设计模式报告

1. 基本功能描述

1.1 登录注册功能

对于个人用户以及管理员用户，可以实现在系统的注册、登录、注销的功能，管理员账号为初始设定好的，不可注册。

1.2 个人管理功能

对于个人用户，可以对自己的个人信息进行编辑和修改，可以对资金进行相应的充值。

1.3 卖家管理功能

每一个用户可以作为交易平台上的卖家，可以上传自己的商品进入交易市场。在有用户购买自身商品后，可在商家界面中查看，并在平台上进行发货处理。

1.4 商品市场功能

本网站提供一个大型的商品交易市场，所有用户可以在此添加自身喜爱的收藏商品，可以在此浏览、搜索商品，并对相应的商品进行购买。

1.5 后台管理功能

对于后台管理员，管理员可以查看到所有的交易信息，有可视化图表显示近来的交易数目；并且可以查看商品市场上的所有商品，对不合规的商品进行操作处理；此外，管理员也可以对用户信息进行查看和编辑等操作

2. 设计模式选择

2.1 体系结构分析

本项目是围绕着数据的增删改查的数据中心模型，项目的主要操作是有关数据的操作。因此我们的设计模式也应该围绕着这一特点展开。同时，由于我们的项目围绕着交易市场展开，市场中的商品，用户，订单等都可以用面向对象的思想进行建模。

2.2 设计模式选用

设计模式是针对特定软件设计问题的通用解决方案，它们在软件工程领域中出现，并通过多年经验和应用不断发展和改进。设计模式提供了一种通用的术语和符号，能够帮助开发人员更清晰和明确地描述软件设计，并可极大地促进开发团队间的交流。按照设计模式去处理软件设计问题，可以提高软件开发效率、可读性、可维护性和可扩展性。在我们的项目中，我们采用了不少经典的设计模式来优化提高团队之间的沟通效率与协作能力。

根据《设计模式：可复用面向对象软件的基础》，可以将课程项目采用的设计模式分为“创建型模式”“结构型模式”与“行为型模式”。

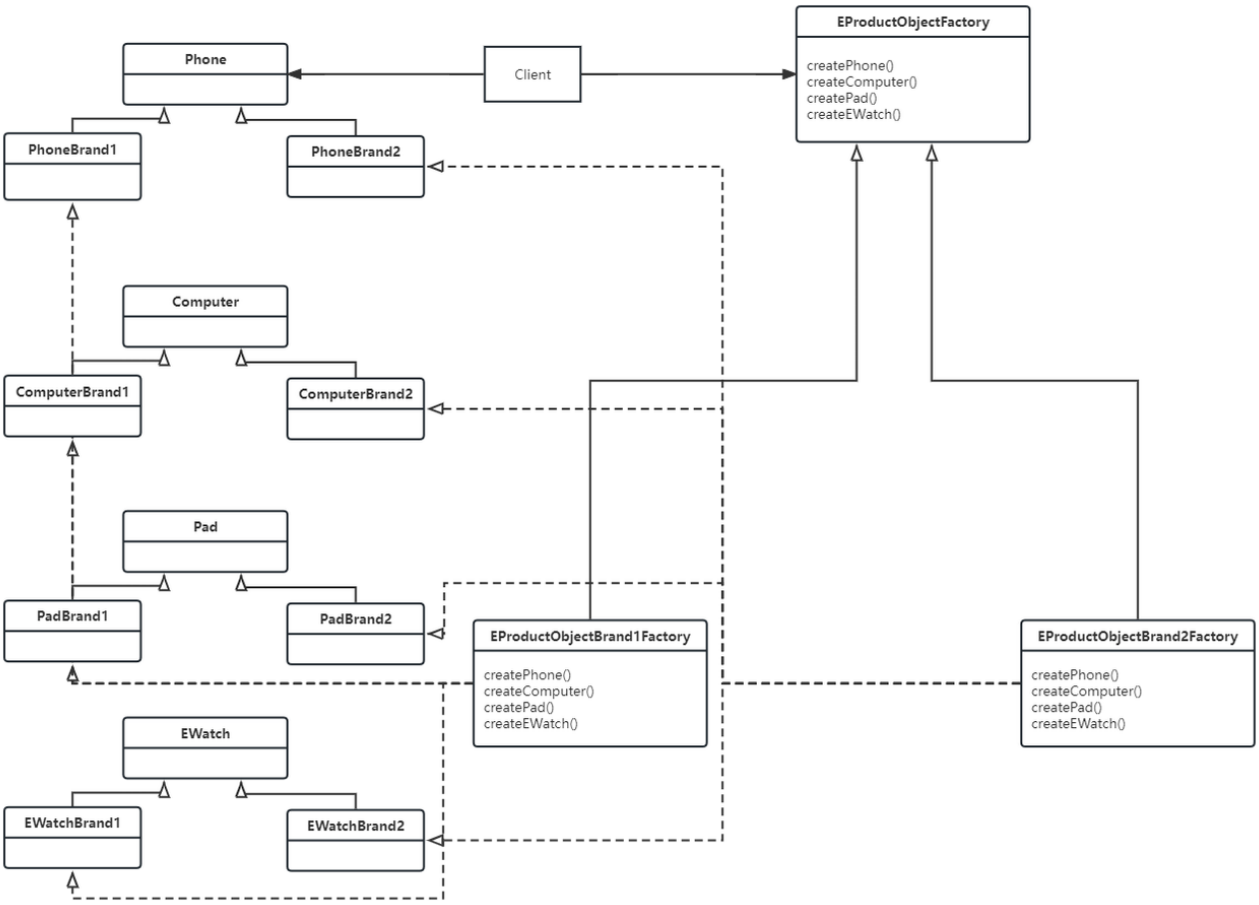
2.2.1 创建型模式

1. 抽象工厂（Abstract Factory）——对象创建型模式

抽象工厂能够提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。这非常适合用于系统内一系列相关产品的创建设计。

抽象工厂可将整个对象创建过程封装在工厂接口和工厂实现中，从而有效消除重复的代码，并提高代码复用率。在我们的项目中，市场内的产品主要分为“图书音像”“家居生活”“服饰箱包”和“电子产品”4种类型，而这些商品大类可以认为是由具体种类的商品类组合实现的。这样的业务逻辑非常适合应用抽象工厂去实现。以“电子产品”类型为例，客户可以通过`EProductObjectFactory`进行生产`Phone`、`Computer`、`Pad`、`EWatch`等类型。以上的创建过程逻辑相似，而运用抽象工厂的设计模式可以大大提高代码的可重用性。

同时，抽象工厂通过工厂接口和工厂实现，可轻松添加和扩展新的对象类型，而无需更改现有对象的创建过程。同样以“电子产品”类型为例。假如在后续的开发设计中，我们想要在“电子产品”类型的商品中增加新品牌的一系列产品，那么我们不需要对代码进行大规模的修改，只需要在`EProductObjectFactory`中添加对应的一个子类，并在这个子类内实现具体的创建逻辑即可。这样的设计模式能够为项目的后续维护完善提供非常优秀的基础。



2. 生成器（Builder）——对象创建型模式

生成器能够将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。使用生成器模式，能够显著优化复杂对象的创建。

生成器允许我们在创建对象时根据需求灵活地添加/删除组件，这意味着使用生成器创建的对象具有更大的可配置性，能够适应不同的使用场景。在我们的项目中，商品类是一个比较复杂的对象，它包括商品ID、商品名称、商品价格、商品类型、存货数量、销量、商品描述、销售者ID、邮费等属性。而假如销售者在提供商品信息时，并没有提供完整的信息，那么可以利用生成器模式在创建商品时调整对应的属性组合，从而达到成功创建商品的目的。

相关的简略代码如下所示。

```
import java.util.UUID;
•
public class Good {
    private String goodId;
    private String goodName;
    private double price;
    private int type;
```

```
8       private int storageAmount;
9       private int salesVolume;
10      private String description;
11      private String sellerId;
12      private double postage;
13
14      // 构造函数
15      public Good(String goodId, String goodName, double price, int type,
    int storageAmount, int salesVolume, String description, String sellerId,
    double postage) {
16          this.goodId = goodId;
17          this.goodName = goodName;
18          this.price = price;
19          this.type = type;
20          this.storageAmount = storageAmount;
21          this.salesVolume = salesVolume;
22          this.description = description;
23          this.sellerId = sellerId;
24          this.postage = postage;
25      }
26
27      // Getter和Setter方法
28      // ...
29
30      //Builder设计模式
31      public static class GoodBuilder {
32          private String goodId = UUID.randomUUID().toString();
33          private String goodName;
34          private double price;
35          private int type;
36          private int storageAmount;
37          private int salesVolume;
38          private String description;
39          private String sellerId;
40          private double postage;
41
42          public GoodBuilder setGoodName(String goodName) {
43              this.goodName = goodName;
44              return this;
45          }
46
47          public GoodBuilder setPrice(double price) {
```

```
48              this.price = price;
49              return this;
50          }
51  •
52          public GoodBuilder setType(int type) {
53              this.type = type;
54              return this;
55          }
56  •
57          public GoodBuilder setStorageAmount(int storageAmount) {
58              this.storageAmount = storageAmount;
59              return this;
60          }
61  •
62          public GoodBuilder setSalesVolume(int salesVolume) {
63              this.salesVolume = salesVolume;
64              return this;
65          }
66  •
67          public GoodBuilder setDescription(String description) {
68              this.description = description;
69              return this;
70          }
71  •
72          public GoodBuilder setSellerId(String sellerId) {
73              this.sellerId = sellerId;
74              return this;
75          }
76  •
77          public GoodBuilder setPostage(double postage) {
78              this.postage = postage;
79              return this;
80          }
81  •
82          public Good build() {
83              return new Good(goodId, goodName, price, type, storageAmount,
    salesVolume, description, sellerId, postage);
84          }
85      }
86  }
```

我们利用上述代码来创建不同种类的商品，这使得我们不需要关心商品构造的具体过程与细节，简化了创建思路，同时也提高了提高代码的可重用性、可读性和可维护性。

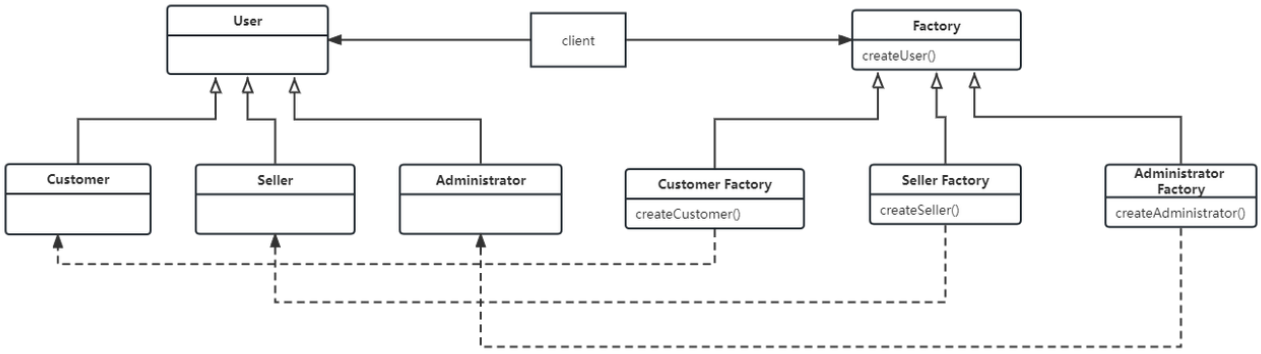
3. 工厂方法（Factory Method）——对象创建型模式

工厂方法能够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。简而言之，工厂方法使一个类的实例化延迟到其子类。这非常适合当一个类希望由它的子类来指定它所创建的对象的时候。

在我们的项目中，针对具有相似属性的用户分为三种：买家，卖家和管理员。为了创建这些用户，我们使用了工厂方法设计模式。在实现中，该工厂方法将延迟用户类的实例化，直到我们获得更多信息并可以确定用户应该属于哪个子类时。

我们定义了一个用户工厂类，它接受用户的一组初始信息，并根据需要创建适当的用户子类。这样，当工厂方法接收到关于新用户的信息时，它将检查一些逻辑以确定用户类的类型，并使用相应的子类进行实例化，然后返回该实例。

使用工厂方法的好处在于我们可以将用户类的创建和实现从客户端代码中分离出来，这使得我们可以更加灵活地管理代码，可以很方便地添加新用户类的子类或者调整对象的实现方式。同时，我们也可以更容易地测试代码。



4. 原型（Prototype）——对象创建型模式

原型这种设计模式能够用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。这非常适合用于需要频繁创建的对象，并且需要在不同的环境下复用现有对象的场景。在这种情况下，原型模式可以复用已有对象，从而节省了创建对象的时间和开销。同时，原型模式能够降低代码的耦合性，易于维护和升级，因此在大规模复杂应用系统中广泛应用。

在我们的项目中，并没有上述频繁创建对象的需求，因此项目程序没有涉及原型这一设计模式的相关实现代码。但是为了方便对象的拷贝复制，我们利用相关的拷贝构造函数，实现了创建一个新对象并复制原始对象的值。

拷贝构造函数适用于需要在不同的环境下复用现有对象，并且这些对象的创建不需要进行大规模性能优化的场合。在这种情况下，我们可以通过拷贝构造函数来实现对象的拷贝复制，从而避免手动复制对象值的繁琐过程，提高代码编写的效率。

通过综合考虑，在本项目中，我们采用拷贝构造函数进行对象的复制来进行相关代码的管理与维护。

5. 单件（Singleton）——对象创建型模式

单件模式能够保证一个类只有一个实例，并提供访问这个实例的全局点。单件模式可以防止多个实例被创建，从而减少了资源的消耗，也避免了对象状态和行为的分散。在程序开发中，开发者可以利用静态变量和枚举类型等方法来实现单件模式。在多线程环境中，多个线程可以共享同一个单件对象，避免了线程之间的竞争和锁定，能够提高程序的执行效率。

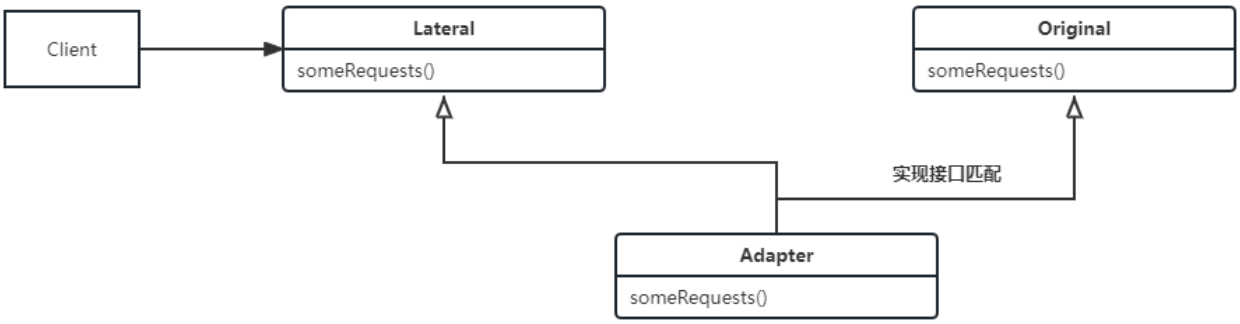
在我们的项目中，没有相关的类有唯一实例的需求，因此我们没有在实际代码中运用单件这一设计模式。

2.2.2 结构型模式

1. 适配器（Adapter）——类对象结构型模式

适配器模式的主要目的是将一个类的接口转换成客户端所期望的另一个接口，从而使原本不兼容的类能够协同工作。适配器模式非常适合用于旧代码的重构或在组件间接口不兼容时进行协调。

由于本项目由多人进行合作开发，在实际开发推进的过程中，可能会产生接口不兼容的问题，导致代码无法正常运行。此时，我们可以利用适配器模式进行调整与完善。假如我们原本设计了一个类`Original`，而在后续项目推进的过程中，我们最终决定用另一个类`Lateral`去调用`Original`，这就会造成类之间接口的不匹配问题。为了解决这个问题，我们可以再设计一个`Adapter`类来实现这两个类之间接口的匹配。

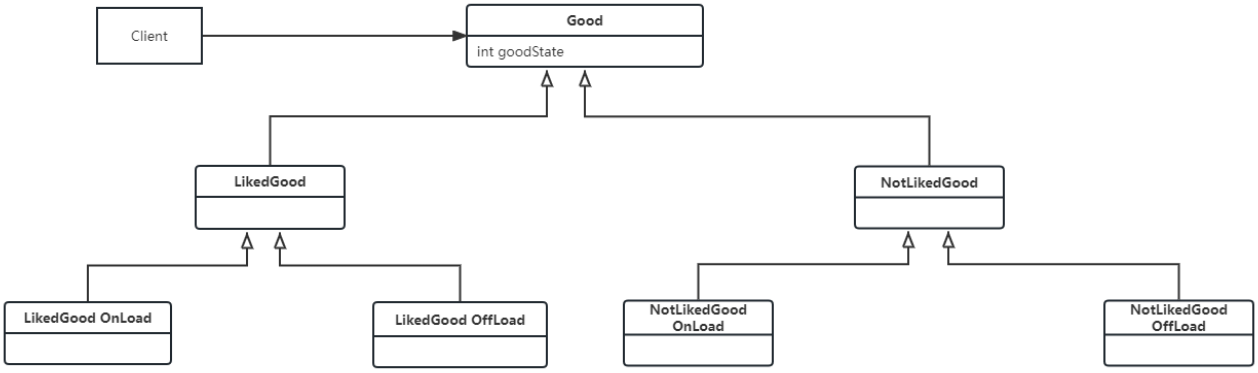


当然，上述是单向适配器的设计模式。如果有需求，也可以设计双向适配器。双向适配器可以将一个类转换为另一个接口，同时也可以将另一个接口转换为该类，从而实现双向转换。这种模式在需要在多个类之间进行复杂而灵活的对象转换时非常有用。

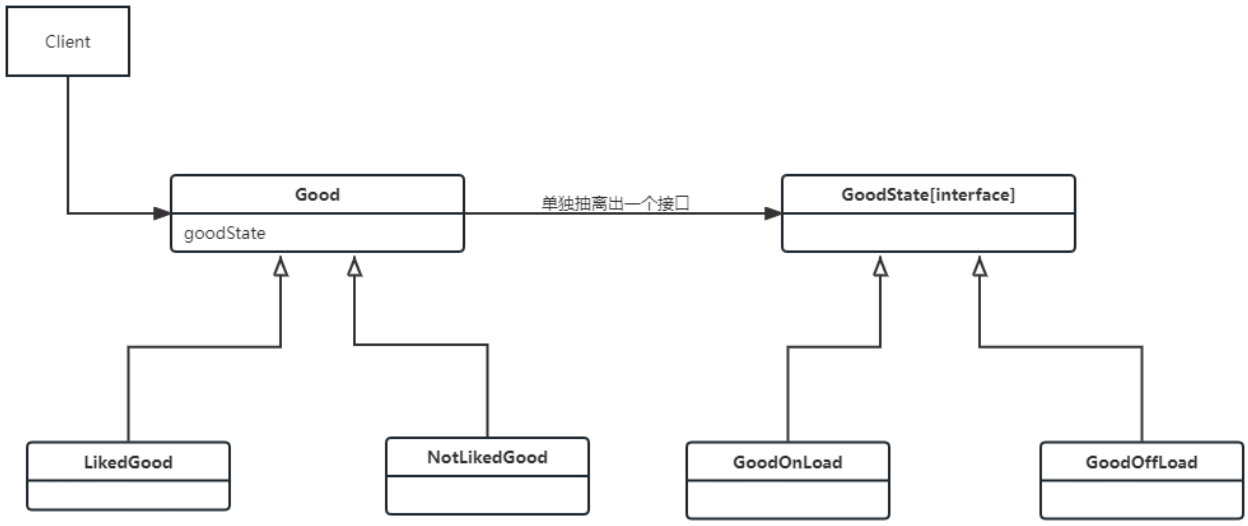
2. 桥接（Bridge）——对象结构型模式

桥接模式能够将抽象部分与它的实现部分分离，使它们都可以独立地变化。桥接模式的核心理念是"组合优于继承"，它通过将接口和实现分离开来，从而使得它们可以随意组合，并且能够独立地改变这两个部分中的任何一个而不影响到另一个。

在本项目中，商品类的state属性根据买方操作，可以分为未收藏和收藏这2种状态。而state属性根据销售者操作，可以分为已上架和已下架这2种状态。如果直接进行继承，那么整体结构会相对复杂。同时，这样的继承方式存在一定的冗余信息，非常不利于提高代码的可复用性。



如果采用桥接模式，那么继承模式会被优化到如下所示。



在采用桥接模式之后，我们能够对Like和Load层次结构进行各自的进一步扩充，这大大优化了代码之间的解耦合，使得模块化的思想原则在项目程序中得到了更好的体现。

3. 组合（Composite）——对象结构型模式

组合这一设计模式能够将对象组合成树形结构以表示“部分-整体”的层次结构，使得用户对单个对象和组合对象的使用具有一致性。组合模式中通常有两种类型的对象：叶子节点（Leaf）和组合节点（Composite）。叶子节点代表单个对象，而组合节点则包含叶子节点和其他组合节点。组合这一设计模式比较适合用于以下场景：系统内存在复杂对象，且该复杂对象是由多个系统内的简单对象组合而成。针对上述情况，组合模式可以统一接口。即使在处理复杂的树形结构时，用户也只需要调用同一个接口即可，而不需要关心具体的实现细节。

在我们的项目中，由于涉及到的对象彼此之间并没有包含和从属的明显关系，因此在项目实现中，并没有运用组合这一设计模式。

4. 装饰（Decorator）——对象结构型模式

装饰这一设计模式允许在运行时动态地为对象添加新的职责，同时又不改变其原有的结构和行为。这样可以在不修改原始代码的基础上进行功能扩展或修改，从而提高代码的可维护性和可复用性。与直接生成子类相比，装饰模式能够更加灵活地为对象添加功能。

在我们的项目中，由于我们实现的是一个基础的电子商务系统，涉及到的主要操作是商品买卖和金额充值等，因此没有明显的为对象添加新功能的需求。

但是我们设计保留了部分接口，可以在之后拓展商品促销价格计算的功能。而该功能可以利用装饰模式来进行高效合理的实现。在系统中，每个商品都可以看作是一个基础的组件，我们可以为其添加一些额外的促销策略和赠品策略，同时也可以根据不同的用户信息和订单信息，动态地为不同的商品添加不同的装饰器来实现更具体的促销策略和赠品策略。

大致的代码思路如下。

```
// 定义商品接口
Interface Good
    getName() // 获取商品名
    getPrice() // 获取商品价格
End Interface

// 实现商品接口的基类
Class BaseGood Implements Good
    String name // 商品名
    Decimal price // 商品价格

    Constructor(name, price) // 构造函数
        this.name = name
        this.price = price

    Function getName() // 获取商品名

    Function getPrice() // 获取商品价格

End Class

// 定义装饰模式的核心部分 - 装饰器基类
Abstract Class GoodDecorator Implements Good
    Protected good // 待装饰的商品

    Constructor(good) // 构造函数
        this.good = good

    Function getName() // 获取商品名

    Function getPrice() // 获取商品价格

End Class

```

```
// 一个具体装饰类 - 折扣商品类
Class DiscountGood Extends GoodDecorator
    Decimal discount // 折扣率

    Constructor(good, discount) // 构造函数
        Super(good) // 调用父类构造函数，传入待装饰商品
        this.discount = discount

    Function getPrice() // 获取商品价格
        // 调用待装饰商品的getPrice()方法，并乘以折扣率
        Return this.good.getPrice() * this.discount
    End Function
End Class

// 另一个具体装饰类 - 赠品商品类
Class GiftGood Extends GoodDecorator
    String gift // 赠品

    Constructor(good, gift) // 构造函数
        Super(good) // 调用父类构造函数，传入待装饰商品
        this.gift = gift

    Function getName() // 获取商品名
        // 调用待装饰商品的getName()方法，并在其后面添加赠品信息
        Return this.good.getName() + " + " + this.gift
    End Function
End Class
```

抽象类GoodDecorator作为装饰类，实现了Good接口。DiscountGood和GiftGood分别作为GoodDecorator类的子类，实现了不同的装饰行为。

5. 外观（Facade）——对象结构型模式

外观这一设计模式能够为复杂的子系统提供一个简化的接口。这个接口隐藏了子系统的复杂性和让其易于使用。这一设计模式非常适合用于系统子模块功能比较复杂和数量较多的情况。在这种场景下，外观模式有助于松耦合系统中的对象，并将系统分解为更小的模块，这些模块可以更易于管理、维护和升级。外观模式可以使得各个子系统拥有自己的独立实现，系统整体表现为一致的接口，对客户端隐藏了子系统的复杂性。

同时，在多人协作的项目中，外观模式也能在一定程度上实现模块兼容协调的问题。

6. 享元（Flyweight）——对象结构型模式

享元这一设计模式通过共享细粒度对象来实现对大量对象的有效管理，以减少程序运行时的内存开销。享元模式的核心思想是共享对象，它在多次使用相同的对象时可以避免重复对象的创建，只需引用已有对象即可。这一设计模式比较适合用于存在大量类似对象或存在不可变对象的场景。在这些情况下，使用共享的方法可以在保证安全和效能的前提下降低开销。

在我们的项目中，由于涉及到的对象彼此之间相互独立且大部分属性可变，因此在项目实现中，并没有运用享元这一设计模式。

7. 代理（Proxy）——对象结构型模式

代理这一设计模式能够在访问对象时引入一定程度的间接性，从而可以在不改变对象的情况下控制访问。代理模式可以通过定义代理类来实现，代理类与所代理对象具有相同的接口，从而可以对外表现为同样的对象。这一设计模式非常适合用于以下场景：需要访问远程对象、需要限制对真实对象的访问、需要在访问真实对象时添加额外的行为（例如记录日志、计算执行时间等）等。

在我们的项目中，由于并没有涉及到上述的相似场景，因此在项目实现中，并没有运用代理这一设计模式。

2.2.3 行为型模式

1. 职责链（Chain of Responsibility）——对象行为型模式

职责链这一设计模式能够使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。这些对象会被连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

职责链模式能够降低耦合，从而可以降低系统的耦合度和增加系统的可维护性。同时，职责链模式可以动态添加和删除处理器，从而提高系统的灵活性和可扩展性。此外，职责链模式可以将类的功能单一化，使得每个类只需要专注于处理自己的角色，简化了对象的复杂度。

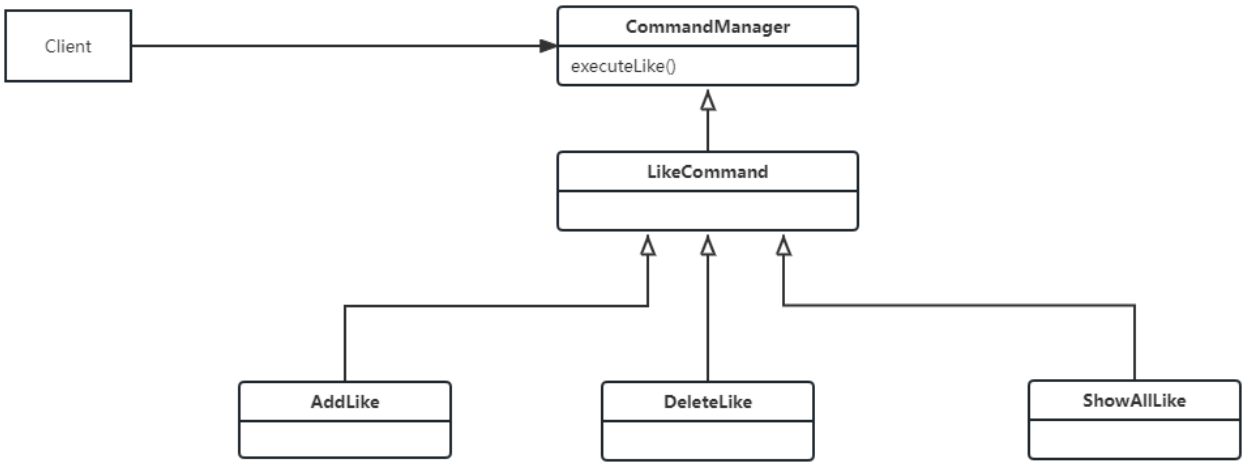
在我们的项目中，由于并没有成链的业务逻辑关系，因此在项目实现中，并没有运用职责链这一设计模式。

2. 命令（Command）——对象行为型模式

命令这一设计模式能够将请求转换为一个包含请求信息的对象，从而允许发送方和接收方彻底解耦。它允许请求的发送者与接收者完全解耦，使得发送者不需要知道接收者的存在，而接收者也不需要知道发送者的存在。该模式将请求封装在一个对象中，并将参数和操作绑定在一起。这一设计模式非常适合用于需要支持撤销和重做操作、需要在系统中基于事件驱动的架构实现等场景。

命令模式通过将命令封装为独立的对象，使得其在不同的请求和操作之间能够提供更好的参数化和抽象。另外，命令模式也能够增强灵活性、降低耦合度、简化代码设计等方面都有所改善。

在我们的项目中，用户收藏商品这一行为是通过命令模式实现的。



在购物流程中每个用户执行商品收藏操作时，可以创建相应的具体命令对象，然后将命令对象作为参数传递给CommandManager对象的executeLike()方法，执行命令时通过CommandManager对象调用executeLike()方法触发具体命令对象的操作。

3. 解释器（Interpreter）——类行为型模式

解释器这一设计模式定义了一种语言来解释表达式，其主要目的是为给定语言定义一种文法，并按照该文法解释语言中的句子。这一设计模式可以简化语法树中的节点数量，从而简化代码的实现。同时，解释器通过把复杂的语言文法分解为一些简单的小部件，可以为开发人员提供更好的掌握语言的体系结构。

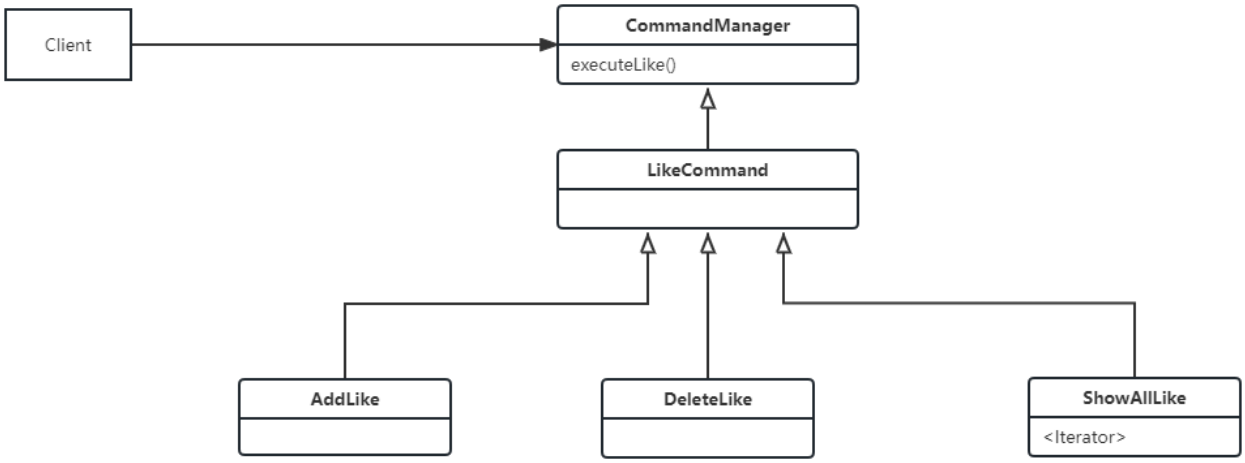
在我们的项目中，由于并没有涉及到对某种语言的文法规定需求，因此在项目实现中，并没有运用代理这一设计模式。

4. 迭代器（Iterator）——对象行为型模式

迭代器这一设计模式能够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。这种模式提供了一种简单而通用的方式，用于在访问集合中的元素时避免代码中的大量重复。

迭代器模式的核心是创建一个迭代器类，负责遍历集合并提供对其元素的访问操作（比如`next()`方法）。然后，客户端代码使用该迭代器类遍历集合，而不必关心其内部实现。这可使代码更加干净，可维护性更高。而且，迭代器可以提供不同的遍历方式，例如倒序遍历。

在我们的项目中，显示用户收藏的商品这一行为是通过迭代器模式实现的。



和直接使用`for`循环相比，使用迭代器可以避免将整个集合都加载到内存中，从而提高了代码的运行效率和内存使用效率。此外，迭代器可以让代码更加直观，更加简洁易懂，特别是遍历嵌套结构时，使用迭代器可以使代码更加清晰。

5. 中介者（Mediator）——对象行为型模式

中介者这一设计模式通过用一个中介对象来封装一系列的对象交互。这使得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。这一设计模式比较适合用于以下场景：系统涉及一组对象，其中每个对象都需要相互通信，但不希望彼此之间直接耦合。这时，可以引入一个中介者对象，负责协调对象之间的交互。可以将每个对象的消息转发给中介者，由中介者处理和分发，而不是互相之间直接通信。

使用中介者模式不仅能够减少对象之间的直接耦合，还能提高代码可重用性。同时，中介者可以对对象之间的交互进行统一管理，这使得对代码进行修改和维护更加便捷。

在我们的项目中，买家与卖家通过商品进行交互，由于此处的交互较为简单直接，因此并没有设计中介者来介入。如果之后有更复杂的交互功能待开发，那么可以在此处引入中介者来优化整体代码。

6. 备忘录（Memento）——对象行为型模式

备忘录这一设计模式能够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。因此，系统就具备了将该对象恢复到原先保存的状态的能力。使用这一设计模式还能隐藏状态的实现细节：通过备忘录对象，状态信息被隐藏在内部，其他对象不能直接访问，提高了程序的安全性。同时，备忘录模式支持撤销操作，以实现类似“撤销”和“重做”的功能。

在我们的项目中，不允许购买操作、充值操作的“撤回”，因此在项目实现中，并没有运用备忘录这一设计模式。

7. 观察者（Observer）——对象行为型模式

观察者这一设计模式使得一个对象的状态变化能够通知其他依赖于该对象的对象。观察者模式提供了一种松散耦合的方式，使得我们可以在不影响其他对象的情况下，更改一个对象的状态。

在观察者模式中，存在一个主题对象（也称为被观察者或可观察者对象），其状态的变化可能会影响其他对象。此外，还存在一组观察者对象，它们依赖于主题对象的状态并希望在状态变化时得到通知。当主题对象的状态发生变化时，通常会发送通知给所有的观察者对象，以便它们可以更新自己。

在我们的项目中，买家购买商品，卖家销售商品，两者通过商品进行交互，并没有存在更新通知的业务需求，因此在项目实现中，并没有运用观察者这一设计模式。

8. 状态（State）——对象行为型模式

状态这一设计模式允许对象在其内部状态发生变化时改变其行为。状态模式将对象的行为委托给表示其内部状态的一个或多个对象，从而实现状态转换时的行为变化。使用状态模式的核心思想是将与特定状态相关的行为分离到一个状态类中，以便对象在其内部状态发生变化时可以更改其行为而不是通过大量条件语句来实现。

在网站搭建中，TCP连接是典型的运用状态模式的例子。TCP连接是一种有序、可靠、双向、基于字节流的面向连接协议。在TCP连接的建立、传输和终止过程中，客户端和服务器端会根据不同的阶段采用不同的状态。因此，TCP连接的不同阶段可以使用状态模式来进行有用的建模。

TCP连接通常会经过以下三个阶段：

1. 建立连接 (establishment)

在建立连接时，TCP连接的状态由CLOSED转变为SYN_SENT。在这个过程中，客户端向服务器端发送一份SYN请求报文，提示自己想要开始一个连接。

2. 数据传输 (data transfer)

在数据传输时，TCP连接的状态被改为ESTABLISHED，即客户端和服务器端之间的连接已经建立起来。在这个状态下，数据不断地在连接上进行传输。

3. 断开连接 (termination)

在断开连接时，TCP连接的状态由ESTABLISHED转变为FIN_WAIT_1，即开始发送关闭连接报文的一方处于“半关闭”状态。随后，另一方收到“关闭连接”报文时会进入TIME_WAIT状态，等待最终关闭。最终，双方都进入CLOSED状态，连接关闭完成。

状态模式可以将每个阶段视为一个单独的状态对象，这个对象实现了一个公共接口，其中包括能够表示当前状态的所有方法。当TCP连接从一个状态转换到另一个状态时，将把其状态更改为新状态的对象。

9. 策略（Strategy）——对象行为型模式

策略这一设计模式定义了一系列的算法，把它们一个个封装起来，并且使它们可相互替换。这种做法使得算法可独立于使用它的客户而变化。在策略模式中，一个算法被封装在一个对象中，并且可以由其他对象调用。这些对象可以共享相同的算法，也可以使用不同的算法。

使用策略模式能够允许客户端在运行时选择算法，这可以根据不同的输入参数动态地选择算法。

在我们的项目中，并没有涉及到多种算法选择的业务需求，因此在项目实现中，并没有运用策略这一设计模式。

10. 模板方法（Template Method）——类行为型模式

模板方法这一设计模式能够定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。它使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

在用户注册登录模块，注册与登录属于两种操作，但是其实现逻辑存在共通之处。可以通过如下思路用模板方法实现用户登录注册。

```
// 抽象类
abstract class User {
    // 输入用户信息（抽象方法）
    abstract void inputUserInfo();
```

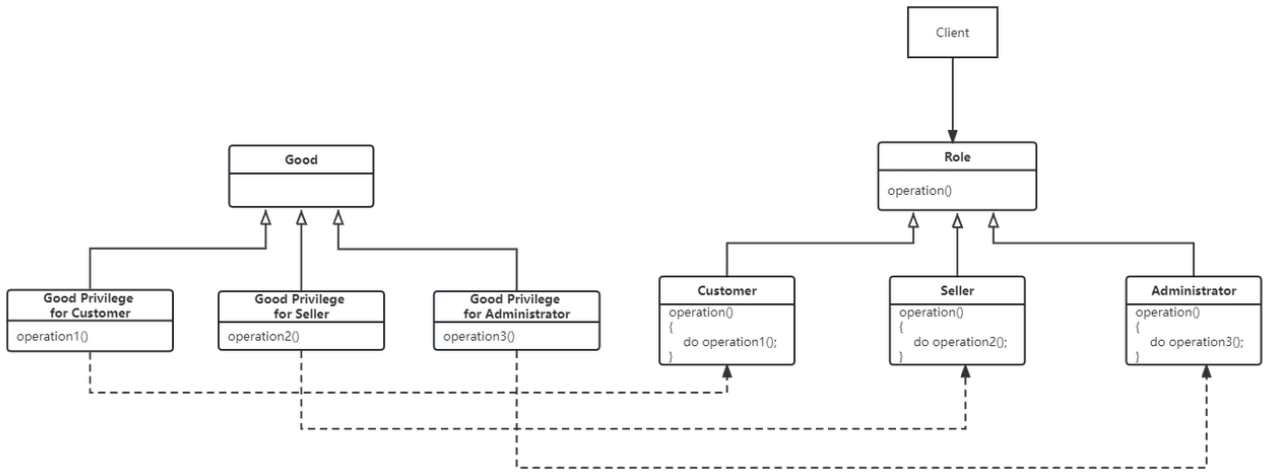
```
5
6       //处理用户信息（抽象方法）
7       abstract void handleInfo();
8
9       // 登录或注册流程（模板方法）
10      public final void loginOrRegister() {
11          inputUserInfo();
12          handleInfo();
13      }
14  }
15  •
16  // 具体子类--登录
17  class Login extends User {
18      // 输入用户信息
19      @Override
20      void inputUserInfo() {
21          ...
22      }
23
24      // 处理用户信息
25      @Override
26      void handleInfo() {
27          ...
28      }
29  }
30  •
31  // 具体子类--注册
32  class Register extends User {
33      // 输入用户信息
34      @Override
35      void inputUserInfo() {
36          ...
37      }
38
39      // 处理用户信息
40      @Override
41      void handleInfo() {
42          ...
43      }
44  }
```

上述的设计思路能够提高代码复用性，降低代码维护成本，同时提供了更好的扩展性与维护了一致性。

11. 访问者（Visitor）——对象行为型模式

访问者这一设计模式能够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。该模式定义了一个称为访问者的新对象，该对象通过对其他对象执行操作来更改其行为。访问者模式最适合于处理复杂的结构而不影响其元素的类层次结构。

在我们的项目中，存在三种身份的用户：顾客、销售者与管理员。不同用户的权限不同，因此他们的可执行操作也并不相同，用访问者模式可以较好地实现该逻辑。



使用访问者模式可以将数据结构和操作分开，使得代码更加清晰、易于扩展和维护。而且，假如需要新增操作，也只需要添加相应的访问者，不需要修改原有代码，符合“开闭原则”。总体而言，访问者模式适用于大型复杂结构的数据操作，可以避免大量嵌套的条件语句。

3. 个人小结

在现代商务管理系统开发中，合理运用设计模式是非常重要的。首先，设计模式能够帮助我们更好地组织代码结构，使其更加清晰易懂，易于维护和扩展。其次，设计模式可以提高代码的复用性，使代码更加简洁高效，从而减少了开发成本和时间。最后，设计模式能够提高代码的可靠性和健壮性，有效地降低了软件出错的风险。

总之，设计模式在现代商务管理系统的开发中具有非常重要的作用。合理地运用设计模式可以使代码更加简洁高效、易于维护和扩展，并且能够提高软件的可靠性和健壮性，从而为实现现代商务管理系统提供了强有力的支持。

4. 参考文献

[1]设计模式（第二版） 清华大学出版社 刘伟

[2]Java设计模式及实践 机械工业出版社 卡玛尔米特.辛格

[3]Java设计模式 清华大学出版社 刘伟