Refactoring

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

Outcome

After today's lecture you will be able to:

- Define the process of refactoring
- Understand the describe the basic refactoring process
- Use refactoring in your daily practice
- Understand the uses of refactoring in practice
- Understand why we refactor and what code smells are

Inspiration

“But in our enthusiasm, we could not resist a radical overhaul of the system, in which all of its major weaknesses have been exposed, analyzed, and replaced with new weaknesses.” – Bruce Leverett

“Refactoring provides enough energy to a system for it to relax into a new and more comfortable state, a new local minimum.” – Kevlin Henney

What is Refactoring

- Refactoring is the process of changing a software system such that
  - The external behavior of the system does not change
    - e.g., functional requirements are maintained
  - but the internal structure of the system is improved
- This is sometimes called
  - “improving the design after it has been written”

(Very) Simple Example

- Consolidate duplicate conditional fragments

This

```
if (isSpecialDeal()) {
     total = price * 0.95;
     send();
} else {
     total = price * 0.98;
     send();
}
```

Becomes This

```
if (isSpecialDeal()) {
     total = price * 0.95;
} else {
     total = price * 0.98;
}
send();
```

(Another) Simple Example

- Replace magic number with symbolic constant:

This

```
double potentialEnergy(double mass, double height){
    return mass * 9.81 * height;
}
```

Becomes This

```
static final double GRAVITATIONAL_CONSTANT = 9.81;
double potentialEnergy(double mass, double height) {
  return mass * GRAVITATIONAL_CONSTANT * height;
}
```

In This way, refactoring formalizes good programming practices

But, Refactoring is Dangerous!

- Although refactoring helps to reduce bugs, it can also introduce new bugs into the code
- Manager’s point of view
  - If my programmers spend time “cleaning up the code” then that’s less time implementing required functionality (and my schedule is slipping as it is!)
- To address these concerns
  - Refactoring needs to be **systematic**, **incremental**, and **safe**

Refactoring is Useful Too

- The idea behind refactoring is to acknowledge that it will be difficult to get a design right the first time and, as a program's requirements change, the design may need to change
  - refactoring provides techniques for evolving the design in small incremental steps
- Benefits
  - Often code size is reduced after refactoring
  - Confusing structures are transformed into simpler structures
    - which are easier to maintain and understand

A “Cookbook” can be Useful

- Refactoring: Improving the Design of Existing Code
  - by Martin Fowler (and Kent Beck, John Brant, William Opdyke, and Don Roberts)
- Similar to the Gang of Four’s Design Patterns
  - Provides “refactoring patterns”
- Also
  - http://www.refactoring.com/catalog
  - http://sourcemaking.com/refactoring
  - http://refactoring.guru/

Principles in Refactoring

- Fowler's definition
  - Refactoring (noun)
    - a **change made to the internal structure of software** to make it **easier to understand and cheaper to modify** without changing its observable behavior
  - Refactoring (verb)
    - to **restructure software** by applying a series of refactorings without changing its observable behavior

Principles, continued

- The purpose of refactoring is
  - to make software **easier to understand and modify**
  - **no functionality is added,** but the code is **cleaned up**, make easier to understand and modify, and sometimes is reduced in size
- Contrast this with performance optimization
  - functionality is not changed, only internal structure;
  - however, performance optimizations often involve making code harder to understand (but faster!)

Principles, continued

How do you make refactorings safe? - First, use refactoring “patterns” - Fowler’s book (and Website) assigns “names” to refactorings in the same way that the GoF’s book assigned names to patterns

- Second, test constantly!
  - This ties into the agile design paradigm
    - you write tests **before** you write the code
    - after you refactor, you run the tests and check that they all pass
    - if a test fails, the refactoring broke something **but you know about it right away** and can fix the problem before you move on

Why Should you Refactor?

- **Refactoring improves the design of software**
  - without refactoring, a design will "decay" as people make changes to a software system
- **Refactoring makes software easier to understand**
  - because structure is improved, duplicated code is eliminated, etc.
- **Refactoring helps you find bugs**
  - Refactoring promotes a deep understanding of the code at hand, and this understanding aids the programmer in finding bugs and anticipating potential bugs
- **Refactoring helps you program faster**
  - because a good design enables progress

When Should you Refactor?

- The Rule of Three
  - Three “strikes” and you refactor
    - refers to duplication of code
- Refactor when you add functionality
  - do it before you add the new function to make it easier to add the function
  - or do it after to clean up the code after the function is added
- Refactor when you need to fix a bug
- Refactor as you do a code review

Problems with Refactoring

- Databases
  - Business applications are often tightly coupled to underlying databases
    - code is easy to change; databases are not
- Changing interfaces (!!)
  - Some refactorings **require that interfaces be changed**
    - if you own all the calling code, no problem
    - if not, the interface is “published” and can’t change

Refactoring: Where to Start?

- How do you identify code that needs to be refactored?
  - Fowler uses an olfactory analogy (attributed to Kent Beck)
  - Look for “Bad Smells” in code
    - A chapter in Fowler’s book
    - Several online sources (e.g., http://sourcemaking.com/refactoring/bad-smells-in-code)
    - They present examples of “bad smells” and then suggest refactoring techniques to apply

Bad Smells in Code

- **Duplicated Code**
  - Bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!
- **Long method**
  - Long methods are more difficult to understand
    - performance concerns with respect to short methods are largely obsolete

Bad Smells in Code

- **Large Class**
  - Large classes try to do too much, which reduces cohesion
- **Long Parameter List**
  - Hard to understand, can become inconsistent if the same parameter chain is being passed from method to method
- **Divergent Change**
  - symptom: one type of change requires changing one subset of methods; another type of change requires changing another subset
  - e.g., "I have to change these three methods every time I get a new database."
  - Related to cohesion

Bad Smells in Code

- **Shotgun Surgery**
  - A change requires lots of little changes in a lot of different classes
- **Feature Envy**
  - A method requires lots of information form some other class
    - Move it closer!
- **Data Clumps**
  - Attributes that clump together (are used together) but are not part of the same class

Bad Smells in Code

- **Primitive Obsession**
  - Characterized by a reluctance to use classes instead of primitive data types
- **Switch Statements**
  - Switch statements are often duplicated in code; they can typically be replaced by use of polymorphism (let OO do your selection for you!)
- **Parallel Inheritance Hierarchies**
  - Similar to shotgun surgery; each time I add a subclass to one hierarchy, I need to do it for all related hierarchies
    - Note: some design patterns encourage the use of parallel inheritance hierarchies (so they are not always bad!)

Bad Smells in Code

- **Lazy Class**
  - A class that no longer “pays its way”
    - e.g., may be a class that was downsized by a previous refactoring, or represented planned functionality that did not pan out
- **Speculative Generality**
  - “Oh, I think we need the ability to do this kind of thing someday”
    - thus have all sorts of hooks and special cases to handle things that aren’t required
- **Temporary Field**
  - An attribute of an object is only set/used in certain circumstances;
    - but an object should need all of its attributes

Bad Smells in Code

- **Message Chains**
  - a client asks an object for another object and then asks that object for another object etc.
    - client depends on the structure of the navigation
    - any change to the intermediate relationships requires a change to the client
- **Middle Man**
  - If a class is delegating more than half its responsibilities to another class, do you really need it? Involves trade-offs, some design patterns encourage this (e.g., Decorator)
- **Inappropriate Intimacy**
  - Pairs of classes that know too much about each other's implementation details (loss of encapsulation)

Bad Smells in Code

- **Data Class (information holder)**
  - These are classes that have fields, getting and setting methods for the fields, and nothing else; they are data holders, but objects should be about data AND behavior
- **Refused Bequest**
  - A subclass ignores most of the functionality provided by its superclass
  - Subclass may not pass the “IS-A” test
- **Comments** (!)
  - Comments are sometimes used to hide bad code
    - “…comments are often used as a deodorant”(!)

Are there any questions?