

CMSC 433 – Programming Language Technologies and Paradigms Spring 2004

Refactoring
March 4, 2004

Lots of material taken from Fowler, *Refactoring:
Improving the Design of Existing Code*

1

Evolving Software

- Problem
 - The requirements of real software often change in ways that cannot be handled by the current design
 - Moreover, trying to anticipate changes in the initial implementation can be difficult and costly
- Solution
 - Redesign as requirements change
 - **Refactor** code to accommodate new design

2

Example

- (p204) Replace Magic Number with Symbolic Constant

```
double potentialEnergy(double m, double h) {
    return m * 9.81 * h;
}
```
- becomes...

```
static final double G = 9.81;
double potentialEnergy(double m, double h) {
    return m * G * h;
}
```

3

Some Motivations for This Refactoring

- Magic numbers have special values
 - But why they have those values is not obvious
 - So we're like to give them a name
- Magic numbers may be used multiple times
 - Easy to make errors
 - May make a typo when putting in a number
 - May need to change a number later (more digits of G)

4

Conventional Wisdom: The Design is Fixed

- Software process looks like this:
 - Step 1: Design, design, design
 - Step 2: Build your system
- Once you're on step 2, don't change the design!
 - You might break something in the code
 - You need to update your design documents
 - You need to communicate your new design with everyone else

5

What if the Design is Broken?

- You're kind of stuck
 - Design changes are very expensive
 - When you're "cleaning up the code," you're not adding features
- Result: An inappropriate design
 - Makes code harder to change
 - Makes code harder to understand and maintain
 - Very expensive in the long run

6

Refactoring Philosophy

- It's hard to get the design right the first time
 - So let's not even pretend
 - Step 1: Make a *reasonable* design that should work, but...
 - Plan for changes
 - As implementers discover better designs
 - As your clients change the requirements (!)
- But how can we ensure changes are safe?

7

Refactoring Philosophy (cont'd)

- Make all changes small and methodical
 - Follow mechanical patterns (which could be automated in some cases) called *refactorings*, which are *semantics-preserving*
- Retest the system after each change
 - By rerunning all of your unit tests
 - If something breaks, you know what caused it
 - Notice: we need fully automated tests for this case
 - We're going to be running them a lot

8

Two Hats

- Refactoring hat
 - You are updating the design of your code, but not changing what it does. You can thus rerun existing tests to make sure the change works.
- Bug-fixing/feature-adding hat
 - You are modifying the functionality of the code.
- May switch hats frequently
 - But know when you are using which hat, to be sure that you are reaching your end goal.

9

Principles of Refactoring

- In general, each refactoring aims to
 - Decompose large objects into smaller ones
 - Distribute responsibility
- Like design patterns
 - Adds composition and delegation (read: indirection)
 - In some sense, refactorings are ways of applying design patterns to existing code

10

Principles of Refactoring

- Refactoring improves design
 - Fights again "code decay" as people make changes
- Refactoring makes code easier to understand
 - Simplifies complicated code, eliminates duplication
- Refactoring helps you find bugs
 - In order to make refactorings, you need to clarify your understanding of the code. Makes bugs easier to spot.
- Refactoring helps you program faster
 - Good design = rapid development

11

When to Refactor

- The "Rule of Three"
 - Three strikes and you refactor
 - The third time you duplicate something, refactor
- Refactor when you add a feature
 - Make it easier for you to add the feature
- Refactor when you have a bug
 - Simplify the code as you're looking for the bug
 - (Could be dangerous...)
- Refactor when you do code reviews
 - ...if you'd be embarrassed to show someone the code

12

When to Refactor: An Analogy

- Unfinished refactoring is like going into debt
- Debt is fine as long as you can meet the interest payments (extra maintenance costs)
- If there is too much debt, you will be overwhelmed
 - [Ward Cunningham]

13

Barriers to Refactoring

- May introduce errors
 - Mitigated by testing
 - Clean first, *then* add new functionality
- Cultural issues
 - Producing negative lines of code
 - “We pay you to add new features, not to improve the code!”
- If it ain't broke, don't fix it

14

Barriers to Refactoring (cont'd)

- Tight coupling with implementations
 - E.g., databases that rely on schema details
- Public interfaces
 - If others rely on your API, you can't easily change it
 - I.e., you can't refactor if you don't have all the code
- Designs that are hard to refactor
 - It might be hard to see a path from the current design to the new design
 - You may be better off starting from scratch

15

What Code Needs to be Refactored?

- Bad code exhibits certain characteristics that can be addressed with refactoring
 - These are called “smells”
- Different smells suggest different refactorings

16

Feature Envy

- A method seems more interested in a class other than the one it is actually in
 - E.g., invoking lots of methods
- Move Method
 - Move method from one class to another
- Extract Method
 - Pull out code in one method into a separate method

17

Move Method




- Should other methods also be moved?
- What about sub- and superclasses?
- What about access control (public, protected)?

18

Extract Method

```
void printOwning(double amt) {  
    printBanner();  
    System.out.println("name" + n);  
    System.out.println("amount" + amt);  
}  
  
void printDetails(double amt) {  
    System.out.println("name" + name);  
    System.out.println("amount" + amt);  
}
```



```
void printOwning(double amt) {  
    printBanner();  
    printDetails(amt);  
}
```

- Are you ever going to reuse this new method?
- Local variable scopes?
- Extra cost of method invocation?

19


Long Method

- A method is too long. Long methods are harder to understand than lots of short ones.
- Can decompose with Extract Method
- Replace Temp with Query
 - Remove code that assigns a method call to a temporary, and replace references to that temporary with the call
- Replace Method with Method Object
 - Use the command pattern to build a “closure”

20

Replace Temp with Query

```
double basePrice = num * price;  
if (basePrice > 1000)  
    return basePrice * 0.95;  
else  
    return basePrice * 0.98;
```



```
double basePrice() {  
    return num * price;  
}  
if (basePrice() > 1000)  
    return basePrice() * 0.95;  
else  
    return basePrice() * 0.98;
```

- Does this aid other refactorings?
 - E.g., Extract Method

21