# OOP七大设计原则-合成/聚合复用原则

概述：
　　简单地说，尽量使用合成/聚合的方式复用，尽量少用继承；组合/聚合复用就是在一个新对象里使用一些已有的对象，使这些已有对象成为新对象的一部分，新对象通过对这些已有对象的引用达到复用已有功能的目的.

什么是聚合?
聚合用来表示“拥有”关系或者整体与部分的关系．代表部分的对象有可能会被多个代表整体的对象所共享，而且不一定会随着某个代表整体的对象被销毁或破坏而被销毁或破坏，部分的生命周期可以超越整体．例如班级和学生，当班级删除后，学生还能存在，学生可以被培训机构引用.

聚合关系类似下图：




示例代码：

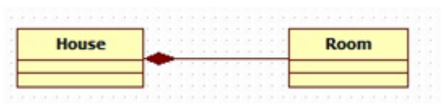
In [1]:
```
class Student:
    pass

class ClassRoom:
    def __init__(self, student):
        self.student = student

if __name__ == '__main__':
    s = Student()
    c = ClassRoom(s)
#ClassRoom被销毁了，但是学生实例仍然是存在的.
```

什么是合成?
合成用来表示一种强得多的"拥有"关系．在一个合成关系里，部分和整体的生命周期是一样的．一个合成的新对象完全拥有对其组成部分的支配权，包括它们的创建和销毁等．使用程序语言的术语来说，合成而成的新对象对组成部分的内存分配、内存释放有绝对的责任．一个合成关系中的成分对象是不能与另一个合成关系共享的，一个成分对象在同一个时间内只能属于一个合成关系．如果一个合成关系湮灭了，那么所有的成分对象要么自己销毁所有的成分对象.
例如：一个人由头、四肢和各种器官组成，人与这些具有相同的生命周期，人死了，这些器官也就挂了；房子和房间的关系，当房子没了，房间也不可能独立存在.

示例代码：




In [2]:
```
class Room:
    def createRoom(self):
        print('创建了一个房间')

class House:
    def __init__(self):
        self.room = Room()

    def createHouse(self):
        self.room.createRoom()

if __name__ == '__main__':
    h = House()
    h.createHouse()
```

创建了一个房间

复用的实现方式：
　　方式一：组合复用
　　组合可以将已有的对象纳入到新的对象中，使之成为新对象的一部分，因此新对象就可以调用已有对象的功能.

　　1.优点：

　　新对象存取成分对象的唯一方法是通过成分对象的接口。
　　这种复用是黑箱复用，因为成分对象的内部细节是新对象所看不见的。
　　这种复用支持包装。
　　这种复用所需要的依赖较少。
　　每一个新的类可以将焦点集中到一个任务上。
　　这种复用可以在运行时间动态进行，新对象可以使用合成/聚合关系将新的责任委派到合适的对象。

　　2.缺点：

　　用组合复用建造的系统会有较多的对象需要管理。

方式二：继承复用
　　组合几乎可以用到任何环境中去，但是继承只能用到一些环境中．继承复用通过扩展一个已有对象的实现来得到新的功能，基类提供了共同的属性和方法，而子类通过增加新的属性和方法来扩展超类的实现.

　　1.优点：

　　新的实现比较容易，因为基类的大部分功能都可以通过继承自动的进入子类.
　　修改或扩展继承而来的实现较为容易.

　　2.缺点：

　　继承复用破坏了包装，因为继承超类的实现细节暴露给了子类．由于超类的内部细节常常对子类是透明的，因此这种复用是透明的复用，又称"白箱”复用。 如果超类的实现发生改变，那么子类的实现也不得不发生改变．因此，当一个基类发生改变时，这种改变就会像水中投入石子引起的水波一样，将变化一圈又一圈的传导到一级又一级的子类，使设计师不得不相应地改变这些子类，以适应超类的变化.

如何选择复用方式，继承还是组合?

　　在复用时，应优先考虑使用组合而不是继承，判定的条件有以下：

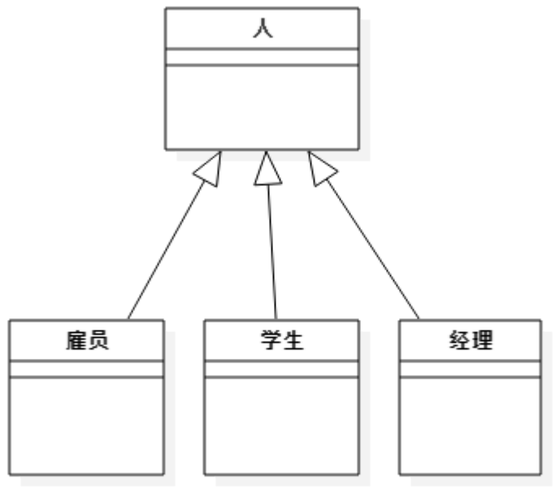
　　1.子类是超类的一个特殊种类，而不是超类的一个角色，也就是区分"Has-A"和"Is-A"，满足"Is-A"的关系才能继承，而组合却是“Has-A”的关系.

　　2.子类具有扩展超类的责任，而不是具有置换调(override)或注销掉(Nullify)超类的责任．如果一个子类需要大量的置换掉超类的行为，那么这个类就不应该是这个超类的子类.

　　Is-A顾名思义，就是"是一个"，意思是一个类是另一个类的一种；

　　Has-A是"有一个"，意思是一个类是另一个类的一部分．错误的使用继承而不是使用组合，就是把"Has-A"当做"Is-A"

请看下面的例子：




"人"被继承到"学生"、"经理"和"雇员"等子类; 而实际上, "学生"、"经理"和"雇员"分别描述一种角色, 而"人"可以同时有几种不同的角色. 比如, 一个人既然是"经理", 就必然是"雇员"; 而"人"可能同时还参加MBA课程, 从而也是一个"学生", 使用继承来实现角色, 则只能使每一个"人"具有Is-A角色, 而且继承是静态的, 这会使得一个"人"在成为"雇员"身份后, 就永远为"雇员", 不能成为"学生"和"经理", 而这显然是不合理的(在JAVA中是这样, Python是支持多重继承的)

以下示例错误的把HAS-A当成了IS-A代码如下:

In [1]:
```
from abc import ABCMeta, abstractmethod
class Person:
    __metaclass__ =  ABCMeta
    @abstractmethod
    def show(self):
        pass

class staff(Person):
    def show(self):
        print('i am a staff')

class student(Person):
    def show(self):
        print('i am a student')

class manager(Person):
    def show(self):
        print('i am a manager')

if __name__ == '__main__':
    p = staff()
    p.show()

##此时就不能再次成为其他的角色了, 这是不合理的.
```

```
i am a staff
```

人被继承到雇员, 学生, 经理子类. 而实际上雇员、学生和经理分别描述一种角色, 而人可以同时有几种不同的角色. 比如, 一个人既然是经理了就一定是雇员, 使用继承来实现角色, 则只能使用每一个人具有一种角色, 这显然是不合理的. 错误的原因就是把角色的等级结构和人的等级结构混淆起来, 把Has-A的关系误认为是Is-A的关系, 通过下面的改正就可以正确的做到这一点.




从上图可以看出, 每一个人都可以有一个以上的角色, 所以一个人可以同时是雇员又是经理. 从这个例子可以看出, 当一个类是另一个类的角色时, 不应该使用继承描述这种关系.

遵循了合成/复用原则的示例代码如下:

In [3]:
```
from abc import ABCMeta, abstractmethod
class Role:
    __metaclass__ = ABCMeta
    @abstractmethod
    def show(self):
        pass

class staff(Role):
    def show(self):
        print('i am a staff')

class student(Role):
    def show(self):
        print("i am a student")

class manager(Role):
    def show(self):
        print('i am a manager')

class Person:
    def set_role(self, role):
        self.role = role

    def get_role(self):
        return self.role.show()

if __name__ == '__main__':
    p = Person()
    p.set_role(student())
    p.get_role()
    p.set_role(manager())
    p.get_role()
```

```
i am a student
i am a manager
```

角色与经理 学生 雇员之间是IS-A关系, 因为经理 学生 雇员都是角色的一种. 人与角色之间是HAS-A关系, 因为角色与人是一种聚合关系.

In [ ]: