Evaluations Calendar

Object Oriented Programming
1st Project, version 1.0 – 2022-04-05
Contact: mgoul@fct.unl.pt

Important remarks

Deadline until 23h55 (Lisbon time) of May 2, 2022.

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 *Evaluations Calendar*

1.1 Problem description

The goal of this project is to develop an application that supports an evaluation calendar for the Pedagogic Committee of your degree. With this application, committee members can create a calendar for all the evaluations scheduled during a semester, supporting the detection of conflicts between evaluations dates, to help building a more balanced evaluation calendar during the semester. The application will support data on the students, the professors, the courses, the enrolment of students in courses, the assignment of professors to those courses and the evaluations in each of those courses. We will consider two kinds of evaluations: tests and project deadlines. The application will support the detection of conflicting dates for tests, to avoid students having two or more tests in the same time slot. The application will also support monitoring how busy the evaluations calendar is for any given user, as well as detecting the most stressful moments in the calendar, where there is a higher concentration of evaluation events.

Each person in the system has a unique name and is associated to the courses the person participates in, either in the role of professor or the role of student. Students also have a unique id number. Each course has a unique course name, a set of professors assigned to it, and a set of students enrolled in it. The course also has a set of evaluations. These include tests and project deadlines. All professors and students participating in a course are automatically associated to

all evaluation events of that course. In other words, a student is assumed to participate in all tests of all courses the student is enrolled in, and to have all the deadlines defined for those courses, as well.

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 in this document.

Commands are case insensitive strings with no blank spaces inside. 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 COMMAND. Type help to see available commands.). For example, the non existing command `someRandomCommand` would have the following effect:

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

If there are additional tokens in the line (e.g. a 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 SOMERANDOM. Type help to see available commands.↵
Unknown command 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, in the correct order. 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 always succeeds. When executed, it terminates the program execution. This command does not require any arguments. It outputs the message (Bye!) before terminating the program. The following scenario illustrates its usage.

```
exit↵
Bye!↵
```

2.2 `help` command

Shows the available commands. This command does not require any arguments and always succeeds. When executed, it shows the available commands. The following scenario illustrates its usage.

```
help↵
Available commands:↵
people - lists all people↵
professor - adds a new professor↵
student - adds a new student↵
courses - lists all courses↵
course - adds a new course↵
roster - lists the professors and students of a course↵
assign - adds a teacher to a course↵
enrol - adds students to a course↵
intersection - lists all the people involved in all the given courses↵
coursedeadlines - lists all deadlines in a given course↵
personaldeadlines - lists all the deadlines of a given person↵
deadline - adds a new deadline↵
coursetests - lists all tests in a given course↵
personaltests - lists all tests for a given student↵
schedule - add a new test to a course↵
superprofessor - presents the professor with more students↵
stressometer - presents the students with the top N stressful sequences of evaluations↵
help - shows the available commands↵
exit - terminates the execution of the program↵
```

2.3 `people` command

Lists all people (professors and students) in the system. This command does not receive any arguments and always succeeds. If the database is empty, the output is (No people registered!). Otherwise, it will present summary information about all the registered people, in their order of registration, one by line, with the following format:

1. If the person is a student, the format is ([<id>] <name> (<courses>)), representing the student's number, name and the number of courses the student is enrolled in.

2. If the person is a professor, the format is (<name> (<courses>)), representing the professor's name and number of courses the professor lectures.

The following scenario illustrates its usage.

```
people↵
No people registered!↵
... a few students and professors added...
people↵
[34242] Tom Sawyer (0)↵
[34234] Huckleberry Finn (0)↵
Mr. Walters (0)↵
[36423] Becky Thatcher (0)↵
Mother Hopkins (0)↵
Mr. Dobbins (0)↵
[12342] Joe Harper (0)↵
```

2.4 `professor` command

Adds a new professor to the system. The command receives as argument:

- the **name**, which can be composed of one or more tokens.

The expected behaviour is as follows: If there is no registered person with that **name**, the professor is added and the message (<name> added.) is written as feedback. The following scenario illustrates its usage.

```
professor Mr. Dobbins↵
Mr. Dobbins added.↵
```

The following error may occur:

1. If the **name** already exists, the adequate error message is (<name> already exists!). Note this can occur even if there is a student with the same name. The name must be unique.

```
professor Mr.  Dobbins↵
Mr. Dobbins already exists!↵
professor Tom Sawyer↵
Tom Sawyer already exists!↵
```

2.5 `student` command

Adds a new student to the system. The command receives as arguments:

- the **student number**, which is an integer.
- the **name**, which can be composed of one or more tokens.

The expected behaviour is as follows: If there is no registered person with that **name**, the student is added and the message (<name> added.) is written as feedback. The following scenario illustrates its usage.

```
student 34242 Tom Sawyer↵
Tom Sawyer added.↵
```

The following errors may occur:

1. If the `student number` already exists, the adequate error message is (There is already a student with the number <student number>!).

2. If the **`name`** already exists, the adequate error message is (<name> already exists!). Note this can occur even if there is a professor with the same name. The name must be unique.

```
student 34242 Tom Sawyer↵
There is already a student with the number 34242!↵
student 34242 Sid Sawyer↵
There is already a student with the number 34242!↵
student 27446 Tom Sawyer↵
Tom Sawyer already exists!↵
student 16661 Mr.  Dobbins↵
Mr. Dobbins already exists!↵
```

2.6 `courses` command

Lists all courses in the system. This command does not receive any arguments and always succeeds. The expected behaviour is as follows: If there are no registered courses, the output is (No courses registered!). Otherwise, it will present summary information about all the registered courses, in their order of registration, one by line, with the format (<course name>: <professors> professors, <students> students, <tests> tests and <deadlines> deadlines.) representing the course name, number of professors lecturing it, number of enrolled students in the course, number of scheduled tests and deadlines, respectively. The following example illustrates this.

```
courses↵
No courses registered!↵
... a few courses added with no professors, students, tests or deadlines yet...
courses↵
Object-Oriented Programming: 0 professors, 0 students, 0 tests and 0 deadlines.↵
Calculus II: 0 professors, 0 students, 0 tests and 0 deadlines.↵
Discrete Maths: 0 professors, 0 students, 0 tests and 0 deadlines.↵
Physics: 0 professors, 0 students, 0 tests and 0 deadlines.↵
```

2.7 `course` command

Adds a new course to the system. The command receives as argument:

- the **`course name`**, which can be composed of one or more tokens.

The expected behaviour is as follows: If there is no course registered with the same **`course name`**, the course is created and added to the system. The system returns the message (<course name> added.) as feedback. The following example illustrates this.

```
course Object-Oriented Programming↵
Course Object-Oriented Programming added.↵
```

The following error may occur:

1. If the **`course name`** already exists, the adequate error message is (Course <course name> already exists!).

```
course Object-Oriented Programming↵
Course Object-Oriented Programming already exists!↵
```

2.8 `roster` command

Lists all the professors and students of a course. First, it lists the professors, and then it lists the students. The command receives as argument:

- the **course name**, which can be composed of one or more tokens.

The expected behaviour is as follows: If there is a course registered with the same **course name**, the command prints the lists of professors and students. Each professor and student is presented in a different line. Professors and students are presented in order of association to the course. First, all the professors. Then, all the students. The list of professors starts with the header (Professors:), followed by the professors, with the format (<name>). The list of students starts with the header (Students:), and is followed by the students, with the format (<Id> <name>). The following example illustrates this.

```
roster Object-Oriented Programming↵
Professors:↵
Mr. Dobbins↵
Mr. Walters↵
Students:↵
34242 Tom Sawyer↵
36423 Becky Thatcher↵
12342 Joe Harper↵
34324 Huckleberry Finn↵
38876 Sid Sawyer↵
```

The following error may occur:

1. If the **course name** does not exist, the adequate error message is (Course <course name> does not exist!).

```
course Wine tasting↵
Course Wine tasting does not exist!↵
```

2.9 `assign` command

Adds a professor to a course. The command receives as arguments:

- the **professor name**, which can be composed of one or more tokens.
- the **course name**, which can be composed of one or more tokens.

The expected behaviour is as follows: If there is a professor with **professor name**, a course with **course name** and the professor is not yet assigned to the course, the professor is assigned to the course and a message with the format (Professor <professor name> assigned to <course name>.). The following example illustrates this.

```
roster Object-Oriented Programming↵
Professors:↵
Mr. Dobbins↵
Students:↵
34242 Tom Sawyer↵
36423 Becky Thatcher↵
12342 Joe Harper↵
assign Mr. Walters↵
Object-Oriented Programming↵
Professor Mr. Walters assigned to Object-Oriented Programming.↵
roster Object-Oriented Programming↵
Professors:↵
Mr. Dobbins↵
Mr. Walters↵
Students:↵
34242 Tom Sawyer↵
36423 Becky Thatcher↵
12342 Joe Harper↵
```

The following errors may occur:

1. If the **professor name** does not exist, the adequate error message is (Professor <professor name> does not exist!).
2. If the **course name** does not exist, the adequate error message is (Course <course name> does not exist!).
3. It the professor **professor name** is already assigned to course **course name**, the adequate message is (Professor <professor name> is already assigned to course <course name>!).

```
assign Mr. Darcy↵
Object-Oriented Programming↵
Professor Mr. Darcy does not exist!↵
assign Mr. Dobbins↵
Wine tasting↵
Course Wine tasting does not exist!↵
assign Mr. Dobbins↵
Object-Oriented Programming↵
Professor Mr. Dobbins is already assigned to course Object-Oriented Programming!↵
```

2.10 `enrol` command

Adds one or more students to a course. The command receives as arguments:

- the **number of students** to enrol;
- the **course name**, which can be composed of one or more tokens;
- the **student names**, which can be composed of one or more tokens, and are inserted one per line.

The expected behaviour is as follows: The user will insert **number of students** student names to enrol in the course **course name**. This will add all the inserted students who are not yet enrolled in the course. Students who are already attending the course will not be added. The system outputs a feedback message with the format (<successfully enrolled students count> students added to course <course name>.). The following example illustrates this.

```
enrol 3 Object-Oriented Programming↵
Tom Sawyer↵
Becky Thatcher↵
Joe Harper↵
3 students added to course Object-Oriented Programming.↵
roster Object-Oriented Programming↵
Professors:↵
Students:↵
34242 Tom Sawyer↵
36423 Becky Thatcher↵
12342 Joe Harper↵
```

The following errors may occur:

1. If the **number of students** to add is less than or equal to 0, the adequate error message is (Inadequate number of students!);
2. If the **course name** does not exist, the adequate error message is (Course <course name> does not exist!);
3. If the **student name** does not exist, the adequate error message is (Student <student name> does not exist!);
4. It the student **student name** is already assigned to course **course name**, the adequate message is (Student <student name> is already enrolled in course <course name>!).

```
enrol -2 Object-Oriented Programming↵
Inadequate number of students!↵
enrol 0 Object-Oriented Programming↵
Inadequate number of students!↵
enrol 2 Wine tasting↵
Tom Sawyer↵
Becky Thatcher↵
Course Wine tasting does not exist!↵
enrol 2 Object-Oriented Programming↵
Tom Sawyer↵
Darth Vader↵
Student Darth Vader does not exist!↵
1 students added to course Object-Oriented Programming.↵
enrol 1 Object-Oriented Programming↵
Tom Sawyer↵
Student Tom Sawyer is already enrolled in course Object-Oriented Programming!↵
```

2.11 `intersection` command

Returns the list of teachers and students that are enrolled in the provided courses. This command receives as arguments:

- the **number of courses** for which we want to know the intersection;
- the **course names**, which can be composed of one or more tokens, and are inserted one per line.

The expected behaviour is as follows: The user will insert the **number of courses** for which we need the intersection, followed by the **course names**. The program will then list all the professors and students which participate, simultaneously, in all the intersected courses, using a similar format to the one in the `roster` command. Professors and students are listed in order of insertion in the first course (so, the order of assignment, for professors, and of enrolment, for students) for which we are computing the intersection. If the intersection is empty, the program will display the message (No professors or students to list!).

The following example illustrates this.

```
intersection 3 Object-Oriented Programming↵
English Literature↵
Physics↵
No professors or students to list!↵
intersection 2 Physics↵
Chemistry↵
Professors:↵
Students:↵
34242 Tom Sawyer↵
36423 Becky Thatcher↵
```

The following errors may occur:

1. The number of courses is invalid. We need at least 2 courses to compute their intersection. The adequate error message is (Inadequate number of courses!).
2. If one of the **courses** names does not exist, the adequate error message is (Course <course name> does not exist!). If more than one course does not exist, this message will be applied to the first course that does not exist.

```
intersection 1 Introduction to Programming↵
Inadequate number of courses!↵
intersection 0↵
Inadequate number of courses!↵
intersection -1↵
Inadequate number of courses!↵
intersection 2 Beer tasting 101↵
Card games from the 20th Century↵
Course Beer tasting 101 does not exist!↵
```

2.12 `coursedeadlines` command

Returns the list of the deadlines defined for a given course. This command receives as arguments:

- the **course name**, which can be composed of one or more tokens.

The expected behaviour is as follows: The user will insert the **course name**. The program then lists all the deadlines available for that course, one per line, sorted by ascending order of date and, in case of ties, by ascending alphabetic order of deadline name, with the format (<deadline name>: <date>). If the course has no deadlines, the adequate message is (No deadlines defined for <course name>!). The following example illustrates this.

```
coursedeadlines Object-Oriented Programming↵
First project: 2022-05-02↵
Second project: 2022-06-03↵
coursedeadlines Advanced Procastination↵
No deadlines defined for Advanced Procastination↵
```

The following errors may occur:

1. If the course name does not exist, the adequate error message is (Course <course name> does not exist!).

```
coursedeadlines Surf↵
Course Surf does not exist!
```

2.13 `personaldeadlines` command

Returns the list of the deadlines defined for a given person (student or professor). This command receives as arguments:

- the **person name**, which can be composed of one or more tokens.

The expected behaviour is as follows: The user will insert the **person name**. The program then lists all the deadlines available for that person, one per line, sorting in ascending order by date an, in case of tie, by alphabetic order of course name. If the person has no deadlines, the adequate message is (No deadlines defined for <person name>!). The following example illustrates this.

The following example illustrates this.

```
personaldeadlines Tom Sawyer↵
[Computer Architecture] Project 1: 2022-04-07↵
[Computer Architecture] Project 2: 2022-04-21↵
[Object-Oriented Programming] First project: 2022-04-28↵
[Computer Architecture] Project 3: 2022-05-19↵
[Introduction to Procastination] Ludicrous idea: 2022-05-19↵
[Computer Architecture] Project 4: 2022-06-02↵
[Object-Oriended Programming] Second project: 2022-06-03↵
personaldeadlines Injun Joe↵
No deadlines defined for Injun Joe↵
```

The following errors may occur:

1. If the person name does not exist, the adequate error message is (Person <person name> does not exist!).

```
personaldeadlines Santa Claus↵
Santa Claus does not exist!↵
```

2.14 `deadline` command

Defines a new deadline for a given course. This command receives as arguments:

- the **`course name`**, which can be composed of one or more tokens;
- the **`deadline`**, which is a date;
- the **`deadline name`**, which can be composed of one or more tokens.

The expected behaviour is as follows: The user inserts a **`course name`**, a **`deadline`** and a **`deadline name`** and the program adds a new deadline to that course on the specified deadline (date) and outputs the message (Deadline <deadline> added to <course name>.). The following example illustrates this.

```
deadline Advanced Procastination↵
2053 8 4 Start thinking about the project↵
Deadline 2053-08-04 added to Advanced Procastination.↵
```

The following errors may occur:

1. If the **`course id`** does not exist, the adequate error message is (Course <course id> does not exist!).
2. If the course already has a deadline with the same name **`deadline name`**, the adequate error message is (Deadline <deadline name> already exists!).

```
deadline Wine tasting↵
Course Wine tasting does not exist!↵
deadline Advanced Procastination↵
2053 8 4 Start thinking about the project↵
Deadline 2053-08-04 added to Advanced Procastination.↵
deadline Advanced Procastination↵
2022 4 5 Start thinking about the project↵
Deadline Start thinking about the project already exists!↵
```

2.15 `coursetests` command

Lists the scheduled tests for a given course. This command receives as arguments:

- the **`course name`**, which can be composed of one or more tokens.

The expected behaviour is as follows: The user inserts a **`course name`** and the program produces a list of all the tests scheduled for their course, sorted in ascending order by date and starting time, with the format (<date> <starting time>-<ending time>: <test name>). If the course has no scheduled tests yet, the adequate message is (No scheduled tests for <course name>!). The following example illustrates the expected behaviour.

```
coursetests Object-Oriented Programming↵
2022-05-03 14h30-16h30: 1st test↵
2022-06-06 18h30-20h30: 2nd test↵
coursetests Functional Programming↵
No scheduled tests for Functional Programming!↵
```

The following errors may occur:

1. If the **course id** does not exist, the adequate error message is (Course <course id> does not exist!).

```
coursetests Nerd-Oriented Programming↵
Course Nerd-Oriented Programming does not exist!↵
```

2.16 `personaltests` command

Lists the scheduled tests for a given person. This command receives as arguments:

- the **person name**, which can be composed of one or more tokens.

The expected behaviour is as follows: The user inserts a **student name** and the program produces a list of all the tests scheduled for the courses that student is attending, sorted in ascending order by date, starting time and alphabetic order of the course name, with the format (<date> <starting time>-<ending time>: <course name> - <test name>). If the student has no scheduled tests yet, the adequate message is (No scheduled tests for <student name>!). The following example illustrates the expected behaviour.

The following example illustrates this.

```
personaltests Tom Sawyer↵
2022-05-03 14h30-16h30: Object-Oriented Programming - 1st test↵
2022-05-03 19h00-20h00: Advanced Hearts - Round 1↵
2022-05-31 12h00-14h00: Advanced Hearts - Round 2↵
2022-06-06 18h30-20h30: Object-Oriented Programming - 2nd test↵
2022-05-07 14h00-16h00: Advanced Hearts - Final Round↵
personaltests Sid Sawyer↵
No scheduled tests for Sid Sawyer!↵
```

The following errors may occur:

1. If the student named **person name** does not exist, the adequate error message is (Student <person name> does not exist!).

```
personaltests Santa Claus↵
Student Santa Claus does not exist!↵
personaltests Mr.  Dobbins↵
Student Mr. Dobbins does not exist!↵
```

2.17 `schedule` command

Schedules a test for a given course. This command receives as arguments:

- the date and time of the test, which should be read as a sequence of **`year`**, **`month`**, **`day`** and **`hour`**;
- the **`duration`** of the test, in hours;
- the **`course name`**, which can be composed of one or more tokens;
- the **`test name`**, which can be composed of one or more tokens.

The expected behaviour is as follows: the user inserts the test information. The system creates a new test and warns about potential conflicts, if any, with other scheduled tests. Conflicts may occur if there is another test from a different course on the same date, in a course which has intersections of students with the one the test is being defined for. The conflicts may be classified as:

- *free*, if no conflict is identified at the time of scheduling;
- *mild*, if there is a test from another course on the same date, with professors or students intersections, but at a time that does not intersect with the test being scheduled;
- *severe*, if there is a test from another course on the same date and intersecting times, with professors or students;

Upon successful scheduling, the output will have the format (<conflict classification> <course name> <test name> <test date> <test starting time> <test ending time> (<professors with conflict>, <students with conflict>])). The following example illustrates this.

```
schedule 2022 5 3 14 30 2 Object-Oriented Programming↵
1st test↵
free Object-Oriented Programming 1st test 2022-05-03 14h30-16h30 (0, 0)↵
schedule 2022 6 6 18 30 2 Object-Oriented Programming↵
2nd test↵
free Object-Oriented Programming 2nd test 2022-06-06 18h30-20h30 (0, 0)↵
schedule 2022 5 3 9 0 1 Stream Processing↵
Test 1↵
mild Stream Processing Test 1 2022-05-03 9h00-10h00 (0, 3)↵
schedule 2022 6 3 18 0 1 Stream Processing↵
Test 2↵
severe Stream Processing Test 2 2022-06-06 18h00-19h00 (0, 3)↵
```

The following errors may occur:

1. If the **`course name`** does not exist, the adequate error message is (Course <course name> does not exist!).
2. If the **`test name`** already exists in the course (**`course name`**, the adequate error message is (Course <course name> already has a test named <test name>!).
3. If there is another test of the same course at intersecting times, the adequate error message is (Cannot schedule test <test name> at that time!).

```
schedule 2022 5 3 14 30 2 Objet-Orienteed Programing↵
1st test↵
Course Objet-Orienteed Programing does not exist!↵
schedule 2022 5 3 14 30 2 Object-Oriented Programming↵
1st test↵
free Object-Oriented Programming 1st test 2022-05-03 14h30-16h30 (0, 0)↵
schedule 2022 6 6 18 30 2 Object-Oriented Programming↵
1st test↵
Course Object-Oriented Programming already has a test named 1st test!↵
schedule 2022 5 3 15 30 2 Object-Oriented Programming↵
2nd test↵
Cannot schedule test 2nd test at that time!↵
```

2.18 `superprofessor` command

Returns the professor with more students. This command receives no arguments and always succeeds.

The expected behaviour is as follows: The command presents the name of the professor with more students, with the format (<professor name> (<number of students>).). In case of a tie, the program presents the first professor to reach that number of students. If no professors have students, then the first professor to be inserted in the system is displayed. Finally, if there are no professors in the system, the adequate message is (There are no professors!).

The following example illustrates this.

```
superprofessor↵
There are no professors!↵
Several assignments later...↵
superprofessor↵
Mr. Dobbins (0).↵
Several assignments later...↵
superprofessor↵
Mr. Walters (43).↵
```

2.19 `stressometer` command

Lists the students with the top N stressful sequences of evaluations. This command receives as arguments:

- the **`number of students`** to list.

The expected behaviour is as follows: the user provides the **`number of students`** to be listed, and the program returns up to that **`number of students`**, sorted by the longest sequence of consecutive days with tests or deadlines. In case of ties, students with more evaluations in the same number of days are considered to be more stressed. If ties still persist, students enrolled in more courses are considered to be more stressed. Finally, if all else fails to break the tie, students with the highest student numbers are more stressed. To make it into this list, a student needs to have at least one evaluation. The output format is (<student number> <student name> (<consecutive evaluation days> days <number of evaluations> evaluations)). If there are less than **`number of students`** with evaluations, the command only lists those

with at least one evaluation. If there are no students with evaluations, the adequate feedback message is (There are no stressed students right now!). The following example illustrates this.

```
stressometer 5↵
There are no stressed students right now!↵
A few evaluations scheduled for two students...
stressometer 6↵
34242 Tom Sawyer (3 days, 5 evaluations)↵
36423 Becky Thatcher (3 days, 4 evaluations)↵
Some more evaluations scheduled for 10 students...
stressometer 3↵
12342 Joe Harper (3 days, 5 evaluations)↵
34242 Tom Sawyer (3 days, 5 evaluations)↵
36423 Becky Thatcher (3 days, 4 evaluations)↵
```

The following errors may occur:

1. If the **`number of students`** to list is less or equal to zero, the adequate error message is (Invalid number of students!).

```
stressometer 0↵
Invalid number of students!↵
```

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 evaluations (e.g. mandatory class presence), or new kinds of people (e.g. course monitors).

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 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, add professors and students. Once these are correctly inserted and listed, move on to adding and listing courses kinds. When that is done, move on to enrolling students and assigning professors to courses. 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.**

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 POO2022-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, which is completely predictable. 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

You should create your own tests as you build up your implementation. The official Mooshak tests verify incrementally the implementation of the commands. They will be be made publicly available on April 13, 2021. 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 should already have a lot more than those to test, anyway. If not, you are getting late! When your program passes all official Mooshak tests on your machine, it should pass them as well on Mooshak.

4.3 Code quality

The evaluation of your project covers not only the extent to which it successfully passes the tests, but also how well it is programmed. Please review the code quality criteria published on Moodle with this project. Use them as a checklist for how you implement your code. Your instructors will certainly use them as part of their checklist when grading it! Good luck!