Course Application Design

Refactoring

Michiel Noback
Institute for Life Sciences and Technology
Hanze University of Applied Sciences

Introduction

- This presentation deals with aspects that you should apply all the time: refactoring
- Refactoring code is about improving readability, flexibility and extensibility without changing the functionality
- This involves renaming, extracting methods, classes, interfaces, and applying design patterns and SOLID principles

A base example

What do you do?

```
private List<int[]> theList;

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

BaseExampleFirst.java

What do you do?

```
/**
 * This is a Minesweaper game. The board is
 * represented by a List of cells. The cells are
 * represented by int arrays, where index 0 holds
 * the cell status: 4 means “flagged”
 */
private List<int[]> theList;
public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<>();
    for(int[] x : theList) {
        if(x[0] == 4){
            list1.add(x);
        }
    }
    return list1;
}
```

nl.bioinf.refactoring.BaseExampleFirst.java

Rename

```
private List<int[]> gameBoard;

public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<>();
    for(int[] x : gameBoard) {
        if(x[0] == 4){
            flaggedCells.add(x);
        }
    }
    return flaggedCells;
}
```

nl.bioinf.refactoring.BaseExampleSecond.java

Extract constants

```
private static final int STATUS_VALUE_INDEX = 0;
private static final int STATUS_FLAGGED = 4;
private List<int[]> gameBoard;

public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<>();
    for(int[] x : gameBoard) {
        if(x[STATUS_VALUE_INDEX] == STATUS_FLAGGED){
            flaggedCells.add(x);
        }
    }
    return flaggedCells;
}
```

nl.bioinf.refactoring.BaseExampleThird.java

Extract classes

```
private List<Cell> gameBoard;
private static final int STATUS_VALUE_INDEX = 0;

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

static class Cell {
    private static final int STATUS_FLAGGED = 4;
    private int status;

    public boolean isFlagged() {
        return status == STATUS_FLAGGED;
    }
}
```

nl.bioinf.refactoring.BaseExampleFourth.java

Names

Meaningful names

- Names should reveal the purpose of what they refer to, as specific as possible

Variable names

- If a name requires a comment, it does not reveal its intent
- Use names that specifiy what is measured and in which unit

```
//Elapsed time in days
private int t;                          //BAD

private int elapsedTimeInDays;          //GOOD
```

Class names

- Classes should have a noun or noun phrase

```
FileParse                    //BAD
ParseFile                    //BAD
FastaReader                  //GOOD

Processor                    //BAD
GffFileParser                //GOOD
```

Method names

- Methods should have verb or verb phrases
- Accessors, mutators and predicates should be named for value they intend to publish and prefixed with *get*, *set* and *is* - the Javabean standard.

```
name()                          //BAD
getName()                       //GOOD

doIt()                          //BAD
parseGffFile()                  //GOOD

process()                       //BAD
processGffElement()             //GOOD
```

Constructors

Factory methods over Constructors

- When constructors are overloaded, consider using Factory methods
- Instead of this:

```
public Cell() {}

public Cell(int status) {
    this.status = status;
}
```

- Try this

```
public Cell() {}

public static Cell createFlaggedCell() {
    Cell cell = new Cell();
    cell.status = STATUS_FLAGGED;
    return cell;
}
//Or, more generic
public static Cell fromStatusCode(int status) {
    Cell cell = new Cell();
    cell.status = status;
    return cell;
}
```

- So instead of this

```
new Cell(4);
//OR
new Cell(Cell.STATUS_FLAGGED);
```

- You can do this

```
//BETTER!
Cell.fromStatusCode(Cell.STATUS_FLAGGED);
//Best?
Cell.createFlaggedCell();
```

Catch *null* as soon as possible

- Null values are the most common cause of bugs
- Catch them as soon as possible, preferably in the constructor
- Use Objects.requireNonNull

```
this.wheel = Objects.requireNonNull(
                    wh, "Wheel should be non-null!");
```

Place variables in a clear context

- Names are usually more meaningfull when placed in a context
- With this class, you need to study carefully what the variables represent – for instance, is the variable **numberOfLogins** related to the datasource or something else? And what about **port**?

```
public class UserManagement {
        private long id;
        private String name;
        private String street;
        private int number;
        private String numberPostfix;
        private String zipCode;
        private int numberOfLogins;
        private String dataSourceUrl;
        private int port;
        //much code
```

- This represents the same data now ordered in logical entities - **context**

```
public class UserManagement {
    private User user;
    private Address address;
    private DataSource;
    //much code
}
```

```
public class User {
    private long id;
    private String name;
    private int numberOfLogins;
}
```

```
public class DataSource {
    private String dataSourceUrl;
    private int port;
}
```

```
public class Address {
    private String street;
    private int number;
    private String numberPostfix;
    private String zipCode;
}
```

Part two

Functions / Methods

Good methods…

- have very descriptive name (verb)
- are small
- have a maximum of two indentation levels
- do one thing only (**SRP!**)
- preferably define no more than 2 parameters

(in general)

Wrap multiple arguments

- When you have multiple parameters for a method, consider wrapping them in an object

```
abstract Circle createCircle(double x, double y, double radius);
```

```
abstract Circle createCircle(Point center, double radius);
```

Comments

To comment or not to comment

- In general, ***comment your API methods***, as concise as possible.
- For the rest: make your code self-explanatory

```
if (employee.getWorkingYears() >= MINIMUM_COMPLETE_PENSION_PERIOD
        && employee.getFteSize() >= 0.95) { }

//OR this?
if (employee.isEligibleForFullPension()) { }
```

Useless comments

```
/**
 * The employee that is being dealt with
 */
private Employee employee;


/**
 * This method returns whether the employee is
 * eligible for full pension.
 * @return eligible
 */
public boolean isEligibleForFullPension() {
    return eligibleForFullPension;
}
```

Code block comments

- When you need code block comments, your method is (way) too long

```
 wordCount += words.length;
    } //while
} //try
```

Error Handling

- Use Exceptions rather than Error codes
- Use unchecked exceptions rather than checked exceptions
- Don’t return null but rather the **SPECIAL CASE object** or Optional

Classes

Getting repetitive…

- Classes should be small
- Classes should adhere to the SRP
- Classes should be cohesive

Cohesive?

- The class on the next slide is highly cohesive because two out of three methods deal with both instance variables and one method with one instance variable

```java
public class Stack {
    private int topOfStack = 0;
    private List<Integer> elements = new LinkedList<Integer>();

    public int size() {
        return topOfStack;
    }

    public void push(int element) {
        topOfStack++;
        elements.add(element);
    }

    public int pop() throws EmptyStackException {
        if (topOfStack == 0)
            throw new EmptyStackException();
        int element = elements.get(--topOfStack);
        elements.remove(topOfStack);
        return element;
    }
}
```

Classes should not spill their guts

- Never deal out your (mutable) collection to the outside
- Staying with the Stack example, instead of this, which creates a risk of data corruption, makes your design easier to break and harder to change (rigid & fragile)

```
private List<Integer> elements = new LinkedList<Integer>();

public List<Integer> getStackAsList() {
    return elements;
}
```

- you should do this

You can change data representation without breaking anything

```
private List<Integer> elements = new LinkedList<Integer>();

public List<Integer> getStackAsList() {
    return Collections.unmodifiableList(elements);
}
```

Can’t touch this!

CAINI

- (Code Against Interfaces Not Implementations)
- Rely on abstractions not implementations
  - Interfaces
  - Abstract classes
- Do not limit the variety of implementations

Want to know more?