软件设计原则----LisKov替换原则（LSP）

2011-10-21 10:32 2170人阅读 评论(0) 收藏[1] 举报

“一个软件实体如果使用的是一个基类的话，一定适用于其子类，而且根本不能觉察出基类对象和子类对象的区别。”

陈述：

- 子类型（Subtype）必须能够替换他们的基类型(Basetype)

Barbara Liskov对原则的陈述：
若对每个类型S的对象o1,都存在一个类型T的对象o2，使得在所有针对T编写的程序P中，用o1替换o2后，程序P的行为功能不变，则S是T的子类型。

通俗地讲，就是子类型能够完全替换父类型，而不会让调用父类型的客户程序从行为上有任何改变。

我们在客户程序在调用某一个类时，实际上是对该类的整个继承体系设定了一套约束，继承体系中的所有类必须遵循这一约束，即前置条件和后置条件必须保持一致。这为对象继承加上了一把严格的枷锁。显然，**LSP**原则对于约束继承的泛滥具有重要意义。

分析：

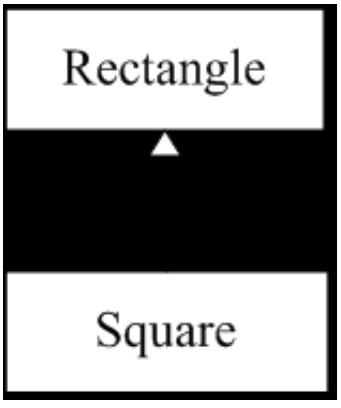
- 违反这个职责将导致程序的和对OCP的违反

例如：基类Base，派生类Derived，派生类实例d,函数f(Base* p);

- f(&d) 会导致错误

  显然D对于f是脆弱的。
- 如果我们试图编写一些测试，以保证把d传给f时可以使f具有正确的行为。那么这个测试违反了OCP——因为f无法对Base的所有派生类都是封闭的。

经典例子：长方形与正方形驳论




```
1. class Rectangle
2. private
3.     width;
```

```
4.      height;
5. public
6.     setWidth( width)
7.       ->width = width;
8.     getWidth()
9.       return->width;
10.    setHeight( height)
11.      ->height = height;
12.    getHeight()
13.      return->height;
14. //正方形类
15. class Square
16. private
17. <span style="white-space: pre; ">  </span> side;
18. public
19. <span style="white-space: pre; ">  </span> setSide( side)
20.      ->side = side;
21.    getSide()
22.      return side;
```

```
class Rectangle
{
private:
     long width;
     long height;
public:
    void setWidth(long width)
    {
        this->width = width;
    }
    long getWidth()
    {
        return this->width;
    }
    void setHeight(long height)
    {
        this->height = height;
    }
    long getHeight()
    {
        return this->height;
    }
};

//正方形类
class Square
{
private:
long side;

public:
```

```
void setSide(long side)
    {
        this->side = side;
    }
    long getSide()
    {
        return side;
    }
};
```

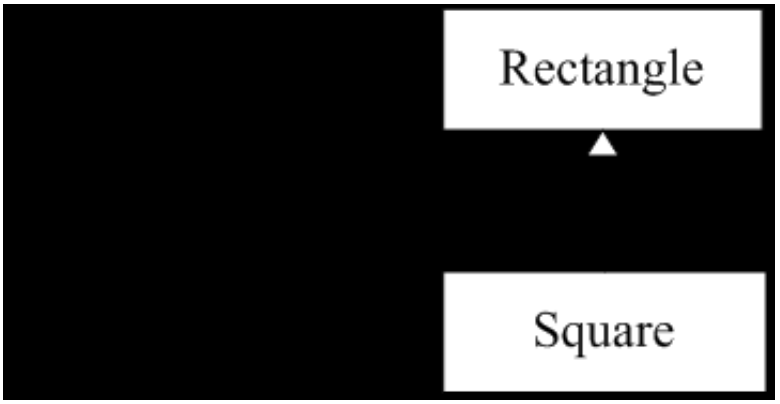
1. //正方形类（如果继承自长方形类）：
2. class Square : public Rectangle
3. private
4.   side;
5. public
6.   setWidth(width)
7. setSide(width);
8. getWidth()
9. return getSide();
10. setHeight(height)
11. setSide(height);
12. getHeight()
13. return getSide();
14. getSide()
15. return side;
16. setSide(side)
17. ->side = side;

```
//正方形类（如果继承自长方形类）：
class Square : public Rectangle
{
private:
long side;
public:
void setWidth(long width)
    {
        setSide(width);
    }
    long getWidth()
    {
        return getSide();
    }
    void setHeight(long height)
    {
        setSide(height);
    }
    long getHeight()
    {
        return getSide();
    }
    long getSide()
```

```
    {
        return side;
    }
    void setSide(long side)
    {
        this->side = side;
    }
};
```

```
1. class SmartTest
2. public
3.     resize(Rectangle r)
4.    while (r.getHeight() <= r.getWidth() )
5.       r.setWidth(r.getWidth() + 1);
```

```
class SmartTest
{
public:
    void resize(Rectangle r)
    {
        while (r.getHeight() <= r.getWidth() )
        {
            r.setWidth(r.getWidth() + 1);
        }
    }
};
```



从上面小函数可见，只想改变长方形的宽时，如果把正方形看成一种长方形的话，则正方形的长和宽都被改变了。LSP原则被破坏了，Square不应成为Rectangle的子类。

里氏代换与通常的数学法则和生活常识有不可混淆的区别。
考虑一个设计是否恰当时，不能孤立的看待并判断，应该从此设计的使用者所作出的假设来审视它！

这个看似明显正确的模型怎么会出错呢?
“正方形是一种长方形”
对不是SmartTest函数的编写者而言，正方形可以是长方形，但是对SmartTest函数的编写者而言，Square绝对不是Rectangle！！

OOD中对象之间是否存在**IS-A**关系，应该从行为的角度来看待。

->而行为可以依赖客户程序做出合理的假设。

引入一个Quadrangle（四边形）类，并将Rectangle 与Square变成它的具体子类，解决了Rectangle 与Square的关系不

符合里氏替换原则的问题。

```
1. class Quadrangle
2. public
3.     virtual getWidth() = 0;
4.     virtual getHeight() = 0;
```

```
class Quadrangle
{
public:
        virtual long getWidth() = 0;
        virtual long getHeight() = 0;
};
```

Quadrangle类只声明两个取值方法，不声明任何的赋值方法。

//长方形类：

```
1. class Rectangle : public Quadrangle
2. private
3.     width;
4.     height;
5. public
6.     setWidth( width)
7.       ->width = width;
8.     getWidth()
9.       return->width;
10.    setHeight( height)
11.      ->height = height;
12.    getHeight()
13.      return->height;
```

```
class Rectangle : public Quadrangle
{
private:
        long width;
    long height;

public:
        void setWidth(long width)
    {
        this->width = width;
    }
    long getWidth()
    {
        return this->width;
    }
    void setHeight(long height)
    {
        this->height = height;
    }
    long getHeight()
```

```
    {
        return this->height;
    }
};
```

//正方形类:

```
1. class Square : public Quadrangle
2. private
3.     side;
4. public
5.     setSide( side)
6.       ->side = side;
7.     getSide()
8.       return side;
9.     getWidth()
10.      return getSide();
11.    getHeight()
12.      return getSide();
```

```
class Square : public Quadrangle
{
private:
        long side;

public:
        void setSide(long side)
    {
                this->side = side;
    }

    long getSide()
    {
        return side;
    }

    long getWidth()
    {
        return getSide();
    }

    long getHeight()
    {
        return getSide();
    }
};
```

问题如何得以避免?

基类Quadrangle类没有赋值方法，因此类似于 SmartTest的resize()方法不可能适用于Quadrangle类型，而只能适用于不同的具体子类Rectangle 和Square，因此里氏替换原则不可能被破坏。

- 尽量从抽象类继承，而不从具体类继承。
- 如果有两个具体类A和B有继承关系，那么一个最简单的修改方案应当是建立一个抽象类C，让类A和B成为抽象类C的子类。

- 更进一步： 如果有一个由继承关系形成的等级结构的话，那么在等级结构的树图上面所有的树叶节点都应该是具体类，而所有的树枝节点都应该是抽象类或接口。

相应设计模式：

- Strategy
- Composite
- Proxy

参考资源：

《设计模式：可复用面向对象软件的基础》，ERICH GAMMA RICHARD HELM RALPH JOHNSON JOHN VLISSIDES著作，李英军 马晓星 蔡敏 刘建中译，机械工业出版社，2005.6

《敏捷[2]软件开发：原则、模式与实践》，Robert C. Martin著，邓辉译，清华大学出版社，2003.9

《设计模式解析》，Alan Shalloway等著（徐言声译），人民邮电出版社，2006.10

顶
踩

Links

1. javascript:void(0);
2. http://lib.csdn.net/base/agile