/Theory/In/Practice

The Art of Agile Development

O'REILLY®

James Shore & Shane Warden

The Art of Agile Development

James Shore and Shane Warden

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

The Art of Agile Development
by James Shore and Shane Warden

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472

Editor: Mary O'Brien
Copy Editor: Sarah Schneider
Production Editor: Sarah Schneider
Proofreader: Sada Preisch

Indexer: Joe Wizda
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
October 2007: First Edition.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

RepKover™

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN-10: 0-596-52767-5
ISBN-13: 978-0-596-52767-9

[C]

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?"

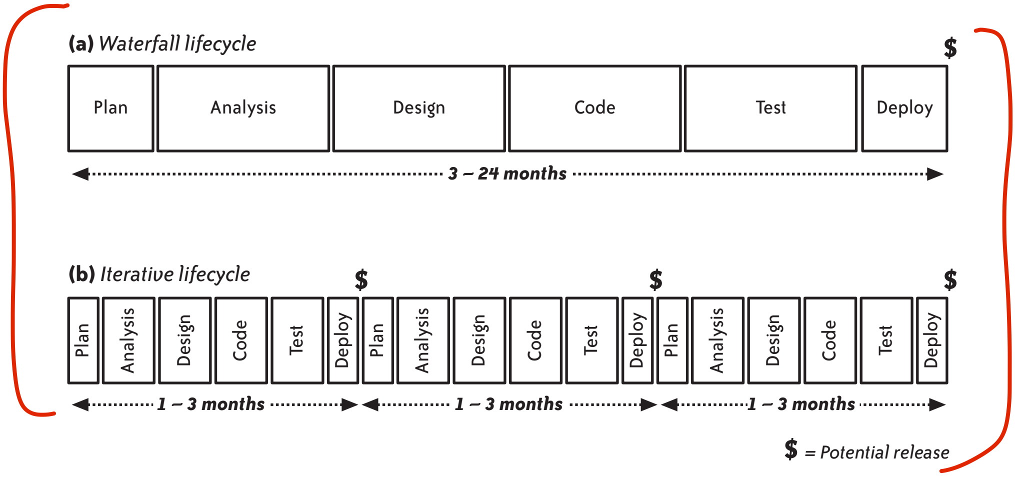

Figure 3-1. Traditional lifecycles

“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?”

“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.”

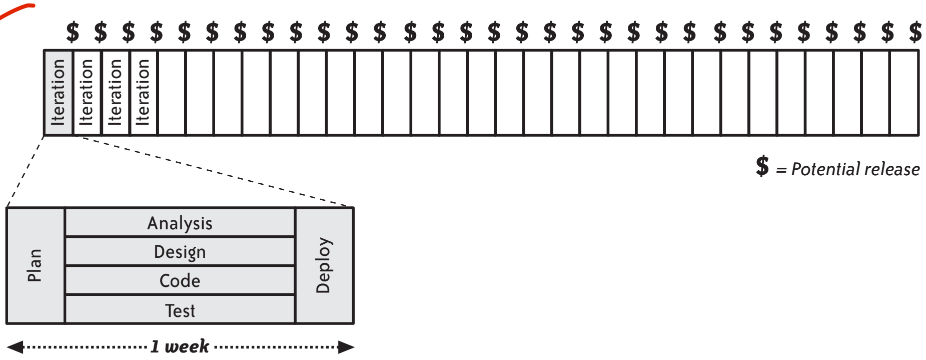

Figure 3-2. XP lifecycle

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.

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.

This Book	2nd Edition XP[a]	1st Edition XP[b]	Scrum[c]
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
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

Repasar brevemente los roles de cada miembro. Leer el Rol del TESTER

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.

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.

* This disc
XP doesn't include testers as a distinct role.

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.

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.

Prefer *better* to *bigger*.

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.

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.)

Fractional assignment is dreadfully counterproductive.

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).

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. 303)

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

CONCEPTO EXPLICADO EN CLASE

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.