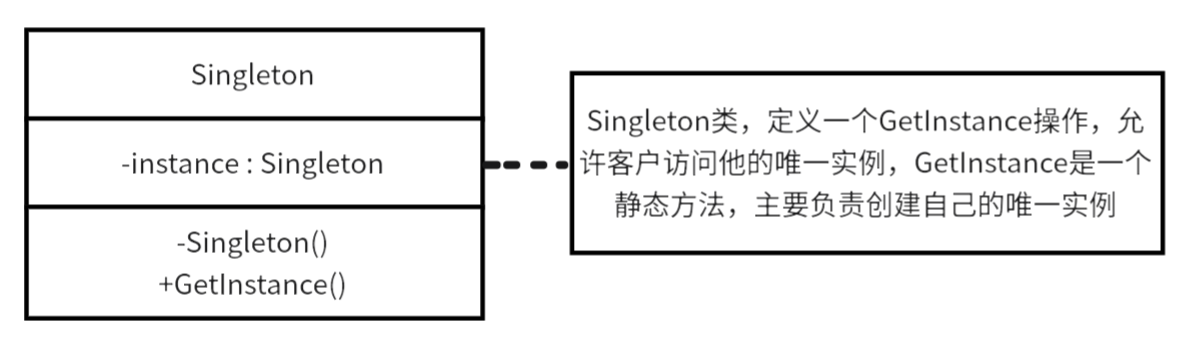
**设计模式列举**

1. **单例模式**

单例模式是一种对象创建型模式，使用单例模式，可以保证为一个类只生成唯一的实例对象。也就是说，在整个程序空间中，该类只存在一个实例对象。

结构图：



示例代码：

class Singleton:

    \_instance = None

    def \_\_new\_\_(cls):

        if cls.\_instance is None:

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

        return cls.\_instance

# 使用单例模式

singleton1 = Singleton()

singleton2 = Singleton()

print(singleton1 is singleton2)  # 输出: True

特点：

唯一实例：确保一个类只有一个实例，并提供全局访问点。

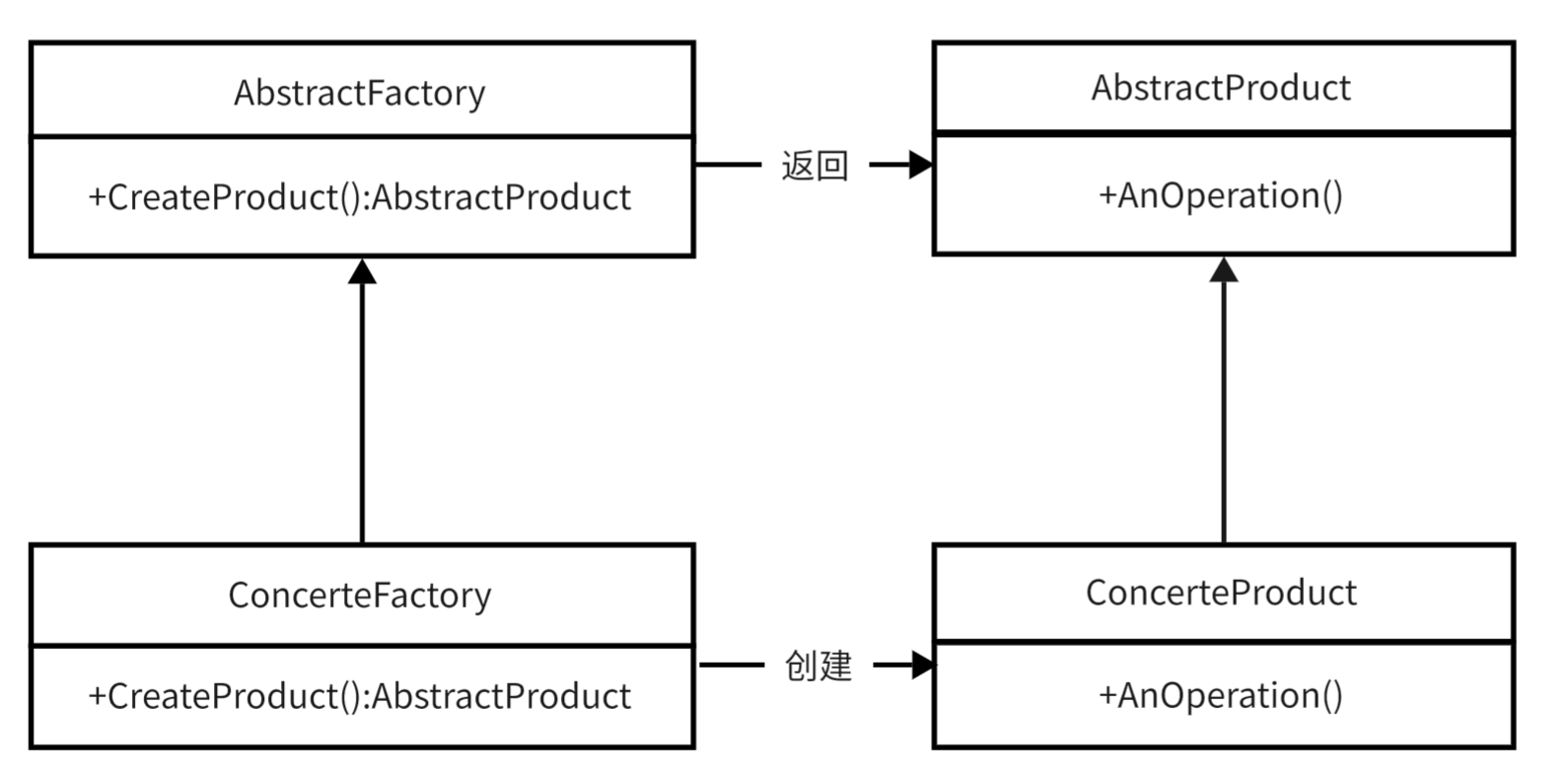
延迟初始化：实例只有在第一次使用时才被创建。

线程安全：在多线程环境下需要确保实例的唯一性。

1. **工厂模式**

工厂方法模式定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个。工厂方法使一个类的实例化延迟到其子类。

模式图：



示例代码：

class Animal:

    def speak(self):

        pass

class Dog(Animal):

    def speak(self):

        return "Woof!"

class Cat(Animal):

    def speak(self):

        return "Meow!"

class AnimalFactory:

    def create\_animal(self):

        pass

class DogFactory(AnimalFactory):

    def create\_animal(self):

        return Dog()

class CatFactory(AnimalFactory):

    def create\_animal(self):

        return Cat()

# 使用工厂方法创建对象

dog\_factory = DogFactory()

dog = dog\_factory.create\_animal()

print(dog.speak())  # 输出: Woof!

cat\_factory = CatFactory()

cat = cat\_factory.create\_animal()

print(cat.speak())  # 输出: Meow!

特点：

多个工厂子类：每个子类工厂负责创建一种产品对象。

分散化：创建逻辑分散在不同的子类中，符合开闭原则，易于扩展。

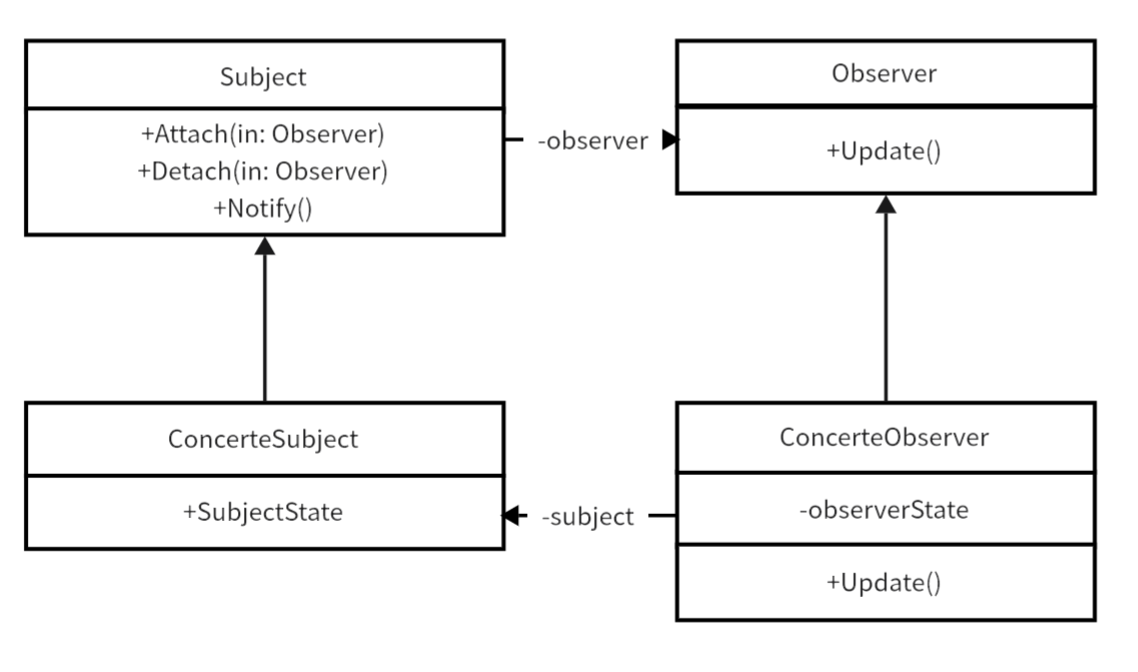
增加复杂性：比简单工厂模式更复杂，需要更多的类。

1. **观察者模式**

观察者模式是行为模式之一，它的作用是当一个对象的状态发生变化时，能够自动通知其 他关联对象，自动刷新对象状态。

观察者模式提供给关联对象一种同步通信的手段，使某个对象与依赖它的其他对象之间保 持状态同步。

模式图：



示例代码：

class Subject:

    def \_\_init\_\_(self):

        self.\_observers = []

    def attach(self, observer):

        self.\_observers.append(observer)

    def detach(self, observer):

        self.\_observers.remove(observer)

    def notify(self):

        for observer in self.\_observers:

            observer.update()

class Observer:

    def update(self):

        pass

class ConcreteObserver(Observer):

    def update(self):

        print("Observer updated.")

# 使用

subject = Subject()

observer = ConcreteObserver()

subject.attach(observer)

subject.notify()  # 输出: Observer updated.

特点：

发布-订阅机制：定义对象间的一对多依赖关系，当一个对象状态改变时，所有依赖它的对象都会收到通知并自动更新。

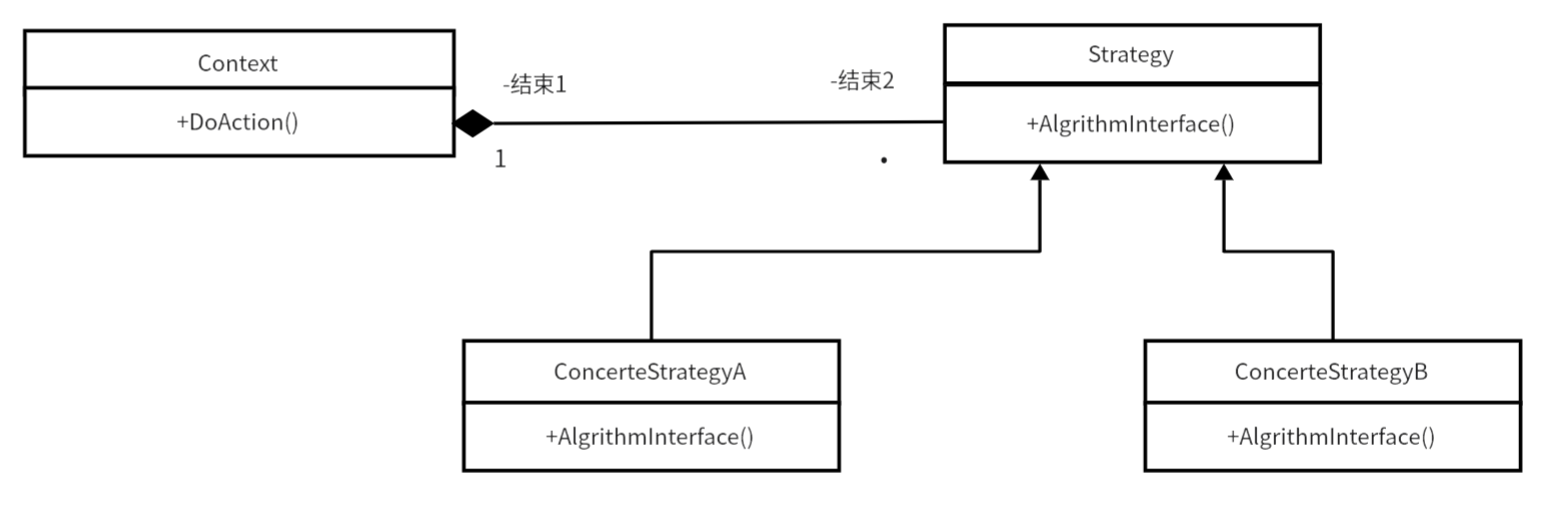
松散耦合：观察者和被观察者之间的耦合度低，增加新的观察者容易。

动态关联：在运行时可以动态地增加或删除观察者

1. **策略模式**

策略模式对一些列算法加以封装，为所有算法定义一个抽象的算法接口。并通过继承该抽象算法接口对所有的算法加以封装和实现，具体 的算法选择交由客户端决定。策略模式主要用来平滑地处理算法的切换 。

模式图：



示例代码：

class Strategy:

    def execute(self, a, b):

        pass

class AddStrategy(Strategy):

    def execute(self, a, b):

        return a + b

class SubtractStrategy(Strategy):

    def execute(self, a, b):

        return a - b

class Context:

    def \_\_init\_\_(self, strategy):

        self.\_strategy = strategy

    def set\_strategy(self, strategy):

        self.\_strategy = strategy

    def execute\_strategy(self, a, b):

        return self.\_strategy.execute(a, b)

# 使用策略模式

context = Context(AddStrategy())

print(context.execute\_strategy(5, 3))  # 输出: 8

context.set\_strategy(SubtractStrategy())

print(context.execute\_strategy(5, 3))  # 输出: 2

特点：

行为封装：将具体算法封装在独立的类中，使得算法可以相互替换。

易于扩展：增加新的策略类时，不需要修改现有的策略接口和上下文类。

策略独立：使得算法可以独立于使用它的客户端变化。