OOP七大设计原则-依赖倒置原则

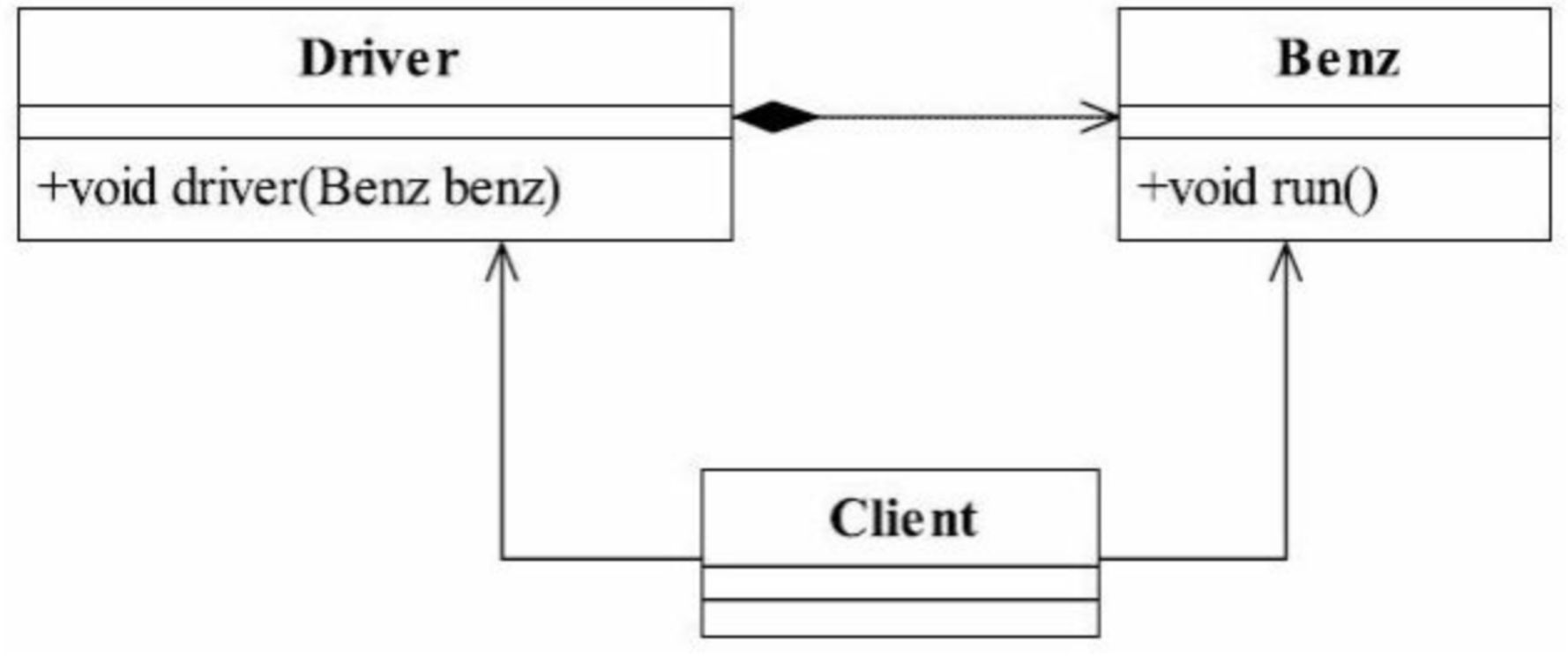
概述：
依赖倒置原则的包含如下的三层含义：
1.高层模块(由原子逻辑组装而成)不应该依赖低层模块(不可分割的原子逻辑)，两者都应该依赖其抽象.
2.抽象(抽象类)不应该依赖细节(实现类).
3.细节应该依赖抽象.

每一个逻辑的实现都是由原子逻辑组成的，不可分割的原子逻辑就是低层模块(一般是接口，抽象类)．原子逻辑的组装就是高层模块，在Python语言中，抽象就是指接口和或抽象类，两者都不能被直接实例化．细节就是实现类，实现接口或继承抽象类而产生的类就是细节，可以被直接实例化．下面是依赖倒置原则在Python语言中的表现：

1.模块间的依赖通过抽象发生，实现类之间不发生直接的依赖关系，其依赖关系是通过接口或抽象类产生的.
2.接口或抽象类不依赖于实现类
3.实现类依赖于接口或抽象类

采用依赖倒置原则可以减少类间的耦合性，提高系统的稳定性，降低并行开发引起的风险，提高代码的可读性和可维护性.

假设如下场景：



从上面的类图中可以看出，司机类和奔驰车类都属于细节，并没有实现或继承抽象．它们是对象级别的耦合，通过类图可以看出司机有一个drive()方法，用来开车，奔驰车有一个run()方法，用来表示车辆运行，并且奔驰车类依赖于司机类，用户模块表示高层模块，负责调用司机类和奔驰车类.

未采用依赖倒置原则的示例代码如下：

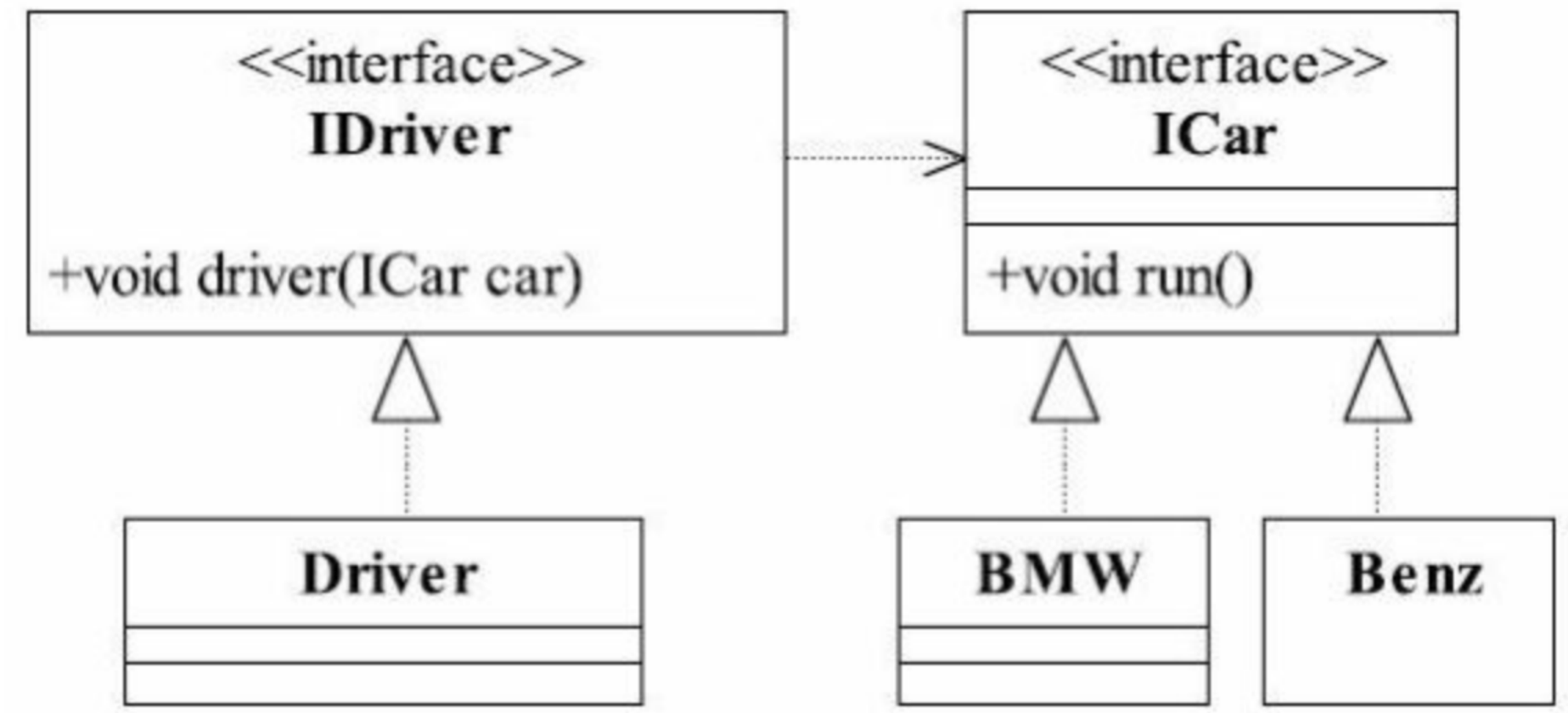
In [3]:

```
class Driver:
    """
    最不灵活的写法，driver如果新买了一个bmw的话，他无法使用
    """
    def __init__(self, name):
        self.name = name
    def drive(self, benz):
        print('%s is driving a benz' %self.name)
        benz.run()


class Driver:
    """
    将代码改成这样，如果benz或bmw的运行方法不是run，那么就会报错．此时需要将Car类的run方法进行抽象，来对子类进行约束.
    """
    def __init__(self, name):
        self.name = name
    def drive(self, car):
        print('%s is driving a %s' %(self.name, car.name))
        car.run()

class Benz:
    def __init__(self, name='benz'):
        self.name = name

    def run(self):
        print('%s is running..' %self.name)

class BMW:
    def __init__(self, name='bmw'):
        self.name = name
    def run(self):
        print('%s is running...'%self.name)


if __name__  == '__main__':
    d = Driver("张三")
    benz = Benz()
    d.drive(benz)
    print('#' * 20)
    bmw = BMW()
    d.drive(bmw)
```

```
张三 is driving a benz
benz is running..
####################
张三 is driving a bmw
bmw is running...
```

上面的设计没有使用依赖倒置原则，我们已经郁闷的发现，模块与模块之间耦合度太高，生产力太低，只要需求一变就需要大面积重构，说明这样的设计是不合理．现在我们引入依赖倒置原则，重新设计的类图如下：

In [ ]:



示例代码：

In [3]:
```
from abc import ABCMeta, abstractmethod

class Driver:
    __metaclass__ = ABCMeta
    def __init__(self, name):
        self.name = name

    @abstractmethod
    def drive(self, car):
        pass

class CarDrive(Driver):
    def drive(self, car):
        print("%s is driving a car.." %self.name)
        car.run()

class Car:
    __metaclass__ =  ABCMeta
    def __init__(self, name):
        self.name = name

    @abstractmethod
    def run(self):
        pass

class BenzCar(Car):
    def run(self):
        print("%s is running.." %self.name)

class BmwCar(Car):
    def run(self):
        print("%s is running.." %self.name)


if __name__ == '__main__':
    zs = CarDrive("张三")
    benz = BenzCar("奔驰")
    bmw = BmwCar("宝马")
    zs.drive(benz)
    zs.drive(bmw)
    print("##" * 10)
    ls = CarDrive("李四")
    ls.drive(benz)
    ls.drive(bmw)
```

```
张三 is driving a car..
奔驰 is running..
张三 is driving a car..
宝马 is running..
####################
李四 is driving a car..
奔驰 is running..
李四 is driving a car..
宝马 is running..
```

在新增低层模块时，只修改了高层模块(业务场景类)，对其他低层模块(Driver类)不需要做任何修改，可以把"变更"的风险降低到最低．抽象是对实现的约束，是对依赖者的一种契约，不仅仅约束自己，还同时约束自己与外部的关系，其目的就是保证所有的细节不脱离契约的范畴，确保约束双方按照规定好的契约(抽象)共同发展，只要抽象这条线还在，细节就脱离不了这个圈圈．就好比一场篮球比赛，已经定好了规则，大家如果按照规则来打球，那么会很愉快．但是假如大家脱离了规则，那么也许比赛就无法顺利进行．