# CS 335 — Software Development

Refactoring

Oct 29, 2007

Material adapted from Martin Fowler, *Refactoring*.

# What is refactoring?

Refactoring is the process of changing a software system in such a way that it *does not alter the external behavior* of the code yet *improves its internal structure*. It is a disciplined way to *clean up code* that minimizes the changes of introducing bugs. In essence when you refactor you are *improving the design of the code after it has been written*. [Fowler, pg xvi, emphasis added]

- Preserve interface and behavior (contract)
- Preserve correctness
- Improve design
- Improve *performance*

*Behavior* vs. *performance*

# Principles of refactoring

When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature. [Folwer, pg 7]

Before you start refactoring, check that you have a solid suite of tests. These tests must be self-checking [Fowler, pg 8]

# Principles of refactoring

> Refactroing changes the programs in small steps. If you make a mistake, it is easy to find the bug. [Fowler, pg 13]

When should you refactor?

- Three strikes and you refactor.

  > The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor.

- Refactor when you add functionality.
- Refactor when you need to fix a bug.
- Refactor as you do a code review. [Fowler, 58-59]

# Bad smells in code

Precise criteria difficult. Instead, look for bad smells.

*If it stinks, change it.*

# Bad smells in code

## Duplicated code

- Same expression in two methods of the same class.
- Same expression in two sibling subclasses
- Duplicated code in two unrelated classes

Use *Extract Method*, *Pull Up Method*, and *Extract class*.

# Bad smells in code

## Long method

> [O-O] programs that live best and longest are those with short methods. [To novices, it seems like] no computation every takes place, that [O-O] programs are endless sequences of delegation. . . . however, you learn just how valuable all those little methods are.
>
> . . . the longer a procedure is, the more difficult it is to understand. . . . be much more aggressive about decomposing methods. A heuristic we follow is that whenever we feel the need to comment something, we write a method instead. [Fowler, 76-77]

Use *Extract Method* and others.

# Bad smells in code

## Large class

When a class is trying to do too much, it often shows up as too many instance variables. When a class has too many instance variables, duplicated code cannot be far behind.

A class with too much code is prime breeding ground for duplicated code, chaos, and death. [Fowler, pg 78]

Use *Extract Class* and *Extract Subclass*.

# Bad smells in code

## Long parameter list

In our early programming days were were taught to pass in as parameters everything needed by a routine. [T]he alternative was global data, and global data is evil and usually painful. Objects change this situation because if you don't have something you need, you can always ask another object to get it for you.

[L]ong parameter lists are hard to understand because they become inconsistent and difficult to use and because you are forever changing them as you need more data. [Fowler, 78-79]

Use *Replace Parameter with Method* and *Preserve Whole Object*.

# Bad smells in code

## Divergent change and Shotgun surgery

When we make a change, we want to be able to jump to a single clear point in the system and make the change.

Divergent change occurs when one class is commonly changed in different ways for different reasons . . . you likely have a situation where two objects are better than one.

Shotgun surgery is similar but opposite—when every time you make a change, you have to make a lot of little changes to a lot of different classes. [Fowler, 79-80]

Use *Extract Class*; *Move Method* and *Inline Class*

# Bad smells in code

## Feature envy

Objects are a technique to package data with the processes used on that data. A classic smell is a method more interested in a class other than the one it is in The most common focus of the envy is the data.

The heuristic we use is to determine which class has most of the data and put the method with that data. . . Strategy and Visitor [break this rule]. [Fowler, 80]

Use *Move Method* and *Extract Method*.

# Bad smells in code

## Data clumps

> Data items tend to be like children; they enjoy hanging around in groups together. [Such groups] ought to be made into their own object. The immediate benefit is that you can shrink a lot of parameter lists. [Fowler, 81]

Use *Extract Class*.

# Bad smells in code

## Primitive obsession

People new to objects usually are reluctant to use small objects for small tasks, such as money classes, telephone numbers, ZIP codes. . .

Use *Replace Data Value with Object*

# Bad smells in code

## Switch statements

- Hallmark of OO: lack of switch statements
- Problem: duplication; many clauses in scattered locations need change
- When you see a switch statement, replace with polymorphism

  Use *Extract Method* and *Replace Conditional with Polymorphism*.

# Bad smells in code

## Comments

> [C]omments often are used as a deodorant. [L]ook at thickly commented code and notice that the comments are there because the code is bad.
>
> Comments lead us to bad code that has all the other bad smells. Our first action is to remove the bad smells by refactoring. When we're finished, we often find that the comments are superfluous.

*When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous.*

# Bad smells in code

**Inappropriate intimacy.**

Sometimes classes become far too intimate and spend too much time delving in each others' private parts. Over intimate classes need to be broken up as lovers were in ancient days. If classes do have common interests, put the commonality in a safe place and make honest classes of them, or let another class act as go-between.

Inheritance often can lead to overintimacy. Subclasses are always going to know more about their parents than their parents would like them to know. If it's time to leave home, replace inheritance with delegation.

# Bad smells in code

**Parallel inheritance hierarchies.** Shotgun surgery at the class level. Compare Abstract Factory, Bridge, Decorator. . .

**Lazy class.** A class that isn't doing enough to pay for itself should be eliminated.

**Speculative generality.** "I think I'll need it in the future." Are abstract classes doing much? Are the only users of a method the test cases?

**Temporary field.** Some classes have instance variables set only under certain circumstances.

# Bad smells in code

**Message chains.** Long line of "get this" methods; client is coupled to the structure of navigation.

**Middle man.** Encapsulation leads to delegation, which is fine; if half a class's methods are delegating to another class, it has gone too far.

**Alternative classes with different interfaces.** Move behavior to classes until the protocols are the same.

# Bad smells in code

**Incomplete library class.** Reuse is overrated; builders of library classes are rarely omniscient; use selectively.

**Data class.** Classes that have fields, getters, and setter; data classes are like children. They are ok as a starting point, but to participate as a grownup object, they need to take some responsibility.

**Refused bequest.** What if subclasses don't want or need what they are given? The hierarchy is wrong (maybe).

# Categorization

**Scope:** Intra-method; inter-method/intra-class; inter-class.

**Sophistication:** Obvious; standard, slick; artful.

# Intra method, obvious

- Inline temporary, replace temporary with query
- Consolidate duplicate conditional fragments
- Replace nested conditional with guards
- Replace conditional with polymorphism
- Introduce assertion
- Replace error code with exception
- Replace exception with test

# Intra method, standard

## Inline/extract 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);
}
```

```
int getRating() {
    return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
    return numberOfLateDeliveries > 5;
}
```

```
int getRating() {
    return (numberOfLateDeliveries > 5) ? 2 : 1;
}
```

# Intra-method, standard

## Introduce explaining variable

```
if ((platform.toUpperCase().indexOf("MAC") > -1) &&
     (browser.toUpperCase().indexOf("IE") > -1) &&
      wasInitialize && resize > 0) {
   // do something
}
```

```
final boolean isMacOS = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;
final boolean wasResize = resize > 0;

if (isMacOS && isIEBrowser && wasInitialized() && wasResized) {
   // do something
}
```

# Intra-method, standard

## Split temporary variable

```
double temp = 2 * (height + width);
System.out.println(temp);
temp = height * width;
System.out.println(temp);
```

```
final double perimeter = 2 * (height + width);
System.out.println(perimeter);
final double area = height * width;
System.out.println(area);
```