2.创建型模式-工厂模式

概念1：封闭开放原则
关于开放封闭原则，其核心的思想是：软件实体应该是可扩展，而不可修改的．也就是说对扩展是开放的，而对修改是封闭的．因此开放封闭原则主要体现在两个方面：
1）对扩展开放，意味着有新的需求或变化时，可以对现有代码进行扩展，以适应新的情况．
2）对修改封闭，意味着类一旦设计完成，就可以独立完成其工作，而不要对类进行任何修改．

"需求总是变化"、"世界上没有一个软件是不变的"，这些言论是对软件需求最经典的表白．从中透射出一个关键的意思就是，对于软件设计者来说，必须在不需要对原有的系统进行修改的情况下，实现灵活的系统扩展，而如何能做到这一点呢?

In [1]:
```python
#示例代码1：未使用工厂模式

from abc import ABCMeta, abstractmethod

class LeiFeng:
    __metaclass__ = ABCMeta
    @abstractmethod
    def do_something(self):
        pass
    @abstractmethod
    def do_work(self):
        pass

class Stuent(LeiFeng):
    def do_something(self):
        return "学生正在做好事"
    def do_work(self):
        return "学生正在干累活"

class Volunteer(LeiFeng):
    def do_something(self):
        return "志愿者正在做好事"
    def do_work(self):
        return "志愿者正在干累活"

if __name__ == '__main__':

    s1 = Stuent()
    print(s1.do_work(), s1.do_something())

    v1 = Volunteer()
    print(v1.do_work(), v1.do_something())
```

```
学生正在干累活 学生正在做好事
志愿者正在干累活 志愿者正在做好事
```

未使用工厂方法时，调用者可以通过实例化不同的类(学生，志愿者)来生成示例对象，进而进行操作．但如果类有很多，调用者就需要记住很多类名，使用不便，所以通常我们使用一种工厂来生成对象．

简单工厂模式：

In [5]:
```python
from abc import ABCMeta, abstractmethod

class LeiFeng:
    __metaclass__ = ABCMeta
    @abstractmethod
    def do_something(self):
        pass
    @abstractmethod
    def do_work(self):
        pass

class Stuent(LeiFeng):
    def do_something(self):
        return "学生正在做好事"
    def do_work(self):
        return "学生正在干累活"

class Volunteer(LeiFeng):
    def do_something(self):
        return "志愿者正在做好事"
    def do_work(self):
        return "志愿者正在干累活"


class LeiFeng_factory:

    def create_new_leifeng(self, leifeng):
        maps = {
            "学生": Stuent(),
            "志愿者": Volunteer()
        }

        if leifeng in maps:
            return maps[leifeng]
        else:
            raise ValueError("请输入学生或志愿者")


if __name__ == '__main__':
    s1 = LeiFeng_factory().create_new_leifeng("学生")
    s2 = LeiFeng_factory().create_new_leifeng("学生")
    s3 = LeiFeng_factory().create_new_leifeng("学生")
    print(s1.do_something(), 's1')
    print(s2.do_work(), 's2')
    print(s3.do_something(), 's3')
    print(s3.do_work(), 's3')
```

```
学生正在做好事 s1
学生正在干累活 s2
学生正在做好事 s3
学生正在干累活 s3
```

在简单工厂中，如果需要新增类，例如加一个中学生类（MiddleStudent，就需要新写一个类，同时要修改工厂类的maps，加入'中学生':MiddleStudent()，这样就违背了封闭开放原则中的一个类写好后，尽量不要修改里面的内容这个原则．

工厂模式：

In [6]:
```
#coding:utf8

from abc import ABCMeta, abstractmethod


class LeiFeng:
    __metaclass__ = ABCMeta
    @abstractmethod
    def do_something(self):
        pass
    @abstractmethod
    def do_work(self):
        pass

class Stuent(LeiFeng):
    def do_something(self):
        return "学生正在做好事"
    def do_work(self):
        return "学生正在干累活"

class Volunteer(LeiFeng):
    def do_something(self):
        return "志愿者正在做好事"
    def do_work(self):
        return "志愿者正在干累活"


class Factory:
    @abstractmethod
    def create_new_leifeng(self):
        pass

class student_factory(Factory):
    def create_new_leifeng(self):
        return Stuent()

class volunteer_factory(Factory):
    def create_new_leifeng(self):
        return Volunteer()

if __name__ == '__main__':
    myfactory = student_factory

    s1 = myfactory().create_new_leifeng()
    s2 = myfactory().create_new_leifeng()
    s3 = myfactory().create_new_leifeng()

    print(s1.do_something(), s1.do_work(), 's1')
    print(s2.do_something(), s2.do_work(), 's2')
    print(s3.do_something(), s3.do_work(), 's3')
```

```
学生正在做好事 学生正在干累活 s1
学生正在做好事 学生正在干累活 s2
学生正在做好事 学生正在干累活 s3
```

优势1：
    在工厂方法中，需要增加一个中学生类和一个中学生工厂类（MiddleStudentFactory），虽然比较繁琐，但是符合封闭开放原则.
    在简单工厂方法中，将判断输入的类型，返回相应的类这个过程从工厂类中移到了客户端中实现，所以当需要新增类时，也是要修改代码的，不过是改客户端的代码而不是工厂类的代码.

优势2：
    对代码的修改会更加方便，例如在调用者代码中，需要将Student的实现改为Volunteer，如果在简单工厂中，
    就需要把leifeng1 =LeiFengFactory().create_lei_feng('大学生')中的大学生改成社区志愿者，这里就需要改三处地方.

    但是在工厂方法中，只需要把
    myfactory = StudentFactory
    改成
    myfactory = VolunteerFactory

In [1]:
抽象工厂模式：

    前面我们讨论了"简单工厂模式"和"工厂方法模式"，这次我们来学习设计模式中最后的一种工厂模式—抽象工厂模式.

    抽象工厂模式其实是工厂方法模式的一种扩展，应用抽象工厂模式可以创建一系列的产品(产品族)，而不是像工厂方法模式中的只能创建一种产品. 先我们来看一下抽象工厂模式的标准定义：为创建一组相关或相互依赖的对象提供一个接口，而且无需指定他们的具体类.

    例如富士康公司给两个品牌作代工产品：苹果和三星；众所周知，这两个品牌都有手机和平板产品，由于生产工艺的不同，富士康开设了两条生产线，一条线只生产手机，另一条线只生产平板，总负责人是车间主任老王. 一个卖苹果设备的采购商找到老王，说先给我来1台苹果的iPad,老王转身到生产平板的生产线上的操作台，往电脑里输入“苹果牌”三个字，很快1台iPad生产出来了. 采购商又说，再给我来1台苹果的iPhone吧，老王又转身到手机的生产线,在电脑里输入“苹果牌”，很快一台iPhone又造好了.

    这里有两种抽象的产品(苹果产品和三星产品)，而每种抽象的产品都有两种产品角色(手机和平板电脑)，这样就要建立两种工厂(手机工厂和平板工厂)分别负责不同产品角色的实例化. 老王就是工厂的总接口，他负责帮你找到正确的生产工厂，并且拿到你想要的那一种类型的产品.

In [2]:
```
#coding:utf8
from abc import ABCMeta,abstractmethod


class Apple:
    __metaclass__ = ABCMeta
    @abstractmethod
    def appleStyle(self):
        pass

class Sumsung:
    __metaclass__ = ABCMeta
    @abstractmethod
    def sumsungStyle(self):
        pass

class iphone(Apple):
    def appleStyle(self):
        print("this is iphonex")

class ipad(Apple):
    def appleStyle(self):
        print("this is ipad pro")

class note3(Sumsung):
    def sumsungStyle(self):
        print("this is sumsung note3")

class galaxy(Sumsung):
    def sumsungStyle(self):
        print("this is sumsung galaxy!!")

class Factory:
    __metaclass__ = ABCMeta
    @abstractmethod
    def createAppleProduct(self):
        pass
    @abstractmethod
    def createSumsungProduct(self):
        pass

class Factory_phone(Factory):
    def createAppleProduct(self):
        return iphone()
    def createSumsungProduct(self):
        return galaxy()

class Factiory_pad(Factory):
    def createAppleProduct(self):
        return ipad()
    def createSumsungProduct(self):
        return note3()

if __name__ == '__main__':
    x = Factory_phone()
    x.createAppleProduct().appleStyle()
    x.createSumsungProduct().sumsungStyle()

    y = Factiory_pad()
    y.createAppleProduct().appleStyle()
    y.createSumsungProduct().sumsungStyle()
```

```
this is iphonex
this is sumsung galaxy!!
this is ipad pro
this is sumsung note3
```

抽象工厂模式最大的缺点就是对产品族的扩展非常困难，如果要添加一个新的品牌联想的话，看看我们的改动会有多大吧，首先要在Factory接口中声明新方法：

```
class Factory:
    __metaclass__ = ABCMeta
    def createAppleProduct(self):
        pass
    def createSumsungProduct(self):
        pass
    def createLenovoProduct(self):
        pass
```

然后在所有现有的工厂实现类中分别实现这个新的createLenovoProduct()方法，如果工厂类有很多，改动的地方也会很多的.违反了开闭原则，并且作为契约的接口修改了，其他所有和接口有关的代码可能都要改.

反过来想，如果对产品角色扩展，比如我要添加一个新的角色"电脑"，改动的地方有哪些呢？只需要新建的各品牌电脑产品类和一个电脑工厂而已，都是扩展而不是修改，这样就又符合了开闭原则. 所以说，抽象工厂模式对于产品角色的扩展是很容易的.