REFACTORING

Recommended reading:

Code Complete by Steve McConnell

Refactoring by Fowler et al.

Sources: https://sourcemaking.com/, Based on slides by Professor Christine Julien and Dr. Marty Stepp

What is Refactoring?

- Semantic-preserving program transformations
  - A change made to the internal structure of a program without modifying its observable behavior to make it
    - Easier to understand
    - Cheaper to modify
- **Refactoring patterns**
  - “Improving the design after the code has been written”
    - Seems a bit odd since we usually design first then code
    - Refactoring usually entails small changes with large cumulative effects

Why Refactor?

- Problem: "Bit rot"
  - After new versions, modifications, new features: design decays
  - Code's structure must evolve, even if code was initially well-designed
- Why fix a part of your system that isn't broken?
- Your code has three purposes:
  - Execute its functionality
  - Allow change
  - Understandability/readability
- If code doesn't do all three, it's broken → refactor

Bad Code Smells and Refactoring

- **Code smells**
  - Indicative of bad software design
  - List of bad smells: http://blog.codinghorror.com/code-smells/
  - Useful “catalog” of refactorings: http://www.refactoring.com/catalog/
  - Mapping of smells to refactorings: http://www.industriallogic.com/wp-content/uploads/2005/09/smellstorefactorings.pdf

Refactorings

- The Fowler book is basically a catalog of common refactoring patterns
  - Each includes a name, summary, motivation, mechanics, and examples
- Not formal (they can't be, since determining program equivalence is undecidable)
- Similar in nature to design patterns
  - Defining a shared vocabulary

EXAMPLES OF CODE SMELLS

And associated refactorings

Code Smells (Fowler)

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
  - change one place → must change others
- Feature Envy
- Primitive Obsession
- Switch Statements
- Lazy Class
- Speculative Generality
- Temporary Field
- Message Chains
- Inappropriate Intimacy
- Refused Bequest
  - subclass doesn't use inherited members much
- Data Clumps
- Comments

Refactorings

- **Composing Methods**
  - Extract Method
  - Inline Method
  - Replace Temp with Query
- **Moving Features between Objects**
  - Move Method
  - Move Field
  - Extract Class
  - Hide Delegate
  - Remove Middle Man
  - Inline Class
- **Organizing Data**
  - Replace Data Value with Object
  - Change Unidirectional Association to Bidirectional
  - Encapsulate Collection
  - Replace Type Code with Class
  - Replace Type Code with State/Strategy
  - Replace Data Value with Object
  - Replace Magic Number with Symbolic Constant
- **Simplifying Conditional Expressions**
  - Replace Conditional with Polymorphism
  - Introduce Null Object
- **Simplifying Method Calls**
  - Rename Method
  - Remove Parameter
  - Introduce Parameter Object
  - Replace Constructor with Factory Method
  - Preserve Whole Object
- **Dealing with Generalization**
  - Pull Up Field
  - Pull Up Method
  - Push Down Method
  - Push Down Field
  - Extract Subclass; Extract Superclass
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation
  - Replace Delegation with Inheritance

IDE Support for Refactoring

- Eclipse and IntelliJ
  - variable/method/class renaming
  - method extraction
  - extract redundant code snippets
  - change method signature
  - method inlining
  - ...

https://www.jetbrains.com/help/idea/refactoring-source-code.html

Refactoring Flow

- Write unit tests that verify code's external correctness
  - Should pass on current poorly designed code
  - Helps ensure that refactor doesn't cause regression
- Determine if code should be refactored
  - Tests are passing, no impending deadlines
- Refactor the code
  - Do unit tests break? Fix bugs.
  - Perform integration testing. Fix issues.
- Code review the changes
- Check in refactored code
  - Each refactoring should be small – refactor one issue at a time
  - Check in should only contain refactor – no added features

Style Smells

- Comments
  - Often used to deodorize other smells
  - Not necessarily bad – copious comments can indicate bad code
- Naming
  - Avoid placing types in method or variable names (if you change the type, you'll have to change the name)
  - Names of methods/variables should succinctly describe what the purpose is
  - Pick naming standard and stick with it,
    make sure that analog functions have analog names
    (e.g., if you can open() you ought to be able to close())
- Dead code
  - Delete it. Use version control.

#1: Duplicated Code

- You've done this before
- You know it's bad
- Explicit and subtle duplication
  - E.g., identical code (explicit) vs. structures or processing steps that appear different but are essentially the same (subtle)
- Potential useful refactorings:
  - **Extract method**, **Extract class**, **Template method pattern**, **Strategy pattern**

Extract Method

- Applies when you have a code fragment inside some code block where the lines of code should always be grouped together

✓ *Turn the fragment into a method whose name explains the purpose of the block of code*

Extract Method Refactoring Example

```
void printOwing() {
  printBanner();

  //print details
  System.out.println ("name:  " + _name);
  System.out.println ("amount " + getOutstanding());
}
```

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails (double outstanding) {
  System.out.println ("name:  " + _name);
  System.out.println ("amount " + outstanding);
}
```

Extract Class

- You have one class doing work that should be done by two different classes

✓ *Create a new class and move the relevant fields and methods from the old class to the new class*

Extract Class Example

Template Method Pattern

- A Template Method describes the skeleton behavior of a method
  - Defers some substeps to subclasses
- By defining the “primitive operations” comprising the template method, the subclasses provide different behaviors

Template Method Example

- Barista recipes for preparing beverages

Coffee Recipe
1. Boil water
2. Brew coffee in boiling water
3. Pour coffee in cup
4. Add sugar and milk

Tea Recipe
1. Boil some water
2. Steep tea in boiling water
3. Pour tea in cup
4. Add lemon

Template Method Example

```
public class Coffee {
   void prepareRecipe() {
        boil Water();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
   }

   public void boilWater() {
        SOPln("Boiling water");
   }

   public void brewCoffeeGrinds(){
        SOPln("Drip coffee...");
   }
...
```

```
public class Tea {
   void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
   }

   public void boilWater() {
        SOPln("Boiling water");
   }

   public void steepTeaBag() {
        SOPln("Steeping the tea...");
   }
...
```

Template Method Example

- Code duplication in these classes:
  - boilWater(), pourInCup() are same
  - Both recipes follow same outline
- Get rid of duplication

Template Method Example: Abstract Parent Class

Template method is final

Primitive operations implemented in subclasses, Coffee and Tea

```
1  public abstract class CaffeineBeverage {
2
3      final void prepareRecipe() {
4          boilWater();
5          brew();
6          pourInCup();
7          addCondiments();
8      }
9
10     abstract void brew();
11
12     abstract void addCondiments();
13
14     void boilWater() {
15         System.out.println("Boiling water");
16     }
17
18     void pourInCup() {
19         System.out.println("Pouring into cup");
20     }
21 }
22
```

Template Method Example: Subclasses

```
1  public class Coffee extends CaffeineBeverage {
2      public void brew() {
3          System.out.println("Dripping Coffee through filter");
4      }
5      public void addCondiments() {
6          System.out.println("Adding Sugar and Milk");
7      }
8  }
9
10 public class Tea extends CaffeineBeverage {
11     public void brew() {
12         System.out.println("Steeping the tea");
13     }
14     public void addCondiments() {
15         System.out.println("Adding Lemon");
16     }
17 }
18
```

Template Method Example: Recap

- Original classes had similar algorithms
- Eliminated duplication by introducing parent class
- Made algorithm steps more abstract, specified algorithm structure in parent
  - **Eliminated "implicit" duplication in classes**
- Subclasses implemented the primitive operations, which were abstract in parent class, in appropriate way for them
- **Benefits:**
  - Parent class contains algorithm structure and protects it
  - Parent class shares common code with subclasses
  - Change in algorithm likely only affects parent class
  - New caffeinated beverages can easily be plugged in
  - Parent centralizes algorithm – subclasses plug in missing pieces

In Class Exercise/Quiz

Template Method Pattern

- Code is currently in Customer class
- Other classes in software: Rental, Movie

Move code to a Statement hierarchy:

- abstract class Statement with two subclasses, HTMLStatement and TextStatement
- template method value()
- hook methods produce header, footer

#2: Long Method

- Methods with many statements, loops, variables
- Short methods help explain code
  - Easier to read, understand, maintain, debug
- Potential useful refactorings:
  - **Extract method** (vast majority of the time)

#3: Large Class

- Class may be small initially but gets bloated as program grows
- Too many instance variables, methods, lines
  - A class is trying to do too much – too many responsibilities
- Potential refactorings
  - **Extract class**, **Extract subclass**
  - **Observer**
    - Common for GUIs

#4: Long Parameter List

- Hard to understand long lists of parameters
- Consider which parameters are essential
  - Leave the rest to the object to track down as necessary
- Potential refactorings:
  - **Replace parameter with method call**, **Introduce parameter object**, **Preserve whole object**

Replace Parameter with Method

- An object invokes a method then passes the result as a parameter for a method
  - The receiver can also invoke this method

✓ *Why the indirection? Remove the parameter and let the receiver invoke the method.*

Example:
Replace Parameter with Method Call

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice, discountLevel);
```

```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice (basePrice);
```

Introduce Parameter Object

- Group of parameters encountered in multiple methods

✓ *Replace them with a single object*

Introduce Parameter Object Example

Preserve Whole Object

- You get a bunch of values from an object but then pass those values together to another method call

✓ *Maybe you should just pass the whole object instead.*

Preserve Whole Object Example

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

```
withinPlan = plan.withinRange(daysTempRange());
```

#5: Divergent Change

- A class is changed in different ways for different reasons
  - Separating divergent responsibilities decreases the chance that one change negatively affects a different function
  - E.g., in `class X`, change `mA()`, `mB()`, and `mC()` every time we add a new database; change `mD()`, `mE()`, and `mF()` every time we add a new financial instrument
- Potential refactoring:
  - **Extract class**

#6: Shotgun Surgery

- Opposite of divergent change
- One change requires altering many classes
  - Easy to miss an important change
- Potential refactorings:
  - **Move method**, **Move field**, **Inline class**

Move Method

- A method is used more by another class, or uses more code in another class, than its own class.

✓ *Well, then, move it. Create a new method with a similar body in the class it uses most. Turn the old method into a simple delegation or remove it altogether.*

Move Method Example

Move Method (another) Example

```
class Project {
  Person[] participants;
}

class Person {
  int id;
  boolean participate(Project p) {
    for(int i=0; i<p.participants.length; i++) {
          if (p.participants[i].id == id) return(true);
    }
    return(false);
  }
}

... if (x.participate(p)) ...
```

```
class Project {
  Person[] participants;
  boolean participate(Person x) {
    for(int i=0; i<participants.length; i++) {
          if (participants[i].id == x.id) return(true);
    }
    return(false);
  }
}

class Person {
  int id;
}

... if (p.participate(x)) ...
```

#7: Feature Envy

- A method in a class seems more interested in some other class's internals than its own
  - The most common target of the envy is data
  - E.g., a class repeatedly calls getter and setter methods on some other class
- Potential refactorings:
  - **Extract method**, **Move method**, **Move field**

#8: Data Clumps

- Bunches of data that hang around together should be made into their own object
  - Fields in several classes, parameters that are always chained together, etc.
  - Ask yourself the question: are the others sensible when one is removed?
- Potential refactorings:
  - **Extract class**, **Preserve whole object**, **Introduce parameter object**

#9: Primitive Obsession

- Using primitives instead of small objects for simple things, e.g., money, intervals, special strings for phone numbers
- Result: over-emphasis on primitives (strings, arrays, integers, etc.)
- Classes generally provide a simpler and more natural way to directly model things than primitives do
  - Higher level abstractions clarify code
- Potential refactorings:
  - **Replace data value(s) with object**, **Replace type code with class**, **Replace type code with state/strategy**

Replace Data Value with Object

- You have a data item that needs additional data or behavior
  - Really, try not to start with primitives and add more and more primitives that are conceptually (but not concretely) linked

✓ *Instead, turn the data item into an object*

Replace Data Value with Object Example

Replace Type Code with Class

- A class has a (numeric) type code that does not affect its behavior

✓ *Replace the number with a new class*

Replace Type Code with Class Example

Replace Type with Class: Code Examples

```
class Person {
  public static final int O = 0;
  public static final int A = 1;
  public static final int B = 2;
  public static final int AB = 3;

  private int _bloodGroup;

  public Person (int bloodGroup) {
      _bloodGroup = bloodGroup;
  }

  public void setBloodGroup(int arg) {
      _bloodGroup = arg;
  }

  public int getBloodGroup() {
      return _bloodGroup;
  }
}
```

Replace Type with Class: Code Examples (cont.)

```
class BloodGroup {
  public static final BloodGroup O = new BloodGroup(0);
  public static final BloodGroup A = new BloodGroup(1);
  public static final BloodGroup B = new BloodGroup(2);
  public static final BloodGroup AB = new BloodGroup(3);
  private static final BloodGroup[] _values = {O, A, B, AB};

  private final int _code;

  private BloodGroup (int code ) {
      _code = code;
  }

  public int getCode() {
      return _code;
  }

  public static BloodGroup code(int arg) {
      return _values[arg];
  }
}
```

Replace Type with Class: Code Examples (cont.)

```
class Person {
    public static final int O = BloodGroup.O.getCode();
    public static final int A = BloodGroup.A.getCode();
    public static final int B = BloodGroup.B.getCode();
    public static final int AB = BloodGroup.AB.getCode();

    private BloodGroup _bloodGroup;

    public Person (int bloodGroup) {
        _bloodGroup = BloodGroup.code(bloodGroup);
    }

    public int getBloodGroup() {
        return _bloodGroup.getCode();
    }

    public void setBloodGroup(int arg) {
        _bloodGroup = BloodGroup.code (arg);
    }
  }
```

Replace Type with Class: Code Examples (cont.)

```
class Person {
    ...

    pub

    }

    pub

    }
}
```

```
class Person {
    ...

    public Person (int bloodGroup) {

    }

    publ

    }
}
```

```
class Person {
    ...

    public void setBloodGroup(int arg) {
        _bloodGroup = BloodGroup.code (arg);
    }

    public void setBloodGroup(BloodGroup arg) {
        _bloodGroup = arg;
    }
}
```

Replace Type with Class: Code Examples (cont.)

```
Person thePerson = new Person(BloodGroup.A);
```

```
class Person ...
  public static final int O = BloodGroup.O.getCode();
  public static final int A = BloodGroup.A.getCode();
  public static final int B = BloodGroup.B.getCode();
  public static final int AB = BloodGroup.AB.getCode();
  public Person (int bloodGroup) {
      _bloodGroup = BloodGroup.code(bloodGroup);
  }
  public int getBloodGroup() {
      return _bloodGroup.getCode();
  }
  public void setBloodGroup(int arg) {
      _bloodGroup = BloodGroup.code (arg);
  }
```

Replace Type Code with State/Strategy

- You have a type code that affects the behavior of the class, but you cannot use subclassing

✓ *Replace the type code with a state/strategy object*

Replace Type Code with State/Strategy Example

Replace Type Code with State/Strategy Example

```
class Employee {
    private EmployeeType type;
    private float salary;
    private float commission;
    ...
    public void setEmployeeType(EmployeeType type) {
        this.type = type;
    }
    public float salary(
        return salary;
    }
    ...
    public float pay() {
        return type.pay(
    }
}
```

```
class Engineer extends EmployeeType {
    float pay(Employee employee) {
        return employee.salary();
    }
}

class Salesman extends EmployeeType {
    float pay(Employee employee) {
        return employee.salary() +
               employee.commission();
    }
}
```

Replace Type Code with State/Strategy Example

```
class Employee {
    private EmployeeType type;
    private float salary;
    private float commission;
    ...
    public void setEmployeeType(EmployeeType type) {
        this.type = type;
    }
    public float salary(
        return salary;
    }
    ...
    public float pay() {
        return type.pay(
    }
}
```

```
enum EmployeeType {
    ENGINEER {
        float pay(Employee employee) {
            return employee.salary();
        }
    },
    SALESMAN {
        float pay(Employee employee) {
            return employee.salary() +
                   employee.commission();
        }
    };
    abstract float pay(Employee employee);
}
```

In Class Exercise

- See handout

#10: Switch statements

- Ugh.
- Switch statements often end up duplicated across the system
- Indicate lack of OO style and underuse of polymorphism
- Special case: a conditional that chooses different behavior based on the type of an object
- Potential refactorings:
  - **Extract method**, **Move method**, **Replace type code with subclasses**, **Replace type code with state/strategy**, **Replace conditional with polymorphism**

Replace Conditional with Polymorphism

- You have a condition that chooses different behavior depending on the type of object

✓ *Move each leg of the conditional to an overriding method in a subclass*

- Make the original method abstract (why?)
  - Otherwise, you're introducing an instance of the **Refused Bequest** smell… coming up

Replace Conditional with Polymorphism Example

```
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 ("Shou
}
```

#11: Lazy Class

- A class doesn't do enough to justify its maintenance
  - Lazy class can result from downsizing or adding things speculatively
- Potential refactorings:
  - **Collapse hierarchy**, **Inline class**

#12: Speculative Generality

- Unused class, method, field, parameter
  - E.g., “we might need to a method to do X some day”
  - Code created "just in case" some future feature requires it
- Potential refactorings:
  - **Collapse hierarchy**, **Remove parameter**, **Inline class**

#13: Temporary Field

- Including an instance variable that is only set in some circumstances
- The rest of the time, the field is empty or (worse) contains irrelevant data
  - This hampers understandability and can lead to accidental errors based on context
- Potential refactorings
  - **Extract class**, **Introduce null object**

Introduce Null Object

- You have repeated checks for a null values
  - Ugh. It's ugly and hard to read.

✓ *So replace the null value with a null object!*

Introduce Null Object Example

```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```

Introduce Null Object

```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```

```
class NullCustomer extends Customer {
   boolean isNull() {return true;}
   Plan getPlan() {
      return new NullPlan();
   }

   ...
}
```

```
customer = (order.customer != null) ?
   order.customer : new NullCustomer();
plan = customer.getPlan();
```

#14: Message Chains

- Occur when you see a long line of method calls or temporary variables to get some data
  - E.g., long string of `getThis().getThat().getSomething()`
- Client dependent on class structure
- Potential refactorings:
  - **Hide delegate**, **Extract method**, **Move method**

Hide Delegate

- A client is calling a delegate class of an object
  - getB().getSomethingFromB()

✓ *Create methods on the server to hide the delegate*

Hide Delegate Example

#15: Middle Man

- Delegation is good, and that's why we make objects
- But sometimes, all an object is doing is passing along calls to another object
  - For no apparent reason (e.g., an Adapter would be an exception)
- There is a fine line between information hiding and delegation overhead.
- Potential refactorings:
  - **Remove middle man** (duh!), **Inline method**, **Replace delegation with inheritance**

Remove Middleman

- A class is doing too much simple delegation

✓ *Get the client to call the delegate directly*

- This is the exact dual for **Hide Delegate**

Remove Middleman Example

Inline Method

- A method's body is just as clear as its name

✓ *So put the method's body into the body of its callers and remove the method*

Inline Method Example

```
int getRating() {
  return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
  return _numberOfLateDeliveries > 5;
}
```

```
int getRating() {
  return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

#16: Inappropriate Intimacy

- Classes sometimes end up delving too much into each others' private methods and fields
- Related: *Data Class* – classes that have fields and getters and setters but nothing else
  - Almost assuredly being manipulated in far too much detail by others
- Potential refactorings:
  - **Move method**, **Move field**, **Change bidirectional association to unidirectional association**, **Extract class** (if classes do in fact have common interests), **Hide delegate** (allow another class to act as a go-between), **Encapsulate collection** (for Data Class)

Change Bidirectional Association to Unidirectional

- You have a two-way association but one class no longer needs access to the other

✓ *So drop the unneeded end of the association*

Change Bidirectional Association to Unidirectional

Encapsulate Collection

- A method returns a collection
  - This can be confusing because it may seem to the caller that he can make changes to the collection

✓ *Make it return a read-only view and provide add/remove methods*

Encapsulate Collection Example

#17: Alternative Classes with Different Interfaces

- Classes can be completely different on the outside but end up being the same internally
- Basically, you should find the similarities in the two classes, the refactor them to share a common interface
- Potential refactorings:
  - **Extract superclass**

#18: Refused Bequest

- Happens when you inherit code you don't want
  - i.e., a child class uses very little of the functionality of some parent (base) class
- The worst (strongest smell) here is when the child reimplements the behavior from the parent class
- Potential refactorings:
  - **Push down field**, **Push down method**, **Replace inheritance with delegation**

Replace Inheritance with Delegation

- A subclass uses only part of a superclass's interface or does not want to inherit data

✓ *Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing*

Replace Inheritance with Delegation Example

QUESTIONS?