INFORMATICS LARGE PRACTICAL 2019-20

COURSEWORK 2 REPORT

LIANA AHMED (S1703935)

Software Architecture Description

When deciding on what classes to split the program into, I chose to follow the Single Responsibility Principle[1] to improve the readability and maintainability of my code. This means that the classes I have chosen have little to no overlap between them, as each class deals with a separate aspect of the software.

1 App

App is the class which contains our implementation of the main method. The primary function of this class is to take in the user input arguments, and invoke the necessary methods required to run the specified drone. This class is also used to ensure that the correct arguments are inputted by the user, and that the correct exceptions are thrown should they not.

2 Position

The purpose of the *Position* class is to keep track of/manage our drone's coordinates, as well as any hold any other relevant functions such as calculating the next set of coordinates after a move.

3 Direction

Direction is an enum which holds all 16 compass directions our drone can move in, as well as their associated angles.

4 Battery

Battery is a simple class used to manage and keep track of our drone's current charge.

5 Coins

Coins is a simple class used to manage and keep track of our drone's current coin balance.

6 Map

The *Map* class is used for anything related to the Geo-JSON used for each map. This includes dealing with the necessary server-side connections we need to make in order to obtain our Geo-JSON map; as well as obtaining the features (and their associated objects) of that map; and the function for writing the final Geo-JSON file after our drone's flight is complete.

While *Map* could have potentially been split into two separate classes (one specifically for dealing with the server-side connections, and another for the Geo-JSON); I personally did not feel this was necessary as both of these are closely related enough to not justify it.

[1] https://en.wikipedia.org/wiki/Single_responsibility_principle

7 RandomDirectionGenerator

RandomDirectionGenerator is a class whose sole purpose is to create a new Random Object based on the seed number inputted by the user, and get the associated random value from our *Direction* enum.

While this could have simply been implemented inside of the Stateless drone implementation, I felt it was necessary to create a new class in order to save the current Random Object. As using a seed value means the random values are given in a particular order, I felt it was important to protect this Object as its own private variable so that there is no way to accidentally reset it.

8 Drone

Drone is the parent class for our two drone types (Stateless and Stateful). It is there to declare and initialise all the variables and methods which are used by **both** drones.

Implementing the drones in this way means that we are able to avoid repeating a lot of code. Furthermore, if it is decided to add another drone type further along in the development process, it can simply be added as another child class.

8.1 Stateless

The class *Stateless* is a child class of *Drone*. It contains the entire implementation of the Stateless drone's methods; including how it moves and how it picks which direction to move in.

8.2 Stateful

The class *Stateful* is a child class of *Drone*. It contains the entire implementation of the Stateful drone's methods; including how it moves, how it picks which direction to move in, and how it decides on which feature to visit next.

Class Documentation

1 App

1.1 Main

This is the application's main entry point. It takes in the 7 required arguments from the user input, and invokes either the Stateless or Stateful drones as required. It then outputs to the console what the total number of moves, coins and battery are.

1.1.1 Parameters (User Input):

- **String day:** The day from the date of our desired map
- **String month:** The month from the date of our desired map
- **String year:** The year from the date of our desired map
- **double startingLat:** The initial latitude of the drone
- **double startingLong):** The initial longitude of the drone
- **int seedNum:** The chosen seed number for the Stateless drone
- **String droneType:** Either "stateful" or "stateless"

1.1.2 Exceptions thrown:

- **IllegalArgumentException:**
  - If incorrect number of arguments are inputted by user
  - If incorrect arguments are inputted by user

1.1.3 Output

- **To Console:** Total number of moves, coins and battery after drone's flight

1.2 Methods

createMapString (**String day**, **String month**, **String year**)
This method is used to create the String for the URL of our desired Geo-JSON map.

Parameters:

- **String day**: The day from the date of our desired map
- **String month**: The month from the date of our desired map
- **String year**: The year from the date of our desired map

Output:

- **Return:** The URL created by our function

createFileString (**String day**, **String month**, **String year**, **String droneType**)
This method is used to create the String for the file name used for the text and Geo-JSON files outputted at the end of the application.

Parameters:

- **String day**: The day from the date of our desired map
- **String month**: The month from the date of our desired map
- **String year**: The year from the date of our desired map
- **String droneType**: The type of drone we are using

Output:

- **Return:** The URL created by our function

2 Position

2.1 Global Variables

- public double latitude
- public double longitude

2.2 Constructors

- **public Position** (**double latitude**, **double longitude**)
  - Initialises a new position with the specified longitude and latitude

2.3 Getters

Getter	Parameters	Output
getDist: Finds the Euclidean distance between the current latitude and longitude of of our Position object and given longitude and latitude.	List<Double> **nextCoords**: The second set of coordinates for finding our distance. This is of the form of a List<Double> with the longitude being the first value and the latitude being the second value	The Euclidean distance as a double.
getCoordinates: Retrieves the current longitude and latitude of Position in the form of a list.	**N/A**	The current coordinates of **Position**. This is of the form of a **List**<**Double**> with the longitude being the first value and the latitude being the second value.

2.3 Other Methods

nextPosition(**Direction direction**)
This method is used to determine what the next latitude and longitude of the drone would be given the current position and direction the drone will be moving.

Parameters:
- **Direction direction**: The specific compass direction the drone will be moving in

Output:
- **Return:** A new Position object initialised with the new coordinates

inRange(**double dist**)
Tells us whether or not the given distance is less than or equal to 0.00025.

Parameters:
- **double dist**: The distance we are checking

Output:
- **Return:** A boolean value indicating if the distance is in range or not

inPlayArea()
Checks if the current coordinates are inside the designated play area.

Output:
- **Return:** A boolean value indicating if the coordinates are inside the play area

3 Direction (enum)

3.1 Values

All 16 compass directions our drone can move in and their related angles.

3.2 Global Variables

- Private final double angle

3.3 Constructors

- **private Direction** (**double angle**)
    - Initialises each direction in the enum with their respective angles.

3.4 Getters

Getter	**Output**
getAngle: Returns the angle of a chosen direction	The double value of the angle of the direction.

4 Map

4.1 Global Variables

- private URL mapURL
- private HttpURLConnection conn
- private String mapSource
- public List<Feature> features
- public List<Feature> goodFeatures
- public List<Feature> badFeatures

4.2 Constructors

- **public Map** (**String mapString**)
    - Connects to the URL described in mapString
    - Retrieves and organises the features from the respective Geo-JSON.

4.3 Getters

Getter	**Parameters**	**Output**
getCoordinates: Retrieves the coordinates of a given charging station.	**Feature f**: The charging station would like to retrieve this data from	The coordinates in the form of a **List<Double>** with the longitude being the first value and the latitude being the second value.
getCoins: Retrieves the value of coins of a given charging station	**Feature f**: The charging station would like to retrieve this data from	The value of coins in the form of a **Float**

getPower: Retrieves the value of the charge of a given charging station	**Feature f**: The charging station would like to retrieve this data from	The charging value in the form of a Float
getMarkerSymbol: Retrieves the marker symbol of a given charging station	**Feature f**: The charging station would like to retrieve this data from	The marker symbol in the form of a String
getMarkerColour: Retrieves the marker colour of a given charging station	**Feature f**: The charging station would like to retrieve this data from	The marker colour in the form of a String
getID: Retrieves the ID of a given charging station	**Feature f**: The charging station would like to retrieve this data from	The ID in the form of a String

4.4 Other Methods

mapConn(**String mapString**)
Private helper function which initialises a new URL with mapString before creating and opening a new HttpURLConnection with it.

Parameters:
- **String mapString**: The String version of the desired URL

Output:
- **Return:** A new HttpURLConnection of the URL from mapString

Exceptions Thrown:
- **MalformedURLException**
  - If mapString is not a valid URL
- **IOException**
  - If we cannot connect to the server

streamToString(**InputStream is**)
Private helper function which converts an InputStream object into a String

Parameters:
- **InputStream is**: The InputStream we would like to convert into a String

Output:
- **Return**: The new String version of the InputStream we just created

Exceptions Thrown:
- **IOException**
  - If we cannot connect to the server

writeFlightPath(List<Position> **flightPath**, **String fileName**)
Outputs a new Geo-JSON file of the map and drone's final flight path.[2]

Parameters:
- List<Position> **flightPath**: A list containing every Position the drone has had
- **String fileName**: What we would like to name our new Geo-JSON file

Output:
- **File Output**: The new Geo-JSON file

Exceptions Thrown:
- **ParseException**
  - If an array of Features from the original source is not found
- **FileNotFoundException**
  - If the file being written to is not found

5 Battery

5.1 Global Variables

- private double charge (Initialised with 250.0)

5.2 Getters

Getter	Output
getCharge: Returns the current charge	The current value in the variable charge

5.3 Other Methods

chargeBattery(**double amount**)
Increases the battery with the specified amount

Parameters:
- **double amount**: The amount we want to increase our battery by

consumeBattery(**double amount**)
Decreases the battery with the specified amount

Parameters:
- **double amount**: The amount we want to decrease our battery by

[2] Follows a similar method as found in https://stackoverflow.com/questions/37986806/create-geojson-using-simple-json-java

6 Coins

6.1 Global Variables

- private double collected (Initialised with 0.0)

6.2 Getters

Getter	Output
getCoins: Returns the current number of coins collected	The current value in the variable collected

6.3 Other Methods

addCoins(**double amount**)
Increases the number of coins collected with the specified amount

Parameters:

- **double amount**: The amount we want to add to our total coins

removeCoins(**double amount**)
Decreases the number of coins collected with the specified amount

Parameters:

- **double amount**: The amount we want to remove from our total coins

7 RandomDirectionGenerator

7.1 Global Variables

- private Random rnd

7.2 Constructors

- **public RandomDirectionGenerator** (**int seedNumber**)
  - Initialises the global variable rnd with a new Random object with the seed

7.3 Getters

Getter	Output
getRandomDirection: Retrieves a random direction from our Direction enum	A random Direction

8 Drone (Parent)

8.1 Global Variables

- public int moveCount
- public double latitude
- public double longitude
- public String mapString
- public String fileName
- public PrintWriter txtWriter
- public Position position
- public Battery battery
- public Coins coins
- public Map map
- public Direction[ ] directions
- public List<Position> flightPath

8.2 Constructors

- **public Drone** (**String mapString**, **double latitude**, **double longitude**, **Position position**, **String fileName**)
  - Initialises a new drone by initialising the respective global variables with the given parameters
  - Initialises **map** with a new **Map** object using **mapString**
  - Initialises **txtWriter** with a new **PrintWriter** for writing the end text file required.
  - Throws a **FileNotFoundException** should the file specified for **txtWriter** not exist

8.3 Getters

Getter	Output
getCount: Retrieves the number of moves completed	The current value in **moveCount**

9 Stateless (Child)

9.1 Global Variables

- public int seedNum
- public RandomDirectionGenerator rdg
- public List<Position> flightPath

9.2 Constructors

- **public Stateless** (**String mapString**, **double latitude**, **double longitude**, **int seedNum**, **Position position**, **String fileName**)
  - Invokes the super constructor with the relevant parameters
  - Initialises **rdg** with a new **RandomDirectionGenerator** using the given seed number **seedNum**

9.3 Methods

Move( )
Invoking this method will trigger the stateless drone to move around the map visiting stations, until either all 250 moves have been completed or the battery runs out. Each move/new position is documented along the way; and the current status of the drone for each move is written onto a text file with the given **fileName**. The stateless drone is the 'easy drone', as it can only make decisions based off of what it can immediately see around it (i.e. where it will be for each of the 16 directions it can move in). It avoids the bad charging stations, attempting to prioritise only visiting good charging stations with the highest utility[3].

Output:

- **File Output:**
  - The final text file which stores the Stateless drone's status during each move.
  - Triggers the method **writeFilePath** from **Map** to be invoked so that the final Geo-JSON file can also be outputted.

bestDirection(**HashMap**<**Direction**, **Feature**> **moves**, **List**<**Direction**> **illegalDirections**)
This method is a private helper function which takes in a HashMap of every valid direction a drone can go (alongside whatever good charging stations may be visited should the drone choose to visit in that direction), and a list of directions the drone cannot go, and return the best direction out of those available for the drone to visit (as well as the feature associated with that direction should there be one). In order to do this, **bestDirection** either picks whichever direction leads to the charging station with the highest utility or picks a random direction (being careful to ensure said direction is not a part of **illegalDirections**) should there be no good charging stations for the drone to visit.

Parameters:

- **HashMap**<**Direction**, **Feature**> **moves**: A HashMap of each valid direction the drone can take and their associated good feature (should said direction lead to any)
- **List**<**Direction**> **illegalDirections**: A list of every direction the drone cannot go (including directions which will lead it outside of the designated play area and directions which lead to bad charging stations).

[3] Utility in this context refers to the coin/battery gain of a charging station. If the Stateless drone is caught between choosing one charging station or another, can choose to pick based on what the drone at the time will benefit from more. For example, if the battery is low the drone will choose the station which has a higher battery gain. Otherwise, it will always opt to go for the station with the larger coin gain as that is the utility we are trying to maximise for most of the game.

Output:

- **Return:** A new **Object**[ ] where the first value is the best Direction found, and the second value is the (optional) feature associated with it (the second value can be **null**)

10 Stateful (Child)

10.1 Global Variables

- public List<List<Double>> flightCoordinates

10.2 Constructors

- **public Stateful** (**String mapString**, **double latitude**, **double longitude**, **Position position**, **String fileName**)
  - Invokes the super constructor with the given parameters

10.3 Methods

Move( )
Invoking this method will trigger the stateful drone to move around the map visiting stations, until either all 250 moves have been completed, the battery runs out or every good station has been visited. Each move/new position is documented along the way; and the current status of the drone for each move is written onto a text file with the given **fileName**. The stateful drone is the 'hard drone', which aims to visit every good station and avoid every bad station (except for the cases where it cannot).

Output:

- **File Output:**
  - The final text file which stores the Stateful drone's status during each move.
  - Triggers the method **writeFilePath** from **Map** to be invoked so that the final Geo-JSON file can also be outputted.

endMove(**Direction closestDirection**)
This method is a Private helper function which does everything required after the direction which drone will be moving in has been decided. This includes:

- Writing the status of the move in the output text file
- Updating position
- Consuming the battery by 1.25
- Iterating the moves and updating the flight path

Parameters:

- **Direction closestDirection**: The direction which was chosen for the drone to move in

Output:

- **OutputFile: txtWriter** from the parent class gets updated

getClosestFeature(**List<Feature> currentFeatures**)

This method is a **Private helper function** which figures out which is the next closest good charging station the drone can visit.

Parameters:

- List<Feature> currentFeatures: A list of charging stations the drone has not visited yet

Output:

- **Return**: The feature closest to our drone's current position

Stateful Drone Strategy

1 Overall Idea

My drone begins by scanning all the available good charging stations, and return whichever one was closest to its base coordinates (calculated using the Euclidean distance). If there are more than one charging stations of equal distance, the one which provides the highest number of coins would then be picked.

For each move, the drone would first scan how close each direction would take them to the closest charging station we had picked; while simultaneously filtering out any 'illegal' directions (i.e. directions that would lead us to a position outside of our play area, to a bad charging station, or to a set of coordinates the drone had previously had). It would then pick the direction out of the remaining valid directions which is closest to our target charging station. If the drone has no valid directions available, it then picks the first available direction which would make it pass through a bad charging station.

If this move takes us to the charging station itself, we remove that particular feature from our list of good features, update the coins, battery and end the move as normal; before repeating the process again with the next closest good charging station. Otherwise, we repeat the previous process of picking whichever move is closest to our target station. If our drone has attempted to move towards the same charging station for over 40 moves, it is deemed that said charging station is likely unreachable. It is then removed from our list of good charging stations, and the drone picks a new good station to move towards instead.

The above process is repeated until either our drone has run out run out of battery, made 250 moves, or it has visited every good charging station that is available.

2 Stateless vs Stateful

While there are several large ways in which the Stateful drone differs from the Stateless drone in terms of memory and implementation; they both have access to the following basic details:

- The current position of the drone
- The current battery percentage of the drone

- The number of coins the drone currently has
- The number of moves it has made so far

My implementation of the **Stateless drone** only has direct access to the following details for each move (these are all reset after each move is complete):

- What lies ahead in each of the 16 following directions from the current position of the drone (this includes both good and bad features, as well as whether or not it leaves the play area).

  This data is then split into the following:

  - A HashMap of valid directions the drone can move in, alongside the respective charging stations those directions may move to
  - A list of every 'bad' direction the drone should avoid if it has to pick a random direction (should there not be any charging stations it can immediately visit)

As the Stateless drone has a rather narrow field of vision, it can only keep track of what is immediately in front of its current position. While it does have some ability to choose the way it moves (if it is immediately surrounded by more than one good charging station, it can choose towards whichever one has a higher utility), there is still a random element involved when deciding how it moves, making it highly inefficient in terms of both how it collects coins and how many moves it takes to reach a feature.

My implementation of the **Stateful drone** has direct access to these following details:

- The coordinates from each of its previous positions (thus whether or not the current position matches one that the drone has previously been in)

- A list of all the unvisited good charging stations available on the map (and thus the ability to search for individual stations and remove them as necessary)
-
- The next available closest (good) charging station

- What lies ahead in each of the 16 following directions from the current position of the drone (this includes both good and bad features, whether or not it leaves the play area, or whether it is a direction which will lead the drone to repeat a previous position).

  This data is then split into the following:

  - A HashMap of valid directions the drone can move in, alongside the euclidean distance between the respective new position and the current feature we are trying
  - A list of every 'bad' direction the drone should avoid when picking the closest direction to move in

What really improves the performance with my Stateful drone, is its ability to scan the entire map and move towards the closest feature available each time. As it also avoids repeating positions, it is impossible for this drone to visit the same station multiple times or get stuck in loop. This also makes the drone more efficient in terms of moves, as it picks a path which minimises the number of moves it needs to make in order to visit each feature.

This can be demonstrated clearly with the following figures from the 11/11/2019 map:

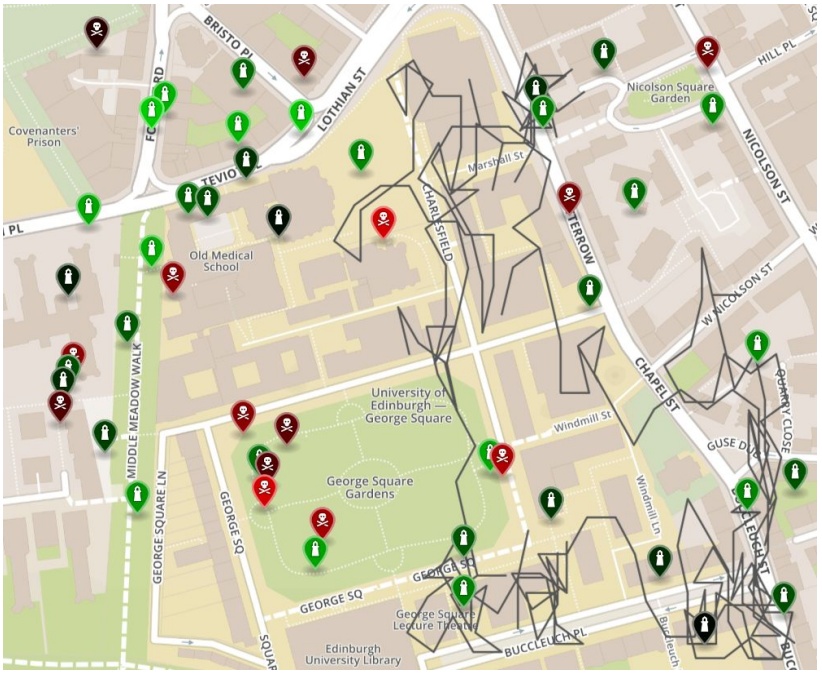

Figure 2.2: Stateless Drone flight path (final coins collected: 846.3601)

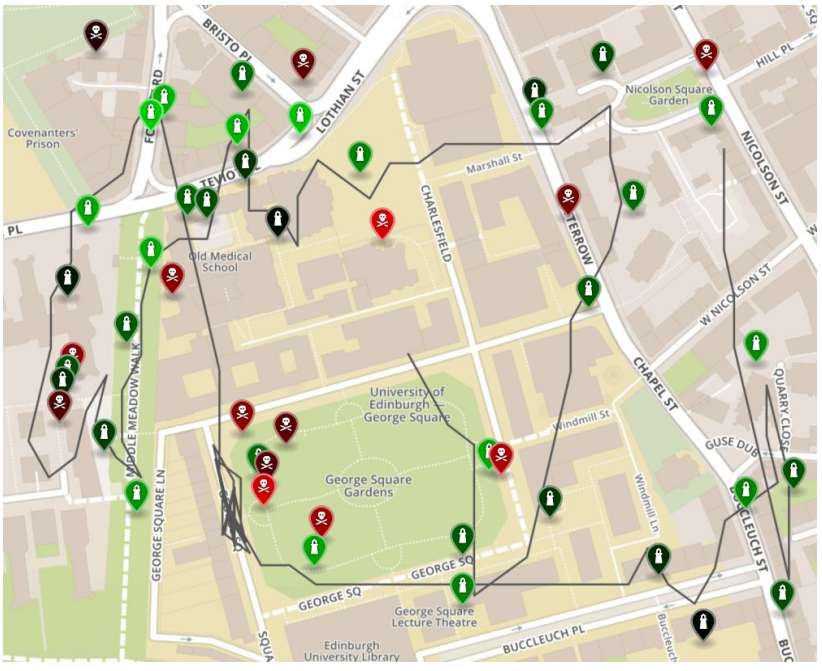

Figure 2.2: Stateful Drone flight path (final coins collected: 1974.8003)