

CS 619 Introduction to OO Design and Development

Refactoring

Fall 2012

Refactoring

- Refactoring is:
 - restructuring (rearranging) code in a series of small, semantics-preserving transformations (i.e. the code keeps working) in order to make the code easier to maintain and modify
- Refactoring is not just arbitrary restructuring
 - Code must still work
 - Small steps only so the semantics are preserved (i.e. not a major re-write)
 - Unit tests to prove the code still works
 - Code is more loosely coupled, more cohesive modules and more comprehensible

The refactoring environment

- Traditional software engineering is modeled after traditional engineering practices
 - Design first, then code)
 - Assumptions: The desired end product can be determined in advance
- Agile is based on different assumptions:
 - Requirements (and therefore design) change as users become acquainted with the software
 - Programmers are professionals with varying skills and knowledge
 - Programmers are in the best position for making design decisions
- Refactoring is fundamental to agile programming
 - Refactoring is sometimes necessary in a traditional process, when the design is found to be flawed

When to refactor

- When should you refactor?
 - Any time you find that you can improve the design of existing code
 - You detect a “bad smell” (an indication that something is wrong) in the code
- When can you refactor?
 - You should be in a supportive environment (agile programming team, or doing your own work)
 - You are familiar with common refactorings
 - Refactoring tools also help
 - You should have an adequate set of unit tests

Refactoring Process

- Make a small change
 - a single refactoring
- Run all the tests to ensure everything still works
- If everything works, move on to the next refactoring
- If not, fix the problem, or undo the change, so you still have a working system

Code Smells

- If it stinks, change it
 - Code that can make the design harder to change
- Examples:
 - Duplicate code
 - Long methods
 - Big classes
 - Big switch statements
 - Long navigations (e.g., a.b().c().d())
 - Lots of checking for null objects
 - Data clumps (e.g., a Contact class that has fields for address, phone, email etc.) - similar to non-normalized tables in relational design
 - Data classes (classes that have mainly fields/properties and little or no methods)
 - Un-encapsulated fields (public member variables)

1: switch statements

- switch statements are very rare in properly designed object-oriented code
 - Therefore, a switch statement is a simple and easily detected “bad smell”
 - Of course, not all uses of switch are bad
 - A switch statement should not be used to distinguish between various kinds of object
- There are several well-defined refactorings for this case
 - The simplest is the creation of subclasses

Example

```
class Animal {
    final int MAMMAL = 0, BIRD = 1, REPTILE = 2;
    int myKind; // set in constructor
    ...
    String getSkin() {
        switch (myKind) {
            case MAMMAL: return "hair";
            case BIRD: return "feathers";
            case REPTILE: return "scales";
            default: return "skin";
        }
    }
}
```



```
class Animal {
    String getSkin() { return "skin"; }
}
class Mammal extends Animal {
    String getSkin() { return "hair"; }
}
class Bird extends Animal {
    String getSkin() { return "feathers"; }
}
class Reptile extends Animal {
    String getSkin() { return "scales"; }
}
```

2: Encapsulate Field

- Un-encapsulated data is a no-no in OO application design. Use property get and set procedures to provide public access to private (encapsulated) member variables.

```
public class Course
{
    public List students;
}
```



```
public class Course
{
    private List students;
    public List getStudents()
    {
        return students;
    }
    public void setStudents(List s)
    {
        students = s;
    }
}
```

```
int classSize = course.students.size();
```

```
int classSize = course.getStudents().size();
```

3. Extract Class

- Break one class into two, e.g. Having the phone details as part of the Customer class is not a realistic OO model, and also breaks the Single Responsibility design principle. We can refactor this into two separate classes, each with the appropriate responsibility.

```
public class Customer
{
    private String name;
    private String workPhoneAreaCode;
    private String workPhoneNumber;
}
```



```
public class Customer
{
    private String name;
    private Phone workPhone;
}

public class Phone
{
    private String areaCode;
    private String number;
}
```

4. Extract Interface

Extract an interface from a class.

It is better to have more specialized interfaces than to have one multi-purpose interface. **interface-segregation principle (ISP)**

```
public class Customer
{
    private String name;

    public String getName(){ return name; }

    public void setName(String string)
    { name = string; }

    public String toXML()
    {
        return "<Customer><Name>" +
            name + "</Name></Customer>";
    }
}
```



```
public class Customer implements SerXML
{
    private String name;

    public String getName(){ return name; }

    public void setName(String string)
    { name = string; }

    public String toXML()
    {
        return "<Customer><Name>" +
            name + "</Name></Customer>";
    }
}
```

```
public interface SerXml {
    public abstract String toXML();
}
```

5. Extract Method

Sometimes we have methods that do too much. The more code in a single method, the harder it is to understand and get right. It also means that logic embedded in that method cannot be reused elsewhere. The Extract Method refactoring is one of the most useful for reducing the amount of duplication in code.

```
public class Customer
{
    void int foo()
    {
        ...
        // Compute score
        score = a*b+c;
        score *= xfactor;
    }
}
```



```
public class Customer
{
    void int foo()
    {
        ...
        score = ComputeScore(a,b,c,xfactor);
    }

    int ComputeScore(int a, int b, int c, int x)
    {
        return (a*b+c)*x;
    }
}
```

6. Extract Subclass

When a class has features (attributes and methods) that would only be useful in specialized instances, we can create a specialization of that class and give it those features. This makes the original class less specialized (i.e., more abstract), and good design is about binding to abstractions wherever possible.

```
public class Person
{
    private String name;
    private String jobTitle;
}
```



```
public class Person
{
    protected String name;
}

public class Employee extends Person
{
    private String jobTitle;
}
```

7. Extract Super Class

When you find two or more classes that share common features, consider abstracting those shared features into a super-class. Again, this makes it easier to bind clients to an abstraction, and removes duplicate code from the original classes.

```
public class Employee
{
    private String name;
    private String jobTitle;
}

public class Student
{
    private String name;
    private Course course;
}
```



```
public abstract class Person
{
    protected String name;
}

public class Employee extends Person
{
    private String jobTitle;
}

public class Student extends Person
{
    private Course course;
}
```

8. Form Template Method

When you find two methods in subclasses that perform the same steps, but do different things in each step, create methods for those steps with the same signature and move the original method into the base class

```
public abstract class Party { }

public class Company extends Party
{
    private String name;
    private String companyType;
    private Date incorporated;
    public void PrintNameAndDetails()
    {
        System.out.println("Name: " + name + " " + companyType);
        System.out.println("Incorporated: " + incorporated.toString());
    }
}

public class Person extends Party
{
    private String firstName;
    private String lastName;
    private Date dob;
    private String nationality;
    public void printNameAndDetails()
    {
        System.out.println("Name: " + firstName + " " + lastName);
        System.out.println("DOB: " + dob.toString() + ", Nationality: " + nationality);
    }
}
```

Refactored

```
public abstract class Party
{
    public void PrintNameAndDetails()
    {
        printName();
        printDetails();
    }

    public abstract void printName();
    public abstract void printDetails();
}

public class Company extends Party
{
    private String name;
    private String companyType;
    private Date incorporated;
    public void printDetails()
    {
        System.out.println("Incorporated: " + incorporated.toString());
    }

    public void printName()
    {
        System.out.println("Name: " + name + " " + companyType);
    }
}

public class Person extends Party
{
    private String firstName;
    private String lastName;
    private Date dob;
    private String nationality;
    public void printDetails()
    {
        System.out.println("DOB: " + dob.toString() + ", Nationality: " + nationality);
    }

    public void printName()
    {
        System.out.println("Name: " + firstName + " " + lastName);
    }
}
```

9. Move Method

If a method on one class uses (or is used by) another class more than the class on which its defined, move it to the other class

```
public class Student
{
    public boolean isTaking(Course course)
    {
        return (course.getStudents().contains(this));
    }
}

public class Course
{
    private List students;
    public List getStudents()
    {
        return students;
    }
}

public class Student
{
}

public class Course
{
    private List students;
    public boolean isTaking(Student student)
    {
        return students.contains(student);
    }
}
```

10. Introduce Null Object

Provide an object as a surrogate for the lack of an object of a given type. The Null Object provides intelligent do nothing behavior, hiding the details from its collaborators.

```
public class User
{
    Plan getPlan()
    {
        return plan;
    }
}

if (user == null)
    plan = Plan.basic();
else
    plan = user.getPlan();

public class User
{
    Plan getPlan()
    {
        return plan;
    }
}

public class NullUser extends User
{
    Plan getPlan()
    {
        return Plan.basic();
    }
}
```

11. Replace Error Code with Exception

A method returns a special code to indicate an error is better accomplished with an Exception.

```
int withdraw(int amount)
{
    if (amount > balance)
        return -1;
    else {
        balance -= amount;
        return 0;
    }
}

void withdraw(int amount)
    throws BalanceException
{
    if (amount > balance)
    {
        throw new BalanceException();
    }
    balance -= amount;
}
```

12. Replace Exception with Test

- Conversely, if you are catching an exception that could be handled by an if-statement, use that instead.

```
double getValueForPeriod (int periodNumber)
{
    try
    {
        return values[periodNumber];
    }
    catch (ArrayIndexOutOfBoundsException e)
    {
        return 0;
    }
}

double getValueForPeriod (int periodNumber)
{
    if (periodNumber >= values.length)
        return 0;
    return values[periodNumber];
}
```

13. Nested Conditional with Guard

- A method has conditional behavior that does not make clear what the normal path of execution is. Use Guard Clauses for all the special cases.

```
double getPayAmount() {
    double result;
    if (isDead) result = deadAmount();
    else {
        if (isSeparated) result = separatedAmount();
        else {
            if (isRetired) result = retiredAmount();
            else result = normalPayAmount();
        }
    }
    return result;
}
```

```
double getPayAmount() {
    if (isDead) return deadAmount();
    if (isSeparated) return separatedAmount();
    if (isRetired) return retiredAmount();
    return normalPayAmount();
};
```

14. Replace Parameter with Explicit Method

You have a method that runs different code depending on the values of an enumerated parameter. Create a separate method for each value of the parameter.

```
void setValue (String name, int value) {
    if (name.equals("height")) {
        height = value;
        return;
    }
    if (name.equals("width")) {
        width = value;
        return;
    }
    Assert.shouldNeverReachHere();
}
```

```
void setHeight(int arg)
{
    height = arg;
}

void setWidth (int arg)
{
    width = arg;
}
```

15. 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 the expression. The new method can then be used in other methods and allows for other refactorings.

```
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;
}
```

16. Rename Variable or Method

Perhaps one of the simplest, but one of the most useful that bears repeating: If the name of a method or variable does not reveal its purpose then change the name of the method or variable.

```
public class Customer
{
    public double getInvcdtLmt();
}
```

```
public class Customer
{
    public double getInvoiceCreditLimit();
}
```

More on Refactorings

- Refactoring Catalog
 - <http://www.refactoring.com/catalog>
- Java Refactoring Tools
 - NetBeans 4+ – Built In
 - JFactor – works with VisualAge and JBuilder
 - RefactorIt – plug-in tool for NetBeans, Forte, JBuilder and JDeveloper. Also works standalone.
 - JRefactory – for jEdit, NetBeans, JBuilder or standalone
- Visual Studio 2005+
 - Refactoring Built In
 - Encapsulate Field, Extract Method, Extract Interface, Reorder Parameters, Remove Parameter, Promote Local Var to Parameter, more.

Eclipse

- Eclipse (and some other IDEs) provide significant support for refactoring

Refactor	
Rename...	\##R
Move...	\##V
Change Method Signature...	\##C
Extract Method...	\##M
Extract Local Variable...	\##L
Extract Constant...	
Inline...	\##I
Convert Anonymous Class to Nested...	
Convert Member Type to Top Level	
Convert Local Variable to Field...	
Extract Superclass...	
Extract Interface...	
Use Supertype Where Possible...	
Push Down...	
Pull Up...	
Introduce Indirection...	
Introduce Factory...	
Introduce Parameter...	
Encapsulate Field...	
Generalize Declared Type...	
Infer Generic Type Arguments...	
Migrate JAR File...	
Create Script...	
Apply Script...	
History...	

Summary

Know when is as important as know-how

- Refactored designs are more complex
 - ➔ Introduce a lot of extra small classes/methods
 - Use "code smells" as symptoms for refactoring opportunities
 - + Duplicated code
 - + Nested conditionals
 - + Large classes/methods
 - + Abusive inheritance
- Rule of the thumb:
 - + All system logic must be stated *Once and Only Once*
 - ➔ a piece of logic stated more than once implies refactoring