OOP七大设计原则-迪米特原则

概述：
简单地说，最少知道原则；即一个软件实体尽可能少地与其他实体发生相互作用．类与类之间的关系越密切，耦合度就越大，扩展和复用就越难．如果两个类之间没必要直接通信，那么这两个类就不应该发生直接交互，可以通过合理引入第三方解耦这两个类之间的相互关系．一个类对另一个类知道的越少越好,尽量降低类中成员变量及成员函数的访问权限就相当于不把相应的变量和方法暴露给其他类，这样可以尽量少地影响其他类或模块，扩展会相对容易．所以迪米特法则有一个更为直白的定义：不要和陌生人说话，只与直接的朋友通信．哪些类、对象、变量可以成为直接的朋友呢？只要两个对象之间有耦合关系就可以说这两个对象是朋友关系，耦合关系包括：继承、实现、依赖、关联、聚合、组合等．所以当前对象的直接朋友包括：

1）当前对象本身

2）当前对象创建的对象

3）当前对象的成员对象

4）当前对象的实例变量所引用的对象

5）以参数形式传入到当前对象方法中的对象

满足上面任一条件的对象都是当前对象的”直接朋友“否则就是陌生人．

迪米特法则强调了下面两点：

1．从依赖者的角度来看，只依赖应该依赖的对象．
2．从被依赖者的角度：只暴露应该暴露的方法或属性，即编写相关的类时确定方法和属性的权限(public or private)

我们先举例演示第一点，在我们生活中会有这样的情况，比如张三去找李四帮忙做一件事，对于李四来说这件事也很棘手，李四也做不了，但是李四有一个好哥们王五却能完成这件事，所以李四就把这件事交给王五去办(在本例中，张三和王五是不认识的)．现在我们暂定张三为A,李四为B，王五为C．

不符合迪米特原则的示例代码如下：

In [1]:
```
#coding:utf8

class A:
    def __init__(self, name):
        self.name = name

    def get_b(self, name):
        return B(name)

    def work(self):
        b = self.get_b("李四")
        c = b.get_c("王五")
        c.work()

class B:
    def __init__(self, name):
        self.name = name
    def get_c(self, name):
        return C(name)

class C:
    def __init__(self, name):
        self.name = name
    def work(self):
        print('%s把这个事情做好了!' %self.name)


if __name__ == '__main__':
    a = A("张三")
    a.work()
```

```
王五把这个事情做好了!
```

上面的设计输出答案是正确的，王五确实把事情办妥了．但是我们仔细看业务逻辑确发现这样做事不对的，张三和王五互相不认识，那为什么代表张三的A类中会有代表王五的C类呢，这样明显是违背了迪米特法则的．现在我们对上面的代码进行重构，根据迪米特法则的第二点：从依赖者的角度来看，只依赖应该依赖的对象，在本例中，张三只认识李四，那么只能依赖李四．重构后代码如下：

符合迪米特原则的示例代码，并且符合只依赖应该依赖的对象这一特点．

In [2]:
```
class A:
    def __init__(self, name):
        self.name = name

    def get_b(self, name):
        return B(name)

    def work(self):
        b = self.get_b("李四")
        b.work()

class B:
    def __init__(self, name):
        self.name = name
    def get_c(self, name):
        return C(name)

    def work(self):
        c = self.get_c("王五")
        c.work()

class C:
    def __init__(self, name):
        self.name = name
    def work(self):
        print('%s把这个事情做好了!' %self.name)


if __name__ == '__main__':
    a = A("张三")
    a.work()
```

```
王五把这个事情做好了!
```

继续举例演示第二点：当我们按下计算机的按钮的时候，计算机会指行一系列操作：保存当前任务 关闭相关服务 接着关闭显示屏 最后关闭电源 这些操作完成则计算机才算关闭．

未符合迪米特原则第2个特点的(只暴露应该暴露的方法和属性)的示例代码如下：

In [ ]:
```
class Computer:
    def save_task(self):
        print("任务保存了")

    def close_service(self):
        print("服务关闭了")

    def close_screen(self):
        print("屏幕关闭了")

    def close_power(self):
        print("电源关闭了")

    def close(self):
        self.save_task()
        self.close_service()
        self.close_screen()
        self.close_power()

class Person:
    def __init__(self, c=Computer()):
        self.c = c

    def clickCloseButton(self):
        self.c.save_task()
        self.c.close_power()
        self.c.close()

        #或者是
        self.c.close()
        self.c.close_screen()

if __name__ == '__main__':
    p = Person()
    p.clickCloseButton()
```

对于人来说，我期待的结果只是按下关闭电钮然后计算机就关掉，而不是需要我去小心的去保存当前正在执行的任务等等；在上面的代码中，c是一个完全暴露的对象，它的方法是完全公开的，对于Person来说，手里面就如同多出了好几把钥匙，至于具体用哪一把他不知道，所以只能一把一把的去试一遍，显然这样的设计是不对的.

根据迪米特法则的第二点：从被依赖者的角度，只暴露应该暴露的方法. 在本例中，应该暴露的方法就是close()，关于计算机的其他操作不是依赖者应该关注的问题，应该对依赖者关闭. 所以代码应该重构为如下样子：

符合迪米特原则第2个特点的示例代码如下：

In [4]:
```
#coding:utf8

class Computer:

    def __save_task(self):
        print("任务保存了")

    def __close_service(self):
        print("服务关闭了")

    def __close_screen(self):
        print("屏幕关闭了")

    def __close_power(self):
        print("电源关闭了")

    def close(self):
        self.__save_task()
        self.__close_service()
        self.__close_screen()
        self.__close_power()

class Person:
    def __init__(self, c=Computer()):
        self.c = c

    def clickCloseButton(self):
        self.c.close()

if __name__ == '__main__':
    p = Person()
    p.clickCloseButton()
```

```
任务保存了
服务关闭了
屏幕关闭了
电源关闭了
```

注意事项：

1)在不违反需求的条件下尽量降低每个类中成员变量和成员函数的访问权限

2)合理引入第三方降低对象之间直接交互的耦合度

3)尽量使用不可变类

4)过度应用迪米特法则会产生大量的代理类或中介类，导致系统过于复杂. 所以需要权衡使用，既满足高内聚低耦合，又能够做到结构清晰.