Dependency Inversion Principle

As a Java programmer, you've likely heard about code coupling and have been told to avoid tightly coupled code. Ignorance of writing "*good code*" is the main reason of tightly coupled code existing in applications. As an example, creating an object of a class using the **new** operator results in a class being tightly coupled to another class. Such coupling appears harmless and does not disrupt small programs. But, as you move into enterprise application development, tightly coupled code can lead to serious adverse consequences.

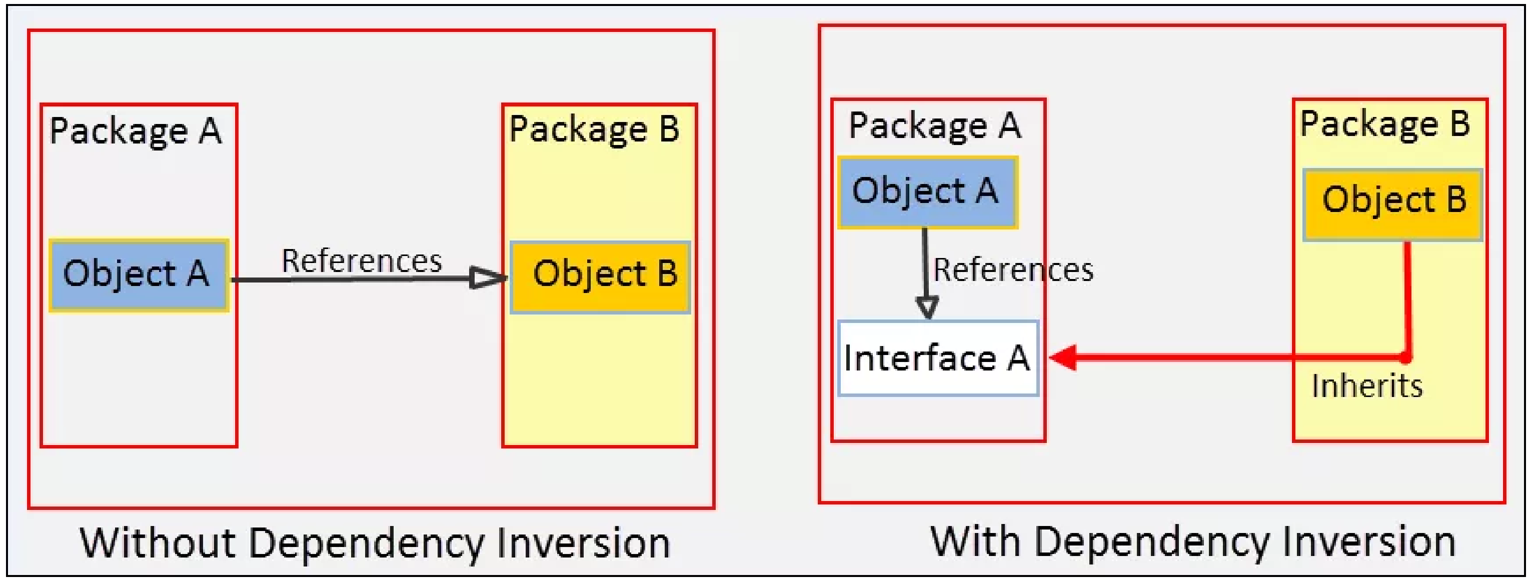
When one class knows explicitly about the design and implementation of another class, changes to one class raise the risk of breaking the other class. Such changes can have rippling effects across the application making the application fragile. To avoid such problems, you should write "*good code*" that is loosely coupled, and to support this you can turn to the Dependency Inversion Principle.

The Dependency Inversion Principle represents the last "D" of the five SOLID principles of object-oriented programming. Robert C. Martin first postulated the Dependency Inversion Principle and published it in 1996. The principle states:

> *"A. High-level modules should not depend on low-level modules. Both should depend on abstractions.*
> *B. 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.



In the figure above, without Dependency Inversion Principle, Object A in Package A refers Object B in Package B. With Dependency Inversion Principle, an Interface A is introduced as an abstraction in Package A. Object A now refers Interface A and Object B inherits from Interface A. 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).

Before we write code that follows the Dependency Inversion Principle, let's examine a typical violating of the principle.

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

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

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

```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;
        }

    }
}
```

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

```
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

```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

```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

```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. To test our example, let’s write this unit test.

ElectricPowerSwitchTest.java

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

import guru.springframework.blog.dependencyinversionprinciple.lowlevel.Fan;
import guru.springframework.blog.dependencyinversionprinciple.lowlevel.LightBulb;
import org.junit.Test;

public class ElectricPowerSwitchTest {

    @Test
    public void testPress() throws Exception {
     Switchable switchableBulb=new LightBulb();
     Switch bulbPowerSwitch=new ElectricPowerSwitch(switchableBulb);
     bulbPowerSwitch.press();
     bulbPowerSwitch.press();

    Switchable switchableFan=new Fan();
    Switch fanPowerSwitch=new ElectricPowerSwitch(switchableFan);
    fanPowerSwitch.press();
    fanPowerSwitch.press();
    }
}
```

The output is:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.2.3.RELEASE)

Running guru.springframework.blog.dependencyinversionprinciple.highlevel.ElectricPowerSwitchTest
LightBulb: Bulb turned on...
LightBulb: Bulb turned off...
Fan: Fan turned on...
Fan: Fan turned off...
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.016 sec - in guru.springframework.blog.dependencyinversionprinciple.highlevel.ElectricPo
```

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.

Radio Shack
Rosin Core Solder
DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency Inversion Principle and the Spring Framework

You may think the Dependency Inversion Principle is related to Dependency Injection as it applies to the Spring Framework, and you would be correct. Uncle Bob Martin coined this concept of Dependency Inversion before Martin Fowler coined the term Dependency Injection. The two concepts are highly related. Dependency Inversion is more focused on the structure of your code, its focus is keeping your code loosely coupled. On the other hand, Dependency Injection is how the code functionally works. When programming with the Spring Framework, Spring is using Dependency Injection to assemble your application. Dependency Inversion is what decouples your code so Spring can use Dependency Injection at run time.

7 comments on "Dependency Inversion Principle"

Charles Morin November 5, 2015 at 11:28 am # Reply

Very interesting as always. Thank you

jt November 5, 2015 at 11:32 am # Reply

Thanks!

boni octavianus August 10, 2016 at 12:54 am # Reply

really helpful ! thanks 😀

1111@a.com March 14, 2017 at 9:00 am # Reply

Good one!

Ganesh Tiwari April 18, 2017 at 1:07 am # Reply

Really a good example!! Thanks!

Sundar Rajagopal August 4, 2017 at 7:47 am # Reply

Very Good Explanation on ‘Dependency Inversion’ principle. Thanks much.