**Khushi Desai - 21BCP264**

**Computer Engineering**

**Pandit Deendayal Energy University**

**Educational Initiatives**

**Exercise 1:**

**Problem Statement on Design patterns**

* **Behavioral Design Patterns:**

|  |  |
| --- | --- |
| **1.** | **Observer Pattern: Weather Station** |
| UseCase: | A weather station that notifies various displays (like current conditions, statistics, and forecast) when there are changes in weather data. |
| Subject: | WeatherData |
| Observer: | CurrentConditionsDisplay, StatisticsDisplay, ForecastDisplay |
|  | import java.util.ArrayList;  import java.util.List;  // Subject  class WeatherData {      private List<Observer> observers;      private float temperature;      public WeatherData() {          observers = new ArrayList<>();      }      public void registerObserver(Observer observer) {          observers.add(observer);      }      public void removeObserver(Observer observer) {          observers.remove(observer);      }      public void notifyObservers() {          for (Observer observer : observers) {              observer.update(temperature);          }      }      public void setTemperature(float temperature) {          this.temperature = temperature;          notifyObservers();      }  }  // Observer  interface Observer {      void update(float temperature);  }  // Concrete Observer  class CurrentConditionsDisplay implements Observer {      public void update(float temperature) {          System.out.println("Current temperature: " + temperature);      }  } |
| **2.** | **Strategy Pattern: Payment Processing** |
| Use Case: | An e-commerce application that allows users to choose different payment methods (CreditCard, PayPal, etc.). |
| Context: | ShoppingCart |
| Strategy: | PaymentStrategy |
|  | // Strategy Interface  interface PaymentStrategy {      void pay(int amount);  }  // Concrete Strategies  class CreditCardPayment implements PaymentStrategy {      public void pay(int amount) {          System.out.println("Paid " + amount + " using Credit Card.");      }  }  class PayPalPayment implements PaymentStrategy {      public void pay(int amount) {          System.out.println("Paid " + amount + " using PayPal.");      }  }  // Context  class ShoppingCart {      private PaymentStrategy paymentStrategy;      public void setPaymentStrategy(PaymentStrategy paymentStrategy) {          this.paymentStrategy = paymentStrategy;      }      public void checkout(int amount) {          paymentStrategy.pay(amount);      }  } |

* **Creational Design Patterns**

|  |  |
| --- | --- |
| **3.** | **Singleton Pattern: Database Connection** |
| UseCase: | A class that manages the database connection ensuring only one instance exists. |
|  | class DatabaseConnection {      private static DatabaseConnection instance = null;      private DatabaseConnection() {          // private constructor      }      public static DatabaseConnection getInstance() {          if (instance == null) {              instance = new DatabaseConnection();          }          return instance;      }      public void connect() {          System.out.println("Connected to database");      }  } |
| **4.** | **Factory Method Pattern: Vehicle Creation** |
| Use Case: | An application that creates different types of vehicles (Car, Bike) through a factory method. |
| Product: | Vehicle |
| Concrete Products: | Car, Bike |
| Creator: | VehicleFactory |
|  | // Product  abstract class Vehicle {      public abstract void drive();  }  // Concrete Products  class Car extends Vehicle {      public void drive() {          System.out.println("Driving a car.");      }  }  class Bike extends Vehicle {      public void drive() {          System.out.println("Riding a bike.");      }  }  // Creator  abstract class VehicleFactory {      public abstract Vehicle createVehicle();  }  // Concrete Creator  class CarFactory extends VehicleFactory {      public Vehicle createVehicle() {          return new Car();      }  }  class BikeFactory extends VehicleFactory {      public Vehicle createVehicle() {          return new Bike();      }  } |

* **Structural Design Patterns:**

|  |  |
| --- | --- |
| 5. | Adapter Pattern: Media Player |
| Use Case: | A media player that can play different formats (like MP3 and VLC) using an adapter to convert them into a common interface. |
| Target: | MediaPlayer |
| Adaptee: | VlcPlayer, Mp3Player |
| Adapter: | MediaAdapter |
|  | // Target  interface MediaPlayer {      void play(String audioType, String fileName);  }  // Adaptee  class VlcPlayer {      public void playVlc(String fileName) {          System.out.println("Playing VLC file: " + fileName);      }  }  class Mp3Player {      public void playMp3(String fileName) {          System.out.println("Playing MP3 file: " + fileName);      }  }  // Adapter  class MediaAdapter implements MediaPlayer {      private VlcPlayer vlcPlayer;      private Mp3Player mp3Player;      public MediaAdapter(String audioType) {          if (audioType.equalsIgnoreCase("vlc")) {              vlcPlayer = new VlcPlayer();          } else if (audioType.equalsIgnoreCase("mp3")) {              mp3Player = new Mp3Player();          }      }      public void play(String audioType, String fileName) {          if (audioType.equalsIgnoreCase("vlc")) {              vlcPlayer.playVlc(fileName);          } else if (audioType.equalsIgnoreCase("mp3")) {              mp3Player.playMp3(fileName);          }      }  } |
| 6. | Composite Pattern: Graphic Editor |
| Use Case: | A graphic editor that allows for grouping of shapes (like Circle, Rectangle) into a composite shape. |
| Component: | Shape |
| Leaf: | Circle, Rectangle |
| Composite: | CompositeShape |
|  | // Component  import java.util.ArrayList;  import java.util.List;  interface Shape {      void draw();  }  // Leaf  class Circle implements Shape {      public void draw() {          System.out.println("Drawing a Circle.");      }  }  class Rectangle implements Shape {      public void draw() {          System.out.println("Drawing a Rectangle.");      }  }  // Composite  class CompositeShape implements Shape {      private List<Shape> shapes = new ArrayList<>();      public void add(Shape shape) {          shapes.add(shape);      }      public void draw() {          for (Shape shape : shapes) {              shape.draw();          }      }  } |

* **Summary:**
* **Behavioral Patterns**: Observer (Weather Station), Strategy (Payment Processing)
* **Creational Patterns**: Singleton (Database Connection), Factory Method (Vehicle Creation)
* **Structural Patterns**: Adapter (Media Player), Composite (Graphic Editor)