

Informatics Large Practical Coursework 2

Theodor Amariucai



Stephen Gilmore, Paul Jackson

School of Informatics

UNIVERSITY OF EDINBURGH

**Table of Contents**

[1. Software architecture description 2](#__RefHeading___Toc778_56519270)

[General structure 2](#__RefHeading___Toc1252_56519270)

[Object-Oriented Programming principles 2](#__RefHeading___Toc1258_56519270)

[Cohesion 2](#__RefHeading___Toc1277_56519270)

[Coupling 3](#__RefHeading___Toc1279_56519270)

[2. Class documentation 3](#__RefHeading___Toc902_56519270)

[App class 3](#__RefHeading___Toc1217_56519270)

[Game class 3](#__RefHeading___Toc1232_56519270)

[Map class 4](#__RefHeading___Toc1234_56519270)

[Station class 4](#__RefHeading___Toc1236_565192701)

[Drone class 4](#__RefHeading___Toc1236_56519270)

[Stateless class 5](#__RefHeading___Toc1238_56519270)

[Stateful class 5](#__RefHeading___Toc1240_56519270)

[Position class 5](#__RefHeading___Toc1242_56519270)

[Direction class 5](#__RefHeading___Toc1246_56519270)

[3. Stateful drone strategy 5](#__RefHeading___Toc904_56519270)

# Software architecture description

## General structure

My application is comprised of 9 different Java classes, each with a separate, well-defined purpose: *App*, *Game*, *Map*, *Station*, *Drone*, *Stateless*, *Stateful*, *Position* and *Direction*. I identified these classes as being the right ones for my application because they properly address the 4 principles of Object-Oriented Programming: Inheritance, Encapsulation, Abstraction and Polymorphism. This software architecture facilitates desirable cohesion and coupling properties, and also makes use of design patterns such as the Singleton design pattern. You do not need to discuss App, Direction and Position in Section 1.

## Object-Oriented Programming principles

* Considering the hierarchical relationships between classes, Stateless and Stateful are both subclasses of abstract class Drone. Through **inheritance**, they acquire all the properties and behaviors of the parent class Drone, thus greatly improving code reusability. This still allows them to diverge from the base class Drone through the use of different variables and methods (in accordance with their clearly distinct strategies).
* **Encapsulation** is used in my application to hide the values or state of a structured data object inside classes, thus preventing unauthorized parties' direct access to them.1 I implemented this by restricting access to data fields which are only used internally in the class through the use of the private keyword, and by creating getters and setter to provide public access only when absolutely required.
* **Abstraction**
* **Compile time polymorphism** is . **Run time polymorphism** is achieved e.g. through the overriding of the *move* method within the Stateful class. By way of dynamic method dispatch, Java determines which version(superclass/subclasses) of the method is to be executed based upon the type of the object being referred to at the time the call occurs.2

## **Cohesion**

My application has a high class cohesion (which, intuitively, is the degree to which the elements inside a module belong together)3 because most variables and methods which are grouped together contribute to a single well-defined task of a given class e.g. class *Map* and its methods *getAllStations*, *getCollectedStations*, *getUncollectedStations, getPositiveUncollectedStations* etc. This directly translates to increased robustness, reliability, and understandability.

## **Coupling**

On the other hand, the application also displays loose coupling

# Class documentation

Provide concise documentation for each class in your application. Explain each class as through providing documentation for a developer who will be maintaining your application in the future. Java design patterns?

## App class

The *App* class defines the entry point in the program thorough its *main* method in which command line arguments are parsed according to their specified meaning. The map is then retrieved from the web using method *readFromURL*. Logic is then passed over to the Game class which handles the drone’s flight simulation.

Methods *generateResultFiles* and *mapPerfectScore* are only used for generating (submission) output files.

## **Game** class

The *Game* class, on instantiation, in the constructor, computes the shifts in latitude and longitude for all 16 possible directions through the *precomputeMovementShift method*. This vastly improves the application runtime, as calculations will be stored in memory and reused.

Private variable *path (List<Point>)* is used to track the points where the drone has been to. This is only ever used in the *createOutputGeoJson* method, which creates a new *FeatureCollection* in GeoJson format adding the drone’s tracked path to the initial stations, which is finally printed to a *.geojson* file using *pathWriter*.

Method *play* is the backbone of the entire flight simulation logic: while the drone has moves left and has not run out of power, choose a direction to move in based on the current position and known map stations, move the drone in that direction and record it through variable *moveWriter.* At the end of the simulation, it populates the *.txt* file.

Method *getGameScore* is only used when (optionally) calculating the perfect score for a map in the generation of (submission) output files.

## **Map** class

The *Map* class contains an immutable list (enforced through the *final* keyword) of all stations on the map (*mapStations*), and it also records the already collected stations in a HashSet (*collectedStations*).

Another purpose of this class is to control all of the station management logic through methods which return or filter upon the aforementioned variables, thus providing the drones with insightful information: *getUncollectedStations*, *getPositiveUncollectedStations*, *getAllStations*, *getCollectedStations .*

Methods *distanceBetweenPoints* and *arePointsInRange* are also used to provide the drones with information: the positioning of stations on the map.

## **Station** class

The *Station* class represents the template of an in-game station, and contains three immutable variables (*coins*, *power* and *position*), all of which are used extensively throughout the game logic. Its constructor takes an instance of type *Feature* passed as an argument, and extracts those three field values from it by using MapBox library functions: *getNumberProperty(String key), geometry(),* and *coordinates().*

## **Drone** class

The *Drone* class defines It also provides a measure for calculating the utility of a station via its method *stationUtility.*

As part of the Drone class, I implemented the Singleton design pattern to ensure that only a single instance of the Drone class will ever be instantiated. This is a safe and straightforward way of complying with the logic of the project: under no circumstances should the Drone class have multiple instances in the same game. To enforce this in a single-threaded application such as PowerGrab, it is sufficient for the Drone class to implement the field and method as below:

private static Drone *instance* = null**;**  
  
static Drone createInstance(Position position**,** long seed**,** boolean submissionGeneration) {  
 if (*instance* == null || submissionGeneration)  
 *instance* = new Stateless(250**,** 0**,** position**,** seed)**;**  
return *instance***;**  
}

## **Stateless** class

The *Stateless* class defines

## **Stateful** class

The *Stateful* class defines

Class *DirectionOption* is an inner class of class *Stateful* and has the sole purpose of wrapping together boolean variable *isIdeal* and double variable *distance*, which are used in the aforementioned EnumMap *safeDirectionsStateful*.

## **Position class**

The class*Position*defines three immutable fields: *latitude* and *longitude* (accurately locating a point on the game map) and *moveShift.*

The latter is a static HashMap whose values are pre-computed in method *precomputeMovementShift* of class *Game* atinstantiation, and which is used in method *nextPosition* to perform the computation required to calculate the new position much faster.

Method *inPlayArea()* compares the current instance of class *Position* with two points marking the upper left (Forrest Hill) and lower right (Buccleuch St bus stop) bounds of the play area.

The sole reason for overriding the *toString()* method of this class is to enable its string representation to be used as a key in the *isStuck* method of class *Stateful*. This will ensure that identical positions will return the same hash code when run through the hash function.

## **Direction class**

The *Direction* class is an enumeration of 16 cardinal directions.

In addition to this, it provides two static variables which store the values of the enumeration in a list, and the size of this list in an integer. Those are then used in the static method *randomDirection* which takes a generator of type *Random* as parameter and returns a random direction out of the 16 possible choices.

# Stateful drone strategy

This section explains the strategy which is used by your stateful drone to improve their score relative to the stateless drone. You should explain what is remembered in the state of the stateful drone and how this is used to improve the drone’s score.

This section of your report should contain two graphical figures (similar to Figure 6 and Figure 7 in this document) which have been made using the http://geojson.io website, rendering one flight of your stateless drone and one flight of your stateful drone on a PowerGrab map of your choosing. It can be any of the available PowerGrab maps, but make sure that the same map is used for both the stateless drone and the stateful drone.

**The maximum page count of your project report is 15 pages in total counting everything. Add appendices?**

Bibliography

1: , , , https://en.wikipedia.org/wiki/Encapsulation\_(computer\_programming)

2: , , , https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/

3: Larry LeRoy Constantine, Edward Nash Yourdon, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, 1