Agile Processes - Extreme Programming

CS169
Lecture 3
(with some materials from Alex Aiken and Kurt Keutzer)

Administrivia

- Today is the deadline for project proposals
  - We will pre-select ~ 20 proposals
  - You will bid (on Monday) on those to select the 10 project topics

Review: The Waterfall Model

Gather Requirements
Specification
Design
Implementation
Integration
Product

When Does Waterfall work?

Cost of a Design Change

- Many organizations have created slow laborious software design processes
  - Then, of course there is a high cost of design change
- Bad reasons not to change the design
  - Diffusing change expensive - silly today
  - GUI change could confuse customer - current GUI may be even more confusing
- Good reasons not to change the design
  - Life critical situations
  - Situations requiring significant retraining

Review: Waterfall drawbacks

- The major risks are:
  - Relies heavily on being able to accurately assess requirements at the start
  - Little feedback from users until very late
    - Unless they understand specification documents
  - Problems in the specification may be found very late
    - Coding or integration
  - Whole process can take a long time before the first working version is seen
Extreme Programming

- Waterfall model inspired by civil engineering
- Civil engineering metaphor is not perfect
  - Software is more organic than concrete
  - Much higher rate of change; you "grow the software" to meet changing requirements
- Extreme Programming (XP) addresses this
  - A version of the iterative model discussed before
  - Recommended reading: "Extreme Programming Explained" by Kent Beck

Software Process Control Variables

- Control variables in a software project
  - Time
  - Quality
  - Scope
- If you fix all three then the hardest to measure (quality) will suffer
- Management fixes two and the development team controls the third
  Scope is the control variable for XP

Extreme Programming (XP)

- XP: like iterative but taken to the extreme

<table>
<thead>
<tr>
<th>Time</th>
<th>Waterfall</th>
<th>Iterative</th>
<th>XP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analyse</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

XP Development Cycle

Short cycle (2 weeks):
1. Meet with client to elicit requirements
   - User stories + acceptance tests
2. Planning game
   - Break stories into tasks, estimate cost
   - Client prioritizes stories to do first
3. Implementation
   - Write programmer tests first
   - Simplest possible design to pass the tests
   - Occasionally refactor the code
4. Evaluate progress and reiterate from step 1

XP Customer

- Expert customer is part of the team
  - (ideally) on site, available constantly
  - XP principles: communication and feedback
  - Make sure we build what the client wants
- Customer involved actively in all stages:
  - Clarifies the requirements
  - Negotiates with the team what to do next
  - Writes and runs acceptance tests
  - Constantly evaluates intermediate versions

User Stories

- Write on index cards
  - meaningful title
  - short (customer-centered) description
- Focus on "what" not the "why" or "how"
- Uses client language
  - Client must be able to test if a story is completed
- No need to have all stories in first iteration
Example User Stories: Accounting Software

1. Create account
   "I can create named accounts"
2. List accounts
   "I can get a list of all accounts"
   - Question: How is the list ordered?
   "I can get an alphabetical list of all accounts"
3. Query account balance
   - "I can query account balance"
4. Delete account
   "I can delete a named account"
   - Question: Even if the balance is not zero?
   "I can delete a named account if the balance is not zero"

Customer Tests

- Client must describe how the user stories will be tested
  - With concrete data examples,
  - Associated with (one or more) user stories
- Concrete expressions of user stories

Example: Accounting Customer Tests

- Tests are associated with (one or more) stories
1. If I create an account "savings", then another called "checking", and I ask for the list of accounts I must obtain: "checking", "savings"
2. If I now try to create "checking" again, I get an error
3. If now I query the balance of "checking", I must get 0.
4. If I try to delete "stocks", I get an error
5. If I delete "checking", it should not appear in the new listing of accounts

Automate Acceptance Tests

- Customer can write and later (re)run tests
  - E.g., customer writes an XML table with data examples, developers write tool to interpret table
- Tests should be automated
  - To ensure they are run after each release

Tasks

- Each story is broken into tasks
- To split the work and to improve cost estimates
- Story: customer-centered description
- Task: developer-centered description
- Example:
  - Story: "I can create named accounts"
  - Tasks: ask the user the name of the account
  - check to see if the account already exists
  - create an empty account
- Break down only as much as needed to estimate cost

- Validate the breakdown of stories into tasks with the customer
- If a story has too many tasks: break it down
- Team assigns cost to tasks
  - We care about relative cost of task/stories
  - Use abstract "units" (as opposed to hours, days)
  - Experience will tell us how much a unit is
  - Developers can assign units by bidding: "I can do this task in 2 units"
Play the Planning Game

- Customer chooses the important stories for the next release
- Development team bids on tasks
  - After first iteration, we know the speed (units/week) for each subteam
- Pick tasks => find completion date
- Pick completion date, pick stories until you fill the budget
- Customer might have to reprioritize stories

Planning Game

Test-driven development

- Write unit tests before implementing tasks
- Unit test: concentrate on one module
  - Start by breaking acceptance tests into units
- Example of a test
  ```
  addAccount("checking");
  if(balance("checking") != 0) throw ...;
  try { addAccount("checking");
  throw ...;
  } catch(DuplicateAccount e) {};
  ```
  Test both good and bad behavior
- Think about names and calling conventions

Why Write Tests First?

- Ensures that you think about testability first
  - How will you know when you are done?
  - In what order do you need to test components?
  - What infrastructure you need to test?

Test-Driven Development. Bug Fixes

- Fail a unit test
  - Fix the code to pass the test
- Fail an acceptance test (user story)
  - Means that there aren’t enough user tests
  - Add a user test, then fix the code to pass the test
- Fail on beta testing
  - Make one or more unit tests pass the failing scenario
- Always write code to fix tests
  - Ensures that you will have a solid test suite
**Simplicity (KISS)**

- **Just in time design**
  - design and implement what you know right now; don’t worry about future design decisions

- **No premature optimization**
  - You are not going to need it (YAGNI)

- **In every big system there is a simple one waiting to get out**

**Refactoring: Improving the Design of Code**

- **Make the code easier to read/use/modify**
  - Change “how” code does something

- **Why?** Incremental feature extension might outgrow the initial design
  - Expected because of lack of extensive early design

- **But needed even for waterfall model code**
  - Plan for it, no point in trying to avoid it.

**Refactoring: Remove Duplicated Code**

- **Why?** Easier to change, understand

- **Inside a single method: move code outside conditionals**

  ```java
  if(…) { c1; c2 } else { c1; c3}
  c1. if(…) { c2 } else { c3 }
  ```

- **In several methods: create new methods**

- **Almost duplicate code**
  - ... balance + 5 ... and ... balance - x ...
  - int incrBalance(int what) { return balance + what; }
  - ... incrBalance(5) ... and ... incrBalance(-x) ...

**Refactoring: Change Names**

- **Why?** A name should suggest what the method does and how it should be used

- **Examples:**
  - moveRightIfCan, moveRight, canMoveRight

  - Meth1: rename the method, then fix compiler errors
    - Drawback: many edits until you can re-run tests

  - Meth2: copy method with new name, make old one call the new one, slowly change references
    - Advantage: can run tests continuously

**Refactoring and Regression Testing**

- **Comprehensive suite needed for fearless refactoring**

- **Only refactor working code**
  - Do not refactor in the middle of implementing a feature

- **Plan your refactoring to allow frequent regression tests**

- **Modern tools provide help with refactoring**

- **Recommended book: Martin Fowler’s “Refactoring”**

**XP: Pair programming**

- **Pilot and copilot metaphor**
  - Driver and navigator

- **Pilot types, copilot monitors high level issues**
  - simplicity, integration with other components, assumptions being made implicitly

- **Disagreements point early to design problems**

- **Pairs are shuffled**
Benefits of Pairing

• Results in better code
  – instant and complete and pleasant code review
  – copilot can think about big-picture
• Reduces risk
  – collective understanding of design/code
• Improves focus and productivity
  – instant source of advice
• Knowledge and skill migration
  – good habits spread

Why Some Programmers Resist Pairing?

• “Will slow me down”
  – Even the best programmer can learn something
  from even the lowliest hacker
• It is stressful to relate to people all the time
• Need time alone to figure things out
• Afraid to show you are not a genius
  – Neither is your partner
  – Best way to learn

Why Some Managers Resist Pairing?

• Myth: Inefficient use of personnel
  – That would be true if the most time consuming part
  of programming was typing!
  – 15% increase in dev. cost, and same decrease in bugs
  • 2 individuals: 50 loc/h each, 1 bug/33 loc
  • 1 team: 80 loc/h, 1 bug/40 loc
  • 1 bug fix costs 10 hours
  • 50kloc program 2 individuals: 10000 devel + 15000 bug fix
  • 50kloc program 1 team: 12500 devel + 12500 bug fix
• Resistance from developers
  – Ask them to experiment for a short time
  – Find people who want to pair

Evaluation and Planning

• Run acceptance tests
• Assess what was completed
  – How many stories
• Discuss problems that came up
  – Both technical and team issues
• Compute the speed of the team
• Re-estimate remaining user stories
• Plan with the client next iteration

Putting It All Together

XP is...

• A lightweight development process that
  emphasizes
  – active and ongoing customer involvement
  – pay-as-you-go design
  – test-before-you-implement
What's Different About XP

- No specialized analysts, architects, programmers, testers, and integrators
  - every XP programmer participates in all of these critical activities every day.

- No complete up-front analysis and design
  - start with a quick analysis of the system
  - team continues to make analysis and design decisions throughout development.

What's Different About XP

- Develop infrastructure and frameworks as you develop your application
  - not up-front
  - quickly delivering business value is the driver of XP projects.

When to (Not) Use XP

- Use for:
  - A dynamic project done in small teams (2-10 people)
  - Projects with requirements prone to change
  - Have a customer available

- Do not use when:
  - Requirements are truly fixed
  - Cost of late changes is very high
  - Your customer is not available (e.g., space probe)

Class Approach?

- "Extreme Classical"
- Classical:
  - Staged waterfall development
  - Generation of project documentation as you go
- Extreme
  - XP planning game to move from customer requirements (user stories) to design specification
  - Test-driven development
  - Continuous system integration
  - Pair-programming (encouraged)

Conclusion

- Extreme Programming is an incremental software process designed to cope with change

- Core principles: customer-on-team, planning game, test first, rapid cycle, simplicity

- Additional principles: team programming, sustainable rhythm

- With XP you never miss a deadline; you just deliver less content