CHAPTER ELEVEN

Mike Cohn on Planning

Anyone who's followed the world of software development anytime after the late '90s has at least heard of the agile movement. One area that agile development overlaps with the work we've done over the years is in the practices used on agile projects—especially practices for planning and estimation, and how that impacts teams and the software that they build. Mike Cohn has written books and spoken frequently on exactly this topic.

Andrew: We wanted to start out by asking you about your background. How did you come to know about the things you know about how teams work?

Mike: Well, I guess I got kind of lucky in one of the first programming jobs I had. My boss didn't know what to do with us. He wasn't a manager of programmers at all, so he ended up putting me in charge, even though I didn't have a lot of experience doing that myself. I had a little bit more than the other developers that I was working with, as there were just four of us at the time. I just kind of ended up being a team lead there. From there, I ended up running various software development teams, and worked my way up to being a VP of engineering at a handful of companies. I got a lot of exposure to different types of companies from that.

Jenny: Through all of that experience, can you tell us specifically what characterized some of the better teams that you've been on, and if there's anything that they've had in common?

Mike: Normally what the better teams I've been on have really benefited from is having what's called a clear, elevating goal. They had some sort of mission that they were after and that the team really focused on. When you have something like that, it gets rid of all of the other crap and politics and stuff that get in your way.

Andrew: Have you ever felt like you understood the goal, but your team didn't quite get it? How did you get everybody in sync on the goal?

Mike: Well, a lot of times the team doesn't understand the goal right off. That's part of the project manager's job, or in agile it's part of the product owner's or ScrumMaster's job to help make sure the team understands what that goal is.

Going back to the importance of this, a lot of us have been in companies that have had the goal of "make a ton of money." That's not a very elevating goal. I'm looking for teams to have something that's a little more intrinsically valuable than that to pursue.

There's nothing wrong with making a ton of money, but that's not the type of thing that's going to pull a team together to really achieve spectacular things. Yes, they'll work hard because they think they're going to make a ton of money, but I'm looking for people to go beyond the productivity we get just from those types of motivations.

I'll give you one example. One of the companies I was at was a healthcare company. Essentially, we were the first company to put nurses on the telephone giving out medical advice. That's a very common thing these days, but back in the early '90s, it wasn't.

A lot of people in that company were there because they felt we were going to significantly improve people's lives. It would be great—you would be able to call a nurse rather than go to a doctor, and get medical advice right then.

We had some archetypal phone calls we wanted to support. One we used a lot was a young mother with a sick baby who can't get a hold of the doctor. She calls, talks to a nurse, and her mind is put at ease because she knows what to do for her baby.

What we wanted to do was make calls like this very tangible to people. We started putting out an email a day of a significant phone call that had happened the previous day. Some of those phone calls were life-saving.

I remember one guy had called in with a question about whether he should go to his chiropractor or not. He'd hurt his back. He had pain in the left side of his back and really wasn't sure what to do about it. Should he go to a chiropractor? Should he just take some aspirin and wait? It turned out he was having a heart attack. Our nurse on the phone was able to diagnose that by using the software, plus her own nursing expertise. So she was able to figure this out and dispatch an ambulance to his house. She saved his life.

We started sending around the phone call of the day by email to team members to help them understand the impact that we were having. We did things like that to try to make a team understand why we were doing something.

Andrew: I can definitely see how that would be huge for the morale of the team and really pull people together as a group rather than just a bunch of individuals. I'm wondering: did that also have an effect on the quality of the software? Because it seems like it might.

Mike: I guess that's a tough question. How do you ever do a double-blind study? Do you say, "Hey, you're not motivated today" and "Today you are" and compare?

I believe there's a direct correlation between a team with a clear, elevating goal and a team that writes high-quality code, lots of tests, and such.

The way I always say it is teams that go well go fast. If you write a bunch of high-quality code, you're going to go fast because there's not a lot of bugs dragging you down.

The teams that I've worked with that have had this type of clear, elevating goal as motivation certainly went fast. So I'm assuming that we can draw the analogy they also must have been writing high-quality code to be able to do that.

Andrew: I really like the direct line you drew between getting the job done fast and getting the job done right or well.

Jenny: That's actually a really, really difficult point for a lot of people to comprehend right off the bat. It's something that takes a lot of experience for a lot of people to come to.

I know that both Andrew and I have had a tough time getting that idea across to all different kinds of people, because when you talk about investing upfront in testing and in practices that are going to help people to write quality software, it seems like a lot of extra bureaucracy and a lot of extra time sometimes.

Andrew: A lot of programmers—and just not programmers, but managers, too—often have a lot of trouble connecting the dots between spending time doing stuff other than writing code—stuff like figuring out and writing down what you're going to build and *how you'll build it—and ending up not just with better code, but with better code more quickly than they would otherwise.*

Mike: I think a lot of it is because we don't communicate about all of the goals of the application or the product that we're trying to build. Occasionally, it's not worth writing really high-quality code and we have to acknowledge that, and in those situations, perhaps, rush through it.

I work with some clients that do websites for advertising or marketing campaigns. Let's say we're doing a website to do a tie-in to a movie that's coming out. That site's got to last for a couple months. I don't need the same quality of code for that as I do if I'm writing something like Microsoft Office. If I expect code to live for 10 years or longer, I need to make an investment in writing good code.

Jenny: Let's go back to your initial statement that if you write good code it ends up going more quickly than if you write bad code. How does that fit with what you just said?

Mike: Well, because the payback is over time. Writing high-quality code is an investment, and if I'm only going to amortize that investment over two months of a short-lived website, I may not have the chance to earn back the investment in quality.

If the application's going to exist for years, the investment I take in going very methodically, being very careful with what I write, that's going to have much more time to pay back.

Andrew: Let me make sure I get what you're saying. Are you talking about "paying back" effort in terms of spending less on maintenance going forward? I guess I buy that. Your team is going to have a much easier time extending and maintaining the code and there'll be less of a chance for people to even run across bugs—you can live with more bugs out in the field and distributed to the users. If it's going to be there for years, users will find a lot more bugs and will put up with a lot less hassle than if they only need to use it for a couple of weeks or a couple of months. Is that part of it?

Mike: Absolutely. It sounds like there's some skepticism here. Let me prove this.

Imagine you're working on some application you've done in the past. You're very diligent; you're the most diligent programmer out there in terms of quality and code consciousness, and you want to have this really high-quality stuff. There are two parts to this application: the application itself, and a data import routine you'll run once to move data in from an old system.

It should be clear that almost everyone in that situation would write higher-quality code for the application since it's going to live for years. The import routine gets run once. It just doesn't need the same rigor that the rest of the application does.

I feel like I'm justifying writing crappy code here, so I want to be careful with that. What I'm trying to say with this is that most teams don't have this type of discussion, and they should. This should be discussed with stakeholders: "How long is this system planning to last? What are our big goals here?" We miss big opportunities with that.

I'm going to contend that if we're writing this system, and it's going to live for 10 years, I'm going to write it to a higher quality of beautiful code than I am something that's a one-time "import the data" routine. I certainly have to get the thing correct, but I run all my tests, I check it out, I run some sample data, it looks good, and I'm done.

Andrew: You know what? I'll buy that. I'll buy that at the beginning of the project you should actually choose; come up with a quality standard. In fact, from a traditional quality management perspective, isn't that what acceptance criteria are all about? Or target defect rates? That's all stuff we read about in textbooks, but it sounds like exactly what you're talking about in the real world.

Mike: I think it is something that we should talk about that isn't normally talked about.

Think of how some projects would change. I mean, I don't know that we can ever be this honest, but imagine if you had the key stakeholder who came in and said, "Here are my goals for this project. I am after my next promotion. I want this project to get done. I want it to come out well enough that I get that promotion. I expect that to be in the next six to 12 months. After that, I'm going to a completely different division and I don't care."

We would know where that stakeholder's incentives are. I'm not necessarily even putting that person down. That's the type of incentive some companies create. But that might be how somebody's thinking about an application, and it's why teams get pushed to write lower-quality code than they might be inclined to otherwise in most cases.

Jenny: So how do you figure out the team's incentives? A team can really be driven to release good software, but if the quality is poor, won't that affect the team's morale? And what happens if people on the team are driving toward different goals? What if they have different ideas of what they're building and why they're building it?

Mike: I want to give a concrete example of a company I went and visited in Boston. I met with the VP, who brought me in to consult with his organization. I started by asking him about the application he brought me in to help with.

They were building a new system to replace an existing workflow system. The original system had been built around 2000–2002 and had amazing JavaScript stuff in it, what today we would call Ajax. This was built by some consultants they'd brought in who must have been absolute geniuses to have figured all this out way back then. But the application was extremely fragile because of how it was done with Ajax before Ajax existed.

I asked the VP why they were rewriting the system, and it was purely to get a stable application, one that his team didn't have to spend as much time per year maintaining. That was his only goal.

Then I went and met with the teams. I wasn't looking for miscommunications. I wasn't looking for communication errors. But I just asked them; I said, "Why are you doing this application? What are the big goals for this project?" I got answers like "We need faster performance" and "We need faster throughput because we process more documents now." One answer was, "Well, we're changing to a new technology because our CTO read about it in an in-flight magazine on an airplane and he decided we should switch technologies."

Not one of the five or six developers I spoke with individually had the right answer. No one gave the same answer as the VP who had initiated the project. They were making decisions that were leading them to create the same poor maintainability issues with their application.

The company had an interesting approach where they only released applications to their internal users every three months. This was interesting because it made planning real easy: "Can we meet the June 30 deadline? If not, let's go to the September 30 deadline."

So the boss had said, "Hey, I'd like it by June 30 if you can, but I can understand if it's not ready by then because I know that's a little aggressive." The team misinterpreted this as schedule pressure and was cutting quality corners to meet the June 30 date.

When I told that to the boss, he had a fit.

This was a team that didn't have that conversation at the beginning, and was going off and making completely wrong decisions.

Andrew: You know, it's funny. Hearing you talk about that, I can almost hear Jenny's voice talking about a specific team that she worked with awhile back that had almost exactly the same thing happen. There were very clear goals that had nothing to do with the deadline, but she had a real disconnect with the team about that. She had a heck of a time getting them to let the deadline slip and concentrate on quality instead. Jenny, do you remember that one?

Jenny: Yeah, and the way that we were able to finally fix it was to clearly write down the scope and make sure everybody understood it. But you're right, people had a tendency to fixate on the release dates, and on making sure that everything fit into the timeline. They weren't really thinking hard enough about the actual goal that they were trying to achieve.

Andrew: I guess that gets back to the idea of a "clear, elevating goal" that you've been talking about.

Mike: Yep.

Andrew: So what about that question Jenny brought up about team morale and quality?

Mike: You mean, how is team morale affected by some of these quality issues? I think that was the question.

Jenny: The question, to me, is about the morale of a team that's building poor-quality software. I mean, if you know you're building something that's, say, slightly crappier, is that all right? How does asking a team to cut corners affect them?

Mike: Again, I want to be real careful with this, because 99% of the time when a team thinks they are being pressured to write crappy code, they aren't. Usually it's the result of the stakeholders and developers not being in alignment on the goals of the project. Earlier

I was trying to make the point that most of the time we should be writing far higher-quality code than we are, and that one of the ways to get that some of the time is to realize that not everything in an application needs to be coded to the same quality level.

One of the correlations I draw there is I remember that when I started hiring people into different organizations, I noticed something unusual about the fresh college graduates. They felt like every assignment had to be their A-quality work. They were trying to impress me with their A-quality work on everything they did.

I thought about it and I realized that was what they were used to. You're in college; you've got five or six different professors. One professor doesn't say, "Oh, well, I'll grade you up a little bit because I know you had a hard time in biology class this semester, so I'll take it easy on you. I only need your B-quality work and I'll give you an A."

You were graded independently on each thing. I realized that as I was giving people different assignments, sometimes I had to say, "Do this the best you can." But for other times—let's make it about something other than coding, like doing a vendor assessment—I'd say, "We're going to pick a calendar widget from our application. There are eight good choices. It's not like we're going to go *that* wrong. Go spend two hours checking them out and pick one."

I'd have somebody fresh out of school feel like they had to go spend 20 hours on it and make the absolutely perfect decision. "We'll never regret this decision!" they'd think. Meanwhile, I would have preferred a good decision and 18 hours left for other work.

That's the kind of distinction I'm trying to draw there: not everything needs somebody's A-quality work. We're not even, in most cases, given the opportunity to do that.

Does that make sense? I want to make sure I'm not coming across as saying that I want crappy code. I mean, yes, I gave one example; sometimes it's OK, but normally it's not.

Andrew: That makes sense.

Mike: I still want to stick with a comment on this morale thing for a moment. It's a little bit different from anything you exactly asked, but I think it gets to the real question here. I want to tell you about a team that I worked with.

I was managing a very large project in one of the companies I worked at, and we had a secondary project going on. It wasn't as important, but any project is important. We had some people on that project who were under some very weird management restrictions.

For example, they were told they were not allowed to do any error handling. If an error happened, the boss didn't care. The application could crash. He didn't care. It was just bizarre.

I had heard about a few of these unusual things, but I was swamped with the project I had, the critical project. I didn't get involved in this other one. I just heard about some aspects of it.

A lot of nights I don't sleep well, so I'll get up and go into the office early. It was a stressful project, and I went in real early one morning, about 5:00. It's five a.m. when I get into the office, and I meet the other two developers, the two main developers on the other project, Jeff and Donna.

I said, "What are you guys doing here?" They didn't want to talk at first. But they eventually confessed: "We're here adding error handling and doing testing. The boss doesn't let us do it on the application, but our pride is tied to this thing. We can't put out an application that is like what he's asking for."

For the past three months, they had been coming in at five in the morning and working until seven. At 7:00 a.m., they would go out to the local Denny's for breakfast. They'd have breakfast from 7:00 to 8:00, show up at 8:00, and pretend they were just getting there for the day! They'd then work until 5:00 or 6:00 at night on their application.

After I ran into them that one morning at 5:00, we talked it out. I was a manager. They were programmers. They said, "Can you please help us out? Talk to our boss; figure out what's going on. We can't keep doing this." And this is where it gets at the morale question, because they were beating themselves up over their boss's insistence that they put out a low-quality product.

So I went to talk to the boss, and I said, "What is going on here? Why aren't you letting them add error handling, do testing, and write a high-quality product? I agree with them that it's wrong." I told him what had been going on. And he came clean. Here was the situation.

This was back when it was kind of in vogue to name projects after cities. Microsoft had done it and so companies were naming projects with code names after cities. I was working on a project called the Napa Project, after Napa, California. Their project was called Dodge City, like the old western Dodge City.

The boss had named their project Dodge City because of the idea of walking down a Universal Studios set—you see Dodge City on the side and it's just a façade. It's just the front of the building. There's nothing behind it. There's no depth, it's not real.

He was having them work on an application that they could demo at an upcoming trade show to entice people to renew their licenses because of the great new version that was coming. But that version would never be released. He was never going to come out with the next new version. He was going to upsell people into the related product I was working on.

These developers were on a fake application, the ultimate vaporware, which is why they didn't need any error handling, why it was meant to just be like a good, old Norton demo type of thing from the '80s.

Here was a team killing themselves to do things that legitimately weren't needed. Apart from the ethics of lying to your customer base like this, what a horrible thing for that team.

Andrew: How frustrating for them! If the boss had been transparent about everything from the beginning, had he been upfront with the team about that goal, they would have understood and probably would have done a better job making it look as good as possible in a way that absolutely didn't have to work. Because when you think about it, your hands are really tied by projects that have to work. If they don't have to work, you can do some things that look amazingly cool. All the boss needs is something he can talk about and give hints to his customers.

Mike: Right. What he needed were the paths through it that he could demonstrate at a big trade show. If there was an error, he just wouldn't demo that little path, or he'd make up some other data and go through a different path to do it. He could have a carefully scripted demo and run that only.

Andrew: I like that story—it really ties together the two ideas we've been talking about: the idea of time, quality, and morale, and the idea of how understanding the vision, understanding the main goal of the project, really does have a huge effect on quality.

I guess that brings me to one of the reasons we really wanted to talk to you more than anything else because you spent a lot of time talking about practices, about things that programmers do to make their code better, to improve the quality of the code.

Let's say I put you on the spot; you had to choose one, two, three practices that really could make a difference in quality of code and how well you plan your project. What would you—if a team that was just stuck and wasn't sure what to do—what would you tell them to do first?

Mike: The first one's easy. The absolute first thing I would want a team to do is a continuous integration approach. I just think that is the greatest thing to do.

I remember—this is going way back, it was about 1992. One of the teams I was working with had the typical C++ application of the era, where just due to some external dependencies and things like that our build times had started to go through the roof.

We didn't know quite enough about networking to set something up like a good build server. For various reasons, Novell print queues just happened to be something we'd been using with our application.

I know this just sounds bizarre, but it was fantastic at the time. We actually wrote a build server that monitored a Novell print queue. Novell print queues were wonderful. They could hold anything. We put compile jobs into the Novell print queue, and we had a build server that would just monitor it and kick off compiles whenever it noticed anything getting inserted in there.

It was a completely cheesy, crappy way to do this network communication, but just on that application it had tremendous benefits. We had a little thing you inserted into the queue and the build would kick off.

I don't know why we never took that to the next idea of having it monitor the version control system. It took another seven or eight years before I ever saw a team doing that.

That completely changed what I wanted to do. It's just such a wonderful technique.

So my first one would be continuous integration. I think that's huge for teams.

Andrew: I like continuous integration as a first step, because it's something a team can do by themselves. No manager will ever object to it, because any manager who understands it well enough will generally be in favor of it. It seems like just the sort of stuff programmers do anyway.

Mike: Yeah, great point. There's nothing you can object to about it. I think to your point it's not even one I think you have to go to your manager and say you're doing. You do it.

I'm not trying to mislead managers. I mean, I've been one for many years. But I'll meet teams that obsess about it: "How do I sell my manager on continuous integration?" You don't. I mean, how do you sell your manager on the fact that you need to compile your code? It's a fact of life. You do it.

You're right, that's not one I think you have to go and beg a lot of permission for. Maybe you've got to buy some hardware or something, but just go find an old desktop lying around and it's good enough to get started.

The second thing that I really want teams to do is to get some level of test automation in place. This can obviously tie to the first goal of continuous integration. It's one thing to have the application building continuously; at least we know that everything at least integrates at a compiled level. But I'd sure like to get some automated tests going with that. That's a big challenge for many teams.

I don't really care what level they're testing at, whether it's unit tests or what I'd call the service layer, with tools like FitNesse, or even if it's a little bit more of a scripted user interface type of testing.

I certainly have preferences on how they phase in those tests, but it's most important to just start somewhere. Often it's easiest to start with some high-level user interface tests. Automate that regression test that's sitting around that you have to run manually. Get that automated at a user interface level and then start really working on getting the other types of testing added in.

Andrew: I like that you addressed just the code and build and quality testing. But what about planning? That's something that can have a huge effect on the project, but it's also something that can be surprisingly hard to sell to the team, or to the boss, or both.

Mike: I wrote a book on planning because I got so frustrated with agile teams running around saying, "We're agile. We don't plan." This was common in the early days of agile, through perhaps 2004.

It just bothered me because my experience up until then had been as a vice president of engineering at a couple of different companies, and I'd have been furious if one of my teams had come to me with a process I didn't know about and told me how wonderful it was. "It's faster and customers like the results and it's higher quality," and all these wonderful things that we often get from agile teams. But then they threw in the kicker: "But we have to give up all predictability."

I would have been furious, because that would have been teasing me with a great process I couldn't use. I don't care how good the rest of the process is. If it doesn't allow some degree of predictability, I can't use it.

From my experience having done it with various teams, I knew that agile teams could plan. I ended up writing that book on it just because I wanted to try to dispel this myth of "We're agile. We don't plan."

I don't hear that argument a whole lot anymore, so I hope I've been somewhat successful. That and, I guess, it's mostly been a matter of time and teams proving it can be done. There are enough agile teams out there doing planning. That's what probably has the biggest impact: when you see other teams doing agile planning, it's hard to say that you can't.

I definitely want teams to plan. But your initial question was, what were the first couple things I'd want a team to go do.

Normally, the first one is not planning. Teams have to start getting their act together with a lot of other stuff before they get a chance to really worry about anybody's opinion of their planning capabilities.

Jenny: That's a good point. Initially, when we started this conversation you were saying that not understanding the goal was a big problem in keeping teams in line with what they needed to build.

I wonder; all of the tools that you mentioned were about executing the project and building the code, and not really about understanding the project's goals. Do you think that that's something that teams can work on? Are there specific practices around that that you would recommend?

Mike: Around improving the planning? Yes, absolutely. Jim Highsmith documented a number of them in his book, *Agile Project Management*. He talks about having teams write what's called an elevator statement. This is a two-sentence statement that you can give a stranger who meets you in the elevator and asks, "What do you do?"

But beyond some of those, I've done things like have a team write the magazine review that you'd like to have come out about your product once it's done.

I did this with a company that was doing anti-spyware software. Their product had just been reviewed by a magazine as the second best product in their field. It was the runner-up to an editor's choice award. Now, one way to think about this is if I were the director of the second best movie of the year, I'd be pretty happy. I'd make a ton of money. You've got the second best movie. People go to more than one movie. But the second best anti-spyware product? People buy only one of those.

Being second best in a niche like that was not a good thing for them. So they got serious about some of their process changes they were putting in place, and I asked their Scrum-

Master who was working with them, Erin, to have the team write a magazine review that would summarize what they wanted to have said about them six months later when the next review cycle happened.

This was about establishing their clear, elevating goal. And it worked. They wrote the magazine review, and I remember the day about six months later when the real review came out. I remember it was Halloween and I was home. I wasn't traveling or working that day. I took my kids trick-or-treating, came home, and checked my email at about maybe 10:15 at night before getting ready for bed. I had a three-word email from Erin saying, "We did it!"

A week later, when I could read the real magazine review, I noticed there were sentences in it that were very close to what the team had written in their version of the review: "Finds more spies than any other spyware product." "Removes spies that other products couldn't even find." There were sentences like these that the team had established as its goal in writing a review six months in advance.

That was a testament to the team working with its stakeholder, figuring out what it is they were trying to achieve in this release over the next six months, and then making it happen because they had that very tangible goal very clearly in mind.

Have teams do things like that to solidify that goal. The vision is then shared among all team members.

Andrew: I had one more question about planning. Specifically, I want to talk a little bit about task boards. I've seen a reasonable amount of writing about task boards attributed to you.

More importantly, what do you think of when you think about the differences between working on an agile project and working on a traditionally project-managed project? It's certainly clear that we don't have a schedule in the traditional sense, or structured estimates, or those things that you normally associate with a dry, dusty binder full of paper. But for a lot of agile projects you've got this task board, a living whiteboard that seems to do a reasonably good job of actually planning things out—in a way that programmers actually embrace.

Now, I'm a little bit torn here, because I've definitely gotten a lot of success over the years out of those dusty binders full of planning and project schedules. But I can't deny that task boards and other agile planning tools have been really successful. So would you include something like a task board in your two or three practices that you'd tell a team to adopt immediately? Can you give us a little background, tell us how a task board works?

Mike: A task board is a practice that I would have a team start doing right off if they were doing agile. So toss it in as one of the three practices from your prior question.

I remember when I first started doing that with my Scrum teams. I wasn't aware of anyone else doing them; it was just an idea I had that I thought would help a particular team I was working with. This team wasn't getting the concept of teamwork. We had done a nice planning meeting. We had a big list of things that needed to be done. But people didn't understand that they were to be a team. They were just a group of individuals.

This was a team that had switched from a traditional process. I was introducing Scrum to them. I grabbed some big 2-foot by 3-foot sheets of paper and I said, "OK, this sheet of paper is for this user story and that sheet of paper is for that other user story." And we just taped index cards to them.

There was no logical arrangement to the cards. Each big sheet of paper represented one user story or feature being worked on. Each thing we had to do was represented by one index card.

About a week later, I could see that things had gotten a little better with this group of individuals, but it still wasn't connecting with them that they had to be a team. People would come in and we'd do our daily standup Scrum meetings, but people would only talk about their own tasks. They had no interest in anybody else's work.

We finished up that meeting, and I remember taking the sheets of paper down and organizing them onto a great big whiteboard that this company had. I stretched it out and I drew columns: "Here are the things that we haven't started yet; the to-do column." I made a column of work that's in process, and a column for work that was already done. I said, "We're just going to move the cards very tangibly across this," and it clicked with that team right off.

Since then, this has been something that I always encourage teams to do. Any sort of collocated team in the same building, I think, is going to benefit from putting an iteration plan on index cards or Post-its, and putting them up into some sort of loosely structured form—a task board—up on the wall.

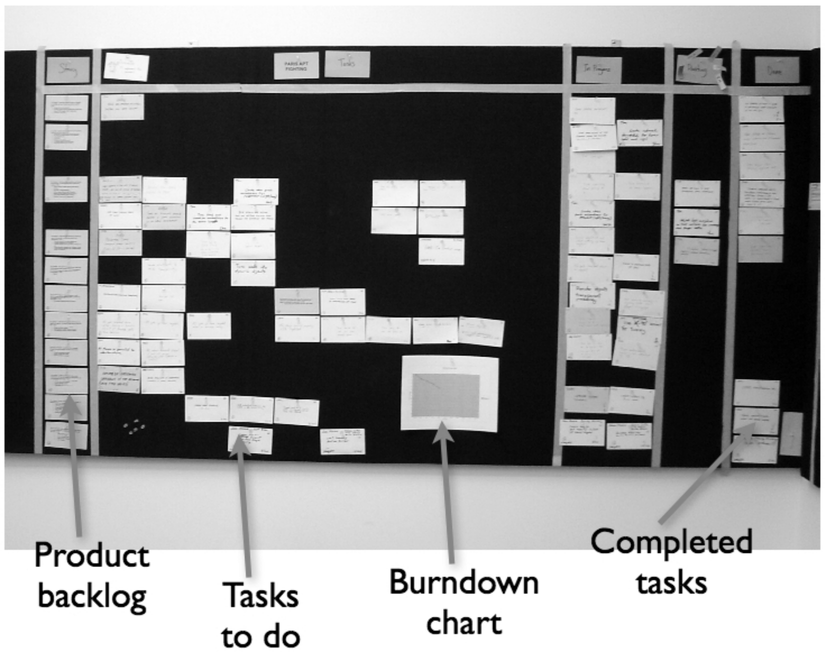
The task board outlines the steps that work has to go through. Whether it's a step like the design, coding, and even testing, just something that shows this is not started, this is being done, and this is done. A task board like that is about as complicated as I ever want one to get, but I've certainly seen teams do far more with that.

Andrew: It sounds to me like it's less about what are we going to be doing in a week, two weeks, three weeks, a month, two months, a year, and more about what we're doing right now. It's about figuring out where the project stands, at a glance.

Mike: I think that's part of it. Ideally, the team walks in and they see a bunch of work up on the board and they say, "What's the most important thing for me to do right now, today? Of all that work up there, what's the most important thing for me to do?"

We're not looking out six months on a task board. Task planning is normally done at the start of a two- or four-week iteration. The task board is wiped clean at the end and starts all over again for the next iteration.

It doesn't have to be complicated. I remember when I first started doing these. I had incredible fears that as kind of the traditionally trained project manager, I would be the



A typical task board, with all of its parts labeled

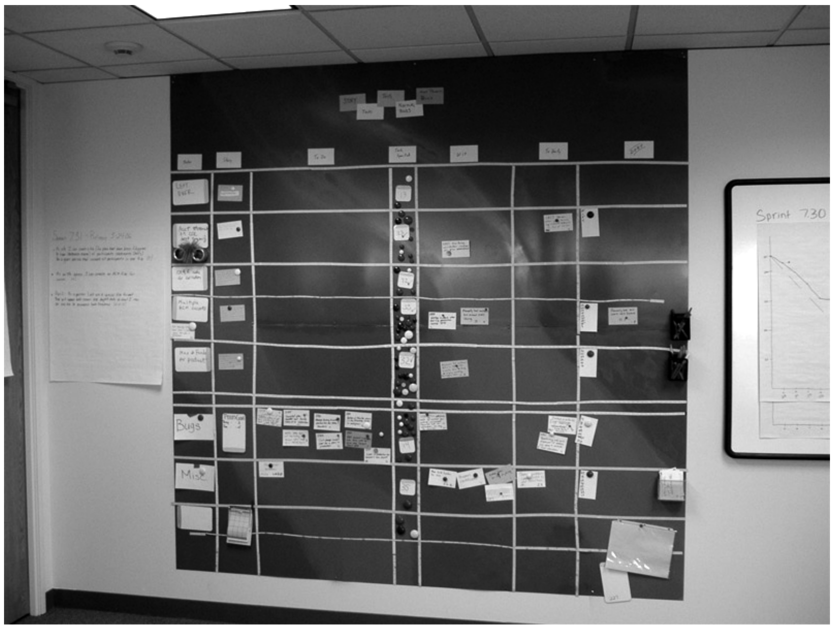
only one to see a critical path. There'd be a critical path of work and no one would see it but me, the project manager.

I worried about that for the first, maybe, six months. After awhile, I eventually realized that no matter how smart I thought I was my team collectively was far smarter than me. If there was a critical path up there on the board, they would see it. This freed me up from having to worry about those things. I still haven't had a team that bumped up against the last day and said, "Uh-oh, there's not that many hours of work left, but there are sequential dependencies between them and now we're screwed." I've never had a team make that type of mistake.

The short time boxes and typically small team sizes that we have eliminate that whole class of issues that we would otherwise have.

Jenny: Actually, that comes back to something you said earlier about being frustrated with agile teams that say they can't deliver predictability. It's definitely clear how you can do planning for a short-term iteration. You're looking at the next three weeks, or the next four weeks, or six weeks, the next sprint or the next iteration. How does that fit with being able to give your manager enough information to plan the next six months, or the next year?

Mike: You absolutely have to be able to plan the next six months. You might even have to do a little planning the next year or two years. But the key is you have to be able to plan them with far less precision.



A task board that uses index cards held up with magnets. The left-hand side shows categories of tasks to be done ("Left Over", "Bugs", "Misc"), and the top shows the status of the tasks ("To do", "WIP", "Done").

I'm sitting here on October 22. There's not a project around that has to be able to say, right now, today, "OK, we've got our delivery date. It will be next September 16." I can't even imagine a scenario where that has to be the case. What would happen if it was September 15 or September 17? That level of precision is just not necessary over that longer term.

Now, I want to be careful. I'm not saying that we don't have to commit to a September 15 date 11 months in advance. But if we do, then we need to have some flexibility in the scope. The key is we don't have to lock those things down 11 or 12 months in advance. Teams need a little bit of flexibility and the business should be fine with that.

If we're looking at a project that we want out 12 months from now, we have to know a couple of different things. I have to know from the team about how long it's going to take, and I have to know from somebody in product management or marketing how much money are we going to make.

But I don't need to know how much money we're going to make down to the nearest dollar or euro, just like I don't need to know the schedule down to the nearest day. Giving me those things with less precision, as long as they're accurate, is OK. We can still make an appropriate decision, which is all that's really necessary.

Andrew: I have one more question I wanted to ask. Jenny and I have done a lot of speaking and training all over the world, and we've talked to a lot of people from all around the software industry. And one thing I've noticed is that there seems to be a small

group of highly zealous agile evangelists. I think you've sort of referenced this a little bit: I'm talking about the sort of person for whom agile is a be-all and end-all answer to everything.

Now, personally, I don't believe there's one silver bullet that solves all project problems. I do believe that many agile teams use some very good practices that can help a lot of teams, agile or otherwise, to build better software. You've given some really good examples of those practices. I also believe that everyone, myself included, should always keep an open mind about the way we do our jobs, and we should always be looking for ways to improve our own software development.

But there are a small number of zealots and evangelists who, as far as I can tell, apparently claim that there's only one true way to build software, and that everyone else in the world should change the way they do things, even if their current methods work just fine.

And I've noticed that you don't do that at all. So I want to know if you've noticed the zealotry that I'm talking about. How have you avoided falling into that? Or, on the other hand, do you think that I have it wrong? Is vocal agile evangelism—to the exclusion of all other ways of thinking about building software—a good thing that people should be falling into? I know that's kind of a loaded question, but I think you know what I mean.

Mike: Well, I don't think agile zealotry really gets us anywhere. I think the agile approaches are the right way to do things. I'm certainly a huge advocate of that. But I think two things prevent me from becoming an overzealous agile advocate.

One is that it's just totally by luck that this is how I've always done software. When I started out, in my first real job in a company (other than a very small company), I was doing software in a division of Anderson Consulting back in what was called their Litigation Information Services division. We worked on lawsuits. We worked on projects for people and companies who were being sued.

I remember working on very high-profile lawsuits. When you're working on those types of projects, you can't take a month or two to get something done. An attorney would come to you and say, "We need a program to solve this problem. We need it now." We needed to have very short cycles and turnaround periods on things like that.

I got lucky, and just kind of learned how to build software in that way back in the 1980s. I became agile just by luck and circumstance. The nature of the businesses I was in led me to do this.

I think the second thing that influences me is having lived through the object revolution. I think agile is almost the second wave of the object-oriented revolution. Agile is a continuation of that idea, which is why we saw that so many of the early, great OO thinkers found the agile movement. They were just continuing what started with objects.

I don't hear people saying things like, "I have to go to the OO design meeting today." They just say, "I have to go to the design meeting." Objects won; nobody talks about OO. Nobody asks, "Is this a structured design meeting or an object-oriented design meeting?"

Objects won, as they should have. For the most part, teams do object-oriented development. (I know there are some applications that don't and can't.)

I want agile to be the same type of thing. I'd like the word *agile* to go away. I'd like eventually not to be talking about agile software development. I'd rather just be talking about software development. I don't know if it's another five or 10 years or whatever it takes, but eventually we'll stop talking about agile and it will just be how we do it.

Some projects are more or less agile than others, just like some projects make better use of objects than others. We won't need books with "agile" in the title. We won't need an Agile Software Conference. It'll just be the Software Conference.

Agile goes away as a concept and it wins. It simply becomes how we do things.