Dependency Inversion Principle Violation (Bad Example)

Consider the example of an electric switch that turns a light bulb on or off. We can model this requirement by creating two classes: `ElectricPowerSwitch` and `LightBulb`. Let's write the `LightBulb` class first.

LightBulb.java

```
1 public class LightBulb {
2    public void turnOn() {
3       System.out.println("LightBulb: Bulb turned on...");
4    }
5    public void turnOff() {
6       System.out.println("LightBulb: Bulb turned off...");
7    }
8 }
```

In the LightBulb class above, we wrote the `turnOn()` and `turnOff()` methods to turn a bulb on and off.

Next, we will write the `ElectricPowerSwitch` class.

ElectricPowerSwitch.java

```
1   public class ElectricPowerSwitch {
2      public LightBulb lightBulb;
3      public boolean on;
4      public ElectricPowerSwitch(LightBulb lightBulb) {
5         this.lightBulb = lightBulb;
6         this.on = false;
7      }
8      public boolean isOn() {
9         return this.on;
10     }
11     public void press(){
12        boolean checkOn = isOn();
13        if (checkOn) {
14           lightBulb.turnOff();
15           this.on = false;
16        } else {
17           lightBulb.turnOn();
18           this.on = true;
```

```
19        }
20
21    }
22 }
```

In the example above, we wrote the `ElectricPowerSwitch` class with a field referencing `LightBulb`. In the constructor, we created a `LightBulb` object and assigned it to the field. We then wrote a `isOn()` method that returns the state of `ElectricPowerSwitch` as a `boolean` value. In the `press()` method, based on the state, we called the `turnOn()` and `turnOff()` methods.

Our switch is now ready for use to turn on and off the light bulb. But the mistake we did is apparent. Our high-level `ElectricPowerSwitch` class is directly dependent on the low-level `LightBulb` class. if you see in the code, the `LightBulb` class is hardcoded in `ElectricPowerSwitch`. But, a switch should not be tied to a bulb. It should be able to turn on and off other appliances and devices too, say a fan, an AC, or the entire lightning system of an amusement park. Now, imagine the modifications we will require in the `ElectricPowerSwitch` class each time we add a new appliance or device. We can conclude that our design is flawed and we need to revisit it by following the Dependency Inversion Principle.

Following the Dependency Inversion Principle

To follow the Dependency Inversion Principle in our example, we will need an abstraction that both the `ElectricPowerSwitch` and `LightBulb` classes will depend on. But, before creating it, let's create an interface for switches.

Switch.java

```
1 public interface Switch {
2     boolean isOn();
3     void press();
4 }
```

We wrote an interface for switches with the `isOn()` and `press()` methods. This interface will give us the flexibility to plug in other types of switches, say a remote control switch later on, if required. Next, we will write the abstraction in the form of an interface, which we will call `Switchable`.

Switchable.java

```
1 public interface Switchable {
2     void turnOn();
3     void turnOff();
4 }
```

In the example above, we wrote the `Switchable` interface with the `turnOn()` and `turnoff()` methods. From now on, any switchable devices in the application can implement this interface and provide their own functionality. Our `ElectricPowerSwitch` class will also depend on this interface, as shown below:

ElectricPowerSwitch.java

```
1   public class ElectricPowerSwitch implements Switch {
2       public Switchable client;
3       public boolean on;
4       public ElectricPowerSwitch(Switchable client) {
5           this.client = client;
6           this.on = false;
7       }
8       public boolean isOn() {
9           return this.on;
10      }
11     public void press(){
12         boolean checkOn = isOn();
13         if (checkOn) {
14             client.turnOff();
15             this.on = false;
16         } else {
17               client.turnOn();
18               this.on = true;
19         }
20      }
21  }
```

In the `ElectricPowerSwitch` class we implemented the `Switch` interface and referred the `Switchable` interface instead of any concrete class in a field. We then called the `turnOn()` and `turnoff()` methods on the interface, which at run time will get invoked on the object passed to the constructor. Now, we can add low-level switchable classes without worrying about modifying the `ElectricPowerSwitch` class. We will add two such classes: `LightBulb` and `Fan`.

LightBulb.java

```
1  public class LightBulb implements Switchable {
2      @Override
3      public void turnOn() {
4          System.out.println("LightBulb: Bulb turned on...");
5      }
6       @Override
7      public void turnOff() {
8          System.out.println("LightBulb: Bulb turned off...");
9      }
10 }
```

Fan.java

```
1  public class Fan implements Switchable {
2      @Override
3      public void turnOn() {
4          System.out.println("Fan: Fan turned on...");
5      }
6       @Override
7      public void turnOff() {
8          System.out.println("Fan: Fan turned off...");
9      }
10 }
```

In both the `LightBulb` and `Fan` classes that we wrote, we implemented the `Switchable` interface to provide their own functionality for turning on and off. While writing the classes, if you have missed how we arranged them in packages, notice that we kept the `Switchable` interface in a different package from the low-level electric device classes. Although, this did not make any difference from coding perspective, except for an import statement, by doing so we have made our intentions clear- We want the low-level classes to depend (inversely) on our abstraction. This will also help us if we later decide to release the high-level package as a public API that other applications can use for their devices. To test our example, let's write this unit test.

ElectricPowerSwitchTest.java

```
1  public class ElectricPowerSwitchTest {
2      @Test
3      public void testPress() throws Exception {
4       Switchable switchableBulb=new LightBulb();
5       Switch bulbPowerSwitch=new ElectricPowerSwitch(switchableBulb);
6       bulbPowerSwitch.press();
7       bulbPowerSwitch.press();
8      Switchable switchableFan=new Fan();
9      Switch fanPowerSwitch=new ElectricPowerSwitch(switchableFan);
10     fanPowerSwitch.press();
11     fanPowerSwitch.press();
12     }
13 }
```

The output is:

```
1  LightBulb: Bulb turned on...
2  LightBulb: Bulb turned off...
3  Fan: Fan turned on...
4  Fan: Fan turned off...
```