Project Report

Group 11
COMP2021 Object-Oriented Programming (Fall 2023)
Author: Ileana Pal 21094312D
Other group members:
Bailin ZENG 21107431D
Xin DAI 23096373D
Caryse CHONG Jia Wern 22104071D

1. Introduction

The purpose of this individual report is to reflect on my contribution and experiences during the development of the command-line task management system (TMS) as part of the COMP2021 group project. This report highlights my involvement in various aspects of the project and collaboration within the group.

2. Role and Responsibilities

My main responsibilities as a group member were code documentation, user manual preparation, and testing.

1. Code documentation

I was in charge of improving the code's readability by thoroughly documenting the source, adding comments, and clearly outlining intricate logic. This was quite a complex task, as the code went through several iterations and drafts, and since several people were working on it, it was very important to make sure that everyone understands the logic and structure of how the code is working.

Additionally, since a lot of the time I was commenting on code that I had not written, it was important for me to understand it completely, and communicate with my teammates who had written the code, to make sure that the documentation was accurate.

2. User Manual Preparation

I was also in charge of the production of the user manual, ensuring that it included precise instructions on how to operate the system. I wanted to ensure that I had a very clear and logical structure in the user manual, especially for a beginner. I sectioned the manual into clear parts that would gradually guide the user to be able to use the TMS without any difficulty. I also made it a point to include FAQs about questions that I myself had about the task, like what the difference was between sub-tasks and prerequisites, and why we need Criterion. I think that answering the questions that we had would be a good way of targeting any future questions that the user would have.

3. Testing

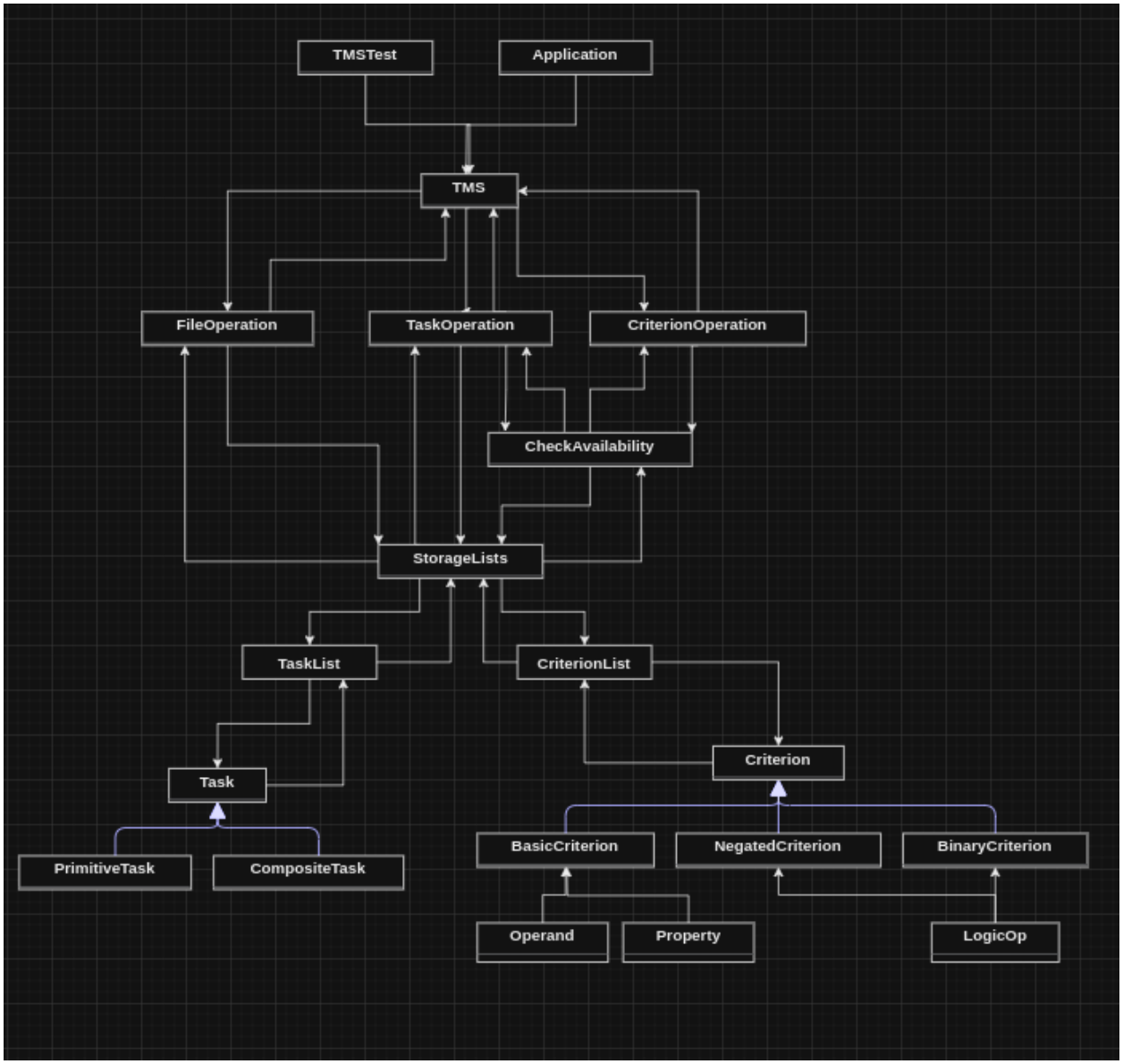
It was my responsibility to design an extensive testing plan for the command-line interface task management system. I created and carried out a number of test cases to verify the robustness and functionality of the system. The main class to test the system is TMSTest which runs the tests for all the Task, Criterion and File commands using the testRun method defined in TMS. This test simply loops through all the commands that could be run through the particular command and fails if there is an exception thrown during its running.

3. A Command-Line Task Management System

This section covers the general architecture and specifics of the system's implementation.

3.1. Design

The following diagram shows the design of the TMS implementation.



1.TMS.java:

TMS is an insatiable class, which acts as the entry and exit of the system through the run() method. By allowing it to extend the Thread class, this module can be further expanded to facilitate concurrent programming.

To test the input of numerous command lines as String arrays, there is also a test run method.

The access to the Controller is the operation method within this module. After creating a general StorageLists model, it will forward it to the controller. We ought to make improvements to this in the future.

2. TaskOperation.java:

This task makes up a part of the controller, and is hence hidden from the user. It uses static methods and cannot be instantiated. It takes input from View and outputs raw input data into CheckAvailability. After getting a converted type from the CheckAvailability, it creates or modifies the model. It will return the result of the operation back to View.

3. CriterionOperation.java:
CriterionOperation.java serves the same control function as TaskOperation, except it deals with the operation of criteria.

4. FileOperation.java:
This Java file contains only Store and Load static methods that allow the user to store the StorageLists model in the form of files.

5. CheckAvailability.java:
This is a checker for determining the availability and legality of the majority of raw inputs. It is made up of static techniques. It aids in the input's conversion into the model as well. To convert a task name into a task object, for instance, you can call the model StorageLists' searching method.
This class also throws out most exceptions when the input is illegal. The exceptions will be printed out as output in the View.

6. StorageLists.java:
This is the class that packs up all the data. It is regarded as the Model's core component. It is composed of two ArrayLists: one that holds the criteria and the other that holds the task. These two lists are contained and are only accessible by use of particular techniques. Additionally, it overrides the toString method in order to implement print and file storage features.

7. Task.java, PrimitiveTask.java and CompositeTask.java:
Task is an abstract class inherited and implemented by PrimitiveTask and CompositeTask. They have implemented the essential operation for a Task, such as storing the name, description, etc. The prerequisite Tasks and subtasks are stored as ArrayList in each task class.
The CheckDuration, toString, etc, are implemented differently in PrimitiveTask and CompositeTask. We considered this a perfect utilization of Polymorphism to enhance the modulation and reduce the difficulty of redevelopment of our project. We also simplified the code structure a lot by utilizing this Polymorphism feature. These three classes are part of the Model.

8. Criterion.java, BasicCriterion.java, CompositeCriterion.java, BinaryCriterion.java and NegatedCriterion.java:
The general Criterion class only contains the basic name of a criterion. It also contains a Check method which will be performed differently in each Criterion subclasses.
These are the classes for storing the information of Criteria. The information on property and operand has already been converted to the Enum class in the CheckAvailability.java. The corresponding object is stored in the essential criteria. The BinaryCriterion contains two Criteria, and the NegatedCriterion includes one.

9. Operand.java, Property.java and LogicOp.java:
These are the helper Enum for establishing and maintaining the Model. It helps to convert String input into objects. Meanwhile, the evaluating functions and checking functions for the criteria are implemented in these Enum classes. Again, Polymorphism is applied for different operands and operators, and the system should return a different result.

3.2. Requirements

The requirements REQ1 to REQ16 have all been implemented. The requirements are as shown below:

[REQ1] [REQ2]:

1) Implemented

2) After the inputs are received from View, which is the TMS, the TaskOperation executes the logic control. The CheckAvailability will check the input. A new task will be created if no exceptions are generated.
3) Exceptions:
    a) “The task name has an illegal format.”
    b) “The task description has an illegal format.”
    c) “We use a name to define the uniqueness of a task, so defining two tasks with the same name is not allowed.”
    d) “Duration must be a number.”
    e) “Duration cannot be negative.”
    f) “The task does not exist.” : We do not allow the user to create a task with a non-recognized prerequisite or subtask input. A prerequisite or subtask must already exist in the system if the user wants to define a new task based on it.

[REQ3]

1) Implemented
2) The data flow is the same as previous requirements.
3) Exceptions:
    a) Task not found: The required task to be deleted mush exist in the system to be deleted.
    b) The task is the prerequisite for another task. Users cannot delete a task that is the prerequisite of another primitive task.
    c) The subtask is the prerequisite of another task. When deleting a composite task, it is required to delete the subtasks that belong to it. If one of its subtasks is the prerequisite of another primitive task, the user cannot delete it.

[REQ4]

1) Implemented
2) The CheckAvailability.java will help the TaskOperation.java to call the search list function in StorageLists.java. If the task is found, the setProperty method will be executed.
3) Exceptions:

a) Cannot set duration for a composite task. Duration is the feature of a simple task.
b) Cannot set prerequisites for a composite task and vice versa.
c) A task cannot be a prerequisite or subtask of itself.
d) It is illegal to create a task referencing loop. For example, task t1 is the prerequisite of t2, while t2 is the prerequisite of t3. If we want to set a loop by defining t3 as the prerequisite of t1, it's illegal since we cannot decide the duration of these three tasks.

[REQ5][REQ6]

1) Implemented
2) The print single task and all tasks are executed in the TaskOperation.java by calling the toString method in each Task class and the printAllTasks in the StorageList class.
3) Exception: There are no exceptions. The system will output no-task notation if there are no tasks created in the system.

[REQ7][REQ8]

1) Implemented
2) The report duration of a composite task and the report of the earliest finish time are the same. Polymorphism is utilized so that the getDuration method in the Task class performs differently in these two types of Tasks. The getDuration returns the maximum hours of getDuration result for each task object from its prerequisites or subtasks. This is performed recursively until a task object has no subtasks prerequisites.
3) There are no exceptions for getting the duration. The only exception that may be thrown is that the task name cannot be found in the system.

[REQ9]

1) Implemented
2) defineBasicCriterion will first check whether the user input is allowed to be used to create a new basic criterion using the checkAvailability class, and then if none of them throws an exception, a new criterion is added to the storage list.
3) The following exceptions may be thrown by this implementation:

a) "The task name is illegal."
b) "This criterion already exists.”
c) "Invalid property input.”
d) "Invalid operand input."
e) "Property and Operand do not match."
f) "Too much input for value."

[REQ10]

1) Created
2) This is created at the constructor of StorageLists.
3) Does not throw any exceptions

[REQ11]

1) Implemented
2) defineNegatedCriterion is implemented in CriterionOperation and calls storageLists.defineNegatedCriterion if certain checks have passed, which simply creates a new NegatedCriterion object and adds it to the storage list. defineBinaryCriterion is implemented in CriterionOperation and calls storageLists.defineBinaryCriterion if certain checks have passed, which simply creates a new BinaryCriterion object and adds it to the storage list.
3) The following exceptions may be thrown by this implementation:
    a) "The task name is illegal."
    b) "This criterion already exists."
    c) "The required criterion does not exist."
    d) "Invalid logical operand."

[REQ12]

1) Implemented
2) Simply prints all criteria in the storage list. Prints "There is no criterion currently..." if there are no criteria in the list.
3) Does not throw any exceptions.

[REQ 13]

1) Implemented

2) The method begins by checking if the specified criterion exists as seen in 'CheckAvailability.checkCriterionExists'. If the criterion does not exist, it throws an exception. If the criterion exists, it proceeds to search for tasks that meet the criterion using 'storageLists.search(criterion)'. The output is a message that includes the task names that meet the criterion.
3) The following exceptions may be thrown by this implementation:
    a) If the criterion does not exist, the method throws an exception.
    b) If no tasks meet the criterion, it returns a message indicating that no corresponding tasks were found. The program also uses 'StringBuilder' to handle cases where the last character is a comma by removing it before returning the final message.

[REQ 14]

1) Implemented
2) The method takes a 'StorageLists' object and a file path as parameters. It uses a 'try-with-resources' statement to automatically close the file streams after the write operation is complete. After, it creates a 'FileOutputStream' and an 'ObjectOutputStream' to write the 'StorageLists' object to the specified file path. If the write operation is successful, it returns a success message.
3) The following exceptions may be thrown by this implementation:
    a) If the specified file path is not found, the method throws an exception with the message "The file path is illegal".
    b) If there is an IO exception during the file write operation, it throws an exception with the message "IO exception".
    c) The method also declares that it throws a general "Exception", and it's caught in the calling code.

[REQ 15]

1) Implemented
2) When Load<path> is entered, the file path is extracted from the command. The 'FileOperation.readFile' method is called to load data from the specified file path. If successful, the program updates its state with the loaded data and informs the user that the data has been loaded.
3) The following exceptions may be thrown by this implementation:

a) If the specified file path does not exist, the method throws an exception with the message “The file cannot be found”.
b) If there is an IO exception during the file write operation, it throws an exception with the message “IO exception”.
c) If the class definition is not found while deserializing the object, ‘ClassNotFoundException’ is caught and an exception message is thrown: “The file content is unreadable”.
d) The method also declares that it throws a general “Exception”, and it’s caught in the calling code.

[REQ 16]

1) Implemented
2) When the ‘quit’ method is called, it prints a farewell message to the console. It then calls ‘System.exit(0)’ to terminate the program.
3) No exceptions are thrown.