**Design Pattern Lab File**

*Submitted in Partial Fulfilment of requirements for the award of*

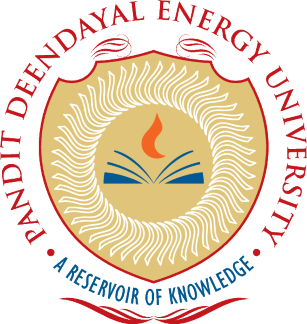
*Degree of Bachelor of Technology in Computer Science and Engineering*

**Submitted by**

Heet Dobariya (22BCP177)

**Submitted to**

Prof. Meera Khanna



***Department of Computer Science and Engineering***

***School of Technology***

***Pandit Deendayal Energy University, Gandhinagar***

***April 2024***

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Experiment Name** | **Date** | **Sign** |
| 1 | Factory Design Pattern (Operating System Example) | 11/01/2024 |  |
| 2 | Factory Design Pattern (Car Example) | 11/01/2024 |  |
| 3 | Builder Design Pattern (Phone Example) | 15/01/2024 |  |
| 4 | Builder Design Pattern (Car Example) | 15/01/2024 |  |
| 5 | Prototype Design Pattern (Vehicle Example) | 18/01/2024 |  |
| 6 | Prototype Design Pattern (Car Example) | 18/01/2024 |  |
| 7 | Abstract Design Pattern (Tech Example) | 29/01/2024 |  |
| 8 | Abstract Design Pattern (Car Example) | 29/01/2024 |  |
| 9 | Abstract Design Pattern (Packaging Example) | 29/01/2024 |  |
| 10 | Singleton Design Pattern | 05/02/2024 |  |
| 11 | Singleton Design Pattern (Packaging Example) | 05/02/2024 |  |
| 12 | Composite Design Pattern | 08/02/2024 |  |
| 13 | Composite Design Pattern (Packaging Example) | 08/02/2024 |  |
| 14 | Facade Design Pattern (Phone Example) | 12/02/2024 |  |
| 15 | Adapter Design Pattern (Media Player Example) | 22/02/2024 |  |
| 16 | Adapter Design Pattern (Packaging Example) | 07/03/2024 |  |
| 17 | Fly-Weight Design Pattern (Shape Example) | 11/03/2024 |  |
| 18 | Fly-Weight Design Pattern (Packaging Example) | 11/03/2024 |  |
| 19 | Decorated Design Pattern (Coffee Example) | 18/03/2024 |  |
| 20 | Decorated Design Pattern (Packaging Example) | 18/03/2024 |  |
| 21 | State Design Pattern (Mobile Example) | 29/03/2024 |  |
| 22 | State Design Pattern (Delivery Example) | 01/04/2024 |  |
| 23 | Observer Design Pattern (NewsLetter Example) | 11/04/2024 |  |
| 24 | Iterator Design Pattern (Iterator Example) | 15/04/2024 |  |
| 25 | Memento Design Pattern (Text Editor Example) | 18/04/2024 |  |
| 26 | MVC Design Pattern (Restaurant Example) | 22/04/2024 |  |

**Factory Pattern Assignment - 1**

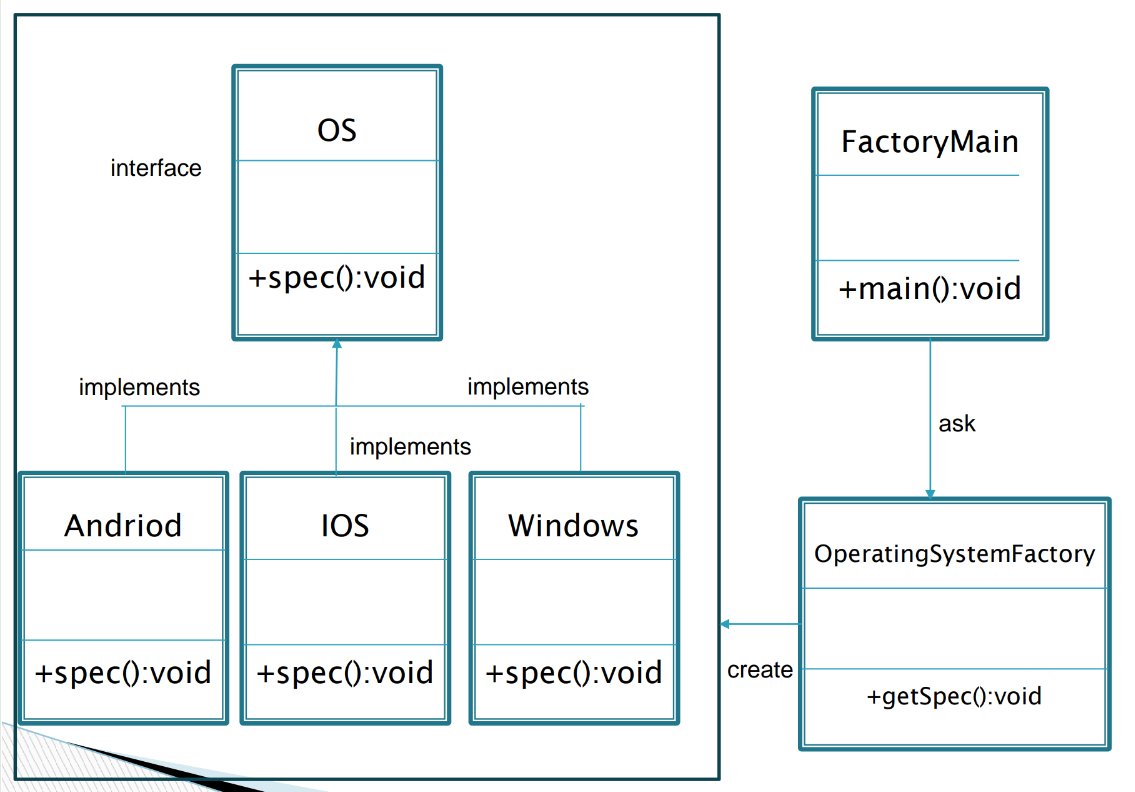
Name: Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Factory Design :**

A creational design pattern called the Factory Design Pattern gives subclasses the ability to modify the kind of objects that are created while still providing an interface for doing so in a super class. It entails specifying an interface for object creation, allowing subclasses to choose the type of object to be created, and then constructing the instance in a method.

By abstracting the object creation process, this design facilitates loose coupling by facilitating changes or extensions to the creation logic without requiring changes to the client code.

* **Program :** Create a Factory Design Pattern for Mobile Factory example.
* **UML Diagram :**



* **Code :**

public interface OS

{

    public void spec();

}

public class Android implements OS

{

    @ Override

    public void spec()

    {

        System.out.println("Most Powerful OS");

    }

}

public class IOS implements OS

{

    @ Override

    public void spec()

    {

        System.out.println("Most Secured OS");

    }

}

public class Windows implements OS

{

    @ Override

    public void spec()

    {

        System.out.println("I am about to die!");

    }

}

public class OperatingSystemFactory

{

    public OS getInstance(String str)

    {

        if (str.equals("Open"))

            return new Android();

        else if (str.equals("Closed"))

            return new IOS();

        else

            return new Windows();

    }

}

public class FactoryMain

{

    public static void main(String[] args)

    {

        OperatingSystemFactory osf = new OperatingSystemFactory();

        OS obj = osf.getInstance("Closed");

        obj.spec();

    }

}

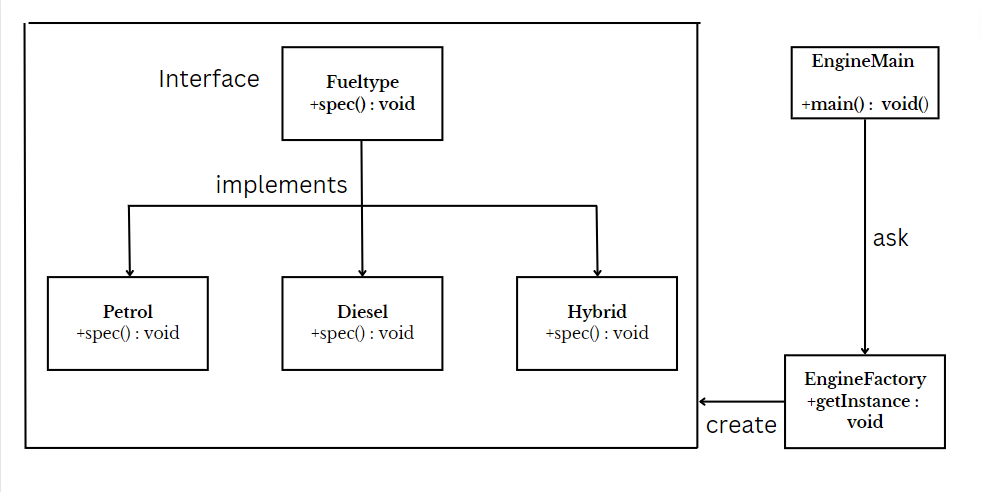
* **Output :**

Screenshot (222)

**Factory Pattern Assignment - 2**

Name: Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Program :** Create a Factory Design Pattern by taking suitable example.
* **UML Diagram :**



* **Code :**

package FactoryDesign2;

public interface FuelType

{

public void spec();

}

package FactoryDesign2;

public class Petrol implements FuelType

{

@ Override

public void spec()

{

System.out.println("Petrol Engine - SPress the gas & feel the power!!");

}

}

package FactoryDesign2;

public class Diesel implements FuelType

{

public void spec()

{

System.out.println("Diesel Engine - Want economy, this is for you");

}

}

package FactoryDesign2;

public class Hybrid implements FuelType

{

@ Override

public void spec()

{

System.out.println("Hybrid Engine - Are you a tech lover?");

}

}

package FactoryDesign2;

public class EngineFactory

{

public FuelType getInstance(String str)

{

if (str.equals("Grawl"))

return new Petrol();

else if (str.equals("Smoky"))

return new Diesel();

else

return new Hybrid();

}

}

package FactoryDesign2;

public class EngineMain

{

public static void main(String[] args)

{

EngineFactory ftf = new EngineFactory();

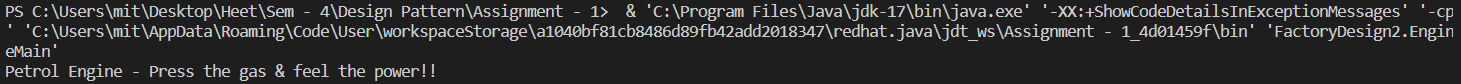
FuelType obj = ftf.getInstance("Grawl");

obj.spec();

}

}

* **Output :**



**Builder Design Pattern**

**Assignment - 1**

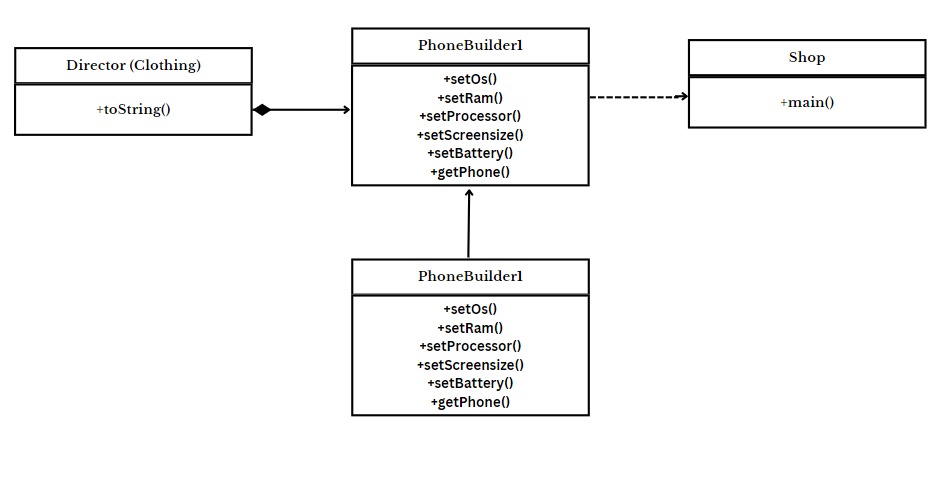
Name: Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Builder Design :**

The Builder Design Pattern is a creational design pattern that provides a way to construct a complex object step by step. It separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

The main idea behind the Builder pattern is to have a dedicated "Builder" class responsible for constructing an object with a complex structure. This builder class typically has a set of methods for configuring different parts of the object. There is also a "Director" class that orchestrates the construction process by using the builder to create the final object.

* **Program :** Create a Builder Design Pattern for Phone example.
* **UML Diagram :**



* **Code :**

package PhoneBuilder;

public class Phone

{

private String os;

private int ram;

private String processor;

private double screensize;

private int battery;

public Phone(String os, int ram, String processor, double screensize, int battery)

{

this.os = os;

this.ram = ram;

this.processor = processor;

this.screensize = screensize;

this.battery = battery;

}

@Override

public String toString()

{

return "Phone [OS ="+os+", RAM ="+ram+", Processor =" +processor+", Screensize ="+screensize+", Battery ="+battery+"]";

}

}

package PhoneBuilder;

public class PhoneBuilder

{

private String os;

private int ram;

private String processor;

private double screensize;

private int battery;

public PhoneBuilder setOs(String os1)

{

this.os = os1;

return this;

}

public PhoneBuilder setRam(int ram1)

{

this.ram = ram1;

return this;

}

public PhoneBuilder setProcessor(String processor1)

{

this.processor = processor1;

return this;

}

public PhoneBuilder setScreensize(double screensize1)

{

this.screensize = screensize1;

return this;

}

public PhoneBuilder setBattery(int battery1)

{

this.battery = battery1;

return this;

}

public Phone getPhone()

{

return new Phone(os, ram, processor, screensize, battery);

}

}

package PhoneBuilder;

public class PhoneBuilder1

{

private String os;

private int ram;

private String processor;

private double screensize;

private int battery;

public void setOs(String os1)

{

this.os = os1;

}

public void setRam(int ram1)

{

this.ram = ram1;

}

public void setProcessor(String processor1)

{

this.processor = processor1;

}

public void setScreensize(double screensize1)

{

this.screensize = screensize1;

}

public void setBattery(int battery1)

{

this.battery = battery1;

}

public Phone getPhone()

{

return new Phone(os, ram, processor, screensize, battery);

}

}

package PhoneBuilder;

public class Shop

{

public static void main(String[] args)

{

Phone p = new PhoneBuilder().setOs("Android").setRam(2).setBattery(3100).getPhone();

System.out.println(p);

// You can set the no. of parameters by yourself sequence does not matter, set them in any order

// If you don't want the above cascaded calls for Phone Object creation

// then create PhoneBuilder1 class object and execute the setters call separately

PhoneBuilder1 p1 = new PhoneBuilder1();

p1.setOs("Android");

p1.setRam(16);

p1.setBattery(5000);

System.out.println(p1.getPhone());

}

}

* **Output :**

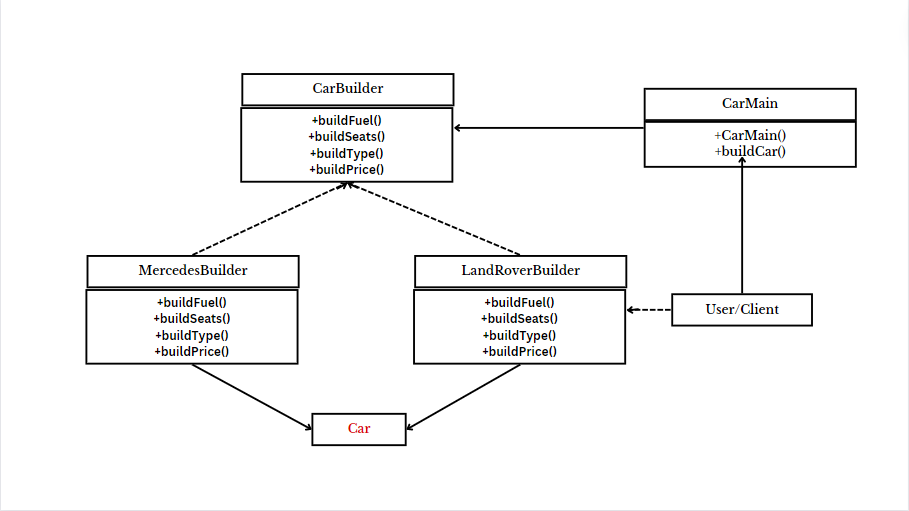


**Builder Design Pattern**

**Assignment - 2**

Name: Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Program :** Create a Builder Design Pattern by taking any suitable example.
* **UML Diagram :**



* **Code :**

package CarBuilder;

public class Car

{

private String fuel;

private int seats;

private String cartype;

private double price;

public void setFuel(String fuel1)

{

this.fuel = fuel1;

}

public void setSeats(int seats1)

{

this.seats = seats1;

}

public void setType(String type1)

{

this.cartype = type1;

}

public void setPrice(double price1)

{

this.price = price1;

}

public void showCar()

{

System.out.println("Car [Fueltype ="+fuel+", No. of seats ="+seats+", Car Type =" +cartype+", Price ="+price+"]");

}

}

package CarBuilder;

public interface CarBuilder

{

public void buildFuel();

public void buildSeats();

public void buildType();

public void buildPrice();

public Car getCar();

}

package CarBuilder;

public class LandRoverBuilder implements CarBuilder

{

private Car c;

public LandRoverBuilder()

{

this.c = new Car();

}

@ Override

public void buildFuel()

{

c.setFuel("Diesel");

}

@ Override

public void buildSeats()

{

c.setSeats(5);

}

@ Override

public void buildType()

{

c.setType("SUV");

}

@ Override

public void buildPrice()

{

c.setPrice(15000000);

}

public Car getCar()

{

Car newcar = this.c;

this.c = new Car();

return newcar;

}

}

package CarBuilder;

public class MercedesBuilder implements CarBuilder

{

private Car c;

public MercedesBuilder()

{

this.c = new Car();

}

@ Override

public void buildFuel()

{

c.setFuel("Petrol");

}

@ Override

public void buildSeats()

{

c.setSeats(5);

}

@ Override

public void buildType()

{

c.setType("SUV");

}

@ Override

public void buildPrice()

{

c.setPrice(30000000);

}

public Car getCar()

{

Car newcar = this.c;

this.c = new Car();

return newcar;

}

}

package CarBuilder;

public class CarMain

{

private CarBuilder cbuild;

public CarMain(CarBuilder cbuild)

{

this.cbuild = cbuild;

}

public void buildCar()

{

cbuild.buildFuel();

cbuild.buildSeats();

cbuild.buildType();

cbuild.buildPrice();

}

}

package CarBuilder;

public class CarShop

{

public static void main(String[] args)

{

CarBuilder cbuild;

CarMain cmain;

Car c;

cbuild = new MercedesBuilder();

cmain = new CarMain(cbuild);

cmain.buildCar();

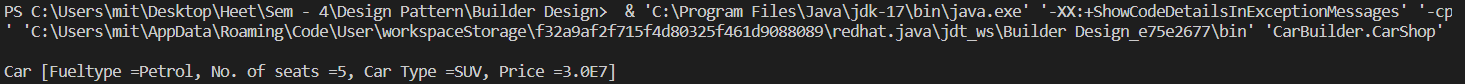
c = cbuild.getCar();

c.showCar();

}

}

* **Output :**



**Prototype Design Pattern**

**Assignment - 1**

Name: Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Prototype Design Pattern :**

The Prototype Design Pattern is a creational design pattern that deals with object creation mechanisms, trying to create objects in a manner suitable to the situation. The pattern creates new objects by copying an existing object, known as the prototype. This process is especially useful when the cost of creating an object is more expensive or complex than copying an existing one.

The Prototype pattern involves creating new objects by copying an existing object, known as the prototype. The basic idea is to use a prototype instance as a blueprint for creating new objects, allowing the copying of properties, data, and structure from the prototype to the new instances.

* **Types of Clones :**

1. Shallow Clone:

--> A shallow clone creates a new object but does not create copies of the nested objects within the original object.

--> In other words, the cloned object and the original object share references to the same nested objects.

--> Changes made to the nested objects in the cloned instance may affect the original instance and vice versa.

1. Deep Clone:

--> A deep clone creates a new object and also creates copies of all the nested objects within the original object.

--> The cloned object and the original object have their independent copies of nested objects.

--> Changes made to the nested objects in the cloned instance do not affect the original instance and vice versa.

* **Program :** Create a Prototype Design Pattern for Vehicle Example.
* **Code :**

import java.util.\*;

public class Vehicle implements Cloneable

{

private List<String> vehiclelist;

public Vehicle()

{

this.vehiclelist = new ArrayList<String>();

}

public Vehicle(List<String> list)

{

this.vehiclelist = list;

}

public void insertData()

{

vehiclelist.add("Mercedes AMG G63");

vehiclelist.add("Land Rover Defender");

vehiclelist.add("Mercedes G600");

vehiclelist.add("Cadillac Escalade V");

vehiclelist.add("Volkswagen Virtus");

}

public List<String> getVehicleList()

{

return this.vehiclelist;

}

// Shallow Copy

public Object clone1() throws CloneNotSupportedException

{

return super.clone();

}

// Deep Copy

public Object clone2() throws CloneNotSupportedException

{

List<String> tempList = new ArrayList<String>();

for (String s: this.vehiclelist)

{

tempList.add(s);

}

return new Vehicle(tempList);

}

}

import java.util.\*;

public class PrototypePattern

{

public static void main(String[] args) throws CloneNotSupportedException

{

Vehicle a = new Vehicle();

a.insertData();

Vehicle b = (Vehicle) a.clone1();

System.out.println("For Shallow Copy:");

System.out.println("A: "+a.getVehicleList());

System.out.println("B: "+b.getVehicleList());

Vehicle c = (Vehicle) a.clone2();

System.out.println("For Deep Copy:");

System.out.println("A: "+a.getVehicleList());

System.out.println("B: "+c.getVehicleList());

List<String> list = c.getVehicleList();

list.add("Porsche Boxter");

System.out.println("B: "+c.getVehicleList());

c.getVehicleList().remove("Mercedes G600");

System.out.println("B: "+list);

System.out.println("A: "+a.getVehicleList());

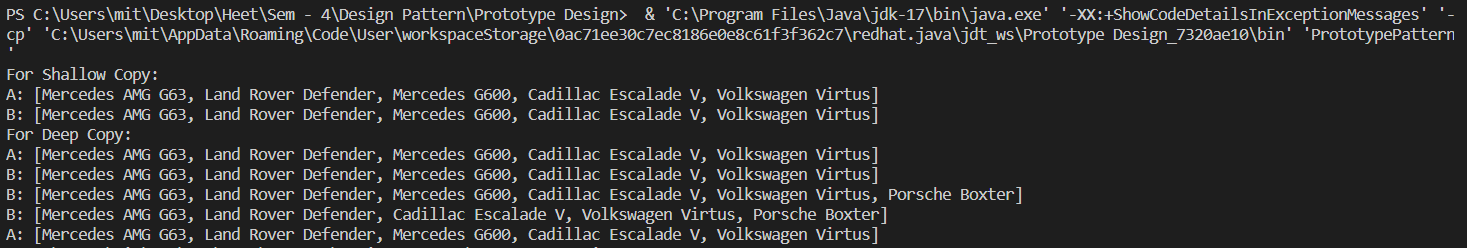
// b.modify();

}

}

// Deep Copy can copy nested objects also but Shallow Copy will not copy the nested objects.

* **Output :**

****

* **Advantages of Prototype Design Pattern :**

1) Efficient creation: Clone existing items to save time and

resources.

2) Flexibility: Enables dynamic runtime changes to object

structures.

1. Reuse promotes code reuse by utilising existing designs.

* **Disadvantages of Prototype Design Pattern :**

1) Lifecycle Management: Managing the lifecycle of cloned

objects, particularly those with nested structures, can be

challenging.

2) Initialization Challenges: It can be challenging to properly

initialise cloned objects, particularly ones with dependencies.

3) Unwanted Modifications: Because prototypes and clones share

structures, they might cause unwanted changes that must be

handled carefully.

**Prototype Design Pattern**

**Assignment - 2**

Name: Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Program :** Create a Prototype Design Pattern for Car Specifications Example.
* **Code :**

import java.util.\*;

public class CarSpecification implements Cloneable

{

private List<String> carspecs;

public CarSpecification()

{

this.carspecs = new ArrayList<String>();

}

public CarSpecification(List<String> list)

{

this.carspecs = list;

}

public void insertData()

{

carspecs.add("V8 Engine");

carspecs.add("SUV");

carspecs.add("Bi-Turbo");

carspecs.add("Quad-Tip Exhaust");

}

public void modify(String oldItem, String newItem)

{

int index = carspecs.indexOf(oldItem);

if (index != -1)

{

carspecs.set(index, newItem);

System.out.println("Modification successful.");

}

else

{

System.out.println(oldItem + " not found in the clothing list.");

}

}

public List<String> getSpecList()

{

return this.carspecs;

}

// Shallow Copy

public Object clone1() throws CloneNotSupportedException

{

return super.clone();

}

// Deep Copy

public Object clone2() throws CloneNotSupportedException

{

List<String> tempList = new ArrayList<String>();

for (String s: this.carspecs)

{

tempList.add(s);

}

return new CarSpecification(tempList);

}

}

import java.util.\*;

public class PrototypePattern2

{

public static void main(String[] args) throws CloneNotSupportedException

{

CarSpecification a = new CarSpecification();

a.insertData();

CarSpecification b = (CarSpecification) a.clone1();

System.out.println("For Shallow Copy: ");

System.out.println("Original List: "+ a.getSpecList());

System.out.println("Clone List: "+ b.getSpecList());

CarSpecification c = (CarSpecification) a.clone2();

System.out.println("For Deep Copy:");

System.out.println("Original List: "+a.getSpecList());

System.out.println("Clone List: "+c.getSpecList());

List<String> list = c.getSpecList();

list.add("Mafia SUV");

System.out.println("Clone list after addition of one element: "+c.getSpecList());

c.modify("V8 Engine","Roaring V8");

System.out.println("Clone list after modifying one element: "+c.getSpecList());

c.getSpecList().remove("SUV");

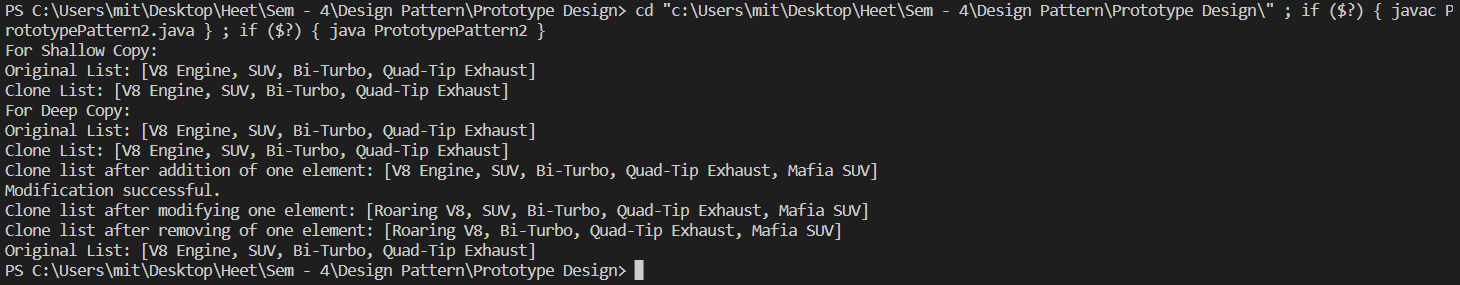
System.out.println("Clone list after removing of one element: "+list);

System.out.println("Original List: "+a.getSpecList());

}

}

* **Output :**

****

**Abstract Design Pattern**

**Assignment - 1**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Abstract Design :**

The Abstract Factory design pattern is a creational pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It falls under the Gang of Four (GoF) design patterns and aims to abstract the process of object creation, making it possible to create families of related objects without specifying their concrete classes.

Key components of the Abstract Factory pattern include:

1. \*Abstract Factory Interface:\*

- Declares a set of methods that create the abstract product objects. These methods typically represent the various types of objects that can be created within a family.

2. \*Concrete Factories:\*

- Implement the Abstract Factory interface and are responsible for creating concrete product objects. Each concrete factory corresponds to a specific family of related products.

3. \*Abstract Product:\*

- Declares the interface for a type of product object created by the abstract factory. These products form a family of related objects.

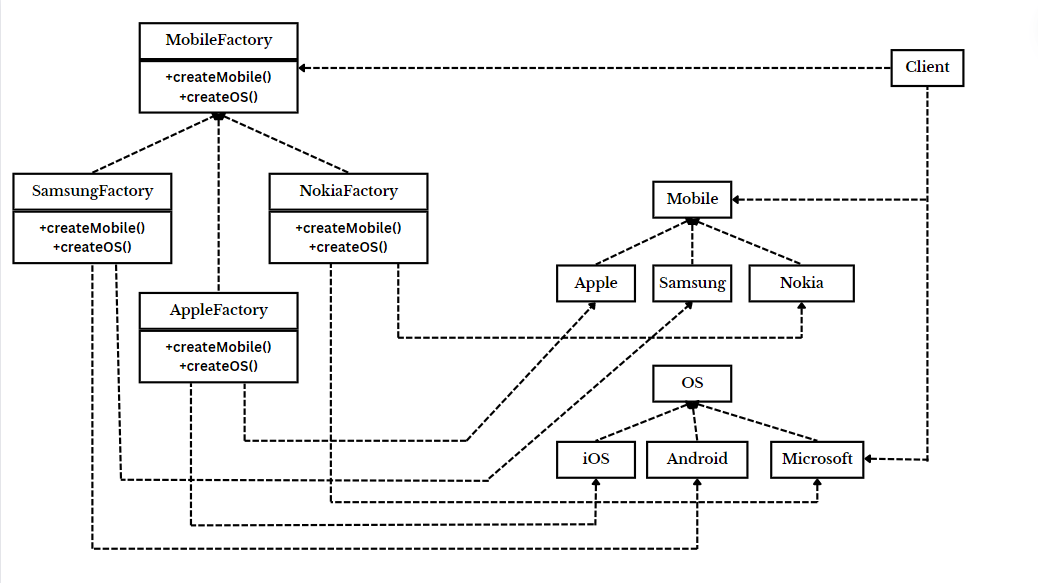
4. \*Concrete Products:\*

- Implement the Abstract Product interface and represent the actual product objects that are created by the concrete factories.

The Abstract Factory pattern allows a client to use an abstract interface to create families of related or dependent objects, making it easy to substitute different families of objects without altering the client code. This promotes flexibility and helps in designing systems that are independent of the way their objects are created, composed, and represented.

One common use case for the Abstract Factory pattern is in graphical user interface (GUI) libraries, where different platforms (such as Windows, macOS, or Linux) may have different implementations for buttons, windows, and other UI elements. The Abstract Factory pattern enables the creation of UI components that are specific to a particular platform, ensuring consistency within each family of related objects.

* **Program :** Create a Builder Design Pattern for Mobile Factory example.
* **UML Diagram :**



* **Code :**

// Abstract Product OS

interface OS

{

void displayOS();

}

// Concrete Products of OS: AndroidOS, WindowsOS, iOS

class AndroidOS implements OS

{

@Override

public void displayOS()

{

System.out.println("Powered by Android");

}

}

class WindowsOS implements OS

{

@Override

public void displayOS()

{

System.out.println("Powered by Windows");

}

}

class iOS implements OS

{

@Override

public void displayOS()

{

System.out.println("Powered by iOS");

}

}

// Abstract Product Mobile

interface Mobile

{

void displayInfo();

}

// Concrete Products of Mobile SamsungMobile, NokiaMobile, AppleMobile

class SamsungMobile implements Mobile

{

@Override

public void displayInfo()

{

System.out.println("This is Samsung Mobile");

}

}

class NokiaMobile implements Mobile

{

@Override

public void displayInfo()

{

System.out.println("This is Nokia Mobile");

}

}

class AppleMobile implements Mobile

{

@Override

public void displayInfo()

{

System.out.println("This is Apple Mobile");

}

}

// Abstract Factory MobileFactory

interface MobileFactory

{

Mobile createMobile();

OS createOS();

}

// Concrete Factories SamsungFactory, NokiaFactory, AppleFactory

class SamsungFactory implements MobileFactory

{

@Override

public Mobile createMobile()

{

return new SamsungMobile();

}

@Override

public OS createOS()

{

return new AndroidOS();

}

}

class NokiaFactory implements MobileFactory

{

@Override

public Mobile createMobile()

{

return new NokiaMobile();

}

@Override

public OS createOS()

{

return new WindowsOS();

}

}

class AppleFactory implements MobileFactory

{

@Override

public Mobile createMobile()

{

return new AppleMobile();

}

@Override

public OS createOS()

{

return new iOS();

}

}

// Client Code

public class AbstractFactoryExample

{

public static void main(String[] args)

{

MobileFactory samsungFactory = new SamsungFactory();

Mobile samsungMobile = samsungFactory.createMobile();

OS samsungOS = samsungFactory.createOS();

samsungMobile.displayInfo();

samsungOS.displayOS();

MobileFactory nokiaFactory = new NokiaFactory();

Mobile nokiaMobile = nokiaFactory.createMobile();

OS nokiaOS = nokiaFactory.createOS();

nokiaMobile.displayInfo();

nokiaOS.displayOS();

MobileFactory appleFactory = new AppleFactory();

Mobile appleMobile = appleFactory.createMobile();

OS appleOS = appleFactory.createOS();

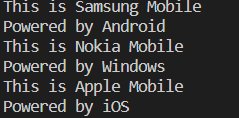
appleMobile.displayInfo();

appleOS.displayOS();

}

}

* **Output :**

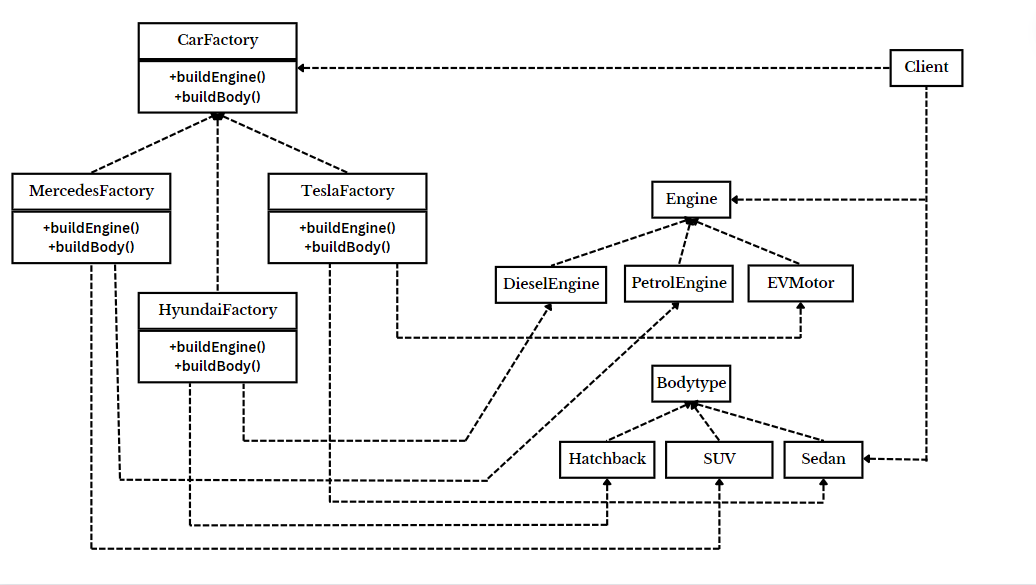


**Abstract Design Pattern**

**Assignment - 2**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Program :** Create a Abstract Design Pattern for suitable example.
* **UML Diagram :**



* **Code :**

package EnginewithBodytype;

interface Engine

{

void displayEngine();

}

package EnginewithBodytype;

// Concrete Products of Engine: PetrolEngine, DieselEngine, EVMotor

class PetrolEngine implements Engine

{

@Override

public void displayEngine()

{

System.out.println("Powered by Petrol, which gives mind blowing thrust!");

}

}

package EnginewithBodytype;

class DieselEngine implements Engine

{

@Override

public void displayEngine()

{

System.out.println("Powered by Diesel, which is bit more polluting!");

}

}

package EnginewithBodytype;

class EVMotor implements Engine

{

@Override

public void displayEngine()

{

System.out.println("Powered by Electricity, Huhh kids!");

}

}

package EnginewithBodytype;

interface BodyType

{

void displayBody();

}

package EnginewithBodytype;

class SUV implements BodyType

{

@Override

public void displayBody()

{

System.out.println("SUV: Having a body roll!");

}

}

package EnginewithBodytype;

class Sedan implements BodyType

{

@Override

public void displayBody()

{

System.out.println("Sedan: Most stable on road!");

}

}

package EnginewithBodytype;

class Hatchback implements BodyType

{

@Override

public void displayBody()

{

System.out.println("Hatchback: Known for compactness!");

}

}

package EnginewithBodytype;

interface CarFactory

{

Engine buildEngine();

BodyType buildBody();

}

package EnginewithBodytype;

class MercedesFactory implements CarFactory

{

@Override

public Engine buildEngine()

{

return new PetrolEngine();

}

@Override

public BodyType buildBody()

{

return new SUV();

}

}

package EnginewithBodytype;

class HyundaiFactory implements CarFactory

{

@Override

public Engine buildEngine()

{

return new DieselEngine();

}

@Override

public BodyType buildBody()

{

return new Hatchback();

}

}

package EnginewithBodytype;

class TeslaFactory implements CarFactory

{

@Override

public Engine buildEngine()

{

return new EVMotor();

}

@Override

public BodyType buildBody()

{

return new Sedan();

}

}

package EnginewithBodytype;

public class Client

{

public static void main(String[] args)

{

System.out.println("For Petrol SUVs:");

CarFactory mercedesFactory = new MercedesFactory();

Engine petrolEngine = mercedesFactory.buildEngine();

BodyType suv = mercedesFactory.buildBody();

petrolEngine.displayEngine();

suv.displayBody();

System.out.println("For Diesel Hatchbacks:");

CarFactory suzukiFactory = new HyundaiFactory();

Engine dieselEngine = suzukiFactory.buildEngine();

BodyType hatchback = suzukiFactory.buildBody();

dieselEngine.displayEngine();

hatchback.displayBody();

System.out.println("For EV Sedans:");

CarFactory teslaFactory = new TeslaFactory();

Engine evMotor = teslaFactory.buildEngine();

BodyType sedan = teslaFactory.buildBody();

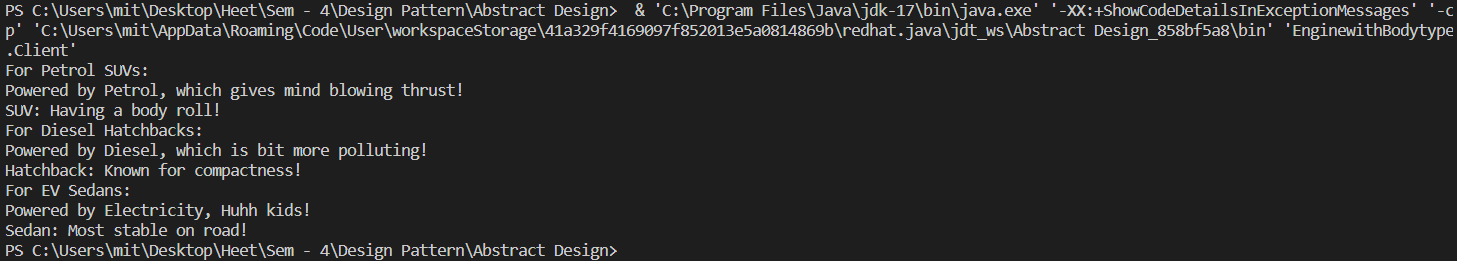
evMotor.displayEngine();

sedan.displayBody();

}

}

* **Output :**

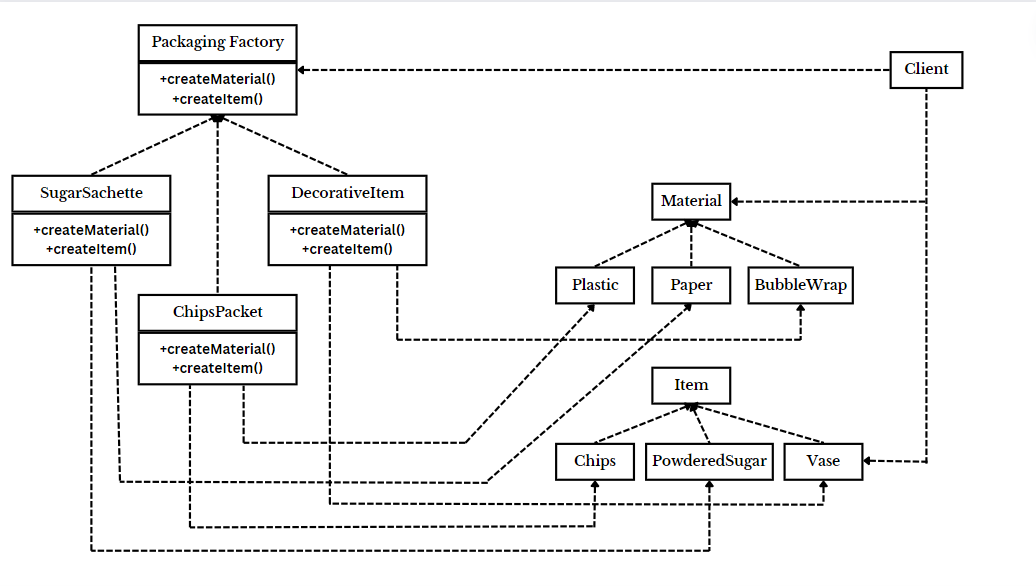
****

**Abstract Design Pattern**

**Assignment - 3**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Program :** Create a Builder Design Pattern for Packaging Factory example.
* **UML Diagram :**



* **Code :**

package Packaging;

interface Material

{

void displayMaterial();

void displayColour();

}

package Packaging;

class Plastic implements Material

{

@Override

public void displayMaterial()

{

System.out.println("Plastic Material: Can be used where durability is needed.");

}

@Override

public void displayColour()

{

System.out.println("Green in Colour");

}

}

package Packaging;

class Paper implements Material

{

@Override

public void displayMaterial()

{

System.out.println("Paper Material: It is easy to recycle.");

}

@Override

public void displayColour()

{

System.out.println("White Colour is the colour of paper");

}

}

package Packaging;

class BubbleWrap implements Material

{

@Override

public void displayMaterial()

{

System.out.println("Bubble Wrap: Can be used for fragile items.");

}

@Override

public void displayColour()

{

System.out.println("Transparent in colour.s");

}

}

package Packaging;

interface Item

{

void displayItem();

}

package Packaging;

class PowderedSugar implements Item

{

@Override

public void displayItem()

{

System.out.println("Powdered Sugar is the item used.");

}

}

package Packaging;

class Wafers implements Item

{

@Override

public void displayItem()

{

System.out.println("Wafers and Chips is the item used.");

}

}

package Packaging;

class Vase implements Item

{

@Override

public void displayItem()

{

System.out.println("Vase is the used.");

}

}

package Packaging;

interface PackagingFactory

{

Material createMaterial();

Item createItem();

}

package Packaging;

class SugarSachette implements PackagingFactory

{

@Override

public Material createMaterial()

{

return new Paper();

}

@Override

public Item createItem()

{

return new PowderedSugar();

}

}

package Packaging;

class ChipsPacket implements PackagingFactory

{

@Override

public Material createMaterial()

{

return new Plastic();

}

@Override

public Item createItem()

{

return new Wafers();

}

}

package Packaging;

class DecorativeItem implements PackagingFactory

{

@Override

public Material createMaterial()

{

return new BubbleWrap();

}

@Override

public Item createItem()

{

return new Vase();

}

}

package Packaging;

public class Client

{

public static void main(String[] args)

{

System.out.println("For sugar sachette:");

PackagingFactory sugarSachette = new SugarSachette();

Material material1 = sugarSachette.createMaterial();

Item item1 = sugarSachette.createItem();

material1.displayMaterial();

material1.displayColour();

item1.displayItem();

System.out.println("For chips packet:");

PackagingFactory chipsPacket = new ChipsPacket();

Material material2 = chipsPacket.createMaterial();

Item item2 = chipsPacket.createItem();

material2.displayMaterial();

material2.displayColour();

item2.displayItem();

System.out.println("For decorative item:");

PackagingFactory decorativeItem = new DecorativeItem();

Material material3 = decorativeItem.createMaterial();

Item item3 = decorativeItem.createItem();

material3.displayMaterial();

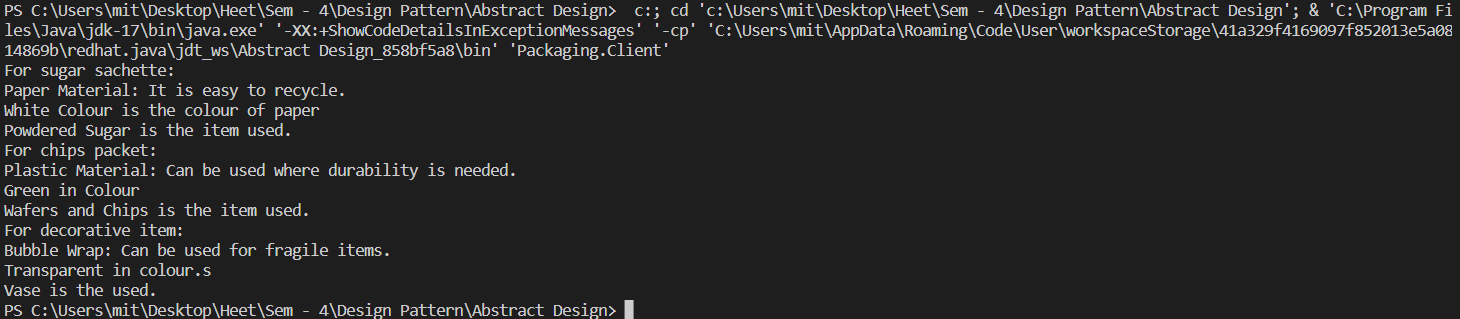
material3.displayColour();

item3.displayItem();

}

}

* **Output :**



**Singleton Design Pattern**

**Assignment - 1**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Singleton Design :**

The Singleton design pattern is a creational pattern that ensures a class has only one instance and provides a global point of access to that instance. In other words, it restricts the instantiation of a class to a single object and provides a way to access that instance from any point in the application.

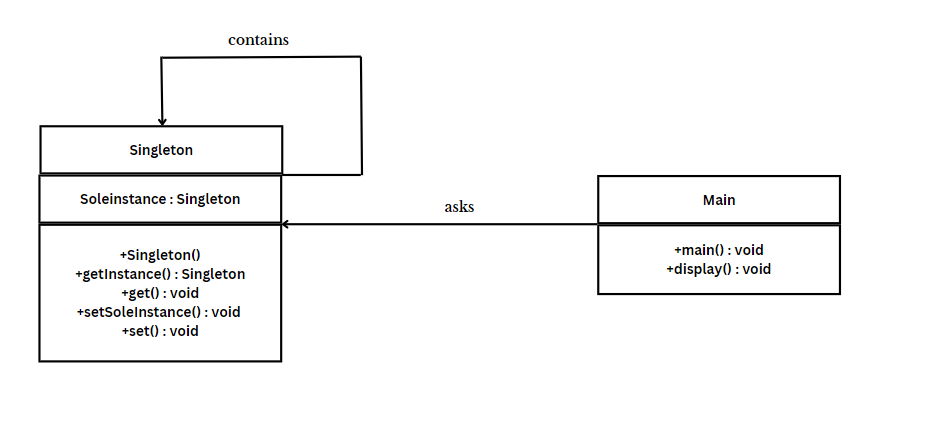
* The key features of a Singleton pattern include:

1. Single Instance: There is only one instance of the class that is created and shared.
2. Global Access: The instance is globally accessible, meaning that it can be accessed from any part of the application.
3. Lazy Initialization (optional): The instance is created only when it is first needed, not necessarily when the program starts.

A common implementation of the Singleton pattern involves a private constructor to prevent direct instantiation, a static method to provide access to the instance, and a static variable to hold the single instance.

It's important to note that while the Singleton pattern has its use cases, it should be used judiciously as it introduces a global state, which might lead to issues such as tight coupling and difficulties in unit testing. Additionally, in multi-threaded environments, special attention must be paid to ensure thread safety during the initialization of the singleton instance.

* **Program :** Give an example of Singleton Design Pattern.
* **UML Diagram :**



* **Code :**

1. **For Lazy Initialization -**

public class Singleton

{

// private static Singleton soleInstance = new Singleton();

private static Singleton soleInstance;

public int i;

private Singleton() // private constructor

{

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

}

public static Singleton getInstance() // global access point

{

if(soleInstance == null)

{

soleInstance = new Singleton();

}

return soleInstance;

}

public int get()

{

return i;

}

public static void setSoleInstance(Singleton soleInstance)

{

Singleton.soleInstance = soleInstance;

}

public void set(int i)

{

this.i = i;

}

}

public class TestClass

{

public static void main(String[] args)

{

// Get the Singleton instance

Singleton s1 = Singleton.getInstance();

Singleton s2 = Singleton.getInstance();

s1.set(5);

s2.set(10);

System.out.println(s1.get());

s2.i = s1.i + s2.i;

System.out.println(s2.get());

print("S1",s1);

print("S2",s2);

}

static void print(String name, Singleton obj)

{

System.out.println(String.format("Object: %s, Hashcode: %d", name, obj.hashCode()));

}

}

1. **For Eager Initialization -**

public class Singleton

{

// Private static instance variable

private static Singleton soleInstance = new Singleton();

public int i;

// Private constructor to prevent instantiation from outside the class

private Singleton()

{

// Initialize the Singleton instance

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

}

// Public method to provide global access to the instance

public static Singleton getInstance()

{

return soleInstance;

}

public int get()

{

return i;

}

public static void setSoleInstance(Singleton soleInstance)

{

Singleton.soleInstance = soleInstance;

}

public void set(int i)

{

this.i = i;

}

}

public class TestClass

{

public static void main(String[] args)

{

// Get the Singleton instance

Singleton s1 = Singleton.getInstance();

Singleton s2 = Singleton.getInstance();

s1.set(5);

s2.set(10);

System.out.println(s1.get());

s2.i = s1.i + s2.i;

System.out.println(s2.get());

print("S1",s1);

print("S2",s2);

}

static void print(String name, Singleton obj)

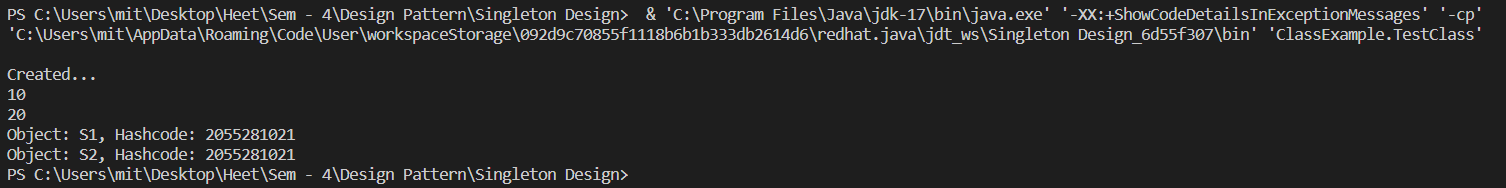
{

System.out.println(String.format("Object: %s, Hashcode: %d", name, obj.hashCode()));

}

}

* **Output :**

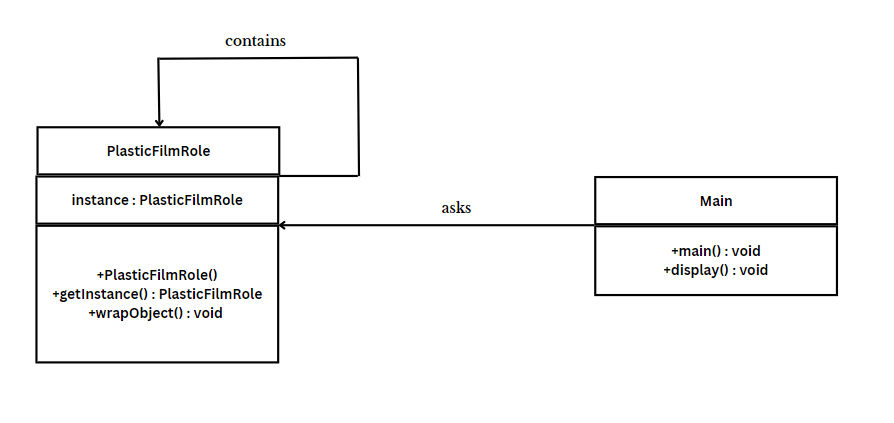
****

**Singleton Design Pattern**

**Assignment - 2**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Program :** Create a Singleton Design Pattern for Packaging example.
* **UML Diagram :**



* **Code :**

public class PlasticFilmRole

{

private static PlasticFilmRole instance;

private PlasticFilmRole()

{

System.out.println("Plastic Film Role instance created.");

}

public static PlasticFilmRole getInstance()

{

// Lazy initialization: create the instance only if it doesn't exist yet

if (instance == null) {

instance = new PlasticFilmRole();

}

return instance;

}

// Method to wrap an object with plastic film

public void wrapObject(String objectName)

{

System.out.println("Wrapping " + objectName + " with plastic film.");

}

public static void main(String[] args)

{

// Get the PlasticFilmRole instance

PlasticFilmRole plasticFilmRole = PlasticFilmRole.getInstance();

// Wrap different objects using the same instance

plasticFilmRole.wrapObject("Book");

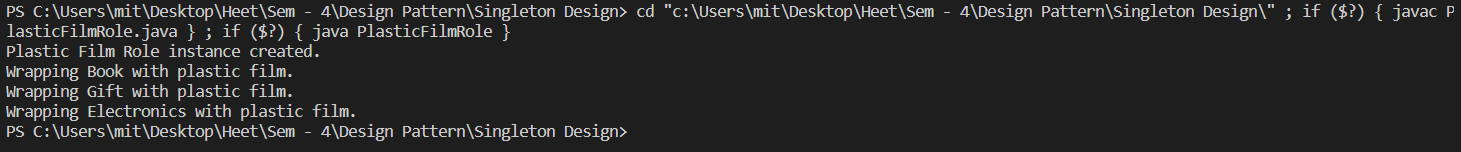
plasticFilmRole.wrapObject("Gift");

plasticFilmRole.wrapObject("Electronics");

}

}

* **Output :**



**Composite Design Pattern**

**Assignment - 1**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Composite Design :**

The Composite Design Pattern is a structural design pattern that allows us to compose objects into tree-like structures to represent part-whole hierarchies. This pattern creates a unified interface for individual objects and compositions of objects, allowing clients to treat both in a uniform manner.

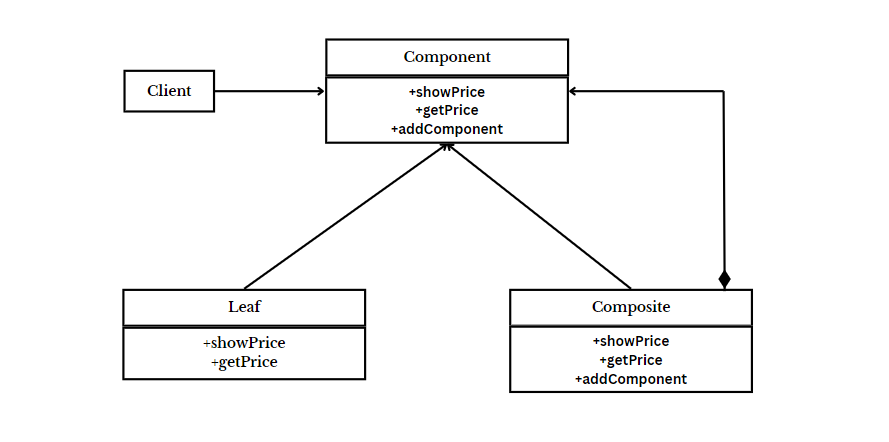
In the Composite pattern, there are three main components:

1. Component: This is the common interface or abstract class that represents both individual objects and compositions. It declares the common operations that can be performed on both types of objects.
2. Leaf: This represents individual objects in the composition. Leaf objects are the end nodes of the tree structure and do not have any children.
3. Composite: This represents the composite objects that contain leaf objects or other composite objects as children. Composite objects implement the operations defined in the component interface, delegating to their children as necessary.

The key idea behind the Composite pattern is that clients can interact with individual objects or compositions of objects uniformly through the component interface. This simplifies client code, as it does not need to distinguish between different types of objects in the hierarchy.

The Composite pattern is commonly used in scenarios where we have tree-like structures and want to treat individual objects and compositions of objects in a uniform manner. Examples include graphical user interfaces, file systems, organization hierarchies, and any other hierarchical structures where you need to represent part-whole relationships.

* **Program :** Implement composite design pattern for the computer as composite component.
* **UML Diagram :**



* **Code :**

public interface Component

{

void showPrice();

int getPrice();

}

public class Leaf implements Component

{

int price;

String name;

Leaf(String name, int price)

{

this.price = price;

this.name = name;

}

@Override

public void showPrice()

{

System.out.println("Leaf -> "+name+" : "+price);

}

@Override

public int getPrice()

{

return price;

}

}

import java.util.\*;

public class Composite implements Component

{

String name;

List<Component> components = new ArrayList();

public Composite(String name)

{

super();

this.name = name;

}

public void addComponent(Component com)

{

components.add(com);

}

@Override

public int getPrice()

{

int p=0;

for(Component c:components)

{

p += c.getPrice();

}

return p;

}

@Override

public void showPrice()

{

System.out.println("Composite -> "+name+" : Price -> "+getPrice());

System.out.println("Leaf of " +name);

for(Component c:components)

{

c.showPrice();

}

}

}

public class CompositeTest

{

public static void main(String[] args)

{

Component hd = new Leaf("Hard Drive", 4000);

Component mouse = new Leaf("Mouse", 500);

Component monitor = new Leaf("Monitor", 8000);

Component ram = new Leaf("RAM", 2000);

Component cpu = new Leaf("CPU", 10000);

Composite ph = new Composite("Peripheral");

Composite cabinet = new Composite("Cabinet");

Composite mb = new Composite("MotherBoard");

Composite computer = new Composite("Computer");

mb.addComponent(cpu);

mb.addComponent(ram);

ph.addComponent(mouse);

ph.addComponent(monitor);

cabinet.addComponent(hd);

cabinet.addComponent(mb);

computer.addComponent(ph);

computer.addComponent(cabinet);

ram.showPrice();

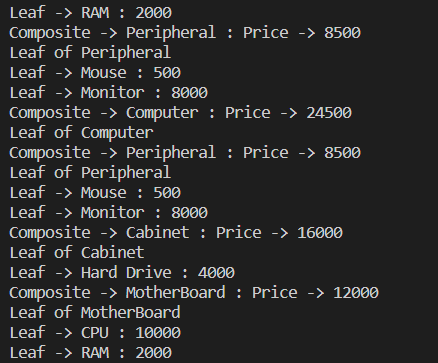
ph.showPrice();

computer.showPrice();

}

}

* **Output:**

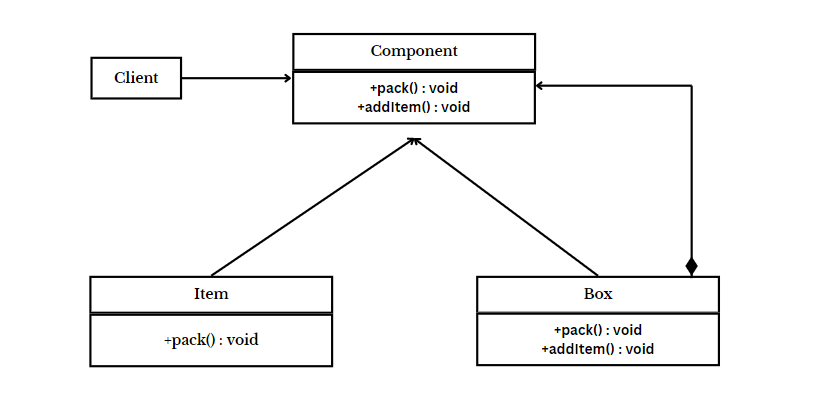
****

**Composite Design Pattern**

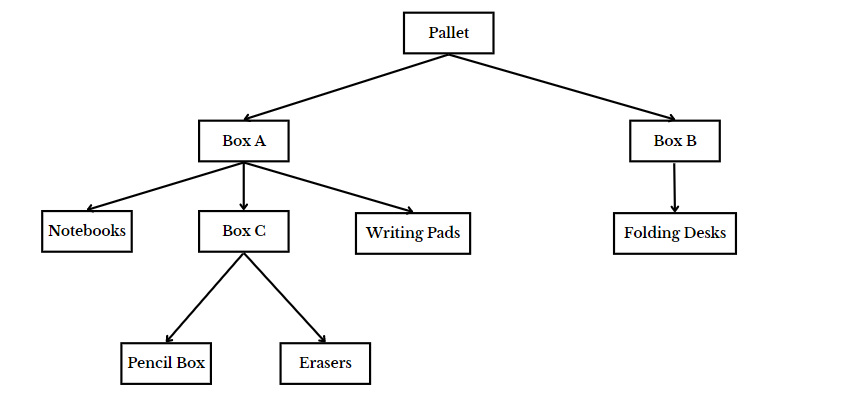
**Assignment - 2**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Program :** Implement composite design pattern for the packaging as composite component.
* **UML Diagram :**



* **Tree Diagram :**



* **Code :**

import java.util.ArrayList;

import java.util.List;

// Component interface

interface PackagingComponent

{

void pack();

}

// Leaf class (individual item)

class Item implements PackagingComponent

{

private String itemName;

public Item(String itemName)

{

this.itemName = itemName;

}

@Override

public void pack()

{

System.out.println("Packing individual item (Leaf): " + itemName);

}

}

// Composite class (box or pallet)

class Box implements PackagingComponent

{

private List<PackagingComponent> items = new ArrayList<>();

private String boxName;

public Box(String boxName)

{

this.boxName = boxName;

}

public void addItem(PackagingComponent item)

{

items.add(item);

}

@Override

public void pack()

{

System.out.println("Packing box (Composite): " + boxName);

// Pack individual items in the box

for (PackagingComponent item : items)

{

item.pack();

}

}

}

// Client code

public class PackagingFactoryExample

{

public static void main(String[] args)

{

// Creating individual items

Item item1 = new Item("Notebooks");

Item item2 = new Item("Writing Pads");

Item item3 = new Item("Pencil Box");

Item item4 = new Item("Erasers");

Item item5 = new Item("Foldable Desk");

// Creating composite boxes

Box box1 = new Box("Box C");

Box box2 = new Box("Box A");

Box box3 = new Box("Box B");

// Adding individual items to boxes

box2.addItem(item1);

box2.addItem(item2);

box1.addItem(item3);

box1.addItem(item4);

box2.addItem(box1);

box3.addItem(item5);

// Creating a pallet as a composite of boxes

Box pallet = new Box("Pallet");

pallet.addItem(box2);

pallet.addItem(box3);

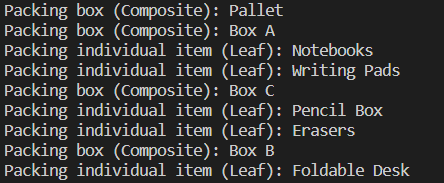
// Packing the entire structure

pallet.pack();

}

}

* **Output :**

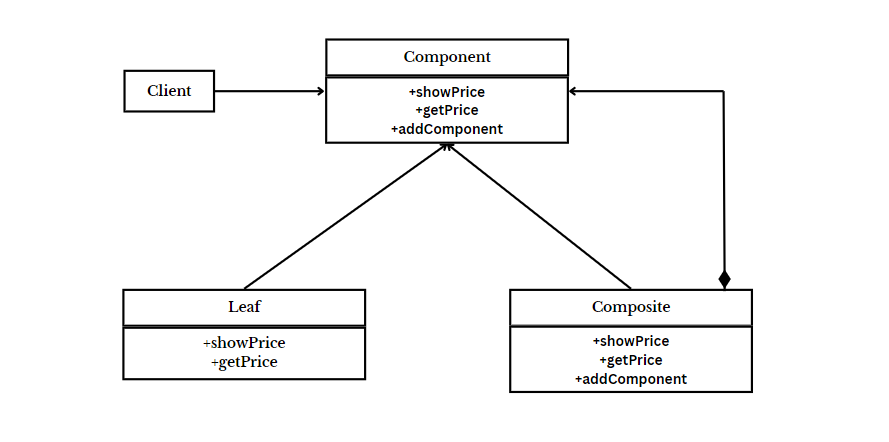
****

**Composite Design Pattern**

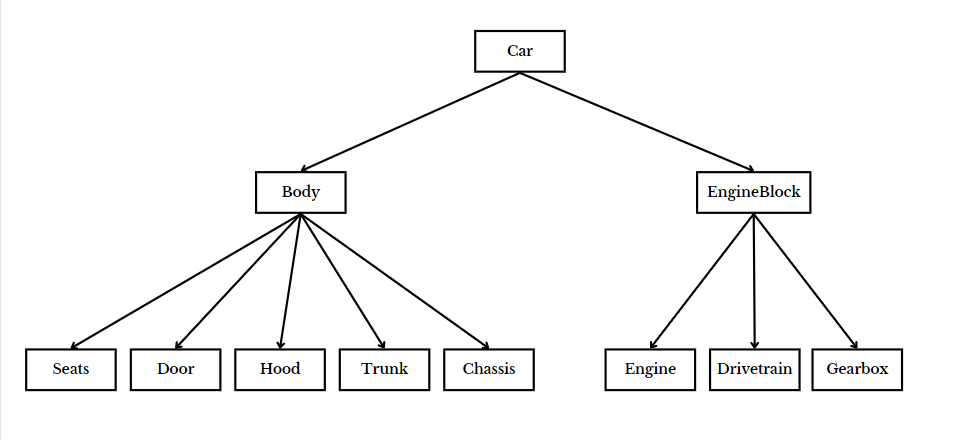
**Assignment - 3**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Program :** Implement composite design pattern for the car as composite component.
* **UML Diagram :**



* **Tree Diagram :**



* **Code :**

import java.util.ArrayList;

import java.util.List;

interface Component

{

void showPrice();

int getPrice();

}

class Leaf implements Component

{

int price;

String name;

Leaf(String name, int price)

{

this.price = price;

this.name = name;

}

@Override

public void showPrice()

{

System.out.println("Leaf -> "+name+" : "+price);

}

@Override

public int getPrice()

{

return price;

}

}

class Composite implements Component

{

String name;

List<Component> components = new ArrayList();

public Composite(String name)

{

super();

this.name = name;

}

public void addComponent(Component com)

{

components.add(com);

}

@Override

public int getPrice()

{

int p=0;

for(Component c:components)

{

p += c.getPrice();

}

return p;

}

@Override

public void showPrice()

{

System.out.println("Composite -> "+name+" : Price -> "+getPrice());

System.out.println("Leaf of " +name);

for(Component c:components)

{

c.showPrice();

}

}

}

public class CarFactory

{

public static void main(String[] args)

{

Component seats = new Leaf("Seats", 40000);

Component chassis = new Leaf("Chassis", 500000);

Component doors = new Leaf("Doors", 100000);

Component trunk = new Leaf("Trunk", 20000);

Component hood = new Leaf("Hood", 50000);

Component engine = new Leaf("Engine", 500000);

Component drivetrain = new Leaf("Drivetrain", 700000);

Component gearbox = new Leaf("Gearbox", 350000);

Composite body = new Composite("Body");

Composite engineBlock = new Composite("Engine Block");

Composite car = new Composite("Car");

body.addComponent(seats);

body.addComponent(chassis);

body.addComponent(doors);

body.addComponent(trunk);

body.addComponent(hood);

engineBlock.addComponent(engine);

engineBlock.addComponent(drivetrain);

engineBlock.addComponent(gearbox);

car.addComponent(body);

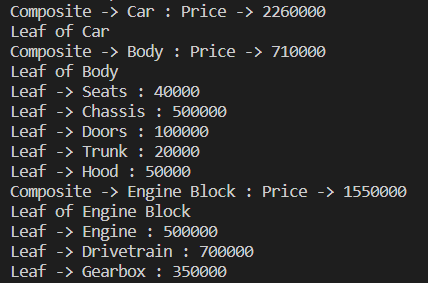
car.addComponent(engineBlock);

car.showPrice();

}

}

* **Output :**



**Facade Design Pattern**

**Assignment - 1**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Façade Design Pattern :**

The facade design pattern is a structural design pattern used in software development. It involves creating a simplified interface that hides the complexities of a system or a set of classes behind it. This interface provides a unified interface to a larger body of code, making it easier to use and understand. In other words, the facade pattern provides a single, simplified interface to a complex system, making it easier for clients to interact with.

The facade design pattern is commonly used in software development when there is a need to provide a simplified interface to a complex subsystem or set of classes. Some common usage scenarios include:

1. Complex API or library: When working with a complex API or library, developers may implement a facade to provide a simpler and more intuitive interface for common tasks, abstracting away the underlying complexity.

2. Legacy code integration: When integrating legacy code into a new system, a facade can be used to encapsulate the legacy code's functionality and provide a modern, streamlined interface for interacting with it.

3. Cross-platform development: Facades can be useful in cross-platform development to provide a unified interface for platform-specific features, allowing developers to write code that works across different platforms without having to deal with platform-specific intricacies directly.

Advantages of using the facade design pattern include:

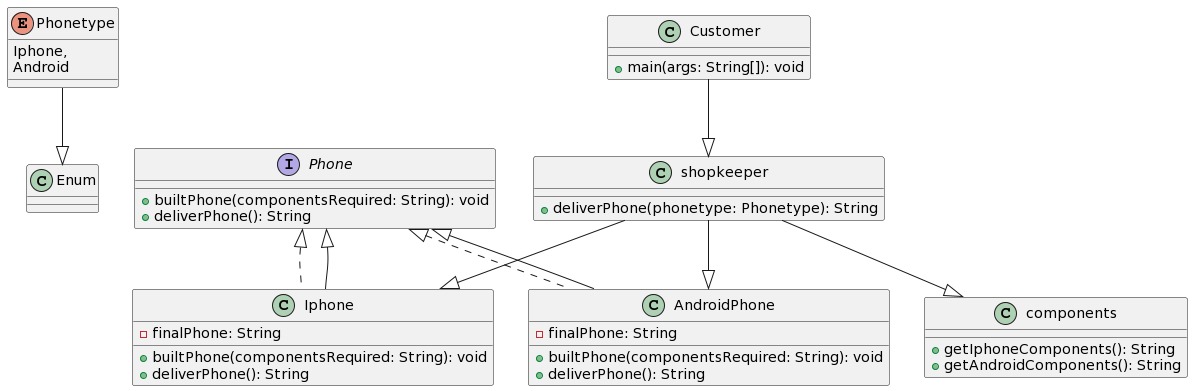
1. Simplified interface: Facades provide a simplified and easy-to-understand interface to a complex system, reducing the learning curve for developers who need to work with the system.

2. Encapsulation of complexity: By encapsulating the complex implementation details behind a facade, developers can hide the complexity of the underlying system and focus on using the provided interface.

3. Improved maintainability: Facades promote code maintainability by decoupling the client code from the implementation details of the subsystem. This makes it easier to make changes to the underlying system without affecting the clients that use it.

4. Enhanced flexibility: Facades can help improve flexibility by allowing developers to change the underlying implementation of a subsystem without affecting the clients that rely on its interface. This makes it easier to evolve the system over time as requirements change.

* **Program :** Implement facade design pattern for phone example.
* **UML Diagram :**



* **Code :**

public enum PhoneType{

iPhone,

Android

}

public interface Phone{

    public void builtPhone(String componentsrequired);

    public String deliverPhone();

}

public class Components {

    public String getiPhoneComponents(){

        return "IOS,3200,5.5";

    }

    public String getAndroidComponents(){

        return "Qualcom,2600,5";

    }

}

public class AndroidPhone implements Phone{

    public String finalPhone;

    @Override

    public void builtPhone(String componentsrequired){

        finalPhone = "Android phone with components: " + componentsrequired;

    }

    @Override

    public String deliverPhone(){

        return finalPhone;

    }

}

public class IPhone implements Phone{

    public String finalPhone;

    @Override

    public void builtPhone(String componentsrequired){

        finalPhone = "Apple phone with components: " + componentsrequired;

    }

    @Override

    public String deliverPhone(){

        return finalPhone;

    }

}

public class Shopkeeper {

    // facade which hides the implementation

    public static String deliverPhone(PhoneType phoneType){

        Components comp = new Components();

        switch(phoneType){

            case iPhone:

                Phone IPh=new IPhone();

                String phComp=comp.getiPhoneComponents();

                IPh.builtPhone(phComp);

                return IPh.deliverPhone();

            case Android:

                Phone Android=new AndroidPhone();

                String PhComp=comp.getAndroidComponents();

                Android.builtPhone(PhComp);

                return Android.deliverPhone();

        }

        return null;

    }

}

public class Customer {

    public static void main(String args[]){

        //Using facade

        System.out.println("--------Facade--------");

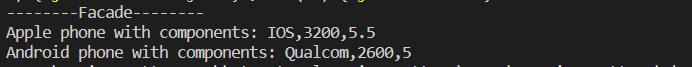
        System.out.println(Shopkeeper.deliverPhone(PhoneType.iPhone));

        System.out.println(Shopkeeper.deliverPhone(PhoneType.Android));

    }

}

* **Output :**

****

**Adapter Design Pattern**

**Assignment - 1**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Adapter Design Pattern :**

The adapter design pattern finds common application in software development across various scenarios. Here's how it can be put:

Usage scenarios:

1. Incorporating third-party libraries: When dealing with external libraries or APIs that don't align with the existing codebase's interfaces, the adapter pattern proves useful. It allows for creating intermediary wrappers that adapt the external interfaces to conform with the application's interface standards.

2. Integration with legacy systems: When modern applications need to interface with legacy systems, the adapter pattern bridges the gap between their disparate interfaces, facilitating seamless communication.

3. Standardizing interfaces: In complex systems with multiple subsystems or modules, each might adhere to its own interface conventions. Employing the adapter pattern helps in standardizing interfaces across various components, simplifying system maintenance and extension.

Advantages:

1. Reusability: Adapters serve as reusable components, capable of being employed across different parts of a system or even in separate projects, as they encapsulate the logic for interface adaptation.

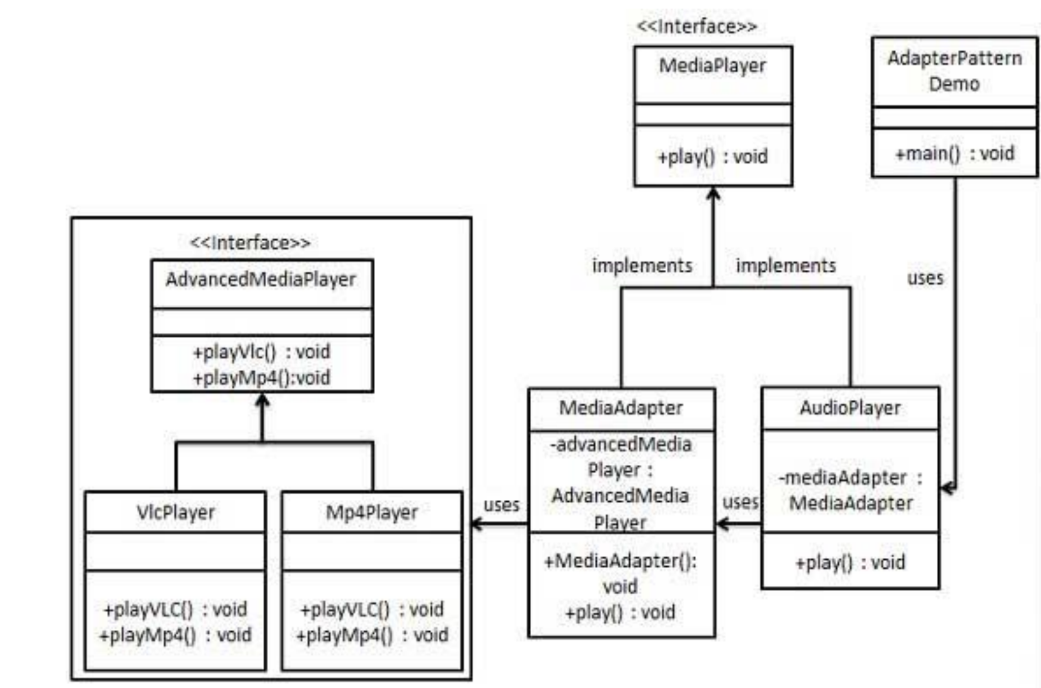
2. Flexibility: By facilitating integration between components boasting different interfaces, the adapter pattern fosters system flexibility, easing maintenance and evolution.

3. Interoperability: By harmonizing incompatible interfaces, the adapter pattern promotes seamless interaction among diverse components, systems, or modules.

4. Encapsulation: Adapters encapsulate the intricacies of interface adaptation, shielding client code from the nitty-gritty details. This enhances code maintainability and comprehensibility.

In summary, the adapter design pattern proves invaluable for seamlessly integrating disparate components or systems featuring incompatible interfaces, all while fostering code reusability and maintainability.

* **Program :** Implement adapter design pattern for Media Player example.
* **UML Diagram :**



* **Code :**

interface MediaPlayer{

    public void play(String audioType, String fileName);

}

interface AdvancedMediaPlayer{

    public void playVlc(String fileName);

    public void playMp4(String fileName);

}

class VlcPlayer implements AdvancedMediaPlayer{

    @Override

    public void playVlc(String fileName){

        System.out.println("Playing VLC file name : " + fileName);

    }

    public void playMp4(String fileName){

        // Do nothing

    }

}

class Mp4Player implements AdvancedMediaPlayer{

    @Override

    public void playVlc(String fileName){

        // Do nothing

    }

    public void playMp4(String fileName){

        System.out.println("Playing MP4 file name : " + fileName);

    }

}

class MediaAdapter implements MediaPlayer{

    AdvancedMediaPlayer advancedMusicPlayer;

    public MediaAdapter(String audioType){

        if (audioType.equalsIgnoreCase("vlc")){

            advancedMusicPlayer=new VlcPlayer();

        }

        else if(audioType.equalsIgnoreCase("mp4")){

            advancedMusicPlayer=new Mp4Player();

        }

    }

    public void play(String audioType, String fileName){

        if (audioType.equalsIgnoreCase("vlc")){

            advancedMusicPlayer.playVlc(fileName);

        }

        else if(audioType.equalsIgnoreCase("mp4")){

            advancedMusicPlayer.playMp4(fileName);

        }

    }

}

class AudioPlayer implements MediaPlayer{

    MediaAdapter mediaAdapter;

    @Override

    public void play(String audioType,String fileName){

        // inbuilt support to play mp3 music files

        if(audioType.equalsIgnoreCase("mp3")){

            System.out.println("Playing MP3 file name : " + fileName);

        }

        else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){

            mediaAdapter = new MediaAdapter(audioType);

            mediaAdapter.play(audioType,fileName);

        }

        else{

            System.out.println("Invalid Media" + audioType + "format not supported.");

        }

    }

}

class Client{

    public static void main(String[] args){

        AudioPlayer audioPlayer = new AudioPlayer();

        audioPlayer.play("mp3" , "Beyond the Horizon.mp3");

        audioPlayer.play("mp4" , "Alone.mp4");

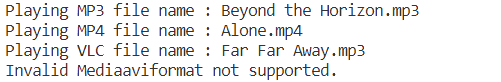
        audioPlayer.play("vlc" , "Far Far Away.mp3");

        audioPlayer.play("avi" , "Mind Me.mp3");

    }

}

* **Output :**

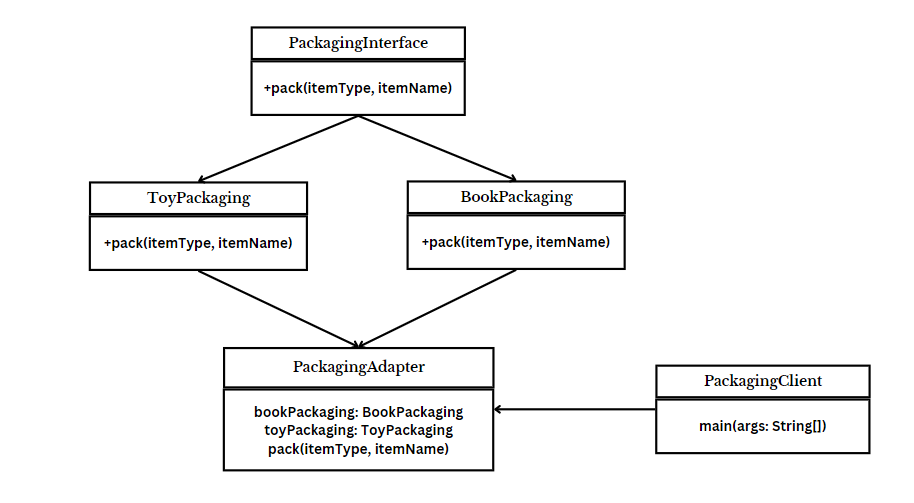


**Adapter Design Pattern**

**Assignment - 2**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Program :** Implement adapter design pattern for Packaging example.
* **UML Diagram :**

ss

* **Code :**

interface PackagingInterface

{

void pack(String itemType, String itemName);

}

class BookPackaging implements PackagingInterface

{

@Override

public void pack(String itemType, String itemName)

{

System.out.println("Packaging " + itemType + ": " + itemName);

}

}

class ToyPackaging implements PackagingInterface

{

@Override

public void pack(String itemType, String itemName)

{

System.out.println("Packaging " + itemType + ": " + itemName);

}

}

class PackagingAdapter implements PackagingInterface

{

BookPackaging bookPackaging;

ToyPackaging toyPackaging;

public PackagingAdapter(String itemType)

{

if (itemType.equalsIgnoreCase("book"))

{

bookPackaging = new BookPackaging();

} else if (itemType.equalsIgnoreCase("toy"))

{

toyPackaging = new ToyPackaging();

}

}

public void pack(String itemType, String itemName)

{

if (itemType.equalsIgnoreCase("book"))

{

bookPackaging.pack(itemType, itemName);

} else if (itemType.equalsIgnoreCase("toy"))

{

toyPackaging.pack(itemType, itemName);

}

}

}

public class PackagingClient

{

public static void main(String[] args)

{

PackagingAdapter packagingInterface = new PackagingAdapter("book");

packagingInterface.pack("book", "The Great Gatsby");

packagingInterface = new PackagingAdapter("toy");

packagingInterface.pack("toy", "LEGO Set");

}

}

* **Output :**



**Flyweight Design Pattern**

**Assignment - 1**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Flyweight Design Pattern :**

The Flyweight Design Pattern is a structural design pattern that focuses on optimizing memory usage by sharing common parts of object state among multiple objects, instead of each object storing its own copy. This pattern is particularly useful when dealing with a large number of similar objects, which would otherwise consume a significant amount of memory if each object maintained its own data.

The key idea behind the Flyweight pattern is to separate intrinsic state (shared state) and extrinsic state (unique state) of an object. The intrinsic state is shared among multiple objects and can be stored externally, while the extrinsic state varies from object to object and must be provided externally when needed.

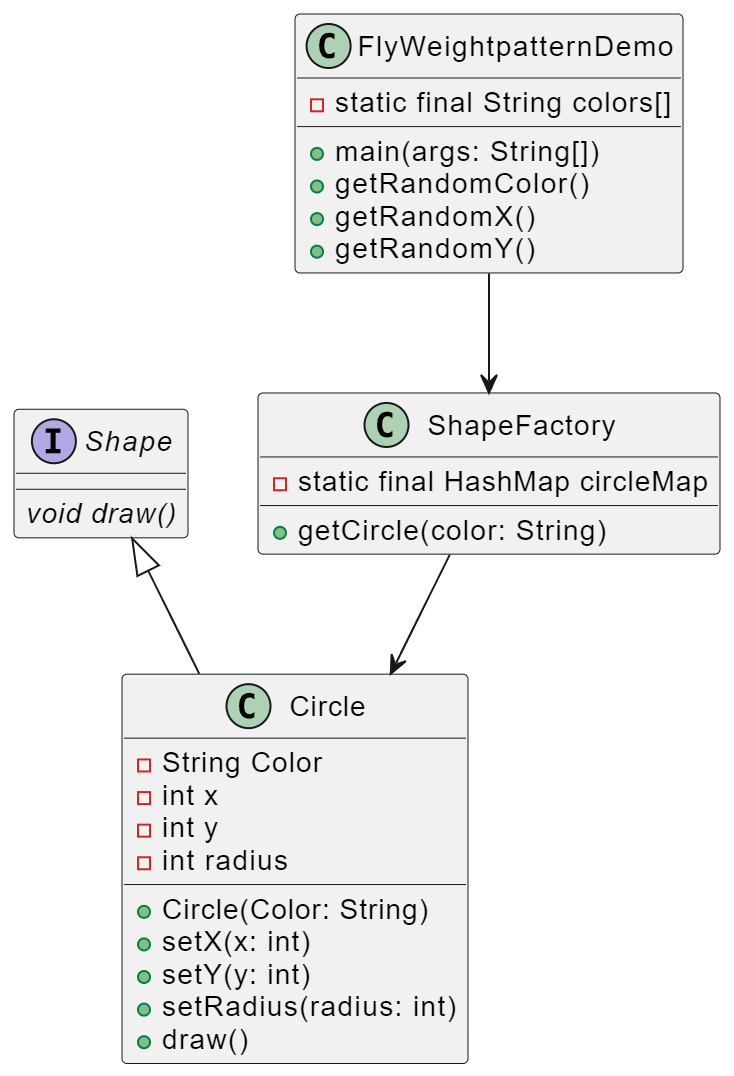
By minimizing memory usage through sharing common state, the Flyweight pattern helps improve performance and reduces memory footprint, especially in scenarios where a large number of similar objects need to be created.

Key components of the Flyweight pattern include:

* Flyweight Interface/Abstract Class: This defines the interface for flyweight objects. It typically includes methods to set and get intrinsic state.
* Concrete Flyweight: This class implements the Flyweight interface and stores the intrinsic state that can be shared among multiple objects.
* Flyweight Factory: This is responsible for managing flyweight objects. It typically maintains a pool of existing flyweight objects and provides a way to retrieve a flyweight object based on certain criteria.
* Client: This is the code that uses flyweight objects. It may need to provide extrinsic state to flyweight objects when interacting with them.

By using the Flyweight pattern, developers can improve the performance and efficiency of their applications, especially in scenarios where large numbers of objects with shared state are involved. However, it's important to carefully consider the trade-offs, as introducing shared state can potentially lead to increased complexity and reduced encapsulation.

* **Program :** Implement flyweight design pattern for Shape example.
* **UML Diagram :**



* **Code :**

public interface Shape

{

void draw();

}

public class Circle implements Shape

{

private String color;

private int x;

private int y;

private int radius;

public Circle(String color)

{

this.color = color;

}

public void setX(int x)

{

this.x = x;

}

public void setY(int y)

{

this.y = y;

}

public void setRadius(int radius)

{

this.radius = radius;

}

@Override

public void draw()

{

System.out.println("Circle: Draw() [Color : "+ color +", x : "+x+", y : "+y+"]");

}

}

import java.util.HashMap;

public class ShapeFactory

{

private static final HashMap circleMap = new HashMap();

public static Shape getCircle(String color)

{

Circle circle = (Circle)circleMap.get(color);

if(circle == null)

{

circle = new Circle(color);

circleMap.put(color, circle);

System.out.println("Creating circle of color : "+color);

}

return circle;

}

}

public class FlyweightPatternDemo

{

private static final String colors[] = { "Red", "Green", "Blue", "White", "Black"};

public static void main(String[] args)

{

for (int i=0; i<20; ++i)

{

Circle circle = (Circle)ShapeFactory.getCircle(getRandomColor());

circle.setX(getRandomX());

circle.setY(getRandomY());

circle.setRadius(100);

circle.draw();

}

}

private static String getRandomColor()

{

return colors[(int)(Math.random()\*colors.length)];

}

private static int getRandomX()

{

return (int)(Math.random()\*100);

}

private static int getRandomY()

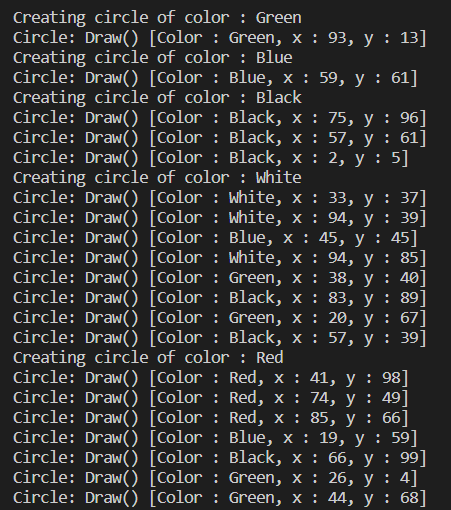
{

return (int)(Math.random()\*100);

}

}

* **Output :**

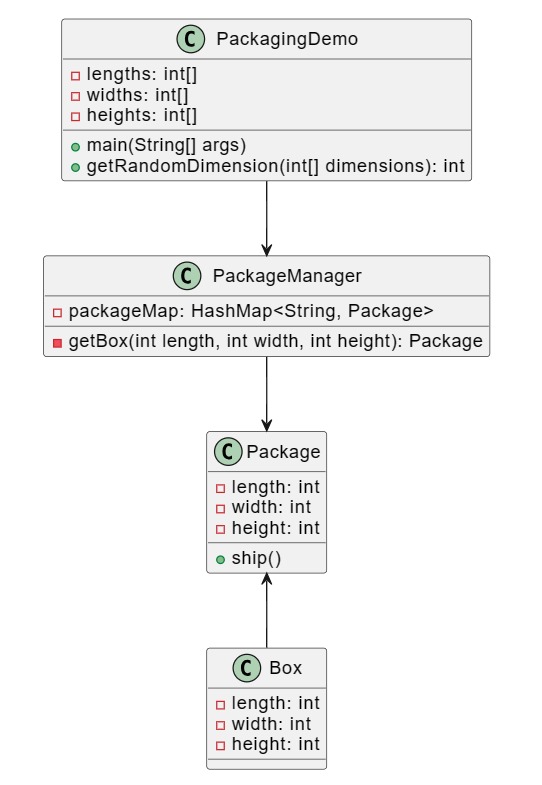


**Flyweight Design Pattern**

**Assignment - 2**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Program :** Implement flyweight design pattern for Packaging example.
* **UML Diagram :**



* **Code :**

import java.util.HashMap;

interface Package

{

void ship();

}

class Box implements Package

{

private int length;

private int width;

private int height;

public Box(int length, int width, int height)

{

this.length = length;

this.width = width;

this.height = height;

}

@Override

public void ship() {

System.out.println("Shipping a box with dimensions: " + length + "x" + width + "x" + height);

}

}

// Package Factory

class PackageManager {

private static final HashMap<String, Package> packageMap = new HashMap<>();

public static Package getBox(int length, int width, int height) {

String key = length + "-" + width + "-" + height;

Package box = packageMap.get(key);

if (box == null) {

box = new Box(length, width, height);

packageMap.put(key, box);

System.out.println("Creating a new box with dimensions: " + key);

}

return box;

}

}

// Main class to demonstrate the usage

class PackagingDemo {

private static final int[] lengths = {10, 20, 30};

private static final int[] widths = {5, 10, 15};

private static final int[] heights = {5, 10, 15};

public static void main(String[] args) {

for (int i = 0; i < 9; ++i) {

int length = getRandomDimension(lengths);

int width = getRandomDimension(widths);

int height = getRandomDimension(heights);

Package box = PackageManager.getBox(length, width, height);

box.ship();

}

}

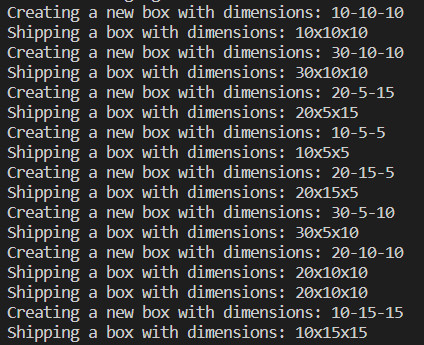
private static int getRandomDimension(int[] dimensions) {

return dimensions[(int) (Math.random() \* dimensions.length)];

}

}

* **Output :**



**Decorator Design Pattern**

**Assignment - 1**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

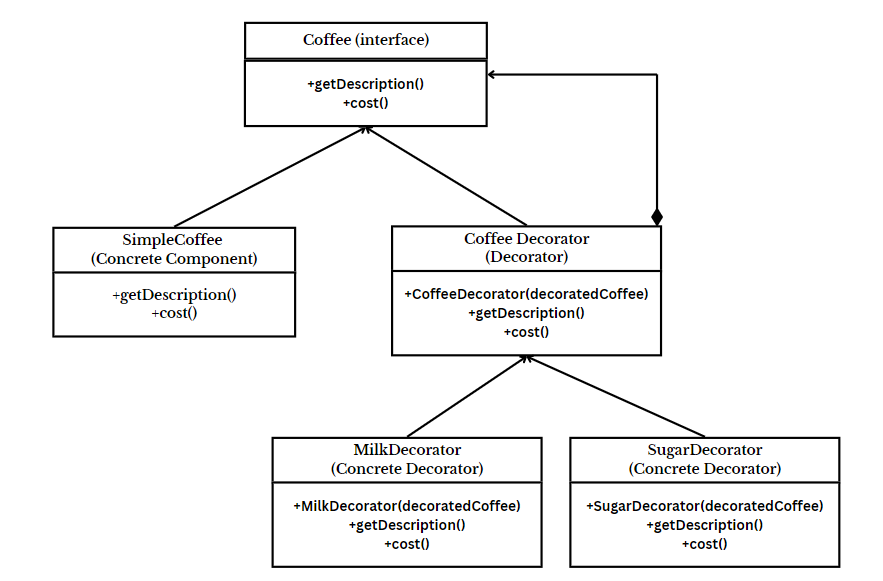
* **Decorator Design Pattern :**

The Decorator Design Pattern is a structural design pattern that allows behavior to be added to individual objects dynamically, without affecting the behavior of other objects from the same class. It's often used when you need to add new functionality to objects at runtime, or when it's impractical to extend them using subclassing.

Here's how the Decorator Pattern works:

1. Component Interface: Define an interface or abstract class for the objects that can have responsibilities added dynamically. This interface usually contains common methods that all components must implement.
2. Concrete Component: Implement the Component interface with a concrete class. This class represents the base object to which additional functionality can be added.
3. Decorator: Create an abstract class (or interface) representing the decorator. This class has a reference to a Component object and implements the Component interface itself. It acts as a base class for all concrete decorators.
4. Concrete Decorators: Implement concrete decorator classes by extending the Decorator class. Each concrete decorator adds its own behavior or responsibilities to the component by overriding methods of the Component interface and calling the methods of the wrapped component.

* **Program :** Implement decorator design pattern for coffee example.
* **UML Diagram :**



* **Code :**

interface Coffee

{

String getDescription();

double cost();

}

class SimpleCoffee implements Coffee

{

@Override

public String getDescription()

{

return "Simple Coffee";

}

@Override

public double cost()

{

return 30.0;

}

}

abstract class CoffeeDecorator implements Coffee

{

protected Coffee decoratedCoffee;

public CoffeeDecorator(Coffee decoratedCoffee)

{

this.decoratedCoffee = decoratedCoffee;

}

@Override

public String getDescription()

{

return decoratedCoffee.getDescription();

}

@Override

public double cost()

{

return decoratedCoffee.cost();

}

}

class MilkDecorator extends CoffeeDecorator

{

public MilkDecorator(Coffee decoratedCoffee)

{

super(decoratedCoffee);

}

@Override

public String getDescription()

{

return super.getDescription() + ", with Milk";

}

@Override

public double cost()

{

return super.cost() + 5.5;

}

}

class SugarDecorator extends CoffeeDecorator

{

public SugarDecorator(Coffee decoratedCoffee)

{

super(decoratedCoffee);

}

@Override

public String getDescription()

{

return super.getDescription() + ", with Sugar";

}

@Override

public double cost()

{

return super.cost() + 3.5;

}

}

public class DecoratorPatternExample

{

public static void main(String[] args)

{

// Create a simple coffee

Coffee coffee = new SimpleCoffee();

System.out.println("Cost: " + coffee.cost() + ", Description: " + coffee.getDescription());

// Decorate the simple coffee with milk

Coffee milkCoffee = new MilkDecorator(coffee);

System.out.println("Cost: " + milkCoffee.cost() + ", Description: " + milkCoffee.getDescription());

// Decorate the simple coffee with sugar

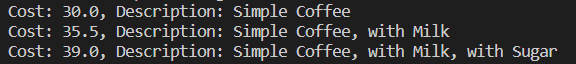
Coffee sweetCoffee = new SugarDecorator(milkCoffee);

System.out.println("Cost: " + sweetCoffee.cost() + ", Description: " + sweetCoffee.getDescription());

}

}

* **Output :**

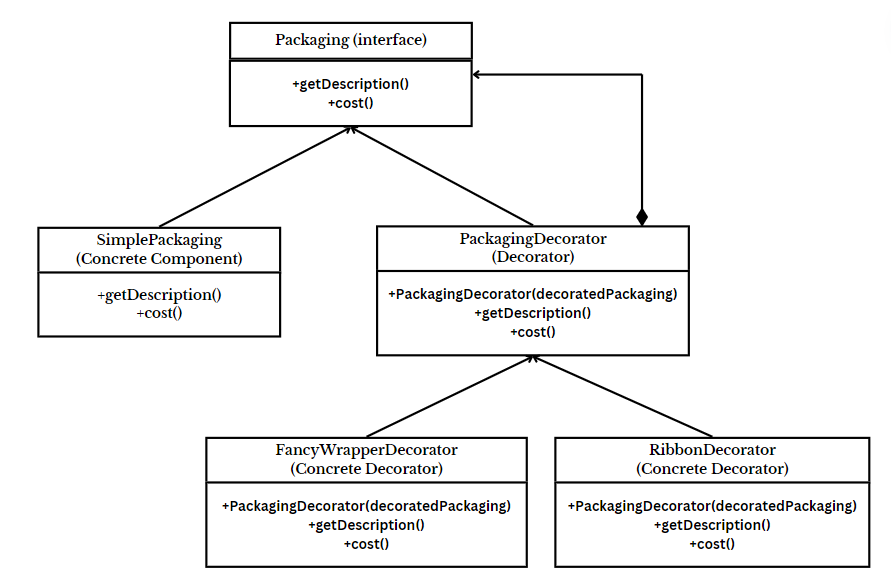


**Decorator Design Pattern**

**Assignment - 2**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Program :** Implement decorator design pattern for packaging example.
* **UML Diagram :**



* **Code :**

interface Packaging {

String getDescription();

double cost();

}

class SimplePackaging implements Packaging {

@Override

public String getDescription() {

return "Simple Packaging";

}

@Override

public double cost() {

return 5.0;

}

}

abstract class PackagingDecorator implements Packaging {

protected Packaging decoratedPackaging;

public PackagingDecorator(Packaging decoratedPackaging) {

this.decoratedPackaging = decoratedPackaging;

}

@Override

public String getDescription() {

return decoratedPackaging.getDescription();

}

@Override

public double cost() {

return decoratedPackaging.cost();

}

}

class FancyWrapperDecorator extends PackagingDecorator {

public FancyWrapperDecorator(Packaging decoratedPackaging) {

super(decoratedPackaging);

}

@Override

public String getDescription() {

return super.getDescription() + ", with Fancy Wrapper";

}

@Override

public double cost() {

return super.cost() + 10.0;

}

}

class RibbonDecorator extends PackagingDecorator {

public RibbonDecorator(Packaging decoratedPackaging) {

super(decoratedPackaging);

}

@Override

public String getDescription() {

return super.getDescription() + ", with Ribbon";

}

@Override

public double cost() {

return super.cost() + 7.0;

}

}

class PackagingDecoratorPatternExample {

public static void main(String[] args) {

// Create a simple packaging

Packaging packaging = new SimplePackaging();

System.out.println("Cost: " + packaging.cost() + ", Description: " + packaging.getDescription());

// Decorate the simple packaging with fancy wrapper

Packaging fancyPackaging = new FancyWrapperDecorator(packaging);

System.out.println("Cost: " + fancyPackaging.cost() + ", Description: " + fancyPackaging.getDescription());

// Decorate the simple packaging with ribbon

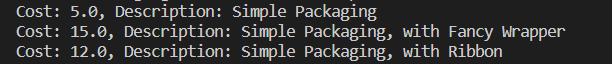
Packaging ribbonPackaging = new RibbonDecorator(packaging);

System.out.println("Cost: " + ribbonPackaging.cost() + ", Description: " + ribbonPackaging.getDescription());

}

}

* **Output :**



**State Design Pattern**

**Assignment - 1**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **State Design Pattern :**

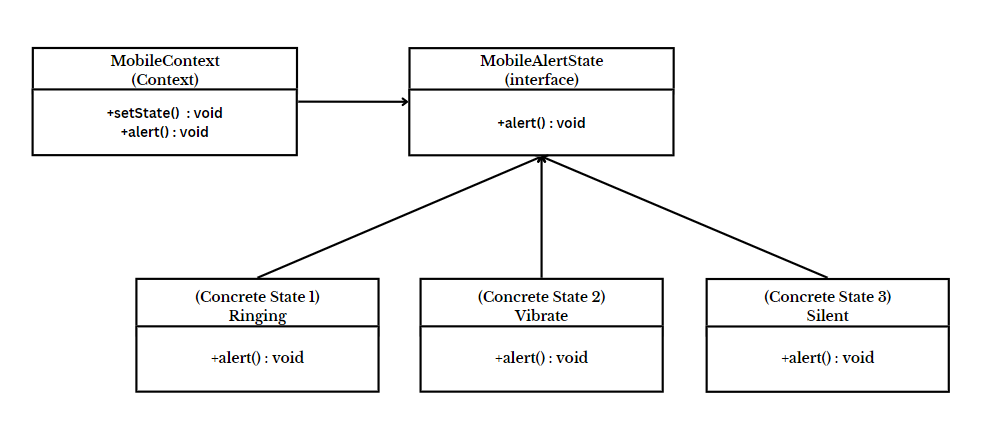
The State Design Pattern is a behavioral design pattern that allows an object to change its behavior when its internal state changes. It is useful when an object's behavior depends on its state and needs to change dynamically based on the state it's in.

The key idea behind the State pattern is to encapsulate each state of an object into a separate class, and the object itself holds a reference to the current state. This way, the object can delegate behavior to the current state class, and when its state changes, it can switch to a different state class seamlessly.

Here's a basic structure of the State Design Pattern:

1. Context: This is the class that maintains the current state and delegates the behavior to the current state object.
2. State: This is an interface or abstract class that defines a set of methods representing the behavior of the context object. Each concrete state subclass implements these methods.
3. Concrete States: These are the concrete implementations of the State interface, each representing a different state of the context object. They implement behavior specific to their state and may trigger transitions to other states.

* **Program :** Implement state design pattern for Mobile example.
* **UML Diagram :**



* **Code :**

interface MobileAlertState

{

public void alert();

}

class Ringing implements MobileAlertState

{

public void alert()

{

System.out.println("Phone is in Ringing Mode.");

}

}

class Silent implements MobileAlertState

{

public void alert()

{

System.out.println("Phone is in Silent Mode.");

}

}

class Vibrate implements MobileAlertState

{

public void alert()

{

System.out.println("Phone is in Vibrate Mode.");

}

}

class MobileContext

{

private MobileAlertState currentState;

public MobileContext()

{

currentState = new Ringing();

}

public void setState(MobileAlertState state)

{

currentState = state;

}

public void alert()

{

currentState.alert();

}

}

class MobileState

{

public static void main(String[] args)

{

MobileContext mc = new MobileContext();

mc.alert();

mc.setState(new Vibrate());

mc.alert();

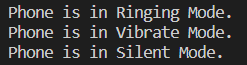
mc.setState(new Silent());

mc.alert();

}

}

* **Output :**

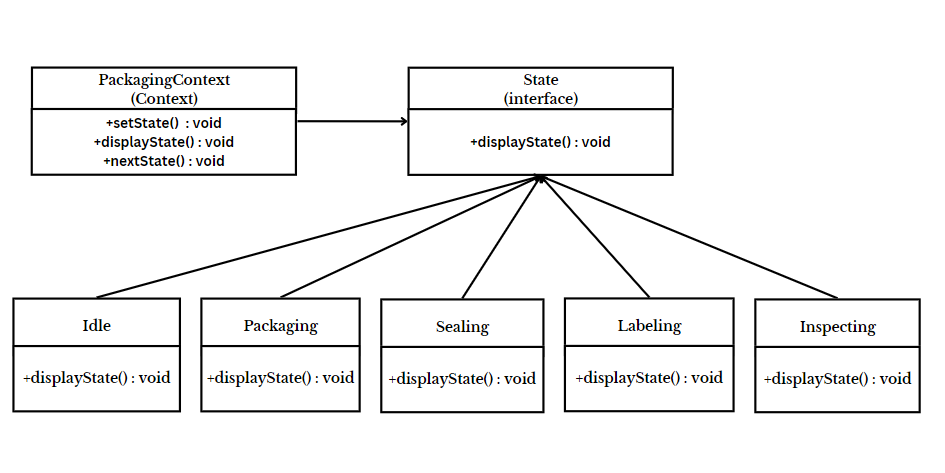


**State Design Pattern**

**Assignment - 2**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Program :** Implement state design pattern for Packaging example.
* **UML Diagram :**



* **Code :**

interface State

{

public void displayState();

}

class Idle implements State

{

public void displayState()

{

System.out.println("Waiting for the package to be processed.");

}

}

class Packaging implements State

{

public void displayState()

{

System.out.println("Item is packed.");

}

}

class Sealing implements State

{

public void displayState()

{

System.out.println("Item is sealed.");

}

}

class Labeling implements State

{

public void displayState()

{

System.out.println("Item is labelled.");

}

}

class Inspecting implements State

{

public void displayState()

{

System.out.println("Inspection of the item is done.");

}

}

class PackagingContext

{

private State currentState;

public PackagingContext()

{

currentState = new Idle();

}

public void setState(State state)

{

currentState = state;

}

public void displayState()

{

currentState.displayState();

}

// This method will give the next state

public void nextState() {

if (currentState instanceof Idle) {

System.out.println("Next state will be: Packaging");

}

else if (currentState instanceof Packaging) {

System.out.println("Next state will be: Sealing");

}

else if (currentState instanceof Sealing) {

System.out.println("Next state will be: Labeling");

}

else if (currentState instanceof Labeling) {

System.out.println("Next state will be: Inspecting");

}

else if (currentState instanceof Inspecting) {

System.out.println("Next state will be: Completed");

}

else {

System.out.println("Invalid state");

}

}

}

class PackagingState

{

public static void main(String[] args)

{

PackagingContext pc = new PackagingContext();

pc.displayState();

pc.setState(new Packaging());

pc.displayState();

pc.nextState();

pc.setState(new Sealing());

pc.displayState();

pc.nextState();

pc.setState(new Labeling());

pc.displayState();

pc.nextState();

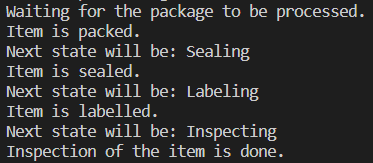
pc.setState(new Inspecting());

pc.displayState();

}

}

* **Output :**



**Observer Design Pattern**

**Assignment - 1**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Observer Design Pattern :**

The Observer Design Pattern is a behavioral design pattern that defines a one-to-many dependency between objects, so that when one object changes state, all its dependents (observers) are notified and updated automatically. This pattern is useful in scenarios where one object's state change should trigger changes in other objects, while keeping the objects loosely coupled.

In the Observer pattern, there are typically three main components:

1. Subject (Observable): This is the object that maintains a list of its dependents (observers), and notifies them of any changes in its state. The subject provides methods for attaching, detaching, and notifying observers.

2. Observer: This is the interface or abstract class defining the update method, which is called by the subject to notify the observer of changes. Observers register with the subject to receive updates.

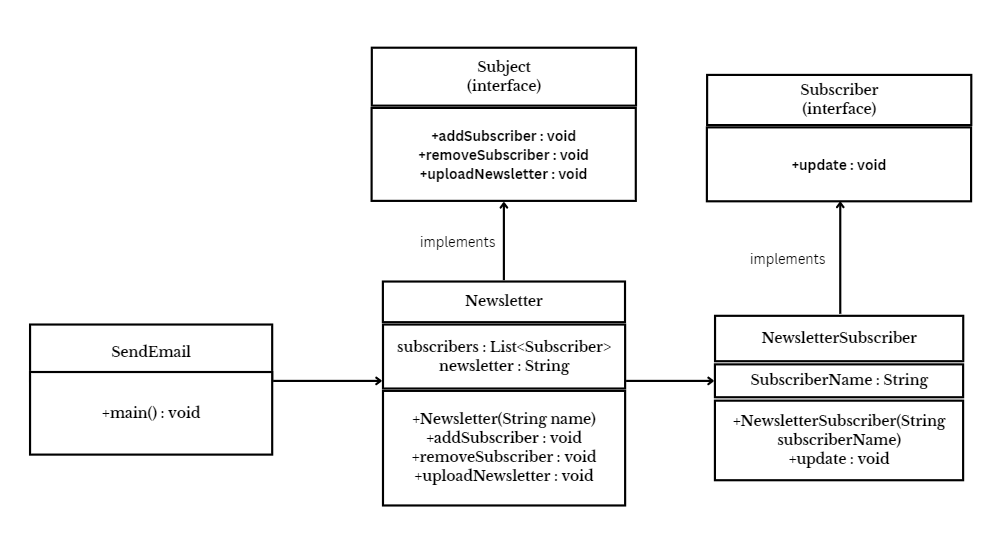
3. Concrete Subject: This is a concrete implementation of the subject interface. It maintains the state and sends notifications to observers when the state changes.

4. Concrete Observer: This is a concrete implementation of the observer interface. It defines the behavior to be executed in response to updates from the subject.

The Observer pattern promotes loose coupling between objects, as the subject doesn't need to know the details of its observers, it only knows they implement a certain interface. This makes it easy to add or remove observers without affecting the subject or other observers.

This pattern is commonly used in user interface frameworks (e.g., event handling in GUIs), distributed systems, and in many other scenarios where the state changes of one object need to be propagated to multiple other objects in a decoupled manner.

* **Program :** Implement observer design pattern for newsletter example.
* **UML Diagram :**



* **Code :**

import java.util.\*;

interface Subject

{

public void addSubscriber(Subscriber subscriber);

public void removeSubscriber(Subscriber subscriber);

public void uploadNewsletter(String newsletter);

}

interface Subscriber

{

public void update(String newsletter);

}

class Newsletter implements Subject

{

private List<Subscriber>subscribers = new ArrayList<>();

public void addSubscriber(Subscriber subscriber)

{

subscribers.add(subscriber);

}

public void removeSubscriber(Subscriber subscriber)

{

subscribers.remove(subscriber);

}

public void uploadNewsletter(String newsletter)

{

System.out.println("Uploading newsletter: "+newsletter);

for (Subscriber subscriber : subscribers)

{

subscriber.update(newsletter);

}

}

}

class NewsletterSubscriber implements Subscriber

{

private String subscriberName;

public NewsletterSubscriber(String subscriberName)

{

this.subscriberName = subscriberName;

}

public void update(String newsletter)

{

System.out.println(subscriberName+" recieved newsletter: "+newsletter);

}

}

class SendEmail

{

public static void main(String[] args)

{

Newsletter newsletter = new Newsletter();

NewsletterSubscriber s1 = new NewsletterSubscriber("Heet");

NewsletterSubscriber s2 = new NewsletterSubscriber("Devanshi");

NewsletterSubscriber s3 = new NewsletterSubscriber("Maitri");

NewsletterSubscriber s4 = new NewsletterSubscriber("Meet");

newsletter.addSubscriber(s1);

newsletter.addSubscriber(s2);

newsletter.addSubscriber(s3);

newsletter.addSubscriber(s4);

newsletter.uploadNewsletter("February Highlights");

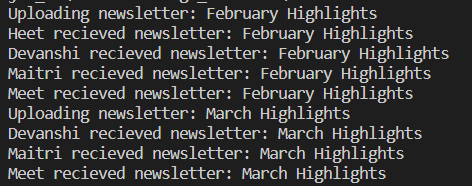
newsletter.removeSubscriber(s1);

newsletter.uploadNewsletter("March Highlights");

}

}

* **Output:**



**Iterator Design Pattern**

**Assignment - 1**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Iterator Design Pattern :**

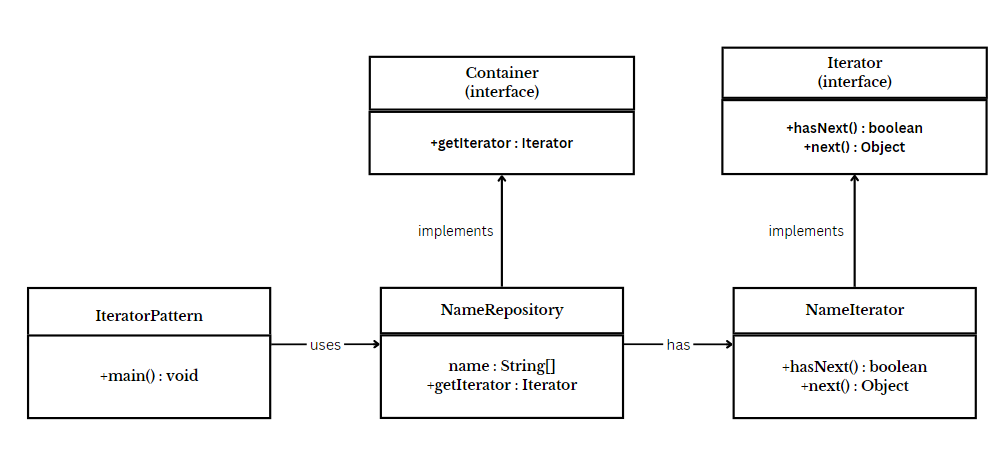
The Iterator Design Pattern is a behavioral design pattern that provides a way to access elements of an aggregate object sequentially without exposing its underlying representation. It is commonly used to traverse collections of objects in a uniform manner without needing to know the internal structure of the collection.

The key components of the Iterator pattern are:

1. Iterator Interface: This interface defines methods for traversing the collection of objects. It typically includes methods like next(), hasNext(), first(), last(), etc., depending on the requirements.
2. Concrete Iterator: This class implements the Iterator interface and maintains the current position in the traversal of the aggregate object. It is responsible for managing the iteration over the collection.
3. Aggregate Interface: This interface defines methods for creating iterators. It could be an abstract class or an interface depending on the design.
4. Concrete Aggregate: This class implements the Aggregate interface and provides methods for creating iterators. It represents a collection of objects that the iterator can traverse.

By using the Iterator pattern, the client code can traverse through elements of a collection without knowing its internal structure. This promotes a clean separation of concerns between the client and the collection, making the code more modular and easier to maintain.

* **Program :** Implement iterator design pattern for name example.
* **UML Diagram :**



* **Code :**

interface Iterator

{

public boolean hasNext();

public Object next();

}

interface Container

{

public Iterator getIterator();

}

class NameRepository implements Container

{

public String names[] = {"Heet", "Devanshi", "Meet", "Maitri"};

public Iterator getIterator()

{

return new NameIterator();

}

class NameIterator implements Iterator

{

int index;

public boolean hasNext()

{

return index < names.length;

}

public Object next()

{

if (this.hasNext())

{

return names[index++];

}

return null;

}

}

}

public class IteratorDesign

{

public static void main(String[] args)

{

NameRepository nameRepository = new NameRepository();

for (Iterator iter = nameRepository.getIterator(); iter.hasNext(); )

{

String name = (String) iter.next();

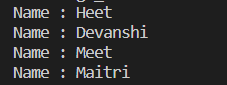
System.out.println("Name : " + name);

}

}

}

* **Output :**



**Memento Design Pattern**

**Assignment - 1**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Memento Design Pattern :**

The Memento design pattern is a behavioral pattern used in software engineering. It allows you to capture and externalize an object's internal state so that the object can be restored to that state later, without violating encapsulation. This pattern is especially useful when you need to implement undo mechanisms or save and restore functionality in an application.

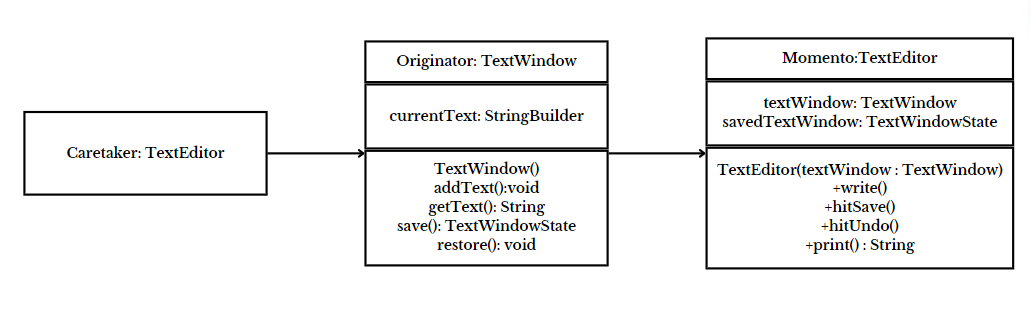
Here's a brief overview of how the Memento pattern typically works:

1. Originator: This is the object whose state needs to be saved. It creates a memento object containing a snapshot of its current state.
2. Memento: This is an immutable object that stores the state of the originator. It provides methods for getting the state and possibly applying it back to the originator.

3. Caretaker: This is responsible for keeping track of mementos. It can store multiple mementos, usually in a stack or some other data structure. It doesn't modify or examine the contents of the memento.

By using this pattern, you can ensure that the encapsulation of the originator is not compromised while still providing the ability to save and restore its state. It promotes separation of concerns by keeping the responsibility of managing state separate from the originator itself.

* **Program :** Implement memento design pattern for word example.
* **UML Diagram :**



* **Code :**

class TextWindow

{

private StringBuilder currentText;

public TextWindow()

{

this.currentText = new StringBuilder();

}

public void addText(String text)

{

currentText.append(text);

}

public String getText()

{

return currentText.toString();

}

public TextWindowState save()

{

return new TextWindowState(currentText.toString());

}

public void restore(TextWindowState save)

{

currentText = new StringBuilder(save.getText());

}

}

class TextWindowState

{

private String text;

public TextWindowState(String text)

{

this.text = text;

}

public String getText()

{

return text;

}

}

class TextEditor

{

private TextWindow textWindow;

private TextWindowState savedTextWindow;

public TextEditor(TextWindow textWindow)

{

this.textWindow = textWindow;

}

public void write(String text)

{

textWindow.addText(text);

}

public void hitSave()

{

savedTextWindow = textWindow.save();

}

public void hitUndo()

{

textWindow.restore(savedTextWindow);

}

public String print()

{

return textWindow.getText();

}

}

class Main

{

public static void main(String[] args)

{

TextEditor textEditor = new TextEditor(new TextWindow());

textEditor.write("The Memento Design Pattern \n");

textEditor.write("Initial State: Welcome \n");

textEditor.hitSave();

textEditor.write("Add some more text to the document\n");

System.out.println(textEditor.print());

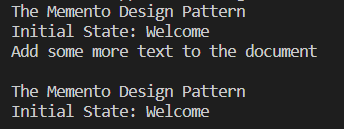
textEditor.hitUndo();

System.out.println(textEditor.print());

}

}

* **Output :**



**MVC Design Pattern**

**Assignment - 1**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **MVC Design Pattern :**

The MVC (Model-View-Controller) design pattern is a software architectural pattern commonly used in developing user interfaces. It divides an application into three interconnected components to separate the internal representations of information from the ways that information is presented and accepted by the user.

Here's a brief overview of each component in the MVC pattern:

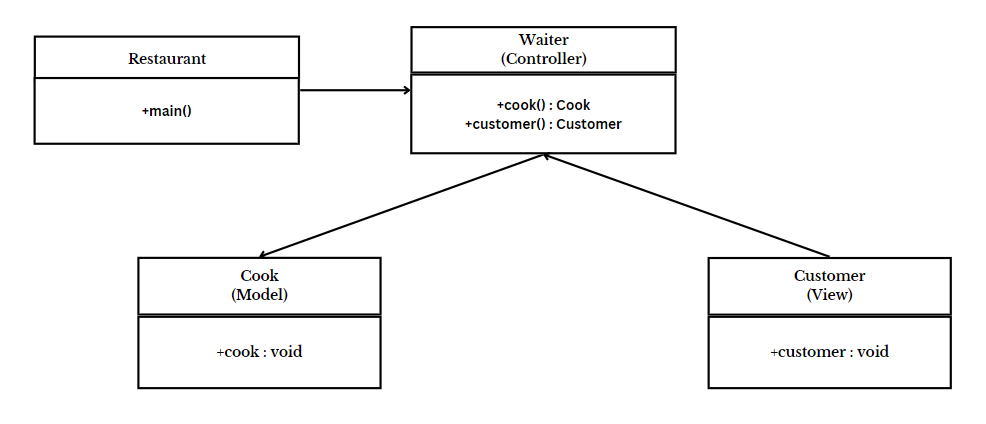
1. Model: The Model represents the application's data and core business logic. It is responsible for managing the data, processing it, and responding to requests for information about its state. The Model component does not depend on the user interface or how the data is presented. Instead, it focuses on the application's behavior and state.

2. View: The View is responsible for the presentation of the data to the user. It displays the data from the Model in a format suitable for interaction, such as a graphical interface or a web page. The View component is designed to be as lightweight as possible, often retrieving data from the Model and rendering it to the user.

1. Controller:
2. The Controller acts as an intermediary between the Model and the View. It receives user input from the View, processes that input (such as handling button clicks or form submissions), and updates the Model accordingly. The Controller is responsible for updating the View to reflect changes in the Model's state. It essentially controls the flow of the application, deciding which actions to take in response to user input.

The MVC pattern promotes a clean separation of concerns, making applications easier to develop, test, and maintain. It's widely used in various software development frameworks and environments, including web development, desktop applications, and mobile apps.

* **Program :** Implement MVC design pattern for restaurant example.
* **UML Diagram :**



* **Code :**

import java.util.\*;

class Cook

{

public void prepareFood(String dish)

{

System.out.println("Preparing " + dish + "...");

System.out.println(dish + " is ready!");

}

}

class Customer

{

public void placeOrder(String dish)

{

System.out.println("Placing order for " + dish);

}

}

class Waiter

{

private Cook cook;

private Customer customer;

public Waiter()

{

this.cook = new Cook();

this.customer = new Customer();

}

public void takeOrder(String dish)

{

customer.placeOrder(dish);

cook.prepareFood(dish);

System.out.println(dish + " is served!");

}

}

class RestaurantExample

{

public static void main(String[] args)

{

String dish;

Scanner sc = new Scanner(System.in);

System.out.println("Enter the dish you want to order: ");

dish = sc.nextLine();

Waiter waiter = new Waiter();

waiter.takeOrder(dish);

}

}

* **Output :**

