CHAPTER 2

How to Be Agile

What does it mean to “be agile”?

The answer is more complicated than you might think. Agile development isn’t a specific process you can follow. No team practices *the* Agile method. There’s no such thing.

Agile development is a philosophy. It’s a way of thinking about software development. The canonical description of this way of thinking is the Agile Manifesto, a collection of 4 values (Figure 2-1) and 12 principles (Figure 2-2).

To “be agile,” you need to put the agile values and principles into practice.

Agile Methods

A *method*, or *process*, is a way of working. Whenever you do something, you’re following a process. Some processes are written, as when assembling a piece of furniture; others are ad hoc and informal, as when I clean my house.

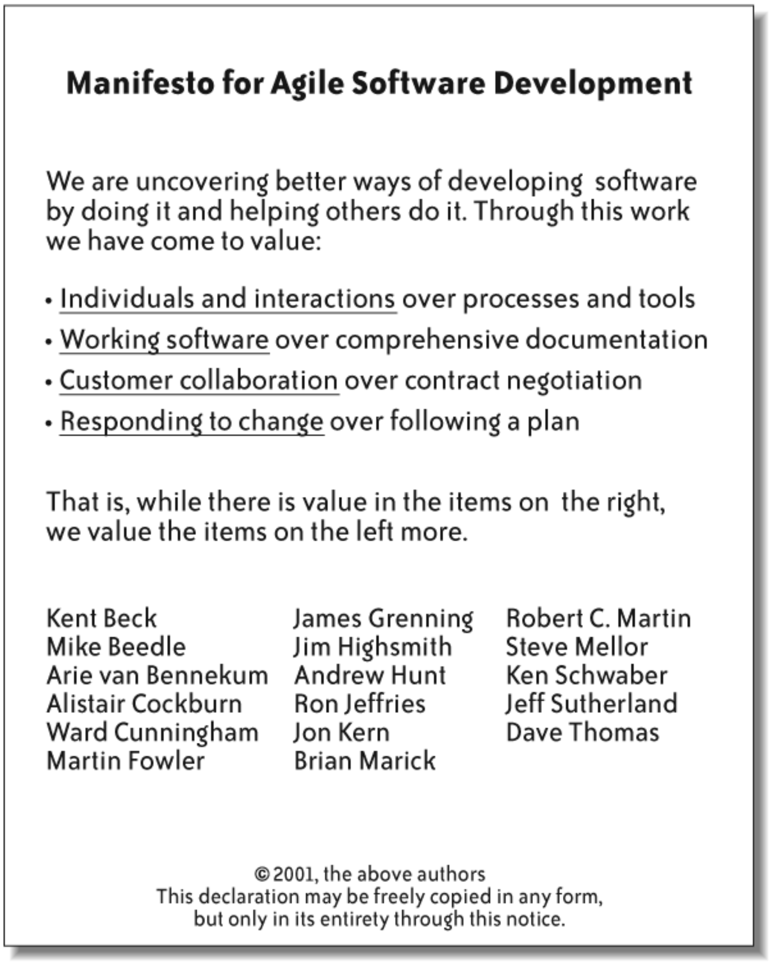
Agile methods are processes that support the agile philosophy. Examples include Extreme Programming and Scrum.

Agile methods consist of individual elements called *practices*. Practices include using version control, setting coding standards, and giving weekly demos to your stakeholders. Most of these practices have been around for years. Agile methods combine them in unique ways, accentuating those parts that support the agile philosophy, discarding the rest, and mixing in a few new ideas. The result is a lean, powerful, self-reinforcing package.

Don’t Make Your Own Method

Just as established agile methods combine existing practices, you might want to create your own agile method by mixing together practices from various agile methods. At first glance, this doesn’t seem too hard. There are scores of good agile practices to choose from.

However, creating a brand-new agile method is a bad idea if you’ve never used agile development before. Just as there’s more to programming than writing code, there’s more to agile development than the practices. The practices are an expression of underlying agile principles. (For more on agile principles, see Part III.) Unless you understand those principles intimately—that is, unless you’ve already mastered the art of agile development—you’re probably not going to choose the right practices. Agile practices often perform double- and


Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

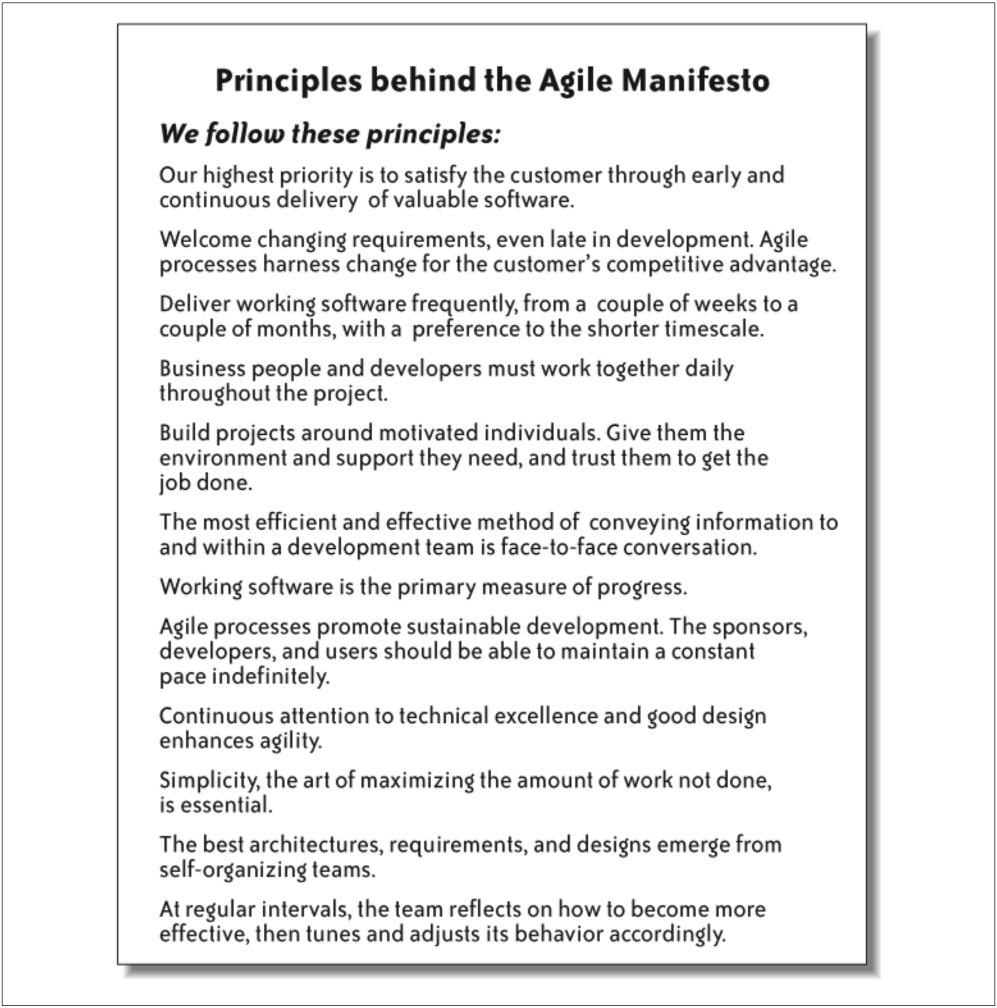
Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

© 2001, the above authors
This declaration may be freely copied in any form,
but only in its entirety through this notice.

Figure 2-1. Agile values

triple-duty, solving multiple software development problems simultaneously and supporting each other in clever and surprising ways.

Every project and situation is unique, of course, so it's a good idea to have an agile method that's customized to your situation. Rather than making an agile method from scratch, start with an existing, proven method and iteratively refine it. Apply it to your situation, note where it works and doesn't, make an educated guess about how to improve, and repeat. That's what experts do.



Principles behind the Agile Manifesto

We follow these principles:

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity, the art of maximizing the amount of work not done, is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Figure 2-2. Agile principles

The Road to Mastery

The core thesis of this book is that mastering the art of agile development requires real-world experience using a specific, well-defined agile method. I've chosen Extreme Programming for this purpose. It has several advantages:

- Of all the agile methods I know, XP is the most complete. It places a strong emphasis on technical practices in addition to the more common teamwork and structural practices.
- XP has undergone intense scrutiny. There are thousands of pages of explanations, experience reports, and critiques out there. Its capabilities and limitations are very well understood.

- I have a lot of experience with XP, which allows me to share insights and practical tips that will help you apply XP more easily.

To master the art of agile development—or simply to use XP to be more successful—follow these steps:

1. Decide why you want to use agile development. Will it make your team and organization more successful? How? (For ideas, see "Enter Agility"" in Chapter 1.)

Teams new to XP often underapply its practices.

2. Determine whether this book's approach will work for your team. (See "Is XP Right for Us?"" in Chapter 4.)
3. Adopt as many of XP's practices as you can. (See Chapter 4.) XP's practices are self-reinforcing, so it works best when you use all of them together.
4. Follow the XP practices rigorously and consistently. (See Part II.) If a practice doesn't work, try following the book approach *more* closely. Teams new to XP often underapply its practices. Expect to take two or three months to start feeling comfortable with the practices and another two to six months for them to become second nature.
5. As you become confident that you are practicing XP correctly—again, give it several months—start experimenting with changes that *aren't* "by the book." (See Part III.) Each time you make a change, observe what happens and make further improvements.

Find a Mentor

As you adapt XP to your situation, you're likely to run into problems and challenges. I provide solutions for a wide variety of common problems, but you're still likely to encounter situations that I don't cover. For these situations, you need a *mentor*: an outside expert who has mastered the art of agile development.

NOTE
If you can get an expert to coach your team directly, that's even better. However, even master coaches benefit from an outside perspective when they encounter problems.

The hardest part of finding a mentor is finding someone with enough experience in agile development. Sources to try include:

- Other groups practicing XP in your organization
- Other companies practicing XP in your area
- A local XP/Agile user group
- XP/Agile consultants
- The XP mailing list: *extremeprogramming@yahoogroups.com*

I can't predict every problem you'll encounter, but I can help you see when things are going wrong. Throughout this book, I've scattered advice such as: "If you can't demonstrate progress weekly, it's a clear sign that your project is in trouble. Slow down for a week and figure out what's going wrong. Ask your mentor for help."

When I tell you to ask your mentor for help, I mean that the correct solution depends on the details of your situation. Your mentor can help you troubleshoot the problem and offer situation-specific advice.

CHAPTER 3

Understanding XP

"Welcome to the team, Pat," said Kim, smiling at the recent graduate. "Let me show you around. As I said during the interview, we're an XP shop. You may find that things are a little different here than you learned in school."

"I'm eager to get started," said Pat. "I took a software engineering course in school, and they taught us about the software development lifecycle. That made a lot of sense. There was a bit about XP, but it sounded like it was mostly about working in pairs and writing tests first. Is that right?"

"Not exactly," said Kim. "We do use pair programming, and we do write tests first, but there's much more to XP than that. Why don't you ask me some questions? I'll explain how XP is different than what you learned."

Pat thought for a moment. "Well, one thing I know from my course is that all development methods use the software development lifecycle: analysis, design, coding, and testing [see Figure 3-1]. Which phase are you in right now? Analysis? Design? Or is it coding or testing?"

"Yes!" Kim grinned. She couldn't help a bit of showmanship.

"I don't understand. Which is it?"

"All of them. We're working on analysis, design, coding, *and* testing. Simultaneously. Oh, and we deploy the software every week, too."

Pat looked confused. Was she pulling his leg?

Kim laughed. "You'll see! Let me show you around.

"This is our team room. As you can see, we all sit together in one big workspace. This helps us collaborate more effectively."

Kim led Pat over to a big whiteboard where a man stood frowning at dozens of index cards. "Brian, I'd like you to meet Pat, our new programmer. Brian is our product manager. What are you working on right now?"

"I'm trying to figure out how we should modify our release plan based on the feedback from the user meeting last week. Our users love what we've done so far, but they also have some really good suggestions. I'm trying to decide if their suggestions are worth postponing the feature we were planning to start next week. Our success has made us visible throughout the company, so requests are starting to flood in. I need to figure out how to keep us on track without alienating too many people."

"Sounds tough," Kim said. "So, would you say that you're working on requirements, then?"

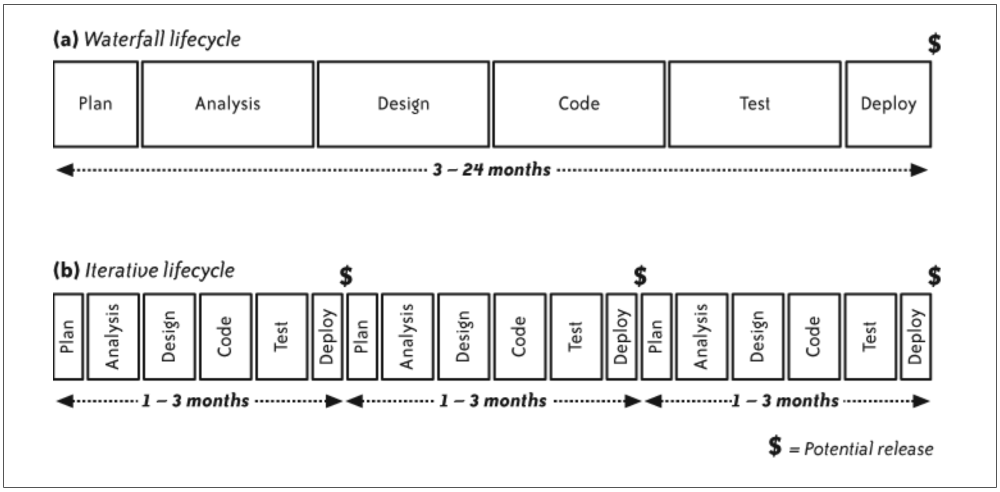


Figure 3-1. Traditional lifecycles

"I'm working on making our stakeholders happy," Brian shrugged, turning back to the whiteboard.

"Don't mind him," Kim whispered to Pat as they walked away. "He's under a lot of pressure right now. This whole project was his idea. It's already saved the company two and a half million dollars, but now there's some political stuff going on. Luckily, we programmers don't have to worry about that. Brian and Rachel take care of it—Rachel's our project manager."

"Wait... I thought Brian was the project manager?"

"No, Brian is the *product* manager. He's in charge of deciding what we build, with the help of stakeholders and other team members, of course. Rachel is the *project* manager—she helps things run smoothly. She helps management understand what we're doing and gets us what we need, like when she convinced Facilities to tear down the cubicle walls and give us this nice open workspace.

"Let me introduce you to some more members of the team," Kim continued, leading Pat over to two people sitting at a workstation. "This is Mary and Jeff. Mary is a mechanical engineer. She normally works in manufacturing, but we asked her to join us as an on-site customer for this project so she can help us understand the issues they face on the floor. Jeff is one of our testers. He's particularly good at finding holes in requirements. Guys, this is Pat, our new programmer."

Pat nodded hello. "I think I recognize what you're doing. That looks like a requirements document."

"Sort of," Jeff replied. "These are our customer tests for this iteration. They help us know if the software's doing what it's supposed to."

"Customer tests?" Pat asked.

Mary spoke up. "They're really examples. This particular set focuses on placement of inventory in the warehouse. We want the most frequently used inventory to be the easiest to access, but there are other concerns as well. We're putting in examples of different selections of inventory and how they should be stored."

"You can see how things are progressing," Jeff continued. "Here, I'll test these examples." He pressed a button on the keyboard, and a copy of the document popped up on the screen. Some sections of the document were green. Others were red.

"You can see that the early examples are green—that means the programmers have those working. These later ones are red because they cover special cases that the programmers haven't coded yet. And this one here is brand-new. Mary realized there were some edge cases we hadn't properly considered. You can see that some of these cases are actually OK—they're green—but some of them need more work. We're about to tell the programmers about them."

"Actually, you can go ahead and do that, Jeff," said Mary, as they heard a muffled curse from the area of the whiteboard. "It sounds like Brian could use my help with the release plan. Nice to meet you, Pat."

"Come on," said Jeff. "Kim and I will introduce you to the other programmers."

"Sure," said Pat. "But first—this document you were working on. Is it a requirements document or a test document?"

"Both," Jeff said, with a twinkle in his eye. "And neither. It's a way to make sure that we get the hard stuff right. Does it really matter what we call it?"

"You seem pretty casual about this," Pat said. "I did an internship last year and nobody at that company could even *think* about coding until the requirements and design plans were signed off. And here you are, adding features and revising your requirements right in the middle of coding!"

"It's just crazy enough to work," said Jeff.

"In other words," Kim added, "we used to have formal process gates and signoffs, too. We used to spend days arguing in meetings about the smallest details in our documents. Now, we focus on doing the right things right, not on what documents we've signed off. It takes a lot less work. Because we work on everything together, from requirements to delivery, we make fewer mistakes and can figure out problems much more easily."

"Things were different for me," Jeff added. "I haven't been here as long as Kim. In my last company, we didn't have any structure at all. People just did what they felt was right. That worked OK when we were starting out, but after a few years we started having terrible problems with quality. We were always under the gun to meet deadlines, and we were constantly running into surprises that prevented us from releasing on time. Here, although we're still able to do what we think is right, there's enough structure for everyone to understand what's going on and make constant progress."

"It's made our life easier," Kim said enthusiastically. "We get a lot more done..."

"*...and* it's higher quality," Jeff finished. "You've got to watch out for Kim—she'll never stop raving about how great it is to work together." He grinned. "She's right, you know. It is. Now let's go tell the other programmers about the new examples Mary and I added."

The XP Lifecycle

One of the most astonishing premises of XP is that you can eliminate requirements, design, and testing phases as well as the formal documents that go with them.

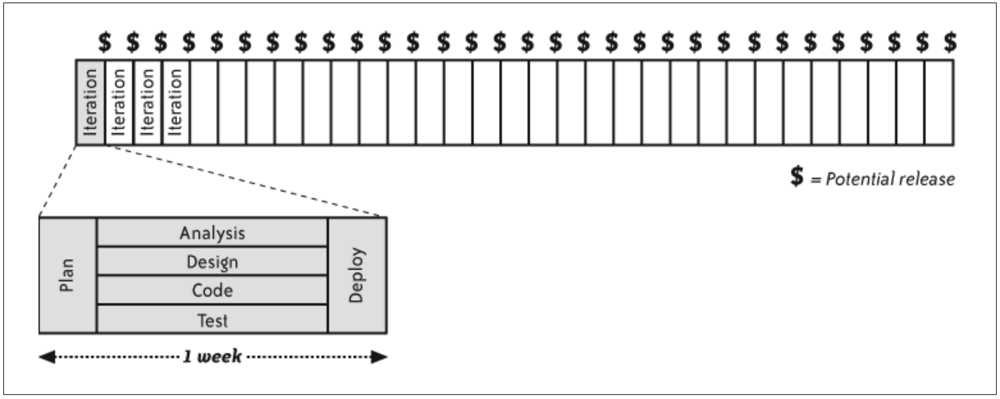


Figure 3-2. XP lifecycle

This premise is so far off from the way we typically learn to develop software that many people dismiss it as a delusional fantasy. "These XP folks obviously don't know what they're talking about," they say. "Just last month I was on a project that failed due to inadequate requirements and design. We need *more* requirements, design, and testing, not *less*!"

That's true. Software projects *do* need more requirements, design, and testing—which is why XP teams work on these activities every day. Yes, every day.

You see, XP emphasizes face-to-face collaboration. This is so effective in eliminating communication delays and misunderstandings that the team no longer needs distinct phases. This allows them to work on all activities every day—with *simultaneous phases*—as shown in Figure 3-2.

Using simultanous phases, an XP team produces deployable software every week. In each iteration, the team analyzes, designs, codes, tests, and deploys a subset of features.

Although this approach doesn't necessarily mean that the team is more productive,* it does mean that the team gets feedback much more frequently. As a result, the team can easily connect successes and failures to their underlying causes. The amount of unproven work is very small, which allows the team to correct some mistakes on the fly, as when coding reveals a design flaw, or when a customer review reveals that a user interface layout is confusing or ugly.

The tight feedback loop also allows XP teams to refine their plans quickly. It's much easier for a customer to refine a feature idea if she can request it and start to explore a working prototype within a few days. The same principle applies for tests, design, and team policy. Any information you learn in one phase can change the way you think about the rest of the software. If you find a design defect during coding or testing, you can use that knowledge as you continue to analyze requirements and design the system in subsequent iterations.

* Productivity is notoriously difficult to study. I'm not aware of any formal research on XP productivity, although anecdotal evidence indicates that agile teams are more productive than traditional teams.

How It Works

XP teams perform nearly every software development activity simultaneously. Analysis, design, coding, testing, and even deployment occur with rapid frequency.

That's a lot to do simultaneously. XP does it by working in *iterations*: week-long increments of work. Every week, the team does a bit of release planning, a bit of design, a bit of coding, a bit of testing, and so forth. They work on *stories*: very small features, or parts of features, that have customer value. Every week, the team commits to delivering four to ten stories. Throughout the week, they work on all phases of development for each story. At the end of the week, they deploy their software for internal review. (In some cases, they deploy it to actual customers.)

The following sections show how traditional phase-based activities correspond to an XP iteration.

Planning

Every XP team includes several business experts—the *on-site customers*—who are responsible for making business decisions. The on-site customers point the project in the right direction by clarifying the project vision, creating stories, constructing a release plan, and managing risks. Programmers provide estimates and suggestions, which are blended with customer priorities in a process called *the planning game*. Together, the team strives to create small, frequent releases that maximize value.

The planning effort is most intense during the first few weeks of the project. During the remainder of the project, customers continue to review and improve the vision and the release plan to account for new opportunities and unexpected events.

In addition to the overall release plan, the team creates a detailed plan for the upcoming week at the beginning of each iteration. The team touches base every day in a brief stand-up meeting, and its informative workspace keeps everyone informed about the project status.

Analysis

Rather than using an upfront analysis phase to define requirements, on-site customers sit with the team full-time. On-site customers may or may not be real customers depending on the type of project, but they are the people best qualified to determine what the software should do.

On-site customers are responsible for figuring out the requirements for the software. To do so, they use their own knowledge as customers combined with traditional requirements-gathering techniques. When programmers need information, they simply ask. Customers are responsible for organizing their work so they are ready when programmers ask for information. They figure out the general requirements for a story before the programmers estimate it and the detailed requirements before the programmers implement it.

Some requirements are tricky or difficult to understand. Customers formalize these requirements, with the assistance of testers, by creating *customer tests*: detailed, automatically checked examples. Customers and testers create the customer tests for a story around the same time that programmers implement the story. To assist in communication, programmers use a ubiquitous language in their design and code.

The user interface (UI) look and feel doesn't benefit from automated customer tests. For the UI, customers work with the team to create sketches of the application screens. In some cases, customers work alongside programmers as they use a UI builder to create a screen. Some teams include an interaction designer who's responsible for the application's UI.

Design and coding

XP uses incremental design and architecture to continuously create and improve the design in small steps. This work is driven by *test-driven development* (*TDD*), an activity that inextricably weaves together testing, coding, design, and architecture. To support this process, programmers work in pairs, which increases the amount of brainpower brought to bear on each task and ensures that one person in each pair always has time to think about larger design issues.

Programmers are also responsible for managing their development environment. They use a version control system for configuration management and maintain their own automated build. Programmers integrate their code every few hours and ensure that every integration is technically capable of deployment.

To support this effort, programmers also maintain coding standards and share ownership of the code. The team shares a joint aesthetic for the code, and everyone is expected to fix problems in the code regardless of who wrote it.

Testing

XP includes a sophisticated suite of testing practices. Each member of the team—programmers, customers, and testers—makes his own contribution to software quality. Well-functioning XP teams produce only a handful of bugs per month in completed work.

Programmers provide the first line of defense with test-driven development. TDD produces automated unit and integration tests. In some cases, programmers may also create end-to-end tests. These tests help ensure that the software does what the programmers intended.

Likewise, customer tests help ensure that the programmers' intent matches customers' expectations. Customers review work in progress to ensure that the UI works the way they expect. They also produce examples for programmers to automate that provide examples of tricky business rules.

Finally, testers help the team understand whether their efforts are in fact producing high-quality code. They use exploratory testing to look for surprises and gaps in the software. When the testers find a bug, the team conducts root-cause analysis and considers how to improve their process to prevent similar bugs from occuring in the future. Testers also explore the software's nonfunctional characteristics, such as performance and stability. Customers then use this information to decide whether to create additional stories.

The team *doesn't* perform any manual regression testing. TDD and customer testing leads to a sophisticated suite of automated regression tests. When bugs are found, programmers create automated tests to show that the bugs have been resolved. This suite is sufficient to prevent regressions. Every time programmers integrate (once every few hours), they run the entire suite of regression tests to check if anything has broken.

The team also supports their quality efforts through pair programming, energized work, and iteration slack. These practices enhance the brainpower that each team member has available for creating high-quality software.

Deployment

XP teams keep their software ready to deploy at the end of any iteration. They deploy the software to internal stakeholders every week in preparation for the weekly iteration demo. Deployment to real customers is scheduled according to business needs.

As long as the team is active, it maintains the software it has released. Depending on the organization, the team may also support its own software (a batman is helpful in this case; see "Iteration Planning"" in Chapter 8). In other cases, a separate support team may take over. Similarly, when the project ends, the team may hand off maintenance duties to another team. In this case, the team creates documentation and provides training as necessary during its last few weeks.

XP PRACTICES BY PHASE

The following table shows how XP's practices correspond to traditional phases. Remember that XP uses iterations rather than phases; teams perform every one of these activities each week. Most are performed every day.

Table 3-1. XP Practices by phase

XP Practices	Planning	Analysis	Design & Coding	Testing	Deployment
Thinking					
Pair Programming			•	•	
Energized Work	•	•	•	•	•
Informative Workspace	•				
Root-Cause Analysis	•			•	
Retrospectives	•			•	
Collaborating					
Trust	•	•	•	•	•
Sit Together	•	•	•	•	
Real Customer Involvement		•			
Ubiquitous Language		•			
Stand-Up Meetings	•				
Coding Standards			•		
Iteration Demo					•
Reporting	•	•	•	•	•
Releasing					
"Done Done"			•		•
No Bugs			•	•	
Version Control			•		
Ten-Minute Build			•		•

XP Practices	Planning	Analysis	Design & Coding	Testing	Deployment
Continuous Integration			•		•
Collective Code Ownership			•		
Documentation					•
Planning					
Vision	•	•			
Release Planning	•	•			
The Planning Game	•	•			
Risk Management	•				
Iteration Planning	•		•		
Slack	•		•		
Stories	•	•			
Estimating	•				
Developing					
Incremental Requirements		•	•		
Customer Tests		•		•	
Test-Driven Development			•	•	
Refactoring			•		
Simple Design			•		
Incremental Design and Architecture			•		
Spike Solutions			•		
Performance Optimization			•		
Exploratory Testing				•	

Our Story Continues

“Hey, guys, I’d like you to meet Pat, our new programmer,” Kim announced. She, Jeff, and Pat had walked over to a series of large tables. Index cards and pencils were scattered around the tables, and whiteboards covered the walls. Six programmers were working in pairs at three of the workstations.

“Hi, Pat,” the team chorused. Kim introduced everyone.

“Welcome aboard,” said Justin, one of the programmers. He stood up and grabbed a rubber chicken from a nearby desk. “Kevin and I are going to integrate our changes now,” he announced to the rest of the group. A few people nodded abstractly, already intent on their work.

"Mary and I just made some changes to the customer tests," said Jeff. "Who's working on the warehouse story?"

"That's us," said Justin. "What's up?"

"We added some examples to cover some new edge cases. I think they're pretty straightforward, but if you have any questions, let us know."

"Will do," Justin replied. "We were about to finish off the last of the business logic for that story anyway. We'll take a look at it as soon as we've checked in."

"Thanks!" Jeff went off to meet Mary and Brian at the planning board.

"Before you start on your next task, Justin, do you have a few minutes?" Kim asked.

Justin glanced at his pairing partner, Kevin, who was listening in. He gave Justin a thumbs up.

"Sure. We're just waiting for the build's tests to finish, anyway."

"Great," Kim said. "I'm helping Pat get oriented, and he wanted to know which phase of development we're working on. Can you explain what you've been doing?"

Justin flashed Pat a knowing look. "She's on the 'simultaneous phases' kick, huh? I'm sorry." He laughed. "Just kidding, Kim! It *is* pretty different.

"I'm sure Kim is dying to hear me say this: yes, we're doing testing, design, coding, and integration all at once. We deploy every week—because we do internal development, we actually deploy to production—but deployment happens automatically. The real deployment work happens when we update our deployment scripts, which we do as needed throughout the week."

"Right now Kevin and I are integrating, which is something everybody does every few hours. Before that, we were using test-driven development and refactoring to test, code, and design simultaneously." Justin pointed to another pair at a workstation. "Jerry and Suri are doing a in-depth review of a story we finished yesterday. Suri is our other tester—she's very good at exploratory testing." He looked at the third pair, who were talking intensely and sketching on the whiteboard. "And it looks like Mark and Allison over there are working on a larger design problem. We use incremental design and architecture, so we're constantly looking for ways to improve our design."

Allison looked up. "Actually, I think this is something the whole group should see. Can we get everyone's attention for a moment?"

"Sure," said Jerry, pushing away from his desk. He glanced at his partner, who nodded. "We needed a break anyway."

As Allison started sketching on the whiteboard, Pat thought about what he'd seen. Brian was working on planning. Jeff and Mary were working on requirements and tests—no, examples. Kevin and Justin were integrating, and had been testing, designing, and coding a few minutes earlier. Mark and Allison were designing. And Jerry and Suri were doing more testing.

Except that it wasn't quite that simple. Everything seemed fluid; people kept changing what they were working on. Jeff and Mary had gone backward, from requirements to planning, and all the programmers had jumped from integration and testing to design. He frowned. How was he going to keep track of it all?

"Don't worry," said Kim quietly. She had noticed his discomfort. "It's not as confusing as it seems. Jeff has it right: don't worry about what to call what we're doing. We're all doing the same thing—getting stuff done. We work together to make it happen, and we jump back and forth between all the normal development activities as the situation dictates. If you focus on what we're *delivering* rather than trying to figure out what phase of development we're in, it will all make sense."

Pat shook his head. "How do you know what you have to do, though? It looks like chaos."

Kim pointed at a whiteboard filled with index cards. "There's our plan for this week. Green are finished and white are remaining. It's Monday morning and we release on Wednesday, so we have about half a week to go. You tell me; what's our progress?"

Pat looked at the board. About half the cards had a green circle around them. "About halfway through?" he guessed.

Kim beamed. "Perfect! We keep those cards up-to-date. Just keep that in mind for the first week or two and you'll do fine!"

A LITTLE LIE

I've pretended that the only way to do XP is the way I've described it in this book. Actually, that's not really true. The essence of XP isn't its practices, but its approach to software development. [Beck 2004] describes XP as including a philosophy of software development, a body of practices, a set of complementary principles, and a community. Every experienced XP team will have its own way of practicing XP. As you master the art of agile development, you will, too.

In Parts I and II of this book, I'm going to perpetuate the little lie that there's just one way to do XP. It's a lot easier to learn XP that way. As you learn, keep in mind that no practice is set in stone. You're always free to experiment with changes. See "The Road to Mastery"" in Chapter 2 and then turn to Part III for guidance.

There are a few differences between my approach to XP and what you'll find in other XP books. Most of the differences are minor stylistic issues, but I've also made a few important changes to XP's planning and acceptance testing practices. I made these changes to address common problems I observed when working with XP teams. Like XP itself, these changes are the result of several years of iterative refinement.

One of the problems I've noticed with XP teams is that the on-site customers often have difficulty with release planning. XP gives them the freedom to change their mind at will, and they do—too much. They slip into *agile thrashing*, which means they overreact to every opportunity, changing direction every time something new comes up. As a result, the team never finishes anything valuable and has trouble showing progress.

To help prevent this problem, I've added the Vision practice and gone into more detail about appropriate release planning. I've also added the Risk Management practice to help teams understand how to make reliable long-term commitments.

I've also noticed that teams struggle with XP's automated acceptance tests. Although they are intended to "allow the customer to know when the system works and tell the programmers what needs to be done" [Jeffries et al.], I've found that they're often too limited to truly fulfill this role. In fact, some customers worry that, by defining acceptance tests, they'll be stuck with software that passes the letter of the tests but fails to fulfill the spirit of their expectations. Worse, most teams implement these tests as system-level tests that take a lot of work to create and maintain. They run slowly and lead to build and integration problems. In short, I've found that automated acceptance tests cost more than they're worth.

In this book, I've replaced automated acceptance tests with customer reviews, customer testing, and exploratory testing. Customer reviews involve customers throughout the development process, allowing them to communicate the spirit as well as the letter of their needs. Customer testing is a variant of automated acceptance testing that focuses on small, targeted tests rather than system-level tests. Its purpose is to communicate complicated business rules rather than confirming that a story has been completed properly. Exploratory testing provides an after-the-fact check on the team's practices. When the team is working well, they should produce nearly zero bugs. Exploratory testing helps the team have confidence that this is true.

Despite these changes, I've only added three brand-new practices in this book (Table 3-2). None of these practices are my invention; I adapted them from other methods and have proven them in real-world projects.

Table 3-2. Practices new to XP

New practice	Reason
Vision	Focuses efforts and helps counteract common "agile thrashing" problem
Risk Management	Improves team's ability to make and meet commitments; reduces costs
Exploratory Testing	Improves quality and helps integrate testers

However, there are several practices that mature XP teams practice intuitively, but aren't explicitly listed as practices in other XP books. I've added those as well (Table 3-3).

Table 3-3. Clarifying practices

Clarifying practice
Retrospectives
Trust
Ubiquitous Language
Stand-Up Meetings
Iteration Demo
Reporting
"Done Done"
No Bugs
Version Control
Documentation
Customer Testing
Spike Solutions
Performance Optimization

There are a few practices in standard XP that I've rarely seen in use and don't practice myself (Table 3-4). I've removed them from this book.

Table 3-4. Practices not in this book

Removed practice	Reason
Metaphor	Replaced with Ubiquitous Language
Shrinking Teams	No personal experience; not essential
Negotiated Scope Contract	Minimal personal experience; not essential
Pay-Per-Use	No personal experience; not essential

Table 3-5 shows how the practices in this book correspond to Beck's XP practices. I've also included a column for Scrum, another popular agile method.

Key: **n/a**: This practice is not part of the method, although some teams may add it. *implied*: Although this idea isn't a practice in the method, its presence is assumed or described in another way.

Table 3-5. Practices cross-reference

This Book	2nd Edition XP[a]	1st Edition XP[b]	Scrum[c]
Thinking			
Pair Programming	Pair Programming	Pair Programming	**n/a**
Energized Work	Energized Work	40-Hour Week	*implied*
Informative Workspace	Informative Workspace	*implied*	*implied*
Root-Cause Analysis	Root-Cause Analysis	*implied*	*implied*
Retrospectives	*implied*	*implied*	*implied*
Collaborating			
Trust	*implied*	*implied*	*implied*
(in Trust)	Team Continuity	**n/a**	*implied*
Sit Together	Sit Together	*implied*	Open Working Environment
implied	Whole Team	On-Site Customer	Scrum Team
Real Customer Involvmenet	Real Customer Involvement	*implied*	*implied*
Ubiquitous Language	*implied*	(replaces Metaphor)	**n/a**
Stand-Up Meetings	*implied*	*implied*	Daily Scrum
Coding Standards	*implied*	Coding Standards	**n/a**
Iteration Demo	*implied*	*implied*	Sprint Review
Reporting	*implied*	*implied*	*implied*
Releasing			
"Done Done"	*implied*	*implied*	*implied*
No Bugs	*implied*	*implied*	**n/a**

This Book	2nd Edition XP[a]	1st Edition XP[b]	Scrum[c]
Version Control	*implied*	*implied*	**n/a**
(in Version Control)	Single Code Base	*implied*	**n/a**
Ten-Minute Build	Ten-Minute Build	*implied*	**n/a**
Continuous Integration	Continuous Integration	Continuous Integration	**n/a**
Collective Code Ownership	Shared Code	Collective Ownership	**n/a**
Documentation	*implied*	*implied*	*implied*
Planning			
Vison	**n/a**	**n/a**	*implied*
Release Planning	Quarterly Cycle	Small Releases	Product Backlog
(in Release Planning)	Incremental Deployment	*implied*	*implied*
(in Release Planning)	Daily Deployment	*implied*	**n/a**
The Planning Game	*implied*	The Planning Game	*implied*
Risk Management	**n/a**	**n/a**	**n/a**
Iteration Planning	Weekly Cycle	*implied*	Sprints
Slack	Slack	*implied*	*implied*
Stories	Stories	*implied*	Backlog Items
Estimating	*implied*	*implied*	Estimating
Developing			
Incremental Requirements	*implied*	*implied*	*implied*
Customer Tests	*implied*	Testing	**n/a**
Test-Driven Development	Test-First Programming	Testing	**n/a**
Refactoring	*implied*	Refactoring	**n/a**
Simple Design	Incremental Design	Simple Design	**n/a**
Incremental Design and Architecture	Incremental Design	Simple Design	**n/a**
Spike Solutions	*implied*	*implied*	**n/a**
Performance Optimization	*implied*	*implied*	**n/a**
Exploratory Testing	**n/a**	**n/a**	**n/a**
(Not in This Book)			
n/a	Shrinking Teams	**n/a**	**n/a**
n/a	Negotiated Scope Contract	*implied*	**n/a**
n/a	Pay-Per-Use	**n/a**	**n/a**
implied	*implied*	*implied*	Scrum Master
implied	*implied*	*implied*	Product Owner

This Book	2nd Edition XP[a]	1st Edition XP[b]	Scrum[c]
n/a	**n/a**	**n/a**	Abnormal Sprint Termination
n/a	**n/a**	**n/a**	Sprint Goal

[a] [Beck 2004]

[b] [Beck 1999]

[c] [Schwaber & Beedle]

The XP Team

Working solo on your own project—"scratching your own itch"—can be a lot of fun. There are no questions about which features to work on, how things ought to work, if the software works correctly, or whether stakeholders are happy. All the answers are right there in one brain.

Team software development is different. The same information is spread out among many members of the team. Different people know:

- How to design and program the software (programmers, designers, and architects)
- Why the software is important (product manager)
- The rules the software should follow (domain experts)
- How the software should behave (interaction designers)
- How the user interface should look (graphic designers)
- Where defects are likely to hide (testers)
- How to interact with the rest of the company (project manager)
- Where to improve work habits (coach)

All of this knowledge is necessary for success. XP acknowledges this reality by creating *cross-functional teams* composed of diverse people who can fulfill all the team's roles.

The Whole Team

XP teams sit together in an open workspace. At the beginning of each iteration, the team meets for a series of activities: an iteration demo, a retrospective, and iteration planning. These typically take two to four hours in total. The team also meets for daily stand-up meetings, which usually take five to ten minutes each.

Other than these scheduled activities, everyone on the team plans his own work. That doesn't mean everybody works independently; they just aren't on an explicit schedule. Team members work out the details of each meeting when they need to. Sometimes it's as informal as somebody standing up and announcing across the shared workspace that he would like to discuss an issue. This *self-organization* is a hallmark of agile teams.

On-Site Customers

On-site customers—often just called *customers*—are responsible for defining the software the team builds. The rest of the team can and should contribute suggestions and ideas, but the customers are ultimately responsible for determining what stakeholders find valuable.

Customers' most important activity is release planning. This is a multifaceted activity. Customers need to evangelize the project's vision; identify features and stories; determine how to group features into small, frequent releases; manage risks; and create an achievable plan by coordinating with programmers and playing the planning game.

On-site customers may or may not be real customers, depending on the type of project. Regardless, customers are responsible for refining their plans by soliciting feedback from real customers and other stakeholders. One of the venues for this feedback is the weekly iteration demo, which customers lead.

In addition to planning, customers are responsible for providing programmers with requirements details upon request. XP uses requirements documents only as memory aids for customers. Customers themselves act as living requirements documents, researching information in time for programmer use and providing it as needed. Customers also help communicate requirements by creating mock-ups, reviewing work in progress, and creating detailed customer tests that clarify complex business rules. The entire team must sit together for this communication to take place effectively.

Typically, product managers, domain experts, interaction designers, and business analysts play the role of the on-site customer. One of the most difficult aspects of creating a cross-functional team is finding people qualified and willing to be on-site customers. Don't neglect this role; it's essential to increasing the value of the product you deliver. A great team will produce technically excellent software without on-site customers, but to truly succeed, your software must also bring value to its investors. This requires the perspective of on-site customers.

NOTE
Include exactly one product manager and enough other on-site customers for them to stay one step ahead of the programmers. As a rule of thumb, start with two on-site customers (including the product manager) for every three programmers.

[Coffin] describes an experience with two nearly identical teams, one that did not have on-site customers and one that did. The team with no on-site customers took fifteen months to produce a product with mediocre value:

> The total cost of the project exceeded initial expectations and the application under delivered on the user's functional expectations for the system... real business value was not delivered until the second and third [releases] and even then the new system was not perceived as valuable by its users because it required them to change while providing no significant benefit.

A team composed of many of the same developers, at the same company, using the same process, later produced a product with compelling value in less than three months:

> The first production release was ready after 9 weeks of development... it surpassed scheduling and functional expectations, while still coming in on-budget.... In the first

> two months of live production usage over 25,000 citations were entered using the new system. The application development team continued to deliver new releases to production approximately every six weeks thereafter. Every release was an exciting opportunity for the team of developers and users to provide value to the company and to improve the user's ability to accomplish their jobs.

One of the primary reasons for this success was customer involvement:

> Many of the shortcomings of the [first] system stemmed from a breakdown in the collaborative atmosphere that was initially established. Had users been more involved throughout the project, the end result would have been a system that much more closely aligned with their actual needs. They would have had a greater sense of ownership and communication between the various groups would have been less tense.
>
> ...
>
> The success of the [second] system caused many people in the organization to take note and embrace the lessons learned in this project... other projects teams restructured their physical arrangements into a shared project room as the [second] team had done.

Customer involvement makes a huge difference in product success. Make an extra effort to include customers. One way to do so is to move to their offices rather than asking them to move to your office. Make sure the customers agree and that there's adequate space available.

If the customers won't move to the team, move the team to the customers.

WHY SO MANY CUSTOMERS?

Two customers for every three programmers seems like a lot, doesn't it? Initially I started with a much smaller ratio, but I often observed customers struggling to keep up with the programmers. Eventually I arrived at the two-to-three ratio after trying different ratios on several successful teams. I also asked other XP coaches about their experiences. The consensus was that the two-to-three ratio was about right.

Most of those projects involved complex problem domains, so if your software is fairly straightforward, you may be able to have fewer customers. Keep in mind that customers have a lot of work to do. They need to figure out what provides the most value, set the appropriate priorities for the work, identify all the details that programmers will ask about, and fit in time for customer reviews and testing. They need to do all this while staying one step ahead of the programmers, who are right behind them, crunching through stories like freight trains. It's a big job. Don't underestimate it.

The product manager (aka product owner)

The product manager has only one job on an XP project, but it's a doozy. That job is to *maintain* and *promote* the product vision. In practice, this means documenting the vision, sharing it with stakeholders, incorporating feedback, generating features and stories, setting priorities for release planning, providing direction for the team's on-site customers, reviewing work in progress, leading iteration demos, involving real customers, and dealing with organizational politics.

> **NOTE**
> **In addition to maintaining and promoting the product vision, product managers are also often responsible for ensuring a successful deployment of the product to market. That may mean advertising and promotion, setting up training, and so forth. These ordinary product management responsibilities are out of the scope of this book.**

The best product managers have deep understandings of their markets, whether the market is one organization (as with custom software) or many (as with commercial software). Good product managers have an intuitive understanding of what the software will provide and why it's *the most important thing* their project teams can do with their time.

A great product manager also has a rare combination of skills. In addition to vision, she must have the authority to make difficult trade-off decisions about what goes into the product and what stays out. She must have the political savvy to align diverse stakeholder interests, consolidate them into the product vision, and effectively say "no" to wishes that can't be accommodated.

Product managers of this caliber often have a lot of demands on their time. You may have trouble getting enough attention. Persevere. Theirs is one of the most crucial roles on the team. Enlist the help of your project manager and remind people that software development is very expensive. If the software isn't valuable enough to warrant the time of a good product manager —a product manager who could mean the difference between success and failure—perhaps it isn't worth developing in the first place.

Make sure your product manager is committed to the project full-time. Once a team is running smoothly, the product manager might start cutting back on his participation. Although domain experts and other on-site customers can fill in for the product manager for a time, the project is likely to start drifting off-course unless the product manager participates in every iteration. [Rooney] experienced that problem, with regrettable results:

> We weren't sure what our priorities were. We weren't exactly sure what to work on next. We pulled stories from the overall list, but there was precious little from the Customer [product manager] in terms of what we should be working on. This went on for a few months.
>
> Then, we found out that the Gold Owner [executive sponsor] was pissed—really pissed. We hadn't been working on what this person thought we should.

In a predictable environment, and by delegating to a solid set of on-site customers, a product manager might be able to spend most of his time on other things, but he should still participate in every retrospective, every iteration demo, and most release planning sessions.

Some companies have a committee play the role of product manager, but I advise against this approach. The team needs a consistent vision to follow, and I've found that committees have trouble creating consistent, compelling visions. When I've seen committees succeed, it's been because one committee member acted as *de facto* product manager. I recommend that you explicitly find a product manager. Her role may be nothing more than consolidating the ideas of the committee into a single vision, and that's likely to keep her hands full. Be sure to choose a product manager with plenty of political acumen in this case.

Domain experts (aka subject matter experts)

Most software operates in a particular industry, such as finance, that has its own specialized rules for doing business. To succeed in that industry, the software must implement those rules faithfully and exactly. These rules are *domain rules*, and knowledge of these rules is *domain knowledge*.

Most programmers have gaps in their domain knowledge, even if they've worked in an industry for years. In many cases, the industry itself doesn't clearly define all its rules. The basics may be clear, but there are nitpicky details where domain rules are implicit or even contradictory.

The team's domain experts are responsible for figuring out these details and having the answers at their fingertips. *Domain experts*, also known as *subject matter experts*, are experts in their field. Examples include financial analysts and PhD chemists.

Domain experts spend most of their time with the team, figuring out the details of upcoming stories and standing ready to answer questions when programmers ask. For complex rules, they create customer tests (often with the help of testers) to help convey nuances.

NOTE
On small teams, product managers often double as domain experts.

Interaction designers

The user interface is the public face of the product. For many users, the UI *is* the product. They judge the product's quality solely on their perception of the UI.

Interaction designers help define the product UI. Their job focuses on understanding users, their needs, and how they will interact with the product. They perform such tasks as interviewing users, creating user personas, reviewing paper prototypes with users, and observing usage of actual software.

NOTE
Don't confuse graphic design with interaction design. *Graphic designers* convey ideas and moods via images and layout. *Interaction designers* focus on the types of people using the product, their needs, and how the product can most seamlessly meet those needs.

You may not have a professional interaction designer on staff. Some companies fill this role with a graphic designer, the product manager, or a programmer.

Interaction designers divide their time between working with the team and working with users. They contribute to release planning by advising the team on user needs and priorities. During each iteration, they help the team create mock-ups of UI elements for that iteration's stories.

As each story approaches completion, they review the look and feel of the UI and confirm that it works as expected.

The fast, iterative, feedback-oriented nature of XP development leads to a different environment than interaction designers may be used to. Rather than spending time researching users and defining behaviors before development begins, interaction designers must iteratively refine their models concurrently with iterative refinement of the program itself.

Although interaction design is different in XP than in other methods, it is not necessarily diminished. XP produces working software every week, which provides a rich grist for the interaction designer's mill. Designers have the opportunity to take real software to users, observe their usage patterns, and use that feedback to effect changes as soon as one week later.

Business analysts

On nonagile teams, business analysts typically act as liaisons between the customers and developers, by clarifying and refining customer needs into a functional requirements specification.

On an XP team, business analysts augment a team that already contains a product manager and domain experts. The analyst continues to clarify and refine customer needs, but the analyst does so in support of the other on-site customers, not as a replacement for them. Analysts help customers think of details they might otherwise forget and help programmers express technical trade-offs in business terms.

Programmers

A great product vision requires solid execution. The bulk of the XP team consists of software developers in a variety of specialties. Each of these developers contributes directly to creating working code. To emphasize this, XP calls all developers *programmers*.

NOTE

Include between 4 and 10 programmers. In addition to the usual range of expertise, be sure to include at least one senior programmer, designer, or architect who has significant design experience and is comfortable working in a hands-on coding environment. This will help the team succeed at XP's incremental design and architecture.

If the customers' job is to maximize the value of the product, then the programmers' job is to minimize its cost. Programmers are responsible for finding the most effective way of delivering the stories in the plan. To this end, programmers provide effort estimates, suggest alternatives, and help customers create an achievable plan by playing the planning game.

Programmers spend most of their time pair programming. Using test-driven development, they write tests, implement code, refactor, and incrementally design and architect the application. They pay careful attention to design quality, and they're keenly aware of technical debt (for an explanation of technical debt, see "XP Concepts"" later in this chapter) and its impact on development time and future maintenance costs.

Programmers also ensure that the customers may choose to release the software at the end of any iteration. With the help of the whole team, the programmers strive to produce no bugs in completed software. They maintain a ten-minute build that can build a complete release package at any time. They use version control and practice continuous integration, keeping all but the last few hours' work integrated and passing its tests.

This work is a joint effort of all the programmers. At the beginning of the project, the programmers establish coding standards that allow them to collectively share responsibility for the code. Programmers have the right and the responsibility to fix any problem they see, no matter which part of the application it touches.

Programmers rely on customers for information about the software to be built. Rather than guessing when they have a question, they ask one of the on-site customers. To enable these conversations, programmers build their software to use a ubiquitous language. They assist in customer testing by automating the customers' examples.

Finally, programmers help ensure the long-term maintainability of the product by providing documentation at appropriate times.

Designers and architects

Everybody codes on an XP team, and everybody designs. Test-driven development combines design, tests, and coding into a single, ongoing activity.

Expert designers and architects are still necessary. They contribute by guiding the team's incremental design and architecture efforts and by helping team members see ways of simplifying complex designs. They act as peers—that is, as programmers—rather than teachers, guiding rather than dictating.

Technical specialists

In addition to the obvious titles (programmer, developer, software engineer), the XP "programmer" role includes other software development roles. The programmers could include a database designer, a security expert, or a network architect. XP programmers are generalizing specialists. Although each person has his own area of expertise, everybody is expected to work on any part of the system that needs attention. (See "Collective Code Ownership"" in Chapter 7 for more.)

Testers

Testers help XP teams produce quality results from the beginning. Testers apply their critical thinking skills to help customers consider all possibilities when envisioning the product. They help customers identify holes in the requirements and assist in customer testing.*

NOTE

Include enough testers for them to stay one step ahead of the programmers. As a rule of thumb, start with one tester for every four programmers.

* This discussion of tester responsibilities is part of my variant of XP (see the sidebar "A Little Lie"," earlier in this chapter). Classic XP doesn't include testers as a distinct role.

Testers also act as technical investigators for the team. They use exploratory testing to help the team identify whether it is successfully preventing bugs from reaching finished code. Testers also provide information about the software's nonfunctional characteristics, such as performance, scalability, and stability, by using both exploratory testing and long-running automated tests.

However, testers *don't* exhaustively test the software for bugs. Rather than relying on testers to find bugs for programmers to fix, the team should produce nearly bug-free code on their own. When testers find bugs, they help the rest of the team figure out what went wrong so that the team as a whole can prevent those kinds of bugs from occurring in the future.

These responsibilities require creative thinking, flexibility, and experience defining test plans. Because XP automates repetitive testing rather than performing manual regression testing, testers who are used to self-directed work are the best fit.

Some XP teams don't include dedicated testers. If you don't have testers on your team, programmers and customers should share this role.

WHY SO FEW TESTERS?

As with the customer ratio, I arrived at the one-to-four tester-to-programmer ratio through trial and error. In fact, that ratio may be a little high. Successful teams I've worked with have had ratios as low as one tester for every six programmers, and some XP teams have no testers at all.

Manual script-based testing, particularly regression testing, is extremely labor-intensive and requires high tester-to-programmer ratios. XP doesn't use this sort of testing. Furthermore, programmers create most of the automated tests (during test-driven development), which further reduces the need for testers.

If you're working with existing code and have to do a lot of manual regression testing, your tester-to-programmer ratio will probably be higher than I've suggested here.

Coaches

XP teams *self-organize*, which means each member of the team figures out how he can best help the team move forward at any given moment. XP teams eschew traditional management roles.

Instead, XP leaders lead by example, helping the team reach its potential rather than creating jobs and assigning tasks. To emphasize this difference, XP leaders are called *coaches*. Over time, as the team gains experience and self-organizes, explicit leadership becomes less necessary and leadership roles dynamically switch from person to person as situations dictate.

A coach's work is subtle; it enables the team to succeed. Coaches help the team start their process by arranging for a shared workspace and making sure that the team includes the right people. They help set up conditions for energized work, and they assist the team in creating an informative workspace.

One of the most important things the coaches can do is to help the team interact with the rest of the organization. They help the team generate organizational trust and goodwill, and they often take responsibility for any reporting needed.

Coaches also help the team members maintain their self-discipline, helping them remain in control of challenging practices such as risk management, test-driven development, slack, and incremental design and architecture.

NOTE

The coach differs from your mentor (see "Find a Mentor"" in Chapter 2). Your mentor is someone outside the team who you can turn to for advice.

The programmer-coach

Every team needs a programmer-coach to help the other programmers with XP's technical practices. Programmer-coaches are often senior developers and may have titles such as "technical lead" or "architect." They can even be functional managers. While some programmer-coaches make good all-around coaches, others require the assistance of a project manager.

Programmer-coaches also act as normal programmers and participate fully in software development.

The project manager

Project managers help the team work with the rest of the organization. They are usually good at coaching nonprogramming practices. Some functional managers fit into this role as well. However, most project managers lack the technical expertise to coach XP's programming practices, which necessitates the assistance of a programmer-coach.

Project managers may also double as customers.

NOTE

Include a programmer-coach and consider including a project manager.

Other Team Members

The preceding roles are a few of the most common team roles, but this list is by no means comprehensive. The absence of a role does not mean the expertise is inappropriate for an XP team; an XP team should include exactly the expertise necessary to complete the project successfully and cost-effectively. For example, one team I worked with included a technical writer and an ISO 9001 analyst.

The Project Community

Projects don't live in a vaccuum; every team has an ecosystem surrounding it. This ecosystem extends beyond the team to the *project community*, which includes everyone who affects or is affected by the project.* Keep this community in mind as you begin your XP project, as everybody within it can have an impact on your success.

Two members of your project community that you may forget to consider are your organization’s Human Resources and Facilities departments. Human Resources often handles performance reviews and compensation. Their mechanisms may not be compatible with XP’s team-based effort (see “Trust”” in Chapter 6). Similarly, in order to use XP, you’ll need the help of Facilities to create an open workspace (see “Sit Together”” in Chapter 6).

Stakeholders

Stakeholders form a large subset of your project community. Not only are they affected by your project, they have an active interest in its success. Stakeholders may include end users, purchasers, managers, and executives. Although they don’t participate in day-to-day development, do invite them to attend each iteration demo. The on-site customers—particularly the product manager—are responsible for understanding the needs of your stakeholders, deciding which needs are most important, and knowing how to best meet those needs.

The executive sponsor

The executive sponsor is particularly important: he holds the purse strings for your project. Take extra care to identify your executive sponsor and understand what he wants from your project. He’s your ultimate customer. Be sure to provide him with regular demos and confirm that the project is proceeding according to his expectations.

XP PRACTICES BY ROLE

The following table shows the practices you should learn to practice XP. You can always learn more, of course! In particular, if you’re in a leadership role (or would like to be), you should study all the practices.

Table 3-6. XP Practices by role

XP Practices	On-Site Customers	Programmers	Testers	Coaches
Thinking				
Pair Programming		●		
Energized Work	•	•	•	●
Informative Workspace	•	•	•	●
Root-Cause Analysis	•	•	•	●
Retrospectives	•	•	•	●
Collaborating				
Trust	•	•	•	●

• = You will be involved with this practice. Studying it will be helpful, but not necessary.

● = You should study this practice carefully.

* Thanks to David Schmaltz and Amy Schwartz of True North pgs, Inc., for this term.

XP Practices	On-Site Customers	Programmers	Testers	Coaches
Sit Together	•	•	•	●
Real Customer Involvement	•			●
Ubiquitous Language		●		
Stand-Up Meetings	•	•	•	●
Coding Standards		•		●
Iteration Demo	●	●	•	•
Reporting	•	•	•	●
Releasing				
"Done Done"	●	●	●	●
No Bugs	•	●	●	●
Version Control	•	●	•	
Ten-Minute Build		●		
Continuous Integration		●		
Collective Code Ownership		●		
Documentation	•	•		●
Planning				
Vision	●			●
Release Planning	●			●
The Planning Game	•	•		●
Risk Management	•	•	•	●
Iteration Planning	•	●		●
Slack		•		●
Stories	•	•	•	●
Estimating		●		
Developing				
Incremental Requirements	●		●	●
Customer Tests	●	●	●	
Test-Driven Development		●	•	
Refactoring		●		
Simple Design		●		
Incremental Design and Architecture		●		
Spike Solutions		●		
Performance Optimization	•	●		

• = You will be involved with this practice. Studying it will be helpful, but not necessary.

● = You should study this practice carefully.

XP Practices	On-Site Customers	Programmers	Testers	Coaches
Exploratory Testing			•	

• = You will be involved with this practice. Studying it will be helpful, but not necessary.

• = You should study this practice carefully.

Filling Roles

The exact structure of your team isn't that important as long as it has all the knowledge it needs. The makeup of your team will probably depend more on your organization's traditions than on anything else.

In other words, if project managers and testers are typical for your organization, include them. If they're not, you don't necessarily need to hire them. You don't have to have one person for each role—some people can fill multiple roles. Just keep in mind that someone has to perform those duties even if no one has a specific job title saying so.

At a minimum, however, I prefer to see one person clearly designated as "product manager" (who may do other customer-y things) and one person clearly defined as "programmer-coach" (who also does programmer-y things).

The other roles may blend together. Product managers are usually domain experts and can often fill the project manager's shoes, too. One of the customers may be able to play the role of interaction designer, possibly with the help of a UI programmer. On the programming side, many programmers are generalists and understand a variety of technologies. In the absence of testers, both programmers and customers should pick up the slack.

Team Size

The guidelines in this book assume teams with 4 to 10 programmers (5 to 20 total team members). For new teams, four to six programmers is a good starting point.

Applying the staffing guidelines to a team of 6 programmers produces a team that also includes 4 customers, 1 tester, and a project manager, for a total team size of 12 people. Twelve people turns out to be a natural limit for team collaboration.

XP teams can be as small as one experienced programmer and one product manager, but full XP might be overkill for such a small team. The smallest team I would use with full XP consists of five people: four programmers (one acting as coach)and one product manager (who also acts as project manager, domain expert, and tester). A team of this size might find that the product manager is overburdened; if so, the programmers will have to pitch in. Adding a domain expert or tester will help.

On the other end of the spectrum, starting with 10 programmers produces a 20-person team that includes 6 customers, 3 testers, and a project manager. You can create even larger XP teams, but they require special practices that are out of the scope of this book.

Before you scale your team to more than 12 people, however, remember that large teams incur extra communication and process overhead, and thus reduce individual productivity. The combined overhead might even reduce overall productivity. If possible, hire more experienced, more productive team members rather than scaling to a large team.

Prefer *better* to *bigger*.

A 20-person team is advanced XP. Avoid creating a team of this size until your organization has had extended success with a smaller team. If you're working with a team of this size, continuous review, adjustment, and an experienced coach are critical.

Full-Time Team Members

All the team members should sit with the team full-time and give the project their complete attention. This particularly applies to customers, who are often surprised by the level of involvement XP requires of them.

Some organizations like to assign people to multiple projects simultaneously. This *fractional assignment* is particularly common in *matrix-managed organization*s. (If team members have two managers, one for their project and one for their function, you are probably in a matrixed organization.)

If your company practices fractional assignment, I have some good news. You can instantly improve productivity by reassigning people to only one project at a time. Fractional assignment is dreadfully counterproductive: fractional workers don't bond with their teams, they often aren't around to hear conversations and answer questions, and they must task switch, which incurs a significant hidden penalty. "[T]he minimum penalty is 15 percent... Fragmented knowledge workers may look busy, but a lot of their busyness is just thrashing" [DeMarco 2002] (p. 19–20).

Fractional assignment is dreadfully counterproductive.

NOTE
If your team deals with a lot of ad hoc requests, you may benefit from using a batman, discussed in "Iteration Planning"" in Chapter 8.

That's not to say everyone needs to work with the team for the entire duration of the project. You can bring someone in to consult on a problem temporarily. However, while she works with the team, she should be fully engaged and available.

XP Concepts

As with any specialized field, XP has its own vocabulary. This vocabulary distills several important concepts into snappy descriptions. Any serious discussion of XP (and of agile in general) uses this vocabulary. Some of the most common ideas follow.

Refactoring

There are multiple ways of expressing the same concept in source code. Some are better than others. *Refactoring* is the process of changing the structure of code—rephrasing it—without changing its meaning or behavior. It's used to improve code quality, to fight off software's unavoidable entropy, and to ease adding new features.

Ally

Refactoring (p. 306)

Technical Debt

Imagine a customer rushing down the hallway to your desk. "It's a bug!" she cries, out of breath. "We have to fix it now." You can think of two solutions: the right way and the fast way. You just *know* she'll watch over your shoulder until you fix it. So you choose the fast way, ignoring the little itchy feeling that you're making the code a bit messier.

Technical debt is the total amount of less-than-perfect design and implementation decisions in your project. This includes quick and dirty hacks intended just to get something working *right now!* and design decisions that may no longer apply due to business changes. Technical debt can even come from development practices such as an unwieldy build process or incomplete test coverage. It lurks in gigantic methods filled with commented-out code and "TODO: not sure why this works" comments. These dark corners of poor formatting, unintelligible control flow, and insufficient testing breed bugs like mad.

The bill for this debt often comes in the form of higher maintenance costs. There may not be a single lump sum to pay, but simple tasks that ought to take minutes may stretch into hours or afternoons. You might not even notice it except for a looming sense of dread when you read a new bug report and suspect it's in *that* part of the code.

Left unchecked, technical debt grows to overwhelm software projects. Software costs millions of dollars to develop, and even small projects cost hundreds of thousands. It's foolish to throw away that investment and rewrite the software, but it happens all the time. Why? Unchecked technical debt makes the software more expensive to modify than to reimplement. What a waste.

XP takes a fanatical approach to technical debt. The key to managing it is to be constantly vigilant. Avoid shortcuts, use simple design, refactor relentlessly... in short, apply XP's development practices (see Chapter 9).

Timeboxing

Some activities invariably stretch to fill the available time. There's always a bit more polish you can put on a program or a bit more design you can discuss in a meeting. Yet at some point you need to make a decision. At some point you've identified as many options as you ever will.

Recognizing the point at which you have enough information is not easy. If you use *timeboxing,* you set aside a specific block of time for your research or discussion and stop when your time is up, regardless of your progress.

This is both difficult and valuable. It's difficult to stop working on a problem when the solution may be seconds away. However, recognizing when you've made as much progress as possible is an important time-management skill. Timeboxing meetings, for example, can reduce wasted discussion.

The Last Responsible Moment

XP views a potential change as an opportunity to exploit; it's the chance to learn something significant. This is why XP teams delay commitment until the *last responsible moment*.[*]

Note that the phrase is the last *responsible* moment, not the last *possible* moment. As [Poppendieck & Poppendieck] says, make decisions at "the moment at which failing to make a decision eliminates an important alternative. If commitments are delayed beyond the last responsible moment, then decisions are made by default, which is generally not a good approach to making decisions."

By delaying decisions until this crucial point, you increase the accuracy of your decisions, decrease your workload, and decrease the impact of changes. Why? A delay gives you time to increase the amount of information you have when you make a decision, which increases the likelihood it is a correct decision. That, in turn, decreases your workload by reducing the amount of rework that results from incorrect decisions. Changes are easier because they are less likely to invalidate decisions or incur additional rework.

See "Release Planning"" in Chapter 8 for an example of applying this concept.

Stories

Stories represent self-contained, individual elements of the project. They tend to correspond to individual features and typically represent one or two days of work.

Stories are customer-centric, describing the results in terms of business results. They're not implementation details, nor are they full requirements specifications. They are traditionally just an index card's worth of information used for scheduling purposes. See "Stories"" in Chapter 8 for more information.

Iterations

An *iteration* is the full cycle of design-code-verify-release practiced by XP teams. It's a timebox that is usually one to three weeks long. (I recommend one-week iterations for new teams; see "Iteration Planning"" in Chapter 8) Each iteration begins with the customer selecting which stories the team will implement during the iteration, and it ends with the team producing software that the customer can install and use.

The beginning of each iteration represents a point at which the customer can change the direction of the project. Smaller iterations allow more frequent adjustment. Fixed-size iterations provide a well-timed rhythm of development.

Though it may seem that small and frequent iterations contain a lot of planning overhead, the amount of planning tends to be proportional to the length of the iteration.

See "Iteration Planning"" for more details about XP iterations.

* The Lean Construction Institute coined the term "last responsible moment." [Poppendieck & Poppendieck] popularized it in relation to software development.

Velocity

In well-designed systems, programmer estimates of effort tend to be *consistent* but not *accurate*. Programmers also experience interruptions that prevent effort estimates from corresponding to calendar time. *Velocity* is a simple way of mapping estimates to the calendar. It's the total of the estimates for the stories finished in an iteration.

In general, the team should be able to achieve the same velocity in every iteration. This allows the team to make iteration commitments and predict release dates. The units measured are deliberately vague; velocity is a technique for converting effort estimates to calendar time and has no relation to productivity. See "Velocity"" in Chapter 8 for more information.

Theory of Constraints

[Goldratt 1992]'s *Theory of Constraints* says,in part, that every system has a single constraint that determines the overall throughput of the system. This book assumes that programmers are the constraint on your team. Regardless of how much work testers and customers do, many software teams can only complete their projects as quickly as the programmers can program them. If the rest of the team outpaces the programmers, the work piles up, falls out of date and needs reworking, and slows the programmers further.

Therefore, the programmers set the pace, and their estimates are used for planning. As long as the programmers are the constraint, the customers and testers will have more slack in their schedules, and they'll have enough time to get their work done before the programmers need it.

Although this book assumes that programmers are the constraint, they may not be. Legacy projects in particular sometimes have a constraint of testing, not programming. The responsibility for estimates and velocity always goes to the constraint: in this case, the testers. Programmers have less to do than testers and manage their workload so that they are finished by the time testers are ready to test a story.

What should the nonconstraints do in their spare time? Help eliminate the constraint. If testers are the constraint, programmers might introduce and improve automated tests.

Mindfulness

Agility—the ability to respond effectively to change—requires that everyone pay attention to the process and practices of development. This is *mindfulness*.

Sometimes pending changes can be subtle. You may realize your technical debt is starting to grow when adding a new feature becomes more difficult this week than last week. You may notice the amount and tone of feedback you receive from your customers change.

XP offers plenty of opportunities to collect feedback from the code, from your coworkers, and from every activity you perform. Take advantage of these. Pay attention. See what changes and what doesn’t, and discuss the results frequently.

CHAPTER 10

Values and Principles

Until now, I've talked about a very specific approach to agility: one style of applying XP's practices. That's only the beginning.

No process is perfect. Every approach to development has some potential for improvement. Ultimately, your goal is to remove every barrier between your team and the success of your project, and fluidly adapt your approach as conditions change. *That* is agility.

To master the art of agile development, you need experience and mindfulness. *Experience* helps you see how agile methods work. *Mindfulness* helps you understand your experiences. Experience again allows you to experiment with changes. Mindfulness again allows you to reflect on why your experiments worked—or didn't work—in practice. Experience and mindfulness, endlessly joined, are the path to mastery.

So far, this book has focused on experience. Before you can reflect on how agile methods work, you need to experience an agile method working. With its emphasis on practicing XP, this book has given you the tools to do so.

Yet practicing XP isn't enough—you need mindfulness, too. You must pay attention to what happens around you. You must think about what's happening and, more importantly, *why* it's happening. Ask questions. Make small changes. Observe the results. I can't teach you mindfulness; only you have the power to do it.

I can, however, give you some things to think about as you learn. The XP practices are a manifestation of deeper agile values and principles. Think about these as you use XP. When your situation changes, use the values and principles to guide you in changing your practices.

Commonalities

Can any set of principles really represent agile development? After all, agility is just an umbrella term for a variety of methods, most of which came about long before the term "agile" was coined.

The answer is yes: agile methods do share common values and principles. In researching this part of the book, I collected over a hundred different values and principles from several agile sources.* They formed five themes: Improve the Process, Rely on People, Eliminate Waste, Deliver Value, and Seek Technical Excellence. Each is compatible with any of the specific agile methods.

* [Beck et al.], [Beck 1999], [Beck 2004], [Cockburn], [Highsmith], [Poppendieck & Poppendieck], [Schwaber & Beedle], [Subramaniam & Hunt], [Shore 2005b], and [Stapleton].

The following chapters explain these themes in terms of principles and practices. Each chapter includes anecdotes about applying the principles to situations beyond standard XP. Where possible, I've borrowed existing names for specific principles.

About Values, Principles, and Practices

Values are ideals. They're abstract, yet identifiable and distinct. For example, XP's values are:

Courage
: To make the right decisions, even when they're difficult, and to tell stakeholders the truth when they need to hear it

Communication
: To give the right people the right information when they can use it to its maximum advantage

Simplicity
: To discard the things we want but don't actually need

Feedback
: To learn the appropriate lessons at every possible opportunity

Respect
: To treat ourselves and others with dignity, and to acknowledge expertise and our mutual desire for success

Principles are applications of those ideals to an industry. For example, the value of simplicity leads us to focus on the essentials of development. As [Beck 2004] puts it, this principle is to "travel light." [Cockburn] says, "Excess methodology weight is costly," and "Discipline, skills, and understanding counter process, formality, and documentation."

Practices are principles applied to a specific type of project. XP's practices, for example, call for colocated teams of up to 20 people. "Sit Together"" in Chapter 6 and "The Whole Team"" in Chapter 3 embody the principles of simplicity because face-to-face communication reduces the need for formal requirements documentation.

Further Reading

The following books are excellent resources that go into far more detail than I have room for here. Each has a different focus, so they complement each other nicely.

Agile Software Development [Cockburn] focuses on the "individuals and interactions" aspect of agile development. Most of his thoughts correspond to the principles I outline in Chapter 12.

Lean Software Development: An Agile Toolkit for Software Development Managers [Poppendieck & Poppendieck] applies concepts from Lean Manufacturing to agile development, with emphasis on the principles that I describe in Chapter 13 and Chapter 14.

Agile Management for Software Engineering [Anderson 2003] is a bit dense, but it has detailed coverage of the Theory of Constraints, which I discuss in "Pursue Throughput"" in Chapter 13.

Practices of an Agile Developer [Subramaniam & Hunt] is an easy-to-read collection of guidelines and advice for agile developers, similar to what I discuss in Chapter 15.

Agile Software Development Ecosystems [Highsmith] looks at several agile methods through the lens of people and principles. Read this if you want to understand agile development in depth.

Extreme Programming Explained [Beck 1999], [Beck 2004] discusses the thought process behind XP. It will give you more insight into how agile principles relate to the XP practices. Try to get both editions, if you can; they describe the same basic process from very different perspectives. Studying both books and looking for the underlying commonalities will teach you a lot about agile values and principles.

CHAPTER 11

Improve the Process

Agile methods are more than a list of practices to follow. When your team has learned how to perform them effectively, you can become a great team by using the practices to modify your process.

Throughout this book, I've drawn attention to places where the way you perform XP may vary from how I explain it. No two teams are exactly alike. You'll do some things differently because you have different people and different needs. As you master the art of agile development, you'll learn how and when to modify your process to take advantage of your specific situation and opportunities.

Understand Your Project

To improve your process, you must understand how it affects your project. You need to take advantage of feedback—from the code, from the team, from customers and stakeholders—so you can understand what works well and what doesn't. Always pay attention to what's happening around you. Ask "why": why do we follow this practice? Why is this practice working? Why *isn't* this practice working?

Ask team members for their thoughts. There's an element of truth in every complaint, so encourage open discussion. As a team, reflect on what you've learned. When you discover something new, be a mentor; when you have questions, ask a mentor. Help each other understand what you're doing and why.

In Practice

XP is full of feedback loops—giant conduits of information that you should use to improve your work. Root-cause analysis and retrospectives clearly improve the team's understanding, and other practices reinforce the principles in more subtle ways.

For example, sitting and working together as a whole team gives team members opportunities to observe and absorb information. This is tactical feedback, and it can reveal strategic issues when something unexpected happens. Stand-up meetings and the informative workspace contribute to an information-rich environment.

Perhaps counterintuitively, the practices of energized work, slack, and pair programming also spread useful information. When team members are under pressure, they have trouble thinking about ways they can improve their work. Energized work and slack reduce that pressure. Pair programming gives one person in each pair time to think about strategy.

Test-driven development, exploratory testing, real customer involvement, iteration demos, and frequent releases all provide information about the project, from code to user response.

Beyond Practices

Large and successful free and open source software projects often see a lot of turnover. In the five years I've worked on one of these projects, dozens of people have come and gone. That's normal, especially for volunteers. It can take some time and effort to manage that change in personnel.

Recently, the project lead and our most prolific contributor had to reduce their time commitments. It took us a few months to realize we had lost ground, especially because no one had taken over the project lead's job of producing timely releases. With everyone else reviewing changes and implementing features, our process assumed the lead was still making releases.

This experience helped us see that tying such an important task to a single person was a mistake. To fix it, we all agreed to divide the work. Every month, one of us takes final responsibility for creating the release; this person produces the final tested bundle, makes the appropriate announcements, and uploads the bundle to the master distribution site. With six people available and a release schedule planned to the day, we've found a much healthier rhythm for making regular releases. If one or more developers are unavailable, several other people can perform the same function. It's worked—we've regained our velocity and started to attract new developers again.

Tune and Adapt

When you see the need for a change, modify your process. Make the change for your team alone; though your team may be one of many, it's OK to do things differently. Every team's needs are different.

These changes require tuning. Think of them as experiments; make small, isolated changes that allow you to understand the results. Be specific about your expectations and about the measurements for judging success. These changes are sources of feedback and learning. Use the results of your experiments to make further changes. Iterate until you're satisfied with the results.

Some experiments will fail, and others may actually make the process worse. Team members need to be flexible and adaptive. Your team needs to have the courage to experiment and occasionally fail.

Changing your process requires you to have a holistic view of what you do and why. New agile teams should be cautious about changing their process, as they don't yet have the experience necessary to give them that holistic understanding. Once you have the experience, use the feedback from your changes to improve your process and your understanding of agility.

In Practice

Tuning and adapting is implicit in XP; teams are supposed to make changes whenever they have a reason to do so. Many XP teams use retrospectives to give themselves a more explicit venue for considering changes. I've made retrospectives an explicit practice in this book as well.

The courage to adapt is an important principle, but it's not explicit in any single XP practice. Rather, it's a facet of XP's *Courage* value. Similarly, the need for a holistic view has been an ongoing theme in this book, but no specific practice reflects that principle. Instead, it's part of XP's *Feedback* value.

Beyond Practices

My anecdote about changing our team's release process made it sound like it went more smoothly than it really did. When the project lead left, he took with him much of the knowledge needed to produce a release, knowledge that existed only in his head and not in permanent documents anywhere. We knew this, but decided to take over the release process anyway.

Why? First, we had no choice if we wanted to return to monthly releases, even if there were problems with the process. More importantly, this was the best way we could think of to *identify* problems with our written process. If someone could follow the instructions to produce a full release, even if he had never made a release before, the instructions were good. If not, he could make a list of problems and we'd address them as a group.

That's what happened. We discovered several problems—our distribution system wouldn't distribute our release to the global mirrors, it didn't index our files correctly, and our automatic installer completely failed to configure the source code for custom builds. Fixing each problem required us to improve our overall process.

We made some mistakes, but it was worth it. We've conducted several monthly releases so far. Though the first few were rocky, they've improved over time. It's painless to release our software again.

Break the Rules

Rules are important—they exist for a reason. Yet rules can't anticipate all situations. When established conventions thwart your attempts to succeed, it's time to break the rules.

How do you know when to break the rules? First, you need to understand them and their reasons for existing. That comes with experience. Once you understand the rules, exercise *pragmatic idealism*: establish an underlying set of ideals—such as the agile principles—based on practical results. Embrace your ideals, but ground them in pragmatism. For example, "We want to avoid integration hell" is a pragmatic result that leads to the ideal of "We will *never* check in code that doesn't build or pass its tests."

With the guidance of your principles, question existing conventions. Ask yourself, "Why do we follow this rule? Do we really need it?" Modify, work around, or break the rules that prevent you from achieving success.

Remember, though, that organizational support is central to success. If you break a rule, you might step on someone's toes. Be prepared to explain your experiment. You'll find it's easier to get away with breaking rules when you've demonstrated that you're trustworthy and effective.

In Practice

Rule-breaking exists more in XP folklore than in XP practices. For example, early XP teams told stories of coming in to work on a weekend to dismantle cubicle walls, assuming that it would be easier to ask forgiveness than permission. Ron Jeffries, one of XP's earliest proponents, is famous for saying, "They're just rules"[*] in regard to the XP practices. He later clarified his statement:

> They're not rules, OK? They're techniques. They're tools we apply. They're habits. They're practices—things we practice.... They are, however, darn good things to know how to do, and do well.[†]

Beyond Practices

One of the seemingly inviolate rules of XP is that you always keep code quality high. It's even an agile principle (see "Eliminate Technical Debt"" in Chapter 15). Yet even this rule is just a rule.

As cofounder of a brand-new startup, I had an opportunity to show our software at an industry conference. We had a spike solution (see "Spike Solutions"" in Chapter 9) that demonstrated some concepts of our software, but it had no tests. We had four weeks to produce a compelling demo.

The right thing to do would have been to redevelop the spike using proper test-driven development. In another situation, I would have done so. Yet in this case, my partner wasn't as familiar with TDD as I was. I had other time commitments and couldn't do much development work. Our choices were for me to forgo my other commitments and use TDD, introduce technical debt by developing the spike, or have no demo for the conference.

We chose to develop the spike. Breaking the rules didn't bother us as much as the fact that developing the spike would incur large amounts of technical debt, but the trade-off seemed worthwhile. We created a great demo to show, and the product was a big hit at the conference. Then we came home and felt the pain of our decision. Four weeks of accumulated technical debt stalled our little startup for almost three *months*. That's a long time.

* "They're just rules!", *http://www.xprogramming.com/Practices/justrule.htm*.

† "I was wrong. They're not rules!", *http://www.xprogramming.com/xpmag/jatNotRules.htm*.

Still, breaking the rules was the right decision for us under the circumstances. The buzz we generated around our product made the cost of technical debt worthwhile. The key to our success was that we *carefully and knowledgeably* broke a rule to achieve a specific purpose. We were also lucky. None of our competitors introduced a similar feature in the several months that we spent paying down our debt.

CHAPTER 13

Eliminate Waste

It's difficult to change the course of a heavy cruise ship, whereas a river kayak dances through rapids with the slightest touch of the paddle. Although a cruise ship has its place, the kayak is much more agile.

Agility requires flexibility and a lean process, stripped to its essentials. Anything more is wasteful. Eliminate it! The less you have to do, the less time your work will take, the less it will cost, and the more quickly you will deliver.

You can't just cut out practices, though. What's really necessary? How can you tell if something helps or hinders you? What actually gets good software to the people who need it? Answering these questions helps you eliminate waste from your process and increase your agility.

Work in Small, Reversible Steps

The easiest way to reduce waste is to reduce the amount of work you may have to throw away. This means breaking your work down into its smallest possible units and verifying them separately.

Sometimes while debugging, I see multiple problems and their solutions at once. *Shotgun debugging* is tempting, but if I try several different solutions simultaneously and fix the bug, I may not know which solution actually worked. This also usually leaves a mess behind. *Incremental change* is a better approach. I make one well-reasoned change, observe and verify its effects, and decide whether to commit to the change or revert it. I learn more and come up with better—and cleaner—solutions.

This may sound like taking *baby steps*, and it is. Though I can work for 10 or 15 minutes on a feature and get it mostly right, the quality of my code improves immensely when I focus on a very small part and spend time perfecting that one tiny piece before continuing. These short, quick steps build on each other; I rarely have to revert any changes.

If the step doesn't work, I've spent a minute or two learning something and can backtrack a few moments to position myself to make further progress. These frequent course corrections help me get where I really want to go. Baby steps reduce the scope of possible errors to only the most recent changes, which are small and fresh in my mind.

In Practice

The desire to solve big, hairy problems is common in developers. Pair programming helps us encourage each other to take small steps to avoid unnecessary embellishments. Further, the navigator concentrates on the big picture so that both developers can maintain perspective of the system as a whole.

Test-driven development provides a natural rhythm with its think-test-design-code-refactor cycle. The successful end of every cycle produces a stable checkpoint at which the entire system works as designed; it's a solid foothold from which to continue. If you go wrong, you can quickly revert to the prior known-good state.

At a higher level, stories limit the total amount of work required for any one pairing session. The maximum size of a step cannot exceed a few days. As well, continuous integration spreads working code throughout the whole team. The project makes continual, always-releasable progress at a reliable pace.

Finally, refactoring enables incremental design. The design of the system proceeds in small steps as needed. As developers add features, their understanding of the sufficient and necessary design will evolve; refactoring allows them to refine the system to meet its optimal current design.

Beyond Practices

Last summer, I introduced a friend to pair programming. She wanted to automate a family history project, and we agreed to write a parser for some sample genealogical data. The file format was complex, with some interesting rules and fields neither of us understood, but she knew which data we needed to process and which data we could safely ignore.

We started by writing a simple skeleton driven by tests. Could we load a file by name effectively? Would we get reasonable errors for exceptional conditions?

Then the fun began. I copied the first few records out of the sample file for test data and wrote a single test: could our parser identify the first record type? Then I pushed the keyboard to her and said, "Make it pass."

"What good is being able to read one record, and just the type?" she wondered, but she added two lines of code and the test passed. I asked her to write the next test. She wrote one line to check if we could identify the person's name from that record and pushed the keyboard back my way.

I wrote three lines of code. The test passed. Then I wrote a test to identify the next record type. Of course it failed. As I passed back the keyboard, we discussed ways to make it pass. I suggested hardcoding the second type in the parsing method. She looked doubtful but did it anyway, and the tests all passed.

"It's time to refactor," I said, and we generalized the method by reducing its code. With her next test, I had to parse another piece of data from both record types. This took one line.

We continued that way for two hours, adding more and more of the sample file to our test data as we passed more tests. Each time we encountered a new feature of the file format, we nibbled away at it with tiny tests. By the end of that time, we hadn't finished, but we had a small amount of code and a comprehensive test suite that would serve her well for further development.

Fail Fast

It may seem obvious, but failure is another source of waste. Unfortunately, the only way to avoid failure entirely is to avoid doing anything worthwhile. That's no way to excel. As [DeMarco & Lister 2003] said, "Projects with no real risks are losers. They are almost always devoid of benefit; that's why they weren't done years ago."

Instead of trying to avoid failure, embrace it. Think, "If this project is sure to fail, I want to know that as soon as possible." Look for ways to gather information that will tell you about the project's likelihood of failure. Conduct experiments on risk-prone areas to see if they fail in practice. The sooner you can cancel a doomed project, the less time, effort, and money you'll waste on it.

Failing fast applies to all aspects of your work, to examples as small as buying a bag of a new type of coffee bean rather than a crate. It frees you from worrying excessively about whether a decision is good or bad. If you're not sure, structure your work so that you expose errors as soon as they occur. If it fails, let it fail quickly, rather than lingering in limbo. Either way, invest only as much time and as many resources as you need to be sure of your results.

With these principles guiding your decisions, you'll fear failure less. If failure doesn't hurt, then it's OK to fail. You'll be free to experiment and take risks. Capitalize on this freedom: if you have an idea, don't speculate about whether it's a good idea—try it! Create an experiment that will fail fast, and see what happens.

In Practice

One of the challenges of adopting XP is that it tends to expose problems. For example, iterations, velocity, and the planning game shine the harsh light of fact on your schedule aspirations.

This is intentional: it's one of the ways XP helps projects fail fast. If your desired schedule is unachievable, you should know that. If the project is still worthwhile, either reduce scope or change your schedule. Otherwise, cancel the project. This may seem harsh, but it's really just a reflection of the "fail fast" philosophy.

This mindset can be foreign for organizations new to XP. It's particularly troubling to organizations that habitually create unrealistic schedules. When XP fails fast in this sort of organization, blame—not credit—often falls on XP. Yet cancelling a project early isn't a sign of failure in XP teams; it's a success. The team prevented a doomed project from wasting hundreds of thousands of dollars! That's worth celebrating.

Beyond Practices

I once led a development team on a project with an "aggressive schedule." Seasoned developers recognize this phrase as a code for impending disaster. The schedule implicitly required the team to sacrifice their lives in a misguided attempt to achieve the unachievable.

I knew before I started that we had little chance of success. I accepted the job anyway, knowing that we could at least fail fast. This was a mistake. I shouldn't have assumed.

Because there were clear danger signs for the project, our first task was to gather more information. We conducted three two-week iterations and created a release plan. Six weeks after starting the project, we had a reliable release date. It showed us coming in very late.

I thought this was good news—a textbook example of failing fast. We had performed an experiment that confirmed our fears, so now it was time to take action: change the scope of the project, change the date, or cancel the project. We had a golden opportunity to take a potential failure and turn it into a success, either by adjusting our plan or cutting our losses.

Unfortunately, I hadn't done my homework. The organization wasn't ready to accept the possibility of failure. Rather than address the problem, management tried to force the team to meet the original schedule. After realizing that management wouldn't give us the support we needed to succeed, I eventually resigned.

Management pressured the remaining team members to work late nights and weekends—a typical death-march project—to no avail. The project delivered very late, within a few weeks of our velocity-based prediction, and the company lost the client. Because the organizational culture made it impossible to fail fast, the project was doomed to fail slowly.

How could the team have succeeded? In a sense, we couldn't. There was no way for this project to achieve its original schedule, scope, and budget. (Unsurprisingly, that plan had no connection to the programmers' estimates.) The best we could have done was to fail fast, then change the plan.

What could I have done differently? My mistake was taking on the project in the first place. I knew the existing schedule was unrealistic, and I assumed that objective proof of this would be enough to change everyone's mind. I should have checked this assumption and—when it proved incorrect—declined the project.

What could the organization have done differently? Several things. It could have adopted an incremental delivery strategy, where we'd release the software after every iteration in the hope of delivering value more quickly. It could have cut the scope of the first milestone to a manageable set of features. Finally, it could have increased the available resources (up to a point) to allow us to increase our velocity.

Unfortunately, this organization couldn't face failure, which prevented them from changing their plan. Organizations that *can* face failure are capable of embracing change in their projects. Paradoxically, facing failure gives them the ability to turn potential failures into successes.

Maximize Work Not Done

The agile community has a saying: "Simplicity is the art of maximizing the work not done." This idea is central to eliminating waste. To make your process more agile, do less.

However, you can't take so much out of your process that it no longer works. As Albert Einstein said, "Everything should be made as simple as possible, but not one bit simpler." Often, this means approaching your work in a new way.

Simplifying your process sometimes means sacrificing formal structures while increasing rigor. For example, an elegant mathematical proof sketched on the back of a napkin may be rigorous, but it's informal. Similarly, sitting with customers decreases the amount of formal requirements documentation you create, but it substantially increases your ability to understand requirements.

Solutions come from feedback, communication, self-discipline, and trust. Feedback and direct communication reduce the need for intermediate deliverables. Self-discipline allows team members to work without the binding overhead of formal structures. Trust can replace the need to wait days—or longer—for formal signoffs.

Paring down your practices to the responsible essentials and removing bottlenecks lets you travel light. You can maximize the time you spend producing releasable software and improve the team's ability to focus on what's really important.

In Practice

XP aggressively eliminates waste, more so than any method I know. It's what makes XP *extreme*.

By having teams sit together and communicate directly, XP eliminates the need for intermediate requirements documents. By using close programmer collaboration and incremental design, XP eschews written design documents.

XP also eliminates waste by reusing practices in multiple roles. The obvious benefit of pair programming, for example, is continuous code review, but it also spreads knowledge throughout the team, promotes self-discipline, and reduces distractions. Collective code ownership not only enables incremental design and architecture, it removes the time wasted while you wait for someone else to make a necessary API change.

Beyond Practices

ISO 9001 certification is an essential competitive requirement for some organizations. I helped one such organization develop control software for their high-end equipment. This was the organization's first XP project, so we had to figure out how to make ISO 9001 certification work with XP. Our challenge was to do so without the waste of unnecessary documentation procedures.

Nobody on the team was an expert in ISO 9001, so we started by asking one of the organization's internal ISO 9001 auditors for help. (This was an example of the "Let the Right People Do the Right Things" principle, covered in Chapter 12.) From the auditor, we learned that ISO 9001 didn't mandate any particular process; it just required that we had a process that achieved certain goals, that we could prove we had such a process, and that we proved we were following the process.

This gave us the flexibility we needed. To keep our process simple, we reused our existing practices to meet our ISO 9001 rules. Rather than creating thick requirements documents and test plans to demonstrate that we tested our product adequately, we structured our existing customer testing practice to fill the need. In addition to demonstrating conclusively that our software fulfilled its necessary functions, the customer tests showed that we followed our own internal processes.

Pursue Throughput

A final source of waste isn't immediately obvious. The manufacturing industry calls it *inventory*. In software development, it's *unreleased software*. Either way, it's partially done work—work that has cost money but has yet to deliver any value.

Partially done work represents unrealized investment. It's waste in the form of *opportunity cost*, where the investment hasn't yet produced value but you can't use the resources it cost for anything else.

Partially done work also hurts *throughput*, which is the amount of time it takes for a new idea to become useful software. Low throughput introduces more waste. The longer it takes to develop an idea, the greater the likelihood that some change of plans will invalidate some of the partially done work.

To minimize partially done work and wasted effort, maximize your throughput. Find the step in your process that has the most work waiting to be done. That's your *constraint*: the one part of your process that determines your overall throughput. In my experience, the constraint in software projects is often the developers. The rate at which they implement stories governs the amount of work everyone else can do.

To maximize throughput, the constraint needs to work at maximum productivity, whereas the other elements of your process don't. To minimize partially finished work, nonconstraints should produce only enough work to keep the constraint busy, but not so much that there's a big pile of outstanding work. Outstanding work means greater opportunity costs and more potential for lost productivity due to changes.

Minimizing partially done work means that everyone but the constraint will be working at less than maximum efficiency. That's OK. Efficiency is expendable in other activities. In fact, it's important that nonconstraints have extra time available so they can respond to any needs that the constraint might have, thus keeping the constraint maximally productive.

NOTE

These ideas come from *The Theory of Constraints*. For more information, see [Goldratt 1997], an excellent and readable introduction. For discussion specific to software development, see [Anderson 2003].

In Practice

XP planning focuses on throughput and minimizing work in progress. It's central to the iteration structure. Every iteration takes an idea—a story—from concept to completion. Each story must be "done done" by the end of the iteration.

XP's emphasis on programmer productivity—often at the cost of other team members' productivity—is another example of this principle. Although having customers sit with the team full-time may not be the most efficient use of the customers' time, it increases programmer productivity. If programmers are the constraint, as XP assumes, this increases the team's overall throughput and productivity.

Beyond Practices

Our project faced a tight schedule, so we tried to speed things up by adding more people to the project. In the span of a month, we increased the team size from 7 programmers to 14 programmers, then to 18 programmers. Most of the new programmers were junior-level.

This is a mistake as old as software itself. Fred Brooks stated it as *Brooks' Law* in 1975: "Adding manpower to a late software project makes it later" [Brooks] (p. 25).

In this particular project, management ignored our protestations about adding people, so we decided to give it our best effort. Rather than having everyone work at maximum efficiency, we focused on maximizing throughput.

We started by increasing the size of the initial development team—the Core Team—only slightly, adding just one person. The remaining six developers formed the SWAT Team. Their job was not to work on production software, but to remove roadblocks that hindered the core development team. Every few weeks, we swapped one or two people between the two teams to share knowledge.

This structure worked well for us. It was a legacy project, so there were a lot of hindrances blocking development. One of the first problems the SWAT Team handled was fixing the build script, which would often fail due to Windows registry or DLL locking issues. By fixing this, the SWAT Team enabled the Core Team to work more smoothly.

Later, we had to add four more inexperienced programmers. We had run out of space by this time. Lacking a better option, we put the new programmers in a vacant area two flights of stairs away. We continued to focus our efforts on maintaining throughput. Not wanting to spread our experienced developers thin, we kept them concentrated in the Core Team, and since the SWAT Team was working well, we decided to leave the new team out of the loop. We deliberately gave them noncritical assignments to keep them out of our hair.

Overall, this approach was a modest success. By focusing on throughput rather than individual efficiency, we were able to withstand the change. We more than doubled our team size, with mostly junior developers, without harming our productivity. Although our productivity didn't go up, such a deluge of people would normally cause a team to come to a complete standstill. As it was, pursuing throughput allowed us to maintain our forward momentum. In better circumstances—fewer new developers, or developers with more experience—we could have actually increased our productivity.

CHAPTER 14

Deliver Value

Your software only begins to have real value when it reaches users. Only at that point do you start to generate trust, to get the most important kinds of feedback, and to demonstrate a useful return on investment. That's why successful agile projects deliver value early, often, and repeatedly.

Exploit Your Agility

Simplicity of code and process are aesthetically pleasing. Yet there's a more important reason why agility helps you create great software: it improves your ability to recognize and take advantage of new opportunities.

If you could predict to the hour how long your project would take, know what risks would and wouldn't happen, and completely eliminate all surprises, you wouldn't need agility—you would succeed with any development method.

However, what you don't know is exciting. A new crisis or happy discovery next week could completely change the rules of the game. You may discover a brilliant new technique that simplifies your code, or your customer may develop a new business practice that saves time and money.

Want to deliver real value? Take advantage of what you've learned and change your direction appropriately. Adapting your point of view to welcome changing requirements gives you great opportunities. Delivering value to your customer is your most important job. Aggressively pursuing feedback from your customer, from real users, from other team members, and from your code itself as early and as often as possible allows you to continue to learn and improve your understanding of the project. It also reveals new opportunities as they appear.

Agility requires you to work in small steps, not giant leaps. A small initial investment of time and resources, properly applied, begins producing quantifiable value immediately. As well, committing to small amounts of change makes change itself more possible. This is most evident when customer requirements outline their needs at a very high level, through stories that promise further clarification during development.

Aggressively seeking feedback and working in small steps allows you to defer your investment of resources until the last responsible moment. You can start to see the value from a piece of work as you need it, rather than hoping to benefit from it someday in the future.

In Practice

XP exploits agility by removing the time between taking an action and observing its results, which improves your ability to learn from this feedback. This is especially apparent when the whole team sits together. Developing features closely with the on-site customer allows you to identify potential misunderstandings and provides nearly instant responses to questions. Including real customers in the process with frequent deliveries of the actual software demonstrates its current value to them.

XP allows changes in focus through short work cycles. Using simultaneous phases enables you to put a lesson into practice almost immediately, without having to wait for the next requirements gathering or testing or development phase to roll back around in the schedule. The short work unit of iterations and frequent demos and releases create a reliable rhythm to make measured process adjustments. Slack provides spare time to make small but necessary changes within an iteration without having to make difficult choices about cutting essential activities.

Beyond Practices

I worked for a small startup whose major product was an inventory management system targeted at a specific retail industry. We had the top retailers and manufacturers lined up to buy our project. We were months away from delivery when our biggest customer ran into a problem: the license for their existing point-of-sale system suddenly expired. The software included a call-home system that checked with the vendor before starting.

Our plans included developing our own POS system, but that was at least several months away. Our current system managed inventory, but we hadn't done specific research on supporting various terminal types, credit card scanners, and receipt printers.

This was our largest customer, and without its business, we'd never succeed.

After discussing our options and the project's requirements with the customer, we decided that we could build just enough of the POS system within six weeks that they could switch to our software and avoid a $30,000 payment to the other vendor.

We were very fortunate that our existing customers needed very little work from us in that period. We shifted two-thirds of our development team to the POS project. Though we started from an open source project we'd found earlier, we ended up customizing it heavily.

After two weeks of development, we delivered the first iteration on a new machine we had built for the customer to test. We delivered weekly after that. Though it was hard work, we gradually added new features based on the way our customer did business. Finally, the only remaining task was to change a few lines in the GUI configuration to match the customer's store colors. We saved our customer plenty of money, and we saved our account.

Only Releasable Code Has Value

Having the best, most beautiful code in the world matters very little unless it does what the customer wants. It's also true that having code that meets customer needs perfectly has little value unless the customer can actually use it. Until your software reaches the people who need it, it has only potential value.

Delivering actual value means delivering real software. Unreleasable code has no value. Working software is the primary measure of your progress. At every point, it should be possible to stop the project and have actual value proportional to your investment in producing the software.

Any functional team can change its focus in a week or a month, but can they do so while maximizing their investment of time and resources to deliver software regularly? Do they really finish code—does "done" mean "done done"—or do they leave an expanding wake of half-finished code? Do they keep their project releasable at any time?

Agility and flexibility are wonderful things, especially when combined with iterative incremental development. Throughput is important! Besides reducing thrashing and waste, it provides much better feedback, and not just in terms of the code's quality. Only code that you can actually release to customers can provide real feedback on how well you're providing value to your customers.

That feedback is invaluable.

In Practice

The most important practice is that of "done done," where work is either complete or incomplete. This unambiguous measure of progress immediately lets you know where you stand.

Test-driven development produces a safety net that catches regressions and deviations from customer requirements. A well-written test suite can quickly identify any failures that may reduce the value of the software. Similarly, continuous integration ensures that the project works as a whole multiple times per day. It forces any mistakes or accidents to appear soon after their introduction, when they're easiest to fix. The 10-minute build reduces the bottlenecks in this process to support its frequency.

Practicing "no bugs" is a good reminder that deferring important and specific decisions decreases the project's value—especially if those decisions come directly from real customer feedback.

Beyond Practices

I was once on a project that had been running for four months and was only a few weeks away from its deadline when it was abruptly halted. "We're way over budget and didn't realize it," the manager told me. "Everybody has to go home tomorrow."

We were all contractors, but the suddeness of the project's cancellation surprised us. I stayed on for one more week to train one of the organization's employees—"Joe"—just in case they had the opportunity to pick up the code again in the future.

Although this wasn't an XP project, we had been working in iterations. We hadn't deployed any of the iteration releases (I now know that would have been a good idea), but the last iteration's result *was* ready to deploy. Joe and I did so, then spent the rest of the week fixing some known bugs.

This might have been a textbook example of good project management, except for one problem: our release wasn't usable. We had worked on features in priority order, but our customers gave us the wrong priorities! A feature they had described as least important for the first release was in fact vitally important: security. That feature was last on our list, so we hadn't implemented it. If we had deployed our interim releases, we would have discovered the problem. Instead, it blindsided us and left the company without anything of value.

Fortunately, the company brought us back a few months later and we finished the application.

Deliver Business Results

What if you could best meet your customer's need without writing any software? Would you do it? Could you do it?

Someday that may happen to you. It may not be as dramatic as telling a recurring customer that he'll get better results if you *don't* write software, but you may have to choose between delivering code and delivering business results.

Value isn't really about software, after all. Your goal is to deliver something useful for the customer. The software is merely how you do that. The single most essential criterion for your success is the fitness of the project for its business purposes. Everything else is secondary—not useless by any means, but of lesser importance.

For example, agile teams value working software over comprehensive documentation. Documentation is valuable—communicating what the software must do and how it works is important—but your first priority is to meet your customer's needs. Sometimes that means producing good documentation. Usually it means delivering good software, but not always. The primary goal is always to provide the most valuable business results possible.

In Practice

XP encourages close involvement with actual customers by bringing them into the team, so they can measure progress and make decisions based on business value every day. Real customer involvement allows the on-site customer to review these values with end-users and keep the plan on track. Their vision provides answers to the questions most important to the project.

XP approaches its schedule in terms of customer value. The team works on stories phrased from the customer's point of view and verifiable by customer testing. After each iteration, the iteration demo shows the team's current progress to stakeholders, allowing them to verify that the results are valuable and to decide whether to continue development.

Beyond Practices

A friend—"Aaron"—recently spent a man-month writing 1,500 lines of prototype code that generated $7.8 million in revenue during its first demo.

As a graduate student, he interned with a technology company doing research on handwriting recognition with digital pens containing sensors and recording equipment. A customer made an off-hand remark about how useful it might be to use those special pens with printed maps. Suddenly, Aaron had a research assignment.

The largest potential customer used an existing software package to send map data to field agents to plan routes and identify waypoints. Aaron modified the pen software to send coordinate information on printed pages. Then he found a way to encode the pen's necessary calibration data on color laser printouts. The final step was to use the API of the customer's software to enter special pen events—mark waypoint, identify route, etc. In effect, all of his code merely replaced the clunky mouse-based UI with the act of drawing on a custom-printed map, then docking the pen.

A few minutes into the first demo, the customer led the sales rep to the control room for a field exercise. After installing the software and connecting the pen's dock, the rep handed the pen and a printed map to one of the techs. The tech had never seen the product before and had no training, but he immediately circled an objective on the map and docked the pen. In seconds, the objective appeared on the vehicle displays as well as on the PDAs of the field agents.

The customer placed an order for a license and hardware for everyone at the location. That's business results.

Deliver Frequently

If you have a business problem, a solution to that problem today is much more valuable than a solution to that problem in six months—especially if the solution will be the same then as it is now. Value is more than just doing what the customer needs. It's doing what the customer needs *when* the customer needs it.

Delivering working, valuable software *frequently* makes your software more valuable. This is especially true when a real customer promotes the most valuable stories to the start of the project. Delivering working software as fast as possible enables two important feedback loops. One is from actual customers to the developers, where the customers use the software and communicate how well it meets their needs. The other is from the team to the customers, where the team communicates by demonstrating how trustworthy and capable it is.

Frequent delivery tightens those loops. Customers see that their involvement in the process makes a real difference to their work. Developers see that they're helping real people solve real problems. The highest priority of any software project is to deliver value, frequently and continuously, and by doing so, to satisfy the customer. Success follows.

In Practice

Once you've identified what the customer really needs and what makes the software valuable, XP's technical practices help you achieve fast and frequent releases. Short iterations keep the schedule light and manageable by dividing the whole project into week-long cycles, culminating in a deliverable project demonstrated in the iteration demo. This allows you to deliver once a week, if not sooner.

Practicing the art of "done done" with discipline keeps you on track, helping you identify how much work you have finished and can do while reminding you to pursue throughput. Keeping a ten-minute build reminds you to reduce or remove any unnecessary technical bottlenecks to producing a release. The less work and frustration, and the more automation, the easier it is to deliver a freshly tested, working new build.

Beyond Practices

According to founder Cal Henderson,* the photo-sharing web site Flickr has practiced frequent delivery from its earliest days. There was no single decision to do so; it was just an extension of how its founders worked. Rather than batching up new features, they released them to users as soon as possible. This helped simplify their development and reduced the cost of fixing bugs.

The early days of Flickr were somewhat informal. When there were only three committers on the project, asking if the trunk was ready to deploy required asking only two other people. Now the team assumes that the trunk is always ready to deploy to the live site. The most important component of this process is a group of strong and responsible developers who appreciate the chance to manage, code, test, stage, and deploy features. The rest of the work is standard agility—working in small cycles, rigorous testing, fixing bugs immediately, and taking many small risks.

The results are powerful. When a user posts a bug to the forum, the team can often fix the problem and deploy the new code to the live site within minutes. There's no need to wait for other people to finish a new feature. It's surprisingly low-risk, too. According to Henderson, "The number of nontrivial rollbacks on the Flickr code base is still zero."

* Via personal communication.