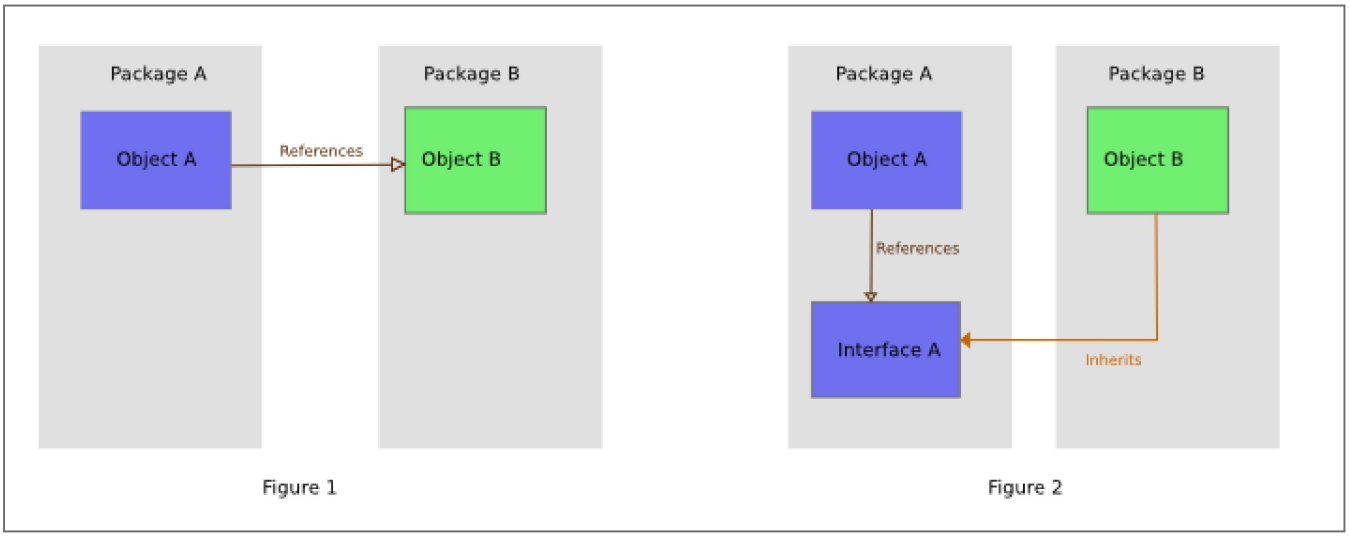
KD Knowledge Diet Follow
Apr 10 · 5 min read · 4:23

Dependency Inversion Principle: How Google Developers write code



To become a high-paid developer, you need to learn TDD. Basically, you have to develop software with TDD to get into a big company. If there are a few concepts you should know before doing TDD, it is ***Dependency Inversion Principle*** of SOLID Principle that you should know. It may not be easy for novice developers to understand, but luckily you have just found the easiest tutorial in the world. I am sure this short tutorial will be of great help to you in your career.

TDD: Test Driven Development

What is Dependency Inversion Principle?

Dependency Inversion Principle corresponds to **D** among SOLI'**D**' Principles. Its

Well-organized code always has a hierarchy. There is a high-level modules and low-level modules. But sometimes rookie developers misunderstand this concept, and they bring directly low-level modules to high-level modules.

Can you see what's wrong with this code?


```
import java.util.Arrays;
import java.util.List;

// High Level Module
class ProductCatalog {

    public void listAllProducts() {

        SQLProductRepository sqlProductRepository = new
SQLProductRepository();

        List<String> allProductNames =
sqlProductRepository.getAllProductNames();

        // Display product names

    }

}

// Low Level Module
class SQLProductRepository {

    public List<String> getAllProductNames() {
        return Arrays.asList("soap", "toothpaste", "shampoo");
    }

}
```

What do you think? Do you see a problem? `Produc`

but it depends on its submodule `SQLProductReposi`

This is a mistake often made by novice developers.

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

You are now seeing the violation of **Dependency Inversion Principle** because High Level Module `ProductCatalog` depends on its submodule `SQLProductRepository`.

Now you understand the problem. Then how do you fix it? Before you fix it, let's see what `abstraction` means in software design.

What is Abstraction?

(1) Code without abstraction

```
class Benz  {

    public void drive() {

    }

}

class CarUtil {

    public static void drive(Benz benz) {
        benz.drive();
    }

}
```

As you can see the code above, CarUtil class's static drive method is dependent on Benz. You should provide `Benz` instance in order for CarUtil's `drive()` method to function. In software design, it is called 'tight-cou
change `drive()` method inside `Benz` class, CarUti
make bugs.

> *Tight Coupling is the most undesirable feature in Software*


```
interface Car {
    public void drive();
}

class Benz implements Car {

    @Override
    public void drive() {


    }

}

class Tesla implements Car {

    @Override
    public void drive() {

    }

}

class CarUtil {

    public static void drive(Car car) {
        car.drive();
    }

}
```

This code looks perfect. CarUtil's static drive method doesn't depend on Benz, but it depends on `Car` interface. Now, it can take any argument which implements Car Interface. This is called abstraction. It is also called 'loose-coupling'.

Now, let's go back to the previous code example.

Refactoring Previous Code with Abstracti

> *"Abstractions should not depend on details. Details should depend on*


```
// High Level Module
class ProductCatalog {

    public void listAllProducts() {

        // High Level Module depends on Abstraction
        ProductRepository productRepository =  new
SQLProductRepository();

        List<String> allProductNames =
productRepository.getAllProductNames();

        // Display product names

    }

}

interface ProductRepository {

    List<String> getAllProductNames();

}

// Low Level Module
class SQLProductRepository implements ProductRepository {

    public List<String> getAllProductNames() {
        return Arrays.asList("soap", "toothpaste", "shampoo");
    }

}
```

Now, `ProductCatalog` depends on `ProductRepository` instead of `SQLProductRepository`.

Why doing this again?

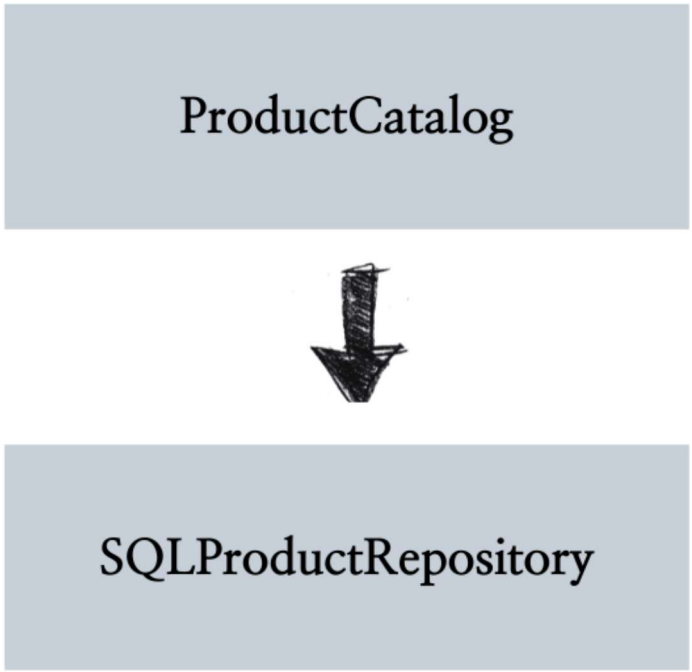
First, you don't know what database you are going to use. It may not be specifically `SQL`.

Second, `ProductCatalog`'s `listAllProducts()` doe

This means, when you change code in `SQLProductI`

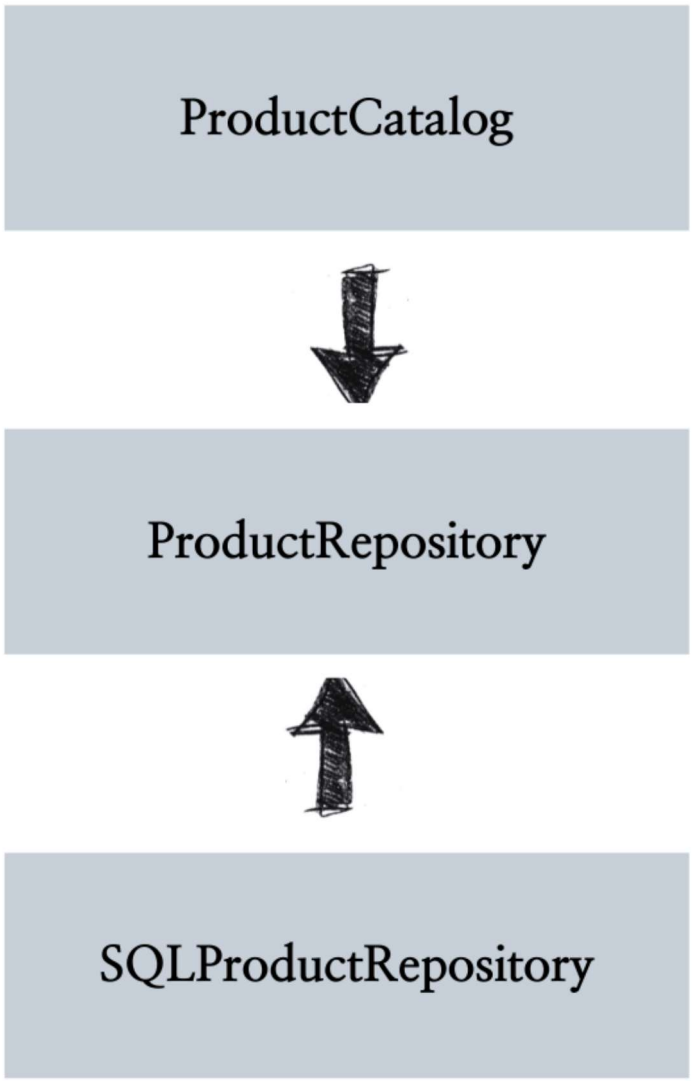
directly affected. You just have achieved `loose-coupling`.

What you have achieved now



Before refactoring

Before refactoring, `ProductCatalog` was dependen



After Refactoring

After Refactoring, `ProductCatalog` depends on `ProductRepository` and `SQLProductRepository` is also dependent on `ProductRepository` .

Let me remind you of Dependency Inversion Princ

> ***"High-level modules should not depend on low-l***
> ***on abstractions."***

One strep forward, Dependency Injection

```
import java.util.Arrays;
import java.util.List;

class ProductCatalog {

    private ProductRepository productRepository;

    public ProductCatalog(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public void listAllProducts() {

        List<String> allProductNames =
productRepository.getAllProductNames();

        // Display product names

    }

}

interface ProductRepository {

    List<String> getAllProductNames();

}

class SQLProductRepository implements ProductRepository {

    public List<String> getAllProductNames() {
        return Arrays.asList("soap", "toothpaste", "shampoo");
    }

}

class EcommerceApplication {

        public static void main(String[

            ProductRepository productRe
SQLProductRepository();

            ProductCatalog productCatal
ProductCatalog(productRepository);
```

You're now injecting `ProductRepository` to `ProductCatalog`. This is a common practice and even recommended way to build objects. You will understand its usefulness when you learn Mock and TDD concepts. But if I explain it now, I can't keep the promise of **"The Easiest Tutorial of The World".**

Conclusion

- "High-level modules should not depend on low-level modules. Both should depend on abstractions."
- "Abstractions should not depend on details. Details should depend on abstractions."

If you can understand these two statements, you have fully understood Dependency Inversion Principle! Congratulations!

But if not, don't worry. You can't easily understand Dependency Inversion Principle unless you have experience. This also needs 'intentional practice'.

Other Articles for Solid Principles

Single Responsibility Principle: What? working as "a software engineer" you don't know it?!?!

Have you ever experienced that code refactoring doesn't make your code better? Adding spaghetti on top of spaghetti...

paigeshin1991.medium.com

Open Closed Principle: Make your code cost-free and flexible

Liskov Substitution Principle: Top Developer's technique which improves 2.5x

As you write code, doesn't anyone ever question whether your code is logically correct or consistent enough? To solve...

paigeshin1991.medium.com

Interface Segregation Principle: Your only way to be a super competent developer

During development, the code tends to be long. At some point, the codebase becomes so bloated that you can't handle it...

paigeshin1991.medium.com

Dependency Inversion Principle: How Google Developers write code

To become a high-paid developer, you need to learn TDD. Basically, you have to develop software with TDD to get into a...

paigeshin1991.medium.com