面向对象编程之七大设计原则

作者：valarchie
原文链接：https://ld246.com/article/1566313342510
来源网站：链滴

七大设计原则 — SOLID

这六大原则是业界在面向对象设计中经过总结精炼得出，在英文表示下各个原则
首字母缩写就是SOLID。

- Single Responsibility Principle：单一职责原则
- Open Closed Principle：开闭原则
- Liskov Substitution Principle：里氏替换原则
- Interface Segregation Principle：接口隔离原则
- Dependence Inversion Principle：依赖倒置原则

另外还有两个设计原则。

- Law of Demeter：迪米特法则
- Composite/Aggregate Reuse Principle：组合/聚合复用原则

单一职责原则(SRP)

概念：

一个类，应该只有一个引起它变化的原因。通俗的讲就是一个类应该只负责一个职责，如果这个类需修改的话，也只是因为这个职责的变化了才引发类的修改。

例子:

```
public class Car {
    /**
     * 启动引擎
     */
    public void start() {
        System.out.println("车辆启动了！");
    }
    /**
     * 熄火
     */
    public void stop() {
        System.out.println("车辆熄火了！");
    }
    /**
     * 加速
     */
    public void speed() {
        System.out.println("车辆加速中！");
    }
    /**
     * 接送乘客
     */
    public void pickUpPassenger(){
```

```
        System.out.println("接送乘客中！");
    }
    /**
     * 去加油站加油
     */
    public void gasUp(){
        System.out.println("去加油站加汽油！");
    }
}
```

在以上的例子当中，如果汽车启动的参数变化了就需要修改这个Car类，如果接送乘客的规则改变的也需要修改这个类当中的代码。而正确的设计应该将接送客人的方法和去加油站加油的方法提取出来予TaxiDriver类中。将代码修改带来影响的范围限定得越小越好。一旦类包含的职责越来越多的话，就会变得低内聚高耦合。

```
public class Car {
    /**
     * 启动引擎
     */
    public void start() {
        System.out.println("车辆启动了！");
    }
    /**
     * 熄火
     */
    public void stop() {
        System.out.println("车辆熄火了！");
    }
    /**
     * 加速
     */
    public void speed() {
        System.out.println("车辆加速中！");
    }
}

class TaxiDriver{
    /**
     * 汽车实例
     */
    public Car car;
    /**
     * 接送乘客
     */
    public void pickUpPassenger(){
        System.out.println("接送乘客中！");
    }

    /**
     * 去加油站加油
     */
    public void gasUp(){
        System.out.println("去加油站加汽油！");
    }
```

```
}
```

在以上这个例子中就将汽车的操作与接送乘客、加油的操作进行了解耦。

优点：

1. 降低了类的复杂性。
2. 提高类的可读性，提高系统的可维护性。
3. 降低变更引起的风险（降低对其他功能的影响）。

开闭原则(OCP)

概念：

一个实体（类、函数、模块等）应该对外扩展开放，对内修改封闭。某实体应该易于扩展，在扩展某的功能时应该通过添加新的代码来实现而不是修改其内部的代码。

例子:

```
public class Driver {
    public void drive(Car car) {
        // 做一些驾驶前准备
        if ("BMW".equals(car.getBrand())) {
            System.out.println("驾驶宝马车！");
        }
        if ("BENZ".equals(car.getBrand())) {
            System.out.println("驾驶奔驰车！");
        }
    }
}

class Car {
    /**
     * 汽车品牌
     */
    private String brand;

    public String getBrand() {
        return brand;
    }
    public void setBrand(String brand) {
        this.brand = brand;
    }
}
```

在上面的例子当中，司机类当前只会驾驶宝马车和奔驰车，那如果我想要让司机也需要驾驶奥迪车呢这个时候我们就需要去修改司机的drive方法了。那新增某一个功能却需要去修改原本的代码时，很易引发bug，而原本的代码时经过测试好的，一经修改的话又需要进行重新的测试。而所有引用到此法的地方也需要进行修改，维护的成本就变得极高。

```
public class Driver {
```

```
    public void drive(Car car) {
        // 做一些驾驶前准备
        car.start();
    }
}

class Car {
    /**
     * 汽车品牌
     */
    private String brand;

    public String getBrand() {
        return brand;
    }
    public void setBrand(String brand) {
        this.brand = brand;
    }
    public void start() {

    }

}

class BMW extends Car {
    public void start() {
        System.out.println("驾驶宝马车！");
    }
}

class BENZ extends Car {
    public void start() {
        System.out.println("驾驶奔驰车！");
    }
}
```

在以上的例子当中，如果需要给司机在新增一种品牌车型的驾驶技能的时候，就不需要去修改原本的drve方法了，而仅需要新增一种品牌车型，并重写Car类中的start方法即可。

还有一个比较通俗的例子：商品的价钱需要根据会员等级进行打折，如果在获取价钱的方法当中直接根据会员等级计算出价钱返回，就违反了开闭原则。而是应该通过关闭对商品价钱方法的修改，新增一个会员类，通过调用商品获取价钱的方法和会员等级计算出价钱进行返回。

优点：

1. 对类的功能扩展变得灵活。
2. 扩展变得灵活的话，维护性自然就提高了。

里氏替换原则(LSP)

概念：

任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当子类可以替换掉基类软件单位的功能不受到影响时，基类才能真正被复用。个人理解里氏替换原则是用来检验继承是否合的原则。

例子：

```
public class Bird {
    // 飞行速度
    int velocity;
    // 飞行操作
    public void fly() {
        System.out.println("扑打翅膀！ ");
        velocity = 20;
    }
    public int getVelocity() {
        return velocity;
    }
    public void setVelocity(int velocity) {
        this.velocity = velocity;
    }
}

/**
 * 鸵鸟类
 */
class Ostrich extends Bird {
    // 由于鸵鸟不会飞 所以方法里面为空
    public void fly() {
        //I do nothing
    }
}


/**
 * 测试鸟类
 */
class TestBird {
    public void getFlyTime() {
        // 测试普通鸟类 将会得到飞行时间为10分钟
        Bird bird = new Bird();
        int flyTimeBird = 200 / bird.getVelocity();
        // 测试鸵鸟 将会得到无限的飞行时间 程序将直接报错
        Bird ostrich = new Ostrich();
        int flyTimeOstrich = 200 / ostrich.getVelocity();
    }
}
```

在以上的例子当中，Bird类可以计算出飞行时间，而Ostrich类计算飞行时间就会出错。这种就是子类法替换父类的例子。而当你的同事调用计算飞行时间方法的时候，并不会去查看每个子类中的方法实，所以就会得出莫名其妙的结果。

原文链接：面向对象编程之七大设计原则

里氏替换原则的实践可以归纳为以下四点：

- 子类必须实现父类的抽象方法，但不得重写（覆盖）父类的非抽象（已实现）方法。
- 子类中可以增加自己特有的方法。
- 当子类覆盖或实现父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽。
- 当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。

优点：

1. 规范了父类与子类之间安全的继承，继承是安全的，代码的复用才是安全的。
2. 继承也提高了代码可维护性，方便修改父类的公共方法和子类的特定方法。

依赖倒置原则(DIP)

概念：

依赖倒置原则（Dependence Inversion Principle）是程序要依赖于抽象接口，不要依赖于具体实现。简单的说就是要求对抽象进行编程，不要对实现进行编程，这样就降低了客户与实现模块间的耦合。

例子：

```
public class Driver {
    public void drive(BMW car) {
        car.start();
    }
}
class BMW {
    public void start() {
        System.out.println("驾驶宝马车！");
    }
}
```

在以上的例子中呢，司机类直接依赖于宝马车进行驾驶，如果有一天我们想让司机开奔驰车呢？只能修改原方法的参数类型，改为奔驰类。而我们一开始就为所有车抽象出一个抽象车类呢？此时就不用心去修改原方法了，只需要传参的时候传入奔驰车即可。

例子：

```
public class Driver {
    // 此时可以传BMW车，也可以传入BENZ车
    // 如果想让司机换开奔驰车的话，不需要更改这部分的代码
    public void drive(Car car) {
        car.start();
    }
}
```

```
class Car {
    void start() {

    }
}

class BMW extends Car{
    public void start() {
        System.out.println("驾驶宝马车！");
    }
}
class BENZ extends Car{
    public void start() {
        System.out.println("驾驶奔驰车！");
    }
}
```

这种依赖抽象的例子很多，例如我们在开发的时候定义DAO，在Service当中调用这些DAO时是通过明接口引用而不是通过声明实现类引用。假设我们需要将数据库框架从Hibernate转到Mybatis的时，Service可以毫无感知的无缝切换。因为之前Service当中并无与实现类耦合。还有比如我们项目中日志工具，记录日志依赖的都是规范好的日志接口，如果需要从log4j迁移到logback也是对代码无侵的。

优点：

1. 减少类之间的耦合。
2. 低耦合使得代码更容易进行维护和扩展。

接口隔离原则(ISP)

概念：

客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。接口隔离原和单一职责原则很像。单一职责是从对象的职责上面去限定，而接口隔离原则希望每个接口都是最小接口，接口小的话才使得复用接口变得更加容易。

例子：

```
/**
 * 交通工具类
 */
public interface Vehicle {
    // 飞行
    void fly();
    // 航行
    void sail();
    // 陆行
    void run();
}

class AirPlane implements Vehicle {
```

```
    @Override
    public void fly() {
        System.out.println("飞行");
    }
    @Override
    public void sail() {

    }
    @Override
    public void run() {

    }
}
```

在以上的例子中，交通工具类集合了所有交通的方式，这种方式不可取，如果飞机实现这个接口，却要去实现与飞机不相关的航行和陆行方法。我们应该缩小接口的粒度。

```
/**
 * 飞行接口
 */
public interface Fly {
    // 飞行
    void fly();
}

/**
 * 航行接口
 */
interface Sail {
    // 航行
    void sail();
}


/**
 * 陆行接口
 */
interface Run {
    // 陆行
    void run();
}

/**
 * 飞行器类
 */
class AirCraft implements Fly {
    @Override
    public void fly() {
        System.out.println("飞行");
    }
}
```

优点：

1. 避免大接口被许多子类实现，造成耦合。降低了耦合，代码也变得好维护。

2. 小接口可以赋予特定的含义，使得代码更好理解（例如Comparable接口和Serialization接口一目然）。

3. 减少没必要实现的冗余代码。

组合/聚合复用原则(CARP)

概念：

尽量使用组合和聚合，尽量少使用继承。为什么呢？继承不是面向对象的良好特性吗？

继承有很多局限性。首先，继承属于一种硬编码。如果没有遵守里氏替换原则，父类一旦修改，所有类都需要进行改变。

例子:

```
/**
 * 手机类有打电话、游戏、音乐功能
 */
public class Phone extends GameBoy {
    // 拨打电话
    void call() {
        System.out.println("播放电话");
    }
}

/**
 * 播放音乐类
 */
class Pods {
    void playMusic() {
        System.out.println("播放音乐");
    }
}

/**
 * 游戏机类，有游戏和音乐功能
 */
class GameBoy extends Pods{
    void playGame() {
        System.out.println("玩游戏");
    }
}
```

在以上的例子当中，如果Pods类增加了播放磁带的功能，而手机类并不需要播放磁带音乐的功能。或游戏机类增加了摇杆按键功能，而我们的手机同样不需要这个功能。这种继承会导致手机类平台无故继承了不需要的方法。或者这时候我们新出了一个智能手机类SmartPhone类，又想使用之前的所有能呢？再一次继承Phone类，使得继承的层级更深了，各个类耦合得更紧密。还有一点，Gameboy也不一点需要播放音乐的功能。

在下面的例子，我们使用组合/聚合的方式来实现我们的手机类。

```
public class Phone {
    private Pods pods;
    private GameBoy gameBoy;
    // 拨打电话
    void call() {
        System.out.println("播放电话");
    }
    // 播放音乐
    void playMusic() {
        pods.playMusic();
    }
    // 玩游戏
    void playGame() {
        gameBoy.playGame();
    }
}

/**
 * 播放音乐类
 */
class Pods {
    void playMusic() {
        System.out.println("播放音乐");
    }
}

/**
 * 游戏机类，有游戏和音乐功能
 */
class GameBoy {
    void playGame() {
        System.out.println("玩游戏");
    }
}
```

此时我们通过组合的方式，去除掉了之前的继承耦合。

优点：

1. 修改各个复用到的类变得容易，不用担心会影响到其子类。
2. 可以动态的替换各个复用类。
3. 各个复用类各司其职，在其他地方一样也可以使用，提高了代码复用。

迪米特法则(LOD)

概念：

又叫作最少知识原则（Least Knowledge Principle 简写LKP），就是说一个对象应当对其他对象有可能少的了解,不和陌生人说话。也就是说类应该尽可能地少的了解其他对象的细节。如果对象A知道象B的所有细节，那么对象A就可能会去使用到这些细节。如果你修改了其中对象B中的细节，就会不意影响到A。

例子:

```
/**
 * 煮汤类
 * 一共有四个步骤
 */
public class CookSoup {
    void addWater() {
        System.out.println("加水");
    }
    void addFood() {
        System.out.println("加食物");
    }
    void addSalt() {
        System.out.println("加盐");
    }
    void heat() {
        System.out.println("加热");
    }
}

/**
 * 这个例子当中用户知道煮汤的步骤
 */
class TestCookSoup {
    public void testCook() {
        CookSoup cookSoup = new CookSoup();
        cookSoup.addWater();
        cookSoup.addFood();
        cookSoup.addSalt();
        cookSoup.heat();
        System.out.println("得到一碗汤");
    }
}
```

再以上的这个例子中，用户必须知道煮汤的顺序，如果不知道的话，就会把汤煮坏。又或者煮汤类修了一条规则，addFood()加食物方法前一定要将食物切好。那么所以引用这段代码的地方都需要进行加切食物的动作。对其他对象了解得越多，或是了解越多对象都会导致对象之间的强烈耦合，一旦耦的话，修改一处代码就会造成其他耦合对象也需要跟着更改。

```
/**
 * 煮汤类
 * 一共有四个步骤
 */
public class CookSoup {
    void addWater() {
        System.out.println("加水");
    }
    void addFood() {
        System.out.println("加食物");
    }
    void addSalt() {
        System.out.println("加盐");
    }
```

```
    void heat() {
        System.out.println("加热");
    }
    /**
     * 封装煮汤步骤
     */
    void cook() {
        addWater();
        addFood();
        addSalt();
        heat();
    }
}


/**
 * 这个例子当中用户并不知道煮汤的具体步骤
 */
class TestCookSoup {
    public void testCook() {
        CookSoup cookSoup = new CookSoup();
        cookSoup.cook();
        System.out.println("得到一碗汤");
    }
}
```

在这个例子中，调用者不需要知道煮汤的步骤，直接调用cook()方法即可，就不会再煮出一锅难吃的。而如果厨师想在煮汤操作中间添加一些特殊操作的话，也不会影响到调用煮汤的代码。因为调用者不知道真正的煮汤方法。

优点：

1. 降低了类之间的耦合。
2. 提高了可维护性。
3. 减少了代码的可维护性。