EXTREME PROGRAMMING AND AGILE SOFTWARE DEVELOPMENT METHODOLOGIES

Lowell Lindstrom and Ron Jeffries

Several agile (i.e., lightweight) development methodologies, especially extreme programming (XP), have been argued to be a solution to many of the problems that continue to plague software development projects. The authors provide a useful evaluation of such approaches, including a discussion of the values that underlie the XP methodology.

LOWELL LINDSTROM has been a software professional for 20 years. He currently runs a consulting firm that helps software intensive businesses transition to more agile business processes. He can be reached at lindstrom@ objectmentor.com

RON JEFFRIES is an independent consultant. He is the senior author of Extreme Programming Installed *from Addison-Wesley, and has just released a new book,* Extreme Programming Adventures in C#, *from Microsoft Press. He can be reached at ronjeffries@acm.org.*

*A*S A STAKEHOLDER OF A SOFTWARE project, how does the following sound to you? You can have releases as often as you like. The small number of defects is unprecedented. All of the features in the system are the most valuable ones to your business. At anytime, you have access to complete, accurate information as the status of any feature and of the quality of the system as a whole. The team developing your project works in an energized space with constant communication about the project. You are not dependent on any one or even two programmers for the continued success of the project. If your needs change, the development team welcomes the change of direction.

As a developer of a software project, how does the following sound to you? No one estimates your tasks but you, period! You always have access to a customer to clarify details about the features you are implementing. You are free (and required) to clean up the code whenever necessary. You complete a project every two weeks. You can work in any part of the system that you wish. You can pick who will help you on any given task. You are not required to constantly work long hours.

Does this sound too good to be true? Teams are achieving these advantages using a relatively new set of software methodologies. Collectively, they are referred to as Agile Software Development Methodologies. The most pervasive is Extreme Programming. This article introduces these exciting, popular, yet controversial new approaches to software development.

BACKGROUND
Trends in Software Development
The last half-century has seen a dizzying progression of technical advancement in the areas of computer, software, and communications technology. With each advance came rapid changes in the way society works and lives. The impact of technology is increasingly pervasive. Even as the current economic downturn limits capital investment, innovators and entrepreneurs are pushing the limits in the areas of biotechnology and nanotechnology.

The most popular project management techniques focus on developing a plan and sticking to the plan. This improves coordination but reduces the ability of the project to adapt to new information regarding the requirements or the implementation details.

Supporting all these advances is software running on some kind of computer. From the spreadsheets to anti-lock brakes to phones to toys, logic that was once implemented in mechanics, circuitry, or pencil and paper is now a set of instructions controlling one or many computers. These computers communicate with each other over networks ignorant to the limits of geography and even wires. The notion of a program running on a computer is largely a memory as the emergence of distributed computing models, most recently Web services, allows many different computers to participate in "running" a program or system to yield a given output.

In this explosive environment, software professionals have the challenge to deliver a seemingly infinite backlog of software projects, while keeping abreast of the latest advances. The results are debatable at best. Survey after survey continues to confirm that most software projects fail against some measure of success. Most software developers have many more stories of failure than success. Given the friction and finger-pointing that accompanies the end of a failed project, it is difficult to research the causes of software failures. However, typically, projects fail for one or more of the following reasons:

- Requirements that are not clearly communicated
- Requirements that do not solve the business problem
- Requirements that change prior to the completion of the project
- Software (code) that has not been tested
- Software that has not been tested as the user will use it
- Software developed such that it is difficult to modify
- Software that is used for functions for which it was not intended
- Projects not staffed with the resources required in the project plan
- Schedule and scope commitments are made prior to fully understanding the requirements or the technical risks

Despite these delivery problems, strong demand seems to hold steady. Reports still suggest a shortage of software professionals and college graduates, despite the implosion of the Internet industry slowing the growth of demand and the emergence of an offshore programming market increasing supply.

Improvement Remains Elusive

Efforts to improve the success rate of software projects have existed since the first bug was detected. Recently, these efforts have focused on four distinct parts of the software development process: requirements gathering, designing and developing the software, testing the results, and overall project management. Formal requirements definition and analysis addressed the problem of requirements that were incomplete or did not reflect the needs of the customer. Formal design before implementation addressed the goals of reuse, consistency of operation, and reducing rework. Testing caught defects before they reached the users. Project management addressed the problem of coordinating the efforts of multi-department teams. These areas of focus follow a logical approach to process improvement: improve the quality of the inputs (requirements), improve the quality of the output (designing and developing, project management), and improve the detection and elimination of defects prior to shipping (testing). Methods and tools focusing on these four distinct areas of software development have proliferated.

For some projects, these efforts have been effective. For others, they yielded silos of responsibility, with poor communication between the groups and distributed ownership and accountability. If a project is late, it is easy for the programmers to blame the requirements gatherers, and vice versa. If the users detect defects, finger-pointing ensues between developers and testers. Project management techniques try to better coordinate the activities of the multiple groups involved in delivering the project, but add yet another silo and area of accountability. The most popular project management techniques focus on developing a plan and sticking to that plan. This improves coordination but reduces the ability of the project to adapt to new information regarding the requirements or the implementation details.

The Emergence of Agile Methods

The additional process steps, roles, and artifacts helped many teams to enjoy higher success rates and more satisfied customers. Unfortunately, many projects failed attempting to use the same techniques. Some projects got lost in the documents and never implemented any code, missing the window of opportunity for the software. Others did not leave enough time at the end for implementation and testing

*T**he set of known successes with XP continues to stretch the breadth of projects applicable for XP.***

and delivered systems inconsistent with the documents and designs on which most of the project time was spent.

At the same time, numerous projects were very successful that did not follow methods with binders of documents, detailed designs, and project plans. Many experienced programmers were having great success without all these extra steps. The determining factor of project success seemed more and more to be the people on the project, not the technology or the methods that were being used. After all, people end up writing the software at some point. To some, the developers who did not embrace the new methodologies appeared to be undisciplined and indifferent to quality, despite their successes at delivering quality software that people wanted to use.

A few people started to author papers about these disciplined, yet lighter approaches to software development. They called them Extreme Programming, SCRUM, Crystal, Adaptive, etc. Different authors emphasized different aspects of software development. Some focused on approaches to planning and requirements; some focused on ways to write software that could be changed more easily; and some focused on the people interactions that allow software developers to more easily adapt to their customers' changing needs. These various efforts created a focal point for a community that furthered the set of practices that succeed without many of the activities and artifacts required by more defined methodologies.

In the fall of 1999, *Extreme Programming, Embrace Change[1]* was published and the trend had found its catalyst. In early 2001, the different innovators who were creating different agile methodologies held a retreat and scribed the "Agile Manifesto for Software Development."[2] By the spring of 2002, Computerworld.com ran the following headline: "More than two-thirds of all corporate IT organizations will use some form of 'agile' software development process within 18 months, Giga Information Group predicted this week at its application development conference here."[3]

Agile Methodologies

At the retreat in early 2001, a number of leaders of the agile software development movement held a retreat to discuss their approaches and to explore the commonalities and differences. What emerged was the Agile Manifesto for Software Development. The manifesto (see Table 1) articulates core values and principles that guide agile methodologies.

EXTREME PROGRAMMING

Extreme Programming (XP) is the most widely used agile methodology. XP shares the values espoused by the Agile Manifesto for Software Development but goes further to specify a simple set of practices. Whereas many popular methodologies try to answer the question "What are all of the practices I might ever need on a software project?," XP simply asks, "What is the simplest set of practices I could possibly need and what do I need to do to limit my needs to those practices?"

The significance of this difference cannot be understated. The most frequent critique of XP is that it is too simple to work beyond a narrow set of project criteria. Yet, the set of known successes with XP continues to stretch the breadth of projects applicable for XP. It would seem that the parameters that we use to determine what methods are appropriate for what project are still inadequate.

To many, XP is a set of 12 interdependent software development practices. Used together, these practices have had much success, initially with small teams, working on projects with high degrees of change. However, the more one works with XP, the more it is apparent that the practices do not capture the essence of XP. As with the heavier methods, some teams have great success with the XP practices, some less so. Some larger teams have greater success than smaller ones. Some teams with legacy code have success; others do not. There is something more than just the practices that enables teams to succeed with XP. This extra attribute of XP is XP Values.

What Is Extreme Programming?

Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.

In XP, every contributor to the project is a member of the "Whole Team," a single business/development/testing team that handles all aspects of the development. Central to the team is the "Customer," one or more business representatives who sit with the team and work with them daily.

TABLE 1 Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others to 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 on the items on the right, we value the items on the left more.

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.
- Welcoming 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 shorter time scales.
- 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-organized teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Agile Methods

The number of methods that claim to align the Agile Manifesto will continue to grow with the popularity of the agile software methodologies. The early initial methodologies include:*

- Extreme Programming
- SCRUM
- Crystal
- Feature Driven Development
- Lean Development
- Adaptive Software Development
- DSDM

* Jim Highsmith, *Agile Software Development Ecosystems*, Addison-Wesley, 2002, provides a comparison of these methodologies.

XP teams use a simple form of planning and tracking to decide what to do next and to predict when any desired feature set will be delivered. Focused on business value, the team produces the software in a series of small, fully integrated releases that pass all the tests that the Customer has defined. The core XP practices for the above are called Whole Team, Planning Game, Small Releases, and Acceptance Tests. There are specific recommendations for all of these, which are briefly discussed here and as the chapter progresses.

Extreme Programmers work together in pairs and as a group, with simple design and obsessively tested code, improving the design continually to keep it always just right for the current needs.

The core XP practices here are Pair Programming, Simple Design, Test-Driven Development, and Design Improvement.

The XP team keeps the system integrated and running all the time. The programmers write all production code in pairs, and all work together all the time. They code in a consistent style so that everyone can understand and

TABLE 2 Comparison of Methodologies

Methodology[a]	Values	Principles	Practices
CMMI	No	No	Yes (KPAs)
SA/SD	No	No	Yes
RUP	No	Yes	Yes
Agile	Yes	Yes	No
XP	Yes	Yes	Yes

[a] CMMI: Capability Maturity Model Integration, Software Engineering Institute.

SA/SD: Structured Analysis/Structured Design.
RUP: Rational Uni ed Process , Rational Corporation.

improve all the code as needed. The additional practices here are called Continuous Integration, Team Code Ownership, and Coding Standard.

The XP team shares a common and simple picture of what the system looks like. Everyone works at a pace that can be sustained indefinitely. These practices are called Metaphor and Sustainable Pace.

XP Values

The XP Values are Communication, Simplicity, Feedback, and Courage.

> The essence [of XP] truly is simple. Be together with your customer and fellow programmers, and talk to each other. Use simple design and programming practices, and simple methods of planning, tracking, and reporting. Test your program and your practices, using feedback to steer the project. Working together this way gives the team courage.[4]

These values guide our actions on the project. The practices leverage these values to remove complexity from the process. The impact of the XP Values is significant and unique. XP remains the only methodology that is explicit in its values and practices. This combination gives specific guidance not only on what (the practices) to do on a project, but also on how to react (defer to the values) when the practices do not seem to be working or are not sufficient. Most methods are specific on practices, some specify principles, but few combine both.[5] For example, CMMI describes Key Practice Areas (KPAs) but does not articulate a set of values or principles. RUP provides guiding principles, such as Develop Iteratively, but does not include values that give guidance beyond the software development practices.

How these values are used to guide the team in its use of the practices is described later in the section "Fitting XP to Your Project." See Table 2 for a comparison of several well-known methodologies.

Organization

On a project using XP, there are two explicit roles or teams defined: the Customer and the Programmer. In keeping with the value of simplicity, most of the XP literature describes the customer as a single person who can represent the requirements, acceptance criteria, and business value for the project. In practice, it is a team of people that communicates with one voice with the Programming Team. As such, this role is also referred to as the Customer Team. This chapter uses the term "Customer" to describe the role, whether acted on by an individual or a team. The Programmer is a member of the Programming Team that implements the XP Customer Team's requirements. Again, the convention will be to use the term "Programmer" to describe an individual or the team.

On all but the smallest projects, there will also be a Management Team that allocates resources for the teams, manages the alignment of the project to the goals of the business, and removes any obstacles impeding the team's progress. Extreme Programming does not specify management practices. XP attempts to simplify management by empowering the Customer and Programmer to make most of the decisions regarding the project. Often, XP teams are described as self-managing. As projects grow in size and complexity, more management is typically required to coordinate the efforts of different teams. Many of the other emerging agile methodologies are focusing more attention on management practices, such as Scrum,[6] Lean Development,[7] and Extreme Project Management.[8]

The Rhythm of an XP Project

An XP project proceeds in iterations of two weeks in length. Each iteration delivers fully developed and tested software that meets the most valuable small set of the full project's requirements. Figure 1 shows the primary activities of the Customer and Programmer during the initial iterations of a project. The project proceeds in a steady rhythm of delivering more functionality. The Customer determines at what point in time the full system can be released and deployed.
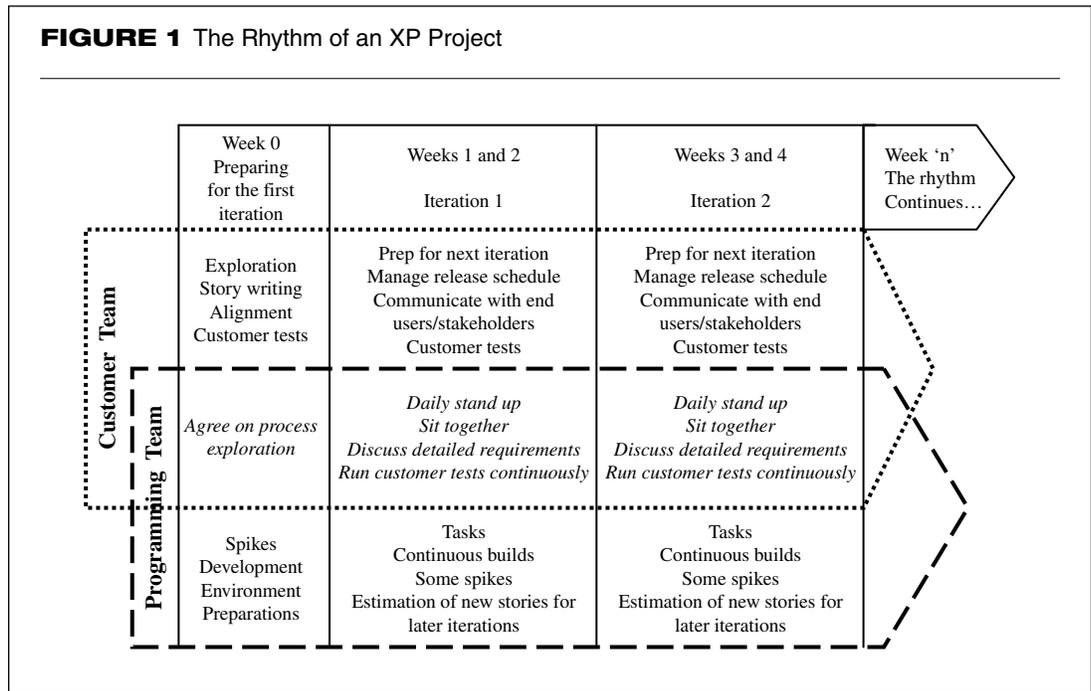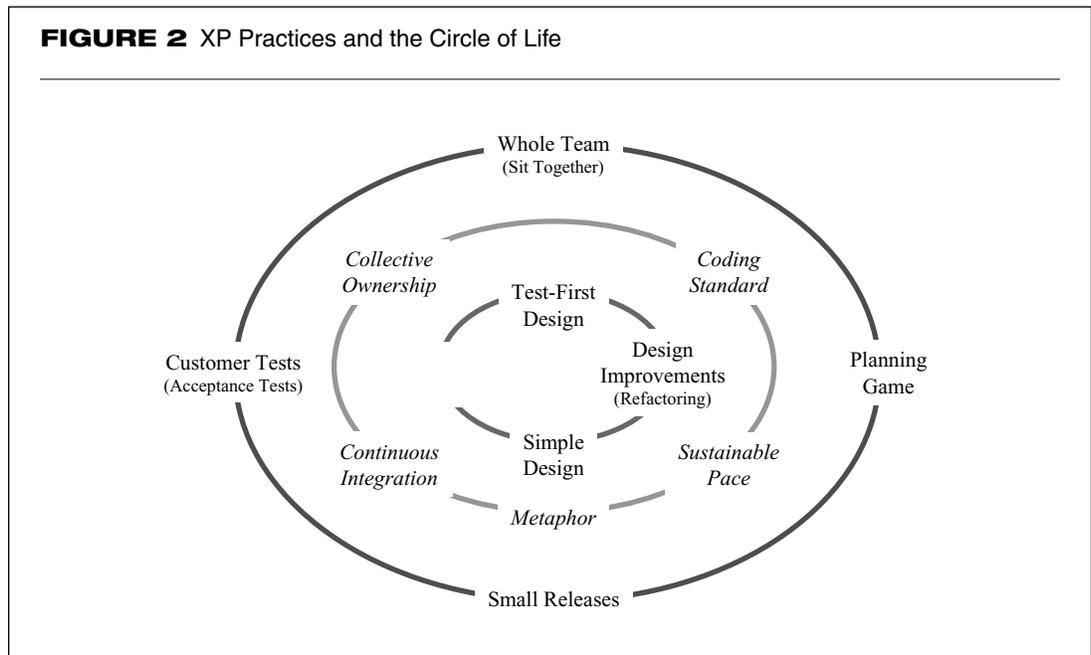
FIGURE 1 The Rhythm of an XP Project

	Week 0 Preparing for the first iteration	Weeks 1 and 2 Iteration 1	Weeks 3 and 4 Iteration 2	Week 'n' The rhythm Continues…
Customer Team	Exploration Story writing Alignment Customer tests	Prep for next iteration Manage release schedule Communicate with end users/stakeholders Customer tests	Prep for next iteration Manage release schedule Communicate with end users/stakeholders Customer tests	
Programing Team	*Agree on process exploration*	*Daily stand up Sit together Discuss detailed requirements Run customer tests continuously*	*Daily stand up Sit together Discuss detailed requirements Run customer tests continuously*	
	Spikes Development Environment Preparations	Tasks Continuous builds Some spikes Estimation of new stories for later iterations	Tasks Continuous builds Some spikes Estimation of new stories for later iterations	

FIGURE 2 XP Practices and the Circle of Life

Core Practices

There are 12 core practices that define XP. Teams new to XP should focus on using and developing skills with these practices. Over time, as the team matures in its use of XP, it will continue to check its proficiency with these practices, but will also tailor the practices to the project needs. XP teams are encouraged to use feedback from their project to adapt, add, and eliminate practices as needed. A number of other practices are popular on XP teams and some of these are described later.

The practices can be described as a cycle of activities (see Figure 2). The inner circle describes the tight cycle of the Programmers. The outer loop describes the planning cycle that occurs between the Customers and Programmers. The middle loop shows practices that help the team communicate and coordinate the delivery of quality software.

The best XP teams treat their customer tests the same way they do programmer tests: once the test runs, the team keeps it running correctly thereafter.

Whole Team. All the contributors to an XP project sit together as members of one team. This team must include a business representative — the Customer — who provides the requirements, sets the priorities, and steers the project. It is best if the Customer or one of her aides is a real end user who knows the domain and what is needed. The team will, of course, have programmers. The team will typically include testers, who help the Customer define the customer acceptance tests. Analysts may serve as helpers to the Customer, helping to define the requirements. There is commonly a coach who helps the team stay on track and facilitates the process. There may be a manager, providing resources, handling external communication, and coordinating activities. None of these roles is necessarily the exclusive property of just one individual. Everyone on an XP team contributes in any way that he or she can. The best teams have no specialists, only general contributors with special skills.

Planning Game. XP planning addresses two key questions in software development: predicting what will be accomplished by the due date, and determining what to do next. The emphasis is on steering the project — which is quite straightforward — rather than on exact prediction of what will be needed and how long it will take — which is quite difficult. There are two key planning steps in XP:

1. *Release planning* is a practice where the Customer presents the desired features to the programmers, and the programmers estimate their difficulty. With the cost estimates in hand, and with knowledge of the importance of the features, the Customer lays out a plan for the project. Initial release plans are necessarily imprecise; neither the priorities nor the estimates are truly solid, and until the team begins to work, we will not know just how fast they will go. Even the first release plan is accurate enough for decision making, however, and XP teams revise the release plan regularly.
2. *Iteration planning* is the practice whereby the team is given direction every couple of weeks. XP teams build software in two-week "iterations," delivering running, useful software at the end of each iteration. During Iteration Planning, the Customer presents the features desired for the next two weeks. The programmers break them down into tasks and estimate their cost (at a finer level of detail than in Release Planning).

Based on the amount of work accomplished in the previous iteration, the team signs up for what will be undertaken in the current iteration.

These planning steps are very simple yet they provide very good information and excellent steering control in the hands of the Customer. Every couple of weeks, the amount of progress is entirely visible. There is no "90 percent done" in XP: a feature story was completed, or it was not. This focus on visibility results in a nice little paradox. On the one hand, with so much visibility, the Customer is in a position to cancel the project if progress is not sufficient. On the other hand, progress is so visible, and the ability to decide what will be done next is so complete, that XP projects tend to deliver more of what is needed, with less pressure and stress.

Customer Tests. As part of presenting each desired feature, the XP Customer defines one or more automated acceptance tests to show that the feature is working. The team builds these tests and uses them to prove to themselves, and to the customers, that the feature is implemented correctly. Automation is important because in the press of time, manual tests are skipped. That is like turning off your lights when the night gets darkest.

The best XP teams treat their customer tests the same way they do programmer tests: once the test runs, the team keeps it running correctly thereafter. This means that the system only improves, always notching forward, and never backsliding.

Small Releases. XP teams practice small releases in two important ways. First, the team releases running, tested software, delivering business value chosen by the Customer, with every iteration. The Customer can use this software for any purpose, either for evaluation or even for release to end users (which is highly recommended). The most important aspect is that the software is visible, and given to the customer at the end of every iteration. This keeps everything open and tangible. Second, XP teams also release software to their end users frequently. XP Web projects release as often as daily, in-house projects monthly or more frequently. Even shrink-wrapped products are shipped as often as quarterly.

It might seem impossible to create good versions this often but XP teams are doing it all the time. See the section on "Continuous

*E*xtreme programming focuses on delivering business values in every iteration.

Integration" for more on this, and note that these frequent releases are kept reliable by XP's obsession with testing, as described in the sections on "Customer Tests" and "Test-Driven Development."

Simple Design. XP teams build software to a simple design. They start simple, and through programmer testing and design improvement, they keep it that way. An XP team keeps the design exactly suited for the current functionality of the system. There is no wasted motion, and the software is always ready for what is next.

Design in XP is neither a one-time thing nor an up-front thing, but it is an all-the-time thing. There are design steps in release planning and iteration planning, plus teams engage in quick design sessions and design revisions through refactoring, throughout the course of the entire project. In an incremental, iterative process like Extreme Programming, good design is essential.

Pair Programming. In XP, two programmers, sitting side by side at the same machine, build all production software. This practice ensures that all production code is reviewed by at least one other programmer, resulting in better design, better testing, and better code.

It may seem inefficient to have two programmers doing "one programmer's job," but the reverse is true. Research on pair programming shows that pairing produces better code in about the same time as programmers working singly. That is right: two heads really are better than one!

It does take some practice to do well, and you need to do it well for a few weeks to see the results. Most programmers who learn pair programming prefer it, so we highly recommend it to all teams.

Pairing, in addition to providing better code and tests, also serves to communicate knowledge throughout the team. As pairs switch, everyone gets the benefits of everyone's specialized knowledge. Programmers learn, their skills improve, and they become move valuable to the team and to the company. Pairing, even on its own outside of XP, is a big win for everyone.

Test-Driven Development. XP is obsessed with feedback; and in software development, good feedback requires good testing. XP teams practice "test-driven development," working in very short cycles of adding a test, then making it work. Almost effortlessly, teams produce code with nearly 100 percent test coverage, which is a great step forward in most shops. (If your programmers are already doing even more sophisticated testing, more power to you. Keep it up, it can only help.)

It is not enough to write tests; you have to run them. Here, too, XP is extreme. These "programmer tests," or "unit tests," are all collected together, and every time any programmer releases any code to the repository (and pairs typically release twice a day or more), every single one of the programmer tests must run correctly. One hundred percent, all the time! This means that programmers get immediate feedback on how they are doing. Additionally, these tests provide invaluable support as the software design is improved.

Design Improvement. XP focuses on delivering business value in every iteration. To accomplish this over the course of the whole project, the software must be well designed. The alternative would be to slow down and ultimately get stuck. So, XP uses a process of continuous design improvement called "refactoring."[9]

The refactoring process focuses on the removal of duplication (a sure sign of poor design), and on increasing the "cohesion" of the code while lowering the "coupling." High cohesion and low coupling have been recognized as the hallmarks of well-designed code for at least 30 years.[10] The result is that XP teams start with a good, simple design, and always have a good, simple design for the software. This lets them sustain their development speed and, in fact, generally increase speed as the project goes forward.

Refactoring is, of course, strongly supported by comprehensive testing that ensures that as the design evolves, nothing is broken. Thus, the customer tests and programmer tests are a critical enabling factor. The XP practices support each other: they are stronger together than separately.

Continuous Integration. XP teams keep the system fully integrated at all times. We say that daily builds are for wimps; XP teams build multiple times per day. (One XP team of 40 people builds at least eight or ten times per day!) The benefit of this practice can be seen by thinking back on projects you may have heard about (or even been a part of), where the build process was weekly or less frequently and usually led to "integration hell," where everything broke and no one knew why.

The specifics of the standard are not important; what is important is that all the code looks familiar, in support of collective ownership.

Infrequent integration leads to serious problems on a software project. First of all, although integration is critical to shipping good working code, the team is not practiced at it, and often it is delegated to people who are not familiar with the whole system. Second, infrequently integrated code is often — or usually — buggy code. Problems creep in at integration time that are not detected by any of the testing that takes place on a nonintegrated system. Third, a weak integration process leads to long code freezes. Code freezes mean that you have long time periods when the programmers could be working on important shippable features, but that those features must be held back. This weakens your position in the market or with your end users.

Collective Code Ownership. On an XP project, any pair of programmers can improve any code at anytime. This means that all code gets the benefit of many people's attention, which increases code quality and reduces defects. There is another important benefit as well: when code is owned by individuals, required features are often put in the wrong place as one programmer discovers that he needs a feature somewhere in code that he does not own. The owner is too busy to do it, so the programmer puts the feature in his own code, where it does not belong. This leads to ugly, hard-to-maintain code, full of duplication and with low (bad) cohesion.

Collective ownership could be a problem if people worked blindly on code they do not understand. XP avoids these problems through two key techniques: (1) the programmer tests catch mistakes, and (2) pair programming, which means that the best way to work on unfamiliar code is to pair with the expert. In addition to ensuring good modifications when needed, this practice spreads knowledge throughout the team.

Coding Standard. XP teams follow a common coding standard so that all the code in the system looks as if a single — very competent — individual wrote it. The specifics of the standard are not important; what is important is that all the code looks familiar, in support of collective ownership.

Metaphor. XP teams develop a common vision of how the program works, which we call the "metaphor." At its best, the metaphor is a simple, evocative description of how the program works, such as "this program works like a hive of bees, going out for pollen and bringing it back to the hive" as a description for an agent-based information retrieval system.

Sometimes, a sufficiently poetic metaphor does not arise. In any case, with or without vivid imagery, XP teams use a common system of names to be sure that everyone understands how the system works and where to look to find functionality or to find the right place to put the functionality that is about to be added.

Sustainable Pace. XP teams are in it for the long term. They work hard, but at a pace that can be sustained indefinitely. This means that they work overtime when it is effective, and that they normally work in such a way as to maximize productivity week in and week out. It is well understood these days that "death march" projects are neither productive nor produce quality software. XP teams are in it to win, not to die.

Other Common Practices

The core practices of XP do not specify all of the activities that are required to deliver a software project. As teams use XP, many find that other practices aid in their success, in some cases as significantly as some of the core practices. The following are some other practices commonly used by successful XP teams.

Open Workspace. To maximize communication among the Whole Team, the team works together in an "open workspace." This is a large room, with tables in the center that can typically seat two to four pairs of developers. Figure 3 shows an example of an XP workstation for three pairs of developers. By sitting together, all team members can establish instant communication when needed for the project. Teams establish their own rules concerning their space to ensure that everyone can work effectively. The walls of the "open workspace" are used to display information about the project. This will include big, visible charts of metrics such as passing acceptance tests and team productivity. There may be designs drawn on whiteboards. Project status will be displayed so that any participant or stakeholder of the project can always see progress.

Retrospectives. The XP practices provide feedback to the team as to the quality of the code and its alignment to the Customers' needs. The team also needs feedback on how it is performing. Is it following the practices with discipline? Are there adaptations to the practices

FIGURE 3 Extreme Programming Workstation
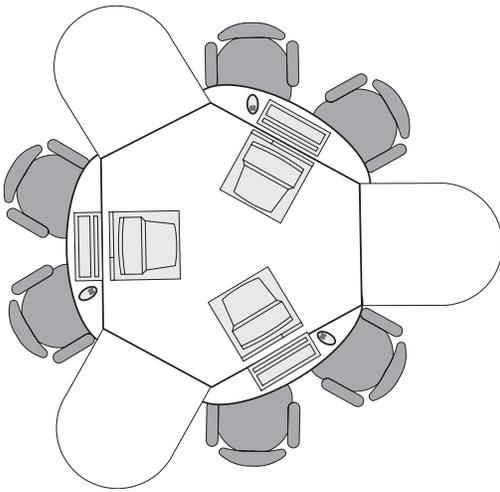
that would benefit the team? The practice commonly used for this is the Retrospective.[11] After each iteration, the team does a short reflection on what went well during the iteration and what should be improved in the next iteration. After a release of the product, a more in-depth Retrospective is performed on the whole project.

Self-Directed Teams. A practice that is common among most of the agile methods is self-directed teams. The best people to make decisions about the project are those closest to the details, as long as they have an understanding of the overall goals of the project. Open communication allows team members to have the information required to make decisions. Managers are part of the communication loop but not bottlenecks in the decision-making flow.

Customer Team. As XP is used on projects with more complex requirements, a team performs the Customer function. For larger or more complex projects, the Customer team may even exceed the Programming team in size. Some of the challenges faced by the Customer team include communicating with and balancing the needs of multiple stakeholders, allocating resources to the appropriate projects or features, and providing sufficient feedback to ensure that the requirements implemented achieve the stakeholders' goals. The specific Customer team practices are still emerging in the agile community. The practices are guided by the same values as the other XP practices.

FITTING XP TO YOUR PROJECT
Is My Project a Good Fit for XP?
Probably the most commonly debated question regarding XP is whether it can be used successfully on a particular type of project. Experience is proving that, as with other approaches to software development, the limitations often include the characteristics of the project, the people on the team, and the organization in which they work. To evaluate whether the XP practices can help a team achieve greater success on their project, consideration must be given to the project characteristics, the people on the team, and the cultures of the organizations involved in the project.

The XP Values can be used as a template to test the fit of XP to a project, team, and organization. Simply evaluate the degree to which each value is currently held by the team and the organization.

Communication. Does the team communicate constantly and effectively? Does this communication extend to the customer? Is the team's software readable and understandable (i.e., is it easy for Programmers to communicate with the code)?

Simplicity. Is the team comfortable with simple solutions? Can the team implement, without a complete design, the system prior to coding? Is the team comfortable with some ambiguity as to the exact requirements and designs? Can the team adapt often to changing requirements? Is the team working new code or code that is well designed and refactored?

Feedback. Can the team get feedback on its tasks and deliverables often? Does the team accept feedback constructively? When there are problems, does the team focus on the process to identify root causes (rather than the people)? How often does the team integrate, build, and test the complete software system?

Courage. Does the organization encourage individuals to not fear failure? Are individuals and teams encouraged to show initiative and make decisions for their projects? Are organizational boundaries easily crossed to solve problems on the project?

Typically, the greater the degree to which the team can answer these questions affirmatively, the fewer changes will be required and the easier it is for the team and organization to adopt XP. Some specific project and team

guidelines for getting started are provided next.

Getting Started

When selecting an initial project on which to try XP, one must consider the challenges of using the new practices. New practices introduce risk to a project. Care must be taken to select an initial project that is not burdened by all of the most difficult obstacles to using XP, but does address enough typical obstacles so that the success of the initial project can provide the basis for expanding to the rest of the organization.

Although most initial XP projects are not this fortunate, ideally, the initial project will have many of the following characteristics:

- Primarily new code versus legacy updates
- An identified and available source of requirements and feedback (i.e., on-site customer)
- Delivers important business value and has management visibility
- Uses an OO language/environment
- Is typical of the projects the organization will be doing in the future
- Has a co-located team in an open workspace
- Can be delivered to the end user incrementally, with a new stage once in at least every four to six weeks

In selecting the initial XP Project Team, the main attribute of the team members should be a strong commitment to delivering the project and achieving its goals using the new practices. Some healthy skepticism about XP is acceptable as long as the team members are willing to use the practices and let data and experience from the project guide any adaptations. The team ideally will have a few technical leaders familiar with other projects in the organization, but it is not desirable to have a team full of the most senior people. XP is a collaborative approach to development and, as such, the initial project will benefit from members with strong "soft" skills who prefer collaborative work environments. Beyond these characteristics, the team should be representative of teams that the organization will use in the future.

The simplest way to reduce risk on an initial project is maximize the skill of the team as quickly as possible. This can be achieved through recruiting team members that are already skilled in XP, training, or experienced coaching for an inexperienced team.

Adaptations

As teams begin adopting the XP practices, numerous obstacles and constraints must be confronted. The team may have trouble gaining access to the Customer every day. The team may have trouble co-locating to an open workspace. The team may be so large that communicating without formal documentation is not feasible. How do we adjust? Must we abandon XP? The XP Values guide teams in solving these process problems with their projects.

The Courage value guides us to aggressively confront and remove any obstacles that would add steps, artifacts, or complexity to the process. This often means letting common sense outweigh bureaucracy. For example, teams sometimes do not feel empowered to change the physical work environment to have an open workspace (i.e., change the cubicles). Often, a little courage, negotiating, and a power screwdriver will remove this obstacle. Some teams struggle to have a customer sitting with the team. The programmers develop from a requirements document and have never spoken to the customer. Although the thought of having a customer present is desirable, the logistics can seem impossible, particularly if the best person to sit with the team does not live near the team or is constantly traveling. Often, with a slight reorganization and a modified communication infrastructure, a customer can be identified who can sit with the team on a frequent basis.

Of course, courage can only take us so far. There will be constraints that interfere with our ability to implement the practices as described. A common example is legacy code. Many teams work with large code bases that do not have tests and are in dire need of design improvement. We want to aggressively move to the state where all of the code has passing tests, is understandable, and is well designed. The initial attempt is to rapidly get the code up to our new standard. Can we toss it and rewrite it? Would it really be that expensive and time-consuming to fix it? Is there other, cleaner code available with which we can replace it? Very often, the answers are No, Yes, and No, respectively, leaving team members no choice but to live with the smelly code and improve as they can.

XP Values give the team a helpful, simple tool to deal with this difficult, yet inevitable challenge. The constraint that causes a practice to be modified or abandoned is reviewed against each of the XP Values, using the following question: How will the influence of this XP

C are must be taken to select an initial project that is not burdened by all of the most difficult obstacles to using XP, but does address enough typical obstacles so that the success of the initial project can provide the basis for expanding to the rest of the organization.

TABLE 3 Constraint: Legacy Code Prevents Test-Driven Development

Value	Impact	Adaptation Alternatives
Communication	Dif cult to communicate the code's intent and design	• Wiki[a] pages to document the team's understanding of the code • Reverse-engineering tools to create models of the code • Changing pair partners at least twice a day
Simplicity	Complexity restricts some simple design alternatives	• Targeting areas of the system that warrant simpli cation • When changes are being made in a complex area, ensuring that one of the pair partners is new to the code
Feedback	It will take longer to be aware of and x errors	• Ensuring that the tests that exist are run every day • Add at least one automated test every time a defect is xed
Courage	Cannot proceed as aggressively with certain simple design	• When stories require changes to legacy code, increasing the amount of design discussion during the Iteration Planning Meeting

[a] A Wiki is a Web-based collaborative repository popular with XP teams. See www.wiki.org for more information.

Value be diminished as a result? In the case of our untestable legacy code, a quick brainstorming session by the team might yield the ideas in the Impact column of Table 3. The team discusses ways to adapt the process that is guided by the values, yielding something similar to the Adaptation Alternatives column.

Each alternative that the team considers is checked for its alignment to the values. A misaligned example, an alternative that states "all legacy code changes must be approved by a Change Control Board (CCB) prior to implementation," may be viable, but it is not simple to implement. It reduces the frequency of feedback while we wait for the CCB to meet, and takes empowerment away from the programmers, thus reducing their Courage. Other alternatives that address the constraint and that align closer to the XP Values are preferred.

Using this simple technique, teams adapt the XP Practices to their project and team needs. The importance of starting with Courage cannot be overstated. Many teams have been able to achieve a level of simplicity in their practices beyond what was thought possible. Although this may appear to introduce risk, Retrospectives after each iteration mitigate that risk by helping the team understand where additional adaptations are required.

SUMMARY

The pace of change in the software development industry remains at high. People continue to push the boundaries of known techniques and practices in an effort to develop software as efficiently and effectively as possible. Extreme Programming and Agile Software Methodologies have emerged as an alternative to comprehensive methods designed primarily for very large projects. Teams using XP are delivering software often and with very low defect rates. As the industry continues to evolve, we are likely to see additional insights on how to leverage collaborative work on more and more types of projects. ▲

Notes

1. Beck, K., *Extreme Programming Explained*, Addison Wesley Longman, 2000.
2. www.agilemanifesto.org.
3. Sliwa, C., "Agile Programming Techniques Spark Interest," Computerworld.com, March 14, 2002.
4. Jeffries, R. et al., *Extreme Programming Installed,* Addison Wesley Longman, 2001, 172.
5. It is likely true that skilled practitioners of most methods are guided by a set of values, perhaps dictated by the culture of the organization, perhaps dictated by the leadership of the team. It is the expression of the values as integral to the practices that makes XP unique.
6. Schwaber, K. and Beedle, M., *Agile Software Development with Scrum*, Prentice Hall, 2002.
7. Mary Poppendieck, *Lean Development: A Toolkit for Software Development Managers*, Addison-Wesley, to be published in April 2003.
8. Thomsett, R., *Radical Project Management*, Prentice Hall, 2002.
9. Fowler, M. et al., *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 1999.
10. For more information on good software design principles and their application on agile software projects, see Martin, R.C., *Agile Software Development: Principles, Patterns, and Practices,* Pearson Education, 2003.
11. Kerth, N., *Project Retrospectives*, Dorsett House, 2001.