Name: Mihir Makwana

Roll NO: 21BIT247

Assignment of Educational Initiatives

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 patterns.

2. Two use cases to demonstrate two creational design patterns.

3. Two use cases to demonstrate two structural design patterns.

1. Behavioural Design Patterns

Case-1: Observer Pattern – Stock Market Ticker

Code:

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

interface Observer {
    void update(Stock stock);
}

class Stock {
    private String name;
    private double price;
    private List<Observer> observers = new ArrayList<>();

    public Stock(String name) {
        this.name = name;
    }

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

    public void setPrice(double price) {
        this.price = price;
        notifyObservers();
    }

    private void notifyObservers() {
        for (Observer observer : observers) {
```

```
            observer.update(this);
        }
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }
}

class Investor implements Observer {
    private String name;

    public Investor(String name) {
        this.name = name;
    }

    @Override
    public void update(Stock stock) {
        System.out.println(name + " notified! New price of " + stock.getName()
+ " is " + stock.getPrice());
    }
}

public class ObserverPatternExample {
    public static void main(String[] args) {
        Stock stock = new Stock("AAPL");
        Investor investor1 = new Investor("ABC");
        Investor investor2 = new Investor("XYZ");

        stock.addObserver(investor1);
        stock.addObserver(investor2);

        stock.setPrice(150.0); // Both investors are notified
    }
}
```

Output:

Output

```
java -cp /tmp/jicfVNfy8m/ObserverPatternExample
ABC notified! New price of AAPL is 150.0
XYZ notified! New price of AAPL is 150.0
```

Case-2: Strategy Pattern – Payment System

Code:

```
interface PaymentStrategy {
    void pay(int amount);
}

class CreditCardPayment implements PaymentStrategy {
    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Credit Card");
    }
}

class PayPalPayment implements PaymentStrategy {
    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using PayPal");
    }
}

class BitcoinPayment implements PaymentStrategy {
    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Bitcoin");
    }
}

class Checkout {
    private PaymentStrategy paymentStrategy;

    public Checkout(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void pay(int amount) {
        paymentStrategy.pay(amount);
    }
}

public class StrategyPatternExample {
    public static void main(String[] args) {
        Checkout checkout = new Checkout(new CreditCardPayment());
        checkout.pay(100);

        checkout = new Checkout(new PayPalPayment());
        checkout.pay(200);
    }
```

```
}
```

Output:

```
Output
java -cp /tmp/vwNjJJEwF4/StrategyPatternExample
Paid 100 using Credit Card
Paid 200 using PayPal
```

2. Creational Design Patterns

Case-1: Singleton Pattern – Database Connection

Code:

```
class DatabaseConnection {
    private static DatabaseConnection instance;

    private DatabaseConnection() {
        // Private constructor to prevent instantiation
    }

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

public class SingletonPatternExample {
    public static void main(String[] args) {
        DatabaseConnection db1 = DatabaseConnection.getInstance();
        DatabaseConnection db2 = DatabaseConnection.getInstance();

        System.out.println(db1 == db2); // True, both refer to the same
instance
    }
}
```

Output:

```
Output
java -cp /tmp/UmlNdZkse2/SingletonPatternExample
true
```

Case-2: Factory Method Pattern – Vehicle Factory

Code:

```
interface Vehicle {
    String create();
}

class Car implements Vehicle {
    @Override
    public String create() {
        return "Car created";
    }
}

class Bike implements Vehicle {
    @Override
    public String create() {
        return "Bike created";
    }
}

class VehicleFactory {
    public Vehicle getVehicle(String vehicleType) {
        if (vehicleType == null) {
            return null;
        }
        if (vehicleType.equalsIgnoreCase("CAR")) {
            return new Car();
        } else if (vehicleType.equalsIgnoreCase("BIKE")) {
            return new Bike();
        }
        return null;
    }
}

public class FactoryMethodExample {
    public static void main(String[] args) {
        VehicleFactory factory = new VehicleFactory();

        Vehicle vehicle1 = factory.getVehicle("CAR");
        System.out.println(vehicle1.create());

        Vehicle vehicle2 = factory.getVehicle("BIKE");
        System.out.println(vehicle2.create());
    }
}
```

Output:

```
Output
java -cp /tmp/F6zIbODtRK/FactoryMethodExample
Car created
Bike created
```

3. Structural Design Patterns

Case 1: Adapter Pattern – Charging Devices

Code:

```
class PowerOutlet220V {
    public String supplyPower() {
        return "220V power supplied";
    }
}

class Device110V {
    public void usePower(String power) {
        if ("110V power supplied".equals(power)) {
            System.out.println("Device is using 110V power");
        } else {
            System.out.println("Cannot use this power supply");
        }
    }
}

class PowerAdapter {
    private PowerOutlet220V outlet;

    public PowerAdapter(PowerOutlet220V outlet) {
        this.outlet = outlet;
    }

    public String convertPower() {
        return "110V power supplied";
    }
}

public class AdapterPatternExample {
    public static void main(String[] args) {
        PowerOutlet220V outlet = new PowerOutlet220V();
        PowerAdapter adapter = new PowerAdapter(outlet);
        Device110V device = new Device110V();
```

```
        String power = adapter.convertPower();
        device.usePower(power);
    }
}
```

Output:

```
Output
java -cp /tmp/bsrLdJeCX9/AdapterPatternExample
Device is using 110V power
```

Case-2: Facade Pattern – Home Automation System

Code:

```
class Light {
    public void on() {
        System.out.println("Light is on");
    }

    public void off() {
        System.out.println("Light is off");
    }
}

class AirConditioner {
    public void on() {
        System.out.println("Air Conditioner is on");
    }

    public void off() {
        System.out.println("Air Conditioner is off");
    }
}

class SecuritySystem {
    public void activate() {
        System.out.println("Security System activated");
    }

    public void deactivate() {
        System.out.println("Security System deactivated");
    }
```

```
}

class HomeAutomationFacade {
    private Light light;
    private AirConditioner ac;
    private SecuritySystem security;

    public HomeAutomationFacade() {
        light = new Light();
        ac = new AirConditioner();
        security = new SecuritySystem();
    }

    public void enterHome() {
        light.on();
        ac.on();
        security.deactivate();
    }

    public void leaveHome() {
        light.off();
        ac.off();
        security.activate();
    }
}

public class FacadePatternExample {
    public static void main(String[] args) {
        HomeAutomationFacade home = new HomeAutomationFacade();

        home.enterHome();
        home.leaveHome();
    }
}
```

Output:

Output

```
java -cp /tmp/lUGjuFZwPF/FacadePatternExample
Light is on
Air Conditioner is on
Security System deactivated
Light is off
Air Conditioner is off
Security System activated
```

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 candidate's 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!

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. Behavioural 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.

Intent of this exercise is to gauge the candidate's practical knowledge of design patterns, OOP, and coding best practices... This will serve as an insightful lens through which to assess the depth and breadth of their skills.

Code:

```
import java.util.HashSet;
import java.util.Set;

// Command Interface
interface Command {
    void execute();
}

// Concrete Command Classes
class MoveCommand implements Command {
    private Rover rover;

    public MoveCommand(Rover rover) {
        this.rover = rover;
    }

    @Override
    public void execute() {
        rover.moveForward();
    }
}
```

```
class TurnLeftCommand implements Command {
    private Rover rover;

    public TurnLeftCommand(Rover rover) {
        this.rover = rover;
    }

    @Override
    public void execute() {
        rover.turnLeft();
    }
}

class TurnRightCommand implements Command {
    private Rover rover;

    public TurnRightCommand(Rover rover) {
        this.rover = rover;
    }

    @Override
    public void execute() {
        rover.turnRight();
    }
}
class Rover {
    private int x;
    private int y;
    private String direction;
    private Grid grid;

    private static final String[] DIRECTIONS = {"N", "E", "S", "W"};

    public Rover(int x, int y, String direction, Grid grid) {
        this.x = x;
        this.y = y;
        this.direction = direction;
        this.grid = grid;
    }

    public void moveForward() {
        switch (direction) {
            case "N":
                if (grid.isValidMove(x, y + 1)) {
                    y += 1;
                }
                break;
            case "E":
```

```
                if (grid.isValidMove(x + 1, y)) {
                    x += 1;
                }
                break;
            case "S":
                if (grid.isValidMove(x, y - 1)) {
                    y -= 1;
                }
                break;
            case "W":
                if (grid.isValidMove(x - 1, y)) {
                    x -= 1;
                }
                break;
        }
    }

    public void turnLeft() {
        int currentIndex =
java.util.Arrays.asList(DIRECTIONS).indexOf(direction);
        direction = DIRECTIONS[(currentIndex - 1 + DIRECTIONS.length) %
DIRECTIONS.length];
    }

    public void turnRight() {
        int currentIndex =
java.util.Arrays.asList(DIRECTIONS).indexOf(direction);
        direction = DIRECTIONS[(currentIndex + 1) % DIRECTIONS.length];
    }

    public String getStatus() {
        return "Rover is at (" + x + ", " + y + ") facing " + direction + ".";
    }
}


class Grid {
    private int width;
    private int height;
    private Set<Obstacle> obstacles;

    public Grid(int width, int height, Set<Obstacle> obstacles) {
        this.width = width;
        this.height = height;
        this.obstacles = obstacles;
    }

    public boolean isValidMove(int x, int y) {
```

```
        if (x >= 0 && x < width && y >= 0 && y < height && !isObstacle(x, y))
{
            return true;
        }
        return false;
    }

    private boolean isObstacle(int x, int y) {
        for (Obstacle obs : obstacles) {
            if (obs.getX() == x && obs.getY() == y) {
                return true;
            }
        }
        return false;
    }
}

class Obstacle {
    private int x;
    private int y;

    public Obstacle(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}


public class MarsRoverSimulation {
    public static void main(String[] args) {
        // Define grid size and obstacles
        Set<Obstacle> obstacles = new HashSet<>();
        obstacles.add(new Obstacle(2, 2));
        obstacles.add(new Obstacle(3, 5));
        Grid grid = new Grid(10, 10, obstacles);

        // Create a rover starting at (0, 0) facing North
        Rover rover = new Rover(0, 0, "N", grid);

        // Define commands to be executed
        Command[] commands = {
```

```
            new MoveCommand(rover),
            new MoveCommand(rover),
            new TurnRightCommand(rover),
            new MoveCommand(rover)
        };

        // Execute each command
        for (Command command : commands) {
            command.execute();
        }

        // Print final rover status
        System.out.println(rover.getStatus());
    }
}
```

Output:

```
Output
java -cp /tmp/NplYCLG7ui/MarsRoverSimulation
Rover is at (1, 2) facing E.
```