ILP Coursework 2 Report

B179485

PizzaDronz

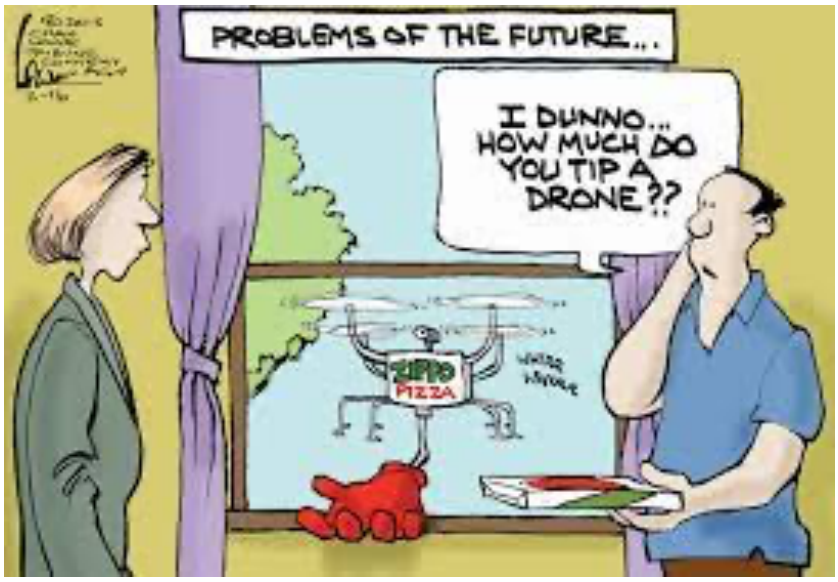


Table of Contents

1. Introduction

This report covers several aspects in relation to the implementation details of the PizzaDronz delivery system. This system is designed as a Java application, built using the Maven build system. The application works by reading in data from a REST server and using this data to make pizza deliveries from various restaurants across Edinburgh to Appleton Tower.

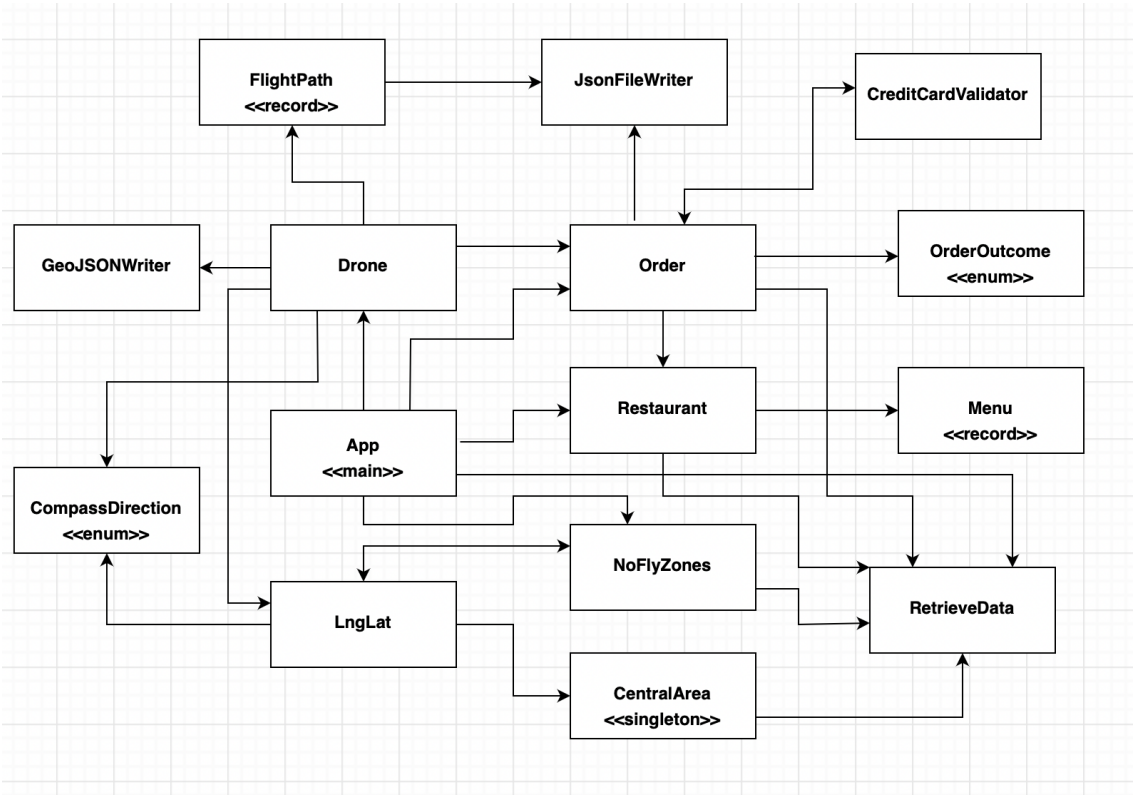
The report will cover the following two main aspects: software architecture description, and drone control algorithm. The first will look more at the underlying implementation of the general application service, whilst the latter specifically focuses on the algorithm used by the drone to make deliveries. Lastly, I will conclude the report by including snapshots of the flightpaths that the algorithm produces for two given dates, which can be viewed on https://geojson.io.

2. Software architecture description

In this section, I will discuss the software architecture of the project. I will start by first considering a UML diagram, which showcases all the classes, and the connections between these classes. After analysing that, I will consider each class individually, and give an insightful description of their functionality. Lastly, I will briefly discuss the exceptions package.

2.1 UML diagram of all classes in the project

Here is the UML diagram, showcasing the connections between each class.



As can be seen from the diagram, the classes communicate with each other quite heavily. The class that communicates most with the rest of the code is App, but this is to be expected, since App is essentially the controller component of the app that binds everything together.

One key thing to note based on my implementation is that there is ***loose coupling***: there are very few strong relationships/ heavy dependencies between the classes. For instance, if changes were made to the NoFlyZone class (more no fly zones were added), this would have a minimal impact on the checking algorithm inside the LngLat class, and so there wouldn't be any changes that need to be made despite the changes to the NoFlyZone class. This allows for a high degree of ***maintainability*** and ***extensibility*** should we decide to further evolve the PizzaDronz project.

2.2 Individual explanations of what each class does

There are a total of 15 component classes that constitute the PizzaDronz app. These are: **App, CentralArea, CompassLocation, CreditCardValidator, Drone, FlightPath, GeoJSONWriter, JsonFileWriter, LngLat, Menu, NoFlyZone, Order, OrderOutcome, Restaurant and RetrieveData**. Below are short descriptions of what each of the 15 component classes do.

2.2.1 App

The App class serves as the main controller class: it gets the input from the command line, validates it, and feeds it to the appropriate classes to be able to produce the desired output. After validating the inputs (checking that there are enough arguments passed as input and ensuring the input can be used by the app), this class retrieves all the necessary data from the REST server and passes it to the Drone class in order to produce all the json and GeoJSON files.

2.2.2 CentralArea

This **singleton** class is used to retrieve the co-ordinates for the Central Area campus from the REST server. The purpose of the Central Area class is to store the location of the Central Area campus as a polygon, to be used in the LngLat class for checking if a point is inside or outside the campus area.

2.2.3 CompassDirection

The CompassLocation class is an Enum class that is used to store all the possible directions in which a drone can go. There are 16 allowed directions: ***East***, ***North***, ***North-East***, ***North-North-East***, ***East-North-East***, ***West***, ***North-West***, ***North-North-West***, ***West-North-West***, ***South***, ***South-East***, ***East-Southeast***, ***South-South-East***, ***South-West***, ***South-South-West***, ***West-South-West***. In addition, the Enum value used to represent a drone's hover move is **null**.

2.2.4 CreditCardValidator

This class is used to validate the credit card details for every order. Since the order validation process was quite large, and a big part of it was related to credit card validation, it seemed appropriate to create a separate class that handles validation for the card number, expiry date and cvv. In addition to performing validation checks, the order outcome is updated accordingly in case an issue is detected.

2.2.5 Drone

The Drone class allows the app to plan a flight path for delivering orders for a given date. This class provides the core of the algorithm that controls our drone: it gives it a list of valid orders and allows it to filter which orders it delivers, and how it delivers them (the specific route taken). In addition, this class co-ordinates all the file writing once the flight planning is done and orders have been delivered. More details on the class's algorithm implementation can be found in ***Section 3***.

2.2.6 FlightPath

This class is used to store details about each individual move a drone makes along a flight path. These objects hold information about each move, such as the longitude and latitude for the current and next position, as well as details about the current order being handled (the order number and order date). In addition, it keeps track of the time (in nanoseconds) needed to compute that specific move. Lastly, the class has a specific method which allows us to write a list of FlightPath objects to a json file. This class is a record, since all the information needed to create these objects can easily be provided by the Drone class, and it also makes use of the new Java 18 features.

2.2.7 GeoJSONWriter

The GeoJSONWriter class is used solely to write objects to a GeoJSON file. In the case of the PizzaDronz app, the Drone objects are the ones being written to GeoJSON: the flight path co-ordinates produced for delivering orders in a given date are written to GeoJSON. This allowes us to make visualisations such as the ones in ***Section 4*** of this report, where we can see the paths as 2D lines on a map, amongst other landmarks such as Appleton Tower, the Central Area campus, the no-fly zones, and the restaurants available on the service.

2.2.8 JsonFileWriter

The JsonFileWriter class is used to write the FlightPath and Order objects to json files. The class has 3 methods: two of them are for creating the necessary json objects depending on which object is being written (Order or FlightPath). Once the necessary json objects have been set up, the third method is called, which is in charge of actually writing the json objects to the appropriate file.

2.2.9 LngLat

The LngLat class allows us to store any location on the map as a pair of co-ordinates in the format (longitude, latitude). These objects allow us to keep track of all the key features on the map: the starting location of any flight plan (Appleton Tower), the edges of all the no fly zones, the edges of the Central Area campus, the restaurant locations, and any new position that the drone moves to when delivering orders. The class also provides ways of checking if a given point is inside a polygon (be it the central area or any of the no fly zones), and also checks if line segments between two points intersect the no fly zones. Lastly, the class allows us to check if a point is close to a given location (useful when delivering orders), as well as computing the Pythagorean distance between 2 points and moving the drone in a given compass direction.

2.2.10 Menu

This class is used to store information about each item on a restaurant's menu. The data for each object in this class is retrieved from the REST server, and each object has a *name* attribute, storing the name of the item as a String, and a *priceInPence* attribute, storing the total price of each item in pence. This class is actually a record, as it does not require any additional fields other than the ones mentioned, and it also makes use of the new Java 18 features.

2.2.11 NoFlyZone

The NoFlyZone class is used to store information about all the no-fly zones that a drone cannot enter. This class stores the information from the REST server and allows other parts of the program to check if points or line segments go through any of the no fly zones (most of this being done inside the LngLat and Drone classes).

2.2.12 Order

This class is used to store all the necessary information about the orders placed on the PizzaDronz app. It allows us to create Order objects containing all the information from the REST server, as well as validate orders according to certain criteria. In addition, it keeps an up-to-date list of all the valid orders for any given date (stored as a static variable), which can be used by the Drone class when planning order deliveries. Lastly, it has a specific method which allows us to write a list of Order objects to a json file.

2.2.13 OrderOutcome

OrderOutcome is an Enum class that stores all the possible ways to label an order. This Enum is used to differentiate between valid and invalid orders, and mark orders as invalid according to what criteria they fail to meet/ violate. There is an additional value to label orders as delivered (once a drone successfully completes an order). The possible values for this Enum are: ***Delivered***, ***ValidButNotDelivered***, ***InvalidCardNumber***, ***InvalidExpiryDate***, ***InvalidCvv, InvalidTotal, InvalidPizzaNotDefined, InvalidPizzaCount, InvalidPizzaCombinationMultipleSuppliers, Invalid***.

2.2.14 Restaurant

This class stores all the necessary information about the restaurants available on the PizzaDronz app. Restaurant objects have a name, list of menu items (corresponding to the restaurant's menu), and location (which is stored as a LngLat object). The location is used as part of the path finding algorithm in the Drone class, whereas the list of menu items is used as part of the order validation process, to check if the list of ordered items is valid. Moreover, the Restaurant class provides a static method called *getRestaurantsFromServer()* which allows us to retrieve all the restaurant data stored on the REST server without needing a Restaurant object.

2.2.15 RetrieveData

The RetrieveData class is used to fetch data from the REST server. This class has one generic method that allows the app to get data for all different objects. In this way, fetching Restaurant, Order, NoFlyZone and CentralArea objects can all be done using the same generic *getData()* method, by simply specifying the appropriate extension and needed return type as method parameters.

2.3 Exceptions package

The exceptions package stores a list of exception classes that allow the app to display customised exceptions in case of invalid orders. These exceptions are strongly interconnected with the OrderOutcome Enum class, since, for each invalid outcome, a corresponding exception class was created. Currently the code does not make use of the exceptions package, but it was included in there in case this will be needed when the PizzaDronz service is extended.

3. Drone control algorithm

Having looked at the software architecture, I will now focus on my drone control algorithm. This algorithm, which is a ***greedy algorithm***, represents how my drone filters the orders, makes decisions on which orders to deliver, and creates the appropriate flight paths to make deliveries.

3.1 Concept of greedy algorithms & how it relates to my algorithm

As mentioned earlier, the algorithm is a ***greedy algorithm***. This means that it makes the best optimal choice at each small stage with the goal of this eventually leading to a globally optimum solution. Essentially, the algorithm picks the best solution at the moment without regard for consequences. By picking the best immediate output and not consider the big picture, this is why the algorithm is regarded as greedy.

In our case, the algorithm always tries to find the best angle and closest distance in that direction in order to make a move. In doing so, it gets one step closer to the destination restaurant each time; however, in certain cases, these moves can be suboptimal in the long run. One such case is when trying to avoid no fly zones: instead of going around a shorter route (which would be identified by an algorithm such as A* search), the greedy algorithm takes a longer route around the no fly zone that still gets it to the destination (Appleton Tower), but by using a greater number of moves.

3.2 How the algorithm plans a route to a restaurant

The main method that is used to create a flight plan is the *planFlightPath()* method in the Drone class. In this method, we loop over all the valid orders, and for each individual order, retrieve the restaurant location (as a LngLat object) and check if the drone's battery allows for that order to be delivered. If it can be delivered, we deliver it, and set the order's outcome to be ***OrderOutcome.Delivered***. Once all the orders have been considered, we write the appropriate output files (json and GeoJSON).

The check for being able to deliver an order is performed by the *canDeliverOrder()* method. In this method, we create a deep clone of the drone's current position and run the *computeFlightPath()* method to compute the actual route to the restaurant. This method is called twice since we also need to account for the return path back to the drop-off location close to Appleton Tower. Once we have computed the total number of moves needed to make the delivery, we check that the drone's current battery suffices. If it does, we update the number of moves left, and the drone's current position, and return true (since a delivery could be made). Otherwise, we return false since the drone's battery is not sufficient.

As previously mentioned, the actual route planning method is the *computeFlightPath()* method. This method essentially implements the greedy algorithm. First, we start by checking if we are close to the destination (in this case, the restaurant). If we are, then we simply make a hover move to collect the order (or alternatively deliver the order if we're returning to Appleton Tower). If we are not close to the destination, we then explore all

possible compass directions to identify the best angle. For each angle, we check that the follow-up map location is not in a no-fly zone, or that the line segment between the current and follow-up locations does not cross a no-fly zone. Once we have checked this, we gather all possible angles into a list, and filter them based on which one gives the closest distance. The angle that results from this filter is the angle that we pick, and we make a move accordingly.

One thing worth noting about the algorithm is that it used to have one fatal flaw: the drone could get stuck in an infinite loop trying to figure out a path around no-fly zones. This has been remedied, however, and the program now runs as expected. The remedy solution was to have a variable that keeps track of the previous angle being taken, constantly update this variable and discard it as a potential option each time we try to make a move. In this way, the drone avoids being stuck in an infinite loop by never going back to the previous position that might have caused that infinite loop.

3.3 How the algorithm plans a return route to Appleton Tower

The algorithm uses the same procedure outlined in ***Section 3.2*** to create a new route from the restaurant back to the Central Area campus and identify a drop-off point that is close to Appleton Tower. It is worth noting that my algorithm does not retrace the steps taken to get to the restaurant from Appleton, and reverse that path, but rather it computes a new, different path to deliver the order.

3.4 Maximising the *sampled average number of pizza orders delivered* metric

The algorithm manages to deliver an average of around 20 orders per day, which is nearly 50% of all the available valid orders. One thing that the algorithm could do to improve this score is prioritising orders that take fewer moves to complete. This might result in an improvement of having an additional 2-3 orders delivered per day (on average).

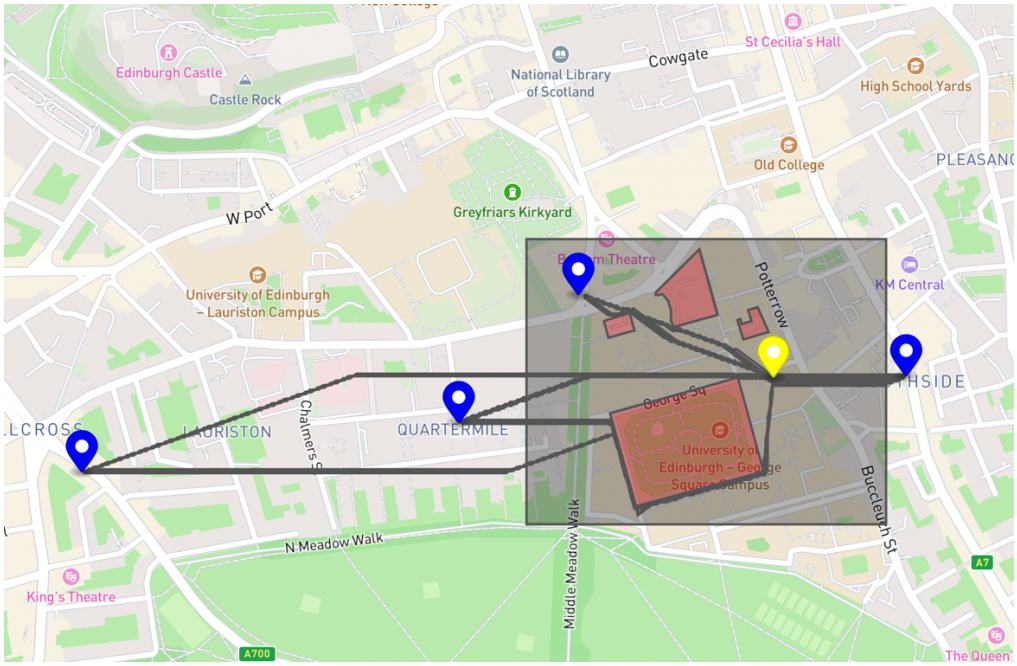
However, there is one key reason behind not implementing this heuristic: it could bias the algorithm against restaurants that are far away, and in the long run potentially monopolising the closest restaurant if too many valid orders from that restaurant are made. Therefore, in order to avoid a situation where some restaurants miss out on the PizzaDronz service simply due to their relative location to Appleton Tower, the algorithm selects the orders using more of a Round Robin approach, ensuring more diversity.

This is ensured by the randomised ordering of the valid orders list. The orders from the REST server are added to a list of valid orders after they have passed all validation checks. However, these orders are stored in a random order, and hence this ensures that an order from a far-away restaurant will not be ignored.

4. Figures of GeoJSON output

Lastly, to finish off this report, I will include snapshots of two GeoJSON output files that my algorithm produces. The two dates being considered here will be “2023-03-19” and “2023-05-18”.

4.1 First GeoJSON graph (2023-03-19)



4.2 Second GeoJSON graph (2023-05-18)