out of these people's way. It's simple things like managing the stupid meetings that the organization requires as it grows—but at the same time, you don't want to make them unconstrained. It's like, Nerf Wars in the hallways are fun, too, but you want to make sure these people are focused.

Jenny: And how does a manager make sure the people on the team are focused?

Grady: Three things come to mind. One is that the best leaders I've found in this space tend to be really articulate. They can write great code, but they can also talk to people outside that space—they can rise above the code itself.

The second is they do care about interpersonal relationships. The best leaders, I find, are not just total gear heads, but people who really do understand the human experience in building software and respect that, too. People have off days—their dog dies, their kid's having problems in school, these are all things that affect the development team. A good leader understands these human issues.

Third, a good leader is able to work at many levels of abstraction simultaneously. A person who can go down to the code, then go up and talk to the CEO, these people are very rare indeed, but they're the best kind of leaders that I can find because they provide that context.

Andrew: It's funny that you say that, because a lot of times, especially early in my career, I've been on a team and thought that to myself: "I'm just not speaking the same language as the people above me." I definitely needed a "translator," and I think maybe I started out a manager simply because I had to rise to the occasion out of sheer frustration. What would you say to somebody else, somebody who finds himself thrust into that role because nobody else will fill it?

Grady: I always give this advice to such a person: please take on that role gently and without arrogance, because you're dealing oftentimes with people who are highly technically skilled, who have a certain amount of ego, and the thing that's going to irritate them the most is that sense of arrogance without any kind of respect. So move into such roles gently. But do so boldly, and don't be afraid to fail.

Another thing that one needs to grow to in that space, and I know it's difficult for somebody starting out, because your very livelihood is at risk, but you really want to be in a position where you can speak truth to power—where you can say to management, "That is the most insane schedule possible, and no human on Earth can do this. It is unrealistic." You have to be truthful; otherwise, your whole process becomes a series of lies built atop one another. Speaking truth to power is important. That's another aspect of shielding your team from the strange politics that go on.

But it's difficult to stand up, because sometimes management doesn't even understand the process of developing software. Software, to them, is just another huge cost item in the organization, and they'd prefer to outsource it all. In that case, the leader has to be one who also gently educates upper management, and that's a long process. And, if you fail, you can always find another job: life's too short to work for a crappy organization.

Andrew: It's funny—Jenny and I spend a lot of time talking to people in a lot of different companies about why projects fail. In fact, we put together a talk called "Why Projects Fail" and took it on the road. And what we keep finding over and over again is that people get really uncomfortable when they hear people talking about failure. That's a

shame, because I think they need to start talking about failure if they're going to learn from it. Have you noticed that?

Grady: One of the signs that I have for the health of an organization is that they're reticent to fail. Organizations that are totally anal and refuse to fail at all tend to be the least innovative organizations, and they're hardly any fun, because these people are so fearful of failing they take the most conservative actions. On the other hand, organizations that are freer in failing, not in a way that will destroy the business, but are given some freedom to fail, are the ones that are more productive, because they're not in fear for their life with every line of code. Because, you know, every line of code you write, it's going to be wrong. You have to have the opportunity to refactor it over time.

This is a big problem for organizations transitioning from building something that doesn't have a legacy to having a legacy. I've seen this scenario happen so many times. You get a group—especially if they're using something lightweight like Ruby on Rails—and they have no legacy whatsoever. Maybe there's a database out there, but somebody says "go build X" and they go build X because they have nothing else that they have to work with. Maybe there are a few services or APIs, but they're pretty much running across virgin territory. The problem is that starts accumulating. Every line of code I write today becomes tomorrow's legacy. And you get to the point where you say, "Gee I can blow this up and start over," but then you get to a point where you say, "I can't blow this up anymore, there's so much stuff here. I simply can't rewrite all of this." And you get to the point where you realize you have some choices. One is to abandon the code and start over, and that's rarely useful, because you never get the behavior you want. Another choice is to isolate it and put some wrappers around it and say, "Let's leave it off in a corner here, it works well enough, we'll get to it eventually," although you never do get to it. There are other cases where you might want to harvest it: "Throw it away, but harvest the best things we can do, and rewrite it, although not from scratch."

Enduring organizations I encounter tend to refactor on a continuous basis, not just at the programmatic level but at the architectural level as well. But that's a challenge because your upper organization will say, "We paid for this code already, why are we rewriting it?" And the answer is that we're doing it to make it simpler, which will give us opportunities for innovation and degrees of freedom we could not have had otherwise. But that's a tough sell, and that's part of the education process that I spoke of earlier.

Jenny: I suspect there are people who will see the word refactoring and immediately jump to agile development. Is that what you mean here? What does that mean for a team's productivity?

Grady: It's just a label, for me, for some best practices I've seen here for a long time. No matter what label you give it, the best process that I've seen hyper-productive organizations across domains tend to follow is this, in three parts. These are organizations that tend to incrementally and iteratively produce releases of executable architectures. So what are the three pieces?

First, a focus on executables. Pushing out raw code as your primary product, that's the most important thing, and that's obviously an agile process. Second, doing so incrementally and iteratively means you have some regular rhythms, where you can introduce refactoring, in which you have an opportunity for failing a little bit along the way and have parallel paths that let you try things. The third element, which is less so one of the agile community and more one of the RUP community, is a focus on architecture as a means of governance. Because, from my experience, as I start dealing with systems of any kind of meaningful intellectual or physical size, there become a set of accumulated design decisions that are part of the tribal memory, and it's very expensive, if not impossible, for me to throw those decisions away and start over. And so what I tend to govern on, as I grow the system over time, is preserving those architectural decisions so I don't end up with a totally random piece of software, but I end up with one that's intentional over time.

Now, call it agile, call it whatever you want, those are the practices that I find to be consistent among the really hyper-productive teams.