Andrea Nabil Belahouane
nabil.belahouane.17@ucl.ac.uk
Student Number: 17023334

The task:

Our client's request was to change a legacy code base and implement a new functionality whilst maintaining the original code's features.

The new functionality was that the code must have the following: "The new rule is that if you enter the zone before 2pm, you are charged £6, and you can stay in the zone for up to 4 hours. If you enter the zone after 2pm, you will be charged £4. If you leave and come back within 4 hours, you should not be charged twice. If you stay inside the zone for longer than 4 hours on a given day then you will be charged £12."

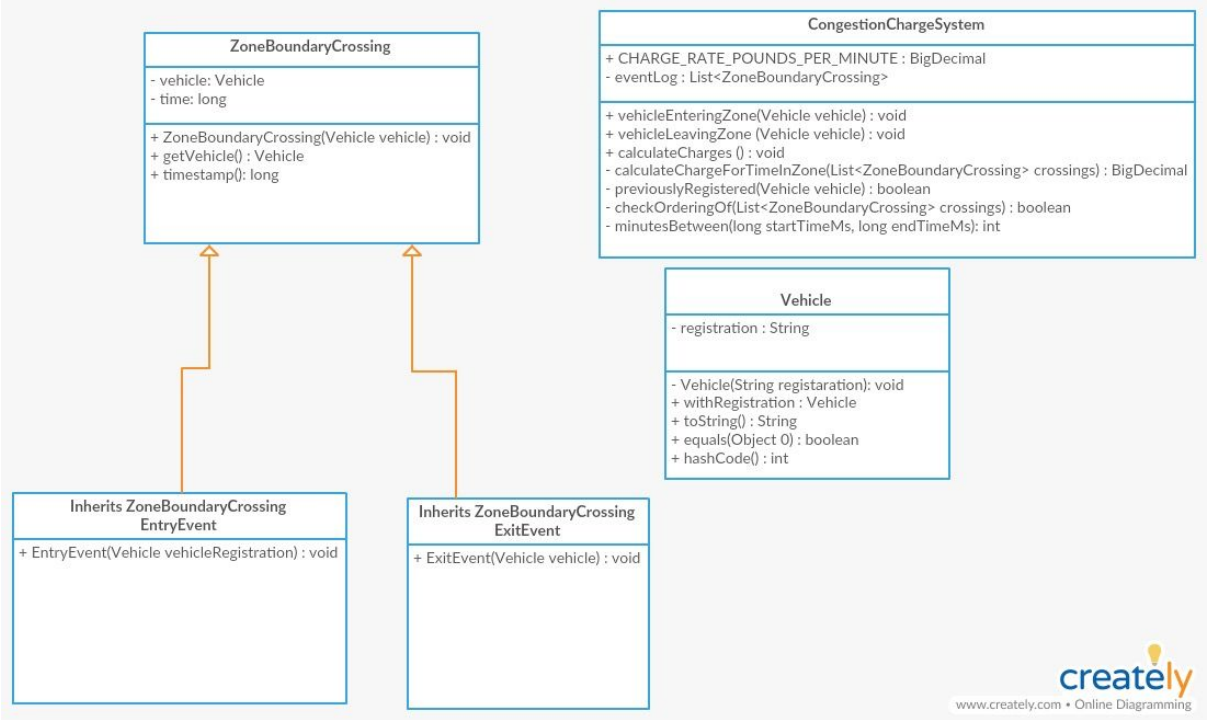
Approach:

Our approach to the task was split it up into multiple steps as shown below:

1. Analysis of the code and its functionality and drawbacks
2. Refactor the code to remove unnecessary duplication and make the code easier to read and test.
3. Unit test the current code to ensure that it works the way it is meant to
4. Implement new functionality
5. Unit test the new functionality and ensure all possible test cases are covered

Code Analysis

In order to develop a greater understanding of the legacy code base we set out to create UML diagram:



We saw that the **template method** was already being implemented by the ZoneBoundaryCrosssing Class but the CongestionChargeSystem could be improved upon by allowing it become more testable and more modular.

Code refactoring

The CongestionChargeSystem class had no constructors which meant that it was hard to test, to resolve this we added a package private constructor which allowed us to pass in mock objects and a custom eventlog. Then we also added package private constructors to both the entry and exit event classes.

We decided to keep the **template method design pattern**, as it saves us code duplication by categorising Entry and Exit events as ZoneBoundaryCrossings.

In the Congestion Charge System class we removed the hashmap from CalculateCharges function and placed it within the scope of the class this allowed us to test the CalculateCharges function.

For the Vehicle class we made an interface for it and replaced all references to the vehicle class with a reference to the vehicle interface, this allowed us to follow the strategy design pattern, the reasoning behind implementing the **strategy design pattern** here was that it allows us greater flexibility for future usage where a client may ask us to create different set of rules for different types of vehicles.

In the Congestion Charge System class we replaced all calls to the external code base to calls to adaptor classes (CustomerAccountsService and OperationsService), we then

created interfaces for the adaptor classes so we could the mock the classes for testing purposes.
Our motivation for doing this, was to implement the **adaptor design pattern** so that mocking calls to the external code base would be possible and testing would be possible.

As a general rule, we created interfaces for classes we wanted to mock.

Unit Testing before implementing the new feature

For the testing of the original code base we used JMock to interact with the interfaces and pass through our conditions to the original code base. We then used asserts to check if the value of the charge was calculated properly. Below is an example of the code.

```
   public void testCalculateChargesOneEntryWithTimeDiff() throws
Exception{
       context.checking(new Expectations() {{


exactly(1).of(mockAccountsService).deductChargeFrom(with(equal(testVehicle
)),
with(Matchers.comparesEqualTo(congestionChargeSystem.CHARGE_RATE_POUNDS_PE
R_MINUTE.multiply(new BigDecimal(30.00)))));
```

Furthermore, we checked for other errors that could occur. Such as two consecutive entry events and exit event or an inconsistent record of entries and exits because a vehicle did not enter or exit.

Implementing the new functionality.

The instructions given to us by the client were a bit vague and as a result of this there were many possible meanings of the instructions.

Two meanings which stood out were, a car could stay for a period of 4 hours and it does not have to be in one consecutive 4-hour time block and the other meaning was that once a vehicle enters it has a 4-hour consecutive time block till it gets charged the £12 rate.

We decided to go with the latter meaning as this would fit with the clients requirement of controlling congestion in the city centre.

So the new method of calculating charges would work by charging £12 if the total time exceeds 4 hours, entry before 14:00 and a less than 4 hour stay would result in a charge of £6 and an entry after 14:00 and a less than 4 hour stay would result in a charge of £4.

In order to implement the new functionality we created a *timeFromEpochToHourOfDay* function which worked as a modular helper function to the Calculate Charges function.

```
private double timeFromEpochToHourOfDay(long millis){
    SimpleDateFormat sdf = new SimpleDateFormat( s: "HH:mm");
    Date resultantDate = new Date(millis);
    String[] timeOfDay = sdf.format(resultantDate).split( s: ":");
    return (Double.parseDouble(timeOfDay[0])+(Double.parseDouble(timeOfDay[1])/60.0));

}
```

We did this so future developers would more easily understand how our code was functioning.

Unit testing the new feature

The Unit testing of the new functionality was more difficult as we had to consider all possible cases that may occur with a crossing vehicle.

Some problems that could occur were the following:

- What to charge if the time of entry is exactly 14:00? (£6 or £4)
- What should be done if a vehicle enters from 05:00 then exits at 06:00 then returns at 11:00 and leaves again at 12:00? Should they be charged £6 twice or not.
- If a person enters at 13:00 and then leaves at 14:00 and then returns at 14:30 and leaves at 15:00 do they get charged £6 and then £4 or just £6?
- A more pedantic case, entry at 00:00 exit at 00:30, then entry at 04:00 and exit at 04:30 and then entry at 08:00 and exit at 08:30 and finally entry at 12:00 and then exit at 12:30. Are they charged £24 or £6?

Some examples of our unit tests:

```
@Test
public void testCalculateChargesInterFourHourIntervalCrossingsAfter2PM() throws Exception{
    context.checking(new Expectations() {{
        exactly( count: 1).of(mockAccountsService).deductChargeFrom(with(equal(testVehicle)), with(Matchers.comparesEqualTo(new BigDecimal( v: 8.00)) ));
    }});
    addCrossingEvent( EntryHour: 14, ExitHour: 14.5,testVehicle);
    addCrossingEvent( EntryHour: 17.8, ExitHour: 18.5,testVehicle);
    congestionChargeSystem.calculateCharges();
}

@Test
public void testCalculateChargesMultipleCrossingsUnderFourHoursOverFourHourIntervals() throws Exception{
    context.checking(new Expectations() {{
        exactly( count: 1).of(mockAccountsService).deductChargeFrom(with(equal(testVehicle)), with(Matchers.comparesEqualTo(new BigDecimal( v: 12.00)) ));
    }});
    addCrossingEvent( EntryHour: 14, ExitHour: 14.5,testVehicle);
    addCrossingEvent( EntryHour: 18.1, ExitHour: 18.5,testVehicle);
    addCrossingEvent( EntryHour: 22.1, ExitHour: 23.5,testVehicle);
    congestionChargeSystem.calculateCharges();
}
```

The above tests were for the CalculateCharge function but we also had tests for EntryEvent ExitEvent and Vehicle class so we could achieve more coverage.

Code optimisation

We removed the eventLog in the Congestion Charge System class and instead we directly added objects into the hashmap. This meant that we now dealt directly with eventLogs for each vehicle separately, whereas earlier we had to sort the eventLog by vehicle before calculating charges. This saved us some time complexity and it also made the code easier to understand.