Version Control System

Object Oriented Programming
2nd Project, version 1.5 – 19-05-2021
Contact: carla.ferreira@fct.unl.pt

Important remarks

Deadline: Until 23h59 (Lisbon time) of June 03, 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 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 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, if necessary, defining preconditions for their usage. Students may and should discuss any questions 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 website.

1 Development of a *Version Control System*

1.1 Problem description

The objective of the work is the implementation of a version control system for a software house. This system is responsible for managing changes to software projects which include computer programs, documents, class diagrams, or other artefacts. Changes are identified by a number, termed the "revision number". For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on. Each revision is associated with a timestamp and the employee making the update. To simplify the development, each project artefact will have its own independent revision number that starts with the number 1 and is incremented each time the artefact is updated, as discussed before. Given that some projects developed have confidentiality concerns, in addition to managing revisions, the system should also ensure that the personnel working on these projects have the necessary clearance level. The application will support data on the employees, either software developers or project managers, the projects and their associated artefacts, the definition of project teams of developers, and the changes/updates made to project artefacts.

Users For each registered user the system maintains the username (unique identifier) and their job position. The job positions to be considered are a project manager and software developer. Company personnel also maintain a clearance level, limiting the projects and artefacts that can be updated. Software developers are associated with a project manager, responsible for managing their work. Project managers are also responsible for creating new projects and associating a team for each project they are managing. Developers work on the projects they were assigned by project managers, but project managers can also work on projects that are not managed by them.

Project Each project includes the following information: a unique identifier, the manager, and the relevant keywords. There are two types of projects: in-house or outsourced. Outsourced projects have no associated developing team, instead, it is just kept the name of the company developing the project. In-house projects have a confidentiality level, a developing team, and associated artefacts.

Each artefact has the following information: an owner – the developer that added the artefact to the project, a name that must be unique within the project, a description (e.g. java code, UML class diagram) and a sequence of revisions. Each revision includes the username of the team member responsible for updating the artefact, a date, and a short description of the revision.

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 in 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↵
register - adds a new user↵
users - lists all registered users↵
create - creates a new project↵
projects - lists all projects↵
team - adds team members to a project↵
artefacts - adds artefacts to a project↵
project - shows detailed project information↵
revision - revises an artefact↵
manages - lists developers of a manager↵
keyword - filters projects by keyword↵
confidentiality - filters projects by confidentiality level↵
workaholics - top 3 employees with more artefacts updates↵
common - employees with more projects in common↵
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 `register` command

Registers a user in the system. The command receives as arguments the **job position**, which can be either project **manager** or software **developer** followed by their **username**. If registering a developer the command receives their **manager username**. The last argument is the **clearance level** – a value between 0 and 5, with 5 being the highest clearance level. In case of success, the message presented is (User <username> was registered as <job position> with clearance level <clearance level>.).

```
register manager peralta 4↵
User peralta was registered as project manager with clearance level 4.↵
```

```
register developer boyle peralta 3↵
User boyle was registered as software developer with clearance level 3.↵
```

This command will fail if:

1. The **job position** is unknown, the adequate error message is (Unknown job position.).
2. The **username** already exists. In this case, the error feedback message is (User <username> already exists.).
3. The **manager username** does not exist or it does not belong to a manager. In this case, the error feedback message is (Project manager <manager username> does not exist.).

```
register intern alice 5↵
Unknown job position.↵
register developer peralta diaz 3↵
User peralta already exists.↵
register developer alice boyle 3↵
Project manager boyle does not exist.↵
register developer alice bob 3↵
Project manager bob does not exist.↵
```

It is not necessary to consider the error situation where the clearance level is less than 0 or greater than 5.

2.4 `users` command

Lists all registered users. This command does not receive any arguments and always succeeds. If no user is registered, the output is (No user is registered.). Otherwise, it will present a header line (All registered users:) followed by summary information about all the registered people, in alphabetical order of username, one in each line, with the following format:

1. If the username is a manager, the format is (manager <username> [<number of developers>, <projects as manager>, <projects as members>]), representing the manager's username, the number of developers managed by them, and the number of participating projects as a manager and as a team member.

2. If the person is a software developer, the format is (developer <username> is managed by <manager> [<number of projects>]), representing the developer's username, their manager, and the number of participating projects as a team member.

The following scenario illustrates its usage.

```
users↵
No users registered.↵
... a few registrations later...
users↵
All registered users:↵
developer boyle is managed by holt [0]↵
manager holt [0, 0, 0]↵
developer linetti is managed by holt [0]↵
manager peralta [1, 0, 0]↵
```

2.5 `create` command

Creates a new project. The command receives as arguments the **`username`** of the project manager, the type of project, which can be either **`inhouse`**, or **`outsourced`** followed by the **`project name`**, an integer **`value`** greater than zero specifying how keywords are going to be associated with the project. This parameter is followed by the **`keywords`** that best describe the project. For outsourced projects, the last argument is the **`name of the company`** developing the project, while for in-house projects is the project **`confidentiality level`** (the upper bound for the confidentiality level of any artefact associated with the project). In case of success, the message presented is (<project name > project was created.).

```
create santiago inhouse Online occurrence reporting↵
3 web javascript mysql↵
2↵
Online occurrence reporting project was created.↵
```

```
create diaz outsourced Online occurrence reporting UI↵
2 css javascript↵
UI4All↵
Online occurrence reporting UI project was created.↵
```

This command will fail if:

1. The **`project type`** is unknown, the adequate error message is (Unknown project type.).

2. The **`username`** does not exist or it does not belong to a manager. In this case, the error feedback message is (Project manager <username> does not exist.).

3. The **project name** already exists. In this case, the error feedback message is (<project name> project already exists.).

4. The project **confidentiality level** is higher than the clearance level of its manager. In this case, the error feedback message is (Project manager <username> has clearance level <level>.).

```
create diaz other My fabulous app↵
4 c# redis json html↵
3↵
Unknown project type.↵
create alice inhouse My fabulous app↵
4 c# redis json html↵
3↵
Project manager alice does not exist.↵
create boyle inhouse My fabulous app↵
4 c# redis json html↵
3↵
Project manager boyle does not exist.↵
create diaz inhouse My fabulous app↵
4 c# redis json html↵
3↵
My fabulous app project was created.↵
create diaz inhouse My fabulous app↵
4 c# redis json html↵
3↵
My fabulous app project already exists.↵
create diaz inhouse Confidential fabulous app↵
4 c# redis json html↵
5↵
Project manager diaz has clearance level 4.↵
```

2.6 `projects` command

Lists all projects. This command does not receive any arguments and always succeeds. If no user is registered, the output is (No projects added.). Otherwise, it will present a header line (All projects:) followed by summary information about all the existing projects, by insertion order, one in each line, with the following format:

1. If the project is an in-house project, the format is (in-house <project name> is managed by <username> [<level>, <number of members>, <number of artefacts>, <number of revisions>]), representing the project name, the manager username, confidentiality level, and the number of project members, artefacts, and the total number of artefacts revisions.

2. If the project is an outsourced project, the format is (outsourced <project name> is managed by <username> and developed by <company>), representing the project name, the manager username, and the company developing the project.

The following scenario illustrates its usage.

```
projects↵
No projects added.↵
... a few projects later...
projects↵
All projects:↵
in-house Online occurrence reporting is managed by santiago [2, 0, 0, 0]↵
in-house Wordle is managed by santiago [1, 0, 0, 0]↵
outsourced Online occurrence reporting UI is managed by diaz and developed by UI4All↵
in-house Top secret project is managed by holt [5, 0, 0, 0]↵
```

2.7 `team` command

Adds team members to a project. The command receives as arguments the manager **username**, the in-house **project name**, followed by an integer **value** greater than zero specifying how many usernames we are trying to associated with the project. For each candidate team member, the program shows the outcome of trying to add that member to the project team, as described next. The program will start by presenting a header line (Latest team members:) followed by the outcome of the command for each team member. This outcome can be: i) (<username>: added to the team.), ii) (<username>: already a member.), iii) (<username>: insufficient clearance level.), iv) (<username>: does not exist.). This command can be called multiple times for a project.

```
team diaz Yet another project↵
5 boyle peralta linetti jeffords alice↵
Latest team members:↵
boyle: insufficient clearance level.↵
peralta: added to the team.↵
linetti: already a member.↵
jeffords: added to the team.↵
alice: does not exist.↵
```

1. The **username** does not exist or it does not belong to a manager. In this case, the error feedback message is (Project manager <username> does not exist.).
2. The **project name** does not exist or if it is an outsourced project. In this case, the error feedback message is (<project name> project does not exist.).
3. Project **project name** is not managed by **username**. In this case, the error feedback message is (<project name> is managed by <manager username>.).

```
team alice My fabulous app↵
5 boyle peralta linetti jeffords alice↵
Project manager alice does not exist.↵
team linetti My fabulous app↵
5 boyle peralta linetti jeffords alice↵
Project manager linetti does not exist.↵
team peralta Wordle app↵
1 boyle↵
Wordle app project does not exist.↵
team diaz Online occurrence reporting UI↵
2 scully holt↵
Online occurrence reporting UI project does not exist.↵
team peralta Stationary wish list↵
2 linetti holt↵
Stationary wish list is managed by santiago.↵
```

2.8 `artefacts` command

Adds artefacts to a project. The command receives as arguments the teams' member **`username`**, the in-house **`project name`**, followed by a **`date`** and an integer **`value`** greater than zero specifying how many artefacts are going to be added to the project. For each artefact, it is received its **`name`**, its **`confidentiality level`**, and a short **`description`**. In case of success, it will present a header line (Latest project artefacts:) followed by the outcome of the command for each artefact. This outcome can be: i) (<name>: added to the project.), ii) (<name>: already in to the project.), iii) (<name>: exceeds project confidentiality level.). This command can be called multiple times for a project.

```
artefacts diaz Yet another project↵
23-03-2022↵
4↵
Main.java 3 java↵
diagram.jpg 4 jpg image↵
todo.txt 3 text file↵
passwords.xlsx 5 passwords in an excel file!?↵
Latest project artefacts:↵
Main.java: added to the project.↵
diagram.jpg: already in the project.↵
todo.txt: added to the project.↵
passwords.xlsx exceeds project confidentiality level.↵
```

1. The **`username`** does not exist. In this case, the error feedback message is (User <username> does not exist.).
2. The **`project name`** does not exist or if it is an outsourced project. In this case, the error feedback message is (<project name> project does not exist.).
3. User **`username`** is not a member of the team of **`project name`**. In this case, the error feedback message is (User <username> does not belong to the team of <project name>.).

```
artefacts alice My fabulous app↵
25-12-2021↵
1↵
Main.java 3 java↵
User alice does not exist.↵
artefacts holt Wordle app↵
25-12-2021↵
1↵
Main.java 3 java↵
Wordle app does not exist.↵
artefacts diaz Online occurrence reporting UI↵
25-12-2021↵
1↵
Main.java 3 java↵
Online occurrence reporting UI project does not exist.↵
artefacts peralta Stationary wish list↵
25-12-2021↵
1↵
Main.java 3 java↵
User peralta does not belong to the team of Stationary wish list.↵
```

2.9 `project` command

Shows the detailed information of an in-house project This command receives as argument the **`project name`** and presents its detailed information. The program will write the message (<project name> [<project level>] managed by <manager> [<manager level>]:) in the first line and then print the team members by insertion order, followed by the project artefacts are ordered by the date of the last revision number (last revision first) and then by name. The format for each team member is (<member username> [<member level>]). The format for each project artefact is (<artefact name> [<artefact level>]: <artefact description>) followed by a list of revision (last revision first). Each revision presents (<revision number> <username> <date> <comment>). The first revision is the initial upload of the artefact and its description.

```
project Wordle app↵
Wordle app [2] managed by santiago [5]:↵
scully [3]↵
boyle [4]↵
wordle.cpp [2]↵
revision 3 scully 12-06-2021 updated comments↵
revision 2 boyle 10-06-2021 refactored code↵
revision 1 santiago 01-06-2021 c++ file↵
readme [0]↵
revision 1 scully 02-06-2021 compiling instructions↵
```

This command will fail if:

1. The **`project name`** does not exist. In this case, the error feedback message is (<project name> project does not exist.).
2. The project is outsourced. In this case, the error feedback message is (<project name> is an outsourced project.).

```
project Some app↵
Some app project does not exist.↵
project Online occurrence reporting UI↵
Online occurrence reporting UI is an outsourced project.↵
```

2.10 `revision` command

Revises an artefact. The command receives as arguments the **`username`** of the team member updating the artefact, the **`project name`**, the **`artefact name`**, the date **`date`**, and **`comment`** describing the revision. In case of success, the revision is registered and the following message is shown (Revision <revision number> of **`artefact name`** was submitted.). It can be assumed that for any two consecutive revisions the date for the first revision has to be less or equal to the date for the second revision.

```
revision boyle Wordle app↵
wordle.cpp 04-05-2022 minor fix↵
Revision 3 of artefact wordle.cpp was submitted.↵
```

This command will fail if:

1. The **`username`** does not exist. In this case, the error feedback message is (User <username> does not exist.).
2. The **`project name`** does not exist or if it is an outsourced project. In this case, the error feedback message is (<project name> project does not exist.).

3. The **artefact name** does not exist in the project. In this case, the error feedback message is (<artefact name> does not exist in the project.).

4. User **username** is not a member of the team of **project name**. In this case, the error feedback message is (User <username> does not belong to the team of <project name>.).

```
revision alice Some app↵
Main.cpp 12-12-2019 refactored code↵
User alice does not exist.↵
revision holt Some app↵
Main.cpp 12-12-2019 refactored code↵
Some app project does not exist.↵
revision holt Online occurrence reporting UI↵
Main.cpp 12-12-2019 refactored code↵
Online occurrence reporting UI project does not exist.↵
revision santiago Wordle app↵
Scheduler.cpp 12-12-2019 refactored code↵
Scheduler.cpp does not exist in the project.↵
revision holt Wordle app↵
Main.cpp 12-12-2019 refactored code↵
User holt does not belong to the team of Wordle app.↵
```

2.11 `manages` command

Shows the detailed information about the developers supervised by a given manager. This command receives as an argument the manager **username** and presents information about the developers supervised by the given manager. The program will write the message (Manager <username>:) in the first line and then print the developers ordered by alphabetical order, and for each developer, the revisions made ordered by the date (from newest to oldest), then by revision number (last revision first), and then by project name. The format for each revision is (<project name>, <artefact name>, <revision number>, <date>, <comment>).

```
manages holt↵
Manager holt:↵
boyle↵
Restaurant guide app, cover.html, revision 5, 09-04-2022, refactored code↵
Wordle app, wordle.cpp, revision 5, 09-04-2022, refactored code↵
Restaurant guide app, cover.html, revision 2, 03-04-2022, update table↵
linetti↵
Yet another app, Main.java, revision 12, 12-04-2022, minor fix↵
Yet another app, Main.java, revision 11, 10-04-2022, fix command interpreter↵
Find my laptop, search.py, revision 7, 10-04-2022, commented algorithm↵
```

This command will fail if:

1. The **username** does not exist or does not belong to a manager. In this case, the error feedback message is (Project manager <username> does not exist.).

```
manages alice↵
Project manager alice does not exist.↵
manages scully↵
Project manager scully does not exist.↵
```

2.12 `keyword` command

Filters projects by keyword. This command receives as an argument a **`keyword`**. The command always succeeds, but if no project is associated with the given **`keyword`**, the output is (No projects with keyword <keyword>.). Otherwise, it will present a header line (All projects with keyword <keyword>:) followed by summary information about the filtered projects. In-house projects should appear first, sorted by the most recent revision, then by the number of updates, and lastly project name. These projects are then followed by outsourced projects sorted by project name.

1. If the project is an in-house project, the format is (in-house <project name> is managed by <username> [<level>, <members>, <artefacts>, <revisions>, <last updated>]), representing the project name, the manager username, confidentiality level, and the number of project members, artefacts, and artefacts revisions.

2. If the project is an outsourced project, the format is (outsourced <project name> is managed by <username> and developed by <company>), representing the project name, the manager username, and the company developing the project.

The following scenario illustrates its usage.

```
keyword java↵
No projects with keyword java.↵
... a few projects later...
keyword java↵
All projects with keyword java:↵
in-house Online occurrence reporting is managed by santiago [2, 4, 20, 120, 10-05-2022] ↵
in-house Wordle is managed by santiago [1, 3, 12, 35, 09-05-2022]↵
in-house Top secret project is managed by holt [5, 5, 12, 33, 09-05-2022]↵
outsourced Online occurrence reporting UI is managed by diaz and developed by UI4All↵
```

2.13 `confidentiality` command

Filters in-house projects by confidentiality level. This command receives as an argument two integers **`lower limit`** and **`upper limit`**. The command should return all in-house projects with a confidentiality level between the **`lower limit`** and **`upper limit`**. The command always succeeds, but if no project exists within the confidentiality range, the output is (No projects within levels <lower level> and <upper level>.). Otherwise, it will present a header line (All projects within levels <lower level> and <upper level>:) followed by summary information about filtered projects sorted by the project name. The format is (<project name> [<level>] is managed by <username> and has keywords <list of keywords>.), representing the project name, confidentiality level, the manager username, and the list of keywords separated by a space. The following scenario illustrates its usage.

```
confidentiality 1 2↵
No projects within levels 1 and 2.↵
... a few projects later...
confidentiality 1 2↵
All projects within levels 1 and 2:↵
Restaurant guide app is managed by boyle and has keywords android, kotlin.↵
Wordle is managed by santiago and has keywords puzzle, javascript.↵
confidentiality 2 1↵
All projects within levels 1 and 2:↵
Restaurant guide app is managed by boyle and has keywords android, kotlin.↵
Wordle is managed by santiago and has keywords puzzle, javascript.↵
confidentiality 1 1↵
All projects within levels 1 and 1:↵
Wordle is managed by santiago and has keywords puzzle, javascript.↵
```

2.14 `workaholics` command

Determines the 3 employees with more artefacts updates. The program prints a header (Workaholics:) and returns up to 3 employees that have made more updates to artefacts. In the case of ties, employees that participate in more projects as managers or developers are considered to be more workaholics. Next, ties are broken using the date of the last update, newest to oldest. Lastly, if all else fails, ties are broken by using the alphabetical order of the employees' usernames. The output format is (<username>: <number of updates> updates, <number of projects> projects, last update on <date>.). If there are no employees or if no employee has made an update, the adequate feedback message is (There are no workaholics.). The following example illustrates this.

```
workaholics↵
There are no workaholics.↵
After multiple commands...↵
workaholics↵
holt: 12 updates, 3 projects, last update on 11-05-2022↵
peralta: 12 updates, 3 projects, last update on 09-05-2022↵
After multiple commands...↵
workaholics↵
diaz: 15 updates, 1 projects, last update on 09-04-2022↵
holt: 13 updates, 4 projects, last update on 11-05-2022↵
peralta: 13 updates, 3 projects, last update on 09-05-2022↵
```

2.15 `common` command

Determines the two employees that have more projects in common. The command presents the usernames of the two employees with more projects in common (<username> and <username> have <number> projects in common.)). In case of a tie, the program presents the employees that have participated in the most recently updated projects, ordered in alphabetical order. In case of another tie, the command should select the employees using alphabetical order. If no employees have common projects (Cannot determine employees with common projects.).

The following example illustrates this.

```
common↵
Cannot determine employees with common projects.↵
After multiple commands...↵
common↵
holt linetti have 4 projects in common.↵
After multiple commands...↵
common↵
diaz santiago have 12 projects in common.↵
```

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 projects.

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, implement the register command and the users listing operation, so that you can create users and then check they are ok. Then implement the commands that add projects and its associated listing operation. 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 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 POO2122-TP2 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

The Mooshak tests verify incrementally the implementation of the commands. They will be made publicly available on May 21, 2022. 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!