Project Report
Group 3
COMP2021 Object-Oriented Programming (Fall 2023)
Author: Tahmin Anower
Other group members: Angad Singh Grover, Sidtharth Sreekumar,Ilyas Akramov

1 Introduction

This document describes group 3'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

In the development of the Command-Line Task Management System (TMS) project, I actively contributed to several key aspects, showcasing a clear distinction in my role compared to other group members. Here is a breakdown of my specific contributions:

Requirement Implementation:

- I took a lead role in implementing REQ3, REQ4, REQ5, and REQ6 of the project requirements.
- I took a helping role in implementing REQ1 and REQ2 of the project requirements.

- For REQ3, I designed and implemented the task deletion functionality, ensuring that simple primitive tasks could be deleted if not prerequisites and composite tasks could be deleted without affecting other tasks.

- In REQ4, I implemented the ChangeTask command, enabling users to modify task properties such as name, description, duration, and prerequisites. I ensured

the system handles both primitive and composite tasks appropriately.

- REQ5 was addressed with the implementation of the PrintTask command, allowing users to view detailed information about a specific task.

- For REQ6, I designed and implemented the PrintAllTasks command, providing a comprehensive overview of all existing tasks in a readable format.

User Manual Development:

- I played a pivotal role in drafting the user manual for the application, collaborating with team members to ensure clarity, completeness, and user-friendliness.
- The user manual covers all aspects of the TMS, including detailed instructions on using each command, examples, and troubleshooting guidelines.

Presentation Slides:

- Collaborated with the team in creating presentation slides for project showcases.
- Ensured that the slides effectively communicated the functionality of the TMS, emphasizing the implemented requirements and their significance.

Code Review and Integration:

- Actively participated in code review sessions, providing constructive feedback to enhance code quality and adherence to project requirements.

- Collaborated in integrating individual components developed by team members to ensure seamless functionality of the entire TMS.

Communication and Coordination:

- Maintained effective communication within the team, ensuring that everyone was on the same page regarding project progress, challenges, and solutions.
- Coordinated efforts to align individual contributions, fostering a cohesive and well-integrated final product.

Additional Contribution to the Code:

- In addition to formulating the core logic of the code, my responsibilities encompassed the composition and definition of each class, method, and parameter for the provision of Javadoc documentation. This entailed providing concise explanations of the code and its parameters, along with anticipated output and its type. Furthermore, I contributed to the refinement of the codebase by addressing and rectifying minor errors, including instances of magic numbers, occurrences of instanceof errors, and the removal of redundant code files.

3 A Command-Line Task Management System

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

3.1 Design

Concerning the design of the system, we have introduced a total of three main sections, namely, model, utilities and Application - with Application serving as the main driver code that initializes all instances of the data variables found in our codebase whilst also maintaining the mainloop that governs our system.

The main loop is the section of the code that keeps requesting for user input until it is terminated using the Quit feature. It is essentially a while (true) loop with the Quit command being its break condition.

In addition to application, we have the model folder consisting of the classes TMS, PrimitiveTask and CompositeTask, and also the folder titled utilities which consists of the classes Criterion, DefineBinaryCriterion, DefineBasicCriterion, and DefineNegatedCriterion.

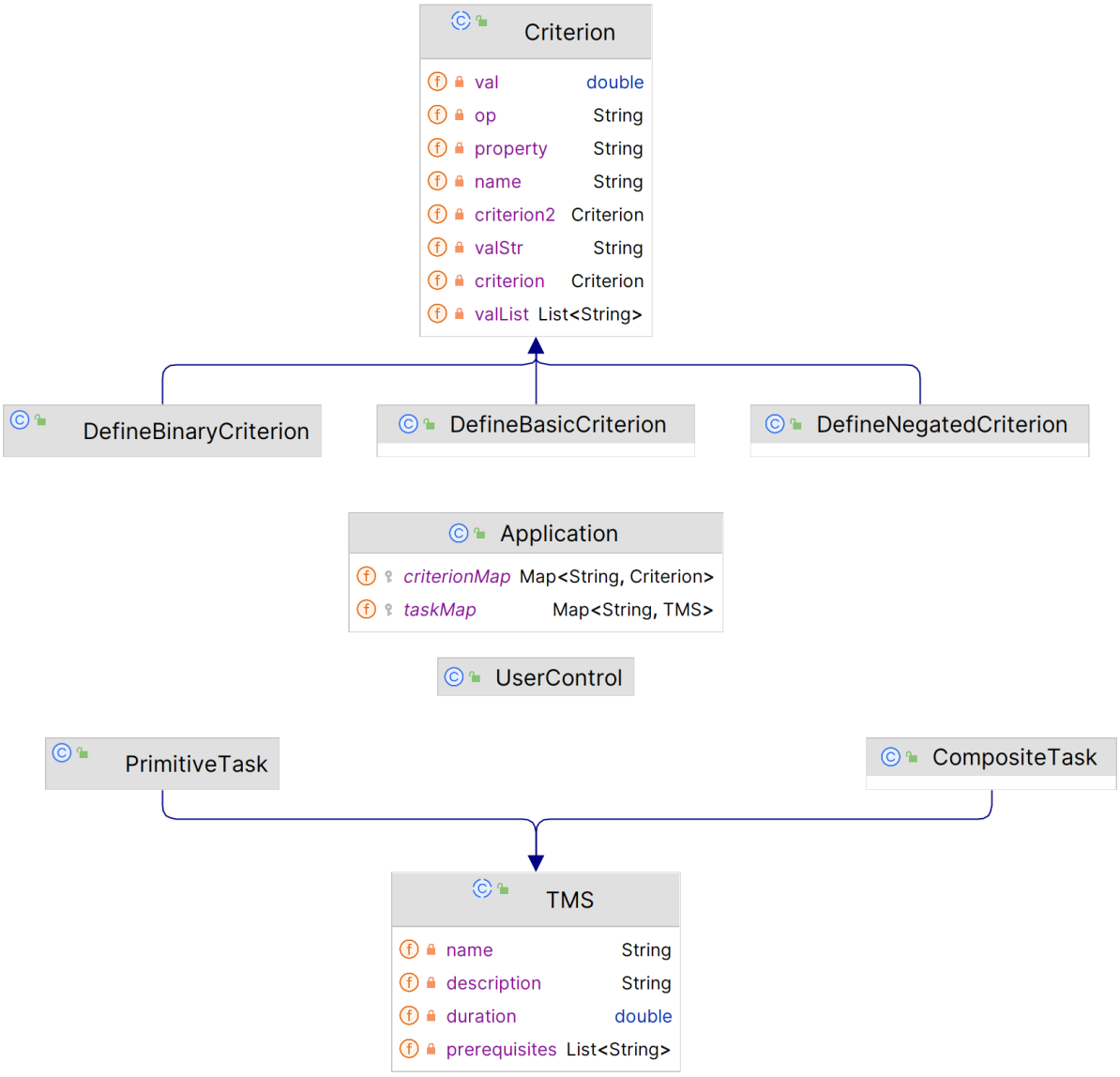


Figure: Simplified Class Diagram

In our implementation, we have two main classes, TMS and Criterion, both of which are abstract classes that best utilizes the concept of abstraction and polymorphism to allow our code to become modular and highly re-usable. The other classes mentioned in the diagram are sub classes that inherit the fields and methods defined within these abstract classes.

Finally, in order to enhance data retrieval and modification efficiency, we instituted the incorporation of two hash maps, designated for the categorization of tasks and criteria, respectively.

They undergo expeditious accesses and modifications predicated upon distinct names as keys, obviating the requirement for protracted search operations, iterative procedures, file manipulations, or report calculations.

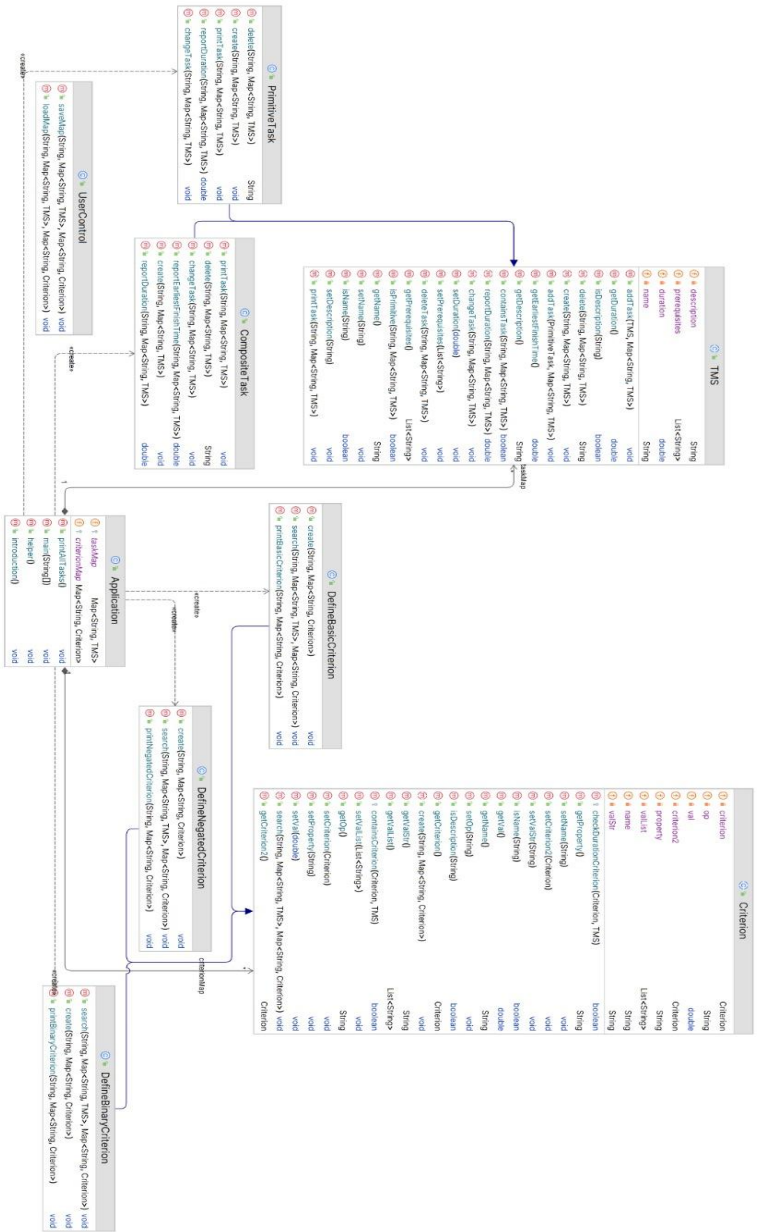


Fig: Detailed Class Diagram

3.2 Requirements

For each (compulsory and bonus) requirement, describe 1) whether it is implemented and, when yes, 2) how you implemented the requirement as well as 3) how you handled various error conditions.

HIGHLIGHTED ONES IS MY PERSONAL CONTRIBUTION

[REQ1] CreatePrimitiveTask

1) **The requirement is implemented.**

```java
/** Method to create a Primitive Task
 * This method expects an instruction string and a Map that stores tasks
 * @param instruction contains a string representation of entire user input
 * @param taskMap contains a map that stores the user information*/
@Override
public void create(String instruction, Map <String, TMS> taskMap) {
    // Method to create an object of the primitive task
    String[] tokens = instruction.split( regex: " ");
    if (tokens.length >= 5 && tokens[4].strip().equals(",") && isName(tokens[1])) {
        String name = tokens[1];
        String description = tokens[2];
        double duration = Double.parseDouble(tokens[3]);
        //List<String> prerequisites = Arrays.asList(tokens[4].split(","));
        if ((!taskMap.containsKey(name)) || taskMap.isEmpty()) {
            TMS TMS = new PrimitiveTask(name, description, duration);
            taskMap.put(name, TMS);
            System.out.println("Simple task created: " + name);
        } else {
            System.out.println("Task with the same name already exists: " + name);
        }
        //System.out.print (prerequisites);
    } else {
        System.out.println("Invalid CreateSimpleTask command format.");
    }
}
```

2) **Implementation Details:**

- The code uses a conditional statement to check if the input command starts with "CreatePrimitiveTask."
- It then splits the input string into tokens to extract relevant information such as task name, description, duration, and prerequisites.
- Checks are in place to ensure that the input format is correct, including verifying the presence of a comma after duration and validating the task name.
- The code utilizes a map (taskMap) to manage tasks, ensuring that no task with the same name already exists before creating a new one.
- If the conditions are met, a new PrimitiveTask object is created, added to the task map, and a success message is printed. If the task with the same name already exists, an appropriate error message is displayed.

3) **Error Conditions and How They Are Handled:**

- **Invalid Format:**
  - If the input command does not follow the expected format, an "Invalid CreateSimpleTask command format" message is printed, ensuring users are aware of the incorrect input.
- **Existing Task Name:**
  - If a task with the same name already exists in the task map, an error message is displayed: "Task with the same name already exists." This prevents the creation of tasks with duplicate names, maintaining uniqueness.

- **Missing Task Name:**
  - The implementation checks if the task name is valid (using the isName method). If not, it ensures that an appropriate error message is displayed.
- **Empty Prerequisites:**
  - The code checks if the prerequisites field is empty (only containing a comma). If so, it proceeds with task creation. This accounts for the case where a simple primitive task may not have prerequisites.

[REQ 2] CreateCompositeTask

1) The requirement is implemented.

```java
/**Method to create a Composite Task
 * This method expects an instruction string and a Map that stores tasks
 * the instruction is split to obtain the required information for task creation
 *
 * @param instruction contains string representation of entire user input
 * @param taskMap contains a map that stores the user information*/
@Override
public void create(String instruction, Map<String,TMS>taskMap) {
    // add code
    String[] tokens = instruction.split( regex: " ");
    if (tokens.length >= 4 && isName(tokens[1])) {
        String name = tokens[1];
        String description = tokens[2];
        String[] subtaskNames = tokens[3].split( regex: ",");
        if (!taskMap.containsKey(name)) {
            TMS TMS = new CompositeTask (name, description, subtaskNames); // Duration is 0 for co
            for (String subtaskName : subtaskNames) {
                if (!taskMap.containsKey(subtaskName)) {
                    System.out.println("Failed to Create Composite Task.\nSubtask not found: " + s
                    return;
                }
            }
            taskMap.put(name, TMS);
            System.out.println("Composite task created: " + name);
        } else {
            System.out.println("Task with the same name already exists: " + name);
        }
    } else {
        System.out.println("Invalid CreateCompositeTask command format.");
    }
```

2)Implementation Details:

- The code begins by checking if the input command starts with "CreateCompositeTask."
- It then splits the input string into tokens to extract relevant information such as the task name, description, and an array of subtask names.
- Checks are in place to ensure that the input format is correct, including validating the task name and checking for the existence of subtasks in the task map.
- The code utilizes a map (taskMap) to manage tasks, ensuring that no task with the same name already exists before creating a new one.
- If the conditions are met, a new CompositeTask object is created, added to the task map, and a success message is printed. If a subtask is not found, an appropriate error message is displayed.

3) Error Conditions and How They Are Handled:

- **Invalid Format:**
  - If the input command does not follow the expected format, an "Invalid CreateCompositeTask command format" message is printed, guiding users on the correct input.
- **Existing Task Name:**
  - If a task with the same name already exists in the task map, an error message is displayed: "Task with the same name already exists." This prevents the creation of tasks with duplicate names.
- **Missing Subtask:**
  - The code iterates over the array of subtask names, checking if each subtask exists in the task map. If a subtask is not found, it prints a message: "Failed to Create Composite Task. Subtask not found: [subtaskName]."

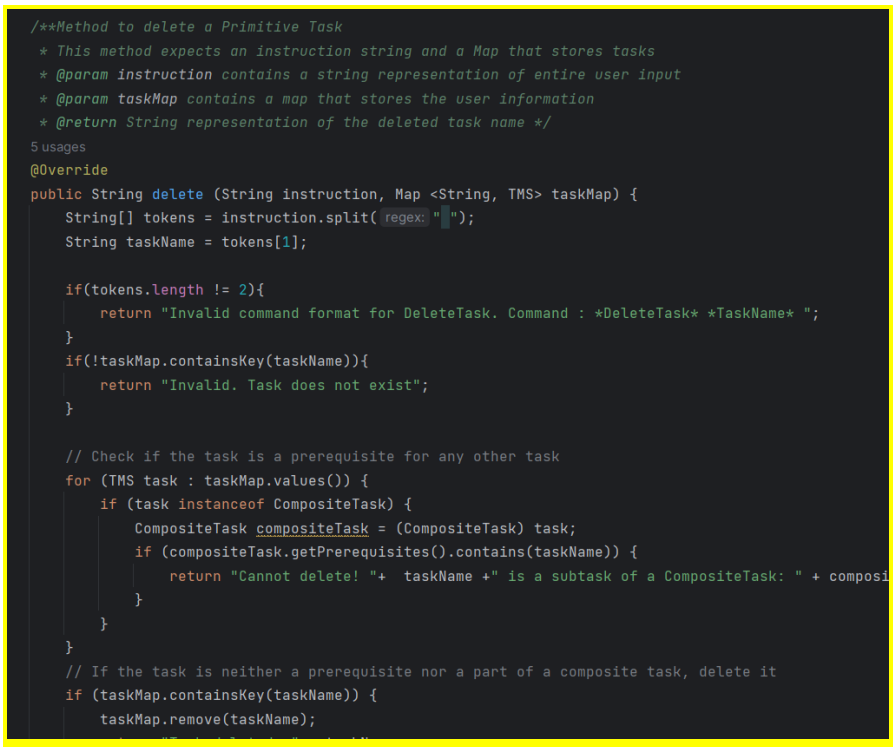
- **Empty Subtasks:**

  - The implementation ensures that the array of subtask names is not empty. If it is, the system does not proceed with task creation and prints an appropriate error message.

.

[REQ 3] DeleteTask

1) **The requirement is implemented.**



```java
/**Method to delete a Primitive Task
 * This method expects an instruction string and a Map that stores tasks
 * @param instruction contains a string representation of entire user input
 * @param taskMap contains a map that stores the user information
 * @return String representation of the deleted task name */
5 usages
@Override
public String delete (String instruction, Map <String, TMS> taskMap) {
    String[] tokens = instruction.split( regex: " ");
    String taskName = tokens[1];

    if(tokens.length != 2){
        return "Invalid command format for DeleteTask. Command : *DeleteTask* *TaskName* ";
    }
    if(!taskMap.containsKey(taskName)){
        return "Invalid. Task does not exist";
    }

    // Check if the task is a prerequisite for any other task
    for (TMS task : taskMap.values()) {
        if (task instanceof CompositeTask) {
            CompositeTask compositeTask = (CompositeTask) task;
            if (compositeTask.getPrerequisites().contains(taskName)) {
                return "Cannot delete! "+  taskName +" is a subtask of a CompositeTask: " + composi
            }
        }
    }
    // If the task is neither a prerequisite nor a part of a composite task, delete it
    if (taskMap.containsKey(taskName)) {
        taskMap.remove(taskName);
```

2) **Implementation Details:**

- Two delete methods are provided, each catering to the deletion of simple primitive tasks and composite tasks, respectively.
- The first method handles the deletion of simple primitive tasks. It checks if the task exists in the task map, and if it does, it checks if the task is a prerequisite for any other composite task. If it is, the system prevents deletion and returns an appropriate message. Otherwise, it deletes the task from the map.
- The second method handles the deletion of composite tasks. It checks if the task exists in the task map and if it is indeed a composite task. It then checks if any of its subtasks are prerequisites for other tasks. If any subtask is a prerequisite, the system prevents deletion and returns an appropriate message. Otherwise, it deletes the composite task and its subtasks from the map.

3) **Error Conditions and How They Are Handled:**

- **Invalid Command Format:**
    - Both methods check if the command format is valid. If not, they return an "Invalid command format for DeleteTask" message.

- **Task Not Found:**
  - If the task to be deleted is not found in the task map, both methods return a "Task not found" message.
- **Task is a Prerequisite for a Composite Task**:
  - The first method checks if the simple primitive task is a prerequisite for any composite task. If it is, deletion is prevented, and a message indicating that the task is a subtask of a composite task is returned.
- **Subtask is a Prerequisite for Another Task:**
  - The second method checks if any subtask of the composite task is a prerequisite for another task. If any subtask is a prerequisite, deletion is prevented, and a message indicating that the subtask is a prerequisite for another task is returned.
- **Deletion Success:**
  - If all conditions are met, and the task or composite task can be safely deleted, a success message is returned.

[REQ4] ChangeTask

1) The requirement is implemented.

```java
/**
 * Modifies a PrimitiveTask based on the provided instruction. <p>
 * This method expects an instruction string and a Map that stores existing tasks. Its main
 * function is to modify an existing task.
 *
 * @param instruction contains string representation of the entire user input that dictates how the
 * @param taskMap contains a map that stores the user's tasks, mapped by their names. The task to b
 */
6 usages
@Override
public void changeTask (String instruction, Map<String, TMS> taskMap) {
    String[] tokens = instruction.split( regex: " ");
    String name = tokens[1];
    String property = tokens[2];
    String newValue = tokens[3];

    if (taskMap.containsKey(name)) {
        PrimitiveTask task = (PrimitiveTask) taskMap.get(name);
        switch (property) {
            case "name":
                task.setName(newValue);
                break;
            case "description":
                task.setDescription(newValue);
                break;
            case "duration":
                task.setDuration(Double.parseDouble(newValue));
                break;
            default:
                System.out.println("Invalid property for a primitive task");
                return;
```

2) Implementation Details:

- Two changeTask methods are provided, each catering to the modification of properties for simple primitive tasks and composite tasks, respectively.
- Both methods split the input instruction to extract the task name, property, and new value.
- The code checks if the task with the given name exists in the task map. If it does, it retrieves the task and performs the property modification based on the switch-case structure.
- For primitive tasks, the property can be "name," "description," "duration," or "prerequisites." For composite tasks, it can be "name," "description," or "subtasks."

- The newValue is converted to the appropriate type (Double or List<String>) depending on the property being modified.

3) **Error Conditions and How They Are Handled:**
- **Task Not Found:**
    - If the task to be modified is not found in the task map, both methods return a "Task not found" message.
- **Invalid Property for Task Type:**
    - If the property to be modified is not compatible with the task type (primitive or composite), an appropriate error message is printed.
- **Invalid Property Name:**
    - If an invalid property name is provided, an appropriate error message is displayed for both primitive and composite tasks.
- **Invalid Value Format:**
    - For the "duration" property of primitive tasks, the code ensures that the newValue can be parsed to a Double. If not, it prints an appropriate error message.
    - For the "subtasks" property of composite tasks, the code ensures that the newValue is a valid comma-separated list of task names.

[REQ 5] PrintTask

1) The requirement is implemented.

```java
/**Method to print a Primitive task
 * This method expects an instruction string and a Map that stores tasks
 * It will format and print all the tasks currently found in the system
 *
 * @param instruction contains string representation of entire user input containing the required
 * @param taskMap contains a map that stores the user information*/
5 usages
@Override
public void printTask(String instruction, Map<String, TMS> taskMap) {
    String[] tokens = instruction.split( regex: " ");
    if (tokens.length >= 2) {
        String name = tokens[1];
        if (taskMap.containsKey(name)) {
            TMS task = taskMap.get(name);
            System.out.println("Task Name: " + task.getName());
            System.out.println("Description: " + task.getDescription());
            if (task instanceof CompositeTask) {
                CompositeTask compositeTask = (CompositeTask) task;
                System.out.println("Subtasks: " + String.join( delimiter: ", ", compositeTask.getPrer
            }
        } else {
            System.out.println("Task not found: " + name);
        }
    } else {
        System.out.println("Invalid PrintTask command format.");
    }}
```

2) Implementation Details:

- Two printTask methods are provided, each catering to the printing of information for simple primitive tasks and composite tasks, respectively.
- Both methods split the input instruction to extract the task name.
- The code checks if the task with the given name exists in the task map. If it does, it retrieves the task and prints relevant information such as task name, description, and, for composite tasks, subtasks.
- The printout is formatted to be easy to read, providing clear information about the task.

3) Error Conditions and How They Are Handled:

- **Task Not Found:**
  - If the task to be printed is not found in the task map, both methods return a "Task not found" message.
- **Invalid PrintTask Command Format:**
  - If the PrintTask command does not follow the expected format, an "Invalid PrintTask command format" message is printed.

[REQ 6] PrintAllTasks

1) The requirement is implemented.

```java
/**
 * Prints information about all the tasks stored in the task map.
 * If no tasks are in map, a message indicating it will be displayed.
 */

2 usages
public static void printAllTasks() {
    if (taskMap.isEmpty()) {
        System.out.println("No tasks available.");
    } else {
        for (TMS task : taskMap.values()) {
            if (task.isPrimitive(task.getName(), taskMap)) {
                PrimitiveTask primitiveTask = (PrimitiveTask) task;
                System.out.println("Task Name: " + primitiveTask.getName());
                System.out.println("Description: " + primitiveTask.getDescription());
                System.out.println("Duration: " + primitiveTask.getDuration());
                System.out.println();
            } else if (!task.isPrimitive(task.getName(), taskMap)) {
                CompositeTask compositeTask = (CompositeTask) task;
                System.out.println("Task Name: " + compositeTask.getName());
                System.out.println("Description: " + compositeTask.getDescription());
                System.out.println("Subtasks: " + String.join( delimiter: ", ", compositeTask.getPrereq
                System.out.println();
            }
        }
    }
}
```

2) Implementation Details:

- The printAllTasks method is provided to print information about all tasks in the task map.
- The code first checks if the task map is empty. If it is, it prints a message indicating that there are no tasks available.

- If there are tasks, it iterates through the task map and prints information about each task.
- For each task, it checks if it is a primitive task using the isPrimitive method, and based on the result, it prints relevant information such as task name, description, duration (for primitive tasks), and subtasks (for composite tasks).
- The printout is formatted to be easy to read, providing clear information about each task.

3) Error Conditions and How They Are Handled:

- **No Tasks Available:**
  - If the task map is empty, the code prints a message: "No tasks available."
- **Task Type Identification:**
  - The code uses the isPrimitive method to identify whether a task is primitive or composite, ensuring the correct printing of information based on the task type.

[REQ 7] ReportDuration

1) The requirement is implemented.

```java
/**Method to report the duration of a Composite task
 * This method expects an instruction string and a Map that stores tasks
 * It will calculate the time required to finish a task
 *
 * @param taskName contains string representation of entire user input
 * @param taskMap contains a map that stores the user information
 * @return Double representation of duration that reports the duration of the task*/
6 usages
@Override
public double reportDuration(String taskName, Map<String, TMS> taskMap) {
    if (taskMap.containsKey(taskName)) {
        TMS task = taskMap.get(taskName);
        if (isPrimitive(taskName, taskMap)) {
            return ((PrimitiveTask) task).getDuration();
        } else if (!isPrimitive(taskName, taskMap)) {
            double duration = 0;
            for (String subtaskName : task.getPrerequisites()) {
                TMS subtask = taskMap.get(subtaskName);
                duration += reportDuration(subtaskName, taskMap);
            }
            duration += task.getDuration();
            return duration;
        }
    }
    return 0;
}
```

2) Implementation Details:

- Two reportDuration methods are provided, each catering to the reporting of the duration for simple primitive tasks and composite tasks, respectively.
- Both methods take the task name and the task map as parameters.
- If the task is primitive, the code directly retrieves and returns the duration of the primitive task.
- If the task is composite, the code iterates through its subtasks, recursively calls reportDuration for each subtask, and accumulates their durations. The duration of the composite task is the sum of the durations of its subtasks plus its own duration.

3) Error Conditions and How They Are Handled:

- **Task Not Found:**
    - If the task to report duration for is not found in the task map, both methods return 0, indicating that the task was not found.

- **Task Type Identification:**
    - The code uses the isPrimitive method to identify whether a task is primitive or composite, ensuring the correct reporting of duration based on the task type.
- **Invalid Task Duration:**
    - If a task has a negative duration or if there are issues with the task duration format, the code handles this gracefully by returning 0.

[REQ 8] ReportEarliestFinishTime

1) The requirement is implemented.

```java
/**Defined Function: Method to calculate and report the earliest finish time for a given task
 *                  in a map of tasks.
 * Common function - for Primitive and Composite tasks.
 *
 * @param taskName contains string representation of task name
 * @param taskMap contains a map that stores the user information
 * @return Double representation of earliest finish time for given tasks*/
3 usages
public double reportEarliestFinishTime(String taskName, Map<String, TMS> taskMap) {
    double earliestFinishTime = 0.0;
    TMS task = taskMap.get(taskName);

    if (!isPrimitive(taskName, taskMap)) {
        CompositeTask compositeTask = (CompositeTask) task;
        for (String subtaskName : compositeTask.getPrerequisites()) {
            double subtaskFinishTime = compositeTask.reportEarliestFinishTime(subtaskName, taskMap);
            earliestFinishTime = Math.max(earliestFinishTime, subtaskFinishTime);
        }
    } else if (isPrimitive(taskName, taskMap)) {
        earliestFinishTime = task.getDuration();
    }

    earliestFinishTime += task.getDuration();
    return earliestFinishTime;
}}
```

2) Implementation Details:

- The reportEarliestFinishTime method is provided to calculate and report the earliest finish time for a given task.
- The code initializes earliestFinishTime to 0.0 and retrieves the task from the task map.

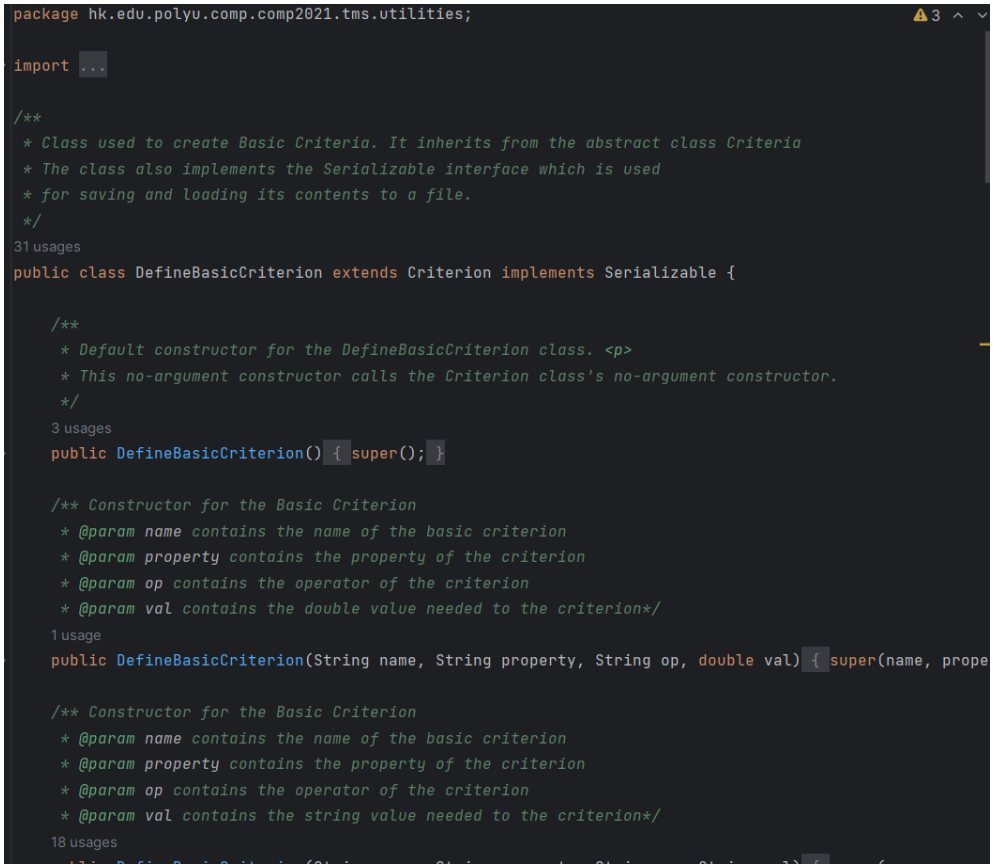
- If the task is composite, the code iterates through its prerequisites (subtasks), recursively calls reportEarliestFinishTime for each subtask, and updates the earliestFinishTime based on the maximum finish time of its prerequisites.
- If the task is primitive, the code sets the earliestFinishTime to its own duration.
- The earliestFinishTime is then updated by adding the duration of the task itself.

3) Error Conditions and How They Are Handled:

- **Task Not Found:**
  - If the task to report the earliest finish time for is not found in the task map, the method returns 0, indicating that the task was not found.
- **Task Type Identification:**
  - The code uses the isPrimitive method to identify whether a task is primitive or composite, ensuring the correct calculation of the earliest finish time based on the task type.
- **Invalid Task Duration:**
  - If a task has a negative duration or if there are issues with the task duration format, the code handles this gracefully by returning 0.

[REQ 9] DefineBasicCriterion

1) The requirement is implemented.



```java
package hk.edu.polyu.comp.comp2021.tms.utilities;

import ...

/**
 * Class used to create Basic Criteria. It inherits from the abstract class Criteria
 * The class also implements the Serializable interface which is used
 * for saving and loading its contents to a file.
 */
31 usages
public class DefineBasicCriterion extends Criterion implements Serializable {

    /**
     * Default constructor for the DefineBasicCriterion class. <p>
     * This no-argument constructor calls the Criterion class's no-argument constructor.
     */
    3 usages
    public DefineBasicCriterion() { super(); }

    /** Constructor for the Basic Criterion
     * @param name contains the name of the basic criterion
     * @param property contains the property of the criterion
     * @param op contains the operator of the criterion
     * @param val contains the double value needed to the criterion*/
    1 usage
    public DefineBasicCriterion(String name, String property, String op, double val) { super(name, prope

    /** Constructor for the Basic Criterion
     * @param name contains the name of the basic criterion
     * @param property contains the property of the criterion
     * @param op contains the operator of the criterion
     * @param val contains the string value needed to the criterion*/
    18 usages
```

2) Implementation Details:

- The create method is provided to handle the DefineBasicCriterion command.
- The method splits the input instruction to extract name, property, operator (op), and value.
- It checks the validity of the command format, ensuring that it has the correct number of tokens.
- It checks if a criterion with the same name already exists and handles the error condition by printing an appropriate message.
- It validates the name format using the isName method.
- Based on the property, it constructs a new DefineBasicCriterion object with the specified name, property, operator, and value.
- The method handles different property types ("name," "description," "duration," "prerequisites") and their respective formats.
- The constructed criterion is then added to the criterion map.

3) Error Conditions and How They Are Handled:

- **Invalid Command Format:**
    - If the DefineBasicCriterion command does not follow the expected format, an "Invalid DefineBasicCriterion command format" message is printed.
- **Criterion with the Same Name Exists:**
    - If a criterion with the same name already exists in the criterion map, the code prints a message: "Task with the same name already exists: name."
- **Invalid Name Format:**
    - If the name does not follow the specified format, the code prints an "Invalid CreateSimpleTask name format" message.
- **Invalid Duration Operator:**
    - If the property is "duration" and the operator is not one of >, <, >=, <=, ==, or !=, the code prints an "Invalid duration op: op" message.
- **Invalid Duration Value:**
    - If the property is "duration" and the value cannot be parsed to a Double, the code prints an "Invalid duration value: value" message.
- Invalid Prerequisites Format:
    - If the property is "prerequisites" and the operator is not "contains," the code prints an "Invalid prerequisites op: op" message.

[REQ 10] isPrimitive

1) The requirement is implemented.

```java
/** Common function - For all types of tasks
 * Defined Function: returns true if the current task is Primitive
 *                   It includes both Primitive and Composite Tasks.
 *
 * @param name contains a string representation of task name
 * @param taskMap contains a map that stores user's tasks
 * @return boolean if the task is a primitive task
 */
10 usages
public boolean isPrimitive (String name, Map<String, TMS> taskMap){
    //String [] tokens = instruction.split (" ");
    if (taskMap.containsKey(name)){
        try{
            if(taskMap.get(name).getPrerequisites() == null) return true;
        }
        catch (Exception e) {e.printStackTrace();}
    }
    return false;
}
```

2) Implementation Details:

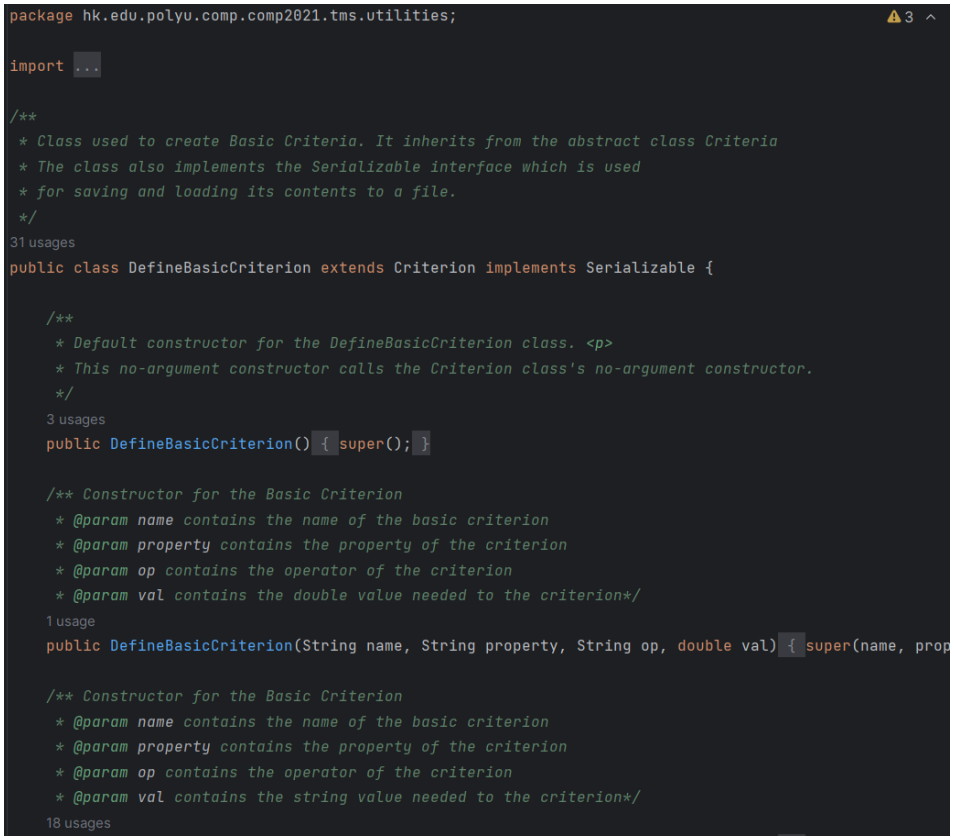
- The isPrimitive method is provided to evaluate the IsPrimitive criterion.
- The method takes the task name and the task map as parameters.
- It checks if the task map contains the given task name.
- If the task is found, it attempts to retrieve the prerequisites of the task. If the task is primitive, its prerequisites should be null.
- If the prerequisites are null, the method returns true, indicating that the task is primitive. Otherwise, it returns false.

3) **Error Conditions and How They Are Handled:**

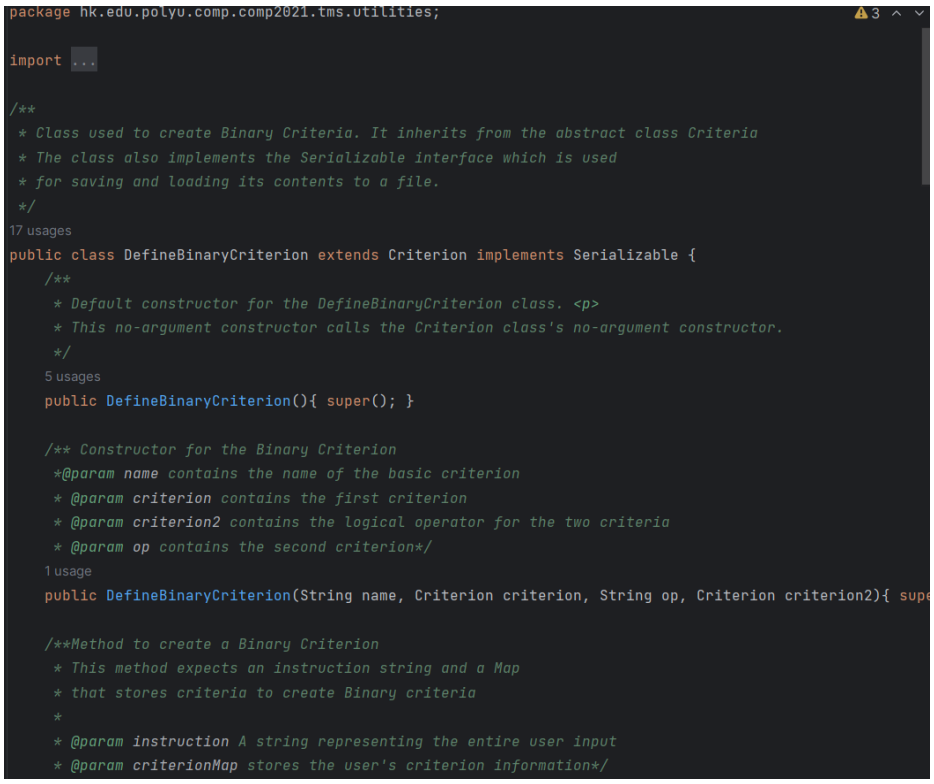
- **Task Not Found**:
  - If the task to check for primitiveness is not found in the task map, the method returns false.
- **Exception Handling:**
  - The code uses a try-catch block to catch any exceptions that might occur while accessing the prerequisites. If an exception occurs, it prints a stack trace but does not affect the result.

[REQ11] DefineNegatedCriteria and DefineBinaryCriterion

1) The requirement is implemented.



```java
package hk.edu.polyu.comp.comp2021.tms.utilities;

import ...

/**
 * Class used to create Basic Criteria. It inherits from the abstract class Criteria
 * The class also implements the Serializable interface which is used
 * for saving and loading its contents to a file.
 */
31 usages
public class DefineBasicCriterion extends Criterion implements Serializable {

    /**
     * Default constructor for the DefineBasicCriterion class. <p>
     * This no-argument constructor calls the Criterion class's no-argument constructor.
     */
    3 usages
    public DefineBasicCriterion() { super(); }

    /** Constructor for the Basic Criterion
     * @param name contains the name of the basic criterion
     * @param property contains the property of the criterion
     * @param op contains the operator of the criterion
     * @param val contains the double value needed to the criterion*/
    1 usage
    public DefineBasicCriterion(String name, String property, String op, double val) { super(name, prop

    /** Constructor for the Basic Criterion
     * @param name contains the name of the basic criterion
     * @param property contains the property of the criterion
     * @param op contains the operator of the criterion
     * @param val contains the string value needed to the criterion*/
    18 usages
```



```java
package hk.edu.polyu.comp.comp2021.tms.utilities;

import ...

/**
 * Class used to create Binary Criteria. It inherits from the abstract class Criteria
 * The class also implements the Serializable interface which is used
 * for saving and loading its contents to a file.
 */
17 usages
public class DefineBinaryCriterion extends Criterion implements Serializable {
    /**
     * Default constructor for the DefineBinaryCriterion class. <p>
     * This no-argument constructor calls the Criterion class's no-argument constructor.
     */
    5 usages
    public DefineBinaryCriterion(){ super(); }

    /** Constructor for the Binary Criterion
     *@param name contains the name of the basic criterion
     * @param criterion contains the first criterion
     * @param criterion2 contains the logical operator for the two criteria
     * @param op contains the second criterion*/
    1 usage
    public DefineBinaryCriterion(String name, Criterion criterion, String op, Criterion criterion2){ supe

    /**Method to create a Binary Criterion
     * This method expects an instruction string and a Map
     * that stores criteria to create Binary criteria
     *
     * @param instruction A string representing the entire user input
     * @param criterionMap stores the user's criterion information*/
```

2) Implementation Details:

- DefineNegatedCriterion:
    - Parses the input and checks for the correct number of arguments.
    - Validates the input, ensuring the criterion name is not already present and follows the correct format.
    - Retrieves the existing criterion (name2) from the criterion map and creates a new DefineNegatedCriterion based on its properties.
    - Handles different property types (name, description, duration, prerequisites).
    - Adds the new negated criterion to the criterion map.
- DefineBinaryCriterion:
    - Parses the input and checks for the correct number of arguments.
    - Validates the input, ensuring the criterion name is not already present, follows the correct format, and the logical operator is either "&&" or "||".
    - Retrieves existing criteria (name2 and name3) from the criterion map and creates a new DefineBinaryCriterion based on their properties and the logical operator.
    - Adds the new binary criterion to the criterion map.

Error Conditions and How They Are Handled:

- DefineNegatedCriterion:
    - Checks if the specified property is valid and handles accordingly.
    - Handles invalid duration operators or values.
    - Prints error messages for invalid input formats, existing criterion names, or invalid property formats.
- DefineBinaryCriterion:
    - Checks for the validity of the logical operator (&& or ||).
    - Prints error messages for invalid input formats, existing criterion names, invalid logical operators, or missing criteria.

[REQ12] PrintAllCriteria

1) The requirement is implemented.

```java
/**
 * Prints information about all the criteria stored in the task map.
 * If no tasks are in map, a message indicating it will be displayed.
 */

1 usage
public static void printAllCriteria() {
    if (criterionMap.isEmpty()) {
        System.out.println("No criteria available.");
    } else{
        for (Criterion criteria : criterionMap.values()) {
            criteria.printCriterion(criteria.getName(), criterionMap);
        }
    }
}
```

2) Implementation Details:

- The printCriterion method takes a criterion name and a map of criteria.
- It first checks if the criterion map is empty, and if so, it prints a message indicating that there are no criteria defined.
- If the provided criterion name exists in the criterion map, it retrieves the corresponding criterion.
- It then prints the following information about the criterion:
    - Criterion Name
    - Property
    - Operator
    - Value (with different handling based on whether it's a string, list, or double)
- The printed information is formatted for readability.

3) Error Conditions and How They Are Handled:

- **Empty Criterion Map:**
    - If the criterion map is empty, the code prints a message: "There are no criteria defined."

[REQ 13] Search name

1) The requirement is implemented.

```java
/**Method to search Tasks based on Basic Criteria
 * This method expects an instruction string, a Map that stores tasks,
 * and another Map that stores criteria in order to search tasks based on the given criteria
 *
 * @param instruction A string representing the entire user input
 * @param taskMap A map that stores the user's tasks, mapped by their names.
 * The task to be modified should be present in this map.
 * @param criterionMap stores the user's criterion information */
17 usages
@Override
public void search(String instruction, Map<String, TMS> taskMap, Map <String, Criterion> criterionM
    String[] tokens = instruction.split( regex: " ");
    if (tokens.length != 2) {
        System.out.println("Invalid search command format.");
        return;
    }

    String name = tokens[1];
    if (!criterionMap.containsKey(name)) {
        System.out.println("Criterion with the given name does not exist: " + name);
        return;
    }

    Criterion criterion = criterionMap.get(name);
    String property = criterion.getProperty();
    String op = criterion.getOp();
```

2) The search method takes the instruction, a map of tasks (taskMap), and a map of criteria (criterionMap).

- It first checks the validity of the command format.
- It retrieves the criterion with the given name from the criterionMap.
- Based on the type of criterion (basic or composite), it performs the search operation.
- The search is done by iterating over tasks and checking if they meet the criteria.
- Matching tasks are stored in an ArrayList, and the results are printed.
- For composite criteria with logical operators (&& or ||), it evaluates both sub-criteria.
- The output includes the tasks that match the given criterion.

3) Error Conditions and How They Are Handled:

- **Invalid Command Format:**
    - If the command format is invalid (e.g., incorrect number of tokens), it prints: "Invalid search command format."

- **Non-Existent Criterion:**
    - If the criterion with the given name does not exist in the criterionMap, it prints: "Criterion with the given name does not exist: {name}."
- **No Matching Tasks:**
    - If no tasks match the given criterion, it prints: "No tasks match the given criterion."
- **Matching Tasks:**
    - If tasks match the given criterion, it prints: "Tasks matching the given criterion:" followed by the list of matching task names.

[REQ 14] Store

1) The requirement is implemented.

```java
/**
 * Method serves to save the current state of task and criterion maps to file.
 *
 * @param instruction A string representing the entire user input
 * @param taskMap A map that stores the user's tasks, mapped by their names. The task to be modif
 * @param criterionMap stores the user's criterion information
 * @throws IOException checks if an input/output error occurs during the save operation
 */
3 usages  Sid-005
public static void saveMap(String instruction, Map<String, TMS> taskMap, Map<String, Criterion> c
    String[] inputArray = instruction.split( regex: " ");
    if (inputArray.length == 2) {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(inputArray[1]))
            oos.writeObject(taskMap);
            oos.writeObject(criterionMap);
            System.out.println("Files were updated successfully");
        } catch (FileNotFoundException e) {
            System.out.println("The file path was not found. Try again");
        } catch (NotSerializableException e) {
            System.out.println("There is an implementation error.");
        }
    } else {
        System.out.println("Invalid format for Store function");
    }
}
```

2) Implementation Details:

- The saveMap method takes the instruction, a map of tasks (taskMap), and a map of criteria (criterionMap).
- It splits the input to get the file path.
- It uses ObjectOutputStream to write the taskMap and criterionMap objects into the specified file.
- It catches and handles exceptions such as FileNotFoundException and NotSerializableException.
- If the file is successfully written, it prints: "Files were updated successfully."
- The code uses ObjectOutputStream for serialization, which is a suitable approach for saving objects to a file.
- It provides feedback to the user about the success or failure of the store operation.

3) Error Conditions and How They Are Handled:

- **Invalid Command Format:**
  - If the command format is invalid (e.g., incorrect number of tokens), it prints: "Invalid format for Store function."
- **File Not Found:**
  - If the file path is not found, it catches FileNotFoundException **and prints: "The file path was not found. Try again."**
- **Serialization Issue:**

If there is an issue with serialization (e.g., NotSerializableException**), it prints: "There is an implementation error."**

[REQ 15] Load

1) The requirement is implemented.

```java
/**
 * Methods serves to load task and criterion maps from a specified file.
 *
 * @param instruction A string representing the entire user input
 * @param taskMap A map that stores the user's tasks, mapped by their names. The task to be modifi
 * @param criterionMap stores the user's criterion information
 * @throws IOException checks if an input/output error occurs during the save operation
 * @throws ClassNotFoundException checks if the class of a serialized object can not be found duri
 */
6 usages  Sid-005
public static void loadMap(String instruction, Map<String, TMS> taskMap, Map<String, Criterion> cr
    String[] inputArray = instruction.split( regex: " ");
    if (inputArray.length == 2) {
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(inputArray[1]))) {
            taskMap.clear();
            criterionMap.clear();
            taskMap.putAll((Map<String, TMS>) ois.readObject());
            criterionMap.putAll((Map<String, Criterion>) ois.readObject());
            System.out.println("All lines within file have been read.");
        } catch (FileNotFoundException e) {
            throw new FileNotFoundException("Specified directory is not found or cannot be accesse
        } catch (EOFException e) {
            throw new EOFException("End of file reached.");
        }
    } else {
        throw new IllegalArgumentException("Invalid Syntax for Load");
    }
}
```

2) Implementation Details:

- The loadMap method takes the instruction, a map of tasks (taskMap), and a map of criteria (criterionMap).
- It splits the input to get the file path.
- It uses ObjectInputStream to read the taskMap and criterionMap objects from the specified file.
- It clears the existing taskMap and criterionMap before populating them with the loaded objects.
- It catches and handles exceptions such as FileNotFoundException and EOFException.
- If the file is successfully read, it prints: "All lines within the file have been read."
- The code effectively clears the existing maps before loading new tasks and criteria from the file.
- It provides specific exception messages to help diagnose issues during loading.

3) Error Conditions and How They Are Handled:

- **Invalid Command Syntax:**
  - If the command syntax is invalid (e.g., incorrect number of tokens), it throws an IllegalArgumentException with the message: "Invalid Syntax for Load."
- **File Not Found:**
  - If the file path is not found, it throws a FileNotFoundException with the message: "Specified directory is not found or cannot be accessed."
- **End of File Reached (EOFException):**
  - If the end of the file is reached unexpectedly (e.g., if the file is corrupted), it throws an EOFException with the message: "End of file reached."

[REQ16] Quit

1) The requirement is implemented.

```
else if (input.equalsIgnoreCase( anotherString: "Quit")) {
    System.out.println("Thank You for using the System.");
    System.out.println ("All unsaved changes will be discarded.");
    System.out.println ("Would you like to proceed? (Y?N)");
    String choice = scanner.nextLine();
    if (choice.equalsIgnoreCase( anotherString: "Y")){
        System.out.println("Shutting Down...");
        break;
    }
    System.out.println ("Shut down process halted.");
```

2) Implementation Details:

- When the user enters "Quit" (case-insensitive), the system prompts the user with a message:
  - "Thank You for using the System."
  - "All unsaved changes will be discarded."

    - "Would you like to proceed? (Y/N)"
- The system reads the user's choice.
- If the choice is "Y" (case-insensitive), the system prints "Shutting Down..." and breaks out of the loop, effectively terminating the execution.
- If the choice is not "Y", the system prints "Shutdown process halted."
- The implementation involves user interaction to confirm the shutdown, providing a level of safety to prevent accidental termination.

3) Error Conditions and How They Are Handled:

There are no specific error conditions associated with the Quit command. The user's input is checked for "Y" or other values, and appropriate messages are printed based on the choice.