Lecture 27: Refactoring

Kenneth M. Anderson
Object-Oriented Analysis and Design
CSCI 6448 - Spring Semester, 2003

Credit where Credit is Due

- Some of the material for this lecture and lecture 26 is taken from “Refactoring: Improving the Design of Existing Code” by Martin Fowler; as such some material is copyright © Addison Wesley, 1999

Last Lecture

- Design Patterns
  - Covered Examples
    - Iterator
    - Flyweight
    - Decorator
    - Observer
    - State
    - Composite

Goals for this lecture

- Introduce the concept of Refactoring and cover a few examples
- In lecture 28, we will present a tutorial that will introduce a few additional refactoring techniques

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 (page 243); 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();
```

Refactoring is thus Dangerous!

- 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 this concern
  - 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 a refactoring
  - Confusing structures are transformed into simpler structures
    - which are easier to maintain and understand

A “cookbook” can be useful

- “New” Book
  - 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”

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
- contrast this with performance optimization
  - again functionality is not changed, only internal structure; however performance optimizations often involve making code harder to understand (but faster!)

Principles, continued

- When you systematically apply refactoring, you wear two hats
  - adding function
    - functionality is added to the system without spending any time cleaning the code
  - refactoring
    - no functionality is added, but the code is cleaned up, made easier to understand and modify, and sometimes is reduced in size

Principles, continued

- How do you make refactoring safe?
  - First, use refactoring “patterns”
    - Fowler’s book assigns “names” to refactorings in the same way that the GoF’s book assigned names to patterns
  - Second, test constantly!
    - This ties into the extreme programming paradigm, you write tests before you write code, after you refactor code, you run the tests and make sure they all still 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
  - Design Changes that are difficult to refactor
    - This is why Extreme Programming says that software engineers need to have “courage”!

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 very valuable chapter in Fowler’s book
    - It presents examples of “bad smells” and then suggests 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 lots of 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
- Divergent Change
  - Deals with cohesion; symptom: one type of change requires changing one subset of methods; another type of change requires changing another subset

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 from some other class (move it closer!)
- Data Clumps
  - attributes that clump 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

Bad Smells in Code

- Lazy Class
  - A class that no longer "pays its way"
    - e.g. may be a class that was downsized by 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"
- Temporary Field
  - An attribute of an object is only set 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. Bad because client depends on the structure of the navigation
- Middle Man
  - If a class is delegating more than half of its responsibilities to another class, do you really need it?
- Inappropriate Intimacy
  - Pairs of classes that know too much about each other's private details

Bad Smells in Code

- Alternative Classes with Different Interfaces
  - Symptom: Two or more methods do the same thing but have different signature for what they do
- Incomplete Library Class
  - A framework class doesn't do everything you need

Bad Smells in Code

- Data Class
  - 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
- Comments (!)
  - Comments are sometimes used to hide bad code
    - “…comments often are used as a deodorant” (!)

The Catalog

- The refactoring book has 72 refactoring patterns!
  - I’m only going to cover a few of the more common ones, including
    - Extract Method
    - Replace Temp with Query
    - Move Method
    - Replace Conditional with Polymorphism
    - Introduce Null Object

Extract Method

- You have a code fragment that can be grouped together
- Turn the fragment into a method whose name explains the purpose of the fragment
- Example, next slide

Extract Method, continued

```
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);
}
```

Replace Temp with Query

- You are using a temporary variable to hold the result of an expression
- Extract the expression into a method; Replace all references to the temp with the expression. The new method can then be used in other methods
- Example, next slide

Replace Temp with Query, continued

```
double basePrice = _quantity * _itemPrice
if (basePrice > 1000)
        return basePrice * 0.95;
else
        return basePrice * 0.98;
==============================
if (basePrice() > 1000)
        return basePrice() * 0.95;
else
        return basePrice() * 0.98;
…
double basePrice() {
        return _quantity * _itemPrice;
}
```

Move Method

- A method is using more features (attributes and operations) of another class than the class on which it is defined
- Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether
- An example of move method is available on the class website (it can’t fit into the slides!)

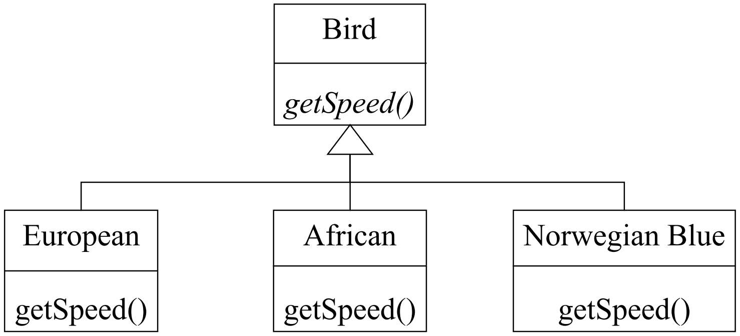
Replace Conditional with Polymorphism

- You have a conditional that chooses different behavior depending on the type of an object
- Move each “leg” of the conditional to an overriding method in a subclass. Make the original method abstract

Replace Conditional with Polymorphism, continued

```
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(“Unreachable”)
}
```

Replace Conditional with Polymorphism, continued

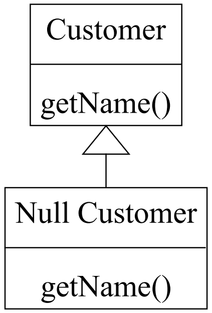


See example available from class website for more details.

Introduce Null Object

- Repeated checks for a null value
- Replace the null value with a null object

```
if (customer == null) {
    name = “occupant”
} else {
    name = customer.getName()
}
if (customer == null) {
…
```



Introduce Null Object

```
if (customer.isNull()) {
    name = “occupant”
} else {
    name = customer.getName()
}
===========================
public class nullCustomer {
   public String getName() { return “occupant”;}
}
===========================
customer.getName();
```

The conditional goes away entirely!!