Project Report

COMP2021 Object-Oriented Programming (Fall 2023)

Group 24 Author:
Zhu Jin Shun

Other Group Members:
Wang Ruijie
Zeng Tianyi
Liu Yuyang

1 Introduction

This document aims at introducing and explaining group 24's design and implementation of a command-line-based task management system. The project is part of the subject COMP2021 Object-Oriented Programming at PolyU.

2 My Contribution and Role Distinction

In this project, my primary role is that of a coder, where I am responsible for writing and implementing the necessary requirement codes and the whole responder for the user manual design and writing. I play a crucial role in translating the project's requirements into functional code.

2.1 Work Distributed

My work distribution is mainly as one of the person who is responsible for coding in this project, my contribution is mainly about:

1. Focusing on coding requirements and work distributed;
2. Providing different class or instances for other group members to write on specific requirements ;
3. Regularly providing help for other group members to complete assigned tasks and providing technical support;
4. Arranging the requirement code for presentation and user manual, and ensuring the successful execute.
5. Designing format and main writer for the user manual

2.2 Assistance for system designer

As an assistance on designing the object-oriented system, I am concentrated on:

1. Providing help on designing a command-line user interface to achieve friendly interaction with accurate input and output control;
2. Creating the classes of *Negated Criterion and Binary Criterion* to standardize the properties and the behaviors of *criterion*;
3. Conceptualize and put forward the structure of the classes of *Criterion*.

2.3 Programmer for specific requirements

I also implemented the following requirements and revised their functions independently or with other members:

1. *DeleteTask*;
2. *ChangeTask*;
3. *ReportEarliestFinishTime*;
4. *DefineBasicCriterion*;
5. *DefineNegatedCriertion*;
6. *DefineBinaryCriterion;*
7. *Search.*

2.4 Use of GenAI tools

Generally speaking, all the *Javadoc-style* comments, and some of the testing session for my requirement code has GenAI's participation for better improvement. I also used GenAI to polish the individual report and user manual. The GenAI tool used by me in this project is *Assistant* supported by *Poe*. In detail, I use the GenAI mainly for:

- automatic generation of *Javadoc*;
- inspection and examination of existing code;
- debugging;
- polishing and expressing better use of language in the individual report and user manual.

The verification and examination of the generated contents is as follows:

- Check whether the generated *Javadoc* contents are at the correct position, with accuracy, and completed (e.g., no missing @*param* or @*return*);
- Execute the generated requirement code or test code against the target system and observe the behavior with integration of existing code, and improve efficiency with 100% correctness;
- Share the generated contents with other group members for a peer review;
- Manually review for any instances of unclear or mismatched expressions within the generated content and the contexts of the communication with GenAI.

To effectively leverage GenAI, my approach focuses on assigning it technical tasks or tasks that involve high repeatability. Prior to assigning specific tasks, it is crucial to provide GenAI with comprehensive context and background information regarding the task at hand. Additionally, it is important to establish clear expectations for GenAI's output, including desired formats or specifications. It should be noted that when dealing with projects that encompass a wide range of content and requirements, GenAI should not be solely relied upon for generating numerous source code files or connecting classes and interfaces together. Ultimately, GenAI is not suited for handling complex logic-oriented or creative tasks within the project, as it may struggle with effectively dividing and managing intricate workloads.

3 A Command-Line Task Management System

3.1 Design

3.1.1 The structure of the Command-Line Task Management System

The structure of the Command-Line Task Management System (TMS) is displayed in the UML diagram below:

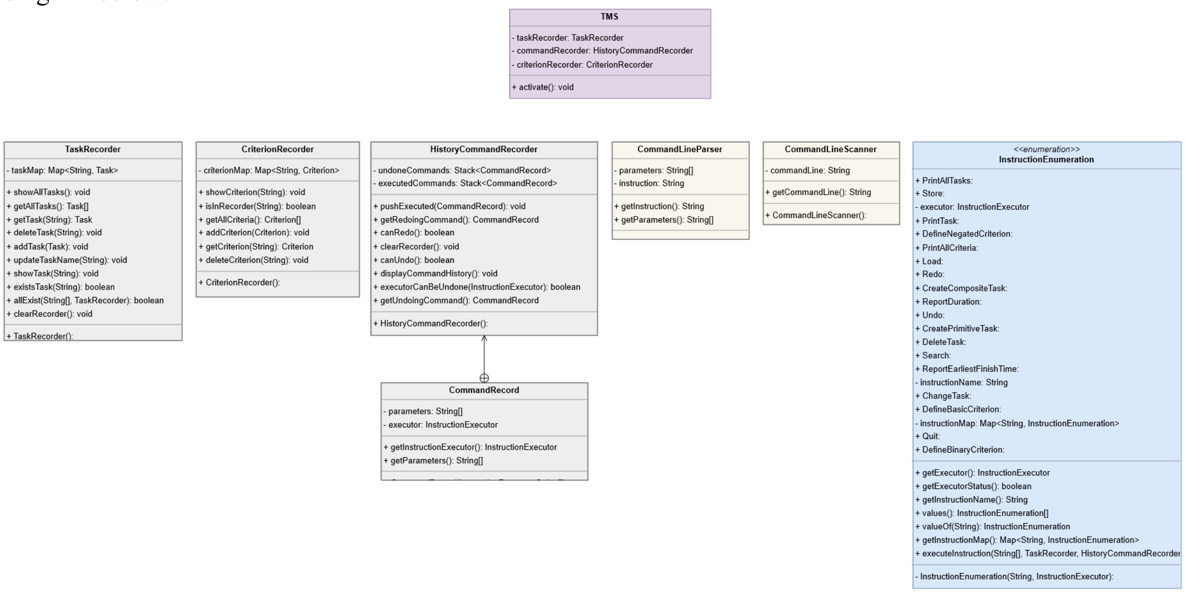

Figure 1: Basic structure of TMS

TMS can be initialized by constructing itself and then calling *TMS.activate()*. Once TMS is constructed, three recorders, *taskRecorder*, *commandLineRecorder* and *criterionRecorder*, enjoying the equal lifetime of TMS itself will be generated, which are instances of the classes *TaskRecorder*, *HistoryCommandLineRecorder* and *CriterionRecorder* separately.

```
public TMS() {

    this.taskRecorder = new TaskRecorder();
    this.commandRecorder = new HistoryCommandRecorder();
    this.criterionRecorder = new CriterionRecorder();

}
```

After calling *TMS.activate()*, a scanner of *CommandlineScanner* and a parser of *CommandlineParser* will be created for each time of user input for division and analysis of the input. We will divide a piece of user input in *String* into *String[]*, and *String[0]* is regarded as the instruction name in *String*, while the other elements in *String[]* are regarded as *String[] parameters*.

```
public class CommandLineScanner {

        private String commandLine;
        public CommandLineScanner() {
        Scanner commandLine = new Scanner(System.in);
```

```
            if (commandLine.hasNextLine()) {
            this.commandLine = commandLine.nextLine();
            }
            }

        public String getCommandLine() {
        return this.commandLine;
        }

}
```

```
public class CommandLineParser {

    private final String instruction;
    private final String[] parameters;

        public CommandLineParser(String commandLine) {
        String[] commands = commandLine.split(" ");
        // Split the command line by spaces
        this.instruction = commands[0];
        // The first element is the instruction
        this.parameters = new String[commands.length - 1];
        // The remaining elements are parameters
        System.arraycopy(commands, 1, this.parameters, 0, commands.length - 1);
        // Copy the parameters to the array
        }

}
```

```
public void activate() {

            for ( ; ; ) {
            CommandLineScanner scanner = new CommandLineScanner();
            CommandLineParser parser =
            new CommandLineParser(scanner.getCommandLine());
            String instructionName = parser.getInstruction();
            // not ended here
            }

}
```

Inside the TMS, we use an Enum class, *InstructionEnumeration*, to define all valid instruction with its name in *String* and the method of instantiating the corresponding *Executor*. After that, we establish the relationship between the name and the instruction itself using a *HashMap*. For example:

```
public enum InstructionEnumeration {

```

```java
    // Examples of available instructions
    CreatePrimitiveTask("CreateSimpleTask", new CreatePrimitiveTaskExecutor()),
    CreateCompositeTask("CreateCompositeTask", new CreateCompositeTaskExecutor()),
    DeleteTask("DeleteTask", new DeleteTaskExecutor());

    private final String instructionName;
    private final InstructionExecutor executor;

        InstructionEnumeration(String instructionName, InstructionExecutor executor) {
        this.instructionName = instructionName;
        this.executor = executor;
        }

            // Create a HashMap for indexing and searching available instruction names
            // and the instruction itself (and corresponding executor)
            private static final Map<String, InstructionEnumeration> instructionMap =
            new HashMap<>();
            static {
            for (InstructionEnumeration instruction : InstructionEnumeration.values()) {
            instructionMap.put(instruction.getInstructionName(), instruction);
            }
            }

}
```

Via the *instructionMap*, we can find out the specific instruction in *activate()*.

```java
public void activate() {

    // Continued here

        Map<String, InstructionEnumeration> instructionMap =
        InstructionEnumeration.getInstructionMap();
        // Retrieve instructionMap
        InstructionEnumeration instructionEnumeration =
        instructionMap.get(instructionName);
        // Use instructionName to search the corresponding Instruction

                if (instructionEnumeration != null) {
                System.out.println("Instruction found: " + instructionEnumeration);
                instructionEnumeration
                .executeInstruction(parser.getParameters(), taskRecorder,
                commandRecorder, criterionRecorder);

                        if (executorCanBeUndone(instructionEnumeration.getExecutor())) {
                        HistoryCommandRecorder.CommandRecord commandRecord =
                        new HistoryCommandRecorder
                        .CommandRecord(instructionEnumeration.getExecutor(),
                        parser.getParameters());
                        commandRecorder.pushExecuted(commandRecord);
                        }

        System.out.println(Arrays.toString(parser.getParameters()));
```

```
        boolean status = instructionEnumeration.getExecutorStatus();
        System.out.println(status);
        System.out.println();
        commandRecorder.displayCommandHistory();
        } else {
        System.out.println("Instruction not found");
    }

}
```

Within the *activate()* method, we construct a perpetual loop that continues until the user inputs ”Quit,” causing the system to exit via *System.exit(0)*, so that users are able to iteratively input their commands.

3.1.2 Task and Criterion design

The implementation of the classes of *Task* and *Criterion* is given below:

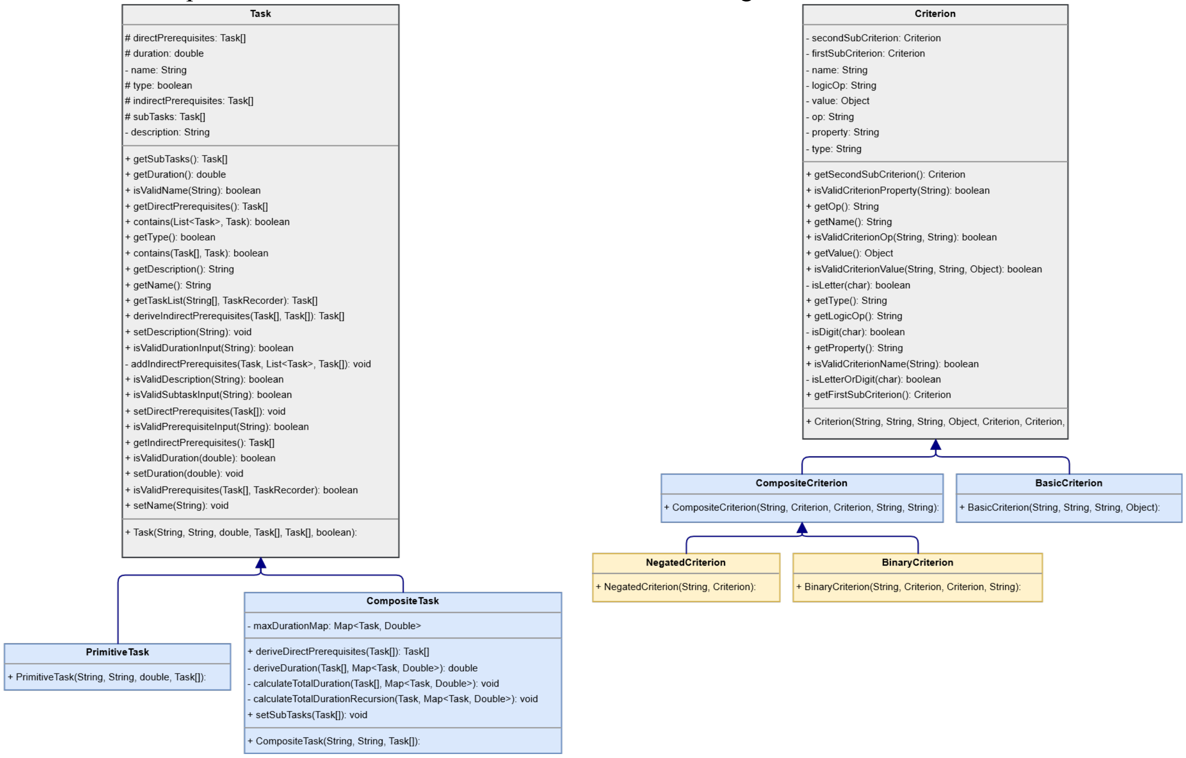

Figure 2: Implementation of Task and Criterion

For *Task* class, we define its fields containing *String name*, *String description*, *double duration*, *Task[] directPrerequisites* , *Task[] indirectPrerequisites*, *boolean type* and *Task[] subtasks*. The field of *indirectPrerequisites* will be automatically derived from the task's *directPrerequisites*, and we will explain the details and usage of the derivation mechanism in the following sections.

```
public class Task {

    private String name;
    private String description;
    protected double duration;
    protected Task[] directPrerequisites;
    protected Task[]
              indirectPrerequisites;
    protected Task[] subTasks;
    protected final boolean type;

            public Task(String name, String description, double duration,
            Task[] directPrerequisites, Task[] subTasks, boolean type) {
            this.name = name;
            this.description = description;
            this.duration = duration;
            this.directPrerequisites = directPrerequisites;
            this.indirectPrerequisites =
            deriveIndirectPrerequisites(directPrerequisites, subTasks);
            this.subTasks = subTasks;
            this.type = type;
```

```
            // true for a primitive task and false for a composite task
            }

    //not ended here
    }
```

The classes inheriting from *Task*, i.e., *PrimitiveTask* and *CompositeTask* have the same field as *Task*. However, in the corresponding constructors, the values of the field will be determined according to the practical meaning of them. For example, *subtasks* us meaningless for *PrimitiveTask*. Thus, a primitive task's subtasks will claimed as an empty task list.

```
public class PrimitiveTask extends Task{

                public PrimitiveTask(String name, String description,
                double duration, Task[] directPrerequisites) {
                super(name, description, duration,
                directPrerequisites, new Task[0], true);
                }

}
```

Likewise, the program will derive the duration and the prerequisites (both direct and indirect) of a composite task, though users are not required to provide them in the input command-lines.

```
public class CompositeTask extends Task {

            public CompositeTask(String name, String description, Task[] subTasks) {
            super(name, description, deriveDuration(subTasks, maxDurationMap),
            deriveDirectPrerequisites(subTasks), subTasks, false);
            }

            public void setSubTasks(Task[] subTasks) {
            this.subTasks = subTasks;
            this.directPrerequisites = deriveDirectPrerequisites(subTasks);
            this.indirectPrerequisites =
            deriveIndirectPrerequisites(this.directPrerequisites, subTasks);
            this.duration = deriveDuration(subTasks, maxDurationMap);
            }

    //not ended here
    }
```

The definition and implementation of *Criterion* is quite similar as *Task's*. Based on the context, we determine *NegatedCriterion* and *BinaryCriterion* as two kinds of different composite criteria.

```
public class Criterion {

```

```java
    private final String name;
    private final String property;
    private final String op;
    private final Object value;

    private final Criterion firstSubCriterion;
    private final Criterion secondSubCriterion;
    private final String logicOp;
    private final String type;

            public Criterion(String name, String property, String op, Object value,
            Criterion firstSubCriterion, Criterion secondSubCriterion,
            String logicOp, String type) {

        this.name = name;
        this.property = property;
        this.op = op;
        this.value = value;
        this.firstSubCriterion = firstSubCriterion;
        this.secondSubCriterion= secondSubCriterion;
        this.logicOp = logicOp;
        this.type = type;

    }
    // not ended here

}
```

```java
public class BasicCriterion extends Criterion{

        public BasicCriterion(String name, String property, String op, Object value) {
        super(name, property, op, value, null, null, null, "Basic");
        }

}
```

```java
public class CompositeCriterion extends Criterion {

                public CompositeCriterion(String name, Criterion firstSubCriterion,
                Criterion secondSubCriterion, String logicOp, String type) {
                super(name, null, null, null, firstSubCriterion,
                secondSubCriterion, logicOp, type);
                }

}
```

```java
public class BinaryCriterion extends CompositeCriterion {

        public BinaryCriterion(String name, Criterion firstSubCriterion,
        Criterion secondSubCriterion, String logicOp) {
        super(name, firstSubCriterion, secondSubCriterion, logicOp, "Binary");
        }

}
```

```java
public class NegatedCriterion extends CompositeCriterion {

        public NegatedCriterion(String name, Criterion firstSubCriterion) {
        super(name, firstSubCriterion, null,null, "Negated");
        }

}
```

Please not that in the implementation of *NegatedCriterion*, the criterion to be negated is not allowed to be *negated* or *binary*. Only a basic criterion is accepted by the executor. However, the sub-criteria of a binary criterion can be *basic*, *negated* and *binary*

Once an instance of *Task* or *Criterion* is created, it will be put into the *LinkedHashMap* which is inside the corresponding recorder. The use of textitLinkedHashMap is to maintain the order of being constructed.

3.1.3 Interface of the instruction executors

We package all source code of the instruction executors into a separate package for convenient management. All the classes of the instruction executor, e.g., *CreatePrimitiveTaskExecutor, PrintAllTasksExecutor, SearchExecutor, UndoExecutor, etc.*, are the implementation of an interface. And all these classes all implement three methods, *executeInstruction()* (the logic of execute the instruction while being called), *undoInstruction()* (the undo logic of each instruction that can be undone) and *getStatus()* (to detect if an instruction is successfully executed and can be undone logically). The diagram below showcases an example of the implementation relationship.

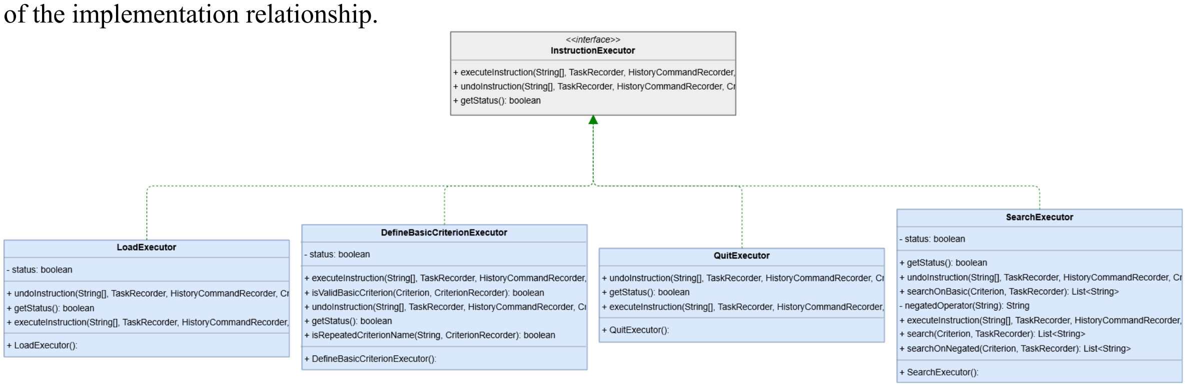

Figure 3: The instruction executor interface and its implementation

We take the recorders into the methods for any operation related to search, change or deletion of existing instances. You can get a closer look of the interface:

```
public interface InstructionExecutor {

            void executeInstruction(String[] parameters, TaskRecorder taskRecorder,
            HistoryCommandRecorder commandRecorder,
            CriterionRecorder criterionRecorder);
            void undoInstruction(String[] parameters, TaskRecorder taskRecorder,
            HistoryCommandRecorder commandRecorder,
            CriterionRecorder criterionRecorder);
            boolean getStatus();

}
```

3.2 Requirements

[REQ1]

1. The requirement has been implemented by Wang Ruijie and Liu Yuyang.

2. After the user input the command-line of creating a primitive task, e.g., *"CreatePrimitiveTask coding java-programming 3 , "*, the *CreatePrimitiveTaskExecutor* will retrieve the *parameters* and begin the validity examination of the string array by checking the length of *parameters*, the validity of input *name, description, duration*, and *prerequisites*. The program will also examine the existence of the creating task and return if the task is already existed. The program will ensure that all prerequisite tasks is existed. All methods for examination are defined in *Task* class. We will display the implementation of them in this section as an example. Due to space limitations, we cannot provide a detailed presentation of all the detection details. In the subsequent explanations of the requirements, we will not repeat similar coding contents.

```
        public void executeInstruction(String[] parameters,
        TaskRecorder taskRecorder, HistoryCommandRecorder commandRecorder,
        CriterionRecorder criterionRecorder) {

    if (parameters.length != 4) {
        System.out.println("The parameters of the command is invalid.");
        return;
        }

    String name = parameters[0];
    if (!Task.isValidName(name)) {
        System.out.println("The input name is not valid.");
        return;
        }

    if (taskRecorder.existsTask(parameters[0])) {
        System.out.println("The task has been created.");
        return;
        }

    String description = parameters[1];
    if (!Task.isValidDescription(description)) {
        System.out.println("The input description is not valid.");
        return;
        }
```

```
        if (!Task.isValidDurationInput(parameters[2])) {
        System.out.println("The input duration
            contains invalid characters.");
        return;
        }

        double duration = Double.parseDouble(parameters[2]);
        if (!Task.isValidDuration(duration)) {
        System.out.println("The input duration
            is not a positive number.");
        return;
        }

        if (!Task.isValidPrerequisiteInput(parameters[3])) {
System.out.println("The input prerequisite is not valid.
            Your input should only be comma-separated.");
        return;
        }

        String[] prerequisiteNames = parameters[3].split(",");
        if (!allExist(prerequisiteNames, taskRecorder)) {
System.out.println("The input prerequisite(s) has not been created.");
        return;
        }

}
```

Most of the examination methods are of general use in other executors and defined in *Task*. They are as follows:

```
        public static boolean isValidName(String name) {
        if (name == null) {
        return false;
        }
        if (name.length() > 8) {
        return false;
        }
        if (Character.isDigit(name.charAt(0))) {
        return false;
        }
        for (char c : name.toCharArray()) {
        if (!Character.isLetterOrDigit(c)) {
        return false;
        }
        }
        return true;
        }

      public static boolean isValidDescription(String description) {
      for (char c : description.toCharArray()) {
      if(!Character.isLetterOrDigit(c) c != '-') {
      return false;
```

```
            }
        }
        return true;
        }

    public static boolean isValidDuration(double duration) {
    return !(duration <= 0);
    }

        public static boolean isValidDurationInput(String durationInput) {
        try {
        Double.parseDouble(durationInput);
        return true;
        } catch (NumberFormatException e){
        return false;
        }
        }

        public static boolean
        isValidPrerequisiteInput(String prerequisiteInput) {
        return prerequisiteInput.contains(",");
        }

    public static boolean isValidSubtaskInput(String subtaskInput) {
    return isValidPrerequisiteInput(subtaskInput);
    }
```

After the examination of validity, we will construct a new primitive task and the task will be recorded. At that point, the status of the executor will be set to be *true* because the corresponding instruction is able to be undone and is successfully executed. Otherwise, the status remains *false*. The undo logic of *CreatePrimitiveTaskExecutor* is just to delete the created task from the *taskRecorder*.

3. Most of the error handling strategies have been shown within the examination section. In all, the program will end the executor and ensure that there is no side effect in case of users' incorrect input command-line. Users can use the instruction again if his or her input is examined to be wrong.

[REQ2]

1. The requirement has been implemented by Wang Ruijie.

2. Similar to *CreatePrimitiveTask*, the program will do examination on the user's input. The only difference is that the subtasks of the creating *CompositeTask* will be examined to find out the existence of them. On the basis of given *subtasks*, the executor will derives the *directPrerequisites*, *indirectPrerequisites* and *duration* of the creating *CompositeTask* automatically. We will introduce these important mechanisms in the sections of *PrintTask* and *ReportEarliestFinishTime* later. Likewise, the undo logic is to delete the created task from the *taskRecorder*.

3. The error handling strategies are the same as *CreatePrimitiveTask*.

[REQ3]

1. The requirement has been implemented by Wang Ruijie and Zhu Jin Shun.

2. Firstly, the executor examines whether the task to be delete exists in the *taskRecorder*. Secondly, the task to be delete will be checked if it is one of the subtasks of a composite task or a prerequisite of any other task. Before deleting the task from the *taskRecorder*, the executor will

store the task into its field *List<Task> deletedTasks* in case of undo. While redoing the *DeleteTask*, the program can read the list and put the task into the *LinkedHashMap* of the *taskRecorder* again.

When trying to delete a composite task, the program will recursively check the subtasks (TMS permits that the subtask of a subtask is a composite task), and, if applicable, recursively delete the subtasks before deleting the composite task itself. When Redoing, the subtasks can be retrieved by recursion as well to be added again.

```
public static boolean subtaskCheck(Task[] allExistingTask, Task[] subtasks) {
for (Task subtask: subtasks) {
if (subtask.getSubTasks().length == 0) {
for (Task existingTask : allExistingTask) {
if (contains(existingTask.getDirectPrerequisites(), subtask)
|| contains(existingTask.getIndirectPrerequisites(), subtask)) {
return false;
}
}
} else {
subtaskCheck(allExistingTask, subtask.getSubTasks());
}
}
return true;
}

public void deleteSubtasks(Task task, TaskRecorder taskRecorder) {
Task[] subtasks = task.getSubTasks();
for (Task subtask : subtasks) {
if (subtask.getSubTasks().length == 0) {
taskRecorder.deleteTask(subtask.getName());
} else {
deleteSubtasks(subtask, taskRecorder);
}
}
}

private void collectSubtasks(Task task, List<Task> subtasks) {
for (Task subtask : task.getSubTasks()) {
subtasks.add(subtask);
collectSubtasks(subtask, subtasks);
}
}

private void restoreSubtasks(Task task, TaskRecorder taskRecorder) {
for (Task subtask : task.getSubTasks()) {
taskRecorder.addTask(subtask);
restoreSubtasks(subtask, taskRecorder);
}
}
```

3. Error handling is exactly the same as other executors.

[REQ4]

1. The requirement has been implemented by Wang Ruijie and Zhu Jin Shun.

2. Similar to the previous requirements, the executor also does examination on user input with the identical criteria on *name*, *description*, etc.. However, what should be paid attention to is that the executor refuses the command-lines intending to change the subtasks of a primitive task or the duration and prerequisites of a composite task through the following code:

```
Task changingTask = taskRecorder.getTask(taskName);
boolean isValidProperty = (changingTask instanceof PrimitiveTask
? Objects.equals(property, "name") ||
Objects.equals(property, "description") ||
Objects.equals(property, "duration") ||
Objects.equals(property, "prerequisites")
: Objects.equals(property, "name") ||
Objects.equals(property, "description") ||
Objects.equals(property, "prerequisites") ||
Objects.equals(property, "subtasks"));
```

We note that it is of great importance to update all other tasks' property after *ChangeTask* of one existing task is committed, and that is because the change of a property of one existing task might do impact on others'. E.g., the change of the (direct) prerequisite task of a primitive task will influence other tasks' (indirect) prerequisite or other composite tasks' duration. Therefore, we need to update all other tasks. We take the case of changing prerequisites of a primitive task as an example to show this mechanism. Other properties such as *name*, *duration* and *subtasks* also share it.

```
switch (property) {
case "prerequisites":

        // Check if newValue is valid for the description property
        if (isValidPrerequisiteInput(newValue)) {

            String[] prerequisiteNames = newValue.split(",");

                    if (TaskRecorder.allExist(prerequisiteNames, taskRecorder)
                    || Objects.equals(prerequisiteNames[0], "")) {
                    Task[] prerequisites =
                    Task.getTaskList(prerequisiteNames, taskRecorder);
                    this.changedProperty = "prerequisites";
                    this.changedContent =
                    changingTask.getDirectPrerequisites();
                    changingTask.setDirectPrerequisites(prerequisites);

                            // Update all
                            // Derive the directPrerequisites of all others
                            for (Task task : taskRecorder.getAllTasks()) {
                            task.setDirectPrerequisites(task.getDirectPrerequisites());
                            if (task instanceof CompositeTask) {
                            ((CompositeTask) task)
                            .setSubTasks(task.getSubTasks());
                            }
                            }
                            this.status = true;
```

```
                    } else {
                    System.out.println("Invalid prerequisites.
                Please provide a valid prerequisites.");
            }
            } else {
            System.out.println("Invalid prerequisite input.");
        }
        break;
        }
```

The undo logic of *ChangeTask* is to call another *ChangeTaskExecutor* to change the changed task back to the previous status. For example if a task's name has been changed by input *ChangeTask doJava name doOCaml*, we will create a new parameters which is able to describe how to call the executor can the task can be changed back, which is the same as by inputting *ChangeTask doOCaml name doJava*.

```
switch (getChangedProperty()) {

        case "name":
        newParameters[0] = parameters[2];
        newParameters[2] = parameters[0];
        break;

        case "description": case "duration":
        newParameters[0] = parameters[0];
        newParameters[2] = (String) getChangedContent();
        break;

            case "prerequisites": case "subtasks":
            newParameters[0] = parameters[0];
            StringBuilder prerequisiteNameString = new StringBuilder();
            prerequisiteNameString.append(",");
            for (Task prerequisite : (Task[]) getChangedContent()) {
            prerequisiteNameString.append(prerequisite.getName()).append(",");
            }
            newParameters[2] = prerequisiteNameString.toString();
            break;

}
```

Then we call another executor to change the task back.

3. The error handling operation is to point out the incorrect input and to let users to input their command-line again.

[REQ5]

1. The requirement has been implemented by Wang Ruijie.
2. This requirement is easy for printing the information of a primitive task. We will focus on the derivation of *indirectPrerequisites* of all tasks, *duration* of composite tasks, and *indirectPrerequisites* and *directPrerequisites* of composite tasks in our explanation, as they are not input by users from their command-lines.

For *indirectPrerequisites*, the executor recursively examines each of the direct prerequisite tasks and collects their prerequisites until there is no prerequisite task, and all the duplicate tasks are removed. If the direct prerequisite tasks of a composite task have been already derived, the method is the same for *CompositeTask*. Please note that, a prerequisite task of a composite task can not be the subtask of the same composite task.

```
public static Task[] deriveIndirectPrerequisites(
Task[] directPrerequisites, Task[] subTasks) {
List<Task> indirectPrerequisites = new ArrayList<>();
for (Task prerequisite : directPrerequisites) {
indirectPrerequisites.add(prerequisite);
addIndirectPrerequisites(prerequisite,
indirectPrerequisites, subTasks);
// Ensure that there are no duplication
}
for (Task prerequisite : directPrerequisites) {
    indirectPrerequisites.remove(prerequisite);
}
return
indirectPrerequisites.toArray(new
Task[0]);
}

private static void addIndirectPrerequisites(Task task,
List<Task> indirectPrerequisites, Task[] subTasks) {
if (task == null) {
return;
}
for (Task directPrerequisite : task.getDirectPrerequisites()) {
if (!indirectPrerequisites.contains(directPrerequisite)
!contains(subTasks, directPrerequisite)) {
// Can not be contained in the subtasks of a composite task
indirectPrerequisites.add(directPrerequisite);
}
// Recursion until a task has no prerequisite
addIndirectPrerequisites(directPrerequisite,
indirectPrerequisites, subTasks);
}
}
```

When calculating the duration time of a composite task with subtasks, the executor gets the maximum sum the duration of the subtasks that are in the same *prerequisite task chain*, and there might be multiple such chains within one composite task. The meaning of the *prerequisite task chain* is shown in the figure below: Consider that *task1* to

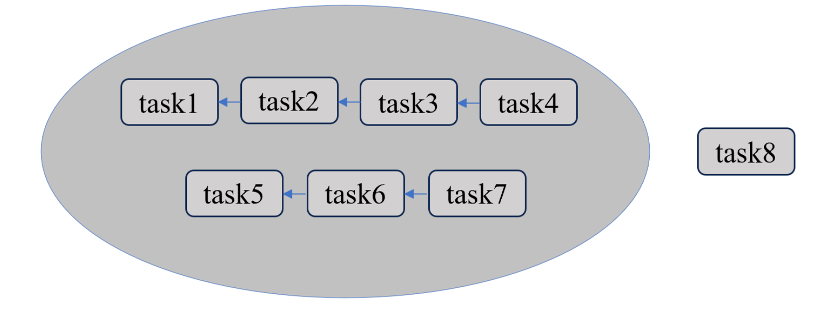

Figure 4: Calculation of the duration of the composite tasks

task7 are all subtasks of *task8*, and the arrows represents the relationship of being as the prerequisite task. In this case, there exists two *prerequisite task chains*, and the maximum between $\sum_{i=1}^{4} task_i.duration$ and $\sum_{j=5}^{7} task_j.duration$ is the duration of *task8*. In case that a composite task is the subtask of another composite, the executor uses recursion to obtain the duration of each subtask.

```
public static Task[] deriveDirectPrerequisites(Task[] subTasks) {
List<Task> directPrerequisites = new ArrayList<>();
for (Task subTask : subTasks) {
if (subTask.getType()) {
for (Task directPrerequisite : subTask.getDirectPrerequisites()) {
if (!directPrerequisites.contains(directPrerequisite)
!contains(subTasks, directPrerequisite)) {
directPrerequisites.add(directPrerequisite);
}
}
} else {
CompositeTask compositeSubTask = (CompositeTask) subTask;
deriveDirectPrerequisites(compositeSubTask.getSubTasks());
}
}
return directPrerequisites.toArray(new Task[0]);
}

private static void calculateTotalDuration(Task[] subTasks,
Map<Task, Double> maxDurationMap) {

double totalDuration = 0;

// subtask traversal
for (Task subTask : subTasks) {
double subTaskDuration = 0;

        // If the subtask is a simple task
        // add its duration directly
        if (subTask.getType()) {
        subTaskDuration = subTask.getDuration();
        }
        // If the subtask is a composite task
        // recursively calculate the duration
            else {
```

```
CompositeTask compositeSubTask = (CompositeTask) subTask;
calculateTotalDuration(compositeSubTask.getSubTasks(),
maxDurationMap);
}
// Update
maxDurationMap.put(subTask, subTaskDuration);
totalDuration = Math.max(totalDuration, subTaskDuration);
}

for (Task subTask : subTasks) {
calculateTotalDurationRecursion(subTask, maxDurationMap);
}
}

private static double deriveDuration(Task[] subTasks, Map<Task, Double> maxDurationMap) {
calculateTotalDuration(subTasks, maxDurationMap);
Map.Entry<Task, Double> maxEntry = null;
for (Map.Entry<Task, Double> entry : maxDurationMap.entrySet()) {
if (maxEntry == null || entry.getValue().compareTo(maxEntry.getValue()) > 0) {
maxEntry = entry;
}
}
assert maxEntry != null;
return maxEntry.getValue();
}
```

After every necessary information of a task is prepared, the executor will print them based on the task's type. E.g., the information about *subtasks* of a primitive task is not displayed as output.

3. If the task to print has not been created yet, nothing will be output on the screen.

[REQ6]

1. The requirement has been implemented by Wang Ruijie.
2. The mechanism of *PrintAllTasks* is to iteratively call *PrintTask* for all tasks recorded by *taskRecorder*.
3. If no task exists, nothing will be printed on the screen.

[REQ7]

1. The requirement has been implemented by Wang Ruijie.

2. The difficulty of this requirement lies in calculating the duration of the composite task, and we have already discussed it above in [REQ5] *PrintTask*. Once *ReportDuration* is input with a task name, the executor will read the duration of the task and print it on the screen.

3. The executor examines the existence of the task input, and nothing will be printed if the task has not been created yet.

[REQ8]

1. The requirement has been implemented by Wang Ruijie and Zhu Jin Shun.

2. We have discussed the method of deriving all indirect and direct prerequisite task of a specific task. Since then, the executor of *ReportEarliestFinishTime* simply calculates the sum of the duration of all the prerequisites and the task itself, and finally print it on the screen. The process will

mainly be Frist ,to calculate the earliest finish time, we initialize a double value named EarliestFinishTime to store the duration of the task. We then iterate through the duration of the reportingTask itself and add it to the EarliestFinishTime. Next, we check if there are any prerequisites or subtasks associated with the reportingTask. If there are, we continue to iterate through all the prerequisites and subtasks to obtain their respective duration times and add them to the EarliestFinishTime. Once all the calculations are completed, we print out the final result, which represents the earliest finish time for the reporting task..

3. If $task_1$ is the prerequisite task of $task_2$, and $task_2$ is the prerequisite task of $task_1$, our system regards them as a pair of tasks that has to be done concurrently, and the time for finishing both is $task_1.duration + task_2.duration$, as concurrency does not mean that the total finish time equals to the maximum of the two durations. The system does not recognize such situations as a *dead lock* error. Also, the executor examines the existence of the task input, and nothing will be printed if the task has not been created yet.

[REQ9]

1. The requirement has been implemented by Zhu Jin Shun.

2. We implemented *DefineBasicCriterion* by accepting four values from the user input and transferring them as a string for the name, property, op, and object for value. Then, we checked if the name is repeated in the criteria recorder and if the values and property are valid according to the conditions and some of the is valid methods written in the criterion class. If not, the command will return. If all the inputs are valid, we created a new criterion based on the user input using the fields we created and recorded it back into the criterion recorder as a basic type. As a result, a new basic criterion is created for this method.

3. There were several errors encountered when creating the basic criterion. We needed to ensure that all inputs are valid. For the value part, we made a mistake by initially accepting only integer values for duration, which is incorrect, as values can be any real numbers. To rectify this, we changed the data type to double, allowing all real values to be accepted and used for duration comparison. Another mistake was that when checking the validity of the criterion's name, we only verified if it met the basic requirements for a name (such as containing only English letters and digits, not starting with a digit, and having a maximum length of eight characters). However, we failed to check whether the name had already been used in the *criterionRecorder*. To address this issue, we added an additional step to check if the criterion's name already exists in the *criterionRecorder*. If it does, we consider it invalid.

[REQ10]

1. The requirement has been implemented by Wang Ruijie.

2. The special criterion *IsPrimitive* is regarded as a built-in criterion that the user does not need to create such a criterion manually. Meanwhile, *IsPrimitive* is not recorded by *criterionRecorder*, as it can not be created by the user. The user can input the commandline *Search IsPrimitive* to filter the existing tasks, and all primitive tasks will be displayed. Hence, in TMS, *IsPrimitive* is a keyword for *Search* rather than a technical *Criterion*.

3. If there exists no primitive task, no task will be displayed. TMS blocks the user to manually create a criterion named *IsPrimitive* to ensure that the function will not be covered throughout the system.

[REQ11]

1. The requirement has been successfully implemented by Zhu Jin Shun.

2. For the negated criterion, we accept two values from the user. Firstly, we check if the second value corresponds to an existing criterion in the criterion recorder. If it is not found, we return. Next, we verify if the name for the new negated criterion is already used in the criterion recorder. If it is, we return to avoid duplicates.

If all the conditions are valid, we create a new negated criterion by inserting the existing criterion as the sub criterion1 for the new criterion. The user-defined name is assigned to the name of the new criterion, *negated* is set as the *logicOp*, and the type is specified as *negated*. Finally, we add the new negated criterion to the criterion recorder.

Regarding the binary criterion, we accept four values from the user. Initially, we check if any of the three names provided are repeated. If a repetition is found, we return to prevent duplicates. Then, we ensure that both name2 and name3 exist in the criterion recorder. If any of them is not found, we return.

Additionally, we verify if name1 is already present in the criterion recorder. If it is, we return to avoid duplications. Furthermore, we check if the *logicOp* entered by the user is either && or ||. If it is not, we return.

If all these conditions are met, we create a new binary criterion named name1. The name2 is assigned to sub-criterion1, the sub name3 is assigned to criterion2, the logicOp is set as either *and* or *or* based on user input, and the type is specified as *binary*. Finally, we add the new binary criterion to the criterion recorder.

3. The executor will do basic examination of the validity of the input of fields such as *name*. During the validation of the entered names, we encountered some error conditions. We realized that the conditions for checking the validity of the names should ensure that all names entered are not repeated. Previously, the conditions were separated using the *or* operator, which allowed passing the test if at least one name was valid. This led to code that did not fulfill the necessary requirements. To rectify this, we modified the conditions to ensure that all names are validated for duplication.

DefineNegatedCriterionExecutor does not allow users to input a existing composite task (negated or binary) to be the task, and relevant examination will be done. However, *DefineNegatedCriterionExecutor* accepts such input.

Once the errors mentioned above occurs, the executor will return and the user can correct his or her input.

[REQ12]

1. The requirement has been implemented by Zeng Tianyi

2. Implemented by utilizing the class named CriterionRecorder that defined by us. We first create an array of type Criterion and assign all the criteria in the input criterionRecorder to this array. We then use for-loop to traverse this array. To do the traversal, we first consider the type of each criterion, i.e., *Basic, Binary*, or *Negated*. If a criterion corresponds to the type Basic, we print its property name, op, and value, and print a message. If a criterion corresponds to the type *Binary*, we print the property names of its first sub-criterion and second sub-criterion, the ops and the values of the two criteria, and print a message to indicate the logicOp of this binary criterion. If a criteria corresponds to the type *Negated*, we first print the property name of its first sub-criterion (i.e., the criterion to be negated). Then, we will evaluate the op of the first sub-criterion, do a corresponding negation operation (e.g., if the op of the first sub-criterion is >, we set the op of the criterion of type *Negated* as <=), and print it out. After that, we print the value of the Negated-type criterion. At last, we indicate that the logic op of the Negated-type criterion is negation.

3. If no criterion has been created before *PrintAllCriteria*, no information will be displayed on the screen.

[REQ13]

1. The requirement has been implemented by Zeng Tianyi, Zhu Jin Shun and Wang Ruijie.

2. When searching on a basic criterion or a negated, the *SearchExecutor* does traversal on all task records and filters those meeting the property. Specifically for *NegatedCriterion*, the executor finds the complementary set of the tasks that are searched based on the criterion being negated. For example, if basic criterion $criterion_1$ indicates $task_1$, and in the *taskRecorder* there

exists three tasks in total, $task_1$, $task_2$ and $task_3$, the executor will return the complementary set of set {$task_1$}, i.e., {$task_2$, $task_3$}.

However, when searching on a binary criterion, the executor adapts the strategy of recursion to find out all tasks satisfied the criterion, as TMS allows users to define a binary criterion based on one or two previously defined binary criteria. In that case, the executor needs to extract the tasks indicated by its sub-criteria and do set operation on the extracted task list, based on the given *logicOp*.

```
public List<String> search(Criterion criterion, TaskRecorder taskRecorder) {

    List<String> resultTaskNameList = new ArrayList<>();

        if (Objects.equals(criterion.getType(), "Basic")) {
        resultTaskNameList = searchOnBasic(criterion, taskRecorder);
        }
        if (Objects.equals(criterion.getType(), "Negated")) {
        resultTaskNameList = searchOnNegated(criterion, taskRecorder);
        }
        if (Objects.equals(criterion.getType(), "Binary")) {

        List<String> firstList = new ArrayList<>();
        List<String> secondList = new ArrayList<>();

                // Recursion
                if  (Objects.equals(criterion.getFirstSubCriterion().getType(),  "Basic"))
                {
                firstList.addAll(searchOnBasic
                (criterion.getFirstSubCriterion(), taskRecorder));
                }
                if (Objects.equals(criterion.getFirstSubCriterion().getType(), "Negated"))
                {
                firstList.addAll(searchOnNegated
                (criterion.getFirstSubCriterion(), taskRecorder));
                }
                if (Objects.equals(criterion.getFirstSubCriterion().getType(), "Binary"))
                {    firstList.addAll(search
                (criterion.getFirstSubCriterion(),
                taskRecorder));
                }

                if  (Objects.equals(criterion.getSecondSubCriterion().getType(),"Basic"))
                {
                secondList.addAll(searchOnBasic
                (criterion.getSecondSubCriterion(), taskRecorder));
                }
                if            (Objects.equals(criterion.getSecondSubCriterion().getType(),
                "Negated")) {
                secondList.addAll(searchOnNegated
                (criterion.getSecondSubCriterion(), taskRecorder));
                }
                if (Objects.equals(criterion.getSecondSubCriterion().getType(), "Binary"))
                {
                secondList.addAll(search
                (criterion.getSecondSubCriterion(), taskRecorder));
```

```
                }

                    if (Objects.equals(criterion.getLogicOp(), "and")) {
                    List<String> intersection = firstList.stream()
                    .filter(secondList::contains)
                    .collect(Collectors.toList());

            resultTaskNameList.addAll(intersection);
            }

            if (Objects.equals(criterion.getLogicOp(), "or")) {
            resultTaskNameList.addAll(firstList);
            resultTaskNameList.removeAll(secondList);
            resultTaskNameList.addAll(secondList);
            }

    }

    return resultTaskNameList;

}
```

3. The error handling is main about if the searched exists, and the executor will return if it is null.

[REQ14]

1. The requirement has been implemented by Wang Ruijie.
2. When *StoreExecutor* is constructed, it will go through the *LinkedHashMap* of the *taskRecorder* and *criterionRecorder* to extract the information of tasks and criteria which is necessary to create or define them again. For example, we must know the type of a specific task or criterion to utilize the correct executor to create or define it. After that, we use *try (FileWriter writer = new FileWriter(filePath))* new detect the accessability of the file or the directory the file belongs to. If the file does not exist, a new file with the given name will be created, and the executor will write in the file. The executor uses the prefixes of *"task: "* and *"criterion: "* to indicate the contents stored, which is also convenient for loading. For each piece of information, the elements are space-separated.

The idea of using *LinkedHashMap* rather than *HashMap* in the recorders is that the order of being created or defined shall be maintained during the *Store* and *Load* operations to ensure successful loading. For example, if a composite task is loaded prior to the loading of its subtasks, there will be an error. Therefore, such error shall be eliminated during *Store*. An example of *Store* after the user input *CreatePrimitiveTask t1 source-code 10 , ,* *CreatePrimitiveTask t2 test-code 5 ,*, *CreateCompositeTask t3 OOP-project t1,t2*, *DefineBasicCriterion c1 duration > 8*, *DefineNegatedCriterion c2 c1*, and *DefineBinaryCriterion c3 c1* && *c2* is as follows:

```
TASK:
task: t1 source-code 10.0 , Primitive ,
task: t2 test-code 5.0 , Primitive ,
task: t3 OOP-project 10.0 , Composite t1,t2,
CRITERION:
criterion: c1 Basic duration > 8
criterion: c2 Negated c1
```

```
criterion: c3 Binary c1 c2 and
```

3. The most common error condition is that the operating system block the program to read or write some files or such directories or files is without accessability. In such cases, an *IOException* will be thrown.

[REQ15]

1. The requirement has been implemented by Wang Ruijie.

2. The *LoadExecutor* will visit the file with given filepath and read the contents line by line. For each line of contents, the executor determines which instructions to be constructed according to the prefix and the type given, and combine the elements into a *String[] newParameters* according to the command-line format. Then, the executor constructs a new *commandRecord* and it will be pushed into the *executedStack* after the *newParameters* is executed along with the corresponding *instructionExecutor*. Hence, although the instruction of *Load* can not be directly undone, the user can undone what is loaded in order.

3. We use *LinkedHashMap* to prevent the incorrect order of loading. If the file to load does not exist, the executor will throw an *IOException*.

[REQ16]

1. The requirement has been implemented by Wang Ruijie

2. Once *QuitExecutor* is called, the program will terminate by *System.exit(0)*. Hence, there is no need to break the loop in *main*.

3. No error situation exists.

[BON1]

1. This requirement has not been implemented.

[BON2]

1. This requirement has been implemented by Wang Ruijie.

2. For better storage of history command-lines, we define a static class (data structure), *CommandRecord*, inside *HistoryCommandRecorder*, where we combine a *executor* that is constructed for one user command-line, and the corresponding *parameters* as an instance stored in *commandRecorder*. Afterwards, two stacks for storing executed (also redone) *commandRecords* and undone *commandRecords* respectively are established.

```java
// HistoryCommandLineRecorder.java
// Fields of the HistoryCommandLineRecorder class

private final Stack<CommandRecord> executedCommands;
private final Stack<CommandRecord> undoneCommands;

    public HistoryCommandRecorder() {
    this.executedCommands = new Stack<>();
    this.undoneCommands = new Stack<>();
    }
```

Once an *instructionExecutor* is successfully executed and it is meaningful to undo or redo the instruction, its *status* will be set to be *true*. TMS checks the status of each constructed executors and those with *true* status will be pushed into the stack *executedCommands*. When undoing them, the corresponding *commandRecords* will be popped and pushed into the stack *undoneCommands*. In the situation that the old was undone and a new executor constructed is with *true* status, the stack *undoneCommands* will be cleared, as TMS does not allow insert new instructions in between *Undo* and *Redo*. Those methods are defined as follows:

```java
// HistoryCommandLineRecorder.java

    public void pushExecuted(CommandRecord executedCommand) {
    this.executedCommands.push(executedCommand);
    this.undoneCommands.clear();
    }

    public CommandRecord getUndoingCommand() {
    CommandRecord undoingCommand = this.executedCommands.pop();
    this.undoneCommands.push(undoingCommand);
    return undoingCommand;
    }

    public CommandRecord getRedoingCommand() {
    CommandRecord redoingCommand = this.undoneCommands.pop();
    this.executedCommands.push(redoingCommand);
    return redoingCommand;
    }
```

The reason of using stacks is that, when undoing or redoing, the instructions always obey *First-In-Last-Out*.