

COMP 150-SEN

Software Engineering Foundations

Refactoring

Spring 2019

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

Conventional Wisdom: Fixed Design

- (Abbreviated) “waterfall mode” of software process
 - Step 1: Design, design, design
 - Step 2: Build your system
- Once youi’ve done 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
 - Here, “design” is something lower-level than “software architecture”
 - 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;  
}
```

Motivations for This Refactoring

- Magic numbers have special values
 - But why they have those values is not obvious
 - So we'd like to give them a name
- Magic numbers may be used multiple times
 - Might make a typo when putting in a number
 - Might 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 implementors 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 called *refactorings*
 - Should be *semantics-preserving*
 - In theory, could be automated
- Retest the system after each change
 - By rerunning all the unit tests
 - If something breaks, you know what caused it
 - Notice: we need fully automated tests here
 - We're going to be running them a lot

Two Hats

- Refactoring hat
 - Updating code design, but not changing behavior
 - Can rerun existing tests to ensure change works
- Bug-fixing/feature-adding hat
 - Modifying functionality of code
 - Now some tests might break, need to fix them
- May switch hats frequently
 - But know when you are wearing each hat!

Principles of Refactoring

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

Principles of Refactoring (cont'd)

- Refactoring improves design
 - Fights against “code decay” as developers make changes
- Refactoring makes code easier to understand
 - Simplifies complicated code, eliminates duplication
- Refactoring might help you find bugs
 - To refactor code, you need to understand it!
- 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, though!)
- Refactor when you do code reviews
 - ...especially 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

Barriers to Refactoring

- Refactoring might introduce errors
 - Mitigated by testing
- Cultural issues
 - Producing negative lines of code!
- If it ain't broke, don't fix it
- Tight coupling with implementations
- Public interfaces
 - If others rely on your API, you can't refactor it
 - I.e., you can't refactor if you don't have all the code
- Designs that are hard to refactor
 - You might be better off starting from scratch

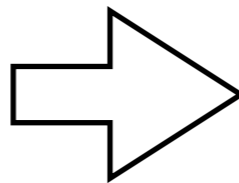
Code Smells

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

Smell: Feature Envy

- Method more interested in a class other than this
- Refactoring: Move Method

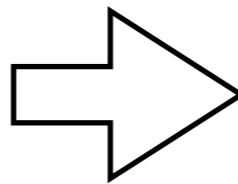
```
class A { m(); }  
class B
```



```
class A  
class B { m(); }
```

- Move other methods? Sub-/superclasses? public/private?
- Refactoring: Extract Method

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



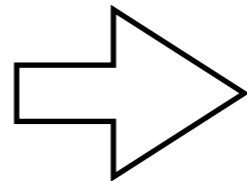
```
void printDetails(double amt) {  
    println("name" + n);  
    println("amount" + amt);  
}  
  
void printOwning(double amt) {  
    printBanner();  
    printDetails(amt);  
}
```

- Will the method be reused? Local variable scopes?

Smell: Long Method

- Can decompose with Extract Method
- Replace Temp with Query

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

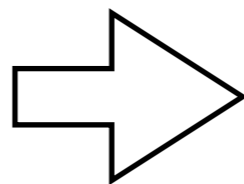


```
double basePrice() {
    return num * price;
}
if (basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;
```

- (Does this aid other refactorings?)

- Replace Method with Method Object

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



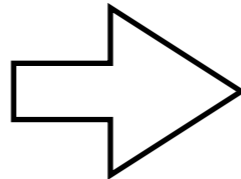
```
class PriceCalculator {
    double primaryBasePrice;
    double secondaryBasePrice;
    double compute() { ... }
}
```

- Change `price()` to `new PriceCalculator(this).compute()`
- Now apply refactorings to break up `compute()`

Smell: Switch Statements

- Usually not needed in OO programming
- Replace Type Code with State/Strategy

```
class Employee {  
    final int ENGINEER;  
    final int SALESMAN;  
    int type;  
}
```



```
interface EType {...}  
class Engineer implements EType { ... }  
class Salesman implements EType { ... }  
class Employee { EType typ; }
```

- Replace Conditional with Polymorphism

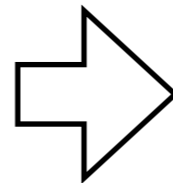
```
double getSpeed() {  
    switch (kind) {  
        case EUROPEAN: return getBaseSpeed();  
        case AFRICAN: return getBaseSpeed()-loadFactor()*numberOfCoconuts;  
        case NORWEGIAN_BLUE: return (isNailed) ? 0 : getBaseSpeed(voltage);  
    } }  
}
```

```
interface Bird { double getSpeed(); }  
class European implements Bird { double getSpeed() { ... } }  
class African implements Bird { double getSpeed() { ... } }  
class NorwegianBlue implements Bird { double getSpeed() { ... } }
```

Smell: Duplicated Code

- Same expression in different places in same class
 - Use Extract Method to pull into a single method
- Same expression in two subclasses with same superclass
 - Extract Method in each, then PullUp method into parent

```
class Employee { ... }
class Engineer extends Employee {
    String getName() { ... }
}
class Salesman extends Employee {
    String getName() { ... }
}
```



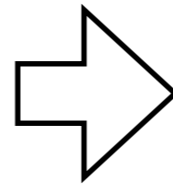
```
class Employee {
    String getName() { ... }
}
class Engineer extends Employee {
    ... }
class Salesman extends Employee {
    ... }
}
```

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

Smell: Duplicated Code (cont'd)

- Duplicated code in two unrelated classes
 - Extract Class to break up class into smaller classes

```
class Person {
    String name;
    int officeAreaCode;
    int officeNumber;
    int getTelephoneNumber() { ... }
}
```



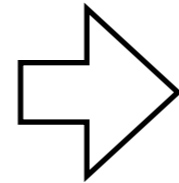
```
class Person {
    String name;
    TelephoneNumber num;
    int getTelephoneNumber() {
        num.getTelephoneNumber();
    }
}
class TelephoneNumber {
    int officeAreaCode;
    int OfficeNumber;
    int getTelephoneNumber() { ... }
}
```

- How to decide what goes in new class?
- Do fields still need to be access in original class?

Smell: Long Parameter List

- Replace Parameter with Method

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

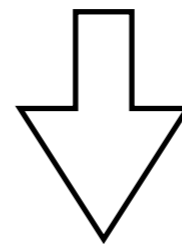


```
double base = num * price;
double finalPrice =
    discountedPrice(base);
```

- `discountedPrice` can call `getDiscount()` itself

- Introduce Parameter Object

```
class Customer {
    amtInvoiced(Date start, Date end) { ... }
    amtReceived(Date start, Date end) { ... }
    amtOverdue(Date start, Date end) { ... }
}
```



```
class DateRange { Date start, end; }
class Customer {
    amtInvoiced(DateRange r) { ... }
    amtReceived(DateRange r) { ... }
    amtOverdue(DateRange r) { ... }
}
```

Refactoring with Tools

- Many refactorings can be performed automatically
 - Reduces possibility of making a silly mistake\
- Eclipse provides support for Java refactorings
 - <http://www.eclipse.org>

Smell: Divergent Change

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

Smell: Shotgun Surgery

- Every time I make change X, I have to make lots of little changes to different classes
 - Opposite of divergent change
- Try these refactorings:
 - 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
 - Every time we add a subclass in one place, we need to add a corresponding subclass in another place
- Lazy Class
 - A class just isn't useful any more
- Speculative Generality
 - “Oh, I think we will need this ability some day”
- Temporary Field
 - Instance variable only used in some cases

Other Bad Smells (cont'd)

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

Other Bad Smells (cont'd)

- Data Class
 - Classes that act as “structs,” with no computation
- Refused Bequest
 - Subclass doesn't use features of superclass
- 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 become superfluous.”

More Information

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