COMP 3711

OOD

Test Driven Development And Refactoring

Larman Chapter 21

What is Extreme Programming

It is a ...

- **Discipline**
- **Software Development Methodology**
- **Lightweight**
- **Evolutionary Design**
- **Humanistic**
- **Incremental Planning**
- **Flexible scheduling**
- **Automated testing**

XP – 12 Core Practices

1. Planning
2. Small Releases
3. System Metaphor
4. Simple Design
5. Continuous Testing
6. Refactoring

7. Paired Programming
8. Collective Ownership
9. Continuous Integration
10. 40-Hour Week
11. On-site Customer
12. Coding Standards

XP Extremos

The 2 Extremos - fathers of eXtreme Programming:

Ward Cunningham - the inventor

Kent Beck - the articulator

Ron Jeffries - the realizer

XP – Continuous Testing

Test a little → Code a little

1. Planning	*7. Paired Programming*
2. Small Releases	*8. Collective Ownership*
3. System Metaphor	*9. Continuous Integration*
4. Simple Design	*10. 40-Hour Week*
5. Continuous Testing	*11. On-site Customer*
6. Refactoring	*12. Coding Standards*

5. Continuous Testing

- Unit tests (test single class or cluster of classes) - written by developer
- Acceptance testing (overall system is functioning) - written by users
- Paired programming allows better test plans
- Simple design allows frequent automated testing

Test Driven Development (TDD)

- Unit Test refers to a test of individual parts of an application, as oppose to Application Test
- The unit test code is written before the class to be tested (not an after-thought exercise)
- Developer writes unit testing code for all production code
- The unit test is written first, imagining the code to be tested is written and the challenge is to write code that will pass the test

Test Driven Development (TDD)

- Early errors
  - *syntax errors* - compiler will spot
- Later errors
  - *logic errors* - compiler can’t help
  - *bugs* - some logical errors have no immediately obvious manifestation

TDD – Popular testing framework

- xUnit jUnit NUnit

- TDD
  1. Create the fixture
  2. Do something to it (test)
  3. Evaluate the results are as expected

- Write one unit test method at a time, implement the solution to make it pass, and then repeat this process

- Analogy of level line in laying bricks or limestone

Development Unit Test

- Understand what the unit is supposed to do (i.e. its *intent* or its *contract)*
  - look for violations
  - use positive tests and negative tests
  - test *boundaries*
    - zero, one, full
    - search empty collection
    - add to a full collection

Development Unit Test - Assertion

- To check if the code is behaving as expected, use an *assertion* method
- There are many asertions available in Junit, a few examples:
    - assertTrue
    - assertEquals
    - assertSame
    - assertNull
    - assertNotNull
    - assertFalse

TDD – NextGen POS Example

- Write the SaleTest class first before writing Sale class

A group of objects used in more than one test

- Write a unit testing method in SaleTest class:
  1. Create Sale (the *fixture*) to be tested
  2. Add line items with makeLineItem method (this is the method to be tested)
  3. Ask for total, and verify it is the expected value, using `assertTrue (sale.getTotal().equals(total));`

- Create a separate testing method for each Sale method targeted to be tested (ignore trivial ones- such as *getters* and *setters*)

Larman p388

TDD – Steps

Steps suggested by *Bill Wake* in "*XP Explored*":

1. Write a single test.
2. Compile it. It shouldn't compile because no implementation code written.
3. Implement just enough code to get test to compile.
4. Runt the test and see it fail.
5. Implement just enough code to pass test.
6. Run the test and see it pass.
7. Refactor for clarity and "once and only once"
8. Repeat.

TDD - Benefits

- TDD is more than just doing Unit Testing
- By writing test first before code, it helps to clarify the detailed design (e.g. method visibility, name, return value, parameters, interfaces, behaviour)
- It improves programmer satisfaction (tests are passed)
- It builds confidence that speeds up development
- Provides provable, repeatable and automated verification
- Spinoff - best class-level documentation without needing to write documentation later
- Testing as preventive mechanism .. not final inspection

Many Test Types?

- Unit Test
- Structural Test
- Integration Test
- Interface Test
- Positive Test
- Acceptance Test
- Regression Test
- Smoke Test
- Path Test
- Component Test
- Stress Test
- Negative Test
- Functional Test
- Performance Test

XP – Refactoring

1. Planning	*7. Paired Programming*
2. Small Releases	*8. Collective Ownership*
3. System Metaphor	*9. Continuous Integration*
4. Simple Design	*10. 40-Hour Week*
5. Continuous Testing	*11. On-site Customer*
6. Refactoring	*12. Coding Standards*

6. Refactoring

•Restructure and simplify the system without changing its behavior

•Refactor out duplications (based on needs by systems and code)

•Simple Design, Continuous Integration, Collection Ownereship and Paired Programming foster an environment that provides confidence to refactor

Refactoring Defined

- A structured, disciplined method to rewrite or restructure existing code without changing its external behviour, applying small transformation steps combined with re-executing tests each step…. [Martin Folwer99]

- Fixing what is not broken …

- Fixing “*code smell*” *or* “*code stench*”

Code Smell / Code Stench

- Some symptoms:
  - Violate RDD principles
  - Classes are highly coupled (e.g. public fields)
  - Classes are not cohesive
  - *Methods are not cohesive*
  - *Code duplication*

Refactoring Goals

- Remove duplicate code
- Shorten long code
- Improve clarity
- Reduce too many instance variables
- Make long methods shorter
- Remove use of hard-coded literal constants
- Remove high coupling ....etc....

→ End results of refactored code:

  - short, tight, clear and without duplication

Refactoring Names

- Over 100 refactoring names
- A few sample examples:

Refactoring	Description
Extract Method	Transform long method into shorter one by factoring out a portion into a private helper method (see fig 21.2,21.3)
Extract Constant	Replace literal constant with constant variable
Explaining Variable	Put result of expression in a temporary variable with a name that explains purpose (see fig 21.4,21.5)
Replace Constructor Call with Factory Method	Replace new operator and constructor call with invoking a helper method that creates the object (information hiding)

Refactoring – Explaining Variable

```
// code is ok but logic is not clear
Boolean isLeapYear (int year)
 {
   return ((( year %400) == 0) ||
          ((( year % 4 ) == 0 &&
          (( year % 100) !=0)));
 }
```

```
// refactor by using explaining variable
Boolean isLeapYear (int year)
 {
   boolean isFourthYear = (( year % 4 ) == 0);
   boolean isHundrethYear = (( year % 100 ) == 0);
   boolean is4HundrethYear = (( year % 400 ) == 0);
   return (
          is 4HundrethYear ||
          (isFourthYear && ! isHundrethYear ));
}
```

Larman figs 21.4, 21.5

When Not-To Refactor?

- When code doesn’t work and needs rewriting
- Do not refactor if not sure how to make it better (do not do without a plan)
- Do not refactor beyond what is needed (diminishing marginal utility)

Some Reference Sites

- www.refactoring.com
- www.junit.org
- www.testdriven.com
- www.testing.com/agile
- www.testinglessons.com
- c2.com/cgi/wiki