There are 11 classes in my implementation. I will explain the reasoning why I used these ones in this section.

1. The “**Database**” class interacts with the database. Its job is to collect information from the database, store it into its own attributes and produce one of the two output tables. This class provides multiple methods to extract columns from the existing tables as well as one method to create a table. This class purely serves the functionality of connection and interactivity with the database and does **not** modify or process any of the data.

2. The “**Shop**” class is for deserialization of information from the “menus” folder in the web content. We are interested in the location, item and price. The structure of this java class is purposely built exactly like the menus folder so that the application can access the information.

3. The “**Details**” class is for deserialization of information from the “words” folder in the web content. We are interested in the coordinate that corresponds to the 3 words. The structure of this java class is purposely built exactly like the words folder so that the application can access the information.

4. The “**Polygon**” class is for deserialization of information from the “buildings” folder in the web content. We are interested in the coordinates which make up the polygons. The structure of this java class is purposely built exactly like the buildings folder so that the application can access the information.

5. The “**HTTPConnection**” class is one with HTTP connection and some JSON parser functionality to obtain information from the website content. The class creates a connection to the server, sends an HTTP request and receives an HTTP response. It also parses the JSON files from the web server and store them into its attributes.

6. The “**Order**” class represents the orders that the drone has to deliver. Each object of this class speaks for one order and contains details for the order. These include order number, the delivery location in What3Words, the delivery location in longitude and latitude, the shop locations in What3Words, the shop locations in longitude and latitude and the delivery fee.

7. The “**AllOrders**” class is the contains a collection of all the orders (some Order objects) in the day which was given in the parameters as well as a way to assemble the orders all into the collection. This class is needed since there are lots of order in one day and we do not know exactly how many until the parameter is given so we need an array list to store them. There are extracted and unprocessed information about the orders in the HTTPConnection and the Database class and this class utilises the information to form a bunch of orders.

8. The “**LongLat**” class is for coordinates of the location for the drone, shop locations and delivery locations. The class determines the coordinates by longitude and latitude.

9. The “**NoFlyZones**” class is for processing the no-fly-zone information obtained in the HTTPConnection class which was a list of polygons. This class can turn the information into useful collection of coordinates and paths for the ease of the flight algorithm. Our algorithm requires us to know where the vertices of the no-fly-zone polygons are. There are also methods that detect if we are crossing the no-fly-zones or not and if we are too close to the no-fly-zones.

10. The “**Drone**” class contains the algorithm for the drone’s flight path. It takes care of the drone’s trip from Appleton tower, passing through multiple shop locations and delivery locations and finally getting back to Appleton tower. It will produce a table uploaded to the database at the end and create a GeoJSON file that helps us visualise the whole flight path.

11. The “**App**” class is the main class of this project. It is responsible for calling the functions from other class to reach the ultimate goal of drone delivery job. After the main function of this class has finished running, the output (2 tables and 1 file) is expected to be produced and the program is expected to end safely.

I will analyse how the implemented drone control algorithm works in this section with logical details of each function, including 2 examples of graphical presentation.

The idea is that we will first find the stops (midway landmark stops + delivery stops + shop stops), then we will make the drone move. The planning is done before the drone moves and the drone will move strictly according to the plan.

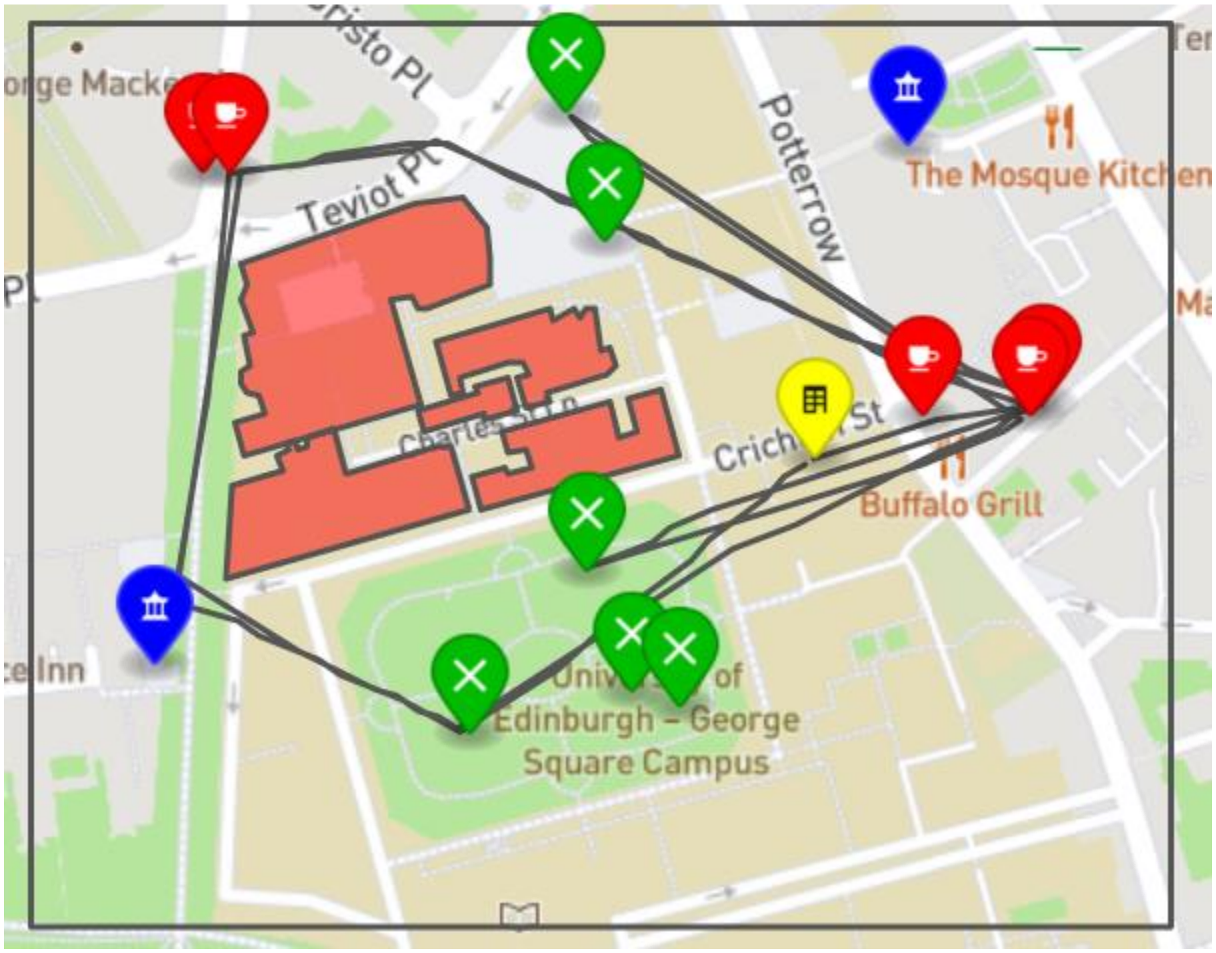
1. The "**findLandmarks**" method discovers 1521 landmarks (39*39). We first divide the longitude and latitude into 40 equal parts, then combine them to make 1521 landmarks that equally distribute in the confined area. The landmarks do not touch the edge of the confined area. There are some landmarks that are in the no-fly-zone area but later we will filter them. The landmarks are stored into an attribute that is an array list because we need to use them frequently. It is worth noting that by increasing the number of landmarks we can slightly shorten the overall number of moves the drone has to do, however I decide not to do so since I need to take care of the run time and the number of landmarks is a great driving factor to it. This method is called in the constructor method of the "Drone" class.

2. The "**findPath**" method considers whether to use a landmark given a start and a destination. The logic behind it is that if there is no-fly-zone detected, we call the function "bestLandmark" to find the optimised landmark for us. The discovered landmark will be added to the returned array list as well as the destination. If it is fine to go straight from the start to destination, then we will not consider using any landmark and will just return the destination itself as an array list. This method is called in the method "findShortestOrder".

3. The "**bestLandmark**" method first filters out the landmarks that connect the start and destination without crossing the no-fly-zones. The okay ones go into an array list. Then we do a second filtering process to these coordinates so that we obtain the ones that are not too close to the no-fly-zones. The reason behind it is that there are different types of float precision in the java program and the GeoJSON visualisation, so we have to make the drone a bit more distant to the no-fly-zone to prevent crossing due to the inaccuracy of precision change. Now after two successful filtering processes, we find the best one which requires the shortest distance. The result is return to the "findPath"

method as a LongLat object. As you can see below, the flight path is not so tightly fitted. This method is called in the “findPath” method.

Example Figure number 1 (drone-01-01-2022.geojson):



4. The “**findShortestOrder**” method discovers the order that requires the shortest distance based on the current location and add the stops to the plan. Specifically, it first calculates the distance of every currently not delivered order, and stores that order in a local variable. We will then find the stops required for this order. The stops include the landmark turning points, the food shops, and the delivery location. These stops will be added to our plan “allStops”. Then we set the location to the delivery location of this order since we are at this location after delivering to the person. We will remove this order from the order list since we have done planning for it. This method uses multiple auxiliary methods to determine the path length of orders because we have to consider the case when there are two food shops to go. This method is called by the “findAllStops” method repeatedly.

5. The “**pathlength**” method is a simple auxiliary method that finds the distance sequentially according to the given LongLat objects in the input array list. This method is called by findShortestOrder.

6. The “**getPath1**” method is another simple auxiliary method that finds the path length for orders that have 2 shops to go. This method returns the distance of going to the first food shop and then the second one. (Called by optimisedShopSequence)

7. The “**getPath2**” method is another simple auxiliary method that finds the path length for orders that have 2 shops to go. It is similar to the last one, but this method returns the distance of going to the second food shop first. (Called by optimisedShopSequence)

8. The “**optimisedShopSequence**” method is a straight-forward method that calls the previous 2 methods. It finds out the distances going to the first food shop first versus going to the second food shop first. It returns true if it is more efficient to go to the second food shop first and false otherwise. (Called by findShortestOrder)

9. The “**findAllStops**” method: Now we have the functionality of finding the next shortest order available. We now determine the overall sequence of orders and obtain all the stops. We use a while loop and repeatedly call the “findShortestOrder” method. After the loop finishes, we will have all the stops ready, and our plan is finally completed. We will add the final stop which is the Appleton Tower.
10. The “**makeAMove**” method tells the drone to make **one** move based on its current battery power and location. The method uses “if…else” to divide into two different situations: when the battery power is 40 or more, and when it is less than 40.

    a. When the battery power is 40 or more, we fetch the second stop in the “allStops” which is from the plan we have done in the previous methods. We fetch the second stop instead of the first one since the first one is Appleton Tower (the start position). We again need to do determine what kind of move the drone needs based on the location, there are 3 branches:

        1. When the drone is close to the next stop: we have arrived at the stop, and we remove this one from the "allStops" list.
        2. When the drone arrives at the destination Appleton Tower: we finish the flight.
        3. When the drone is not close to next stop, we make it keep moving.

    Regardless which of these three branches, we always keep track of the order number and angle since we need these two properties. They are recorded and stored into corresponding attributes. We also take 1 battery power off.

    b. When the battery power is less than 40, we attempt to make the return to Appleton Tower:
        1. we first check if it is close to Appleton Tower. If it is not, then a move is made in order to return.
        2. If the drone is indeed close to Appleton Tower, this means that we have finished our flight.

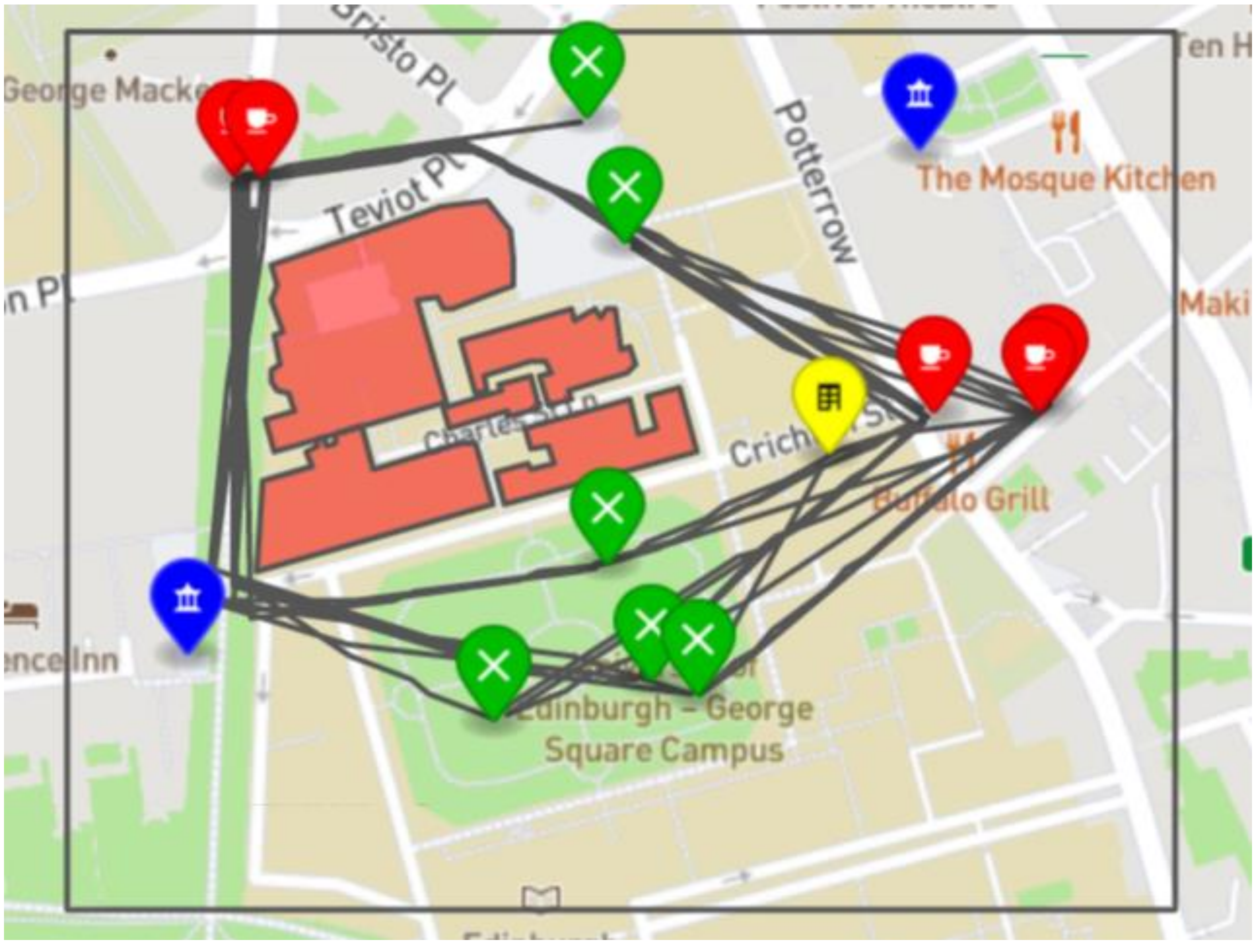
    Again, regardless which of these two branches, we always keep track of the order number and angle since we need these two properties. They are recorded and stored into corresponding attributes. We also take 1 battery power off.

11. The "**makeAllMoves**" method called the previous method "makeAMove" repeatedly so that the drone keeps making moves till the flight is finished. It utilises a "while" loop to check two conditions:
    1. If "allStops" has a size larger than 1, we keep making move. This is because the first stop is Appleton Tower (the start position) and it will never be removed. When we finish delivering all the orders the "allStops" list should drop to length of 1 and when this happens, we stop making moves.
    2. If "finished" Boolean is false, we keep making move. Consider the case when the drone does not have enough battery power to carry through all the orders, it attempts to make its return to Appleton Tower. In this case the "allStops" list will not be length of 1 and this is the reason we need to check on the "finished" Boolean.
12. The "**findAngle**" method is an auxiliary method that discovers the angle that is optimised so that the drone is efficiently moving towards the next stop. Since there are only limited angles available (multiple of 10), we attempt to move in every possible angle from the current location. The attempts will be recorded, and we find the angle that produces the shortest distance to the next stop. This angle is returned. This method is called by "makeAMove" repeatedly.

In the main method of the main class App, two methods stated will be called:

1. **findAllStops**: this will do the planning to obtain all the stops.
2. **makeAllMoves**: this will move the drone from start to finish according the plan.

Example Figure number 2 (drone-12-12-2022.geojson): This is a much longer flight yet the drone still managed to avoid touching the no-fly-zones and makes its return to Appleton Tower.


Bristo Pl
George Macke
Teviot Pl
Potterrow
The Mosque Kitchen
Crich St
Buffalo Grill
Charles St
Edinburgh – George
Square Campus
Ten H
Maki
ence Inn