An Introduction to Extreme Programming

Ken Auer

kauer@rolemodelsoft.com
http://www.rolemodelsoft.com
RoleModel Software, Inc.
5004 Rossmore Dr.
Fuquay-Varina, NC 27526
919-557-6352
To change Hardware
- Soldering Iron
- Screwdrivers
- Change PCB

To change Software
- Just change some symbols
The Reality? of Software

- Changing Software is Hard
  - Dependencies
The Promise of Specifications

Examples from Traditional Engineering
  ➢ Do the Drawings first
  ➢ Design work is done on the drawings
  ➢ Construction is a separate act (at least in theory)

In Software
  ➢ Capture Requirements up front
  ➢ Do design up front
  ➢ Hand over to coders
Almost all method(ologie)s focus on this assumption no matter what they say about the evils of waterfalls!
The Uncomfortable Reality

- The Customers change requirements
  - Customers believe software is easy to change
  - Real requirements hard to nail down
  - They’re learning, conditions change

- Hard to get costs of features
  - Materials keep changing
  - Productivity of people/teams vary widely
  - Later development tasks hard to estimate

- Developers learn as they program

- Too much flexibility costs
  - Cannot design for unanticipated changes
  - Often anticipate wrong changes
  - ...

Copyright © 1999, RoleModel Software, Inc.
Design Decisions

- **In the moment**
  - Consider current needs
  - Patch code when new needs appear... is this really design???

- **Up front design**
  - Consider current needs and possible future needs
  - Design to minimize change with future needs
  - Patch code if unforeseen need appears

- **Evolutionary design**
  - Consider current needs and possible future needs
  - Trade off cost of current flexibility versus cost of later refactoring
  - Refactor as changes appear
Object-orientation helps manage the complexity of software and reduce dependencies...

Why don’t methods reflect this?

Why don’t methods exploit this?

Many have tried but failed because they threw out the baby (discipline) when they threw out the bathwaterfall...
Another Approach

Cost of Making Changes

Reduce cost of later change

Time

- The flatter this is, the less...
  - Benefit of up-front design
  - Cost of up-front design
  - Cost of maintaining design artifacts
“Recognizing that the cost of changing a program no longer grows exponentially over time, XP relies on the complex interactions between simple development practices to reduce project risk, improve responsiveness to business and technical learning, and make programming more fun”

- Kent Beck
Playing to Win

- Don’t just “survive”
- Tackle problems aggressively
- Tackle only the problems that you KNOW exist in the current context
- Communicate to expose and reduce risk
- Be confident in what you’ve done and are doing based on feedback
The Four Values

- Simplicity
- Communication
- Feedback
- Aggressiveness
Simplicity

❑ What is the simplest thing that will possibly work?
❑ No code duplication
❑ Don’t look too far ahead...
  ➢ Simple things are easier to change
  ➢ You may not need it
Communication

- People talk to people
- People work with people
- People are honest when they don’t know
- People share what they’ve learned!
- Code communicates with people
- No documentation for documentation’s sake
Feedback

- **Measure things that matter**
- **Look at the measurements**
- **Share the measurements**
- **Believe the measurements tell you something**
- **Change what you can, accept what you can’t**
Aggressiveness

- You’ve got to believe that you can do it
- Jump into the problem, don’t be frozen by all the things that can go wrong
- Believe that when things do go wrong, you can fix them
XP Iterations

- **Limited Up Front Design**
  - Detailed work in iterations
  - Up front work for estimates and risk assessment

- **Planning Game**
  - Customers develop stories (use cases)
  - Development estimates stories
  - Joint scheduling based on customer value and development risk

- **Disciplined development**
  - Write tests first
  - Only focus on story in hand
  - Short development episodes in pairs
  - Continuous Integration
  - Always working, always clean
Development Practices

A Day in the Life...
A Day In The Life

- Stand up meeting
- Load latest stuff
- Identify tasks
- Pair up
- Test first, code second
  - What exists?
  - Does it support what we need to do?
  - Gotta test for that?
  - Do the tests work?
  - Is the code clean (enough)?
- If it’s working, integrate it
  - Do the tests work?
- Next task?
XP® Development Practices

A Little More Detail
Split Business & Technical

- Customer
  - chooses scope
  - sets priorities

- Developers
  - estimate time required to implement requirements
  - schedule within a release...
Small Releases

- **Customer**
  - chooses smallest, most valuable set of requirements to put into production next

- **Developers**
  - keep design simple to get requirements met as fast as possible
Metaphor

- Everyone tells the story of the system consistent with a central metaphor
- Validate the metaphor quickly with a concrete design
- Often 5-20 CRC level concepts
Simple Design

- At all times the system has
  - simplest design that runs all test cases
  - no code duplication
  - no unnecessary classes or methods

- Constantly evolved through refactoring
Having Tests

- **Developers**
  - write and satisfy unit tests for all development
  - write the tests first

- **Customers/Testers**
  - write functional tests for each requirement

- Ensure all tests can be run with one command

- Test often... every change to a line of code??

- Run all tests with every build

- Tests should be about 1/3 of production code
Having Tests

- Take the confidence of the moment, and preserve it
- When did it break?
- Allows
  - refactoring
  - collective code ownership
  - feedback
  - trust...
Pair Programming

- All Production Code...
  - written with two people sitting at one machine
  - pass the keyboard and mouse back and forth

- Refactoring code wherever it needs clean up

- Write tests

- Everyone knows a little about the whole system
Refactoring

- Pair brings code back to simplicity the instant they see it drifting away
- Short episodes...
Continuous Integration

- System is built and successfully tested several times a day
  - each time a pair gets the test cases for a simple development task running
  - integration machine
Collective Code Ownership

- Code is fixed wherever it is found to be deficient
- Tests verify you haven’t broken anything
  - New tests verify it doesn’t happen again
Iterations

- During Iteration Planning
  - Cards can be shifted
  - Time cannot

- During Iteration
  - Clarification as we learn
  - That’s another iteration
Final Thoughts

- Undisciplined coding leads to a steep cost of change curve
- BDUF doesn’t change the curve
  - It attempts to reduce number of times you are bitten by the curve
  - It adds up front cost that may never be recovered
- XP focuses on pushing the "cost of change" curve down
- Any of the XP practices might help your project
- Adopting all of the practices will allow you to attain a high level of productivity and produce high quality product at a sustained rate over an extended period of time
To Learn More...

- http://www.armaties.com
- Books from AW-L:
  - *Extreme Programming Explained: Embrace Change*
    Kent Beck
  - *Refactoring: Improving the Design of Existing Code*
    Martin Fowler
  - *Extreme Programming Applied: Playing to Win*
    (not yet available) Beck, Auer, Cunningham, Fowler, Jeffries