CSE 352

Lecture 8

Refactoring Principles / Bad Smells in Code

Refactoring Principals

- Defining Refactoring
- Why Should You Refactor?
- When Should You Refactor?
- What Do I Tell My Manager?
- Problems with Refactoring
- Refactoring and Design
- Refactoring and Performance
- Where Did Refactoring Come From?

Defining Refactoring

- ***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.*
- ***Refactor*** *(verb): to restructure software by applying a series of refactorings without changing its observable behavior.*

Why Should You Refactor?

- Refactoring improves the design of software
  - Without refactoring the design of the program will decay
  - Poorly designed code usually takes more code to do the same things, often because the code does the same thing in different places
- Refactoring makes software easier to understand
  - In most software development environments, somebody else will eventually have to read your code
- Refactoring helps you find bugs
- Refactoring helps you program faster

When Should You Refactor?

- The Rule of three (Don Roberts)
- Refactor when you add function
  - Helps you to understand the code you are modifying
  - Sometimes the existing design does not allow you to easily add the feature
- Refactor when you need to fix a bug
  - If you get a bug report its a sign the code needs refactoring because the code was not clear enough for you to see the bug in the first place
- Refactor as you do a code review
  - Code reviews help spread knowledge through the development team
  - Works best with small review groups
  - XP pair programming is active code review taken to its limit

What Do I Tell My Manager?

- If the manager is technically savvy, introducing the subject may not be that hard
- Stress the quality aspects if the manager is *genuinely* quality oriented
  - Position refactoring as part of the review process
- If the manager is schedule driven (or is a PHB), consider a "don't ask don't tell" strategy

Problems With Refactoring

- Databases
  - Most business applications are tightly coupled to the database schema that supports them
  - You need to isolate changes to either the database or object model by creating a layer between the models
  - Such a layer adds complexity buts enhances flexibility
- Changing interfaces
  - Don't publish interfaces prematurely -- modify your code ownership policies to smooth refactoring
- Design changes that are difficult to refactor
- When shouldn't you refactor?
  - A clear sign that a rewrite is in order is when the code does not work

Refactoring and Design

- Upfront design
- XP advocates that you code the first approach that comes in to your head, get it working, and then refactor it into shape
- The point is that refactoring changes the role of upfront design
- You are no longer looking for the perfect solution when doing design but *a reasonable one*
- Refactoring can lead to simpler designs without sacrificing flexibility

Refactoring and Performance

- A common concern with refactoring is its effect on performance
- Refactoring will make software slower but it also makes the software more amenable to performance tuning
  - The secret to fast software , in all but hard real-time contexts, is to write tunable software first and then to tune it for sufficient speed
- Profiling ensures that you focus your tuning efforts in the right places

Where Did Refactoring Come From?

- Two of the first people to recognize the importance of refactoring were Kent Beck and Ward Cunningham
  - The worked with Smalltalk from the 80's onward
  - Refactoring has always been an important element in the Smalltalk community
- Ralph Johnson's work with refactoring and frameworks has been an important contribution
- Bill Opdyke focused his doctoral research on developing refactoring tools for use in frameworks
  - Outside of Fowler's book, Opdyke's doctoral thesis is the most substantial work in this area

Bad Smells in Code

1. Duplicated Code
2. Long Method
3. Large Class
4. Long Parameter List
5. Divergent Change
6. Shotgun Surgery
7. Feature Envy
8. Data Clumps
9. Primitive Obsession
10. Switch Statements
11. Parallel Inheritance Hierarchies
12. Lazy Class
13. Speculative Generality
14. Temporary Field
15. Message Chains
16. Middle Man
17. Inappropriate Intimacy
18. Alternative Classes with Different Interfaces
19. Incomplete Library Class
20. Data Class

Duplicated Code

- If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them
- The simplest duplicated code problem is when you have the same expression in two methods of the same class
  - Perform *Extract Method* and invoke the code from both places
- Another common duplication problem is having the same expression in two sibling subclasses
  - Perform *Extract Method* in both classes then *Pull Up Field*
- If you have duplicated code in two unrelated classes, consider using *Extract Class* in one class and then use the new component in the other

Long Method

- The longer a procedure is the more difficult it is to understand
- Nearly all of the time all you have to do to shorten a method is *Extract Method*
- If you try to use *Extract Method* and end up passing a lot of parameters, you can often use *Replace Temp with Query* to eliminate the temps and slim down the long list of parameters with *Introduce Parameter Object* and *Preserve Whole Object*

Large Class

- When a class is trying to do too much, it often shows up as too many instance variables. When a class has too many instance variables, duplicated code cannot be far behind
- A class with too much code is also a breeding ground for duplication
- In both cases *Extract Class* and *Extract Subclass* will work

Long Parameter List

- With objects you don't need to pass in everything the method needs, instead you pass in enough so the method can get to everything it needs
- This is goodness, because long parameter lists are hard to understand, because they inconsistent and difficult to use because you are forever changing them as you need more data
- Use *Replace Parameter with Method* when you can get the data in one parameter by making a request of an object you already know about

Divergent Change

- Divergent change occurs when one class is commonly changed in different ways for different reasons
  - If for example you have to change 4 different methods every time the database changes you might have a situation where two objects are better than one
- To clean this up you identify everything that changes for a particular cause and use *Extract Class* to put them all together

Shotgun Surgery

- This situation occurs when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes
- When the changes are all over the place they are hard to find, and its easy to miss an important change.
- You want to use *Move Method* and *Move Field* to put all the changes in a single class
- If no current class looks like a good candidate then create one
- Often you can use *Inline Class* to bring a whole bunch of behavior together

Feature Envy

- A classic smell is a method that seems more interested in a class other in the one that it is in
- The method clearly wants to be elsewhere, so use *Move Method* to get it there
- Sometimes only part of the method suffers from envy so in that case you can use *Extract Method* on the jealous bit and *Move Method* to get it home
- There are several patterns that break this rule, including:
  - Strategy (Gang of Four)
  - Visitor (Gang of Four)
  - Self Delegation (Beck)
- These patterns are used to combat the divergent change smell

Data Clumps

- Often you will see the same three or four data items together in lots of places:
  - Fields in a couple of classes
  - Parameters in many method signatures
- Bunches of data that hang around together really ought to be made into their own object
- The first step is to look for where the clumps appear as fields and use *Extract Class* to turn the clumps into an object
- For method parameters use *Introduce Parameter Object* or *Preserve Whole Object* to slim them down

Primitive Obsession

- Java has primitives for numbers, but strings and dates, which are primitives in many environment, are classes
- People new to objects are sometimes reluctant to use small objects for small tasks, such as money classes that combine numbers and currency ranges with an upper and lower, and special strings such as telephone numbers and ZIP codes
- You can use *Replace Data Value with Object* on individual data values
- If the data value has a type code, use *Replace Type Code with Class* if the value does not effect the behavior
- If you have conditional that depend on the type code use *Replace Type Code with Subclass* or *Replace Type Code with State/Strategy*

Switch Statements

- Most times when you see a switch statement you should consider polymorphism
- Use *Extract Method* to extract the switch statement and then *Move Method* to get it into the class where the polymorphism is needed
- If you only have a few case that effect a single method then polymorphism is overkill. In this case *Replace Parameter with Explicit Methods* is a good option
- If one of your conditional cases is null, try *Introduce Null Object*

Parallel Inheritance Hierarchies

- Is really a special case of shotgun surgery
- In this case every time you make a subclass of one class, you have to make a subclass of another
- The general strategy for eliminating the duplication is to make sure that instances of one hierarchy refer to instance of another
- If you use *Move Method* and *Move Field*, the hierarchy on the referring class disappears

Lazy Class

- Each class you create costs money and time to maintain and understand
- A class that is not carrying its weight should be eliminated
- If you have subclasses that are not doing enough try to use *Collapse Hierarchy* and nearly useless components should be subjected to *Inline Class*

Speculative Generality

- You get this smell when someone says "I think we need the ability to do this someday" and you need all sorts of hooks and special cases to handle things that are not required
- This smell is easily detected when the only users of a class or method are test cases
- If you have abstract classes that are not doing enough then use *Collapse Hierarchy*
- Unnecessary delegation can be removed with *Inline Class*
- Methods with unused parameters should be subject to *Remove Parameter*
- Methods named with odd abstract names should be repaired with *Rename Method*

Temporary Field

- Sometimes you will see an object in which an instance variable is set only in certain circumstances
- This can make the code difficult to understand because we usually expect an object to use all of its variables
- Use *Extract Class* to create a home for these orphan variables by putting all the code that uses the variable into the component
- You can also eliminate conditional code by using *Introduce Null Object* to create an alternative component for when the variables are not valid

Message Chains

- Message chains occur when you see a client that asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another object, etc.
- This smell may appear as a long line of getThis methods, or as a sequence of temps
- Navigating this way means the client is structured to the structure of the navigation
- The move to use in this case is *Hide Delegate* at various points in the chain

Middle Man

- One the major features of Objects is encapsulation

Encapsulation often comes with delegation,Sometimes delegation can go to far

- For example if you find half the methods are delegated to another class it might be time to use *Remove Middle Man* and talk to the object that really knows what is going on
- If only a few methods are not doing much, use *Inline Method* to inline them into the caller
- If there is additional behavior, you can use *Replace Delegation with Inheritance* to turn the middle man into a subclass of the real object

Inappropriate Intimacy

- Sometimes classes become far too intimate and spend too much time in each other's private parts
- Use *Move Method* and *Move Field* to separate the pieces to reduce the intimacy
- If the classes do have common interests, use *Extract Class* to put the commonality in a safe place or use *Hide Delegate* to let another class act as a go-between

Alternative Classes with Different Interfaces

- Most of the time you can use *Rename Method* on any methods that do the same thing but have different signatures for what they do
- If this does not go far enough that means the classes are not yet doing enough so keep using *Move Method* to move behavior to other classes until the protocols are the same
- If you have to redundantly move code to accomplish this, you may be able to use *Extract Superclass*

Incomplete Library Class

- Library builders have a tough job
- The problem is that if the library is insufficient for your needs it is usually impossible to modify a library class to do something that you would like it to do
- If there are just a couple of methods that you wish the library class had, use *Introduce Foreign Method*
- If there is more extra behavior you need, use *Introduce Local Extension*

Data Class

- These are classes that have fields, getting and setting methods, and nothing else
- Such methods are dumb data holders and are manipulated in far too much detail by other classes
- If in a previous life the classes were public fields, apply *Encapsulate Field*
- If you have collection fields, check to see if they are properly encapsulated and apply *Encapsulate Collection* if they are not
- Use *Remove Setting Method* on any field that should not be changed
- Look for where these getters and setters are used by other classes and try to use *Move Method* to move behavior into the data class
- If you can't move a whole method, use *Extract Method* to create a method that can be moved

References

http://sourcemaking.com/refactoring

Refactoring Examples

```
Class ff
   float ConvertToTL (float amount, String Curr)
       { float x, y ;

               if (Curr = ‘Euro’)
                           {
                                 x= 3.01;
                                 y = 1000;
                            }

                 if (Curr = ‘USD’)
                           {
                                 x= 2.11;
                                 y = 2000;
                            }


 if   (amount  >=y)
                 if  (Currency = ‘Euro’)
                    x = x – 0.2;
                else
                    x = x – 0.1;

  Return (x* amount);
}
------------------
```

Client

```
  Print (ff.ConvertToTL(1000, “USD”));
```

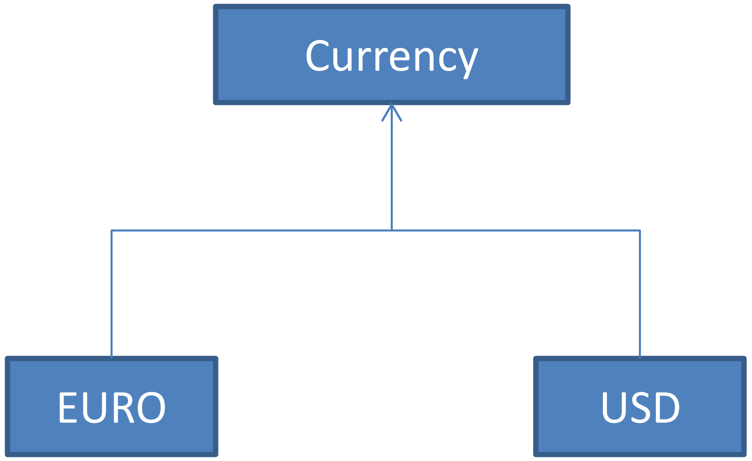
Class ff

```
                                getEuroRate();
                                getEuroLimit();
                                GetUSDRate()
                                getUSDlimit()
                               getEuroDiscount();
                              getUSDDiscount

  float ConvertToTL (float amount, String Curr)
     { float x, y ;

             if (Curr = 'Euro')
                          {
                               x= getEuroRate();
                                y=getEuroLimit();

                           }

               if (Curr = 'USD')
                          {
                              x=  GetUSDRate()
                                y= getUSDlimit()
                           }


 if   (amount  >=y)
               if  (Currency = 'Euro')
                  x = x – getEuroDiscount();
              else
                  x = x – getUSDDiscount;

 Return (x* amount);
}
------------------
```

Client

Print (ff.ConvertToTL(1000, "USD"));

```
Class Currency{
String name;
 float  ExchangeRate;
float   Limit;
Float Discount;
-------------------------
---Gets, and sets

ConverttoTL()
}
```

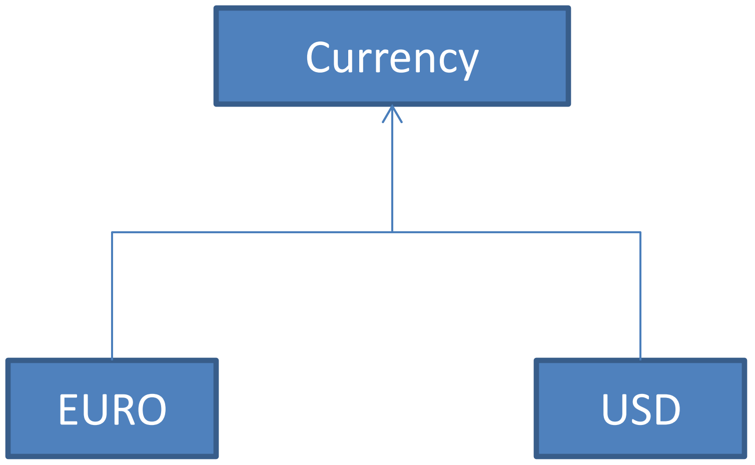


```
Class ff
   float ConvertToTL (float amount, Currency Curr)
       { float x ;

                 x= Curr.Get-ExchangeRate();
                if   (amount  >= Curr.Get-Limit() )   x= x -  Get-Discount();

Return (x* amount);
}
------------------
```

Client

Print (Currency**.ConvertToTL(1000, new USD()));**

```
Class Currency{
String name;
 float  ExchangeRate;
float   Limit;
Float Discount;
-------------------------
---Gets, and sets

Boolean DiscountedAmount()

ConverttoTL()
}
```

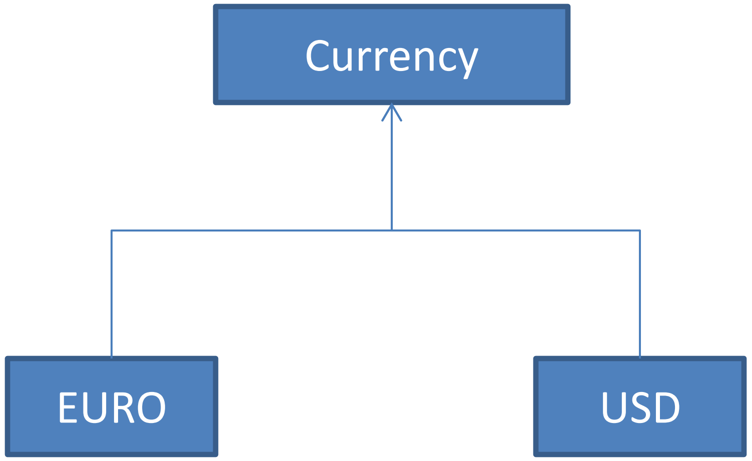


```
Class ff
   float ConvertToTL (float amount, Currency Curr)
       { float x ;


                x= Curr.Get-ExchangeRate();
               if   (DiscountedAmount(amount))   x= x -  Get-Discount();
                Return (x* amount);
}
------------------
```

Client

  Print (Currency**.ConvertToTL(1000, new USD()));**

```
Class Currency{
String name;
 float  ExchangeRate;
float   Limit;
Float Discount;
-------------------------
---Gets, and sets

Boolean DiscountedAmount()
Float  GetDiscountedRate();
Float ComputeExchangeRate(amount)

ConverttoTL()
}
```



```
Class ff
   float ConvertToTL (float amount, Currency Curr)
       {
               return( amount *ComputeExchangeRate(amount))
        }
------------------
```

Client

```
  Print (Currency.ConvertToTL(1000,  new USD()));
```

```
ComputeExchangeRate(amount)
{
  if  (DiscountedAmount(amount))
         GetDiscountedRate()
   Else
        Get-ExchangeRate();


}
```