Advanced Programming Coursework 2018

Matthew Henderson – 2031944H

Assumptions

Several assumptions were made whilst designing the programme. Firstly, as stated in the project specification, this is just the infrastructure of the code. As such, there was no controller class created that could be attached to a view, and the main class was simply used as a controller. It was assumed that should this programme be expanded to include a user interface, a controller and view class would be added. Furthermore, since there is no user input, no validation is performed on any of the parameters. Again, it is assumed that should the programme be expanded, user input validation would be added.

Regarding the flow of traffic, it was assumed that no one vehicle had priority over another. Hence, if two vehicles are waiting to get into the same space, the vehicle that occupies it first will be the one that is on the thread the java virtual machine chooses to run first.

It is assumed that the only reason that a vehicle needs to stop is for traffic, and that otherwise it moves at a constant speed. Furthermore, it is assumed that one lane will not have two cars going in different directions. This scenario would cause a deadlock.

Class Design

The design of the programme can be seen in the UML diagram below. It works as follows. The main class creates a grid object, a repainter object, a statistics object and a trafficManager object. Both the repainter and trafficManager objects are threaded. The grid object is passed to the repainter object which repeatedly prints it to the console. The grid is also passed to the traffic manager, along with all maximum speed, minimum speed, number of vehicles, time between vehicles, lanes which the traffic manager controls and whether the vehicles will be going forward and backwards. Using this information, the traffic manager creates vehicle objects and starts them running on their own threads. These vehicles are passed the grid, along with randomly generated parameters such as speed, lane and direction. The vehicle calls the addToGrid method of the grid passing itself as a parameter. It then repeatedly calls the move method, again with itself as a parameter, until it reaches the end of the grid. Once at the end of the grid, the vehicles setFinished method is called and the thread stops running. After all traffic managers have stopped running, the getVehicleTimes method is called by the main and the resulting array is passed to the statistics class which prints all statistics to the console. A reentrantLock and condition in the grid class ensures that there are no "collisions" between vehicles.

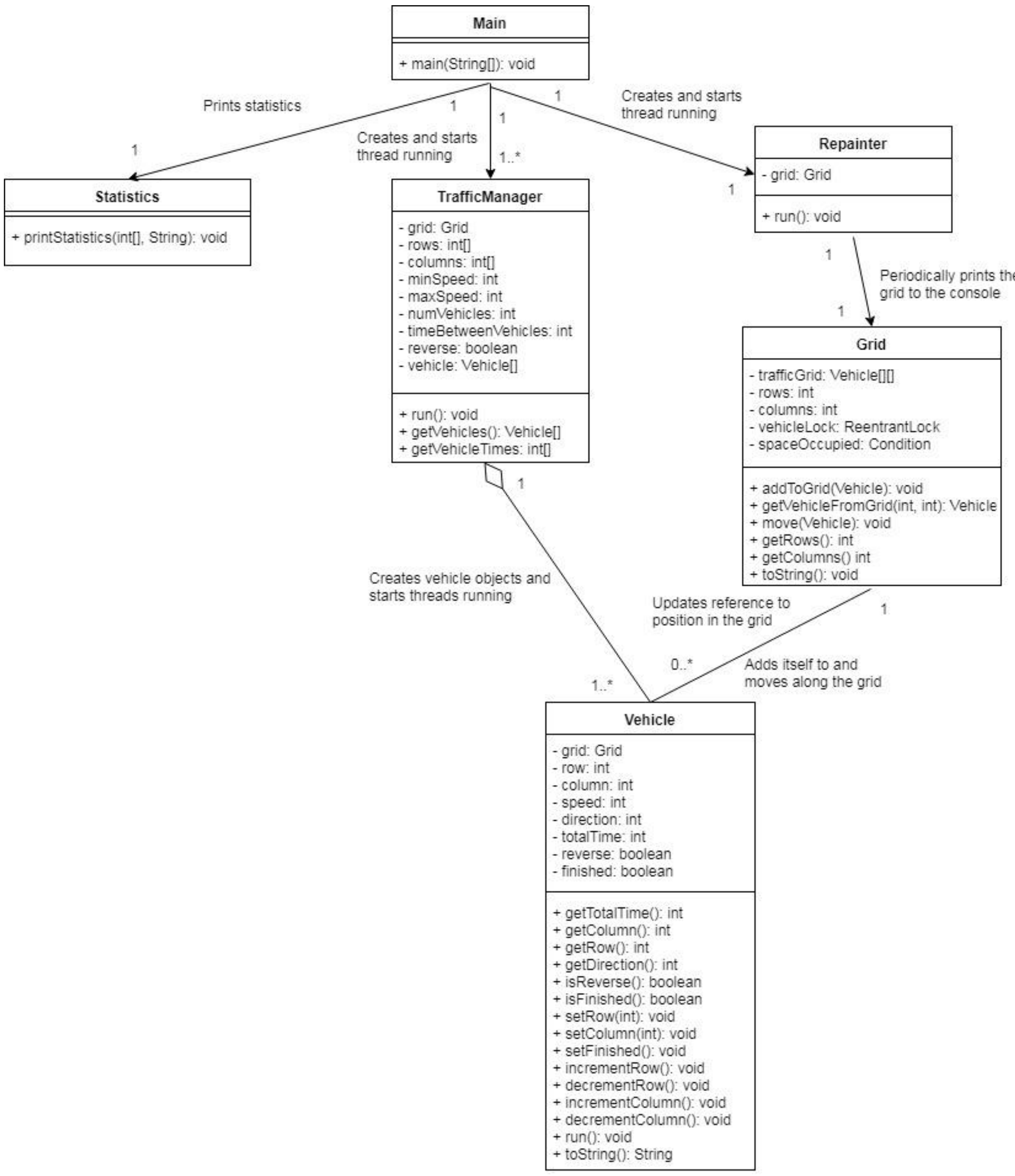

Figure 1. UML diagram of the project

Testing

A selection of JUnit tests were written to automatically assess the functionality of the system. This makes it easy to retest the system if any change is made. Table 1 below shows all the scenarios that tests have been designed for, as well as the result of the tests at the time of submission. The JUnit tests can be seen in the TrafficTest class that has been handed in alongside all the other java files. Note that since the number of cars in is always the same as the number of cars out, the re-entrant lock must be functioning. If it was not, references to vehicles would be overridden in the grid and fewer cars would make it to the end of the grid.

In addition to those JUnit tests shown in table 1, the statistics class was also tested by inserting known data into the printStatistics method. The results were then compared against data calculated by hand. This can be seen in table 2.

Table 1. JUnit test cases

Scenario	Result of test	Expected?
Grid has the correct number of rows	Pass	Yes
Grid has the correct number of columns	Pass	Yes
Vehicle is added to the top of the grid	Pass	Yes
Vehicle is added to the bottom of the grid	Pass	Yes
Vehicle is added to the left of the grid	Pass	Yes
Vehicle is added to the right of the grid	Pass	Yes
Vehicle moves south	Pass	Yes
Vehicle moves north	Pass	Yes
Vehicle moves east	Pass	Yes
Vehicle moves west	Pass	Yes
Same number of vehicles out as in	Pass	Yes
All vehicles have logged a time	Pass	Yes
Same number of vehicles out as in (dense traffic)	Pass	Yes
All vehicles have logged a time (dense traffic)	Pass	Yes
Same number of vehicles out as in (big grid)	Pass	Yes
All vehicles have logged a time (big grid)	Pass	Yes
Same number of vehicles out as in (small grid)	Pass	Yes
All vehicles have logged a time (small grid)	Pass	Yes
Same number of vehicles out as in (multiple traffic manager)	Pass	Yes
All vehicles have logged a time (multiple traffic managers)	Pass	Yes
Logged time is accurate to within 5 percent	Pass	Yes
Head on collision (deadlock)	TestTimedOut exception	Yes

Table 2. Testing the statistics class

Input (ms)	Max time (s)	Min time (s)	Mean (s)	Variance	Expected?
{1000,2000,3000,4000}	4	1	3	1250	Yes

Questions

1) Currently the system could not handle switching lanes and overtaking of slow moving vehicle. It can be seen by simply witnessing the programme run that there are many situations where a fast moving vehicle will get stuck behind slow moving vehicles. In a real life scenario the faster vehicle would simply move lanes and overtake the slower vehicle. However, the system does not currently facilitate lane switching. Furthermore, deadlocks would almost certainly occur in situations where the parallel lane has traffic flowing in the opposite direction.

2) A traffic light system to allow for pedestrians crossing the road could be implemented. This could be done with another condition that blocks the flow of traffic through certain squares once active. This would require the addition of a Boolean instance variable that is true when the light is red. Then by using an if statement, the vehicle could be put in the await state when the light is red and the vehicle is about to move into the crossing zone. Furthermore, it would not be hard to write a pedestrian class that could be implemented into the system to simulate people crossing the road. The would operate in a very similar way to the Vehicle class.

3) This could be achieved by having a setter method for the grid instant variable in the vehicle class. When a vehicle reaches the end of the grid, the current grid calls the addToGrid method on the new grid using the vehicle as an argument. It then calls the new setGrid method in the vehicle class to update the grid variable. As long as the setFinished method is not called on the vehicle object, the vehicle will now start calling the move method in the new grid object. Particular care must be taken to ensure that the vehicle enter the correct row/column of the next grid, but this completely depends upon how the user wishes to string the grids together.

4) The first step I took when approaching this project was drawing out an overview of the class structure in UML. I started with the aim of completing both parts of the specification so structured the project as such. During the development of the programme some of the methods moved and some more methods were added, however the loose structure was the same.
I found the main challenge of the programme was knowing where to place the re-entrant lock and the condition. Initially I had placed these in the vehicle class. However, once I realised these belonged in the grid class, the rest of the programme came relatively naturally. In particular, I found that I was able to write the statistics class quickly. Furthermore, since I had started the project with the intent of completing the second specification, there was no major change that needed to be made in the code to incorporate it.
Once completed I found one major bug in the programme. This was caused by the signalAll method of the condition not being called when a vehicle exited the grid. The result of this was that occasionally vehicles got stuck in the await state and the programme did not end. This did not happen every time the programme ran. For me this emphasises the importance of extensively testing programmes that are seemingly working well. Especially threaded programmes which can potentially run differently every time.
If I was to do this project again there is one thing I would do differently. That is to adopt more of a test-driven approach. All of my JUnit tests were written retrospectively. Had I written these before I had started the programme, it would have allowed for easy continuous testing at every stage of the project. This would have made it easier to detect bugs and errors in the code.