Dung Pham - Design Patterns: Bridge

Introduction

The Bridge Pattern is a structural design pattern that decouples an abstraction from its implementation, allowing the two to be extended independently. The pattern consists of an abstraction (interface or abstract class) and an implementation, with a bridge connecting the two (the abstraction maintains reference to the implementation's object). This enables changes in either hierarchy without affecting the other.

I selected the implementation provided by Refactoring Guru in Java, which showcases a system with devices and remotes. The original example can be found here:

https://refactoring.guru/design-patterns/bridge/java/example

In the original implementation, there are two interfaces: Remote and Device, with various implementations for different types of remotes and devices. The Bridge Pattern is created to separate the abstraction (remote) from its implementation (device).

New Functionality

I extended the implementation by adding a new device called SmartTv and a new remote called SmartRemote. The goal behind this addition is to demonstrate how the Bridge Pattern allows for simple integration of a new functionalities without changing existing code. The SmartTv introduces a method browseInternet(), while the SmartRemote includes a voiceControl() method.

Implementation:

```java
public class SmartTv extends Tv {
    3 usages
    @Override
    public void printStatus() {
        System.out.println("------------------------------------");
        System.out.println("| I'm Smart TV");
        System.out.println("| I'm " + (on ? "enabled" : "disabled"));
        System.out.println("| Current volume is " + volume + "%");
        System.out.println("| Current channel is " + channel);
        System.out.println("------------------------------------\n");
    }

    1 usage
    public void browseInternet() {
        System.out.println("| Smart Tv: open Internet browser");
    }
}
```

```java
public class SmartRemote extends AdvancedRemote {
    1 usage
    public SmartRemote(Device device) {
        super(device);
    }

    1 usage
    public void voiceControl() {
        System.out.println("Remote: voice control activated, please speak into the mic");
    }
}
```

Verification:

To verify the new functionality, I created instances of SmartTv and SmartRemote:

```java
public class Main {
    public static void main(String[] args) {
        testDevice(new Tv());
        testDevice(new Radio());

        SmartTv smartTv = new SmartTv();
        testDevice(smartTv);
        smartTv.browseInternet();
    }

    3 usages
    public static void testDevice(Device device) {
        System.out.println("Tests with basic remote.");
        BasicRemote basicRemote = new BasicRemote(device);
        basicRemote.power();
        device.printStatus();

        System.out.println("Tests with advanced remote.");
        AdvancedRemote advancedRemote = new AdvancedRemote(device);
        advancedRemote.power();
        advancedRemote.mute();
        device.printStatus();

        System.out.println("Test with smart remote.");
        SmartRemote smartRemote = new SmartRemote(device);
        smartRemote.power();
        smartRemote.mute();
        smartRemote.voiceControl();
        device.printStatus();
    }
}
```

Upon running the main method, I got the expected outputs confirming that the new functions work as planned.

```
Test with smart remote.
Remote: power toggle
Remote: mute
Remote: voice control activated, please speak into the mic
-----------------------------------
| I'm Smart TV
| I'm enabled
| Current volume is 0%
| Current channel is 1
-----------------------------------

| Smart Tv: open Internet browser
```

Conclusion

In conclusion, the implementation of the Bridge Patter demonstrates its flexibility in upgrading new functionalities more easily, without modifying existing code of either hierarchy. This design decision promotes code maintainability and scalability, allowing for easy adaptation to evolving requirements.