Introducing eXtreme Programming

A new discipline of software development

"If you knew that your customer could afford only one day of software development, what would you do different? …

especially if, by doing something different, he might be able to afford yet another day of software development"

What is Extreme Programming?

- A discipline of program development.
  It relies on:
  - automated tests
  - oral communication
  - a gradual design process
  - an incremental planning approach
  - close collaboration of ordinary programmers
  - use of "instinctive" practices

its reliance on automated tests written by developers and customers to monitor the progress of development and allow the system to evolve

its reliance on oral communication and source code instead of written documentation,

its reliance on a gradual design process that lasts as long as the system lasts,

its incremental planning approach that quickly comes up with an overall plan which is expected to evolve through the life of the project,

its reliance on the close collaboration of programmers with ordinary skills,

its reliance on practices that work with the short-term instincts of developers

I say that this is the short answer, because I don't like the word „methodology". I prefer to call XP a discipline of software development. It is a discipline because there are certain things that you have to do to be doing XP. You don't get to choose whether or not you will write tests- if you don't, you aren't extreme, end of discussion.

The Basic Problems - The XP Answers

- Schedule slips - short delivery cycles
- Project cancelled - the smallest release that makes the most business sense
- System goes sour - comprehensive suite of tests
- Defect rate -
  - developers write tests method-by-method and
  - customers write tests program feature-by-program feature.
- Business misunderstood - customer is part of the team
- Business changes - shortens the delivery cycle
- Staff Fluctuation - developers accept responsibility for estimating and completing their own work. There is less chance for a developer to get frustrated by management incompetence.

Schedule slips- short delivery cycles, scope of any slip is limited. 2-4 week customer-visible iterations give finer grained feedback about progress.

Project cancelled - choose the smallest release that makes the most business sense, less to go wrong before going into production, software value is greatest.

System goes sour- creates and maintains a comprehensive suite of tests to ensure quality baseline. continual evolution of design so the cost of maintenance never rises sharply.

Defect rate- XP tests from the perspective of both developers writing tests method-by-method and customers writing tests program feature-by-program feature.

Business misunderstood- customer is an integral part of the development team. The specification of the project is continuously refined as development continues,

Business changes- shortens the delivery cycle, so there is less change during the development of a single release. During a release, the customer is welcome to substitute new functionality for functionality not yet developed.

Turnover- XP asks developers to accept responsibility for estimating and completing their own work, and honors those estimates. The rules are clear. So, there is less chance for a developer to get frustrated by management incompetence. XP also encourages human contact among the team, reducing the loneliness that is often at the heart of job dissatisfaction. Finally, XP incorporates an explicit model of turnover. New team members are encouraged to gradually accept more and more responsibility, and are assisted along the way by each other and by more senior developers.

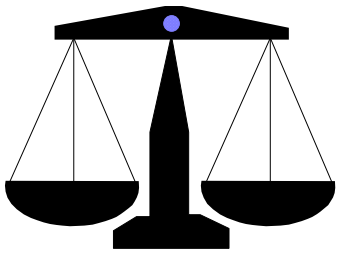
How to do it

Extreme Programming	Our Rules
• Focus: simplicity	• Do the simplest thing that could possibly work
• Build for NOW	• You ain't gonna need it (YAGNI)
• Customers specify, Developers estimate	• Planning Game, Iteration Planning
• Self management	• Release Plan, Project Values

Responsibilities

Customer

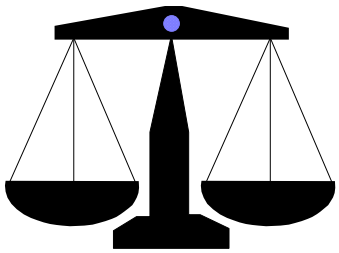
- Need
- Stories
- Resources
- Priorities
- Acceptance



Developer

- Time estimates
- Design
- Code
- Quality

What they need to know



Customer

- How long
- What's done
- How good

Developer

- What to do
- When to do it
- When done

The Economics of XP

The business story

How to increase project value

- Push investment out so you don't have to pay so much interest and you are less likely to have to pay
- Pull revenue in so that you get more interest and you are more like to receive the money
- Increase the chances of survival

9

There are three ways you can increase the overall value of a project analyzed with this model:

Push investment out so you don't have to pay so much interest and you are less likely to have to pay

Pull revenue in so that you get more interest and you are more like to receive the money

Increase the chances of survival

Sometimes you can combine the strategies. For example, you can split a project into phases, which simultaneously pushes some of the investment into the future, pulls some of the payback closer, and increases the chance of the project surviving.

This model is by no means complete. The biggest cost of the risk of software development is not the money that you may or may not make. Often, the cost of risk is dominated by the cost of the lost opportunities. You are sure you're new system will be in production in a year, so you stop developing the old system. By the time it is obvious that the new system isn't going to ship, the old system is so far behind the competition that you have lost a significant share of the market.

Economic Strategy of XP

- You would like to invest money gradually, over time, rather than all at once.
- You would like to begin earning a payback from the system sooner rather than later.
- Above all, you should do anything possible to reduce the risk that the development will fail, since improving the mortality curve has the greatest effect on the value of the project.

10

However, the model makes a few pieces of strategy clear:

You would like to invest money gradually, over time, rather than all at once.

You would like to begin earning a payback from the system sooner rather than later.

Above all, you should do anything possible to reduce the risk that the development will fail, since improving the mortality curve has the greatest effect on the value of the project.

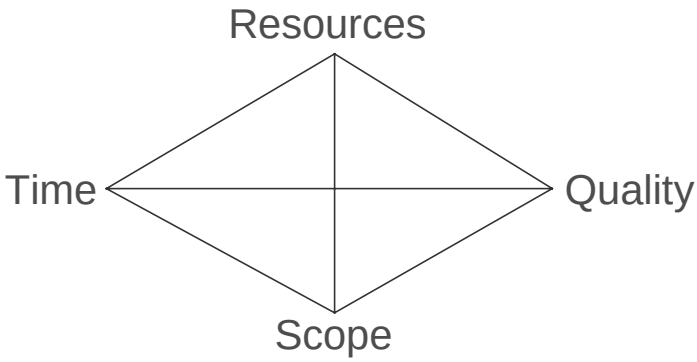
At the limit, designing a software process with this model would lead to a project that went into production at the end of its first day. Each day you would invest a little and get a little more in return. Each day you would review the direction of the project, to be sure that you were always creating the most possible value.

You wouldn't develop blindly, though, thinking only of today. You would develop with an eye towards balancing the economics of today with the economics of the indefinite future.

Every day you would examine the investment you were about to make compared with the returns you were to get. The day you made more money by stopping development, you would stop.

The real world is far more complex. No top manager is willing to review a software project every day.

Four Variables



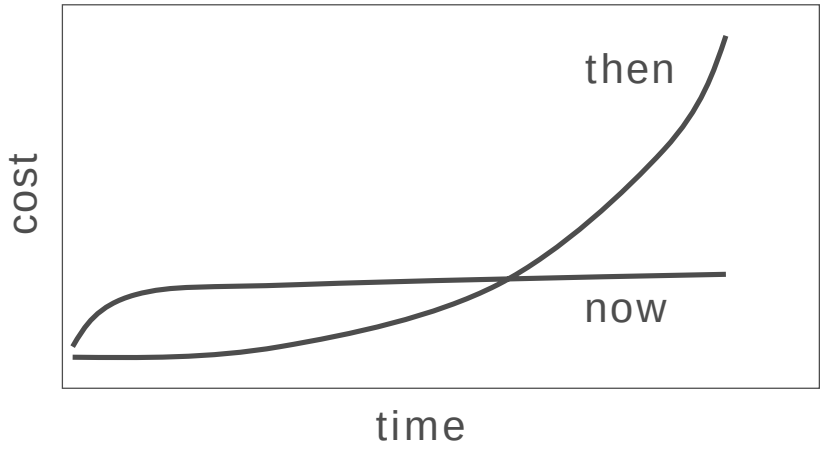
11

The way the software development game is played is that external forces (customers, managers) get to pick the values of three of the variables. The development team gets to pick the resultant value of the fourth variable.

Some managers and customers believe they can pick the value of all four variables. „You are GOING to get all these requirements done by the first of next month with exactly this team. And quality is job one here, so it will be up to our usual standards.“ When this happens, quality always goes out the window (this is generally up to the usual standards, though), since nobody does good work under too much stress. Also likely to go out of control is time. You get crappy software late.

The solution is to make the four variables visible. If everyone, developers, customers, and managers, can see all four variables, they can consciously choose which variables to control. If they don't like the result implied for the fourth variable, they can change the inputs, or they can pick a different three variables to control.

Cost of Change



Cost of Change

$1	$10	$100	$1000	$10000	$100000
Requirements	Analysis	Design	Implementation	Testing	Production

This curve is based on experimental evidence -
of 30 years ago!

$1	$3	$5	$30	$30
Day 1	Week 1	Month 1	Year 1	Decade 1

This is one of the basic premises of XP - if the cost of change rose slowly over time, you would act completely differently than if it rose exponentially

Change is Easy

- automation simplifies tasks
- modularity limits impact
- tests detect mistakes
- change enables further change
- practice makes perfect

Basic Principles

- Coding -- to have something
- Testing -- to know when done
- Listening -- to have things of value
- Refactoring -- to continue forever

"Listening, Testing, Coding, Refactoring.
That's all there is to software.
Anyone who tells you different is selling something."
- Kent Beck

15

What are the things that absolutely matter?

You code because if you don’t code, if at the end of the day, if the program doesn't run and make money for your client, you haven’t done anything.

You test because if you don’t test, you don’t know when you are done coding. If you're smart, you'll write them first so you'll know the instant you're done. Otherwise, you're stuck thinking you maybe might be done, but knowing you're probably not, but you're not sure how close you are.

You listen because if you don’t listen you don’t know what to code or what to test. You don't know the numbers yourself, so you have to get good at listening to clients - users, managers, and business people.

You have to take what your program tells you about how it wants to be structured and feed it back into the program. Otherwise, you'll sink under the weight of your own guesses.

The Four Basic Values

Communication, Simplicity, Feedback and Courage

Communication comes from:

- User Stories
- Release Plannng
- Iteration Planning
- CRC Card design
- Pair Programming
- Continuous Integration
- Listening to what the objects tell you

Communication leads to:

- Higher quality software
- Better business understanding
- Risk reduction through early recognition of problems
- Team building
- Shared knowledge

Simplicity comes from:

- Do the simplest thing that could possibly work
- Refactor mercilessly
- You ain't gonna need it
- Spike solution
- Supported by: Worst things first

Simplicity leads to:

- Better understandable code
- Easy modification
- Confidence in the system

Feedback comes from:

- Continuous Integration
- Relentless Testing
  - Unit Tests
  - Functional Tests
- Customer on site
- Iteration Planning
- Pair Programming

Feedback leads to:

- Early problem recognition
- Better quality software
- Self-confident programmers

Courage comes from:

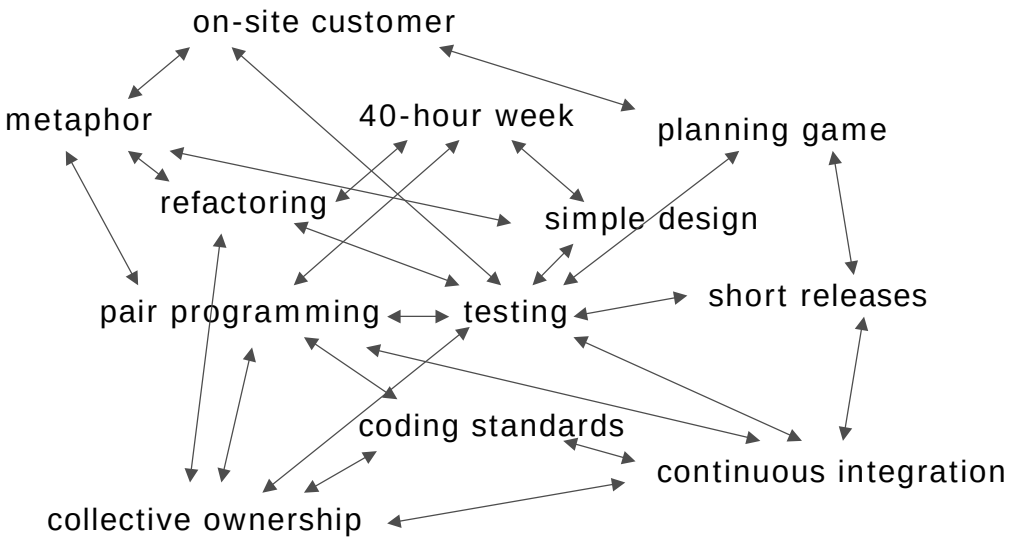
- Continuous Integration
- Relentless Testing
- Pair Programming
- Success

Courage leads to:

- Refactoring Mercilessly
- Collective Code Ownership
- Honest time estimation
- Experimental implementation (Spike solution)
- Honest communication

The Practices

Mutual Support



Practice: Planning Game

- scope of next release
- business sets priorities
- development provides estimates
- find shortest path to most value

Practice: Small Releases

- get into production quickly
- release new versions frequently
- everything is maintenance

Practice: Guiding Metaphor

- smallest architecture
- names elements

Practice: Simple Design

- passes tests
- avoids duplication
- shows intentions
- has fewest parts

Practice: Unit Tests First

- declaration of goal
- defends the simple
- stands for programmer

Practice: Refactoring

- change to improve expression
- motivated only by need
- relentless, small steps

Practice: Pair-Programming

- temporary associations
- assemble experience
- bolster courage
- spread wisdom

Practice: Collective Code Ownership

- universal obligation to add value
- (also must understand “value”)
- more responsibility than no owner
- more ability than individual owner

Practice: Continuous Integration

- code stays ready to integrate
- integrate as attention wanes
- synchronize at single station
- unit tests at 100% good

Practice: 40-hour Week

- not enough to just “try”
- overtime conceals larger problems
- find performance maximum
- don’t exceed it two weeks in a row

Practice: On-Site Customer

- intimate reference
- able and likely to use system
- highly valued people
- but less so than good system

Practice: Coding Standards

- code reveals authors' intention
- cast as solutions to problems
- spread by pairing
- accepted by professionals

Why is XP hard?

- It's hard to do simple things
- It's hard to admit you don't know
- It's hard to collaborate
- It's hard to break down emotional walls
- It's hard to accept responsibility for your development process

When you shouldn't try XP

- Culture
- Big specification
- Paper-driven programming
- Long hours expected
- Smart and clever developers
- Size matters
- Exponential cost curve
- Long feedback cycle
- Wrong physical environment

Culture
Business culture. Any power culture is going to have problems with a team that insists on steering

Big specification
If a customer or manager insists on a big spec (and analysis and design) before programming, there's going to be friction. You must essentially ask them to trade something that gives them control for a dialog that requires continuous involvement.

Paper-driven programming
If management or customers demand piles of documentation, it becomes a task. After a while, the card always seems to disappear.

Long hours expected
Proves your "commitment to the company". XP is strenuous; you can't do it tired. If the production rate of a good team at top speed isn't enough for your company, XP can't help.

Smart and clever developers
Smart people have a very difficult time finding the simplest thing that could possibly work. They can't trade the "guess right" game for close communication, sharing, and continuous evolution.

You don't HAVE to do XP

- Don't force it
  - If the team can't agree on adopting new practices
  - If you can't find a coach
- The time may not be ripe. The team may not be ripe. It may never be ripe.
- You don't HAVE to do XP.

If the team can't agree on adopting new practices, or you can't find a coach, forget it. The time may not be ripe. The team may not be ripe. It may never be ripe. You don't HAVE to do XP.

First Steps

- Find a coach
- Play to win
- Rapid feedback
- Concrete experiments
- Assume simplicity
- Open, honest communication
- The simplest thing that could possibly work
- EMBRACE CHANGE

Contact

For further information please contact

Daedalos Consulting GmbH
Ruhrtal 5
D-58456 Witten
Germany

Tel.: +49 (0)2302 979-0
Fax: +49 (0)2302 979-199
E-Mail: info@daedalos.com

Daedalos Consulting AG
Seestrasse 510
8038 Zürich
Switzerland

Tel.: +41 (1) 481 07 20
Fax: +41 (1) 481 07 24
E-Mail: info@daedalos.com