## 第24章 购买手机选项多，如何选购是难题——策略模式

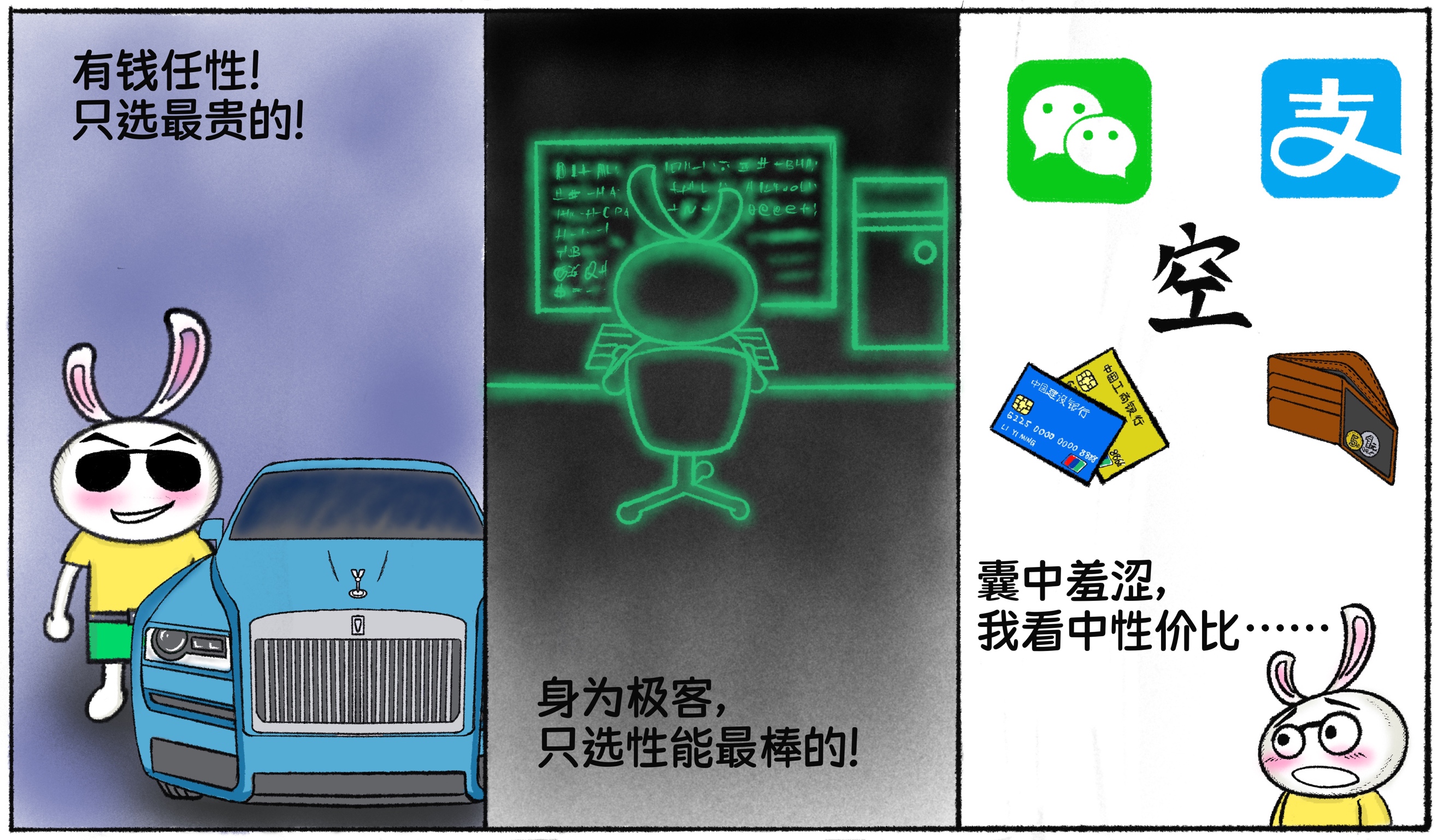
## 24.1 如何挑选一部手机

兔小白：熊小猫，我的手机屏幕摔坏了，听说你对手机很了解，能不能给我推荐一部？

熊小猫：换个屏幕就可以了，不需要换手机吧？

兔小白：嘿嘿，其实我早就想换手机了，已经研究了一段时间，但是种类太多，已经挑花了眼。你给我点建议？

熊小猫：没问题！但是我得知道你的需求是什么，我才知道用哪种策略给你推荐。



兔小白：我更倾向于选择性价比高的手机。不过你还是按照不同的策略都给我推荐一部吧。我再综合考虑。

熊小猫：你这是把我推荐的工作量翻了3倍呀！我得按照3种不同的策略帮你选手机。

兔小白：我纠结好几天了，帮帮我吧......

熊小猫：这样吧，咱们用程序来解决手机推荐的问题。我可以指导你开发程序。

兔小白：一个小问题，不至于开发程序来解决吧吧？

熊小猫：程序是可以复用的，类似的选择问题在生活中十分常见。另外，这个问题虽然不大，但是程序设计却值得思考哦！

兔小白：听你这样说，我倒是很有兴趣挑战一下！

## 24.2 简单工厂模式实现手机推荐程序

熊小猫：咱们就根据你挑选手机的需求来开发程序。这里有3种不同的手机推荐策略，不同的推荐策略，算法不同。具体的推荐策略如下。

1. 选择价格最高的手机。
2. 选择性能最好的手机
3. 选择性价比最高的手机。性价比的计算规则为手机性能得分除以价格。

兔小白：这个需求的场景似曾相识，我先想想如何设计程序。

20分钟后，兔小白写完了程序。

兔小白：我就说这个场景很熟悉嘛！我先根据推荐手机所需的参数定义了手机类。

public class Mobile {  
    private final String name;  
    private final double price;  
    private final double performanceScore;  
​  
    public Mobile(String name, int price, int performanceScore) {  
        this.name = name;  
        this.price = price;  
        this.performanceScore = performanceScore;  
   }  
​  
    public String getName() {  
        return name;  
   }  
​  
    public double getPrice() {  
        return price;  
   }  
​  
    public double getPerformanceScore() {  
        return performanceScore;  
   }  
}

之前讲的简单工厂模式，正是适用于推荐手机场景。在程序中，先定义推荐策略接口，然后再实现3个推荐策略类。

RecommendStrategy为策略接口，定义推荐手机的方法。

public interface RecommendStrategy {  
    Mobile recommendMobile(List<Mobile> mobiles);  
}

该接口有三种不同推荐策略的实现类。HighestPriceStrategy为最高价格推荐策略实现类。

public class HighestPriceStrategy implements RecommendStrategy {  
    public Mobile recommendMobile(List<Mobile> mobiles) {  
        Mobile recommendMobile = null;  
​  
        for (Mobile mobile : mobiles) {  
            if (recommendMobile == null) {  
                recommendMobile = mobile;  
           } else if (mobile.getPrice() > recommendMobile.getPrice()) {  
                recommendMobile = mobile;  
           }  
       }  
        
        return recommendMobile;  
   }  
}

HighestPerformanceStrategy为最优性能推荐策略实现类。

public class HighestPerformanceStrategy implements RecommendStrategy {  
    public Mobile recommendMobile(List<Mobile> mobiles) {  
        Mobile recommendMobile = null;  
​  
        for (Mobile mobile : mobiles) {  
            if (recommendMobile == null) {  
                recommendMobile = mobile;  
           } else if (mobile.getPerformanceScore() >  
                    recommendMobile.getPerformanceScore()) {  
                recommendMobile = mobile;  
           }  
       }  
​  
        return recommendMobile;  
   }  
}

CostPerformanceStrategy为最优性价比推荐策略实现类。

public class CostPerformanceStrategy implements RecommendStrategy {  
    public Mobile recommendMobile(List<Mobile> mobiles) {  
        Mobile recommendMobile = null;  
​  
        for (Mobile mobile : mobiles) {  
            if (recommendMobile == null) {  
                recommendMobile = mobile;  
           } else {  
                double highestScore = recommendMobile.getPerformanceScore()  
                        / recommendMobile.getPrice();  
                double score = mobile.getPerformanceScore() / mobile.getPrice();  
                if (score > highestScore) {  
                    recommendMobile = mobile;  
               }  
           }  
       }  
​  
        return recommendMobile;  
   }  
}

RecommendStrategyFactory是推荐策略工厂，负责创建推荐策略实例。

public class RecommendStrategyFactory {  
    public RecommendStrategy createStrategy(String category) throws Exception {  
        switch (category){  
            case "price":  
                return new HighestPriceStrategy();  
            case "performance":  
                return new HighestPerformanceStrategy();  
            case "costPerformance":  
                return new CostPerformanceStrategy();  
            default:  
                throw new Exception();  
       }  
   }  
}

在客户端代码中，使用工厂创建推荐策略对象，用不同的推荐策略对象进行推荐。

List<Mobile> mobiles = new ArrayList<>();  
mobiles.add(new Mobile("Huawei", 3200, 5100));  
mobiles.add(new Mobile("Xiaomi", 2500, 4800));  
mobiles.add(new Mobile("Vivo", 3000, 5200));  
​  
RecommendStrategyFactory recommendStrategyFactory =  
  new RecommendStrategyFactory();  
​  
RecommendStrategy highestPriceStrategy =   
  recommendStrategyFactory.createStrategy("price");  
Mobile mobile = highestPriceStrategy.recommendMobile(mobiles);  
System.out.println("推荐的最具性价比手机是" + mobile.getName());  
​  
RecommendStrategy highestPerformanceStrategy =   
  recommendStrategyFactory.createStrategy("performance");  
mobile = highestPerformanceStrategy.recommendMobile(mobiles);  
System.out.println("推荐的性能最优手机是" + mobile.getName());  
​  
RecommendStrategy costPerformanceStrategy =   
  recommendStrategyFactory.createStrategy("costPerformance");  
mobile = costPerformanceStrategy.recommendMobile(mobiles);  
System.out.println("推荐的价格最高手机是" + mobile.getName());

从程序输出可以看到，每种推荐策略都按照各自的逻辑推荐了一部手机。[[1]](#footnote-1)

推荐的最具性价比手机是Huawei  
推荐的性能最优手机是Vivo  
推荐的价格最高手机是Xiaomi

简单工厂是你最早给我讲的设计模式，我掌握的还算熟练吧？

熊小猫：使用简单工厂确实也能解决这个问题。但我的目的肯定不是让你复习简单工厂。

## 24.3 加入推荐人的手机推荐程序

熊小猫：程序设计需要贴合真实场景，这样的设计才符合直觉。无论自己开发还是别人阅读代码，都容易的多。我们回归到真实的场景，你想一想，给你推荐手机的是推荐策略吗？

兔小白：推荐策略是算法实现，给我推荐手机的是你。

熊小猫：没错，是我运用不同的策略来给你推荐手机。但现在程序的写法，好比你拿到了三种推荐策略的文档，先研究清楚每种推荐策略，然后再自己做选择。但真实情况是，我作为推荐人，你只需要告诉我你的需求，我就可以按照不同的需求给你推荐手机。

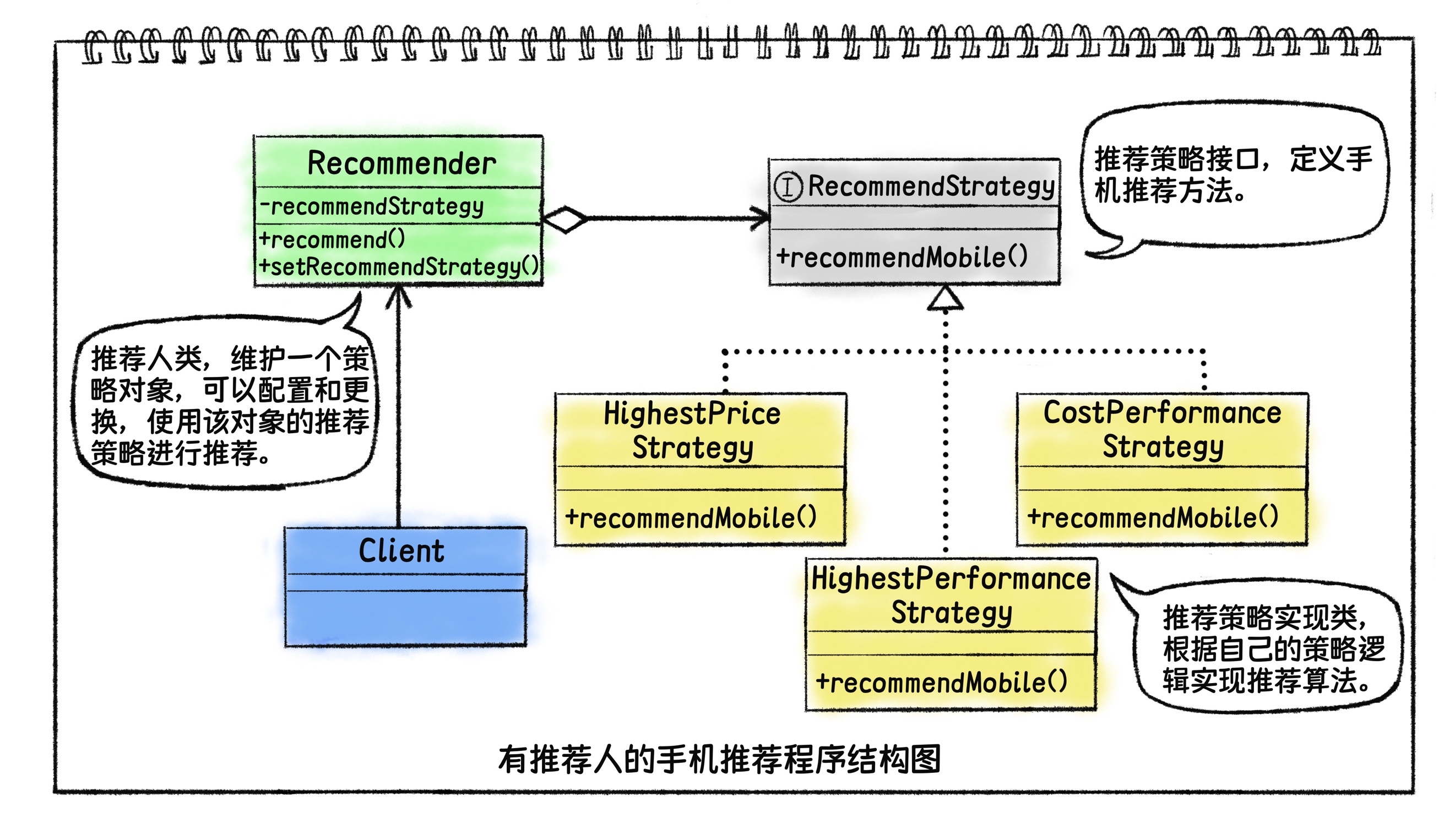
兔小白：你的意思是我应该定义推荐人对象？客户端应该和推荐人对象打交道，而不是直接和推荐策略打交道。

熊小猫：这样做不但符合真实场景，而且符合迪米特法则。客户端并不需要和那么多的推荐策略“对话”。推荐人需要留有切换策略的接口，可以按需要改变自己的推荐策略。客户端和推荐策略之间加入推荐人，也是一种代理的思想。

兔小白：代理的思想真是无处不在。我再想想怎么优化。

15分钟后，兔小白优化完了程序。

兔小白：策略接口和三个策略实现类没有任何修改。我增加了推荐人Recommender类。我们先看看程序的结构图。



Recommender维护一个策略对象，可以配置和更换，使用该对象的推荐策略进行推荐。

public class Recommender {  
    private RecommendStrategy recommendStrategy;  
​  
    public Recommender(RecommendStrategy recommendStrategy) {  
        this.recommendStrategy = recommendStrategy;  
   }  
​  
    public void setRecommendStrategy(RecommendStrategy recommendStrategy) {  
        this.recommendStrategy = recommendStrategy;  
   }  
​  
    public Mobile recommend(List<Mobile> mobiles) {  
        return recommendStrategy.recommendMobile(mobiles);  
   }  
}

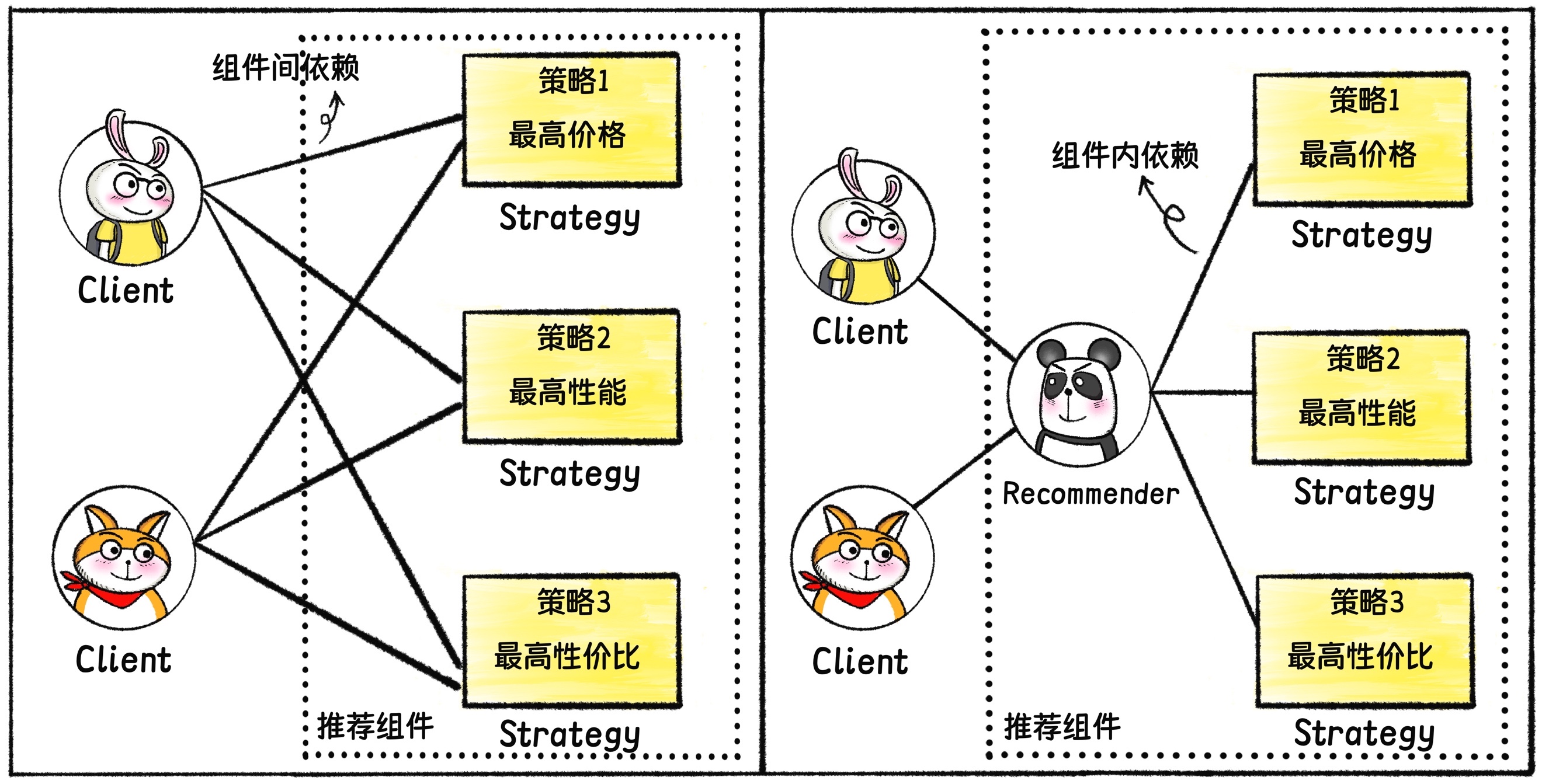
在客户端代码中，创建Recommender对象，使用该对象进行推荐。客户端可以通过Recommender对象的setRecommendStrategy方法切换推荐策略。

List<Mobile> mobiles = new ArrayList<>();  
mobiles.add(new Mobile("Huawei", 3200, 5100));  
mobiles.add(new Mobile("Xiaomi", 2500, 4800));  
mobiles.add(new Mobile("Vivo", 3000, 5200));  
​  
Recommender panda = new Recommender(new CostPerformanceStrategy());  
Mobile mobile = panda.recommend(mobiles);  
System.out.println("推荐的最具性价比手机是" + mobile.getName());  
​  
panda.setRecommendStrategy(new HighestPerformanceStrategy());  
mobile = panda.recommend(mobiles);  
System.out.println("推荐的性能最优手机是" + mobile.getName());  
​  
panda.setRecommendStrategy(new HighestPriceStrategy());  
mobile = panda.recommend(mobiles);  
System.out.println("推荐的价格最高手机是" + mobile.getName());

熊小猫：优化的不错，正是我想看到的代码。现在客户端只需要和Recommender对象“对话”，Recommender按照客户端设置的Strategy进行推荐。在原来的代码中，客户端需要和三个Strategy对象直接“对话”。

兔小白：但是Recommender还是需要和3个Strategy对象“对话”呀！它需要使用不同的推荐策略来推荐。依赖关系只是被转移了，并没有被消除。

熊小猫：依赖关系确实没有被消除，但转移是有意义的。Recommender和Strategy构成了“推荐组件”，Recommender和Strategy的依赖关系是组件内依赖。而客户端和Strategy的依赖是组件间的依赖。当出现多个客户端调用推荐组件时，优劣便显而易见。



兔小白：确实将依赖关系转移到组件内部后，整个程序的依赖关系会简单很多。

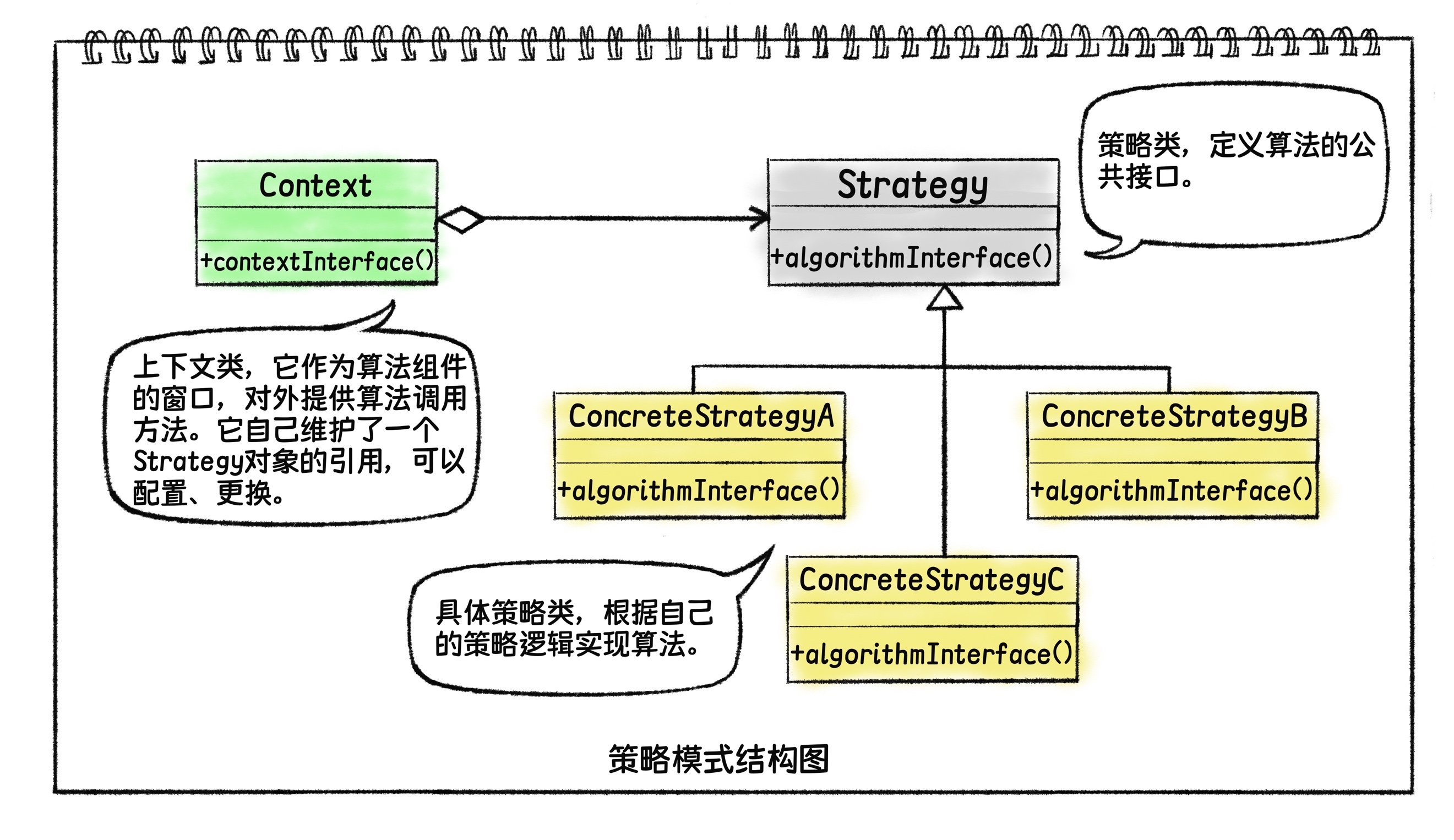
熊小猫：类是一种封装，组件也是一种封装。类和类之间的依赖要尽量减少，组件和组件间的依赖更要尽量减少。这就像在公司里，部门内部的问题都好解决，一旦跨部门，问题就会变的复杂。

兔小白：这让我想起了外观模式。Recommender就像是推荐组件的外观类。

熊小猫：在这一点上，两者的设计思想是类似的。但外观模式适合更为复杂的组件场景。策略模式有其独特的适用场景。下面我们来详细讲讲策略模式。

## 24.4 策略模式适用场景

熊小猫：我们先来看看策略模式的结构图。



Strategy是策略类，定义算法的公共接口。ConcreteStrategy是具体策略类，根据自己的策略逻辑实现算法。Context是上下文类，它作为算法组件的窗口，对外提供算法调用的方法。它自己维护了一个Strategy对象的引用，可以配置、更换。它对算法的实现，实际是调用了Strategy对象。

策略模式的特点是算法封装，封装算法的策略类可扩展，并且Context可以灵活切换算法。我们看看策略模式适用场景的特点。

1. 同样的接口定义，但存在多种算法。比如例子中的商品推荐策略；驾车路线推荐的时间优先和费用优先算法。
2. 需要灵活切换算法。既然有多种算法，一般都需要灵活切换，但这也取决于业务需要。

## 24.5 策略模式与简单工厂模式

熊小猫：有些时候，开发人员会把应该使用策略模式的场景，错误的使用了简单工厂模式。这是因为策略类可以独立提供服务，客户端通过简单工厂创建需要的策略对象，也可以实现需求。

兔小白：你就直接报我名字吧......不过你说的确实是事实。

熊小猫：其实两种模式的区别很大。我们需要站在客户端的角度来看这两种模式。简单工厂模式属于创建型设计模式，重点在于创建对象。而策略模式属于行为型模式，重点在于封装不同的算法，对外提供统一调用入口。当你发现简单工厂的产品家族具有的行为很少，不同的产品实现类只是算法略有区别，那么可能使用策略模式更为适合。

两者还可以结合起来使用。我之前说策略模式符合迪米特法则，这个说法其实并不准确。

兔小白：我也发现了这个问题。客户端虽然不用和策略对象直接“对话”，但在使用Context对象时，还是需要先创建策略对象，然后为Context设置策略对象。

熊小猫：Context需要设置策略对象，那么策略对象从何而来呢？创建对象自然要交给创建型的设计模式。这里可以结合简单工厂进行优化。你要不要来挑战一下？

兔小白：好啊，两种设计模式都学完了，我看看怎么结合起来！

10分钟后，兔小白改好了程序。

兔小白：我结合简单工厂创建产品对象的方式，改造了Recommender设置Strategy对象的方法。直接看代码吧！

public class Recommender {  
    private RecommendStrategy recommendStrategy;  
​  
    public Recommender(String category) throws Exception {  
        setRecommendStrategy(category);  
   }  
​  
    public void setRecommendStrategy(String category) throws Exception {  
        switch (category) {  
            case "price":  
                this.recommendStrategy = new HighestPriceStrategy();  
                break;  
            case "performance":  
                this.recommendStrategy = new HighestPerformanceStrategy();  
                break;  
            case "costPerformance":  
                this.recommendStrategy = new CostPerformanceStrategy();  
                break;  
            default:  
                throw new Exception();  
       }  
   }  
​  
    public Mobile recommend(List<Mobile> mobiles) {  
        return recommendStrategy.recommendMobile(mobiles);  
   }  
}

改造后，客户端代码不用去创建策略对象，只需要告诉Recommender对象，自己想要使用的算法名字即可。

Recommender panda = new Recommender("costPerformance");  
Mobile mobile = panda.recommend(mobiles);  
System.out.println("推荐的最具性价比手机是" + mobile.getName());  
​  
panda.setRecommendStrategy("performance");  
mobile = panda.recommend(mobiles);  
System.out.println("推荐的性能最优手机是" + mobile.getName());  
​  
panda.setRecommendStrategy("price");  
mobile = panda.recommend(mobiles);  
System.out.println("推荐的价格最高手机是" + mobile.getName());

熊小猫：客户端代码不用知道任何策略类，和策略类彻底说拜拜了！推荐组件的内部实现被完全隐藏，程序的依赖关系更为简单，并且完全符合迪米特法则。

1. 手机信息为虚构，推荐结果仅作为练习举例，不反映真实情况。 [↑](#footnote-ref-1)