Extreme Programming: A Humanistic Discipline of Software Development

Kent Beck

CSLife and First Class Software
P.O. Box 226
Boulder Creek, CA
email: 70761.1216@compuserve.com

Once a farmer and his friend were sitting at a battered kitchen table. The farmer said, "It's time for the mule's lesson." Out they went into the yard. The farmer picked up a thick tree branch lying on the ground. He walked up to the mule and hit it over the head so hard the branch shattered. The mule staggered. His friend's jaw dropped. The friend said,"You can't teach a mule a lesson like that." "That wasn't the lesson," replied the farmer, "that was just to get his attention."

Introduction

Extreme Programming is a discipline of software development. It is intended to guide development of medium sized projects by small teams, getting ordinary programmers to achieve extraordinary goals. It emphasizes productivity, flexibility, informality, teamwork, and limited use of technology outside of programming. This paper presents the values underlying Extreme Programming, its assumptions, and some ways to begin incorporating its ideas into your work.

My plan was not to write this paper for these proceedings. My plan was to write a 25 page manifesto from which you could infuriate, amuse, enlighten, and motivate yourself. Instead, I'm typing this with three day old Joëlle Karen on my chest, trying to decide what all to leave out. If there's one lesson in Extreme Programming, it is that life and software development must balance. So, instead of a carefully reasoned argument leading you gently step by step to the conclusions of Extreme Programming, I'm going to take the time I have to hit you over the head with XPs most bizarre and surprising assumptions. I'll trust you to keep an open mind, and incorporate those practices that make sense to you.

Values

Any discipline has to start with a set of values. What does the discipline hold dear? How will it measure its success? Extreme Programming emphasizes four values:

Communication The bottlenecks in software development all stem from the difficulty in communication between people. XP aims to enhance communication developer-to-developer, developer-to-client, and developer-to-manager.

Simplicity Every bit of complexity on a project adds a little to the required effort, but it adds a lot to risk. XP reduces risk by constantly asking "What is the simplest thing that could possibly work?" Requirements should be simple, programs should be simple, and the discipline itself should be simple. Communication enhances simplicity by making it easier to find unnecessary complexity. Simplicity enhances communication because it is much easier to communicate simple structures.

Testing If a program feature has no automated test, it doesn't work. If it does have an automated test, it might work. XP relies on two levels of testing: developers write their own unit tests in minute-by-minute synchronization with their code, and separate functional testers write tests from the client's perspective. Testing enhances communication by recording the interfaces of the program. Communication enhances testing by improving the odds that a test will discover a defect. Testing enhances simplicity because with tests you can refactor code much more readily. Simplicity enhances testing because there is that much less to test in a simple program

Aggression With the first three values in place, the team can go like hell. Nothing beats experience for dispelling fear. XP developers throw code away when it isn't going well. XP developers spend as much time refactoring as they do coding, even late in development. Communication enhances aggression because two people together are much more likely to try something difficult but valuable than one person alone. Aggression enhances communication because developers are not afraid to learn about new parts of the system. Simplicity enhances aggression because you can much more easily manipulate a simple program. Aggression enhances simplicity because it is often late in development when you discover how the system really should have been structured to be simple, and if you are aggressive you will make it so. Testing enhances aggression because you are much more likely to try something if you can test it afterwards.

Lying beneath all of these values is the recognition that software development is a human process. People want programs. People write programs. People use programs. Computers have a profound effect on all of these people's lives. Acknowledging and honoring humanity in this process is the only way to realize the potential of computers. Treating them like balky bits of machinery will not provide the best results for anyone.

If you believe that humanity is part of software development, then you have to pay attention to how everyone feels. People will be afraid, everybody, whether because my job might disappear, because I haven't learned about Java yet, because I've forgotten how to program, or because I don't like giving so much

power to nerds. You have to address those fears with listening, with victories, with celebration, with punishment where necessary. The good news is that if you pay attention to emotions, you will find them a sensitive barometer of progress, risk, and deviation.

Assumptions

This section hints at how XP is a different way of developing software. I have tried to state the assumptions without much justification (in deference to the shortness of time and this fussing baby). Each assumption also hints at XPs reaction to the assumption. It's been frustrating writing this section, as I want to explain each assumption enough to convince you, and explain the practices that follow from it enough to let you try them. I keep reminding myself of the mule story – I'm just trying to get your attention.

Most developers have no idea of the cost or benefits of their decisions. Therefore, begin each project by making a chart listing five years' income and expenses for the company with the project and without it. Refer to this business justification when "How much is enough?" arises.

Many, perhaps a majority of, projects fail to deliver any value beyond experience. Therefore, do whatever you can to reduce the chance of project failure. Simplify requirements, ignore reuse, eliminate useless documentation, ignore unfamiliar technology.

Planning has limited accuracy. Therefore, only give it limited investment at first. Before you commit to a schedule, implement enough of the tricky bits to be comfortable. Be prepared to revisit planning as you learn throughout the project.

Requirements will change. Therefore, be prepared to change with them. Collect brief stories from the client describing everything the system has to do, and which set of stories would constitute a useful first release. Implement the most valuable features first. Invest no more than necessary in the features you do implement. Create a dialog with the client so you know as soon as possible that they are changing the requirements.

Clients need to see progress. Therefore, break the schedule up into bundles of stories no more than four weeks long. Implement functional tests for each story and try to make sure they pass before the end of their iteration.

Many systems never have a big picture, or the big picture is so vague as to be useless. Therefore, agree on a common metaphor before beginning. Make the first iteration of the system fully functional in some way (print a real check, make a real phone call).

The users don't need everything they ask for. The users don't know what is hard and what is easy. The developers don't know what is important. Therefore, make real users a full-time part of the development team. Let them decide what order to implement features. Let them decide what will be in a release based on estimates from the developers and a fixed amount of available development.

Most features implemented in anticipation of future requirements are never used. Therefore, implement today what you need today. Trust your ability to refactor should your design prove naive.

Most program documentation is never read. Therefore, don't write the parts that won't be read. Instead, make every effort to communicate through code (common metaphor, coding conventions, small procedures, separation of concerns). Write information that doesn't fit well in the code in the form of technical memos.

Technology gets in the way of communication. Therefore, avoid it whenever possible. In particular, CASE tools and project management tools interfere with communication, so don't use them. Write information on index cards, stacked, stapled, color coded, and rubber-banded as necessary.

Face-to-face communication is far more effective than any other kind. Therefore, locate your entire team in the same office and ask everyone to keep roughly the same hours. Program in pairs, two developers at one machine, when writing production code.

Developers need freedom to make changes where they make the most sense. Therefore, integrate and test several times a day. Throw away unintegrated code after a couple of days and start over. Ignore code ownership. Program in pairs. Don't integrate without unit tests.

Changing analysis and design decisions is easy. Therefore, don't make lots of big decisions early in development when you don't have any experience. Make them later when you actually have a chance of being right. Accept refactoring as part of the price of doing business this way, but do refactor a little each day.

Tuning the performance of a simple, well-factored system is easy. Therefore, ignore performance as long as possible. Half way through development, create an automated performance test and graph the results of the test weekly.

First Steps

If you are saying, "I want to throw away everything I am doing now and adopt Extreme Programming on my current project," you have missed the point of XP. (Unless you know you are going to fail and you're desperate. Then a leap might be called for.) Instead, I'd like you to think about little things that you can change that might make a big difference. Here are some suggestions.

If you are a programmer:

Pair Programming The next time you have a couple hours of programming to do, ask a colleague to sit beside you while you do it. Offer to reciprocate for their next programming task. You will be much more successful doing this if you move your computer out of the corner and onto a flat table or even better the protruding end of a table, where you can pass the keyboard and mouse back and forth without moving your chairs.

Unit Testing The next time you have a programming task, write a test before you begin that will only succeed when you are done. If you are using Smalltalk or Java, try a programmatic testing framework like the one at `http://ic.net/~jeffries`. Run your tests after every compile.

Literate Programming The next time you write a tricky bit of code, before you go on to the next task, write it up as a story. Interleave text, pictures, and source code in the document. Rewrite the code if necessary to make it easier to explain. When I wrote my first literate program, I realized just how much more I could say with my code than I was.

If you are a manager:

Stories Write down (or better yet, have a real user write down) the next 20 or 50 things your system has to do. Write each feature on an index card – a name and a couple of sentences are enough. In a meeting with developers and users, have the users sort the cards into three piles: must have, should have, and like to have. Schedule the must have's next.

Iterations Next Monday, have all the developers tell you what 2 or 5 things they are going to do that week and how long (in ideal engineering days) each will take. Next Friday, review what was actually accomplished and how long it took.

Conclusion

You should have been able to conclude by now that XP is not a polished edifice. Indeed, it would be against its own spirit if it ever were. Instead, it is and should always be an evolving fabric of mutually supporting practices. These practices will continue to evolve, perhaps radically, over the next few years.

XP discards much of the received wisdom of the past. Some baby has certainly been thrown out with the bath water. I expect to see some familiar practices return to XP, although in reduced form. For example, diagrammatic notations are entirely ignored by XP currently. There is certainly a role to play for graphics as a design tool.

If you are skeptical about Extreme Programming as presented here, great. It has only been used in its pure form on one project. That makes me suspicious, as I have seen far too many one-project-methodologies. However, if it does no more than make you take a fresh look at what you see as essential to software engineering, I will have partly achieved my purpose. However, none of the ideas in XP is new. The novelty is in putting them all together, and asking that they be taken seriously as a whole discipline of development.

Extreme Programming is named after the currently popular trend of extreme sports. There is a macho, chest-thumping, "I can get closer to dying than you"

aspect to extreme sports that I find repugnant. However, underlying all the extreme sports (and all traditional sports at their highest level), is a marriage of courage and skill that I find an apt metaphor to strive for in software development.

The twin waves of business and technology change are heading for the beach. You can get out your shovels and your buckets and dig like mad, and make bigger buckets and bigger shovels when that doesn't work, or you can get on your surf board and ride.

Acknowledgments

Short iterations/lots of tests came from Jon Hopkins. Many of the scheduling ideas came from Ward Cunningham's Episodes pattern language. Ward was also the person who first began preaching pair programming. Baptism under fire was provided by the brave folks in C3: Ron Jeffries, Bob Coe, Chet Hendrickson, Rich Garzaniti, Margaret Fronczak, Denny Gore, Ralph Beattie, Don Wells, Marie DeArment, Brian Hacker, and Matt Saigeon. Stories are Ivar Jacobson's use cases in warm, fuzzy clothes. Courage to talk about this as more than just a better way to hack came from Alistair Cockburn. Martin Fowler kept reassuring me that I did analysis and design, I just did it weird.