Refactoring

What is Refactoring?

- Refactoring is the process of changing a software system such that
  - the external behavior of the system does not change
    - e.g., functional requirements are maintained
  - but the internal structure of the system is improved
- This is sometimes called
  - “Improving the design after it has been written”
- Refactoring formalizes good programming practices

Simple Examples

- Consolidate duplicate conditional fragments

Before

```
if (isSpecialDeal()) {
     total = price * 0.95;
     send();
} else {
     total = price * 0.98;
     send();
}
```

After

```
if (isSpecialDeal()) {
     total = price * 0.95;
} else {
     total = price * 0.98;
}
send();
```

Simple Examples

- Replace magic number with symbolic constant

Before

```
double potentialEnergy(double mass, double height){
    return mass * 9.81 * height;
}
```

After

```
static final double GRAVITATIONAL_CONSTANT = 9.81;
double potentialEnergy(double mass, double height) {
    return mass * GRAVITATIONAL_CONSTANT * height;
}
```

But, Refactoring is Dangerous!

- Although refactoring helps to reduce bugs, it can also introduce new bugs into the code
- Manager’s point of view
  - if my programmers spend time “cleaning up the code” then that’s less time implementing required functionality (and my schedule is slipping as it is!)
- To address these concerns
  - refactoring needs to be systematic, incremental, and safe

Refactoring is Useful Too

- The idea behind refactoring is to
  - acknowledge that it will be challenging to get a design right the first time and, as a program's requirements change, the design may need to change
  - refactoring provides techniques for evolving the design in small incremental steps
- Benefits
  - often, code size is reduced after refactoring
  - confusing structures are transformed into simpler structures
    - which are easier to maintain and understand

A “Cookbook” can be Useful

- Refactoring: Improving the Design of Existing Code
  - by Martin Fowler (and Kent Beck, John Brant, William Opdyke, and Don Roberts)
- Similar to the Gang of Four’s Design Patterns
  - provides “refactoring patterns”
- Also
  - http://www.refactoring.com/catalog
  - http://sourcemaking.com/refactoring

Principles in Refactoring

- Fowler’s definition
- 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
- Refactoring (verb)
  - to restructure software by applying a series of refactoring without changing its observable behavior

Principles, continued

- The purpose of refactoring is
  - to make software easier to understand and modify
  - no functionality is added, but the code is cleaned up, made easier to understand and modify, and sometimes is reduced in size
- Contrast this with performance optimization
  - functionality is not changed; only internal structure
  - however, performance optimizations often involve making code harder to understand (but faster!)

Principles, continued

- How do you make refactoring safe?
  - First, use refactoring “patterns”
    - Fowler assigns “names” to refactoring in the same way that the GoF’s book assigned names to patterns
  - Second, test constantly!
    - this ties into the agile design paradigm
    - you write tests before you write the code
    - after you refactor, you run the tests and check that they all pass
    - if a test fails, the refactoring broke something, but you know about it right away and can fix the problem before you move on

Why Should you Refactor?

- Refactoring improves the design of software
  - without refactoring, a design will “decay” as people make changes to a software system
- Refactoring makes software easier to understand
  - because structure is improved, duplicated code is eliminated, etc.
- Refactoring helps you find bugs
  - Refactoring promotes a deep understanding of the code at hand, and this understanding aids the programmer in finding bugs and anticipating potential bugs
- Refactoring helps you program faster
  - because a good design enables progress

When Should you Refactor?

- The Rule of Three
  - Three “strikes” and you refactor
- Refactor when you add functionality
  - do it before you add the new function to make it easier to add the function
  - or do it after to clean up the code after the function is added
- Refactor when you need to fix a bug
- Refactor as you do a code review

Problems with Refactoring

- Databases
  - Business applications are often tightly coupled to underlying databases
  - code is easy to change; databases are not
- Changing interfaces
  - Some refactoring requires that interfaces be changed
  - if you own all the calling code, no problem
  - if not, the interface is “published” and can’t change

Refactoring: Where to Start?

- How do you identify code that needs to be refactored?
- Look for “Bad Smells” in code
  - A chapter in Fowler’s book
  - Several online sources
    - http://sourcemaking.com/refactoring/bad-smells-in-code
- They present examples of “bad smells” and then suggest refactoring techniques to apply
- Tools such as Codegrip, Checkstyle, PMD, FindBugs, and SonarQube can automatically identify code smells

Bad Smells in Code

- **Duplicated code**
  - Bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!
- **Long method**
  - Long methods are more difficult to understand
  - performance concerns with respect to short methods are largely obsolete
- **Large Class**
  - Large classes try to do too much, which reduces cohesion
- **Long Parameter List**
  - Hard to understand, can become inconsistent if the same parameter chain is being passed from method to method

Duplicated Code

Duplicated Code

```
public class CustomerNameChanger
{
    public void ChangeName(CustomerDbContext context, int customerId, string name)
    {
        var customer = context.Customer.SingleOrDefault(x => x.CustomerId == customerId);
        if(customer == null)
            throw new Exception(string.Format("Customer {0} was not found.", customerId));

        customer.Name = name;
    }
}

public class CustomerAddressChanger
{
    public void ChangeAddress(CustomerDbContext context, int customerId, string address,
        string postalCode, string city)
    {
        var customer = context.Customer.SingleOrDefault(x => x.CustomerId == customerId);
        if(customer == null)
            throw new Exception(string.Format("Customer {0} was not found.", customerId));

        customer.Address = address;
        customer.PostalCode = postalCode;
        customer.City = city;
    }
}
```

Bad Smells in Code

- **Divergent Change**
  - Symptom: one type of change requires changing one subset of methods; another type of change requires changing another subset
  - e.g., “I have to change these three methods every time I get a new database.”
  - Related to cohesion
- **Shotgun Surgery**
  - a change requires lots of little changes in a lot of different classes
- **Feature Envy**
  - a method requires lots of information from some other class
  - Move it closer!
- **Data Clumps**
  - attributes that clump together (are used together) but are not part of the same class

Shotgun Surgery

```
public void debit(int debit) throws Exception
{
        if(amount <= 500)
        {
                throw new Exception("Mininum balance shuold be over 500");
        }

        amount = amount-debit;
        System.out.println("Now amount is" + amount);

}

public void transfer(Account from,Account to,int cerditAmount) throws Exception
{
        if(from.amount <= 500)
        {
                throw new Exception("Mininum balance shuold be over 500");
        }

        to.amount = amount+cerditAmount;

}
```

Bad Smells in Code

- **Primitive Obsession**
  - characterized by a reluctance to use classes instead of primitive data types
- **Switch Statements**
  - Switch statements are often duplicated in code; they can typically be replaced by the use of polymorphism (let OO do your selection for you!)
- **Parallel Inheritance Hierarchies**
  - Similar to shotgun surgery; each time I add a subclass to one hierarchy, I need to do it for all related hierarchies
  - Some design patterns encourage the use of parallel inheritance hierarchies (so they are not always bad!)

Bad Smells in Code

- **Lazy Class**
  - A class that no longer “pays its way”
  - e.g., maybe a class that was downsized by a previous refactoring, or represented planned functionality that did not pan out
- **Speculative Generality**
  - “Oh, I think we need the ability to do this kind of thing someday”
  - thus have all sorts of hooks and special cases to handle things that aren’t required
- **Temporary Field**
  - An attribute of an object is only set/used in certain circumstances;

Bad Smells in Code

- **Message Chains**
  - a client asks an object for another object and then asks that object for another object etc.
  - The client depends on the structure of the navigation
  - any change to the intermediate relationships requires a change to the client
- **Middle Man**
  - If a class is delegating more than half its responsibilities to another class, do you really need it? Involves trade-offs; some design patterns encourage this (e.g., Decorator)
- **Inappropriate Intimacy**
  - Pairs of classes that know too much about each other's implementation details (loss of encapsulation)

Bad Smells in Code

- **Data Class (information holder)**
  - These are classes that have fields, getting and setting methods for the fields, and nothing else; they are data holders, but objects should be about data and behavior
- **Refused Bequest**
  - A subclass ignores most of the functionality provided by its superclass
  - Subclass may not pass the “IS-A” test
- **Comments**
  - Comments are sometimes used to hide bad code
  - “…comments are often used as a deodorant”

Refused Bequest

```
class Government {
    protected double computeBaseTax() { //... }

    protected double addPersonalTax(double tax) { //... }

    public double getTax() {
        double tax = computeBaseTax();
        return addPersonalTax(tax);
    }
}
```

```
class Company extends Government {
    private double computeInitialTax() { //... }

    @Override
    public double getTax() {
        double tax = computeInitialTax();
        return addPersonalTax(tax);
    }
}
```

The Catalog of Refactoring Patterns

- Large list of refactoring patterns (http://www.refactoring.com/catalog)
  - Extract Method
  - Extract Variable
  - Extract Class
  - Replace Temp with Query
  - Move Method
  - Replace Conditional with Polymorphism
  - Introduce Null Object
  - Separate Query for Modifier
  - Introduce Parameter Object
  - Encapsulate Collection
  - Replace Nested Conditional with Guard Clauses

Extract Method

- You have a code fragment that can be grouped together
- Turn the fragment into a method whose name explains the purpose of the fragment

```
void printOwing(double amount) {
    printBanner()
    //print details
    System.out.println("name: " + _name);
    System.out.println("amount: " + amount);
}

==============================================

void printOwing(double amount) {
    printBanner()
    printDetails(amount)
}

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

Extract Variable

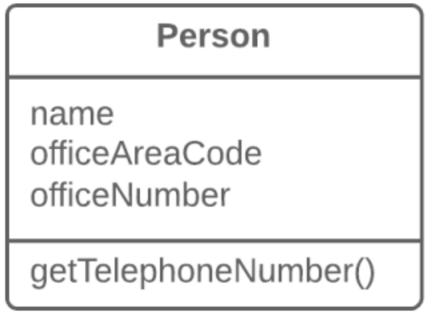
- You have an expression that's hard to understand
- Place the result of the expression or its parts in separate variables that are self-explanatory

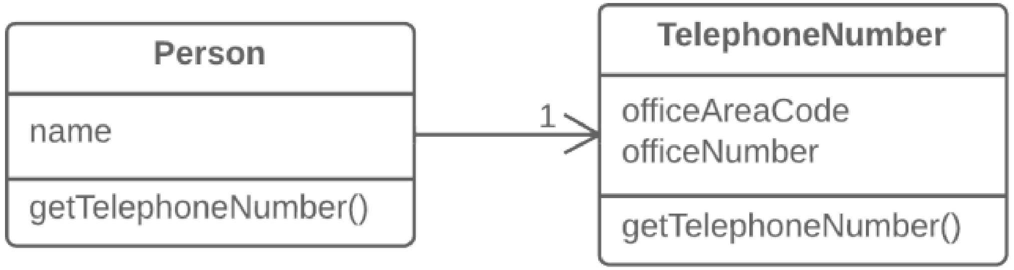
```
void renderBanner() {
  if ((platform.toUpperCase().indexOf("MAC") > -1) &&
       (browser.toUpperCase().indexOf("IE") > -1) &&
        wasInitialized() && resize > 0 )
  {
    // do something
  }
}
```

```
void renderBanner() {
  final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
  final boolean isIE = browser.toUpperCase().indexOf("IE") > -1;
  final boolean wasResized = resize > 0;

  if (isMacOs && isIE && wasInitialized() && wasResized) {
    // do something

  }
}
```

Extract Class

- When one class does the work of two, awkwardness results
- Instead, create a new class and place the fields and methods responsible for the relevant functionality in it





Replace Temp with Query

- You are using a temporary variable to hold the result of an expression
  - Extract the expression into a method
  - Replace all references to the temp with an expression
  - The new method can then be used in other methods

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;

==============================

if (basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;
…
double basePrice() {
    return _quantity * _itemPrice;
}
```

Move Method

- A method is using more features (attributes and operations) of another class than the class on which it is defined
- Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation or remove it altogether

Move Method

1

```
class Account {
    ...
    double overdraftCharge() {
        if (_type.isPremium()) {
            double result = 10;
            if (_daysOverdrawn > 7 ) {
                result += (_daysOverdrawn - 7) * 0.85;
            }
            return result;
        } else {
            return _daysOverdrawn * 1.75;
        }
    }

    double bankCharge() {
        double result = 4.5;
        if (_daysOverdrawn > 0) {
            result += overdraftCharge();
        }
        return result;
    }

    private AccountType _type;
    private int _daysOverdrawn;
}
```

2

```
class AccountType {
    ...
    double overdraftCharge(int daysOverdrawn) {
        if (isPremium()) {
            double result = 10;
            if (daysOverdrawn > 7 ) {
                result += (daysOverdrawn - 7) * 0.85;
            }
            return result;
        } else {
            return daysOverdrawn * 1.75;
        }
    }
    ...
}
```

Move Method

3

```
class Account {
    ...
    double overdraftCharge() {
        return _type.overdraftCharge(_daysOverdrawn);
    }

    double bankCharge() {
        double result = 4.5;
        if ( daysOverdrawn > 0) {
            result += overdraftCharge();
        }
        return result;
    }

    private AccountType _type;
    private int _daysOverdrawn;
}
```

4

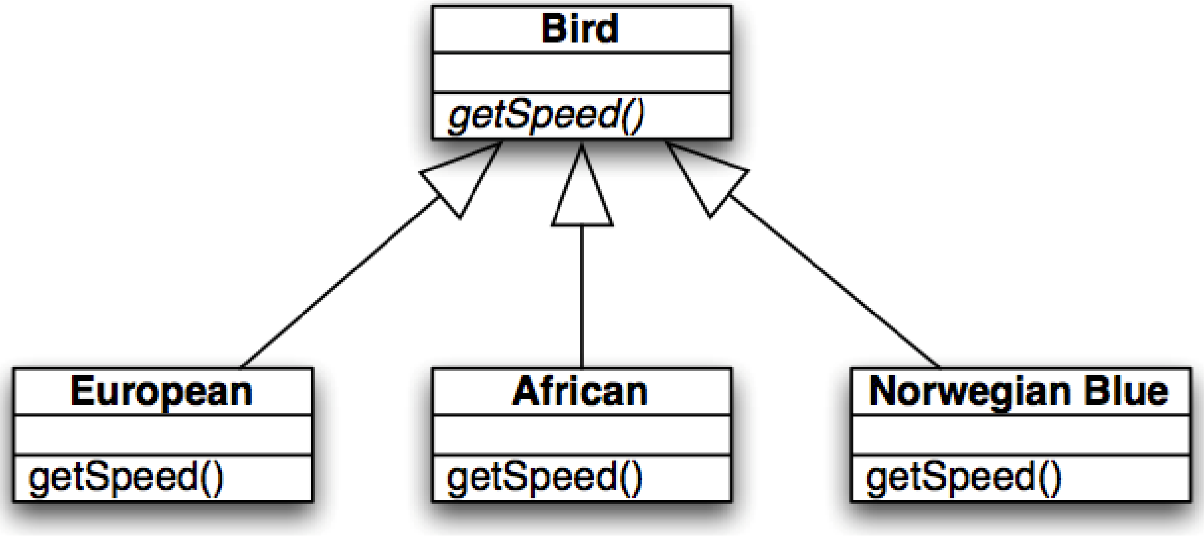
```
class Account {
    ...
    double bankCharge() {
        double result = 4.5;
        if ( daysOverdrawn > 0) {
            result += _type.overdraftCharge(_daysOverdrawn);
        }
        return result;
    }

    private AccountType _type;
    private int _daysOverdrawn;
}
```

Replace Conditional with Polymorphism

- You have a conditional that chooses different behavior depending on the type of an object
- Move each “leg” of the conditional to an overriding method in a subclass. Make the original method abstract.

```
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(“Unknown Type of Bird”)
}
```

Replace Conditional with Polymorphism

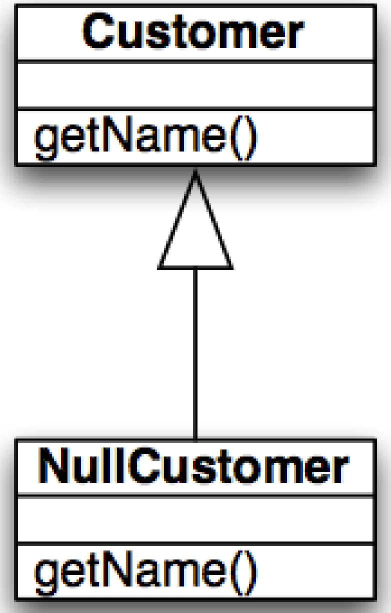



```
void printSpeed(Bird[] birds) {
    for (int i=0; i<birds.length; i++) {
        System.out.println(“” + birds[i].getSpeed());
    }
}
```

Introduce Null Object

- Repeated checks for a null value
- Rather than returning a null value from findCustomer() return an instance of a “null customer” object

```
...
Customer c = findCustomer(...);
...
if (customer == null) {
    name = “occupant”
} else {
    name = customer.getName()
}
if (customer == null) {
...
```



Introduce Null Object

- The conditional goes away entirely!

```
public class NullCustomer {
  public String getName() { return "occupant";}
}
=========================
Customer c = findCustomer(...);
name = c.getName();
```

Separate Query for Modifier

- Sometimes you will encounter code that does something like this
  - **getTotalOutstandingAndSetReadyForSummaries**()
- It is a query method, but it is also changing the state of the object being called
- This change is known as a “side effect” because it’s not the primary purpose of the method
- It is generally accepted practice that queries should not have side effects so this refactoring says to split methods like this into:
  - **getTotalOutstanding**()
  - **setReadyForSummaries**()
- Try as best as possible to avoid any side effects in query methods

Introduce Parameter Object

- You have a group of parameters that go naturally together
  - Stick them in an object and pass the object
- Imagine methods like
  - **amountInvoicedIn**(Date start, Date end);
  - **amountOverdueIn**(Date start, Date end);
- This refactoring says to replace them with something like
  - **amountInvoicedIn**(DateRange dateRange);
- The new class starts out as a data holder but will likely attract methods to it

Encapsulate Collection

- A method returns a collection
- Make it return a read-only version of the collection and provide add/remove methods
- Student class with
  - Map getCourses();
  - void setCourses(Map courses);
- Change to
  - ReadOnlyList getCourses();
  - addCourse(Course c);
  - removeCourse(Course c);

```java
public class CollDemo
{
    public static void main(String[] argv) throws Exception
    {
        List stuff = Arrays.asList(new String[] { "a", "b" });
        List list = new ArrayList(stuff);
        list = Collections.unmodifiableList(list);
        Set set = new HashSet(stuff);
        set = Collections.unmodifiableSet(set);
        Map map = new HashMap();
        map = Collections.unmodifiableMap(map);
        System.out.println("Collection is read-only now.");
    }
}
```

Replace Nested Conditional with Guard Clauses

- This refactoring relates to the purpose of conditional code
  - One type of conditional checks for a variation in “normal” behavior
  - The system will do either A or B; both are considered “normal” behavior
  - The other type of conditional checks for unusual circumstances that require special behavior; if all of these checks fail then the system proceeds with “normal behavior”
- We want to apply this refactoring when we encounter the latter type of conditional
- This refactoring is described in Fowler’s book as:
  - “A method has conditional behavior that does not make clear the normal path of execution; Use guard clauses for all special cases”

Replace Nested Conditional with Guard Clauses

```
double getAmount() {
  double result;
  if (_isDead) {
    result = deadAmount();
  } else {
    if (_isSeparated) {
      result = separatedAmount();
    } else {
      if (_isRetired) {
        result = retiredAmount();
      } else {
        result = normalAmount();
      }
    }
  }
  return result;
}
```

- This type of code may be the result of a novice programmer or due to a programming constraint imposed by some companies that a method can only have a single return
- Often, this constraint causes more confusion than it is worth

Replace Nested Conditional with Guard Clauses

```
double getAmount() {
  if (_isDead) return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) return retiredAmount();
  return normalAmount();
}
```

- With this refactoring, all of the code trying to identify special conditions are turned into one-line statements that determine whether the condition applies and if so handles it
- That’s why these statements are called “guard clauses”
- Even though this method has four returns, it is easier to understand than the method before the refactoring

Wrapping Up

- Refactoring is a useful technique for making non-functional changes to a software system that result in
  - better code structures
  - less code
  - Many refactorings are triggered via the discovery of duplicated code
  - The refactorings then show you how to eliminate duplication
- Bad Smells
  - Useful analogy for discovering places in a system “ripe” for refactoring

Acknowledgments

- Dr. Emily Navarro, University of California, Irvine
- Professor Kenneth M. Anderson, University of Colorado at Boulder