Part 1: Software architecture description

Part 1.1: Reasoning behind choosing these classes

-App : This is the main class of the project. It is where all of the features of the drone merge together to do the expected functions. At the end of operations, it produces the target geojson file and create the two tables deliveries & flightpath table in database.

-Database reads the databases orders and orderdetails at the database port and interpret the database content at a particular time. This class includes helper functions to read the database data in to arraylist<orders> with corresponding parameters. And also functions to create databases deliveries and flightpath and insert values. It's needed to handle database operations.

- Drone plan the movement of the drone by producing a ArrayList<LongLat> path from the order of the given orders and produce the final ArrayList<Point> pointList avoiding the No-fly-zones, for the final geojson file output. It's needed to planning the flight path of the drone by reordering the list of orders based on the price/distance efficiency of each order. It also controls the drone to avoid the no-fly-zones using the landmarks.

-File contains the simple write to jason function that writes the flight path in geojson format to a new file in the directory using a buffered writer.

-LongLat provides representation of the drone's position using attributes longitude, latitude and angle. It's needed as it contains several helper functions to make the drone updates it's current position and check whether it's close to its target location. It also contains the particular longitude/latitude fields to hold values such that magic numbers don't appear in the main body of code.

-Order contains the order constructor for each order, containing the order number, customer number, delivery date, shop locations, pick up locations and the price for the order including the delivery fees and also a comparable method for the orders in terms of their price-distance efficiency.

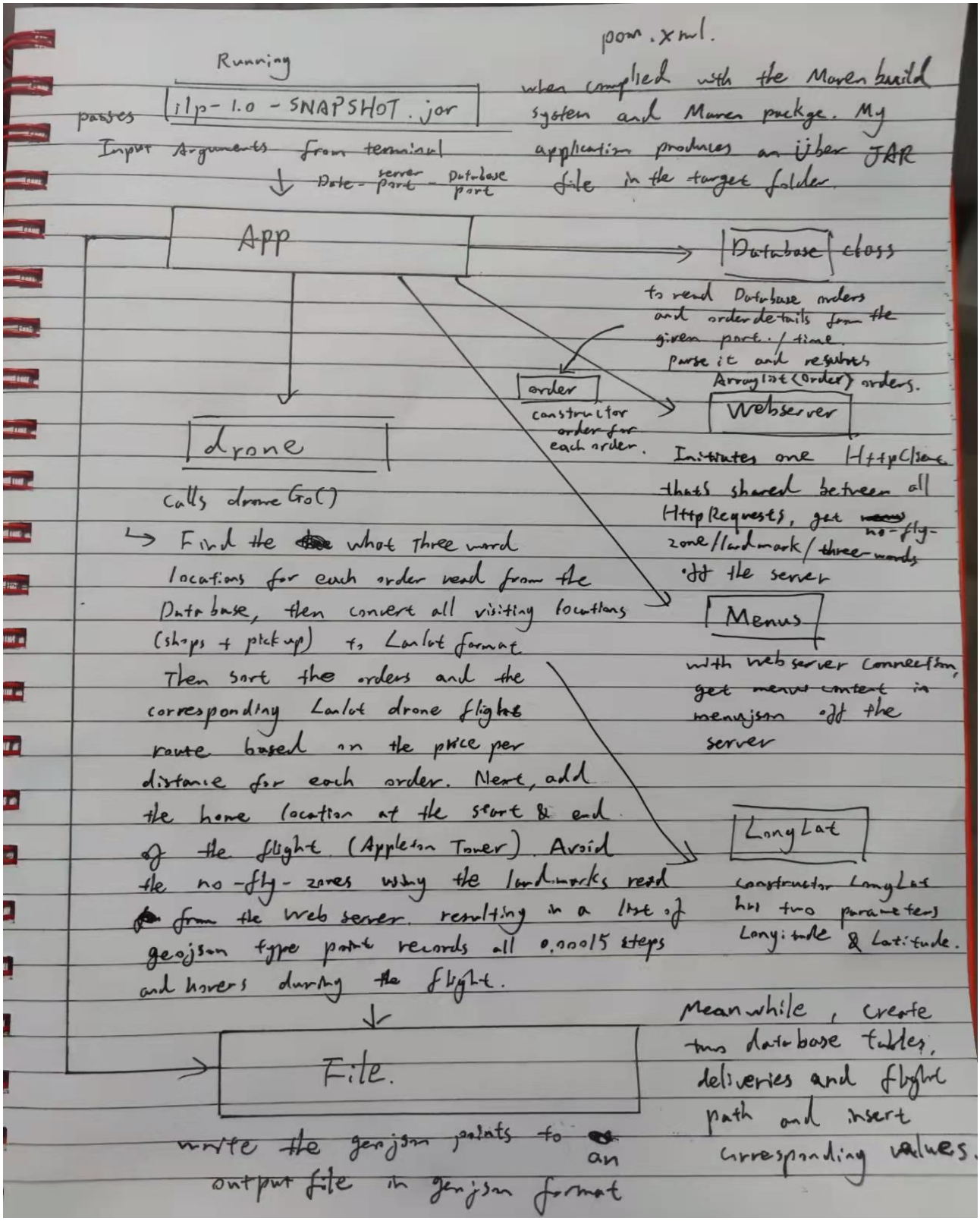
-WebServer initiates an HttpClient, create http requests and get information for landmarks/no fly zones/ three words locations/ menus off the server.

-Menus calls webServer to send Http requests and gets a Http response of JSON format which it parses and store into menujson. It also contains a helper function that calculates the delivery costs for orders.

-Word Parser calls web server for the three-word locations and pack the corresponding longitude and latitude as coordinates. Its constructor to store the location represented

by the what three words. It's also needed to improve the modularity and readability of the code. And make the conceptual idea easier for me.

Part 1.2: A flowchart demonstrating the work flow of my application, between the classes.



Part 1.3: Class Documentation

Class Name	Method Name	Method Description	Return
Database	getJDBCString	get JDBC String for the database	JDBC String for the database.
	readOrders	read table orders	ArrayList<Order> orders
	readOrderDetails	read table orderdetails where orderNo = given orderNo	Arraylist<String> items for the given orderNo
	createTableFlightpath	create an empty table FLIGHTPATH in the database	Void
	createTableDeliveries	create an empty table DELIVERIES in the database	Void
	insertToTableDelivery	insert into the empty table DELIVERIES in the database	Void
	insertToTableflightPath	insert into the empty table DELIVERIES in the database	Void
Drone	isNoFlyZone	check whether the linestring between the given two coordinates crosses the No-Fly-Zones	boolean
	getAngle	calculate the angle between the two Longlat coords, if they cross a no-fly-zone, give an angle that doesn't cross for the current longlat.	Int angle
	findOrderShopLocations	insert the three word locations into each order's orderShopLocations based on the order items by parsing menus.	Void
	getVisitingLongLat	get the LongLat locations of all shops/pickup points and add them to the order's route. with three Word Parser. * meanwhile calculate the totalDeliveryDistance from shop to shop, shop to pick up for each order * and their pricePerDistance based on thier price.	Void
	sortOrders	Sort the orders which are read from the database in terms of decreasing pricePerDistance.	Void

	createLangLatPath	add all order's route to one arraylist path, with appleton tower at the start and end.	Void
	avoidNoFlyZone	if there is noflyzone in between two coords in path, add a valid landmark between the two, to avoid the noflyzone	Void
	planPath	Based on the Arraylist<LongLat> path, add all visiting points to ArrayList<Point> pointList in steps of a single drone move of 0.00015. Meanwhile, handle other helper holders, longPath, completePath, pickUpPath	Void
	checkDelivered	Mark order objects order.isDelivered true if it's indeed delivered.	Void
	deliveriesInsertion	Insert orders into database deliveries, based on their isDelivered field.	Void
	flightPathInsertion	Insert the flight path into database flightpath, and assign the correct orderno to completed deliveries and BACKHOME to the last move back to appleton tower.	Void
	droneGo	A summary function that calls all necessary functions to make the drone move	Void
File	writeToJSon	simple write to jason function that writes the flight path in geojson format to a new file in the * directory using a buffered writer.	Void
LongLat	distanceTo	Calculate the distance between two points	Double, distance between two the points
	closeTo	Determine whether the distance between two points is within the standard distance tolerance	Boolean, True if the points are within the distance tolerance, False if not.
	parseAngle	A helper function to parse the input angle for the nextPosition() method	Ture if the input angle indicates a valid drone movement direction, False if it represents drone-hovering.

	nextPosition	Calculate the next position using based on the input angle.	LongLat, the new position of the drone after making one move in the input angle.
Menus	MenusJson	java objects of the Json class Menus	
	parseMenus	sends request to the server and gets response, Deserializing JSON list to a Java object using its type.	ArrayList<MenusJson> menusJsonList
	getDeliveryCost	Calculates the total delivery cost for a given order	Int, the total delivery cost
Order	Order	Constructor Order * @param orderNo order number * @param deliveryDate date of delivery * @param customer customer matriculation no * @param deliverTo three word pick up point * @param item items of the order * @param price price for the order including delivery fees	
WebServer	getURLStringForMenus	Simple helper function to put up a valid server string to get menus	String, valid url
	getURLStringForNoFlyZones	simple helper function to put up a valid server string to get no fly zones	String, valid url
	getURLStringForLandmarks	simple helper function to put up a valid server string to get landmarks	String, valid url
	getURLStringForThreeWordsLocation	simple helper function to put up a valid server string to get three word references	String, valid url
	createResponse	A helper function that constructs web server connection	HttpResponse<String> response
	getLandMarks	get landmark coords from server to the landmarks list	Void
	getNoFlyZones	Get no fly zones from the server	ArrayList<Line2D> no fly zones in Line2D
WordParser	Word	Constructor Word With coordinates longitude and latitude	
	parseWord	calls web server for the three word locations and packed in Word	the parsed three word locations with valid coordinates

Part 2: Drone control algorithm

My application produces the geojson file with an arraylist of geojson.point called pointList, and that pointList contains all the points visited by the drone, in steps of 0.00015 drone movement distance. And this PointList is generated based on the ArrayList<LongLat> route parameter of each order.

This route list simply contains the shop + shop (if there's second shop) +⋯ + pick up point of the order, in the form of LangLat. My isNoFlyZone() method checks if any two coordinates crosses the NoFlyZone boundary using the Line2D library and if the LangLat in the route list do cross, my avoidNoFlyZone() Method will loop through Path List (The complete LangLat list adding the route list of every order within orders) and insert a landmark inbetween the two coordinates which does not cross the No-Fly-Zones.

```
/**
 * if there is noflyzone in between two coords in path, add a valid landmark between the two, to avoid the noflyzone
 */
public void avoidNoFlyZone() {
    for(int counter = 0; counter < path.size()-1; counter++){
        if(isNoFlyZone(path.get(counter).longitude, path.get(counter).latitude, path.get(counter+1).longitude, path.get(counter+1).latitude)){
            for(int i = 0; i< server.landmarks.size(); i++) {
                if(!(isNoFlyZone(path.get(counter).longitude, path.get(counter).latitude, server.landmarks.get(i).longitude, server.landmarks.
                        !(isNoFlyZone(path.get(counter+1).longitude, path.get(counter+1).latitude, server.landmarks.get(i).longitude, server.l
                    path.add( index: counter+1, new LongLat(server.landmarks.get(i).longitude, server.landmarks.get(i).latitude));
                    break;
                }
            }
            counter ++; // if the drone moves to a landmark, then we increment that as well
```

Figure 1, the avoidNoFlyZone method in Drone.class

Once that is done, my planPath() method will loop through the path list containing all the key locations visited by the drone and iterate/update the current position holder currentPosition to add all the little 0.00015 moves inbetween the key locations visited by the drone to the final pointList as geojson.Point. And it also identifies the pick up points and shops for which the coordinates need to appear twice in the pointList.

```
TargetPositionFromHome = (int)(currentPosition.distanceTo(targetPosition) / LongLat.DISTANCE_TOLERANCE +
        targetPosition.distanceTo(path.get(path.size()-1)) / LongLat.DISTANCE_TOLERANCE);
if(pointList.size() < (MAXIMUM_NO_OF_MOVES - TargetPositionFromHome)) {
    while(!currentPosition.closeTo(targetPosition)){
        LongLat thiscurrent = currentPosition;
        currentPosition = currentPosition.nextPosition(getAngle(currentPosition,targetPosition));
        LongLat nextcurrent = currentPosition;
        if(isNoFlyZone(nextcurrent.longitude,nextcurrent.latitude,thiscurrent.longitude,thiscurrent.latitude))
            currentPosition = thiscurrent.nextPosition(getAngle(thiscurrent,nextcurrent));
        }
        pointList.add(Point.fromLngLat(currentPosition.longitude,currentPosition.latitude));
        longPath.add(currentPosition);
        pickUpPath.add(currentPosition);
        completePath.add(currentPosition);
```

Figure 2, part of the planPath method in Drone.class, that's used to iterate steps to get the drone to the next key visiting point

And if the line between nextposition and current position crosses the no-fly-zone, the method will try to get another angle and recalculate the nextposition to move around the no-fly-zone.

```
while(isNoFlyZone(current.longitude, current.latitude, target.longitude, target.latitude)) {
    angle += 10;
    if(angle == 360){
        angle = 0;
    }
    target = current.nextPosition(angle);
```

Figure 3 part of the getAngle method in Drone.class that search for an angle to move around the no-fly-zone

The ordering of the visiting points within path list directly come from the ordering of the orders list of order. The orders are comparable to each other in terms of their unit price per distance value. Which is calculated in the getVisitingLongLat method in Drone.class. Price per distance value is defined as the total price of order divided by the distance between shops and final pick up point of the order.

```
    distance += longLat.distanceTo(shops.get(shops.size()-1));
    order.totalDeliveryDistance = distance;
    order.pricePerDistance = (int) Math.round(order.price/order.totalDeliveryDistance);
```

Figure 4 part of the getVisitingLongLat method in Drone.class.

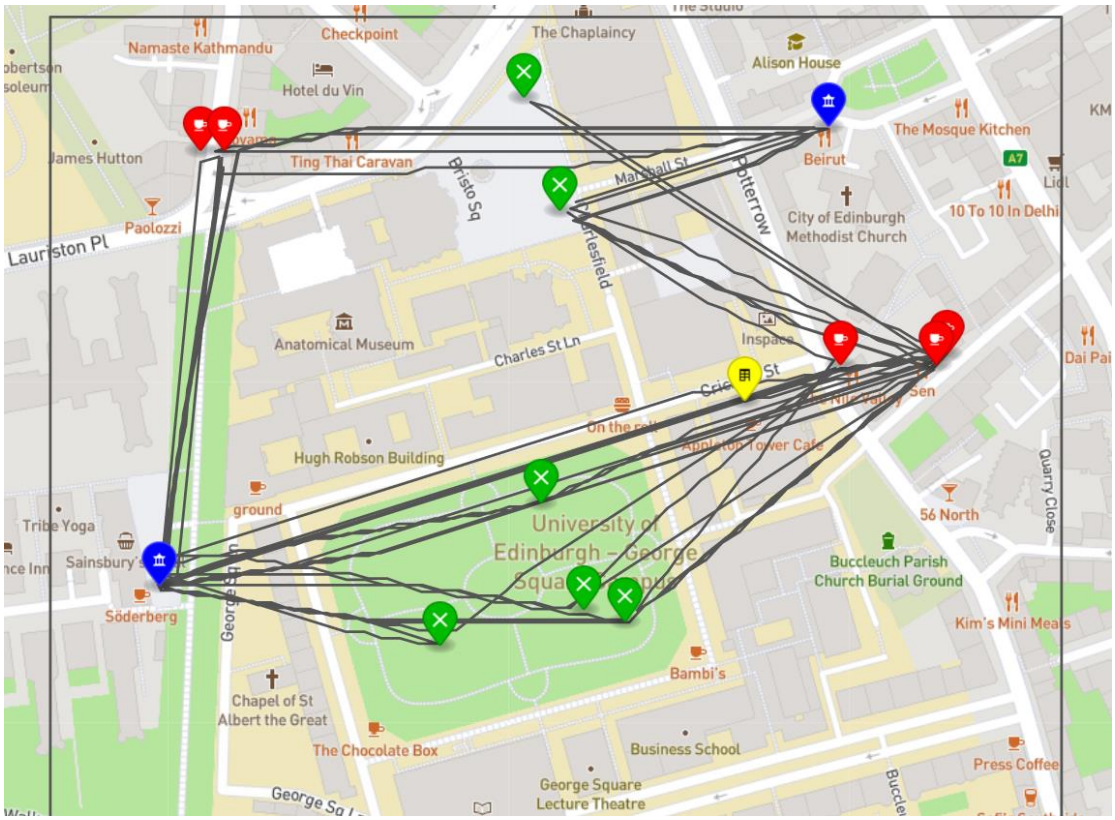


Figure 5, drone flight path on 11-11-2023 with extra markers

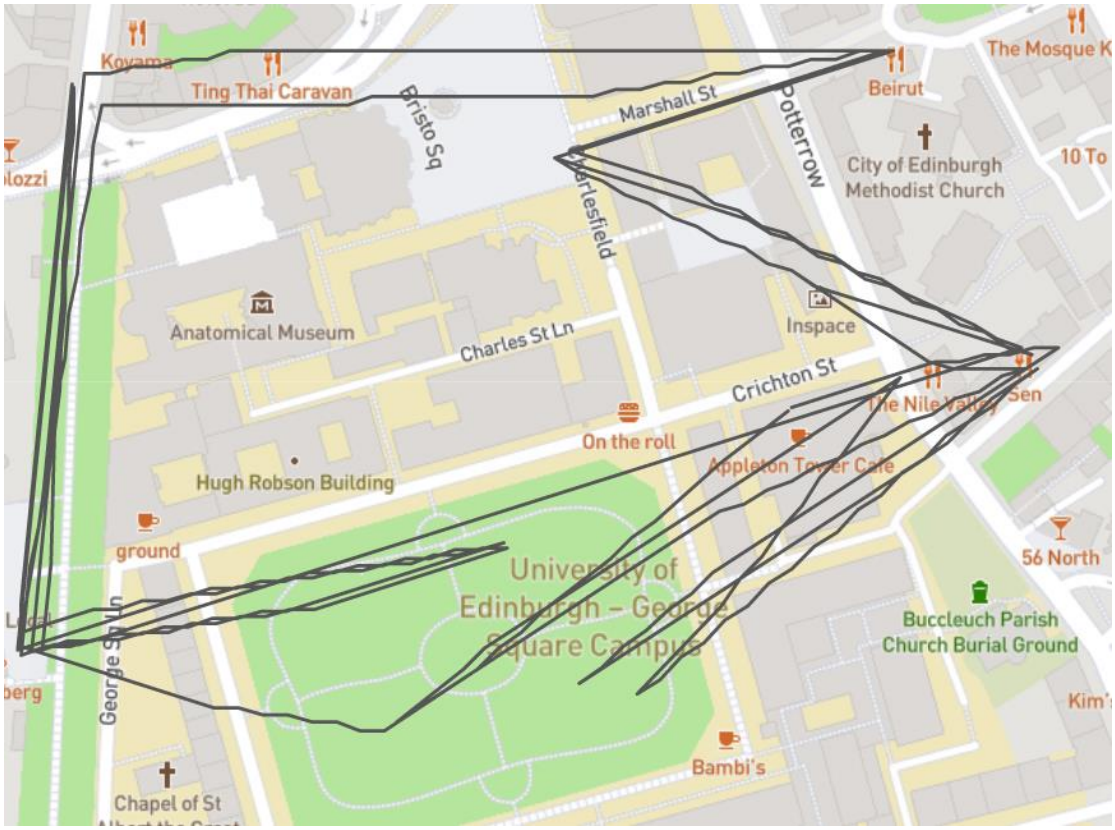


Figure 5, drone flight path on 06-06-2022 without extra markers, the direct outcome of the resulting geojson file produced by the application, without human modification.

Reference

[1] https://geojson.io/ for illustrating the drone flight paths.