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 pattern.

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

3. Two use cases to demonstrate two structural design pattern.  
  
**Program:  
1. Behavioral : Strategy Pattern (Sorting Algorithms)**

// SortStrategy.java

public interface SortStrategy {

void sort(int[] numbers);

}

// QuickSort.java

import java.util.Arrays;

public class QuickSort implements SortStrategy {

@Override

public void sort(int[] numbers) {

Arrays.sort(numbers); // shortcut

System.out.println("QuickSort: " + Arrays.toString(numbers));

}

}

// BubbleSort.java

import java.util.Arrays;

public class BubbleSort implements SortStrategy {

@Override

public void sort(int[] numbers) {

for(int i=0;i<numbers.length-1;i++)

for(int j=0;j<numbers.length-i-1;j++)

if(numbers[j] > numbers[j+1]){

int t=numbers[j]; numbers[j]=numbers[j+1]; numbers[j+1]=t;

}

System.out.println("BubbleSort: " + Arrays.toString(numbers));

}

}

// StrategyDemo.java

public class StrategyDemo {

public static void main(String[] args) {

int[] arr = {5, 3, 8, 1};

SortStrategy quick = new QuickSort();

SortStrategy bubble = new BubbleSort();

quick.sort(arr.clone());

bubble.sort(arr.clone());

}

}

It Shows **Strategy Pattern**: interchangeable algorithms (QuickSort, BubbleSort).

**2. Behavioral : Observer Pattern (Weather Station)**

// Observer.java

public interface Observer {

void update(float temp);

}

// PhoneDisplay.java

public class PhoneDisplay implements Observer {

@Override

public void update(float temp) {

System.out.println("Phone shows temperature: " + temp);

}

}

// WeatherStation.java

import java.util.\*;

public class WeatherStation {

private List<Observer> observers = new ArrayList<>();

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

public void setTemperature(float temp){

for(Observer o : observers) o.update(temp);

}

}

// ObserverDemo.java

public class ObserverDemo {

public static void main(String[] args) {

WeatherStation ws = new WeatherStation();

ws.addObserver(new PhoneDisplay());

ws.setTemperature(32.5f);

}

}

It Shows **Observer Pattern**: WeatherStation notifies observers.

**3. Creational: Factory Pattern (Notifications)**

// Notification.java

public interface Notification { void notifyUser(); }

// EmailNotification.java

public class EmailNotification implements Notification {

public void notifyUser(){ System.out.println("Sending Email Notification"); }

}

// SMSNotification.java

public class SMSNotification implements Notification {

public void notifyUser(){ System.out.println("Sending SMS Notification"); }

}

// NotificationFactory.java

public class NotificationFactory {

public Notification createNotification(String type){

switch(type.toLowerCase()){

case "email": return new EmailNotification();

case "sms": return new SMSNotification();

default: throw new IllegalArgumentException("Unknown type");

}

}

}

// FactoryDemo.java

public class FactoryDemo {

public static void main(String[] args){

NotificationFactory factory = new NotificationFactory();

Notification n = factory.createNotification("email");

n.notifyUser();

}

}

It Shows **Factory Pattern**: centralized creation logic.

**4. Creational: Singleton Pattern (Database Connection)**

// Database.java

public class Database {

private static Database instance;

private Database() {}

public static synchronized Database getInstance(){

if(instance == null) instance = new Database();

return instance;

}

public void query(String sql){

System.out.println("Executing query: " + sql);

}

}

// SingletonDemo.java

public class SingletonDemo {

public static void main(String[] args){

Database db1 = Database.getInstance();

Database db2 = Database.getInstance();

db1.query("SELECT \* FROM users");

System.out.println("Same instance? " + (db1 == db2));

}

}

It Shows **Singleton Pattern**: single DB instance reused.

**5. Structural: Adapter Pattern (Charger Adapter)**

// Charger.java

public interface Charger { void charge(); }

// TypeCCharger.java

public class TypeCCharger implements Charger {

public void charge(){ System.out.println("Charging with Type-C Charger"); }

}

// MicroUSBCharger.java

public class MicroUSBCharger {

public void microUsbCharge(){ System.out.println("Charging with Micro-USB Charger"); }

}

// ChargerAdapter.java

public class ChargerAdapter implements Charger {

private MicroUSBCharger micro;

public ChargerAdapter(MicroUSBCharger m){ this.micro = m; }

public void charge(){ micro.microUsbCharge(); }

}

// AdapterDemo.java

public class AdapterDemo {

public static void main(String[] args){

Charger typec = new TypeCCharger();

Charger adapter = new ChargerAdapter(new MicroUSBCharger());

typec.charge();

adapter.charge();

}

}

It Shows **Adapter Pattern**: adapts MicroUSB to Type-C interface.

**6. Structural: Composite Pattern (File System)**

// FileSystem.java

public interface FileSystem { void show(); }

// FileLeaf.java

public class FileLeaf implements FileSystem {

private String name;

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

public void show(){ System.out.println("File: " + name); }

}

// Folder.java

import java.util.\*;

public class Folder implements FileSystem {

private String name;

private List<FileSystem> children = new ArrayList<>();

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

public void add(FileSystem fs){ children.add(fs); }

public void show(){

System.out.println("Folder: " + name);

for(FileSystem fs : children) fs.show();

}

}

// CompositeDemo.java

public class CompositeDemo {

public static void main(String[] args){

Folder root = new Folder("Root");

root.add(new FileLeaf("file1.txt"));

root.add(new FileLeaf("file2.txt"));

Folder sub = new Folder("SubFolder");

sub.add(new FileLeaf("subfile.txt"));

root.add(sub);

root.show();

}

}