Extreme Programming Explained note

Brief Introduction

Extreme Programming Explained explains the virtues of the Extreme Programmer and shows you how to develop them. The book is clear and readable -- even funny. Chapters are short and to the point. It talks about the thinking behind XP—its roots, philosophy, stories, myths. It is intended to help you make an informed decision about whether or not to use XP on your project. Beck uses the metaphor of driving to bring his point across. (Driving is not about pointing in the right direction and maintaining that course, it's about making slight corrections all of the time.)

Abstract

I: The Problem

The basic problem of software development is risk. Here are some examples of risk:

- Schedule slips
- Project canceled
- System goes sour
- Defect rate
- Business misunderstood
- Business changes
- False feature rich
- Staff turnover

That's the whole XP development cycle. Notice that:

- Pairs of programmers program together.
- Development is driven by tests. You test first, then code. Until all the tests run, you aren't done. When all the tests run, and you can't think of any more tests that would break, you are done adding functionality.
- Pairs don't just make test cases run. They also evolve the design of the system. Changes aren't restricted to any particular area. Pairs add value to the analysis, design, implementation, and testing of the system. They add that value wherever the system needs it.
- Integration immediately follows development, including integration testing.

there are four variables in software development:

- Cost
- Time
- Quality
- Scope

There is a strange relationship between internal and external quality. External quality is quality as measured by the customer. Internal quality is quality as measured by the programmers.

If you dropped important functionality at the end of every release cycle, the customer would soon get upset. To avoid this, XP uses two strategies:

1. You get lots of practice making estimates and feeding back the actual results. Better estimates reduce the probability that you will have to drop functionality.
2. You implement the customer's most important requirements first, so if further functionality has to be dropped it is less important than the functionality that is already running in the system.

Several factors come out of the story about what made our code easy to modify, even after years of production:

- A simple design, with no extra design elements—no ideas that weren't used yet but were expected to be used in the future.
- Automated tests, so we had confidence we would know if we accidentally changed the existing behavior of the system.
- Lots of practice in modifying the design, so when the time came to change the system, we weren't too afraid to try it.

We need to control the development of software by making many small adjustments, not by making a few large adjustments, kind of like driving a car. This means that we will need the feedback to know when we are a little off, we will need many opportunities to make corrections, and we will have to be able to make those corrections at a reasonable cost.

The four values of XP are:

- Communication
- Simplicity
- Feedback
- Courage

The effect of testing, pairing, and estimating is that programmers and customers and managers have to communicate.

XP's design strategy resembles a hill-climbing algorithm. You get a simple design, then you make it a little more complex, then a little simpler, then a little more complex. The problem with hill- climbing algorithms is reaching local optima, where no small change can improve the situation, but a large change could.

Here are the fundamental principles:

- Rapid feedback—Learning psychology teaches that the time between an action and its feedback is critical to learning.
- Assume simplicity—Treat every problem as if it can be solved with ridiculous simplicity.
- Incremental change—Big changes made all at once just don't work.
- Embracing change—The best strategy is the one that preserves the most options while actually solving your most pressing problem.
- Quality work—Nobody likes working sloppy.

If we code it together, though, I can see in the logic you write the precise shape of your ideas. Again, I see the shape of your ideas not as you see them in your head, but as they find expression to the outside world.

Since we must have the source code, we should use it for as many of the purposes of software engineering as possible. It turns out that code can be used to communicate—expressing tactical intent, describing algorithms, pointing to spots for possible future expansion and contraction. Code can also be used to express tests, tests that both objectively test the operation of the system and provide a valuable operational specification of the system at all levels.

The programmers need to make their confidence concrete in the form of tests so everyone else can share in their confidence. The customers need to prepare a set of tests that represent their confidence, "Well, I guess if you can compute all of these cases, the system must work."

Design is part of the daily business of all programmers in XP in the midst of their coding. But regardless of the strategy used to achieve it, the activity of design is not an option. It must be given serious thought for software development to be effective.

II: The Solution

First, here are all the practices:

- The Planning Game—Quickly determine the scope of the next release by combining business priorities and technical estimates. As reality overtakes the plan, update the plan.
- Small releases—Put a simple system into production quickly, then release new versions on a very short cycle.
- Metaphor—Guide all development with a simple shared story of how the whole system works.
- Simple design—The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.
- Testing—Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write tests demonstrating that features are finished.
- Refactoring—Programmers restructure the system without changing its behavior to remove duplication, improve communication, simplify, or add flexibility.

- Pair programming—All production code is written with two programmers at one machine.
- Collective ownership—Anyone can change any code anywhere in the system at any time.
- Continuous integration—Integrate and build the system many times a day, every time a task is completed.
- 40 hour week—Work no more than 40 hours a week as a rule. Never work overtime a second week in a row.
- On-site customer—Include a real, live user on the team, available full-time to answer questions.
- Coding standards—Programmers write all code in accordance with rules emphasizing communication through the code.

Business people need to decide about

- Scope—How much of a problem must be solved for the system to be valuable in production? The business person is in a position to understand how much is not enough and how much is too much.
- Priority—If you could only have A or B at first, which one do you want? The business person is in a position to determine this, much more so than a programmer.
- Composition of releases—How much or how little needs to be done before the business is better off with the software than without it? The programmer's intuition about this question can be wildly wrong.
- Dates of releases—What are important dates at which the presence of the software (or some of the software) would make a big difference?

Technical people decide about

- Estimates—How long will a feature take to implement?
- Consequences—There are strategic business decisions that should be made only when informed about the technical consequences. Choice of a database is a good example. Business might rather work with a huge company than a startup, but a factor of 2 in productivity may make the extra risk or discomfort worth it. Or not. Development needs to explain the consequences.
- Process—How will the work and the team be organized? The team needs to fit the culture in which it will operate, but you should write software well rather than preserve the irrationality of an enclosing culture.
- Detailed scheduling—Within a release, which stories will be done first? The programmers need the freedom to schedule the riskiest segments of development first, to reduce the overall risk of the project. Within that constraint, they still tend to move business priorities earlier in the process, reducing the chance that important stories will have to be dropped toward the end of the development of a release.

The practices support each other. The weakness of one is covered by the strengths of others.

We will manage the overall project using business basics—phased delivery, quick and

concrete feedback, clear articulation of the business needs of the system, and specialists for special tasks.
Management strategies that are balanced toward centralized control are also difficult to execute, because they require lots of overhead on the part of those being managed.

We will create an open workspace for our team, with small private spaces around the periphery and a common programming area in the middle.
Taking control of the physical environment sends a powerful message to the team. They are not going to let irrational opposing interests in the organization get in the way of success. Taking control of their physical environment is the first step toward taking control of how they work overall.

One key to our strategy is to keep technical people focused on technical problems and business people focused on business problems. The project must be driven by business decisions, but the business decisions must be informed by technical decisions about cost and risk.
Following this model, business people should choose

- The scope or timing of releases
- The relative priorities of proposed features
- The exact scope of proposed features

Planning is the process of guessing what it will be like to develop a piece of software with a customer. Some of the purposes of planning are to

- Bring the team together
- Decide on scope and priorities
- Estimate cost and schedule
- Give everyone involved confidence that the system can actually be done
- Provide a benchmark for feedback

Unlike the management strategy, the development strategy is a radical departure from conventional wisdom—we will carefully craft a solution for today's problem today, and trust that we will be able to solve tomorrow's problem tomorrow.

We will continually refine the design of the system, starting from a very simple beginning. We will remove any flexibility that doesn't prove useful.
XP works against many programmers' instincts. As programmers, we get in the habit of anticipating problems. When they appear later, we're happy. When they don't appear, we don't notice. So the design strategy will have to go sideways of this "guessing at the future" behavior.
This leads us to the following design strategy.
1. Start with a test, so we will know when we are done. We have to do a certain amount of design just to write the test: What are the objects and their visible methods?
2. Design and implement just enough to get that test running. You will have to design

enough of the implementation to get this test and all previous tests running.
3. Repeat.
4. If you ever see the chance to make the design simpler, do it. See the subsection What Is Simplest? for a definition of the principles that drive this.

We will write tests before we code, minute by minute. We will preserve these tests forever, and run them all together frequently. We will also derive tests from the customer's perspective.

III: Implementing XP

Adopt XP one practice at a time, always addressing the most pressing problem for your team. Once that's no longer your most pressing problem, go on to the next problem.
Solving the most pressing problem also addresses the objection to XP that it is "one size fits all." In adopting each practice, you will shape it to your situation. If you don't have a problem, you won't even consider solving it the XP way.

Projects that want to change their existing culture are far more common than projects that can create a new culture from scratch. Adopt XP on running projects a little at a time, starting with testing or planning.
If the choice is switch to XP or be fired, first realize that your chances of consistently adopting new practices aren't very good. Under stress, you revert to old habits. You already have lots of stress. Your chances of successfully making the switch are drastically reduced.

The ideal XP project goes through a short initial development phase, followed by years of simultaneous production support and refinement, and finally graceful retirement when the project no longer makes sense.
Experts sometimes develop habits that are based on value systems not entirely in tune with extreme programming. The team will have to be comfortable with the practices they choose. Saying "The expert said so" isn't very satisfying when the project is spiraling out of control.
The purpose of the planning phase is for the customers and programmers to confidently agree on a date by which the smallest, most valuable set of stories will be done.
Ideally, at the end of every iteration, the customer will have completed the functional tests and they will all run. Make a little ceremony out of the end of each iteration—buy pizza, shoot off fireworks, have the customer sign the completed story cards.
If you begin finding lots of ideas that you can't justify putting into the system for this release, make a visible list so everybody can see where you will be going after this release goes into production.
Maintenance is really the normal state of an XP project. You have to simultaneously produce new functionality, keep the existing system running, incorporate new people into

the team, and bid farewell to members who move on.

Certain roles have to be filled for an extreme team to work—programmer, customer, coach, tracker.
Above all, you must be prepared to acknowledge your fears. Everybody is afraid—

- Afraid of looking dumb
- Afraid of being thought useless
- Afraid of growing obsolete
- Afraid of not being good enough

The best customers are those who will actually use the system being developed, but who also have a certain perspective on the problem to be solved.
The skill you need to cultivate most is the ability to collect the information you need without disturbing the whole process more than necessary.
If you're the big boss, what the team needs most from you is courage, confidence, and occasional insistence that they do what they say they do.

The full value of XP will not come until all the practices are in place. Many of the practices can be adopted piecemeal, but their effects will be multiplied when they are in place together.

Even though the individual practices can be executed by blue-collar programmers, putting all the pieces together and keeping them together is hard. It is primarily emotions—especially fear—that make XP hard.
Driving projects by steering a little at a time goes contrary to the car-pointing metaphor prevalent in lots of organizations. A final difficulty, and one that can easily sink an XP project, is that steering is just not acceptable in many company cultures.

The exact limits of XP aren't clear yet. But there are some absolute showstoppers that prevent XP from working—big teams, distrustful customers, technology that doesn't support graceful change.
You shouldn't use XP if you are using a technology with an inherently exponential cost curve.
Another technology barrier to XP is an environment where a long time is needed to gain feedback.
If you have the wrong physical environment, XP can't work.
Finally, there is absolutely no way you can do XP with a baby screaming in the room.

XP can accommodate the common forms of contract, albeit with slight modifications. Fixed price/fixed scope contracts, in particular, become fixed price/fixed date/roughly fixed scope contracts when run with the Planning Game.

To the degree that XP is my baby, XP reflects my fears. I am afraid of:

- Doing work that doesn't matter

- Having projects canceled because I didn't make enough technical progress
- Making business decisions badly
- Having business people make technical decisions badly for me
- Coming to the end of a career of building systems and realizing that I should have spent more time with my kids
- Doing work I'm not proud of

XP also reflects the things I'm not afraid of:

- Coding
- Changing my mind
- Proceeding without knowing everything about the future
- Relying on other people
- Changing the analysis and design of a running system
- Writing tests

Author

Kent Beck (born 1961) is an American software engineer and the creator of extreme programming, a software development methodology that eschews rigid formal specification for a collaborative and iterative design process. Beck was one of the 17 original signatories of the Agile Manifesto, the founding document for agile software development. Extreme and Agile methods are closely associated with Test-Driven Development (TDD), of which Beck is perhaps the leading proponent. Beck pioneered software design patterns, as well as the commercial application of Smalltalk. He wrote the SUnit unit testing framework for Smalltalk, which spawned the xUnit series of frameworks, notably JUnit for Java, which Beck wrote with Erich Gamma. Beck popularized CRC cards with Ward Cunningham, the inventor of the wiki.