

CMSC 433 – Programming Language Technologies and Paradigms Fall 2006

Refactoring
October 19th, 2006

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

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

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

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

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;  
}
```

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)

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?

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

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.

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

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

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

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]

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

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

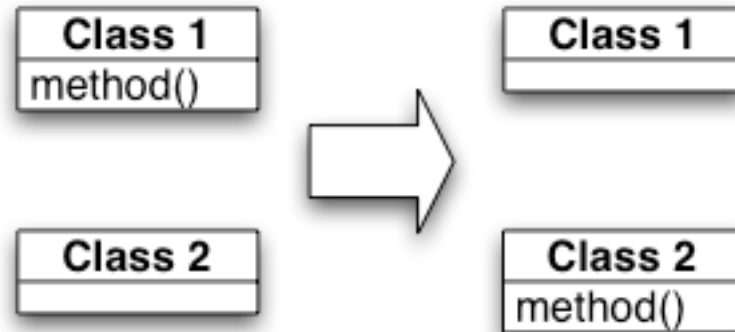
What Code Needs to be Refactored?

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

Feature Envy

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

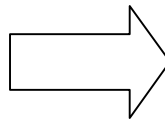
Move Method



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

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);  
}
```

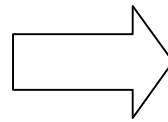
- Are you ever going to reuse this new method?
- Local variable scopes?
- Extra cost of method invocation?

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”

Replace Temp with Query

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

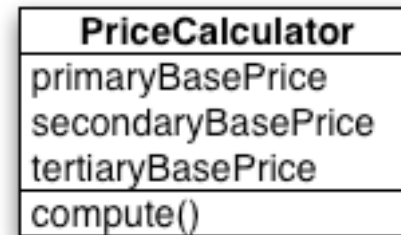
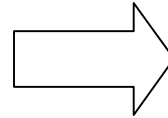


```
double basePrice() {  
    return num * price;  
}  
if (basePrice() > 1000)  
    return basePrice() * 0.95;  
else  
    return basePrice() * 0.98;
```

- Does this aid other refactorings?
 - E.g., [Extract Method](#)

Replace Method with Method Object

```
double price() {  
    double primaryBasePrice;  
    double secondaryBasePrice;  
    double tertiaryBasePrice;  
    // long computation  
    ...  
}
```

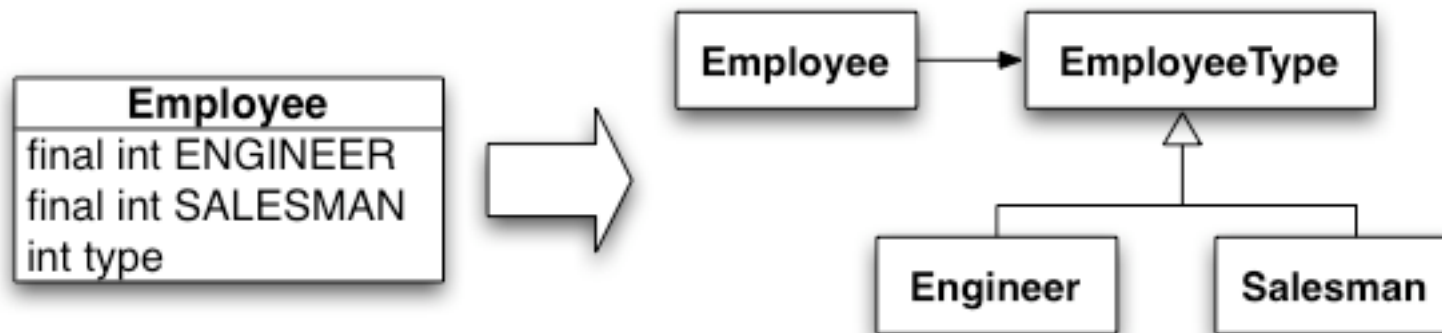


- Calls to price() changed to
 - `new PriceCalculator(this).compute()`
- Now apply techniques to break up compute() in PriceCalculator

Switch Statements

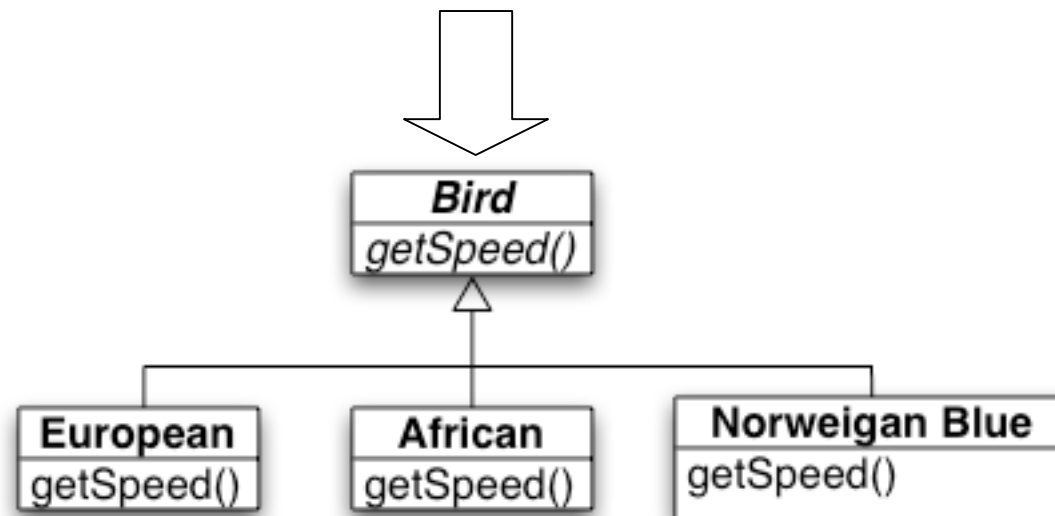
- Usually not necessary in delegation-based OO programming
- Two steps:
 1. Replace Type Code with State/Strategy
 - Define a class hierarchy, a subclass for each type code
 2. Replace Conditional with Polymorphism
 - Call method on state object to perform the check; switching is based on dynamic dispatch

Replace Type Code with State/Strategy



Replace Conditional with Polymorphism

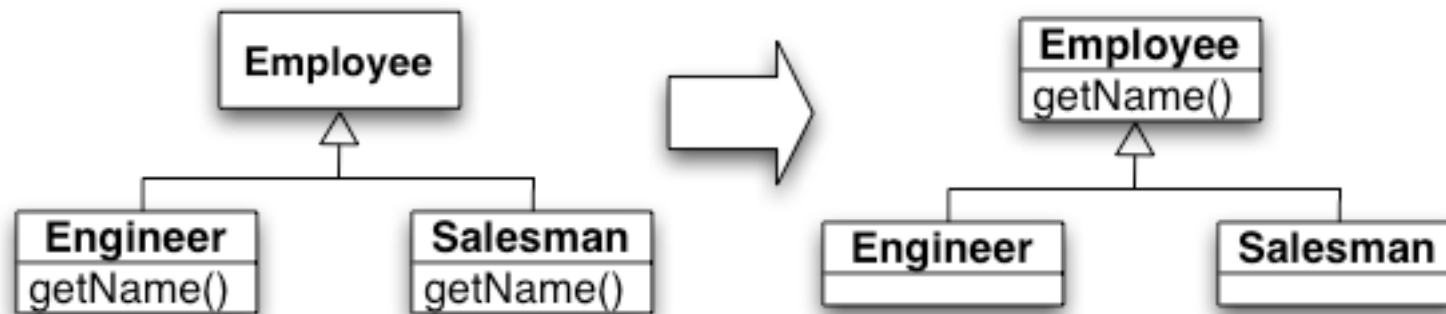
```
double getSpeed() {  
    switch (kind) {  
        case EUROPEAN: return getBaseSpeed();  
        case AFRICAN: return getBaseSpeed()-loadFactor()*numberOfCoconuts;  
        case NORWEGIAN_BLUE: return (isNailed) ? 0 : getBaseSpeed(voltage);  
        throw new RuntimeException("Should be unreachable");  
    }  
}
```



Duplicated Code

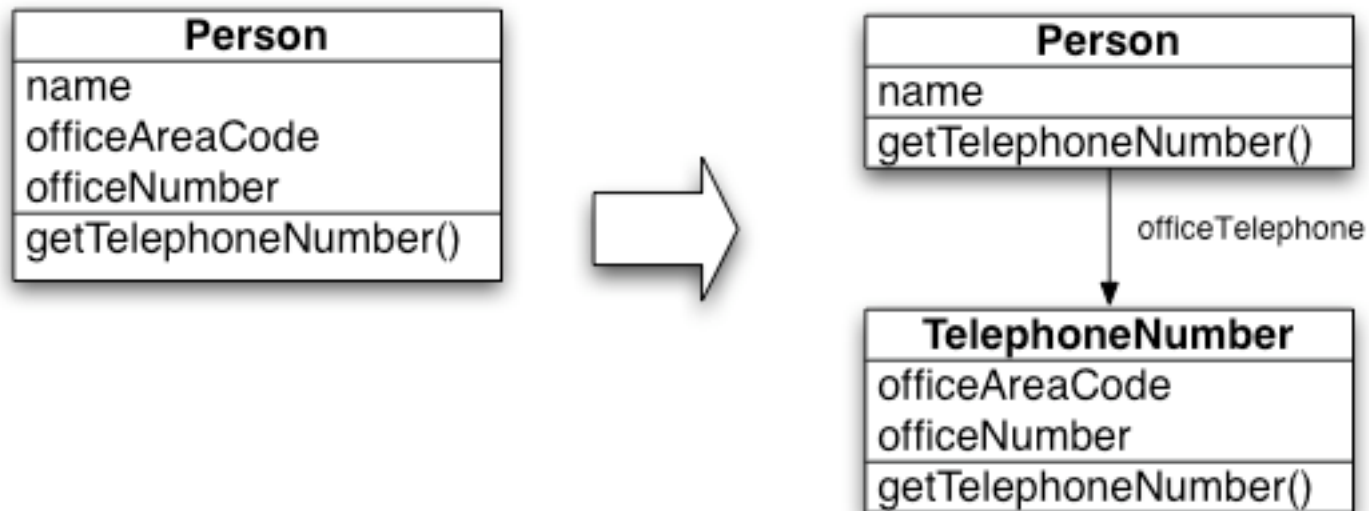
- The same expression used in different places in the same class
 - Use Extract Method to pull it out into a method
- The same expression in two subclasses sharing the same superclass
 - Extract Method in each, then
 - PullUp method into parent
- Duplicated code in two unrelated classes
 - Extract Class - Break a class that does many things into smaller classes, then use new class in both places

Pull Up Method



- Might do other refactorings if methods don't quite match
- What if doesn't appear in all subclasses?

Extract Class



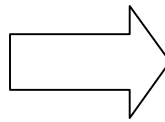
- How do we decide what goes in new class?
- Do fields still need to be accessed in orig class?

Long Parameter List

- Lots of parameters occlude understanding
- Replace Parameter with Method
 - Remove method parameters and instead use some other way to get the parameter value (e.g., method call)
- Introduce Parameter Object
 - Group parameters that go together into a container object

Replace Parameter with Method

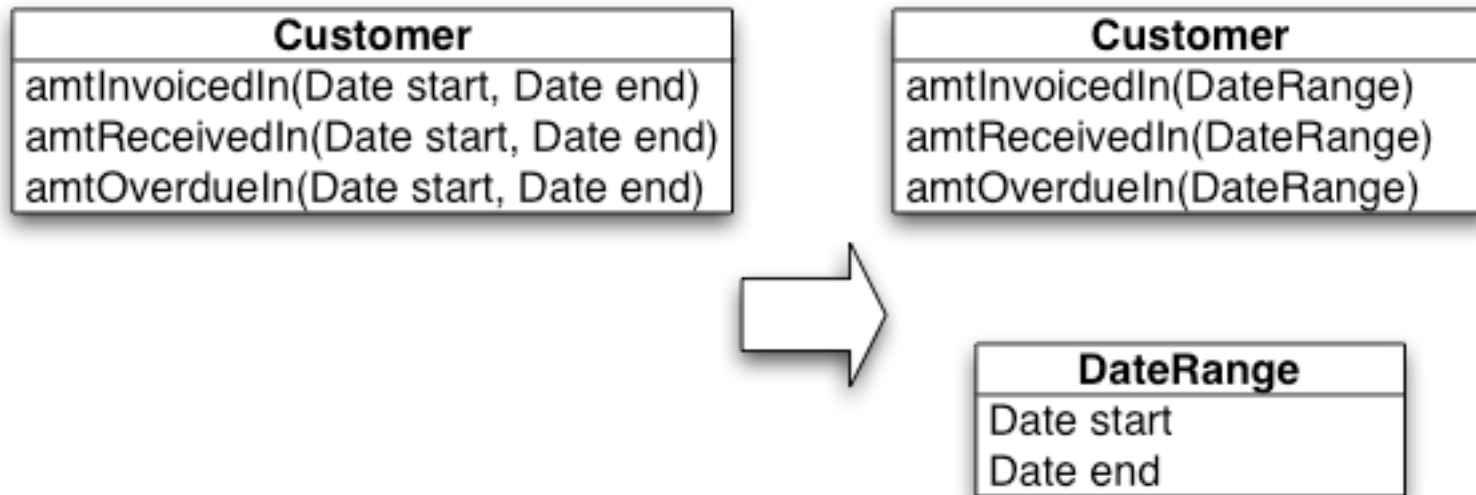
```
double basePrice = num * price;  
double discount = getDiscount();  
double finalPrice =  
    discountedPrice(basePrice, discount);
```



```
double basePrice = num * price;  
double finalPrice =  
    discountedPrice(basePrice);
```

- discountedPrice can call getDiscount() itself

Introduce Parameter Object



Refactoring with Tools

- Many refactorings can be performed automatically
- This reduces the possibility of making a silly mistake
- Eclipse provides support for refactoring in Java
 - <http://www.eclipse.org>

Divergent Change

- One class is commonly changed in different ways for different reasons
 - To add a new database, change these three methods
 - To add a new financial currency, change these four
- Suggests maybe this shouldn't be one object
- Apply Extract Class to group together variations

Shotgun Surgery

- Every time I make change X, I have to make lots of little changes to different classes
 - Opposite of Divergent Change
- Move Method
- Move Field
 - Switch field from one class to another
- Inline Class
 - A class isn't doing very much, so inline its features into its users (reverse of Extract Class)

Other Bad Smells

- Data Clumps
 - Objects seem to be associated, but aren't grouped together
- Primitive Obsession
 - Reluctance to use objects instead of primitives
- Parallel Inheritance Hierarchies
 - Similar to Shotgun Surgery; every time we add a subclass in one place, we need to add a corresponding subclass to another

Other Bad Smells (cont'd)

- Lazy Class
 - A class just isn't useful any more
- Speculative Generality
 - “Oh, I think we need the ability to do this kind of thing someday.”
- Temporary Field
 - Instance variable only used in some cases. Confusing to figure out why it's not being set everywhere.

Other Bad Smells (cont'd)

- Message Chains
 - Long sequences of gets or temporaries; means client is tied to deep relationships among other classes
- Middle Man
 - Too much delegation. If a class delegates lots of its functionality to another class, do you need it?
- Inappropriate Intimacy
 - Classes rely on too many details of each other

Other Bad Smells (cont'd)

- Alternative Classes with Different Interfaces
 - Methods do the same thing but have different interfaces
- Incomplete Library Class
 - Library code doesn't do everything you'd like
- Data Class
 - Classes that act as “structs,” with no computation
- Refused Bequest
 - Subclass doesn't use features of superclass

Other Bad Smells (cont'd)

- Comments!
 - If code is heavily commented, either
 - It's very tricky code (e.g., a hard algorithm), or
 - The design is bad, and you're trying to explain it
 - “When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous.”

More information

- Textbook: Refactoring by M. Fowler
- Catalog of refactorings:
 - <http://www.refactoring.com/catalog/index.html>
- Refactoring to patterns
 - <http://industriallogic.com/xp/refactoring/>