第二篇 应用技术篇

技术本身并没有高低之分，但应用技术的思想有。一名真正的架构师，也许对于某些具体细节的了解不如一线工程师深入，也许对于某些组件背后的机制比不上骨干程序员熟悉，但对技术整体的掌控，对某些业务场景下技术的取舍和运用的策略、方法、形式、时机等方面，应该是要“高级一些”的。这些“高级货”包括但不限于设计模式、领域模型、分布式、大数据、微服务等，也包括但不限于健壮性、稳定性、可用性、易用性、可扩展性、可维护性、可操作性等。

本篇内容不仅均来自于笔者曾在一线主导开发过的项目和产品，是目前市面上同类资料中所没有的真实案例，而且极具代表性。希望本书的这部分内容，能够成为各位读者的“阿里阿德涅之线”。

第 7 章 支付系统设计模式

“支付”是现今所有互联网应用中必不可少的标配功能之一，但大多数公司做的只是简单接入第三方支付 SDK，并非真正的支付系统。

笔者借着曾经负责公司支付项目和了解专业支付公司的机会，对将设计模式应用于支付功能做了一番梳理。本章涉及到的模式包括构造器模式、策略模式、模版方法模式、工厂模式、外观模式、备忘录模式、责任链模式、迭代器模式和组合模式。

在继续进行下去之前，本章先做如下声明：

1. 支付系统涉及到的内容极为复杂，笔者只结合实际应用场景来展开设计模式；
2. 只讲核心问题，只写核心代码。

7.1 业务背景

有一家新成立不久的母婴用品电商公司，其平台的多数功能都是赶工拼凑出来的，不仅问题多多，而且还经了好几道手，健壮性、稳定性、扩展性和可维护性极差，用“命悬一线”来形容都不为过。

为此，公司的技术 BOSS 下定决心要一劳永逸地解决这些麻烦，给后续业务的高速发展铺路。BOSS 提出：现有的支付功能要升为级独立的支付系统，而且在继续开发新功能的同时，还要保质保量地满足公司各种运营的需求。

该母婴公司与支付相关的初始代码在 javabook-cloud 子项目 javabook-application 的 cn.javabook.chapter07.common 包中。

7.2 老板需求

现在，BOSS 对支付功能提出了下面的几项需求：

1. 将 APP 账户分为可用余额与押金两部分。如果可用余额不足，就从押金中扣除一部分（假设这么做不违规且经用户调研后可行）；

2. 如果可用余额 + 押金仍不足以支付，那么支付失败；

3. 业务系统在支付前和支付后需要完成不同的工作，例如支付前需要锁定账户，而支付成后要给账户增加积分；

4. APP 既有自己的钱包，也同时对接了多种不同的第三方接口，而且不同的支付接口（支付宝和微信）、不同的服务（支付和提现），接口的设置可能完全不同，这块的代码需要改善。

仔细思考并分析一下需求，可以得出下面的结论：

1. 需求 1 纯粹属于产品设计的范畴，和技术架构无关。需求 1 其实是多了个分支条件，看来也和技术升级改造扯不上关系。而需求 3，很明显是要做一个面向“切面”的拦截。也就是拦截所有支付请求，在支付前执行某些动作，支付后再完成另一些动作。

2. 至于需求 4，因为是用“if...else”的方式选择不同的支付渠道，而且每次配置属性的时候代码都比较丑陋且冗长，全是 setter 方法。

3. 很明显需求 3 和需求 4 就是需要根据业务需求来改造技术架构的点。那么，在 GoF 的 23 种设计模式中，哪几个匹配需求 4 呢？因为我们希望客户端的设置能够优雅地用一行代码搞定，就像这样：“client.setAppID(.....).setRequest(.....).setNotifyUrl(.....);”。

这其实是典型的构造器（Builder）模式。

而用来消除大量“if...else”的，非策略（Strategy）模式莫属。至少，把不同的分支条件放在不同的类中，就不会再出现多个工程师修改同一个类导致代码冲突了。而且从目前的支付代码来看，如果再增加其他支付接口，就不得不修改源代码。

至于需求 3，其实就是一个通用模板的问题，这个用模板方法（Template Method）模式就能很好解决。当然，代理模式也是可以的。

7.2.1 构造器模式

构造器模式（Builder Pattern）有时也叫生成器模式。在软件开发过程中，有时候需要创建一些复杂的对象，而这些复杂对象的初始化代码可能又要调用很多个其他的类。

如果某个汽车厂只用一条流水线来生产所有款式的汽车的话，就会像图 7-1 所示那样。

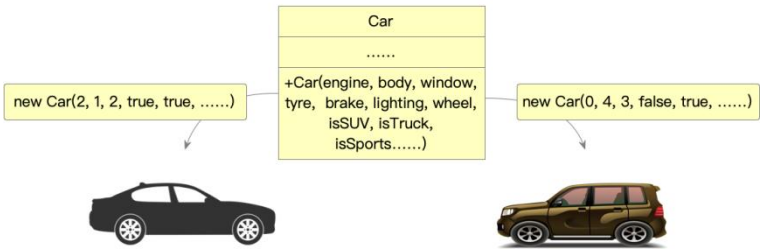

图 7-1 生产汽车的构造器

所有选项的排列组合可以生产出无数种车，但对某款车型来说，大多数的选项都是默认

且无用的。有时候软件工程师们为了偷懒，也会在一个构造器中把对象的构造参数全部加上。当代码复杂到一定程度后，这将产生不可控的复杂性。为了解决这种“参数大杂烩”问题，构造器模式出现了。它将对象的构造方法从类中抽取出来，将其放在一个名为生成器的独立对象中，然后通过一系列步骤，“定制”所需对象。

如果使用构造器的话，那么生产一辆汽车的过程就清晰多了。如图 7-2 所示。

图 7-2 用构造器生产汽车

构造器/生成器模式的类结构如图 7-3 所示。

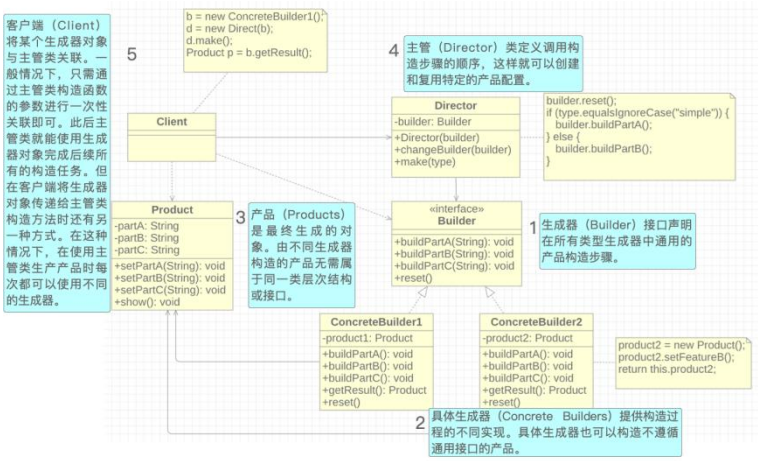

图 7-3 构造器模式类结构图

构造器/生成器模式还可以避免“重叠构造器（telescoping constructor）”的现象。比如像下面这种代码。

```
public class Foobar {
    Foo(int size) { ····· }
    Foo(int size, boolean cheese) { ····· }
    Foo(int size, boolean cheese, boolean pepperoni) { ····· }
    .....
}
```

下面就来给 Payload 增加生成器。如代码清单 7-1 所示。

代码清单 7-1 Payload.java 部分源码

```
public class Payload {
    public Payload() {}

    private Payload(Builder builder) {
        this.channel = builder.channel;
        ......
    }

    public static class Builder {
        private CHANNEL channel;
        ......
        public Builder() {}

        public Builder(CHANNEL channel, ......) {
            this.channel = channel;
            ......
        }

        public Builder setChannel(CHANNEL channel) {
            this.channel = channel;
            return this;
        }
        ......
        public Payload build() {
            return new Payload(this);
        }
    }
    ......
}
```

Payload 创建了一个内部生成器，它和 Payload 的最大不同就是它的每一个参数都有一个相应的构造器。现在用新的生成器来重写支付设置，如代码清单 7-2 所示。

代码清单 7-2 Payment.java 部分源码

```
public class Payment {
    private Payload payload;

    protected boolean pay(int type, final Account account, final Order order) {
        Builder builder = new Builder();
        ......
        if (CHANNEL.ALIPAY.ordinal() == type) {
            builder = builder.setChannel(CHANNEL.ACCOUNT)
                    .setCharge(0.0)
```

```
                .setNotifyUrl("http://localhost:9090");
        }
        payload = builder.setBody("订单备注")
                .setTradeNo("202001234567890123456789")
                .build();
        ......
    }
}
```

这样 Payload 类看起来更优雅了，而且也符合单一职责原则：将复杂的“构造行为”从业务逻辑中分离出来。只是这样可能会增加一点代码本身的复杂性。但如果这种复杂性所带来的好处能抵消不使用它的坏处，那也是值得的。

7.2.2 策略模式

至于代码中过多的“if...else”则可以用策略模式解决。

```
if (CHANNEL.ALIPAY.ordinal() == type) {
    // TODO ......
} else if (CHANNEL.WEIXIN.ordinal() == type) {
    // TODO ......
} else if (CHANNEL.ALIPAY.ordinal() == type) {
    // TODO ......
}
```

策略模式（Strategy Pattern）在 GoF 中属于行为型模式，它将不同的分支条件行为剥离到独立的类中。例如，假设某公司打算开发一款导游 APP，方便用户打卡网红景点。该 APP 需要自动规划前往目的地的路线，让游客能在 APP 地图上看到这些路线。

但程序的首个版本只有自驾路线，所以自驾游旅客对此非常满意。不过很显然，并非所有人都会开车去旅行，因此又添加了步行路线。此后，又添加了公共交通专线，而这只是个开始。据说还要添加骑行路线，特色景点路线等，如图 7-4 所示。

图 7-4 各种不同的出行“策略”

尽管从商业角度来看，项目已经交付了，但后续维护时非常头疼：无论是修复简单缺陷还是微调街道权重，或对某个算法进行修改都会影响到整个类，从而增加引入 Bug 的风险。此外，团队合作变得越来越低效，因为工程师在解决代码冲突的问题上花费了大量不必要的时间。

所以，将每种路线规划都独立出来，分别抽取到独立类中，就是自然而然的改进方法。策略模式的类结构图如图 7-5 所示。

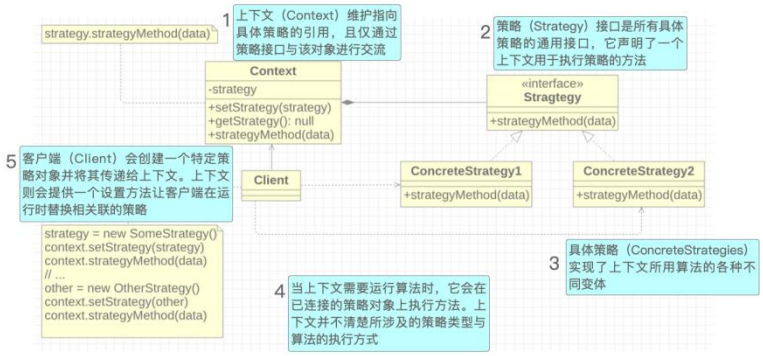

图 7-5 策略模式类结构图

现在回到支付上，使用不同的支付方式，就是典型的“策略”。

代码清单 7-3 PayStrategy.java

```
public interface PayStrategy {
    boolean pay(double amount);
}
```

代码清单 7-4 PayByAlipay.java

```
public class PayByAlipay implements PayStrategy {
    @Override
    public boolean pay(double amount) {
        System.out.println("<<<<<<<<<<< 接口设置 >>>>>>>>>>>");
        Payload.Builder builder = new Payload.Builder();
        Payload payload = builder.setChannel(Payload.CHANNEL.ALIPAY)
                .setCharge(amount * 0.006)
                .setNotifyUrl("http://localhost:7070")
                .setBody("订单备注")
                .setTradeNo("20200123456789012345678 9")
                .build();
        return true;
    }
}
```

这里仅以某一支付方式为例，完整代码在 cn.javabook.chapter07.strategy 包。

代码清单 7-5 Payment.java

```
public class Payment {
    protected PayStrategy strategy;

    protected boolean pay(final double amount) {
        return strategy.pay(amount);
    }
}
```

改造后的 Payment 支付类，瞬间少了大量的代码，而且调用的是接口，与各个具体的“策略”实现完全无关。

7.2.3 模板方法模式

现在再来解决“支付前锁定账户，支付后增加积分”。很多人都会写年终总结，像开头致辞、中间成绩、结尾感谢这些都有现成的“模板”，直接抄就好。

模板方法模式的类结构图如图 7-6 所示。

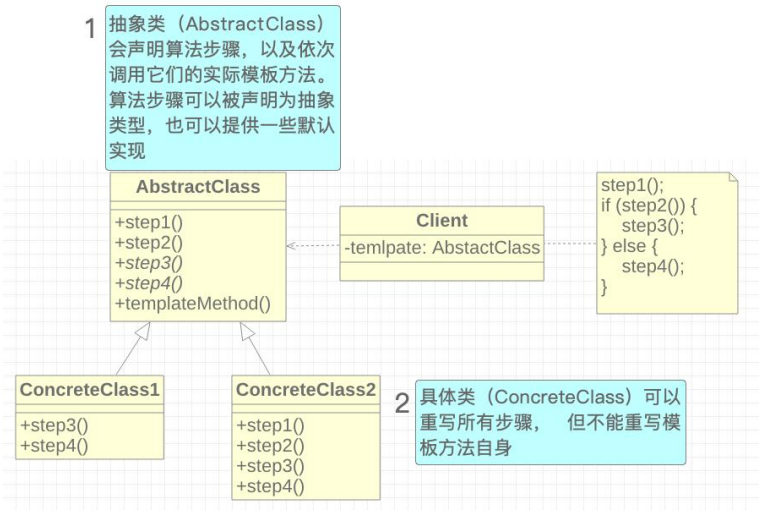

图 7-6 模板方法模式类结构图

模板方法模式特别适合解决特定步骤的扩展或优化，且无需改动整个结构。

依据模板方法模式，首先改造支付类。如代码清单 7-6 所示。

代码清单 7-6 Payment.java

```
public abstract class Payment {
    protected PayStrategy strategy;
    protected abstract void payBefore();
    protected abstract void payAfter();

    public void templateMethod(final double amount) {
        payBefore();
        pay(amount);
        payAfter();
    }
    ......
}
```

然后再创建不同支付子类继承抽象支付类，这里以余额支付为例，如代码清单 7-7 所示。

代码清单 7-7 BalancePayment.java

```
public class BalancePayment extends Payment {
    public BalancePayment(PayByBalance strategy) {
        this.strategy = strategy;
    }

    @Override
    protected void payBefore() {
        System.out.println("payBefore：余额支付前处理");
```

```
    }

    @Override
    protected void payAfter() {
        System.out.println("payAfter：余额支付后处理");
    }
}
```

至此，第一期的需求就完成了。

7.3 实现双赢

随着业务越做越大，BOSS 提出支付系统需要根据不同的结算模式，给账户返利。

1. 选择 T+1 结算方式的，给账户返利订单金额的 0.1%；
2. 选择 T+7 结算方式的，给账户返利订单金额的 0.3%。

这时候，读者是否还记得之前说过的方法——如果有很多分支条件，就应该考虑使用策略模式，所以可以继续套用它。但这一次会有不同吗？

7.3.1 策略工厂模式

按照之前的方式，将每个“if...else”都封装成单独的类是没问题的。但如果就这样直接把所有的策略都暴露出去，显然不够好。而且这种硬编码的方式仍然有很多“if...else”，也不够优雅。因此需要把策略映射成一种选择，而不是直接创建策略类。这需要把策略模式稍稍扩展一下，相对于这种扩展的策略模式，以前的策略模式笔者称呼为“简单策略模式”或者“一般策略模式”。它最大的改进，就是多了一个保存策略的 Map，如图 7-7 所示。

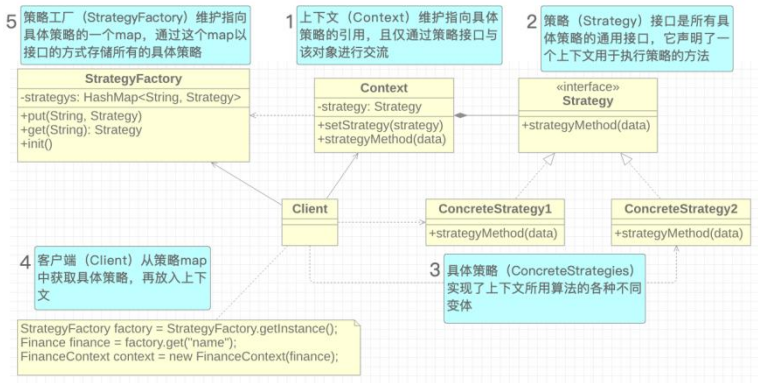

图 7-7 改进的策略模式

首先创建结算策略接口。

代码清单 7-8 Finance.java

```
public interface Finance {
    public void settleAccount(Account account, Order order);
}
```

其次以 T+1 为例创建具体策略。
代码清单 7-9 T1Finance.java

```
public class T1Finance implements Finance {
    @Override
    public void settleAccount(Account account, Order order) {
        account.setBalance(account.getBalance() + 0.001 * order.getAmount());
        System.out.println("账户已变更，T+1 返利：" + 0.001 * order.getAmount() + "元");
    }
}
```

然后再创建策略上下文，封装不同的策略。
代码清单 7-10 FinanceContext.java

```
public class FinanceContext {
    private final Finance finance;

    public FinanceContext(Finance finance) {
        this.finance = finance;
    }

    public void settleAccount(Account account, Order order) {
        this.finance.settleAccount(account, order);
    }
}
```

最后，通过策略工厂实现对策略的初始化及包装。
代码清单 7-11 StrategyFactory.java

```
public class StrategyFactory {
    private static StrategyFactory instance = null;
    private final Map<String, Finance> strategies = new HashMap<String, Finance>();
    private StrategyFactory() {}

    public static StrategyFactory getInstance() {
        if (null == instance) {
            instance = new StrategyFactory();
```

```
                return instance;
            }
            return instance;
        }
        ......
        public void init() {
            strategies.put("t1", new T1Finance());
            strategies.put("t7", new T7Finance());
        }
}
```

修改之前的 AppClient，调用策略工厂，运行后即可看到效果。

7.3.2 外观模式

修改之后没有再出现“if...else”，程序会依据传进来的参数从映射中选取合适的策略，然后再执行结算，这样就优雅多了。但即便是如此，在笔者看来，还是不完美，因为将工厂的初始化及策略包装直接暴露出来，仍然比较生硬。如果这些代码就这么丢在 main()方法里，时间长了肯定会有一些“Bad smell”出现。那有没有什么方式对这一大坨再做一次封装呢？

记得之前笔者学习设计模式的时候看过这样一段话：它创建了一个统一的类，用来包装子系统中一个或多个复杂的类，为子系统中的一组接口提供一个一致的界面。客户端可以直接通过该类来调用内部子系统的方法，从而让客户和子系统之间避免了紧耦合。这段解释有些难懂，简单来说就是：将一系列复杂的操作步骤用一个简单的类封装，操作这个类也就是在操作这些复杂的步骤——这就是外观模式（Facade Pattern），它有时也被称为门面模式。

外观模式是一种结构型设计模式，能为程序库、框架或其他复杂的类提供一个简单的调用接口。比如，如果人们在网上购物时，需要自己去找商家、找仓库、找物流、找售后、自己交税而且还要承担质量风险，恐怕谁都不会去体验这种糟糕的服务，但电商把这些事情都干了，如图 7-8 所示。

图 7-8 电商就是“门面”

所以，如果需要与包含几十种功能的第三方库整合，那么利用外观模式来屏蔽复杂性，就是一个明智且优雅的处理方式。外观模式的类结构如图 7-9 所示。

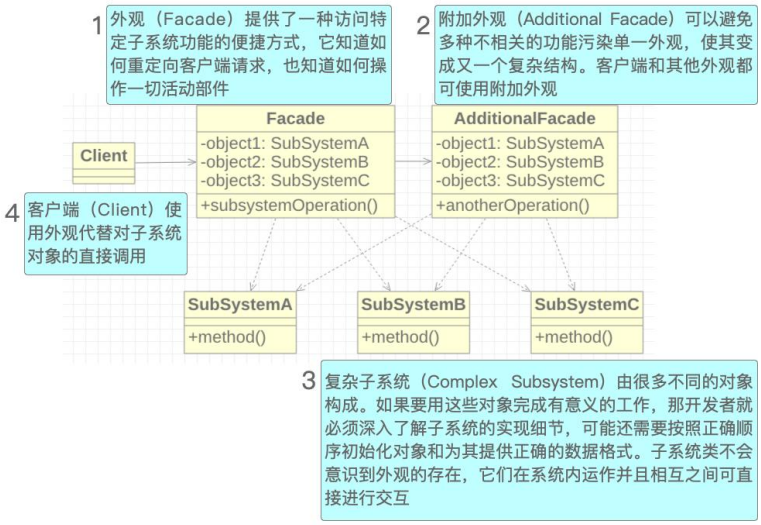

图 7-9 外观模式类结构图

有好处就肯定有坏处：外观模式一不小心就会泛滥成灾，尤其是当 Facade 成为和所有类都紧耦合的“上帝对象”时（所谓上帝对象，就是一个操了太多心的对象）。

外观模式非常简单，如代码清单 7-12 所示。

代码清单 7-12 FinanceFacade.java

```
public class FinanceFacade {
    public static void doSettle(Account account, Order order) {
        StrategyFactory factory = StrategyFactory.getInstance();
        factory.init();
        FinanceContext context = new FinanceContext(factory.get(account.getAccid()));
        context.settleAccount(account, order);
    }
}
```

运行 cn.javabook.chapter07.facade.AppClient 类就可以优雅地调用它了。

7.4 产品发难

当用构造器模式、策略模式、模板方法模式和外观模式满足了 BOSS 的需求后，产品经理在新的产品需求评审会上，对支付系统也提出了如下需求：

1. 交易完成后，要给用户发送消息通知，通知用户交易进度等消息；
2. 交易必须要做到不可抵赖，需要有快照作为证据来保存相关交易数据；
3. 支付系统需要根据订单的不同状态来决定下一步可以执行的动作。比如只有待支付订单才能付款、已取消/已超时/已完成订单就不能再支付、已支付的订单不能重复支付等。

7.4.1 观察者模式

某件事情做完以后，如果需要告知对方结果的话，要么当面告知，要么打对方电话告知。这种事后通知的方法在开发中叫做“回调”，它在设计模式中就是观察者模式。比如当舞台上演员的动作发生变化时（有结果），观众就能马上看到（回调）。那么演员就是发布者（Publisher），而观众就是订阅者（Subscriber），有的地方也叫它观察者（Observer）。

观察者模式是一种发布-订阅机制，它让每个对象都能订阅或取消订阅发布者的事件流。它实际上包括两种东西：

1. 用于存储订阅者对象的列表；
2. 用于添加/删除列表中订阅者的方法。

所以，不管发生什么，发布者都会遍历订阅者并调用它的特定通知方法。观察者模式的类结构图如图 7-10 所示。

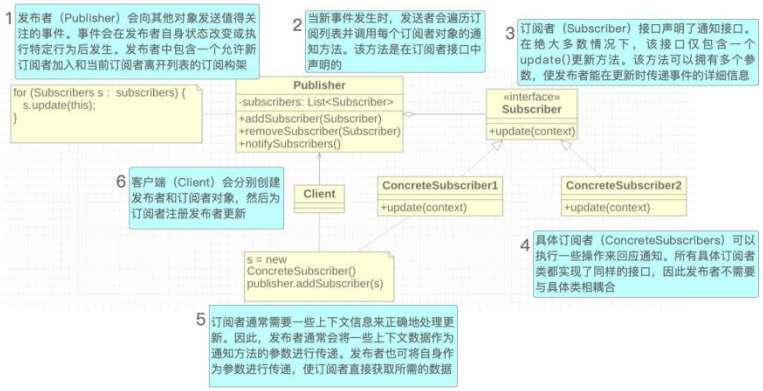

图 7-10 观察者模式类结构图

所以，要实现交易完成后给用户发送通知，就需要先定义订阅者（Subscriber）接口。

代码清单 7-13 Subscriber.java

```
public interface Subscriber {
    public void payUpdate(String channel);
    public void pointUpdate(String channel, int point);
}
```

然后修改用户账户类，让它实现订阅者接口。

代码清单 7-14 Account.java 部分源码

```
public class Account implements Subscriber {
    ......
    @Override
    public void payUpdate(String channel) {
        System.out.println("使用 " + channel + " 支付成功");
    }
```

```
    @Override
    public void pointUpdate(String channel, int point) {
        System.out.println(channel + " 后积分增加 " + point + " 分");
    }
    ......
}
```

接着修改支付抽象类，增加发布者（Publisher）相关方法。

代码清单 7-15 Payment.java 部分源码

```
public abstract class Payment {
    ......
    protected Vector<Subscriber> subscribers = new Vector<>();
    ......
    public void addSubscriber(Subscriber subscriber) {
        subscribers.add(subscriber);
    }

    public void removeSubscriber(Subscriber subscriber) {
        subscribers.remove(subscriber);
    }

    public void notifySubscribers(String channel, int point) {
        for (Subscriber subscriber : subscribers) {
            subscriber.payUpdate(channel);
            subscriber.pointUpdate(channel, point);
        }
    }
}
```

Payment 抽象类增加了 subscribers 列表，并通过 addSubscriber()、removeSubscriber()、notifySubscribers()这三个方法实现了注册、取消和通知订阅者。现在来修改 AppClient 中的支付方法，如代码清单 7-16 所示。完整代码在 cn.javabook.chapter07.observer 包。

代码清单 7-16 AppClient.java 部分源码

```
public class AppClient {
    ......
    private static void pay(Account account, double amount) {
        System.out.print("请选择支付类型：");
        String type = getInput();
        switch (type) {
            ......
```

```
            case "yue":
                ......
                balancePayment.addSubscriber(account);
                balancePayment.notifySubscribers("余额支付", 20);
                break;
            default:
                break;
        }
    }
    ......
}
```

给 AppClient 的 pay()方法增加了通知行为用于向用户发送消息，然后可以调用 main()方法看一下效果。

7.4.2 备忘录模式

为了防止交易系统崩溃或者交易双方发生纠纷，显然需要一个能够“还原现场，无法抵赖”的功能，也就是要在用户支付完成后立即保存一份交易快照。有的读者可能会觉得，直接存一份当时交易场景相关的数据不就行了吗？这确实可以，不过有两个问题：

1. 用户支付后交易并未就此结束，而是还有发货、物流、签收、退换货和评价等流程。如果没有一个统一的机制会非常混乱，因为不同的人做法肯定会有很大的不同；

2. 如果把所有相关字段、属性都集中在一起的话，这个类将会非常庞大臃肿。

在 GoF 的设计模式中，有一种专门针对快照存储的模式，叫做“备忘录模式”。这个模式不太好理解。它的机制简单来说就是：

1. 将创建快照的工作委派给业务对象自己，比如交易中的账户和订单；

2. 然后由原发器来调用业务对象创建交易快照的方法；

3. 之后原发器（Originator）将创建的快照存储在备忘录（Memento）对象中；

4. 最后将备忘录保存在负责人（Caretaker）对象中。但这个 Caretaker 对象和备忘录的交互是受限的，无法修改备忘录内容，只有 Originator 可以。

备忘录模式其实就类似于航空事故的调查和处理过程，如图 7-11 所示。

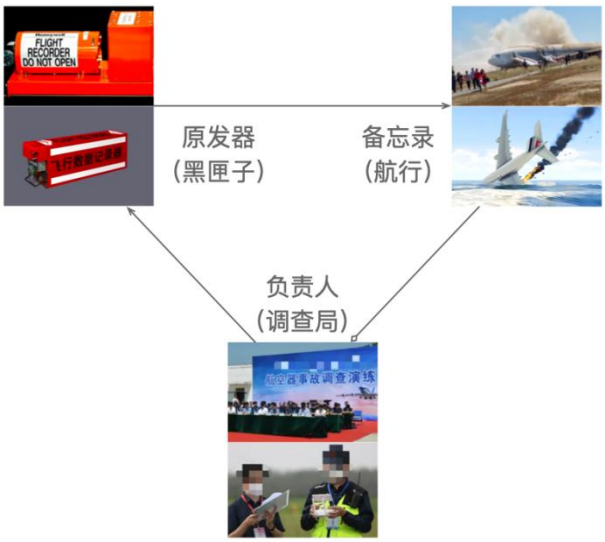

图 7-11 民调查航中的“备忘录模式”

只有航行中的飞机才能产生“快照”（也就是备忘录），也只有黑匣子（相当于原发器）才能“记录快照”。而民航局的事故调查机构（也就是负责人）虽然可以采用黑匣子里面的“快照”数据，但却不能修改和新增“快照”，只能通过黑匣子实现，这就是备忘录的工作原理。备忘录模式的类结构图如图 7-20 所示。

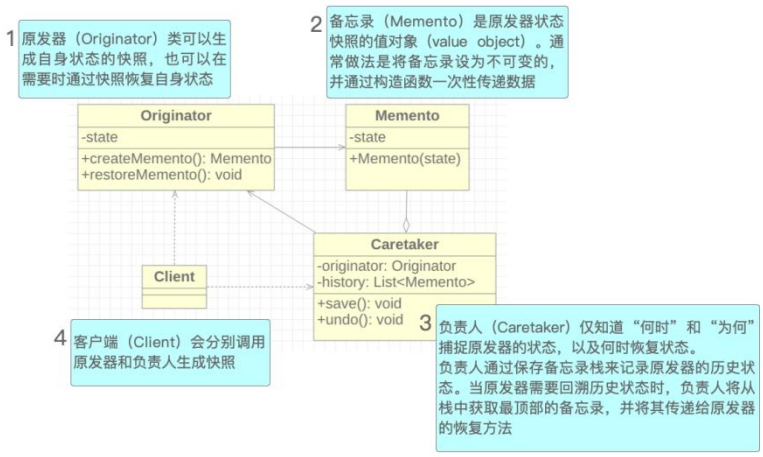

图 7-12 备忘录模式类结构图

首先创建备忘录。

代码清单 7-17 Memento.java

```
public class Memento {
    private String snapshot = "";

    public Memento(String snapshot) {
        this.snapshot = snapshot;
    }
    ......
}
```

接着创建原发器。

代码清单 7-18 Originator.java

```
public class Originator {
    private String snapshot = "";
    ......
    public Memento createMemento() {
        return new Memento(snapshot);
    }

    public void restoreMemento(Memento memento) {
        this.setSnapshot(memento.getSnapshot());
    }
}
```

然后指定负责人。

代码清单 7-19 Caretaker.java

```
public class Caretaker {
    private Originator originator;
    private LinkedList<Memento> history = new LinkedList<>();

    public Caretaker(Originator originator) {
        this.originator = originator;
    }

    public void save() {
        Memento memento = originator.createMemento();
        history.add(memento);
        System.out.println("创建备忘录并保存至历史快照列表");
    }

    public void undo() {
        if (0 < history.size()) {
            Memento memento = history.getLast();
            originator.restoreMemento(memento);
            history.removeLast();
        }
    }
}
```

最后在 AppClient 中实现交易快照。完整代码在 cn.javabook.chapter07.memento 包。可以改变一下 Order 的属性值，然后再使用负责人的 undo()方法试试。感兴趣的读者可以自己动手，把快照数据保存到数据库，然后再从数据库恢复。

7.5 技术重构

经常使用电商的读者对这样的体验肯定不陌生：

1. 已下单的订单后续步骤只能是取消和继续支付二选一；
2. 已取消的订单除了删除什么都做不了；
3. 已支付的订单不能被取消，后续状态只能是已揽件；
4. 只有已揽件的订单才能确认收货或已拒收；
5. 只有确认收货的订单才能评价、申请退换货、删除和再来一单；
6. 已评价的订单除了删除还可以追加评价。

整个订单流转状态大致如图 7-13 所示。

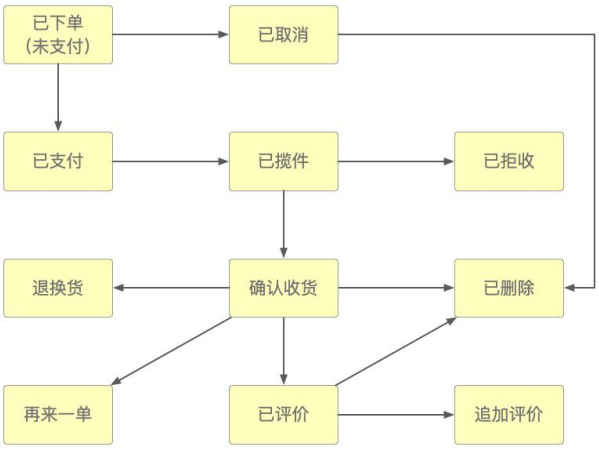

图 7-13 订单状态流转

电商公司早期实现订单状态流转的代码在 cn.javabook.chapter07.state.old 包的 Context 类中。虽然业务逻辑已经实现，但很明显还存在如下问题：

1. 操作订单的每一步都要 switch 判断，重复代码太多；
2. 扩展性很差，如果再来一种状态，所有的 switch 都要改；
3. 很多时候，交易系统的每一步都取决于订单的状态是否满足执行条件，如果在每个地方都去判断，无疑是非常繁琐冗长的。

经过技术评审后，大家一致认为 GoF 设计模式中的状态模式可以消除这种“Bed Smell”。这里为了达到演示的效果，同时又避免陷入太多细节，所以只简化并解决订单流转状态列表的前四个：

1. 已下单的订单后续步骤只能是取消和继续支付二选一；
2. 已取消的订单什么都做不了；
3. 已支付的订单不能被取消，后续状态只能是确认收货；
4. 已完成的订单什么都做不了。

GoF 对状态模式的理解是：状态模式与有限状态机的概念紧密相关，它在任意时刻仅可

处于几种有限的状态中。在任何一个特定状态，程序的行为都不相同，且可瞬间从一个状态切换到另一个状态。总的来说，状态模式的核心做法是为对象的每一个可能的状态都新建一个类，然后将所有状态的对应行为抽取到这些类中。状态模式的类结构如图 7-14 所示。

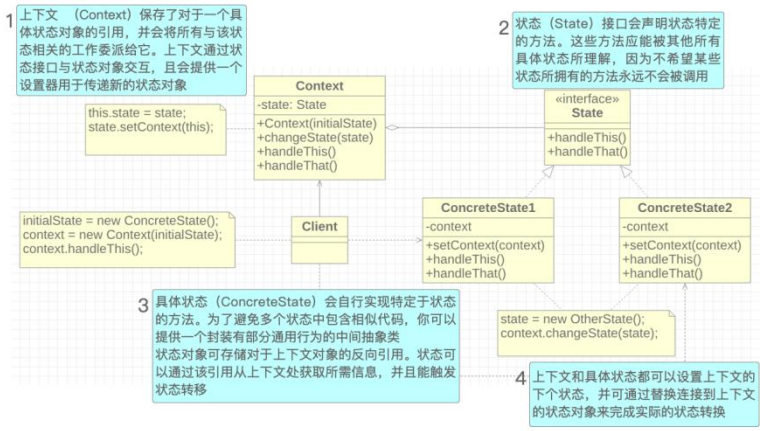

图 7-14 状态模式类结构图

按照状态模式的要求，首先定义状态抽象类。

代码清单 7-20 State.java

```
public abstract class State {
    protected final Context context;

    public State(Context context) {
        this.context = context;
    }

    public abstract String onOrdered();
    public abstract String onCancel();
    public abstract String onPaid();
    public abstract String onConfirm();
}
```

然后创建订单上下文。

代码清单 7-21 Context.java

```
public class Context {
    private State state;
    private boolean ordered = false;

    public Context() {
        this.state = new OrderedState(this);
    }
```

```
    ......
}
```

然后为每种状态都创建一个与之对应的类，并把相关的行为都抽取到类中。完整代码在 cn.javabook.chapter07.state.news 包。可运行 cn.javabook.chapter07.state.news.UI 类进行测试，界面如图 7-15 所示。

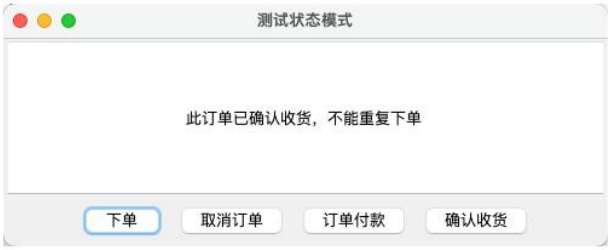

图 7-15 状态流转测试结果

7.6 运营改进

最近公司业务的发展蒸蒸日上，所以运营部门也提出了如下需求：

1. 用户（也就是入驻平台的中小卖家）要有提现功能，但是对提现额度有规定：

1）提现≤2000 元无需审批；

2）提现≤10000 需公司风控部门审批；

3）提现≤100000 需公司副总裁审批，同时要提供身份证明；

4）提现≤1000000 需当面到公司线下的门店柜台交易。

2. 出于质量及安全要求，需要对某些顾客购买的商品进行检验，其实就是是逐一检视；

3. 需要打印出平台顾客购买的商品小票。

7.6.1 责任链模式

仔细思考一下提现需求可以发现，它其实还隐含着一个“审批”流程在其中，也就是提现办理依次从宽到严，逐层升级。再结合整个 GoF 设计模式来看，这种能够同时包含多个“if...else”条件语句和链条结构的，除了责任链模式，就没有第二家了。

作为一种行为设计模式，责任链模式可以将请求沿着处理者链进行发送。收到请求后，每个处理者均可对请求进行处理，或将其传递给链上的下一个处理者。比如，用户可以在平台上下单购买商品，而且后台运营人员也有权限访问用户创建的订单。如果后续需要增加一些订单验证步骤，比如同一个 IP 一天内只能下单一次、通过规则引擎判断订单是否存在刷单嫌疑、判断多级缓存中的订单是否一致等。这样，每次新增一些看似必要的功能都会让代码变得更加臃肿混乱，维护困难。

经典的责任链模式类结构图如图 7-16 所示。

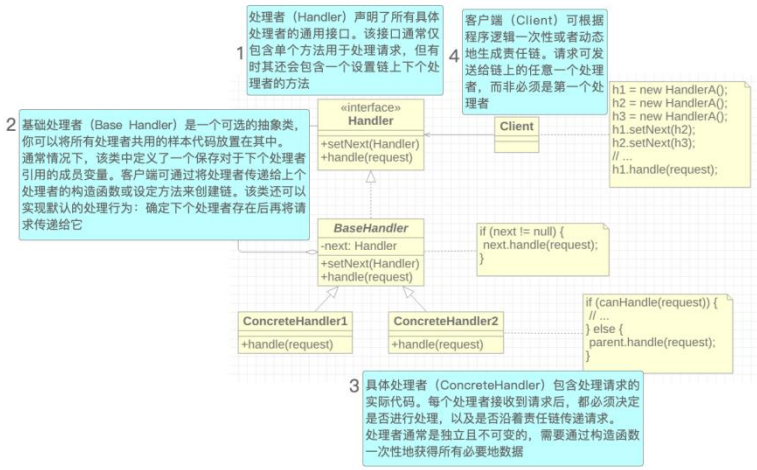

图 7-16 经典的责任链模式

但在这家电商公司中的支付系统中，笔者稍稍做了一些简化。如图 7-17 所示。

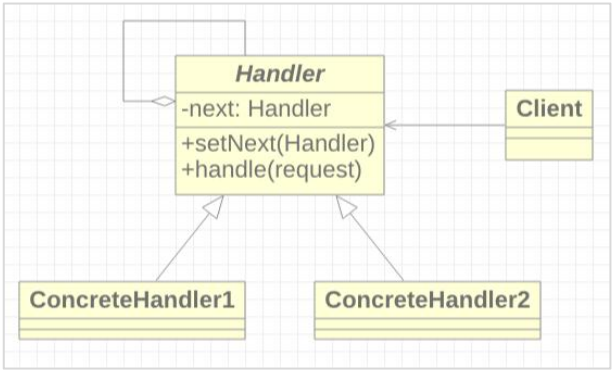

图 7-17 简化的责任链模式

上图去掉了接口，只保留了抽象类。当然也可以反过来，完全依据自己的理解和习惯而定。下面就按照简化后的责任链模式来实现第一个需求。

首先定义 Handler 抽象类，如代码清单 7-22 所示。

代码清单 7-22 WithdrawHandler.java

```
public abstract class WithdrawHandler {
    private WithdrawHandler next;
    ......
    public abstract void handle(double amount);
}
```

再依次实现无需审核的请求、由风控部门审核的请求、由公司副总裁审核的请求以及由线下柜台当面审核的请求，完整代码在 cn.javabook.chapter07.chain 包。最后在 AppClient 中增加 withdraw()方法，如代码清单 7-23 所示。

代码清单 7-23 AppClient.java 部分源码

```
public class AppClient {
    ......
    private static void withdraw(double amount) {
        WithdrawHandler handler1 = new NoAuditHandler();
        WithdrawHandler handler2 = new RiskDeptHandler();
        WithdrawHandler handler3 = new ViceHandler();
        WithdrawHandler handler4 = new BankHandler();
        handler1.setNext(handler2);
        handler2.setNext(handler3);
        handler3.setNext(handler4);
        handler1.handle(amount);
    }
    ......

    public static void main(String[] args) {
        ......
        withdraw(order.getAmount());
        ......
    }
}
```

运行代码后分别输入需要不同审核级别的金额，然后看看效果。

7.6.2 迭代器模式

用户下单购买的商品清单就是一个列表，而检查商品就需要遍历列表中的每个元素。最容易想到的方法是直接用 for 循环，但它能行吗？

假设有多种类型的集合，有的是列表，有的是堆栈，而有的又是树。如果集合基于列表，那么这项任务很简单。但如果是树呢？例如，今天需要使用深度优先算法来遍历树结构，但明天可能就会需要广度优先算法，下周则可能会需要其他方式（比如随机存取树中的元素）。

这种将不同数据结构和行为耦合在一起的结果，会带来职责的臃肿和效率的降低。所以，基于将数据和遍历数据的行为进行分离的想法，就产生了迭代器模式。

迭代器模式的类结构图如图 7-18 所示。

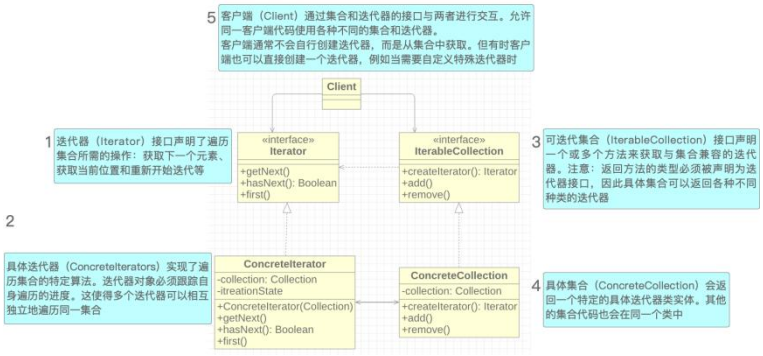

图 7-18 迭代器模式类结构图

首先定义迭代器接口。

代码清单 7-24 Iterator.java

```
public interface Iterator {
    public Product getFirst();
    public Product getNext();
    public boolean hasNext();
}
```

然后用具体迭代器实现它。

代码清单 7-25 ConcreteIterator.java

```
public class ConcreteIterator implements Iterator {
    private List<Product> list = null;
    private int index = -1;

    public ConcreteIterator(List<Product> list) {
        this.list = list;
    }

    @Override
    public Product getFirst() {
        index = 0;
        return list.get(index);
    }

    @Override
    public Product getNext() {
        Product product = null;
        if (this.hasNext()) {
            product = list.get(++index);
        }
```

```
        return product;
    }

    @Override
    public boolean hasNext() {
        return index < list.size() - 1;
    }
}
```

因为商品都是放在购物车中的，所以购物车也是一个商品的集合。定义购物车接口，如代码清单 7-26 所示。

代码清单 7-26 Cart.java

```
public interface Cart {
    public void add(Product product);
    public void remove(Product product);
    public Iterator createIterator();
}
```

然后是购物车的具体实现类以及产品类，完整代码在 cn.javabook.chapter07.iterator 包，最后修改 AppClient 的 main()方法并运行。

虽然迭代器模式看起来和 List 中的 iterator()方法没什么不同，但其本质差别在于一个是具体的工具，一个是设计思想。这种设计思想可以让代码遍历不同的甚至是无法预知的数据结构，并且也有利于对客户端隐藏交互细节。如果数据结构只有集合且也无扩展性时，那还是 iterator()方法更简单直接。

7.6.3 组合模式

现在就剩“打印出平台顾客购买的商品小票”这个需求了。我们去超市买完东西之后，都会收到打印出来的小票，就是商品清单、价格、数量和汇总的信息。

在电商里面，这种“购物小票”换了个马甲，叫“发货单”，如图 7-19 所示。

序号	产品名称	规格型号	单位	数量	备注
1					
2					
3					
合计： 件					
收货单位：					
收货人：		电话：			
发货人：		电话：			
发货日期： 年 月 日		收货日期： 年 月 日			

图 7-19 电商发货单

不同的平台可能会有不同的样式，但本质仍然是商品清单、价格、数量和汇总的信息。

假设现在某用户在平台购买了一些商品，这些商品都是来自于不同商家。如果此时平台要给用户推送一个发货单的消息，那么这个发货单的数据该怎么“造”出来呢？拿线下场景作比方的话，过程应该会是这样的：

1. 顾客买的东西会被放到购物手推车中；
2. 顾客把购物车推到收银台，收银台逐个扫码然后打印出商品名称、单价及总价；
3. 收银员将商品分别放进不同的塑料袋中。

换成线上电商购物过程的话就是：

1. 用户订单会被自动拆分成不同商家的子订单（也就是订单拆分过程）；
2. 平台会为这些子订单生成不同的商家发货单；
3. 平台抽取出商品的名称、单价及总价，形成统一的、和商家信息无关的平台发货单。

平台推送的“购物小票”消息的内容可以通过合并这些子订单的数据得到，也就是对全部子订单进行循环计数，然后再合起来计算总价。

通过分析可以知道，不管是线下“购物车-购物袋”，还是线上“平台订单-商家子订单”，本质上都是“整体-部分”的关系，对这种组合或者复合模式的处理，GoF 设计模式中有一种专门的组合模式，它让单个对象和组合对象具有一致的访问性。

组合模式在 GoF 设计模式中是一种结构型模式，顾名思义，它可以将对象“组装”起来，并且能像使用一个单独的对象一样使用这个组合。就拿前面的例子来说，假设订单中可以有无包装的简单产品，也可以有产品包装盒，那么组合而成的对象应该如图 7-20 所示。

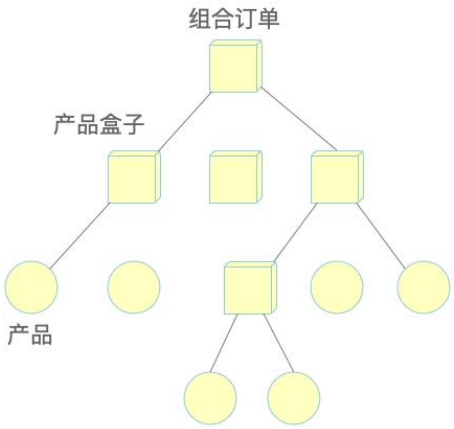

图 7-20 产品组合

组合模式建议使用一个通用接口来与“产品”和“盒子”进行交互，并且在该接口中声明计算总价的方法。该方式的最大优点在于无需了解具体类，也无需了解对象是简单的产品还是复杂的盒子，只需通过接口以相同的方式对其进行处理。组合模式的类结构图如图 7-21 所示。

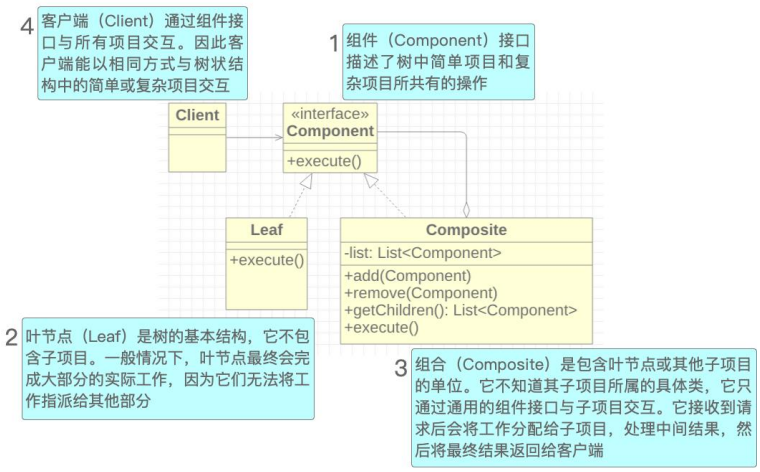

图 7-21 组合模式类结构图

先定义组合接口。

代码清单 7-27 OrderAction.java

```
public interface OrderAction {
    public double calculation();
    public void receipt();
}
```

再定义“叶子”对象。

代码清单 7-28 Product.java 部分源码

```
public class Product implements OrderAction {
    private String name;
    private int quantity;
    private double price;
    ......
    @Override
    public double calculation() {
        return price * quantity;
    }

    @Override
    public void receipt() {
        System.out.println(name + "(数量： " + quantity + "， 单价： " + price + "元)");
    }
}
```

然后用组合对象实现接口。

代码清单 7-29 SubOrder.java

```
public class SubOrder implements OrderAction {
    private String name = "";
    private List<OrderAction> suborder = new ArrayList<>();

    public SubOrder(String name) {
        this.name = name;
    }

    public void add(OrderAction things) {
        suborder.add(things);
    }

    @Override
    public double calculation() {
        double total = 0;
        for (OrderAction action : suborder) {
            total += action.calculation();
        }
        return total;
    }

    @Override
    public void receipt() {
        for (OrderAction action : suborder) {
            action.receipt();
        }
    }
}
```

最后运行 cn.javabook.chapter07.component.AppClient 类，打印出“购物小票”。

7.7 本章小节

设计模式是软件开发中对业务逻辑代码的高度抽象和总结，本章用十多种 GoF 设计模式分别实现了老板、产品经理、技术经理和运营部门提出的各种需求。

这并不是 GoF 设计模式的全部，而且即使是这十多种模式，也仅仅只实现了支付系统中非常非常小的一部分功能，还有大量的业务逻辑都可以用设计模式来改造。