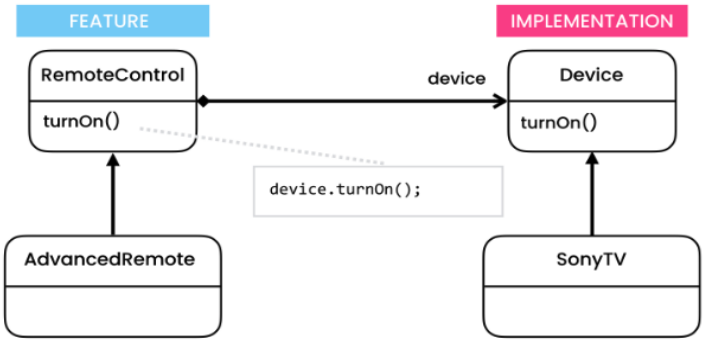
Bridge Pattern:

Definition: Allows representing hierarchies that grow in two different dimensions independently.



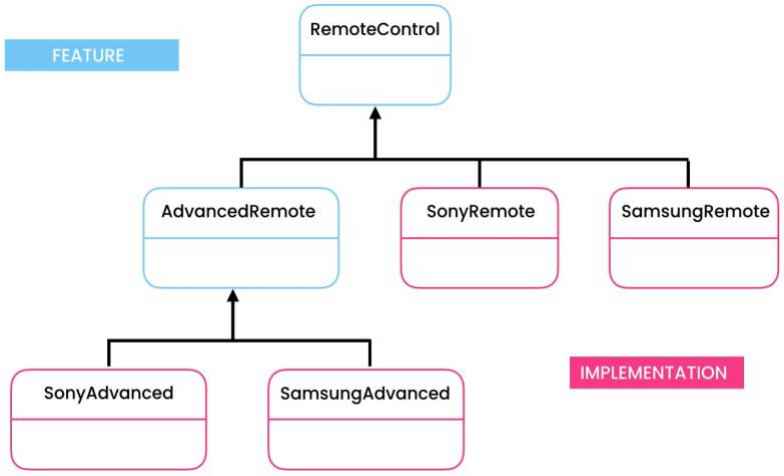
Scenario: We want to implement a remote control. It can be a basic or advanced. Basic remote control means it can be used to turn on and turn of the TV while advanced remote control has also the functionality to set the channels. And multiple brands like Sony and Samsung has their own remotes which can be basic or advanced.

Implementation:

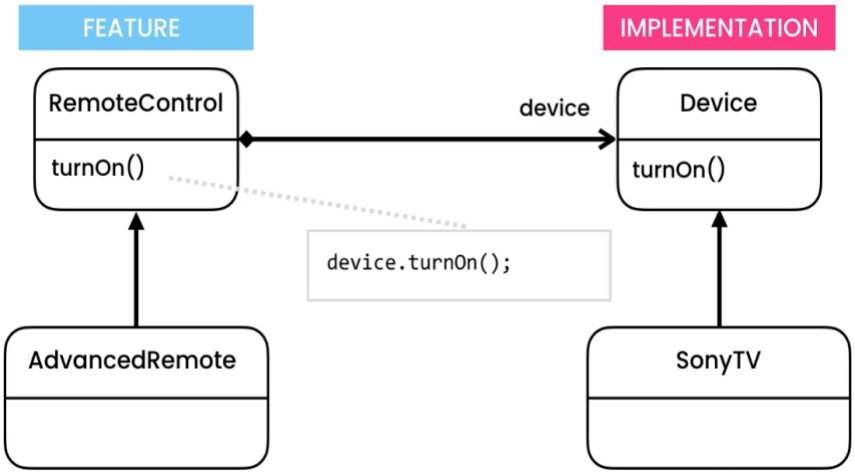
```
public class RemoteControl {
    public void turnOn(){
        System.out.println("TV is on");
    }

    public void turnOff(){
        System.out.println("TV is off");
    }
}

public class AdvancedRemoteControl extends RemoteControl {
    public AdvancedRemoteControl() {
        super();
    }

    public void setChannels(int number){
        System.out.println("Set channels : " + number);
    }
}
```

SonyRemoteControl, SamsungRemoteControl will extend RemoteControl class and SonyAdvancedRemoteControl, SamsungAdvancedRemoteControl will extend AdvancedRemoteControl class.

Problem: We always have to make new classes and structure becomes messy. For example, later if we want to support LGRemote and LGAdvanced remote, our structure will keep growing. Hierarchy is the problem here. We have a dependency between feature and implementation as shown in the figure.



Solution: Feature and implementation must grow independently and bind each other with composition as shown in the figure.



Feature hierarchy can be represented by an interface

```
public interface Device {
   void turnOn();
   void turnOff();
   void setChannels(int number);
}
```

And implementation hierarchy can be implemented separately and feature can be bind here with composition.

```
public class RemoteControl {
    protected Device device;

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

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

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

Now it can be called by concrete classes

```
public class SonyTV implements Device {
    @Override
    public void turnOn() {
        System.out.println("Sony: Turn on");
    }

    @Override
    public void turnOff() {
        System.out.println("Sony: Turn off");
    }

    @Override
    public void setChannels(int number) {
        System.out.println("Sony: Set Channel");
    }
}
```