六大原则之“里氏替代原则（LSP）“笔记

1. 里氏替代原则，(Liskov Substitution Principle, LSP)

定义：Functions that use pointers or referrnces to base classes must be able to use objects of derived classes without knowing it.(所有引用基类的地方必须能透明地使用其子类的对象。)

2.理解：只要父类能出现的地方，子类就可以出现，并且替换为子类也不会产生任何错误或异常，使用者可能根本就不需要知道是父类还是子类。但反之，未要求。

继承机制的优点：

- 代码共享，减少创建类的工作量；
- 提高代码的重用性；
- 子类可以形似父类，又异于父类；
- 提高父类的扩展性，实现父类的方法即可随意而为；
- 提高产品或项。

继承机制的缺点：

- 继承是入侵性的（只要继承，就必须拥有父类的所有属性与方法）；
- 降低了代码的灵活性（子类拥有了父类的属性方法，会增多约束）；
- 增强了耦合性（当父类的常量、变量、方法被修改时，必需要考虑子类的修改）。

定义所包含的四层意思：（另一种通俗的LSP原则讲解：子类可以扩展父类的功能，但不能改变父类原有的功能）

1. 子类可以实现父的抽象方法，但不能覆盖父类的非抽象方法（做系统设计时，经常会定义一个接口或抽象类，然后编码实现（定义的方法或接口），调用类则直接传入接口或抽象类，这里也是LSP的应用体现）；
2. 子类可以有自己的方法和属性（因为LSP可以正用，不能反用：在子类出现的地方，父类未必就可以胜任（本来就无此要求）。即不要死扣——所有的地方（如参数）都要以父类的形式出现（再作转化），实现中有需要依赖子类的情况，这是正常的）；
3. 覆盖或实现父类的方法时输入参数可以被放大（放大的实质为重载，因为参数不同；为什么只能放大？因为父类方法的参数类型相对较小，所以当传入父类方法的参数类型（或更窄类型）时，重载时，将优先匹配父类的方法，因此子类重载的方法并不会对此参数类型被执行，因此保证了LSP，且不会引起想不到的业务逻辑混乱。若为覆写，则程序员必清楚其逻辑要义）；
4. 覆写或实现父类的方法时输出结果可以被缩小（若放大，还能用子类替换父类吗？）。

Liskov替换原则并不是要求子类不能新增父类没有的方法或者属性。因为从调用父类的客户程序的角度来说，它关心的仅仅是父类的行为，只要子类对于父类的行为是可替换的，就不算是违背该原则。

恰恰相反，当你发现父类拥有子类不希望继承，或者勉强继承会对子类造成破坏时 ，正可以说明这个继承体系可能存在问题，违背了Liskov替换原则。这就充分说明，子类并不关心父类的行为，但却需要遵循父类制定的规范或契约，以满足客户调用父类的期望。正所谓"萧规曹随"，如果前人制定的规范我们不遵循，反而要去打破，那就不是继承，而是铁了心要另起炉灶了。
一个经典的违反Liskov替换原则的例子是正方形与矩形之间的关系。这样的例子在谈对象设计的原则时，已经啰嗦得够多，这里我就不再赘述了 。这个例子带来的教训就是，现实世界中继承的例子，不能够完全直接套用在程序世界中。不过，作为设计的参照物，现实世界的很多规律与法则，我们仍然不可忽视。例如鲸鱼和鱼，应该属于什么关系？从生物学的角度看，鲸鱼应该属于哺乳动物，而不是鱼类。

3.问题由来

有一功能P1，由类A完成。现需要将功能P1进行扩展，扩展后的功能为P，其中P由原有功能P1与新功能P2组成。新功能P由类A的子类B来完成，则子类B在完成新功能P2的同时，有会导原有功能P1发生故障。解决方案：遵循LSP：类B在继承类A时，除添加新的方法完成新增功能P2外，尽量不要重写父类A的方法，也尽量不要重载父类的A的方法。

即为正确使用继承的好处。

5.1 如何根据类的继承原则，确定是要继承当前类结构，还是要另起炉赵。

5.2 如何根据情况，在违背Liskov替换原则时，提出一种方案。

6.实践建议

6.1 在类中调用其它类时务必要使用父类或接口，如果不能使用父类或接口，则说明类的设计已违背了LSP；（例：Interface in = new Instance();）；

6.2 如果子类不能完整地实现父类的方法，或者父类的某些方法在子类中已经发生“畸变”，则建议断开父子继承关系，采用依赖、聚集、组合等关系代替继承。

6.3 如果你的程序中if/else之类对子类类型进行判断的条件，则说明类的设计已违背了LSP。

7.1 矩形与正方形（现实中的继承，却不能直接用于程序中）

对于长方形的类，如果它的长宽相等，那么它就是一个正方形，因此，长方形类的对象中有一些正方形的对象。对于一个正方形的类，它的方法有个setSide和getSide，它不是长方形的子类，和长方形也不会符合LSP。

```
1. //长方形类:
2. publicclass Rectangle{
3.    setWidth( width){
4.       .width=width;
5.    setHeight( height){
6.       .height=height
7. //正方形类:
8. publicclass Square{
9.    setWidth( width){
10.      .width=width;
11.      . height=width;
12.   setHeight( height){
13.      .setWidth(height);
14. //例子中改变边长的方法:
15. public resize(Rectangle r){
16.   while(r.getHeight()<=r.getWidth){
17.      r.setHeight(r.getWidth+
```

```
            //长方形类:
public class Rectangle{
    ...
    setWidth(int width){
            this.width=width;
    }
    setHeight(int height){
            this.height=height
```

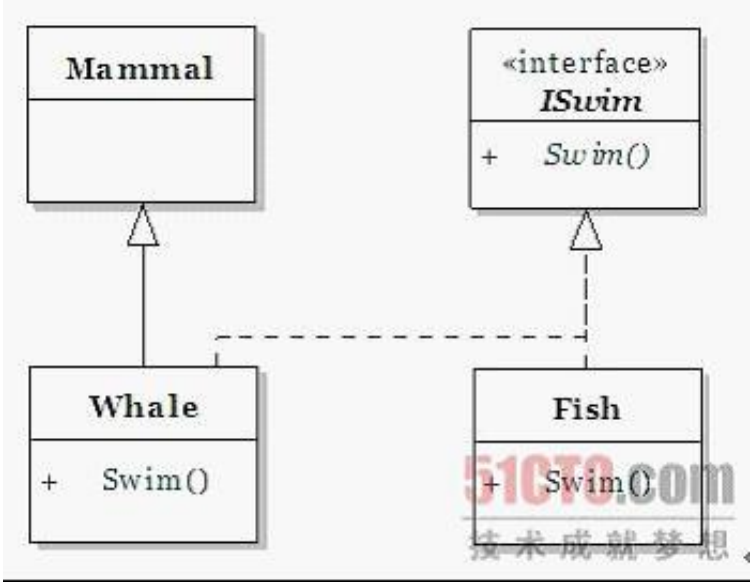
```
    }
}
//正方形类:
public class Square{
    ...
    setWidth(int width){
            this.width=width;
            this. height=width;
    }
    setHeight(int height){
            this.setWidth(height);
    }
}
//例子中改变边长的方法:
public void resize(Rectangle r){
    while(r.getHeight()<=r.getWidth){
            r.setHeight(r.getWidth+1);
    }
}
```

那么，如果让正方形当做是长方形的子类，会出现什么情况呢？我们让正方形从长方形继承，然后在它的内部设置width等于height，这样，只要width或者height被赋值，那么width和height会被同时赋值，这样就保证了正方形类中，width和height总是相等的.现在我们假设有个客户类，其中有个方法，规则是这样的，测试传入的长方形的宽度是否大于高度，如果满足就停止下来，否则就增加宽度的值。现在我们来看，如果传入的是基类长方形，这个运行的很好。根据LSP，我们把基类替换成它的子类，结果应该也是一样的，但是因为正方形类的width和height会同时赋值，这个方法没有结束的时候，条件总是不满足，也就是说，替换成子类后，程序的行为发生了变化，它不满足LSP。

7.2 鲸鱼与鱼（避开违背LSP的一种解决方案）

鲸鱼和鱼，应该属于什么关系？从生物学的角度看，鲸鱼应该属于哺乳动物，而不是鱼类。没错，在程序世界中我们可以得出同样的结论。如果让鲸鱼类去继承鱼类，就完全违背了Liskov替换原则。因为鱼作为父类，很多特性是鲸鱼所不具备的，例如通过腮呼吸，以及卵生繁殖。那么，二者是否具有共性呢？有，那就是它们都可以在水中"游泳"，从程序设计的角度来说，它们都共同实现了一个支持"游泳"行为的接口。

如下所示的设计，可以看做是解决违背Liskov替换原则的一种常规方案，即提取两者之间的共同点，定义一个更为通用的接口，或者新的父类。



7.3 继承的风险

举例说明继承的风险，我们需要完成一个两数相减的功能，由类A来负责：

```
1. class
2.     public func1(
3.         return
4. publicclass Client{
5.     publicstatic main(String[] args){
6.         System.out.println("100-50="+a.func1(
7.         System.out.println("100-80="+a.func1(
8. 运行结果：
```

```
class A{
        public int func1(int a, int b){
                return a-b;
        }
}
public class Client{
        public static void main(String[] args){
                A a = new A();

                System.out.println("100-50="+a.func1(100, 50));
                System.out.println("100-80="+a.func1(100, 80));
        }
}
```

运行结果：

```
100-50=50
100-80=20
```

后来，我们需要增加一个新的功能：完成两数相加，然后再与100求和，由类B来负责。即类B需要完成两个功能：

- 两数相加，再加100

由于类A已经实现了第一个功能，所以类B继承类A后，只需要再完成第二个功能就可以了，代码如下：

```
1. classextends
2.     public func1(
3.         return
4.     public func2(
5.         return func1(a,b)+
6. publicclass Client{
7.     publicstatic main(String[] args){
8.         System.out.println("100-50="+b.func1(
9.         System.out.println("100-80="+b.func1(
10.        System.out.println("100+20+100="+b.func2(
11. 运行结果：
```

```
class B extends A{
        public int func1(int a, int b){
                return a+b;
        }
        public int func2(int a, int b){
                return func1(a,b)+100;
        }
```

```
}
public class Client{
        public static void main(String[] args){
                B a = new B();
                System.out.println("100-50="+b.func1(100, 50));
                System.out.println("100-80="+b.func1(100, 80));
                System.out.println("100+20+100="+b.func2(100, 20));
        }
}
```

运行结果：

```
100-50=150
100-80=180
100+20+100=220
```

我们发现原来运行正常的相减功能发生了错误。原因就是类B在给方法起名时无意中重写了父类的方法，造成所有运行相减功能的代码全部调用了类B的重写后的方法，造成原本运行正常的功能出现了错误。
在本例中，使用A类完成的功能，换成子类B之后，发生了异常。在实际编程中，我们常会通过重写父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的几率非常大。如果非要重写父类的方法，比较通用的做法是：原来的父类和子类都继承一个更通谷的基类，原来的继承关系去掉，采用依赖、聚合、组合等关系替代。

Links