Informatics Large Practical Report

Lewis Hamilton (s1807298)

Contents:

-page 2: Software Architecture Description

-page 4: Class Documentation

-page : Drone Control Algorithm

-page :References/Appendix

*Software Architecture Description*

My implementation contains a number of classes, each of which I see as valuable to the overall project.

*DroneFlightPath/FlightPathOutput*

To start with I’d describe what I believe to be the centre of the implementation in terms of classes. While there are no hierarchical structures in my final implementation these classes serve as the “brain” of it. Rather than have model a class based on the Drone itself, where data, such as its current location, would be constantly changing, I viewed it based on an *input/output* idea. In this case the input is data from the WebServer, such as sensors, noflyzones etc and the output, for each time the application, is run would be a flightpath for the drone. To me this makes it much clearer what is being calculated and where it is being calculated each time the program is run.

One could also view the DroneFlightPath class as an example of the *Factory Pattern*. It is only instantiated once during the program and creates a number of Move Objects that the end user does not know about as it is only sees the output from FlightOutput that the Move objects contribute towards.

The DroneFlightPath class is where the bulk of the calculations for finding the Flightpath take place, with data from other classes such as TodaysSensors, NoFlyZones being the *input* and the overall *output* of the DroneFlightPath class being an ArrayList of type Move (Move as a class is discussed later).

The decision to split the part of the program that calculates the output files is for 2 reasons. The first is to reduce the size/scope of DroneFlightPath class to make it easier to read, understand and maintain. The second is so that future users of the program can easily update the formats in how the output is generated. For example, say a user wants to have the flightpath output as a pdf format then it would simply be the case of writing an additional method in the FlightOutput class to parse the data given into said class into this format. This allows for having unique methods for each type of format of output making it easy for end users to pick how they want the data outputted as- they need only create another method in this class.

*SensorsToday/Sensor/Words*

Starting with Sensor, as a class it is simply the data structure which represents the real-life object of the sensors. Unlike say the Drone, to which data surrounding it would be constantly changing, the Sensor’s location, battery reading etc do not change over the course of calculating the flightpath. Hence modelling the Sensor as a class fits the problem.

The purpose of SensorsToday is to create a batch of sensors that represent the sensors the drone is travelling to and their battery/readings for that particular day. Each time this class is called it parses the JSON file from the WebServer for that particular day (SensorsToday’s Input) and gets an ArrayList of the Sensors with their battery/readings for that day (SensorsToday’s Output). This is a cleaner implementation than say creating every possible Sensor on the map. This is because you are cutting down on any unnecessary data such as redundant sensors that won’t be visited that day or calculations within the program to get the sensors to be visited for that day. Essentially SensorsToday *decouples the dependancy Sensor would have on the DroneFlightPath* class by creating an intermediary class that filters the relevant Sensor data to DroneFlightPath. Like DroneFlightPath it can also be considered a part of the Factory Pattern with SensorsToday also only being instantiated once during the program to create/parse a number of Sensor objects.

The Words class exists as a class in order to facilitate easy JSON parsing of the different fields of Sensors in the maps JSON Files. The Words class mimics the data structure of the each Sensor in the maps files with the static inner class coordinates acting as the field within a field for the coordinates.

*NoFlyZone*

The NoFlyZone class is an example of allowing me separate out data from the DroneFlightPath class to allow me to reduce the scope of DroneFlightPath. I could have easily just had the NoFlyZones as an ArrayList in DroneFlightPath and have all the GeoJSON parsing in that class but this separation allows for better readability overall. This class now contains all the data and calculations to check if the drone went through a NoFlyZone and DroneFlightPath only needs to instantiate it and call one of its methods throughout the whole application.

*Move*

The purpose of the Move class is to represent as a kind of data structure, each move the Drone makes across the map. It allows me to keep all the data of current/previous Longitudes,Latitudes, if the Drone visited a sensor and if so, what sensor in one place. This is useful as since a List of Moves is essentially the output of DroneFlightPath class, it means the FlightPathOutput class only needs to read in the information from each move to create the final output files.

*Direction*

The Direction class is another example of a class being used as a structured way to store data and also to ensure a clean way of input validation. This class is set up so that it can only store the direction in degrees in increments of ten so whenever a new direction for the Drone to go in is calculated within the DroneFlightPath class, any input validation is moved to this specific class.

*StraightLine*

The purpose of this class is to represent the calculate the straight line equation of the line a drone takes from one location to another. Since only the current and previous longitudes and latitudes are stored/calculated with in the Move and DroneFlightPath classes, this class allows us to calculate any other longitude and latitude on that line. This is useful when calculating whether or not the Drone has gone over the NoFlyZone in one move as this class lets us calculate multiple points along the line of travel and feed that information into the NoFlyZone class to ensure none of these points are in a NoFlyZone

*MapDate*

The function of the MapDate class might seem redundant given their are other Java classes such as Calendar that do a similar purpose. However I believe such classes have too much ceremonial processes in trying to store the date data and in the case of Calendar seems to serve a purpose of getting the current data/time and calculating times. My class simply stores the user inputted data in a structured way with relevant getters to get the date information when it is needed in other classes.