eXtreme Programming

Overview

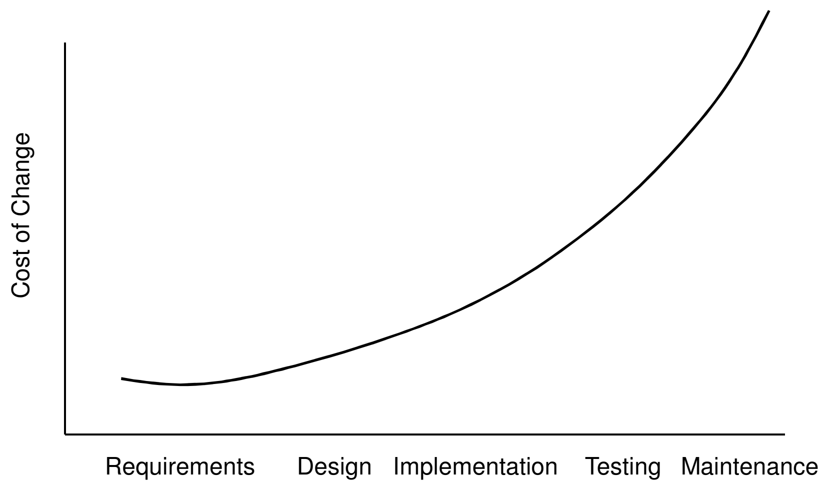
What’s wrong with software today?

- Software development is risky and difficult to manage
- Customers are often dissatisfied with the development process
- Programmers are also dissatisfied

One Alternative: Agile Development Methodologies

- XP = eXtreme Programming
  - It does not encourage blind hacking. It is a systematic methodology.
  - It predates Windows “XP”.
- Developed by Kent Beck
  - XP is “a light-weight methodology for small to medium-sized teams developing software in the face of vague or rapidly changing requirements.”
- Alternative to “heavy-weight” software development models
  (which tend to avoid change and customers)
  - "Extreme Programming turns the conventional software process sideways. Rather than planning, analyzing, and designing for the far-flung future, XP programmers do all of these activities a little at a time throughout development.”
    -- IEEE Computer , October 1999

Traditional Processes are ‘Heavy’



Boehm’s Curve

- To accomplish this:
  - We need lots of up front planning, resulting in “heavy” methodologies
  - Every bug caught early saves money, since models are easier to modify than code
  - Large investments are made in up front analysis and design models, because the of the cost of late error discovery
  - This leads to a waterfall mentality with BDUF (Big Design Up Front)
- Proponents of XP argue that logic is based on development in the 1970’s and 1980’s

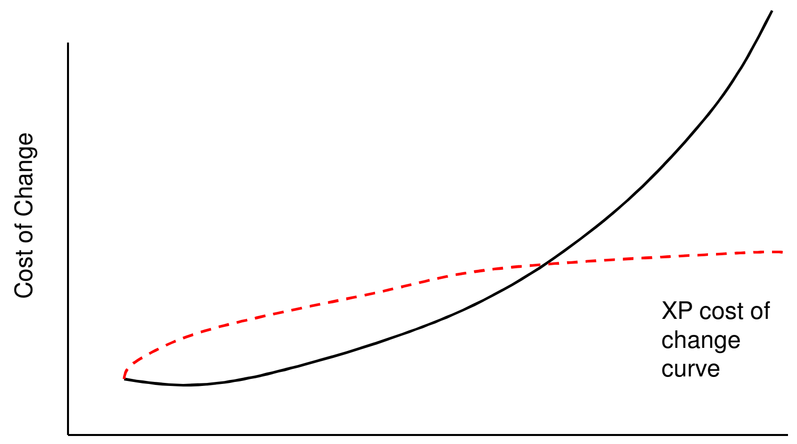
What’s Changed?

- Computing power has increased astronomically
- New tools have dramatically reduced the compile/test cycle
- Used properly, OO languages make software much easier to change
- The cost curve is significantly flattened, i.e. costs don’t increase dramatically with time
- Up front modeling becomes a liability – some speculative work will certainly be wrong, especially in a business environment

Why XP Helps

- Extreme Programming is a “light” process that creates and then exploits a flattened cost curve
- XP is people-oriented rather than process oriented, explicitly trying to work with human nature rather than against it
- XP Practices flatten the cost of change curve.

XP Cost of Change Curve



Embrace change

- In traditional software life cycle models, the cost of changing a program rises exponentially over time
- A key assumption of XP is that the cost of changing a program can be hold mostly constant over time
- Hence XP is a lightweight (agile) process:
  - Instead of lots of documentation nailing down what customer wants up front, XP emphasizes plenty of feedback
  - Embrace change: iterate often, design and redesign, code and test frequently, keep the customer involved
  - Deliver software to the customer in short (2 week) iterations
  - Eliminate defects early, thus reducing costs

Why does XP Help?

- “Software development is too hard to spend time on things that don't matter. So, what really matters? Listening, Testing, Coding, and Designing.” - Kent Beck, “father” of Extreme Programming
- Promotes incremental development with minimal up-front design
- Results in a “pay as you go” process, rather than a high up-front investment
- Delivers highest business value first
- Provides the option to cut and run through frequent releases that are thoroughly tested

More on XP

- XP tends to use small teams, thus reducing
- communication costs.
- XP puts Customers and Programmers in one place.
- XP prefers index cards to expensive round-trip UML diagramming environments
- XP's practices work together in synergy, to get a team moving as quickly as possible to deliver value the customer wants

Successes in industry

- Chrysler Comprehensive Compensation system
  - After finding significant, initial development problems, Beck and Jeffries restarted this development using XP principles
  - The payroll system pays some 10,000 monthly-paid employees and has 2,000 classes and 30,000 methods, went into production almost on schedule, and is still operational today (Anderson 1998)
- Ford Motor Company VCAPS system
  - Spent four unsuccessful years trying to build the Vehicle Cost and Profit System using traditional waterfall methodology
  - XP developers successfully implemented that system in less than a year using Extreme Programming (Beck 2000).

XP Process

- Planning
  - User stories are written
  - Release planning creates the schedule.
  - Make frequent small releases.
  - The Project Velocity is measured.
  - The project is divided into iterations.
  - Iteration planning starts each iteration.
  - Move people around.
  - A stand-up meeting starts each day.

XP Process

- Designing
  - Simplicity.
  - Choose a system metaphor.
  - Use index cards for design sessions.
  - Create spike solutions to reduce risk.
  - No functionality is added early.
  - Refactor whenever and wherever possible.

XP Process

- **Coding**
  - The customer is always available.
  - Code must be written to agreed standards.
  - Code the unit test first.
  - All production code is pair programmed.
  - Only one pair integrates code at a time.
  - Integrate often.
  - Use collective code ownership.
  - Leave optimization until the end.
  - No overtime.

XP Process

- Testing
  - All code must have unit tests.
  - All code must pass all unit tests before it can be released.
  - When a bug is found tests are created.
  - Acceptance tests are run often and the score is published.

Four Core Values of XP

- Communication
- Simplicity
- Feedback
- Courage

Communication

- What does lack of communication do to projects?
- XP emphasizes value of communication in many of its practices:
  - On-site customer, user stories, pair programming, collective ownership (popular with open source developers), daily standup meetings, etc.
- XP employs a *coach* whose job is noticing when people aren't communicating and reintroduce them

Simplicity

- "Do the simplest thing that could possibly work" (DTSTTCPW) principle
  - Elsewhere known as KISS
- A coach may say DTSTTCPW when he sees an XP developer doing something needlessly complicated
- YAGNI principle ("You ain't gonna need it")
- How do simplicity and communication support each other?

Feedback

- Feedback at different time scales
- Unit tests tell programmers status of the system
- When customers write new *user stories,* programmers estimate time required to deliver changes
- Programmers produce new releases every 2-3 weeks for customers to review
- How does valuing feedback turn the waterfall model upside down?

Courage

- The courage to communicate and accept feedback
- The courage to throw code away (prototypes)
- The courage to refactor the architecture of a system
- Do you have what it takes?

ALL OF MY BAD DECISIONS ARE CATCHING UP TO ME. COULD WE DO ANOTHER REORG TO COVER MY TRACKS?
YEAH, I'VE GOT SOME BODIES TO BURY, TOO.

Twelve XP Practices

- The Planning Game
- Small Releases
- Metaphor
- Simple Design
- Test-driven development
- Refactoring
- Pair Programming
- Collective Ownership
- Continuous Integration
- 40-Hours a Week
- On-Site Customer
- Coding Standards

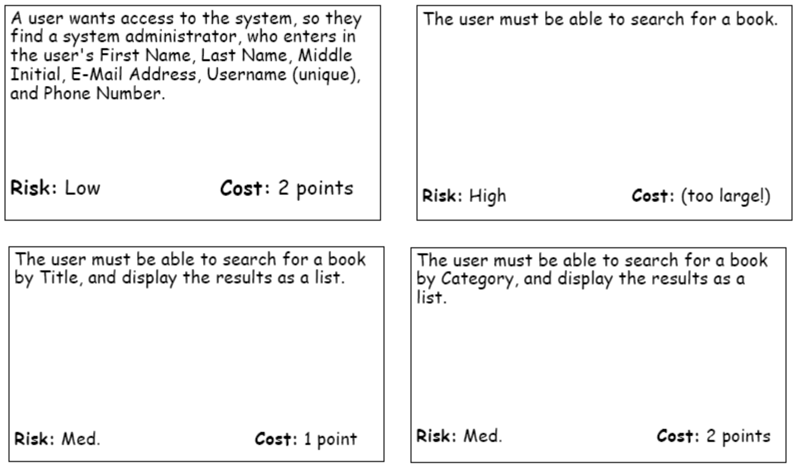
The Planning Game

- Customer comes up with a list of desired features for the system
  - How is this different from the usual requirements gathering?
- Each feature is written out as a **user story**
  - Describes in broad strokes what the feature requires
  - Typically written in 2-3 sentences on index cards
- Developers estimate how much effort each story will take, and how much effort the team can produce in a given time interval (iteration)

User Stories

- Drive the creation of the acceptance tests:
  - Must be one or more tests to verify that a story has been properly implemented
- Different than Requirements:
  - Should only provide enough detail to make a reasonably low risk estimate of how long the story will take to implement.
- Different than Use Cases:
  - Written by the Customer, not the Programmers, using the Customer’s terminology
  - More “friendly” than formal Use Cases

User Story Examples


A user wants access to the system, so they find a system administrator, who enters in the user's First Name, Last Name, Middle Initial, E-Mail Address, Username (unique), and Phone Number.

Risk: Low **Cost:** 2 points

The user must be able to search for a book.

Risk: High **Cost:** (too large!)

The user must be able to search for a book by Title, and display the results as a list.

Risk: Med. **Cost:** 1 point

The user must be able to search for a book by Category, and display the results as a list.

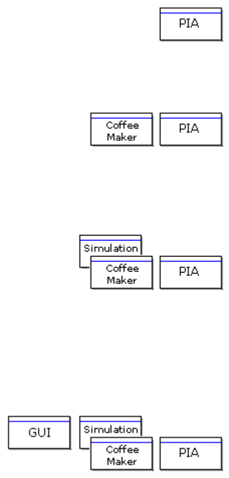
Risk: Med. **Cost:** 2 points

User Stories

- **Project velocity** = how many days can be committed to a project per week
  - Why is this important to know?
- Given developer estimates and project velocity, the customer prioritizes which stories to implement
  - Why let the customer (rather than developer) set the priorities?
- Later we must develop acceptance tests for each story

Design

- No tedious UML
- Use CRC cards
- Web example: http://www.extremeprogramming.org/example/crcsim.html



Small and simple

- Small releases
  - Start with the smallest useful feature set
  - Release early and often, adding a few features each time
  - Releases can be date driven or user story driven
- Simple design
  - Always use the simplest possible design that gets the job done
  - The requirements will change tomorrow, so only do what's needed to meet today's requirements (remember, YAGNI)

Test-driven development

- **Test first:** before adding a feature, write a test for it!
  - If code has no automated test case, it is assumed it does not work
- When the complete test suite passes 100%, the feature is accepted
- Tests come in two basic flavors…
- **Unit Tests** automate testing of functionality as developers write it
  - Each unit test typically tests only a single class, or a small cluster of classes
  - Unit tests typically use a unit testing framework, such as JUnit (xUnit)
  - Experiments show that test-driven development reduces debugging time
  - Increases confidence that new features work, and work with everything
  - If a bug is discovered during development, add a test case to make sure it doesn't come back!

Test-Driven Development

- **Acceptance Tests** (or **Functional Tests**) are specified by the customer to test that the overall system is functioning as specified
  - When all the acceptance tests pass, that user story is considered complete
  - Could be a script of user interface actions and expected results
  - Ideally acceptance tests should be automated, either using a unit testing framework, or a separate acceptance testing framework

Pair programming

- Two programmers work together at one machine
- **Driver** enters code, while **navigator** critiques it
- Periodically switch roles
- Research results:
  - Pair programming increases productivity
  - Higher quality code (15% fewer defects) in about half the time (58%)
  - Williams, L., Kessler, R., Cunningham, W., & Jeffries, R. Strengthening the case for pair programming. *IEEE Software*, 17(3), July/August 2000
  - Requires proximity in lab or work environment

Pair programming in CS classes

- Experiment at NC State
  - CS1— programming in Java
  - Two sections, same instructor, same exams
  - 69 in solo programming section, 44 in paired section
  - Pairs assigned in labs

- Results:
  - 68% of paired students got C or better vs. 45% of solo students
  - Paired students performed much 16-18 points better on first 2 projects
  - No difference on third project (perhaps because lower performing solo students had dropped before the third project)
  - Midterm exam: 65.8 vs. 49.5   Final exam: 74.1 vs. 67.2
  - Course and instructor evaluations were higher for paired students
- Similar results at UC Santa Cruz (86 vs. 67 on programs)

More XP practices

- Refactoring
  - Refactor out any duplicate code generated in a coding session
  - You can do this with confidence that you didn't break anything because you have the tests
- Collective code ownership
  - No single person "owns" a module
  - Any developer can work on any part of the code base at any time
- Continuous integration
  - All changes are integrated into the code base at least daily
  - Tests have to run 100% both before and after integration

More practices

- 40-hour work week
  - Programmers go home on time
  - “fresh and eager every morning, and tired and satisfied every night”
  - In crunch mode, up to one week of overtime is allowed
  - More than that and there’s something wrong with the process
- On-site customer
  - Development team has continuous access to a real live customer, that is, someone who will actually be using the system
- Coding standards
  - Everyone codes to the same standards
  - Ideally, you shouldn't be able to tell by looking at it who on the team has touched a specific piece of code

13th XP practice: Daily standup meeting

- Goal: Identify items to be accomplished for the day and raise issues
  - Everyone attends, including the customer
  - Not a discussion forum
  - Take discussions offline
  - Everyone gets to speak
  - 15 minutes

Kindergarten lessons

- Williams, L. and Kessler, R., "All I Really Need to Know about Pair Programming I Learned In Kindergarten," *Communications of the ACM* (May 2000)
  - *Share everything.* (Collective code ownership)
  - *Play fair.* (Pair programming—navigator must not be passive)
  - *Don't hit people.* (Give and receive feedback. Stay on track.)
  - *Clean up your own mess.* (Unit testing.)
  - *Wash your hands before you eat.* (Wash your hands of skepticism: buy-in is crucial to pair programming.)
  - *Flush.* (Test-driven development, refactoring.)
  - *Take a nap every afternoon.* (40-hour week.)
  - *Be aware of wonder.* (Ego-less programming, metaphor.)