Bridge Pattern

Michal Moravik, SD20w2

Name: Bridge Pattern

Category: Structural

Intent: Decouples an abstraction from implementation. They can be afterwards developed independently. It prefers composition over inheritance.

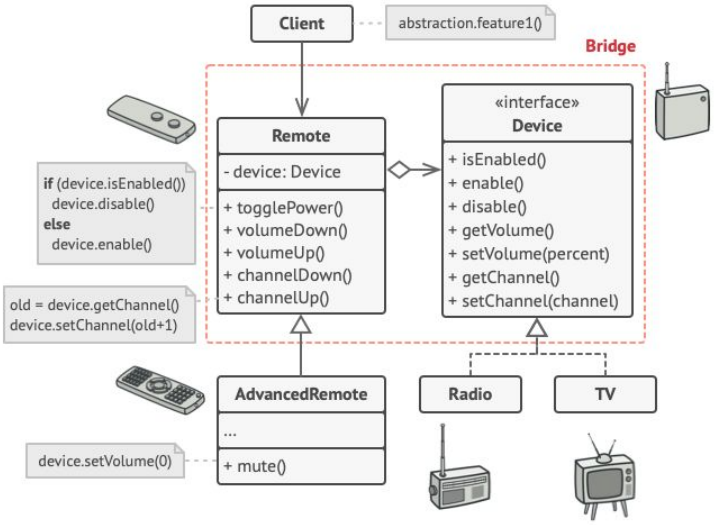
Motivation: We want to use it when we need to extend a class in several independent dimensions. We also want to use it if we want to switch implementations at runtime.

Real case example: Suppose we have devices like TV or Radio. And then we have remotes which control these devices. Devices are, in this case, implementations and remotes are abstractions.

Thanks to the bridge pattern, we do not need to have TVRemote, RadioRemote, TVAdvancedRemote, nor RadioAdvancedRemote. Adding a new device (implementation) in that case would mean adding two new remotes (e.g. MobileRemote, MobileAdvancedRemote). Vica versa in case of adding a new remote (abstraction), we would need to add new combinations with the devices too (TVSuperRemote, RadioSuperRemote, MobileSuperRemote).

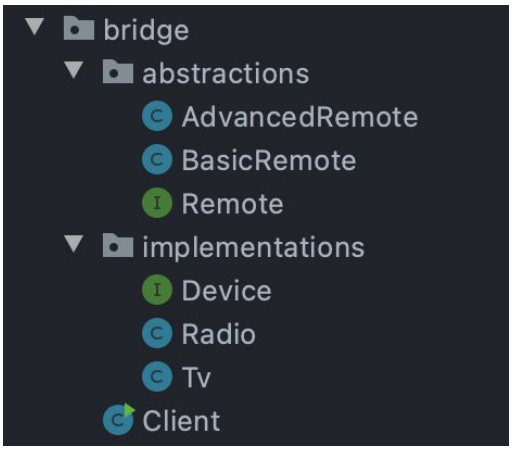
To avoid such a scenario, we can plan the use of Bridge Pattern upfront.

UML:



UML from refactoring.guru/design-patterns/bridge

Implementation:




```
public interface Device {
    boolean isEnabled();
    void enable();
    void disable();
    int getVolume();
    void setVolume(int percent);
    void printStatus();
}
```

```java
public class Tv implements Device {
    private boolean on = false;
    private int volume = 30;

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

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

    @Override
    public void disable() {
        on = false;
    }

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

    @Override
    public void setVolume(int volume) {
        if (volume > 100) {
            this.volume = 100;
        } else if (volume < 0) {
            this.volume = 0;
        } else {
            this.volume = volume;
        }
    }

    @Override
    public void printStatus() {
        System.out.println("------------------------------------");
        System.out.println("| TV ");
        System.out.println("| I'm " + (on ? "enabled" : "disabled"));
        System.out.println("| Current volume is " + volume + "%");
        System.out.println("------------------------------------\n");
    }
}
```

```java
public class Radio implements Device {
    private boolean on = false;
    private int volume = 30;

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

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

    @Override
    public void disable() {
        on = false;
    }

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

    @Override
    public void setVolume(int volume) {
        if (volume > 100) {
            this.volume = 100;
        } else if (volume < 0) {
            this.volume = 0;
        } else {
            this.volume = volume;
        }
    }

    @Override
    public void printStatus() {
        System.out.println("------------------------------------");
        System.out.println("| Radio ");
        System.out.println("| I'm " + (on ? "enabled" : "disabled"));
        System.out.println("| Current volume is " + volume + "%");
        System.out.println("------------------------------------\n");
    }
}
```

```java
public interface Remote {
    void power();
    void volumeDown();
    void volumeUp();
}
```

```java
public class BasicRemote implements Remote {
    protected Device device;

    public BasicRemote() {}

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

    @Override
    public void power() {
        System.out.println("Remote: power toggle");
        if (device.isEnabled()) {
            device.disable();
        } else {
            device.enable();
        }
    }

    @Override
    public void volumeDown() {
        System.out.println("Remote: volume down");
        device.setVolume(device.getVolume() - 10);
    }

    @Override
    public void volumeUp() {
        System.out.println("Remote: volume up");
        device.setVolume(device.getVolume() + 10);
    }
}
```

```java
public class AdvancedRemote extends BasicRemote {

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

    public void mute() {
        System.out.println("Remote: mute");
        device.setVolume(0);
    }
}
```

Consequences:

Pros:

- The client code works with the high-level abstraction (generalization)
- You can introduce new abstractions and implementations independently from each other

Cons:

- Introduces a lot of complexity to the code

Known uses:

- Platform-independent classes and apps

Related patterns:

- Abstract factory - if there is a situation that a specific bridge abstraction can only work with a specific implementation, we can add an abstract factory to encapsulate these relations and hide the complexity from the client code.