

Matthew Barlow (160075342)  
Nkosi Garcia (160138186)  
Rajan Gopalji (160072570)  
Rebecca Holmes (160041808)  
Hannah Miller (160038473)  
Tom Woodcraft (160086832)

**CS1410**

**Java Programming Development Group Report**

2017

**Level 1 submission.**

**Section 1. Description of the design**

Our design consists of the following classes: vehicle, car, motorbike, sedan, truck, queueable, pump, till, station, simulatorView, simulator, ticker and stationGUI.

In our design, we have a vehicle super class which has subclasses called car, motorbike, sedan and truck. Our decision behind making the car, motorbike, sedan and truck classes into subclasses was because they would be very similar to each other and we decided we did not want to repeat code as this would make it higher risk that we made mistakes. Also, if we had to change one component in one transport class we would have to repeat this change in every transport class therefore making the code unnecessarily longer.

The vehicle class is responsible for containing the variables and methods that are mutual to all the transport methods such as a tank size variable, probability that they shopped, the amount they spent in the shop if they shopped and the space each vehicle took up in the queue variables as examples. In the transport method classes, each of these variables had been set to the specific values stated in the brief. This ensured that we did not have to repeat several static field variables in each class and instead could declare them all in the vehicle class making the design slicker and less repetitive.

All the fields in the vehicle class are protected because as vehicle is a superclass, we needed to ensure that all the subclasses could access them however the other classes were not able to, to avoid confusion and errors in the program.

In the constructor of each transport class we included a parameter of type station to enable the vehicle objects to know which station they were going to as before we did this it was causing us problems and this seemed to be the best way to overcome it. There is only one method in the mode of transport classes making them very small and well organised classes. This method’s responsibility was making sure that when the vehicle was generated, all the values necessary such as the probabilities were set correctly for example the shopping probability, tank size and the space they took up in the queue. An example of this in the brief was if cars filled up in less than 5 minutes they had a 0.3 probability that they would shop for 2-4 minutes and spend £5-£10. This resulted in us having to have 2 random generators to generate the time they shopped for if they decided to shop and another one for the amount they spent in the shop and these values were generated in the method generate.

The only difference between these modes of transports and the truck class was that the truck class had extra field variables and methods since we had to take into consideration if they were happy drivers or not and the resulting effect it had on the simulation.

In the truck classes, we had an extra field variable which was of type Boolean called isHappy and the value of this changed according to the refill time of the truck. We included if statements in this class to determine whether the Boolean remained false or changed to true based on the random variable tank size and refill time. We also included methods to control how much the probability of trucks arriving (which was t) changed due to the previous driver’s happiness or not. These were simple protected static methods that determined how much the probability would increase or decrease. Another method we had to include in this class was a public void changeProbability method and this was responsible for the actual alteration of t, based on the Boolean isHappy method. To make it clear for the simulation we wrote an output of the result of the method letting the user know if the trucks were happy or unhappy to avoid ambiguity.

The vehicle class with the car, motorbike, sedan and truck classes are library components since they can be reused for other scenarios and can be edited to change the overall running of the application

There is also another superclass called queueable which has subclasses called pumps and tills. We decided to add this class since both the pumps and the tills contained queues so by having a superclass for the queue objects where the queues are made, it avoided repeating code and having unnecessary code. Queueable was made into an abstract class as because it is only used for till and pump to get the methods for it, to reduce code repetition. The queueable class created an array list of objects of type vehicle that acted as the queue for the pump and till every time it was initiated. This queueable class is responsible for everything that a queue is responsible for such as getting the current length of the queue by counting the number of vehicles in the array and using their queue space values, and checking to see if there is space in the queue by counting the number of vehicles and their designated queue space and returning true or false depending on whether there is space for another vehicle or not.

In this class, we had to use a caller parameter depending on whether the queue for the pump was to be changed or the queue for the tills was. An example of this is the addTo method where if the user calls pump in this method it will get the queue space of the vehicle to be added and will add it to the length of the queue at the pump or if the user calls till it will simply add one to the till length regardless of which type of vehicle is at the pump. Another use of this caller parameter is for the removeF method which updates the length of the queue depending on which queue is to be updated. This method is used when the customer has finished at the till after they have filled up. We decided to add this feature in so the code knew which queue we either wanted to change the length of or wanted to know how long it was and this helped during the testing section of the coursework.

The pump and till classes are very small as all the methods they need to use are in the queueable class. In order for the program to know which queue is to be updated the super class’s methods are either called via the pump or till class which makes it easy to know, especially when testing, which queue should do what when. The pump and till classes only have two methods which are used to remove the first item in the queue to add vehicles to each queue. The reason these methods are repeated in these classes as well as the superclass is because for example when till.addTo method is called it knows to use the parameter “till” when it uses the addTo method in the queueable superclass. We decided to do it like this since it makes the code tidier and without this feature in the superclass if the parameter caller is forgotten then the station wouldn’t know which queue to update.

The queueable class along with the tills and pumps are also library components for the same reason that the vehicle classes were since they can be adapted for any situation.

We have also made a ticker class which is responsible for the timings of the simulation. This class is very simple but was necessary for the simulation to function. The methods are simple return methods such as returning the current tick and the max amount of ticks which in our case was 1440 ticks. Also, a reset ticks method which reset the ticker to zero. We also added an increment method which adds one to the tick for every second the simulation was ran and stops at the max number of ticks. Creating the ticker to have the sole responsibility of keeping track of the timing reduced the coupling and increased the cohesion of the design which is what is best for the code.

There is a station class in which the constructor is used to set up the layout of the petrol station by inserting values in the parameter. These values are number of pumps, number of tills, value of p and q and whether trucks are present in the simulation or not and these are the values the user of the GUI can enter. In this class, there are two array lists that are made which consist of pump and till objects which are used when a vehicle is added to each. In this class, we have two similar methods called scanTillsForChange and scanPumpsForChange we decided to add these methods to iterate through every vehicle in the pump and tills and to ask them if they want to anything on that tick. This is the class in which all the vehicles are generated based on the probability of p, q and t, this happens in this class since after they are generated they are added directly to the pumps so thought this was the best place to have them generated. We also have a series of methods which control the flow of vehicles to the pumps and tills such as remove from shop which calls the method from till to remove the first item and leave because queue is full which is called upon if the length of the queues at the pump is above three units of space so the vehicle cannot fit and this is where the amount of money lost is calculated. There are also two Boolean methods called addVehicleToPump and addToTill which are responsible for checking whether or not the vehicles can be added to the pumps and tills based on the existing queue length, the size capacity of the hopeful vehicle and the shortest queue. We have these methods in the station class because since they are most closely related to the station section which reduces coupling if they were in a different class.

The simulator class is where all the documentation is created after the simulation has been run. This class is responsible for printing out the end result of the simulation telling the user how many vehicles of each transport were created based on their input of p and q. This class also output the total profit made and the total profit the station could have made due to vehicles not being able to fit in. We have done this so it makes it very clear for the user to see how well or not the petrol station is doing in terms of money and they would easily be able to see if having trucks is profitable for the station or not. The simulator class is client code since it uses every other class in order to function, therefore this is specific to the design of the other classes and would not be reused for anything other than this brief.

Throughout the project we kept considering using interfaces for the station classes but decided against this since we didn’t think it was necessary for the coursework brief as there was only one petrol station and all methods to do with the station were contained here. In the future, we could add an interface in case we need to make multiple petrol stations.

Since we have done the level one version of this coursework we have added a GUI as well as a text based console. Since in the brief it states that the user must be able to choose the probability of p, q, whether trucks are present or not, the number of pumps, the number of tills, the price of the gallon, and the period of time the simulation should run for, the team mates responsible for the GUI used labels, text fields, sliders, buttons and radio buttons to enable the user to enter all of the values they required. As we wanted the GUI to be as easy to use as possible there is a help section included where the user can go for help if necessary.

We chose to design the GUI in this way to make it as simple and user friendly as possible. It needed to be easy to look at and understand, whilst also being functional and useable. The combination of different Swing elements used has allowed us to do this as the user can easily set the values they want to use and run the simulation. The use of an extra pop up window when the help button is pushed ensures any confusion can be clarified and explained. This is also why we felt the importance of putting this button first, it should be the first thing the user sees when looking down the GUI. The simulation window itself is separate as not only did we feel this would make it clearer by keeping the different parts of the GUI separate, but it also allows for multiple GUIs to all run simultaneously from the same, central window. The results menu is triggered by the simulation ending. This again is on a separate window as it remains consistent with keeping the GUI clear and simple, but also gives the user a clear indication that the simulation has finished.

**Section 2. Changes to Support Other Simulations**

More Types of Vehicles:

In order to add more types of vehicles to the simulation, a subclass of the Vehicle superclass would be created and all the inherited fields will need to be set to appropriate values for that vehicle in a generate method as well as a probability of it appearing. The generate method should be called in its constructor so that vehicles of this type can be generated in the simulation. In the generateVehicle method of the Station class, an if statement should be added to generate a random number to instantiate an object and should be written in the same syntax of the other if statements in that method.

Multiple Types of Fuel with Different Prices:

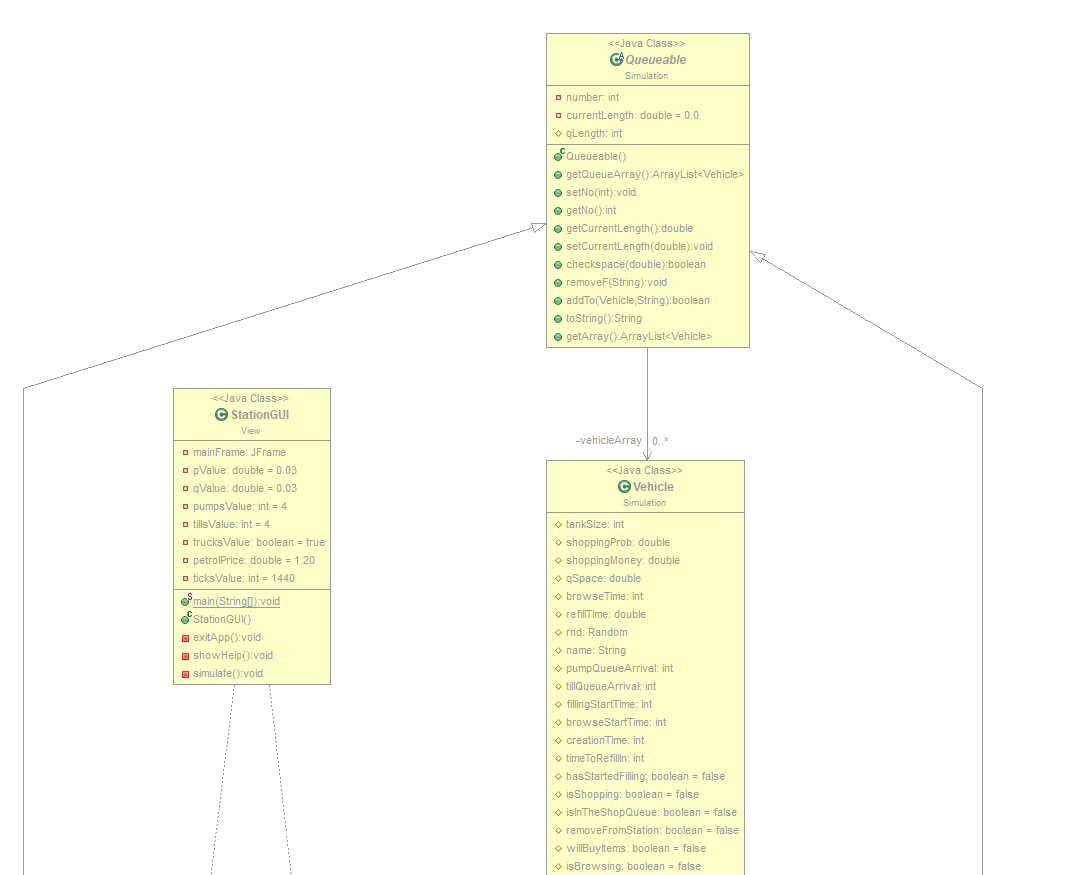
To add multiple types of fuel with different prices, a fuel class could be created with various subclasses of fuel types. A static field of type double, an accessor method and a mutator method would be added to the Station class allowing the price to be set or accessed. In the main method inside the Simulator class, it should call the mutator method created and the price of the fuel should be set in the parameter.

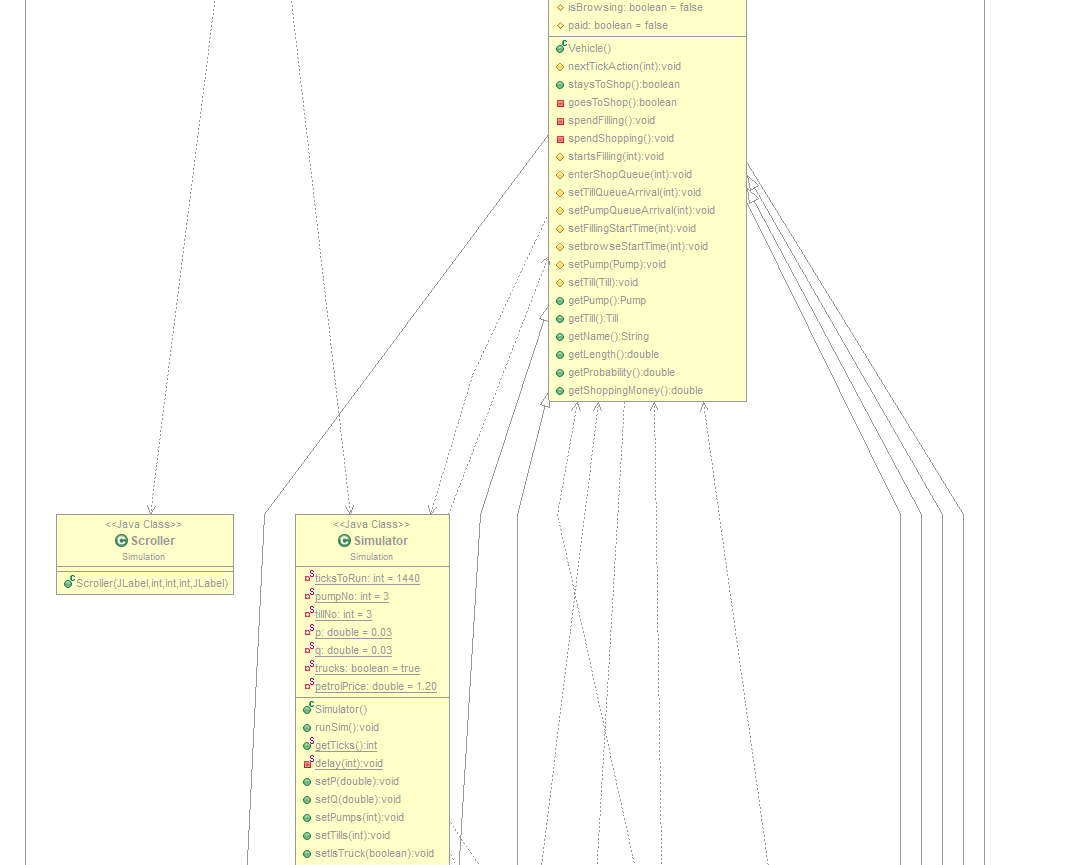
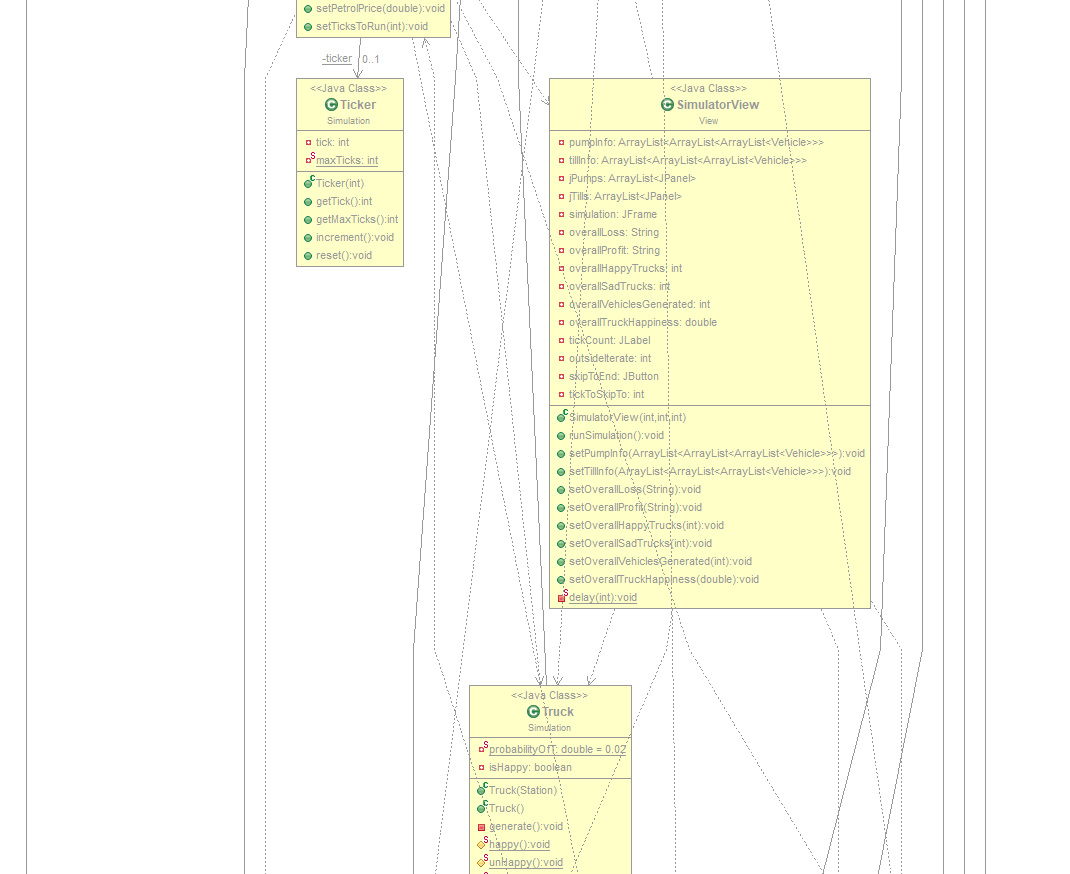
Parking Away from Pump during Shopping:

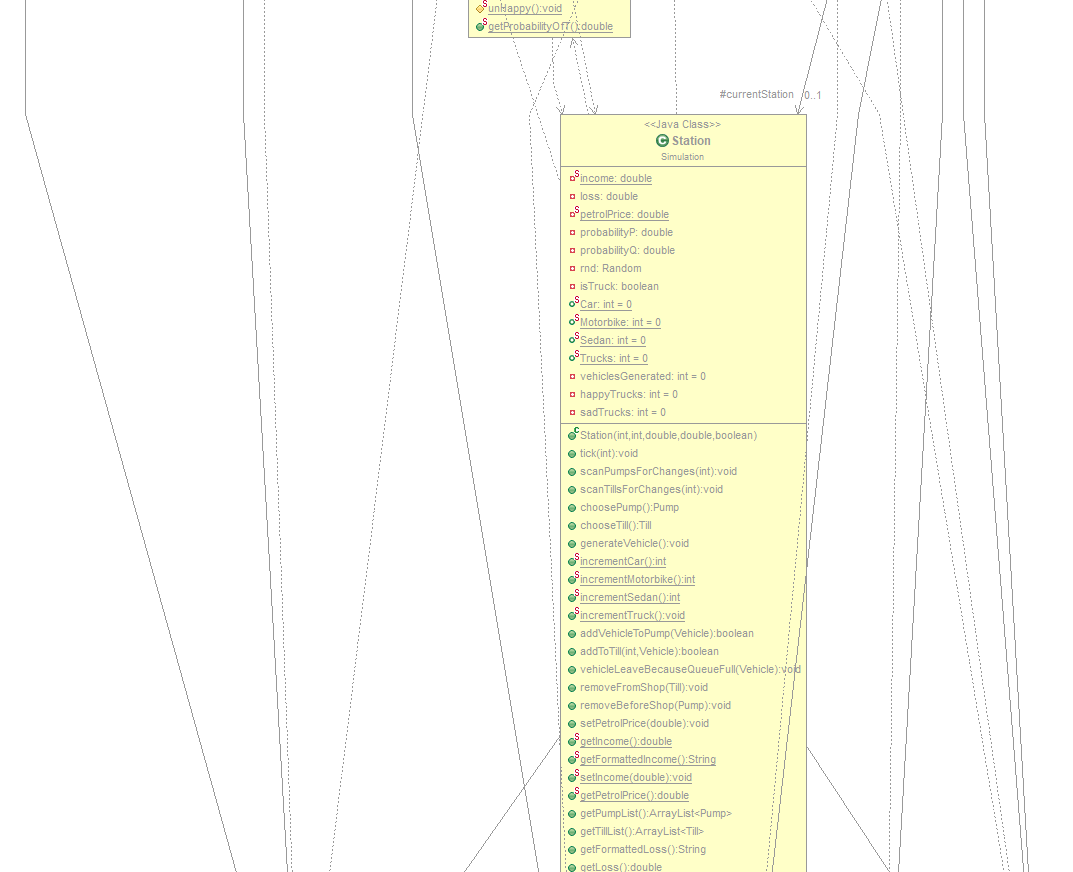
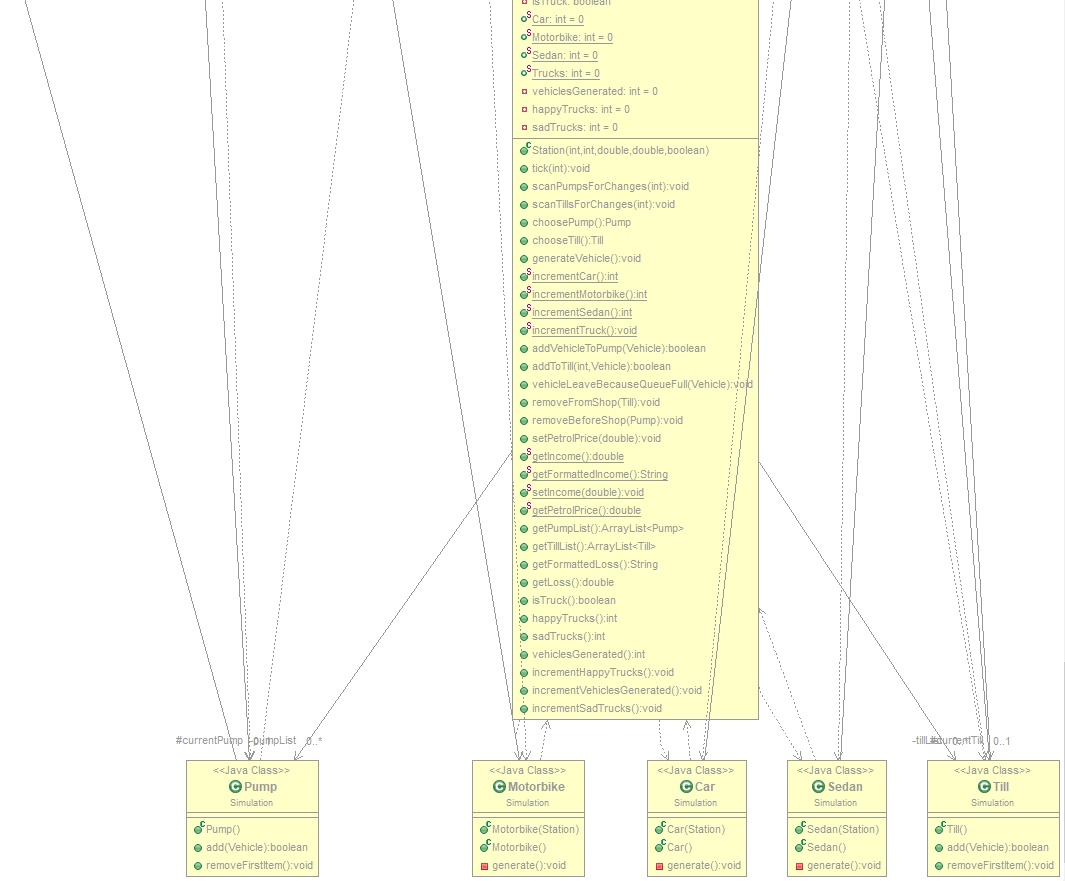
If the vehicle was to be parked away from the pump whilst shopping, a new array list in the Queueable class could be created, storing all the vehicles that are currently parked. The array list could have a maximum value of items representing the number of parking spaces there are and each vehicle parked occupies one space.

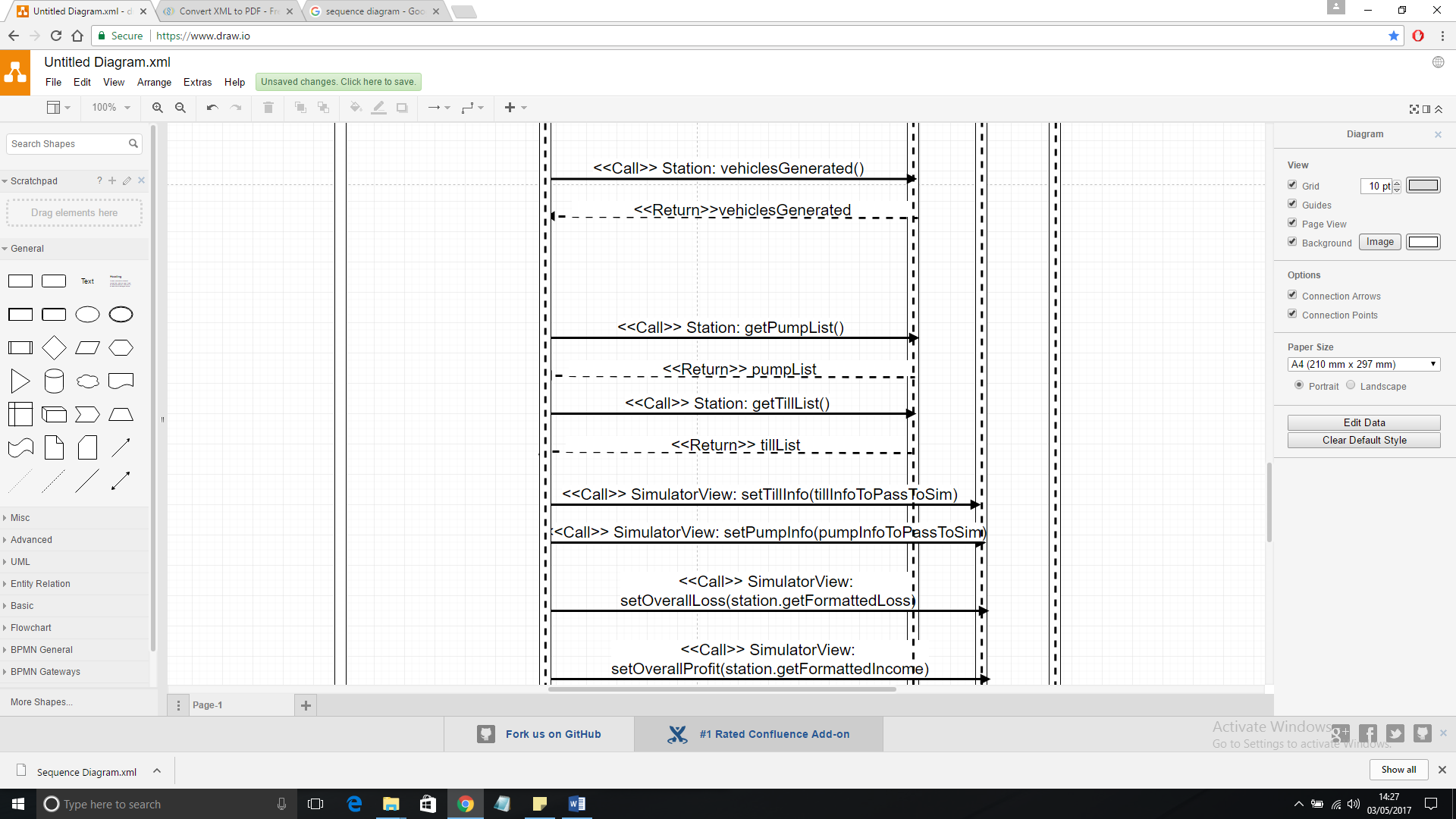
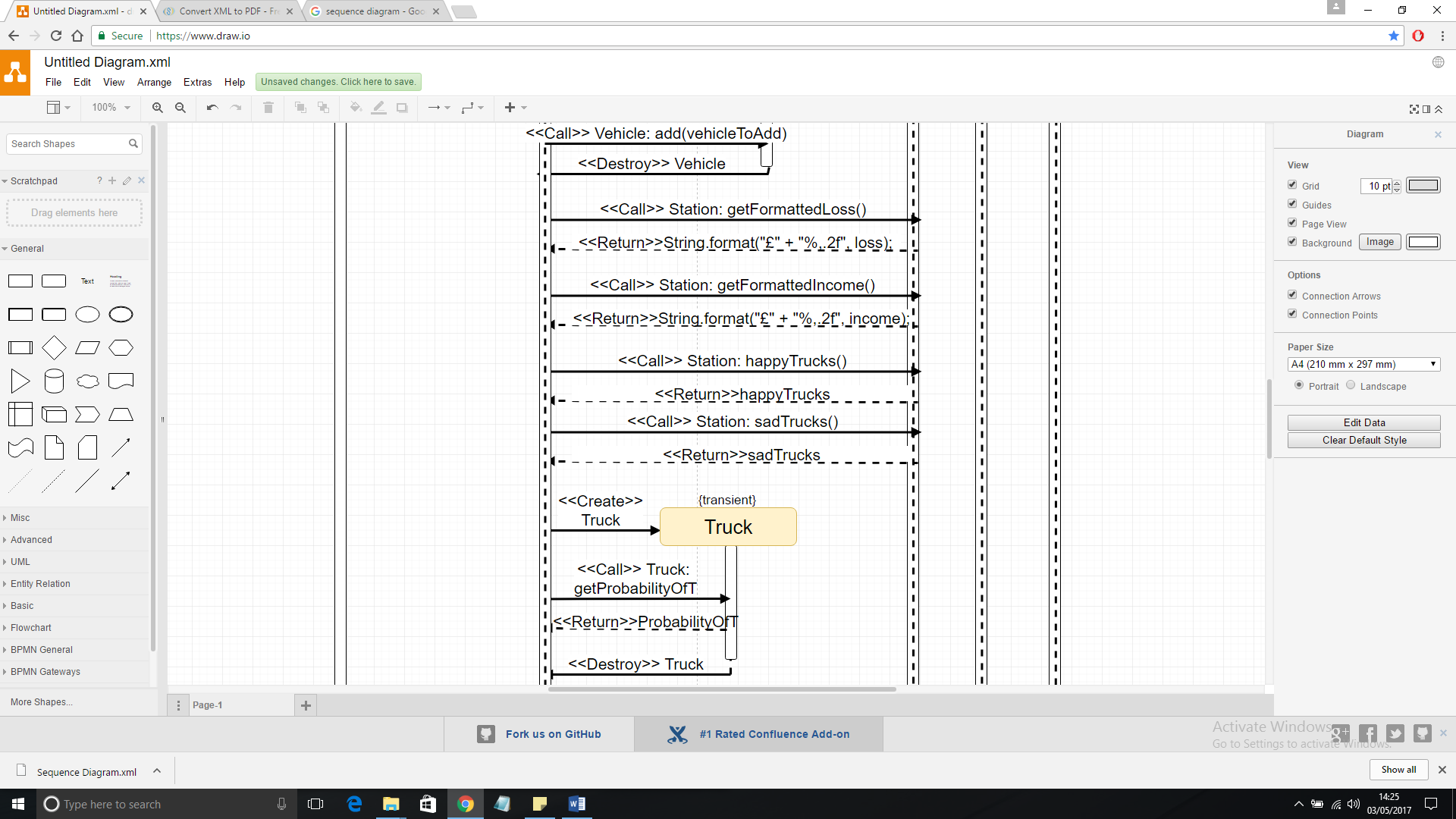
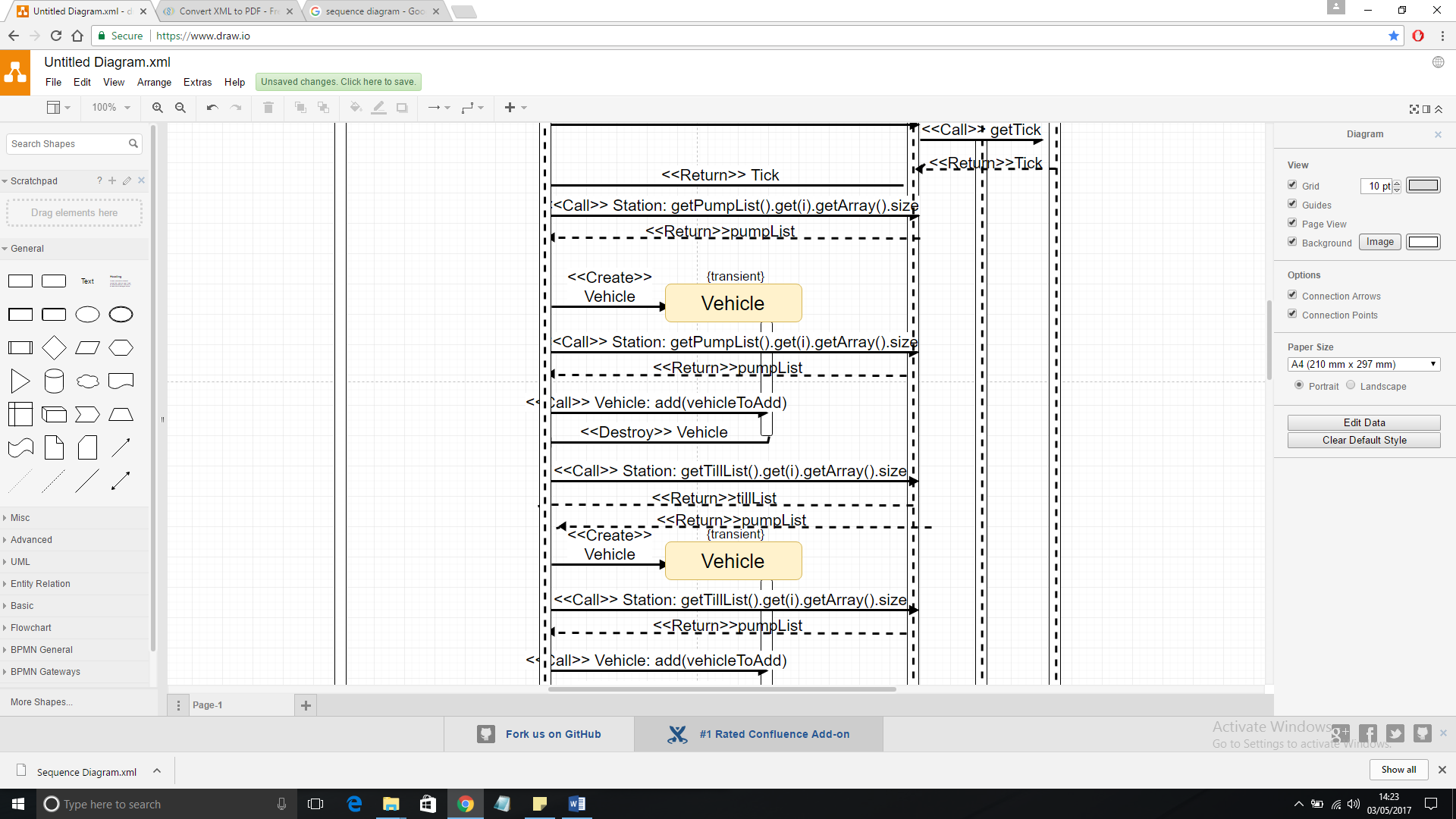
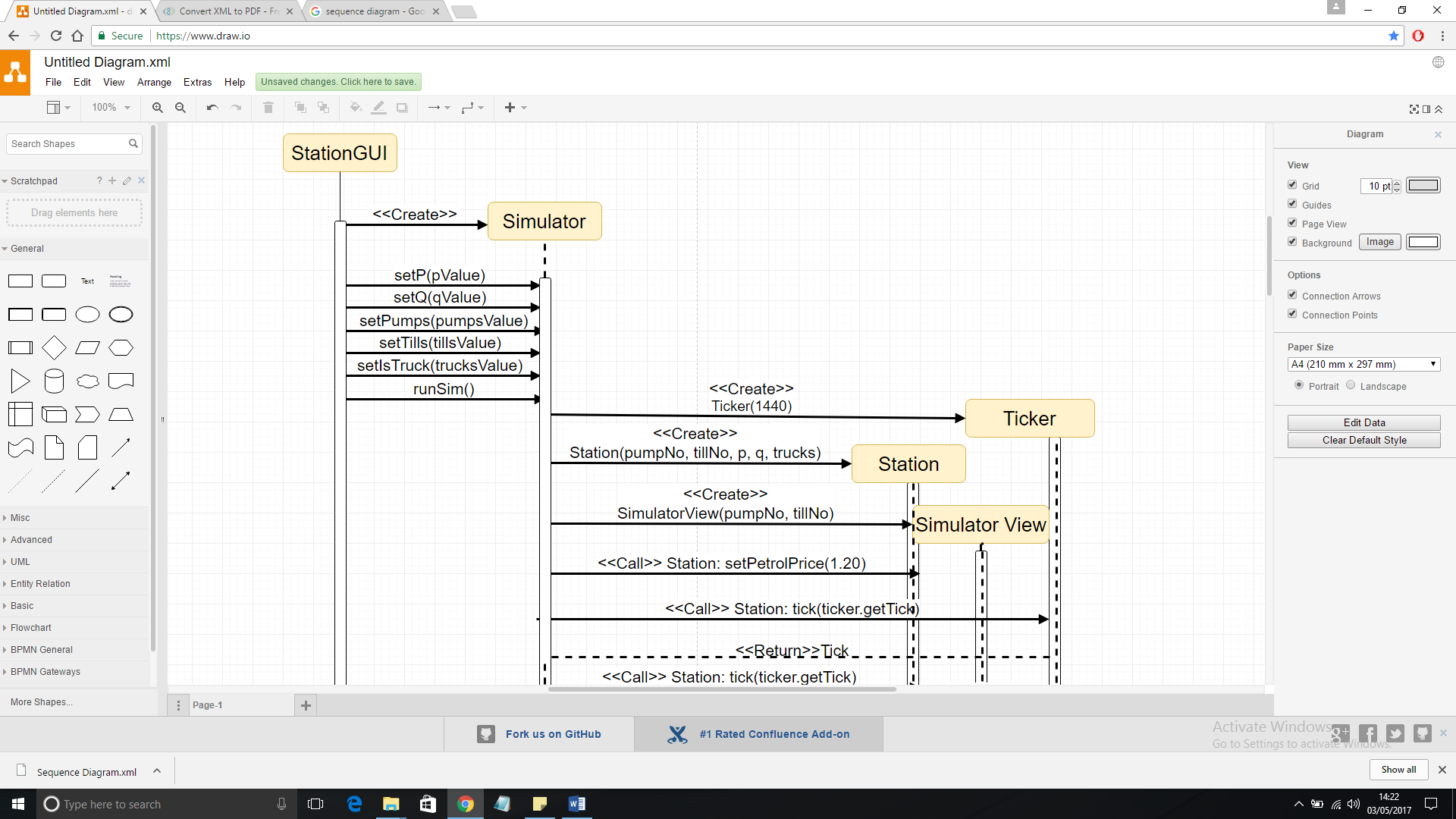
Vehicles Breaking Down during Simulation:

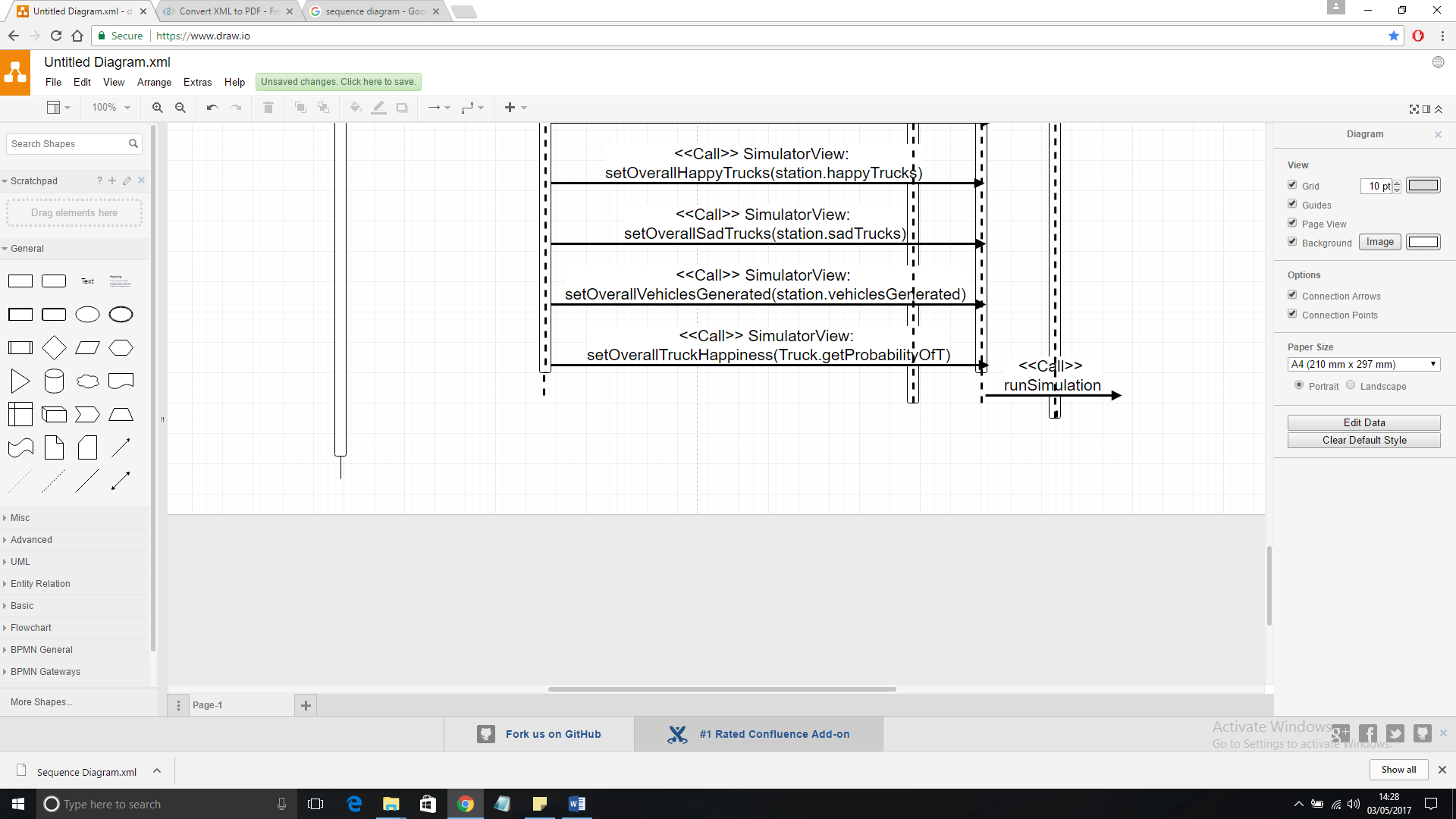
If the simulation was to include the possibility of vehicles breaking down, a probability and time would have to be created and stored inside the Station class. If the vehicle breaks down in the queue or whilst filling up, then the vehicles behind it in the array would have to move to a different queue which is the shortest and prevent any incoming vehicles from entering that queue array. This can be done by writing a method in the Station class that moves the vehicles in the array to the next shortest one and reduce the array in which the vehicle has broken down to a size of 1 temporarily until the broken-down vehicle leaves.











**Section 5. Results of Simulation in Tabular Form:**

p = 0.01, q = 0.01, pumps = 2, tills = 2, trucks = false

|  |  |
| --- | --- |
| Money lost | £29.80 |
| Profit made | £863.00 |
| Number of happy trucks | 0 |
| Number of sad trucks | 0 |
| Number of vehicles generated | 53 |
| Number of Cars generated | 13 |
| Number of Motorbikes generated | 20 |
| Number of Sedans generated | 20 |
| Number of Trucks generated | 0 |

p = 0.01, q = 0.01, pumps = 2, tills = 2, trucks = true

|  |  |
| --- | --- |
| Money lost | £800.20 |
| Profit made | £2,078.20 |
| Number of happy trucks | 22 |
| Number of sad trucks | 1 |
| Number of vehicles generated | 84 |
| Number of Cars generated | 26 |
| Number of Motorbikes generated | 12 |
| Number of Sedans generated | 11 |
| Number of Trucks generated | 25 |

p = 0.04, q = 0.02, pumps = 1, tills = 4 trucks = true

|  |  |
| --- | --- |
| Money lost | £2,315.40 |
| Profit made | £1,390.20 |
| Number of happy trucks | 6 |
| Number of sad trucks | 2 |
| Number of vehicles generated | 149 |
| Number of Cars generated | 55 |
| Number of Motorbikes generated | 39 |
| Number of Sedans generated | 22 |
| Number of Trucks generated | 33 |

p = 0.03, q = 0.04, pumps = 4, tills = 1 trucks = true

|  |  |
| --- | --- |
| Money lost | £208.80 |
| Profit made | £4,788.40 |
| Number of happy trucks | 25 |
| Number of sad trucks | 4 |
| Number of vehicles generated | 199 |
| Number of Cars generated | 48 |
| Number of Motorbikes generated | 41 |
| Number of Sedans generated | 78 |
| Number of Trucks generated | 32 |

Average over 4500 runs:

Income: £2340.38

Loss: £1020.36

What row money lost implies is how much money could have been made if it wasn’t for vehicles not being able to queue up for a pump as all pumps it was full. From this we can optimise the number of pumps and tills to minimise loss.

The profit made row implies the amount of money that contributes to financial gains after factoring cost of fuel. Using this value in conjunction with money lost we can maximise profits by altering the number of pumps and tills to serve more customers and to possibly adjust fuel prices for added income.

The number of happy trucks implies the number of drivers that entered the station to fill up were happy with the service as they could refill their fuel within a given time (8 minutes) whereas unhappy drivers had to spend longer than that to refill their fuel.

The number vehicles generated shows us the number of vehicles that were created and sent to the station, whether they entered the station or left before queuing. The next three rows show the number of vehicles generated for each specific vehicle which implies the type of drivers the station attracts. From this data, the owner can adjust the items in the shops for example, trucks, when enabled, frequently fill up their tanks a majority of the time compared to other vehicles and as they always buy things from the shop, so the owner should have items that appeal to truck drivers more so that they spend more money. Also, could find ways to better the service for those truck drivers as they tell other truck drivers about the service which will attract more customers.

However, we can see from the results, especially in the first two tables, that trucks aren’t beneficial for the business as overall loss was higher when they were involved, shrinking the profit margin. This could be because of their large tank size which takes very long to fill and the unit of space they take up in the pump meaning other vehicles can’t fit in the queues which forces them to leave. This is where money is lost and the correct configuration of pumps and tills to prevent this is:

Optimal layout: 4 pumps, 2 tills and no trucks = maximum profit.

(given that p and q are both 0.04)

**Section 6. Building and Running the Program**

Once the project has been opened in your preferred development environment (we recommend Eclipse), the main class needs to be ran as a Java Application, this is “StationGUI”. This will create the initial menu screen from which you can set preferences such as the probabilities of when cars and motorbikes spawn, as well as several other options. Acceptable values for each field are specified within the help section which can be displayed by pressing the help button. The help section provides the user with a comprehensive guide for using the simulator. It briefly explains the function of the simulator, followed by a clear step-by-step guide for running the simulation with your own specified values. It lists the different fields that can be edited, what they do, and the data types that need to be entered for it to be valid and therefore used when submitted. It also explains the further functionality of the simulator and what is done in the event of invalid values being submit. Once the custom values have been entered (optional), the user should press the “Submit Preferences” button. If the values entered are valid, then the variables used to create the simulation are set to these and will be used for the simulation. If any of the values entered are invalid for this field, then they will be ignored and the default values will be used. Any valid values will still be used alongside the defaults. The user then needs to press the “Simulate” button which will cause a new window to open where the actual simulation will take place and be displayed. The simulation will run for the specified number of ticks and then when it is finished another window will open in front of it detailing the simulation results. During the simulation, there is a “Skip To End” button located at the bottom of the window. This can be used to skip to the end and display the results (the whole simulation doesn’t have to be watched first), this is especially useful for longer simulations if the user is looking to save time. When the simulation is finished and the user is satisfied, they can close the results and simulation windows and then either create a new simulation or exit the program. Creating a new simulation is done in the exact same way. If the user chooses to exit the program, then there is a “Quit” button next to “Simulate” button, pressing this will cause another window to pop up, this time asking the user if they are sure they wish to close the program. They will have an option of either “Yes” or “No”. Clicking “Yes” will close the program properly, clicking “No” will close the box and the user will be returned to the option menu.