Project Report

Group 18
COMP2021 Object-Oriented Programming (Fall 2023)
Author: Tanya Budhrani
Other group members: Pranav Kedia, Jyotsna Venkatesan

Contents

1 Introduction

With the rate at which technology has advanced, society has borne witness to an exponential surge in the use of task scheduling tools. These tools have become indispensable aids in assisting individuals in achieving their daily efficiency goals. Ranging from the simplicity of a to-do list; allowing users to tick-off their day-to-day chores, to the dynamicity of a calendar; granting you the freedom to plan years in advance – the digital realm is teaming with these assistants. Yet, despite their individualized functions, they each share a common goal: to make their user's life a whole lot easier.

Enter the Task Management System: a comprehensive Java-based application designed to facilitate the efficient management of tasks within a project or organizational context. The core components of the system include the representation of tasks as either primitive or composite entities, allowing for the modeling of intricate task hierarchies. When a primitive task is created, users are able to set the name, description, duration, and prerequisites of the task (likewise for a composite, but with a subtask list as opposed to a set of prerequisites). The implementation encompasses the creation, modification, and deletion of tasks; enabling users to organize their to-dos in a way that reflects real-world relationships.

That being said, this project report serves as not only a documentation of the user and development of the Task Management System but also as a lens through which readers can gain insights about the transformative role that technology plays in shaping our lifestyles. As our dependence on digital assistants deepen, those responsible for crafting these tools bear the responsibility of ensuring seamless functionality. Creation of the TMS exemplifies the ongoing dialogue between user expectation and technological innovation; a testament to the belief that, in this day and age, technology is not just a convenience but a necessity.

2 My Contribution and Role Distinction

In our three-member group, we tried to divide our tasks as evenly as possible. As a result, we approached the programming aspect by dividing the responsibilities based on the specified requirements– I was tasked with requirements 11 to 16, which included the **definition of composite tasks, printing criterias, searching**, and **quitting**. Additionally, I took on the responsibility of implementing the file reading methods, namely **store** and **load**. To do this, I utilized the instructions posted on Blackboard, along with some additional help from W3Schools (https://www.w3schools.com/java/java_files_read.asp) and GeeksForGeeks (https://www.geeksforgeeks.org/different-ways-reading-text-file-java/)

Aside from the programming, our group also divided the written and graphical tasks – I partnered up with Jyotsna to create the **user manual** (specifically the step-by-step instructions,

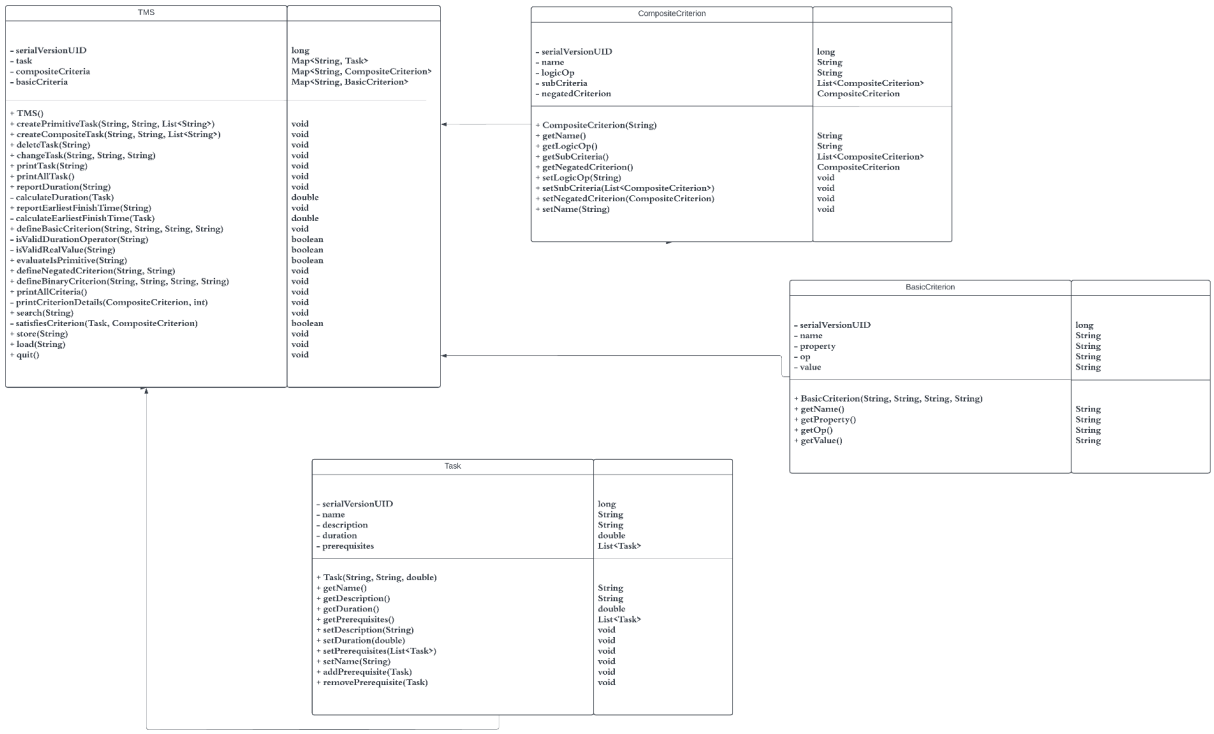
troubleshooting, and additional resources), while Pranav focused on the development of the powerpoint presentation. To create the user manual, it was imperative that we each held a comprehensive understanding of one another's code implementation. To achieve this, the three of us gathered to refine the final program with the proper code formatting. During this collaborative session, we took the opportunity to explain our respective code segments, to ensure that all of us had a mutual understanding of the entire project's functionality.

In addition to this, I was in charge of **writing the JUnit code** to test out our program's functionality. I tried to add as many test cases as I could think of to try out the error handling and final printout of our program.

Finally, regarding the video presentation, I was in charge of **editing our video** and ensuring that it was at its highest quality possible.

A Command-Line Task Management System

Class Diagram



Design

Data structures

The following data structures were utilized in the creation of the final project:

HashMap

- HashMaps are used to store key-value pairs. In the context of the TMS, they were employed to associate the names of a task with the instances of the tasks, composite criteria, and basic criteria.

ArrayList

- ArrayLists are dynamic arrays that provide flexibility in managing a list of objects. In the TMS, ArrayLists were used to represent collections of tasks, subtasks, prerequisites, and subcriteria. For instance, the ‘prerequisites’ ArrayList in the ‘Task’ class holds a list of the tasks that must be completed before the associated task can begin.

Serializable

- This project implements the Serializable interface which allows objects to be stored in files and transmitted over the network – Serialization is crucial for storing the state of the TMS in the files (e.g. during the ‘store’ method) and then later deserializing it back into objects (e.g. during the ‘load’ method)

ObjectOutputStream and ObjectInputStream

- These classes are used for writing and reading objects to and from streams, which enable the serialization and deserialization of an object.

Object-orientation capabilities

The following capabilities can be seen in the TMS project:

Encapsulation

- The technique of restricting access to some components of an object which prevents unauthorized access and modification. This can be seen in the initialization of most of the fields of each class, utilizing the ‘private’

keyword for access control, then the use of getters and setters to allow for the retrieval and update of a variable outside the encapsulating class.

Inheritance

- Allows a class to inherit properties and behaviors from another class which provides code reusability– we utilized inheritance with the extension of the Serializable interface.

Polymorphism

- Enables objects to be treated as instances of their parent class, which provides a single interface for different types– we utilized polymorphism in our method overloading and overriding. For instance createPrimitiveTask and createCompositeTasks are two overloaded methods, while defineBasicCriterion is a method that overrides the BasicCriterion class.

Model-View-Controller (MVC) framework

Model

- Here, the model is represented by the TMS code with additional private methods to aid in the calculation and satisfaction of data (e.g. calculateDuration).

View

- The view represents the Graphical User Interface or GUI of the model– while we don't have an explicit GUI, the task manager utilizes additional code (e.g Task, CompositeCriterion, and BasicCriterion) that the user cannot see as they only witness the model itself (TMS).
- In addition to this, we made use of DELIMITERS to ensure the readability of the printed tasks and criteria:

```
All tasks:
------------------------
------------------------
------------------------
------------------------
------------------------
All criteria:
Criterion: Criterion3
  Criterion: Criterion3
    LogicOp: &&
------------------------
------------------------
Search results for Criterion3:
Tasks satisfying criterion Criterion3:
Task: Task2
Task: Task1
Task: composite1
Task: composite2
------------------------
Report duration for Task1:
------------------------
Report earliest finish time for Task1:
Earliest finish time of task Task1: 0.3 hours
------------------------
Saving the state to task_manager_state.ser
Loading the state from task_manager_state.ser
All tasks after loading:
------------------------
------------------------
------------------------
------------------------
------------------------
Quitting the application:
Exiting the Task Management System. Goodbye!
```

Controller

- Controllers are responsible for handling user input and acts as an intermediary between the model and the view. In this case, this would be the main class in the TMS model that allows users to access the functions provided by both the public and private methods/fields.

Line Coverage

- Our JUnit test has an overall line coverage of 100% while our TMS program has an overall line coverage of 82%. The JUnit covered cases on all of the main functions (e.g. creating tasks, defining tasks, changing tasks, deleting tasks, searching for tasks, printing criterias, checking if a task is primitive or composite, reporting earliest finish time, calculating duration, etc) but we also added additional tests to check if a task doesn't exist (especially in trying to print that task out).

Requirements

Requirements	Was it implemented?	How was it implemented?	How we handled error conditions
Creating primitive tasks	Yes	This method takes parameters such as the task's name, description, duration, and a list of prerequisites. First, it creates a new task object with the provided name, description, and a default duration of 0. If all checks pass, the task is added to a tasks collection, and if a positive duration is provided, it is set for the new task. Finally, the method iterates through the list of prerequisite names, associates them with existing tasks, and adds them as prerequisites to the newly created task. This implementation ensures the creation of valid and well-defined simple tasks in the system while handling potential input errors.	Checks if the task name adheres to certain rules (e.g., alphanumeric, length constraints, and not starting with a digit). Ensures that the duration is non-negative. Verifies that the task description follows certain criteria (e.g., alphanumeric characters and hyphens allowed). Provides informative error messages if any of the checks fail.
Creating composite tasks	Yes	This method takes parameters such as the composite task's name, description, and a list of subtask names. It starts by creating a new task object with the provided name and description, setting the initial duration for composite tasks to 0. The method then iterates through the list of subtask names, retrieving each subtask from the tasks collection and adding it as a prerequisite to the composite task. Additionally, it calculates the duration of the composite task based on the maximum duration among its subtasks.	Ensures that the duration for composite tasks is initially set to 0. Calculates the composite duration based on subtasks and sets it for the composite task. Handles cases where subtask names are invalid or not found, providing feedback.

		The `calculateDuration` function is used for this purpose. Finally, the method sets the calculated duration for the composite task and adds it to the tasks collection.	
Deleting tasks	Yes	This method takes a task name as a parameter, retrieves the corresponding task from the tasks collection, and checks if it exists. If the task exists, it is removed from the tasks collection. Subsequently, the method iterates through the remaining tasks and removes the deleted task from their list of prerequisites. This ensures that any references to the deleted task are appropriately handled, maintaining the integrity of task dependencies. Additionally, when a composite task is deleted, the method ensures that all its subtasks are also removed.	Checks if the task to be deleted exists. If not, no action is taken. Removes the task and updates prerequisites for other tasks.
Changing the properties of an existing task	Yes	This method takes the task name, the property to change, and the new value as parameters. It first retrieves the task with the given name from the tasks collection and checks if it exists. If the task exists, the method uses a switch statement to determine which property needs to be modified and then applies the corresponding changes. For primitive tasks, the properties that can be changed include "name," "description," "duration," and "prerequisites." If the property is "name," the method sets the new name for the task. If it is "description," the description is updated. If it is "duration," the method sets the new	Validates input by checking if the task and specified property exist. Parses and validates duration values. Ensures that prerequisites or subtasks mentioned exist before setting them. Provides feedback if the task or property is not found.

		duration by converting the new value to a double. If it is "prerequisites," the method parses the comma-separated list of prerequisite names, retrieves the corresponding tasks, and sets them as prerequisites for the task. For composite tasks, the allowed properties to change are "name," "description," and "subtasks." The method handles these properties similarly to primitive tasks, updating the name and description accordingly. When the property is "subtasks," it parses the comma-separated list of subtask names, retrieves the corresponding tasks, and sets them as subtasks for the composite task.	
Printing the information of a task	Yes	This method takes the task name as a parameter, retrieves the corresponding task from the tasks collection, and checks if it exists. If the task exists, the method creates a StringBuilder to construct a clear and readable output. The resulting information includes the task's name, description, duration, and, based on whether the task is primitive or composite, either prerequisites or subtasks. For primitive tasks, the output includes a list of prerequisites, each separated by a comma. For composite tasks, the output includes a list of subtasks, also separated by a comma. The information is presented in a format that is easy to read, with each piece of information on a new line for clarity. If the task does not exist, the method returns a message indicating that the task with the given name was not found.	Checks if the task exists before attempting to print its information. Returns an error message if the task is not found.

Printing the information of all tasks	Yes	This method iterates through the collection of tasks and, for each task, calls the `printTask` method to obtain and print its information. After printing the details of each task, a separator line is added to enhance readability and distinguish between the information of different tasks. Each task's information, including name, description, duration, and either prerequisites or subtasks, is displayed in an easily readable format. The separator line serves as a visual delimiter between the information of different tasks.	By calling printTask, the method checks if the task exists before attempting to print its information.
Reporting the duration of a task	Yes	This method takes the task name as a parameter, retrieves the corresponding task from the tasks collection, and checks if it exists. If the task exists, the method calculates and returns the duration of the task using the `calculateDuration` helper method. For simple tasks, the duration is simply the duration of the task itself. For composite tasks, the duration is calculated as the maximum duration among its subtasks. The `calculateDuration` method recursively computes the duration of each subtask and identifies the maximum duration. The returned duration is then formatted into a string, indicating the task name and its calculated duration in hours. If the task does not exist, the method returns a message stating that the task with the given name was not found.	Checks if the task exists before calculating and reporting its duration. Returns an error message if the task is not found.
Reporting the earliest finish time of a task	Yes	This method takes the task name as a parameter, retrieves the corresponding task from the tasks collection, and checks if it exists. If the task exists, the	Checks if the task exists before calculating and reporting its

		method calculates and prints the earliest finish time of the task using the `calculateEarliestFinishTime` helper method. The earliest finish time of a task is determined by the sum of two components: 1) the earliest finish time of all its prerequisites, and 2) the duration of the task itself. The `calculateEarliestFinishTime` method recursively computes the earliest finish time of each prerequisite, ensuring that the earliest finish time is updated based on the maximum among its prerequisites. The calculated earliest finish time is then printed to the console, providing users with information about when a particular task can be completed in the most time-efficient manner. If the task does not exist, no output is generated, preventing potential errors.	earliest finish time. Returns an error message if the task is not found.
Defining basic task selection criteria	Yes	This method takes parameters such as the criterion name, property, comparison operator (op), and value. The method first initializes a `BasicCriterion` object as null and then uses a switch statement to validate the input based on the specified rules. For properties "name" and "description," the method checks if the operator is "contains," and if the value is a string enclosed in double quotes. For the "duration" property, the method ensures that the operator is a valid duration operator (>, <, >=, <=, ==, or !=) and that the value is a valid real value. For properties "prerequisites" and "subtasks," the method checks if the operator is "contains" and that the value	Validates input based on specified rules for different properties. Checks if the criterion is null after validation and only adds it if it is valid.

		is a non-empty list of comma-separated task names. If the input passes the validation checks, a new `BasicCriterion` object is created, and it is added to the collection of basic criteria (`basicCriteria`).	
Checks if a task is primitive	Yes	This method takes the task name as a parameter, retrieves the corresponding task from the tasks collection, and evaluates whether the task is primitive based on the presence of prerequisites. The method returns `true` if the task is primitive (i.e., it has no prerequisites), and `false` otherwise.	Returns false if the task doesn't exist or has prerequisites, effectively handling non-existent tasks and composite tasks.
Defining composite task selection criteria	Yes	The `defineNegatedCriterion` method constructs a composite criterion named `name1` by negating an existing criterion named `name2`. It first retrieves the existing criterion `criterion2` from the `compositeCriteria` collection. If `criterion2` exists, a new `CompositeCriterion` named `negatedCriterion` is created, set as the negation of `criterion2`, and added to the `compositeCriteria` collection under the name `name1`. If `criterion2` is not found, a placeholder negated criterion is created with a placeholder name, and an informational message is printed to the console indicating that the base criterion (`name2`) was not found. The `defineBinaryCriterion` method constructs a composite criterion named `name1` by combining two existing criteria named `name2` and `name3` using a logical operator (`logicOp`), which can be either && (logical AND) or || (logical OR).	Checks if the base criteria exist before creating the negated or binary criterion. Provides feedback if the base criterion is not found.

		It retrieves the existing criteria `criterion2` and `criterion3` from the `compositeCriteria` collection, creates a new `CompositeCriterion` named `binaryCriterion` with the specified logical operator and sub-criteria, and adds it to the `compositeCriteria` collection under the name `name1`.	
Printing all defined criteria	Yes	This method iterates through the collection of composite criteria (`compositeCriteria`) and, for each criterion, calls the `printCriterionDetails` method to print out the criterion details. A separator line is added after each criterion for better readability. The `printCriterionDetails` method recursively prints the details of a composite criterion, including its name, logical operator (`LogicOp`), sub-criteria, and any negated criterion. The indentation parameter is used to format the output with appropriate spaces, enhancing the readability of nested criteria.	Calls printCriterionDetails which first ensures that the criterion is not null before printing.
Searching for tasks based on existing criterion	Yes	This method takes a criterion name as a parameter, retrieves the corresponding composite criterion from the `compositeCriteria` collection, and checks if it exists. If the criterion exists, the method iterates through all tasks and prints the names of those that satisfy the specified criterion. The `satisfiesCriterion` helper method recursively evaluates whether a given task satisfies a composite criterion. It considers both logical AND (`&&`) and logical OR (`||`) operators, as well as negated criteria.	Checks if the specified criterion exists before attempting to search for tasks. Provides feedback if the criterion is not found.

		For each task, the method checks if it satisfies the specified criterion and prints its name if it does. The output provides a list of tasks that meet the specified criterion, contributing to the tool's functionality in helping users identify and manage tasks based on defined criteria.	
Storing defined tasks and criteria into a file	Yes	This method takes a file path as a parameter and uses an `ObjectOutputStream` to serialize and write the `tasks`, `compositeCriteria`, and `basicCriteria` collections to the specified file.	Utilizes Java's exception handling to catch IO exceptions that might occur during file writing. Prints the stack trace if an IOException occurs during file writing.
Loading tasks and criteria from the file at path	Yes	This method takes a file path as a parameter and uses an `ObjectInputStream` to deserialize and read the `tasks`, `compositeCriteria`, and `basicCriteria` collections from the specified file. If successful, the method updates the internal collections (`tasks`, `compositeCriteria`, and `basicCriteria`) with the loaded data from the file. If an exception occurs during the loading process, whether due to I/O issues or class not found, the stack trace is printed to the console for debugging purposes.	Utilizes Java's exception handling to catch IO and ClassNotFoundException exceptions that might occur during file reading. Prints the stack trace if an IOException or ClassNotFoundException occurs during file reading
Terminate current execution of the system	Yes	This method prints a goodbye message to the console, indicating the exit of the Task Management System, and then calls `System.exit(0)` to terminate the program with a status code of 0.	We use 0 to indicate a successful termination, as opposed to any other digit which represents a failed termination.

Graphical User Interface	No		
Undo and redo	No		

,