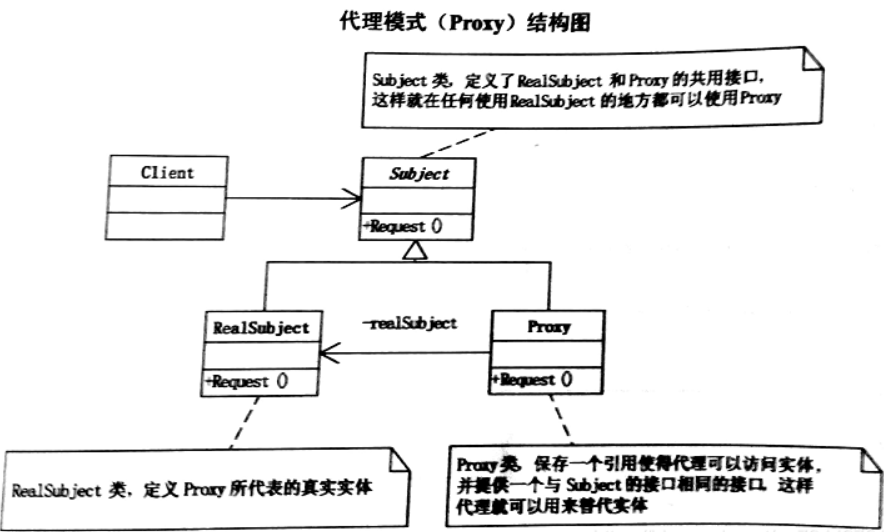
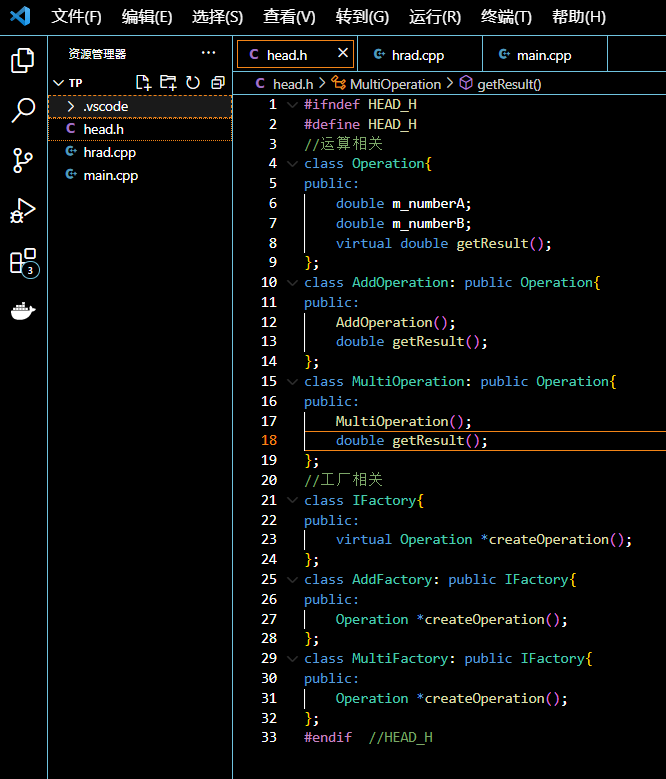
2设计模式应用

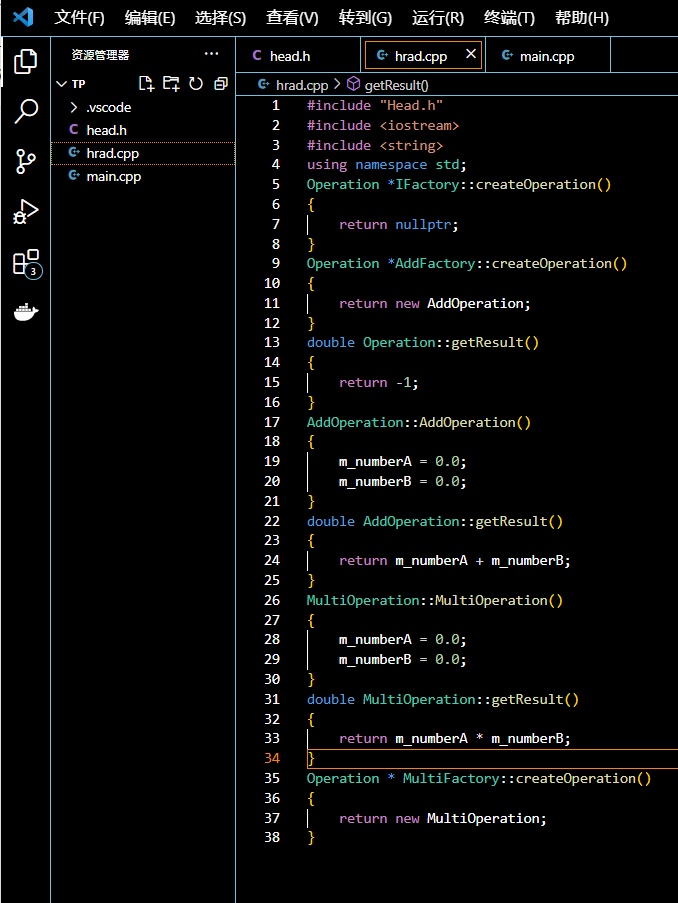
1. **工厂方式模式**

工厂方法模式（Factory Method）：定义一个用于创建对象的接口，让字类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类；

工厂方法模式实现时，客户端需要决定实例化哪一个工厂来实现运算类，选择判断的问题还是存在的，也就是说，工厂方法把简单工厂内部逻辑判断移到了客户端代码来进行。









优点

（1）使用工厂方法来创建用户所需的产品，用户只要关心所需产品对应的工厂，无需关心创建的细节。

（2）所有的具体工厂类都具有同一抽象父类工厂能自行确定创建何种产品对象，而创建的细节完全封装在具体工厂中。

（3）如果新加入产品，就无需修改抽象工厂类和抽象产品类，也无需修改其他具体工厂和产品，而只用添加一个具体工厂和具体产品即可。增加系统的可扩展性，符合开闭原则

缺点

添加新产品会增加具体工厂类和具体产品类，这无疑会增加系统的复杂度，带来更多的开销。

1. **策略模式**

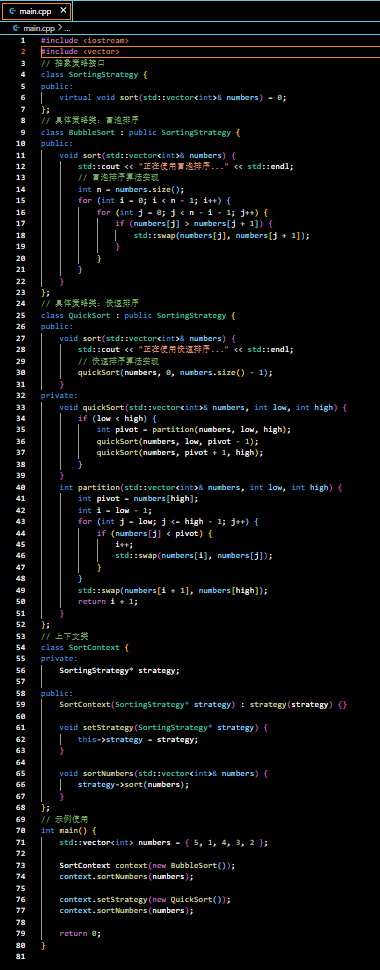
策略模式（Strategy Pattern）是一种行为型设计模式，它允许在运行时根据不同的情况选择算法的行为。在策略模式中，我们定义一组算法，并将每个算法封装在独立的类中，这些算法具有相同的接口，使它们可以相互替换。这样可以使算法的变化独立于使用算法的客户端。

**一般结构：**

1抽象策略（Strategy）：定义了策略类的接口，所有具体策略类都要实现这个接口。

2具体策略（Concrete Strategy）：实现了抽象策略定义的接口，提供了具体的算法实现。

3上下文（Context）：维护一个对抽象策略的引用，用于执行算法。上下文可以通过设置不同的策略对象来改变其行为。



优点：

1提供了灵活性：可以动态地选择算法，而无需更改客户端代码。

2使算法独立于客户端：策略类可以独立于其他类进行修改和扩展。

3可以减少条件语句：将不同的算法封装在不同的策略类中，避免了使用复杂的条件语句。

1. **装饰者模式**

装饰者模式（Decorator Pattern）是一种结构型设计模式，它允许在不改变现有对象结构的情况下，动态地将新功能添加到对象中。

装饰者模式通过创建一个包装器（装饰者）来包裹原始对象，并且使用相同的接口，使得装饰者和原始对象对于客户端来说是可互换的。这样可以在运行时透明地向对象添加新的行为或修改现有行为，而无需改变其结构。

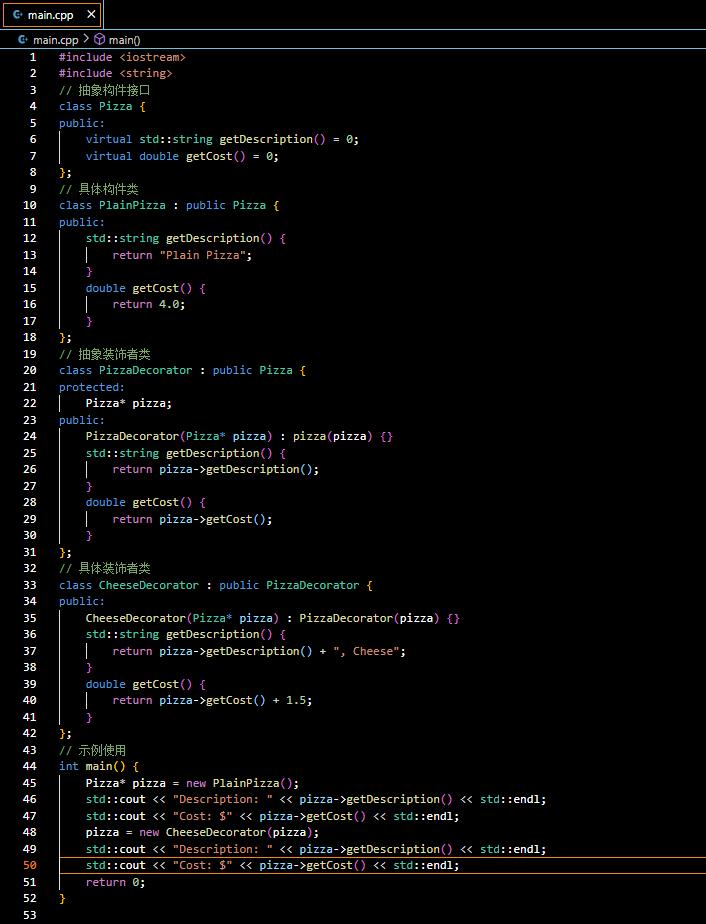
**一般结构**

1抽象构件（Component）：定义了具体构件和装饰者的公共接口。

具体构件（Concrete Component）：实现了抽象构件定义的接口，是被装饰的原始对象。

2抽象装饰者（Decorator）：继承自抽象构件，并持有一个指向抽象构件的引用。在具体装饰者中可以调用原始对象的方法，并可以添加新的行为。

3具体装饰者（Concrete Decorator）：扩展了抽象装饰者定义的接口，并实现了新的行为。



优点：

1动态添加功能：通过装饰者可以在运行时动态地添加新的功能，而不需要修改原始对象的代码。

2遵循开闭原则：无需修改现有代码，可以通过新增装饰者类来扩展功能。

3组合灵活性：可以通过不同方式组合装饰者，实现多种不同的功能组合。

下面是一个简单的示例代码，展示了装饰者模式的应用：