The University of Edinburgh
School of Informatics

ILP project report

Report on the implementation of drone control software

Author
Zhi Qi Lee

December 2020

Contents

Chapter 1

Software architecture

1.1 Introduction

Autonomous drones are the ideal technology for efficiently collecting data from air quality sensors distributed around an urban geographical. My project aims to program an autonomous drone to fly around and collect the sensor readings of air quality and to produce scientific visualisations of the data. This will ultimate help the researchers to analyse urban air quality. If this feasibility study is successful the researchers will apply for funding to conduct their research on a much larger scale, collecting and analysing air quality readings all across the city of Edinburgh.

Through this document, the software architecture and the architectural decisions which have been made on the program will be described.

1.2 Architectural Goals and Constraints

There are some key requirements and constraints that have a significant bearing on the architecture.

1.2.1 The implementation of the drone control software should be considered to be a prototype that would be passed to a research team for maintenance and further development. This implies that the clarity and readability of the code is important.

1.2.2 The researchers have stored the synthetic data and other important files on a web server. Hence, the program must be able to get files named `air-quality-data.json`, `details.json` and `no-fly-zones.geojson` which contain the relevant data for drone programming from the web server.

1.2.3 The program must be able to parse the data that it obtains from the web server .

1.2.4 Given the initial starting location of the drone and the date, the program must be able to calculate a flight path which visits as many of the sensors listed for that date as possible and returns as close to its initial starting location once all sensors have been visited.

1.2.5 The program must be able to produce a report on the drone's flight which takes the form of a Geo-JSON map and a log file which lists where the drone has been and which sensors it connected to.

1.3 Architectural Decisions

1.3.1 I tried to construct a reasonable design without excessive coupling which means the strength of the interconnections between classes are weak to increase understandability and maintainability of the program. Hence, instead of a large class, my program is composed of many small classes.

1.3.2 I tried to achieve high cohesion by making the classes focus on what they should do and ignore the remainder of irrelevant information so that only methods relating to the intention of the class are included. I divided the program into smaller components and make sure all the classes have fairly clear and distinct responsibilities.

1.3.3 Consider the program might be used by a researcher who don't know much about drone and Geo-JSON, I separated the *class App* which known to handling I/O from the details of how the autonomous drone is programmed and the Geo-JSON map is drew. Hence, the program would be less error-prune.

1.3.4 I declared a few classes without any method. They are read-only data structures that generally do not contain any business logic or behavior. They are either used for deserialisation data from web server or as a data transfer object to loose the coupling between the data from web server and this program, allowing us to more easily maintain and do further development.

1.3.5 I tried to achieve encapsulation by making those details inaccessible. I declared the variables as `final` if their values would not be changed after the initialisation otherwise `private` which can only be accessed by using the getter. Other than that, method would only be `package-level` if it is a getter. Besides, `package-level` setter could only be accessed by `private` method in other classes. Thus, they are safer from outside interference and misuse.

1.4 Components

This program consists of 8 top classes and 2 nested classes.

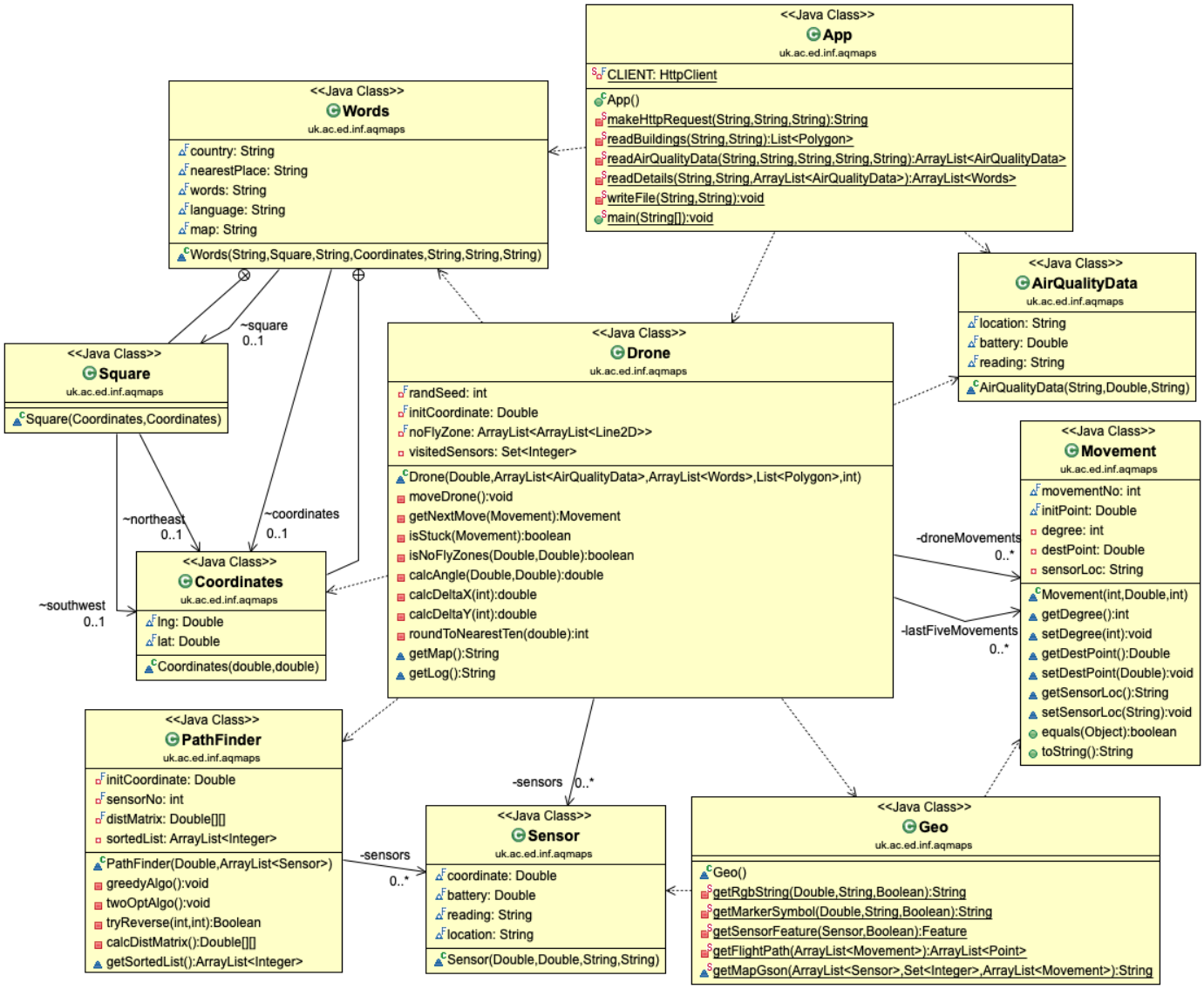


Figure 1.1: The complete UML diagram.

For the 8 top classes, we can separate them into 4 components to make the explanation clear and concise.

1.4.1 Classes without method

There are three classes without any method. *class AirQualityData* and *class Words* are used for parsing the string file obtained from the web server. Whereas, *class Sensor* aggregates data from the above two classes but stored only drone programming related attributes. Besides, *class Words* has two static nested classes, *class Square* and *class Coordinates* which do not have any methods as well. All the fields of the classes mentioned above are declared as final since their values would not be changed after the initialisation. Figure 1.2 on the next page shows classes without method.

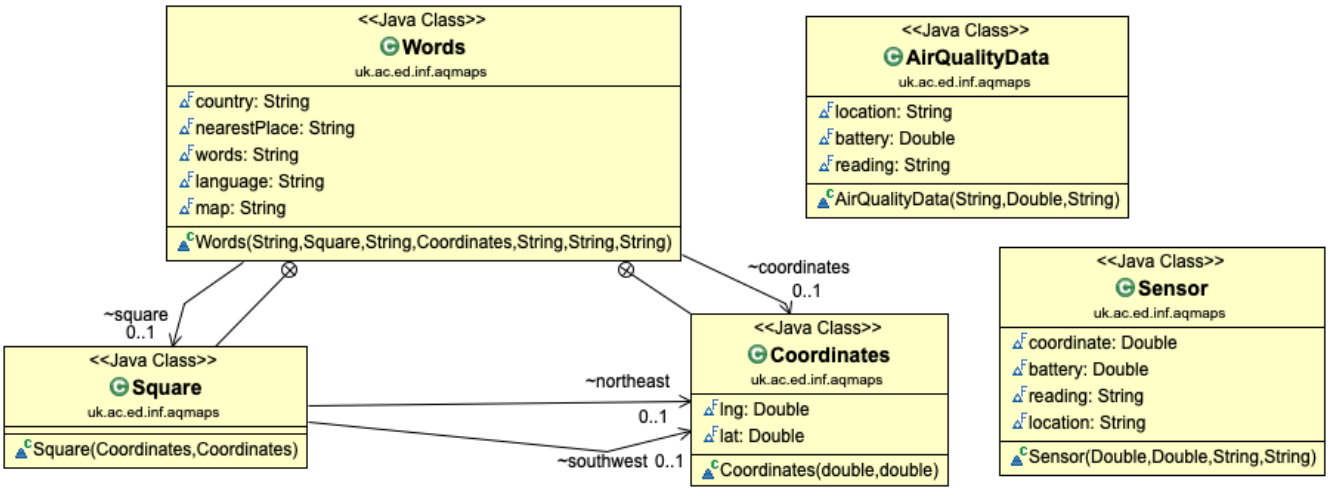


Figure 1.2: Classes without method.

1.4.2 Main class

The main class of this Java program is *class App* and its main() method would be the execution entry point of the whole program. Command line arguments are passed as an string array parameter to the main() method. Due to the considerations mentioned in part 1.3.3, *class App* only handles I/O actions, makes HTTP request to web server, parses JSON string to Java objects and creates two files. It does not have any implementation about the drone control algorithm.

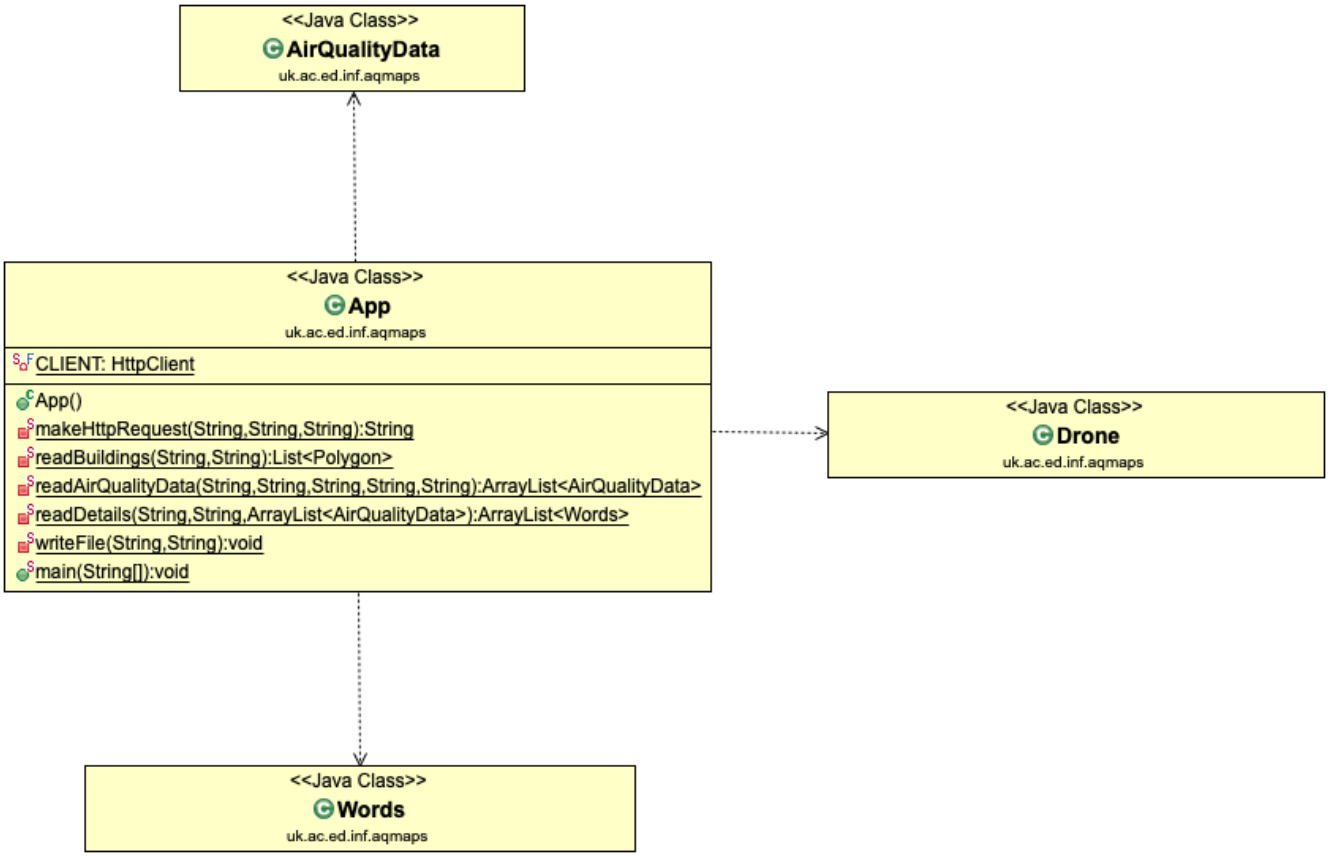


Figure 1.3: the UML diagram of *class App* and all its relationships

1.4.3 Classes for drone control

According to the program specifications, the drone has to visit 33 sensors which are listed on the file named `air-quality-data.json`. The *class Pathfinder* decides the order of sensors to be visited. Then, the *class Drone* decides the drone flight around the air-sensors and back to the start location of the flight, while avoiding all of the no-fly zones and make sure the drone remain strictly inside the confinement area all the times. The *class Movement* is used to describe each movement of the flight. It helps produce the Geo-JSON map and the flight path log file which lists where the drone has been.

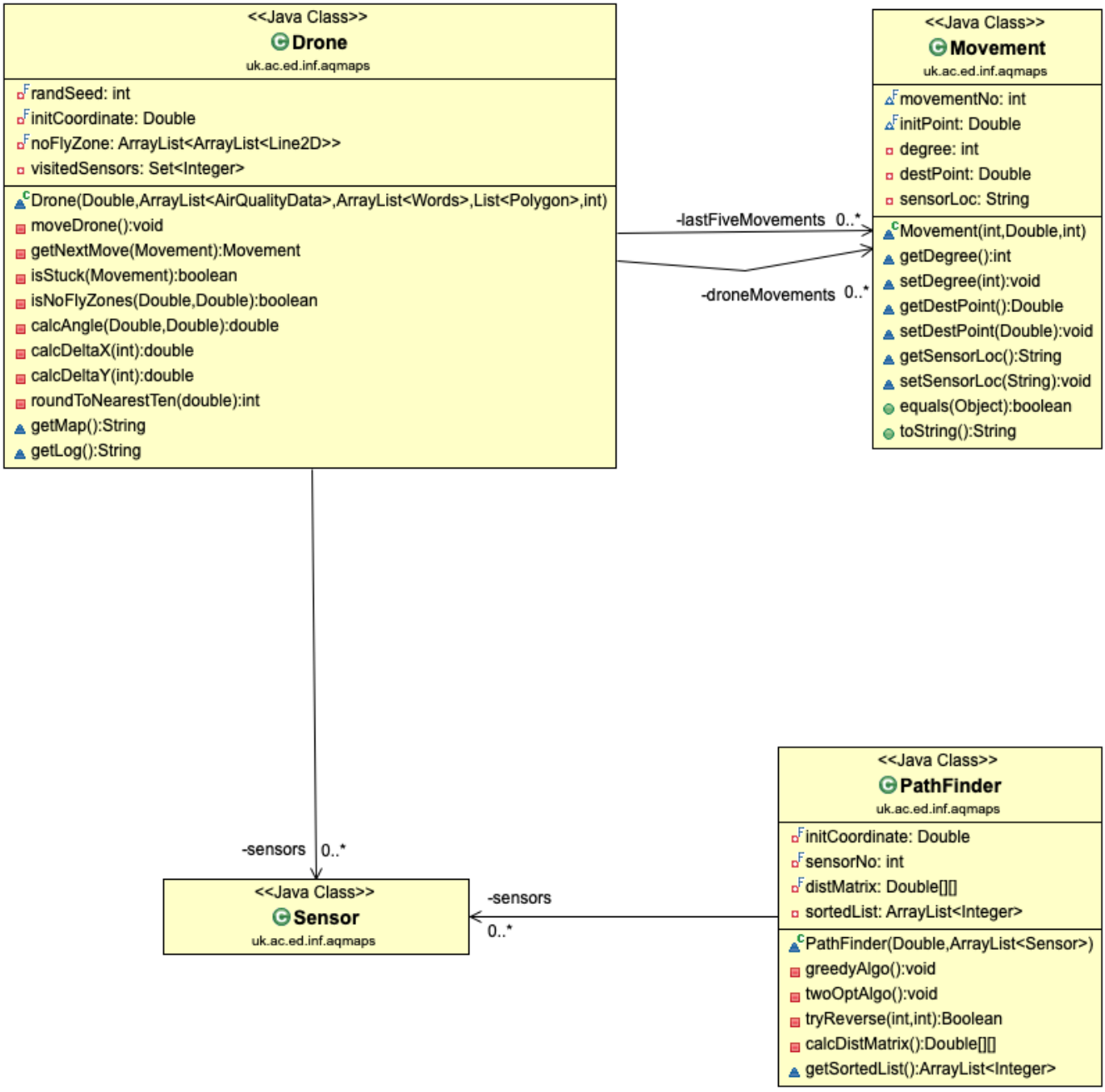


Figure 1.4: the UML diagram of classes for drone control

1.4.4 Classes for generating Geo-JSON map

The *class Geo* plot the drone flight path as a Geo-JSON map. The Geo-JSON map also includes the locations of the sensors plotted using coloured markers which represent the levels of air pollution detected by the sensors.

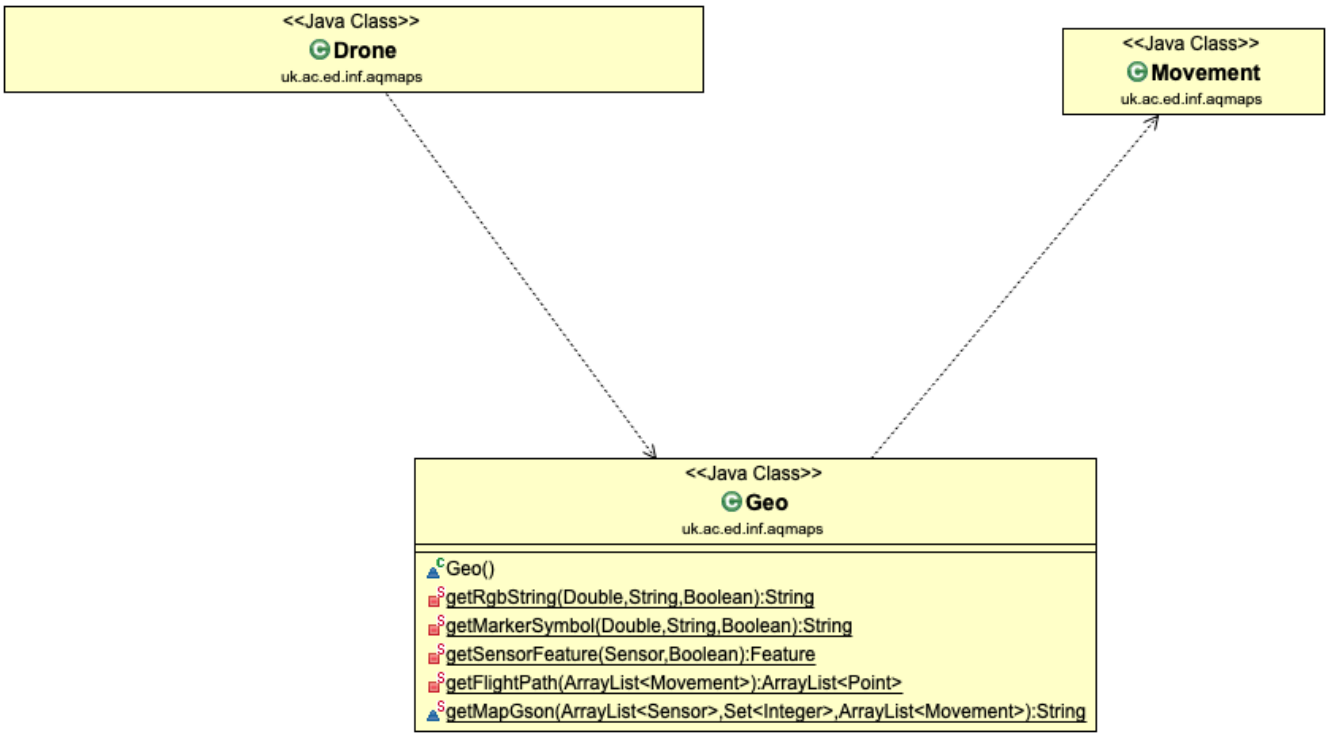


Figure 1.5: the UML diagram of *class Geo* and all its relationships

Chapter 2

Class Documentation

Constructors

Constructor	Description
`App()`	

Method Summary

All Methods | **Static Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`static void`	`main(java.lang.String[] args)`	
`private static java.lang.String`	`makeHttpRequest(java.lang.String url, java.lang.String server, java.lang.String port)`	Make HTTP Get request.
`private static java.util.ArrayList<uk.ac.ed.inf.aqmaps.AirQualityData>`	`readAirQualityData(java.lang.String day, java.lang.String month, java.lang.String year, java.lang.String port, java.lang.String server)`	Get the file named "air-quality-data" in JSON format from the web server.
`private static java.util.List<com.mapbox.geojson.Polygon>`	`readBuildings(java.lang.String port, java.lang.String server)`	Get the file named "no-fly-zones" in Geo-JSON format from the web server.
`private static java.util.ArrayList<uk.ac.ed.inf.aqmaps.Words>`	`readDetails(java.lang.String port, java.lang.String server, java.util.ArrayList<uk.ac.ed.inf.aqmaps.AirQualityData> aqDatas)`	For every what3words location in the `aqDatas` ArrayList, get its file named "details" in JSON format from the web server.
`private static void`	`writeFile(java.lang.String content, java.lang.String filePathName)`	Write the string `content` into a file with the name of `filePathName`

Figure 2.1: class documentation of class App

Constructors

Constructor	Description
`Drone(java.awt.geom.Point2D.Double initCoordinate, java.util.ArrayList<uk.ac.ed.inf.aqmaps.AirQualityData> airQualityData, java.util.ArrayList<uk.ac.ed.inf.aqmaps.Words> wordsList, java.util.List<com.mapbox.geojson.Polygon> polygons, int randSeed)`	Construct and initializes a `Drone` with the specified initial coordinate, list of air quality data, list of What3Words, list of buildings and random number seed for this program.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`private double`	`calcAngle(java.awt.geom.Point2D.Double centerPoint, java.awt.geom.Point2D.Double targetPoint)`	Calculate the angle in the unit of degree between the line defined by these two points and the horizontal axis
`private double`	`calcDeltaX(int degreeToMove)`	Calculate a new longitude caused by travelling in the specific direction
`private double`	`calcDeltaY(int degreeToMove)`	Calculate a new latitude caused by travelling in the specific direction
`(package private) java.lang.String`	`getLog()`	
`(package private) java.lang.String`	`getMap()`	
`private Movement`	`getNextMove(Movement move)`	Get the next move of the drone flight by checking if the move is in No fly zone or lead to a stuck If the move is illegal then start searching a new move
`private boolean`	`isNoFlyZones(java.awt.geom.Point2D.Double currentPoint, java.awt.geom.Point2D.Double nextPoint)`	Check if the move to `nextPoint` will into or pass through any of the No Fly Zones.
`private boolean`	`isStuck(Movement move)`	Check if the drone is stuck by comparing the `move` with the latest five move.
`private void`	`moveDrone()`	Starts the drone flight and updates the flight path into this.droneMovements
`private int`	`roundToNearestTen(double degree)`	Round the degree to the nearest ten

Figure 2.2: class documentation of class Drone

Constructors

Constructor	Description
`PathFinder(java.awt.geom.Point2D.Double initCoordinate, java.util.ArrayList<uk.ac.ed.inf.aqmaps.Sensor> sensors)`	Constructs and initializes a `Pathfinder` from the specified starting coordinate and a list of sensors to be visited.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`private java.lang.Double[][]`	`calcDistMatrix()`	It adds the starting point of the flight as the first node.
`(package private) java.util.ArrayList<java.lang.Integer>`	`getSortedList()`	
`private void`	`greedyAlgo()`	constructs an initial order to visit the sensors and stores it to the `this.sortedList`
`private java.lang.Boolean`	`tryReverse(int i, int j)`	Check if the reverse of the segment between i and j is better than the original
`private void`	`twoOptAlgo()`	improve the initial tour and stores the updated order to visit the sensors to the `this.sortedList`

Figure 2.3: class documentation of class PathFinder

Constructors

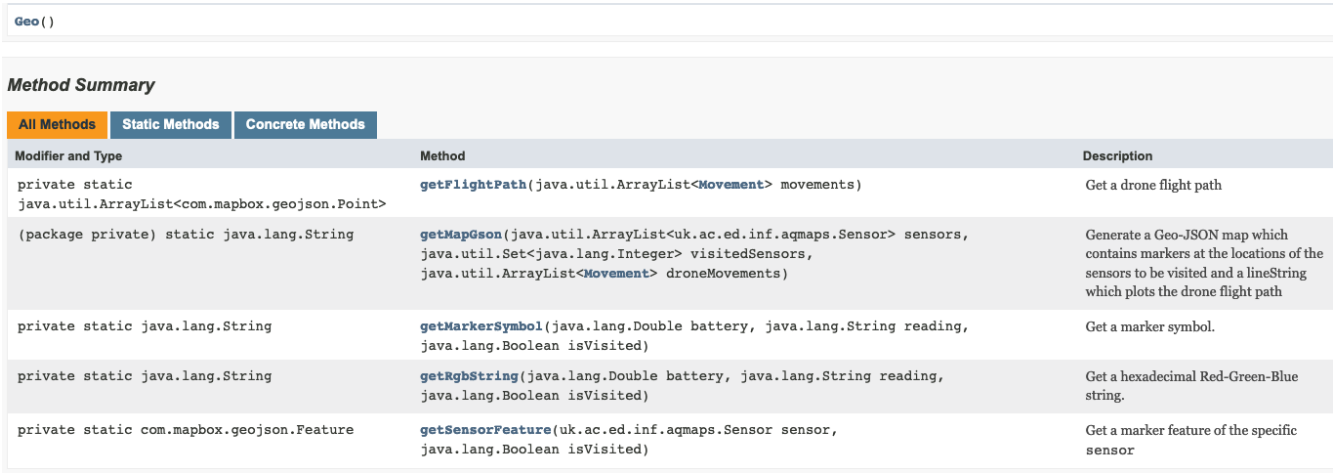
Constructor	Description
`Movement(int movementNo, java.awt.geom.Point2D.Double initPoint, int degree)`	A single move which is a straight line of length 0.0003 degrees.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`boolean`	`equals(java.lang.Object obj)`	
`(package private) int`	`getDegree()`	
`(package private) java.awt.geom.Point2D.Double`	`getDestPoint()`	
`(package private) java.lang.String`	`getSensorLoc()`	
`(package private) void`	`setDegree(int degree)`	
`(package private) void`	`setDestPoint(java.awt.geom.Point2D.Double destPoint)`	
`(package private) void`	`setSensorLoc(java.lang.String sensorLoc)`	
`java.lang.String`	`toString()`	

Figure 2.4: class documentation of class Movement



Geo()

Method Summary

All Methods | Static Methods | Concrete Methods

Modifier and Type	Method	Description
private static java.util.ArrayList<com.mapbox.geojson.Point>	getFlightPath(java.util.ArrayList<Movement> movements)	Get a drone flight path
(package private) static java.lang.String	getMapGson(java.util.ArrayList<uk.ac.ed.inf.aqmaps.Sensor> sensors, java.util.Set<java.lang.Integer> visitedSensors, java.util.ArrayList<Movement> droneMovements)	Generate a Geo-JSON map which contains markers at the locations of the sensors to be visited and a lineString which plots the drone flight path
private static java.lang.String	getMarkerSymbol(java.lang.Double battery, java.lang.String reading, java.lang.Boolean isVisited)	Get a marker symbol.
private static java.lang.String	getRgbString(java.lang.Double battery, java.lang.String reading, java.lang.Boolean isVisited)	Get a hexadecimal Red-Green-Blue string.
private static com.mapbox.geojson.Feature	getSensorFeature(uk.ac.ed.inf.aqmaps.Sensor sensor, java.lang.Boolean isVisited)	Get a marker feature of the specific sensor

Figure 2.5: class documentation of class Geo

Chapter 3

Drone Control Algorithm

3.1 Overview

According to the program specifications, the drone has to visit 33 sensors which are listed on the file named `air-quality-data.json` and return close to the starting point of the flight path. Besides, the drone is a battery-powered device so the number of moves it can make is limited to 150 moves. Hence, we could not let the drone simply fly around and expect it can finish the requirements within 150 moves. In order to satisfy the requirements, there are two important question we have to solve:

- How to decide the order to visit the 33 specific sensors?
- How their flight around the air-quality sensors and back to the start location of their flight, while avoiding all of the no-fly-zones and remains strictly inside the confinement area?

Section 3.2 and 3.3 address these two questions respectively.

3.2 Deciding the order

Since we have to visit all the 33 sensors within 150 moves, the shorter the distance between the sensors, the higher the probability the full flight path is short. Hence, I implemented the Greedy algorithm to address this problem.

The greedy algorithm builds a tour by always moving to the closest sensor (among the sensors not yet been visited) from the starting point which guarantees the distance between every two sensors in the tour is the shortest. There is a flowchart of the Greedy algorithm on the next page.

Although the greedy algorithm makes the optimal choice at each step when it attempts to build the tour, it may still fail to find the shortest tour since it do not consider all the sensors and unaware of future choices it could make. Thus, I think a tour improvement algorithm is required for improving the tour obtained from greedy algorithm. I chose 2-Opt algorithm as the tour improvement algorithm.

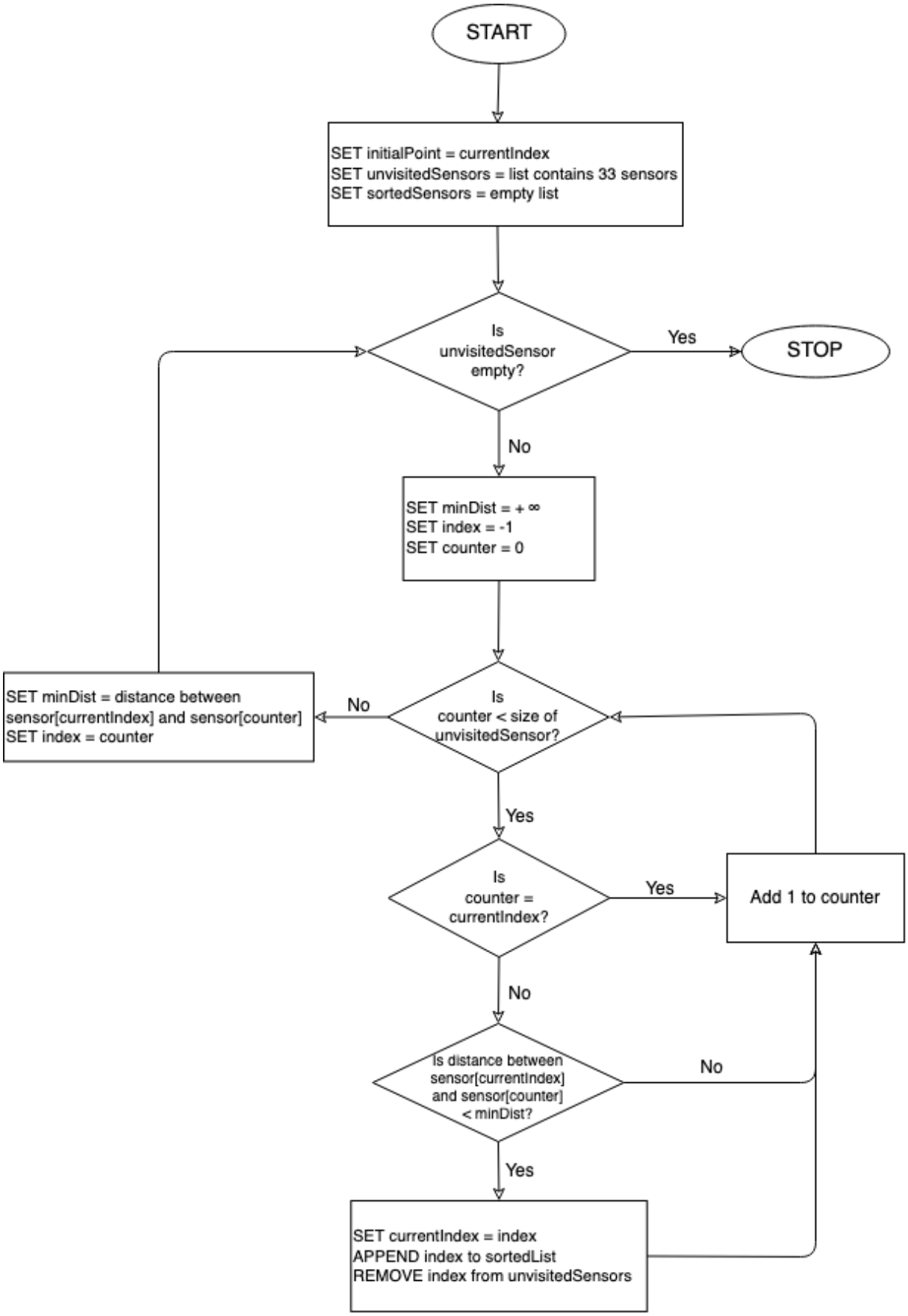


Figure 3.1: the flowchart of the Greedy algorithm

The 2-Opt algorithm considers the cost (in terms of distance) of a contiguous sequence of sensors on the current tour and then considers the effect on cost of these sensors be visited in the reverse order. If the reversal will make the cost of the tour lesser, then change the corresponding part of tour to this reversal. 2-Opt algorithm improve the initial tour gradually until there is no more improvement can be made.

In summary, I use greedy algorithm to construct a initial order to visit the 33 sensors and I then use 2-Opt algorithm to improve the initial tour.

Algorithm	Total number of movements
Greedy	128
Greedy + 2-Opt	110

Figure 3.2: 2020/06/06 - Map using only Greedy algorithm

Figure 3.3: 2020/06/06 - Map using Greedy algorithm then 2-Opt algorithm

3.3 Avoiding obstacles and remaining in confinement area

First of all, We have to define what is an obstacle means in the program. I implemented a single obstacle as a list of *line2D.Double* and a move is considered illegal if the straight line of length 0.0003 degrees intersects with any lines in the list. Due to that, I can now implement the confinement area as 4 *line2D.Double* objects and all the buildings included the confinement area form a big list of *line2D.Double* named `no-fly-zone`. Before making any move, the program would calculate the degree of next possible move:

$$degree = \arctan 2(y_1 - y_2, x_1 - x_2)$$

where (x_1, y_1) is the current point and (x_2, y_2) is the next sensor point. Since the drone can only fly in a direction which is a multiple of ten degrees[1]. The program would round the degree to the nearest ten and check whether if a move in the degree is illegal. If the move is illegal, then the program would start finding the next valid move by adding 10 degrees every two counter and reverse the sign of number every two counter. The program would loop until it found the next valid move.

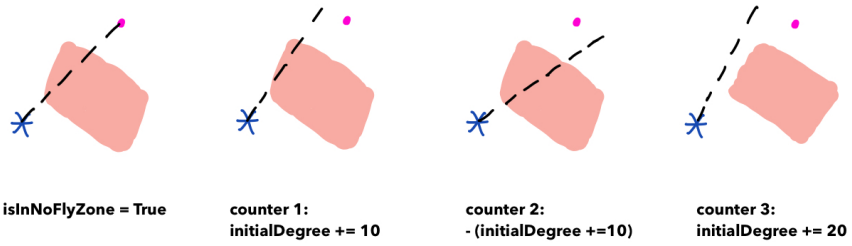


Figure 3.4: While isInNoFlyZone = true, searching for next move

However, there is a flaw in this mechanism. The drone might stuck in somewhere within the confinement area. It stuck flying back and forth until it reaches the maximum limit number of moves and fail to return to the starting point.

Figure 3.5: The drone is stuck.

[1]By convention, 0 means go East, 90 means go North, 180 means go West and 270 means go South, with the other multiples of ten between 0 and 350 representing the obvious directions between these four major compass points.Negative angles and angles greater than 350 are not allowed.

Therefore, random fly might be needed if the drone is stuck. I stored a list of last five *movement*s. If the next move is in the No Fly Zones and is exactly same with one of the last five *movement*s, then the drone would set the next degree to move randomly and check if the move is valid and does not stuck until it found a valid move in order to break out of the cycle.

Figure 3.6: output map for the date 2020/04/04

Figure 3.7: output map for the date 2020/11/11