Viet Tran
25.04.2025
2320123

Bridge Design Pattern

1. Bridge Design Pattern is a structural pattern which would help the developer separate abstraction from implementation. Meaning that you wouldn't necessarily need to create each objects and their controllers to match each other specifically and limited only to them. You can basically create different classes and create the "bridge" like connection between them.

   Link to the original implementation: https://refactoring.guru/design-patterns/bridge/java/example

   Above is the original code from where I started before extending the task. The code provided the Device and Remote interfaces. And also, the implementation of Device and abstraction of Remote. This implementation design basically allows any Remote to connect to any Device.

2. The original code came with a device like TV which I took and used and a Remote with BasicRemote. I extended this with creating the most simplified Remote with the less functionalities as possible. I call it MinimalRemote. The remote is made for people like elderly and children who would just like to control the power button and muting without needing or being restricted to adjust anything else. The mute button would remember the previous volume level and once the user click on it again it would unmute.

3. Code for the new functionality (MinimalRemote):

```
public void mute() {
    currentVolume = device.getVolume();

    // First checks if the volume is not muted
    if (currentVolume != 0) {
      // Remembers the previous volume level
      beforeMute = device.getVolume();
      device.setVolume(0);
      System.out.println("Muted, current volume: " + device.getVolume());
    } else {
      device.setVolume(beforeMute);
      System.out.println("Unmuted, current volume: " + device.getVolume());
    }
  }
```

   Link to the code:

4. I verified the new mute functionality by running the program and observing the expected behavior through console outputs. When using the MinimalRemote, pressing the mute button successfully set the TV volume to 0, and pressing mute again restored the previous volume level. Additionally, I tested unsupported operations (such as volumeUp() on MinimalRemote) and confirmed that an UnsupportedOperationException was correctly thrown and caught, as shown in the provided code snippets.

The code works as supposed:

```
=== Using BasicRemote ===
Remote: power toggle
Remote: volume up
Remote: channel up

------------------------

=== Using MinimalRemote with Mute Feature ===
MinimalRemote: power toggle
Muting...
Muted, current volume: 0
Volume after mute: 0
Unmuting...
Unmuted, current volume: 40
Volume after unmute: 40
Caught unsupported operation: volumeUp not supported
viettran@Viet-MacBook-Pro bridge %
```

Here is also the Main class to support the Bridge logic:

```
3   public class Main {
        Run | Debug
4       public static void main(String[] args) {
5           // Create a TV device
6           Device tv = new TV();
7
8           // Basic Remote section
9           System.out.println(x:"=== Using BasicRemote ===");
10          Remote basicRemote = new BasicRemote(tv);
11          basicRemote.power();
12          basicRemote.volumeUp();
13          basicRemote.channelUp();
14
15          System.out.println(x:"\n------------------------\n");
16
17          // Minimal Remote section
18          System.out.println(x:"=== Using MinimalRemote with Mute Feature ===");
19          MinimalRemote minimalRemote = new MinimalRemote(tv);
20          minimalRemote.power();
21
22          System.out.println(x:"Muting...");
23          minimalRemote.mute();
24          System.out.println("Volume after mute: " + tv.getVolume());
25
26          System.out.println(x:"Unmuting...");
27          minimalRemote.mute();
28          System.out.println("Volume after unmute: " + tv.getVolume());
29
30          // Only wrap volumeUp() because it throws
31          try {
32              minimalRemote.volumeUp();
33          } catch (UnsupportedOperationException e) {
34              System.out.println("Caught unsupported operation: " + e.getMessage());
35          }
36      }
37  }
```

5. The plan was to initially create a multi usable remote control but then instead of adding new features, why couldn't we test if those features could be restricted and adding just one tiny method to prove that adding new features would also work and confirm the Bridge pattern.

   Summarize
   The Bridge Pattern was extended by creating a new abstraction (MinimalRemote). The Bridge's power was proved here by customizing the behaviour of the Remote with restricting it to fewer buttons, but smarter behaviour. The new mute() method adds a practical real-world use case for people who we would like to keep it simple for.