Project Report

Group 2
COMP2021 Object-Oriented Programming (Fall 2023)
Author: Dominicus Dylan HARYOTO
Other group members:
Shusen ZHENG
Bohan YANG
Qinye ZHANG

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

My key contribution to the project was developing the structure for the Task, PrimitiveTask, and CompositeTask classes, which form the project's core functionality. Furthermore, I successfully fulfilled primary requirements REQ1, REQ2, REQ3, and REQ4.

In addition to these fundamental contributions, I introduced the idea of utilizing the HashMap data structure as the main structure for the project. This choice proved to be pivotal in facilitating efficient data management and access throughout the project's lifecycle. By using HashMap, it is guaranteed that the data input and access take better time complexity than using other data structures.

In collaboration with ZHENG Sushen, we implemented the bonus 'undo' and 'redo' features using a Stack data structure. We introduced two stacks, one for storing the undo commands and another one for storing the redo commands. Every time the user inputs undo after their command, it will store the negation of their inputted command to the redo stack. Therefore, when the user decides to redo, we can directly pop the top stack of the redo stack. At the same time, the popped value from the redo stack will be pushed to the undo stack.

One of my critical non-coding contributions was creating the user manual report. I ensured it was readable and user-friendly, and provided comprehensive guidance to end-users. Additionally, I contributed to the creation of our presentation slides.

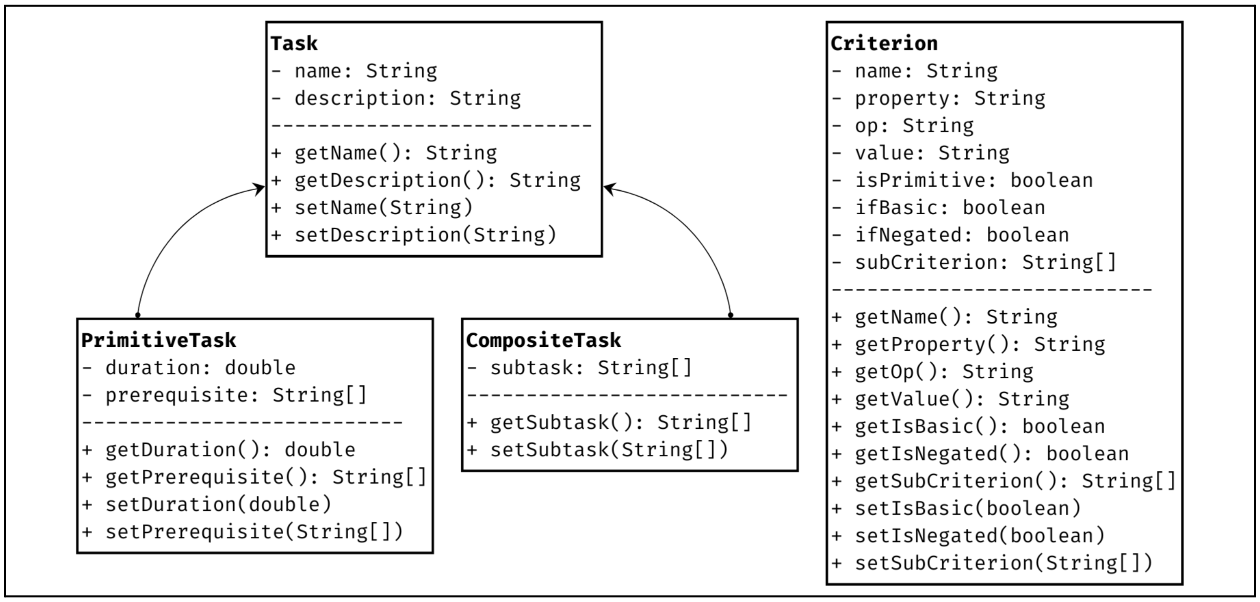
On the project management front, I led weekly team meetings and maintained robust team communication. I also led the planning and execution of our project presentation and video, ensuring our efforts were well-communicated and well-received.

Crucially, to make our code more understandable, I leveraged GenAI to create clear and concise Javadocs. This tool greatly simplified my documentation efforts, making the content straightforward and brief, which significantly benefitted our development process and end-user comprehension.

3. A Command-Line Task Management System

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

3.1 Design



The picture above illustrates the UML diagram of our Command-Line Task Management System. We have four classes as the main structure of our project:

1. **Task**
   The Task class represents a task with a name and description. It is the parent class of the classes PrimitiveTask and CompositeTask. It has the following components:

   Fields
   - private String name → holds the name of the task
   - private String description → holds the description of the task

   Methods
   - public String getName() → returns the current value of the name field
   - public String getDescription() → returns the current value of the description field
   - public void setName(String) → changes the value of the name field
   - public void getDescription(String) → changes the value of the description field

2. **PrimitiveTask**
   The PrimitiveTask class is a subclass of the Task class and represents a basic task in the Task Management System (TMS). It extends Task, meaning it inherits the name and description fields, as well as their associated getter and setter methods.

   Fields
   - private double duration → holds the duration of the task
   - private String[] prerequisite → holds the prerequisite of the task

Methods
- public double getDuration() → returns the current value of the duration field
- public String[] getPrerequisite() → returns the current value of the prerequisite field
- public void setDuration(double) → changes the value of the duration field
- public void setPrerequisite(String[]) → changes the value of the prerequisite field

3. **CompositeTask**
The CompositeTask class is a subclass of the Task class and represents a composite task in the Task Management System (TMS). It extends Task, meaning it inherits the name and description fields, as well as their associated getter and setter methods.

Fields
- private String[] subtask → holds the subtask of the task

Methods
- public String[] getSubtask() → returns the current value of the subtask field
- public void setSubtask(String[]) → changes the value of the subtask field

4. **Criterion**
The Criterion class represents a criterion in the Task Management System (TMS).

Fields
- private String name → holds the name of the criterion
- private String property → holds the property of the criterion
- private String op → holds the operation of the criterion with respect to the property
- private String value → holds the value of the criterion with respect to the operation
- private boolean isPrimitive → holds a flag indicating whether the criterion is of a primitive type
- private boolean isBasic → holds a flag indicating whether the criterion is basic
- private boolean isNegated → holds a flag indicating whether the criterion is negated
- private String[] subCriterion → holds the sub-criteria of the criterion

Methods
- public String getName() → returns the current value of the name field
- public String getProperty() → returns the current value of the property field
- public String getOp() → returns the current value of the operation field with respect to the property
- public String getValue() → returns the current value of the property field with respect to the operation
- public boolean getIsPrimitive() → returns the current state of whether the criterion is of a primitive type
- public boolean getIsBasic() → returns the current state of whether the criterion is basic

- public boolean getIsNegated() → returns the current state of whether the criterion is negated
- private String[] getSubCriterion → returns the current value of the subCriterion field

We have three HashMaps as the structure of the Command-Line Task Management System.

primitiveTask	**compositeTask**	**criterion**
`<String, PrimitiveTask>`	`<String, CompositeTask>`	`<String, Criterion>`
Stores a list of SimpleTask	Stores a list of CompositeTask	Stores a list of Criterion
key: name	key: name	key: name
value: a PrimitiveTask	value: a CompositeTask	value: a Criterion

Since inheritance is applied here, we have the getter method to retrieve the current values of primitiveTask, compositeTask, and criterion:

1. **getPrimitiveTask(): HashMap<String, PrimitiveTask>**
   This method returns the existing 'primitiveTask' HashMap that maps a string to a 'PrimitiveTask' object.

```
public HashMap<String, PrimitiveTask> getPrimitiveTask() {
     return primitiveTask;
}
```

2. **getCompositeTask(): HashMap<String, CompositeTask>**
   This method returns the existing 'compositeTask' HashMap that maps a string to a 'CompositeTask' object.

```
public HashMap<String, CompositeTask> getCompositeTask() {
     return compositeTask;
}
```

3. **getCriterion(): HashMap<String, Criterion>**
   This method returns the existing 'criterion' HashMap that maps a string to a 'Criterion' object.

```
public HashMap<String, Criterion> getCriterion() {
     return criterion;
}
```

3.2 Requirements

[REQ1]

1) Requirement Implemented
2) Implementation Details

The method createPrimitiveTask accepts six parameters - name, description, duration, prerequisite, undoRedo, and undo. Here is a detailed breakdown of how the method works:

a) Task Existence Check: The method first checks if a task with the given name already exists in the primitiveTask or compositeTask maps (using containsKey). If it exists, it prints an error message and returns, preventing the creation of a duplicate task.
b) Validation of name: The method checks if the name is longer than 8 characters and starts with a digit. If either condition is true, it prints an error message and returns. It also checks that the name only contains English letters and digits - if it doesn't, it prints an error message and returns.
c) Validation of description: It checks that the description only contains English letters, digits, and hyphens. If it doesn't, it prints an error message and returns.
d) Validation of duration: It tries to parse duration as a double. If it cannot, it prints an error message and returns. It also checks that the duration is a positive value. If it isn't, it prints an error message and returns.
e) Validation of prerequisite: It splits the prerequisite string into an array of strings (using split), then checks that each prerequisite task exists in the primitiveTask or compositeTask HashMap. If a prerequisite does not exist, it prints an error message and returns.
f) Task Creation: If all validations pass, it creates a new PrimitiveTask object with the given parameters and adds it to the primitiveTask HashMap (using put).
g) Undo/Redo Stack Update: Depending on the states of the undoRedo and undo flags, it adds the appropriate command to the undo or redo stack. This is used to implement undo/redo functionality.

3) Error conditions and how they are handled
Error conditions are handled by checking for specific conditions and printing an error message if the condition is not met. In all cases, the method also returns without creating a new task. These conditions include:
- A task with the given name already exists.
- The name is longer than 8 characters, starts with a digit, or contains characters other than English letters and digits.
- The description contains characters other than English letters, digits, and hyphens.
- The duration cannot be parsed as a double or is negative.
- The prerequisite string contains one or more tasks that do not exist in the primitiveTask or compositeTask HashMaps.

[REQ2]

1) Requirement Implemented
2) Implementation Details
The createCompositeTask method accepts five parameters - name0, description, subtask, undoRedo, and undo. Here is a detailed breakdown of how the method works:

a) Task Existence Check: The method first checks if a task with the given name0 already exists in the primitiveTask or compositeTask maps (using containsKey). If it exists, it prints an error message and returns, preventing the creation of a duplicate task.
b) Validation of name0: The method checks if the name0 is longer than 8 characters and starts with a digit. If either condition is true, it prints an error message and returns. It also checks that the name0 only contains English letters and digits - if it doesn't, it prints an error message and returns.
c) Validation of description: It checks that the description only contains English letters, digits, and hyphens. If it doesn't, it prints an error message and returns.
d) Validation of subtask: It splits the subtask string into an array of strings (using split), then checks that each subtask exists in the primitiveTask or compositeTask maps. If a subtask does not exist, it prints an error message and returns.
e) Task Creation: If all validations pass, it creates a new CompositeTask object with the given parameters and adds it to the compositeTask HashMap (using put).
f) Undo/Redo Stack Update: Depending on the states of the undoRedo and undo flags, it adds the appropriate command to the undo or redo stack. This is used to implement undo/redo functionality.

3) Error conditions and how they are handled
Error conditions are handled by checking for specific conditions and printing an error message if the condition is not met. In all cases, the method also returns without creating a new task. These conditions include:
- A task with the given name0 already exists.
- The name0 is longer than 8 characters, starts with a digit, or contains characters other than English letters and digits.
- The description contains characters other than English letters, digits, and hyphens.
- The subtask string contains one or more tasks that do not exist in the primitiveTask or compositeTask maps.

[REQ3]

1) Requirement Implemented
2) Implementation Details
The deleteTask method accepts three parameters - name, undoRedo, and undo. Here is a detailed breakdown of how the method works:
a) Task Existence Check: The method first checks if a task with the given name exists in the primitiveTask or compositeTask HashMaps (using containsKey).
b) Deletion of a Primitive Task: If the task exists in primitiveTask, it checks if this task is a prerequisite for any other tasks or a subtask in any composite task. If it is a prerequisite for any other tasks, the method prints an appropriate message and returns false. If the task is not a prerequisite, it is removed from primitiveTask and subtask of compositeTask(s) that have that particular task, and a corresponding CreatePrimitiveTask command is pushed to the undo or redo stack depending on the state of undoRedo and undo flags.

c) Deletion of a Composite Task: If the task exists in compositeTask, it attempts to delete all subtasks of this task. If a subtask cannot be deleted (because it has prerequisites), the method prints an appropriate message and returns false. If all subtasks can be deleted, the composite task is removed from compositeTask and a corresponding CreateCompositeTask command is pushed to the undo or redo stack.
d) Task Non-existence: If the task does not exist in either primitiveTask or compositeTask, the method prints an appropriate message and returns false.

After a change is successfully made, the method pushes a corresponding ChangeTask command onto the undo or redo stack depending on the state of undoRedo and undo flags. It also prints a success message.

3) Error conditions and how they are handled
Error conditions are handled by checking for specific conditions and printing an error message if the condition is not met. These conditions include:
   - A task with the given name does not exist.
   - The task is a prerequisite for one or more other tasks.
   - A composite task contains one or more subtasks that cannot be deleted because they have prerequisites.

[REQ4]

1) Requirement Implemented
2) Implementation Details
The changeTask method accepts five parameters - name, property, newValue, undoRedo, and undo. Here's a detailed breakdown of how the method works:
   a) Task Existence Check: The method first checks whether a task with the given name exists in the primitiveTask or compositeTask HashMaps.
   b) Modify Primitive Task: If the task is a primitive task, it checks the property parameter to determine which property to change. The properties that can be changed are "name", "description", "duration", and "prerequisites". The method has appropriate error checks to handle incorrect inputs like a new name that already exists, a duration that is not a positive real number, or a prerequisite that doesn't exist.
   c) Modify Composite Task: If the task is a composite task, it checks the property parameter to determine which property to change. The properties that can be changed are "name", "description", and "subtasks". The method has appropriate error checks to handle incorrect inputs like a new name that already exists, or a subtask that doesn't exist.
   d) Task Non-existence: If the task does not exist in either primitiveTask or compositeTask, the method prints a message indicating that the task does not exist.
3) Error conditions and how they are handled
Error conditions are handled by checking for specific conditions and printing an error message if the condition is not met. These conditions include:

- A task with the given name does not exist.
- The property to be changed does not exist.
- The new value of a property is invalid (e.g., a new name that already exists, a duration that is not a positive real number, a prerequisite or subtask that doesn't exist).

[REQ5]

1) Requirement Implemented
2) Implementation Details
   The printTask method accepts one parameter - taskName. Here's a detailed breakdown of how the method works:
   a) Task Existence Check: The method first checks if a task with the given taskName exists in the primitiveTask or compositeTask maps.
   b) Print Primitive Task: If the task is a primitive task, it prints the task name, description, duration, and prerequisites. If the task does not have any prerequisites, it prints "null".
   c) Print Composite Task: If the task is a composite task, it prints the task name, description, and subtasks. If the task does not have any subtasks, it prints "null".
   d) Task Non-existence: If the task does not exist in either primitiveTask or compositeTask, the method prints a message indicating that the task does not exist.
3) Error conditions and how they are handled
   Error conditions are handled by checking for specific conditions and printing an error message if the condition is not met. These conditions include:
   - A task with the given taskName does not exist.

[REQ6]

1) Requirement Implemented
2) Implementation Details
   The printAllTasks method does not accept any parameters. Here's a detailed breakdown of how the method works:
   a) Printing Primitive Tasks: The method iterates over the keys of the primitiveTask map and calls printTask on each key. This prints the details of each primitive task in the system.
   b) Printing Composite Tasks: The method iterates over the keys of the compositeTask map and calls printTask on each key. This prints the details of each composite task in the system.
3) Error conditions and how they are handled
   Since the printAllTasks method only prints the details of tasks that are already in the system, there aren't any error conditions that need to be handled in this method

[REQ7]

1) Requirement Implemented

2) Implementation Details
   The reportDuration method accepts one parameter - taskName. It uses several helper methods, and here's a detailed breakdown of how the method works:
   a) getDuration: This method accepts a task name and returns its duration. If the task is a primitive task, it directly returns the duration from the primitiveTask map. If it's a composite task, it calculates the total duration by calling getCompositeTaskDuration.
   b) getCompositeTaskDuration: This method calculates the total duration of a composite task. It breaks the task into its primitive subtasks using breakTasksIntoPrimitiveTasks and calculates the maximum duration among them.
   c) breakTasksIntoPrimitiveTasks and breakTask: These methods are used to break a composite task into its primitive subtasks recursively.
   d) calculateDuration: This method uses recursion to calculate the total duration of a task, taking into account its prerequisites.
3) Error conditions and how they are handled
   Error conditions are handled by checking for specific conditions and printing an error message if the condition is not met. These conditions include:
   - A task with the given name does not exist.

[REQ8]

1) Requirement Implemented
2) Implementation Details
   The reportEarliestFinishTime method accepts one parameter - taskName. It uses several helper methods, and here's a detailed breakdown of how the method works:
   a) reportEarliestFinishTime: This method accepts a task name as an argument and prints the earliest finish time of the task by calling getEarliestFinishTime.
   b) getEarliestFinishTime: This method accepts a task name. If the task is a primitive one, it checks whether it has any prerequisites. If not, it returns the task's duration. If it has prerequisites, it calculates the maximum finish time among its prerequisite tasks and adds it to the task's duration. If the task is a composite one, it calculates the maximum finish time among its subtasks.
   c) handleSingleTaskForInput: This method handles the case where a task has no prerequisites. It simply returns the duration of the task.
3) Error conditions and how they are handled
   Error conditions are handled by checking for specific conditions and printing an error message if the condition is not met. These conditions include:
   - A task with the given name does not exist.

[REQ9]

1) Requirement Implemented
2) Implementation Details

The defineBasicCriterion method accepts six parameters - name1, property, op, value, undoRedo, and undo. Here's a detailed breakdown of how the method works:

a) Task/Criterion Existence Check: The method first checks if a task or criterion with the given name already exists. If so, it prints a message indicating that the name is already in use.
b) Name Validation: The method validates the name of the criterion. It checks if the name is longer than 8 characters, starts with a digit, or contains characters other than English letters and digits. If any of these conditions are not met, it prints an error message.
c) Property and Operation Validation: The method validates the property and operation. If the property is "name", "description", "prerequisites", or "subtasks", the operation must be "contains". If the property is "duration", the operation must be one of "<", ">", "<=", ">=", "==", "!=". If these conditions are not met, it prints an error message.
d) Duration Value Validation: If the property is "duration", it checks if the value can be parsed into a double. If not, it prints an error message.
e) Criterion Definition: If all validation checks pass, it creates a new Criterion object with the given parameters, sets it as a basic criterion (not negated), and adds it to the criterion map. It then prints a success message.
f) Undo/Redo Handling: Lastly, it adds an action to the undo or redo stack depending on the boolean parameters.

3) Error conditions and how they are handled
Error conditions are handled by checking for specific conditions and printing an error message if the condition is not met. These conditions include:
   - A task or criterion with the given name already exists.
   - The name is longer than 8 characters, starts with a digit, or contains characters other than English letters and digits.
   - The property or operation is not valid.
   - The value cannot be parsed into a double (for "duration" property).

[REQ10]

1) Requirement Implemented
2) Implementation Details
The IsPrimitive criterion is a special type of criterion that is set up when the system runs. This criterion does not receive any parameters. Here's a detailed breakdown of how it is initialized:
a) Criterion Creation: The Criterion object is created on certain specific occasions with only one parameter - name.
b) IsPrimitive Setup: If the name is "IsPrimitive", the constructor sets up the criterion as a basic, non-negated, primitive criterion with no property and value. The op is also set to "IsPrimitive".
c) System Run: When the system runs, IsPrimitive will be set subsequently.
3) Error conditions and how they are handled

Since it is a system-predefined criterion and does not accept any parameters from the user, there aren't any error conditions that need to be handled in this requirement.

[REQ11-1]

1) Requirement Implemented
2) Implementation Details
   The defineNegatedCriterion method accepts four parameters - name1, name2, undoRedo, and undo. Here's a detailed breakdown of how the method works:
   a) Existence Check: The method first checks if a criterion with the given name1 already exists, or if the criterion name2 does not exist. If either of these conditions is met, it prints an error message and returns.
   b) Name Validation: The method validates the name name1 of the negated criterion. It checks if the name is longer than 8 characters, starts with a digit, or contains characters other than English letters and digits. If any of these conditions are not met, it prints an error message and returns.
   c) Negation of Operation: The method retrieves the property, operation, and value of the criterion name2, and negates the operation. For example, if the operation is ">", it is changed to "<=". If the operation is "||", it is changed to "&&". If the operation is "contains", it is changed to "not-contains", and so on.
   d) IsCompos Handling: If name2 is "IsPrimitive" and name1 is not "IsCompos", it suggests to use "IsCompos" instead of name1, and a criterion "IsCompos" is created.
   e) Negated Criterion Creation: Next, it creates a new Criterion object with name1, the negated operation, and the property and value of name2. It also sets this criterion as a negated and basic criterion and adds it to the criterion map.
   f) Undo/Redo Handling: Finally, it adds an action to the undo or redo stack depending on the undoRedo and undo parameters.
3) Error conditions and how they are handled
   Error conditions are handled by checking for specific conditions and printing an error message if the condition is not met. These conditions include:
   - A criterion with name1 already exists.
   - A criterion with name2 does not exist.
   - The name name1 is longer than 8 characters, starts with a digit, or contains characters other than English letters and digits.

[REQ11-2]

1) Requirement Implemented
2) Implementation Details
   The defineBinaryCriterion method accepts six parameters - name1, name2, op, name3, undoRedo, and undo. Here's a detailed breakdown of how the method works:

    a) Existence Check: The method first checks if a criterion with the given name1 already exists, or if the criterion name2 or name3 does not exist. If any of these conditions are met, it prints an error message and returns.
    b) Name Validation: The method validates the name name1 of the binary criterion. It checks if the name is longer than 8 characters, starts with a digit, or contains characters other than English letters and digits. If any of these conditions are not met, it prints an error message and returns.
    c) Criteria Similarity Check: The method checks if name2 and name3 are the same. If they are, it prints an error message and returns.
    d) Binary Criterion Creation: Next, it retrieves the property, operation, and value of criteria name2 and name3, and combines them using the operation op. It then creates a new Criterion object with name1, the combined property, operation, and value. The new criterion is marked as non-basic and non-negated, and it is added to the criterion map.
    e) Undo/Redo Handling: Finally, it adds an action to the undo or redo stack depending on the undoRedo and undo parameters.
3) Error conditions and how they are handled
   Error conditions are handled by checking for specific conditions and printing an error message if the condition is not met. These conditions include:
    - A criterion with name1 already exists.
    - A criterion with name2 or name3 does not exist.
    - The name name1 is longer than 8 characters, starts with a digit, or contains characters other than English letters and digits.
    - The criteria name2 and name3 are the same.

[REQ12]

1) Requirement Implemented
2) Implementation Details
   The printAllCriteria method does not accept any parameters. Here's a detailed breakdown of how the method works:
    a) Printing Introduction: The method first prints a line saying "Here is the list of Criteria(s)".
    b) Printing Criteria: The method then loops through all the keys in the criterion map, which are the names of the criteria. For each criterion, it checks if the name equals "IsPrimitive" or "IsCompos". If it does, it prints just the name of the criterion.
    c) Printing Details: If the name of the criterion does not equal "IsPrimitive" or "IsCompos", it prints the name of the criterion followed by the property, value, and operation of the criterion. The property, value, and operation are retrieved from the Criterion object associated with the name in the criterion map.
3) Error conditions and how they are handled
   Since it only performs read operations on the existing data structure, there aren't any error conditions that need to be handled in this requirement.

[REQ13]

1) Requirement Implemented
2) Implementation Details
   The search method accepts one parameter - name. Here's a detailed breakdown of how the method works:
   a) Existence Check: The method first checks if a criterion with the given name exists. If it does not, it prints an error message and returns.
   b) Basic Criterion Search: If the criterion is a basic criterion, it calls the basicCriteriaSearch method with name as the argument. This method returns a list of tasks that satisfy the basic criterion. If this list is empty, it means that no tasks satisfy the criterion, and the method prints a message and returns. Otherwise, it prints the list of tasks.
   c) Binary Criterion Search: If the criterion is not a basic criterion, it is a binary criterion. The method retrieves the sub-criteria and the operation of the binary criterion. It then calls the binaryCriteriaSearch method with these as arguments. This method returns a list of tasks that satisfy the binary criterion. If this list is empty, it means that no tasks satisfy the criterion, and the method prints a message and returns. Otherwise, it prints the list of tasks.
3) Error conditions and how they are handled
   Error conditions are handled by checking for specific conditions and printing an error message if the condition is not met. These conditions include:
   - A criterion with the given name does not exist.
   - No tasks satisfy the criterion.

[REQ14]

1) Requirement Implemented
2) Implementation Details
   The store method accepts one parameter - path. Here's a detailed breakdown of how the method works:
   a) File Existence and Creation: The method first creates a File object with the given path. If the parent directory of the file does not exist, it attempts to create it. If the parent directory still does not exist after this attempt, it prints an error message and returns.
   b) Writing to File: If the parent directory exists, the method creates a PrintWriter object for the file and begins writing to it. It writes the details of each primitive task, composite task, and criterion to the file in a specific format.
      i) Primitive Tasks: For each primitive task, it writes a $ symbol, followed by the name, description, and duration of the task. If the task has any prerequisites, it writes them as a comma-separated string; otherwise, it writes NULL. It then writes another $ symbol to mark the end of the task's details.
      ii) Composite Tasks: For each composite task, it writes a % symbol, followed by the name and description of the task. If the task has any subtasks, it

writes them as a comma-separated string; otherwise, it writes null. It then writes another % symbol to mark the end of the task's details.

      iii) Criteria: For each criterion, it writes a ^ symbol, followed by a 1 if the criterion is primitive and a 0 otherwise. It then writes the name, property, operation, and value of the criterion. It writes another 1 if the criterion is primitive and a 0 otherwise, and then a ^ symbol to mark the end of the criterion's details.

   c) Error Handling: If an IOException is thrown while writing to the file, it prints a stack trace and an error message, and then returns.

3) Error conditions and how they are handled
   Error conditions are handled by checking for specific conditions and printing an error message if the condition is not met. These conditions include:
   - The parent directory of the file does not exist and cannot be created.
   - An IOException is thrown while writing to the file.

[REQ15]

1) Requirement Implemented
2) Implementation Details
   The load method accepts one parameter - path. Here's a detailed breakdown of how the method works:
   a) Resetting State: The method first resets the current state of tasks and criteria by creating new, empty HashMaps for the primitive tasks, composite tasks, and criteria.
   b) Reading from File: The method then creates a BufferedReader object for the file at the given path and begins reading from it. It reads each line, and based on the symbol at the beginning of the line ($, %, or ^), it determines whether the line contains the details of a primitive task, a composite task, or a criterion.
      i) Primitive Tasks: If the line starts with a $ symbol, it reads the name, description, and duration of a primitive task from the next few lines. It also reads the prerequisites of the task, if there are any. It then creates a new PrimitiveTask object with these details and adds it to the primitiveTask map.
      ii) Composite Tasks: If the line starts with a % symbol, it reads the name and description of a composite task from the next few lines. It also reads the subtasks of the task. It then creates a new CompositeTask object with these details and adds it to the compositeTask map.
      iii) Criteria: If the line starts with a ^ symbol, it reads the name, property, operation, and value of a criterion from the next few lines. It then creates a new Criterion object with these details and adds it to the criterion map.
   c) Error Handling: If a FileNotFoundException is thrown while reading from the file, it prints a custom error message and a stack trace. If an IOException is thrown, it prints a generic error message and a stack trace.

3) Error conditions and how they are handled
   Error conditions are handled by checking for specific conditions and printing an error message if the condition is not met. These conditions include:
   - The file at the given path does not exist.
   - An IOException is thrown while reading from the file.

[REQ16]

1) Requirement Implemented
2) Implementation Details
   The Quit command allows the user to terminate the current execution of the Task Management System (TMS). This command is implemented in the main event loop of the TMS. Here's how it works:
   a) User Input: The system uses a Scanner to read user input from the command line. After each command, the system prints TaskManagementSystem@COMP2021 ~ % and waits for the next command.
   b) Command Execution: If the input string equals "Quit", the system prints a goodbye message and then breaks from the main event loop, effectively ending the program execution.
3) Error conditions and how they are handled
   There's only one potential error condition related to the Quit command:
   - Incorrect Command: If the user enters a string that is not "Quit", the system simply ignores it and continues with the next iteration of the main event loop. There's no need to print an error message in this case, because the system allows for multiple valid commands, and any string could potentially be a valid command.

[BON1]

1) Requirement Implemented
2) Implementation Details
   The TasklyGUI class is implemented as a graphical user interface (GUI) for the Task Management System (TMS). It extends the JFrame class and includes a reference to a TMS object. Here is a detailed breakdown:
   a) Menu Setup: The class sets up five menus: Task, Duration, Criteria, Interact, and Help. Each of these menus contains several menu items that allow users to perform various operations in the TMS.
   b) Event Handling: The processinput method is invoked when the user enters a command in the text field and presses the Confirm button or the Enter key. This method takes the user's command as a string, splits it into words, and calls the appropriate method from the TMS object based on the first keyword in the command.

   c) Help Dialogs: The UserManual and TeamMember methods are invoked when the user selects the User Manual or Info menu item, respectively. These methods display a new dialog box with relevant information.

3) Error conditions and how they are handled
   This class handles several types of errors and exceptions:
   - Unrecognized Command: If the user enters a command that is not recognized, the processinput method prints an error message to the textarea.
   - IOException: If an IOException occurs while writing to the CustomOutputStream, the write method of the CustomOutputStream class doesn't handle it but instead lets it propagate up the call stack. This exception should be caught and handled by the calling code.
   - Other Exceptions: If any other type of exception occurs during the execution of a command, it will be caught by the event dispatching thread of the Swing framework. An error dialog will be displayed with details about the exception.

[BON2]

1) Requirement Implemented
2) Implementation Details
   The undo and redo methods are designed to reverse or reapply changes to the task management system, respectively. Here's how they work:
   a) undo(): If the undo stack is empty, the method prints "Nothing to undo" and returns immediately. If it's not empty, the method pops the most recent command from the undo stack and splits it into operations and arguments. Depending on the operation, the appropriate method is then called with the original arguments to reverse the change.
   b) redo(): If the redo stack is empty, the method prints "Nothing to redo" and returns immediately. If it's not empty, the method pops the most recent command from the redo stack and splits it into operations and arguments. Depending on the operation, the appropriate method is then called with the original arguments to reapply the change.
3) Error conditions and how they are handled
   There are several error conditions in the method, all of which result in an error message being printed to the console and the method returning without performing the undo or redo operation:
   - Empty Undo/Redo Stack: In the undo method, if the undo stack is empty, the function handles this by printing a message "Nothing to undo" and returning immediately. Similarly in the redo method, if the redo stack is empty, the function handles this by printing a message "Nothing to redo" and returning immediately.
   - Unrecognized Operation: If the operation to be undone or redone is not recognized, the function does nothing and returns, avoiding any potential errors or inconsistencies.