Assignment

Ayush Parmar (21BCP234, G7, CSE)

Exercise 1: Design Patterns Use Cases

1. Behavioral Design Pattern Use Case 1: Observer Pattern

The Observer Pattern is used when an object (subject) maintains a list of its dependents (observers) and notifies them of state changes automatically.

Use Case: Weather Monitoring System

Whenever the weather data changes, the system automatically notifies all subscribed devices.

Code Snippet in Java:

```
// Observer interface

public interface Observer {

    void update(float temperature, float humidity, float pressure);

}


// Concrete Observer (e.g., Phone App)

public class PhoneApp implements Observer {

    public void update(float temperature, float humidity, float pressure) {

        System.out.println("Phone App updated: Temperature = " + temperature);

    }

}


// Subject interface
```

```
public interface Subject {
   void registerObserver(Observer o);
   void removeObserver(Observer o);
   void notifyObservers();
}

// Concrete Subject (Weather Station)
public class WeatherStation implements Subject {
   private List<Observer> observers;
   private float temperature, humidity, pressure;

   public WeatherStation() {
      observers = new ArrayList<>();
   }

   public void setMeasurements(float temperature, float humidity, float pressure) {
      this.temperature = temperature;
      this.humidity = humidity;
      this.pressure = pressure;
      notifyObservers();
   }

   public void registerObserver(Observer o) {
      observers.add(o);
   }
```

```java
    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity, pressure);
        }
    }
}
```

In this case, whenever the `WeatherStation` updates the weather data, all registered observers (like `PhoneApp`) are notified.

---

2. Behavioral Design Pattern Use Case 2: Strategy Pattern

The Strategy Pattern allows a class's behavior to be defined through interchangeable algorithms (strategies).

Use Case: Payment Methods

You have different payment methods (Credit Card, PayPal, etc.) that can be switched without changing the context.

Code Snippet in C#:

csharp

```
// Strategy interface
public interface IPaymentStrategy {
   void Pay(double amount);
}

// Concrete Strategy (CreditCard)
public class CreditCardPayment : IPaymentStrategy {
   public void Pay(double amount) {
      Console.WriteLine("Paid " + amount + " using Credit Card.");
   }
}

// Concrete Strategy (PayPal)
public class PayPalPayment : IPaymentStrategy {
   public void Pay(double amount) {
      Console.WriteLine("Paid " + amount + " using PayPal.");
   }
}

// Context class
public class PaymentContext {
   private IPaymentStrategy paymentStrategy;

   public void SetPaymentStrategy(IPaymentStrategy strategy) {
      paymentStrategy = strategy;
```

```
    }

    public void Pay(double amount) {
        paymentStrategy.Pay(amount);
    }
}
```

You can switch between payment methods (strategies) without changing the context.

---

3. Creational Design Pattern Use Case 1: Singleton Pattern

The Singleton Pattern ensures that a class has only one instance and provides a global point of access to it.

Use Case: Database Connection Manager

You want only one instance of the database connection manager in your system.

Code Snippet in Java:

```java
public class DatabaseConnection {
    private static DatabaseConnection instance;

    private DatabaseConnection() {
        // Private constructor
    }
```

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

    public void connect() {
        System.out.println("Connecting to the database...");
    }
}
```

---

4. Creational Design Pattern Use Case 2: Factory Pattern

The Factory Pattern provides a way to delegate the creation of objects to subclasses.

Use Case: Shape Factory

You want to create different shapes (Circle, Square, etc.) but let a factory decide which one to instantiate.

Code Snippet in C#:

csharp

```
// Product interface
public interface IShape {
```

```
    void Draw();
}

// Concrete Products
public class Circle : IShape {
    public void Draw() {
        Console.WriteLine("Drawing a Circle");
    }
}

public class Square : IShape {
    public void Draw() {
        Console.WriteLine("Drawing a Square");
    }
}

// Factory class
public class ShapeFactory {
    public IShape GetShape(string shapeType) {
        if (shapeType == "Circle") {
            return new Circle();
        } else if (shapeType == "Square") {
            return new Square();
        }
        return null;
```

```java
    }
}
```

The `ShapeFactory` decides which shape to create based on input.

---

5. Structural Design Pattern Use Case 1: Adapter Pattern

The Adapter Pattern is used to make two incompatible interfaces work together.

Use Case: Media Player Adapter

You have an advanced media player and a basic media player interface. The adapter converts one into the other.

Code Snippet in Java:

java

```java
// Target interface
public interface MediaPlayer {
    void play(String audioType, String fileName);
}

// Adapter class
public class MediaAdapter implements MediaPlayer {
    AdvancedMediaPlayer advancedMediaPlayer;

    public MediaAdapter(String audioType) {
```

```
        if (audioType.equalsIgnoreCase("vlc")) {
            advancedMediaPlayer = new VlcPlayer();
        } else if (audioType.equalsIgnoreCase("mp4")) {
            advancedMediaPlayer = new Mp4Player();
        }
    }

    public void play(String audioType, String fileName) {
        if (audioType.equalsIgnoreCase("vlc")) {
            advancedMediaPlayer.playVlc(fileName);
        } else if (audioType.equalsIgnoreCase("mp4")) {
            advancedMediaPlayer.playMp4(fileName);
        }
    }
}
```

This pattern allows different types of media players to work under a common interface.

---

6. Structural Design Pattern Use Case 2: Decorator Pattern

The Decorator Pattern allows behavior to be added to individual objects, dynamically.

Use Case: Coffee Order System

You want to decorate a coffee with different add-ons like milk, sugar, etc.

Code Snippet in C#:

csharp

```
// Component
public abstract class Coffee {
    public abstract double GetCost();
    public abstract string GetDescription();
}

// Concrete Component
public class SimpleCoffee : Coffee {
    public override double GetCost() {
        return 2.00;
    }

    public override string GetDescription() {
        return "Simple Coffee";
    }
}

// Decorator
public abstract class CoffeeDecorator : Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee coffee) {
```

```
        this.decoratedCoffee = coffee;
    }

    public override double GetCost() {
        return decoratedCoffee.GetCost();
    }

    public override string GetDescription() {
        return decoratedCoffee.GetDescription();
    }
}

// Concrete Decorators
public class MilkDecorator : CoffeeDecorator {
    public MilkDecorator(Coffee coffee) : base(coffee) { }

    public override double GetCost() {
        return base.GetCost() + 0.50;
    }

    public override string GetDescription() {
        return base.GetDescription() + ", Milk";
    }
}
```

Exercise 2: Rocket launch simulator

Problem Statement:

You need to simulate a rocket launch in a terminal-based environment, providing real-time updates to the user. The simulator will work in discrete time steps (seconds), simulating the stages of a rocket launch, fuel consumption, altitude gain, and speed changes.

Key Functionalities:

1. Pre-Launch Checks: The user initiates system checks before launch.

2. Launch: The rocket launches, and the system provides real-time updates on fuel, altitude, and speed.

3. Fast Forward: The user can fast-forward the simulation by a specified number of seconds.

4. Stage Separation: As the rocket reaches different stages, updates are provided.

5. Mission Success/Failure: Determine whether the mission is successful (achieving orbit) or a failure (insufficient fuel).

Key Design Patterns:

- Singleton Pattern: Ensure that the mission controller is a single instance managing the entire simulation.

- Observer Pattern (Optional): Could be used for updates to different subsystems (e.g., fuel system, engine system, etc.).

- Command Pattern: Manage different actions (launch, fast forward, etc.) in a flexible way.

Step-by-Step Implementation

1. Singleton Pattern: Mission Controller

The `MissionController` will manage the overall state of the rocket, such as fuel, altitude, speed, and mission progress.

2. Command Pattern: Simulating Launch Steps

Each step in the simulation (like launching, updating altitude, checking fuel) will be encapsulated as a command. This makes it easier to modify the simulation in the future by adding or removing commands.

Code Example

1. Singleton Pattern: MissionController

The `MissionController` manages the overall mission and stores the state of the rocket, including its fuel, altitude, and speed.

Java Example:

java

public class MissionController {

```
private static MissionController instance;

private int fuel;
private int altitude;
private int speed;
private String stage;

private MissionController() {
    this.fuel = 100;  // 100% fuel initially
    this.altitude = 0; // Altitude starts at 0 km
    this.speed = 0;    // Speed starts at 0 km/h
    this.stage = "Pre-Launch";
}

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

public void startChecks() {
    System.out.println("All systems are 'Go' for launch.");
    stage = "Ready for Launch";
}
```

```
public void launch() {
    if (stage.equals("Ready for Launch")) {
        stage = "Stage 1";
        speed = 1000;
        altitude = 10;
        fuel -= 10;
        System.out.println("Rocket has launched! Stage: " + stage);
        updateStatus();
    } else {
        System.out.println("Pre-launch checks not completed. Cannot launch.");
    }
}

public void fastForward(int seconds) {
    for (int i = 0; i < seconds; i++) {
        update();
    }
}

private void update() {
    if (fuel <= 0) {
        System.out.println("Mission Failed due to insufficient fuel.");
        return;
    }

    altitude += 10;     // Simulating altitude gain
```

```
        speed += 500;       // Simulating speed increase
        fuel -= 5;          // Simulating fuel consumption

        if (altitude >= 100 && stage.equals("Stage 1")) {
            stage = "Stage 2";
            System.out.println("Stage 1 complete. Separating stage. Entering Stage 2.");
        }

        if (altitude >= 200) {
            stage = "Orbit";
            System.out.println("Orbit achieved! Mission Successful.");
            return;
        }

        updateStatus();
    }

    private void updateStatus() {
        System.out.println("Stage: " + stage + ", Fuel: " + fuel + "%, Altitude: " + altitude + " km, Speed: " +
speed + " km/h");
    }
}
```

In this class:

- The `MissionController` is a singleton that manages the rocket's state.

- The `startChecks` method initiates the system checks.

- The `launch` method simulates the launch and updates the altitude, fuel, and speed.

- The `fastForward` method skips a specified number of seconds, updating the rocket's state for each second.

---

2. Main Program: Interacting with the Simulation

The main program allows the user to interact with the simulation, triggering launch, fast-forwarding, and checking mission status.

Java Example:

java

```
import java.util.Scanner;

public class RocketLaunchSimulator {
    public static void main(String[] args) {
        MissionController controller = MissionController.getInstance();
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("Enter command (start_checks, launch, fast_forward [seconds], exit): ");
            String input = scanner.nextLine();
            String[] command = input.split(" ");

            switch (command[0]) {
                case "start_checks":
```

```java
                controller.startChecks();
                break;
            case "launch":
                controller.launch();
                break;
            case "fast_forward":
                if (command.length > 1) {
                    int seconds = Integer.parseInt(command[1]);
                    controller.fastForward(seconds);
                } else {
                    System.out.println("Please provide the number of seconds to fast forward.");
                }
                break;
            case "exit":
                System.out.println("Exiting the simulation.");
                return;
            default:
                System.out.println("Unknown command.");
        }
      }
   }
}
```

How the Simulator Works:

1. Pre-Launch:

- The user types `start_checks`, and the system responds with all systems being "Go" for launch.

2. Launch:

   - When the user types `launch`, the rocket lifts off, and the simulation begins updating the altitude, speed, and fuel consumption.

3. Fast Forward:

   - The user can type `fast_forward X` to skip X seconds of the mission, and the state updates accordingly.

4. Stage Separation:

   - After reaching a certain altitude, the system updates the stage to "Stage 2" and eventually reaches "Orbit".

5. Mission Success/Failure:

   - If the rocket achieves orbit, the mission is declared a success. If fuel runs out before reaching orbit, the mission fails.

Optional Enhancements:

1. Detailed Fuel and Speed Calculations: You could introduce more realistic fuel consumption and speed models based on the mass of the rocket and air resistance.

2. Multiple Stages: Add more stages beyond "Stage 2" to simulate a more complex rocket.

3. Error Handling: Introduce checks for edge cases, like launching without sufficient fuel or fast-forwarding past the mission duration.