CSED332: Software Design Methods

Lecture 4: Refactoring

Kyungmin Bae

Department of Computer Science and Engineering
POSTECH

This Lecture: Refactoring

- What is refactoring, and how should we apply it?
- When should we refactor?
- Where may you want to refactor?

A Fairy Tale

Once upon a time there was a Good Software Engineer whose Customers knew exactly what they wanted. The Good Software Engineer worked very hard to design the Perfect System that would solve all the Customers' problems now and for decades. When the Perfect System was designed, implemented and finally deployed, the Customers were very happy indeed. The Maintainer of the System had very little to do to keep the Perfect System up and running, and the Customers and the Maintainer lived happily every after.

`http://scg.unibe.ch/download/oorp/`

The Reality: Software Changes

- Requirements change
- Design changes
- Code changes (e.g., bug fixes)
- Technological changes
- Social changes
- ...

Refactoring

- Changing the structure of program without changing its behavior
  - easier to understand
  - cheaper to modify
- Key part of
  - XP
  - software evolution
  - making reusable software
- Commonly done, often in secret

XP Practices

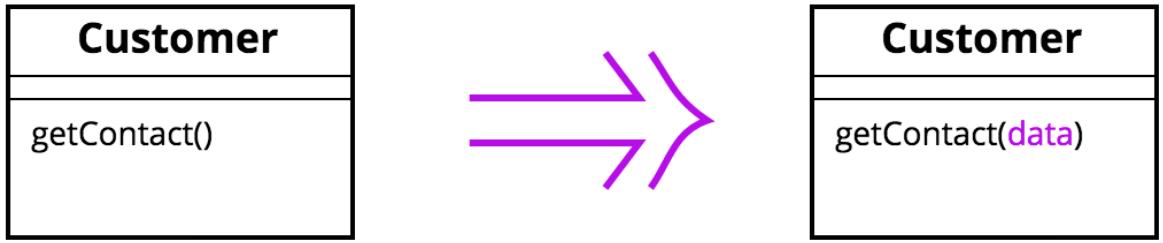
- Whole team (on-site customer)
- System metaphor
- Planning game
- Simple design
- Small releases
- Customer tests
- Pair programming
- Test-driven development
- Design improvement (refactoring)
- Collective code ownership
- Continuous integration
- Sustainable pace (40 hr/week)
- Coding standards

Some Refactoring

- Rename, rename, rename
- Extract method / Inline method
- Extract local variable / Inline local variable
- Move method
- Encapsulate field
- . . . (more in reading)

Add Parameter

- A method needs more information from its caller
  - need to vary a constant
  - forgot to include some information
- Example



Add Parameter: Steps to Perform

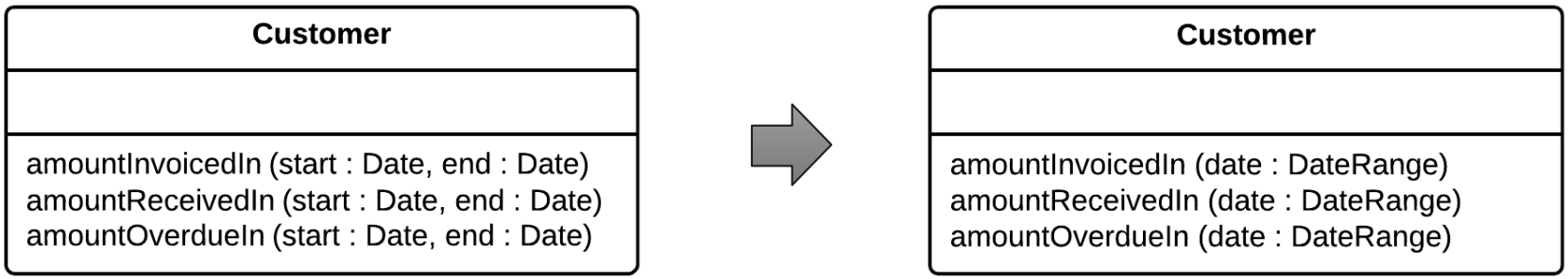
1. Check superclasses and subclasses
2. Make copy of old method, and add parameter
3. Change body of old method so that it calls new one
4. Find all references to the old, and change them to refer to the new
5. Test should run after each change
6. Remove old method

Lessons (1)

- Refactoring requires tests
- Refactoring requires software version control
- Knowing how to perform a refactoring makes it safer and easier
- You can perform a refactoring $\neq$ You should do the refactoring.

Introduce Parameter Object

- There is a group of parameters that naturally go together
  - many methods have same parameters
  - parameters are passed unchanged from one method to another
  - method has too many parameters
- Example



Introduce Parameter Object: Steps to Perform

1. Make a new class for the group of parameters
2. Use "Add Parameter" for the new class
   - use a new object for the parameter in all the callers
3. For each of the original parameters:
   1. Modify caller
      - store parameter in the new object and omit parameter from call
   2. Modify method body
      - omit original parameter and use the value in the new parameter
   3. If method body calls another method with parameter object
      - use existing parameter object instead of making a new one

Example: Introduce Parameter Object (1)

```
class Account {
  double getFlowBetween(Date start, Date end) {
    double result = 0;
    Iterator e = _entries.elements();
    while (e.hasNext()) {
        Entry each = (Entry) e.next();
        Date date = each.getDate();
        if (date.equals(start) || date.equals(end) ||
                (date.after(start) && date.before(end))) {
            result += each.getValue();
        }
    }
    return result;
}
```

Example: Introduce Parameter Object (2)

Make a New Class for the Group of Parameters

```
class DateRange {
    DateRange (Date start, Date end) {
        _start = start;
        _end = end;
    }
    Date getStart() {
        return _start;
    }
    Date getEnd() {
        return _end;
    }
    private final Date _start;
    private final Date _end;
}
```

Example: Introduce Parameter Object (3)

Apply "Add Parameter" to the New Class

```
class Account {
  double getFlowBetween(Date start, Date end, DateRange range) {
    double result = 0;
    Iterator e = _entries.elements();
    while (e.hasNext()) {
        Entry each = (Entry) e.next();
        Date date = each.getDate();
        if (date.equals(start) || date.equals(end) ||
                (date.after(start) && date.before(end))) {
            result += each.getValue();
        }
    }
    return result;
}
```

Example: Introduce Parameter Object (4)

Use a New Object for the Parameter in All the Callers

```
double flow = acc.getFlowBetween(start, end);
```

```
double flow = acc.getFlowBetween(start, end, new DateRange(null, null));
```

Example: Introduce Parameter Object (5)

Modify Caller for the Parameter `start`

```
double flow = acc.getFlowBetween(start, end, new DateRange(null, null));
```

```
double flow = acc.getFlowBetween(end, new DateRange(start, null))
```

Example: Introduce Parameter Object (6)

Modify Method Body the Parameter `start`

```
class Account {
  double getFlowBetween(Date end, DateRange range) {
    double result = 0;
    Iterator e = _entries.elements();
    while (e.hasNext()) {
        Entry each = (Entry) e.next();
        Date date = each.getDate();
        if (date.equals(range.getStart()) || date.equals(end) ||
                  (date.after(range.getStart()) && date.before(end))) {
            result += each.getValue();
        }
    }
    return result;
}
```

Example: Introduce Parameter Object (7)

Modify Caller for the Parameter `end`

```
double flow = acc.getFlowBetween(end, new DateRange(start, null))
```

```
double flow = acc.getFlowBetween(new DateRange(start, end));
```

Example: Introduce Parameter Object (8)

Modify Method Body for the Parameter end

```
class Account {
  double getFlowBetween(DateRange range) {
    double result = 0;
    Iterator e = _entries.elements();
    while (e.hasNext()) {
        Entry each = (Entry) e.next();
        Date date = each.getDate();
        if (date.equals(range.getStart()) ||
                date.equals(range.getEnd()) ||
                    (date.after(range.getStart()) &&
                        date.before(range.getEnd()))) {
            result += each.getValue();
        }
    }
    return result;
}
```

Introduce Parameter Object: Post-Processing

- Look to see if some code should be moved to its methods

```
class DateRange {
    // ...
    boolean includes (Date arg) {
        return arg.equals(_start) || arg.equals(_end) ||
                    (arg.after(_start) && arg.before(_end));
    }
```

```
class Account {
  double getFlowBetween(DateRange range) {
    double result = 0;
    Iterator e = _entries.elements();
    while (e.hasNext()) {
        Entry each = (Entry) e.next();
        Date date = each.getDate();
        if (range.includes(each.getDate())) {
            result += each.getValue();
        }
    }
    return result;
}
```

Lessons (2)

- Refactoring should be small
- Check after each step to make sure you did not make a mistake
- One refactoring leads to another
- Major change requires many refactoring steps

Change Design "XOR" Functionality

- Separate changing behavior from refactoring
  - changing behavior requires new tests
  - refactoring must pass all tests
- Only refactor when you need to
  - before you change behavior
  - after you change behavior
  - to understand

Some Refactoring: Extract Method

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

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

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

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

Some Refactoring: Inline Method

```
int getRating() {
    return (moreThanFiveLateDeliveries()) ? 2 : 1;
}

boolean moreThanFiveLateDeliveries() { // A method's body is clear
    return _numberOfLateDeliveries > 5;
}
```

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

Some Refactoring: Introduce Explaining Variable

```
if ( (x==0) || ((y<17) && name == null) ) {
    // ...
}
```

```
final boolean outOfActiveAreal = (x==0) || ((y<17) && name == null) ;
if (outOfActiveAreal ) {
    // ...
}
```

Some Refactoring: Inline Temporary Variable

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000)
```

```
return (anOrder.basePrice() > 1000)
```

Some Refactoring: Split Temporary Variable

```
double temp = 2 * (_height + _width);
System.out.println (temp);
temp = _height * _width;
System.out.println (temp);
```

```
final double perimeter = 2 * (_height + _width);
System.out.println (perimeter);
final double area = _height * _width;
System.out.println (area);
```

More Refactoring[1]

- Composing methods
  - Extract Method, Replace Method with Method Object, ...
- Moving features between objects
  - Move Field, Move Method, Introduce Foreign Method, ...
- Organizing data
  - Encapsulate Field, Replace Array with Object, ...
- Simplifying conditional expressions
  - Replace Conditional with Polymorphism, ...
- Making method calls simpler
  - Introduce Parameter Object, Parameterize Method, ...
- ...

[1] http://refactoring.com/catalog/

Automated Refactoring Support

- Deciding where to refactor
  - Tools for measuring cohesion, size, etc.
  - Tools for measuring code duplication or cloning
- Performing the change
  - Refactoring Browser for Smalltalk, first
  - Over a dozen of tools for Java
  - Eclipse

Refactoring and Reverse Engineering

- Reverse engineering
  - Converting code into design
  - Requires understanding system
- Refactoring
  - Transforming code
  - Must find all uses of the code, methods that override, etc.
  - Requires tests, but not deep understanding

XP Rules

- Make it work
  - make tests run
- Make it right
  - meaningful and easy to understand
  - as simple as possible
  - eliminate duplication
- Make it fast

Refactoring and Design

- XP creates a design by refactoring
- Refactor parts of the system that are hard to understand
- Eventually nothing is hard to understand
- Code matches the design
- Refactoring is a good way to understand a complex system

Code Smells

Where to Refactor: Code Smells

- Definition

  . . . certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring. – M. Fowler

  . . . symptom[s] in the source code of a program that possibly indicate a deeper problem. . . . usually not bugs ... not technically incorrect and don't currently prevent the program from functioning. Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future. – Wikipedia

- Code smells are clear signs that your design is starting to decay
  - Long term decay leads to "software rot"

Example Code Smells[2]

- Duplicated code
- Long method
- Large class
- Long parameter list
- Message chain
- Feature envy
- Switch statements
- Data class
- Speculative generality
- Temporary field
- Refused bequest
- Shotgun Surgery
- Parallel class hierarchies
- ...

[2] `https://sourcemaking.com/refactoring/smells`

Duplicated Code

- Duplicate methods in subclasses
  - Refactoring: Move to superclass, possibly create superclass
- Duplicate expressions in same class
  - Refactoring: Extract method
- Duplicate expressions in different classes
  - Refactoring: Extract method, move to common component

Example: Duplicated Code

```
sqrt(pow(loc1.getX()-loc2.getX(), 2) + pow(loc1.getY()-loc2.getY(), 2))
```

```
sqrt(square(loc1.getX()-loc2.getX()) + square(loc1.getY()-loc2.getY()))

double square(double d) {
    return pow(d, 2);
}
```

- Pros and cons?

Long Method

- Won't fit on a page
- Cannot think of whole thing at once
- Refactoring: Extract method
  - Loop body
  - Places where there is (or should be) a comment
  - Don't write comments but name functions

Example: Long Method (1)

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + name() + "\n";
    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();
        switch (each.tape().movie().priceCode()) { //determine amounts for each line
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.daysRented() > 2) thisAmount += (each.daysRented() - 2) * 1.5; break;
            case Movie.NEW_RELEASE:
                thisAmount += each.daysRented() * 3; break;
            case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (each.daysRented() > 3) thisAmount += (each.daysRented() - 3) * 1.5; break;
        }
        totalAmount += thisAmount;
        frequentRenterPoints ++; // add frequent renter points
        // add bonus for a two day new release rental
        if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) && each.daysRented() > 1)
            frequentRenterPoints ++;
        //show figures for this rental
        result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(thisAmount) + "\n";
    }
    //add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";
    return result;
}
```

Example: Long Method (2)

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + name() + "\n";
    while (rentals.hasMoreElements()) {
       double thisAmount = 0;
       Rental each = (Rental) rentals.nextElement();

       //determine amounts for each line
       thisAmount = amountOf(each);
       totalAmount += thisAmount;

       frequentRenterPoints ++; // add frequent renter points
       // add bonus for a two day new release rental
       if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) && each.daysRented() > 1)
           frequentRenterPoints ++;
       //show figures for this rental
       result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(thisAmount) + "\n";
    }
    //add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";
    return result;
}
```

Example: Long Method (3)

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + name() + "\n";
    while (rentals.hasMoreElements()) {
       Rental each = (Rental) rentals.nextElement();

       //determine amounts for each line
       totalAmount += amountOf(each);

       frequentRenterPoints ++; // add frequent renter points
       // add bonus for a two day new release rental
       if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) && each.daysRented() > 1)
           frequentRenterPoints ++;
       //show figures for this rental
       result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(amountOf(each)) + "\n";
    }
    //add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";
    return result;
}
```

Example: Long Method (4)

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + name() + "\n";
    while (rentals.hasMoreElements()) {
       Rental each = (Rental) rentals.nextElement();

       //determine amounts for each line
       totalAmount += amountOf(each);

       // add frequent renter points
       frequentRenterPoints += frequentRenterPointOf(each);

       //show figures for this rental
       result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(amountOf(each)) + "\n";
    }
    //add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";
    return result;
}
```

- See: `http://www.cs.unc.edu/~stotts/723/refactor/`

Large Class

- More than a couple dozen methods, or half a dozen variables
- Refactoring:
  - Split into component classes
  - Create superclass
  - If using switch statement, split into subclasses

Long Parameter List

- Refactoring:
  - Introduce parameter object
  - Only worthwhile for several methods with same parameter list and they call each other

Poor Names

- Name of a variable should reflect its function
  - unless the function is obvious
- Also applies to constants, classes, and methods, . . .
  - magic number (e.g., 3.14) vs. named constant (e.g., PI)
- Also applies to data primitive
  - data primitive (e.g., int[]) vs. abstract data type (e.g., Cell)

Example: Poor Names (1)

- Bad

```
x = x - xx;
xxx = fido + SalesTax( fido );
x = x + LateFee( x1, x ) + xxx;
x = x + Interest( x1, x );
```

- Better

```
balance = balance - lastPayment;
monthlyTotal = newPurchases + SalesTax( newPurchases );
balance = balance + LateFee( customerID, balance ) + monthlyTotal;
balance = balance + Interest( customerID, balance );
```

Example: Poor Names (2)

- Bad

```
List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
        if (x[0] == 4) list1.add(x);
    return list1;
}
```

- Better

```
List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard)
        if (cell[STATUS_VALUE] == FLAGGED) flaggedCells.add(cell);
    return flaggedCells;
}
```

- Even better

```
List<Cell> getFlaggedCells() {
    List<Cell> flaggedCells = new ArrayList<Cell>();
    for (Cell cell : gameBoard)
        if (cell.isFlagged()) flaggedCells.add(cell);
    return flaggedCells;
```

Message Chain

- Long list of method calls

```
customer.getAddress().getState();
window.getBoundingbox().getOrigin().getX();
```

- Refactoring: replace by shorter ones

```
customer.getState();
window.leftBoundary();
```

Feature Envy

- Code "wishes" it were in another class

```
teacher.getClasses().add(thisClass);
teacher.setClassLoad(teacher.getClassLoad()+1);
```

- Refactoring: move it

```
teacher.addClass(thisClass);
```

Example: Feature Envy

```
private double amountOf(Rental aRental) {
    double result = 0;
    switch (aRental.tape().movie().priceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (aRental.daysRented() > 2)
                result += (aRental.daysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            result += aRental.daysRented() * 3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (aRental.daysRented() > 3)
                result += (aRental.daysRented() - 3) * 1.5;
            break;
    }
    return result;
}
```

Example: Feature Envy

```
class Rental {
...
    double amountOf() {
        double result = 0;
        switch (tape().movie().priceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented() > 2)
                    result += (daysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented() * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented() > 3)
                    result += (daysRented() - 3) * 1.5;
                break;

        }
        return result;
    }
```

Data Class

- Class has no methods except for getter and setters
- Refactoring:
  - Look for missing methods (feature envy?)
  - Move them to the class
  - Merge with another class
- Example: DateRange from last lecture

Switch Statement

- Replace switch (or nested if) with a method call
- Make subclass for each case
- Use dynamic dispatch/polymorphism
- Use pattern matching (functional programming language)

Example: Switch Statement (Library)

```
long computeWeight(Element e) {
    if (e instanceof Book)
        return ((Book)e).getBookWeight();
    else
        return ((Collection)e).getTotalWeight();
}
```

```
long computeWeight(Element e) {
     return e.getWeight();
}

abstract class Element { // ...
    public abstract long getWeight();
}
class Book { // ...
    public long getWeight() { /* ... */ }
}
class Collection { // ...
    public long getWeight() { /* ... /* }
}
```

Speculative Generality

- Interfaces/abstract classes that are implemented by only one class
- Unnecessary delegation
- Unused parameters

- Refactoring: remove interfaces, remove parameter, ...

Temporary Field

- Instance variable is only used during part of lifetime of object
- E.g., it is only used while the object is initialized

- Refactoring: move variable into another object (or a new class)

Refused Bequest

- Subclass uses only some of methods/fields inherited from parents
- Example: A is a subclass of B
  - A overrides inherited methods of B
  - but A does not use some variables/methods of B
- Refactoring: move common code into a common superclass
  - give A and B a common superclass and move common code into it

Preventing Changes

- Shotgun Surgery
  - making a change requires changing many parts of a program
  - Refactoring: make a new object that represents everything that changes (methods that change together should stay together)
- Parallel class hierarchies
  - adding class in one class hierarchy requires adding class in another
  - Refactoring: make instances of one hierarchy refer to instances of another hierarchy, and remove the hierarchy in the referred class.
- Design patterns can address these problems (we will study later).

Comments

Do not document bad code — rewrite it

(Kernighan and Plauger 1978)

- Comments are used to explain difficult code
- But they should not be used as a crutch to explain bad code

Other Code Smells

- Too many bugs
  - what if one part of the system has more than its share of the bugs
  - redesign, rewrite, refactor
- Too hard to understand
  - hard to fix bugs because you do not understand
  - hard to change because you do not understand
- Too hard to change
  - due to lack of tests
  - dependencies (e.g., global variables) or duplication

Summary

- Refactoring
  - change code while preserving behavior
  - code smells: code pieces with potentially bad design
- Homework 4 (Due 10/9)
  - refactoring: working in pair
- Reading
  - Code Complete 2nd Chapter 24
  - `https://en.wikipedia.org/wiki/Code_refactoring`
  - `https://en.wikipedia.org/wiki/Code_smell`
- No classes on 10/11
  - make-up class will be scheduled

Questions?