下面给出四种设计模式。

1.单例模式（Singleton Pattern）

2.工厂模式（Factory Pattern）

3.观察者模式（Observer Pattern）

4.装饰器模式（Decorator Pattern）

# 1. 单例模式（Singleton Pattern）

单例模式确保一个类只有一个实例，并提供一个全局访问点。

class Singleton:

    \_\_instance = None

    def \_\_new\_\_(cls):

        if cls.\_\_instance is None:

            cls.\_\_instance = super(Singleton, cls).\_\_new\_\_(cls)

        return cls.\_\_instance

# 测试单例模式

if \_\_name\_\_ == "\_\_main\_\_":

    s1 = Singleton()

    s2 = Singleton()

    print(s1 is s2)  # 输出: True

特点：

* 保证一个类只有一个实例。
* 提供一个全局访问点。
* 通常用来管理共享资源。

# 工厂模式（Factory Pattern）

工厂模式定义一个用于创建对象的接口，让子类决定实例化哪一个类。

class Dog:

    def speak(self):

        return "Woof!"

class Cat:

    def speak(self):

        return "Meow!"

class AnimalFactory:

    @staticmethod

    def get\_animal(animal\_type):

        if animal\_type == "dog":

            return Dog()

        elif animal\_type == "cat":

            return Cat()

        else:

            return None

# 测试工厂模式

if \_\_name\_\_ == "\_\_main\_\_":

    factory = AnimalFactory()

    dog = factory.get\_animal("dog")

    cat = factory.get\_animal("cat")

    print(dog.speak())  # 输出：Woof!

    print(cat.speak())  # 输出：Meow!

特点：

* 提高了对象创建的灵活性。
* 客户端不需要知道它所创建的具体类。
* 新的类可以很容易地添加到系统中。

# 观察者模式（Observer Pattern）

观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。当主题对象发生变化时，它的所有依赖者都会收到通知并自动更新。

class Subject:

    def \_\_init\_\_(self):

        self.\_observers = []

    def attach(self, observer):

        if observer not in self.\_observers:

            self.\_observers.append(observer)

    def detach(self, observer):

        try:

            self.\_observers.remove(observer)

        except ValueError:

            pass

    def notify(self, modifier=None):

        for observer in self.\_observers:

            if modifier != observer:

                observer.update(self)

class Data(Subject):

    def \_\_init\_\_(self, name=''):

        super().\_\_init\_\_()

        self.name = name

        self.\_data = 0

    @property

    def data(self):

        return self.\_data

    @data.setter

    def data(self, value):

        self.\_data = value

        self.notify()

class DataViewer:

    def update(self, subject):

        print(f'DataViewer: {subject.name} has data {subject.data}')

# 测试观察者模式

if \_\_name\_\_ == "\_\_main\_\_":

    data1 = Data('Data 1')

    data2 = Data('Data 2')

    viewer1 = DataViewer()

    viewer2 = DataViewer()

    data1.attach(viewer1)

    data1.attach(viewer2)

data2.attach(viewer1)

    data1.data = 10  # DataViewer更新：Data 1 has data 10

    data2.data = 20  # DataViewer更新：Data 2 has data 20

特点：

* 使得观察者能够在不改变目标对象的情况下动态地添加。
* 提高了对象之间的解耦。
* 灵活性高，但是使用不当可能导致代码复杂度增加。

# 装饰器模式（Decorator Pattern）

装饰器模式允许向一个现有的对象添加新的功能，同时又不改变其结构。

class Coffee:

    def cost(self):

        return 5

class MilkDecorator:

    def \_\_init\_\_(self, coffee):

        self.\_coffee = coffee

    def cost(self):

        return self.\_coffee.cost() + 2

class SugarDecorator:

    def \_\_init\_\_(self, coffee):

        self.\_coffee = coffee

    def cost(self):

        return self.\_coffee.cost() + 1

# 测试装饰器模式

if \_\_name\_\_ == "\_\_main\_\_":

    coffee = Coffee()

    milk\_coffee = MilkDecorator(coffee)

    sugar\_milk\_coffee = SugarDecorator(milk\_coffee)

    print(sugar\_milk\_coffee.cost())  # 输出: 8 (5 + 2 + 1)

特点：

* 动态地给一个对象添加一些额外的职责，就是说多次装饰和不装饰的实例无差别。
* 相比于生成子类更为灵活。
* 易于实现，易于扩展。