Published in 程式愛好者

YC Following

Sep 4, 2020 · 5 min read

使人疯狂的SOLID 原则：里氏替换原则(Liskov Substitution Principle)

> *今天我们要说的是第三个原则：里氏替换原则 (LSP)。*

很多人以为里氏替换原则只是指导我们如何定义子类别。

> *LSP 的出处是 Barbara Liskov 的一篇论文：Behavioral Subtyping Using Invariants and Constraints*

> *定义：若对型态 S 的每一个物件 o1，都存在一个型态为 T 的物件 o2，使得在所有针对 T 编写的程式 P 中，用 o1 替换 o2后，程式 P 的行为功能不变，则 S 是 T 的子型态。*

嗯。。。希望你们看得懂，反正我┐(ˇ∀ˇ)┌。虽然定义看起来很复杂，但这样的定义我认为可以很准确地表达出 LSP 的思想，即：

子型态必须遵从父型态的行为进行设计。

简单说，假设我们在写一个模块 P，而模块 P 里面有用到 Car 类别的物件 car，而今天我们用 BMW 类别的物件 bmw 来替代 car，而 P 的功能都不会被影响的话，那 BMW 就是 Car 的子类别。

也就是说，只要 S 跟 T 替换后，整个 P 的行为没有差别，那 S 就是 T 的子型态。从类别上来说， S 完全可以继承 T，成为T的子类别。

那如果我们在子类 Override 父类呢，这样的动作还符合 LSP 的原则吗?

我们可以换个角度来看 LSP 原则。

按照 Design by Contract 设计方法，遵守 LSP 就是遵守以下三个条件：

1. *子型态的先决条件* (Preconditions) *不应被加强。*

先决条件是指执行一段程式前必须成立的条件。用户在使用子型态前，要确保子型态的先决条件不会比父型态的更强，但可以削弱。

如一个整数相加功能，输入的参数必须为 2 个整体并回传一个整数，且输入的数字不能小于 0 及大于 50 (先决条件)。

```
let sum = 0;

// a,b 必须 >= 0 && <= 50
function add(int a, int b)
{
    result = a + b;
```

```
    return result;
}

sum = add(1,5)
```

子型态在覆写这功能时，先决条件不能比父型态强。若父型态输入的数字要求是「不能小于 0 及大于 50」，子型态输入的数字则不能是「不能小于 0 及大于 51」，但可以是「不能小于 0 及大于 30」。

2.*子型态的后置条件* (Postconditions) *不应被削弱。*

后置条件是指执行一段程式后必须成立的条件。用户在使用子型态后，要确保子型态的后置条件不会比父型态的更弱，但可以加强其后置条件。

```
let sum = 0;

// a,b 必须 >= 0 && <= 50
function add(int a, int b)
{
    result = a + b;

    // 回传型态必须为 int
    return result;
}

// result 必须等于 sum
sum = add(1,5)

//加强条件
let sum2 = 0;
sum2 = add(1,5)
```

以相同的例子，这边的后置条件是回传的型别必须为 int。即子型态不能回传非 int 型别，如最后把 int 转成 String 再回传。

子型态加强其后置条件，如上例，除了 result 必须等于 sum 外，子型态也可以加强条件，让 result 也必须等于 sum2。

3. *父型态的不变条件* (Invariants) *必须被子型态所保留。*

不变条件指不管在何时何地都不能改变，这是构成整个型态的重要条件。同样地，子型态必须遵守父型态的不变条件，若然加以修改或不遵守，则会导致多态的重大失败。

所以，只要 Override 有遵守以上三个原则，他就是符合了 LSP 原则。

试想像一下，如果父类与子类在面对一样的参数时，子类抛出错误，而父类并没有，或者一个子类有不可预期的副作用等等，这些都是名不符实，没有真的遵从父类的行为。

以单元测试为例，如果今天写一个多态的测试，但子类的注入得不到跟父类注入时一样的结果，单元测试就不会通过，也就表示这样的子类别不符合 LSP 原则。

其实 LSP 在类别的应用上非常容易明白，但真正难以理解的是要将 LSP 放到软件架构层来看。

> *在软件架构层中，我们会期待被同一群用户所调用的接口都有着一样的行为。*

LSP 是指 T只要能被 S 替代，S 就是 T 的子型态。换句话说，我们不希望在为软件进行某程度的更新后，行为就变得不一致了。如在专案上，现在有套件 A 更新了， 我们会期待套件的更新不会影响原有程式的运作，而不是更新后一堆东西不能用了。

即用户只依赖于接口，不需要瞭解到程式的内部在发生什么事。今天不管是修 bug、是重构、是用全近的语言来写，让版本从 1.0 -> 1.1，我们都是期待一致的行为。如此，就是乎合 LSP 原则的软件架构。

总结一下，**继承请不要随意使用**。因为继承是依赖性超强的一个特性，如果稍有一项没有做对，你的子类就会做出超乎预期的行为，在整个系统已经建构起来后，修改起来会是一程地狱之旅。

如果你觉得我的文章帮助到你，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！

面向对象 Object Oriented 编程 Software Development Lsp