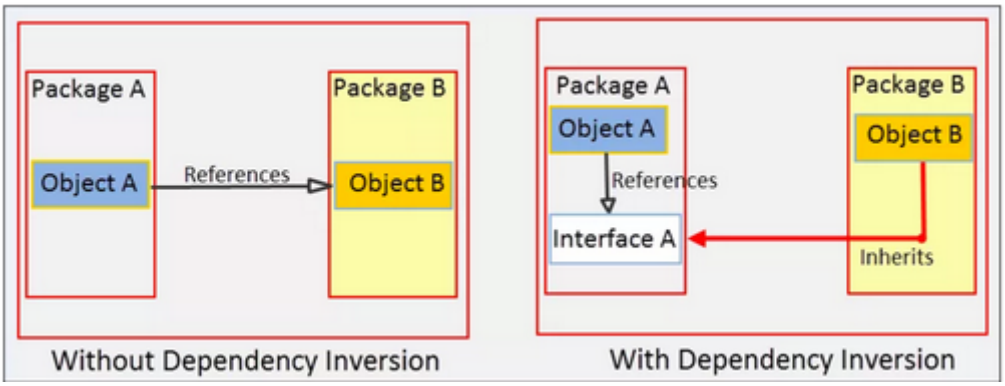
Dependency Inversion Principle

- One of the basic rules of good programming is to avoid tight coupling. Ex, creating an object of a class using the new operator results in a class being tightly coupled to another class.
  - This does not disrupt small applications, but in enterprise application development this can have serious consequences
- When one class knows explicitly about the design and implementation of another class, changes to one class raise the risk of breaking another class
- Dependency Inversion Principle represents the last 'D' of the SOLD principles as published in 1996. The principle states:
  1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
  2. Abstractions should not depend on details. Details should depend on abstractions."
- Conventional application architecture follows a top-down design approach where a high-level problem is broken into smaller parts. In other words, the high-level design is described in terms of these smaller parts. As a result, high-level modules that gets written directly depends on the smaller (low-level) modules.
- What Dependency Inversion Principle says is that instead of a high-level module depending on a low-level module, both should depend on an abstraction. Let us look at it in the context of Java through this figure.



- What the principle has done is:
  1. Both Object A and Object B now depends on Interface A, the abstraction.
  2. It inverted the dependency that existed from Object A to Object B into Object B being dependent on the abstraction (Interface A).

Violation (Bad) Example

Consider the example of an electric switch that turns a light bulb on or off.

LightBulb.java

```
public class LightBulb {
    public void turnOn() {
        System.out.println("LightBulb: Bulb turned on...");
    }
    public void turnOff() {
        System.out.println("LightBulb: Bulb turned off...");
    }
}
```

ElectricPowerSwitch.java

```
public class ElectricPowerSwitch {
    public LightBulb lightBulb;
    public boolean on;
    public ElectricPowerSwitch(LightBulb lightBulb) {
        this.lightBulb = lightBulb;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
    public void press(){
        boolean checkOn = isOn();
        if (checkOn) {
            lightBulb.turnOff();
            this.on = false;
        } else {
            lightBulb.turnOn();
            this.on = true;
        }
    }
}
```

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.

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

```
package guru.springframework.blog.dependencyinversionprinciple.highlevel;


public interface Switch {
    boolean isOn();
    void press();
}
```

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

```
package guru.springframework.blog.dependencyinversionprinciple.highlevel;


public interface Switchable {
    void turnOn();
    void turnOff();
}
```

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

```
package guru.springframework.blog.dependencyinversionprinciple.highlevel;



public class ElectricPowerSwitch implements Switch {
    public Switchable client;
```

```
    public boolean on;
    public ElectricPowerSwitch(Switchable client) {
        this.client = client;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
   public void press(){
      boolean checkOn = isOn();
      if (checkOn) {
          client.turnOff();
          this.on = false;
      } else {
            client.turnOn();
            this.on = true;
      }

   }
}
```

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

```
package guru.springframework.blog.dependencyinversionprinciple.lowlevel;

import guru.springframework.blog.dependencyinversionprinciple.highlevel.Switchable;

public class LightBulb implements Switchable {
    @Override
    public void turnOn() {
        System.out.println("LightBulb: Bulb turned on...");
    }

    @Override
    public void turnOff() {
        System.out.println("LightBulb: Bulb turned off...");
    }
```

```
    }
```

Fan.java

```
package guru.springframework.blog.dependencyinversionprinciple.lowlevel;

import guru.springframework.blog.dependencyinversionprinciple.highlevel.Switchable;

public class Fan implements Switchable {
    @Override
    public void turnOn() {
        System.out.println("Fan: Fan turned on...");
    }

    @Override
    public void turnOff() {
        System.out.println("Fan: Fan turned off...");
    }
}
```

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.

Summary of the Dependency Inversion Principle

Robert Martin equated the Dependency Inversion Principle, as a first-class combination of the Open Closed Principle and the Liskov Substitution Principle, and found it important enough to give its own name. While using the Dependency Inversion Principle comes with the overhead of writing additional code, the advantages that it provides outweigh the extra effort. Therefore, from now whenever you start writing code, consider the possibility of dependencies breaking your code, and if so, add abstractions to make your code resilient to changes.

Revision #2
Created 17 April 2022 00:50:44 by Elkip
Updated 17 April 2022 00:52:12 by Elkip