

Extreme Programming: Fast Iterations Mean Fewer Nasty Surprises

By Bart Eisenberg
Pacific Connection
Software Design magazine

It's a story familiar to every software designer. You start a project, talk to the "customer," write up a detailed spec sheet, get it approved, do the coding, and produce a prototype. The customer looks at it and says: "yes, that's what I asked for. But it's not quite what I want." And so you re-write the spec sheet, modify the code, and try again. Sometimes, this cycle drags on for months, or until the budget runs dry. The Standish Group International, a research firm that has documented the failure rate of application projects, says that in the year 2000, only 28 percent came in on time, within budget, as specified.

There has to be a better way—and perhaps there is. Extreme Programming, usually abbreviated as XP, is a method for producing software in which there are fewer nasty surprises at the end of each development cycle. With XP, the customer is heavily involved throughout the process, development cycles are reduced to a matter of weeks, testing is ongoing and frequent, programmers work in pairs and have normal working schedules, like other people.

Two developers—Kent Beck and Ward Cunningham—invented XP, based on their work in Smalltalk. Beck's book, *Extreme Programming Explained: Embracing Change*, is still the definitive work on the subject, with Cunningham's Wiki Wiki Web project, (<http://c2.com/cgi/wiki?WikiWikiWeb>) a massively hyperlinked resource, has become a place for ongoing discussion.

Ron Jeffries, an independent consultant and one of the earliest XP practitioners (he runs the website www.extremeprogramming.com), says that XP is spreading through "word of mouth and word of Web," through a dozen or so books on the subject, and two generations of programmer consultants. While XP works best among smaller development groups, some large companies are at least testing it. Among the best known are Symantec, pharmaceutical maker GlaxoSmithKline, Hewlett-Packard and Disney. "The objections to XP are becoming more reasoned, less reactionary," he says, with more people willing to make the case online for why the methodology works.

"XP comes from hard-core practitioners who have analyzed the process up and down and know what works," says Joshua Kerievsky, founder of the Berkeley-based consulting firm Industrial Logic. "Many methodologies advocate heavy, up-front design work. Teams build components, but the components don't talk to each other very well because the people don't talk to each other. Organizations aren't co-located and don't share code—so it's no wonder they there are problems when they try to integrate the components."

“The interesting thing about XP is that it's a grass roots movement from practioners, not a theory from management schools or Carnegie Mellon's Software Engineering Institute,” says John Levy, a management consultant and long-time observer of XP*. “With XP, the assumption is that people don't really know what they want from their software until they see it working. That's the driving force. Customers do change their mind; it's human nature. Instead of seeing that as a disturbance, XP sees it as a fundamental part of the process. But because you've avoided complexity in your design, the code is easier to change. In fact, the code is developed with change in mind.”

(* The name “Extreme Programming” has been controversial because it resembles the term “Extreme Sports,” including high-risk versions snowboarding and skateboarding in which the partipants spend part of their time airborne. As one blog writer put it: a manager “is going to picture the long haired youth with bad attitude that cut him up on the ski slopes of Aspen.” But on a recent visit to Lake Tahoe ski resort, I saw far more snowboarders than skiers. If XP were to do anywhere near as well, it would be the most sucessful revolution in software development since object-oriented programming.)

XP principles: a closer look

Levy and others have distilled XP down to a set of principles, some of which seem obvious, others quite radical. The basics of the approach come down to this: Keep the design simple. Your customer should be readily available. Test the code early and often. Don't invest so much in the code that you're afraid to change it. Develop release versions that the customer can try out every three weeks or less. Develop the code in two-programmer teams. Programmers should have a life beyond coding. (For a broader view of XP, see “The 12 Core Practices of XP” below.)

These operational principles become for software development what “rapid prototyping” is for product development—a way of getting user feedback so that the project stays on course. In industrial design and architecture, rapid prototyping can be seen everywhere—from the quick-and-dirty mockups the design firm IDEO built to create the Palm V to Frank Gehry's duct tape and plastic models for his curvaceous buildings. The IDEO creed admonishes its designers to “fail early and often,” because only through rapid mistakes and corrections do they ever get it right. XP has the same philosophical roots. The methodology insists that constant feedback is a necessity—for the programmer and customer alike. That without it, your project will veer off course and become another statistic—coming in late, over-budget, or below expectations. Or not coming in at all.

Keep the design simple

In XP, the multi-page specification sheet is avoided. Instead, software designers collaborate with the customer to write a set of “user stories”—operational descriptions short enough to keep on index cards that decrbe the system and its features. Index cards may seem like a crude way to spec out a system, but the medium is fluid, allowing people

to spread the cards out, pass them around, and trade back and forth so that the most important features of the system come to the foreground.

A user story is a customer-oriented description of what the system should do . That means, it should be written in a way that everyone understand, and should be short enough to fit on a small pad of paper. “A user story keeps you focused on what the customer needs,” says Levy. “That may change over time, but that's why you have the customer nearby. Levy contends that programmers, being who they are, tend to add complexity into a project. “Complexity is interesting; programmers like it.” The problem is that complexity for its own sake also leads to code that fails—and to code that’s harder to change. “Programmers keep adding functionality for anticipated extensions, rather than what is needed now. XP tends to enforce the principle of ‘YAGNI’—which stands for: ‘You ain’t going to need it.’ In other words, to meet the requirements of the next release, don’t put in anything that’s not absolutely needed.”

Joshua Kerievsky says that XP is very good at keeping the goals realistic. Because the iterations are so quick, the team quickly understands what can and cannot be done. “As a result, the team is forced to prioritize, as well as negotiate with customers. You may be able to fool yourself on what can be done over three months, but a two week deadline leaves little to the imagination, and so all parties talk, compare notes, and compromise. The process forces people to go at a steady, realistic pace. More subtly, it forces customers to come up with creative ways get as much as they can out of every iteration.” Maybe the data stream doesn’t have to be complete for the next revision Maybe the entry screen can be simpler. People ask themselves what would work under the circumstances.”

The customer should be readily available.

In pure XP, your customer—the person who will actually use the software--is not just available by email, phone or an occasional meeting. He or she is supposed to be on hand, in the same building, or even the same room. When you have a question about what some feature means, you should be able to lift your head and ask—not that somebody who stops by every two weeks.

Test the code early and often

There are two types of tests in XP: unit tests and acceptance tests. But a clearer nomenclature may be Jeffries’ suggestion: he calls them programmer and customer tests. Programmer tests are built by the programmer to test the quality of the code. Customer tests are designated by the customer to prove that what was requested, was actually delivered.

If there is a core centerpiece to XP, testing is it. XP is based on “test-driven development.” You build the test first, *then* build the code—not the other way around.

Jeffries contends that test-driven development is often misunderstood. “It doesn’t mean I write a test on Monday and work all week to make it work,” says Jeffries. “It means I write a programmer test, and 10 minutes later, I test it and it works. Literally.” Each test represents just a tiny improvement in the code, and as a result, it takes just a few lines of code to pass the test. “I find that in my personal work, if I go a half hour without making a test work, I’m almost certainly in for an ugly debugging session.” And then there’s refactoring—the constantly improvement of code to make it better. “Re-factoring doesn’t mean throwing away a bunch of code and rewriting it. It means re-organizing code to improve its quality while keeping its functionality.” Jeffries says that customer tests can be done less frequently—once a day.

Jeffries thinks that the user stories are really invitations to a conversation between the programmer and customer, trying to figure out what the customer really wants. After that comes confirmation—an acceptance test—that confirms the customer’s expectations. “Most of these tests take the form of inputs and outputs—which is why Ward Cunningham’s FIT product is so apt in this area. (FIT is an executable table—the table, written in HTML, with a “fixture”—code that enables the table to drive the product. The table then displays the results with green and red shading. Acceptance testing could also be done through a spreadsheet or a scripting language.)

“A lot of people, when they first see XP, only see the informal paper requirements—the user stories,” says John Brewer, an independent software consultant. “But each user story should represent a system feature, and each feature should have a test associated with it. Hence if the system passes the test, the assumption for everyone—programmers and customers alike—is that the feature is complete.

Brewer says that the biggest single key to XP’s value is what are called ‘time-boxed iterations.’ “Traditional software projects have long-term milestones, the code may be 80 percent complete for months or years at a time.” That never happens with XP’s short iteration periods. Each test is modest in scope, but it either passes or fails—there’s no in-between. “Ideally, you should have automated acceptance tests for every story, or at least scripted acceptance tests that a neutral third-party could manually verify. Paper requirements are less useful than executable requirements that you can test the system against. Requirements that exist in the form of code are not ambiguous, the way paper requirements written in English or Japanese can be.”

Don’t invest so much in the code that you’re afraid to change it.

“Ordinarily, nobody wants to change a module that works--because also many resources were committed to making it work,” says Levy. “By keeping the modules simple, they tend to work sooner. And because less work is required to produce simple modules, there’s less at stake in improving them.” The XP term for this process of quick modification and improvement is “refactoring.”

“Re-factoring doesn’t mean throwing away a bunch of code and rewriting it,” says Jeffries. “It means re-organizing code to improve its quality while keeping its functionality.”

Develop release versions every three weeks or less.

Traditional software development can go months without the customer seeing any results. With XP, three weeks is the outer limit and some developers think it should be weekly. Consultant John Brewer says that the “shipping muscles” of a development team can atrophy if the releases are months apart. Frequent iterations mean that the development gets used to completion, even if each “shipment” moves just microscopically forward. Frequent iterations also save programmers from the weeks of 18-hour days needed to make a deadline. With XP, you might rush to completion, but not for days at a time.

“With XP release planning the customer breaks the product into a set of features—each of which must be implementable within an iteration. If a feature is too big, you must split it, so that each feature has an atomic unit of business value, that you can test and measure. Because the feedback loop is based on a release cycle, at the end of every cycle, we produce a system that is capable of being released to end users. The early iterations may not do much. A desktop application might put up a window and quit, or log in and bring up an empty web page. But the idea is to get a running system that you can burn on a CD ROM, hand off to the customer and say here’s the newest release. And that system shows everyone precisely where the project is to date.”

Develop the code in two-programmer teams

The most controversial part of XP is what is known as “pair programming”. One programmer codes, the other looks on and critiques. After a while, the two swap places. The advantage, says Levy, is immediate code review, with mistakes caught immediately. “Pair programming also enforces the discipline of explaining to your partner what you are doing, which helps clarify the thinking of the one writing the code,” says Levy. “Pair programming seems like an extravagant use of talent, but when you look at what debugging costs, it pays for itself.”

Programmers should have a life

Not so much a tenet of XP as a result of its practice: programmers should be able to live normal lives. The 18-hour all-night programming session will happen, but it shouldn’t go on for weeks. Doing so not only affects the programmer’s health, well-being, and love life, but results in mistakes and inefficiencies.

Managing expectations

Jeffries says that the low success rate for software projects may have less to do with execution than with expectations—from programmers and customers alike. XP helps keep both groups thinking realistically.

“One of the most important things about XP is that we learn how to estimate more accurately what we can do and how long it will take,” he says. But the discipline of building the software, continuously keeping it integrated, and testing features every week or two has another impact, as well: keeping the project grounded in reality. When asked, people want every feature they can imagine built into the software. But if you press them, you discover that not all features are of equal importance. “If you do the most important third of them first, you’ve probably covered 80 to 90 percent of the product value. The result is a product that’s shippable much more quickly—and shippable every week—with people recognizing that they could actually use it.

Jeffries contrasts XP’s short interval approach with the more widely known Rational Unified Process, or RUP, which is now sold by IBM. RUP also talks about integration, but because the intervals tend to be so long, developers wind up focusing for a big release. By thinking in terms of a long-term deadline, he says, you miss the fact that the software might be shippable right now—if, as XP requires, your iterations have kept up with customer expectations. The problem with traditional development is that customers don’t actually see the software under construction. “When they see their own suggestions coming into shape out of the fog and looking like a finished product, then they get it. As always, it’s the test drive that tells you.” Jeffries thinks this is true both for internal IT products and shippable off-the-shelf software. He argues that a minimal product that meets all the acceptance tests of its customers and is upgraded by the week is worth releasing. People will start to use it and request more features along the way.

Team size and other practicalities

XP proponents argue about how large a development team can get before XP ceases to work. The general consensus seems to be 20, at the outer limit. As you go larger, the project becomes more paper-driven, less test driven. The emails increase, the conversations decrease, the coordination gets bogged down and the project becomes more conventional. One way around this problem is to divide the team up. “We’ve worked with teams as large as 60 people—consisting of several sub teams,” says Joshua Kerievsky. “So it does scale.”

On the other hand, says Kerievsky, “we’ve worked at organizations where XP definitely doesn’t have a place—like corporate environments with heavy-handed methodologies already in place. Some organizations, for example, evaluate strictly on their data. If they demand a design for the database before building the system, it doesn’t work. Under XP, the systems and the database evolve together.”

And then there’s the question of how closely you must adhere to the principles of XP and still derive some benefit. John Brewer says it’s not always possible to practice what he calls “canonical XP.” On one project, for example, he and his partner did pair programming for the first set of iterations, then split off to work on their own. They had to, he says, because they work out of different locations. Nor was it possible to have a client in the room. “Having an onsite customer can be difficult,” he says, “especially in product-oriented development where the product manager is an ideal customer—but is

often out in the field.” The programmers compensated for the distributed team by dropping the time between iterations down to just a single week.

Still, Brewer says that programmers new to XP should try to adhere as closely as possible to the complete methodology. “The biggest thing people don’t get about XP is that some of the practices don’t make much sense in isolation, but they work together in ‘non-obvious’ ways. Which means it’s a good idea to try all the practices together before you make a decision about which to discard.”

But Ron Jeffries argues that while practicality may be a debatable point, it does not get you off the hook—especially if your team is new to XP. If you want the full benefits of Extreme Programming, you need to adhere to it completely. “The value of doing XP completely is that only then will you know how to adapt the methodology to your own environment.” It’s like golf, he says. First learn to swing the club like the pro tell you to. “Then you can add your own personality quirks. If you do that your first day out on the links, you’ll continue to hit the ball into the rough.”

Sidebar:

The 12 core practices of XP—from John Brewer’s Extreme Programming FAQ. (Reprinted with permission)

1. **The Planning Game:** Business and development cooperate to produce the maximum business value as rapidly as possible. The planning game happens at various scales, but the basic rules are always the same:
 1. Business comes up with a list of desired features for the system. Each feature is written out as a **User Story**, which gives the feature a name, and describes in broad strokes what is required. User stories are typically written on 4x6 cards.
 2. Development estimates how much effort each story will take, and how much effort the team can produce in a given time interval (the iteration).
 3. Business then decides which stories to implement in what order, as well as when and how often to produce a production releases of the system.
2. **Small Releases:** Start with the smallest useful feature set. Release early and often, adding a few features each time.
3. **System Metaphor:** Each project has an organizing metaphor, which provides an easy to remember naming convention.

4. **Simple Design:** Always use the simplest possible design that gets the job done. The requirements will change tomorrow, so only do what's needed to meet today's requirements.
5. **Continuous Testing:** Before programmers add a feature, they write a test for it. When the suite runs, the job is done. Tests in XP come in two basic flavors.
 1. **Unit Tests** are automated tests written by the developers to test functionality as they write it. Each unit test typically tests only a single class, or a small cluster of classes. Unit tests are typically written using a unit testing framework, such as [JUnit](#).
 2. **Acceptance Tests** (also known as **Functional Tests**) are specified by the customer to test that the overall system is functioning as specified. Acceptance tests typically test the entire system, or some large chunk of it. When all the acceptance tests pass for a given user story, that story is considered complete. At the very least, an acceptance test could consist of a script of user interface actions and expected results that a human can run. Ideally acceptance tests should be automated, either using the unit testing framework, or a separate acceptance testing framework.
6. **Refactoring:** Refactor out any duplicate code generated in a coding session. You can do this with confidence that you didn't break anything because you have the tests.
7. **Pair Programming:** All production code is written by two programmers sitting at one machine. Essentially, all code is reviewed as it is written.
8. **Collective Code Ownership:** No single person "owns" a module. Any developer is expected to be able to work on any part of the codebase at any time.
9. **Continuous Integration:** All changes are integrated into the codebase at least daily. The tests have to run 100% both before and after integration.
10. **40-Hour Work Week:** Programmers go home on time. In crunch mode, up to one week of overtime is allowed. But multiple consecutive weeks of overtime are treated as a sign that something is very wrong with the process.
11. **On-site Customer:** Development team has continuous access to a real live customer, that is, someone who will actually be using the system. For commercial software with lots of customers, a customer proxy (usually the product manager) is used instead.

12. **Coding Standards:** Everyone codes to the same standards. Ideally, you shouldn't be able to tell by looking at it who on the team has touched a specific piece of code.

#