Group Project: A Command-Line Task Management System

Deadlines

1. Form your groups of 3 to 4 students on or before 10 October 2023. Afterward, students with no group will be randomly grouped, and requests for group change are only approved if written agreements from all members of the involved groups are provided before or on 14 October 2023.
2. Compress the source code, the tests, the presentation slides, and the individual reports in PDF into a single ZIP archive and submit it on Blackboard before or on 25 November 2023.
3. Submit the design review report on Blackboard before midnight, 24 November 2023.

1 Overview

This project aims to develop a command-line-based task management system (TMS) that will enable its users to define, query, and modify the simple tasks they need to complete.

A task can be simple or composite. Simple tasks are the smallest unions of work to manage using the system, while each composite task is a collection of other (simple or composite) tasks. Each task has a unique `name`, a `description`, a `duration`, and a group of `prerequisites`. A task `name` 1) may contain only English letters and digits, 2) cannot start with digits, and 3) may contain at most eight characters. A `description` may contain English letters, digits, and the hyphen letter (-). A `duration` is a positive real number, and it gives the *minimum* amount of time in hours required to complete the task. The `prerequisites` is a comma-separated list of task names. Each task name in the list should be defined already, and a task can only start when all its prerequisite tasks have finished. An empty list is denoted using a single comma (i.e., ",").

Note that the prerequisite relation between tasks is transitive. Consider three tasks `t1`, `t2`, and `t3` for example. If task `t1` is a prerequisite for task `t2`, and task `t2` is a prerequisite for task `t3`, task `t1` is also a prerequisite for task `t3` since task `t3` cannot start before task `t1` is finished. In this example, task `t1` is a *direct* prerequisite for task `t2` and an *indirect* prerequisite for task `t3`. When creating or changing a simple task, you should only specify the direct prerequisites of the task under consideration. The indirect prerequisites will be derived by the TMS automatically.

2 Requirements

The system should satisfy the following requirements.

[REQ1] (2 points) The system should support creating simple tasks.
Command: `CreateSimpleTask name description duration prerequisites`
Effect: Creates a new simple task.
Example: `CreateSimpleTask task1 boil-water 0.3 ,`

[REQ2] (2 points) The system should support creating composite tasks.
Command: `CreateCompositeTask name0 description name1,name2,...,namek`
Effect: Creates a composite task. The new task is named `name0` and has k sub-tasks, named `name1`, `name2`, ..., `namek` ($k \geq 2$), respectively.
Example: `CreateCompositeTask composite1 make-coffee primitive1,primitive2,composite0`

[REQ3] (2 points) It should support deleting tasks.
Command: `DeleteTask name`
Effect: Deletes an existing task.
Example: `DeleteTask task1`
Note: A simple task can be deleted if it is not the prerequisite of any other tasks; A composite task can be deleted if it does not contain any sub-task that is the prerequisite of another task. When a composite task is deleted, all the sub-tasks it contains are deleted too.

[REQ4] (2 points) It should support changing the properties of an existing task.
Command: `ChangeTask name property newValue`
Effect: Sets the property of a specific task to the new value.
Example: `ChangeTask task1 duration 0.5`
Note: (1) If the task is primitive, `property` can be `name`, `description`, `duration`, or `prerequisites`; If the task is composite, `property` can be `name`, `description`, or `subtasks`. The `newValue` may take any valid value compatible with the corresponding property. (2) The prerequisites of a task are not affected by changes to the other properties of the task.

[REQ5] (1 point) It should support printing the information of a task.
Command: `PrintTask name`
Effect: Prints the information of a task.
Example: `Print task1`
Note: The printout should be easy to read.

[REQ6] (2 points) It should support printing the information of all tasks.
Command: `PrintAllTasks`
Effect: Prints the information of all existing tasks.
Example: `PrintAllTasks`

[REQ7] (4 points) It should support reporting the duration of a task.
Command: `ReportDuration name`
Effect: Reports the duration of a task.
Example: `ReportDuration task1`
Note: The duration of a composite task is the minimum amount of hours needed to finish all its sub-tasks. Suppose that a composite task `t0` has three subtasks `t1`, `t2`, and `t3`, with their durations being 1 hour, 2 hours, and 2 hours, respectively. Further suppose that both `t1` and `t2` are prerequisites for `t3` and there is no prerequisite relation between `t1` and `t2`. The duration of task `t0` is then 4 hours.

[REQ8] (4 points) It should support reporting the earliest finish time of a task.
Command: `ReportEarliestFinishTime name`
Effect: Reports the earliest finish time of a task.
Example: `ReportEarliestFinishTime task1`
Note: The earliest finish time of a task is calculated as the sum of 1) the earliest finish time of all its prerequisites and 2) the duration of itself.

[REQ9] (2 points) It should support defining basic task selection criteria.
Command: `DefineBasicCriterion name1 property op value`
Effect: Defines a basic task selection criterion.
Example: `DefineBasicCriterion criterion1 duration > 0.1`
Note: `name1` is a unique name for the new basic criterion. The construction of criterion names follows the same rules as task names. `property` is either `name`, `description`, `duration`, or `prerequisites`. If `property` is `name` or `description`, `op` must be `contains` and *value* must be a string in double quotes; If `property` is `duration`, `op` can be `>`, `<`, `>=`, `<=`, `==`, or `!=`, and `value` must be a real value. If `property` is `prerequisites` or `subtasks`, `op` must be `contains`, and `value` must be a list of comma-separated task names.

[REQ10] (1 point) It should support a criterion to check whether a task is primitive.
`Criterion name: IsPrimitive`
`Effect:` Evaluates to `true` if and only if on a primitive task.

[REQ11] (3 points) It should support defining composite task selection criteria.
Command: `DefineNegatedCriterion name1 name2`
Command: `DefineBinaryCriterion  name1 name2 logicOp name3`
Effect: Construct a composite criterion named *name1*. The new criterion constructed using `DefineNegatedCriterion` is the negation of an existing criterion

named `name2`; The new criterion constructed using `DefineBinaryCriterion` is `name2 logicOp name3`, where `name2` and `name3` are two existing criteria, while `logicOp` is either `&&` or `||`. The logical operators `negation`, `and` (`&&`), `or` (`||`) have the conventional precedence, associativity, and semantics.

[REQ12] (3 points) The tool should support printing all defined criteria.
Command: `PrintAllCriteria`
Effect: Print out all the criteria defined. All criteria should be resolved to the form containing only property names, `op`, `value`, `logicOp`, and `IsPrimitive`.

[REQ13] (5 points) The tool should support searching for tasks based on an existing criterion.
Command: `Search name`
Effect: List all tasks that satisfy the criterion with the given name.

[REQ14] (3 points) The tool should support storing the defined tasks and criteria into a file.
Command: `Store path`
Effect: Stores the defined tasks and criteria into the file at `path`.
Example: `Store d:\tasks.data`

[REQ15] (3 points) The tool should support loading tasks and criteria from a file.
Command: `Load path`
Effect: Loads the defined tasks and criteria from the file at `path`.
Example: `Load d:\tasks.data`
Note: Loading the tasks and criteria from a file will cause all tasks and criteria defined before the `Load` command to be discarded.

[REQ16] (1 point) The user should be able to terminate the current execution of the system.
Command: `Quit`
Effect: Terminates the execution of the system.

The system may be extended with the following *bonus* features:

[BON1] (8 points) Support for a graphical user interface (GUI) to visually show the tasks created or selected and their relationship. If you implement this bonus feature, your GUI should be able to support all the required functionalities. Note that you still need to implement the command line interface even if your TMS has a GUI.

[BON2] (4 points) Support for the `undo` and `redo`[1] of all the required commands except `PrintTask`, `PrintAllTasks`, `ReportDuration`, `ReportEarliestFinishTime`, `PrintAllCriteria`, `Search`, `Store`, `Load`, and `Quit`.

The requirements above are incomplete on purpose, and you need to decide on the missing details when designing and implementing the TMS. For example, you need to gracefully handle situations where a command is ill-formed or cannot be executed successfully, e.g., because a task name is already used or not defined. Poor design in handling those situations will lead to point deductions.

3 Group Forming

This is a group project. Each group may have 3 to 4 members. Form your group on or before 10 October 2023. Afterward, students with no group will be randomly grouped, and requests for group change are only approved if written agreements from all members of the involved groups are provided before or on 14 October 2023. No group change will be allowed after 14 October 2023.

[1] https://en.wikipedia.org/wiki/Undo

4 Code Inspection

The inspection tool of IntelliJ IDEA[2] checks your program to identify potential problems in the source code. We have prepared a set of rules for grading (`COMP2021_PROJECT.xml` in the project material package). Import the rules into your IntelliJ IDEA IDE and check your implementation against the rules. Note that unit tests do not need to be checked against these rules.

The inspection tool can check for many potential problems, but only a few of them are considered errors by the provided rule set. 2 points will be deducted for each *error* in your code reported by the tool.

5 Line Coverage of Unit Tests

The line coverage[3] achieved by a suite of unit tests is the percentage of source code lines that the tests exercise, and it is an important measure of the test suite's quality. The higher the coverage, the more thoroughly the suite of tests exercised a program.

You should follow the Model-View-Controller (MVC) pattern[4] in designing the system. Put all Java classes for the model into package `hk.edu.polyu.comp.comp2021.tms.model` (see the structure in the sample project) and *write tests for the model classes.*

During grading, we will use the coverage tool of IntelliJ IDEA[5] to get the line coverage of your tests on package `hk.edu.polyu.comp.comp2021.tms.model`. You will get 10 base points for line coverage between 90% and 100%, 9 base points for coverage between 80% and 89%, and so on. You will get 0 base points for line coverage below 30%. The final points you get for line coverage of unit tests will be calculated as your base points multiplied by the percentage of your requirements coverage. For example, if you only implement 60% of the requirements and you achieve 95% line coverage in testing, you will get 6 = 10 * 60% points for this part.

6 Design Review

The 3-hour lecture time in Week 13 will be devoted to the peer review of your design. The review will be conducted in clusters of X groups (X is to be decided later); Each group needs to review the designs of the other groups in its cluster and fill out the review reports. In a review report, a group needs to point out which design choices from another group are good, which are questionable, which should have been avoided, and explain the reasons.

Details about the design review's organization and the report's format will be announced before the review.

7 Individual Report

It is crucial for each member to submit an individual report. The report should be prepared from your own perspective and reflect your understanding of the project. It is mandatory for the individual report to demonstrate a distinct difference of at least 30% from the reports of other group members.

Additionally, your report should include a dedicated subsection titled "My Contribution and Role Distinction." In this subsection, you should provide a detailed account of your individual contributions to the project and highlight the unique aspects of your role in comparison to other group members. The content of all other sections, except the "User Manual," should align with the claims made in this subsection.

Use of GenAI tools: If you use GenAI tools in preparing any of the deliverables, you need to clearly state in the subsection titled "My Contribution and Role Distinction" of your individual report 1) which deliverable(s) and which part(s) of the deliverable(s) were prepared with the help of GenAI tools, 2) how you used the GenAI tools, and 3) what you did to verify/revise/improve the results produced by the tools. Without such explanations in your report, we will consider all the deliverables your original work. Using GenAI tools in the project is permitted, but you should give proper acknowledgement to all the content from those tools that you included in your

[2] https://www.jetbrains.com/help/idea/code-inspection.html
[3] http://en.wikipedia.org/wiki/Code_coverage
[4] https://en.wikipedia.org/wiki/Model-view-controller
[5] https://www.jetbrains.com/help/idea/code-coverage.html

deliverables. Failure to reference externally sourced, non-original work can be considered plagiarism. You may refer to https://libguides.lb.polyu.edu.hk/academic-integrity/GenAI_and_Plagiarism for more guidance on GenAI and plagiarism.

8 Submission

Each group is required to submit a single ZIP file containing their source code, user manual, report, and presentation slides. You should organize the files and directories in a consistent structure to ensure clarity and ease of access. It is important to note that individual reports of group members should be included in the "Report" directory, with student IDs incorporated into the file names as identifiers. Here are the guidelines for preparing the ZIP file:

1. Create a main directory with your project's name or a suitable identifier.
2. Within the main directory, create subdirectories for each component of the submission, such as Source Code, User Manual, Report, and Presentation Slides.
3. Place the relevant files in their respective directories. For example:
   - The IntelliJ IDEA project folder should be placed inside the Source Code directory.
   - The user manual document should be placed inside the User Manual directory.
   - The report document should be placed inside the Report directory.
   - The presentation slides should be placed inside the Presentation Slides directory.
4. Ensure that all files are appropriately named and in the required file formats.
5. Double-check that the files and directories are organized correctly and follow the specified structure.
6. Select all the directories and files within the main directory.
7. Compress the selected items into a ZIP file.

Example Directory Structure:

```
GroupID
├── Source Code
│   └── The IntelliJ IDEA project folder
├── User Manual
│   └── user_manual.pdf
├── Report
│   ├── StudentID1_report.pdf
│   ├── StudentID2_report.pdf
│   └── ...
└── Presentation Slides
    └── presentation.pdf
```

9 Project Grading

You can earn at most 100 points in total in the project from the following components:

- Requirements coverage (in total 40 points, as listed in Section 2)
- Code quality (10 points for good object-oriented design and 10 for good code style, as reported by the code inspection tool)
- Line coverage of unit tests (10 points)
- Presentation (7 points)
- Design review report (8 points)

- Individual report and user manual (15 points)
- Bonus points (12 points)

  Note: Bonus points can be used to reach, but not exceed, 100 points.

Individual grading: Each group member is expected to contribute fairly to the project. While group members will generally receive the same grade for their collective work, the grading process will be conducted individually. Each member's grade will be based on their individual contributions to the project, and it is necessary for each member to provide evidence of their own contributions.

10 General Notes

- Java SE Development Kit Version 17[6] and IntelliJ IDEA Community Edition Version 2023.2[7] will be used in grading your project. Make sure you use the same versions of tools in your development. Note that you need to configure "File|Project Structure..." in IntelliJ IDEA as the following:
  - Project SDK: JDK 17
  - Project language level: 8 - Lambdas, type annotations etc.
- Your code should not rely on any library beyond the standard Java SE 17 API.
- The project material package also includes three other files:
  - `SampleProject.zip`: Provides an example for the structure of the project directory. Feel free to build the interpreter based on the sample project.
  - `IntelliJ IDEA Tutorial.pdf`: Briefly explains how certain tasks relevant to the project can be accomplished in IntelliJ IDEA.
  - `COMP2021_PROJECT.xml`: Contains the rules for code inspection.

If you use other tools and/or IDEs for your development, make sure all your classes and tests are put into an IntelliJ IDEA project that is ready to be compiled and executed. 50 points will be deducted from a project not satisfying this requirement or with compilation errors!

[6] https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html
[7] https://www.jetbrains.com/idea/download/other.html

11 Project Report Template

Project Report
Group XYZ
COMP2021 Object-Oriented Programming (Fall 2023)
Author: Member1
Other group members:
Member2
Member3

1 Introduction

This document describes group XYZ's design and implementation of a command-line-based task management system. The project is part of the course COMP2021 Object-Oriented Programming at PolyU.

2 My Contribution and Role Distinction

Use this subsection to outline your specific contributions to the project and highlight the unique aspects of your role compared to other members of the group. It is essential to ensure that the content of your individual report aligns with the claims made in this subsection.

3 A Command-Line Task Management System

In this section, we describe the system's overall design and implementation details.

3.1 Design

Use class diagram(s) to give an overview of the system design and explain how different components fit together. Feel free to elaborate on the used design patterns and/or anything else that might help others understand your design.

3.2 Requirements

For each (compulsory and bonus) requirement, describe 1) whether it is implemented and, when yes, 2) how you implemented the requirement as well as 3) how you handled various error conditions.

[REQ1] 1) The requirement is implemented.
2) Implementation details.
3) Error conditions and how they are handled.

[REQ2] 1) The requirement is not implemented.

[REQ3] ...

12 User Manual

To prepare a user manual, it's important to note that there is no one-size-fits-all template. User manuals can differ significantly based on the system being documented and the specific needs of the users. However, the following guidelines can help you create an effective user manual:

- Introduction: Begin by providing an overview of the system and its purpose. Explain how the system works from a user's perspective.
- Commands Description: Describe all the commands that the system supports. Provide detailed explanations of each command, including their functionalities and how they can be used.
- Screenshots: For each command, consider including screenshots to visually demonstrate the results under different inputs. Capture screenshots that showcase the system's response or output when specific commands are executed. Include captions or annotations to provide additional context or explanations for each screenshot.
- Step-by-Step Instructions: Provide clear and concise step-by-step instructions for users to follow when using each command. Break down the process into logical steps, ensuring that users can easily understand and replicate the actions.
- Troubleshooting: Include a troubleshooting section to address common issues or errors that users may encounter while using the system. Provide solutions or workarounds for each problem, ensuring that users can resolve issues on their own.
- Additional Resources: If applicable, include additional resources such as references, FAQs, or contact information for technical support. Provide users with avenues to seek assistance or further information if needed.
- Formatting and Organization: Ensure that the user manual is well-formatted and organized. Use headings, subheadings, and bullet points to improve readability. Consider using a consistent and intuitive structure throughout the manual.

Remember, the purpose of a user manual is to provide clear instructions and support to users. Tailor the manual to meet the specific needs of your system and its users.