**Day 25 – August 09th**

**TASK1 – Bridge Method**

The **Bridge Pattern** is a **structural design pattern** that **decouples abstraction from implementation**, so they can evolve independently.

Instead of having one big class hierarchy that mixes **what** an object does and **how** it does it, we separate them into two different hierarchies:

* **Abstraction** → Defines the high-level control logic (what to do).
* **Implementation** → Defines the low-level operations (how to do it).

**//Implementor Interface**

* package BridgeMethod;  
    
  public interface ExcalidrawAPI  
  {  
   void drawSquare(int s);  
  }

**//Concrete Implementors**

public class DrawingPicture implements ExcalidrawAPI{  
 @Override  
 public void drawSquare(int s) {  
 System.*out*.println("draw square in Drawing Picture using Excalidraw with side "+ s);  
  
 }  
}

public class DrawingFrame implements ExcalidrawAPI {  
 @Override  
 public void drawSquare(int s) {  
 System.*out*.println("draw square in Drawing frame using Excalidraw with side "+ s);  
 }  
}

**//Abstraction**

abstract public class Shape {  
 protected ExcalidrawAPI excalidrawAPI;  
  
 protected Shape(ExcalidrawAPI excalidrawAPI) {  
 this.excalidrawAPI = excalidrawAPI;  
 }  
 abstract void draw();  
}

public class Square extends Shape {  
 private int s;  
  
 // Constructor  
 public Square(int s, ExcalidrawAPI excalidrawAPI) {  
 super(excalidrawAPI); // Pass API object to Shape  
 this.s = s;  
 }  
  
 @Override  
 void draw() {  
 excalidrawAPI.drawSquare(s); // Access from Shape  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 System.*out*.println("Bridge Method Design Pattern - Structural DP!");  
 ExcalidrawAPI obj1 = new DrawingFrame();  
 ExcalidrawAPI obj2 = new DrawingPicture();  
  
 Shape square = new Square(5, obj1);  
 Shape square2 = new Square(5, obj2);  
  
  
 square.draw();  
 }  
}

**Output**

Bridge Method Design Pattern - Structural DP!

draw square in Drawing frame using Excalidraw with side 5

**TASK 2 – Composite Method**

**Concept**

The **Composite Pattern** is a **structural design pattern** that lets you treat **individual objects (leaf nodes)** and **groups of objects (composite nodes)** **uniformly**.

It’s perfect for representing **tree-like hierarchies** — e.g., a company with departments and sub-departments.

**Why use Composite?**

* You can **add, remove, and traverse** elements in the hierarchy the same way, regardless of whether they are individual objects or collections.
* Reduces complexity for the client code.
* Makes hierarchical data easy to manage.

**Structure**

* **Component (Interface/Abstract class)** → Declares operations that both leaf and composite classes will implement.
* **Leaf** → Represents end objects (no children). Implements component interface.
* **Composite** → Holds child components (can be leaves or other composites). Implements component interface.

**//Component Interface**

public interface Company {  
 void displayName();  
}

**//Leaf Classes**

public class Software implements Company {  
 private int id;  
 private String name; // variable names in Java typically start lowercase  
  
 // Constructor  
 public Software(int id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 // Implementing interface method  
 @Override  
 public void displayName() {  
 System.*out*.println(getClass().getSimpleName());  
 }  
  
 // Getters  
 public int getId() {  
 return id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 // Setters  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
}

public class HR implements Company {  
 private int id;  
 private String name;  
  
 // Constructor  
 public HR(int id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 // Implementing the Company interface method  
 @Override  
 public void displayName() {  
 System.*out*.println(getClass().getSimpleName());  
 }  
  
 // Getters  
 public int getId() {  
 return id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 // Setters  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
}

**//Composite class**

import java.util.ArrayList;  
import java.util.List;  
// composite component  
public class CompanyHead implements Company{  
 private int id;  
 private String name;  
  
 private List<Company> subDepartments;  
 public CompanyHead(int id, String name) {  
 this.id = id;  
 this.name= name;  
 this.subDepartments = new ArrayList<>();  
 }  
 public void displayName() {  
 subDepartments.forEach(Company::displayName);  
 }  
 public void addDepartments(Company company) {  
 subDepartments.add(company);  
 }  
 // remove details  
}

**//Client**

public class Main {  
 public static void main(String[] args) {  
 System.*out*.println("Composite Method DP - Structural DP");  
  
 Company softwareCompany = new Software(1, "Software");  
 Company hrCompany = new HR(2, "Human Resources");  
  
 CompanyHead companyHead = new CompanyHead(3, "ABC Company");  
 companyHead.addDepartments(softwareCompany);  
 companyHead.addDepartments(hrCompany);  
  
 companyHead.displayName();  
 }  
}

**Output**

Composite Method DP - Structural DP

Software

HR

**TASK3 – Façade Method**

The **Facade Pattern** is a **structural design pattern** that provides **a single simplified interface** to a complex subsystem.

Instead of exposing the client to **multiple classes and methods** directly, you create **one facade class** that handles all the interactions internally.

**When to Use**

* The system has a **lot of classes** with complicated interactions.
* You want to **decouple client code** from subsystem implementation.
* You want to **reduce learning curve** for using your API.

**//Façade class**

public class MallFacade {  
 private StoreStaffService storeStaffService;  
 private BillingCounterService billingCounterService;  
  
 public MallFacade() {  
 this.storeStaffService = new StoreStaffService();  
 this.billingCounterService = new BillingCounterService();  
 }  
 public void getItems(String items) {  
 storeStaffService.getItems(items);  
 }  
 public void payBill(String accountId, String billId, double amount){  
 billingCounterService.payBill(accountId,billId, amount );  
 }  
}

**//Subsystems**

public class StoreStaffService {  
 public void getItems(String items) {  
 System.*out*.println("selecting Mangoes");  
 }  
}

public class BillingCounterService {  
 public void payBill(String accountId, String billId, double amount) {  
 System.*out*.println("paying for Mangoes "+ amount +" for billId "+ billId + " from account " +accountId);  
 }  
}

**//Client**

public class Main {  
 public static void main(String[] args) {  
 System.*out*.println("Facade Method DP - Structural Design Pattern");  
 MallFacade mallFacade = new MallFacade();  
 mallFacade.getItems("Fruits");  
 // mallFacade.transferMoney("123", "7777", 500.0);  
 mallFacade.payBill("123", "billno:44 ", 500.0);  
 }  
}

**Output**

Facade Method DP - Structural Design Pattern

selecting Mangoes

paying for Mangoes 500.0 for billId billno:44 from account 123

**HOMETASK1 – Bridge Method**

### ****Implementor Interface****

public interface Device {

void turnOn();

void turnOff();

void setVolume(int level);

}

**//Concrete Implementors**

public class TV implements Device {

@Override

public void turnOn() {

System.out.println("TV is turned ON");

}

@Override

public void turnOff() {

System.out.println("TV is turned OFF");

}

@Override

public void setVolume(int level) {

System.out.println("TV volume set to " + level);

}

}

public class Radio implements Device {

@Override

public void turnOn() {

System.out.println("Radio is turned ON");

}

@Override

public void turnOff() {

System.out.println("Radio is turned OFF");

}

@Override

public void setVolume(int level) {

System.out.println("Radio volume set to " + level);

}

}

**//Abstraction**

public abstract class RemoteControl {

protected Device device; // bridge to implementation

protected RemoteControl(Device device) {

this.device = device;

}

public abstract void turnOn();

public abstract void turnOff();

public abstract void setVolume(int level);

}

**//Refined Abstraction**

public class BasicRemote extends RemoteControl {

public BasicRemote(Device device) {

super(device);

}

@Override

public void turnOn() {

device.turnOn();

}

@Override

public void turnOff() {

device.turnOff();

}

@Override

public void setVolume(int level) {

device.setVolume(level);

}

}

public class AdvancedRemote extends RemoteControl {

public AdvancedRemote(Device device) {

super(device);

}

@Override

public void turnOn() {

System.out.println("Advanced features enabled.");

device.turnOn();

}

@Override

public void turnOff() {

device.turnOff();

}

@Override

public void setVolume(int level) {

System.out.println("Setting volume with advanced controls.");

device.setVolume(level);

}

}

**//Client**

public class Main {

public static void main(String[] args) {

RemoteControl tvRemote = new BasicRemote(new TV());

tvRemote.turnOn();

tvRemote.setVolume(15);

tvRemote.turnOff();

System.out.println();

RemoteControl radioRemote = new AdvancedRemote(new Radio());

radioRemote.turnOn();

radioRemote.setVolume(8);

radioRemote.turnOff();

}

}

**Output**

**TV is turned ON**

**TV volume set to 15**

**TV is turned OFF**

**Advanced features enabled.**

**Radio is turned ON**

**Setting volume with advanced controls.**

**Radio volume set to 8**

**Radio is turned OFF**

**HOMETASK2 – Fly Weight Method**

The **Flyweight Pattern** is a **structural design pattern** used to **minimize memory usage** by sharing as much data as possible between similar objects.

Instead of creating a new object every time, it **reuses** an existing one when possible.

**When to Use**

* You have **a huge number of objects**.
* Many of these objects share **common (intrinsic) state**.
* The differing state can be passed **externally (extrinsic)** when needed.

**Key Terms**

* **Intrinsic state** → Shared, stored inside the flyweight object (does not change across objects).
* **Extrinsic state** → Supplied by the client at runtime (changes per use).

**//Fly weight Interface**

public interface Shape {  
 void draw(int x, int y); // extrinsic state: position  
}

**//Concrete Flyweight**

public class Circle implements Shape {  
 private String color; // intrinsic state  
  
 public Circle(String color) {  
 this.color = color;  
 }  
  
 @Override  
 public void draw(int x, int y) {  
 System.*out*.println("Drawing " + color + " circle at (" + x + ", " + y + ")");  
 }  
}

**//Flyweight Factory**

import java.util.HashMap;  
import java.util.Map;  
  
public class ShapeFactory {  
 private static final Map<String, Shape> *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 new Circle of color: " + color);  
 }  
 return circle;  
 }  
}

**//Client**

public class Main {  
 private static final String[] *colors* = {"Red", "Green", "Blue"};  
  
 public static void main(String[] args) {  
 for (int i = 0; i < 6; i++) {  
 Shape circle = ShapeFactory.*getCircle*(*colors*[i % *colors*.length]);  
 int x = (int) (Math.*random*() \* 100);  
 int y = (int) (Math.*random*() \* 100);  
 circle.draw(x, y); // extrinsic state: position  
 }  
 }  
}

**Output**

Creating new Circle of color: Red

Drawing Red circle at (21, 29)

Creating new Circle of color: Green

Drawing Green circle at (90, 90)

Creating new Circle of color: Blue

Drawing Blue circle at (63, 53)

Drawing Red circle at (48, 38)

Drawing Green circle at (51, 83)

Drawing Blue circle at (41, 26)

**HOMETASK3 – Decorator Method**

## ****Decorator Pattern – Concept****

* **Purpose:** Add new responsibilities/behaviors to an object **dynamically** without altering its existing code.
* **Key idea:** You wrap the original object inside another object (the decorator), which adds functionality before/after delegating work to the original.
* **Difference from inheritance:** Instead of making new subclasses for every variation, you wrap objects at runtime.

**// Component Interface**

public interface Coffee {  
 String getDescription();  
 double getCost();  
}

**//Concrete Component**

public class SimpleCoffee implements Coffee {  
 @Override  
 public String getDescription() {  
 return "Simple Coffee";  
 }  
  
 @Override  
 public double getCost() {  
 return 2.0;  
 }  
}

**//Decorator Base class**

public abstract class CoffeeDecorator implements Coffee {  
 protected Coffee decoratedCoffee; // Wraps the original  
  
 public CoffeeDecorator(Coffee coffee) {  
 this.decoratedCoffee = coffee;  
 }  
  
 @Override  
 public String getDescription() {  
 return decoratedCoffee.getDescription();  
 }  
  
 @Override  
 public double getCost() {  
 return decoratedCoffee.getCost();  
 }  
}

**//Concrete Decorators**

public class SugarDecorator extends CoffeeDecorator {  
 public SugarDecorator(Coffee coffee) {  
 super(coffee);  
 }  
  
 @Override  
 public String getDescription() {  
 return super.getDescription() + ", Sugar";  
 }  
  
 @Override  
 public double getCost() {  
 return super.getCost() + 0.2;  
 }  
}

public class MilkDecorator extends CoffeeDecorator {  
 public MilkDecorator(Coffee coffee) {  
 super(coffee);  
 }  
  
 @Override  
 public String getDescription() {  
 return super.getDescription() + ", Milk";  
 }  
  
 @Override  
 public double getCost() {  
 return super.getCost() + 0.5;  
 }  
}

**//Client**

public class Main {  
 public static void main(String[] args) {  
 Coffee coffee = new SimpleCoffee();  
 System.*out*.println(coffee.getDescription() + " -> $" + coffee.getCost());  
  
 coffee = new MilkDecorator(coffee);  
 System.*out*.println(coffee.getDescription() + " -> $" + coffee.getCost());  
  
 coffee = new SugarDecorator(coffee);  
 System.*out*.println(coffee.getDescription() + " -> $" + coffee.getCost());  
 }  
}

**Output**

Simple Coffee -> $2.0

Simple Coffee, Milk -> $2.5

Simple Coffee, Milk, Sugar -> $2.7