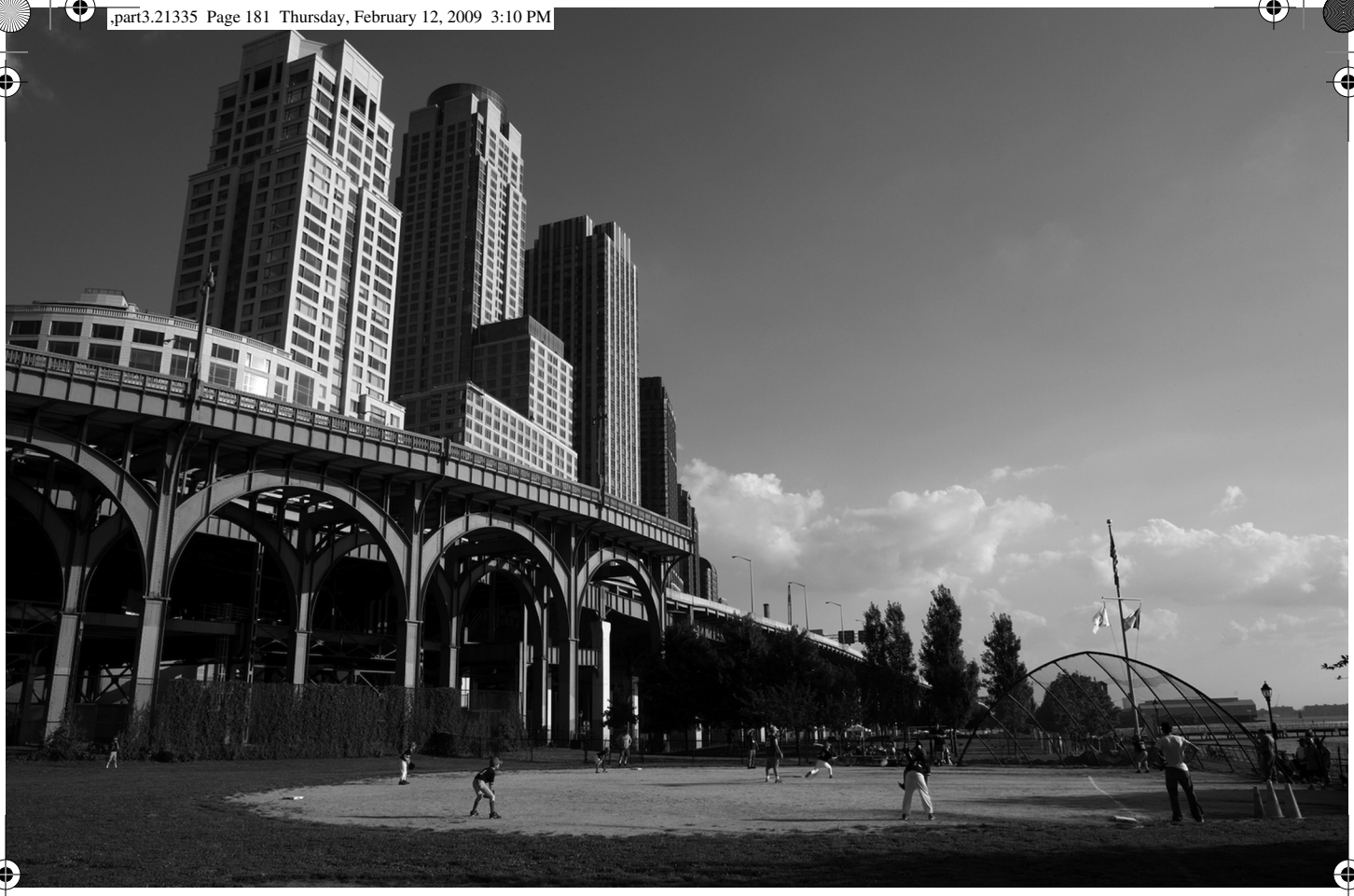

PART THREE

Practices

ALL TEAMS—EVEN TEAMS WITH GREAT PEOPLE WHO WORK really well together—have habitual problems. And when they do, the problems are bigger than the individual people on the team. It's both frustrating and difficult when you just know your team is technically capable of solving their problems, but somehow they keep succumbing to them.

That's where changing the way you work can have a big impact on how your team works. And that's what practices are all about: finding a better way to do things so that your team doesn't get stuck on the same old problems.

The world is full of books about building software: some great, some good, some not so good. The not-so-good books claim to tell you "the right way" to build software. The good and even great ones will tell you that they have a good way—not necessarily *the* way—to build software. What they all have in common is practices.

It's pretty easy to get overwhelmed with practices, because it seems like there are dozens of ways to do any one thing on a project. Just deciding how you'll describe how your users will use the software can be a challenge. Will you use user stories? Use cases? Textual use cases or visual, UML-style use cases? Or maybe more traditional stimulus-response sequences (which predate both use cases and user stories)? Do you write them down in a document, or pin them to a wall? Do you use index cards? Post-its? An Excel spreadsheet? There are literally dozens of ways to write down what amounts to the same information. It all depends on how your team works best and there is definitely no single right answer.

And that's just the problem of writing down how your users will interact with the software. Almost every programmer has a different idea of a perfect code review. Or how to document the architecture the team's come up with. Or how to write down a project schedule—some people hate, hate, *hate* Gantt charts, others can't live without them.

It's hard to overstate just how dangerous change can be to a team's culture. In fact, not just its culture; a change can threaten its very existence. There are two big risks that teams face when they try to adopt a new practice. One is that the practice itself is stupid, but the person pushing it doesn't know that. One of the most popular and unfortunate practices that teams suffer through is the useless status meeting. That's not to say all status meetings are useless. (Ask anyone on a well-run Scrum team, and they'll tell you about their way of having effective meetings!) But there are definitely status meetings that are universally useless, usually done solely for the benefit of a project manager or senior executive. Everyone on the team sits around a table, and each person waits his turn (usually checking his email when he's not speaking). When it's a team member's turn, that person summarizes everything he did in the past week. It's not clear whether that information is ever recorded anywhere, but it's generally not used for anything other than some senior manager's ego gratification. Nobody hears anything anyone else has said, and the whole thing basically just sucks several hours a week of the team's time, with no useful product whatsoever.

A practice such as the useless status meeting is bad for the team, because invariably, nobody ever asks anyone on the team if he gets anything out of it, or if it improves the software. If someone did ask this question, the meeting would be halted immediately.

But what about when the practice is smart, and the people on the team just don't get it? That's the second big pitfall. A lot of teams have run into exactly this problem when trying to put code reviews in place. A lot of programmers agree that code reviews are a good idea. Look at most successful, high-profile open source projects—for example, Apache, Linux, and Firefox—and you'll see a culture that's built up around code reviews. But it's very common for a team to fail to actually put them in place, because the programmers themselves moan and groan every time there's a code review scheduled. When a code review does happen, very little comes of it, because the people participating didn't really believe in its value, and didn't work very hard to review the code. And that creates a self-fulfilling prophecy: everyone decides that the code reviews don't work, and stops doing them. The same thing happens anytime the team doesn't really believe in the value of the new practice.

If you want to put a new practice in place on your team, you need to do two things. First, you need to convince everyone on the team that it's worth doing. And second, you need to get the people who are paying the bills to understand the value of taking the extra time and effort—or, in their view, money—to do the practices. Because a good practice pays for itself in the long run, but that's definitely not intuitive for a lot of people (especially ones who don't actually write the code). Try telling your boss that you want to adopt a new practice that will cause it to take twice as long to build the project, but the testing and deployment will go so much more quickly that it'll be worth the extra time spent upfront, and you'll see what we mean.

It takes a visionary to see the value in a new practice. It takes a salesperson to convince management to pay for it, and to convince the team to do it. And in a lot of cases, it takes an above-average team to be open-minded enough to change the way they work.

The way your team chooses to work has a big impact on how successful your projects will be. And although there's not one correct way, if the team doesn't come to some sort of understanding about how they'll build the software, they risk working against each other.

The stories and interviews in this section are about practices. But more than that, they're about teams finding their way to better practices that suit them, because there's nothing more dangerous to a team's culture than imposing a change that doesn't work for them. We chose these stories because they're about important, pioneering teams who did this well, dramatically changed the way they worked, and had great results. These team leaders saw serious, habitual problems that were preventing the teams from building better software, and they found a way to get their people to embrace a better way of building it.