Martin Ortiz-Cañavate Ortiz

s1913403

Informatics Large Practical (INFR09051)

1st semester of 2021/2022

ILP Final Project: Software Architecture Description

In this section, I will provide you with a description of the software architecture of my application. I will walk you through all the java classes that compose the project and explain my thought process in identifying this as the best way to structure the application.

My project is composed of eight different classes. I've designed each class to take care of a different aspect of the drone app, and then I've coordinated them in the main 'App' class to be executed properly and provide the final result.

So, at the beginning, I identified two main areas in this project. The first one being the drone flight control; and the second one being the area of getting the necessary information from the Apache DB and the Webserver and uploading to the Apache DB the appropriate results. I then decided that taking into account a third area, one in charge of holding the downloaded information from the servers in an easy and accessible way, was most appropiate. This area sort of acts as a bridge between the two aforementioned areas.

Therefore, out of the eight classes which compose my project, two of them are in charge of the drone flight control, three classes are in charge of communicating with the Webserver and the Apache Database, another two classes are parsers for the Webserver's '.json' documents and the last class takes care of the third area I spoke of, generating a data table with both de data from the Webserver and the Apache DB.

First of all, I'm going to start with the 'LongLat' class. This is one of the classes we were asked to create for the first coursework, and I expanded it for the final project.

This class is one of the two classes that collaborate to devise the drone's flightpath. But as it is used to store coordinates it is essential in all the areas of the project.

The class works as follows; given two double numbers, which represent coordinates, this class will generate a 'LongLat' object representing that given location. Apart from this, this class has methods which encompass all the necessary geometrical tools to do the calculations needed for the flightpath: it calculates distances between objects, angles, and using the inner class 'line' (two 'LongLat' objects), it can calculate if two lines intersect. On top of this, this class has the method that checks if the location is inside the flight-zone, and it also checks if two locations are 'close enough' in terms of the drone range.

The second class I've created is the 'Http' class. This class was needed as well for the first coursework, and I expanded it for the large practical.

This class collaborates in getting the necessary information from the servers, generating the 'http client' needed to communicate with the Webserver.

Theis class works as follows; given a 'machine name' and a 'port number', it generates the 'http client' which will establish the connection with the webserver. Once this is done, there are three different methods in the class which are used to access all the different '.json' documents needed.

The third class I've created is the 'shop' class. This class was needed as well for the first coursework, and I have used it for the final project as well.

This class also collaborates in getting the necessary information from the Webserver, and its function is to parse the ‘menus.json’ document retrieved from the website in order to get all the items in the menu, its prices and the locations of the shops where they serve them.

The fourth class I’ve created is the ‘coordinates’ class. This class is very important as it helps in linking the “ThreeWords” locations our customers use with the longitude and latitude coordinates our drone uses to navigate.

This class works as a parser which, given a Webserver’s ‘.json’ document with the information of a “ThreeWords” address, it will parse out the longitude and latitude coordinates corresponding to the location.

The fifth class I’ve created is the ‘Website’ class. This is the principal class in charge of getting the information from the Webserver, meaning that initiating a ‘Website’ class object lets you retrieve all the concise information needed from the ‘Webserver’.

The ‘Website’ class , given a ‘machine name’ and a ‘port number’, initiates the ‘Http’ class client.

An inner class called ‘info’ is used to store the information of a given order contained in the Webserver. This class stores the “ThreeWords” location of the shops involving the order and the total cost of the order’s delivery.

Therefore, combining the methods from the ‘http’ class to get ‘.json’ documents from the Webserver and the parser classes (‘shop’ and ‘coordinates’), the ‘Website’ class has all the methods needed to provide the necessary information in the proper format (i.e. ‘LongLat, int, ‘info’ …).

It has methods to get the coordinates of a “ThreeWords” location, to get the ‘info’ information of an order and to get the no-fly zone’s perimeter as an array of ‘LongLat.line’.

The sixth class I've created is the 'Database' class. This is the class that covers the third area I previously mentioned. This class generates a data table where, in each row (an inner class 'row' generates the objects which represents the rows of the data table), all the necessary information about an order (order number, price, drop-off location and pick-up locations) can be stored.

The 'Database' class contains all the methods to add rows to the data table and get information from the selected row. It also has the method which helps us employ the 'greedy algorithm', which oversees the ordering of rows in the data table from most expensive order to least expensive.

The seventh class I've created is the 'ApacheDB' class. This is the last class involved in getting information from the servers. This class in particular, establishes the connection with the Apache Database; it's used to retrieve information from it and store information into newly created SQL tables.

The class is initiated as follows; given the 'machine name' and the 'apache database port', it connects to the Apache Database. And after checking if the tables 'deliveries' and 'flightpath' don't exist, erasing them if they do, it creates them.

Apart from this, the 'ApacheDB' class contains all the necessary methods to read and write information into the Apache Database. It contains a method that retrieves all the delivery orders of a given day and writes them into the , and then two methods which write into the tables created when initiating the class all the deliveries completed and the flightpath followed by the drone respectively.

The last class I’ve created is the ‘App’ class. It calculates with the help from the ‘LongLat’ class methods the drone’s flightpath using the data stored in the ‘Database’ class. It is also the main class, from where the application is run.

It functions in the following way; the app class is first provided as input a date, the Webserver’s port and the Apache Database’s port. After this, the ‘App’ class will initiate the ‘Website’ (which in turn initiates the ‘Http’ class) and the ‘ApacheDB’ (which creates the SQL tables ‘deliveries’ and ‘flightpath’) classes. Having initiated these classes, ‘App’ uses will also initiate the ‘Database’ class. Then, it uses ’ApacheDB’ and ‘Website’ to get the no-fly zone perimeter, fill the data table generated in the ‘Database’ class with the orders of the day specified in the date given as input, and then it will order this table in a ‘greedy like’ fashion.

After all this is done, the ‘App’ class uses the ordered data in the data table of ‘Database’ and the own methods of ‘App’ to calculate the best possible flightpath which maximises the "sampled average percentage monetary value".

Every time the drone completes a new delivery order, ‘App’ will call methods from the ‘ApacheDB’ class, which will write the necessary order details into the ‘deliveries’ table and it will write the flightpath undergone by the drone to complete the order’s delivery into the ‘flightpath’ table in the Apache Database.

So, as an overview, we can conclude that I’ve designed the software architecture in a way that makes the app work as follows:

- ‘App’ receives date and ports as input ->
- ‘App’ initiates the ‘Website’ ,‘ApacheDB’ and ‘Database’ classes ->

- ‘Website’ and ‘ApacheDB’ use in turn the ‘Http’, ‘shop’ and ‘coordinates’ classes to fill the data table of ‘Database’ with the orders of the day, and order it following a ‘greedy like’ fashion ->
- ‘App’ uses information stored in the data table of ‘Database’ and methods from ‘LongLat’ and ‘ApacheDB’ respectively to calculate the best flightpath to deliver the orders and record the necessary information on the Apache Database ->
- Finally ‘App’ generates the ‘.geojson’ file to help visualize the flightpath.

ILP Final Project: Drone Control Algorithm

In this section, I will explain to you the algorithms I’ve implemented to design the flightpath our drone will follow to fulfill all the possible orders.

I will walk through how the algorithmic approach I’ve implemented maximizes the “sampled average percentage monetary value”, how it decides to which locations of interest to go first when delivering the order, how it calculates the direction of the steps needed to take to arrive to the destinations while avoiding the no-fly zones, and how it ensures that the drone can always go back to the start location before running out of moves.

It will also include two graphical figures (‘.geojson’), which will allow us to visualize the flightpath of the drone on two dates and thus, understand better how our drone behaves while on the air.

First, I will explain to you how I’ve approached the maximization of the “sampled average percentage monetary value”.

As you’ve read on the Software Architecture Description section of this report, before calculating the flightpath of the drone, my ‘App’ is designed so that it first stores all the orders and its data in a data table, and I’ve designed the code such, that my drone will always try to complete the first order on the data table.

Therefore, my approach this method was all about how to order the table (either once or multiple times during the flight) in such way, that the first order on the table would be the best choice to maximize the “sampled average percentage monetary value”.

I first considered the following approach. Every time the drone is about to fulfill a new delivery order, I would calculate the moves needed to complete each order, I would then calculate (order price)/(moves needed) and finally I would order my data table using this parameter.

This is a great way to maximize “sampled average percentage monetary value”, but as I found out, it increased a lot the running time of the application and the results obtained weren’t that superior compared to other simpler but less ‘time consuming’ methods.

After trying the explained approach on different days, I discovered that, more often than not, the best delivery to make next in order to maximize the “sampled average percentage monetary value” is the one with a higher cost. That’s why I decided to use the ‘greedy’ method.

So my approach proceeds as follows; when the data table with all the orders is completed, I order it from the most expensive order to the least expensive. After this the drone will always try to complete the order with the highest cost, if it doesn’t have enough moves left, it will try to complete the second most expensive, and so on. Therefore, our drone uses the ‘greedy approach to decide which orders to deliver first.

This achieved by ordering the data table using the ‘greedyTable’ method in the ‘Database’ class.

Now that we've decide which order our drone is completing, we need to find out how our drone is going go on about delivering the order.

As the orders require food items that need to be collected from up to two different shops, and the drop-off location is the last place the drone must go when fulfilling the order, the question here is from which shop of the two (if the food is only from one shop then its easy) we should pick up the food first.

A priori I thought it would be better for the drone to go first to the shop that is closer, but after trying both possible orders in different days, I found out that the most efficient way is to leave the shop that is closer to the drop-off location last.

Therefore, in order to deliver an order, the method 'distanceOrder' in 'App' is used to choose the order in which food items are picked up following the criterion described above, and then the method 'orderFlight' in 'App' instructs the drone to go to the ordered pick-up locations and then to the drop-off location.

Now that we know all the locations the drone must go to deliver an order, and in what order, we need to work out the sequence of moves that the drone needs to make to go to all these locations.

As you probably already know, the moves the drone makes consist of a fixed step (0.00015) taken in a direction defined by multiple of 10 degrees angle. On top of this, the drone cannot make a move which take it outside a certain area, and it cannot make a move which takes it into the no-flight area.

Solving this problem, I decided to first build a recursive method in charge of instructing the drones to take steps until it reaches (gets 'close' enough) the desired destination. Therefore, the methods 'movementRecursion' and 'movementCalculator' in

'App' oversee the process of recursively taking steps in a certain angle and saving these moves to a 'LongLat' array list respectively.

Now the tricky part was how to choose the angle of the direction in which each step is taken so that the drone ends up arriving to the destination we want in the minimum amount of steps and without violating the flight rules (not leaving the flying area and not entering the no-flight zones).

Using the provided landmarks whenever the drone was in a collision path was the simplest solution, but this way of avoiding the no-fly zone sometimes caused the drone to take big detours which could have been avoided otherwise. Therefore, I approached the choosing of the angle for each step the following way.

As the drone now knows the next location where it has to hover; it calculates for each step the multiple of 10 degrees angle that is closer to the angle which would take the drone in a straight line to said location. This is managed by the method getAngle' in the 'LongLat' class, and it does so by using basic trigonometry and the rounding to the nearest multiple of 10.

Once the best possible angle is chosen, it checks if taking the step with the calculated is a permitted move (doesn't take him outside the fly zone or into the no-fly zone). This is checked by the method 'isMoveGood' in 'App', and it does so by checking if the line segment produced by taking a step in the given angle intersects with any of the line segments which compose the no-fly zone perimeter.

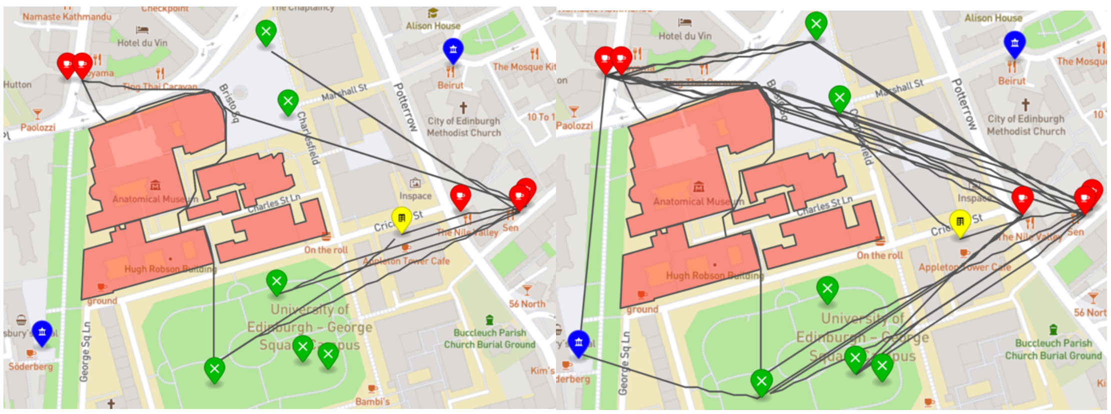
If it isn't a permitted move, it will recursively find the closest multiple of 10 degrees angle that makes the move legal. This is managed by the method 'goodAngle' in 'App', and it does so by trying all the possible angles starting with the closest multiples of 10 to the optimal angle.

And this process is repeated recursively for each step until the drone gets 'close' enough to the intended location, permitting the drone to go around the no fly zones. This is done by the methods 'movementRecursion' and 'movementCalculator' in 'App'.

The problem with this, is that sometimes, it wastes more moves going around a no-fly zone, than using the landmark approach. Therefore after trying different parameters, we've arrived at the conclusion that if the drone is using more than 100 moves to go around a no-fly zone, it's more efficient if it uses the landmarks; and we've instructed the drone to do so. And the method that does this is 'willDetour' in 'App'

This way, we have found an algorithm, that behaves in a 'greedy like' way; that permits the drone to go around the no flight zones taking, in the best possible way, a minimal detour.

Now we only need to make sure that our drone can always go back 'home' before running out of moves. Therefore, before delivering an order, we check if the drone would have enough moves to return 'home' after completing the order. When the drone doesn't have the capacity to deliver any more orders, we instruct the drone to return 'home' using the remaining moves (which will always be enough using this method).



As we can see in these two pictures, our drone has the capacity to go around the no-flight zones, which allows him to take shortcuts, but it will also use the landmarks when necessary. Our drone also always returns to Appleton Tower.