Code Smells

FIT2099: SEMESTER 1 2018

Code smells

- *A code smell is a surface indication that usually corresponds to a deeper problem in the system* – Martin Fowler

- A deeper problem is usually a design problem
- By extension, "design smells" – some small bit of a design that indicates a broader problem.

SWActor cloning

- Cloning “people” is a thing in the Star Wars universe
- So… let’s add the ability to clone a SWActor to the class:

```java
/**
 * Clone a SWActor.  Useful for starting your own Clone War
 * @return a clone of this SWActor
*/
abstract SWActor swclone();
```

- Let's pretend this is implemented and used…and *then* we add Ben Kenobi!

How should BenKenobi implement swclone()?

- There can only be one Ben Kenobi!
  - The class *enforces* this…
- So what should calling swclone() on Ben do?

Liskov Subtitution Principle

- Informally – if B is a subclass of A, you should be able to treat an instance of B as an A
- Design By Contract version:
  - Preconditions can’t be strengthened in a subclass
  - Postconditions can’t be weakened in a subclass
  - Invariants in superclass must be preserved

A smelly "fix"...

```
if(a instanceof BenKenobi) {
   throw Exception("you can't clone a Legend");
} else {
   clone = a.swclone();
}
```

Why is this smelly?

- Introduces a deep, ugly dependency between BenKenobi and every other class that uses a SWActor
  - Could check for SWLegend instead… but would still violate the LSP
- **Branching on type information is a code smell**

Candidate fixes?

- Introduce an interface SWCloneable. Branch on that instead
  - Branching on type information ☹
  - We only have to do one check ☺
- Change the specs for swclone(), such that:
  - It returns null if you attempt to clone an actor that isn't SWCloneable, or
  - It throws an exception (perhaps we define a NonCloneableActor exception) if you attempt to clone an actor that isn't SWCloneable
- A companion isCloneable() method that returns a boolean if an actor is SWCloneable might be an idea

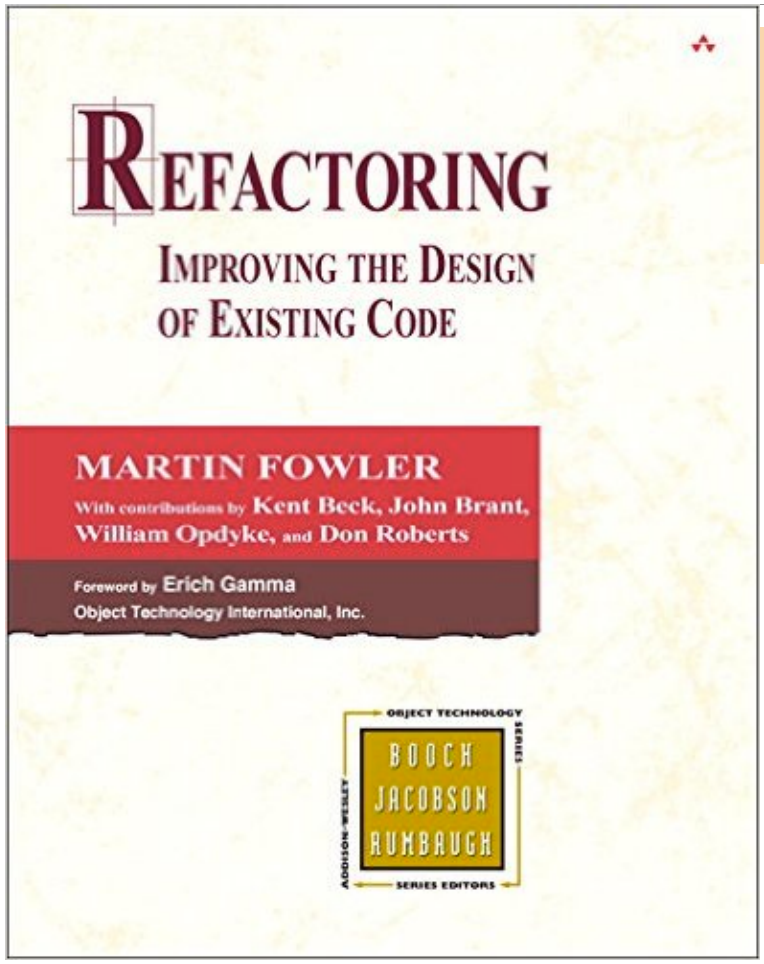
Which is the right answer?

- All of these are potentially viable solutions
- None of them is perfect
- Which is best?
  - Would need more context information to determine

Implementing the fix

- Caveat: we haven't actually tried this.
- If swclone is new and you have the discussion, easy ☺
- If swclone is already *used* in a bunch of places…
  - *Interface* will be changed, not just implementation.
  - Need to check all uses.
  - Not just human review, need to test.
- Automated unit tests make refactoring safer

Refactoring



REFACTORING

IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER

With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma
Object Technology International, Inc.

OBJECT TECHNOLOGY SERIES

BOOCH JACOBSON RUMBAUGH

SERIES EDITORS

ADDISON-WESLEY

…is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. – Martin Fowler

- Refactor to improve the quality of software
- Fowler presents a *technique* for refactoring
- Entire book is available online from the library

Why refactor?

- Improve design (extensibility)
- Improve understandability
- Makes debugging easier

When to refactor?

- Fowler:
  - When adding new features
  - When you need to fix a bug
  - As you do a code review
- You need to switch between “two hats”
  - Don't refactor and add new features at the same time!

Fowler’s taxonomy of code smells

- Fowler built a list of common code smells
- We’ll present them here. Note that they are
  - Not definitive
    - e.g. switching on type information is not on the list ☺
  - Subjective

Long smells

- Duplicated code
  - We’ve been talking about this all semester: Don’t Repeat Yourself!
- Long methods
  - The longer a method is, the more difficult it is to understand
- Large classes
  - Often a “catch-all” class that all the functionality that doesn't go anywhere else has been placed in, often as a result of a misplaced desired for “tidiness”
    - Likely to violated the Single Responsibility Principle
- Long parameter lists
  - More than three arguments to a method is generally an issue
    - Why doesn’t this method already have the data it needs in the class where it lives? Is It in the right place?

Social smells

- Divergent change
  - You have a class that you repeatedly change, in a couple of different ways
    - e.g. you have a class for which you keep having to change three methods at the same time in one context, and in another context you change a different four methods again and again
      - That is a hint that maybe there are two distinct classes there
- Shotgun surgery
  - Shotgun surgery is the opposite – whenever you add functionality, you have to make a lot of small changes to a lot of different classes
    - Easy to miss something that should have changed
    - An indication that encapsulation choices have been poor
- Feature envy
  - a method in one class spends all its effort making multiple method calls to an object of a different class
    - Often because it does not have the data it needs in the class where it is

I smell like Python:
code that doesn’t take advantage of obvious opportunities for object-orientation

- Primitive obsession
  - storing everything in language primitives, rather than creating classes for data types
    - storing everything in arraylists of Strings is the classic newbie Java version
  - Makes validation difficult
- Data clumps
  - Data clumps are groups of variables that pop up repeatedly all over the code to represent the same information
    - Data items that always appear together should likely be attributes of an object
- Switch statements
  - Often the same cascade of switch conditions appears in multiple places. If you want to change or extend it, you have to find a change them all
    - Polymorphism is often the answer
- Data classes
  - Classes that have no logic in them - they’re just passive data stores
    - Data classes are, in Fowler’s words, “almost certainly being manipulated in far too much detail by other classes

You do it…no you do it…no YOU do it…

- Message chains

```
x= a.getThingy().getStuff(b).getAgain()
```

- Middle man

```
aFred.doThing()
…
Class Fred {
    private Ginger worker;
    public void doThing() {
        worker.doActualThing();
   }
```

There are others

- Some Fowler has listed
- Some he hasn’t
- Smells vary a bit by programming environment
- Experienced programmers know what stinks!

Overengineering odours

These are somewhat rarer for novice programmers, but are surprisingly common in industry.

- Speculative Generality
  - when you add a bunch of machinery to make a class extensible for every single special case on Earth (or in a Galaxy Far, Far Away), when it’s never used
- Lazy class
  - a class that doesn’t do much any more, often after other refactorings have been done
    - Arguably, SWLegend is a lazy class ϑ

Refactorings

- Fowler presents a standard set of techniques for eliminating code smells
- Meta-technique:
  - Make small change that eliminates or reduces a smell
  - Test
  - Repeat until passes “whiff” test

A Fowler-style refactor

- There is some whiffy code in the Star Wars game.
- For example – initializeWorld in SWWorld
- 114 lines long!
- This is A LONG METHOD

Luke, Princess Leia, and Han Solo in a whiffy situation, *Star Wars* (1977)

What should we do?

The net effect is that you should 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. Such a method contains the code that was commented but is named after the intention of the code rather than how it does it. We may do this on a group of lines or on as little as a single line of code. We do this even if the method call is longer than the code it replaces, provided the method name explains the purpose of the code. The key here is not method length but the semantic distance between what the method does and how it does it.

Ninety-nine percent of the time, all you have to do to shorten a method is Extract Method . Find parts of the method that seem to go nicely together and make a new method.

-- Martin Fowler

Extract Method

Fowler provides a recipe:

- Create a new method, and name it after the intention of the method
- Figure out visibility
- Copy the extracted code from the source method into the new target method
- *…sort out issues with local variables…*
- Insert a call to method

Find a method to extract

```
public void initializeWorld(MessageRenderer iface) {
    SWLocation loc;
// Set default location string
    for (int row=0; row < height(); row++) {
       for (int col=0; col < width(); col++) {
           loc = myGrid.getLocationByCoordinates(col, row);
           loc.setLongDescription("SWWorld (" + col + ", " + row + ")");
           loc.setShortDescription("SWWorld (" + col + ", " + row + ")");
           loc.setSymbol('.');
        }
     }
…
```

Extracted method…

```
private void setDefaultLocationString() {
    SWLocation loc;
    for (int row=0; row < height(); row++) {
        for (int col=0; col < width(); col++) {
            loc = myGrid.getLocationByCoordinates(col, row);
            loc.setLongDescription("SWWorld (" + col + ", " + row + ")");
            loc.setShortDescription("SWWorld (" + col + ", " + row + ")");
            loc.setSymbol('.');
        }
    }
}
```

Concluding thoughts

- This doesn’t affect the global design much
- We’ll go through a more extensive example of refactorings (via Fowler) next lecture.
  - With architectural changes!