Drones R Us

Object Oriented Programming
1st Project, version 1.0 – 2019-03-26
Contact: mgoul@fct.unl.pt

Important remarks

Deadline until 23h59 (Lisbon time) of April 12, 2019.

Team this project is to be MADE BY GROUPS OF 2 STUDENTS.

Deliverables: Submission and acceptance of the source code to **Mooshak** (score > 0). See the course web site for further details on how to submit projects to Mooshak.

Recommendations: We value the documentation of your source code, as well as the usage of the best programming style possible and, of course, the correct functioning of the project. Please carefully comment both interfaces and classes. The documentation in classes can, of course, refer to the one in interfaces, where appropriate. Please comment the methods explaining what they mean and defining preconditions for their usage. Students may and should discuss any doubts with the teaching team, as well as discuss this project with other students, but may not share their code with other colleagues. This project is to be conducted with full respect for the Code of Ethics available on the course web site.

1 Development of the application *Drones R Us*

1.1 Problem description

The goal of this project is to develop an application that simulates the management of a fleet of drones that deliver packages with orders from an e-commerce web site to their clients. These drones may fly at a low altitude, carrying packages from their bases to their customers' address. There are two kinds of drones used in this service. *"Hermit drones"*, which are drones that fly solo, and *"Sociable drones"*, which are drones that have the ability of forming *swarms*, to increase their capacity to carry packages. A swarm behaves, in many ways, as a bigger drone. The company has several distribution bases from which the drones fly to deliver packages to customers.

Each drone (be it a hermit or a sociable one) has a unique identifier, a maximum distance it can fly before it requires service, and a package carrying capacity. Sociable drones can be combined to form swarms. Swarms can be combined with other drones, or swarms, to create even bigger swarms. A swarm may also be completely dismantled, in which case its drones become available again for usage in solo missions. Swarms combine the capacity of its component drones, by simply adding the capacities of those drones. The maximum range a swarm can fly to is limited by the lowest range of any of its components. Swarms can only be formed by idle drones stationed in a base hangar. After flying a given distance, drones have to be serviced. This implies moving idle drones from the hangar to a service bay, and servicing them, replenishing

the drones' ability to fly up to a maximum distance. Drones within a swarm cannot be serviced. The swarm needs to be disbanded for its drones to get serviced. Swarms can only be created from idle drones in the hangar (including other swarms) and disbanded when the swarm is idle, in the hangar. Drones being serviced, or currently stationed in other bases cannot be added to a swarm.

Drones (and swarms) deliver packages to satisfy orders from the customers and then return to base. Swarm identifiers, just like other drone identifiers, are unique. When flying, drones expend a portion of their total available range. The same happens to drones within a swarm. Both the positions of the base and customer are known and characterized by *latitude* and *longitude* coordinates (for simplicity, we will consider these as integer values). The application user must decide when to send drones for being serviced and, if necessary, to remove them from swarms, service them and getting them back to swarms.

For the sake of simulation, we assume drones to fly at a constant speed of 10 distance units per time unit. Deliveries will take the double of the distance from the source base (for flying from the base to the destination and back to the base). Servicing drones also takes one time unit. After servicing a drone, the drone becomes available again on the base hangar for further missions.

2 Commands

In this section we present all the commands that the system must be able to interpret and execute. In the following examples, we differentiate `text written by the user` from the feedback written by the program in the console. You may assume that the user will make no mistakes when using the program other than those described in this document. In other words, you only need to take care of the error situations described here, in the exact same order as they are described.

Commands are case insensitive. For example, the `exit` command may be written using any combination of upper and lowercase characters, such as `EXIT`, `exit`, `Exit`, `exIT`, and so on. In the examples provided in this document, the symbol ↵ denotes a change of line. Whenever a command is expected from the user, the program should present a *prompt* denoted by the symbol >, which is always followed by a white space. The user input is written right after that white space, in the same line.

If the user introduces an unknown command, the program must write in the console the message Unknown command. Type help to see available commands. For example, the non existing command `some random command` would have the following effect:

> `some random command`↵
> Unknown command. Type help to see available commands.↵

Several commands have arguments. You may assume that the user will only write arguments of the correct type. However, some of those arguments may have an incorrect value. For that reason, we need to test each argument exactly by the order specified in this document. Arguments will be denoted **`with this style`**, in their description, for easier identification.

2.1 `exit` command

Terminates the execution of the program. This command does not require any arguments. The following scenario illustrates its usage.

```
> exit↵
Bye!↵
```

This command always succeeds. When executed, it terminates the program execution.

2.2 `help` command

Shows the available commands. This command does not require any arguments. The following scenario illustrates its usage.

```
> help↵
base - registers a new distribution base↵
listBases - lists existing distribution bases↵
drone - registers a new drone in a starting base↵
service - service all grounded drones requiring service in a base↵
swarm - creates a new swarm from free drones in a base↵
swarmComponents - lists the drones within a swarm↵
disband - disbands the whole swarm↵
listDrones - lists all existing drones (and swarms)↵
flyToBase - fly a drone (or swarm) to a base↵
addOrder - add a new order to a base↵
orders - lists all pending orders from a base↵
allOrders - lists all pending orders from all bases↵
deliver - deliver a package to a customer↵
delivered - lists all delivered orders↵
inTransit - lists all drones (and swarms) currently flying↵
ticTac - advances the simulation clock n timestamps↵
help - shows the available commands↵
exit - terminates the execution of the program↵
```

This command always succeeds. When executed, it shows the available commands.

2.3 `base` command

Registers a new distribution base. The command receives as arguments the base's coordinates (**`Latitude`** and **`Longitude`**) and a **`base name`**. For the sake of simplicity, we will assume that **`Latitude`** and **`Longitude`** are represented as integer values. **`base name`** uniquely identifies the base to be created.

Suppose you want to create three bases named *Montijo*, *Sintra* and *Monte Real*.

```
> base↵
3870 -897 Montijo↵
Base Montijo created at [3870,-897].↵
> base↵
3880 -938 Sintra↵
Base Sintra created at [3880,-938].↵
> base↵
3951 -852 Monte Real↵
Base Monte Real created at [3951,-852].↵
```

If some parameter is incorrect, the base is not created and an adequate error message is presented:

- (1) If there is already a base with the same name, the error message is (Base already exists!).
- (2) If there is already another base at the same location, the error message is (There is already another base at <location>!).

The following example illustrates an interactive session where these error messages would be generated. The problem with the input is highlighted in **red**.

```
> base↵
3870 -897 Montijo↵
Base Montijo created at [3870,-897].↵
> base↵
3870 -897 Montijo↵
Base already exists!↵
> base↵
3870 22 Montijo↵
Base already exists!↵
> base↵
3870 -897 Somewhere else↵
There is already another base at [3870,-897]!↵
```

2.4 `listBases` command

Lists existing distribution bases. This command does not require any parameters and always succeeds. It lists all the existing bases in the order by which they were added. For each base, it presents 3 lines: the first line has the name and location of a base. The second line has a list of the drones which are currently available in the hangar, waiting for a delivery mission. This is a comma separated list. For each drone, the program presents its id, its capacity and its remaining range, before it requires service. The third line presents a list of the drones that are currently in the service station, waiting for being serviced, so they can then be returned to the hangar and become available for new missions. The drones waiting service are presented exactly in the same way as those waiting in the hangar for a mission.

Suppose you have created the three bases (Montijo, Sintra and Monte Real, by this order) and that you have a couple of drones in the hangar and three drones being serviced in Montijo, one drone in the hangar and but none in the service bay at Sintra, and no drones in Monte Real. The output of the `listBases` command would be as follows:

```
> listBases↵
Montijo [3870,-897]↵
Hangar: [cristiano 20 40, pepe 30 100]↵
Service bay: [fesja 20 40, battaglia 30 100, aboubakar 10 25]↵
Sintra [3880,-938]↵
Hangar: [drone01 30 40]↵
The service bay is empty!↵
Monte Real [3951,-852]↵
The hangar is empty!↵
The service bay is empty!↵
```

If there are no bases, the program presents the message There are no bases!.

```
> listBases↵
There are no bases!↵
```

2.5 `drone` command

Registers a new drone in a starting base. The command receives 5 parameters: the new **`drone id`**, the **`base id`**, the **`kind`** of drone (either **`hermit`** or **`sociable`**), the carrying **`capacity`** of the drone, and the **`range`** the drone can travel before it requires going to the service station for maintenance. In case of success, the program presents the feedback message Drone <drone id> created.

In the following example, we create a sociable drone, a hermit and another sociable drone. We add all drones to the base in Montijo. Finally, we list the bases. For the sake of this example, we assume there is only the Montijo base and that it is empty to begin with.

```
> listBases↵
Montijo [3870,-897]↵
The hangar is empty!↵
The service bay is empty!↵
> drone↵
bee01↵
Montijo↵
sociable 20 100↵
Drone bee01 created.↵
> listBases↵
Montijo [3870,-897]↵
Hangar: [bee01 20 100]↵
The service bay is empty!↵
> drone↵
lonelyOne↵
Montijo↵
hermit 40 80↵
Drone lonelyOne created.↵
> drone↵
bee02↵
Montijo↵
sociable 40 80↵
Drone bee02 created.↵
> listBases↵
Montijo [3870,-897]↵
Hangar: [bee01 20 100, lonelyOne 30 40, bee02 40 80]↵
The service bay is empty!↵
```

If some parameter is incorrect, the program is unable to create the drone:

- (1) The company does not use drones with a maximum range lower than 10. If it is lower than 10, the adequate error message is Invalid range for a new drone!
- (2) Capacity has to be a positive integer number (>0). Otherwise, the adequate error message is Capacity has to be a positive integer!
- (3) If there is no base with the given base id, the adequate error message is Base <base id> does not exist!

- (4) If there is already a drone with the same identifier, the adequate error message is Drone <drone id> already exists!
- (5) If the drone kind is unknown, the adequate error message is Invalid drone type!

The following scenario illustrates these issues:

```
> drone↵
lonelyOne↵
Sintra↵
hermit 40 80↵
Drone lonelyDrone created.↵
> drone↵
badDrone↵
Montijo↵
hermit 50 2↵
Invalid range for a new drone!↵
> drone↵
badDrone↵
Montijo↵
hermit -50 2↵
Capacity has to be a positive integer!↵
> drone↵
badDrone↵
Mojito↵
hermit 50 20↵
Base Mojito does not exist!↵
> drone↵
lonelyOne↵
Montijo↵
hermit 50 20↵
Drone lonelyOne already exists!↵
> drone↵
lonelyOne↵
Montijo↵
weirdKindOfDrone 50 20↵
Invalid drone type!↵
```

2.6 `service` command

Service all grounded drones requiring service in a base. The command receives parameters containing the **`base id`** and the **`remaining range threshold`** below which all drones must be moved to the service bay within the base. This provides a convenient way for the system manager to send all drones with a low range left for servicing. The effect of servicing a drone is that its total range is restored. This command only moves drones from the hangar to the service bay. Moreover, it only applies to atomic drones which are not part of a swarm. The service itself takes some time (see **`TicTac`** command for further details on how time is simulated in this program).

```
> listBases↵
Montijo [3870,-897]↵
Hangar: [swarm07 50 120, drone01 20 100]↵
The service bay is empty!↵
Sintra [3880,-938]↵
Hangar: [drone02 20 80]↵
The service bay is empty!↵
Monte Real [3951,-852]↵
The hangar is empty!↵
The service bay is empty!↵


> service↵
Montijo↵
105↵
drone01 moved to service bay.↵
> listBases↵
Montijo [3870,-897]↵
Hangar: [swarm07 50 120]↵
Service bay: [drone01 20 100]↵
Sintra [3880,-938]↵
Hangar: [drone02 20 80]↵
The service bay is empty!↵
Monte Real [3951,-852]↵
The hangar is empty!↵
The service bay is empty!↵
```

If some parameter is incorrect, or there are no idle atomic drones on the base hangar, the program is unable to send drones for servicing:

- (1) If the base id does not correspond to any base owned by the company, the adequate error message is Base <base id> does not exist!
- (2) If the threshold is too low, and there are no drones with a remaining range below it, no drones are sent to the service bay and the adequate message is No drones were sent to the service station! Note that this includes the case where no drones were available in the hangar, to begin with.

The following scenario illustrates these issues:

```
> service↵
Lajes↵
100↵
Base Lajes does not exist!↵
> service↵
Montijo↵
10↵
No drones were sent to the service station!↵
```

2.7 `swarm` command

Creates a new swarm from free drones in a base. The command receives parameters containing the **`base id`**, the **`swarm id`** the **`number of initial drones`**, and the **`drones`**

ids to join the swarm. The following scenario illustrates the creation of a new swarm of three drones. Note that the capacity of the integrated drones is summed and that the available range is the minimum current range of the three drones.

```
> listBases↵
Montijo [3870,-897]↵
Hangar: [drone07 50 120, drone01 20 100, drone03 20 80, drone08 100 100]↵
The service bay is empty!↵
Sintra [3880,-938]↵
The hangar is empty!↵
The service bay is empty!↵
Monte Real [3951,-852]↵
The hangar is empty!↵
The service bay is empty!↵
> swarm↵
Montijo↵
swarm02↵
3↵
drone03↵
drone07↵
drone01↵
swarm02 created↵
> listBases↵
Montijo [3870,-897]↵
Hangar: [drone08 100 100, swarm02 160 80]↵
The service bay is empty!↵
Sintra [3880,-938]↵
The hangar is empty!↵
The service bay is empty!↵
Monte Real [3951,-852]↵
The hangar is empty!↵
The service bay is empty!↵
>swarm↵
Montijo↵
swarm03↵
2↵
swarm02↵
drone08↵
swarm03 created↵
> listBases↵
Montijo [3870,-897]↵
Hangar: [swarm03 260 80]↵
The service bay is empty!↵
Sintra [3880,-938]↵
The hangar is empty!↵
The service bay is empty!↵
Monte Real [3951,-852]↵
The hangar is empty!↵
The service bay is empty!↵
```

If some parameter is incorrect, the program is unable to create the swarm and the drones remain free for any other purposes:

- (1) If the base id does not correspond to any base owned by the company, the adequate error message is Base <base id> does not exist!

- (2) If the number of the initial drones is less than 2, then it is not possible to create a swarm and the adequate error message is Swarm must have at least two drones!
- (3) If one tries to include the same drone more than once, then it is not possible to create a swarm and the adequate error message is Cannot add drone <drone id> twice!
- (4) If one tries to include a hermit drone in the swarm, then it is not possible to create the swarm and the adequate error message is Cannot add hermit drone <drone id>!
- (5) If one tries to include an unavailable drone in a swarm, then it is not possible to create a swarm and the adequate error message is Drone <drone id> is not available in this base!
- (6) If the swarm id is already taken, then it is not possible to create a swarm. The adequate error message is Swarm <swarm id> already exists!

The following scenario illustrates these error messages. For increased readability, the drones have an identifier matching their kind.

```
> listBases↵
Montijo [3870,-897]↵
Hangar: [hermit01 50 120, sociable02 20 100, sociable03 20 80, swarm04 100 100]↵
The service bay is empty!↵
>swarm↵
Lajes↵
swarm05↵
2↵
sociable02↵
sociable03↵
Base Lajes does not exist!↵
>swarm↵
Montijo↵
swarm05↵
1↵
sociable02↵
Swarm must have at least two drones!↵
>swarm↵
Montijo↵
swarm05↵
2↵
sociable02↵
sociable02↵
Cannot add drone sociable02 twice!↵
>swarm↵
Montijo↵
swarm05↵
2↵
sociable02↵
hermit01↵
Cannot add hermit drone hermit01!↵
```

```
>swarm↵
Montijo↵
swarm05↵
2↵
sociable02↵
unavailable06↵
Drone unavailable06 is not available in this base!↵
>swarm↵
Montijo↵
swarm04↵
sociable02↵
sociable03↵
unavailable06↵
Swarm swarm04 already exists!↵
```

2.8 `swarmComponents` command

Lists the drones within a swarm. The command receives 1 parameter containing the **`swarm id`** and lists the drones within the swarm by the order in which they were added to the swarm. It also presents the capacity and current range of the swarm.

```
>swarmComponents↵
swarm02↵
drone03↵
drone07↵
drone01↵
160 80↵
```

If the parameter is incorrect, the program is unable to list the drones within the swarm:

- (1) If there is no swarm with the **`swarm id`**, the adequate error message is <swarm id> is not a swarm!

```
>swarmComponents↵
unknownSwarm↵
unknownSwarm is not a swarm!↵
```

2.9 `disband` command

Disbands the whole swarm. The command receives 2 parameters containing the **`base id`** and the **`swarm id`** and disbands the swarm. The drones in the swarm become available on the base.

```
> listBases↵
Montijo [3870,-897]↵
Hangar: [hermit01 50 120, swarm02 90 80]↵
The service bay is empty!↵
>disband↵
Montijo↵
swarm02↵
swarm02 disbanded.↵
> listBases↵
Montijo [3870,-897]↵
Hangar: [hermit01 50 120, drone07 50 120, drone01 20 100, drone03 20 80]↵
The service bay is empty!↵
```

If some parameter is incorrect, the program is unable to disband the swarm:

- (1) If the base id does not correspond to any base owned by the company, the adequate error message is Base <base id> does not exist!
- (2) If there is no swarm with the **`swarm id`** in this base, the adequate error message is <swarm id> is not at <base id>!

```
> listBases↵
Montijo [3870,-897]↵
Hangar: [hermit01 50 120, swarm02 90 80]↵
The service bay is empty!↵
>disband↵
swarm02↵
swarm02 disbanded.↵
> listBases↵
Montijo [3870,-897]↵
Hangar: [hermit01 50 120, drone07 50 120, drone01 20 100, drone03 20 80]↵
The service bay is empty!↵
```

2.10 `listDrones` command

Lists all existing drones (and swarms). The command receives no parameters and always succeeds. Drones and swarms are listed by their creation order. In the following example, swarm02 was created before hermit01.

```
> listBases↵
Montijo [3870,-897]↵
Hangar: [hermit01 50 120]↵
The service bay is empty!↵
Sintra [3880,-938]↵
The hangar is empty!↵
Service bay: [swarm02 90 80]↵
> listDrones↵
swarm02↵
hermit01↵
```

If there are no drones, the program presents the message There are no drones!.

```
> listDrones↵
There are no drones!↵
```

2.11 `flyToBase` command

Flies a drone (or swarm) to a base. The command receives 3 parameters containing the **`origin base id`** the **`drone id`** and the **`target base id`**, and orders the drone (or swarm) to move to a new base. This puts the drone in transit to the target base. The flight itself will be simulated with the `ticTac` command.

```
> flyToBase↵
Montijo↵
drone03↵
Sintra↵
drone03 flying from Montijo to Sintra.↵
```

If some parameter is incorrect, or the target base is too far for that drone, the command is aborted:

- (1) If the source base id does not correspond to any base owned by the company, the adequate error message is Source base <base id> does not exist!
- (2) If the target base id does not correspond to any base owned by the company, the adequate error message is Target base <base id> does not exist!
- (3) If there is no drone (or swarm) with the **`drone id`**, the adequate error message is <drone id> is not at <base id>!
- (4) If the drone (or swarm) with the **`drone id`** does not have enough range to fly to the destination, the adequate error message is Drone <drone id> cannot reach <base id>!

```
> flyToBase↵
Mojito↵
drone03↵
Sintra↵
Source base Mojito does not exist!↵
> flyToBase↵
Montijo↵
drone03↵
Cintra↵
Target base Cintra does not exist!↵
> flyToBase↵
Montijo↵
unknownDrone↵
Sintra↵
unknownDrone is not at Montijo↵
> flyToBase↵
Montijo↵
drone04↵
Sintra↵
Drone drone04 cannot reach Sintra!↵
```

2.12 `addOrder` command

Add an order to a base. The command receives 5 parameters containing the **`base id`** the **`order id`**, the delivery location (**`latitude`** and **`longitude`**) and the **`dimension`**, and puts the delivery in a pending deliveries list. Those orders will later be processed through the `delivery` command.

```
> addOrder↵
Montijo↵
Order1↵
25 3780 -948↵
Order queued for delivery.↵
```

If some parameter is incorrect, the command is aborted:

- (1) If the source base id does not correspond to any base owned by the company, the adequate error message is Source base <base id> does not exist!
- (2) If the order id already exists, the adequate error message is Order <order id> already registered!
- (3) If the order dimension is not greater than 0, the adequate error message is Order dimension must be a positive integer!

```
> addOrder↵
Mojito↵
Order1↵
25 3780 -948↵
Source base Mojito does not exist!↵
> addOrder↵
Montijo↵
Order1↵
25 3780 -948↵
Order Order1 already exists!↵
> addOrder↵
Montijo↵
Order2↵
-25 3780 -948↵
Order dimension must be a positive integer!↵
```

2.13 `orders` command

Lists all pending orders from a base. The command receives 1 parameter with the base id. It then prints a list of the pending orders from that base. In the following example, there are 3 orders to locations close to Sintra. The dimension of the package to be delivered is presented between the order id and the destination location.

```
> orders↵
Sintra↵
Order1; 25; [3780,-948]↵
Order2; 18; [3860,-928]↵
Order3; 22; [3870,-936]↵
```

If there are no pending orders in the given base, the adequate feedback message is There are no pending orders!

```
> orders↵
Sintra There are no pending orders!↵
```

If the base id parameter is incorrect, the command is aborted:

- (1) if the base does not correspond to an existing base, the adequate message is Base <base id> does not exist!

```
> orders↵
Cintra↵
Base Cintra does not exist!↵
```

2.14 `allOrders` command

Lists all pending orders from all bases. It then prints a list of the pending orders from all bases. In the following example, there are 3 orders to locations close to Sintra and none in the other 2 bases. The dimension of the package to be delivered is presented between the order id and the destination location. For bases with no pending orders, the program prints the message There are no pending orders in <base id>!

```
> allOrders↵
There are no pending orders in Montijo.↵
Orders in Sintra:↵
Order1; 25; [3780,-948]↵
Order2; 18; [3860,-928]↵
Order3; 22; [3870,-936]↵
There are no pending orders in Monte Real.↵
```

If there are no pending orders in any base, the program presents the message There are no pending orders!

```
> allOrders↵
There are no pending orders!↵
```

2.15 `deliver` command

Deliver a package to a customer. This command receives a **`base id`**, a **`drone id`**, and an **`order id`** and assigns the fulfillment of that order to that drone, starting from that base. If all goes well, the drone picks the order and flies to the customer location, which is specified in the order. In the following example, the drone will carry the second order of the list to its customer. This command only starts the delivery process, removing the drone from the hangar. The flight to the customer's location and back is simulated during the execution of the `ticTac` command.

```
> deliver↵
Sintra↵
bee01↵
order23↵
bee01 will deliver order23.↵
```

If some parameter is incorrect, the drone (or swarm) will not be assigned the order, and the corresponding message will be presented:

- (1) If the base does not correspond to an existing base, the adequate message is Base <base id> does not exist!
- (2) If there is no drone (or swarm) with the **drone id**, the adequate error message is <drone id> is not at <base id>!
- (3) If there is no order named **order id**, the adequate message is <order id> is not pending!
- (4) If the order does exist, but the distance is too far for the selected drone to go there and back with it's current range level, the adequate message is <order id> is too far for <drone id>!
- (5) If the order dimension exceeds the capacity of the drone, the adequate message is <order id> is too heavy for <drone id>!

```
> deliver↵
Cintra↵
bee01↵
order23↵
Base Cintra does not exist!↵
> deliver↵
Sintra↵
bee18↵
order23↵
Drone bee18 is not at Sintra!↵
> deliver↵
Sintra↵
bee18↵
order25↵
order25 is not pending!↵
> deliver↵
Sintra↵
bee18↵
order25↵
order25 is not pending!↵
> deliver↵
Sintra↵
bee18↵
order25↵
order25 is not pending!↵
```

```
> deliver↵
Sintra↵
bee19↵
order22↵
order22 is too far for bee19!↵
> deliver↵
Sintra↵
bee20↵
order26↵
order26 is too heavy for bee20!↵
```

2.16 `delivered` command

Lists all delivered orders. An order is considered delivered when the drone carrying it arrives back to the base. This command lists all the deliveries that were successfully concluded and always succeeds. Each delivery is presented in a line that starts with a timestamp (corresponding to the number of simulation time units since the beginning of the simulation where a particular order was marked as delivered), followed by the order id and the base from which that order was delivered. Orders are sorted by the timing in which they were satisfied (i.e., removed from the delivery queue, upon the safe return of the drone carrying it, and added to the already delivered orders list).

In the following scenario, the delivered command lists 3 successfully delivered orders:

```
> delivered↵
3 order23.↵
3 order12.↵
4 order63.↵
```

If no orders were satisfied so far, the adequate feedback is No orders delivered so far!

```
> delivered↵
No orders delivered so far!↵
```

2.17 `inTransit` command

Lists all drones (and swarms) currently flying. Presents a list of all drones flying. In each line, it presents the drone id, the origin base, the destination base, distance covered, total distance, and nature of the mission (either delivery or relocation). This command always succeeds. The distance is computed using the Pitagoras theorem and rounding up the result to the next greater integer. So, for instance, from Montijo to Sintra, we have [3870,-897] and [3880,-938], so, a difference of 10 points in latitude, and 41 points in longitude (actually computing distances with latitudes and longitudes is a bit trickier than this, but we will use this fictional approach for the purposes of our simulation).So, one could compute this distance as $ceil(\sqrt{10^2 + 41^2}) = 43$. We can also assume that the drones fly 10 distance points per time unit.

In the following example, 3 drones are currently flying (one of them is actually a swarm). Assuming the 10 distance points speed, drone02 and swarm07 would complete their flight in the next time tic, being removed from this list after that.

```
> inTransit↵
drone01 Montijo Montijo 10 35 delivery!↵
drone02 Montijo Sintra 40 43 relocation!↵
swarm07 Montijo Montijo 20 22 delivery!↵
```

If there are no drones flying, the command will generate the output feedback No drones are flying!, as in the following example:

```
> inTransit↵
No drones are flying!↵
```

2.18 `ticTac` command

Advances the simulation clock n time units. This command simulates the passage of time in this program. There are two things that occur over time in this program: drones flying and drones servicing. The duration of the flight can be computed by dividing the distance points by 10 and rounding up to the nearest integer value. For example, a flight over a 42 distance will take 5 time ticks to complete. Note that, when delivering packages, we have to double the distance from the base to the delivery point because, naturally, the drones need to fly back to their origin base. Following the same rationale, we **do not** double the distance when we move a drone from one base to another one, as the drone does not need to fly back. Servicing a drone consumes 3 time units. So, when a drone is serviced, it is moved to the service area, where it has to stay for two time ticks. As soon as two time ticks have passed, the drone is automatically put into the hangar area, with its range at the maximum capacity.

In the following example, 3 drones are currently flying (one of them is actually a swarm). The current time tick is 12. drone01 is carrying order23 and swarm07 is carrying order24. drone02 is relocating to Sintra. We will first advance 2 time ticks, and then a third one. After this, all drones have landed. We will also send a drone for servicing and check what happens to it. Service takes 3 time units and the drone is returned to the hangar after that, with its energy back to full level (in this example, the corresponding to flying for 110 time units).

```
> listBases↵
Montijo [3870,-897]↵
The hangar is empty!↵
The service bay is empty!↵
Sintra [3880,-938]↵
The hangar is empty!↵
The service bay is empty!↵
Monte Real [3951,-852]↵
The hangar is empty!↵
The service bay is empty!↵
> inTransit↵
drone01 Montijo Montijo 10 35 delivery!↵
drone02 Montijo Sintra 40 43 relocation!↵
swarm07 Montijo Montijo 20 22 delivery!↵
> delivered↵
No orders delivered so far!↵
```

```
> tictac↵
2↵
> inTransit↵
drone01 Montijo Montijo 30 35 delivery!↵
> delivered↵
13 order24.↵
> tictac↵
1↵
> delivered↵
13 order24.↵
14 order23.↵
> listBases↵
Montijo [3870,-897]↵
Hangar: [swarm07 50 120, drone01 20 100]↵
The service bay is empty!↵
Sintra [3880,-938]↵
Hangar: [drone02 20 80]↵
The service bay is empty!↵
Monte Real [3951,-852]↵
The hangar is empty!↵
The service bay is empty!↵
> service↵
Montijo↵
105↵
drone01 moved to service bay.↵
> listBases↵
Montijo [3870,-897]↵
Hangar: [swarm07 50 120]↵
Service bay: [drone01 20 100]↵
Sintra [3880,-938]↵
Hangar: [drone02 20 80]↵
The service bay is empty!↵
Monte Real [3951,-852]↵
The hangar is empty!↵
The service bay is empty!↵
> tictac↵
1↵
> listBases↵
Montijo [3870,-897]↵
Hangar: [swarm07 50 120]↵
Service bay: [drone01 20 100]↵
Sintra [3880,-938]↵
Hangar: [drone02 20 80]↵
The service bay is empty!↵
Monte Real [3951,-852]↵
The hangar is empty!↵
The service bay is empty!↵
> tictac↵
2↵
```

```
> listBases↵
Montijo [3870,-897]↵
Hangar: [swarm07 50 120, drone01 20 110]↵
The service bay is empty!↵
Sintra [3880,-938]↵
Hangar: [drone02 20 80]↵
The service bay is empty!↵
Monte Real [3951,-852]↵
The hangar is empty!↵
The service bay is empty!↵
```

3 Developing this project

Your program should take the best advantage of the elements taught up until now in the Object-Oriented programming course. You should make this application as **extensible as possible** to make it easier to add, for instance, new kinds of drones. For example, for the time being, hermits are not that different from sociable drones, except that they cannot be part of a swarm. You could be tempted to encode this with a simple boolean attribute. However, when you think about it, hermits have less design constraints, because they do not need to support for combining with other drones. One could use that extra freedom to, say, add a solar panel to the hermits, so they could extend their own battery life while flying, leading to a slower than normal depletion of their remaining range before requiring service. You should build your program so that changes like this one would require as little changes as possible to the rest of your code.

You can start by developing the main user interface of your program, clearly identifying which commands your application should support, their inputs and outputs, and preconditions. Then, you need to identify the entities required for implementing this system. Carefully specify the **interfaces** and **classes** that you will need. You should document their conception and development using a class diagram, as well as documenting your code adequately, with Javadoc.

It is a good idea to build a skeleton of your `Main` class, to handle data input and output, supporting the interaction with your program. In an early stage, your program will not really do much. Remember the **stable version rule**: do not try to do everything at the same time. Build your program incrementally, and test the small increments as you build the new functionalities in your new system. If necessary, create small testing programs to test your classes and interfaces.

Have a careful look at the test files, when they become available. You should start with a really bare bones system with the `help` and `exit` commands, which are good enough for checking whether your commands interpreter is working well, to begin with. Then, you add bases, and the base listing operation, so that you can create bases and then check they are ok. So far, they are empty. The next step is to start creating atomic drones, and adding them to bases. And so on. Step by step, you will incrementally add functionalities to your program and test them. **Do not try to make all functionalities at the same time. It is a really bad idea.**

In this project you will handle several collections. These offer you several opportunities for reusing code, rather than repeating it over and over again. The same holds for iterators. In this particular project, it happens that when iterating over collections, the elements will always be organized in order of introduction in the corresponding collection. So, for example, the notion of collection of drones is likely useful.

Last, but not the least, **do not underestimate the effort for this project**.

4 Submission to Mooshak

To submit your project to mooshak, please register in the mooshak contest POO1819-TP1 and follow the instructions that will be made available on the moodle course website.

4.1 Command syntax

For each command, the program will only produce one output. The error conditions of each command have to be checked in the exact same order as described in this document. If one of those conditions occurs, you do not need to check for the other ones, as you only present the feedback message corresponding to the first failing condition. For example, if you attempt to create a drone with an invalid range, the remaining conditions do not need to be checked. However, the program does need to consume all the remaining input parameters, even if they are to be discarded.

4.2 Tests

The Mooshak tests verify incrementally the implementation of the commands. They will be be made publicly available on April 2. When the sample test files become available, use them to test what you already have implemented, fix it if necessary, and start submitting your partial project to mooshak. Do it from the start, even you just with the exit and help commands. By then you will probably have more than those to test, anyway. Good luck!