Project Report

Group 14
COMP2021 Object-Oriented Programming (Fall 2023)
Author: LIN Zhanzhi
Other group members:
Zirui KONG
Raynell Soon Hong LU
Alex ZHEN ZHANG

1 Introduction [1]

This document outlines the design and implementation of a task management system with a graphical user interface, developed by group 14. The project was undertaken as part of the COMP2021 Object-Oriented Programming course at PolyU.

2 My Contribution and Role Distinction [1]

I made a substantial contribution to the design and programming of the project. In comparison to other group members, I took on the responsibility of programming for the TMS and successfully implemented a majority of the required functionalities.

GenAI tools have been utilized in the preparation of this personal report. They have been employed to revise the draft report written by the author, enhancing the readability, formality, and fluency of the report. The revised texts have been carefully reviewed to ensure their alignment with the claims made in the "My Contribution and Role Distinction" section. Additionally, the GenAI tools have also been employed to revise that section, and the resulting revisions have been verified for factual accuracy and agreement among all group members. [1]

Sections revised by GenAI tools are cited in the section headers.

3 A Task Management System with a GUI [1]

In this section, I would like to present my understanding of the system's overall design and provide details regarding its implementation. [1]

3.1 Design [1]

According to the class diagram (Diagram 1), the system design adheres to the Model-View-Controller (MVC) pattern. In this pattern, the Application class serves as the View component, the TMS class acts as the Controller component, and all the other classes function as the Model components.

The Application, which is essentially a GUI, facilitates communication between the user and the TMS. It handles the retrieval and delivery of input commands from the user to the TMS, as well as the retrieval and display of execution messages from the TMS to the user.

The Controller, represented by the TMS, identifies function calls and arguments from the command, assigns tasks to the corresponding Model component, and relays execution messages back to the user. Additionally, the TMS also manages the program's state, supporting functionalities such as saving the current state, loading the program state from a formatted file source, and reverting to previous states.

The Model components, encompassing the remaining classes, receive arguments from the Controller and execute tasks through mutual communication. They also provide feedback through execution messages.

Overall, the system design effectively implements the MVC pattern, ensuring clear separation of concerns and promoting modularity and maintainability. [1]

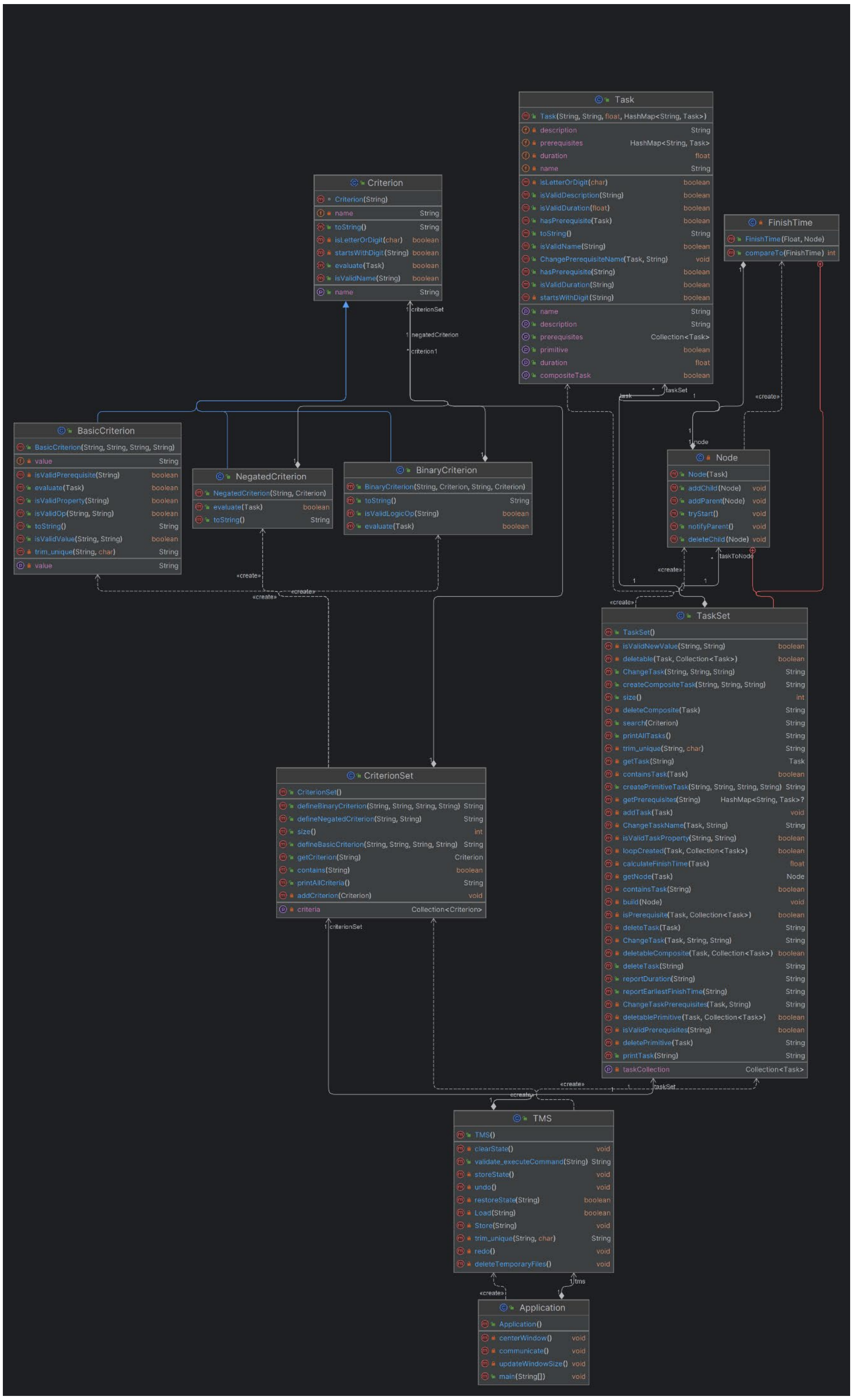
Task
Task(String, String, float, HashMap<String, Task>)
description String
prerequisites HashMap<String, Task>
duration float
name String
isLetterOrDigit(char) boolean
isValidDescription(String) boolean
isValidDuration(float) boolean
hasPrerequisite(Task) boolean
toString() String
isValidName(String) boolean
ChangePrerequisiteName(Task, String) void
hasPrerequisite(String) boolean
isValidDuration(String) boolean
startsWithDigit(String) boolean
name String
description String
prerequisites Collection<Task>
primitive boolean
duration float
compositeTask boolean
Criterion
Criterion(String)
name String
toString() String
isLetterOrDigit(char) boolean
startsWithDigit(String) boolean
evaluate(Task) boolean
isValidName(String) boolean
name String
FinishTime
FinishTime(Float, Node)
compareTo(FinishTime) int
1 criterionSet
1 negatedCriterion
* criterion1
task
* taskSet
1
«create»
1
1 node
BasicCriterion
BasicCriterion(String, String, String, String)
value String
isValidPrerequisite(String) boolean
evaluate(Task) boolean
isValidProperty(String) boolean
isValidOp(String, String) boolean
toString() String
isValidValue(String, String) boolean
trim_unique(String, char) String
value String
NegatedCriterion
NegatedCriterion(String, Criterion)
evaluate(Task) boolean
toString() String
BinaryCriterion
BinaryCriterion(String, Criterion, String, Criterion)
toString() String
isValidLogicOp(String) boolean
evaluate(Task) boolean
Node
Node(Task)
addChild(Node) void
addParent(Node) void
tryStart() void
notifyParent() void
deleteChild(Node) void
* taskToNode
«create»
1
1
«create»
«create»
«create»
«create»
TaskSet
TaskSet()
isValidNewValue(String, String) boolean
deletable(Task, Collection<Task>) boolean
ChangeTask(String, String, String) String
createCompositeTask(String, String, String) String
size() int
deleteComposite(Task) String
search(Criterion) String
printAllTasks() String
trim_unique(String, char) String
getTask(String) Task
containsTask(Task) boolean
createPrimitiveTask(String, String, String, String) String
getPrerequisites(String) HashMap<String, Task>?
addTask(Task) void
ChangeTaskName(Task, String) String
isValidTaskProperty(String, String) boolean
loopCreated(Task, Collection<Task>) boolean
calculateFinishTime(Task) float
getNode(Task) Node
containsTask(String) boolean
build(Node) void
isPrerequisite(Task, Collection<Task>) boolean
deleteTask(Task) String
ChangeTask(Task, String, String) String
deletableComposite(Task, Collection<Task>) boolean
deleteTask(String) String
reportDuration(String) String
reportEarliestFinishTime(String) String
ChangeTaskPrerequisites(Task, String) String
deletablePrimitive(Task, Collection<Task>) boolean
isValidPrerequisites(String) boolean
deletePrimitive(Task) String
printTask(String) String
taskCollection Collection<Task>
CriterionSet
CriterionSet()
defineBinaryCriterion(String, String, String, String) String
defineNegatedCriterion(String, String) String
size() int
defineBasicCriterion(String, String, String, String) String
getCriterion(String) Criterion
contains(String) boolean
printAllCriteria() String
addCriterion(Criterion) void
criteria Collection<Criterion>
1 criterionSet
«create»
«create»
1 1 taskSet
1
TMS
TMS()
clearState() void
validate_executeCommand(String) String
storeState() void
undo() void
restoreState(String) boolean
Load(String) boolean
Store(String) void
trim_unique(String, char) String
redo() void
deleteTemporaryFiles() void
«create»
1 tms
1
Application
Application()
centerWindow() void
communicate() void
updateWindowSize() void
main(String[]) void

Diagram 1 System Class Diagram

3.2 Requirements

[REQ1]

1) The requirement is implemented.
2) First, the arguments are validated. Then they are processed to feed to the Task constructor for a new task. Afterward, the new task is registered in taskSet.
3) Error cases are handled as follows:
   - ➢ If the given name, description, or duration is invalid, the system halts its execution and reports the error to the user.
   - ➢ If the given name is already occupied by another task, the system stops working and notifies the user about the name conflict.
   - ➢ If not all the given prerequisites are defined, the system ceases its operation and informs the user about the missing prerequisites.

   These error cases are effectively handled by the system, ensuring that any issues or inconsistencies are promptly identified and reported to the user, thereby maintaining the integrity and accuracy of the task management process.

[REQ2]

1) The requirement is implemented.
2) The implementation process involves validating the arguments provided by the user. Once validated, the arguments are processed and passed to the Task constructor to create a new task. Subsequently, the newly created task is registered in the taskSet.
3) Error cases are handled as follows:
   - ➢ If the given name or description is invalid, the system halts its execution and reports the error to the user.
   - ➢ If the given name is already occupied by another task, the system stops working and notifies the user about the name conflict.
   - ➢ If not all the given subtasks are defined, the system ceases its operation and informs the user about the missing subtasks.

   These error cases are effectively tackled by the system, ensuring that any issues or inconsistencies are promptly identified and reported to the user, thereby maintaining the integrity and accuracy of the task management process.

[REQ3]

1) The requirement is implemented.
2) The implementation process involves validating the arguments provided by the user. Once validated, the system checks the deletability of the task. If the task is deletable, it is deleted following some rules below.
3) Error cases are handled as follows:
   - ➢ If the given name is invalid, the system halts its execution and reports the error to the user.
   - ➢ If the given name does not correspond to any existing task, the system stops working and notifies the user about the non-existent task.
   - ➢ If the given task is not deletable, the system ceases its operation and informs the user about the inability to delete the task.

   The deletion process follows the following rules:

   - ♦ Deleting a primitive task results in the erasure of the task itself.

- ♦ Deleting a composite task leads to the erasure of the task and all its direct subtasks. In the case of a composite subtask, the deletion of direct subtasks may also result in the erasure of some of its indirect subtasks.

A task is considered deletable if and only if deleting it, according to the above rules, would not result in any remaining tasks that cannot be completed (incompletable tasks).

The deletion process ensures that all relevant tasks, including direct and indirect subtasks, are erased according to the specified rules. The system also checks for any incompletable remaining tasks, which may arise due to non-existent prerequisites or subtasks, or if a task has itself as a prerequisite or subtask. These error cases are effectively handled, ensuring the integrity and consistency of the task management process.

[REQ4]

1) The requirement is implemented.
2) The implementation of this requirement involves several steps. First, the arguments provided are validated to ensure they are in the correct format. Then, the system checks if the new value is changeable for the specified property.
   For a primitive task, the properties that can be changed are name, description, duration, or prerequisites. For a composite task, the properties that can be changed are name, description, or subtasks. The new value must be compatible with the corresponding property.
   - ➢ To change the name of a task, the system checks if the new name is already occupied by another task. If not, the system updates the name of the task in the taskSet registry. Additionally, any other tasks that have the current task as their direct prerequisite or subtask need to be updated to reference the task by its new name.
   - ➢ To change the description or duration of a task, the system only needs to validate the new value and update the corresponding field inside the task itself.
   - ➢ To change the prerequisites or subtasks of a task, the system checks if the new prerequisites or subtasks would result in any incompletable tasks. This is done by assuming the new prerequisites or subtasks and using Depth First Search to determine if the task will become a direct or indirect prerequisite or subtask of itself, under the new prerequisites or subtasks. If no incompletable tasks are detected, the system updates the corresponding field in the taskSet registry.
3) Error cases are handled as follows:
   - ➢ If the task name, property, or new value is not compatible with the command, the system halts its execution and reports the error to the user.
   - ➢ If the given name does not correspond to any existing task, the system stops working and notifies the user about the non-existent task.
   - ➢ If the new name is already occupied by another task, the system will stop working and report the error.
   - ➢ If the new prerequisites or subtasks would result in incompletable tasks, the system will stop working and report the error.

The modification process ensures that relevant tasks, the modified task, and those who have it as their direct prerequisite or subtask, are updated according to the above schedule. The error cases are effectively handled, ensuring the integrity and consistency of the task management process.

[REQ5]

1) The requirement is implemented.

2) The implementation process involves validating the arguments provided by the user. Once validated, the system retrieves the information of the task and prints it in an easy-to-read format, with each attribute on a separate line.
3) Error cases are handled as follows:
    - ➢ If the given argument is invalid, the system halts its execution and reports the error to the user.
    - ➢ If the given name does not correspond to any existing task, the system stops working and notifies the user about the non-existent task.

The printing functionality ensures that the information of a task is displayed in a clear and readable manner. Error cases are effectively handled, ensuring that the user is informed about any issues or inconsistencies in the provided arguments.

[REQ6]

1) The requirement is implemented.
2) The implementation process involves printing the information of all registered tasks. The system displays the total number of existing tasks and proceeds to print the details of each individual task.
3) There are no error cases associated with this requirement. The system successfully prints the information of all tasks without encountering any issues or errors.

This implementation ensures that users can easily access and view the details of all tasks, providing a comprehensive overview of the task management system.

[REQ7]

1) The requirement is implemented.
2) The implementation process begins with the validation of the provided arguments. Once validated, the system proceeds to calculate the duration of the specified task. For primitive tasks, the duration is stored during their definition. However, for composite tasks, the duration is determined by constructing a task graph that includes all direct and indirect subtasks. The system then simulates the multitasking process, starting tasks as soon as their prerequisites are completed. By considering the dependencies and scheduling, the minimum amount of hours needed to finish all subtasks is calculated as the duration of the composite task.
3) Error cases are handled as follows:
    - ➢ If the given name is invalid, the system halts its execution and reports the error to the user.
    - ➢ If the given name does not correspond to any existing task, the system stops working and notifies the user about the non-existent task.

The implementation ensures accurate reporting of the task duration, considering the complexity of composite tasks and their subtasks. By considering the prerequisite relationships and scheduling, the system accurately calculates the minimum duration required to complete a composite task. Error cases are effectively handled, providing reliable and informative user experience.

[REQ8]

1) The requirement is implemented.
2) The implementation process involves validating the arguments provided by the user. The earliest finish time of the task is calculated as the sum of: the earliest finish time of all its prerequisites, and the duration of the task itself. For a composite task, the earliest finish time is equal to its duration. For a primitive task, the earliest finish time is calculated using a simulation approach.
3) Error cases are handled as follows:

- ➢ If the given name is invalid, the system halts its execution and reports the error to the user.
- ➢ If the given name does not correspond to any existing task, the system stops working and notifies the user about the non-existent task.

The system accurately calculates and reports the earliest finish time of a task, considering its prerequisites and duration. Error cases are effectively handled, ensuring the integrity and accuracy of the task management process.

[REQ9]

1) The requirement is implemented.
2) The implementation process begins with the validation of the provided arguments. Once validated, the system processes the arguments to create a new basic criterion using the basic criterion constructor. This newly created criterion is then registered in the registry for future reference.
3) Error cases are handled as follows:
   - ➢ If the given name is invalid, the system halts its execution and reports the error to the user.
   - ➢ If the given name is already occupied by an existing criterion, the system stops working and notifies the user about the name conflict.

By effectively validating the arguments and processing them accordingly, the system ensures the successful creation and registration of basic task selection criteria. Error cases are handled promptly, preventing any conflicts or inconsistencies in the criterion management process.

[REQ10]

1) The requirement is implemented.
2) During the initialization of the Task Management System (TMS), the Controller component creates the IsPrimitive criterion. This criterion is implemented as a basic criterion with a condition that checks if the duration of a task is greater than 0. If the condition is met, the IsPrimitive criterion evaluates to true, indicating that the task is primitive.
3) There are no error cases associated with this requirement. The IsPrimitive criterion is straightforward and does not involve any potential errors or exceptions. It simply evaluates the duration of a task to determine if it is primitive or not.

[REQ11]

1) The requirement is implemented.
2) The implementation process involves validating the arguments provided by the user. Once validated, the system creates a new criterion based on the given arguments and registers it.
3) Error cases are handled as follows:
   - ➢ If the name of the new criterion is invalid, the system halts its execution and reports the error to the user.
   - ➢ If the negated criterion or the two criteria of the binary criterion are not defined, the system stops working and notifies the user about the missing criteria.

These error cases are effectively handled, ensuring that any issues or inconsistencies are promptly identified and reported to the user, thereby maintaining the integrity and accuracy of the composite task selection criteria definition process.

[REQ12]

1) The requirement is implemented.
2) The implementation process involves collecting the defined criteria from the registry, criterionSet, and outputting their information one by one. For criteria that are composed of other criteria, the printing is done recursively until it resolves to the information of a basic

criterion. The output format ensures that all criteria are resolved to the form containing only property names, operators, values, logical operators.

3) There are no error cases associated with this requirement. The system simply retrieves and prints all the defined criteria, ensuring that the information is accurately displayed to the user.

[REQ13]

1) The requirement is implemented.
2) The implementation process involves iterating through all registered tasks and testing them against the given search criterion. When a task satisfies the criterion, it is displayed to the user.
3) Error cases are handled as follows:
   - If the search criterion is not defined, the system halts its execution and reports the error to the user.

   The search functionality effectively filters and displays tasks that meet the specified criterion, providing users with a convenient way to find and access relevant tasks. The system ensures that any issues related to undefined search criteria are promptly identified and reported to the user, maintaining the usability and accuracy of the search feature.

[REQ14]

1) The requirement is implemented.
2) The implementation process involves checking the existence of the file at the given path. If the file exists, the current program state, including the taskSet and criterionSet, is dumped into the file. If the file does not exist, it is created before dumping.
3) Error cases are handled as follows:
   - If the given path is invalid or the file does not exist, the system throws an exception and halts its execution. The error is reported to the user, indicating the invalid path.

   By ensuring the existence of the file and properly dumping the program state, the system effectively stores the defined tasks and criteria into the specified file, allowing for persistence and future retrieval of the data.

[REQ15]

1) The requirement is implemented.
2) The implementation process involves first checking the existence of the file at the specified path. If the file exists, the system attempts to read the current program state, including the taskSet and criterionSet, from the file.
3) Error cases are handled as follows:
   - If the given path is invalid or the file does not exist, the system throws an exception, halts its execution, and reports the error to the user.
   - If the file is wrongly organized or formatted, the system stops working and notifies the user about the issue.

   These error cases ensure that the system can effectively handle situations where the file path is incorrect or the file itself is not properly structured. By providing appropriate error messages, the user is informed about any issues encountered during the loading process, allowing for prompt resolution, and ensuring the successful loading of tasks and criteria from the file.

[REQ16]

1) The requirement is implemented.
2) Upon receiving the "Quit" command, the application performs the following actions:
   - Deletes all temporary files that were used to facilitate program state rollback.
   - Terminates the execution of the system.

3) There are no error cases associated with this requirement. The termination process is straightforward and does not involve any potential issues or error handling.
   The implementation ensures a smooth and efficient termination of the system, allowing the user to gracefully exit the application without encountering any errors or complications.

[BON1]

1) The requirement is implemented.
2) At the start of the program, a graphical user interface (GUI) is created. The GUI consists of a frame with a pane that contains three components. The first component is a text area that displays input commands and output messages. The second component is a text box where users can enter commands. The third component is a button that triggers communication with the Task Management System (TMS) using the entered command.
3) The GUI includes functionality to handle the following:
   - ➢ When the number of messages exceeds the capacity of the result area, a scroll bar is automatically added and kept at the bottom, ensuring that the latest output is always visible.
   - ➢ If the user adjusts the dimensions of the frame, the layout of the three components is adjusted simultaneously, maintaining a visually pleasing and responsive interface.

   The GUI provides a user-friendly and interactive way to interact with the TMS, allowing users to easily view and manage tasks and their relationships. The implemented features enhance the usability and functionality of the Task Management System.

[BON2]

1) The requirement is implemented.
2) To support the undo and redo functionality for all required commands except for PrintTask, PrintAllTasks, ReportDuration, ReportEarliestFinishTime, PrintAllCriteria, Search, Store, Load, and Quit, the program utilizes temporary .data files to store program states after each successful modification to the program state. Two stacks are employed to keep track of the previous and next states by storing the file path of the corresponding .data file.
   - ➢ During an undo operation, the program takes the current state, which is at the top of the undo stack, and moves it to the top of the redo stack. It then restores the state stored at the top of the undo stack, effectively undoing the previous modification.
   - ➢ Conversely, during a redo operation, the program restores the top state from the redo stack and makes it the top of the undo stack.
   - ➢ Additionally, when a modification of the program state is triggered, the redo stack is cleared since the program state develops in a different branch.
3) Error cases are handled as follows:
   - ➢ If the undo stack only contains the current state, the program rejects undoing and reports this to the user.
   - ➢ If the redo stack is empty, the program stops redoing and notifies the user about the lack of redo operations available.

   These error cases ensure that the undo and redo functionality operate smoothly and provide the user with accurate and reliable state restoration options.

References

[1] G.-3. PolyU GenAI, "PolyU GenAI," OpenAI, [Online]. Available: https://genai.polyu.edu.hk/. [Accessed 25 11 2023].