**Lab 6**

**Bridge Design Pattern**

Task1:-

Step 1: Create the color and shape interfaces/

public interface Color {

public void applyColor();

}

public abstract class Shape {

//Composition - implementor

protected Color color;

//constructor with implementor as input argument

public Shape(Color c){

this.color=c;

}

abstract public void applyColor();

}

Step 2: Create concrete implementations of shape class.

public class Triangle extends Shape{

public Triangle(Color c) {

super(c);

}

@Override

public void applyColor() {

System.out.print("Triangle filled with color ");

color.applyColor();

}

}

public class Pentagon extends Shape{

public Pentagon(Color c) {

super(c);

}

@Override

public void applyColor() {

System.out.print("Pentagon filled with color ");

color.applyColor();

} }

Step 3: Create concrete implementations for Color interface:

public class RedColor implements Color{

public void applyColor(){

System.out.println("red.");

}

}

public class GreenColor implements Color{

public void applyColor(){

System.out.println("green.");

}

}

Step 4: Create the client code

public class BridgePatternTest {

public static void main(String[] args) {

Shape tri = new Triangle(new RedColor());

tri.applyColor();

Shape pent = new Pentagon(new GreenColor());

pent.applyColor();

}

}

Task2:-

MainClass.java

public class MainClass {

public static void main(String[] args) {

ThreadSchedular obj=new PreemtiveThreadSchedular(new windowsPts());

obj.preemtiveschedular();

ThreadSchedular obj1=new TimedSliceSchedular(new windowsTsts());

obj1.timesliceschedular();

ThreadSchedular obj2=new PreemtiveThreadSchedular(new UnixPts());

Obj2.preemtiveschedular();

ThreadSchedular obj3=new TimedSliceSchedular(new UnixTsts());

Obj3.timesliceschedular();

}

}

PreemtiveThreadSchedular.java

public class PreemtiveThreadSchedular extends ThreadcShedular {

PreemtiveThreadSchedular(Thread c) {

super(c);

}

@Override

void preemtiveschedular() {

System.out.println("Preemtive Schedular");

}

@Override

void timesliceschedular() {

}

}

Thread.java

public interface Thread {

public void preemtiveschedular();

public void timesliceschedular();

}

ThreadSchedular.java

public abstract class ThreadSchedular {

protected Thread thd;

ThreadSchedular(Thread c){

this.thd=c;

}

abstract void preemtiveschedular();

abstract void timesliceschedular();

}

TimedSliceSchedular.java

public class TimedSliceSchedular extends ThreadSchedular{

public TimedSliceSchedular(Thread c)

{

super(c);

}

@Override

void preemtiveschedular() {

}

@Override

void timesliceschedular() {

System.out.println("TimeSlice Schedular");

}

}

WindowsPts.java

public class windowspts implements Thread {

public void categorytype() {

}

@Override

public void preemtiveschedular() {

System.out.println("Preemtive Schedular");

}

@Override

public void timesliceschedular() {

}

}

WindowsTsts.java

public class windowsTsts implements Thread{

public void categorytype() {

}

@Override

public void preemtiveschedular() {

}

@Override

public void timesliceschedular() {

System.out.println("TimeSlice Schedular");

}

}

UnixPts.java

public class UnixPts implements Thread {

public void categorytype() {

}

@Override

public void preemtiveschedular() {

System.out.println("Preemtive Thread Schedular");

}

@Override

public void timesliceschedular() {

}

}

UnixTsts.java

public class UnixsT sts implements Thread{

public void categorytype() {

}

@Override

public void preemtiveschedular() {

}

@Override

public void timesliceschedular() {

System.out.println("TimeSlice Schedular");

}

}

**Template Pattern**

Step 1: Create an abstract class with a template method being final.

public abstract class Game {

abstract void initialize();

abstract void startPlay();

abstract void endPlay();

//template method

public final void play(){

//initialize the game

initialize();

//start game

startPlay();

//end game

endPlay();

}

}

Step 2: Create concrete classes extending the above class.

public class Cricket extends Game {

@Override

void endPlay() {

System.out.println("Cricket Game Finished!");

}

@Override void initialize() { System.out.println("Cricket Game Initialized! Start playing.");

}

@Override

void startPlay() {

System.out.println("Cricket Game Started. Enjoy the game!");

}

}

public class Football extends Game {

@Override

void endPlay() { System.out.println("Football Game Finished!");

}

@Override

void initialize() {

System.out.println("Football Game Initialized! Start playing.");

}

@Override

void startPlay() {

System.out.println("Football Game Started. Enjoy the game!");

}

}

Step 3: Use the *Game*'s template method play() to demonstrate a defined way of playing game.

public class TemplatePatternDemo {

public static void main(String[] args) { Game game = new Cricket();

game.play();

System.out.println();

game = new Football();

game.play();

}

}

**Chain of Responsibility Pattern**

Step 1: Create a class Currency that will store the amount to dispense and used by the chain implementations.

public class Currency {

private int amount;

public Currency(int amt){

this.amount=amt;

}

public int getAmount(){

return this.amount;

}

}

Step 2: create the base interface which should have a method to define the next processor in the chain and the method that will process the request.

public interface DispenseChain {

void setNextChain(DispenseChain nextChain);

void dispense(Currency cur);

}

Step 3: create different processor classes that will implement the DispenseChain interface and provide implementation of dispense methods.

public class Dollar50Dispenser implements DispenseChain {

private DispenseChain chain;

@Override

public void setNextChain(DispenseChain nextChain) {

this.chain=nextChain;

}

@Override

public void dispense(Currency cur) {

if(cur.getAmount() >= 50){

int num = cur.getAmount()/50;

int remainder = cur.getAmount() % 50;

System.out.println("Dispensing "+num+" 50$ note");

if(remainder !=0) this.chain.dispense(new Currency(remainder));

}else{

this.chain.dispense(cur);

}

}

}

Step 4: Create the chain

import java.util.Scanner;

public class ATMDispenseChain {

public ATMDispenseChain() {

// initialize the chain

private DispenseChain c1= new Dollar50Dispenser();

DispenseChain c2 = new Dollar20Dispenser();

DispenseChain c3 = new Dollar10Dispenser();

// set the chain of responsibility

c1.setNextChain(c2);

c2.setNextChain(c3);

}

public static void main(String[] args) {

ATMDispenseChain atmDispenser = new ATMDispenseChain();

while (true) {

int amount = 0;

System.out.println("Enter amount to dispense");

Scanner input = new Scanner(System.in);

amount = input.nextInt();

if (amount % 10 != 0) {

System.out.println("Amount should be in multiple of 10s.");

return;

}

// process the request

atmDispenser.c1.dispense(new Currency(amount));

}

}

**Command Design Pattern**

Step 1: Create a command interface.

public interface Order {

void execute();

}

Step 2: Create a request class.

public class Stock {

private String name = "ABC";

private int quantity = 10;

public void buy(){

System.out.println("Stock [ Name: "+name+",Quantity: " + quantity +" ] bought"); }

public void sell(){ System.out.println("Stock [ Name: "+name+", Quantity: " + quantity +" ] sold"); } }

Step 3: Create concrete classes implementing the Order interface.

public class BuyStock implements Order { private Stock abcStock;

public BuyStock(Stock abcStock){ this.abcStock = abcStock;

}

public void execute() {

abcStock.buy();

}

}

public class SellStock implements Order { private Stock abcStock;

public SellStock(Stock abcStock){ this.abcStock = abcStock;

}

public void execute() {

abcStock.sell();

}

}

Step 4: Create command invoker class.

import java.util.ArrayList;

import java.util.List;

public class Broker {

private List<Order> orderList = new ArrayList<Order>();

public void takeOrder(Order order){

orderList.add(order);

}

public void placeOrders(){

for (Order order : orderList) { order.execute();

}

orderList.clear();

}

}

Step 5 : Use the Broker class to take and execute commands.

public class CommandPatternDemo {

public static void main(String[] args) {

Stock abcStock = new Stock();

BuyStock buyStockOrder = new BuyStock(abcStock);

SellStock sellStockOrder = new SellStock(abcStock);

Broker broker = new Broker(); broker.takeOrder(buyStockOrder); broker.takeOrder(sellStockOrder); broker.placeOrders();

}

}