*Petrol Station simulation Report: Group 11*

Aston University Engineering and Applied Sciences CS1410: Java Programming Development

*Perkins, Tristan*

*Minguell, Kelvin*

*Bartosevicius, Matas*

*Ramalho, Milton*

*Lees, Jordan*

*Aston University*

Table of Contents

[Distinguishing between Library Components and Client Code Data 3](#_Toc481640794)

[Packages 3](#_Toc481640795)

[Separation of Model-View Structure 3](#_Toc481640796)

[Code Layout 3](#_Toc481640797)

[(Level 1) Required Changes for the Library Classes to Support Simulations 3](#_Toc481640798)

[More types of vehicles 3](#_Toc481640799)

[Multiple types of fuel with different prices 3](#_Toc481640800)

[Parking away from the pump during shopping 3](#_Toc481640801)

[Vehicles breaking down during the simulation 4](#_Toc481640802)

[UML Class Hierarchy 5](#_Toc481640803)

[(Level 1 only) Sequence Diagram for a Scenario in the Simulation 5](#_Toc481640804)

[The results of the simulations in tabular form; 6](#_Toc481640805)

[Results and Implications 6](#_Toc481640806)

[Building and Running the Code 6](#_Toc481640807)

# Distinguishing between Library Components and Client Code Data

## Run as Text

The text version of the program works by running the runText class. It will need to be provided with seven arguments which determine the details of the simulation. These arguments are:

* Steps: How many steps the program should run for.
* P Value: The probability of small cars and motorbikes arriving.
* Q Value: The probability of Family Sedans arriving.
* numPumps: The number of pumps in the Petrol Station.
* numTills: The number of Tills in the Petrol Station’s shop.
* Gallon Price: The price per gallon of fuel in pence.
* trucksChecked: Whether Trucks should be included in the simulation.

Next, it creates the Text View object and feeds it the arguments it was given. The Text View then creates a new Simulator, and the Simulator creates the Config. Then the Text View updates the Config with the information from the arguments. At this point, everything is ready for the simulation to run based exactly on the arguments provided at the start. Therefore, the Text View tells the Simulator to create the Petrol Station. The Petrol Station will be created based on the information stored in the config. This means that if you used the value ‘4’ as your fifth argument, the Petrol Station will have four tills. It is important to note that the Petrol Station is given the instance of the config file in order to allow it to get all other details about how the simulation should run.

The Petrol Station’s constructor reads the instance of the Config and creates its Pumps, Shop and resources. The Shop’s constructor takes in the amount of tills to create from the Config as well. The Pumps will each create their own Vehicle Queue to hold Vehicles and the Tills will each create their own Till Queue to hold Customers. An instance of the output class will also be made. This will hold all the important information that is needed by the user. An instance of a random number generator is also created with the seed in the Config. This Random number generator will be used throughout the program in order to ensure repeatability. The Petrol Station is now ready for Customers, and so, the Text View tells the Simulator to simulate for the number of steps defined by the user at the beginning, as well as providing a boolean telling it that it is not being run for GUI.

This simulate method is a simple loop which runs a Simulate Step method every time it loops. The Simulate Step method takes the GUI boolean as that will determine whether the output string it returns is formatted for printing to the console or displaying in the GUI. There is also a check on this boolean that will run the display method in the GUI which takes the formatted output string and splits it up before updating the GUI to hold this information. In this case, because it is being run from the Text View, this boolean will be false and the output string will be formatted in a way that is easy to read for printing to the console.

The Simulate Step method takes the GUI boolean. It then uses the incStep method in the specific instance of the output class being used by the Petrol Station. This increases the value of the step variable by one, in order to keep track of the current step that the simulation is on. Next, an empty string variable is created. The ToString method is run with the GUI boolean parameter. This method tells runs the Petrol Station’s Run method.

The Petrol Station’s Run method is one of the main method in the simulation and handles a large amount of the running of the simulation. As a result, the order in which it runs the various aspects of the simulation are specifically tailored to meet the specification. It is called once every step and returns a string which details the state of the Petrol Station after the step is over. The first thing this method does is to create an empty String variable called ‘information’. This variable will hold the formatted output string. It will get then add the information about the current step, money missed and the total amount of money made. It retrieves this information from the output class. Next, the Shop’s ‘passTime’ method is run and then the each Pump’s ‘pumpFuel’ method. I will go into details on what these methods do later. The Run method will then attempt to spawn a Vehicle. This method will return true if a Vehicle is spawned, otherwise false. It works by generating a random double and seeing which range it fits into. For example, if a *0.035* is generated, then it will fall within the Motorbikes range and the vehicle will be a Motorbike. It then sets the field ‘generatedV’ to this specific vehicle.

If a vehicle is successfully spawned, the method will return true and so the Petrol Station must now give it to a Pump. It does this by first getting the size of the first Pump’s queue. It then goes through each pump and compares its size. If the size is smaller, then size is set to the size of the new smaller pump queue. The current shortest queue is kept track of in the ‘shortestQueue’ variable. Once it has checked all the available pumps, it will then attempt to add the Vehicle to the shortest queue. This may not be possible however as the shortest queue may still not have enough space to hold the new Vehicle. If there isn’t enough space, then the Vehicle leaves and the method returns ‘false’. The amount of money that would have been made by filling up the vehicle’s tank is added to the Lost Money integer in the output object. However, if there is enough space, the vehicle is simply added to the back of the Pump’s Vehicle Queue.

The last part of the **Petrol Station’s Run** method appends the information about the pump’s and till’s current state to the end of the ‘information’ string and returns it to the caller.

The **‘pumpFuel’** method is run for each of the pumps inside the Petrol Station. The first thing it does is makes the Pump check whether it actually has any vehicles in it. This is to avoid a Null Pointer Exception that could be caused by trying to pump fuel into a Vehicle that isn’t actually there. Next, it sets the current vehicle to be the vehicle at the front of the queue. It then attempts to fill the tank of the Vehicle, taking the parameter for the speed at which fuel is pumped. This will return true while the pump fills the vehicles tank up. Once the tank is full however, it will return false. This will cause the Vehicle to create a Customer for itself. The Customer is created based on the Vehicle which spawned it, so it will contain a reference to the instance of it’s vehicle as well as a unique name based on this vehicle. It will also set it’s various values such as how long it’ll spend shopping based on the type of Vehicle that it owns. In the this **‘createCustomer’** method, the ‘**shoppingTime**’ will be set, then the Customer will either go into the store to buy items, or straight to the till depending on how long it took for it to refuel it’s vehicle. If it goes straight to the till, then the output’s lost money variable will be increased by the amount the customer would have spent if it had gone shopping.

Assuming the Customer was indeed happy, it will go into an array inside the Shop until it has finished shopping i.e. ‘**shoppingTime**’ has reached zero. Finally it will go to the till queue. The customer will also join onto the back of the shortest queue, and once it reaches the front of the queue, will spend 2-3minutes paying. Only once the Customer has finished paying will the output’s ‘**moneyGained**’ variable increase by the fuel cost and the shopping cost.

The Petrol Stations ‘**passTime**’ method tells the shop to go through all its tills and makes them serve the Customers. This is specifically where the serving of the customer is handled and it is what decreases the Customer’s ‘**tillTime**’ variable until it reaches zero and the Customer has paid. It does this by running the **‘paid’** method on whichever Customer is at the front of the queue. The paid method will return false until the **‘tillTime’** reaches zero. At that point, the customer is removed from the Till Queue and runs the **‘leave**’ method. The ‘leave method’ is the specific location where the output class is updated with the money gained and it also makes removes the Customers Vehicle from the Pump’s Vehicle Queue.

This how the basic, text version of the simulation is run. There is however another type of vehicle that acts slightly differently, Trucks. The main difference with this vehicle is that it has the ability to change the probability of future trucks spawning. Assuming it is turned on at the beginning of the simulation, the trucks will use the ‘**getHappy**’ method to first determine whether the amount of time they spent queueing is lower than the limit. If that is so, then it will return true and the Customer will run a method in the config class that takes the happy boolean and increases the probability of trucks arriving i.e. the truck driver telling other truck drivers that they should go to this specific petrol station. If the driver is unhappy with how long it took, the probability decreases instead.

Finally, the program may be run in GUI mode. This does not change the underlying mechanics of the simulation, but simply how the outputted information is displayed. Instead of taking arguments when the program is run, it will instead create a form with sliders and fields allowing you to control/determine how the program will run. These are your **p and q** values etc. the same as in the text version. Once the user is satisfied, they press the Run Simulation button which creates the simulator object and feeds it the parameters from the form. The simulation will then run in the same way that it did for the text version, however the outputted string will be formatted to be read by the Petrol GUI. Once it has finished running the simulation, a window will open which displays all values for the final step as well as the cumulative data such as total money made.

## 

# (Level 1) Required Changes for the Library Classes to Support Simulations

The initial changes to the classes to support the simulation can entail with the following:

## More types of vehicles

If different types of vehicles need to be created, it’ll require creating a brand-new subclass that extends the vehicle class. You must then add the specific details about the vehicle to the config class. These details include the fuel tank size and the shopping probability specific to the new type of vehicle. Finally, you would need to add the new type of vehicle to the options that are spawned by the petrol station in the *spawnVehicle* method.

## Multiple types of fuel with different prices

Because the type of fuel that is being pumped has no effect besides changing the cost of the fuel in pence per gallon. All that would need to be done to add new fuel is to add the other fuel’s costs to the config file. You could also add an argument to the main classes if you want to be able to define the cost of this new fuel whenever the program is run. The customer could simply pay for the amount of fuel he pumped multiplied by the price specific to the type of fuel that he would like. The type of fuel he would like could be decided arbitrarily by the type of vehicle or even a random variable. This would achieve the effect of having multiple fuel types with easy implementation.

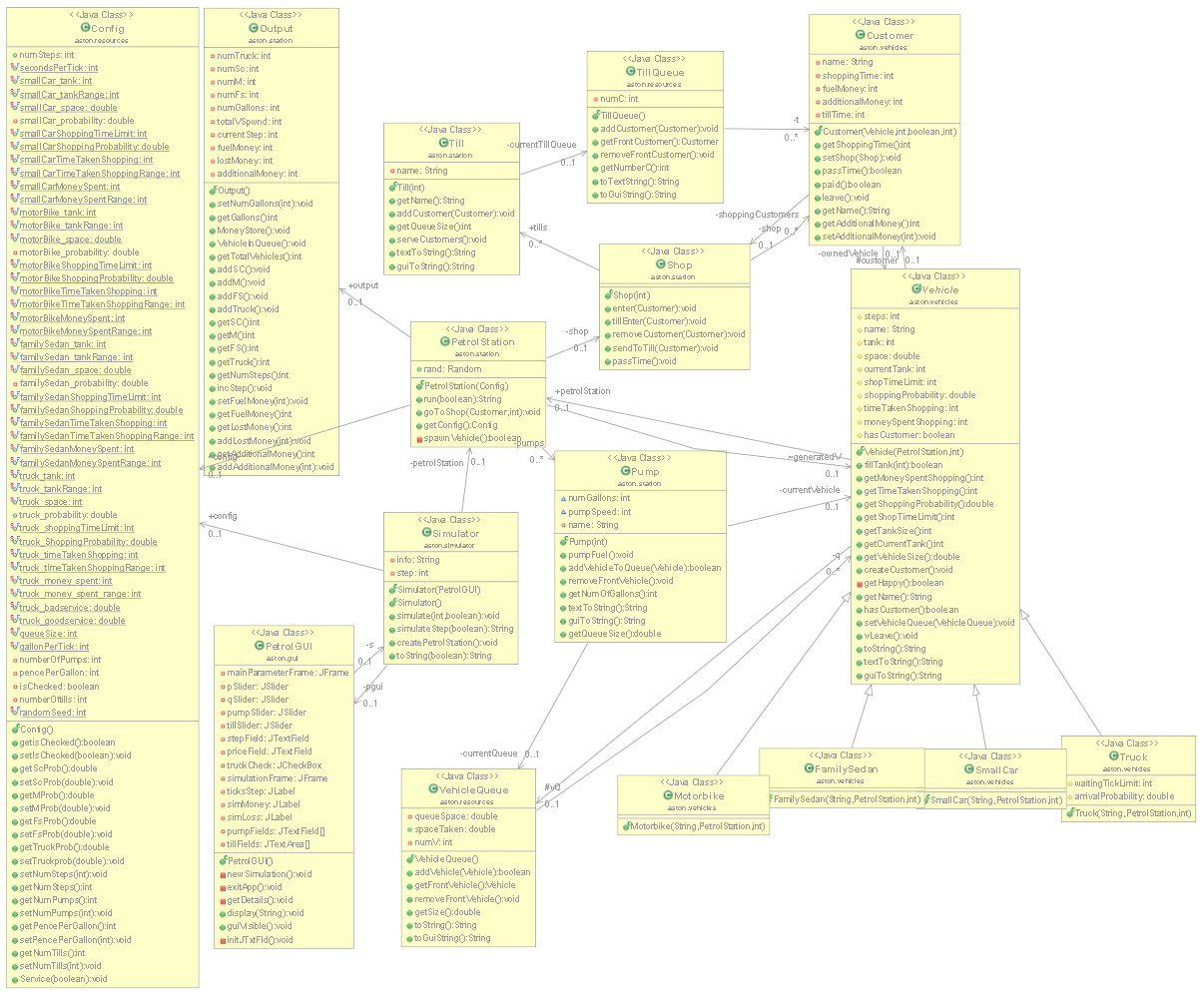
## Parking away from the pump during shopping

This could be achieved by adding an array inside the petrol station which holds the vehicles while the customer pays. This would require modifying the customer to remove the vehicle from this array instead of the queue, and also for the vehicle to get sent to this array when it finishes pumping. This would have the effect of making the Petrol Station work in a very similar way to the shop, where there is an array that holds onto customers until they have finished shopping, but would also have to wait for the customer to finish paying at the till.

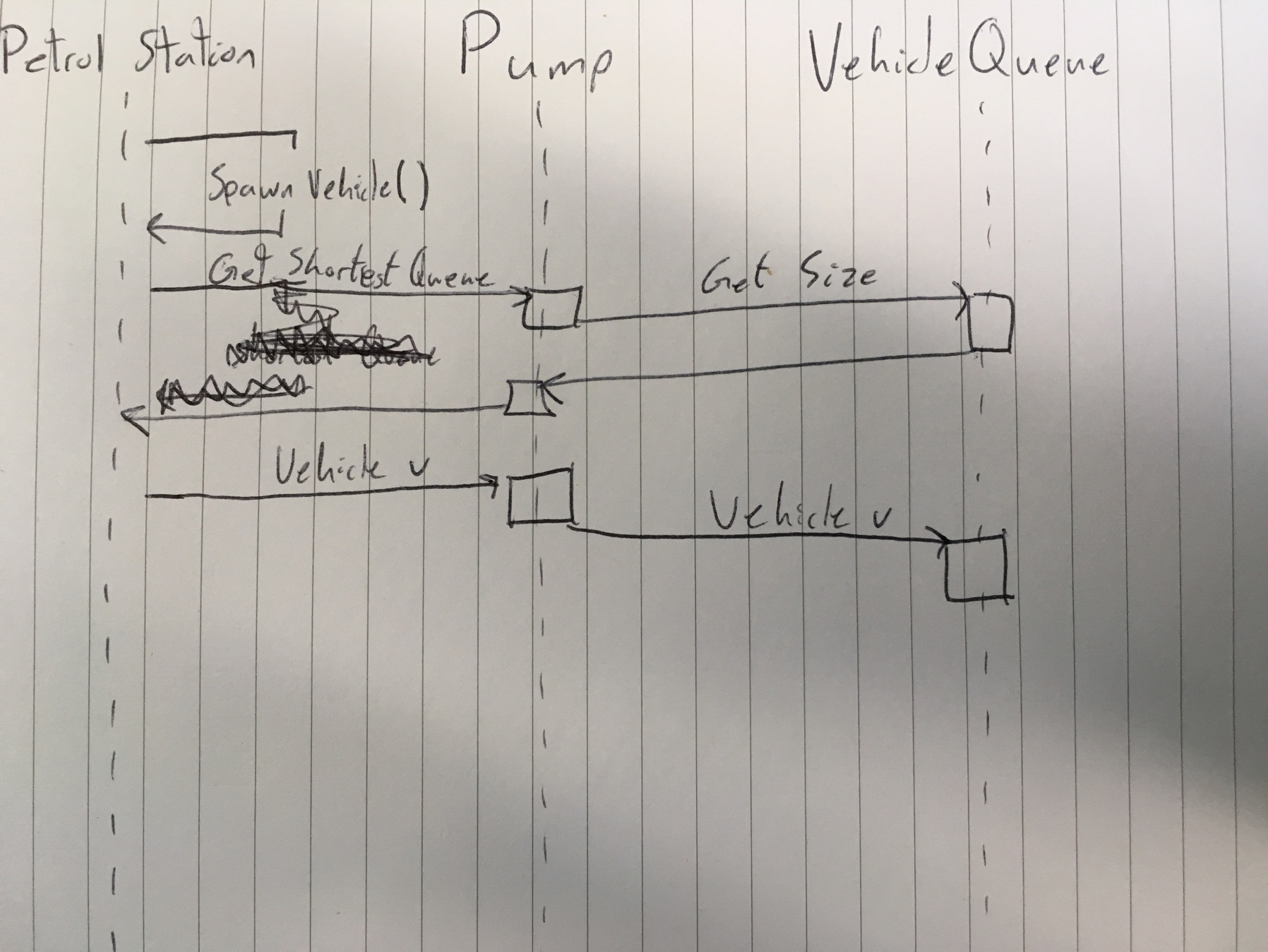
## Vehicles breaking down during the simulation

A vehicle breaking down would obviously cause a block for the pump that it is at. This can be implemented in a variety of ways depending on the requirements given. For example, if a broken down vehicle stayed at the pump location until it was repaired, you would need to add a boolean that determines whether the vehicle is broken down. This could be set by a random breakdown chance. The pump would then have to check whether the vehicle is broken down before it pumps fuel, and not if it is broken. After a certain amount of time has passed the vehicle could become repaired, and this boolean would simply be flipped allowing the pump to continue providing fuel.

# UML Class Hierarchy

**

# (Level 1 only) Sequence Diagram for a Scenario in the Simulation



# The results of the simulations in tabular form;

# Results and Implications

To test the simulation and to evaluate the results we ran a simulation. The simulation was run with 0.3 for probability of q and p as well as £1.2 price per Gallon. The setting for tills and pumps were changed; the ticks were set to 1000 for each run of the simulation and the arrival probability of a truck would be set to active or inactive. These variables had to be changed to see if we could spot a pattern in the results of the simulation and to observe which setup of tills and pumps would create the greatest yield of money while suffering the smallest loss of money possible. The settings for the least money gained and the most money lost were with 1 till and 1 vehicle pump. The money made totaled £583.85 whilst the money lost was £846.25 , the money lost was more than half than of what was made. We learned that the reason for this was due to limited amount of pumps and tills at the station. Station could not accommodate for enough drivers to fit into pumps therefore when they arrived at the station the cars would leave, causing a loss in the possible total of money earned. Money would occasionally be lost due to a long wait time in the pump queue. We also learned that the setting for the greatest yield of money was a setting of 4 pumps and 2 tills; the total money earned amounted to £1601.73. This setting also had the lowest amount of money lost which was £264.17. This increase in money earned between the 2 settings was due to increase in pumps. More pumps accommodate for more vehicles making the money spent larger in comparison as the fuel bought and additional shopping increased the total money earned. There is also a positive correlation that can be seen between the number of pumps and the money earned and a negative correlation between the money lost and the number of pumps.

# Building and Running the Code

There are two different ways to run the program: the first is to open the ‘aston.simulator’ package then run the runGUI class as it contains a main() method. This will compile and present the standard GUI window, which is the Petrol Station Parameter Window. In this window you may change the variables before running the simulation.

The other way is to select the runText class, also in the ‘aston.simulator’ package, and run. You must have set the run configuration to provide the arguments which are required. This will allow you to change the details of how the simulation runs, such as the p and q values.

Selecting the Build Button to create an executable Java file will allow you to build the program. It should be done on whichever of the two run classes you want, runGUI or runText. This will allow it to run independently as long as one defining and appropriate class has the standard main() method. The Build Button will appear as a Green Play Button with a red toolbox on the toolbar, or can be selected by going to Project Menu Tab, then selecting Build.