Assignment

Het Pokar (21BCP223, 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);
    }
```

```
    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:

```
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;
    }
}
```

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
// 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: Real-time Chat Application

Problem Statement:

We need to build a console-based chat application where users can:

1. Create or join chat rooms by entering a unique room ID.

2. Send and receive messages in real-time within a chat room.

3. Display a list of active users in the chat room.

4. Optionally, add private messaging and message history.

Key Focus:

- Behavioral Pattern (Observer): Notify clients of new messages or user activities.

- Creational Pattern (Singleton): Manage the state of the chat rooms.

- Structural Pattern (Adapter): Allow the system to work with different types of client communication protocols (WebSocket, HTTP, etc.).

Proposed Structure:

1. Chat Room (Singleton)

   - Ensures that only one instance of a chat room exists per room ID.

   - Manages users and messages in the room.

2. User (Observer)

   - Each user is an observer who receives updates (messages) from the chat room they are subscribed to.

3. Message Dispatcher (Command/Observer)

   - A central system that handles the sending and receiving of messages, ensuring real-time communication.

Step-by-Step Implementation

1. Singleton Pattern: Chat Room Manager

The `ChatRoomManager` will handle all chat rooms, ensuring that only one instance of each room is created.

Java Example:

java

```java
import java.util.HashMap;

import java.util.Map;


public class ChatRoomManager {

    private static ChatRoomManager instance;

    private Map<String, ChatRoom> chatRooms;


    private ChatRoomManager() {

        chatRooms = new HashMap<>();

    }


    public static ChatRoomManager getInstance() {

        if (instance == null) {

            instance = new ChatRoomManager();

        }

        return instance;
```

```java
    }

    public ChatRoom getChatRoom(String roomId) {
        return chatRooms.computeIfAbsent(roomId, k -> new ChatRoom(roomId));
    }

    public void removeChatRoom(String roomId) {
        chatRooms.remove(roomId);
    }
}
```

This ensures that the `ChatRoomManager` is a singleton and only one instance of it manages all chat rooms.

2. Observer Pattern: User Subscription

Each user acts as an observer that subscribes to a chat room and gets notified when a message is sent.

Java Example:

java

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

public class ChatRoom {
    private String roomId;
    private List<User> users;
    private List<String> messages;
```

```
public ChatRoom(String roomId) {
    this.roomId = roomId;
    users = new ArrayList<>();
    messages = new ArrayList<>();
}

public void joinRoom(User user) {
    users.add(user);
    notifyUsers(user.getUsername() + " has joined the room.");
}

public void leaveRoom(User user) {
    users.remove(user);
    notifyUsers(user.getUsername() + " has left the room.");
}

public void sendMessage(String message, User user) {
    messages.add(user.getUsername() + ": " + message);
    notifyUsers(user.getUsername() + ": " + message);
}

private void notifyUsers(String message) {
    for (User user : users) {
        user.receiveMessage(message);
    }
}
```

}

The `ChatRoom` class notifies all its users whenever a message is received. Users are observers who receive these notifications.

3. User Class: Observer Implementation

Each user is an observer who subscribes to a `ChatRoom` and listens for new messages.

Java Example:

java

```
public class User {
    private String username;

    public User(String username) {
        this.username = username;
    }

    public String getUsername() {
        return username;
    }

    public void receiveMessage(String message) {
        System.out.println(username + " received: " + message);
    }

    public void sendMessage(String message, ChatRoom chatRoom) {
        chatRoom.sendMessage(message, this);
```

```
    }
}
```

The `User` class has methods to send and receive messages in the `ChatRoom`.

4. Main Program: Real-time Interaction

This will simulate the user interaction in a terminal-based application.

Java Example:

```java
import java.util.Scanner;

public class ChatApplication {
    public static void main(String[] args) {
        ChatRoomManager manager = ChatRoomManager.getInstance();
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter your username: ");
        String username = scanner.nextLine();
        User user = new User(username);

        System.out.println("Enter the chat room ID to join or create: ");
        String roomId = scanner.nextLine();
        ChatRoom room = manager.getChatRoom(roomId);

        room.joinRoom(user);
```

```
        while (true) {
            System.out.println("Enter message (or 'exit' to leave): ");
            String message = scanner.nextLine();
            if (message.equalsIgnoreCase("exit")) {
                room.leaveRoom(user);
                break;
            }
            user.sendMessage(message, room);
        }
    }
}
```

This main program allows a user to join or create a chat room, send messages, and receive messages in real-time.

Optional Enhancements:

1. Private Messaging: Add logic to send messages directly to a specific user without broadcasting it to the whole chat room.

2. Message History: Store message history so users can see previous messages when they join a room.

3. Adapter Pattern (Optional): If you plan to extend the system to work with other communication protocols (e.g., WebSocket), you can use the Adapter pattern.

Key Concepts in Action:

- Singleton Pattern ensures there's only one `ChatRoomManager` that manages chat rooms globally.

- Observer Pattern enables real-time message delivery to all users in the chat room.

- Encapsulation and OOP principles ensure modularity and flexibility.