More XP

15-413: Introduction to Software Engineering

Jonathan Aldrich

Project Iteration Structure

- 3 week iterations
  - Weeks 4-6 (starts next week!)
  - Weeks 7-9, 10-12, 13-15+finals
- Documentation for each
  - Beginning: plan, risks
    - Includes functional test definitions
  - End: results
    - cost = person-hours of effort
    - earned value = ideal hours done * load factor
    - new load factor = cost / ideal hours done
    - Chart showing all three over time

Review meetings

- Begin in week 5
- Purpose
  - Communication
    - Status update
    - Discuss issues
    - Answer questions
  - Evaluate XP practices
  - Discuss requirements, plan, risks

Coding starts next week

- Are you ready to do XP?
  - Let's talk about requirements

Planning

- Use cards. Why?
- Customer orders
  - Primarily based on their business value
  - Customer should be aware of risks
    - Obvious importance to put risk first
  - OK for engineering to reserve a portion of the iteration effort for high-risk stories
    - NECESSARY in iteration 1: there will be a prototype requirement
- Ignore dependencies where possible
  - Why?

Dividing the work

- Divvy up the stories
  - Each person takes ideal hours equal to calendar hours / load factor
  - Load factor initially 2
    - You'll adjust later
- Buy into the time estimates
  - Developer with the story should re-estimate if they disagree
  - May require changing story allocation
- Find a buddy to pair with
  - You'll spend half your time being a buddy for someone else's stories

Coding Requirements

- Pair programming
- CVS
- Test first
  - Functional tests tied to stories
  - Unit test for each unit of code (function, class)
  - Automation for all tests
- Refactoring

- Can't enforce these in homework
  - Use review meetings instead
  - Be prepared to discuss and show examples of how you are following the practices

Functional/Acceptance Tests

- Must be written before you implement the story
- Crosses code boundaries
- Conceptually written by customer
  - Must buy in to success criteria

- Ask, what would have to be checked before I am confident this is done?
  - Write a functional tests for each scenario

Unit Tests

- Must be written before any non-trivial functionality
- Must always be at 100% for checked-in code
  - Not true for functional tests
  - Always run tests and ensure at 100% before checking in code
- Should be independent
  - Ideal: each bug makes only one test case fail

Unit Test: Bad or Good?

```
class Car {
  int gas;
  int getGas() { return gas; }
}

void myTest() {
   Car c = new Car(5);
   assert (c.getGas() == 5);
}
```

Unit Test: Bad or Good?

```
class Math {
   float divideBy2(float x) { return x/2; }
}

void myTest() {
   assert(math.divideBy2(10.0)
            == 5.0);
}
```

Unit Test: Bad or Good?

```
class Math {
    int divide5ByX(int x) {
        if (x == 0)
                throw new IllegalArgumentExn();
        return 5/x;
    }
}

void myTest() {
    try {
        math.divide5ByX(0);
        assert(false);
    } catch (IllegalArgumentExn e) {
        assert(true);
    }
}
```

Unit Testing Principles

- Test anything that might fail
  - Unclear interface
  - Complicated implementation
  - Unusual case of usage
  - Defect found
  - About to refactor
- Don't test trivial methods
  - No benefit
  - Makes testing laborious

16 September 2005

Unit Test Challenges

- Creating test objects
  - Design a constructor that completely initializes the object, just for testing
- Collaborating objects
  - Use stubs to unit test separately
  - Refactor to make more independent

16 September 2005

Test Automation

- All your tests should be automatable
  - If they aren't, you better have a good reason
- File I/O: create input, check output manually, automate comparison
- Build input recorder into program
- GUIs/Web
  - Don't test static structure
  - Separate functionality and test programmatically
  - Test interaction if you can
    - [but limit the time sink in trying tools]
- Testing tools
  - http://www.xprogramming.com/software.htm

16 September 2005

Refactoring

- You code must always be clean and well-designed
  - Documentation is *not* required by XP, but may be required by your client!
- Fix as soon as you find out it's not ideal
  - When we review your work, don't ever say, "we're going to fix that"
- Principles
  - Once and only once
  - Keep it simple: write only enough to pass test (with clean code)

16 September 2005

Defects

- Customer classifies as critical or not
- Critical
  - Estimate effort, fix immediately
  - If significant, customer chooses stories of same effort to nix
- Non-critical
  - Write as a story
    - Maybe combine several
  - Customer chooses when to fix

16 September 2005

Questions?

16 September 2005