Extreme Programming: So What?

This talk by Ed Gehringer based on notes by
Roy W. Miller
RoleModel Software, Inc.

Why Extreme Programming?

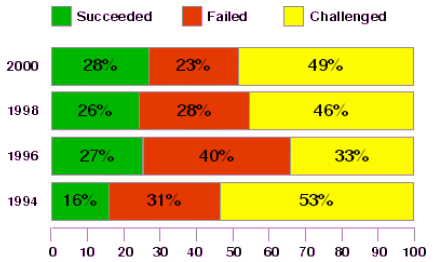
- Be more valuable than your peers.
- Be more productive.
- Make you happier.

The Real Project Lifecycle

- Dream
- Plan
- Capture requirements
- Design a lot, code a little, test if there's time
- Limp to the finish

Create a comprehensive plan, stick to it at all costs, kill change, hope you survive

The Results



Data from 2000 CHAOS Report, Standish Group

- Junk
- Late
- For a lot of money

The software you wanted at the *beginning*, not the *end*

The Source: Taylorism

- Frederick Winslow Taylor, *Principles of Scientific Management* (1911)
- Accepted wisdom by 1950s
- Software began in 1950s
- Software "production" ~= industrial production
- *Exercise:* Find an interesting fact about Taylorism. Submit here.

"[I]n each…trade there is always one method and one implement which is quicker and better than any of the rest. And this **one best method** and best implement can only be discovered or developed through a scientific study and analysis of all of the methods and implements in use, together with accurate, minute, motion and time study. "

Making software is like a factory – an efficiency optimization problem

Software Is Different

Traditional view…

Software like industrial production

- Problem always the same
- Solution always the same
- Optimize process
- Change is disruptive
- Increase predictability

Reality…

Software like predicting the weather

- Problem always different
- Solution always different
- Can't optimize
- Change is constant
- Can't predict accurately

Software is emergent

Growing Software

Need a solution that…

- Allows us not to know
- Allows us to explore
- Gives us feedback to direct us
- Creates the right conditions, lets software emerge
- Lets us produce the right software at the END

XP creates the right conditions for emergent software

XP In a Nutshell

- 4 core values: Simplicity, Communication, Feedback, Courage
- 19 practices
- 1 team
- 3 roles: Customer, Manager, Programmer

What is the simplest thing we can do and still make great software?

The Practices

Joint
- Common Vocabulary
- Iterations
- Open Workspace
- Retrospectives

Development
- Test-First Development
- Pair Programming
- Refactoring
- Collective Ownership
- Continuous Integration
- Just-In-Time Design

Customer
- Storytelling
- Release Planning
- Acceptance Tests
- Frequent Releases

Management
- Accepted Responsibility
- Air Cover
- Quarterly Review
- Mirror
- Sustainable Pace

XP is about more than programming

Joint Practices

Common Vocabulary	Formerly "metaphor" – shared understanding
Iterations	Steering – frequent, regular checkpoints so we can get lots of concrete feedback
Open Workspace	Easy to communicate and learn
Retrospectives	Being "Reflective Practitioners" (Donald Schon), learn as we go

Exercise: Look up one of these practices (your row number mod 4), and find an interesting fact about it. Submit here.

Create an environment where "one team" can exist and thrive

Customer Practices

Storytelling	Describe each system feature in a small chunk that fits in an iteration
Release Planning	Tell programmers which features come first
Customer Tests	Also “acceptance tests” or “functional tests” – tell programmers when they’re done
Frequent Releases	Get software to users so the team can get feedback to steer with

Exercise: Look up one of these practices (your row number mod 4), and find an interesting fact about it. Submit here.

“Drive” the entire process

Management Practices

Accepted Responsibility	Say what needs to be done, let the team decide who does it and how
Quarterly Review	Make sure the team knows what it needs to; make sure management knows what it needs to
Air Cover	Soften up the defenses to make room for the infantry
Sustainable Pace	Help people avoid burnout
Mirror	Point out problems, suggest, advise, encourage

Exercise: Look up one of these practices (your row number mod 5), and find an interesting fact about it. Submit here.

Educate, facilitate, stay out of the way

Development Practices

Test-First Development	No code without a failing programmer test
Pair Programming	All code gets two pairs of eyes
Refactoring	Remove "smells"
Collective Ownership	Everyone owns all of the code
Continuous Integration	Integrate many times each day
Just-In-Time Design	Keep design simple

Test-First Development

- Write tests before you write code
  - http://www.junit.org
  - http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html
- Write just enough code to get each test to pass
- All about confidence
- Programmer tests tell you when the code "works"
- Programmer tests must pass 100% all the time
- Test anything you need to be sure it works

Complete test coverage, simplest code that could possibly work, clear intent

Pair Programming

- 2 developers, 1 computer, solving problems together
- One person “drives,” the other “navigates”
- **Not** Driver/Passenger
- **Not** Pair Watching
- Pairs should rotate
- Love your pair

Continuous code review, more efficient learning, lower project risk

Refactoring

- Changing the design of existing code without changing function
  - http://www.refactoring.com
- All about speed
- Refactor when code “smells”
  - Methods, classes that are too long.
  - Duplicate code (or “almost” duplicate code).
  - Switch statements (instead of polymorphism).
  - “Struct” classes—getters & setters but little else.
- Refactor before adding a feature, and after

Keep code simple, build learning in

Collective Ownership

- Any developer can change any code anytime
- Programmer tests and customer tests tell you if you broke something
- You break it, you fix it

Exercise: Is this a good idea? Look up points pro and con. Submit here.

Convert "my code" to "our code" to lower risk

Continuous Integration

- Integrate changes multiple times each day
- One failing Programmer Test = no integration
- Daily is not enough
- No "Big Bang"

Exercise: Is this a good idea? Look up points pro and con. Submit here.

Maintain speed and spread risk by integrating many times per day

Just-In-Time Design

- Only design for what you're building
- Always keep the design as simple as possible
- Simplicity allows for change
- Change is constant

Exercise: Is this a good idea? Look up points pro and con. Submit here.

Simple design: passes all tests, has no duplication, expresses intent, has least amount of code

All Or Nothing?

- Some practices can stand alone – Refactoring, Test-First Development, Pair Programming
- All is better, some often better than none
- All doesn't mean starting all at once

The closer you get to all, the better off you are

So What?

XP doesn't matter – results do

- XP reflects the true nature of the problem (complex)
- XP is change-tolerant
- XP is realistic
- XP has the potential to facilitate organizational change

XP increases likelihood for success

Resources

http://www.xprogramming.com (Ron Jeffries)

http://www.junit.org (JUnit testing framework)

Addison Wesley XP Series:

Extreme Programming Explained: Embrace Change, Beck
Extreme Programming Installed, Jeffries, Hendrickson, Anderson
Planning Extreme Programming, Fowler and Beck
Extreme Programming Applied: Playing to Win, Auer and Miller

Refactoring, Fowler

IBM developerWorks XP Column, starting in August (http://www.ibm.com/developerWorks)

Growing Software (working title), Addison Wesley, 2003

http://www.roywmiller.com