Gossip!

Object-Oriented Programming
1st Project, version 1.2 – 2023-04-17
Contact: mgoul@fct.unl.pt

`https://discord.gg/8JzBwRqwRv`
`Texto adicionado é apresentado a vermelho`
`Mensagens de comandos modificadas são apresentadas a verde`

Important remarks

Deadline until 23h55 (Lisbon time) of April 30, 2023.

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 website 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 on both interfaces and classes. The documentation in classes can, of course, refer to the one in interfaces, where appropriate. Please comment on the methods explaining what they mean and defining preconditions for their usage. Students may and should discuss any doubts with the teaching team, and 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 website.

1 Development of the application *Gossip*

1.1 Problem description

The goal of this project is to develop an application that tracks the spreading of gossip within a community. With this application, we can keep track of who knows what about whom, within a community. This works as sort of a Big Brother system. All buildings in town are full of cameras and microphones. These detect who meets who, and where. The town mayor wants this because she needs to figure out who created that miserable gossip concerning the authorship of her famous apple pie. Rumour has it is actually her husband who cooks it. Shocking. It is not[1]. But she needs to know who is spreading infamous lies in her community, right? Let's build it for her! No more privacy in this community. Yey!

In this project, we will create the first prototype for the system. A community has people and landmarks. Landmarks are places where people meet. At any given moment, a person is, at most, in only one landmark. When not in any landmark, the person is assumed to be home, but there are no cameras or microphones there, so who cares? In practice, both landmarks and home are places, but, as far as the system can tell, people can only meet other people at

[1] It is the butler, who else?

landmarks. Fun fact... these people just don't talk to each other at home. Great. No gossip sharing there! You see, the surveillance system allows the detection of groups of people in a landmark. And it is within these groups gossip spreads. So, a person within a group may share her/his gossips with all the people in that group at that moment. When a person arrives at a landmark, the person is alone, until (s)he joins someone, or someone joins that person. People can move from one group to another within the landmark, by joining a person within that group. A person can also leave a group, becoming alone within that landmark, even if there are a lot of people around. When a person leaves a landmark, it also leaves the group of people in that landmark. There are three kinds of people: *forgetful* people, *gossipers*, and *sealed lips*:

- *Forgetful* people love to gossip but have a really poor memory. They can only remember the last n gossips they heard (the value of n varies from person to person). Suppose n is *5*. If the forgetful person hears a sixth gossip, they forget the first one, if they hear a seventh gossip, they forget the second one, and so on. They are moderate gossipers. They only share a gossip at a time, within a group. They are also kind of boring. If they switch groups, they start sharing their oldest gossip with the next group.
- *Gossipers* **never** forget a gossip, and they share up to three gossips whenever they gossip. Also, they go around all their gossips before they repeat themselves.
- *Sealed lips* are people who can hear a gossip and do remember it, but never spread it, unless the gossip is about themselves. They only share a gossip a time, and when they run out of gossips they go back and repeat the same ones.

A gossip has a description, the people who are involved in the gossip, and the sequence of people who tried spreading the gossip.

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.

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 `someRandomCommand` would have the following effect:

```
someRandomCommand↵
Unknown command. Type help to see available commands.↵
```

If there are additional tokens in the line (e.g. parameter for the command you were trying to write, the program will try to consume them as commands, as well. So, in this example, `someRandom Command` would be interpreted as two unknown commands: `someRandom` and `Command`, leading to two error messages.

```
someRandom Command↵
Unknown command. Type help to see available commands.↵
Unknown command. Type help to see available commands.↵
```

Several commands have arguments. Unless explicitly stated in this document, 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. Also, for any String arguments in the commands, assume they are case-sensitive. So, "Coffee Shop" and "coffee shop" would be two different landmarks, for instance.

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.

2.2 `help` command

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

```
help↵
landmark - adds a new landmark to the community↵
landmarks - displays the list of landmarks in the community↵
forgetful - adds a forgetful person to the community↵
gossiper - adds a gossiper person to the community↵
sealed - adds a sealed lips person to the community↵
people - lists all the persons in the community↵
go - moves a person to a landmark, or home↵
join - joins a person to a group↵
groups - lists the groups composition in a landmark↵
isolate - makes a person leave the current group, but not the landmark the person is currently on↵
start - starts a new gossip↵
gossip - share a gossip within the current group in the current landmark↵
secrets - lists the gossip about a particular person↵
infotainment - lists the gossips a particular person is aware of↵
hottest - lists the most shared gossip↵
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 `landmark` command

Adds a new landmark to the community. The command receives as arguments the **`capacity`** of the new landmark and the **`name`** of the landmark. When successful, the program will output the feedback message(<landmark name> added.). In the example, we create a new landmark called **`Coffee shop`** with a capacity of up to **`20`** people at any given time. There is no upper limit to the number of landmarks one can create in a community.

```
landmark 20 Coffee shop↵
Coffee shop added.↵
```

The following errors may occur:

1. If the `landmark capacity` is less or equal to `0`, the error message is (Invalid landmark capacity <n>!).
2. If the landmark **`name`** is "home" this is not acceptable, as "home" is a reserved name for that place people go when they are not in any landmark. The error message is (Cannot create a landmark called home. You know, there is no place like home!).
3. If the landmark **`name`** already exists in the community, the error message is (Landmark <name> already exists!

The following example illustrates these error messages.

```
landmark -200 Library↵
Invalid landmark capacity -200!↵
landmark 3 home↵
Cannot create a landmark called home. You know, there is no place like home!↵
landmark 40 Coffee shop↵
Landmark Coffee shop already exists!↵
```

2.4 `landmarks` command

Displays the list of landmarks in the community. The command does not receive any arguments and always succeeds. If there are no landmarks to list, the program will output the feedback message(This community does not have any landmarks yet!). Otherwise, it lists all landmarks, in order of insertion, each line presenting the landmark name, capacity and current occupation, with the format (<landmark name>: <landmark capacity> <current occupation>.). In the example, there are three landmarks available in the community.

```
landmarks↵
Coffee shop: 20 12.↵
Library: 500 127.↵
Garden: 45 14.↵
```

2.5 `forgetful` command

Adds a forgetful person to the community. The command receives as arguments the **`number of gossips`** the forgetful person can remember and the **`name`** of the person. When successful, the program will output the feedback message(<name> can only remember up to

<n> gossips.). In the example, we create a new forgetful person called **Dory** with a capacity to remember up to **3** gossips.

```
forgetful 3 Dory↵
Dory can only remember up to 3 gossips.↵
```

The following errors may occur:

1. If the `gossips capacity` is less or equal to `0`, the error message is (Invalid gossips capacity <n>!).
2. If the forgetful person's **name** already exists in the community, the error message is (<name> already exists!).

The following example illustrates these error messages, illustrating an illegal number of memory capacity and a repeated person in the community, respectively.

```
forgetful -1 Jason Bourne↵
Invalid gossips capacity -1!↵
forgetful 50000 Sheldon Cooper↵
Sheldon Cooper already exists!↵
```

2.6 `gossiper` command

Adds a gossiper person to the community. The command receives as arguments the gossiper person's **name**. When successful, the program will output the feedback message(<name> is a gossiper.). In the example, we create a new gossiper person called **Nate Archibald**.

```
gossiper Nate Archibald↵
Nate Archibald is a gossiper.↵
```

The following errors may occur:

1. If the gossiper person's **name** already exists in the community, the error message is (<name> already exists!).

The following example illustrates this error message, illustrating a repeated person in the community.

```
gossiper Lady Whistledown↵
Lady Whistledown already exists!↵
```

2.7 `sealed` command

Adds a sealed lips person to the community. The command receives as arguments the sealed lips person's **name**. When successful, the program will output the feedback message(<name> lips are sealed.). In the example, we create a new sealed lips person called **Judy Hale**.

```
sealed Judy Hale↵
Judy Hale lips are sealed.↵
```

The following errors may occur:

1. If the sealed lips person's **name** already exists in the community, the error message is (<name> already exists!).

The following example illustrates this error message, illustrating a repeated person in the community.

```
sealed Liza Miller↵
Liza Miller already exists!↵
```

2.8 `people` command

Lists all the persons in the community. The command does not receive any arguments and always succeeds. If there are no people to list, the program will output the feedback message(This community does not have any people yet!). Otherwise, it lists all people, in order of insertion, each line presenting the person's name, current location and count of known gossips, with the format (<person name> at <landmark> knows <number of gossips> gossips.). In the example, there are three people in the community.

```
people↵
Amnesia Louie Scallop at Golf Course knows 3 gossips.↵
Sheldon Cooper at University knows 21 gossips.↵
Self-Centered Christine knows 45 gossips.↵
```

2.9 `go` command

Moves a person to a landmark, or home. The command receives as arguments the person's **name** and the **place** the person is moving to. This can either be a landmark or home. When successful, the program will output the feedback message(<name> is now at <place>.). In the example, **Frankie** goes to **Hollywood**. Note that when a person moves to a landmark a new group is created with only that person. This also implies leaving the last group the person was in. If the person was the only member in that group, the group disappears.

```
go Frankie↵
Hollywood↵
Frankie is now at Hollywood.↵
```

The following errors may occur:

1. If the person's **name** does not exist in the community, the error message is (<name> does not exist!).

2. If the **place** the person is going is not a known landmark in this community, or "home", the error message is (Unknown place <place>!).

3. If the person decides to go to the place the person is already in, this command has no effect but returns the error message (What do you mean go to <place>? <name> is already there!).

4. If the landmark is already full (with as many people there as its max capacity), the person goes home and the error message is (<landmark> is too crowded! <name> went home.).

The following example illustrates these error messages.

```
go Santa Claus↵
Shopping Mall↵
Santa Claus does not exist!↵
go Batman↵
Arkham Asylum↵
Unknown place Arkham Asylum!↵
go Phil Connors↵
Punxsutawney Park↵
What do you mean go to Punxsutawney Park? Phil Connors is already there!↵
go James Blunt↵
Subway↵
Subway is too crowded! James Blunt went home.↵
```

2.10 `join` command

Joins a person to a group. The command receives as arguments the person's **`name`** and the name of another person in the same landmark. The first person then leaves whatever group the person is in, and joins the group of the second person. When successful, the program displays a message(<name> joined <comma separated list of people>, at <landmark>.). In the example, **Belinda** joins **Bella**, who was already with Giselle and Jim, in a group started by Giselle, joined subsequently by **Bella**, Jim, and now **Belinda**. If Belinda was alone in her previous group, that group is eliminated.

```
join Belinda↵
Bella↵
Belinda joined Giselle, Bella, Jim, at the Royal Albert Hall.↵
```

The following errors may occur:

1. If the first and the second person are the same, the error message is (<name> needs company from someone else!).

2. If the first person's **`name`** does not exist in the community, the error message is (<name> does not exist!).

3. If the second person's **`name`** does not exist in the community, the error message is (<name> does not exist!).

4. If the first person is at home, rather than in a landmark, the message is (<name> is at home!).

5. If the second person is not in the same landmark, the error message is (<name> is not in <landmark>!). Note this also covers the case where the second person is at home.

6. If the first person and the second person are already in the same group, the error message is (<name> and <name> are already in the same group!).

The following example illustrates these error messages.

```
join Chuck Noland↵
Chuck Noland↵
Chuck Noland needs company from someone else!↵
join Santa Claus↵
Rudolph the Reindeer↵
Santa Claus does not exist!↵
join Rudolph the Reindeer↵
Santa Claus↵
Santa Claus does not exist!↵
join Kevin McCallister↵
Kevin McCallister is at home!↵
join James Blunt↵
Beautiful woman who smiled at James but was with another man↵
Beautiful woman who smiled at James but was with another man is not in Subway!↵
join Statler↵
Waldorf↵
Statler and Waldorf are already in the same group!↵
```

2.11 `groups` command

Lists the groups composition in a landmark. The command receives as argument the **landmark**. It prints a header (<number of groups> groups at <landmark>:) and then lists all the groups in that landmark, in order of creation. A group is created whenever a person goes to the landmark. The newly created group has only that person. Groups are very volatile. When a person who is alone joins a group, the former single-person group disappears. In the example, the oldest group has only Mr. Hauser, who is alone, while other people are in two other groups. If Mr. Hauser were to join Jonathan Byers, there would be only two groups left (and Mr. Hauser would be the third element in Nancy and Johnathan's group). If then Mr. Hauser left Nancy and Jonathan, he would form a new group with a single element (himself), but now this would be the third group on the list.

```
groups High School↵
3 groups at High School:↵
Mr. Hauser↵
Jane Hopper, Mike Wheeler, Dustin Henderson, Lucas Sinclair↵
Nancy Wheeler, Jonathan Byers↵
```

The following errors may occur:

1. If the question is asked about groups at "home", the error message is (You must understand we have no surveillance tech at home! Privacy is our top concern!).
2. If the landmark does not exist, the error message is (<landmark> does not exist!).
3. If nobody is in the landmark, the error message is (Nobody is at <landmark>!).

The following example illustrates these error messages.

```
groups home↵
You must understand we have no surveillance tech at home! Privacy is our top concern!↵
groups Arkham Asylum↵
Arkham Asylum does not exist!↵
groups Piranha Lake↵
Nobody is at Piranha Lake!↵
```

2.12 `isolate` command

Makes a person leave the current group, but not the landmark the person is currently on. The command receives as arguments the person's **`name`**. When successful, the person leaves the current group and creates a new one, on the same landmark, but where that person is, for the time being, alone. The program will output the feedback message(<name> is now alone at <landmark>.). In the example, **`Mr. Hauser`** leaves a group and becomes alone.

```
isolate Mr. Hauser↵
Mr. Hauser is now alone at High School.↵
```

The following errors may occur:

1. If the person's **`name`** does not exist in the community, the error message is (<name> does not exist!).
2. If the person is at home the error message is (<name> is at home!).
3. If the person is already alone, the command does nothing and returns the error message (<name> is already alone!).

The following example illustrates these error messages.

```
isolate Santa Claus↵
Santa Claus does not exist!↵
isolate Kevin McCallister↵
Kevin McCallister is at home!↵
isolate Chuck Noland↵
Chuck Noland is already alone!↵
```

2.13 `start` command

Starts a new gossip. The command receives as arguments the gossip creator's **`name`**, followed by an integer **`n`** representing the number of people who are targets of that gossip, the names of those people, one per line, followed by the gossip itself. When successful, the program will output the feedback message(Have you heard about <comma-separated list of gossip targets>? <gossip>). In the example, **`Romeo`** and **`Juliet`** are allegedly dating, or something.

```
start William Shakespeare↵
2↵
Romeo Montague↵
Juliet Capulet↵
Romeo and Juliet, sitting on a tree, K I S S I N G! :-)↵
Have you heard about Romeo Montague, Juliet Capulet? Romeo and Juliet, sitting on a tree, K I S
S I N G! :-)↵
```

The following errors may occur:

1. If the person's **name** does not exist in the community, the error message is (<name> does not exist!).
2. If the number **n** of gossip targets is less or equal to 0, the error message is (Invalid number <n> of gossip targets!).
3. If any of the targets of the gossip does not exist, the error message is (<name> does not exist!) The name to be presented is the one for the first target that does not exist.
4. If there is already a similar gossip, with the same creator, involved people, and description, the error message is (Duplicate gossip!).

The following example illustrates these error messages.

```
start Santa Claus↵
2↵
Rudolph the Reindeer↵
Santa Little Helper↵
Santa and Rudolph are secretly negotiating contracts with the Easter
Bunny↵
Santa Claus does not exist!↵
start Rudolph the Reindeer↵
0↵
Santa drinks Pepsi, when nobody is watching!↵
Invalid number 0 of gossip targets!↵
start Rudolph the Reindeer↵
1↵
Santa Claus↵
Santa drinks Pepsi, when nobody is watching!↵
Santa Claus does not exist!↵
```

2.14 `gossip` command

Share a gossip within the current group in the current landmark. The command receives as arguments the person's **name**. The person gossips with whoever is in the same group. Note that which, and what gossips get to be shared depends on the person who is doing the gossip. For instance, a forgetful person will only share one gossip. The same goes for a sealed-lips person, but the latter will only share one gossip involving her/him. A gossiper will share up to 3 secrets (less, if the person does not have 3 gossips to share). Note that gossips will be shared even if the target of the gossip is in the group learning about the gossip. People in this

community don't just gossip behind your back. They gossip in your face, too. When successful, the program will output the feedback message(<name> shared with <comma-separated group members list>, some hot news!↵<gossips>). Note that <gossips> is a list of gossips, one per line, just with the gossip itself. The following scenario illustrates this command. Judy tells her friends she is up all night.

```
gossip Judy Hale↵
Judy Hale shared with Jen, Charlie, Ana, some hot news!↵
I am up all night.↵
```

The following errors may occur:

1. If the person's **name** does not exist in the community, the error message is (<name> does not exist!).
2. If the person is alone, either in a landmark, or at home, the error message is (<name> has nobody to gossip with right now!).
3. If the person has no gossips to share, the error message is (<name> knows nothing!).
4. Some people might be aware of gossips, but not willing to share them. If the person happens to be a "sealed lips" person, that person might actually know of a few gossips, but if all of them are about someone else, the person would not be willing to share any of them. In that case, the error message is (<name> does not wish to gossip right now!).

The following example illustrates these error messages.

```
gossip Santa Claus↵
Santa Claus does not exist!↵
gossip Kevin McAllister↵
Kevin McAllister has nobody to gossip with right now!↵
gossip John Snow↵
John Snow knows nothing!↵
gossip Judy Hale↵
Judy Hale does not wish to gossip right now!↵
```

2.15 `secrets` command

Lists the gossip about a particular person. The command receives as arguments the person's **name**. When successful, the program will print a header line (Here is what we know about <name>:), followed by a list of all existing gossip about that person, one per line. Each line in the output consists simply of the number of people who are aware of the gossip, followed by the gossip itself (<n> <gossip>). Note that gossips may eventually forgotten if all the people who knew about them have forgotten them. Completely forgotten gossips are discarded from the system, so <n> is always greater than 0. In the example, we learn about the two secrets concerning Cynthia Purley.

```
secrets Cynthia Purley↵
Here is what we know about Cynthia Purley:↵
4 Cynthia does not really like Monica.↵
1 Cynthia gave her baby for adoption many years ago.↵
```

The following errors may occur:

1. If the person's **name** does not exist in the community, the error message is (<name> does not exist!).
2. If the person has no secrets, the error message is (<name> lives a very boring life!).

The following example illustrates these error messages.

```
secrets Santa Claus↵
Santa Claus does not exist!↵
secrets Batman↵
Batman lives a very boring life!↵
```

2.16 `infotainment` command

Lists the gossips a particular person is aware of. The command receives as arguments the person's name. These may include gossips the person is aware of, but not willing to gossip about. When successful, the program will output the feedback message(<name> knows things:), followed by all the gossips that person knows, one per line. The following example illustrates this:

```
infotainment Cindy Adams↵
Cindy Adams knows things:↵
A famous public figure who shall remain nameless prefers Yorkshire Terriers to people. They are really cool dogs. Only in New York, kids, only in New York.↵
Just saw an orange old man eating a burger. Only in New York, kids, only in New York.↵
Sarah Jessica Parker and Matthew Broderick are selling their East Village home for 22 million dollars. Only in New York, kids, only in New York.↵
```

The following errors may occur:

1. If the person's **name** does not exist in the community, the error message is (<name> does not exist!).
2. If the person knows nothing, the error message is (<person> knows nothing!).

The following example illustrates these error messages.

```
infotainment Santa Claus↵
Santa Claus does not exist!↵
infotainment John Snow↵
John Snow knows nothing!↵
```

2.17 `hottest` command

Lists the most shared gossip. The command receives no arguments. When successful, the program will output the feedback message(The hottest gossips were shared <n> times!↵) followed by one or more gossips (as many as are tied as the most shared ones). Note this value is independent of the number of people who listen to the gossip. Sharing a gossip with one

person, or with 10 at the same time, will contribute with 1 to the "hotness" of a gossip. In the example, there are two top gossips. Whenever there is a tie, the gossip which reached the top mark first is shown first. Note that if everybody forgets about a particular gossip, it can no longer be considered one of the hottest gossips, as nobody even remembers it ever existed. In other words, gossips only "exist" in the system if at least one person remembers them.

```
hottest↵
The hottest gossips were shared 245 times!↵
These are old news about you know who.↵
These are fresh news about you know who.↵
```

The following errors may occur:

1. There may be no known gossips. Note that, for all practical matters, "completely forgotten gossips" are also considered unknown. In this case, the error message is (No gossips we are aware of!).

2. There may be no shared gossips. This means that even if gossips were created, they were not shared yet. As this command is about shared gossips, not shared gossips are gossips shared 0 times. The adequate error message is (No gossips were shared, yet!).

The following example illustrates these error message.

```
hottest↵
No gossips we are aware of!↵
start William Shakespeare↵
2↵
Romeo Montague↵
Juliet Capulet↵
Romeo and Juliet, sitting on a tree, K I S S I N G! :-)↵
Have you heard about Romeo Montague, Juliet Capulet? Romeo and Juliet, sitting on a tree, K I S
S I N G! :-)↵
hottest↵
No gossips were shared, yet!↵
```

3 Developing this project

Your program should take the best advantage of the elements taught 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 people.

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 error conditions. 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 document your code adequately, with Javadoc. You can and should reuse the package on generic arrays presented in the lectures and labs (you can find a link to source code on Moodle). You cannot change these classes and interfaces. If you need some slightly different functionality, you can either create classes that use those Arrays and iterators as instance variables and add, or constrain, the functionalities you

need for your collections on those so-called wrapper classes. You can also consider extending those classes and interfaces. But do not change them. It is, in general, not a good idea to change library classes and interfaces. Also, please note that you cannot use the standard Java Collection classes in this project. Although they will be introduced in classes while you are still working on this project, we have designed this challenge for you to solve it without those classes. We will have ample opportunity of working with the standard Java Collection classes on the second project.

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, add landmarks and test their creation. Then, move on to adding the different kinds of people, one by one. Check they are ok. Then make people join and leave landmarks. Then, make them join and leave groups within landmarks. And so on. Step by step, you will incrementally add functionalities to your program and test them. **Do not try to make all functionalities simultaneously. It is a really bad idea.**

Last, but not 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 POO2023-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. However, the program does need to consume all the remaining input parameters, even if they are to be discarded.

4.2 Tests

We expect you to create your own tests, based on this specification. Create tests for the "happy day scenario", and also for error situations, so that whenever you add support for a new command, you also create suitable tests for it. So, build your tests as you build your project, incrementally. This mimics what happens with the Mooshak tests. The Mooshak tests verify incrementally the implementation of the commands. They will be made publicly available on April 21, 2023. 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 if you just implemented the exit and help commands. By then you will probably have more than those to test, anyway. Good luck!