******

Informatics Large Project Report

David Jorge

******

s1712653

Contents

[***Software Architecture Description*** 2](#_Toc25785315)

[**App** 2](#_Toc25785316)

[**AppUtil** 3](#_Toc25785317)

[**Drone** 3](#_Toc25785318)

[**Stateless** 3](#_Toc25785319)

[**Stateful** 4](#_Toc25785320)

[**POI** 4](#_Toc25785321)

[**FileOutput** 4](#_Toc25785322)

[**Direction** 4](#_Toc25785323)

[**Position** 4](#_Toc25785324)

[***Class Documentation*** 5](#_Toc25785325)

[**App** 5](#_Toc25785326)

[**main(String[] args)** 5](#_Toc25785327)

[**setupLogger()** 5](#_Toc25785328)

[**getMapSource(String url)** 5](#_Toc25785329)

[**computeMoveSequence(Drone drone, ArrayList<Point> points)** 6](#_Toc25785330)

[**initDrone(Position initialPosition, Random randNumGen, String droneType)** 6](#_Toc25785331)

[**buildJsonFile(String mapSource, ArrayList<Point> points)** 6](#_Toc25785332)

[**parseFeatures(String mapSource)** 7](#_Toc25785333)

[**AppUtil** 7](#_Toc25785334)

[**setupLogger()** 7](#_Toc25785335)

[**inputStreamToString(InputStream inputStream)** 7](#_Toc25785336)

[**arrayToString(ArrayList<String> moveList)** 7](#_Toc25785337)

[**formatTextOutput(Position firstPos, Position secondPos, Direction direction, double coins, double power)** 7](#_Toc25785338)

[**Drone** 7](#_Toc25785339)

[**Drone(Position initialPosition, Random randNumGen)** 8](#_Toc25785340)

[**setupLogger()** 8](#_Toc25785341)

[**getInRange()** 8](#_Toc25785342)

[**makeMove()** 8](#_Toc25785343)

[**updateStatus()** 8](#_Toc25785344)

[**hasPower()** 8](#_Toc25785345)

[**euclideanDist(double xLat, double xLong, double yLat, double yLong)** 8](#_Toc25785346)

[**Stateless** 8](#_Toc25785347)

# ***Software Architecture Description***

The powergrab application requires a mixture of different types of classes to function as intended. Well defined and widely used Java design and creational patterns were incorporated into the application design in order to maximise efficiency, increase modularization and to most importantly correctly implement the intended behaviour of the application as detailed in the design specifications. The main abstract idea behind the development of the powergrab application was to have a controller and the drone where both entities communicate with one another in order to function correctly. The end design includes a total of 9 different class as seen below:

Table 1: Table listing the classes made for this project along with their basic descriptions.

Figure 1: UML diagram of the application implementation.

|  |  |
| --- | --- |
| Class Name | Class Description |
| App | Handles I/O for the application and acts as a controller for the drone. |
| AppUtil | Provides the App class with support methods. |
| Drone | Abstract definition of a drone. |
| Stateless | Definition of stateless drone behaviour. |
| Stateful | Definition of stateful drone behaviour. |
| POI | Class for representing instances of features from the target maps. |
| FileOutput | Responsible for writing content to an output file. |
| Direction | Enum class providing values for the 16 cardinal directions used in the application. |
| Position | Class for representing a position on a 2D lat-long space. |

A screenshot of a cell phone

Description automatically generated

The interclass relationships can be seen from the UML diagram of the application design in *Figure 1*. The App class acts as the controller for the application, it will initialize an instance of one of the drone types, which are implemented as subclasses of an abstract representation of a drone. Utility class methods from AppUtil and FileOutput will also be called from the App class.

## **App**

The App class is an essential building block to the powergrab application. It is necessary for handling the I/O requirements for the application while simultaneously acting as a controller for the drone class.

From the App class the target map will be both read and parsed into usable information which will be used in the Drone class and its subclasses to compute the drone movements in the chosen map. In order to achieve this the methods defined in the AppUtil class will be used in order to support the ones defined in the App class. The App class will also act as a controller for the drone, dictating when it can move. As the Drone class is an abstract definition of a drone, the App class will require a functionality resembling the Factory creational pattern in order to get the desired drone type from an input. The App class will also create the necessary output files of the powergrab application as required by the design specifications. In order to do so it will call upon the FileOutput class in order to accomplish this.

The App class is designed to be a central control unit to the powergrab application, making it an essential class to have. Many functionalities are called from other classes instead of being directly defined in the App class. This was done to increase modularization in order to aid debugging and to make it easier to add specific additional functionalities to the application. Any I/O or map parsing changes should be modified in this class.

## **AppUtil**

The AppUtil class provides essential support methods to the App class. It handles mostly formatting methods that would otherwise clutter the App class if defined there. It is defined as a final class and increases modularization in the application.

It resembles a singleton design pattern by defining its constructor as private, which will prevent making instances of the class. It is designed to be accessed statically by the App class only providing support methods and nothing else, which will aid debugging. Any additional features implemented in the App class should have any support methods defined here.

## **Drone**

The drone class is essential to the powergrab application as it provides the drone functionality. It is an abstract representation of a drone and as such is defined as an abstract class. It is the superclass of the Stateful and Stateless classes which implement the actual concrete definition for the drone behaviour, it thus resembles the strategy behavioural design pattern where different implementations of an abstract method in the superclass are made by the subclasses, where the one executed is dependant on the type of subclass instantiated.

An abstract superclass for both drone types was created instead of having separate classes for the drone types in order to increases efficiency and modularization, avoid code cluttering and repetition by providing an abstract representation for a drone, which could be used for adding additional drone types effectively. This is all accomplished most efficiently by utilizing a hierarchical structure. Any additional shared functionalities between drone types can be modified in this class.

## **Stateless**

The Stateless class is essential to the powergrab application as it defines the behaviour of the stateless drone type. As such it extends the Drone class as it represents a drone. This class will contain all the methods essential for defining the stateless drone behaviour.

The stateless drone type was given its own class in order to increase efficiency and modularization, and take advantage of the abstract drone representation class, which allows it to be separate from the other drone type and therefore it only required to implement functionalities unique to the stateless drone. Any additional features to the stateless drone behaviour can be modified in this class.

## **Stateful**

The Stateful class is essential to the powergrab application as it defines the behaviour of the stateful drone type. As such it extends the Drone class as it represents a drone. This class will contain the methods which make up the stateful drone strategy.

Like the stateless drone type, the stateful drone type is also given its own class to increase efficiency and modularization while taking advantage of an abstract representation of a drone defined by the Drone class. This allows it to be separate from the other drone types, implementing its own strategy while making use of shared methods contained in the Drone class. Any additional features to the stateful drone behaviour can be modified in this class.

## **POI**

The POI class is essential to the powergrab application as it represents an instance of a point of interest in the target map. It will be used to both store and represent the basic information from the features parsed from the target map.

During the execution of the drone strategies it is essential to have a way to represent the information that each feature in the target map holds. This is because feature contain more than one piece of information, thus it is represented as an instance of the POI class.

The POI class is used throughout the other defined classes. Any changes to feature representation can be modified in this class.

## **FileOutput**

The FileOutput class is essential in order to produce the output files made by the powergrab application, which would otherwise clutter the App class if its function was defined there. It is defined as a final class and increases modularization in the application.

It resembles a singleton design pattern by defining its constructor as private, which will prevent making instances of the class. It is designed to be accessed statically by the App class only providing a method for writing to an output file and nothing else, which will aid debugging. Any additional additional features or changes to the way the output file should be written can be modified in this class.

## **Direction**

The Direction class is essential to the powergrab application as it defines a well-defined way to represent the 16 cardinal directions used in the drone functions, increasing efficiency. It is used extensively by the abstract Drone class and its subclasses for implementing the behaviour of the drone types. It is also used by the App class in order to produce the contents of the output files, which includes the directions taken by the drone at each move. Any additional directions or changes can be modified in this class.

## **Position**

The Position class is essential to the powergrab application as it defines a well-defined way to represent a 2D point in a lat-long space. It is used extensively by all classes that require a way to represent the position of an entity. Both the drone and the features have positional information and therefore the use of this class increases efficiency. The behaviour of each drone is also dependant on this class as it requires calculations to be done on positions. Any additional features or changes to the position representation can be modified in this class.

# ***Class Documentation***

## **App**

This class implements the main routine for the application. It consists exclusively of static methods that operate primarily on or return objects representing arguments and outputs of the drone functionality. It’s responsible for handling the I/O of the application and for controlling the drone functionality. Where applicable the methods of this class can throw an IllegalArgumentException, MalformedURLException and IOException, indicating invalid arguments or outputs. This class has 4 attributes: a public ArrayList, POIs, of type POI which store references to all individual features of the target map, a private Logger object, logger, to log statements in this class, and 2 private floats, totalCoins and coinsCollected, representing the total possible amount of coins to be acquired and the total amount acquired by the drone during its move sequence respectively.

### **main(String[] args)**

This method implements the main routine of the application. It will be called from the command line with 7 arguments representing the inputs to the application. The input arguments are ordered as follows: the day, month and year of the target map, the latitude and longitude for the starting position of the drone, the random seen and the drone type.

The method will instantiate the logger by calling the method setupLogger(), and parse all input arguments, after which it will get the target map information, using an URL formatted from the input arguments, by calling the getMapSource() method. The information representing individual features in the target map will then be stored in the POIs attribute by calling the parseFeatures() method with the target map information as its argument. A Position object representing the drone’s initial position will be created from the input arguments, as well as an ArrayList of type Point, which will hold all points visited by the drone and will be used to build the LineString for the output geojson file. The Drone object will then be initialized by calling the initDrone() method with the Position object representing the initial position of the drone, the pseudo-random number generator object and the drone type as its arguments. The type of drone object returned will depend on the drone type argument.

The move sequence of the drone will then be computed by calling the computeMoveSequence() method with the Drone object as its argument. The Arraylist of type Point will also be passed by reference to the method. A String representing information on the state of the drone at each move will be returned, while already being formatted as specified by the design specifications on the text file content. This will then be written to an output text file by calling the writeToFile() method of the FileOutput class with it and the file name as arguments. Finally, the content for the output geojson file will be returned as a String by calling the buildJsonFile() method with the target map information and the ArrayList of type Point as arguments. This will again be written to an output file using the writeToFile() method with it and the file name as arguments.

### **setupLogger()**

This private method is responsible for creating a new Logger from the Logger class using its getLogger() method. A hierarchy of loggers will be created throughout the application with the App class logger as the superclass. The handler for the empty String logger, which all other loggers extend, will be set using the getHandlers() method. The importance level of statements reported by the logger can be modified here to specify the granularity of the information received to aid debugging.

### **getMapSource(String url)**

This private method is responsible for getting the target map information from the internet using a URL. It takes a String representing the URL containing the target map as its input. The method will open a connection using the URL String which will return an InputStream object. The input stream will be converted to a String using the inputStreamToString() method in the AppUtil class. Since opening a connection to the URL is prone to errors, the method will catch any MalformedURLExceptions or IOException. The method also checks whether the resulting String is null or not before returning it, throwing an IllegalArgumentException if it’s the case.

### **computeMoveSequence(Drone drone, ArrayList<Point> points)**

This private method allows the App class to act as a controller for the drone functionality of the application by computing the move sequence of the drone on the target map. Its inputs are a Drone object, representing the instance of the chosen drone type, and an ArrayList of type Point, representing the 2D points the drone visits during its move sequence, which is passed by reference to this method.

The method will use a while loop to call the Drone object’s makeMove() method 250 times for a total of 250 moves, as specified by the design specifications, cutting it short if the hasPower() method from the Drone class returns false. The Position object of the drone before and after making a move, as well as the direction it takes are stored. The formatTextOutput() method from the AppUtil class will then be called to parse this information into an entry of an ArrayList of type String using the format specified for the output text file of the application. The ‘point’ ArrayList will also be updated every move by adding onto it the final position of the drone as a Point object. After the while loop is completed the total amount of coins the drone collected is stored in the class attribute before returning the result of the arrayToString() method of the AppUtil class with the ArrayList of type String with the drone move sequence as its input.

### **initDrone(Position initialPosition, Random randNumGen, String droneType)**

This private method is responsible for instantiating one of the subclasses of the Drone superclass depending on an input. This was implemented in a way resembling the Factory design pattern as it was applicable in this case. Its inputs are a Position object representing the drone’s initial position, a Random object representing the pseudo-random number generator, and a String representing the drone type.

Depending on the input String this method will either return a new instance of the Stateless drone object or a new instance of the Stateful drone object, otherwise if the input String is invalid it throws an IllegalArgumentException.

### **buildJsonFile(String mapSource, ArrayList<Point> points)**

This private method builds the contents of a geojson file containing the geographical locations of the target map plus a line string representing the path the drone took during its move sequence. Its inputs are a String representing the target map information in a geojson format, and an ArrayList of type Point representing the 2D points the drone has visited in its move sequence, which is passed by reference. The geojson file contents are built from scratch, essentially incorporating the information from the original target map geojson file and the positions visited by the drone during its move sequence into one String, returning it in a geojson format.

### **parseFeatures(String mapSource)**

This private method parses the features, representing geographical locations, of the target map from its geojson file content. It takes a String representing the geojson file content describing the target map as its input. For each feature a new POI object is created, storing each feature’s information, and kept in the POIs class attribute. Any positive coin value of a feature is also added on to the class attribute representing the total coins of the target map.

## **AppUtil**

This class defines support methods for the App class. It consists exclusively of static methods, which will be called from the App class, that operate primarily on Strings, formatting them into a desired result. A private attribute, logger, of type Logger is kept by this class in order to make logging statements throughout its methods where applicable. The inherent empty Constructor for the class is made private in order to prevent the creation of instances of this class and the class is made final to prevent it from being extended.

### **setupLogger()**

This public method is responsible for instantiating a separate logger for this class. This logger will be kept as an attribute for the class and will extend the logger created in the App class using the getLogger() method from the Logger class.

### **inputStreamToString(InputStream inputStream)**

This public method is responsible for converting an InputStream object into a String. As an input it takes an InputStream object resulting from a connection attempt to a URL, and it outputs the content of that InputStream object as a String.

### **arrayToString(ArrayList<String> moveList)**

This public method converts an ArrayList of type String into a single String. It takes an ArrayList of type String and it outputs a String where every entity of the input ArrayList is separated by a new line.

### **formatTextOutput(Position firstPos, Position secondPos, Direction direction, double coins, double power)**

This public method takes the information stored at every drone move and formats it into a String. Its inputs are a Position object representing the initial position of the drone before a move, a Position object representing the final position of the drone after a move, the direction the drone took during the move, and the total value of the drone’s coins and power after the move. The output String format complies to the design specification of the output text file.

## **Drone**

This abstract class implements the abstract representation of a drone. It’s the superclass for the Stateless and Stateful classes, consisting of non-static methods which are shared by the two subclasses and the public abstract method makeMove() that is called by the App class to compute the next drone move. This resembles the Strategy behavioural design pattern where the implementation of a method, in this case the makeMove() method, changes depending on the type of drone. This class has 7 protected attributes: a Position object, currentPosition, representing the drone’s current position, 2 double variables, coins and power, representing the drone’s current coin and power values respectively, a final Random object, randNumGen, representing the pseudo-random number generator used by the drone, an ArrayList of type POI, inRange, representing the features in charging range of the drone’s current position, and a Logger object, logger, used by the Drone class and its subclasses.

### **Drone(Position initialPosition, Random randNumGen)**

This is the Constructor for this class. It’s called by the Constructor of its subclasses, initialising all variables shared by the hierarchy. Its inputs are a Position object representing the drone’s initial position, and a Random object representing the pseudo-random number generator. The setupLogger() method is also called to setup the logger for the Drone class and its subclasses.

### **setupLogger()**

This private method is responsible for instantiating a separate logger for this class. This logger will be kept as an attribute for the class and will extend the logger created in the App class using the getLogger() method from the Logger class.

### **getInRange()**

This protected method stores all features that are within charging range (0.00025 degrees) of the drone’s current position in the inRange class attribute. The inRange ArrayList is cleared every time this method is called.

### **makeMove()**

The public abstract declaration of the makeMove() method is done in this class. This method will be called by the App class every time the drone is required to compute its next move. The concrete definition of the method is implemented by the subclasses. It returns a Direction object representing one of the 16 cardinal directions the drone chose to take for the move.

### **updateStatus()**

This protected method updates the drone’s current coin and power values if there is a station in charging range. It only carries out the transaction with the closest station in charging range to the drone’s current position.

If no feature is found to be in charging range of the drone’s current position this method does nothing, otherwise if the closest feature is a lighthouse, the method adds the coin and power values to the drone’s own, setting the feature’s coin and power values to 0. If the closest feature is a danger it subtracts the feature’s coin and power values from the drone’s own. The drone’s value of coins and power can’t be negative, so any excess is kept by the feature, otherwise the feature’s coin and power values are set to 0

### **hasPower()**

This public method returns a Boolean value representing whether the drone has a power value greater than 2.5 (power necessary to make a move) or not.

### **euclideanDist(double xLat, double xLong, double yLat, double yLong)**

This protected method returns the Euclidean distance (L2 norm) between two 2D points.

## **Stateless**

This class extends the abstract Drone class. It represents the stateless drone and consists of non-static methods that implement its behaviour. It also defines the abstract method makeMove() in the superclass. It keeps one private attribute ArrayList, inMoveRange, of type POI representing all features that could be in charging range after a single move.

### **Stateless(Position initialPosition, Random randNumGen)**

Constructor for the Stateless class. It is executed whenever a new instance of the Stateless class is initialized, calling the superclass constructor and the getInRange() method overridden by this class. Its inputs are a Position object representing the initial position of the drone and a Random object representing the pseudo-random number generator.

### **getInRange()**

This protected method overrides the one defined in the superclass. Apart from storing all features that are within 0.00025 degrees of the drone’s current position, it also stores all features that are further than 0.00025 degrees away but less than 0.00055 degrees (move distance plus charging distance) in the inMoveRange ArrayList kept as an attribute by this class. This represents the features that could be in charging range after the drone has made a single move.

### **updateState(ArrayList<Direction> moveList)**

This private method is responsible for updating all necessary attributes of the drone after it has computed which direction(s) to go in. It takes an ArrayList of type Direction, representing the set of possible directions the drone can take, as dictated by the makeMove() method, as an input. It outputs a Direction object representing the final direction the drone has chosen to take.

It chooses a Direction at random, if necessary, from the input ArrayList using the pseudo-random number generator attribute. The current position of the drone is then updated with the nextPosition() method of the Position class, the power attribute is deducted for the move, and the getInRange() and updateStatus() methods are called to get the features now visible by the drone and to charge from the closest feature in range if applicable.

### **makeMove()**

This public method implements the abstract method declared in the superclass for the stateless drone behaviour. It computes sets of direction(s) that are possible for the drone to take at its current position, updates the attributes describing its state at the end of the move and returns a Direction object representing the direction the drone took.

From the 16 possible direction it makes sets of them for each of the following conditions which have priorities as follows, ordered from most important to least important:

1. The drone charges to a lighthouse by taking that direction.
2. The drone is not in charging range of any feature or in charging range of a feature with no coin and power values by taking that direction.
3. The drone moves to a valid position.

The method then calls the updateState() method to update the attributes describing its current state with the most favourable non-empty set of direction(s) it computed, returning the Direction object representing the direction the drone takes in this move.

## **Stateful**

This class extends the abstract Drone class. It represents the stateful drone and consists of non-static methods that implement its behaviour. It also defines the abstract method makeMove() in the superclass. It keeps 5 private attributes: a POI object, target, representing the lighthouse the drone is currently aiming to charge from, a Position object, prevPos, representing the drone’s previous position, a Boolean, hasJustCharged, representing whether or not the drone has charged from a lighthouse in its previous move, an ArrayList, unvisitedPOIs, of type POI representing the lighthouses the drone hasn’t charged from yet, and an int, stuckCounter, representing the amount of moves the drone has taken to reach the current target.

### **Stateful(Position initialPosition, Random randNumGen)**

Constructor for the Stateful class. It is executed whenever a new instance of the Stateful class is initialized, calling the superclass constructor and calling the loadTargets() and getNextTarget() methods in order to store all the lighthouses in an attribute and set the drone’s current target. Its inputs are a Position object representing the initial position of the drone and a Random object representing the pseudo-random number generator.

### **loadTargets()**

This private method iterates through all features in the target map and stores a reference to all features that are lighthouses with positive coin or power values in the class attribute unvisitedPOIs, representing the lighthouses the drone hasn’t charged from yet.

### **getNextTarget()**

This private method sets the class attribute representing the drone’s current target with the feature from the unvisitedPOIs ArrayList, representing the lighthouses the drone hasn’t charged from yet, that is closest to the drone’s current position. If the unvisitedPOIs class attribute is empty, this method sets the value of the attribute representing the drone’s current target to null. If the class attribute is set to a not null value, its reference is also removed from the unvisitedPOIs class attribute.

### **getRandomMove()**

This private method returns a Direction object representing the most favourable ‘random’ direction the drone is made to take. It computes sets of direction(s) that are possible for the drone to take at its current position, updates the attributes describing its state at the end of the move and returns a Direction object representing the direction the drone took.

From the 16 possible direction it makes sets of them for each of the following conditions which have priorities as follows, ordered from most important to least important:

1. The drone doesn’t charge from a danger feature.
2. The drone moves to a valid position.

It calls the updateState() method with the set of directions computed to return a Direction object representing the direction the drone takes.

### **updateState(ArrayList<Direction> moveList)**

This private method is responsible for updating all necessary attributes of the drone after it has computed the set of possible direction(s) to go to. It takes an ArrayList of type Direction, representing the possible directions the drone can take, as dictated by the makeMove() method, as an input. It outputs a Direction object representing the final direction the drone has chosen to take.

It chooses a Direction at random, if necessary, from the input ArrayList using the pseudo-random number generator attribute. The current position of the drone is then updated, the power attribute is deducted for the move, and the getInRange() and updateStatus() methods are called to get the features now visible by the drone and to charge from the closest feature in range if applicable. It will also check if the drone is stuck on its current target if applicable. If the drone currently has a target it will check if the target has been charged from, calling the getNextTarget() method to get the next target if so. Otherwise it checks if other lighthouses, that are not the target, have been charged from, removing them from the unvisitedPOIs class attribute.

### **hasTarget()**

This private method returns a Boolean value representing whether the drone currently has a target or not. The drone doesn’t have a target if its target attribute is null.

### **makeMove()**

This public method implements the abstract method declared in the superclass for the statateful drone behaviour. It computes sets of direction(s) that are possible for the drone to take at its current position, updates the attributes describing its state at the end of the move and returns a Direction object representing the direction the drone took.

From the 16 possible direction it makes sets of them for each of the following conditions which have priorities as follows, ordered from most important to least important:

1. The drone makes progress to charge from the current target by taking that direction.
2. The drone is not in charging range of any danger feature or in charging range of a feature with no coin and power values by taking that direction.
3. The drone moves to a valid position.

The method then calls the updateState() method to update the attributes describing its current state with the most favourable non-empty set of direction(s) it computed, returning the Direction object representing the direction the drone takes in this move.

## **POI**

This class is used to represent instances of features from the target maps. It keeps 7 public attributes describing feature characteristics. It stores the ID, the latitude and longitude, the value of coins and power, the symbol and the colour of a feature instance.

### **POI(String id, double latitude, double longitude, double coins, double power, String symbol, String colour)**

Constructor for the POI class. It stores characteristics of a feature in the attributes of the instance of this class.

## **FileOutput**

This class is responsible for handling output file writing requests. It consists exclusively of static methods which either setup a logger for the class or write contents to an output file. A private attribute, logger, of type Logger is kept by this class in order to make logging statements throughout its methods where applicable. The inherent empty Constructor for the class is made private in order to prevent the creation of instances of this class and the class is made final to prevent it from being extended. Where appropriate, the methods of this class can throw an IOException.

### **setupLogger()**

This public method is responsible for instantiating a separate logger for this class. This logger will be kept as an attribute for the class and will extend the logger created in the App class using the getLogger() method from the Logger class.

### **writeToFile(String filename, String text)**

This public method outputs a file with name and contents specified by its input. Its input are a String, representing the output file name, and a String representing the output file contents. As file writing is error prone, this method catches any IOExceptions.

## **Direction**

This class is used to define a collection of constants representing the 16 cardinal directions along with the angle their angle in degrees.

### **Direction(double angle)**

Constructor for the direction class. Allows for each defined constant to have an angle associated with it.

## **Position**

This class represents an instance of a position in a 2D lat-long space. It consists of non-static methods that operate on the current position. It keeps 3 public attributes for an instance: a double representing the latitude of the position, a double representing the longitude of the position, and a final double representing the distance a drone travels during a move.

### **Position(double latitude, double longitude)**

Constructor for the Position class. It stores the characteristics of a position in the attributes of the instance of this class.

### **nextPosition(Direction direction)**

This public method returns the next position relative to the current position after travelling a distance of 0.0003 degrees in one of the 16 cardinal directions. It takes a Direction object representing one of the 16 cardinal directions and returns the position after the move using trigonometry.

### **inPlayArea()**

This public method returns a Boolean value representing whether or not the current position is within the well-defined playing are of the powergrab application.

# **Stateful Drone Strategy**

Unlike the stateless drone, which is intentionally limited to very restricted information about the target map, the stateful drone is designed to have no limitations, and thus it is free to use any strategy in order to win the game. This includes having no limited lookahead capabilities.

The basic step-by-step idea behind the strategy implemented for the stateful drone is as follows:

1. A list containing information about the position of all lighthouses with positive coin and/or power values in the target map is created.
2. The closest lighthouse to the drone’s current position is selected from the list to be the ‘target’, consequently removing it from the list.
3. The drone will then move in a direction that brings it closer to the ‘target’.
4. Step 3 repeats until the drone is in charging range of the ‘target’. This assumes the drone still has enough power remaining to move.
5. Once it is confirmed that the drone has charged from the ‘target’, the next closest lighthouse to the drone’s current location is selected from the list as the next ‘target’.
6. Steps 3 to 5 will then repeat until the list contains no more entries and the drone has charged from its current target.
7. If the drone still has enough power remaining to make a move, choose a random direction that doesn’t subject the drone to losing any coins or power from charging to a feature.
8. Step 7 repeats until a total of 250 moves have elapsed or the drone has run out of power.

The basic idea listed above doesn’t account for any danger features in the path of the drone while its moving to its current ‘target’, which could deduct coin and power values from the drone. This goes against the design specification of the stateful drone where it requires it to get the highest score possible, therefore a method for the drone to essentially avoid the danger features while going moving to its next target had to be devised.

Unlike the stateless drone, the stateful drone implementation will have an idea of a current ‘target’ that the drone has to aim to charge from before any other feature, which is kept in the drone’s memory, this will give an advantage over the stateless drone as the stateful drone will be able to compare which directions take it closer to the ‘target’, instead of moving in random directions.