CISC/CMPE 327 Software Quality Assurance

Queen's University, 2019-fall

Lecture #19 Inspection & Refactoring

Inspections - Code Refactoring

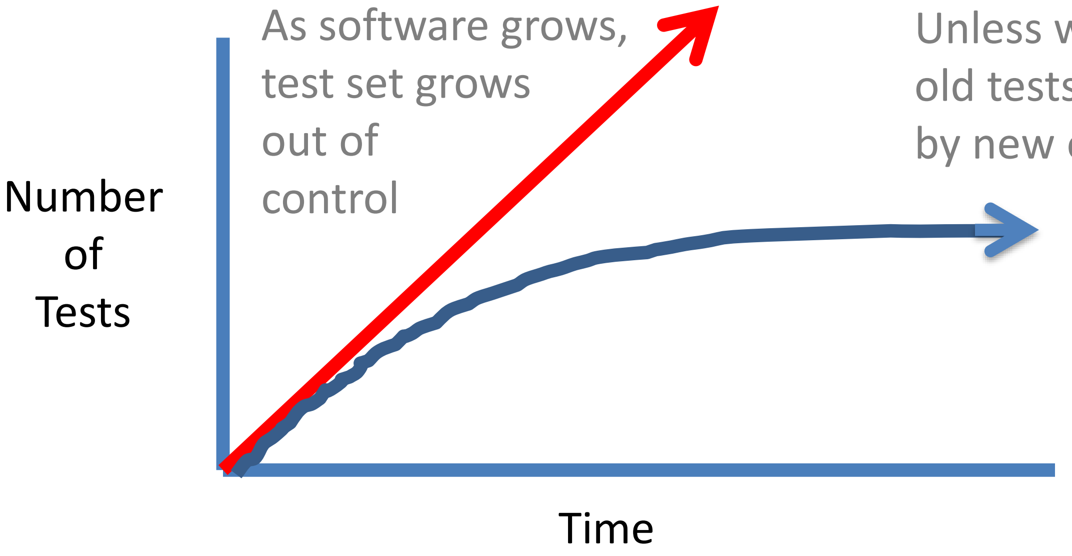
- Outline
  - Today we examine code inspection practices in eXtreme Programming
    - Pair programming
    - Code refactoring
    - Refactoring patterns

Regression Cont'd
Adding and Retiring Tests

- Whenever functionality is added or changed in the software, add and validate new tests for the new or changed functionality, and retire the tests for the replaced old functionality
- Some practitioners retire failure tests after a fixed number of new versions do not exhibit the failure, as a way to keep the number of failure tests from growing too large

Regression Cont'd

- Operational tests must also be maintained, and retired or replaced when they no longer reflect current functionality



Code Inspection in XP

- A Lightweight, Continuous Approach
  - Since XP's goal is rapid high quality software development, traditional inspection processes would take too long
  - Instead, XP uses two lightweight inspection practices continuously in the software development process
    - **Pair programming**:
      continuous immediate inspection of new code
    - **Refactoring**: continuous inspection of existing code for opportunities to improve it

Pair Programming

- Immediate Code Inspection
  - Pair programming is continuous and immediate code inspection
  - Observed to increase both quality and productivity
  - Increases quality because all code being written is inspected
  - Increases productivity because it avoids the cognitive overhead of the programmer continually switching between the code level of understanding and high level of understanding

Pair Programming

- Different Roles
  - Pair programming also involves two roles - the driver and the partner, roughly corresponding to the author and inspector
  - The idea is that the driver can confidently charge forward in the immediate coding task, while the partner keeps track of the big picture
    - Where the whole thing is going (replaces paraphrasing)
  - Normally the partner also watches for simple clerical, coding and style errors that may go unnoticed by the driver (replaces code checklists)

Code Refactoring

- What is Refactoring?
  - In XP, refactoring is to be done all the time
    - After every change to the code!
  - Consists of examining the code for opportunities to abstract or simplify its design to improve its quality and keep it more easily maintainable
  - An example of abstracting is the creation of a new method for a repeated code section when the repetition is made
  - An example of simplification is shortening code by joining similar cases or removing redundancies when new cases are added

Code Refactoring

- Refactoring is not reengineering
  - Both are intended to make software easier to understand and change
  - Reengineering takes place after a system has been maintained for some time
    - Involves modifying a legacy system to create a new system that is more maintainable
  - Refactoring is a continuous process of improvement throughout development and evolution

Code Refactoring

- The object of refactoring is to keep the design of the code as close as possible to its best design
- XP says that the best design is the simplest design
- The simplest design is characterized by four constraints
  1. The system (code plus tests) must communicate everything you want to communicate:
     all of the specification, and all of the solution
  2. The system must contain no duplicate code
  3. The system should have the fewest possible classes
  4. The system should have the fewest possible methods
- The first two of these constitute the "once and only once" rule - everything that must be in the program is in the program, and in only one place

“DRY”

- **D**on’t **R**epeat **Y**ourself
  - “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.”
    (*The Pragmatic Programmer,*
    Thomas and Hunt, 2000)

How to Refactor

- What Do We Need?
  - To refactor, we need five things:
    1. The code to be refactored
    2. Tests for the code (to ensure that we haven't changed the code's external behaviour while refactoring)
    3. A way to identify design flaws to improve
    4. A set of refactorings (templates for design changes that do not affect external behaviour) that we know how to apply
    5. A process to guide us

Identifying Flaws

- Code “Smells”
  - XP people say that when code needs refactoring, it “smells”
  - A code smell is a hint in the source code of a software system that may indicate a more serious problem
  - Code smells are heuristics, educated guesses on where improvement may be necessary

Identifying Flaws

- Code smells include:
  - Classes or methods that are too long
  - Switch statements (instead of polymorphism)*
  - “Struct” classes (classes without much real functionality)
  - Duplicate code
  - Almost (but not quite) duplicate code
  - Too many primitive type variables
  - Useless comments
  - (many, many more...)

* This assumes an OO language. In Haskell and ML, switch statements (pattern matching) smell less.

Refactoring Process

- The Refactor Cycle
  - Refactoring is applied by repeating three steps
    - Identify some code that smells
    - Apply a refactoring to improve it
    - Run the tests
  - This cycle is repeated until we are done
  - We are done when the code
    - Passes its tests
    - Communicates everything it needs to communicate
    - Has no duplication
    - Has as few classes and methods as possible

A Catalog of Refactorings

- The Fowler Catalog
  - Martin Fowler has published a by-example catalog of refactorings that can be applied
  - This catalog is a rough guide for when and why certain refactorings should be used
    - No set of metrics rivals informed human intuition
    - However, these recommendations act as inspiration when a software developer is not sure what to do

Extract Method

- One of the most common refactorings
  - If you have a code fragment that can be grouped together, turn the fragment into a method whose name explains the purpose of the method

```
void printOwing (double amount) {
   printBanner();

   //print details
   System.out.println ("name:" + _name);
   System.out.println ("amount" + amount);
}
```

```
void printOwing (double amount) {
   printBanner();
   printDetails(amount);
}

void printDetails (double amount) {
   System.out.println ("name:" + _name);
   System.out.println ("amount" + amount);
}
```

Duplicated Code

- The most significant smell in source code
  - If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them
    - Copy and paste programming
  - Imagine the (common) situation in which the original duplicated source code fragment has a bug
    - Would you rather fix one instance of the bug, or try to find and fix several dozen?

Long Method

- A common and potent stinky smell
  - The longer a method or function is, the more difficult it is to understand
  - Large methods can be decomposed into several smaller ones
    - Find parts of the method that seem to go nicely together and make a new method
  - One good technique is to look for comments
    - A block of code with a comment that tells you what it is doing can be replaced by a method whose name is based on the comment

CODE SMELL

Long Parameter List

- Hard to understand
  - Parameters are better than globals
  - However, long parameter lists are hard to understand, it can be difficult to maintain variable order, and may always be changing
  - Methods need data though, so what is the alternative?

"If you have a procedure with 10 parameters,
you probably missed some." —Alan Perlis

Replace Parameter with Method

- Reduce parameter lists
  - If a method can get a value that is passed in as a parameter by another means, it should
  - Remove the parameter and let the receiver invoke the method

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice, discountLevel);
```

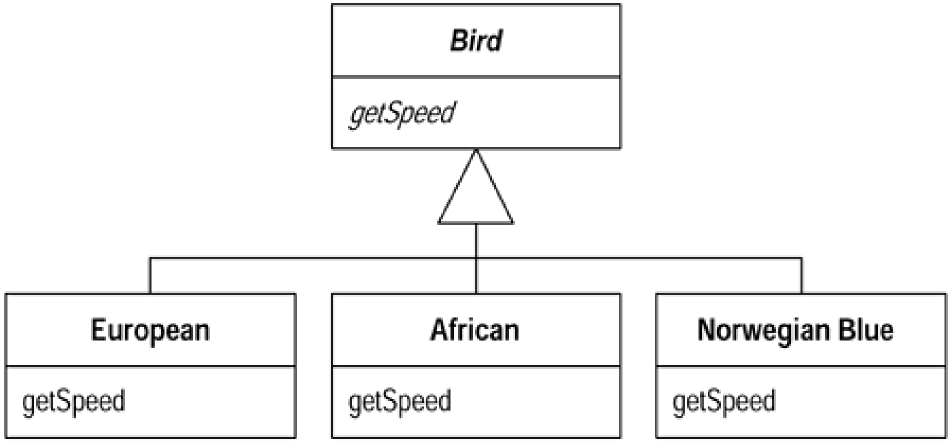
```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice (basePrice);
```

Switch Statements

- Switch statements can lead to duplication
  - Object-oriented code should have comparatively fewer switch statements than imperative code
  - Adding a new conditional case to a switch may require changing other switch statements
  - The object-oriented notion of polymorphism gives you an elegant way to deal with this problem

Replace Switch with Polymorphism

- Move each case of the switch to an overriding method in a subclass, and make the original method abstract



```
double getSpeed() {
      switch (_type) {
         case EUROPEAN:
            return getBaseSpeed();
         case AFRICAN:
            return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
         case NORWEGIAN_BLUE:
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);
      }
      throw new RuntimeException ("Should be unreachable");
   }
```

Identifier Length

- Excessively long identifiers
  - Some description may be implicitly obvious in the context of the statement
- Excessively short identifiers
  - The name of a variable should reflect its function unless it's obvious

there are rules and the rules must be followed…probably

- Often, code smells mean you should refactor
- Sometimes they don’t
  - A long **switch** statement is a reasonable way to implement a finite state machine
- Sometimes it’s a “judgment call”; experience will help you get better at making the right call

Speculative Generality

- "We'll probably need this some day..."
  - Occurs when developers include generality in a program in case it is required in the future
  - The result is often harder to understand and maintain
    - If it was being used, it would be worth it
    - If it isn't, then it just isn't
  - These can often just be removed

And more, and more...

- and more, and more...
  - We can keep improving the code in a similar fashion, using a small set of refactoring rules to improve the code step by step
    - In XP, the idea is to continuously look for opportunities to apply such improvements every time the code is changed
  - We test immediately at every step so that we know right away if we have broken anything (and when we broke it)

Summary

- Code Inspection in XP
  - XP uses continuous lightweight code inspection, in the form of pair programming and code refactoring
  - Refactoring improves the design of code without affecting its external behaviour, using a large catalog of refactoring rules
  - Refactoring is applied one small step at a time, with testing between steps to localize introduced failures
- Reference
  - Wake, Chapter 2, “What is Refactoring?”