

# Adopting XP

The path to—and pitfalls of—implementing  
Extreme Programming **by C. Keith Ray**

EXTREME PROGRAMMING (XP) IS A SOFTWARE DEVELOPMENT process defined by specific, simple practices. XP takes individual practices known to be good and turns all the dials up to maximum, creating a process quite unlike the traditional development process. XP doesn't have a design phase followed by an implementation phase followed by a test phase. As part of the extreme idea of doing certain good things all the time, design and testing are done throughout the project's life.

XP is always focused on the end product: every couple of weeks, an XP project delivers working code. It relies on fast and frequent feedback to help management steer the project to meet shifting requirements, help junior programmers learn good practices, and teach experienced programmers new tricks. Since it is new to many people, implementing an XP project requires careful planning and thought. You must be aware of both the things that you should do, and the things that you should not.

## Prepare Your Team

The best way to learn XP is in an experiential-learning training course. Your entire development team (including the testers, the customer, and the manager) should attend a one-week immersion course on XP. While many programming teams are learning XP based solely on books and information on the Web, it is important to actually do the practices with guidance.

Even if you *do* attend training, the minimum required reading (for at least your team leader) is

- *Extreme Programming Explained* by Kent Beck, which covers all the practices at a high level and why they support each other
- *Extreme Programming Applied* by Ken Auer and Roy Miller, which describes ways to adopt XP and how to persuade managers, customers, and programmers to try it
- *Extreme Programming Installed* by Ron Jeffries, Chet Hendrickson, and Ann Anderson, which gives more detail on all the practices, including examples of test-first and pair programming

## QUICK LOOK

- 10 guidelines to help your project launch smoothly
- 9 warnings to heed along the way

Your customer should read *Planning Extreme Programming* by Kent Beck and Martin Fowler. All programmers on your team should keep a copy of

*Refactoring* by Martin Fowler handy and study it carefully. Encourage your team to read and discuss these books and others on design, design patterns, programming, and testing. (See the StickyNotes at the end of this article for full details on these books.)

*Don't rely on just one book.* In efforts to explain XP more clearly, the description of the process is still evolving. The process described in the first book, *Extreme Programming Explained*, is described differently in the later books; for example, test-first programming (a very important practice) is not described at all in the first book.

### Start an XP Project with a Coach

Based on my team's experience, and that of the other teams in my department, I strongly recommend hiring a full-time coach for at least the first several iterations of your project. Teams need someone who can correct "bad form"—failing to follow a recommended practice, failing to apply a practice correctly, not communicating, or being blind to some other inefficiency. Coaches must also be able to program, so they can work with every member of your team, teaching them patterns for incremental designing and testing.

*Don't just make your team leader the coach.* As a cost-saving measure, you may be considering adopting XP without a formal coach. We made that mistake. Before our first XP project, my team attended a one-day training course and read *Extreme Programming Explained*. We thought that would be enough to get started. Later we realized that we were not applying all of the practices because we didn't have a coach to help us take full advantage of XP strengths. Back when we started, I was new to XP, as were my team members. It was difficult to coach a game that I was still learning. Even now that I have more experience with XP, as a team leader I have my own work to do, so I can't observe the other team members the way a full-time coach could. If your team leader acts as coach, you may end up with only half a coach; doing anything in XP only halfway will prevent you from reaping its benefits fully.

### Choose a Customer

The customer is the person who knows the requirements and has the authority

to decide on interpretations of the requirements. On my team, a product planner plays the role of customer. Yours may be a business analyst, a product or project manager, or an actual end user. The customer doesn't just write down the requirements and leave, but instead must remain easily accessible throughout the life of the project to answer questions about the requirements as they come up. XP acknowledges that a requirements document rarely contains all the relevant information, so a "living requirements document"—embodied in the person of the customer—answers or decides questions that come up during implementation.

The customer, in collaboration with a tester or QA representative, is also responsible for specifying the acceptance test for each story. You need the acceptance test specification because your story implementation is not considered complete until it passes the acceptance test. The customer may also act as the human-interface designer.

If you don't have someone to play the role of customer, then the questions arise: *Who defines your requirements? Who clears up the ambiguities in them? Who knows if your software actually meets the requirements?* If you don't have someone available to answer questions as they come up, then productivity will suffer. Programmers will have to guess at the answers, or postpone work. If they guess, they will not find that they guessed incorrectly until much later, when the real stakeholders see the product. If there is no one to fill the customer role, then maybe no one is interested in this project or its product. You may need to consider whether this project should be done at all.

### Organize Your Project's Requirements before Kickoff

XP does not specify any practices for gathering requirements. It does expect that most of the requirements are known at the beginning of the first iteration, for the release planning meeting. In my team, the customer gathers and defines requirements for months before the programming even starts.

In XP, requirements are represented as small use cases—user-centered stories. There are whole books on writing use cases and doing user-centered human interface design, but XP's format for stories is much simpler—just a few lines of text on an index card. The text on the index card is a promise of future communica-

tion of the detailed requirements. If your company starts a project with a requirements document, have the customer (perhaps with help from the coach, programmers, and testers) break the requirements document into stories before the first release planning meeting. You want a story to describe something useful to the user, like "shopper can see contents of shopping cart with product names, prices, and total." The story should be testable, like "contents display takes less than three seconds," rather than something vague, like "contents display must be fast."

In the release planning meeting at the start of the project, the customer explains these stories to the programmers and testers. The programmers estimate the time, cost, and risk involved for each story. The customer arranges the story cards from highest business value to least, tentatively assigning them to iterations, considering the cost estimates and risk evaluations from the programmers.

### Track Development Speed

There are several ways to estimate and track development speed; my team uses story points. Each story receives a rating of one to three based on the estimated time to complete it. The sum of the story points for each *completed* (partially completed doesn't count) story is the team's velocity. In the release planning meeting, the programmers guess what their velocity will be. After each iteration, we measure the actual velocity. We are then able to give an increasingly accurate estimate of the remaining project duration by dividing the sum of outstanding story points by the velocity. We use the previous iteration's velocity to avoid assigning too many story points to the current iteration.

*Make your story small enough to complete in one iteration.* We have tried rating stories up to five points or higher, but the higher rating has almost always been a mistake. We found that large stories almost always take longer than estimated, and we could seldom complete a large story in one iteration. In the release and iteration meetings, split up those large stories into small, manageable stories.

### Measure and Adjust Each Iteration

The iteration planning meeting at the start of each iteration is an opportunity for the customer to explain stories in more detail to the programmers, and to

add, remove, or reprioritize stories. Adding and dropping stories just before they are to be implemented allows an XP project to follow shifting requirements. Programmers create tasks and focus their design on the current stories, so that no time in the current iteration is wasted on stories that may be dropped or changed in the future.

Track story and task completions during the iteration, so that there are no surprises at the end. You don't want all of the stories to be only 50% done at the

## A word about Legacy Code

Since XP requires frequent testing and simple design in order to make refactoring and incremental development safe, legacy code (written pre-XP) can be a productivity killer. Legacy code rarely has up-to-date automated unit and acceptance tests. Very likely the code is complicated and hard to understand, with poor modularity that prevents unit testing. If the legacy code is a product that has been in use for a long time, then it probably contains many implementations of undocumented requirements. Attempting to rewrite the legacy code from scratch would require rediscovery of those requirements. I have observed that projects with a lot of legacy code have problems doing all of XP's practices—testing in particular.

Whenever possible, when fixing a bug in *any* code, but particularly in legacy code, first write a unit test or acceptance test that fails because of that bug. Then fix the bug. When all the tests pass, you'll have some assurance that you've fixed the bug without creating new bugs. Acceptance tests may be easier to write than unit tests, because the whole may be more easily tested than its parts. Over the course of the project, you will eventually get most of the important parts of the legacy code unit-tested and acceptance-tested, and it becomes safer to refactor. Any new code should be written test-first—this enforces modularity of the code, and it will be easier to insert new, modular code into the appropriate place in the legacy code.

end of the iteration, because then your team will not have delivered *any* value to the customer, and your velocity will be zero. If there is a danger of not finishing all stories by the end, ask the customer to select which stories are most important, so the team can concentrate on those. Don't extend an iteration to complete a story, because that makes velocity less accurate, throwing off planning. Schedule the unfinished stories into the next iteration.

Use the simplest and most visible form of tracking possible within your team. My team posts our stories and tasks onto Wiki Web pages and updates the pages upon completion. The Wiki Web is a simple, collaborative Web application that allows users to create and edit Web pages using any browser. Many teams post their progress on the wall, using white boards and index cards. When the whole team knows every day what has been done and what needs to be done, they can focus on their work with confidence, and have plenty of warning when a story is underestimated or roadblocks prevent progress. Duplicating tracking information in Microsoft Project charts or spreadsheets is unnecessary, unless stakeholders outside the team require those formats.

If velocity is consistently slower than expected, or stories are added to the project, recompute the total project time estimate. Failure to do so will give your stakeholders unpleasant surprises. You don't want hidden scope creep pushing out the schedule. XP recommends reducing scope rather than changing the project duration. Since the most valuable stories are done first, reducing scope usually means dropping the least important stories. You can also reduce the scope by simplifying stories. Though not recommended by XP, some projects will extend the schedule to meet their stakeholder's requirements, rather than reduce scope.

*Keep track of unplanned work.* Often, programmers on your team have to handle "emergency" work on another project when they should be working on your project, effectively lowering the team velocity. Keep a record of this work to make it clear to management why the velocity is slower than expected. I have heard of one team that used red index cards to track these unplanned tasks, and as a result, their management made efforts to decrease the "emergencies" assigned to that team.

## Do Pair Programming

Pair programming consists of collaborative design, testing, and implementation. My experience suggests that the amount of code and tests generated by a pair is roughly equivalent to that of two people working alone without reviews, but the quality is better in pair programming because reviewing is built in. Traditional design and code reviews delay feedback because the review occurs after some units of designing or coding have been completed. Pair programming prevents many coding and design mistakes, reducing rework and bugs.

*Don't let naysayers keep you from pair programming.* When talking about XP, many programmers—about 90%—will say that they don't expect to enjoy pair programming or be productive doing it. Rumor has it that 90% of people who actually learn how to do pair programming say they find it very productive and even enjoy it. (I was in both 90% groups.) When you first try pair programming, you may find yourself making a dozen mistakes in a few minutes, but this just comes from your inexperience. You'll find your partner is just as inept. Talk with your partner and exchange control of the keyboard often. After a week or two of pair programming, you have a good chance of finding yourself in both 90% groups, too.

## Work Toward Simple Design

XP practices simple design. Kent Beck's rules of simple design are (in priority order): "(1) passes all the tests, (2) has no duplicate logic, (3) states every intention important to the programmer, and (4) has the fewest classes and methods." Duplicate code not only allows bugs to remain unfixed; it also slows down making changes for new features. Additionally, I would expand "states every intention" to say that each class and method in a class should have one responsibility. I also recommend that concrete classes should not depend on other concrete classes unnecessarily, because that makes unit testing more difficult, as well as reflecting poor design.

## Test-First Programming

To create new code that has simple design, XP strongly recommends *test-first programming*, also called test-driven development. Programmers write one small test, run it to see it fail, and then write just enough code to pass that test. They

repeat this cycle every few minutes. It is one of the most powerful tools in XP. Test-driven development is a design process very unlike traditional unit testing. The programmer uses the tests to drive the design, saving them into a suite of regression tests (still called *unit tests*, in spite of their nontraditional creation). The tests provide immediate feedback on the design and code in progress and lasting feedback that the code is still correct when programmers modify it to add new features later.

These unit tests must run quickly because you must run them often. To speed up the tests, use boundary analysis, mock objects, and other such testing techniques. Take advantage of their white-box nature to further simplify the tests. To pin down which tests were not running quickly, I added output of timing information to my unit test frameworks. For some slow tests, I simplified the tests; in other cases, I rewrote the code under test to be more efficient.

Unfortunately, many programmers learned and are most comfortable with “test-after programming.” When doing “test-after,” they almost always write some code that is not needed or is unnecessarily complicated. Feedback (“Is this code good?”) is slower, and it is *much* more difficult to get near-100% code coverage. You can do XP with these traditional-style unit tests, but it is harder psychologically to test what you believe is already-working code. This bulky and more complicated “test-after code” is also harder to maintain and refactor. Use pair programming and coaching to encourage your programmers to write test-first. Eventually they will see its benefits when they realize how much extra work they are doing when they write test-after.

*Don't abandon traditional testing!* Acceptance tests are similar to traditional tests and should be written by testers if you have testers on your team. They should include multiple-platform tests, stress tests, etc., as appropriate for your project. Look for ways to automate acceptance testing as early as the first iteration. It may be difficult because prepackaged acceptance testing frameworks are not as easily available as unit-test frameworks. Automated tests ensure, more quickly and easily than manual tests, that previously implemented features don't break as the project goes forward. Additionally, be aware that XP, while empha-

sizing quality, is aimed at “typical” application development. XP projects in especially safety-critical domains must have additional testing and inspections appropriate for that domain.

*Learn from your bugs!* Whenever a bug slips through testing to the customer or hands-on QA, the team should study it and learn how to beef up their testing to prevent that kind of bug from slipping through again. This provides feedback to help the team improve their unit and acceptance tests. In my team, the customer may prioritize a bug fix as if it were a new story in iteration planning. We only did this for bugs that could be considered features (bug: “system limits messages to 32 lines of text, change this to allow 300 lines of text”). Fix other kinds of bugs as soon as possible, since they indicate a problem with either the design or your process.

### Refactoring

Refactoring is the way to keep your code in a state of simple design. It is important for everyone associated with the team to realize that refactoring is not the same as rewriting. Refactoring improves the design of the code while keeping the same functionality. To use a simple example, suppose your code were  $(3 * 4) + (3 * 5)$ . Changing it to  $(3 * (4 + 5))$  would be a refactoring, because the answer is the same. It is better code because if you want to change the 3 to something else, you only have to change it in one place. Many refactorings eliminate duplication.

Refactoring is very disciplined. Programmers learning refactoring should follow the steps in Fowler's *Refactoring* exactly. Refactoring takes very small, nearly foolproof steps, but you still need to run your suite of unit tests after every few steps to make sure you don't break anything.

*Refactoring without unit tests is almost certain doom.* I had to throw away half of a weekend's refactoring work once because I didn't have unit tests and manual testing to confirm every step of the process. Refactoring with the support of unit tests allows me to change the internals of a class, knowing within minutes when I make a minor error in any one of the refactoring steps.

If you don't do enough refactoring, your project will rack up “design debt,” the accumulation of many small code

problems. Design debt is what makes many long-term projects become unmaintainable. Refactoring every day pays back the design debt while it is fresh, and before it has a chance to accumulate. If your team finds that it needs to spend a whole week refactoring (and not delivering new features), then your team hasn't been refactoring often enough.

### Prepare for Cultural Conflicts with XP

If your project is adopting XP, but the rest of your organization does something else, watch out for cultural conflicts and misunderstandings.

If your XP project requires components from other teams, you will need those components to be high quality when they are first delivered. If the team providing the component believes in “get all the features in first, then fix the bugs later,” that conflicts with an XP project incrementally delivering high-quality features. In self-defense, your team may have to write unit tests or acceptance tests of those components in order to easily localize bugs and regressions. You could provide that other team with those tests, but if their culture doesn't mandate it, they are not likely to run them.

If your project is divided into multiple “specialist” teams, not all doing XP, there will be problems synchronizing the teams and integrating their components. The XP team will incrementally deliver some features each iteration, and unless the other teams have the same iterative approach, they will want to deliver “final” versions of all of its features later. Encourage pair programming across specialties—this not only eases integration of code, but also allows everyone on the team to see, understand, and help ease the problems encountered by each specialist.

Some managers forbid pair programming—they haven't realized that XP measures productivity (velocity) at every iteration, and they don't accept that pair programming provides high quality at a quick pace. If management is opposed to pair programming, add collaborative, *detailed* design sessions, and substitute code reviews for the pair programming. Try to get permission to record the team's quality and productivity in order to compare code reviews to pair programming. Try emphasizing the mentoring aspects of pair programming when talking to management. If management refuses to allow even code reviews, then the situation is too dysfunctional for XP.

Dysfunctional organizations where people refuse to work on the same team, or try to hide their code from the view of others (even while pair programming!), or even prevent code reviews, have problems that XP can't solve. XP requires a team that actually works together. There are reports of projects where one or two soloists quit rather than do XP, but the project went on to success anyway; however, if your programming group is composed of committed soloists or your management is opposed to collaboration in this form, XP may not be for you.

*Be prepared to educate others on XP practices.* Those unfamiliar with XP may not immediately accept its practices. For example, *Extreme Programming Examined* reports that a company did an XP project that delivered good value at a good, predictable pace; however, management, not understanding the reasoning behind the practice of Sustainable Pace, did not think of the project as a total success, because they thought the programmers did not put in enough hours.

### **Start Small, Finish Successful**

Be thoughtful when adopting XP. If XP is new to your team, take time to learn new ways of working. Consider starting with a small team for a short-duration project (up to four one-week iterations), creating a small but useful product with an in-house customer. Your project can be as visible within the company as you wish, but you won't be risking your company on an important project while your team is learning a new process. When something seems to be going wrong (more than a few bugs or no production code released each iteration), investigate it. Focus on your end product (the software), but don't ignore design and tests. Your XP project should deliver working code each one- to two-week iteration.

### **Conclusion**

XP projects can be extremely successful when done right. XP focuses programmers on the end product, yet makes design and testing parts of the development process. It doesn't force programmers to do things they find difficult or boring

(writing and reviewing documents), but plays to their strengths (writing software and tests). XP helps programmers learn to write better code and tests. XP helps create order out of chaos and delivers value quickly. Try it with a small project, keeping in mind the path to successful adoption, as well as the pitfalls. I'm sure you'll like the results. **STQE**

---

*C. Keith Ray is a team leader with more than sixteen years of professional software developer experience. He started using XP on a project in 2000, and based on its success, persuaded his department to adopt XP as their development process. He maintains a MiniFaq on XP at <http://homepage.mac.com/keithray/xpminifaq.html>.*

**STQE magazine is produced by STQE Publishing, a division of Software Quality Engineering.**