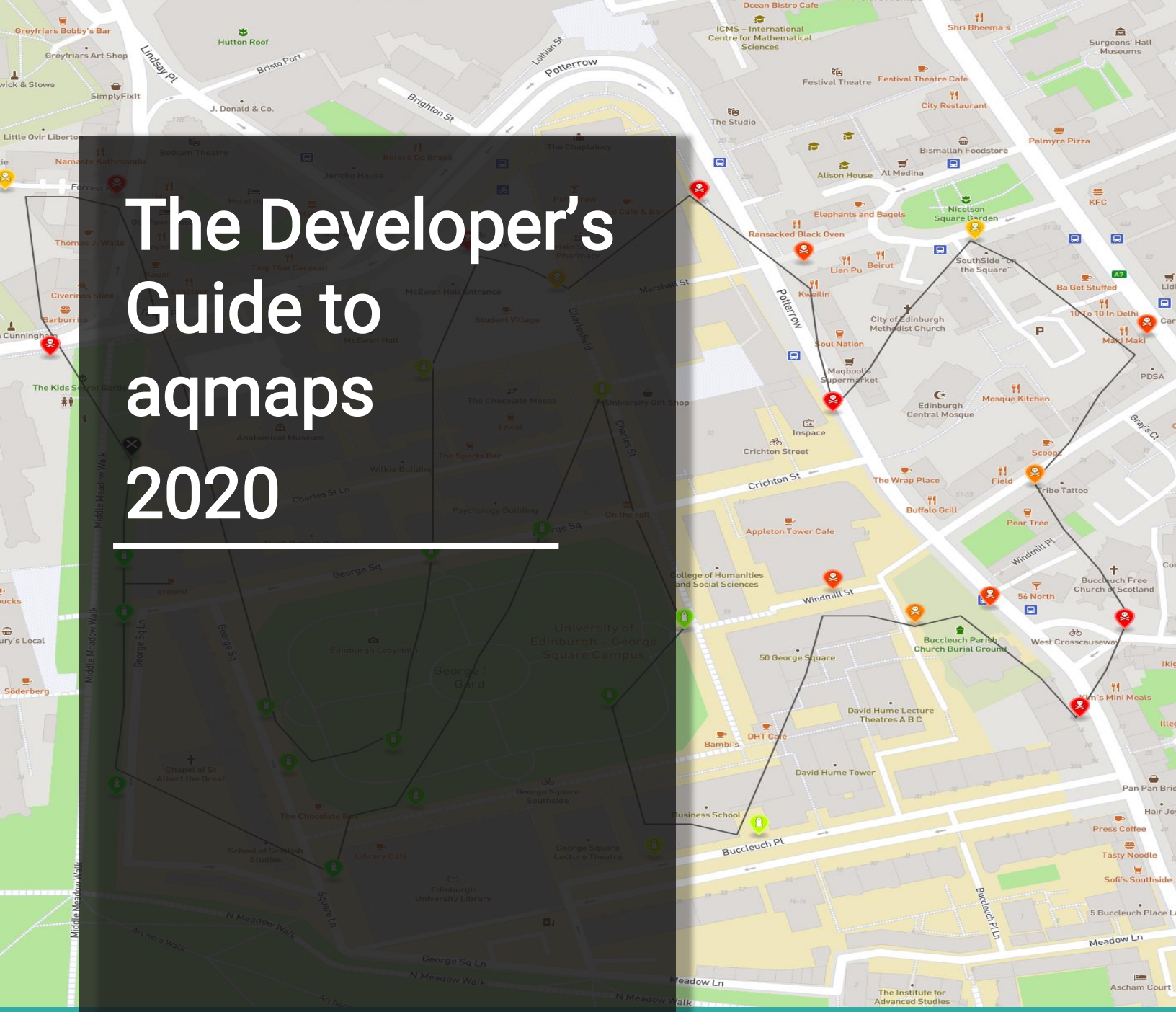

The Developer's Guide to aqmaps 2020

DECEMBER 4TH

Sean Strain; S1832137

Table of Contents:

Section One; Software Architecture Description. 3
- 1.1; Introduction 3
- 1.2; Identifying StakeHolders 3
- 1.3; Goals and Constraints 3
- 1.4; Creating a Faithful Representation 3
- 1.5; Bridging the Gap 4
- 1.6; Building Objects 4
- 1.7; The Resultant Classes 4
- 1.8; UML Class Diagram 5
- 1.9; UML Sequence Diagram 6

Section Two; Class Documentation 7
- 2.1; Introduction 7
- 2.2; App.java 7
- 2.3; Drone.java 8
- 2.4; FileHandler.java 9
- 2.5; NoFlyZone.java 10
- 2.6: SensorList.java 11

Section Three; Drone Control Algorithms 12
- 3.1; Introduction 12
- 3.2; Sensor-to-Sensor; The A* Algorithm 12
- 3.3; Overall; Nearest-Neighbour and Two-Opt 13
- 3.4; Example Results 14

Section One; Software Architecture Description:

*"**Why** does the package have these classes?"*

1.1; Introduction:

This section of the document provides a **high-level overview** of the overall class structure of the aqmaps package. It makes use of the **Unified Modelling Language** to convey the structure in an understandable way. It will define the **goals** and **constraints** that influenced the design.

1.2; Identifying Stakeholders:

The primary focus of any software engineer's design should be the people who are affected by their decisions. To determine how the architecture should be set up, first we have to determine those who will **use, maintain,** and **see** the results of said architecture:

1. The **Researchers** on the project. The primary purpose of the package is to supply the researchers with a visualization of the air pollution sensor readings in an area. They are the end users of the package.
2. The **Students** that will be given the package to maintain. The package is only a prototype, it will be given to a team of students who will configure and further edit the package structure and code base. They must be able to understand the structure first before they can support its further development.

1.3; Goals and Constraints:

1. As the package, at its core, is a prototype for the **Researchers**, it must create a **faithful representation** of what would really happen if the research project were real. This was a crucial part in deciding how the package would work at the fundamental level and was the first and foremost goal the overall class structure had to achieve.
2. The package must interact, connect, and retrieve information from a **Webserver** that simulates the Sensor readings. How the package interacts with this Webserver is a primary concern of the software architecture.
3. The package must deal with the third-party module **Mapbox GeoJson**. This dependency adds additional constraints and requirements to the overall structure.
4. The package is inherently designed for further **modification** and will be subject to significant changes over the course of its continued development. The package must, then, ensure the ease of **maintenance** and uphold a high level of **readability**. The package does this by **modularising** each problem area identified and making use of the **separation of concern**[1] design principle.

1.4; Creating a Faithful Representation:

The primary **use-case** of the project we are emulating can be succinctly condensed into one sentence:

*"A **drone** moves around a list of **sensors** while avoiding **no-fly-zones**"*

In order for the package to stay as true to this as possible, it too has a Drone class, a Sensor class, and a NoFlyZone class. The Drone class simulates the flight its real-life counterpart, it behaves in the **exact** way described in the sentence above; NoFlyZones each define an area the Drone **cannot** go. The decision was made

[1] https://rkay301.medium.com/programming-fundamentals-part-5-separation-of-concerns-software-architecture-f04a900a7c50

to **not** separate the Drone into a physical drone class and a pathfinder class, as in the real-life project, the Drone itself would be calculating its paths.
So, with these 3 classes we meet our first and foremost goal of emulating the real-life project. However, as we a Webserver instead of a receiver, we must find a way to reduce the impact that this difference has on our representation.

1.5; Bridging the Gap:

There are two classes that help bridge the gap between connecting to a Webserver instead of a Sensor. The first of which is the **FileHandler** which handles the transition between the packages internal information and file-system, it **alone** connects to the Webserver and writes files. Second of which is the **SensorList**, which contains a Mapping of every Sensor's WhatThreeWords location to the Sensor object in addition to paths created between them. It acts as a graph for the Drone's pathfinding to solve and facilitates the connection between Drone and Sensor. This decision was made to increase the **cohesion** of each class by reducing the **coupling** of the Drone and SensorList by not linking these classes to the FileHandler. The Drone **only** gains access to a Sensor's reading information if it is within range of it, as it would work in real-life.

1.6; Building Objects:

App.java handles the building of **all** the outer-class objects. It was decided to not use a specified builder/factory class as the problem the package solves has no need for interfacing, the objects of each distinct class are largely the **same**. It is more understandable to keep the small methods that create them in the same place for readability purposes.

1.7; The Resultant Classes:

With all of the above taken into consideration **seven** classes (including inner classes) were decided to be a part of the package. Each was chosen with a specific goal it and **only** it handles:

Goals:	Class:
To facilitate the package's inputs and set up everything for the Drone's flight.	App
To facilitate the translation between the internal package's information and the file-system.	FileHandler
To store a graph of Sensors for the Drone to solve. To allow for the 'connection' between the Drone and a Sensor.	SensorList
To represent the air-quality Sensors of the real-life project.	Sensor
To pathfind; between and over all Sensors.	Drone
To encapsulate all the information needed for the Drone to make a move.	Movement
To handle everything to do with where the Drone cannot go.	NoFlyZone

1.8; UML Class Diagram:

UML class diagrams describe a system's classes, their attributes and methods, and the interaction between objects. It is a useful tool to visualise how the package's classes relate to one another.[2]

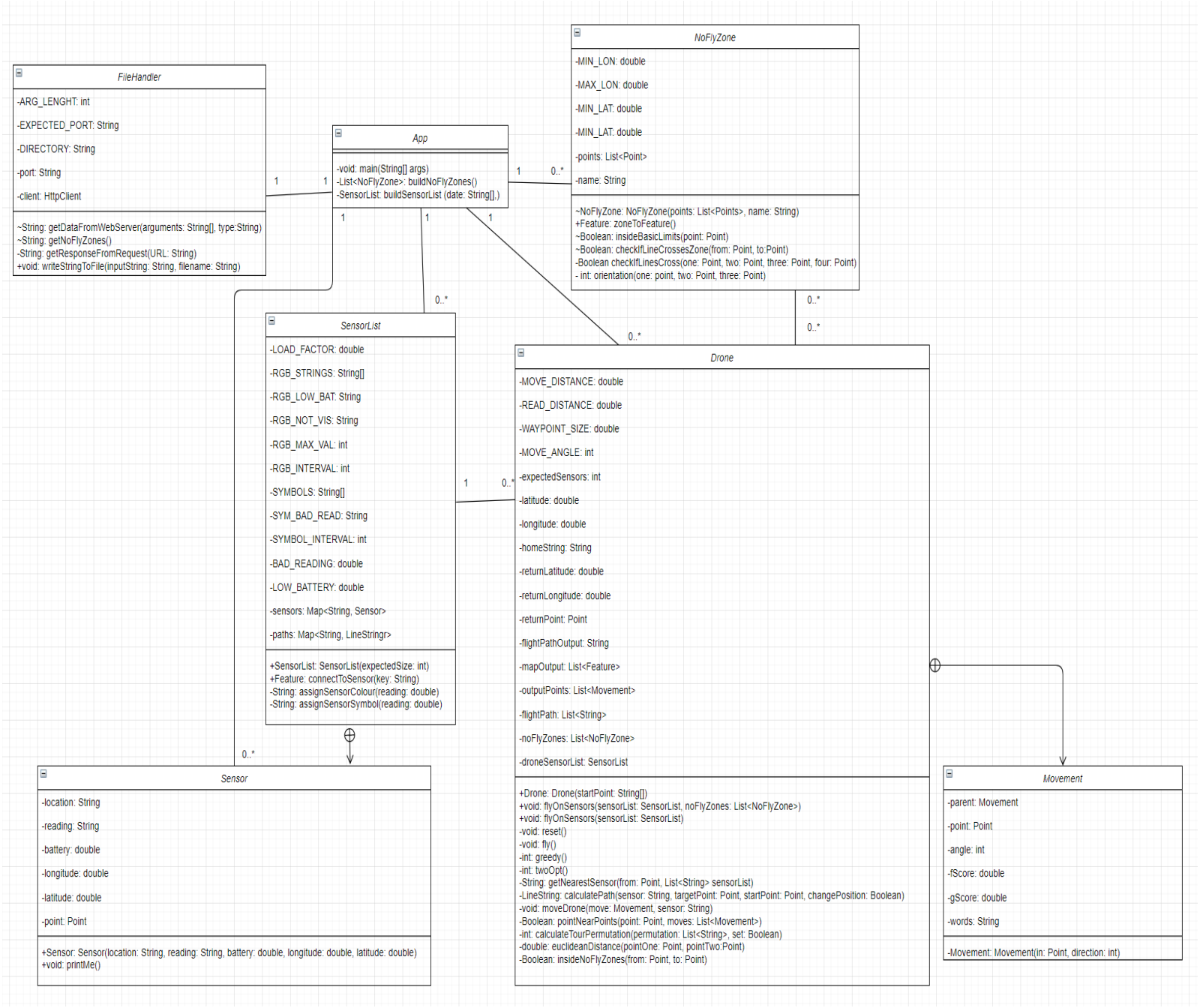

Figure One: A UML class diagram.

[2] https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/

1.9; UML Sequence Diagram:

UML sequence diagrams are used to visualise and clarify the interaction of each component on runtime.[3] The package only has **one** function: to output a flightpath and map of a given day. This primary function's generalised sequence diagram is displayed below:

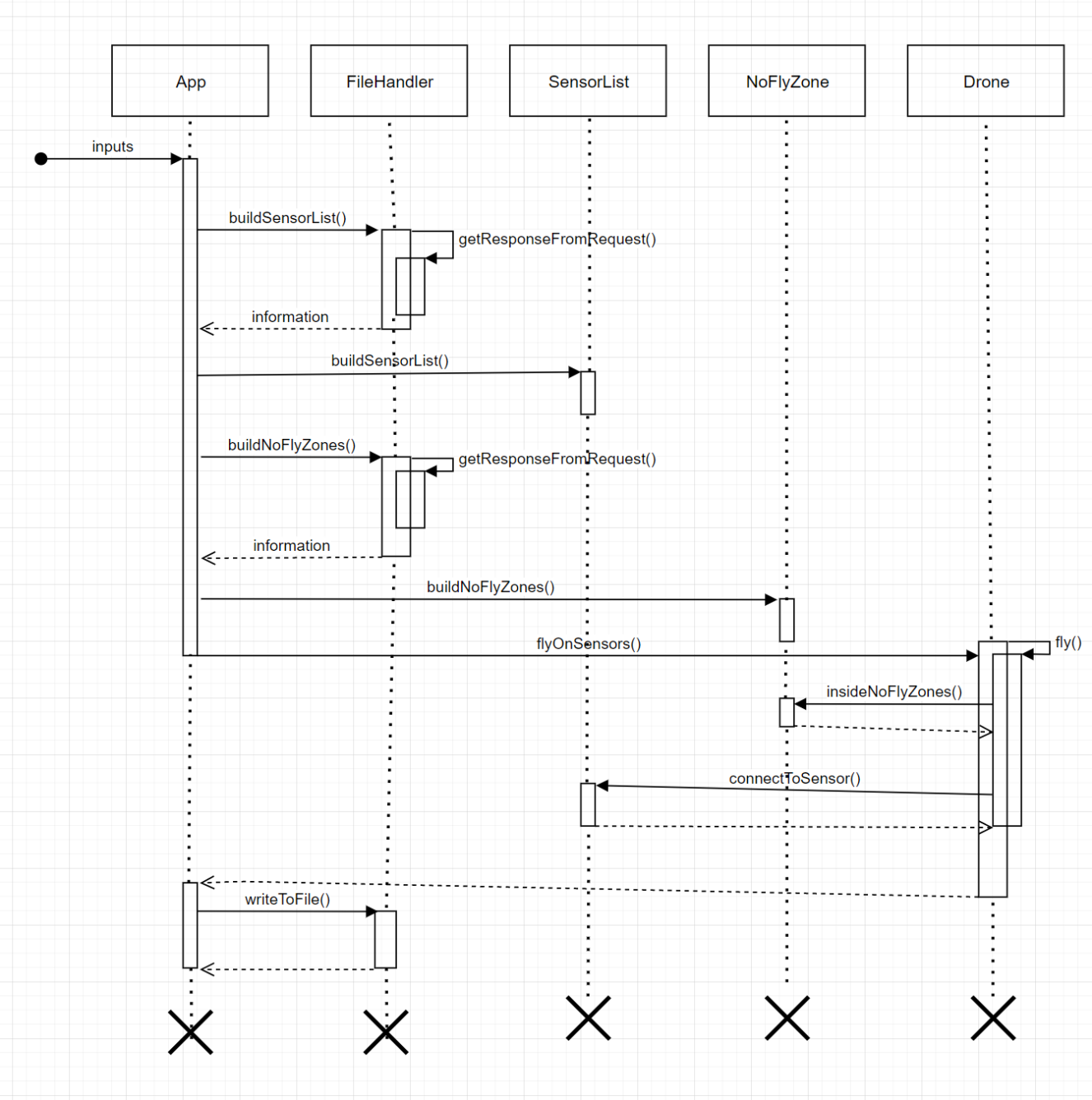

Figure Two: A UML sequence diagram.

[3] https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/

Section Two; Class Documentation:

*"**What** does each class do?"*

2.1; Introduction:

The following section breaks down the goals of each provided class in the package; it outlines the **goal** of that class and concise information on the workings of each method/function, in the approximate order of their execution. Javadoc has been used **extensively** for every method that is not trivial, such as getters, setters, and inherently simple, self-explanatory methods, for instance `addSensor()`. However, this section will go into **much** more extensive detail. Each class will have its own section and they are listed here alphabetically.

2.2; App.java:

*Goal: To **set-up**, **run** and **publish** the outputs of the package.*

The driver and the builder; App.java is a `static` class that contains the package's main function, but also contains the `static` **factory** functions for both the SensorList and NoFlyZones. It handles the creation of all the objects required to create the flight output. The flow of information through the entire package can be seen from `main()`.

`main( String[] args ):`

The main function of aqmaps expects six inputs, the first three are the date by which to read from in the form DD-MM-YYY which is **flipped** to YYYY-MM-DD inside the package as this is the ordering of the Webserver, so take care to keep this inversion in mind. The next two arguments are the longitude and latitude that the Drone class will set as its return point. Next comes a random seed number which can be used for seeded pathfinding algorithms, but it is currently **unused** by this package. Last, the port number to connect the FileHandler too. This argument is set by default to `"80"` within FileHandler but will be overwritten by the main function.

Main follows seven steps everytime the package is called:

1. Set the FileHandler's port.
2. Build a `new` Drone at the start location.
3. Call the `buildSensorList()` function.
4. Call the `buildNoFlyZones()` function.
5. Fly the Drone over the Sensors using `drone.flyOnSensors()`.
6. Read the Drone's outputs.
7. Write the outputs to file using `FileHandler.writeStringToFile()`.

`buildSensorList( String[] date ):`

This `static` function is the **only** place the SensorList and Sensor objects are made. **First**, it uses the FileHandler to get the Json formatted Sensor reading information for this `date`, which includes the WhatThreeWords of each Sensor we are required to access. **Then**, asks the FileHandler for each WhatThreeWords location file given for that `date` and takes the information of the Sensor stored within. It sets the Sensors location and adds it to the SensorList. Once every sensor for that day is stored in the SensorList, it returns the SensorList which will be later handed to the Drone class.

`buildNoFlyZones():`

This function is similar to the buildSensorList function. It reads from the `/buildings/no-fly-zones.geojson` file in the WebServer to get the Points of the no-fly-zone's boundary. It also gives the NoFlyZone its name, which is useful for debugging purposes. As the WebServer file is written in the Json format, the function will first turn this Json format into a FeatureCollection, then separate the collection into its constituent Features which each represent a NoFlyZone. The points of the zone's boundary are added to a list, which is then handed to the NoFlyZone constructor to make a new NoFlyZone. The function will return a list of all the NoFlyZones created by the function.

2.3; Drone.java

*Goal: To **path-find**, both Sensor-to-Sensor and across the SensorList, connecting to each one as it goes. To **emulate**, as closely as possible, the flight of a real-life Drone.*

The Drone class contains the **vast majority** of the package's code. It allows for a flight to be made over a SensorList, both with and without the NoFlyZone(s) through calling its public `flyOnSensors()` method, which is the starting point of the Drone class' information flow. The Drone class is designed for re-use; the most efficient use of the Drone class is to repeatedly call `flyOnSensors()` over each day the flightpath and map is wished for as it resets itself after each flight in preparation for a new one. The pathfinding methods `greedy()`, `twoOpt()`, `calculatePath()`, `pointNearPoints()` and `getNearestSensor()` are described in detail in Section Three they will not be explained here. The Drone class has one subclass, the Movement.

The Movement Subclass:

A Movement **encapsulates** all the information required to move the Drone one step. Each Movement **must** have a Point and an angle in the form of an `int` which refers to the degrees from East moved. Additionally, a Movement **may** have a String referring to a Sensor's WhatThreeWords location, a parent Movement holding the information of the Movement before it, and a f and g score which are used for the A* algorithm (see Section Three).

`flyOnSensors( SensorList sensorList, List<NoFlyZone> NoFlyZones )`

There are two `public` methods named `flyOnSensors()`, one allows the Drone to fly across the Sensors inside `sensorList` without no-fly-zones and the other with them. The method will overwrite the outputs of any previous flight by (re)initialising the class variables. Next, it will call the methods `fly()`, then `reset()`.

`fly():`

Calling the fly method is akin to pressing the 'go' button. The Drone will first calculate the best path it can using the algorithms it possesses using `greedy()` and `twoOpt()`. After this, it will execute two `for` loops:

- First, a permutation of the graph made by the Sensor's in the Sensor list will have been constructed and placed into the class variable `flightPath` by the above graph-solving methods, which is a list of WhatThreeWords locations to visit in order. **For each** of these locations, it will calculate a path using the `calculatePath()` method and this will in turn move the Drone and publish the Sensor to the `mapOutput` class variable. It will then add the Sensor visited to the `visited` class variable, repeating until everything is visited.

- Next, it will generate a flightPath String for this day out of the Movements in the `outputPoints` class variable by iterating over each Movement and publishing them to the class String variable `flightPathOutput`, which contains each Movement the Drone made.

Finally, the `mapOutput` is appended with a LineString which details each Point within the list of Movements made.

After this method is complete, the Drone's outputs are finished and able to be read from.

`reset():`

The reset function clears the SensorList and NoFlyZones and places the drone at its home Point in preparation for the next `flyOnSensors()` call.

`moveDrone( Movement move, String sensor ):`

This method is called every time a path is calculated between two sensors while the Drone is in flight, it moves the Drone to `move.point` publishes this Point to the output. It calls the SensorList using the key `sensor` and, if it is in ***`READ_DISTANCE`*** it will add the feature to the output. It stops the Drone by calling `stop()` if it exceeds ***`MAX_MOVES`***.

`stop():`

This method **stops** the drone from publishing **any** more moves, as it is only called when it has exceeded ***`MAX_MOVES`***. The Sensors in the SensorList that are not in `visited` to are published as *not visited*, seen in the image to the right.

Figure Three: An unvisited Sensor.

`calculateTourPermutation( List<String> permutation, Boolean set ):`

This method counts the lengths of the paths made between each Sensor listed in the `permutation`. If asked, by passing true to the `set` parameter, it will update the class variable `flightPath` to the given `permutation`. It makes use of the SensorList's path map to ensure no path is calculated twice.

`insideNoFlyZones( Point from, Point to ):`

This method will take each NoFlyZone object in the class variable `noFlyZones` and asks it to evaluate if the line-segment made by the two Point parameters violates its boundaries. See `NoFlyZone.checkIfLineCrossesZone()`.

2.4; FileHandler.java

*Goal: To facilitate the **translation** between the internal package's information and the file-system.*

The `static` class FileHandler is an example of a *util* class. It branches the gap between the file-space and WebServer and the package's internal information.

`writeStringToFile( String inputString , String filename ):`

This generic utility function writes the information provided in the first parameter `inputString` to a file named by the second parameter `filename`. It will overwrite any already created file with the same `filename`. It will output to the aqmaps folder.

`getNoFlyZones():`

This function will generate the URL of the file that contains the NoFlyZones and return the information within by calling the `getResponseFromRequest()` function and returning its results.

`getDataFromWebserver( String[] arguments, String type ):`

This function will generate a URL of the type specified in the `type` parameter (either `words` or `dates` but the options can be changed simply by adding `if` statements to the list) which correspond to the type of file the calling class wants to access. It will then send the URL, generated by `arguments`, to the `getResponseFromRequest()` function and return its results.

`getResponseFromRequest( String URL ):`

This function will go the URL parameter and return the String of information contained within the file there. If the file is not found, it will throw a 404 error and return null.

2.5; NoFlyZone.java

*Goal: To handle everything to do with where the Drone **cannot** be, including the limiting area.*

The NoFlyZone class includes the constructor for a NoFlyZone, but also the `static` function `insideBasicLimits()` which determines whether a point provided to it is within or without the basic rectangular limiting area of the map as defined by the four `static final` constants of the class which represent the four points of the rectangle.

`insideBasicLimits( Point point ):`

The `static` function `insideBasicLimits()` takes `point` and evaluates its latitude and longitude. As the limiting area is a simple box, it is sufficient to merely ensure the `point`'s longitude and latitude do not exceed the maxima nor subceeds the minima of the four boundary points. If the limiting area were to become a more complex polygon, one could make the `checkIfLineCrossesZone()` method `static` and use that to determine whether a line segment goes over the boundary.

`checkIfLineCrossesZone( Point from, Point to ):`

Given a line segment denoted by two points, this function will return whether or not that line segment violates the NoFlyZone's boundary by calling `checkIfLinesCross()` against every line in the NoFlyZone.

`checkIfLinesCross( Point one, Point two, Point three, Point four ):`

Evaluates the 2-D lines made between the first two parameters and the last two and determines whether they cross. This is done by checking the orientation of the triplet of points formed by checking each line against each point of the other line. If the orientation of the first triplet is different from the second and the third triplet is different from the fourth, these lines must intersect at some point. It determines the orientations by using `orientation()`.

`orientation( Point one, Point two, Point three ):`

To determine the orientation of a triplet, given in the methods parameters, we consider their gradients. If the gradient of the first to the second is greater than that of the second to the third, the points are clockwise. If the gradient of the first to the second is less than that of the second to the third, the points are counter-clockwise. If they are equal, the lines are collinear.[4]

[4] https://www.geeksforgeeks.org/orientation-3-ordered-points/

2.6; SensorList.java

*Goal: **Store** every Sensor and the paths between them. Allow for the Drone to **connect** to the Sensor's contained within.*

The SensorList class is essentially a **graph**, where the nodes are Sensors, and the edges are LineStrings. It contains two Maps which allow for the Drone to easily access the information of a Sensor or path by simply having the WhatThreeWords location(s). Furthermore, it simulates the connection between the Drone and Sensors through the `connectToSensor()` method. It has one subclass, the Sensor.

The Sensor Subclass:

> The Sensor class encapsulates all the information a Sensor has in the Webserver: a reading, a battery level, a WhatThreeWords location, and a longitude and latitude.

`connectToSensor( String key, Boolean visited )`:

> Allows the calling class to get the Feature defined by the Sensor at the given `key` parameter's values. If the `visited` `boolean` is set to `false`, it will return the Sensor feature as a grey marker. It uses the pair of methods below to do this:

`assignSensorColour( double reading )` and `assignSensorSymbol( double reading )`:

> These methods each do exactly what their names say, they will return the String of the colour or symbol respectively given a `reading` as defined by the class' `static final` constants related to each which are described in detail by the comments written adjacent to them in the .java file. They work by taking the **floor**, rounded to the nearest `int`, of the `reading` **divided** by the maximum value a reading can be **divided** the amount of Strings in the relevant String array, and selecting the String in the array at the index of that `int`.

Section Three; Drone Control Algorithms:

*"**How** does the Drone move?"*

3.1; Introduction:

There are, altogether, five methods the Drone class uses to determine its flightpath around the sensors. `calculatePath()`, `pointsNearPoints()`, `greedy()`, `getNearestSensor` and `twoOpt()`. It also makes use of the NoFlyZone class to ensure it does not violate the boundary of an area it is not permitted.
This section is split into two parts: The **sensor-to-sensor** drone control algorithm and the **overall** drone control algorithm.

Figure Four:
The Drone avoiding a no-fly-zone.

3.2; Sensor-to-Sensor; The A* Algorithm:

To evaluate the best path between two Sensors, the Drone makes use of its `calculatePath()` method, which accepts a String denoting the Sensor it is flying towards, a Point denoting where it starts and a Point denoting its target (additionally it accepts a Boolean informing whether or not the Drone is pre-calculating its path or actually flying).
It works by using a specialised version of the **A* algorithm:**[5]

1. First, a Movement is made at the start Point, this is the start of the path.
2. `while` the Drone is not within `READ_DISTANCE` of its target, it will check every possible Movement it can make (defined by the 360 / `MOVE_ANGLE` points on the circle with radius `MOVE_DISTANCE`),
3. If the line segment formed between the last Movement and this one violates a NoFlyZone or passes over the boundary, it will not be considered. The method describing how this step is done is within NoFlyZone.java on page 10.
4. Moreover, the class variable `WAYPOINT_SIZE` defines a box around each Movement, if the new Movement is within this box, it will not be considered. The Drone uses the `pointsNearPoints()` method for this.
5. Finally, the algorithm selects the Movement that **minimises** the **difference** between the distance from the **end** of the new Movement to the **target** and a third of the distance already **travelled**, plus half the number of moves **already** taken to reach this Movement. This new Movement is then added to the list of points the algorithm has visited and the Movement before it is defined as its parent. These ratios were carefully tested to reach an optimal average path length.

Once the Drone has found a Movement that takes it within the `READ_DISTANCE` of the Sensor, it will reconstruct the path by looking at the chain of parents stemming from that final Movement until there are no more. This is the final path the Drone will fly across.

A* has a high worst case space complexity as it computes multiple branching paths and stores each one, but it is fast and efficient, it often finds the best possible path between points, but not always.

The Drone considers a LineString from Sensor to Sensor, and the path built between does not consider the Sensor after the target. Because of this, we need a separate function for when the Drone is already in range. When this happens, the Drone will simply find the first valid path and return to the point before to read the Sensor. This is inefficient and the Two-opt algorithm will only consider it if there are no better alternatives.

[5] https://brilliant.org/wiki/a-star-search/

3.3; Overall; Nearest-Neighbour and Two-Opt:

To determine the best route around the SensorList graph, the Drone makes use of two pathfinding algorithms: **Nearest-Neighbour** and **Two-Opt**. We will use the term **Sensor** and **node** interchangeably here. A **permutation** refers to a unique sequence of nodes given in order. These methods are defined below:

Figure Five:
A "greedy" path. Notice there is a lot of overlapping paths. (08/04/2020)

Nearest-Neighbour:

The Nearest-Neighbour algorithm, contained within `greedy()`, creates a permutation of the graph's nodes by simply adding the **nearest** (as returned by `getNearestSensor()`) node in terms of the **Euclidean Distance** to the one before, starting at the home-location of the Drone, until no more nodes are left. It is often inefficient as its greedy nature may miss more optimal moves; it may sometimes give a solution that is entirely inadequate. However, it is a good **starting point** for more informed algorithms. It has a worst-case time complexity of $O(n^2)$[6].

Figure Six:
The same path as Figure Five, with Two-Opt enabled.

Two-Opt:

The Two-Opt algorithm, contained within `twoOpt()`, recursively **swaps** the order of the nodes in 'sub-paths' of the permutation, and evaluates whether an improvement has been made in the overall solution. It will continue to do this until it **cannot** find an improvement. The motivation behind this method is to limit the amount of times an overall path crosses over itself, which Nearest-Neighbour will do **very** often. Reversing a list is $O(n)$ and the two nested for loops the algorithm contains amount to $O(n^2)$, so the worst case time complexity is approximately $O(n^3)$[7].

Combined:

The resulting overall algorithm combines the raw speed of Nearest-Neighbour and the careful consideration of Two-Opt to consistently generate a path that is **neat** and **short**; It will always return the same result. These two algorithms were chosen because the size of the graph is relatively small at only 33 nodes, but not small enough to employ an exact algorithm. If the graph size were to increase, Two-Opt will quickly become **inadequate** as it will require a large amount of computation.

[6] https://link.springer.com/chapter/10.1007%2F3-540-10003-2_92
[7] https://link.springer.com/article/10.1007/s00453-013-9801-4

3.4; Example Results:

Figure Seven:
01/02/2020

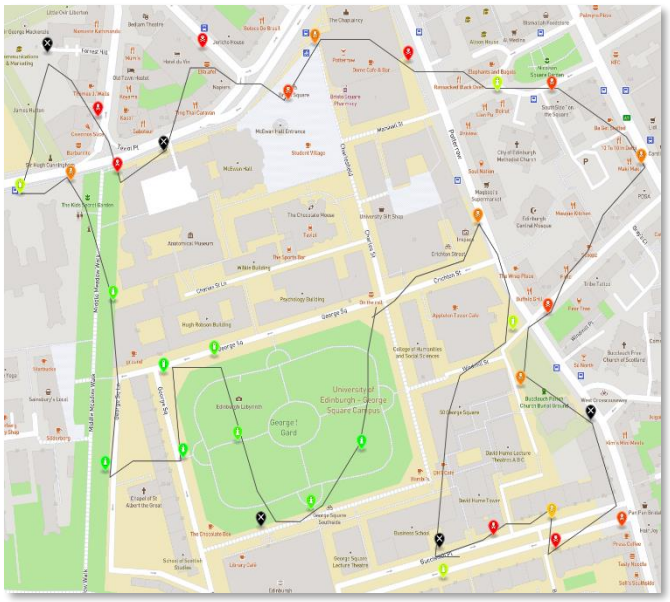

Figure Eight:
04/04/2020

Figure Nine:
04/04/2021

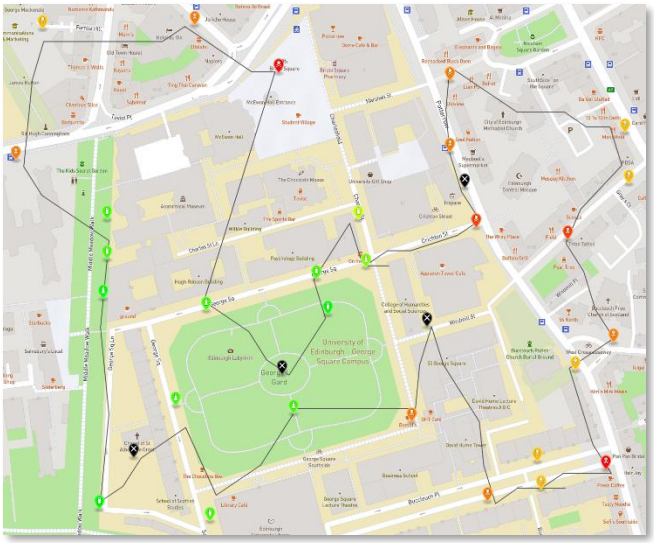

Figure Ten:
31/12/2021