# Informatics Large Practical

# Powergrab

# CW2: Implementation Report

# Matriculation Number: S1705270

## 1. Software Architecture Description

My powergrab application consists of 10 classes and 1 interface: the classes are **App**, **Direction**, **Drone**, **Game**, **GameLogger**, **GameMap**, **GameStatus**, **Position**, **Stateful**, **Stateless**, and the interface is **DroneInterface**. Following the divide-and-conquer strategy, I tried to keep every class short and comprehensible. At the same time, I tried to not create unnecessary classes or interfaces. Because of the strategy I used, all classes are very closely related to each other, and methods defined in one class are very likely to be used in other classes.

**DroneInterface** is an interface specifically for **Drone**, and **Stateful** and **Stateless** are subclasses of **Drone**. Drone class contains common methods that both Stateful and Stateless classes utilize, so I was able to avoid repetition of the same, redundant codes. Also, I needed a method called getDirectionToMove() for Drone class, but its functionality differs for Stateful and Stateless. So, I decided to create DroneInterface and add an abstract function in it so that the method can be defined in Stateful and Stateless, respectively, and therefore the right method gets executed for each type of Drone. This allowed me to avoid the need to specify the type of drone when trying to use the method getDirectionToMove(). If I did not make it an abstract function of Drone, I would need to specify the type of drone every time I want to use the method, and this would require more lines of code and more complexity, therefore making the project more messy and complicated. Also, because I made Drone as a superclass for Stateful and Stateless, I had no need to try hard to write codes for a particular drone, but I could just focus on writing codes for Drone in general. This reduced not only the amount of codes I had to write but also the amount of time I had to spend on building this game. Furthermore, this improves the maintainability of the project; when updates required are not related to the type of drone, we can just update codes in Drone instead of updating both Stateless and Stateful. If updates required are specifically for Stateless or Stateful drone, then we should go into either Stateless or Stateful class to update codes.

**GameStatus** contains a method that reports game status such as whether or not game is finished (game is finished when drone runs out of power or makes 250 moves). It also contains methods that report how much coins or power the drone possess. It also tells how many moves are available to the drone. These coins, power and available moves can be

regarded as status of drone or also as status of game. I hesitated whether I should name the class DroneStatus. However, Drone class already has two subclasses and one interface which it implements, so I decided to name the class GameStatus in the attempt to keep complexity of the project at low level. Also, I thought it would be awkard that the method that reports whether game is finished is contained in a class called DroneStatus. I could pass to Drone class coins, power and available moves as arguments, but I thought that this would violate the divide-and-conquer strategy I followed in this project.

**GameMap** is a very important class as it contains very important methods such as getMap() or getNearestStation(). GameMap receives a stringified url of game map. It uses it to obtain the game map and also to get all the features (or stations) from the map. Without GameMap class, we are unable to get the game map and also an array of features, and this prevents us from executing a lot of important methods.

**GameLogger** is a class specifically for logging the traces of drone. We are required to output a text file that contains all traces of our drone, so this class contributes to logging all the traces and making a stringified, detailed path using the traces.

**Game** is a class that contains the method startGame(). The method startGame() is very concise, and this is possible because I follwed the divide-and-conquer strategy. startGame() uses many complex functions that have already been defined in other classes. For example, it logs the trace of drone using a method from GameLogger class and updates drone's available moves and power using a method from GameStatus class. Because classes are successfully dependent on each other like this, it was possible to keep each class concise, comprehensible and light-weight. **App** class is also very simple; it receives only basic information as arguments, such as latitude and type of drone, and starts game immediately through the use of method startGame() which is defined in Game class. No other method is defined or implemented in **App** class.

## 2. Class Documentation

2.1 Direction

- It is the shortest class in this application. It is actually an enum class, defining directions as constants. It lists 16 directions: N, NNE, NE, ENE, E, ESE, SE, SSE, S, SSW, SW, WSW, W, WNW, NW, NNW, where N represents North, E East, S South and lastly W West. No value or property is given to each direction in this class.

2.2 Position

- Position is defined by two arguments: latitude and longitude both of which are of type Double.
- Five methods are defined in this class: degreeOf(Direction), nextPosition(Direction), inLatitude(), inLongitude(), inPlayArea().
- degreeOf(Direction) returns the degree of the given direction. East is 0 degrees, North 90 degrees, West 180 degrees, and South 270 degrees. It utilizes if-else conditional statements to check which direction is given as an input and returns degree accordingly.
- nextPosition(Direction) computes the next position based on the given direction. Using the given direction, it first computes the degree of the direction. Then, using the degree together with latitude and longitude of current position, it computes the latitude and longitude of next position. This computation uses this formula: 0.0003 * sin(degree) + latitude of current position for latitude of next position and 0.0003 * cos(degree) + longitude of current position for longitude of next position.
- It creates a new Position by passing these new latitude and longitude as arguments and returns it.
- inLatitude() is a boolean method that checks whether or not drone is in allowed range of latitude. Drone must be at latitude greater than 55.942617 and less than 55.946233, so this is explicitly checked.
- inLongitude() is a boolean method that checks whether or not drone is in allowed range of longitude. Drone must be at longitude greater than -3.192473 and less than -3.184319, so this is explicitly checked.
- inPlayArea() is a boolean method that checks whether or not drone is in allowed range of area. It returns AND value of inLatitude() and inLongitude(). This is because if drone is in allowed range of latitude and longitude, then it is in allowed range of area.

2.3 Game

- Game is defined by six arguments: type of drone, latitude, longitude, seed, url of map, and name of file. Their types are String, Double, Double, Long, String and String, respectively.
- Three methods are defined in this class: createDrone(droneType, latitude, longitude, seed), startGame() and writeToFile(fileName, path, contents).
- createDrone(droneType, latitude, longitude, seed) checks if droneType equals “stateless” or “stateful” and returns that type of drone accordingly. For example, if droneType equals “stateless”, the method returns a drone of type Stateless. Latitude, longitude and seed that are received as arguments are passed onto either Stateless or Stateful drone. As explained below, both Stateless and Stateful classes are defined by three arguments: latitude, longitude and seed.
- startGame() is a method that is used in class App in order to play game. It contains a lot of important methods that are defined in other classes. For example, updateDronePositionOnGameMap(drone, gameStatus) is a very important method that is defined in class **GameMap**. startGame() also contains moveInDirection(direction), updateAvailableMoves(), logDrone(previousPosition, directionToMove, gameStatus) and addDronePathToMap(dronePath), which are from **Drone**, **GameStatus**, **GameLogger**, and **GameMap**, respectively. Using “while” loop and a boolean method isFinished() from GameStatus, startGame() runs the game successfully through the use of various important methods from other classes mentioned earlier. When isFinished() returns TRUE, the while loop ends, and then writeToFile() is executed to output .txt and .geojson files.
- writeToFile(fileName, path, contents) is a simple void method that outputs .txt and .geojson files using given arguments.

2.4 GameStatus

- GameStatus is defined by three arguments: coins, power and availableMoves.
- Eight methods are defined in this class: one boolean method, three getters, two setters and two update functions.
- isFinished() is a boolean method that checks if game can still go on or should be finished. When available moves become zero or drone runs out of power, the game is finished, so these two values are checked by the method.
- getCoins(), getPower(), getAvailableMoves() simply return coins, power and available moves, respectively.

- setCoins(coins) and setPower(power) simply set coins and power using the given number.
- updateAvailableMoves() decrements available moves by one, and updatePowerByOneMove() decreases power by 1.25. These methods exist because every move by drone consumes one move and 1.25 power.

2.5 GameMap

- GameMap is defined by only one argument: url of map.
- However, we are able to get the JSON map using the method getMap(url). In the method, we can obtain a particular JSON map using the given url with the use of Java built-in functions HttpURLConnection and InputStream. From the JSON map, we are able to obtain a FeatureCollection using FeatureCollection.fromJson() method, and also from this FeatureCollection, we are able to get all the features (or stations, in this particular case) using FeatureCollection.feature() method. These features are put into an array of Feature.
- getNearestStation(Position) is a method that returns Feature given Position. As the name suggests, its purpose is to return the nearest station from the given position. It uses a for loop to loop over all stations and calculate distance between given position and each station on game map. The station which has the shortest distance to the given position is returned.
- getNearbyStations(Position) returns an array of stations (or features) based on the given position. It computes the subsequent position when drone moves from given position to each of 16 directions and finds the nearest station from the subsequent position using getNearestStation() method. Each nearest station obtained this way is appened to an array, and the array is returned.
- isNearNegativeStation(Position) is a boolean method that returns true if the nearest station from given position is a dangerous (or negative) station. It computes the nearest station from given position and checks if the station's coins are positive or negative.
- addDronePathToMap(dronePath) receives an array of locations (Point) of drone. Using LineString.fromLngLats() method, this array is turned into a type of Geometry, and this is again turned into a type of Feature using Feature.fromGeometry() method. This Feature, which contains all locations drone has been to, is now appended to the array of Feature that we created at the beginning of the class.
- updateDroneAndStationOnGameMap(Drone, GameStatus) receives drone and game status as arguments. We use drone to find its current position through the use of

getCurrentPosition() method. Using this position, we get the nearest station by the use of getNearestStation(Position) method. Then, we get the station's coins and power. We use given game status to find coins and power our drone currently has. Then, we add the station's coins and power to our coins and power, respectively. If each sum is greater than 0, we set our drone's coins and power to the sum, and the station's coins and power become zero because our drone took them. Otherwise, we set our drone's coins and power to zero because our coins and power can't be negative, and the station's coins and power become the negative sum.

- calculateDistance(Position, Point) is a simple mathematical method that returns distance between a location of type Position and another location of type Point. It uses their latitude and longitudes to compute the distance between them.

2.6 GameLogger

- GameLogger is defined by two arguments: latitude and longitude. This class makes two important variables: traceOfDrone and pathOfDrone. traceOfDrone is a String that records every movement of drone during game, and pathOfDrone is an array of drone's every location during game.
- logDrone(Position previousPosition, Direction direction, GameStatus gameStatus) literally logs drone's movement together with how much coins and power it has. This is appended to traceOfDrone, and drone's location after every moment is added to pathOfDrone. By the way, Drone's movement is logged in the way it is specified in the coursework description.
- getDroneTrace() and getDronePath() simply output traceOfDrone (String) and pathOfDrone (Array), respectively.

2.7 Drone

- Drone is defined by three arguments: latitude, longitude and seed. Only two methods, getCurrentPosition() and moveInDirection(Direction), are defined in this class but there is actually one more method defined in DroneInterface: getDirectionToMove(GameMap). This method is actually the reason that DroneInterface exists. Both Stateless and Stateful class extends from Drone, and they both have a getDirectionToMove() method. However, their getDirectionToMove() methods are different and therefore I could not define the method in Drone class. So, I put the method in DroneInterface to let the compiler know that such method exists for Drone, and I specifically defined the method in Stateless and Stateful, respectively. getCurrentPosition() and

moveInDirection(Direction) are common methods for both Stateless and Stateful. The first method returns drone's current position, and the second method is a void function that updates drone's current position to a position after which it moves in the given direction.

2.7.1 Stateless

- Stateless class extends from Drone, so it is also defined by three arguments: latitude, longitude and seed. Using the given seed, it creates a random number generator using a Java built-in function called Random().
- getSumOfCoinsAndPower(Feature) obtains the coins and power of given station (or feature) and simply adds them.
- getHashMap(GameMap) returns a hash map with key being direction and value being sum of coins and power of station nearest from position when drone moves in the direction (key).
- getDirectionToMove(GameMap) obtains a hash map using getHashMap(), and then it returns direction whose value is the greatest. This is because drone wants to move in direction where the nearest station has the biggest sum of coins and power.

2.7.2 Stateful

- Stateful class extends from Drone, so it is also defined by three arguments: latitude, longitude and seed.
- getHashMap(GameMap) returns a hash map with key being Feature (or station) and value being station's value that is computed by evaluateStation() function. We first obtain all stations on map and removes negative stations. Then, we append the pair station-its value for each station to the hash map.
- evaluateStation(Feature) literally evaluates given station. It computes the distance between drone's current position and station. We use reciprocal value of distance because this allows us to have a bigger value if distance is shorter. We return the sum of reciprocal value of distance, station's coins and station's power.
- bestStation(GameMap) simply returns the station with the biggest value (sum of reciprocal value of distance between station and drone's position, station's coins and station's power)
- bestDirectionTowardStation(Feature, GameMap) returns the best direction to move toward given station. It returns the direction that leads drone to get closer to the given

station than any other direction. Direction that leads drone to a position where its nearest station is negative station is detected and therefore not returned.

- getDirectionToMove(GameMap) returns the direction for drone to move. It does not require any other argument aside from GameMap. If there is no positive station on map, this method leads drone to direction where drone will not get close to negative station. If there are one or more positive stations on the given map, the best direction toward the best station is returned by the use of both bestStation() and bestDirectionTowardStation() methods.

2.8 App

- App class is a main calss that receives input from user and runs the game. It receives basic input, such as latitude, longitude and type of drone, as arguments and then starts game. This is possible because all complex methods have already been defined in other classes.

## 3. Stateful drone strategy

The performance of stateful drone is much better than stateless drone because

1. It is able to scan the entire map and all stations, also how much coins and power they have.
2. It is able to find the best station to move toward at every move.
3. It is able to find the best direction to move toward a particular station

3.1 Stateful drone's navigation algorithm

- **Stateful** class obtains an array of all existing stations on game map by creating a clone of such array that is defined in **GameMap** class.
- We delete negative stations from the array.
- We loop over the array of stations and add to a hash map a Station-Value pair for each station. Station's value is computed by evaluateStation() function; how this functions works is defined in 2.7.2.
- When there are one or more positive stations on game map, we move drone in "best direction" toward "best station". "Best station" is the station with the largest value in the hasp map obtained above. "Best direction" is the direction among all 16 possible directions in which drone moves and the distance between drone and "best station" is the shortest. There is an exception to this: even though the direction in which drone moves places drone closer to the "best station" than any other direction, if its nearest station after movement is a negative station, the direction is neglected.
- When there is no positive station on map, any direction that satisfies these two conditions is returned first: the direction does not place drone to a position where the nearest station is a negative station and the direction must lead drone to a position that is in allowed range of latitude and longitude. We loop over all 16 directions, starting from North toward West, so even though there are multiple directions that satisfy these two conditions, the direction that satisfy them first is returned.

The two images below show navigation of stateless drone and stateful drone. The stateful drone's navigation displays how the strategies that have specifically been used for stateful drone are applied. The stateful drone is able to get perfect scores for all maps in 2019 and 2020.

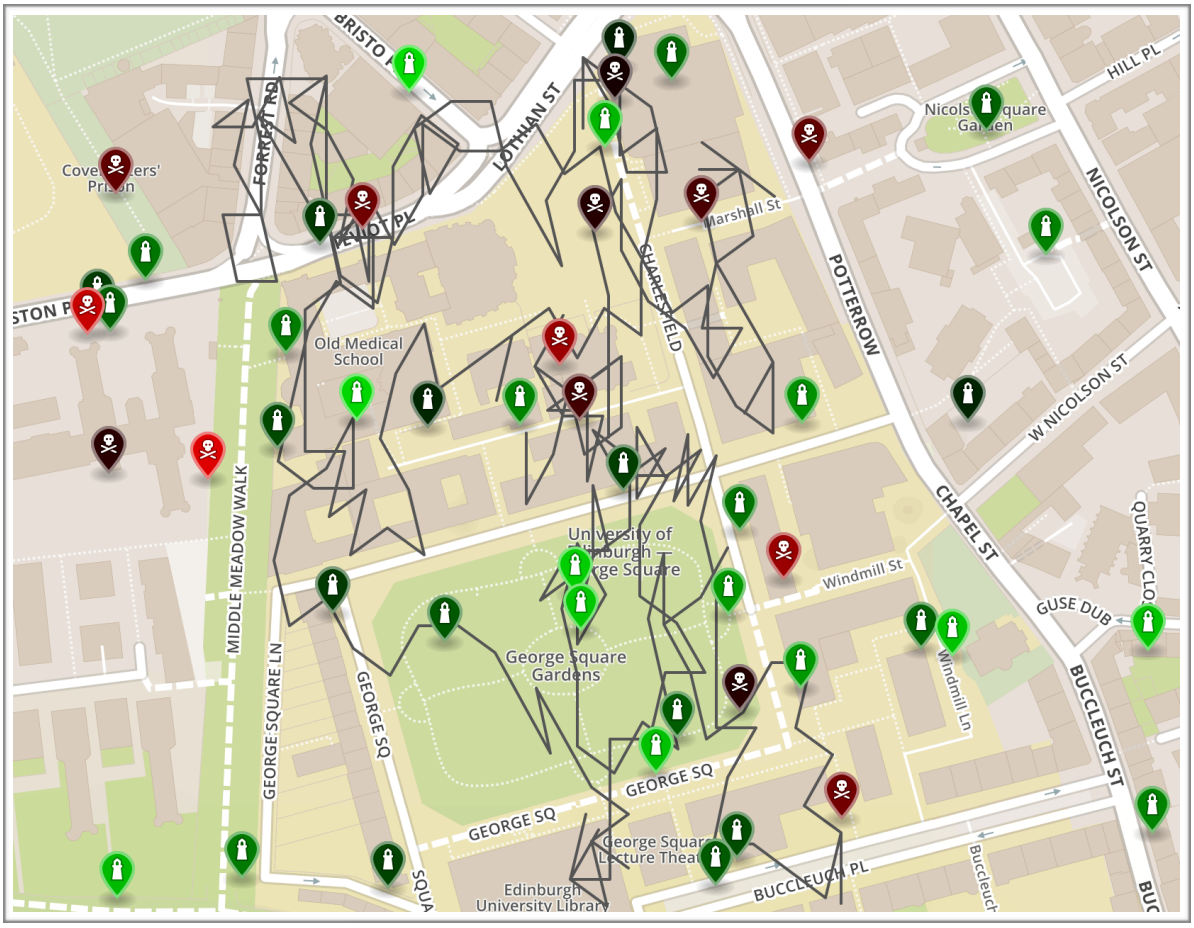


Figure 1: Stateless drone path for 02/02/2019 map starting at 55.944425 -3.188396 with 5678 seed

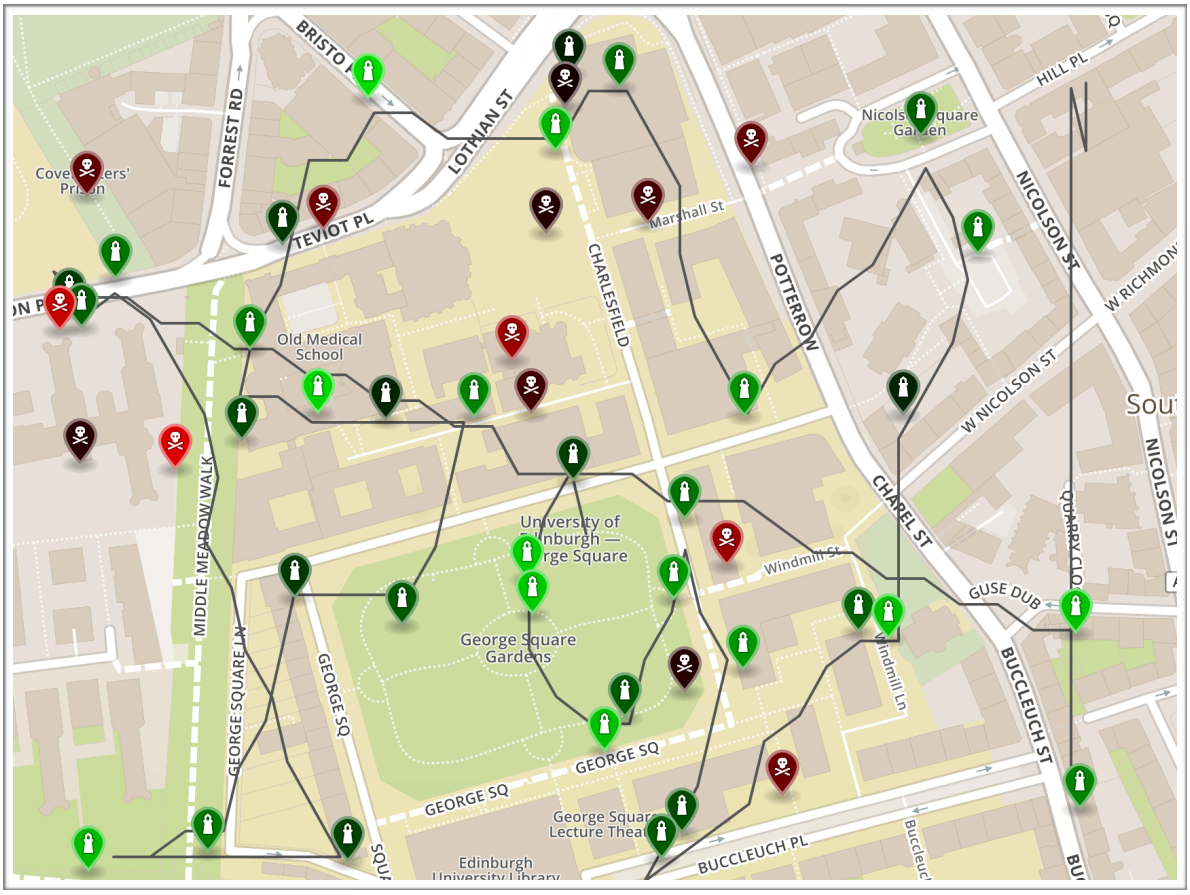


Figure 2: Stateful drone path for 02/02/2019 map starting at 55.944425 -3.188396 with 5678 seed