实验十四 设计模块（三 ）

实验目的：

学习设计模式，能在项目设计中运用设计模式进行面向对象设计

实验内容：

1. 阅读下面设计模式资料（或查阅其它相关资料），结合项目的进程和开发历程，分析项目采用了那些设计模式（李雨桐）

Design Patterns-Elements of Reusable Object-Oriented Software.pdf

The GoF Design Patterns Reference.pdf

[Design Patterns - Wikipedia](https://en.wikipedia.org/wiki/Design_Patterns)

MVC（Model-View-Controller）设计模式：MVC设计模式是一种将应用程序的数据模型、用户界面和控制器逻辑分离的设计方法。在苍穹外卖项目中，MVC模式被用于实现用户界面与后端逻辑的解耦，使得用户界面的变化不会影响到业务逻辑，同时业务逻辑的变化也不会影响到用户界面的显示。

工厂模式（Factory Pattern）：工厂模式用于封装对象的创建过程，使得代码更加灵活和可扩展。在苍穹外卖项目中，工厂模式被用于创建订单、用户、商品等对象，以便在需要时能够方便地扩展新的对象类型。

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。在苍穹外卖项目中，单例模式被用于管理数据库连接、缓存等全局资源，确保在系统中只有一个实例被创建和使用。

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。当主题对象状态发生改变时，所有依赖它的对象都会得到通知并被自动更新。在苍穹外卖项目中，观察者模式可能被用于实现订单状态变更通知、用户登录状态变化等功能。

策略模式（Strategy Pattern）：策略模式定义了一系列的算法，并将每一个算法封装起来，使它们可以互相替换。策略模式使得算法可以独立于使用它的客户端变化。在苍穹外卖项目中，策略模式可能被用于实现不同的支付策略、配送策略等。

适配器模式（Adapter Pattern）：适配器模式将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而无法一起工作的那些类可以一起工作。在苍穹外卖项目中，适配器模式可能被用于处理不同系统或库之间的接口差异。

模板方法模式（Template Method Pattern）：模板方法模式定义了一个操作中的算法的框架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重新定义该算法的某些特定步骤。在苍穹外卖项目中，模板方法模式被用于定义一些通用的业务流程框架，如订单处理流程、用户注册流程等。

1. 给出4种设计模式的例子（语言不限，以组为单位），并总结其特点 （保存到每个小组选定的协作开发平台上）

（王攀 刘易 张勇 丁弘扬 每人举一个例子 ）

王攀：

装饰器模式 (Decorator)

装饰器模式（Decorator Pattern）是一种结构型设计模式，用于动态地给对象添加职责，而无需通过创建子类来扩展对象的功能。

示例 (Python):

python

class Component:

    def operation(self):

        pass

class ConcreteComponent(Component):

    def operation(self):

        return "ConcreteComponent"

class Decorator(Component):

    def \_\_init\_\_(self, component):

        self.\_component = component

    def operation(self):

        return self.\_component.operation()

class ConcreteDecoratorA(Decorator):

    def operation(self):

        return f"ConcreteDecoratorA({self.\_component.operation()})"

class ConcreteDecoratorB(Decorator):

    def operation(self):

        return f"ConcreteDecoratorB({self.\_component.operation()})"

# Usage

component = ConcreteComponent()

decorator1 = ConcreteDecoratorA(component)

decorator2 = ConcreteDecoratorB(decorator1)

print(decorator2.operation())#Output:ConcreteDecoratorB(ConcreteDecoratorA(ConcreteComponent))

特点:

1. 对象的职责动态扩展。装饰器模式允许在运行时添加新的职责，而不是在编译时通过子类实现。这使得可以根据需要动态地增强对象的功能。

2. 替代子类化。传统的扩展对象功能的方法是使用继承，通过创建子类来增加行为。装饰器模式通过组合而不是继承来实现功能扩展，避免了创建大量的子类。

3. 符合开放-封闭原则。装饰器模式符合开放-封闭原则，即对扩展开放，对修改封闭。可以通过添加新的装饰器类来扩展对象的功能，而无需修改现有代码。

4. 细粒度控制。可以使用多个不同的装饰器类，以细粒度的方式组合和排列它们，从而实现复杂的功能增强。每个装饰器类只增加一个特定的功能，这使得职责的分配更加明确和灵活。

5. 透明性。装饰器类和被装饰的原始类具有相同的接口，客户端对象可以透明地使用装饰器类，无需关心它们是具体类还是装饰器。

6. 灵活性和可扩展性。装饰器模式允许按需组合多个装饰器，具有很高的灵活性和可扩展性。可以轻松地添加或移除装饰器，重新组合不同的功能。

优点：

灵活性高：可以自由组合多个装饰器，实现不同的功能组合。

职责单一：每个装饰器类只负责一个特定的功能，符合单一职责原则。

代码复用：通过组合不同的装饰器类，可以避免代码重复，提高代码复用性。

缺点：

复杂性增加：可能会引入大量的小对象，增加系统的复杂性，理解和调试变得困难。

性能开销：由于多层装饰器的嵌套，会增加额外的开销，可能影响性能。

总结：

装饰器模式通过使用组合而不是继承来扩展对象的功能，具有很高的灵活性和可扩展性。它可以在不修改现有代码的情况下动态地添加职责，符合开放-封闭原则。然而，它也可能增加系统的复杂性，需要在使用时权衡利弊。

刘易

适配器设计模式（Adapter）

示例：

正在开发一个电子商务系统，该系统需要与第三方支付网关进行集成。不同的支付网关提供的接口各不相同，但我们希望在系统内部使用统一的支付接口。

现有情况：

PayPal 支付网关：

方法：processPayment(amount)

描述：用于处理支付的具体实现方法。

Stripe 支付网关：

方法：makePayment(value)

描述：用于执行支付的具体实现方法。

我们希望统一使用一个接口来处理支付，例如 PaymentProcessor，它包含一个方法 pay(amount)。

适配器设计模式的解决方案：

目标接口（Target）：

定义一个统一的接口 PaymentProcessor，包含 pay(amount) 方法。

目标接口用于客户代码统一调用支付处理方法。

适配者（Adaptee）：

现有的 PayPal 和 Stripe 支付网关，它们的接口各不相同。

processPayment(amount) 方法用于 PayPal 支付。

makePayment(value) 方法用于 Stripe 支付。

适配器（Adapter）：

为每个支付网关创建一个适配器，将它们的方法适配为统一的 PaymentProcessor 接口。

PayPalAdapter 将 processPayment(amount) 适配为 pay(amount)。

StripeAdapter 将 makePayment(value) 适配为 pay(amount)。

客户端（Client）：

客户端代码通过 PaymentProcessor 接口调用支付处理方法，无需关心具体的支付网关实现。

通过适配器模式，客户端可以透明地使用不同的支付网关进行支付处理。

工作流程：

客户端创建一个 PaymentProcessor 对象（例如 PayPalAdapter 或 StripeAdapter）。

客户端调用 pay(amount) 方法。

适配器将 pay(amount) 方法调用转换为具体的支付网关方法调用（processPayment(amount) 或 makePayment(value)）。

支付网关执行支付操作并返回结果。

总结适配器模式的特点：

意图：

将一个类的接口转换成客户期望的另一个接口，使得原本由于接口不兼容而不能一起工作的类可以协同工作。

适用性：

当系统需要使用现有类而它们的接口不符合系统需要时使用。

创建可重用类来协同工作，这些类可能没有相关接口或者接口不兼容。

需要使用一些现有的子类，但不可能对每个子类都进行子类化来匹配它们的接口。

结构：

目标接口（Target）：客户端所期待的接口。

适配者（Adaptee）：需要被适配的现有接口。

适配器（Adapter）：将适配者转换成目标接口。

参与者：

Target：定义客户所期待的接口。

Adapter：实现 Target 接口并把 Adaptee 的接口转换成 Target 接口。

Adaptee：定义一个已经存在的接口，适配器将使这个接口适配 Target 接口。

Client：通过 Target 接口使用 Adapter。

协作：

客户端通过目标接口调用适配器方法，从而使适配器调用适配者中的方法。

优点：

可以在不修改现有代码的情况下使用现有的类。

提高了类的复用性，使得一个类可以适配多个不相关的接口。

缺点：

过多的使用适配器会导致系统内部出现大量的适配器类，增加系统的复杂度。

有时候直接修改现有的类使其符合新的接口要求可能会更简单。

丁弘扬

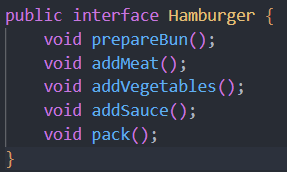
建造者模式：

建造者模式是一种创建型设计模式，用于创建复杂对象的各个部分，然后将这些部分组装成最终的对象。该模式分离了对象的构造过程和对象的表示，使得相同的构建过程可以创建不同的表示。

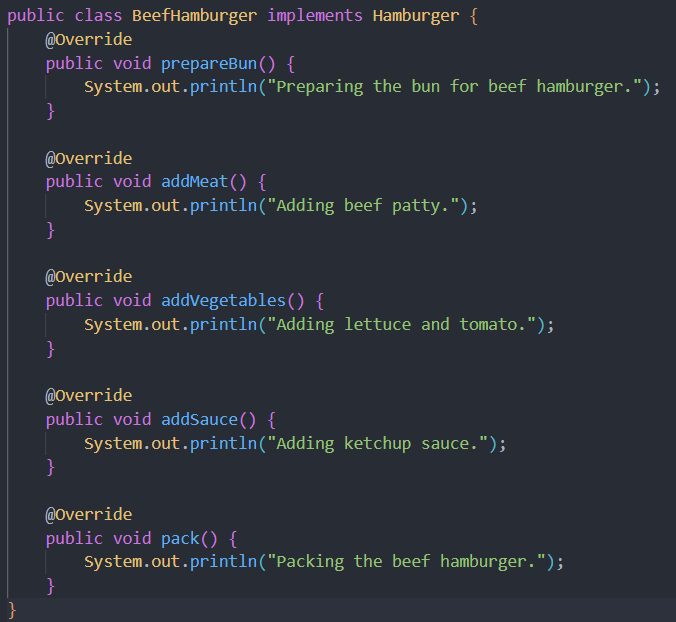
建造者模式示例：

考虑一个汉堡餐厅，菜单上有多种汉堡，每种汉堡都有特定的配料。使用建造者模式可以帮助我们以一种统一的方式来构建各种汉堡，而不会让构造逻辑变得混乱。

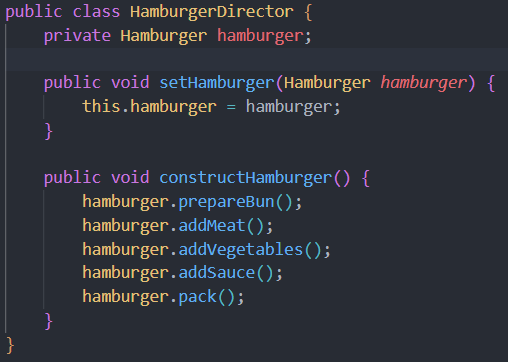
定义汉堡的接口：



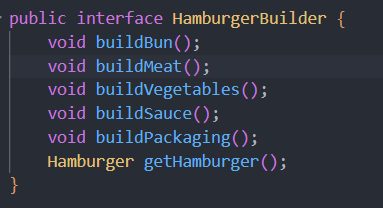
具体的汉堡实现：



创建Director类：



创建Build接口和具体的Build实现：





建造者模式的特点

1.分步骤构造：建造者模式将对象的构造过程分成一系列的步骤，这些步骤可以被重用，以不同的顺序或不同的方式执行。

2.解耦：构造过程与表示解耦，使得构造过程可以独立于产品的组成部分及构造细节。

3.控制构造细节：可以精细地控制对象的构造过程，包括哪些部分应该包含在最终的产品中。

4.易于扩展：当需要添加新的产品类型时，只需新增一个具体的建造者类，而不需要修改已有的代码。

5.多变的内部表示：同一种构建过程可以产生不同的表示，因为具体的建造者可以有不同的实现。

通过使用建造者模式，我们可以有效地管理和控制复杂对象的构造过程，特别是在对象的构造步骤较为繁琐，且可能需要根据不同的需求构造出不同变体的情况下。

张勇

单例模式（Singleton Pattern）是一种常用的软件设计模式，属于创建型模式的范畴。其主要目的是确保一个类在整个系统中仅有一个实例，并提供一个全局访问点来获取这个实例。这在某些场景下特别有用，比如控制对共享资源的访问、管理配置设置或者作为其他组件之间的协调者。

应用场景：外卖系统中可能需要一个全局访问点来管理订单处理、用户会话或是配置信息，这时单例模式就非常适用。例如，可以使用单例模式来确保整个应用程序中只存在一个数据库连接池实例，以减少资源消耗。

核心特征

唯一性：确保类在任何时刻都只有一个实例存在。这是单例模式最本质的特征。

全局访问点：提供一个全局访问方法，使得系统中的任何部分都能轻松地获取到这个唯一实例。

延迟初始化：通常情况下，单例实例会在第一次被请求时才创建（称为懒汉式），以节省资源。但也可以在类加载时就创建好（饿汉式），这取决于具体需求和性能考量。

实现方式

在Java中，实现单例模式有几种常见的方法：

饿汉式：在类加载时就创建单例对象，线程安全，但可能会造成资源浪费。

public class Singleton {

private static final Singleton instance = new Singleton();

private Singleton() {}

public static Singleton getInstance() {

return instance;

}

懒汉式（线程不安全）：在第一次调用getInstance()时创建单例对象，但原始实现非线程安全。

public class Singleton {

private static Singleton instance;

private Singleton() {}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

懒汉式（线程安全，同步方法）：通过在getInstance()方法上加锁保证线程安全，但降低了效率。

public class Singleton {

private static Singleton instance;

private Singleton() {}

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

双重检查锁定（Double-Checked Locking）：结合了懒加载和线程安全，同时提高了效率。

public class Singleton {

private static volatile Singleton instance;

private Singleton() {}

public static Singleton getInstance() {

if (instance == null) {

synchronized (Singleton.class) {

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

}

枚举方式：利用Java枚举的特性实现单例，既简洁又线程安全。

public enum Singleton {

INSTANCE;

// 可以添加其他方法和属性

}

优缺点

优点：

控制资源访问，避免因多个实例而引发的资源竞争或不一致性问题。

提高系统性能，尤其是对于那些创建成本较高的对象。

全局访问点简化了对对象的访问逻辑。

缺点：

违反了单一职责原则，单例类除了负责自己业务之外，还需要考虑单例的创建和管理。

不利于单元测试，因为全局状态难以控制和模拟。

面向对象的原则（如继承、多态）使用受限，因为全局唯一实例限制了对象的多样性。

项目跟踪，建立能反映项目及小组每个人工作的进度、里程碑、工作量的跟踪图或表，将其保存到每个小组选定的协作开发平台上，每周更新。

附加：

PPT上的模式：

设计模式从应用的角度被分为三个大的类型

创建型模式

结构型模式

行为型模式

创建型设计模式

用来创建对象的模式，抽象了实例化过程

单例模式 (Singleton)：保证一个类有且仅有一个实例，提供一个全局访问点

单例设计模式的特点：

1.单例设计模式保证一个类只有一个实例;

2.要提供一个访问该类对象实例的全局访问点。

单例模式最重要的就是要保证一个类只有一个实例并且这个类

易于被访问。

一个全局类使得一个对象可以被访问，但是这样做却不能防止

你实例化多个对象。

单例设计模式的实现：

1.为了避免其它程序过多的建立该类的对象，先禁止其它程序建

立该类对象实例（将构造器私有化）。

2.为了方便其它程序访问该类的对象，只好在本类中自定义一个

对象，由1可知该对象是static的，并对外提供访问方式。

单例模式具体实现有两种：

饿汉式(总结)

对象预先加载，线程是安全的，在类创建好的同时对象生成，

调用获得对象实例的方法反应速度快，代码简练。

懒汉式(总结)

对象延迟加载，效率高，只有在使用的时候才实例化对象，若

设计不当线程会不安全，代码相对于饿汉式复杂，第一次加

载类对象的时候反应不快。

工厂方法 (Factory Method)：父类负责定义创建对象的公共接口，而子类则负责生成具体对象，将类的实例化操作延迟到子类中完成

抽象工厂(Abstract Factory)：为一个产品族提供统一的创建接口。当需要这个产品族的某一系列的时候，可以从抽象工厂中选出相应的系列创建一个具体的工厂类

建造者模式 (Builder)：将复杂对象的构建与它的表示分离，同样的构建过程可以创建不同的表示。允许用户可以只通过指定复杂对象的类型和内容就可以构建它们，用户不知道内部的具体构建细节

建造者模式的由来

在软件系统中，有时候面临着“一个复杂对象”的创建工作

其通常由各个部分的子对象用一定的算法构成

或者按一定的步骤组合而成

这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定

在实际生活中，有很多这样的例子：

譬如一个房屋的构建。创建过程基本不变,但是门窗等组件却是易变的。

又如组装一台电脑，组装过程基本不变，可由主板、CPU、内存等按某种固定的方式组装。但主板、CPU、内存等“部件”本身会经常应需求发生变化

如何应对这种变化？如何提供一种“封装机制”来隔离出“复杂对象的各个部分”的变化，从而保持系统中的“稳定构建算法或步骤”不随着需求改变而改变？这就是我们要说的建造者模式

模式的意图

Separate the construction of a complex object from its representation so that the same construction process can create different representation. – GoF

将一个复杂对象的构建过程与其表示相分离，使得同样的构建过程可以创建不同的表示

以下情况可以使用建造者模式

需要生成的产品对象有复杂的内部结构

创建复杂对象的算法稳定，或建造者模式可以强迫生成一定的顺序

当构造过程允许被构造的对象有不同的表示时

模拟组装一台电脑

假设电脑由主板、CPU和内存三个部件构成

建造者模式的优缺点

模式优点

客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象

每一个具体建造者都相对独立，与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者，扩展方便，符合开闭原则

可以更加精细地控制产品的创建过程

模式缺点

建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，不适合使用建造者模式，因此其使用范围受到一定的限制

如果产品的内部变化复杂，可能会需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大，增加了系统的理解难度和运行成本

原型模式 (Prototype)：通过“复制”一个已经存在的实例来返回新的实例,而不是新建实例。被复制的实例就是我们所称的“原型”，这个原型是可定制的。原型模式多用于创建复杂的或者耗时的实例，因为这种情况下，复制一个已经存在的实例使程序运行更高效；或者创建值相等，只是命名不一样的同类数据

结构型设计模式

结构型模式讨论的是类和对象的结构，它采用继承机制来组合接口或实现（类结构型模式），或者通过组合一些对象，从而实现新的功能（对象结构型模式）

合成（Composite）模式：定义一个接口，使之可以用于单一对象，也可以应用于多个单一对象组成的对象组

装饰（Decorator）模式：动态给一个对象添加一些额外的职责，就好像给一个物体加上装饰物，完善其功能

代理（Proxy）模式：在软件系统中，有些对象有时候由于跨越网络或者其他的障碍，而不能够或者不想直接访问另一个对象，如果直接访问会给系统带来不必要的复杂性，这时候可以在客户程序和目标对象之间增加一层中间层，让代理对象来代替目标对象打点一切，这就是代理（Proxy）模式

享元（Flyweight）模式：Flyweight是一个共享对象，它可以同时在不同上下文（Context）使用

外观（Facade）模式：外观模式为子系统提供了一个更高层次、更简单的接口，从而降低了子系统的复杂度和依赖。这使得子系统更易于使用和管理。外观承担与子系统中类交互的责任

桥梁（Bridge）模式：桥梁模式的用意是将问题的抽象和实现分离开来实现，通过用聚合代替继承来解决子类爆炸性增长的问题

适配器(Adapter)模式：将一个类的接口适配成用户所期待的接口。一个适配器允许通常因为接口不兼容而不能在一起工作的类工作在一起，做法是将类自己的接口包装在一个已存在的类中

行为型设计模式

着力解决的是类实体之间的通讯关系，希望以面向对象的方式描述一个控制流程

模版方法（Template Method）：定义了一个算法步骤，并允许子类别为一个或多个步骤提供其实现方式。让子类别在不改变算法架构的情况下，重新定义算法中某些步骤

观察者（Observer）模式：定义了对象之间一对多的依赖，当这个对象的状态发生改变的时候，多个对象会接受到通知，有机会做出反馈

观察者模式的由来

模式动机

建立一种对象与对象之间的依赖关系，一个对象发生改变时将自动通知其他对象，其他对象将相应做出反应。在此，发生改变的对象称为观察目标，而被通知的对象称为观察者，一个观察目标可以对应多个观察者，而且这些观察者之间没有相互联系，可以根据需要增加和删除观察者，使得系统更易于扩展，这就是观察者模式的模式动机。

实例

某些寻找工作的人对“求职中心”的职业需求信息的变化非常关心，很想跟踪“求职中心”中职业需求信息的变化。“求职者”可以让“求职中心”把自己登记下来，这样求职中心就会及时通知她最新的职业信息需求。

观察者模式

定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。观察者模式又叫做发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式或从属者（Dependents）模式。

观察者模式包含如下角色

Subject: 抽象目标

ConcreteSubject: 具体目标

Observer: 观察者接口

ConcreteObserver: 具体观察者

观察者模式描述了如何建立对象与对象之间的依赖关系，如何构造满足这种需求的系统。

这一模式中的关键对象是观察目标和观察者，一个目标可以有任意数目的与之相依赖的观察者，一旦目标的状态发生改变，所有的观察者都将得到通知。

作为对这个通知的响应，每个观察者都将即时更新自己的状态，以与目标状态同步，这种交互也称为发布-订阅(publish-subscribe)。目标是通知的发布者，它发出通知时并不需要知道谁是它的观察者，可以有任意数目的观察者订阅它并接收通知。

观察者模式的优点

具体目标和具体观察者是松耦合关系。由于目标接口仅仅依赖于观察者接口，因此具体目标只是知道它的观察者是实现观察者接口的某个类的实例，但不需要知道具体是哪个类。同样，由于观察者仅仅依赖于目标接口，因此具体观察者只是知道它依赖的目标是实现目标接口的某个类的实例，但不需要知道具体是哪个类。

观察模式满足“开-闭原则”。目标接口仅仅依赖于观察者接口，这样，我们就可以让创建具体目标的类也仅仅是依赖于观察者接口，因此如果增加新的实现观察者接口的类，不必修改创建具体目标的类的代码。同样，创建具体观察者的类仅仅依赖于目标接口，如果增加新的实现目标接口的类，也不必修改创建具体观察者类的代码。

观察者模式的缺点

如果一个观察目标对象有很多直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。

观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

迭代子（Iterator）模式：提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示

责任链（Chain of Responsibility）模式：很多对象由每一个对象对其下一个对象的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使系统可以在不影响客户端的情况下动态的重新组织链和分配责任

备忘录（Memento）模式：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态

命令（Command）模式：将请求及其参数封装成一个对象，可以对这些请求排队或记录请求日志，以及支持可撤销操作

状态（State）模式：允许一个“对象”在其内部状态改变的时候改变其行为，即不同的状态，不同的行为

访问者（Visitor）模式：表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作

解释器(Interpreter) 模式：给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子

中介者（Mediator）模式：用一个中介对象来封装一系列的对象交互

中介者模式的由来

在用户与用户直接聊天的设计方案中，用户对象之间存在很强的关联性，将导致系统出现如下问题：

系统结构复杂：对象之间存在大量的相互关联和调用，若有一个对象发生变化，则需要跟踪和该对象关联的其他所有对象，并进行适当处理。

对象可重用性差：由于一个对象和其他对象具有很强的关联，若没有其他对象的支持，一个对象很难被另一个系统或模块重用，这些对象表现出来更像一个不可分割的整体，职责较为混乱。

在面向对象的软件设计与开发过程中，根据“单一职责原则”，我们应该尽量将对象细化，使其只负责或呈现单一的职责。

对于一个模块，可能由很多对象构成，而且这些对象之间可能存在相互的引用，为了减少对象两两之间复杂的引用关系，使之成为一个松耦合的系统，我们需要使用中介者模式，这就是中介者模式的模式动机。

中介者模式定义

用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

中介者承担两方面的职责：

中转作用（结构性）：通过中介者提供的中转作用，各个同事对象就不再需要显式引用其他同事，当需要和其他同事进行通信时，通过中介者即可。该中转作用属于中介者在结构上的支持。

协调作用（行为性）：中介者可以更进一步的对同事之间的关系进行封装，同事可以一致地和中介者进行交互，而不需要指明中介者需要具体怎么做，中介者根据封装在自身内部的协调逻辑，对同事的请求进行进一步处理，将同事成员之间的关系行为进行分离和封装。该协调作用属于中介者在行为上的支持。

中介者模式优缺点

中介者模式的优点

简化了对象之间的交互

将各同事解耦

减少子类生成

可以简化各同事类的设计和实现

中介者模式的缺点

在具体中介者类中包含了同事之间的交互细节，可能会导致具体中介者类非常复杂，使得系统难以维护。

策略（Strategy）模式：定义一组算法，将每个算法都封装起来，并且使它们之间可以互换。策略模式使这些算法在客户端调用它们的时候能够互不影响地变化