Heaven's Light Is Our Guide

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Course Code: CSE 3206

Course Title: Software Engineering Sessional

Lab Report: Design Pattern (7,8,9)

Submitted by-

Name	Roll
S.M. SHAREAR RAHMAN	2003157
GOURAB KANTI SAHA	2003158
TAJUL ISLAM	2003159

Submitted to-

Farjana Parvin
Lecturer,
Department of CSE, RUET

Bridge Structural Design Pattern:

Bridge Design Pattern is a structural design pattern that split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

The pattern involves two main components abstraction and implementation layer. Abstraction (also called interface) is a high-level control layer for some entity. This layer isn't supposed to do any real work on its own. It should delegate the work to the implementation layer (also called platform) which implements all the low-level functionality.

The Bridge Design Pattern allows for creating platform-independent classes and apps. In this design pattern client work with only the high-level abstractions. This pattern follows the Open/Closed Principle, enabling new abstractions and implementations to be added independently. The Bridge pattern attempts to solve this problem by switching from inheritance to composition.

Example with and without the Bridge Design Pattern

Let's consider an example where a **Device** can be controlled with different types of **Remotes**. The Remote is the **abstraction**, and the Device is the **implementation**. The Device can be controlled using different types of remote devices.

Without the Bridge Design Pattern:

Without the Bridge pattern, we would need to create separate subclasses for each combination of **Device** and **Remote** ,which leads to a large class hierarchy.

```
// RemoteControl tightly coupled with specific devices
class RemoteControl {
    private Tv tv;
    private Radio radio;
```

```
    private String deviceType; // Tracks which device is currently in use

    public RemoteControl(String deviceType) {
        this.deviceType = deviceType;
        if (deviceType.equals("Tv")) {
            this.tv = new Tv();
        } else if (deviceType.equals("Radio")) {
            this.radio = new Radio();
        }
    }

    public void togglePower() {
        if (deviceType.equals("Tv")) {
            if (tv.isEnabled()) {
                tv.disable();
            } else {
                tv.enable();
            }
        } else if (deviceType.equals("Radio")) {
            if (radio.isEnabled()) {
                radio.disable();
            } else {
                radio.enable();
            }
        }
    }

    public void volumeUp() {
        if (deviceType.equals("Tv")) {
            tv.setVolume(tv.getVolume() + 10);
        } else if (deviceType.equals("Radio")) {
            radio.setVolume(radio.getVolume() + 10);
        }
    }

    public void volumeDown() {
        if (deviceType.equals("Tv")) {
            tv.setVolume(tv.getVolume() - 10);
        } else if (deviceType.equals("Radio")) {
            radio.setVolume(radio.getVolume() - 10);
        }
    }

    public void channelUp() {
        if (deviceType.equals("Tv")) {
```

```
            tv.setChannel(tv.getChannel() + 1);
        } else if (deviceType.equals("Radio")) {
            radio.setChannel(radio.getChannel() + 1);
        }
    }

    public void channelDown() {
        if (deviceType.equals("Tv")) {
            tv.setChannel(tv.getChannel() - 1);
        } else if (deviceType.equals("Radio")) {
            radio.setChannel(radio.getChannel() - 1);
        }
    }
}

// Tv class
class Tv {
    private boolean on = false;
    private int volume = 50;
    private int channel = 1;

    public boolean isEnabled() {
        return on;
    }

    public void enable() {
        on = true;
        System.out.println("TV is now ON.");
    }

    public void disable() {
        on = false;
        System.out.println("TV is now OFF.");
    }

    public int getVolume() {
        return volume;
    }

    public void setVolume(int percent) {
        volume = Math.max(0, Math.min(100, percent));
        System.out.println("TV volume set to " + volume + "%.");
    }

    public int getChannel() {
```

```
        return channel;
    }

    public void setChannel(int channel) {
        this.channel = channel;
        System.out.println("TV channel set to " + channel + ".");
    }
}

// Radio class
class Radio {
    private boolean on = false;
    private int volume = 30;
    private int channel = 1;

    public boolean isEnabled() {
        return on;
    }

    public void enable() {
        on = true;
        System.out.println("Radio is now ON.");
    }

    public void disable() {
        on = false;
        System.out.println("Radio is now OFF.");
    }

    public int getVolume() {
        return volume;
    }

    public void setVolume(int percent) {
        volume = Math.max(0, Math.min(100, percent));
        System.out.println("Radio volume set to " + volume + "%.");
    }

    public int getChannel() {
        return channel;
    }

    public void setChannel(int channel) {
        this.channel = channel;
        System.out.println("Radio channel set to " + channel + ".");
```

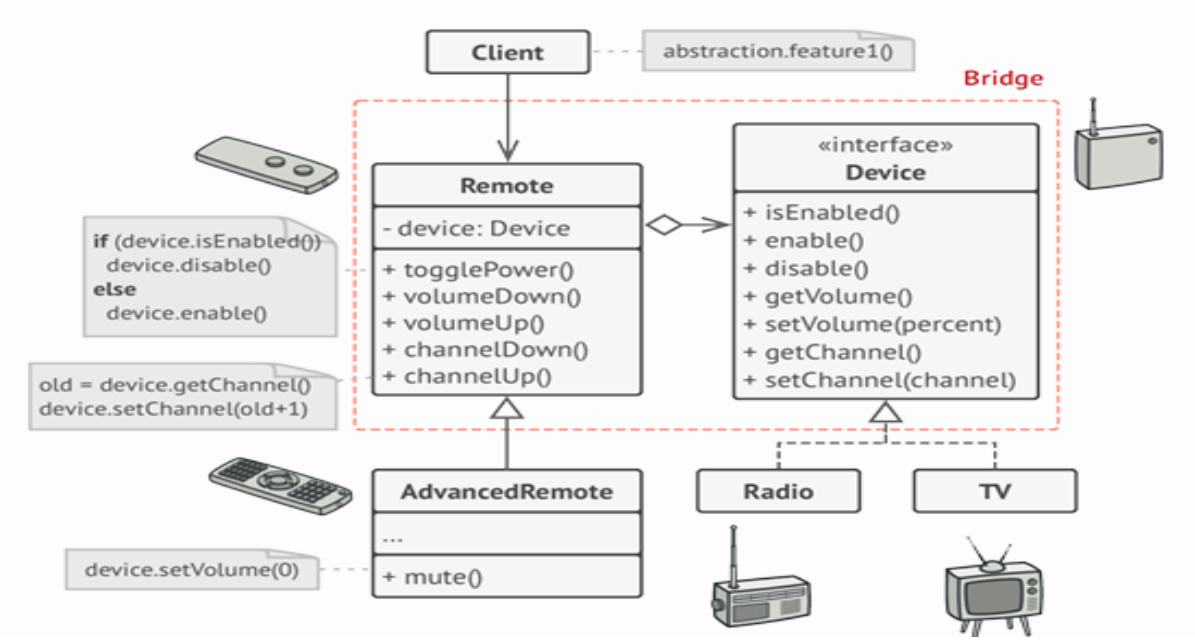
```
    }
}

// Client code
public class WithoutBridgePattern {
    public static void main(String[] args) {
        RemoteControl tvRemote = new RemoteControl("Tv");

        tvRemote.togglePower();
        tvRemote.volumeUp();
        tvRemote.channelUp();
        tvRemote.togglePower();

        RemoteControl radioRemote = new RemoteControl("Radio");

        radioRemote.togglePower();
        radioRemote.volumeUp();
        radioRemote.channelDown();
        radioRemote.togglePower();
    }
}
```

Output:

```
TV is now ON.
TV volume set to 60%.
TV channel set to 2.
TV is now OFF.
Radio is now ON.
Radio volume set to 40%.
Radio channel set to 0.
Radio is now OFF.
```

With the Bridge Design Pattern:

With the Bridge pattern, it can design separating the abstraction (the **Remote**) from the implementation (the **Device**), allowing the **Remote** to change its implementation without needing to create a new class for each combination.



UML Design

```
// Abstraction hierarchy
abstract class RemoteControl {
    protected Device device;

    public RemoteControl(Device device) {
        this.device = device;
    }

    public void togglePower() {
        if (device.isEnabled()) {
            device.disable();
        } else {
            device.enable();
        }
    }

    public void volumeDown() {
        device.setVolume(device.getVolume() - 10);
    }

    public void volumeUp() {
        device.setVolume(device.getVolume() + 10);
    }
```

```
    public void channelDown() {
        device.setChannel(device.getChannel() - 1);
    }

    public void channelUp() {
        device.setChannel(device.getChannel() + 1);
    }
}

class AdvancedRemoteControl extends RemoteControl {

    public AdvancedRemoteControl(Device device) {
        super(device);
    }

    public void mute() {
        device.setVolume(0);
    }
}

// Implementation interface
interface Device {
    boolean isEnabled();
    void enable();
    void disable();
    int getVolume();
    void setVolume(int percent);
    int getChannel();
    void setChannel(int channel);
}

// Concrete implementation - TV
class Tv implements Device {
    private boolean on = false;
    private int volume = 50;
    private int channel = 1;

    @Override
    public boolean isEnabled() {
        return on;
    }

    @Override
    public void enable() {
        on = true;
```

```
        System.out.println("TV is now ON.");
    }

    @Override
    public void disable() {
        on = false;
        System.out.println("TV is now OFF.");
    }

    @Override
    public int getVolume() {
        return volume;
    }

    @Override
    public void setVolume(int percent) {
        volume = Math.max(0, Math.min(100, percent));
        System.out.println("TV volume set to " + volume + "%.");
    }

    @Override
    public int getChannel() {
        return channel;
    }

    @Override
    public void setChannel(int channel) {
        this.channel = channel;
        System.out.println("TV channel set to " + channel + ".");
    }
}

// Concrete implementation - Radio
class Radio implements Device {
    private boolean on = false;
    private int volume = 30;
    private int channel = 1;

    @Override
    public boolean isEnabled() {
        return on;
    }

    @Override
    public void enable() {
```

```
        on = true;
        System.out.println("Radio is now ON.");
    }

    @Override
    public void disable() {
        on = false;
        System.out.println("Radio is now OFF.");
    }

    @Override
    public int getVolume() {
        return volume;
    }

    @Override
    public void setVolume(int percent) {
        volume = Math.max(0, Math.min(100, percent));
        System.out.println("Radio volume set to " + volume + "%.");
    }

    @Override
    public int getChannel() {
        return channel;
    }

    @Override
    public void setChannel(int channel) {
        this.channel = channel;
        System.out.println("Radio channel set to " + channel + ".");
    }
}

// Client code
public class BridgePatternDemo {
    public static void main(String[] args) {
        Device tv = new Tv();
        RemoteControl tvRemote = new AdvancedRemoteControl(tv);

        tvRemote.togglePower();
        tvRemote.volumeUp();
        tvRemote.channelUp();
        tvRemote.togglePower();

        Device radio = new Radio();
```

```
        RemoteControl radioRemote = new AdvancedRemoteControl(radio);

        radioRemote.togglePower();
        radioRemote.volumeUp();
        ((AdvancedRemoteControl) radioRemote).mute(); // Using
AdvancedRemoteControl-specific method
        radioRemote.channelDown();
        radioRemote.togglePower();
    }
}
```

Output:

```
TV is now ON.
TV volume set to 60%.
TV channel set to 2.
TV is now OFF.
Radio is now ON.
Radio volume set to 40%.
Radio volume set to 0%.
Radio channel set to 0.
Radio is now OFF.
```

The **Bridge Design Pattern** helps to decouple abstraction from implementation, making the system more flexible, easier to extend, and maintain. Without the Bridge pattern, the design would face rigid class hierarchies that are difficult to modify.

Composite Design Pattern

The provided code demonstrates the **Composite Pattern**, which is a type of **Structural Design Pattern**. Here's a detailed explanation of each component and how it relates to the **Composite Pattern**:

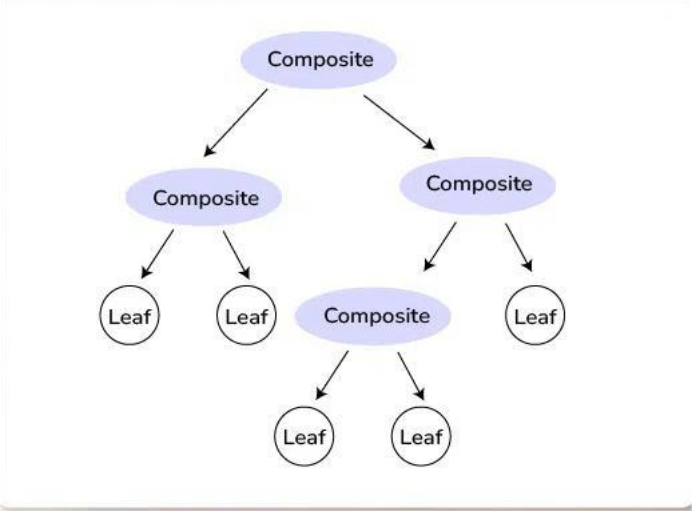
A structural design pattern is a blueprint for combining classes and objects to form a larger structure that achieves multiple goals.Structural Design Patterns are solutions in software design that focus on how classes and objects are organized to form larger, functional structures. These patterns help developers simplify relationships between objects, making code more efficient, flexible, and easy to maintain.

Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy. This type of design pattern comes under structural pattern as this pattern creates a tree structure of group of objects. This pattern creates a class that contains group of its own objects.

Real Life Example:

University Organizational System

Universities are complex organizations consisting of various departments, faculties, professors, and courses. Managing this hierarchical structure and representing it programmatically in a way that allows efficient querying, modification, and traversal is a challenging task.



1. Component Interface:

- This acts as the **Component** interface in the Composite Pattern.
- It defines a common operation (showDetails()) that both individual objects (**Leaf**) and groups of objects (**Composite**) must implement.
- Provides a uniform interface for all university members, whether it's a professor, course, or department.

2. Leaf Classes:

(a) **Professor,** (b) **Course:**

- Acts as a **Leaf**, which represents an individual object in the hierarchy that does not have any children.
- Represents a **single professor**.
- Implements the showDetails() method to print details of the professor.
- Represents a **single course**.
- Implements the showDetails() method to print details of the course.

3. Composite Class

Department:

.Acts as the Composite in the pattern.

.It can hold multiple UniversityMember objects (either Leaf or other Composite).

Represents a department in the university, which can contain professors and courses (Leaf objects), or even other departments (Composite objects).

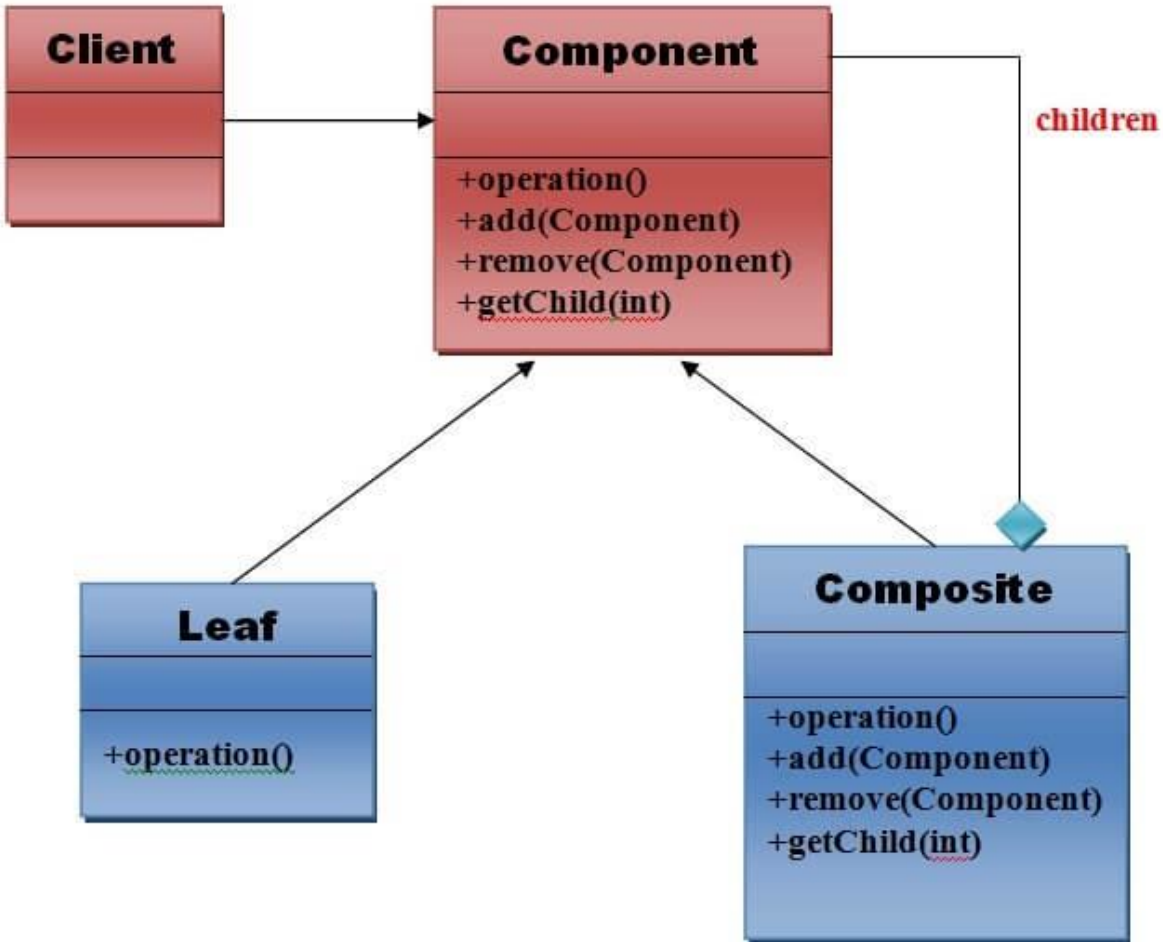
Implements the showDetails() method to display details of the department and recursively call showDetails() on all its members.

---

4. Client Code

Main Class:

- The client (Main) interacts with the UniversityMember interface, treating individual objects (Leaf) and composites (Department) uniformly.
    - Demonstrates how the Composite Pattern allows the client to work with complex hierarchical structures without worrying about whether it is working with an individual object or a group of objects.



```java
// Component Interface
public interface UniversityMember {
    void showDetails();
}

// Leaf: Professor
public class Professor implements UniversityMember {
    private String name;
    private String subject;

    public Professor(String name, String subject) {
        this.name = name;
        this.subject = subject;
    }

    @Override
    public void showDetails() {
        System.out.println("Professor: " + name + ", Subject: " + subject);
    }
}

// Leaf: Course
public class Course implements UniversityMember {
    private String courseName;

    public Course(String courseName) {
        this.courseName = courseName;
    }

    @Override
    public void showDetails() {
        System.out.println("Course: " + courseName);
    }
}

// Composite: Department
import java.util.ArrayList;
import java.util.List;

public class Department implements UniversityMember {
```

```
    private String departmentName;
    private List<UniversityMember> members = new ArrayList<>();

    public Department(String departmentName) {
        this.departmentName = departmentName;

    }

    public void addMember(UniversityMember member) {
        members.add(member);
    }

    public void removeMember(UniversityMember member) {
        members.remove(member);
    }

    @Override
    public void showDetails() {
        System.out.println("Department: " + departmentName);
        for (UniversityMember member : members) {
            member.showDetails();
        }
    }
}

// Client Code
public class Main {
    public static void main(String[] args) {
        // Create Professors
        Professor prof1 = new Professor("Dr. Rahman", "Mathematics");
        Professor prof2 = new Professor("Dr. Karim", "Physics");

        // Create Courses
        Course course1 = new Course("Calculus");
        Course course2 = new Course("Quantum Mechanics");

        // Create a Department and add members
        Department mathDept = new Department("Mathematics Department");
        mathDept.addMember(prof1);
        mathDept.addMember(course1);

        Department physicsDept = new Department("Physics Department");
        physicsDept.addMember(prof2);
        physicsDept.addMember(course2);

```

```
        // Create a Faculty
        Department scienceFaculty = new Department("Faculty of Science");
        scienceFaculty.addMember(mathDept);
        scienceFaculty.addMember(physicsDept);

        // Display the structure
        System.out.println("University Structure:");
        scienceFaculty.showDetails();
    }
}
```

How this example related to Composite Pattern:

1. **Part-Whole Hierarchy:**
    - The hierarchy is built with Faculty at the top, containing multiple Departments, which in turn contain Professors and Courses.
    - This represents a **part-whole relationship** where a whole (e.g., Department) is made up of smaller parts (e.g., Professors and Courses).
2. **Recursive Composition:**
    - The Department class is a Composite that holds child components (UniversityMember objects), enabling recursive traversal through the hierarchy.
3. **Uniform Interface:**
    - The UniversityMember interface ensures that all components (Leaf and Composite) share the same showDetails() method, making it easy for the client to interact with the structure.

This implementation demonstrates the **Composite Pattern** (a type of Structural Pattern) by creating a hierarchical tree structure for university members. The pattern provides a unified way to manage both individual entities and composite groups, enabling recursive traversal and simplified interaction for the client. It is a prime example of how structural patterns help organize and manage complex relationships between objects.

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.This pattern is useful when you need to add functionality to objects in a flexible and reusable way.

Characteristics of the Decorator Pattern

- This pattern promotes flexibility and extensibility in software systems by allowing developers to compose objects with different combinations of functionalities at runtime.
- It follows the open/closed principle, as new decorators can be added without modifying existing code, making it a powerful tool for building modular and customizable software components.
- The Decorator Pattern is commonly used in scenarios where a variety of optional features or behaviors need to be added to objects in a flexible and reusable manner, such as in text formatting, graphical user interfaces, or customization of products like coffee or ice cream

Real-World Example of Decorator Design Pattern

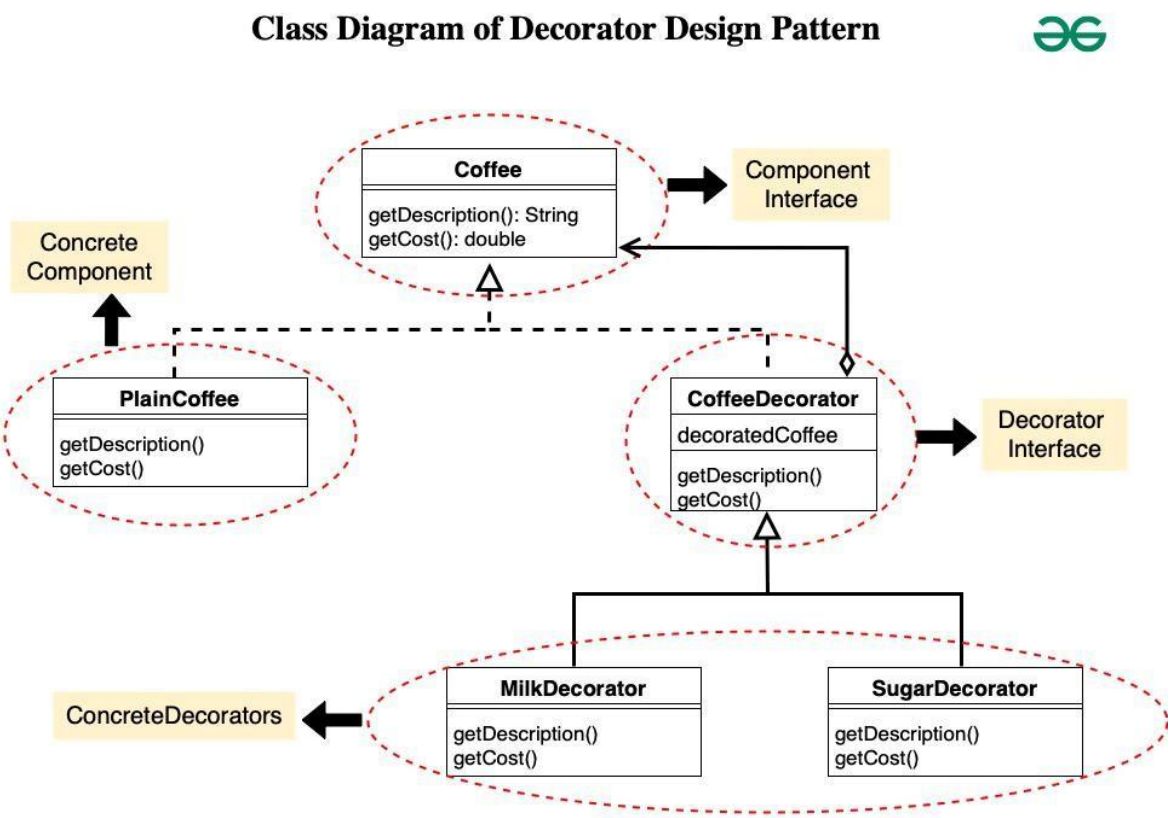
Suppose we are building a coffee shop application where customers can order different types of coffee. Each coffee can have various optional add-ons such as milk, sugar, whipped cream, etc. We want to implement a system where we can dynamically add these add-ons to a coffee order without modifying the coffee classes themselves.

Using the Decorator Pattern allows us to add optional features (add-ons) to coffee orders dynamically without altering the core coffee classes. This promotes code flexibility,

scalability, and maintainability as new add-ons can be easily introduced and combined with different types of coffee orders.

Key Components of the Decorator Design Pattern

- **Component Interface:** This is an abstract class or interface that defines the common interface for both the concrete components and decorators. It specifies the operations that can be performed on the objects.
- **Concrete Component:** These are the basic objects or classes that implement the Component interface. They are the objects to which we want to add new behavior or responsibilities.
- **Decorator**: This is an abstract class that also implements the Component interface and has a reference to a Component object. Decorators are responsible for adding new behaviors to the wrapped Component object.
- **Concrete Decorator**: These are the concrete classes that extend the Decorator class. They add specific behaviors or responsibilities to the Component. Each Concrete Decorator can add one or more behaviors to the Component.



CODE

1. Component Interface(Coffee)

- This is the interface `Coffee` representing the component.
- It declares two methods `getDescription()` and `getCost()` which must be implemented by concrete components and decorators.

```java
// Coffee.java
public interface Coffee {
    String getDescription();
    double getCost();
}
```

2. ConcreteComponent(PlainCoffee)

- `PlainCoffee` is a concrete class implementing the `Coffee` interface.
- It provides the description and cost of plain coffee by implementing the `getDescription()` and `getCost()` methods.

```java
// PlainCoffee.java
public class PlainCoffee implements Coffee {
    @Override
    public String getDescription() {
        return "Plain Coffee";
    }

    @Override
    public double getCost() {
        return 2.0;
    }
}
```

3. Decorator(CoffeeDecorator)

- `CoffeeDecorator` is an abstract class implementing the `Coffee` interface.
- It maintains a reference to the decorated `Coffee` object.

- The `getDescription()` and `getCost()` methods are implemented to delegate to the decorated coffee object.

```java
// CoffeeDecorator.java
public 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 getCost() {
        return decoratedCoffee.getCost();
    }
}
```

4. ConcreteDecorators(MilkDecorator,SugarDecorator)

- `MilkDecorator` and `SugarDecorator` are concrete decorators extending `CoffeeDecorator`.
- They override `getDescription()` to add the respective decorator description to the decorated coffee's description.
- They override `getCost()` to add the cost of the respective decorator to the decorated coffee's cost.

```java
// MilkDecorator.java
public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.5;
    }
}

// SugarDecorator.java
public class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Sugar";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.2;
    }
}
```

OUTPUT

```java
// Main.java
public class Main {
    public static void main(String[] args) {
        // Plain Coffee
        Coffee coffee = new PlainCoffee();
        System.out.println("Description: " + coffee.getDescription());
        System.out.println("Cost: $" + coffee.getCost());

        // Coffee with Milk
        Coffee milkCoffee = new MilkDecorator(new PlainCoffee());
        System.out.println("\nDescription: " + milkCoffee.getDescription());
        System.out.println("Cost: $" + milkCoffee.getCost());

        // Coffee with Sugar and Milk
        Coffee sugarMilkCoffee = new SugarDecorator(new MilkDecorator(new
PlainCoffee()));
        System.out.println("\nDescription: " + sugarMilkCoffee.getDescription());
        System.out.println("Cost: $" + sugarMilkCoffee.getCost());
    }
}
```

```
Description: Plain Coffee
Cost: $2.0

Description: Plain Coffee, Milk
Cost: $2.5

Description: Plain Coffee, Milk, Sugar
Cost: $2.7
```