Entry Five*:* Refactoring

Student Name: Gregory Sloggett

Student Number: 14522247

This assignment presented us with a vehicle renting program where the program calculates and prints a statement of a customer's charges at the rental garage based on the vehicles they've rented and for how long. It then calculates the charges depending on the vehicle type and duration of rental.

Work Done:

In this assignment, we are presented with three classes, a customer class, a vehicle class and a rental class. Both the rental class and vehicle class are data classes when they are given to us and contain only variables and getter and setter methods. In contrast, the customer class represents a God class, in the sense that it does all the work. The 'statement' method in the customer class is quite a long method and strikes me as a good place to start the refactoring process.

The first step taken was to extract the switch statement in the 'statement' method of the customer class. I produced a method called 'rentalCost', which is invoked from inside the 'statement' method, which produced a cost value for the rental vehicle in question.

The same process is performed on the 'frequentRenterPoints' calculation that is done inside the 'statement' method. Again, this is a calculation that is entitled to its own method and as such we use the extract method refactoring process to pull it outside of the 'statement' method.

Many of the variables within the customer class aren't indicative of what the variable holds and many of these variable names were changed as such. I didn't like the variables 'each' and 'thisAmount' in the new 'rentalCost' method. I changed these to 'rental' and 'rentAmount' respectively. In the 'statement' method, 'totalAmount' renames to 'totalRentalAmount', and the 'each' variable that cycles through the car rentals was changed to 'currentRental'. These changes are a better representation of what the variable holds, I feel.

We have cleaned up the 'statement' method a lot know and extracted some methods, but we still have a God class in the customer class, and two other data classes. I noticed when I was refactoring the names in the customer class that both of the new 'rentalCost' and 'frequentRenterPoints' methods do not use information from the customer class, and both methods use information from the rental class. It seems strikingly obvious that these methods are better equipped to be in the rental class rather than the customer class. Hence, these two methods are moved using the Move Method refactoring process. This allows us to remove variables that depended on the rental class from both of these methods.

This clears the customer class of the trait of being a God class, and also adds functionality to the rental class. The big task left, an obvious one, is to add inheritance to the system, by making the vehicle class abstract and adding three additional appropriate sub classes for each vehicle type. These classes contain the method that returns the model of the vehicle, when prompted by the abstract class.

Reflection:

This assignment offers us a wide range of refactoring examples, and it was quite tricky to decide if and where to use certain refactoring methods.

One of the dilemma's I ran into involved the decision of whether to replace some of the temporary variables with methods (Replace Temp With Query) that perform their calculations. An example of this resides in the 'statement' method on the 'totalAmount' and 'frequentRenterPoints' variables. These **could** be replaced with functions, but both of these functions would separately require performing while loops, on top of which there is already a loop in the method. This poses the dilemma of efficiency versus code smells and in my opinion, this is an unnecessary change for this example. However, I do see the potential benefits of making this change, particularly in much larger systems, where temp variables can become a serious nuisance to handle.

Efficiency was a question posed in many areas of this assignment. One particular example arose when I removed the presence of the 'thisAmount' variable in the 'statement' method. This variable held the rental cost for the current rental vehicle in the rentals array list. This was redundant because we have a method which produces this calculation in the rentals class. As such the multiple occurrences of this variable were replaced with the call to that method. However, rather than the method and calculation just being performed once within the 'statement' method, the calculation method is now being called numerous times. If this were a complex calculation, that was called many more times, this could have potentially damaging effects to the efficiency of a program. So, we are sacrificing efficiency for cleaner code. I think in this case, and for the assignment it is necessary, but this dilemma is really one that will be decided on a per case basis, as to me there is no right or wrong here.

One silly error that I ran into, but can be the case in refactoring, arose when I refactored the 'statement' method and took out the switch statement into a new method. Extracting this method meant I had to create a new variable in the method to hold the rental cost due (originally thisAmount in the 'statement' method). Upon creating this method, I made it of type INT, which sparked a pile of errors when I ran the test cases. I often make silly errors of this nature, so it shouldn't come as a surprise, but thankfully the test cases are quite extensive and thorough here and helped me to solve this issue quickly.

In a straightforward system like this these changes can be easy to correct, but in a much larger system with hundreds of variables and a lot more scope for change, extreme care must be taken when doing this refactoring. Making minor refactoring changes and testing in

between each minor change is essential to ensuring that the refactoring process flows smoothly and effectively, and further refactoring can continue. This is the essence of refactoring.

One task I wanted to achieve in this example but didn't end up doing so was replacing the switch statement using polymorphism. The switch statement here should not be based on an attribute of another object, but rather the own data of the class, which is not the case in the rental class. I feel like this should be moved to the vehicle class, but as I've implemented that as abstract it is tricky to implement this and shift the changes down the sub-class. I need more time and more deconstruction of the class to change this. I feel like another class may be necessary to implement this refactoring process, something that handles the pricing, as I did probe this concept and found that it wouldn't work as wished. For this reason, I left it with a simpler inheritance version as it stands, which I believe to be much clearer than the original version.

Overall, this assignment highlighted a variety of different code smells and refactoring issues. This is a relatively small code-base and it is important that this is remembered. Even though these refactoring changes may not be critical in this example, some of these principles have a great effect on the cleanliness and efficiency of coding in a much larger system.