GABRIEL OHOWA OWINO

s1871518

INFORMATICS LARGE PRACTICAL 2020-2021

School of Informatics, University of Edinburgh

Coursework 2

Software Engineering Report

Table Of Contents

Table Of Contents **2**

Background Review **3**

Software Architecture Description **3**

Class Documentation **6**

- Models 6
    - BaseEntity 6
    - Sensor 6
    - NoFlyZone 6
    - Path 7
    - DroneMove 7
    - Drone 8
- Services 9
    - SensorService 9
    - NoFlyZoneService 9
    - FlightService 9
    - OrderSolver 10
    - FlightPathPlanner 10
- HelperClasses 11
    - What3WordsConverter 11
    - GeometryHelper 11

Drone Control Algorithm **12**

References **15**

Background Review

The task at hand is to design and program an autonomous drone that will collect readings from air quality sensors distributed among an urban area as part of a research project to analyse urban air quality. The program needs to be a functional prototype that will be used as part of a pitch to apply for greater funding. In addition, the program will be passed on to a team for further development thus scalability, readability and clarity is vital.

Software Architecture Description

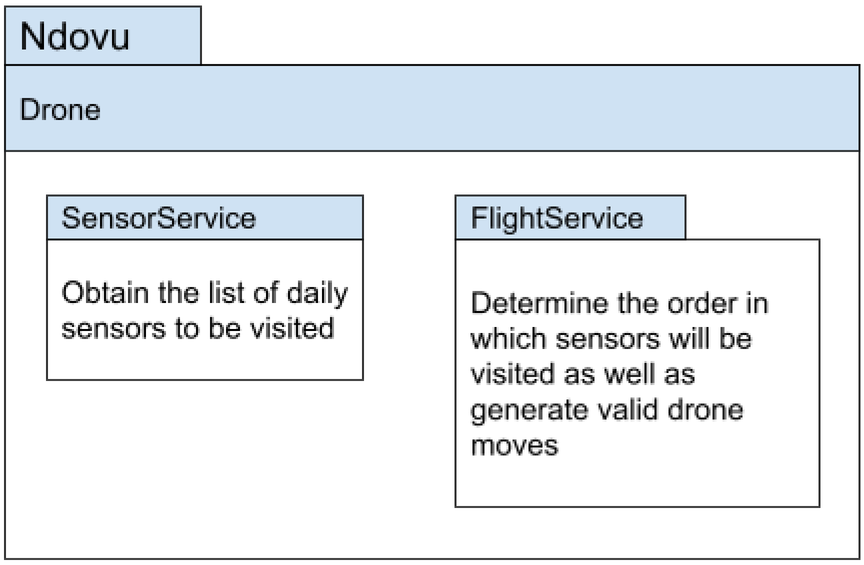
Our drone program should be able to perform the following core tasks:

1. Determine which Sensors are to be visited.
2. Determine which order to visit the sensors.
3. Generate valid moves from its current position towards a sensor. The valid moves can then be interpreted by the drone's hardware to propel it to move appropriately.

With scalability in mind, our program should be open for extension but closed for modification. This means that our program should be designed in such a way that it can be easily extended to support the inclusion of more drones or sensors as the project grows.

Also, classes within our program should be loosely coupled to allow for flexibility within our drone. By doing this, our drone will have the capacity to support multiple functionality i.e. multiple sensor visitation orders; multiple ways of determining the next move etc.

Below is a visual design of the software architecture of our drone control program:



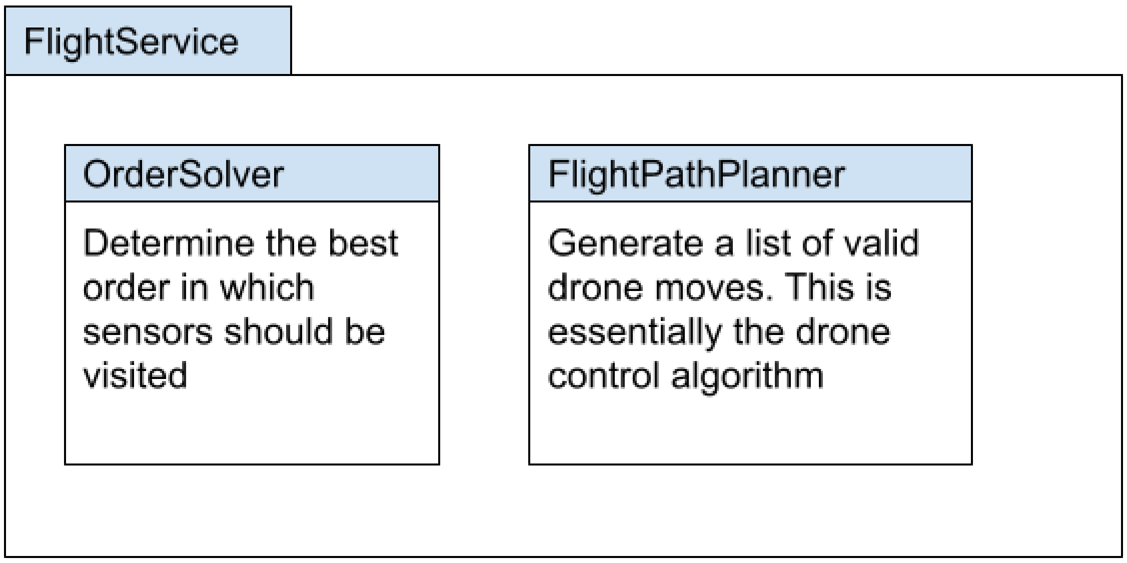
Ndovu
Drone
SensorService
Obtain the list of daily sensors to be visited
FlightService
Determine the order in which sensors will be visited as well as generate valid drone moves

Drone Control module

Our flight service module, will be incharge of deciding the order in which the sensors will be visited and as well as generate valid drone moves. To this end, it will contain 2 classes: ***FlightPathPlanner.java*** and a second class that implements the interface ***OrderSolver.java***. The latter class is responsible for determining the choice of program that determines how it decides on the order of the visited sensors.

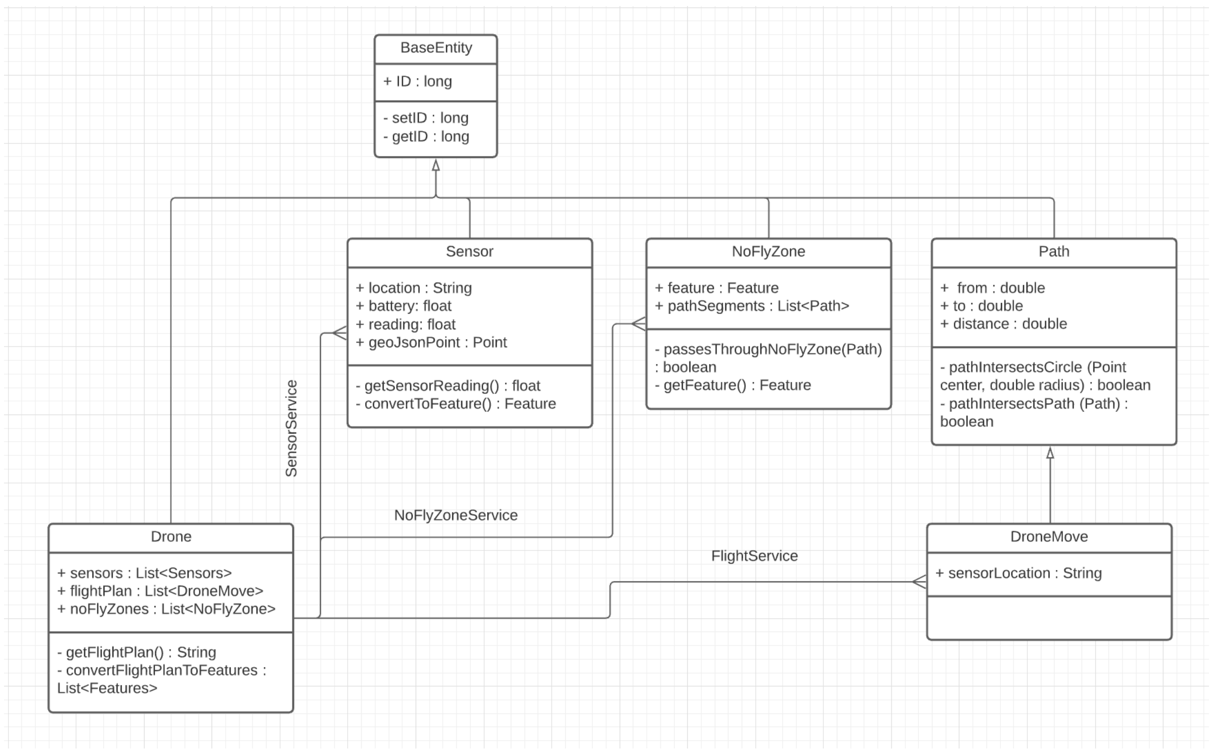
In this case, a third party open source free software is implemented for *OrderSolver.java*. Google OR-Tools are chosen for their reliability, scalability and performance. Using it, we can still be able to determine the optimal visitation order our drone should follow even as the number of visited sensors increase.

The *FlightPathPlanner.java* class is responsible for generating drone moves with regards to the drone control algorithm.


FlightService
OrderSolver
Determine the best order in which sensors should be visited
FlightPathPlanner
Generate a list of valid drone moves. This is essentially the drone control algorithm

FlightService Module

Using object oriented programming principles and design, we can therefore model our program around the Drone, Sensor and NoFlyZones. Here is a complete visual of how the UML class design would look like.


BaseEntity
+ ID : long
- setID : long
- getID : long
Sensor
+ location : String
+ battery: float
+ reading: float
+ geoJsonPoint : Point
- getSensorReading() : float
- convertToFeature() : Feature
NoFlyZone
+ feature : Feature
+ pathSegments : List<Path>
- passesThroughNoFlyZone(Path) : boolean
- getFeature() : Feature
Path
+ from : double
+ to : double
+ distance : double
- pathIntersectsCircle (Point center, double radius) : boolean
- pathIntersectsPath (Path) : boolean
SensorService
NoFlyZoneService
FlightService
Drone
+ sensors : List<Sensors>
+ flightPlan : List<DroneMove>
+ noFlyZones : List<NoFlyZone>
- getFlightPlan() : String
- convertFlightPlanToFeatures : List<Features>
DroneMove
+ sensorLocation : String

Basic UML class based design

Class Documentation

Models

BaseEntity

This is a class that is responsible for assigning id values to objects. The id values are auto generated using javax.persistence APIs taking into account the object type and hierarchy (strategy = IDENTITY). This makes it easier when dealing with multiple objects of the same type or in future when incorporating storage methods.

Methods	**@parameters**	**@return**
Constructor	N/A	N/A
getID - Returns the generated id value	N/A	long id

Sensor

This class models the sensor object. It extends *BaseEntity*.

Methods	**@parameters**	**@return**
Constructor - Instantiates the class as well as converts the given What3Words String location to a GeoJson Point	String location	N/A
float getSensorReading - Returns the sensor reading	N/A	float reading
Feature convertToFeature - Converts the sensor into a GeoJson feature complete with extra properties that tell us more about the sensor	N/A	Feature sensor

NoFlyZone

This class models a NoFlyZone into an object. It extends *BaseEntity*.

Methods	**@parameters**	**@return**
Constructor - Instantiates the class in addition to	Feature noFlyZone	N/A

generating a list of paths defining the bounding region of the NoFlyZone		
getFeature - Returns the NoFlyZone feature	N/A	Feature noFlyZone
passesThroughNoFlyZone - Determines whether a path passes through a NoFlyZone by comparing if the path intersects any of the pathSegments of the NoFlyZone	Path path	boolean isValidPath

Path

This class models a Path Object. It extends *BaseEntity*.

Methods	**@parameters**	**@return**
Constructor - Instantiates the class as well as computes the length of the path and stores it as distance	Point from Point to	N/A
pathIntersectsAnotherPath - Determines whether the path intersects another path and returns a boolean	Path otherPath	boolean intersects
pathIntersectsCircle - Determines whether the path intersects a circle of the given radius and center	Point center double radius	boolean intersects
convertToFeature - Returns a GeoJson LineString object as a Feature	N/A	Feature path

DroneMove

This class inherits from the *Path* class. It is a custom path that contains additional information that is required by the drone for navigation.

Methods	**@parameters**	**@return**
Constructor - Instantiates the class by calling super(from, to) as well as calculates the angle of movement for the drone	Point from Point to String associatedSensor	N/A

toString - Returns debuggable printable version of the drone move	N/A	String droneMove
convertToFeature - Returns the droneMove as a GeoJson Feature	N/A	Feature droneMove
printToFile - Converts the droneMove into the specified output format ready to be logged	N/A	String droneMove

Drone

This models the drone object.

Methods	**@parameters**	**@return**
Constructor - Instantiates the drone object with its initial position - Initializes the SensorService with the date and port parameters and obtains the list of sensors to be Visited - Initializes the NoFlyZoneService with the port parameters and obtains the NoFlyZones - Initializes the FlightService with the obtained sensors, noFlyZones and its initial position in order to generate a valid flight plan	Point startingPoint String date String month String year String port	N/A
convertFlightPlanToFeatures - Converts the drone's flight plan into a series of features that can me mapped	N/A	List<Feature> features
getFlightPlan - Converts the flight plan into Strings		String flightPlan

Services

SensorService

This service is responsible for obtaining the sensors meant to be visited by the drone via a series of HTTP requests. At the moment, we model the service as a static class as all it does is retrieves sensors based on the date parameters. However, this can be easily extended for future reference to assign particular sensors to specific drones thus allowing researchers to use multiple drones to collect readings for a huge area.

Methods	**@parameters**	**@return**
static getDailySensors - Make HTTP requests to our web server in order retrieve the sensors meant to be visited by the drone	String date String month String year String port	List<Sensors> dailySensors

NoFlyZoneService

This service is responsible for making requests in order to get the noFlyZone regions

Methods	**@parameters**	**@return**
static getNoFlyZones - Make HTTP requests to get the noFlyZone regions	String port	List<NoFlyZone> noFlyZones

FlightService

This service is responsible for generation of the drone's flight plan which decides how the drone will move.

Methods	**@parameters**	**@return**
Constructor - Instantiates the service	List<Sensor> sensors List<NoFlyZone> noFlyZones Point currPosition	N/A
getFlightPathPlan - Instantiates an implementation of OrderSolver to get the list	N/A	List<DroneMove> droneMoves

in which the sensors are visited - Based on the visitation order obtain, it instantiates the flightPathPlanner to chart moves based on the drone Control Algorithm		

OrderSolver

This is an interface responsible for determining the order in which the sensors should be visited. I have two implementations of this interface:

1. ***CustomTSPAnnealingSolution.java*** - determines the optimal visitation order based on the Simulated Annealing method for TSP problems
2. ***CustomGoogleORToolsSolution.java*** - implements a third part solver (ORTools) to help determine the optimal visitation order. This was preferred over the first one.

The interface has only one method with the following signature:

List<Sensor> findBestVisitationOrder();

FlightPathPlanner

This is the class that charts the path taken from the drone's current position till it reaches its target sensor. It implements the drone control algorithm and returns the relevant drone moves.

Methods	**@parameters**	**@return**
Constructor - Instantiates the class and converts the What3Words sensor string position into a Point	Point droneCurrentPosition String sensorPositionString List<NoFlyZone> noFlyZones	N/A
planPathToSensor - Uses the drone control algorithm to chart moves to get the drone to the sensor and stores them	N/A	N/A

getPathPlan - Returns the list of drone moves to be taken to get from the droneCurrentPosition to the Sensor	N/A	List<DroneMove> droneMoves

HelperClasses

What3WordsConverter

This class is responsible for the conversion of the What3Words String locations to GeoJSon Points

Methods	**@parameters**	**@return**
static fromStringLocationToPoint - Converts What3Words String location to a GeoJson Point via HTTP requests	String location	Point point

GeometryHelper

This is a class used to help us perform various mathematical vector operations. It implements the following static methods.

Methods	**@parameters**	**@return**
static distanceBetweenTwoPoints - Finds the distance between 2 points	Point pointA Point pointB	double distance
static angleBetweenTwoPoints - Finds the angle between 2 points	Point pointA Point pointB	double angle
static pathsIntersect - Determines whether 2 paths intersect	Path path1 Path path2	boolean intersect
static pathIntersectsCircle - Determines whether the path intersects a circle of the given radius and center	Path path Point center double radius	boolean intersect
static getPointsOfIntersectionOfPathAndCircle - Finds the points of intersection where the path intersects a circle of the given radius and center	Path path Point center double radius	List<Point> points

`static findNewPoint`
- `Determines a point of termination given the angle of travel, starting point and the distance to be travelled`
	`Point from`
	`double angle`
	`Double maxDist`
		`Point end`

Drone Control Algorithm

There are imposed restrictions as to how the drone can move. These include the following:

1. The drone can only move in an angle of 10 degrees or its multiples
2. There is a maximum distance the drone can move without pause
3. The drone can only make up to a maximum number of moves due to its limited battery
4. The drone can not fly over specified regions described as NoFlyZones

Based on the above constraints, below is an algorithm to determine how we find the next move the drone will take to move towards a given sensor given the following variables: *droneCurrentPositon*, *sensorPosition*

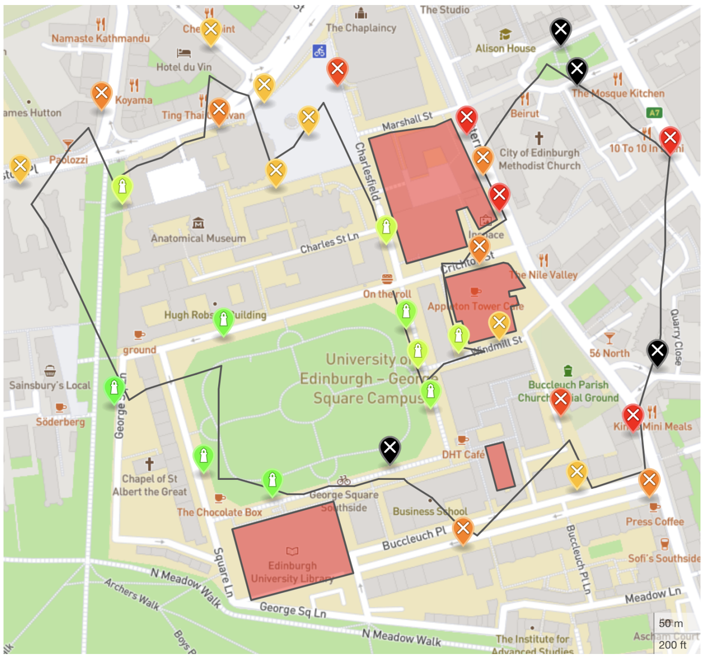
1. find the angle between *droneCurrentPositon* and *sensorPosition*.
2. convert the angle to a valid drone movement angle and generate a *tempPath* that depicts the drone movement in the given angle and distance equal to the drone maximum allowed distance
3. check if this path doesn't pass through any NoFlyZones. If it does, adjust the angle till a valid path is found.
4. determine the distance between the *droneCurrentPositon* and *sensorPosition*
    a. If the *droneCurrentPositon* is within the sensor discovery radius (distance < RADIUS)
        - generate a new drone move from the *droneCurrentPositon* to *sensorPosition*
        - update *droneCurrentPositon* and *sensorPosition* to the next sensor meant to be visited

   b. If the droneCurrentPosition is within one move of being able to get a reading of the sensor (RADIUS < distance < RADIUS+DRONE_MAX_DISTANCE)
      i. if *tempPath* is within sensor discovery range
         - find a point, *tempPoint* within range (where the path intersects the circle)
         - create a new path from the *droneCurrentPositon* to *tempPoint*
         - generate a new drone move based on the new path created
         - update *droneCurrentPositon* and *sensorPosition* to the next sensor meant to be visited
      ii. if *tempPath* is not within sensor discovery range based on the angle
         - readjust the angle until you find a valid path that is within the sensory discovery range
         - generate a new drone move based on the new valid path generated
         - update *droneCurrentPositon* and *sensorPosition* to the next sensor meant to be visited
   c. If the drone is far away from the sensor (ELSE)
         - generate a new drone move similar to *tempPath*
         - update *droneCurrentPositon*
5. If we haven't reached the sensor ie. *sensorPosition* is still the same, repeat the same process from number 1

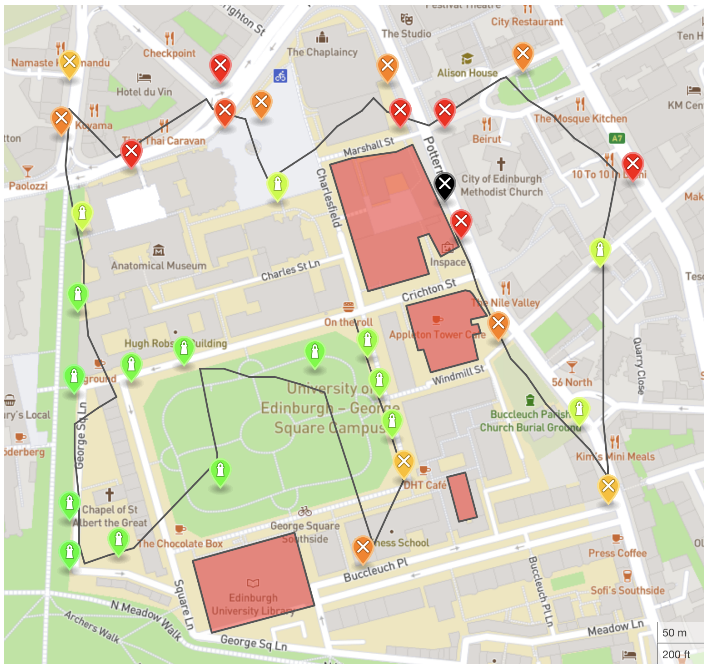
This algorithm maps out how to generate the path from the *droneCurrentPositon* to any given sensor. The sensor is decided based on the visitation order determined by the *OrderSolver.java* class. The *OrderSolver.java* implemented is derived from **ortools**, a third party library routing solver that employs the use of various TSP graph algorithms in order to generate an optimal route for our drone to follow. More about **ortools** at https://developers.google.com/optimization.

The relevant files related to the drone control algorithm include: *FlightPathPlanner.java*, *FlightService.java*, *OrderSolver.java*, *CustomGoogleORToolsTSPSolution.java*, *NoFlyZoneService.java*, *NoFlyZone.java* and *DroneMove.java*

Below are attached snippets of our algorithm in action.



FlightPlan rendering for ***readings-12-06-2021.txt***



FlightPlan rendering for ***readings-17-05-2021.txt***

References

- “OR-Tools | Google Developers.” *Google Developers*, 2019, developers.google.com/optimization.
- “The GeoJSON Format.” *Ietf.org*, 2016, tools.ietf.org/html/rfc7946.
- Ye, Gao, and Xue Rui. "An improved simulated annealing and genetic algorithm for TSP." *2013 5th IEEE International Conference on Broadband Network & Multimedia Technology*. IEEE, 2013.
- “Maven – Maven Documentation.” *Maven.apache.org*, maven.apache.org/guides/index.html