**Name: Ravi kiran C**

**Register Number: 21ETCS002112**

**Email Id:** [**kiranchikkappaiah@gmail.com**](mailto:kiranchikkappaiah@gmail.com)

**College: M S Ramiah University of Applied Sciences, Bangalore.**

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.

**Two use cases to demonstrate two behavioural design Pattern.**

**1. Template Method Pattern.**

**Use Case: Types of Payment**

Java

public abstract class PaymentFlow{

public abstract void validateRequest();

public abstract void calculatefees();

public abstract void debitAmount();

public abstract void creditAmount();

public final void sendMoney(){

validateRequest();

debitAmount();

calculatefees();

creditAmount();

}

}

public class PayToFriend extends PaymentFlow{

@Override

public void validateRequest(){

System.out.println("Validate logic of PayToFriend");

}

@Override

public void debitAmount(){

System.out.println("Debit the amount logic of PayToFriend");

}

@Override

public void calculatefees(){

System.out.println("0% fees charged");

}

@Override

public void creditAmount(){

System.out.println("Credit the full amount");

}

}

public class PayToMerchantFlow extends PaymentFlow{

@Override

public void validateRequest(){

System.out.println("Validate logic of PayToMerchantFlow");

}

@Override

public void debitAmount(){

System.out.println("Debit the amount logic of PayToMerchantFlow");

}

@Override

public void calculatefees(){

System.out.println("2% fees charged");

}

@Override

public void creditAmount(){

System.out.println("Credit the Remaining amount");

}

}

**2. Visitor Pattern.**

**Use Case: Types of Rooms**

java

// Element

public interface RoomElement {

public void accept(RoomVisiter visitor);

}

public class singleroom implements RoomElement{

public int roomPrice = 0;

@Override

public void accept(RoomVisiter visitor){

visitor.visit(singleroomObj:this);

}

}

public class doubleroom implements RoomElement{

public int roomPrice = 0;

@Override

public void accept(RoomVisiter visitor){

visitor.visit(doubleroomObj:this);

}

}

public class deluxeroom implements RoomElement{

public int roomPrice = 0;

@Override

public void accept(RoomVisiter visitor){

visitor.visit(deluxeroomObj:this);

}

}

// Visitor

public interface RoomVisitor{

public void visit(SingleRoom singleRoomObj);

public void visit(DoubleRoom doubleRoomObj);

public void visit(DeluxeRoom deluxeRoomObj);

}

public class RoomPricingVisitor implements RoomVisitor{

@Override

public void visit(SingleRoom singleRoomObj){

System.out.println("Pricing computatio logic of SingleRoom");

singleRoomObj.roomPrice = 1000;

}

@Override

public void visit(DoubleRoom DoubleRoomObj){

System.out.println("Pricing computatio logic of DoubleRoom");

DoubleRoomObj.roomPrice = 2000;

}

@Override

public void visit(DeluxeRoom DeluxeRoomObj){

System.out.println("Pricing computatio logic of DeluxeRoom");

DeluxeRoomObj.roomPrice = 5000;

}

}

public class RoomMaintenanceVisitor implements RoomVisitor{

@Override

public void visit(SingleRoom singleRoomObj){

System.out.println("Performing maintenance of SingleRoom");

}

@Override

public void visit(DoubleRoom DoubleRoomObj){

System.out.println("Performing maintenance of DoubleRoom");

}

@Override

public void visit(DeluxeRoom DeluxeRoomObj){

System.out.println("Performing maintenance of DeluxeRoom");

}

}

**Two use cases to demonstrate two creational design Pattern.**

**1.** **Prototype**

**Use Case: Student details.**

Java

public interface Prototype{

Prototype clone();

}

public class Student implements Prototype{

int age;

private int rollNumber;

String name

Student(){

}

Student(int age,int rollNumber,String name){

this.age = age;

this.rollNumber = rollNumber;

this.name = name;

}

@Override

public Prototype clone{

return new Student(age,rollNumber,name);

}

}

public class Main{

public static void public static void main (String[] args) {

Student obj = new Student(age:20,rollNumber:75,name:"Ram");

Student cloneObj = (Student) obj.clone();

}

}

**2. Factory Method Pattern**

**Use Case: Different Shapes**

Java

public interface Shape{

public void computerarea();

}

public class Square implements Shape{

@Override

public void computerarea(){

System.out.println("Computer Square Area");

}

}

public class Circle implements Shape{

@Override

public void computerarea(){

System.out.println("Computer Circle Area");

}

}

public class shapeInstanceFactory{

public shape getShapeInstance(String value){

if(value.equals("Circle")){

return new Circle();

}

else(value.equals("Square")){

return new Square();

}

return null;

}

public class Main{

public static void main (String[] args) {

shapeInstanceFactory factoryObj = new shapeInstanceFactory();

shape CircleObj = factoryObj.getShapeInstance(value:"Circle");

CircleObj.computerarea();

}

**Two use cases to demonstrate two structural design Pattern.**

**1.Composite Pattern.**

**Use case: File System.**

Java

#Component

public interface fileSystem{

Is();

}

#leaf

public class file implements fileSystem{

void Is(){

//print filename here

}

}

#Composite

public class directory implements fileSystem{

void add(fileSystem fs){

fileSystemList.add(fs);

}

void Is(){

for(fileSystem fsObj:fileSystemList){

fsObj.Is();

}

}

}

public class Main{

public static void main (String[] args) {

directory parentDir = new directory();

fileSystem fileObj1 = new file();

parentDir.add(fileObj1);

directory childDir = new directory();

fileSystem fileObj2 = new file();

childDir.add(fileObj2);

parentDir.add(childDir);

parentDir.Is();

}

}

**2.Bridge Pattern.**

**Use case: Living things.**

Java

public interface BreatheImplementor{

void breatheProcess();

}

public abstract class livingThings{

BreatheImplementor breatheImplementor;

livingThings(BreatheImplementor breatheImplementor){

this.breatheImplementor = breatheImplementor;

}

abstract void breatheProcess();

}

public class Dog extends livingThings{

Dog(BreatheImplementor breatheobj){

super(breatheobj);

}

void breatheProcess(){

breatheImplementor.breatheProcess();

}

}

public class Fish extends livingThings{

Fish(BreatheImplementor breatheobj){

super(breatheobj);

}

void breatheProcess(){

breatheImplementor.breatheProcess();

}

}

public class Tree extends livingThings{

Tree(BreatheImplementor breatheobj){

super(breatheobj);

}

void breatheProcess(){

breatheImplementor.breatheProcess();

}

}

public class Main{

public static void main (String[] args) {

livingThings Fishobj = new Fish(new WaterBreath());

Fishobj.breatheProcess();

}

}