**Lab Assignment 7**

**Template Design Pattern:**

HouseTemplate.java

package com.journaldev.design.template;

public abstract class HouseTemplate {

//template method, final so subclasses can't override

public final void buildHouse(){

buildFoundation();

buildPillars();

buildWalls();

buildWindows();

System.out.println("House is built.");

}

//default implementation

private void buildWindows() {

System.out.println("Building Glass Windows");

}

//methods to be implemented by subclasses

public abstract void buildWalls();

public abstract void buildPillars();

private void buildFoundation() {

System.out.println("Building foundation with cement,iron rods and sand");

}

}

**BuildHouse is the template method and defines the order of execution for performing several steps.**

**Template Method Concrete Classes**

**We can have different type of houses, such as Wooden House and Glass House.**

**WoodenHouse.java**

package com.journaldev.design.template;

public class WoodenHouse extends HouseTemplate {

@Override

public void buildWalls() {

System.out.println("Building Wooden Walls");

}

@Override

public void buildPillars() {

System.out.println("Building Pillars with Wood coating");

}}

**We could have overridden other methods also, but for simplicity I am not doing that.**

**GlassHouse.java**

package com.journaldev.design.template;

public class GlassHouse extends HouseTemplate {

@Override

public void buildWalls() {

System.out.println("Building Glass Walls");

}

@Override

public void buildPillars() {

System.out.println("Building Pillars with glass coating");

}

}

**Template Method Design Pattern Client Let’s test our template method pattern example with a test program.**

HousingClient.java

package com.journaldev.design.template;

public class HousingClient {

public static void main(String[] args) {

HouseTemplate houseType = new WoodenHouse();

//using template method

houseType.buildHouse();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*");

houseType = new GlassHouse();

houseType.buildHouse();

}

}

**Chain of responsibility Design pattern:**

interface Chain

{

    public abstract void setNext(Chain nextInChain);

    public abstract void process(Number request);

}

class Number

{

    private int number;

    public Number(int number)

    {

        this.number = number;

    }

    public int getNumber()

    {

        return number;

    }

}

class NegativeProcessor implements Chain

{

    private Chain nextInChain;

    public void setNext(Chain c)

    {

        nextInChain = c;

    }

    public void process(Number request)

    {

        if (request.getNumber() < 0)

        {

            System.out.println("NegativeProcessor : " + request.getNumber());

        }

        else

        {

            nextInChain.process(request);

        }

    }

}

class ZeroProcessor implements Chain

{

    private Chain nextInChain;

    public void setNext(Chain c)

    {

        nextInChain = c;

    }

    public void process(Number request)

    {

        if (request.getNumber() == 0)

        {

            System.out.println("ZeroProcessor : " + request.getNumber());

        }

        else

        {

            nextInChain.process(request);

        }

    }

}

class PositiveProcessor implements Chain

{

    private Chain nextInChain;

    public void setNext(Chain c)

    {

        nextInChain = c;

    }

    public void process(Number request)

    {

        if (request.getNumber() > 0)

        {

            System.out.println("PositiveProcessor : " + request.getNumber());

        }

        else

        {

            nextInChain.process(request);

        }

    }

}

class TestChain

{

    public static void main(String[] args) {

        //configure Chain of Responsibility

        Chain c1 = new NegativeProcessor();

        Chain c2 = new ZeroProcessor();

        Chain c3 = new PositiveProcessor();

        c1.setNext(c2);

        c2.setNext(c3);

        //calling chain of responsibilities

        c1.process(new Number(90));

        c1.process(new Number(-50));

        c1.process(new Number(0));

    }

}

**Command Design Pattern**

**Command**

public interface Command{

public void execute();

}

Now let's create two concrete commands. One will turn on the lights, another turns off lights:

**Concrete Command**

public class LightOnCommand implements Command{

//reference to the light

Light light;

public LightOnCommand(Light light){

this.light = light;

}

public void execute(){

light.switchOn();

}

}

**Concrete Command**

public class LightOffCommand implements Command{

//reference to the light

Light light;

public LightOffCommand(Light light){

this.light = light;

}

public void execute(){

light.switchOff();

}

}

**Light is our receiver class, so let's set that up now:**

**Receiver**

public class Light{

private boolean on;

public void switchOn(){

on = true;

}

public void switchOff(){

on = false;

}

}

**Our invoker in this case is the remote control.**

**Invoker**

public class RemoteControl{

private Command command;

public void setCommand(Command command){

this.command = command;

}

public void pressButton(){

command.execute();

}

}

**Finally we'll set up a client to use the invoker**

**Client**

public class Client{

public static void main(String[] args) {

RemoteControl control = new RemoteControl();

Light light = new Light();

Command lightsOn = new LightsOnCommand(light);

Command lightsOff = new LightsOffCommand(light);

//switch on

control.setCommand(lightsOn);

control.pressButton();

//switch off

control.setCommand(lightsOff);

control.pressButton();

}

}