Bridge Pattern Implementation Report

Introduction

The Bridge Pattern is a structural design pattern that separates an abstraction from its implementation so that the two can vary independently. It involves an interface which acts as a bridge between the abstraction class and implementor classes.

In this implementation, the Bridge Pattern is used to separate the remote control functionality (abstraction) from the device functionality (implementation). This allows for different types of remotes to control different types of devices without creating a combinatorial explosion of classes.

The original implementation is based on the example from Refactoring Guru (https://refactoring.guru/design-patterns/bridge/java/example), which demonstrates a system with devices and remotes. In this system:

- The `Device` interface defines the operations that all devices must implement, such as power control, volume control, and channel control.
- The `Remote` interface defines the operations that all remotes must implement, which are similar to the device operations but from a remote control perspective.
- The `BasicRemote` class implements the `Remote` interface and delegates the operations to a `Device` object.
- The `Tv` class implements the `Device` interface with basic TV functionality.

New Functionality

I extended the original implementation by adding two new classes:

1. **SmartTv**: A new device that extends the basic `Tv` class with internet browsing capability.
2. **SmartRemote**: A new remote that extends the `BasicRemote` class with voice control capability.

The motivation behind adding these new classes was to demonstrate how the Bridge Pattern allows for easy extension of both the abstraction and implementation hierarchies independently. Smart TVs and smart remotes are common extensions of traditional TVs and remotes in the real world, making them a natural choice for this example.

Implementation

Here is the code for the new functionality:

SmartTv Class

```
package devices;

public class SmartTv extends Tv {

    public SmartTv(Device device) {
        super.device = device;
```

```
    }

    public void browseInternet() {
        System.out.println("Smart TV: browsing the internet");
    }
}
```

The `SmartTv` class extends the basic `Tv` class and adds a new method `browseInternet()` that allows the TV to browse the internet, a feature not available in the basic TV.

SmartRemote Class

```
package remotes;

import devices.Device;

public class SmartRemote extends BasicRemote {

    public SmartRemote(Device device) {
        super.device = device;
    }

    public void voiceControl() {
        System.out.println("Remote: voice control");
    }
}
```

The `SmartRemote` class extends the `BasicRemote` class and adds a new method `voiceControl()` that allows the remote to be controlled by voice, a feature not available in the basic remote.

Verification

To verify that the new functionality works as expected, I created a `Demo` class that tests both the basic and smart devices with both the basic and smart remotes:

```
import devices.Device;
import devices.SmartTv;
import devices.Tv;
import remotes.BasicRemote;
import remotes.Remote;
import remotes.SmartRemote;

public class Demo {
    public static void main(String[] args) {
        testDevice(new Tv());
        testDevice(new SmartTv(new Tv()));
    }

    public static void testDevice(Device device) {
```

```
        System.out.println("Tests with basic remote.");
        BasicRemote basicRemote = new BasicRemote(device);
        basicRemote.power();
        device.printStatus();
        basicRemote.volumeUp();
        basicRemote.channelUp();
        device.printStatus();
        basicRemote.volumeDown();
        basicRemote.channelDown();
        device.printStatus();

        System.out.println("Tests with smart remote.");
        SmartRemote smartRemote = new SmartRemote(device);
        smartRemote.power();
        smartRemote.volumeUp();
        smartRemote.channelUp();
        smartRemote.voiceControl();
        device.printStatus();
        smartRemote.volumeDown();
        smartRemote.channelDown();
        device.printStatus();

        // Test SmartTv specific functionality if the device is a SmartTv
        if (device instanceof SmartTv smartTv) {
            System.out.println("Tests with SmartTv specific
functionality.");
            smartTv.browseInternet();
            device.printStatus();
        }
    }
}
```

When running this demo, we can observe:

1. The basic TV works with both the basic remote and the smart remote.
2. The smart TV works with both the basic remote and the smart remote.
3. The smart TV's internet browsing functionality works correctly.
4. The smart remote's voice control functionality works correctly.

This demonstrates that the Bridge Pattern allows for independent variation of the abstraction and implementation hierarchies, as we can mix and match different types of remotes with different types of devices.

Conclusion

In this implementation, I extended a basic Bridge Pattern example with new functionality by adding a SmartTv class and a SmartRemote class. These extensions demonstrate the flexibility of the Bridge Pattern in allowing independent variation of the abstraction and implementation hierarchies.

The key design decisions I made were:

1. To extend the existing `Tv` and `BasicRemote` classes rather than creating entirely new implementations of the `Device` and `Remote` interfaces. This allowed me to reuse the existing functionality while adding new features.
2. To add methods specific to the smart devices and remotes (`browseInternet()` and `voiceControl()`) that showcase their enhanced capabilities.

An alternative approach would have been to create new implementations of the `Device` and `Remote` interfaces from scratch, but this would have resulted in code duplication. The approach I took is more in line with the principles of object-oriented design, particularly the principle of code reuse through inheritance.

The Bridge Pattern proved to be an effective way to organize this system, as it allowed for easy extension of both the device and remote hierarchies without creating a combinatorial explosion of classes. This is particularly valuable in systems where there are multiple dimensions of variation, as is the case with devices and remotes.