1) Astronaut Daily Schedule organiser programming Exercise:

Exercise 1

i) Behavioural design patterns

Observer Pattern Use Case: An application for weather stations in which temperature, pressure, and humidity variations are recorded by the weather station (subject), and the various display units (observers) are updated accordingly.

Command Pattern Use Case: A home automation system in which various devices (lights, fans, TVs, etc.) receive commands from a central controller to carry out functions such as turning on, off, or altering settings.

ii) Patterns of Creational Design

Use Case for Singleton Pattern: A database connection manager that permits just one connection instance per application. By ensuring that every request makes use of the same connection object, this helps to avoid wasting resources.

Use Case for Factory Pattern: A system for creating automobiles in which various vehicle kinds (cars, trucks, and bikes) are created using a factory technique in response to user input.

iii) Structural Design Patterns

Adapter Pattern Use Case: You wish to use an adapter to enable your legacy music player, which can only play MP3 files, to play MP4 and other formats. Here is the program for media players.

Use Case for façade Pattern: A sophisticated library system in which the façade conceals the intricacy of the underlying activities while offering a straightforward interface for book searches, borrowing, and returning.

Exercise 2

i) Scheduling Manager's Singleton Pattern

To guarantee that a single instance of the Schedule Manager class handles all of the tasks for the day, it should be implemented as a singleton. Restricting the creation of objects and offering a global access point can help achieve this.
https://github.com/12phai/project/blob/6ea0beed1c937365418b259e14f9b2786133ea8a/public%20class%20ScheduleManager%20%7B.java

ii)TaskFactory's Factory Pattern

Several task objects with the necessary properties (description, start time, finish time, and priority) will be created by the TaskFactory. This guarantees the centralization of the creation rationale.
https://github.com/12phai/jhjk/blob/1253a0eea5adbb963d5f5852a880e192552a19ac/2.java

iii) Observer Configuration (Dispute Alerts)

An observer can be used to address task conflicts by alerting users when a new task clashes with an already-existing one. If a job overlaps, the TaskConflictObserver will watch the ScheduleManager and sound an alert.

https://github.com/12phai/jhjk/blob/1253a0eea5adbb963d5f5852a880e192552a19ac/import%20java.util.java

IV) Operations of ScheduleManager

To make sure that the tasks are created, added, and conflict-checked, the ScheduleManager will merge the factory and observer patterns.
https://github.com/12phai/jhjk/blob/1253a0eea5adbb963d5f5852a880e192552a19ac/import%20java.util%204.java

Important Logic Elements:

Add Task: This method starts by utilizing the factory to generate a task. It then uses the observer to see if there are any conflicts, and if not, it adds the job to the schedule.
Eliminate Task: By matching the description, this approach eliminates a task.
View Tasks: This function shows a list of all the tasks arranged by start time.

Examples of Input and Output

i) Favorable Situation:
Task Add("Morning Exercise", "07:00", "08:00", "High") in the input
Results: The task "Morning Exercise" was added with success. No disputes.

ii) Negative Situation:
Task Add("Overlap Task", "07:30", "08:30", "Medium") is the input.
Results: Error: This task interferes with the current "Morning Exercise" task.

This design adheres to SOLID principles, makes effective use of the necessary design patterns, and maintains a clean, modular codebase.

2)Smart Office Facility Exercise

Exercise 1

Key Design Patterns:

i) Singleton Pattern: Make sure that all office reservations and rooms are managed by the OfficeManager class, which has a single instance for all global state.
ii) Observer Pattern: Control systems (air conditioning, lights) and sensors (occupancy detection) should monitor the state of the room and take appropriate action.
iii) Command Pattern: Use command classes to manage reservations, changes, and cancellations for adaptable and scalable operations.

Exercise 2

Design Overview:

i)Singleton Pattern: OfficeManager

The officemanager manages the configuration and state of the entire office.It is responsible for creating rooms, handling bookings, and keeping track of room status.

https://github.com/12phai/jhjk/blob/main/2%201.java

ii)Observer pattern: Room, Sensors and Control Systems

Rooms should notify their observers whenever occupancy changes. The sensors will detect whether the room is occupied accordingly.

https://github.com/12phai/jhjk/blob/main/2%202.java

iii)Command Pattern : Booking , cancellation, and status Commands

The command pattern will allow you to encapsulate booking, cancellation and status updates as commands enabling easier management of operations and extensibility.

https://github.com/12phai/jhjk/blob/main/2%203.java

Key Functionalities:

i)Configure office:

input: Config room count 3

output: office configured with 3 meeting rooms

ii)Set Room capacity:

input: config room max capacity 1 10

output: Room 1 maximum capacity set to 10

iii)Add occupants:

input: Add occupant 1 2

output: Room 1 booked from 09:00 for 60 minutes

iv)Book Room:

input: Block room 1 09:00 60

output: Room 1 booked from 09:00 for 60 minutes.

v)Cancel Room Booking:

input: cancel room 1

output: Booking for Room 1 cancelled successfully

vi)Remove occupants:

input: Add occupant 1 0

output: Room1 is now unoccupied, AC and lights turned off.

Important Logic Elements:

Room Occupancy: The quantity of persons in the room determines the occupancy. The AC and lights are shut off and the room becomes empty if the occupancy fails below two.

Reservation: Reservations are accepted only when the room is not in use. If the accommodation is not used for more than five minutes, the reservation is immediately cancelled.

3)Mars Rover Programming Exercise

Exercise 1

Key Design Patterns:

i)Command Pattern: Movement instructions (M, L, and R) should be encapsulated as distinct objects for flexibility and extensibility in the command pattern.

ii)Composite Pattern: Using a composite structure—where the grid is made up of cells and some cells have the capacity to store obstacles—represent the grid and the obstacles.

Exercise 2

Design Overview:

i)Command Pattern: Movement commands(M,L,R)

Each command will be encapsulated in a separate class, and the rover will execute these commands without using conditional constructs like if-else.

https://github.com/12phai/jhjk/blob/main/3%201.java

ii)Composite pattern: Grid and Obstacles

The grid will be represented as a collection of cell objects , some of which will have obstacles. This allows for easier management of obstacles and grid boundaries.

https://github.com/12phai/jhjk/blob/main/3%202.java

iii)Rover class: Handles position, Direction and Movements

The Rover class will be responsible for maintaining its position , direction and interaction with the grid . It will execute the commands passed to it.

https://github.com/12phai/jhjk/blob/main/3%203.java

Usage Example:

1. **Initialise the Grid:**
   Input: Grid size: (10 x 10)
   Output: A 10 x 10 grid initialised.
   https://github.com/12phai/jhjk/blob/main/3%20ex%201.java

2. **Initialise the Rover:**
   Input: Starting position: (0,0,N)

Output: Rover object initialised at (0,0) facing North.
https://github.com/12phai/jhjk/blob/main/3%20ex%202.txt

3. **Execute Commands:**
Input: Commands: [‘M’,’M’,’R’,’M’,’L’,’M’]
Output: Rover follows the commands and reports status along the way.
https://github.com/12phai/jhjk/blob/main/3%20ex%203%201.java
https://github.com/12phai/jhjk/blob/main/3%20ex%203%202.java

Example Scenario:

1.Command Sequence:

- ‘M’ : Move from (0,0) to (1,0)
- ‘M’ : Move from (0,1) to (0,2)
- ‘R’ : Turn right to face East
- ‘M’ : Move from (0,2) to (1,2)
- ‘L’ : Turn left to face North.
- ‘M’ : Move from (1,2) to (1,3)

2.Final Output:

- Final position: (1,3,N)
- Status Report: “Rover is at (1,3) facing North. No obstacles detected.”

Key Design Considerations:

- Movement instructions (M, L, and R) should be encapsulated as distinct objects for flexibility and extensibility in the command pattern.
- It is simple to add new commands to the Command Pattern, like a command to send a status report. Modular and scalable terrain management is made possible by Composite Pattern for the grid and impediments.
- The functionality for movement, turning, and obstacle detection is appropriately encapsulated within the pertinent classes thanks to encapsulation and OOP principles.

  In order to guarantee maintainability and flexibility in the Mars Rover simulation, this method eliminates conditional constructs like if-else, makes effective use of design patterns, and complies with SOLID principles.

4) Smart Home system programming Exercise

Exercise 1

For this project, we will use the Factory Method, Proxy, and Observer design patterns to create a Smart Home System. The aim is to imitate a system that permits scheduling, automation, and access control and manages several smart devices, including door locks, lights, and thermostats.

<u>Overview of Design Patterns:</u>

<u>i) Observer Pattern :</u> Devices watch the hub for commands or changes in status, using the observer pattern. All associated devices are alerted when a change takes place (for example, when a thermostat reaches a particular temperature).
<u>ii) Factory Method Pattern :</u> Utilizing the factory method pattern, various smart device kinds (such as door locks, thermostats, and lights) can be made without hardcoding their instantiation logic.
<u>iii) Proxy Pattern:</u> Used to manage device access, enabling extra security or features like restricted control over particular devices, authentication, or logging.

Exercise 2

<u>Classes and Design</u>

<u>1)Factory Method Pattern</u> : Creating Smart Devices

We will create a DeviceFactory that instantiates devices based on type(light, thermostat, doorlock).

https://github.com/12phai/jhjk/blob/main/4%201.java

Now, we create a DeviceFactory that generates these device instances.

https://github.com/12phai/jhjk/blob/main/4%201%202.groovy

<u>2)Observer Pattern:</u> Hub and Device Interaction

The smart Hub will manage all devices and notify them of any updates. Devices will subscribe to the hub and act on commands or triggers.

https://github.com/12phai/jhjk/blob/main/4%202.java

<u>3)Proxy Pattern:</u>

To stimulate controlled access to devices, we will implement a proxy for the hub, which can restrict device control.

https://github.com/12phai/jhjk/blob/main/4%203.java

<u>Simulation Examples:</u>

1)Initialise Devices:

https://github.com/12phai/jhjk/blob/main/4%20ex%201.java

2)Control Devices and Set Automations:

https://github.com/12phai/jhjk/blob/main/4%20ex%202.java

3)Check Status:

https://github.com/12phai/jhjk/blob/main/4%20ex%203.java

Example Outputs:

1.Command Execution:

- Light 1 is on
- Door 3 is locked
- Thermostat 2 temperature set to 76 degree F
- Light 1 is off due to automation trigger.

2.Status Report:

- "Light 1 is off. Thermostat 2 is set to 76 degree F. Door 3 is Locked".
- Unauthorised user Charlie attempted to view status.

In summary, a clean device instantiation is ensured by the Factory Method Pattern (Light, Thermostat, DoorLock).
Observer Pattern enables automated reactions to trigger events or changes in device state.
By enforcing access control, Proxy Pattern makes sure that only users with permission may examine device statuses and issue commands.
This approach builds a scalable and maintainable system for controlling smart home devices by following SOLID principles, encapsulating logic within each class, and skillfully utilizing design patterns.

5)Real-Time Chat Application Programming Exercise

Exercise 1

Design Patterns Overview:

1.Observer Pattern: Each chat room will act as the subject, notifying all connected users when a new message is posted.

2.Singleton Pattern: The chat room manager will be a singleton that maintains the state of all active chat rooms.

3.Adapter Pattern: This pattern will abstract the communication layer, allowing the chat system to be extended for different protocols without changing the core logic.

Exercise 2

Classes and Design

1)Singleton Pattern: Chat Room Manager

We will use the Singleton Pattern to ensure there is only one instance managing all chat rooms in the application.

https://github.com/12phai/jhjk/blob/main/5%201.groovy

2)Observer Pattern: Chat Room Users

The Observer Pattern will be used to notify users in a chat room whenever a new message is posted. The chat room acts as the subject, and each user is an observer.

https://github.com/12phai/jhjk/blob/main/5%202.java

3)Adapter Pattern: Communication Protocol

We will implement the Adapter Pattern to decouple the application logic from the communication layer. This allows flexibility in supporting various protocols.

https://github.com/12phai/jhjk/blob/main/5%203.java

Application Workflow:

1)Create or Join Chat Rooms: Users can create or join existing chat rooms by specifying a unique room ID.

https://github.com/12phai/jhjk/blob/main/5%20ex%201.java

2)Send and Receive Messages: Users can send message within the chat room.

https://github.com/12phai/jhjk/blob/main/5%20ex%202.java

3)Display Active Users: The chat room keeps track of active users and can display the current list of participants.

https://github.com/12phai/jhjk/blob/main/5%20ex%203.java

4)Private Messaging: You can extend this by allowing users to send direct messages to each other outside of a chat room.

https://github.com/12phai/jhjk/blob/main/5%20ex%204.java

Example Outputs:

1.User Joins a Room:

```
Alice has joined room Room123.

Bob has joined room Room123.
```

2. Real Time Message Broadcasting:

```
Alice received message: [Alice]: Hello, everyone!

Bob received message: [Alice]: Hello, everyone!

Alice received message: [Bob]: Hi, Alice!

Bob received message: [Bob]: Hi, Alice!
```

3.Display Active Users:

```
Active users: ['Alice', 'Bob']
```

Summary

By using the singleton pattern, all chat rooms are managed by a single instance of the ChatRoomManager.
By broadcasting messages to every user in a chat room, the observer pattern ensures real-time conversation.
Adapter Patterns enable extensibility for various client-server interactions by providing flexibility in communication protocols (WebSocket, HTTP, etc.).
This implementation encourages the separation of concerns, complies with SOLID principles, and is readily expandable. Future features like private messaging and message permanence are made flexible by the design principles.

6) Satellite Command System Programming

Exercise 1

Key Design Patterns:

1.Command Pattern: Encapsulate commands such as rotate, active panels, deactivate Panels, and collect Data into objects

2.Singleton Pattern: Ensure there is only one instance of the satellite that manages its state across the system.

3.Observer Pattern: Can be used to log or notify when the satellite's state changes.

Satellite Attributes:

- **Orientation:** Initially set to "North".
- **Solar Panels:** Initially " Inactive".
- **Data Collected:** Initially 0 .

Exercise 2

Solution Design:

1)Satellite Class(Singleton):

The Satellite class will manage the satellite's state.

https://github.com/12phai/jhjk/blob/main/6%201.java

2)Command Pattern:

Encapsulate each command into individual command classes. This will allow the commands to be executed sequentially and maintain flexibility.

https://github.com/12phai/jhjk/blob/main/6%202.java

3)Command Invoker:

The invoker will manage the command execution process. It stores and executes the list of commands sequentially.

https://github.com/12phai/jhjk/blob/main/6%203.java

Example Usage:

1)Initialise Satellite: The satellite will start with an initial state: orientation “North”, solar panels “ Inactive”, and 0 data collected.

https://github.com/12phai/jhjk/blob/main/6%20ex%201.cs

2)Execute Commands: Sequentially execute commands such as rotating, activating panels, and collecting data.

https://github.com/12phai/jhjk/blob/main/6%20ex%202.java

Class Diagram Overview:

```
+----------------------------+                    +-------------------+

|  CommandInvoker  |     <------->      |     Command     |

+----------------------------+                     +-------------------+

| +add_command()   |                        | +execute()       |

| +execute_commands()|                       +-------------------+

+----------------------------+                           ^

                                                         |

                              +-----------------------+-----------------------+

                    |                |                    |               |

           +------------------------+ +----------------------------+ +------------------------+

           | RotateCommand    | | ActivatePanelsCmd  | | CollectDataCmd        |

+----------------------------------------+ +---------------------------+ +------------------------+

                 | +execute()      | | +execute()       | | +execute()           |

+---------------------------------------+ +-----------------------------+ +------------------------+

                                                    ^

                                                    |

                                      +---------------------------------+

                                      |      Satellite (Singleton)  |

                                      +---------------------------------+

                                      | Orientation: "North"        |

                                      | Solar Panels: "Inactive"     |

                                      | Data Collected: 0             |

                                      +------------------------------------+

                                      | +rotate()                   |

                                      | +activate_panels()          |

                                      | +deactivate_panels()      |
| +collect_data()              |
+----------------------------------+
```

Summary

Every command is contained in a command pattern, which offers flexibility in the order in which commands can be executed and expanded.
The satellite instance is guaranteed to be consistent and globally accessible by the Singleton Pattern.
With its distinct division of responsibilities and modular design, the solution complies with SOLID guidelines.
Robustness is ensured by error handling, and logging can be added for monitoring.
This method ensures a flexible, maintainable architecture by imitating real-world command and control systems used in aerospace and satellite management scenarios.

7) Rocket Launch Simulator Programming Exercise

Exercise 1

Key Design Patterns:

1)State Pattern: Model the rocket's stages – Pre-Launch, Stage1, stage2, and Orbit as states that the system transitions between.

2)Command Pattern: Encapsulate user commands like start_checks, launch, and fast_forward to decouple input handling from the logic.

3)Singleton Pattern: Ensure a single instance of the Rocket Class to track its state globally throughout the simulation.

4)Observer Pattern: Used to notify observers when the rocket's state changes.

Simulation Breakdown:

1)Initial State:

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

2)State Transitions:

- Pre-Launch: User runs system checks.
- Launch: The rocket enters stage 1.
- Stage1 : The rocket climbs until fuel reaches 50 %, then it transitions to stage 2.
- Stage 2: The rocket continues climbing until fuel reaches 0 % .
- Orbit: The rocket achieves a stable Orbit.

Exercise 2

Solution Design:

1)Rocket Class(Singleton):

This class manages the rocket's state , including its fuel, altitude, speed and current stage.

https://github.com/12phai/jhjk/blob/main/7%201.java

2)State Pattern for Rocket Stages:

We will create separate classes to represent each stage of the rocket and define the behaviour specific to that stage.\

https://github.com/12phai/jhjk/blob/main/7%202.java

3)Command Pattern for User Input:

Commands like start_checks, launch , and fast_forward will be implemented using the command Pattern.

https://github.com/12phai/jhjk/blob/main/7%203.java

4)Stage Context:

This context manages the transitions between different stages.

https://github.com/12phai/jhjk/blob/main/7%204.java

Example Usage:

1)Pre-Launch Checks: Initiate system checks using the startChecksCommand.

https://github.com/12phai/jhjk/blob/main/7%20ex%201.java

2)Launch the Rocket: Use the LaunchCommand to initiate the rocket launch.

https://github.com/12phai/jhjk/blob/main/7%20ex%202.java

3)Fast Forward: Use the FastForwardCommand to advance the simulation by a specified number of seconds.

https://github.com/12phai/jhjk/blob/8260fcc59e1c4b5f79760c27e60dbdee675a4956/7%20ex%203.java

Summary:

State Pattern offers a tidy method for controlling the rocket's phases and changes.
The rocket's operational logic and user input are separated by the Command Pattern.
By ensuring that there is only one instance of the rocket, Singleton Pattern maintains the rocket's state constant throughout the simulation.
Since it follows SOLID principles and is modular, the design is simple to expand (e.g., adding new stages, handling more complex scenarios).
This strategy guarantees a realistic, scalable, and maintainable rocket simulation experience that can handle challenging real-world situations.