Agile Methods, eXtreme Programming, and Unit Testing with JUnit

What is Agile Development?
- Family of development processes
- Allow the team to respond to changes in any phase of development
- Accepts change as a fact of life and uses process to enable change
- Most notorious Agile Development Process: Extreme Programming
- A response to the slow predevelopment (upstream) focus of the “traditional” waterfall model

Software Lifecycle
- requirements
- reqts. analysis
- design specs
- validation
- coding
- validation
- testing
- adequacy
- maintenance
- revalidation

Waterfall model
Introduced and criticized by Winston Royce, 1970

eXtreme Programming (XP)
“Extreme Programming is a discipline of software development based on the values of simplicity, communication, feedback, and courage...”

... It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.” - Ron Jeffries

eXtreme Programming (XP)
- Defined by 12 practices
- Claim: XP ‘flattens’ the cost-of-change curve
- Most literature on XP is experience reports
- Rigorous evaluation is needed
12 Practices of XP

**XP Practices: Whole Team**
- Everyone working on project is on one team
- “Customer” – provides requirements, steers planning
- Different roles on team, but:
  - No specialists
  - Generally competent people with special skills

**XP Practices: Planning Game**
- Address two questions:
  - What will be accomplished by the due date?
  - What to do next?
- Release Planning
- Iteration Planning
  - Typically 2 week intervals

**XP Practices: Customer Tests**
- Automated acceptance tests
- Defined by customer
- Implemented by team

**XP Practices: Small Releases**
- Release functional, “useful” software every iteration
  - For evaluation by customer or release to end-users
- Releases are kept reliable by testing
  - Every test must pass

**XP Practices: Continuous Integration**
- Constantly keep the entire system integrated
- Multiple daily builds (10-20 in practice!)
- Problems with infrequent integration
  - Team not experienced with integration
  - Buggy code (problems introduced by integration)
  - Long code freezes
Cleanroom

Incremental Development of a Small System

<table>
<thead>
<tr>
<th>Customer</th>
<th>Requirements</th>
<th>Top-Level Specification</th>
<th>Incremental Design Plan</th>
<th>Complete System</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

XP Practices: Collective Code Ownership
- Anyone on team can work to improve any piece of code
- Avoids asking code “owner” to add feature
- To address problem of working with unfamiliar code
  - Pair with someone familiar with it
  - Use automated tests
- Only works if there are not any difficult components
- Always good to have more than one expert

Cleanroom

XP Practices: Coding Standards
- Common coding standard followed by everyone on team
  - Supports collective code ownership
- Specifics unimportant – as long as code all looks familiar
  - Helps to have a common glossary of terms, naming conventions, etc.

Cleanroom

XP Practices: Metaphor
- Metaphor for function of system shared by team
- For example, an agent-based information retrieval system:
  “This program works like a hive of bees, going out for pollen and bringing it back to the hive.” [www.xprogramming.com]
- Might have some general design principals e.g. no shared state

Cleanroom

XP Practices: Sustainable Pace
- Maintain pace that will be successful in the long run
- Pace should be sustainable indefinitely
- Work overtime when necessary, but don’t burn out and lose productivity

Cleanroom

XP Practices: Pair Programming
- All code written by two programmers working side-by-side
- Ensures code is reviewed
- Communicates knowledge throughout team
- Claim: results in better code
**XP Practices: Simple Design**
- Start with a simple design and keep it that way through design improvement
- Don’t make code unnecessarily general
- No wasted effort – design suited for current functionality

**XP Practices: Refactoring**
- Continuous process of design improvement called “Refactoring”
- Implementation of code change without altering interface
  - When possible
  - Remove duplication

**XP Practices: Test Driven Development**
- Test First, then write code to make it work
  - A form of Design by Contract
- Every test must pass at every build
  - Supports Continuous Integration

**J) Unit Testing**
- Based on the idea that classes should contain their own tests
- Highly localized; test(s) work within a single package
- Tests the interfaces to other packages, but assumes other packages work

**Why Unit Testing?**
- Better able to exercise all code
- Writing tests before writing code helps programmer focus on the issues that must be addressed
  - Focus on boundary conditions and special conditions that make the test fail
- Helps to keep the task of test-writing to a doable size
- Cuts down significantly on debugging time
  - Run tests every time code is compiled
  - If new code breaks a previously-passed test, bug location is easier to pinpoint

**Unit Testing Difficulties**
- Seem to be writing code twice
- Many programmers have never learned to write tests or to think hard about tests
- Overhead of test framework
The Junit Testing Framework
- Used for writing unit tests in Java
- Helps automate testing process
- Provides some basic constructs for running tests

Summary of Agile/XP methods
- A backlash against the waterfall model
- Many good ideas, repackaged
  - Very similar to Clean Room
  - Involve customer early
  - Integration planning
  - Very similar to Code Inspection
  - Multiple people review the products
- Many companies claim to be "AGILE"
- Need to evaluate the ideas carefully to determine what actually improves productivity, quality, predictability

Homework #3
- For at least 10 "interesting" methods
  - Write Junit tests
  - With those tests, try to achieve 100% branch coverage
- Write a short report describing the pros and cons of using Junit and achieving branch coverage