INFORMATICS LARGE PRACTICAL COURSEWORK 2

Theodor Amariucai

STEPHEN GILMORE, PAUL JACKSON
SCHOOL OF INFORMATICS
UNIVERSITY OF EDINBURGH

Table of Contents

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 main simulation loop (making sure the *Drone* object behaves as expected), and is in charge of writing results to the two specified *.txt* and *.geojson* files. This class also keeps a reference to a *Map* object, which provides general station management logic (collected stations, positive uncollected stations etc.)

Class *Map* implicitly relies on the *Station* class which is an abstraction of a station object found in the parsed *GeoJson* string. In a sense, this is equivalent to a Data Transfer Object which is only used to pass and reuse data and does not contain any business logic. It mostly has simple setters and getters.

Finally, 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 necessarily implemented by the aforementioned subclasses.

I identified these 9 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 high cohesion and loose coupling properties as well, and also makes use of design patterns such as the *Singleton* design pattern.

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 by means of different variables and methods (in accordance with their clearly distinct strategies) e.g. class *Stateful* has extra fields *Queue<Position>*

trace (keeps track of the drone's last 5 positions) and *Station target* (keeps track of the drone's desired station to be reached next)

- **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.
- **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]: 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.[Polymorphism]

Cohesion

The program has a high class cohesion (which, intuitively, is the degree to which the elements inside a module belong together)[System 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 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 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 the instance of the *Map* class to analyze options and to make decisions, eventually modifying the *coins* and *power* stored in stations by 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 the Game class which handles the drone's flight simulation on the map.

Methods *generateResultFiles* and *mapPerfectScore* are optional, and should only be 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, to be used in the *static* HashMap of helper class *Position*. This vastly improves the application runtime, as calculations will be stored in memory and reused.

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.

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 then finally printed to a *.geojson* file using *pathWriter* at the end of the simulation.

Method *getGameScore* is only used when (optionally) calculating the perfect score for a map in the generation of (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*, *MOVEDIST* (radius of the circle that the drone can move in), *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*).

Another 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 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 the *Game* class.

Station class

The *Station* class represents the template of an in-game station, and contains three variables (*coins*, *power* and *position*, which is immutable). They are all used extensively throughout the decision-making process of the drones.

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 *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* at instantiation, and which is used in method *nextPosition* to perform the computations required to calculate new positions at a much faster rate.

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 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 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

To improve its score relative to the stateless drone, the stateful drone employs minor modifications in code which end up having a drastic positive influence on the final score. Its strategy takes the Greedy approach, where .

In addition to the above, the state of the stateful drone (also known as the class instance) remembers its last 5 positions and keeps track of the current targeted station.

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

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, 2011