COMP47480 Practical 5: Refactoring

Sukrat Kashyap (14200092)

17 April 2018

1 Work Done

In this practical, first, we tried to find out the long method using the JDeodrant. In that, we found the statement function in the Customer class. We extracted out the switch method calculating the rent for the car from the rental object. This switching method was violating the law of Demeter by accessing Vehicles field. Then we ran the feature envy of the JDeodrant, which gave us the recommendation to move the newly extracted method to the Rental class as it used mostly the Rental class and did not access anything with the Customer class. After this, we realised that the law of Demeter was not resolved for some part as there was still frequentRentalPoints calculation that was breaking the law. We moved this as well to the Rental class by following the extract and move method of the Eclipse IDE. This resolved the problems from the Customer class. We created two private methods for calculation of the totalRental and totalFrequentRentalPoints just to make the code clearer. Switch case was still a code smell in rental class. So, we used strategy pattern and removed the comparison of the vehicle model type by implementing concrete classes for each type of Vehicle, making Vehicles an abstract class. Then logic of calculation was in each of the concrete classes with Rental not caring about its implementation. We also implemented the htmlStatement method returning the statement in html format.

2 Reflections

A lot of refactoring was done in the practical for the code provided. Code refactoring is the process of structuring the code such that it is easier to understand, reduce its complexity, improve source code maintainability and improve extensibility. It is usually motivated by noticing code smell or when it violated or does not follow principles like SOLID for OOPs etc. In this practical, we actually solved a number of issues like long complex method, feature envy, the law of Demeter etc. The important part of refactoring is to improve design and structure without changing the behaviour. To help us give the confidence that our refactoring does not break the functionality. We already had tests written against the main class. The refactoring process is also important. I feel refactoring should be done in stages and not in one go. Stages make sure you are not breaking any previous code. It also helps in reverting back as sometimes refactoring can increase the complexity which we realise after some time spent in refactoring the code. The reasons for this mostly is because of exceptions in the business rule which makes it harder to find a uniform design to tackle the problem.

The first task was to break down the complexity of the statement function of the customer. The function was too long accessing many fields which it does not even relate to (Law of Demeter). The approach I took here was to first divide the function into many small functions.

First for calculating the rentalAmount and second for the frequentRentalPoints. After doing so we realized that the newly created methods actually do not interact with the Customer object at all and only relies on the Rental object. So we moved it to the rental class. This simple approach actually led us to remove violation of Law of Demeter as well as it reduced the complexity of the statement function. Most of the refactoring is like a chain reaction. One refactoring leads another and this way it helps cleaning up the whole code. Hence, it also makes sense to do them in stages. This refactoring was the most important refactoring for further extensions on Customer class. Since the public APIs did not change at all the test could be run in order to check for breaking changes.

Next stage of refactoring was in the rental class. The reason for this was, there is a high probability of changes in the way the points and the amount were calculated on Vehicles. Keeping all the complexity in one place leads to bugs and decreases code extensibility exponentially. Every time change in the calculation will be either the formula or addition of another vehicle. This would involve adding extra cases for switch statement and adding new fields for the model in vehicle class. The refactoring here is usually domain driven meaning it depends on the business model of the company. As the calculation is a business logic rather than coding logic. The rate of change in business logic is always more than the design, tools or the algorithm of the software. To help with this situation I thought and made an assumption that the rental is based on vehicle and days. So when rental wants to calculate the price it should actually ask the vehicle itself the cost. Now since cost has a dependency on the number of days, it is trivial matter as the rental could easily ask the vehicle the cost by giving the object the number of days. Hence, a concrete class was created for each vehicle type (getting rid of the type of vehicle as the rental does not care about type and only cares about the cost). Each class contained their own calculation of the amount for the number of days which was passed in by the Rental class. This design is also commonly known as strategy pattern.

The strategy pattern for calculating frequentRentalPoints. This strategy pattern, of course, has its own advantages and disadvantages. It was preferred in this particular situation as all cars depended on days for calculation of cost and points. If they did not depend on exact same parameters then strategy pattern might have failed. I feel assumptions are the basis on which a pattern or particular design is chosen. Here our assumption was that the car rent only depends on the days. But if it seems to change in the future refactoring would be needed again. Hence, refactoring would never create the perfect solution rather than will create a maintainable software for a certain period of time and then would require refactoring yet again.