2024-25: Coding exercises

Instructions

Please complete the following ***two*** exercises by carefully reading through the instructions.

1. Total duration for coding these two exercises is 7 days.
2. ***MUST Comply***: Coding shall strictly be done in C#, or Java, or TypeScript only.
3. ***MUST Comply***: Code should be loaded in github and ONLY the link be finally shared.
4. ***Key Decision maker***: Candidate should be able to fully walk-us through the code base and explain the decisions taken through the journey! Creativity and spontaneity will carry more weightage.
5. **Good to have**: The code files in the project shall be well organised in adherence to global best practices and standards. e.g., each class shall be in separate file, pick a naming convention and adhere to the same.
   a. The program is expected to run for a long time gathering inputs from users.
   b. No hard coding of boolean flags such as while(true){ // show menu }
   c. Code should have gold standards in - logging mechanism, exception handling mechanism, transient error handling mechanism, defensive programming, validations at all levels, highly optimised for performance.

Exercise 1: Problem Statement on Design patterns

Come up creatively with six different use cases to demonstrate your understanding of the following software design patterns by coding the same.

1. Two use cases to demonstrate two behavioural design pattern.
2. Two use cases to demonstrate two creational design pattern.
3. Two use cases to demonstrate two structural design pattern.

Exercise 2: Problem Statements for Mini-projects

1. ***MUST Comply***: No fancy looking application is required to be built as part of this exercise. It shall be a simple console/terminal based application. Focus shall ONLY be on logic and code quality as described in the points below.
2. ***MUST Comply***: Coding should be done adopting best practices - Behavioural/structural/creational design patterns, SOLID design principles, OOPs programming, language of candidates choice.
3. Candidate shall pick ONE among the EIGHT problem statements provided below, and solve.
4. Note: Please feel free to assume unknowns, and be creative in enhancing the problem statements to demonstrate your excellence!

---

1. Astronaut Daily Schedule Organizer 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.

Functional Requirements

Mandatory Requirements

1. Add a new task with description, start time, end time, and priority level.
2. Remove an existing task.
3. View all tasks sorted by start time.
4. Validate that new tasks do not overlap with existing tasks.
5. Provide appropriate error messages for invalid operations.

Optional Requirements

1. Edit an existing task.
2. Mark tasks as completed.
3. View tasks for a specific priority level.

Non-functional Requirements

1. The application should handle exceptions gracefully.
2. Ensure the application is optimized for performance.
3. Implement a logging mechanism for tracking application usage and errors.

Key Focus

Design Patterns to be used

1. **Singleton Pattern:** Ensure there is only one instance of the schedule manager.
2. **Factory Pattern:** Use a factory to create task objects.
3. **Observer Pattern:** Notify users of task conflicts or updates.

Detailed Instructions

1. Use the Singleton Pattern to create a ScheduleManager class that manages all tasks.
2. Implement a TaskFactory class to create Task objects.
3. Use the Observer Pattern to alert users if a new task conflicts with an existing one.

Possible Inputs and Corresponding Outputs

Positive Cases

1. **Input:** Add Task("Morning Exercise", "07:00", "08:00", "High") **Output:** Task added successfully. No conflicts.
2. **Input:** Add Task("Team Meeting", "09:00", "10:00", "Medium") **Output:** Task added successfully. No conflicts.
3. **Input:** View Tasks **Output:**
   a. 07:00 - 08:00: Morning Exercise [High]
   b. 09:00 - 10:00: Team Meeting [Medium]
4. **Input:** Remove Task("Morning Exercise") **Output:** Task removed successfully.
5. **Input:** Add Task("Lunch Break", "12:00", "13:00", "Low") **Output:** Task added successfully. No conflicts.

Negative Cases

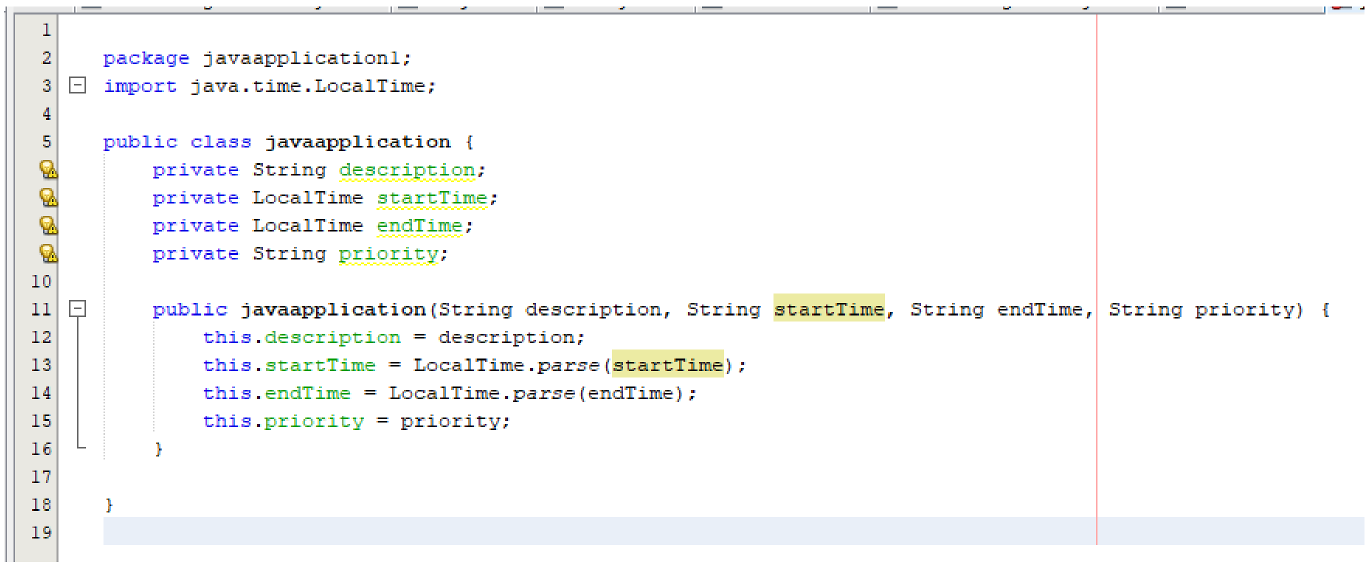
1. **Input:** Add Task("Training Session", "09:30", "10:30", "High") **Output:** Error: Task conflicts with existing task "Team Meeting".
2. **Input:** Remove Task("Non-existent Task") **Output:** Error: Task not found.
3. **Input:** Add Task("Invalid Time Task", "25:00", "26:00", "Low") **Output:** Error: Invalid time format.
4. **Input:** Add Task("Overlap Task", "08:30", "09:30", "Medium") **Output:** Error: Task conflicts with existing task "Team Meeting".

5. **Input:** View Tasks (when no tasks exist) **Output:** No tasks scheduled for the day.

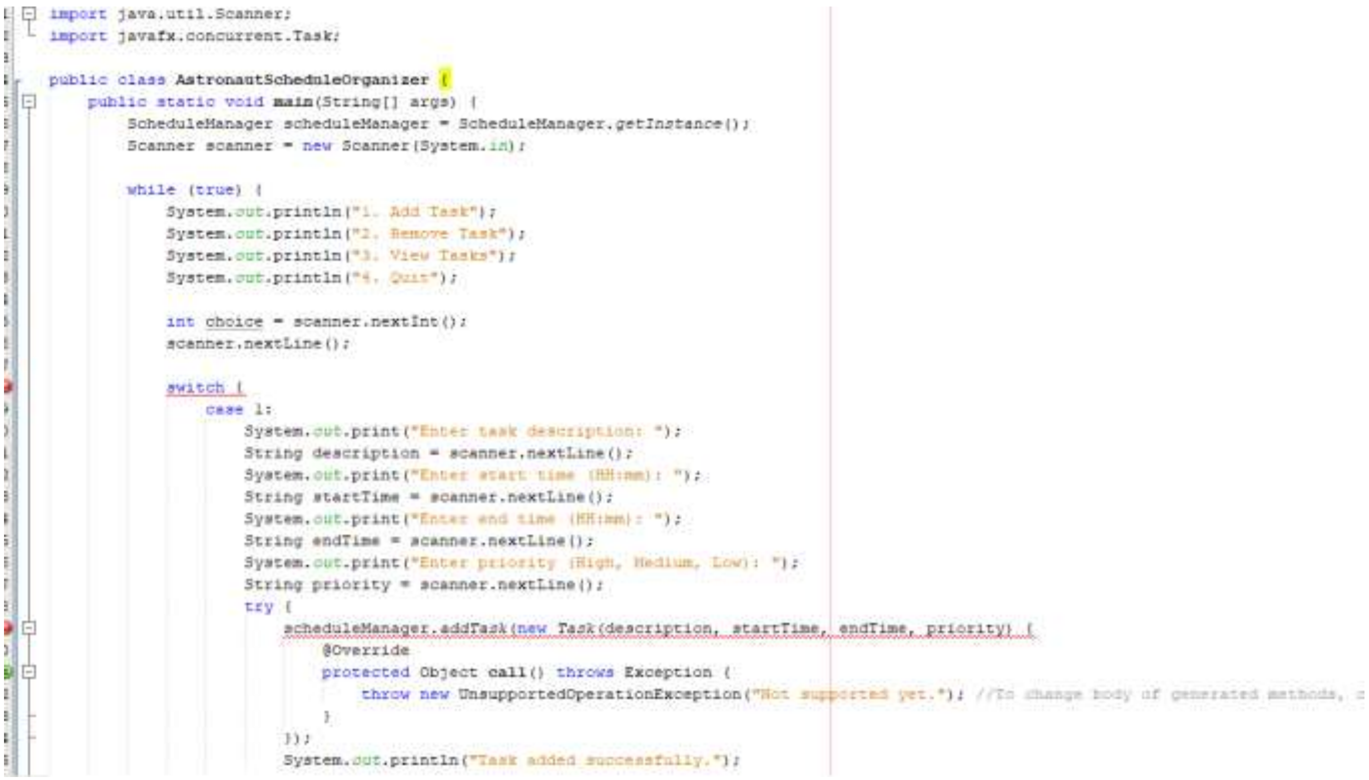
Evaluation

1. **Code Quality:** Adherence to best practices, use of design patterns, SOLID principles, and OOP.
2. **Functionality:** All mandatory requirements implemented correctly.
3. **Error Handling:** Graceful handling of all errors and edge cases.
4. **Performance:** Code is optimized for performance.
5. **Explanation:** Candidate's ability to walk through the code and explain design decisions and logic.
6. **Documentation:** Code is well-documented, and usage instructions are clear.

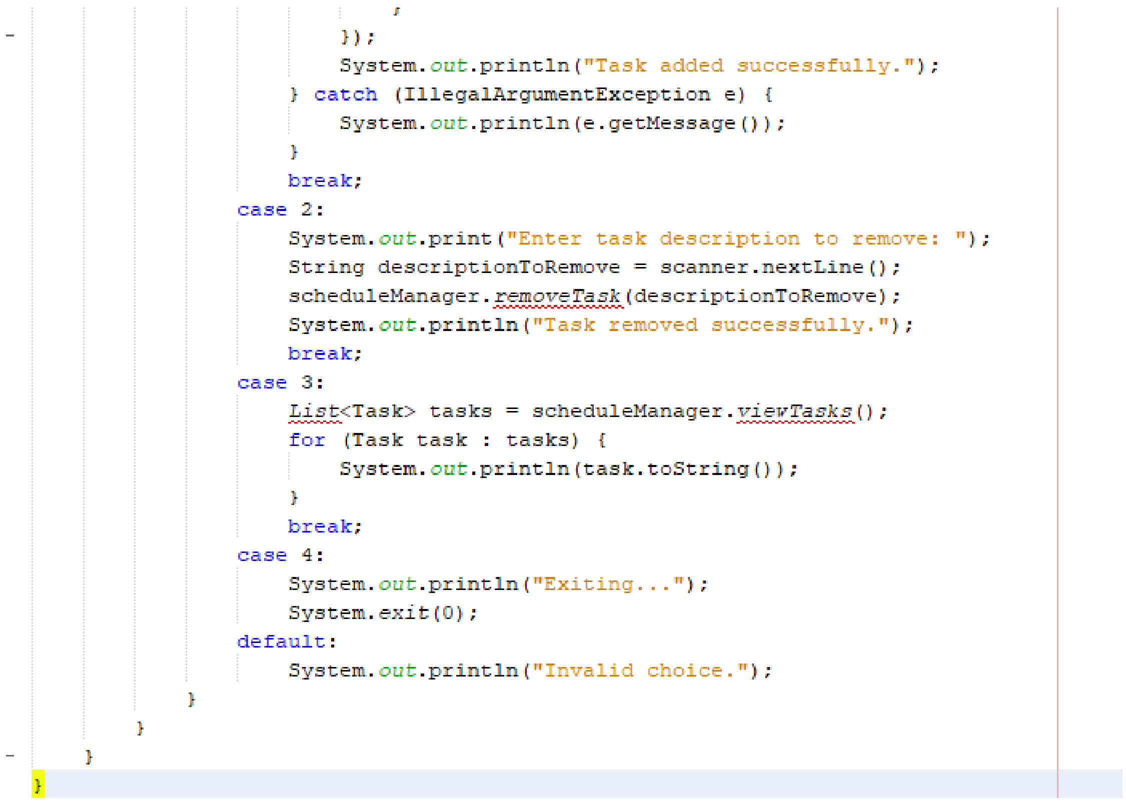
The goal of this exercise is to assess the candidate's coding skills, understanding of design patterns, and ability to produce high-quality, maintainable code.



```java

package javaapplication1;
import java.time.LocalTime;

public class javaapplication {
    private String description;
    private LocalTime startTime;
    private LocalTime endTime;
    private String priority;

    public javaapplication(String description, String startTime, String endTime, String priority) {
        this.description = description;
        this.startTime = LocalTime.parse(startTime);
        this.endTime = LocalTime.parse(endTime);
        this.priority = priority;
    }

}

```



```java
import java.util.Scanner;
import javafx.concurrent.Task;

public class AstronautScheduleOrganizer {
    public static void main(String[] args) {
        ScheduleManager scheduleManager = ScheduleManager.getInstance();
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("1. Add Task");
            System.out.println("2. Remove Task");
            System.out.println("3. View Tasks");
            System.out.println("4. Quit");

            int choice = scanner.nextInt();
            scanner.nextLine();

            switch (
                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 (High, Medium, Low): ");
                    String priority = scanner.nextLine();
                    try {
                        scheduleManager.addTask(new Task(description, startTime, endTime, priority) {
                            @Override
                            protected Object call() throws Exception {
                                throw new UnsupportedOperationException("Not supported yet."); //To change body of generated methods, c
                            }
                        });
                        System.out.println("Task added successfully.");
```



```
                        });
                        System.out.println("Task added successfully.");
                    } catch (IllegalArgumentException e) {
                        System.out.println(e.getMessage());
                    }
                    break;
                case 2:
                    System.out.print("Enter task description to remove: ");
                    String descriptionToRemove = scanner.nextLine();
                    scheduleManager.removeTask(descriptionToRemove);
                    System.out.println("Task removed successfully.");
                    break;
                case 3:
                    List<Task> tasks = scheduleManager.viewTasks();
                    for (Task task : tasks) {
                        System.out.println(task.toString());
                    }
                    break;
                case 4:
                    System.out.println("Exiting...");
                    System.exit(0);
                default:
                    System.out.println("Invalid choice.");
            }
        }
    }
}
```

1

2. Smart Office Facility Programming Exercise

Problem Statement

Design a console-based application to manage a smart office facility. The system should handle conference room bookings, occupancy detection, and automate the control of air conditioning and lighting based on room occupancy. This exercise aims to evaluate the candidate's ability to implement best coding practices, design patterns, and create an efficient, maintainable, and scalable solution.

Functional Requirements

Mandatory Requirements

1. The system should allow users to configure the office facility by specifying the number of meeting rooms.
2. Users should be able to book and cancel bookings for conference rooms.
3. Occupancy should be detected using sensors that register when at least two people enter a room.
4. Unoccupied rooms should automatically release bookings if not occupied within 5 minutes.
5. The air conditioning and lights should turn off if the room is not occupied.

Optional Requirements

1. Provide a summary of room usage statistics.
2. Implement user authentication to restrict access to booking and configuration features.
3. Notify users via email or SMS when their booked room is released automatically.

Key Focus

Design Patterns to be Used

1. **Singleton Pattern:** Ensure that the office configuration and room booking system is a single instance throughout the application.
2. **Observer Pattern:** Implement sensors and control systems (lights, AC) as observers to the room's occupancy status.

3. **Command Pattern:** Handle booking, cancellation, and room status updates through commands, allowing for flexible and extendable operations.

Instructions

1. **Singleton Pattern:** Use this pattern to manage the global state of the office configuration, ensuring that only one instance of the office configuration exists

Observer Pattern: Sensors should subscribe to occupancy changes and trigger actions like turning on/off lights and AC based on room status.

2. **Command Pattern:** Create commands for booking, cancellation, and status updates. This allows for easier addition of new features and operations.

Possible Inputs and Corresponding Outputs

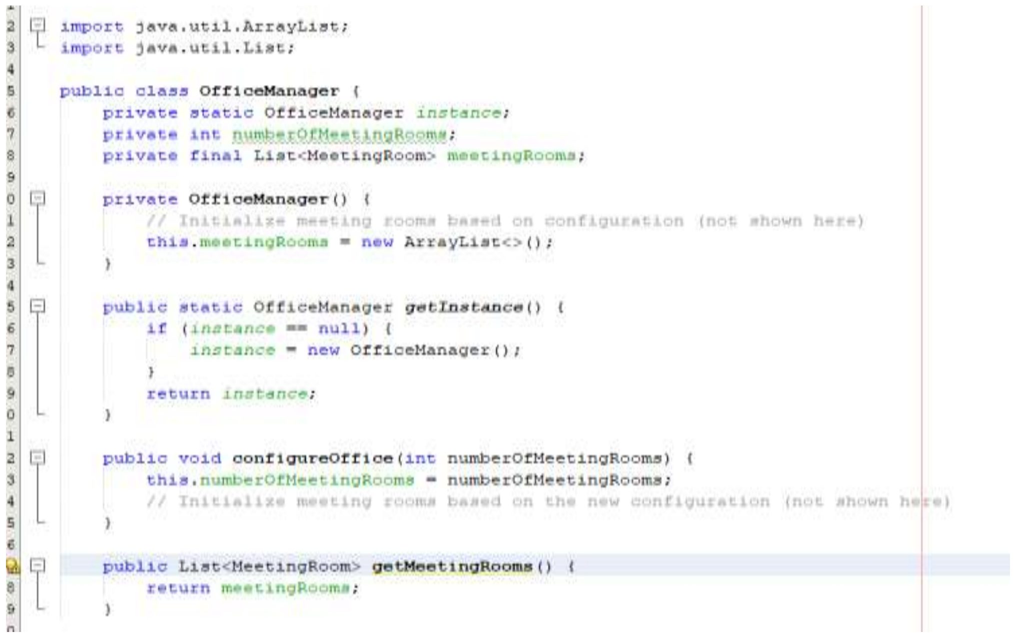
Console Input	Output
Positive Cases	
`Config room count 3`	"Office configured with 3 meeting rooms: Room 1, Room 2, Room 3."
`Config room max capacity 1 10`	"Room 1 maximum capacity set to 10."
`Add occupant 1 2`	"Room 1 is now occupied by 2 persons. AC and lights turned on."
`Block room 1 09:00 60`	"Room 1 booked from 09:00 for 60 minutes."
`Cancel room 1`	"Booking for Room 1 cancelled successfully."
`Add occupant 1 0`	"Room 1 is now unoccupied. AC and lights turned off."
Negative Cases	
`Block room 1 09:00 60` (already booked)	"Room 1 is already booked during this time. Cannot book."
`Cancel room 2` (not booked)	"Room 2 is not booked. Cannot cancel booking."
`Add occupant 2 1`	"Room 2 occupancy insufficient to mark as occupied."
`Add occupant 4 2` (non-existent room)	"Room 4 does not exist."
`Block room A 09:00 60`	"Invalid room number. Please enter a valid room number."
`Config room max capacity 1 -5`	"Invalid capacity. Please enter a valid positive number."
`Room status 1` (unoccupied for > 5 mins)	"Room 1 is now unoccupied. Booking released. AC and lights off."

Evaluation

Candidates will be evaluated on:

1. **Code Quality:** Adherence to best practices, design patterns, SOLID principles, and OOPs.
2. **Organization:** Clear and logical organization of code files, consistent naming conventions.
3. **Functionality:** Correct implementation of the functional requirements, both mandatory and optional.
4. **Efficiency:** Optimized performance, handling of edge cases, and robust error handling.
5. **Documentation:** Clear and comprehensive documentation of code and design decisions.
6. **Presentation:** Ability to walk through the code base, explaining decisions and demonstrating understanding.

This problem statement provides a balanced challenge, enabling candidates to showcase their skills in various aspects of programming and software design.



```java
import java.util.ArrayList;
import java.util.List;

public class OfficeManager {
    private static OfficeManager instance;
    private int numberOfMeetingRooms;
    private final List<MeetingRoom> meetingRooms;

    private OfficeManager() {
        // Initialize meeting rooms based on configuration (not shown here)
        this.meetingRooms = new ArrayList<>();
    }

    public static OfficeManager getInstance() {
        if (instance == null) {
            instance = new OfficeManager();
        }
        return instance;
    }

    public void configureOffice(int numberOfMeetingRooms) {
        this.numberOfMeetingRooms = numberOfMeetingRooms;
        // Initialize meeting rooms based on the new configuration (not shown here)
    }

    public List<MeetingRoom> getMeetingRooms() {
        return meetingRooms;
    }
```

3. Mars Rover Programming Exercise

Problem Statement

Create a simulation for a Mars Rover that can navigate a grid-based terrain. Your Rover should be able to move forward, turn left, and turn right. You'll need to make sure that it avoids obstacles and stays within the boundaries of the grid. Remember to use pure Object-Oriented Programming, design patterns, and avoid using if-else conditional constructs.

Functional Requirements

1. Initialize the Rover with a starting position (x, y) and direction (N, S, E, W).
2. Implement the following commands:
   - 'M' for moving one step forward in the direction the rover is facing
   - 'L' for turning left
   - 'R' for turning right
3. Implement obstacle detection. If an obstacle is detected in the path, the Rover should not move.
4. Optional: Add functionality for the Rover to send a status report containing its current position and facing direction.

Key Focus

1. Behavioral Pattern: Use the Command Pattern to encapsulate 'M', 'L', 'R' as objects for flexibility.
2. Structural Pattern: Use the Composite Pattern to represent the grid and obstacles.
3. OOP: Leverage encapsulation, inheritance, and polymorphism.

Possible Inputs

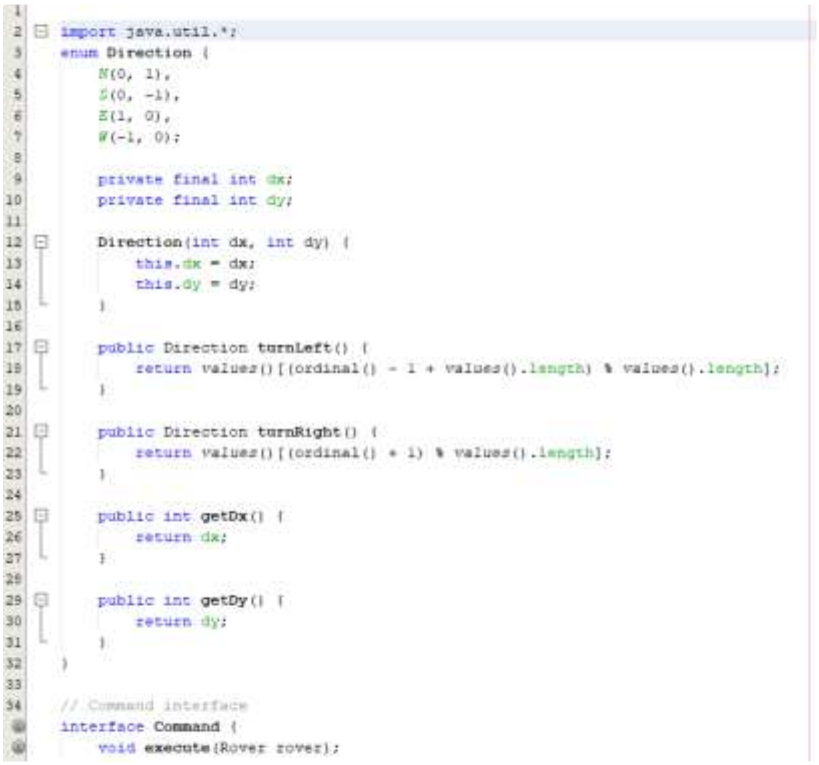
- Grid Size: (10 x 10)
- Starting Position: (0, 0, N)
- Commands: ['M', 'M', 'R', 'M', 'L', 'M']
- Obstacles: [(2, 2), (3, 5)]

Possible Outputs

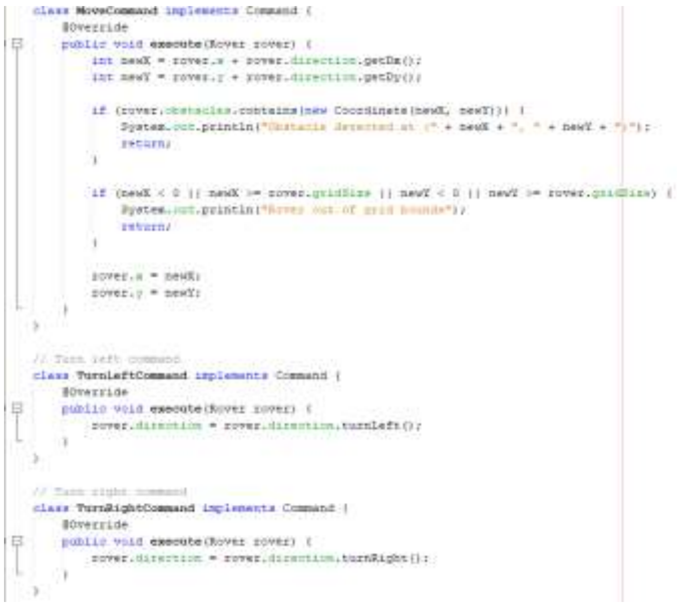
- Final Position: (1, 3, E)
- Status Report: "Rover is at (1, 3) facing East. No Obstacles detected."

Evaluation

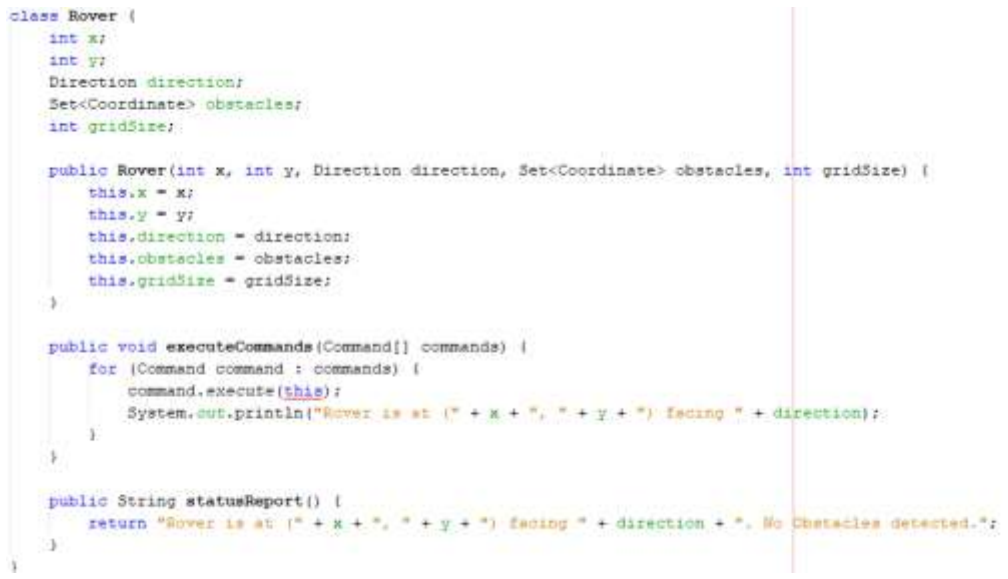
1. **Code Quality**: Are the best practices, SOLID principles, and design patterns effectively implemented?
2. **Functionality**: Does the code perform all the required tasks?
3. **Global Convention**: Is the code written in a way that is globally understandable?
4. **Gold Standards**: Does the code handle logging, exceptions, and validations effectively?
5. **Code Walkthrough**: Ability of the candidate to explain the architecture, design patterns used, and the decisions taken.



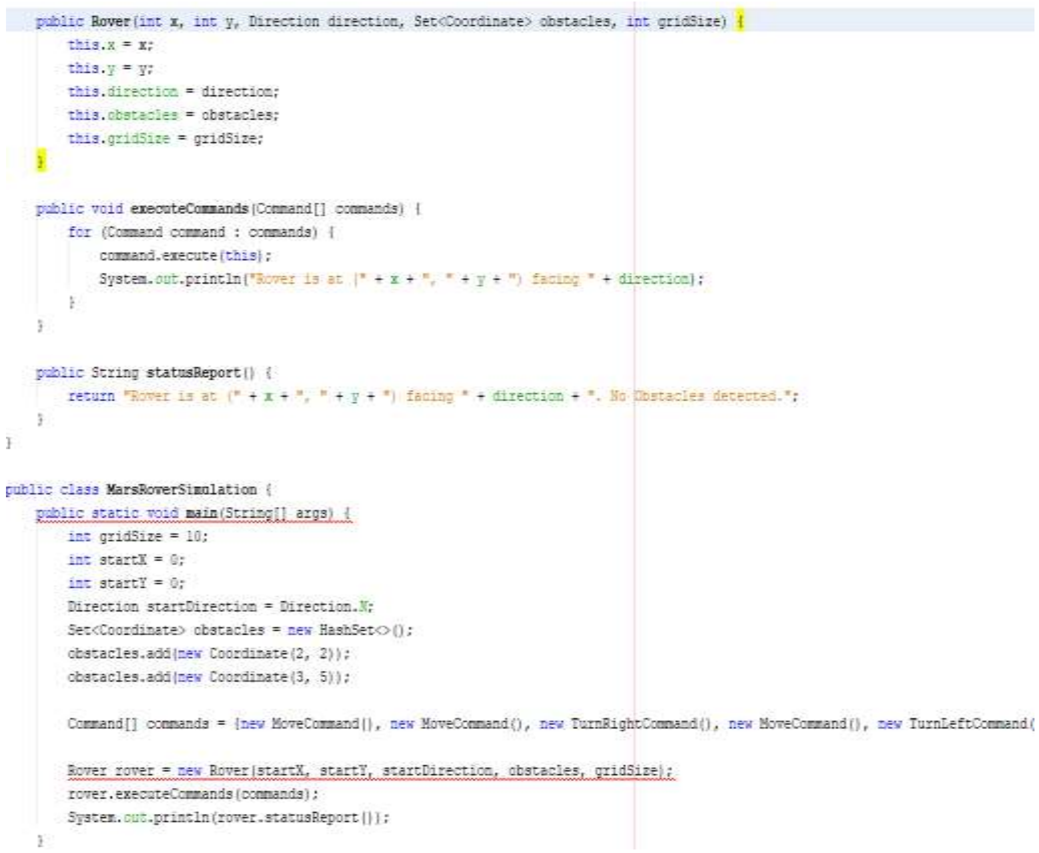
```java

import java.util.*;
enum Direction {
    N(0, 1),
    S(0, -1),
    E(1, 0),
    W(-1, 0);

    private final int dx;
    private final int dy;

    Direction(int dx, int dy) {
        this.dx = dx;
        this.dy = dy;
    }

    public Direction turnLeft() {
        return values()[(ordinal() - 1 + values().length) % values().length];
    }

    public Direction turnRight() {
        return values()[(ordinal() + 1) % values().length];
    }

    public int getDx() {
        return dx;
    }

    public int getDy() {
        return dy;
    }
}

// Command interface
interface Command {
    void execute(Rover rover);
```



```java
class MoveCommand implements Command {
    @Override
    public void execute(Rover rover) {
        int newX = rover.x + rover.direction.getDx();
        int newY = rover.y + rover.direction.getDy();

        if (rover.obstacles.contains(new Coordinate(newX, newY))) {
            System.out.println("Obstacle detected at (" + newX + ", " + newY + ")");
            return;
        }

        if (newX < 0 || newX >= rover.gridSize || newY < 0 || newY >= rover.gridSize) {
            System.out.println("Rover out of grid bounds");
            return;
        }

        rover.x = newX;
        rover.y = newY;
    }
}

// Turn left command
class TurnLeftCommand implements Command {
    @Override
    public void execute(Rover rover) {
        rover.direction = rover.direction.turnLeft();
    }
}

// Turn right command
class TurnRightCommand implements Command {
    @Override
    public void execute(Rover rover) {
        rover.direction = rover.direction.turnRight();
    }
}
```



```java
class Rover {
    int x;
    int y;
    Direction direction;
    Set<Coordinate> obstacles;
    int gridSize;

    public Rover(int x, int y, Direction direction, Set<Coordinate> obstacles, int gridSize) {
        this.x = x;
        this.y = y;
        this.direction = direction;
        this.obstacles = obstacles;
        this.gridSize = gridSize;
    }

    public void executeCommands(Command[] commands) {
        for (Command command : commands) {
            command.execute(this);
            System.out.println("Rover is at (" + x + ", " + y + ") facing " + direction);
        }
    }

    public String statusReport() {
        return "Rover is at (" + x + ", " + y + ") facing " + direction + ". No Obstacles detected.";
    }
}
```



```java
    public Rover(int x, int y, Direction direction, Set<Coordinate> obstacles, int gridSize) {
        this.x = x;
        this.y = y;
        this.direction = direction;
        this.obstacles = obstacles;
        this.gridSize = gridSize;
    }

    public void executeCommands(Command[] commands) {
        for (Command command : commands) {
            command.execute(this);
            System.out.println("Rover is at (" + x + ", " + y + ") facing " + direction);
        }
    }

    public String statusReport() {
        return "Rover is at (" + x + ", " + y + ") facing " + direction + ". No Obstacles detected.";
    }
}
```

```java
public class MarsRoverSimulation {
    public static void main(String[] args) {
        int gridSize = 10;
        int startX = 0;
        int startY = 0;
        Direction startDirection = Direction.N;
        Set<Coordinate> obstacles = new HashSet<>();
        obstacles.add(new Coordinate(2, 2));
        obstacles.add(new Coordinate(3, 5));

        Command[] commands = {new MoveCommand(), new MoveCommand(), new TurnRightCommand(), new MoveCommand(), new TurnLeftCommand(

        Rover rover = new Rover(startX, startY, startDirection, obstacles, gridSize);
        rover.executeCommands(commands);
        System.out.println(rover.statusReport());
    }
```

4. Smart Home System Programming Exercise:

Problem Statement

Create a simulation for a Smart Home System that allows the user to control different smart devices such as lights, thermostats, and door locks via a central hub. The user should be able to set schedules, automate tasks, and view the status of each device.

Functional Requirements

1. Initialize the Smart Home System with different devices, each having their own unique ID and type (light, thermostat, door lock).
2. Implement features to:
   - Turn devices on/off
   - Schedule devices to turn on/off at a particular time
   - Automate tasks based on triggers (e.g., turning off lights when the thermostat reaches a certain temperature)
3. Optional: Provide the ability to add or remove devices dynamically.

Key Focus

1. Behavioral Pattern: Use the Observer Pattern to update all devices when a change occurs in the system.
2. Creational Pattern: Use the Factory Method for creating instances of different smart devices.
3. Structural Pattern: Use the Proxy Pattern to control access to the devices.
4. OOP: Ensure strong encapsulation, modularity, and the application of inheritance and polymorphism.

Possible Inputs

- Devices: `[{id: 1, type: 'light', status: 'off'}, {id: 2, type: 'thermostat', temperature: 70}, {id: 3, type: 'door', status: 'locked'}]`
- Commands: `['turnOn(1)', 'setSchedule(2, "06:00", "Turn On")', 'addTrigger("temperature", ">", 75, "turnOff(1)")']`

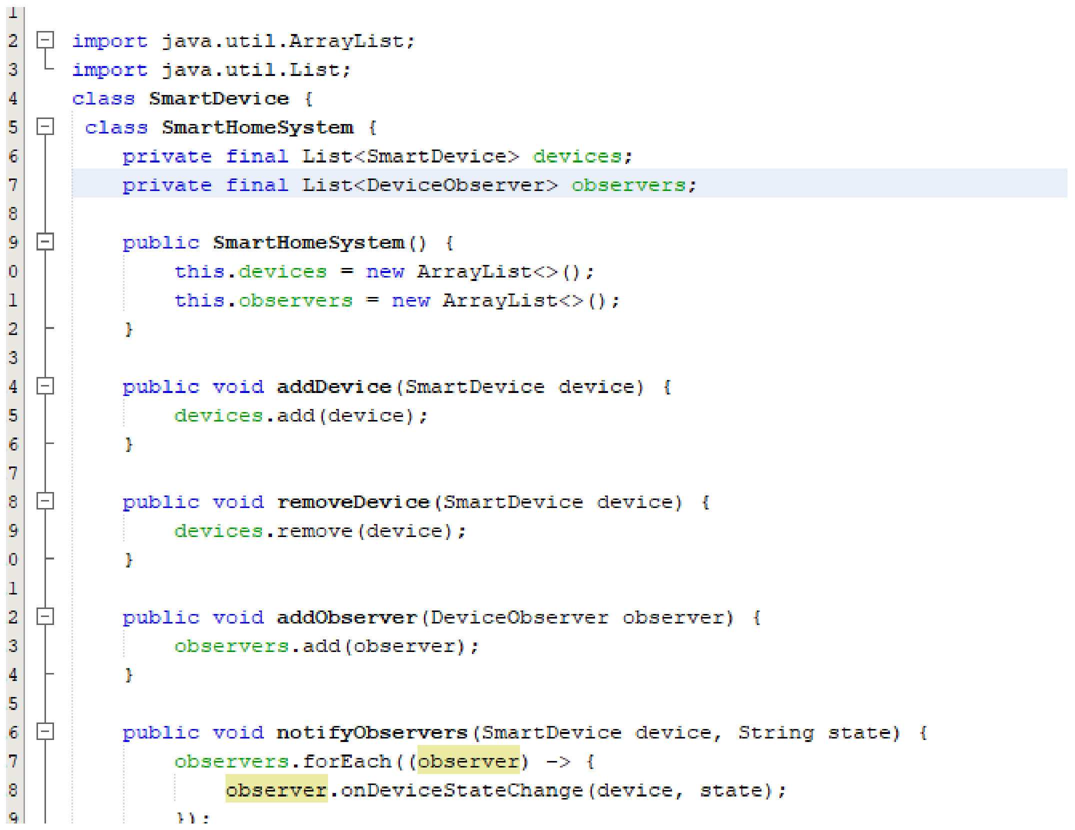
Possible Outputs

- Status Report: `"Light 1 is On. Thermostat is set to 70 degrees. Door is Locked."`
- Scheduled Tasks: `"[{device: 2, time: "06:00", command: "Turn On"}]"`
- Automated Triggers: `"[{condition: "temperature > 75", action: "turnOff(1)"}]"`

Evaluation

1. **Code Quality**: Evaluation criteria remain consistent with best practices, SOLID principles, and effective use of design patterns.
2. **Functionality**: Does the solution meet all the requirements and handle edge cases gracefully?
3. **Global Convention**: Is the code globally understandable and well-documented?
4. **Gold Standards**: Is the code up to the gold standard in terms of logging, error handling, and performance optimization?
5. **Code Walkthrough**: Can the candidate explain their solution coherently, focusing on the architecture, design decisions, and patterns used?

Intent of this Smart Home System exercise is to bring together elements of real-world applicability and technical depth...



```java
import java.util.ArrayList;
import java.util.List;
class SmartDevice {
 class SmartHomeSystem {
    private final List<SmartDevice> devices;
    private final List<DeviceObserver> observers;

    public SmartHomeSystem() {
        this.devices = new ArrayList<>();
        this.observers = new ArrayList<>();
    }

    public void addDevice(SmartDevice device) {
        devices.add(device);
    }

    public void removeDevice(SmartDevice device) {
        devices.remove(device);
    }

    public void addObserver(DeviceObserver observer) {
        observers.add(observer);
    }

    public void notifyObservers(SmartDevice device, String state) {
        observers.forEach((observer) -> {
            observer.onDeviceStateChange(device, state);
        });
```

5. Real-time Chat Application Programming Exercise

Problem Statement

Create a simple real-time chat application where users can join different chat rooms or create their own chat rooms. Users should be able to send and receive messages in real-time.

Functional Requirements

1. Allow users to create/join chat rooms by entering a unique room ID.
2. Enable users to send and receive messages in real-time within a chat room.
3. Display a list of active users in the chat room.
4. Optional: Implement a private messaging feature between users.
5. Optional: Implement message history, so the chat persists even if the user leaves and rejoins.

Key Focus

1. Behavioral Pattern: Use the Observer Pattern to notify clients of new messages or user activities.
2. Creational Pattern: Use Singleton to manage the state of the chat rooms.
3. Structural Pattern: Use the Adapter Pattern to allow the system to work with different types of client communication protocols (WebSocket, HTTP, etc.).
4. OOP: Apply encapsulation, polymorphism, and inheritance effectively.

Possible Inputs

- Chat Room ID: "Room123"
- Messages: "Hello, everyone!", "How's it going?", "Goodbye!"
- Active Users: ["Alice", "Bob", "Charlie"]

Possible Outputs

- Chat Messages: Display a list of messages in the format `[username]: [message]`
- Active Users: Display a list of usernames of the active users in the chat room.

Evaluation

1. **Code Quality**: Does the code adhere to best practices, SOLID principles, and design patterns?
2. **Functionality**: Does the application satisfy all the functional requirements, and does it handle real-time data effectively?
3. **Global Convention**: Is the code structured and written in a globally understandable manner?
4. **Gold Standards**: Are there gold standards in logging, error handling, and other robust features?
5. **Code Walkthrough**: Will the candidate be able to articulate the decisions taken during coding, design patterns used, and the overall architecture?

Intent of this exercise is to cover a balance blend of complexity and practicality, allowing candidates to display both their coding finesse and architectural understanding...

6. Satellite Command System Programming Exercise

Problem Statement

You are tasked with developing a Satellite Command System that simulates controlling a satellite in orbit. The satellite starts in a default initial state and can accept a series of commands to change its orientation, solar panel status, and data collection.

Functional Requirements

1. **Initialize the Satellite**: Create a class or function that initializes the satellite's attributes to their initial state.
2. **Rotate**: Implement a command called 'rotate' that takes a direction parameter (North, South, East, West) and sets the satellite's orientation accordingly.

   Example: `rotate(North)` would set the orientation to "North".
3. **Activate/Deactivate Solar Panels**: Implement commands called 'activatePanels' and 'deactivatePanels' to control the solar panels' status.

   Example: `activatePanels()` would set the solar panels to "Active".
4. **Collect Data**: Implement a command called 'collectData' that increments the 'Data Collected' attribute by 10 units, but only if the solar panels are "Active".

   Example: `collectData()` would set the data collected to 10 if the solar panels are "Active".

Initial State

Your satellite begins with the following attributes:

- Orientation: "North"
- Solar Panels: "Inactive"
- Data Collected: 0

Commands to be executed

The series of commands should be executed in a sequential manner over the initial state, altering the satellite's state accordingly. You could execute them through function calls, or simulate a command-line interface where these commands can be entered.

For example:

- `rotate(South)`
- `activatePanels()`
- `collectData()`

After these commands, the satellite's state would be:

- Orientation: "South"
- Solar Panels: "Active"
- Data Collected: 10

Evaluation Criteria

1. **Code Quality**: Utilization of best practices, SOLID principles, and design patterns is highly encouraged.
2. **Functionality**: Does your implementation correctly execute the commands and maintain the satellite's state?
3. **Global Convention**: Your code should be easily understandable and maintainable.
4. **Gold Standards**: Logging, exception-handling, and transient error-handling mechanisms should be implemented.
5. **Code Walkthrough**: Be prepared to explain your decisions, design patterns used, and the overall architecture during a code review.

The intent behind the 'Satellite Command System' coding exercise is to evaluate a candidate's ability to design and implement a system that manages state transitions and dependencies between various components... The challenge provides a realistic yet simplified simulation of what engineers might encounter in fields like aerospace technology or complex system management.

This question not only assesses the technical skills and coding abilities of the candidates but also their understanding of design patterns, state management, and system design principles. The task requires them to employ SOLID principles and various design patterns, which are crucial for writing maintainable, scalable, and robust software. It also allows candidates to demonstrate their expertise in command execution, error handling, and logging—all of which are highly relevant in real-world applications...

---

7. Rocket Launch Simulator Programming Exercise

Problem Statement

You are to build a terminal-based Rocket Launch Simulator. The goal is to simulate a rocket launch that places a satellite into orbit, while providing real-time updates to the user. The simulator will operate in discrete time steps, each representing one second of the mission.

Initial State

- Stage: "Pre-Launch"
- Fuel: 100%
- Altitude: 0 km
- Speed: 0 km/h

User Input

1. **Pre-Launch Checks**: User must type `start_checks` to initiate system checks.
2. **Launch**: User types `launch` to begin the mission after checks are complete.
3. **Fast Forward**: User types `fast_forward X` to advance the simulation by X seconds.

Console Output

- **Pre-Launch Checks**: "All systems are 'Go' for launch."
- **Each Second of Flight**: "Stage: 1, Fuel: 90%, Altitude: 10 km, Speed: 1000 km/h"
- **Stage Separation**: "Stage 1 complete. Separating stage. Entering Stage 2."
- **Orbit Placement**: "Orbit achieved! Mission Successful."
- **Mission Failure**: "Mission Failed due to insufficient fuel."

Functional Requirements

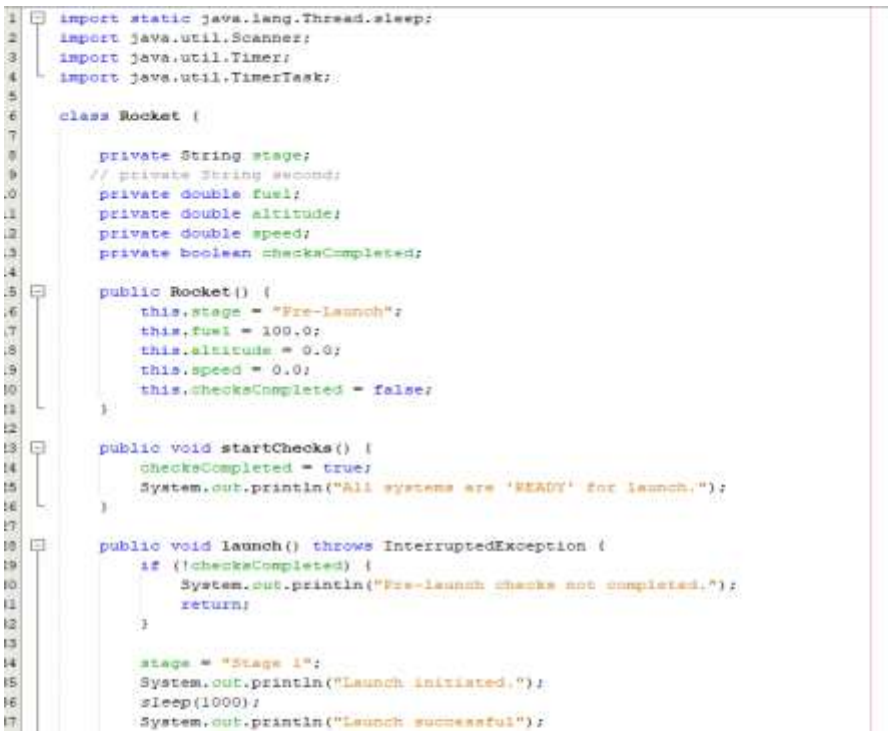
1. **Pre-Launch Checks**: Upon typing `start_checks`, the terminal should output whether all systems are 'Go' for launch.
2. **Launch and Stage Updates**: When the user types `launch`, the terminal starts updating the stage, fuel, altitude, and speed every second.
3. **Fast Forward**: If the user types `fast_forward X`, the program will calculate and display the updated parameters after X seconds.

Evaluation Criteria

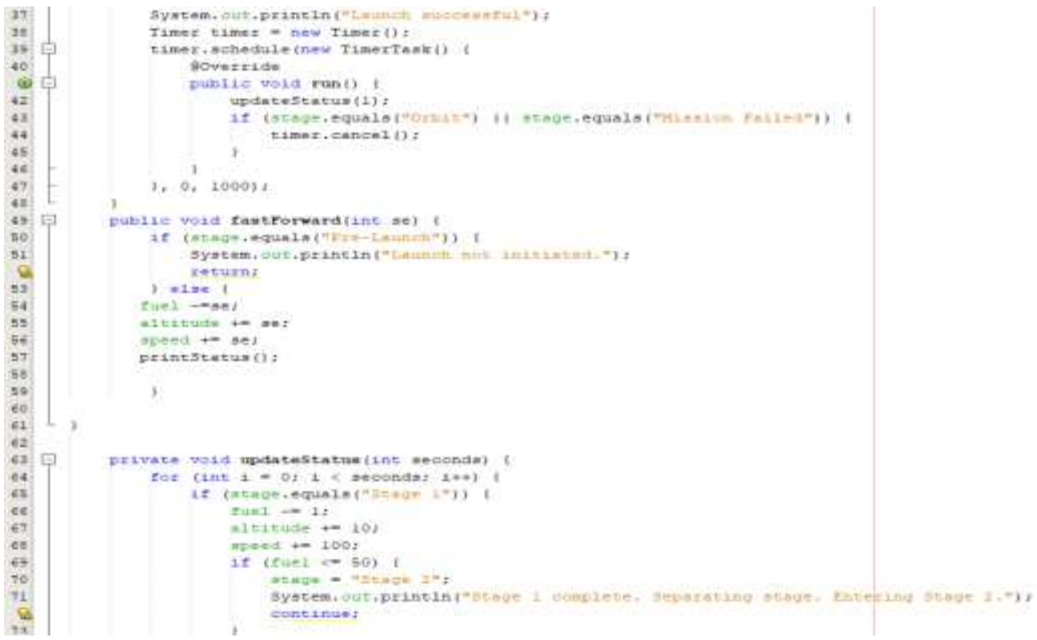
1. **Code Quality**: Adherence to best practices, SOLID principles, and design patterns.
2. **Functionality**: Does the simulator accurately represent the rocket launch process?
3. **Global Convention**: Code should be easily understandable and maintainable.
4. **Gold Standards**: Logging, exception handling, and transient error handling should be implemented.

5. **Code Walkthrough**: Be prepared to walk us through your code, explaining your decisions and any design patterns used.

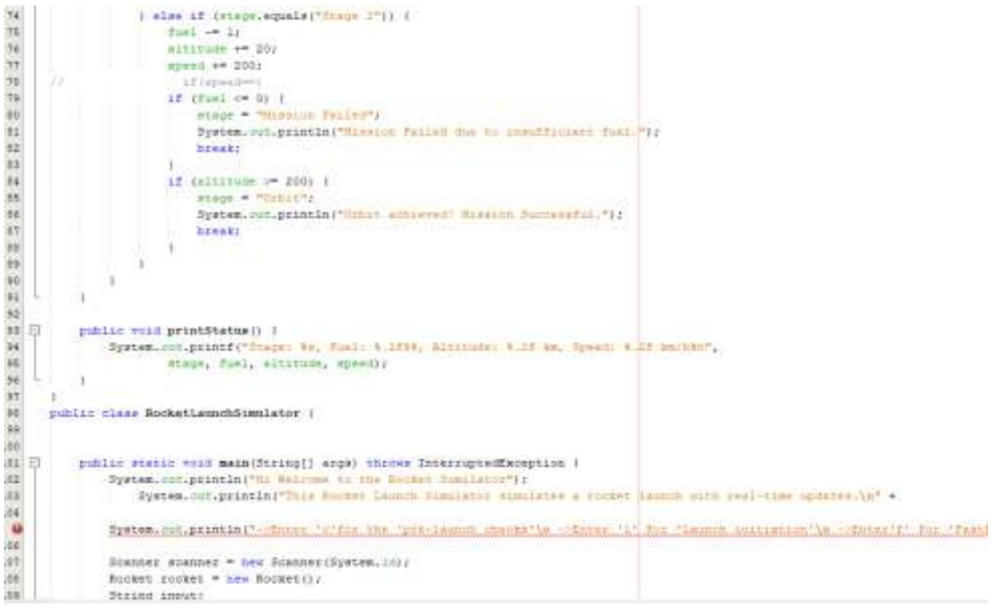
The intent behind this problem statement is to create a scenario where candidates not only apply basic coding skills but also showcase their ability to solve complex real-world problems that involve multiple moving parts and states. In the EdTech industry, these types of problems are common, whether they involve managing the logic of a learning platform or building interactive educational simulations...



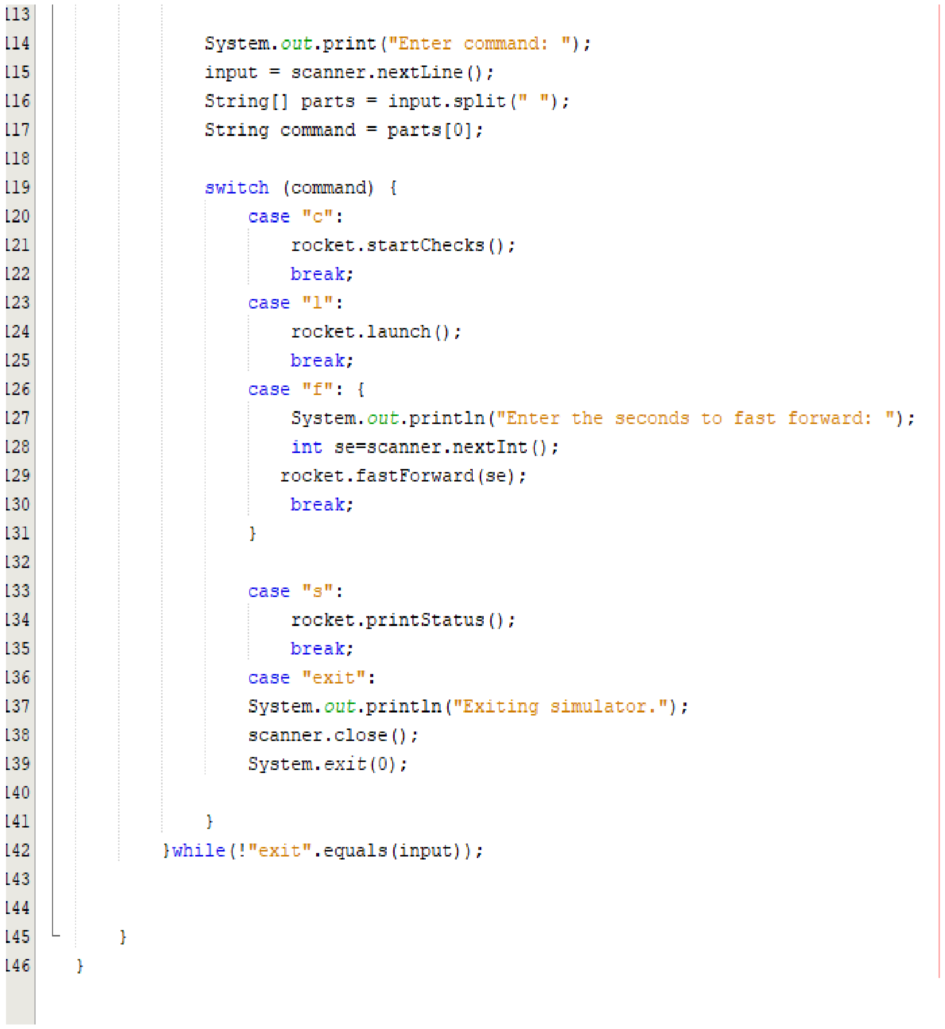
```java
import static java.lang.Thread.sleep;
import java.util.Scanner;
import java.util.Timer;
import java.util.TimerTask;

class Rocket {

    private String stage;
   // private String second;
    private double fuel;
    private double altitude;
    private double speed;
    private boolean checksCompleted;

    public Rocket() {
        this.stage = "Pre-Launch";
        this.fuel = 100.0;
        this.altitude = 0.0;
        this.speed = 0.0;
        this.checksCompleted = false;
    }

    public void startChecks() {
        checksCompleted = true;
        System.out.println("All systems are 'READY' for launch.");
    }

    public void launch() throws InterruptedException {
        if (!checksCompleted) {
            System.out.println("Pre-launch checks not completed.");
            return;
        }

        stage = "Stage 1";
        System.out.println("Launch initiated.");
        sleep(1000);
        System.out.println("Launch successful");
```



```java
        System.out.println("Launch successful");
        Timer timer = new Timer();
        timer.schedule(new TimerTask() {
            @Override
            public void run() {
                updateStatus(1);
                if (stage.equals("Orbit") || stage.equals("Mission Failed")) {
                    timer.cancel();
                }
            }
        }, 0, 1000);
    }
    public void fastForward(int se) {
        if (stage.equals("Pre-Launch")) {
            System.out.println("Launch not initiated.");
            return;
        } else {
       fuel -=se;
       altitude += se;
       speed += se;
       printStatus();

        }

}

    private void updateStatus(int seconds) {
        for (int i = 0; i < seconds; i++) {
            if (stage.equals("Stage 1")) {
                fuel -= 1;
                altitude += 10;
                speed += 100;
                if (fuel <= 50) {
                    stage = "Stage 2";
                    System.out.println("Stage 1 complete. Separating stage. Entering Stage 2.");
                    continue;
                }
```



```java
            } else if (stage.equals("Stage 2")) {
                fuel -= 1;
                altitude += 20;
                speed += 200;
//                if(speed==)
                if (fuel <= 0) {
                    stage = "Mission Failed";
                    System.out.println("Mission Failed due to insufficient fuel.");
                    break;
                }
                if (altitude >= 200) {
                    stage = "Orbit";
                    System.out.println("Orbit achieved! Mission Successful.");
                    break;
                }
            }
        }
    }

    public void printStatus() {
        System.out.printf("Stage: %s, Fuel: %.2f%%, Altitude: %.2f km, Speed: %.2f km/h%n",
                stage, fuel, altitude, speed);
    }
}
public class RocketLaunchSimulator {


    public static void main(String[] args) throws InterruptedException {
        System.out.println("Hi Welcome to the Rocket Simulator");
            System.out.println("This Rocket Launch Simulator simulates a rocket launch with real-time updates.\n" +

        System.out.println("->Enter 'c'for the 'pre-launch checks'\n ->Enter 'l' for 'Launch initiation'\n ->Enter'f' for 'Fast

        Scanner scanner = new Scanner(System.in);
        Rocket rocket = new Rocket();
        String input;
```



```java

            System.out.print("Enter command: ");
            input = scanner.nextLine();
            String[] parts = input.split(" ");
            String command = parts[0];

            switch (command) {
                case "c":
                    rocket.startChecks();
                    break;
                case "l":
                    rocket.launch();
                    break;
                case "f": {
                    System.out.println("Enter the seconds to fast forward: ");
                    int se=scanner.nextInt();
                   rocket.fastForward(se);
                    break;
                }

                case "s":
                    rocket.printStatus();
                    break;
                case "exit":
                System.out.println("Exiting simulator.");
                scanner.close();
                System.exit(0);

            }
        }while(!"exit".equals(input));


    }
}
```

8. Virtual Classroom Manager Programming Exercise

Problem Statement

Imagine you are developing the backend for an EdTech platform that aims to host virtual classrooms. Your task is to create a terminal-based Virtual Classroom Manager that handles class scheduling, student attendance, and assignment submissions.

Initial State

- Number of Classrooms: 0
- Number of Students: 0
- Number of Assignments: 0

User Input

1. **Add Classroom**: User types `add_classroom` followed by the class name to create a new virtual classroom.
2. **Add Student**: User types `add_student` followed by the student ID and the class name to enroll a student in a classroom.
3. **Schedule Assignment**: User types `schedule_assignment` followed by class name and assignment details to add an assignment for a class.
4. **Submit Assignment**: User types `submit_assignment` followed by student ID, class name, and assignment details to mark an assignment as submitted.

Console Output

- **Classroom Addition**: "Classroom [Name] has been created."
- **Student Addition**: "Student [ID] has been enrolled in [Class Name]."
- **Assignment Scheduled**: "Assignment for [Class Name] has been scheduled."
- **Assignment Submission**: "Assignment submitted by Student [ID] in [Class Name]."

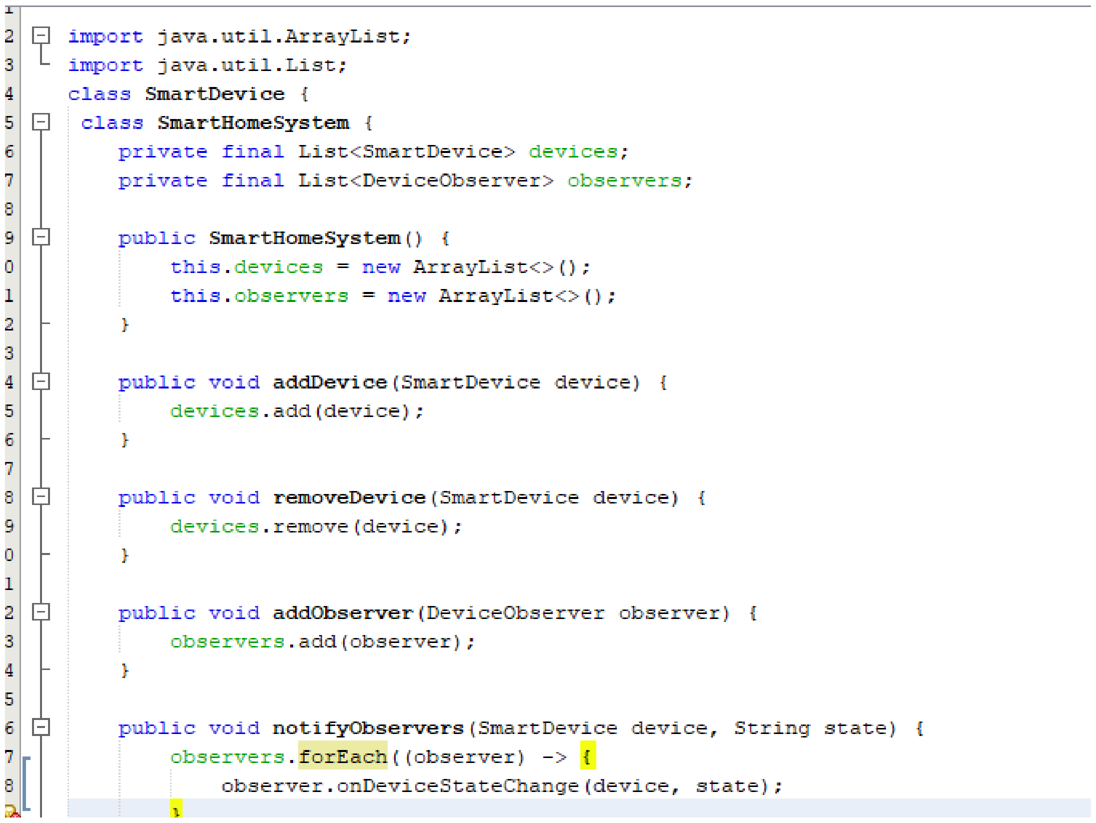
Functional Requirements

1. **Classroom Management**: Ability to add, list, and remove virtual classrooms.
2. **Student Management**: Ability to enroll students into classrooms, and list students in each classroom.
3. **Assignment Management**: Schedule assignments for classrooms and allow students to submit them.

Evaluation Criteria

1. **Code Quality**: Importance will be given to best practices, SOLID principles, and the use of appropriate design patterns.
2. **Functionality**: The terminal-based application should be fully functional and handle various classroom operations efficiently.
3. **Global Convention**: Adherence to coding standards for readability and maintainability.
4. **Gold Standards**: The code should include logging, exception handling, and transient error handling.
5. **Code Walkthrough**: Candidates should be able to fully walk us through their code and the decisions made during development.

The exercise has been designed to echo the real-world complexities that come with managing an educational platform... It's an engaging problem that evaluates a candidate's ability to model relationships between entities like students and classrooms, and manage state, all within the constraints of a terminal-based application.



```
import java.util.ArrayList;
import java.util.List;
class SmartDevice {
 class SmartHomeSystem {
    private final List<SmartDevice> devices;
    private final List<DeviceObserver> observers;

    public SmartHomeSystem() {
        this.devices = new ArrayList<>();
        this.observers = new ArrayList<>();
    }

    public void addDevice(SmartDevice device) {
        devices.add(device);
    }

    public void removeDevice(SmartDevice device) {
        devices.remove(device);
    }

    public void addObserver(DeviceObserver observer) {
        observers.add(observer);
    }

    public void notifyObservers(SmartDevice device, String state) {
        observers.forEach((observer) -> {
            observer.onDeviceStateChange(device, state);
        }
```