Published in 程式爱好者

YC Following
Sep 15, 2020 · 4 min read

Save

使人疯狂的SOLID 原则： 接口隔离原则(Interface Segregation Principle)

> 今天我们要说的是第四个原则：*接口隔离原则* (ISP)。

如果还不知道什么是接口的话，可以先看面向对象中的接口与抽象类别是什么？

接口隔离原则其实并不困难。就如定义所说，因为模块之间的依赖不应有用不到的功能，所以我们可以透过接口来进行分割，把模块分得更合符本身的角色，也让使用接口的角色只能分别接触到应有的功能。

我先举一个简单的例子来说明何为违反 ISP：

```
class Car {

    public function void openEngineMode() { /*...*/ }

    public function void repairWheel() { /*...*/ }

    public function void startEngine() { /*...*/ }

    public function void move() { /*...*/ }

}

class Driver {
    Car myCar = new Car();
    myCar.startEngine();
    myCar.move();
    myCar.openEngineMode(); // 为什么我什么都不会就可以开启工程模式呢?
}

class Mechanic {
    Car clientCar = new Car();
    clientCar.repairWheel();
    clientCar.openEngineMode();
}
```

上例中，我们看到一个普通人就可以使用 Car 类别中的 openEngineMode() 功能，但这功能其实只是给工程师用的，普通人用很容易会把汽车用坏。这就是一个没有把功能隔离好的例子。

Car 的部分功能给了不该用到这些功能的角色使用。

这时我们可以用 interface 来做优化，如下：

```
interface DailyUsage {
    public function void [illegible]
    public function void move();
}

interface RepairUsage {
    public function void openEngineMode();
    public function void repairWheel();
}

class Car implement DailyUsage, RepairUsage {

    public function void openEngineMode() { /*...*/ }

    public function void repairWheel() { /*...*/ }

    public function void startEngine() { /*...*/ }

    public function void move() { /*...*/ }

}

class Driver {
    DailyUsage myCar = new Car();
    myCar.startEngine();
    myCar.move();
}

class Mechanic {
    RepairUsage clientCar = new Car();
    clientCar.repairWheel();
    clientCar.openEngineMode();
}
```

透过 interface 来为我们的 Car 类别的不同功能分类，然后不同的角色再分别引用不同的功能。Mechanic 的程式将依赖于 RepairUsage 和 openEngineMode() ，但不再依赖于 Car。那 Mechanic 也不用关心 DailyUsage 和 Driver 的修改。

那架构层面来看呢?

好比当程式是依赖于 Framework 而 Framework 又赖依于 Database 时，当 Database 更换，如从 MySQL 换到 MongoDB，程式将会直接无法使用。

所以多使用接口来进行解藕、把实作隐藏起来、保持抽象，有助我们程式的弹性。

如果你觉得文章帮助到你，希望你也可以为文章拍手，分别 Follow 我的个人页与程式爱好者出版，按赞我们的粉丝页喔，支持我们推出更多更好的内容创作！

Solid Object Oriented Software Development Isp Software Architecture