Bridge Design Pattern

The Bridge Design Pattern is a structural design pattern that decouples an abstraction from its implementation so that the two can vary independently. This pattern is particularly useful when both the abstraction and the implementation may evolve or change over time.

Customer Requirement:

Imagine you need to design a system that allows different types of remote controls to operate various electronic devices like TVs, Radios, and Projectors. The system should be flexible enough to support new types of remote controls or devices without requiring changes to the existing codebase.

Solution: Bridge Pattern

The Bridge Pattern is well-suited for this scenario. It allows the remote controls (abstractions) to be decoupled from the electronic devices (implementations). This means you can develop new remote controls or new devices independently, and they can work together seamlessly.

Components of the Bridge Pattern:

1. Abstraction: The high-level interface that represents a remote control. It maintains a reference to a device and delegates the control operations to the device.

2. Refined Abstraction: Extends the basic functionality of the RemoteControl. Adds more features like mute or set channel.

3. Implementer: Defines the interface for electronic devices. The abstraction communicates with the device to perform the actual operations like turnOn, turnOff, or setVolume.

4. Concrete Implementer: Provides a concrete implementation of the Device interface. Each device has its own way of implementing the operations.

Example: Remote Control and Devices

1. Implementer (Device):

```java
public interface Device {
    void turnOn();
    void turnOff();
    void setVolume(int volume);
}
```

2. Concrete Implementers (TV, Radio, Projector):

```java
public class TV implements Device {
    private int volume;

    @Override
    public void turnOn() {
        System.out.println("Turning on the TV");
    }

    @Override
    public void turnOff() {
        System.out.println("Turning off the TV");
    }

    @Override
    public void setVolume(int volume) {
        this.volume = volume;
        System.out.println("Setting TV volume to " + volume);
    }
}

public class Radio implements Device {
    private int volume;

    @Override
    public void turnOn() {
        System.out.println("Turning on the Radio");
    }

    @Override
    public void turnOff() {
        System.out.println("Turning off the Radio");
    }

    @Override
    public void setVolume(int volume) {
        this.volume = volume;
        System.out.println("Setting Radio volume to " + volume);
    }
}

public class Projector implements Device {
```

```java
    private int volume;

    @Override
    public void turnOn() {
        System.out.println("Turning on the Projector");
    }

    @Override
    public void turnOff() {
        System.out.println("Turning off the Projector");
    }

    @Override
    public void setVolume(int volume) {
        this.volume = volume;
        System.out.println("Setting Projector volume to " + volume);
    }
}
```

3. Abstraction (RemoteControl):

```java
public class RemoteControl {
    protected Device device;

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

    public void turnOn() {
        device.turnOn();
    }

    public void turnOff() {
        device.turnOff();
    }

    public void setVolume(int volume) {
        device.setVolume(volume);
    }
}
```

4. Refined Abstraction (AdvancedRemoteControl):

```java
public class AdvancedRemoteControl extends RemoteControl {

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

    public void mute() {
        System.out.println("Muting the device");
        device.setVolume(0);
    }

}
```

5. Client Code:

```java
public class BridgePatternDemo {

    public static void main(String[] args) {

        Device tv = new TV();

        Device radio = new Radio();

        Device projector = new Projector();



        RemoteControl remoteControl = new RemoteControl(tv);

        AdvancedRemoteControl advancedRemoteControl = new AdvancedRemoteControl(radio);



        remoteControl.turnOn();

        remoteControl.setVolume(30);

        remoteControl.turnOff();



        advancedRemoteControl.turnOn();

        advancedRemoteControl.setVolume(20);

        advancedRemoteControl.mute();

        advancedRemoteControl.turnOff();

    }

}
```

How It Works:

- Client: The client uses RemoteControl or AdvancedRemoteControl to operate different devices like a TV, Radio, or Projector.

- Abstraction (RemoteControl): The RemoteControl class represents the interface for the remote control. It communicates with the device via the Device interface.

- Implementer (Device): The Device interface defines the operations that all devices must implement (e.g., turnOn, turnOff, setVolume).

- Concrete Implementers: Classes like TV, Radio, and Projector implement the Device interface, providing specific behavior for these operations.

Why It is a Structural Design Pattern:

The Bridge Pattern is a structural design pattern because it focuses on decoupling the abstraction (RemoteControl) from its implementation (Device). This allows the system to grow independently on both sides:

- You can add new devices (like a new type of TV or Radio) without changing the remote controls.
- You can also add new types of remote controls with advanced features without changing the devices.

Benefits of the Bridge Pattern:

1. Decoupling: The Bridge Pattern decouples the abstraction (remote control) from its implementation (devices), allowing both to evolve independently.
2. Extensibility: You can add new devices or new remote control features without modifying existing code.
3. Scalability: The system is more scalable and easier to maintain, as changes in one part do not affect the other.