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
General structure....2
Object-Oriented Programming principles....2
Cohesion....3
Coupling....3
2. Class documentation....4
App class....4
Game class....4
Map class....5
Station class....6
Drone class....6
Stateless class....6
Stateful class....7
Position class....7
Direction class....8
3. Stateful drone strategy....8
Overview....8
Rendering of stateful drone flight, seed 5679, map 12/08/2019....10
Rendering of stateless drone flight, seed 5679, map 12/08/2019....10

1. 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*.

Class **Game** oversees the principal simulation loop in method *play* (controlling the *Drone's* behaviour in its flight over the city of Edinburgh), and is in charge of writing results to the two specified *.txt* and *.geojson* files.

Class *Game* also keeps a reference to a *Map* object, which is a (unique) instance of class **Map**. This class provides general station management logic (collected stations, positive uncollected stations etc.) and Euclidean logic for various localization purposes (distances between stations etc.)

Class *Map* heavily relies on the **Station** class, which, in a sense, is a Data Transfer Object because it is only used to pass and reuse data and does not contain any business logic. It mostly has only getters and setters, and its values are simply retrieved from the parsed *GeoJson* string.

Abstract class **Drone** is a template for the two specialized classes Stateful and Stateless to extend upon, as it contains only base functionality: methods *move*, *chargeFromStation*, *hasPower, hasMovesLeft* etc. Its abstract method *chooseMoveDirection* must be implemented by both those subclasses and specifies the implementation of the drone's strategy.

Class **Stateless** implements the drone's strategy by taking a Greedy approach: whenever it is in range of a positive station, it moves closer to charge from it. Otherwise, it mostly wanders around the map aimlessly in random directions as it is not aware of anything beyond its immediate surroundings.

Finally, class **Stateful** improves upon the strategy of the stateless drone by always targeting the closest positive uncollected station on the entire map, thus gaining an immense advantage.

I identified these 6 additional 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 also displays two desirable properties: high cohesion and loose coupling.

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 behaviours of the parent class Drone, thus greatly improving code re-usability. This still allows them to diverge from the base class Drone by means of different variables and methods (in accordance with their clearly distinct strategies) e.g. class *Stateful* has extra fields *Queue<Position> trace* and *Station target.*
- **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.Encapsulation I implemented this by restricting access to data fields (through the use of the *private* keyword) that are only used internally in the class, and by creating getters and setters (accessors and mutators) which are used to effectively protect the data and to only provide public access when absolutely required and in the way that I specify.
- **Abstraction**, very similar in nature to the process of generalization, allows us to preserve information that is relevant in the context of our flight simulation game over the city of Edinburgh, and to forget information that is irrelevant in this context Introduction to Computation and Programming Using Python: we don’t model everything from the shape and aerodynamics of the drone to the buildings, trees, and wind speed, but merely essential information: coins, power, positions, directions etc.
- **Run time polymorphism** is achieved, in my application, through the overriding of the *move* method within the *Stateful* class: this allows the stateful drone to extend the base movement functionality by also keeping track of the last 5 moves made. This is also known as dynamic method dispatch: Java determines what version (super/sub class) of the method should be executed based upon the type of the object being referred to at the time the call occurs.Types of polymorphism in Java

Cohesion

The program has a high class cohesion (which, intuitively, is the degree to which the elements inside a module belong together)Systems Design: Fundamentals of a Discipline of Computer Programme and Systems Design because most variables and methods which are grouped together in a class contribute to a single well-defined task e.g. class *Map* and most of its methods *getAllStations*, *getCollectedStations*, *getUncollectedStations, getPositiveUncollectedStations* all handle station management and map logic etc. This directly translates to increased robustness, reliability, and understandability.

Coupling

On the other hand, the architecture also displays loose coupling, whose overall goal is to reduce the risk that a change made within one element will create unanticipated changes within other elements.Loose coupling

Consequently, most classes in my application are largely independent: *App* is the entry point, *Game* pre-computes shifts in position for the *nextPosition* method and only ever holds a reference to the abstract class *Drone* to simulate flight logic, *Direction* and *Position* are helper classes to aid in orientation and positioning, class *Map* manages stations and Euclidean logic, and class *Station* is a Data Transfer Object with no business logic.

The only exceptions are the *Stateless*, *Stateful* and (to some extent) the *Drone* class which all heavily rely on our instance of the *Map* class (inside of the *Game* class) to analyse options and make decisions. Consequently, it constantly queries the *Map* instance for information regarding stations, and it also eventually modifies the *coins* and *power* stored in stations (instances of class *Station*) when charging from them.

Limiting interconnections in this way will lead to loose coupling and can help isolate problems when things go wrong. It also simplifies testing, maintenance and troubleshooting procedures.Loose coupling

2. Class documentation

App class

The *App* class defines the entry point of the program through its *main* method in which command line arguments are parsed, in order, according to their specific meaning. The map is then retrieved from the web using method *readFromURL,* and logic is passed over to class *Game*, which handles the drone's flight simulation.

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

Game class

Class *Game*, on instantiation (in the constructor), computes the shifts in latitude and longitude for all 16 possible directions through the *precomputeMovementShift* method, to be used in the *static EnumMap* of helper class *Position*. This vastly speeds up the application runtime, not only because calculations will be stored in memory and reused, but also because of the *EnumMap* type which brings some minor improvements over *HashMaps*: zero probability of collision, specialized optimization using arrays etc.

Private variable *path (List<Point>)* is used to track the points where the drone has been to. This is recorded in method *play*, and only ever used in method *createOutputGeoJson*, which generates a new *FeatureCollection* in GeoJson format adding the drone's tracked path to the initial stations (and thus bringing the number of Features in the collection to 51). This is then finally printed to a *.geojson* file using map*PathWriter* at the end of the simulation.

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 *moveChoiceWriter.* During the simulation this populates the *.txt* file, and at the end of the simulation it populates the *.geojson* file.

Method *getGameScore* is only used when (optionally) retrieving the obtained coins for a map, to calculate and print the score percentage when generating (submission) output files.

Game constants are, intuitively, stored in immutable (enforced through the *final* keyword) static variables in the *Game* class: *RANGEDIST* (maximum distance for the drone to be considered in range of a station), *MOVEPOWERCOST* (amount of power drained when the drone moves in a direction), *MOVEDIST* (radius of the circle that the drone can move in).

In addition to those constants which are provided in the assignment description, I created two more parameters that are used by the drone to determine the utility of a station: *POWERWEIGHT* (the importance given to power when considering station utility), *COINSWEIGHT* (1-*POWERWEIGHT*, the importance given to coins when considering station utility).

Map class

The *Map* class contains a list of all stations on the map (*mapStations*), and it also records the already collected stations in a HashSet (*collectedStations*). Its constructor transforms a list of features into a list of stations with the help of class *Station* and its own constructor.

The main purpose of this class is to control all of the station management logic, thus providing the drones with insightful information for decision-making purposes. Therefore, *getUncollectedStations*, *getPositiveUncollectedStations*, *getAllStations*, *getCollectedStations* simply return or filter upon the stations of the map based on certain criteria, methods *distanceBetweenPoints* and *arePointsInRange* provide the drones with Euclidean logic, and method *stationUtility* estimates the utility of a station based on *COINSWEIGHT* and *POWERWEIGHT* constants defined in class *Game*.

Station class

The *Station* class represents the template of an in-game station, and contains three variables: *coins*, *power* and *position* (which is immutable). Instances of this class are all used extensively during the decision-making process of the drones through references stored in the *Map* class.

The constructor of the *Station* class takes an instance of type *Feature* passed as an argument, and extracts the three variables mentioned above by using MapBox library functions: *getNumberProperty(String key), geometry(),* and *coordinates().*

Drone class

The abstract class *Drone* defines base properties which are common for all drone specializations: *movesLeft*, *power*, *coins*, *position*, *randomDirGen*.

The same can be said about the common behaviour among drones, which is displayed in this class in the form of methods (shared by all classes that extend it): *move*, *hasMovesLeft*, *hasPower*, *chargeFromStation*. Abstract method *chooseMoveDirection* is a contract stating that drones should have a way of deciding where to go, but this is not specified in the base class: it is up to the specialized classes to implement this.

This is particularly useful as the main simulation in method *play()* of class the *Game* heavily relies on this feature: irregardless of whether the drone is stateless or stateful or of a completely different type, it will always call the same methods which could be (and most often are) differently implemented in the specialized drone classes and which are resolved later on, at runtime (type of polymorphism).

Stateless class

The *Stateless* class extends the *Drone* class and thus implements the *chooseMoveDirection* method, but also adds a new helper method *randomSafeDirection.*

In *chooseMoveDirection*, the drone simply iterates through all 16 directions, filters the valid ones (which are within map boundaries), calculates all the new positions which can be reached and for each one of them considers only the closest station within range (if any). If that station happens to be non-negative, it remembers it in a set of stations (*safeDirections)*. When time comes to make a decision, if out of all directions the maximum utility gain is 0, then the choice is between neutral directions and negative directions, and the drone picks a random neutral direction using method *randomSafeDirection*. Otherwise, it either picks the best choice among the worst when *maxGain*<0, or just the best choice overall when *maxGain*>0 (ideal).

Stateful class

The *Stateful* class also extends the *Drone* class and is very similar to the *Stateless* class, but provides two additional features which drastically improve performance:

1. It remembers the drone's last 5 moves to make sure it doesn't get stuck. This is done using variable *Queue<Position> trace,* the overridden method *move*, and method *isStuck()*.
2. It keeps track of the targeted station, always trying to get closer to it. This is done by remembering the target in variable *target*, and by constantly moving in the closest safe direction to it. At each step it also reassesses the integrity of the target (Has it been collected? Is the drone stuck? Is there even a target left or have all the positive stations been collected?). This is done in method *assessTargetPosition*.

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 *EnumMap* whose values are pre-computed in method *precomputeMovementShift* of class *Game* at instantiation, and which is used in method *nextPosition* to simply retrieve the computations required to calculate new positions rather than recalculate them, thus speeding up the algorithm considerably.

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 to make sure that game constraints are respected.

The only 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 string representations of positions will return the same hash code when run through the hash function of the HashMap used in method *isStuck()*.

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 number generator of type *Random* as parameter and returns a random direction out of the 16 possible choices. This is *static* because it does not belong to any instance specifically, but to the concept of "direction" in general.

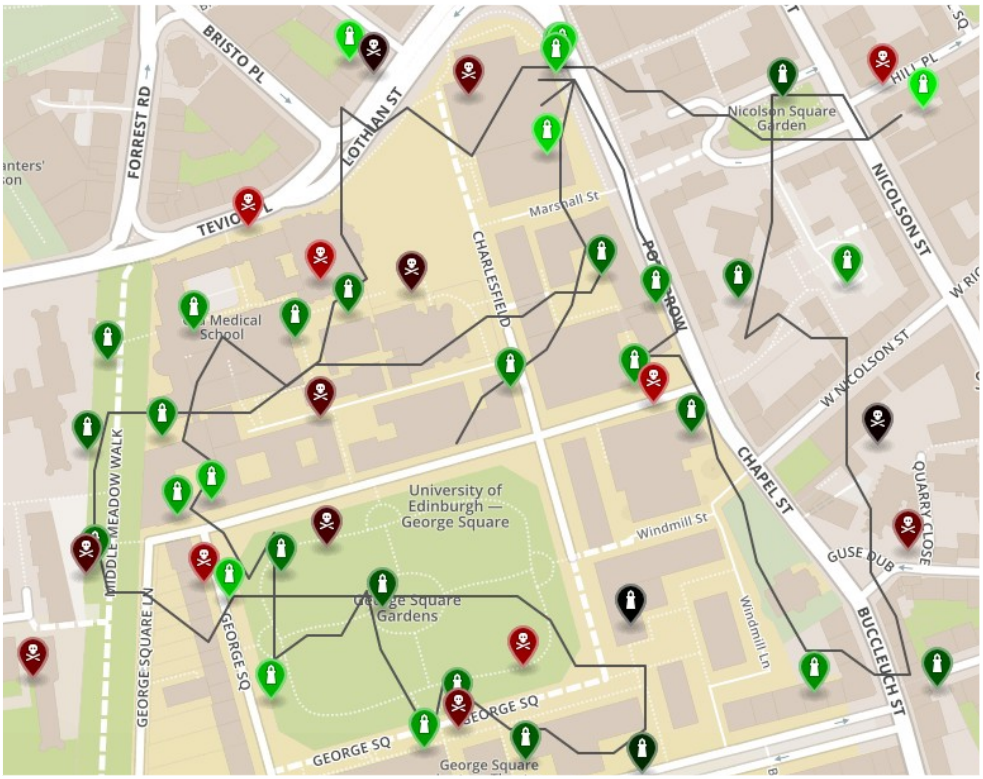
3. Stateful drone strategy

Overview

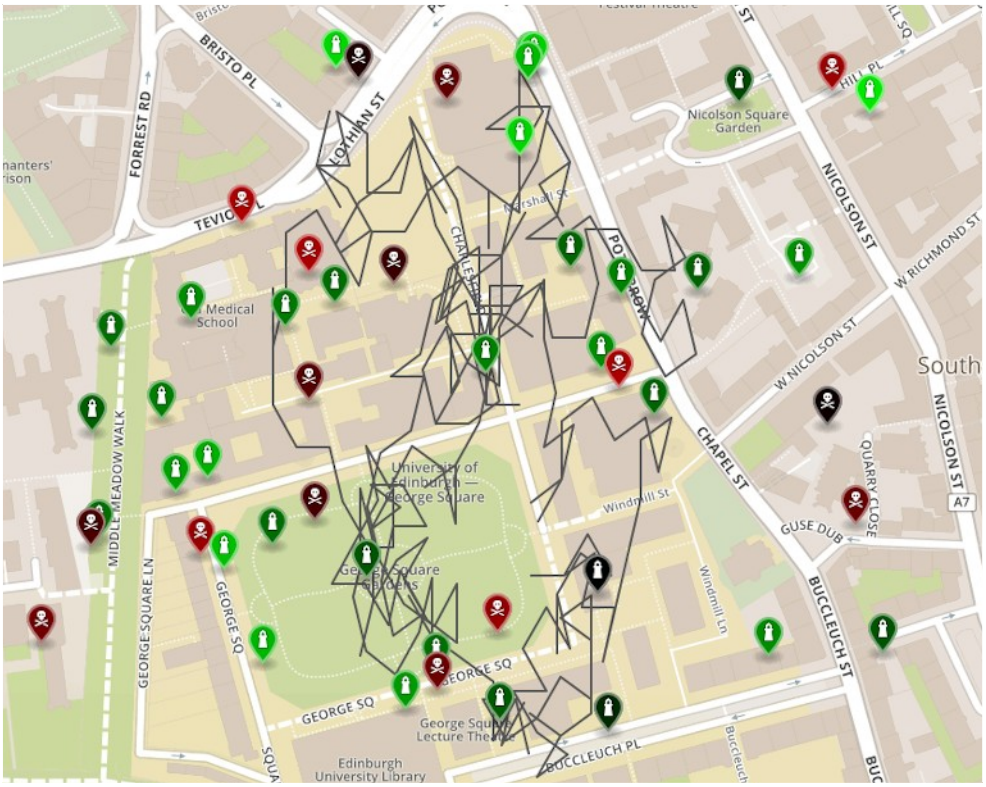
As briefly discussed above, to improve its score relative to the stateless drone, the stateful drone employs some minor modifications in code. Although the stateful drone also takes a Greedy approach, the modifications end up having a drastic positive influence on the final score. The state of the stateful drone (class instance) remembers its last 5 positions and keeps track of the current targeted station.

1. To remember the drone's last 5 positions, we use a FIFO data structure (*Queue<Position> trace*) and we override method *move* to always remove the last remembered position and refresh the newest position by adding it to the front of the queue. Method *isStuck()* is called in method *assessTargetPosition*: if the drone is stuck e.g. when behind a wall of negative stations, we consider retargeting any random station in the hope of getting it unstuck. This is proven to work excellently in practice.
2. The targeting of stations is used to improve the drone's score by defining, across the entire map, a global goal to reach that is most beneficial to the drone. Incidentally, this happens to be the closest positive uncollected station. The drone then keeps track of the targeted station, always trying to get closer to it. This is done by firstly remembering it in variable *target*, and secondly by constantly moving in the closest safe direction by using method *safeDirectionClosestToTarget*. At each step the drone also reassesses the integrity of the target (Has it been collected? Is the drone stuck? Is there even a target left or have all the positive stations been collected?). This is done in method *assessTargetPosition* at every step. When all the positive stations have been collected, the drone returns to station indexed with 0 and flies close to it until 250 moves have been completed.

Rendering of stateful drone flight, seed 5679, map 12/08/2019



Rendering of stateless drone flight, seed 5679, map 12/08/2019



Bibliography

Encapsulation: Dave Braunschweig, Encapsulation, , https://press.rebus.community/programmingfundamentals/chapter/encapsulation/

Introduction to Computation: Guttag, John V., Introduction to Computation and Programming Using Python, 2013

Polymorphism: Chaitanya Singh, Types of polymorphism in Java,

System Design: Edward Yourdon, Larry L. Constantine, Systems Design: Fundamentals of a Discipline of Computer Programme and Systems Design, 1979,

Loose Coupling : Margaret Rouse, Loose coupling, 2011

Loose Coupling : Margaret Rouse, Loose coupling,