Refactoring Exercise
COMP 47480

Tools Used:
- Eclipse IDE for Java;
- JUnit 4;
- It may be interesting to install a the code smell detector plug-in as well; JDeodorant[1] is a well-known one. Not essential.

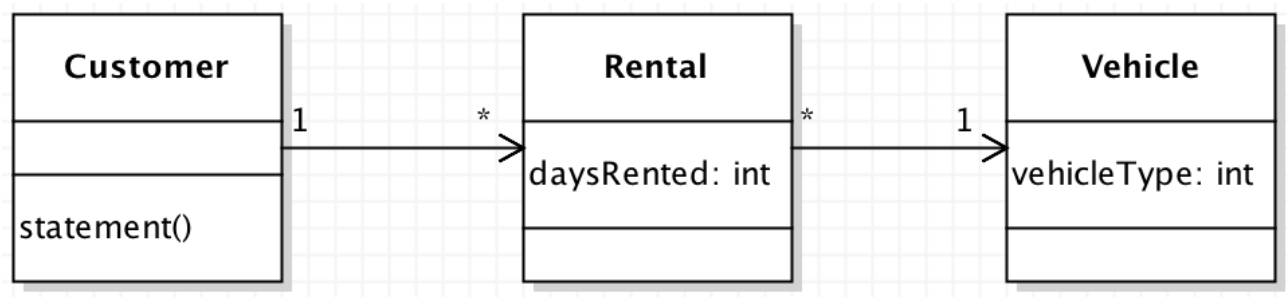
Preamble

The purpose of this practical is to provide you with a space in which to play around with refactoring. You should follow the steps below, and then experiment as you please. Write up what you did, learned and observed in your journal entry. Any questions below are issues that your journal entry should cover. Your submission artefact is the final refactored version of your program. Don't worry too much about how much refactoring you do; your journal entry is as important. Also, you are definitely not expected to fully refactor the application.

1. The Initial Application

First download and look at the Vehicle Rental example. It's a small Java program that calculates and prints a statement of a customer's charges at a vehicle rental agency. The program is told which vehicles a customer rented and for how long. It then calculates the price to be charged, which depends on how long the vehicle is rented for, and the type of vehicle. There are three types of vehicle: regular car, all-terrain, and motorbike. In addition to calculating charges, the statement also computes frequent rental points, which vary depending on the type of the vehicle rented.

This is the basic structure of the application:



The class diagram looks ok, though it already smells. Can you see what the problem is?

Spend some time looking at the source code and determine what areas are most in need of improving, i.e., what smells or principle violations or simply rotten-looking code do you see? Make note of these.

2. Future Evolution

The code works, as can be demonstrated by running the test cases. If this is all the code would ever be expected to do, then it wouldn't matter if it were smelly or not. However, a number of new requirements are planned as follows:
- The the current `statement` method returns a string. It is planned to extend the program to produce HTML output
- The classification scheme for vehicles will be changed periodically, and this will affect both the way customers are charged for rentals and the way that frequent renter points are calculated.

[1] https://marketplace.eclipse.org/content/jdeodorant

Extending the `statement` method to return HTML output can really only be achieved by writing a new method with the same structure as the current one. The vehicle classification scheme is embedded in the `statement` method and it would be a nightmare if this scheme were to change regularly. We want to refactor to code to make it more amenable to these changes.

3. Code Smells

The `Vehicle` and `Rental` classes suffer from the same code smell. What is it? What sort of refactoring comes to mind to fix these smells?

The `Customer` class has a different type of problem. What is it? It also violates some of the principles we looked at.

Taking these code smells/issues as a whole, what general direction do you think the refactoring of the program should take? In this exercise, you'll be directed to apply a few refactorings, and then you are free to refactor on your own.

4. Splitting up a Long Method: the Extract Method refactoring

The `statement` method is clearly a Long Method. There is a lot happening and it looks ugly.

Consider the `switch` statement. What is it doing? Can you think of a good name for this functionality? It is performing a single task so it's a good candidate to extract into a new method. In order to apply Extract Method to a code block, you need to consider the following:

1. Does the code block embody a coherent piece of functionality? If you can't think of a good name for it, it doesn't make sense to extract it out into a new method.
2. Consider the extent of the code block to extract. We're looking at the `switch` statement, but does it make sense the extract the statement preceding the switch statement (`double thisAmount = 0`) as well?
3. Does the code change one single variable? If so, this variable can simply be returned by the new method. If it changes several variables then it gets trickier to apply Extract Method, and perhaps this refactoring isn't useful in this context after all.
4. What variables does the code block need access to from its enclosing method? These should be passed as parameters to the new method.
5. Note that variables created and used only inside the moved code block will become local variables in the new method.

Apply Extract Method to this code block *by hand*. Do not use the IDE's refactoring! Decide what argument the new method needs and what its return type will be. Make sure the testcases work correctly after you finish.

Revert to the initial program and use the Extract Method refactoring on the Refactor menu. compare the result with your by-hand refactoring.

Try applying Extract Method to other portions of code just to see what happens (and undo the refactoring each time). Get a sense also of what its limits are, e.g. if you try to extract the `switch` statement and the following `frequentRenterPoints++` statement it won't work. Why?

5. Fixing poor names with the Rename refactoring

A new method came into existence in the first refactoring. Now is good time to look at the names and check that they make sense. Did you use a clear method name? If not, use the IDE's Rename refactoring to change it to a better one. Your method probably has a parameter called `each`. This is a terrible name; rename it to something better (hint: it represents a rental). What other names could be improved?

6. Feature Envy! Use Move Method to get the method to it rightful home

You've created a new method and reduced the Long Method smell of the `statement` method. However this new method suffers badly from feature envy. What class is it envious of? We're going to move this method to its appropriate class.

As for the Extract Method refactoring above, *perform this refactoring by hand first*. Make sure you did it correctly by running the testcases. Then undo your work and use the Move Method refactoring in the Eclipse IDE to perform the move.

Eclipse will ask if you want to keep the original method as a delegate to the new one. Think about this. Which option do you prefer and why? You could try both versions of the refactoring and see which result you prefer.

When you've performed this refactoring check that the testcases work ok. Should you add some new testcases now?

Look at the new method it its new class. Should you rename it to a better name? It still contains a comment. Does this comment serve a useful purpose, or would you rather update/delete it?

7. Now Refactor Mercilessly (don't worry if you don't get far with this)

Continue refactoring the program and playing around with some of the refactorings we looked at in lectures. Some hints as to what you might do:

- The `statement` method has a local variable you might get rid of.
- The `statement` method has two statements that update the `frequentRenterPoints` variable. These could form a cohesive method on their own. What class should this method be moved to?
- The computation of `totalAmount` in the `statement` method stretches across the entire method. We've all coded like this, but it is clearer if this computation is moved to a new method and invoked once in the `statement` method. This type of refactoring is called Replace Temp With Query. The resulting code is much clearer, though less efficient[2].
- The same refactoring as in the previous bullet can be applied to `frequentRenterPoints`.
- (The previous two refactorings added two new methods to the Customer class. Should they be made public or private?)
- You can now create the `htmlStatement` method to output the customer statement as HTML with minimal copying of code[3]. This is an example of how refactoring a program can make it it amenable to being extended with new requirement.
- The new method created in (4) above switches on a type field in another class. This is awful. Move this method to the class that owns the type field and then consider using subclassing and polymorphism to get rid of the switch statement completely.
- We were warned in (2) above that the rules around price and frequent renter points are likely to change often. This suggests that they should be packaged off into a class on their own. Again, a design pattern, namely *Strategy*, is very useful to build this flexibility into the code.

If you make much progress in this section, well done! The later bullet points about are just there to give you a notion of how the refactoring might progress, and the way design patterns (the final topic in the module) often have a place in refactoring.

---

[2] In most applications, clear code is far more important that efficiency. Code clearly first, then if necessary use run-time profiling to improve performance where it matters.

[3] The code duplication could be reduced entirely using a design pattern called *Template Method*.