ENGI3255 Software Testing & Quality Assurance

Project 2 – Reverse Engineering

Refactoring & Object Oriented Design

1. Aims

The aim of this project is to give you more experience with refactoring, and especially refactoring that relates to multiple classes. Some of the refactoring that we'll do here could be justified well in a larger system. However, the only way to get good at refactoring is to practice, and smaller examples are easier to begin with. You will also learn how to recover system design from the code.

Objectives:

On completion of this project the student should be able to:

a) Carry out a variety of refactoring, including: renaming variables, extracting and moving methods closer to "home", and creating classes and interfaces.
b) Develop a strategy where refactoring is carried out in small steps, with the tests being run after each change so that one can quickly recover from any faults that are introduced as part of the refactoring process.
c) Introduce changes as part of the refactoring process to enhance the software-design from an object-oriented point of view.
d) Develop an understanding that there is often a tension between the different ways that code can be structured, and it's not always obvious that a planned refactoring will lead to an improvement.

2. Background

In software engineering, *"refactoring"* source code means modifying it without changing its behavior, and is sometimes informally referred to as "cleaning it up". Refactoring neither fixes bugs nor adds new functionality, though it might precede either activity. Rather it improves the understandability of the code and changes its internal structure and design, and removes dead code, to make it easier to comprehend, more maintainable and amenable to change. Refactoring is usually motivated by the difficulty of adding new functionality to a program or fixing a bug in it.

In extreme programming and other agile methodologies, refactoring is an integral part of the software development cycle: developers first write tests, then write code to make the tests pass, and finally refactor the code to improve its internal consistency and clarity. Automatic unit testing ensures that refactoring preserves correctness.

Code smells (i.e. any symptom in the source code that indicates something may be wrong) are a heuristic to indicate when to refactor, and what specific refactoring techniques to use. An example of a trivial refactoring is to change a variable name into something more meaningful, such as from a single letter 'i' to 'interestRate'. A more

complex refactoring is to turn the code within an if block into a subroutine. An even more complex refactoring is to replace an if conditional with polymorphism.

While "cleaning up" code has happened for decades, the key insight in refactoring is to intentionally "clean up" code separately from adding new functionality, using a known catalogue of common useful refactoring methods, and then separately testing the code, to ensure that existing behavior is preserved. The aim is to improve an existing design without altering its intent or behavior.

List of refactoring techniques:
Here is a very incomplete list of code refactorings. A longer list can be found in Fowler's Refactoring book and in Fowler's Refactoring Website (see references).

- Techniques that allow for more abstraction
  - Encapsulate Field - force code to access the field with getter and setter methods
  - Generalize Type - create more general types to allow for more code sharing
  - Replace type-checking code with State/Strategy
  - Replace conditional with polymorphism

- Techniques for breaking code apart into more logical pieces
  - Extract Method, to turn part of a larger method into a new method. By breaking down code in smaller pieces, it is more easily understandable. This is also applicable to functions.
  - Extract Class moves part of the code from an existing class into a new class.

- Techniques for improving names and location of code
  - Move Method or Move Field - move to a more appropriate Class or source file
  - Rename Method or Rename_Field - changing the name into a new one that better reveals its purpose
  - Pull Up - in OOP, move to a superclass
  - Push Down - in OOP, move to a subclass
  - Shotgun surgery - in OOP, when a single change affects many classes. In this case we keep all affecting code in a single class so that we make changes only at one place.

3. Project Description

You are supplied with the Java classes Movie, Rental and Customer. A Movie is one of three types: CHILDRENS, REGULAR and NEW RELEASE. The type determines the charge for the rental and the number of frequent-renter points for that movie. A Rental is of a Movie for a number of days. A Customer may make several Rentals. The Customer method statement() provides a summary of the rented movies, charges and frequent-

renter points. The class TestMovieRental has some simple unit tests. The last unit test is commented out; we'll make use of that in a later step. The aim is to add an HTML summary to the program. However, the program as provided is very smelly – it appears that the author of the program doesn't understand object-orientation. So we will first refactor the code step by step to make it easier to add the new code. We then try to recover part of the system design by drawing the class diagram of the Movie rental system.

4. Required Tasks

The source code for this project is available as *project1.zip* under *Resources for Project 1* on D2L course website. If you wish, you may start by converting the program to use JDK under eclipse. This project consists of 17 refactoring steps as described below. You will need to understand the code you modify, by the end of the series of changes. After finishing refactoring the software, you will draw the class diagram of the software. You are to work in **groups of no more than 2 people**.

Step 1: Extract method *chargeForRental()*
The method *statement()* in *Customer* is rather long and unwieldy. We should be able to use some of this code for generating HTML, but we would end up with a lot of duplication if we simply copied and altered it as it is. Extract out a *private double* method *chargeForRental()* that is responsible for calculating the charge for a particular *Rental*. Make sure that the tests still pass.

Step 2: Rename variables
If you haven't already done so, improve the names of the two variables in the code you extracted into *chargeForRental()*, because they need to be more specific and helpful. Eg, change *each* to *rental*, and change *thisAmount* to *charge*. Make sure that the tests still pass.

Step 3: Move method *costForRental()*
Now that we have *chargeForRental()* extracted, it's clear that all of the logic is concerned with the *Rental* and *Movie* and it does not depend on the *Customer*. So move the method *chargeForRental()* into the class *Rental*, rename it *charge()* and make it public. In doing so, the code changes to take account of its new context (eg, so that a reference to *rental.getDaysRented()* changes to *this.getDaysRented()* and then to *getDaysRented()*).

The statement calling this method in *Customer* changes from:

```
charge = chargeForRental(rental);
```

to

```
charge = rental.charge();
```

Make sure that the tests still pass.

Step 4: Extract method *charge()* and move it

Now we can see that the method *charge()* in class *Rental* is making use of the internals of *Movie*. So we need to extract most of it and move it into *Movie*. But the logic depends on the number of days rented, so this needs to be passed as a parameter. So the method in *Rental* turns into:

```
public double charge() {
      return movie.charge(daysRented);
}
```

Step 5: Extract method *frequentRenterPoints()*
As with the previous two steps, there is still some code in *Customer* that doesn't belong there. Extract out a method *frequentRenterPoints()* and then move it into the class *Rental*. It is may be easier to do this kind of refactoring in two steps (first extract the method locally, then move it to the other class), running the tests in between. Make sure that the tests still pass.

Step 6: Extract method *isNewRelease()*
Now that we get it into *Rental*, we can see that it contains the price code test that's rather specific to *Movie*. Extract a *boolean* method *isNewRelease()* and place it in *Movie*, where it belongs. Make sure that the tests still pass.

Step 7: Rename variable; Change to for loop
Now we look at what's left, we can see that there is a variable in method *statement()* in *Customer* that could be improved. Change the *Rental each* to *Rental rental*. We also see that a *while* loop is used to process each of the rentals, but it would be clearer to use a *for* loop. Change that. Make sure that the tests still pass after each little change.

Step 8: Write test case for HTML
We can now introduce a test case for the HTML statement that we want to be produced.

```
public void testHtmlCustomer() {
      customer.addRental(rentMatrix);
      customer.addRental(rentMatrix2);
      assertEquals("<html><head><title>Rentals: John Hood</title>
      </head><body>\n"+"<h1>Rentals: John Hood</h1>\n"+
        "<table border=1><tr><th>Days</th><th>Title</th><th>
        Charge</th></tr>\n"+"<tr><td align=right>4</td> <td>
        Matrix<td align=right>$5.0</td></tr>\n"+ "<tr><td
        align=right>5</td><td>Matrix2<td align=right>$15.0</td>
        </tr>\n"+ "<tr><td></td><td><i>total</i><td align=right>
        $20.0</td></tr>\n"+ "</table><p>Frequent renter points=3
        </p>\n"+"</body></html>\n",
      customer.htmlStatement());
}
```

But don't uncomment it yet, as we need to do some refactoring before we can implement it in Step 17.

Step 9: Extract method *rentalLine()*
To get closer to handling HTML, let's extract the "display" methods, and we'll then use them to generalize the output. First, introduce a *private* method *rentalLine()* that takes three arguments and generates the string. So the code that used to be:

result += rental.getDaysRented() +" days of '" + rental.getMovie().getTitle() +"' $" + charge + "\n";

changes to:

```
result += rentalLine(rental.getDaysRented(), rental.getMovie().getTitle(),charge);
```

Make sure that the tests still pass.

Step 10: Move method *rentalLine()*
Now that we have extracted this method, it's clear that it depends entirely on the *Rental*. So let's move it to the class *Rental* and remove all of the arguments. So we end up in *Customer* with:

```
result += rental.rentalLine();
```

Notice that we can now remove the methods *getDaysRented()* and *getMovie()* from class *Rental*, as they are no longer needed. We can use the method *charge()* directly in the method *rentalLine()* in class *Rental*. It doesn't matter that we end up calling this method twice (it's not worth losing simplicity unless we know that some code causes a performance problem). Make sure that the tests still pass.

Step 11: Extract method *headerLine()*
We apply the same extraction to create *headerLine()*, which takes one argument. So the code that used to be:

```
String result = "Rentals: " + getName() + "\n";
```

changes to:

```
String result = headerLine(getName());
```

Step 12: Extract method *footerLine()*
We apply the same extraction to create *footerLine()*, which takes two arguments. So the code that used to be:

```
//add footer lines
result += "Total = $" + totalAmount + "\n";
result += "Frequent renter points ="+frequentRenterPoints + "\n";
return result + "---\n";
```

changes to:

```
return result + footerLine(totalAmount, frequentRenterPoints);
```

Make sure that the tests still pass.

Step 13: Extract class *Report*
Now we can create a new class *Report* and move methods *headerLine()* and *footerLine()* from *Customer* into it. So now inside *statement*, we replace:

```
String result = headerLine(getName());
...
return result + footerLine(totalAmount, frequentRenterPoints);
```

by:

```
Report report = new Report();
String result = report.headerLine(getName());
...
return Result+report.footerLine(totalAmount,frequentRenterPoints);
```

Make sure that the tests still pass.

Step 14: Extract part of *rentalLine()* to *Report*

The remaining part of the report is generated by *rentalLine()* in class *Rental*. So we need to pass the *Report* to it. In class *Customer*, we end up with the call:

```
result += rental.rentalLine(report);
```

and in class *Rental*, we end up with:

```
public String rentalLine(Report report) {
    return report.rentalLine(daysRented, movie.getTitle(), charge());
}
```

and, finally, *Report* contains the method:

```
public String rentalLine(int days, String title, double charge) {
      return days + " days of '" + title +"' $" + charge + "\n";
}
```

Make sure that the tests still pass.

Step 15: Add *Report* parameter to *statement()*

We want to have a different type of *Report* for generating HTML. So let's add a parameter of type *Report* to *statement()* in class *Customer*. That gives us:

```
public String statement() {
      return statement(new Report());
}

public String statement(Report report) {
      ....
```

Of course, we remove the declaration of the *Report* from inside the second *statement()* method.

Step 16: Extract interface *Report*

Before we can creates another type of report for HTML, we need to extract out an interface. Let's call that interface *Report* and the previous report class *TextReport*. That gives us:

```
public interface Report {
  public String headerLine(String name);
  public String rentalLine(int days, String title, double charge);
  public String footerLine(double totalCharge, int frequentRenterPoints)
}
```

and the new class:

```
public class TextReport implements Report {
	public String headerLine(String name) {
		return "Rentals: " + name + "\n";
	}
	public String rentalLine(int days, String title, double charge) {
	....
}
```

We need to alter the code in the *statement()* method (with no arguments) in class *Customer*, as changed in Step 15, so that a *TextReport* is created instead. Make sure that the tests still pass.

Step 17: Extend with HTML

So now we can define another class that implements *Report* to generate an HTML statement. It's time to uncomment the html test case method in class *TestMovieRental*, so that we can test it. Here's class *HtmlReport*:

```
public class HtmlReport implements Report {
	public String headerLine(String name) {
		return "<html><head><title>Rentals: " + name + "</title>
		</head><body>\n"+ "<h1>Rentals: " + name   + "</h1>\n"+
		"<table border=1><tr><th>Days</th><th>Title</th><th>
		Charge</th></tr>\n";
	}

	public String rentalLine(int days, String title, double charge) {
		return "<tr><td align=right>"+days + "</td><td>" + title +
		"<td align=right>$" + charge + "</td></tr>\n";
	}

	public String footerLine(double totalCharge, int frequentRenterPoints) {
		return "<tr><td></td><td><i>total</i><td align=right>$"+
		totalCharge+"</td></tr>\n"+"</table><p>Frequent renter points = "
		+ frequentRenterPoints +"</p>\n</body></html>\n";
	}
}
```

And then we can introduce a new method in class *Customer* to utilize this new class:

```
public String htmlStatement() {
	return statement(new HtmlReport());
}
```

Make sure that the tests all pass.

5. Deliverables

The completion criteria for this project are:

1- Complete all of the refactoring specified in steps 1 to 17.
2- Make sure that the resulting code has eliminated all of the smells that were identified in those steps.
3- Make sure that all unit tests pass.
4- Build a class diagram for the newly refactored software

So, you need to submit the following items:

- Softcopy of the refactored software. You need to properly document your software.
- The class diagram of the system

There will be a software demonstration session were all group members will be asked questions. So, all members of the group should be able to answer questions about the new program structure, and how it works.

All students are reminded that acquiring solutions from other colleagues is a punishable academic offence.

6. Useful Resources and References

Automated code refactoring CASE tools:

Many software editors and IDEs have automated refactoring support. Here is a list of a few of these editors, or so-called refactoring browsers.

- IntelliJ IDEA for Java
- Eclipse's Java development kit (JDK) (recommended for this project)
- NetBeans (for Java)
- Visual Studio (for .NET)

Further Reading

- Refactoring techniques in Fowler's refactoring Website https://martinfowler.com/

- Martin Fowler's book *Refactoring: Improving the Design of Existing Code* is the canonical reference. http://en.wikipedia.org/wiki/Martin_Fowler
- What Is Refactoring? (c2.com article) http://c2.com/cgi/wiki?WhatIsRefactoring
- Refactoring To Patterns Catalog (https://industriallogic.com/xp/refactoring/catalog.html)
- Refactoring Java Code (http://www.methodsandtools.com/archive/archive.php?id=4)
- Wake, William C. (2003). *Refactoring Workbook*. Addison-Wesley. ISBN 0-321-10929-5. https://www.amazon.ca/Refactoring-Workbook-William-C-Wake/dp/0321109295

Acknowledgement

- The Java classes used in this project are based on an example by Martin Fowler: "the refactoring differ".
- The original version of this project was developed by Dr. Ian Warren, Auckland University. However, the text of this project has partly been modified to reflect the objectives of this course.