**Exercise 2: Problem Statements for Mini-projects**

1. Astronaut Daily Schedule Organize Programming Exercise

**Problem Statement**:Design and implement a console-based application that helps astronauts organize their daily schedules. The application should allow users to add, remove, and view daily tasks. Each task will have a description, start time, end time, and priority level. The intent behind this problem statement is to evaluate your ability to implement a basic CRUD (Create, Read, Update, Delete) application, manage data efficiently, and apply best coding practices.

**ALGORITHM:**

1. Initialize Schedule Manager:

- Create singleton instance of ScheduleManager using `ScheduleManager.getInstance()`.

2. Initialize Scanner:

- Create a Scanner object for user input.

3. Show Main Menu:

- Display options: Add Task, Remove Task, View Tasks, View Tasks by Priority, Exit.

- Prompt user for choice.

4. Process User Choice:

- Read user input choice.

- Use switch statement to handle menu options.

5. Add Task:

- Prompt for task details: description, start time, end time, priority.

- Create Task using `TaskFactory.createTask()`.

- Add task to schedule with `ScheduleManager.addTask()`.

- Handle exceptions for invalid input.

6. Remove Task:

- Prompt for task description to remove.

- Remove task using `ScheduleManager.removeTask()`.

7. View Tasks:

- Display all tasks using `ScheduleManager.viewTasks()`.

8. View Tasks by Priority:

- Prompt for priority.

- Display tasks by priority using `ScheduleManager.viewTasksByPriority()`.

9. Exit:

- Exit the program.

10. Invalid Input Handling:

- Handle invalid menu choices, prompt user again.

11. Recursive Menu Call:

- Recursively call `showMenu` after handling a choice.

12. Main Method:

- Instantiate `ScheduleManager`.

- Instantiate `Scanner`.

- Call `showMenu`.

13. showMenu Method:

- Print menu options.

- Try to read user choice.

- Handle menu options with a switch case.

- Handle exceptions.

- Recursively call `showMenu`.

14.Task Class:

- Attributes: description, startTime, endTime, priority.

- Constructor to initialize attributes.

- Method to check for task overlap.

- `toString` method for task representation.

15. TaskFactory Class:

- Static method to create Task object.

16. ScheduleManager Class:

- Singleton instance.

- List of tasks.

- Methods: `addTask`, `removeTask`, `viewTasks`, `viewTasksByPriority`.

17. Logger Configuration:

- Configure logger for `ScheduleManager`.

18. Add Task Details Handling:

- Ensure task details are valid and handle invalid formats.

19. View Tasks Filtering:

- Implement filtering logic in `viewTasksByPriority`.

20. Exit Handling:

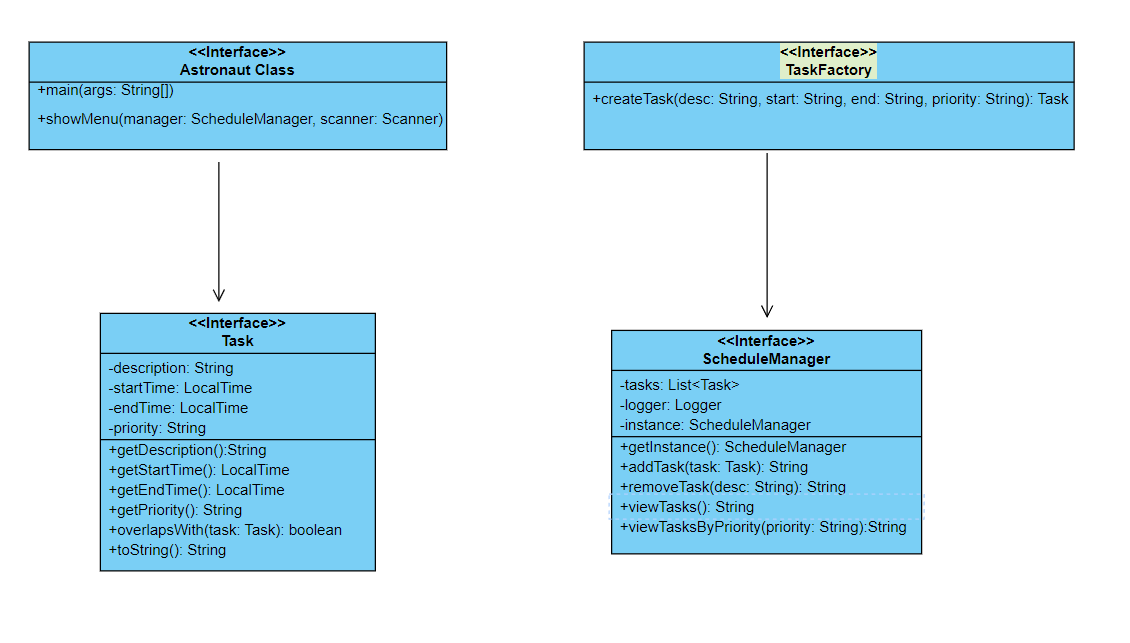
- Gracefully terminate the program.

**Design:**

**1. Overview**

The AstronautTest program is a console-based task scheduling application that allows users to add, remove, and view tasks, as well as filter tasks by priority. The application is designed using a combination of singleton and factory design patterns.

**2.Class Diagram**



**3. Class Descriptions**

1. **AstronautTest**
   * The main entry point of the program.
   * Contains the main method to initiate the program.
   * Provides a showMenu method to display the menu options and handle user input.
2. **Task**
   * Represents a task with description, start time, end time, and priority.
   * Provides methods to get attributes, check for overlapping tasks, and a toString method for string representation.
3. **TaskFactory**
   * A factory class to create Task objects.
   * Contains a createTask method to instantiate a Task object.
4. **ScheduleManager**
   * Manages a list of tasks.
   * Singleton pattern ensures only one instance of ScheduleManager exists.
   * Provides methods to add, remove, view all tasks, and view tasks by priority.
   * Logs actions using a Logger.

**4. Design Patterns**

1. **Singleton Pattern**
   * Used in ScheduleManager to ensure a single instance manages the task list.
   * getInstance method provides the single instance.
2. **Factory Pattern**
   * Used in TaskFactory to encapsulate the creation of Task objects.
   * Simplifies task creation and handles parsing and validation internally.

**5. Methods and Functionality**

1. **AstronautTest**
   * main: Initializes ScheduleManager and Scanner, calls showMenu.
   * showMenu: Displays menu, processes user input, calls appropriate methods, and handles exceptions.
2. **Task**
   * Constructor: Initializes task attributes and checks for valid start and end times.
   * getDescription, getStartTime, getEndTime, getPriority: Getter methods.
   * overlapsWith: Checks if the task overlaps with another task.
   * toString: Provides string representation of a task.
3. **TaskFactory**
   * createTask: Creates and returns a Task object.
4. **ScheduleManager**
   * getInstance: Returns the singleton instance.
   * addTask: Adds a task to the schedule, checks for conflicts, and logs the action.
   * removeTask: Removes a task by description and logs the action.
   * viewTasks: Returns a string of all tasks.
   * viewTasksByPriority: Returns a string of tasks filtered by priority.

**6. Exception Handling**

* Handles invalid time formats and illegal arguments during task creation.
* Catches invalid input and prompts the user again.
* Ensures the program continues to show the menu after handling a choice.

**7. Logging**

* Logs actions such as adding and removing tasks.
* Uses Logger to record info, warning, and severe messages.

**CODING:**

package astronaut;

import java.time.LocalTime;

import java.time.format.DateTimeParseException;

import java.util.Scanner;

import java.util.ArrayList;

import java.util.Comparator;

import java.util.List;

import java.util.logging.Level;

import java.util.logging.Logger;

public class AstronautTest {

public static void main(String[] args) {

ScheduleManager manager = ScheduleManager.getInstance();

Scanner scanner = new Scanner(System.in);

showMenu(manager, scanner);

scanner.close();

}

public static void showMenu(ScheduleManager manager, Scanner scanner) {

System.out.println("\n1. Add Task\n2. Remove Task\n3. View Tasks\n4. View Tasks by Priority\n5. Exit");

System.out.print("Enter choice: ");

try {

int choice = scanner.nextInt();

scanner.nextLine();

switch (choice) {

case 1:

System.out.print("Enter task description: ");

String description = scanner.nextLine();

System.out.print("Enter start time (HH:MM): ");

String startTime = scanner.nextLine();

System.out.print("Enter end time (HH:MM): ");

String endTime = scanner.nextLine();

System.out.print("Enter priority (Low, Medium, High): ");

String priority = scanner.nextLine();

try {

Task task = TaskFactory.createTask(description, startTime, endTime, priority);

System.out.println(manager.addTask(task));

}

catch (DateTimeParseException e) {

System.out.println("Error: Invalid time format. Please use HH:MM.");

}

catch (IllegalArgumentException e) {

System.out.println("Error: " + e.getMessage());

}

break;

case 2:

System.out.print("Enter task description to remove: ");

String removeDescription = scanner.nextLine();

System.out.println(manager.removeTask(removeDescription));

break;

case 3:

System.out.println(manager.viewTasks());

break;

case 4:

System.out.print("Enter priority to filter (Low, Medium, High): ");

String filterPriority = scanner.nextLine();

System.out.println(manager.viewTasksByPriority(filterPriority));

break;

case 5:

System.out.println("Exiting...");

return;

default:

System.out.println("Invalid choice. Please enter a number between 1 and 5.");

break;

}

}

catch (Exception e) {

System.out.println("Error: Invalid input. Please enter a valid number.");

scanner.nextLine();

}

// Recursively call showMenu to show the menu again

showMenu(manager, scanner);

}

}

class Task {

private String description;

private LocalTime startTime;

private LocalTime endTime;

private String priority;

public Task(String description, String startTime, String endTime, String priority) {

this.description = description;

this.startTime = LocalTime.parse(startTime);

this.endTime = LocalTime.parse(endTime);

this.priority = priority;

if (this.startTime.isAfter(this.endTime)) {

throw new IllegalArgumentException("Start time must be before end time.");

}

}

public String getDescription() {

return description;

}

public LocalTime getStartTime() {

return startTime;

}

public LocalTime getEndTime() {

return endTime;

}

public String getPriority() {

return priority;

}

public boolean overlapsWith(Task other) {

return !(this.endTime.isBefore(other.startTime) ||

this.startTime.isAfter(other.endTime));

}

@Override

public String toString() {

return String.format("%s - %s: %s [%s]", startTime, endTime, description, priority);

}

}

class TaskFactory {

public static Task createTask(String description, String startTime, String endTime,String priority){

return new Task(description, startTime, endTime, priority);

}

}

class ScheduleManager {

private static ScheduleManager instance;

private List<Task> tasks;

private static final Logger logger = Logger.getLogger(ScheduleManager.class.getName());

private ScheduleManager() {

tasks = new ArrayList<>();

logger.setLevel(Level.INFO);

}

public static synchronized ScheduleManager getInstance() {

if (instance == null) {

instance = new ScheduleManager();

}

return instance;

}

public String addTask(Task task) {

for (Task existingTask : tasks) {

if (task.overlapsWith(existingTask)) {

logger.warning("Task conflict: " + task);

return "Error: Task conflicts with existing tasks.";

}

}

tasks.add(task);

tasks.sort(Comparator.comparing(Task::getStartTime));

logger.info("Task added: " + task);

return "Task added successfully. No conflicts.";

}

public String removeTask(String description) {

for (Task task : tasks) {

if (task.getDescription().equals(description)) {

tasks.remove(task);

logger.info("Task removed: " + task);

return "Task removed successfully.";

}

}

logger.severe("Task not found");

return "Error: Task not found.";

}

public String viewTasks() {

if (tasks.isEmpty()) {

return "No tasks scheduled for the day.";

}

StringBuilder sb = new StringBuilder();

for (Task task : tasks) {

sb.append(task).append("\n");

}

return sb.toString().trim();

}

public String viewTasksByPriority(String priority) {

List<Task> filteredTasks = new ArrayList<>();

for (Task task : tasks) {

if (task.getPriority().equalsIgnoreCase(priority)) {

filteredTasks.add(task);

}

}

if (filteredTasks.isEmpty()) {

return "No tasks with priority " + priority + ".";

}

StringBuilder sb = new StringBuilder();

for (Task task : filteredTasks) {

sb.append(task).append("\n");

}

return sb.toString().trim();

}

}