## 第13章 为什么加盟快餐店越来越多？——外观模式

## 13.2 连锁加盟，助你轻松当老板

结束了一天繁忙的工作，兔小白和熊小猫一起来到快餐店吃晚饭。

熊小猫：你有没有发现，现在大街上连锁品牌快餐店很多？

兔小白：确实是这样啊！几乎所有的快餐品类，都被连锁品牌占据了。

熊小猫：连锁快餐店之所以能快速占领市场，是因为开放了加盟。加盟让开饭店这件事变得容易多了！品牌方为加盟商提供了很多便利。你想一想，如果不通过加盟开一家饭店，需要做哪些工作？

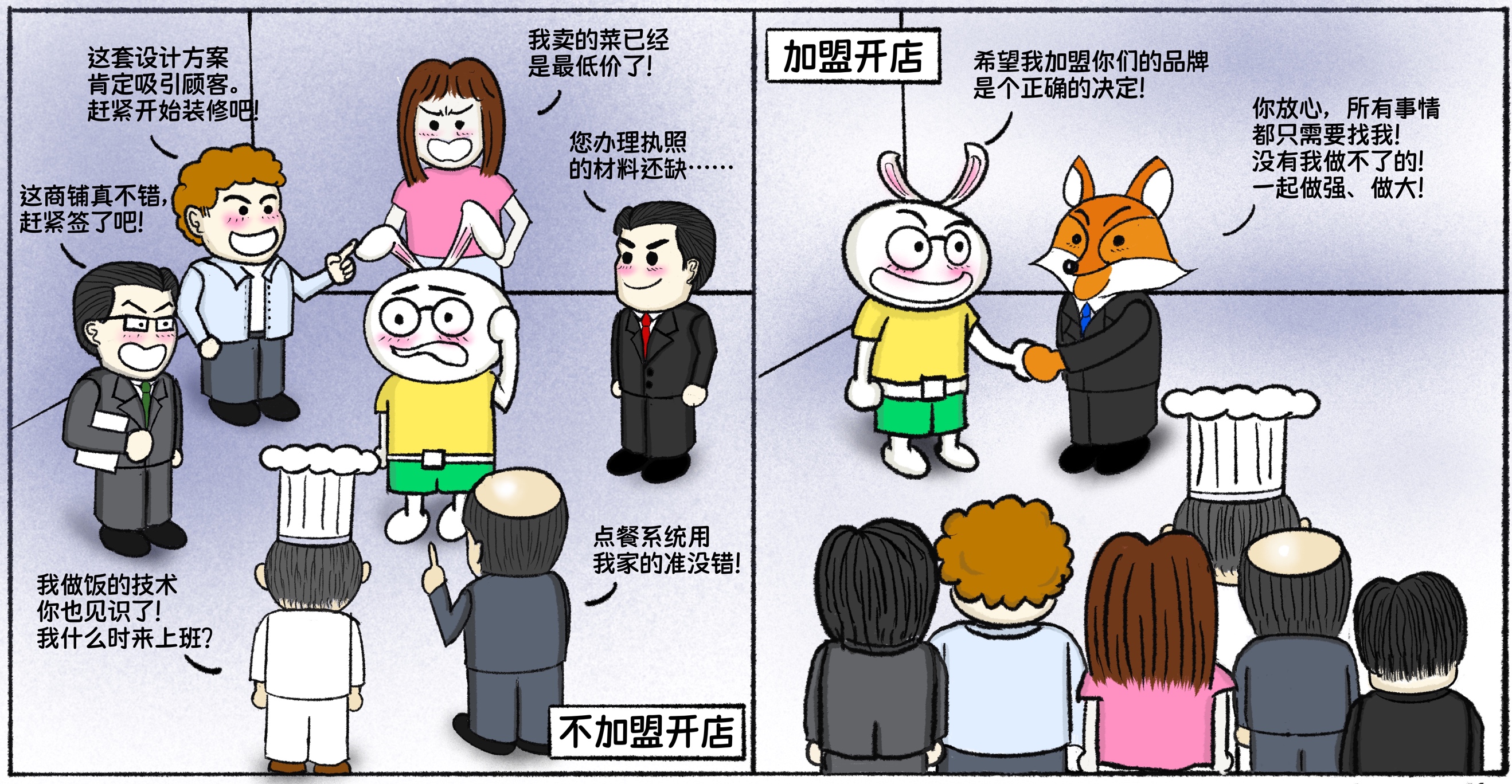
兔小白：

熊小猫：你还忘了最重要的一个步骤，办理营业执照啊！

兔小白：哎呀，步骤实在太多，一不小心把最重要的给漏掉了。

熊小猫：开饭店之所以麻烦，是因为步骤繁多。而且每个步骤你都会面临如何寻找供货方、如何选择供货方、如何和供货方沟通等问题。

但如果选择加盟一家餐饮品牌，这些麻烦事就全都没有了！品牌方可以帮助你餐厅选址。品牌方有合作的装修公司。品牌方甚至可以直接给你派遣厨师和服务员。菜单、软件这些开店必需品，品牌方也都有现成的。营业执照可以交给品牌方去办理，你只需要提供材料。食材的采购就更简单了，直接从品牌方订货。几乎开店的所有步骤，你都只需要和品牌方打交道，免去了和众多形形色色的人沟通的麻烦。



兔小白：选择加盟开店确实省事！这么说我也可以当老板去开饭店了？

熊小猫：加盟能让餐饮品牌快速扩张的原因是，老板只需要和品牌方打交道，不需要和那么多的对接方直接对接，从而吸引更多的人加盟开店。加盟降低开店门槛的思想，在软件设计中也有运用。有一种叫做外观模式的设计模式。你要真想以后开饭店，咱们可以先在程序中过过瘾，来学习外观模式！

## 13.1 独立开店，我的店面我做主

熊小猫：咱们先不用加盟的方式来实现开饭店程序，让你过足开饭店的瘾。我把需求精简一下，假设老板自己有店面，开店仅需要几个主要步骤——装修、办营业执照、招聘员工和订购食材。你来写代码实现吧！

兔小白：这可比真正开饭店简单多了，很快搞定！

10分钟后，兔小白完成了没有加盟商的开饭店程序。

兔小白：程序涉及到对象比较多。有装修公司、政府部门、招聘网站、供应商，还有开饭店的人。但是都不复杂，咱们逐一看一下代码。

装修公司类DecorationCompany，负责装修工作。

public class DecorationCompany {  
    public void decorate(){  
        System.out.println("装修店面完成。");  
   }  
}

招聘网站类RecruitingWebsite。招聘的工作要找它。

public class RecruitingWebsite {  
    public void hireTalent(){  
        System.out.println("雇佣员工完成。");  
   }  
}

政府机关类GovernmentOffice，处理营业执照申请。

public class GovernmentOffice {  
    public void applyBusinessLicense(){  
        System.out.println("申请营业执照完成。");  
   }  
}

最后是食材供应商类Supplier。可以在它这里订购蔬菜和肉类。

public class Supplier {  
    public void orderVegetables(){  
        System.out.println("订购蔬菜完成。");  
   }  
​  
    public void orderMeat(){  
        System.out.println("订购肉类完成。");  
   }  
}

由于没有加盟商，所以开饭店的人就得麻烦一点，需要分别和以上几个对象打交道，完成开饭店的步骤。

public class People {  
    public void openRestaurant(){  
        DecorationCompany decorationCompany = new DecorationCompany();  
        GovernmentOffice governmentOffice = new GovernmentOffice();  
        RecruitingWebsite recruitingWebsite = new RecruitingWebsite();  
        Supplier supplier = new Supplier();  
​  
        decorationCompany.decorate();  
        governmentOffice.applyBusinessLicense();  
        recruitingWebsite.hireTalent();  
        supplier.orderMeat();  
        supplier.orderVegetables();  
​  
        System.out.println("饭店成功开业！");  
   }  
}

客户端代码非常简单，调用People对象的openRestaurant方法。

People people = new People();  
people.openRestaurant();

执行结果如下，饭店这就开起来啦！

装修店面完成。  
申请营业执照完成。  
雇佣员工完成。  
订购肉类完成。  
订购蔬菜完成。  
饭店成功开业！

熊小猫：这个例子看起来似乎很简单，真实情况要复杂的多。我们把开饭店所用到的这些类集合在一起，他们构成了一个“开饭店子系统”，对外提供开饭店所需要的服务。

现在这个子系统并不复杂，但是真实的子系统随着多年的系统演进，会变得越来庞大，类的数量会越来越多。此时，你如果再想使用这个子系统，需要花大量时间了解子系统内部的实现，每个类的职责是什么、有哪些接口可以用。这就好比你自己开饭店，一开始什么都不知道，只能从零学起。

兔小白：哎，我把问题想简单了。你说的这种情况确实普遍存在。

熊小猫：更严重的问题是，当你把子系统内部的实现搞明白，写完程序没多久，另外一位程序员又要用到这个子系统进行开发。他又要把你走过的路全部走一遍。这时候我们就需要思考，如何让这条路变得好走一点。

从程序结构来看，People类依赖了太多子系统的内部类，导致People类使用子系统过于复杂。而且他们之间过于紧密的耦合，会让子系统内部类的变化，直接影响到People类。

不要觉得连锁加盟意义不大，一个好的品牌方能带来很多收益。要不你再考虑考虑以加盟的方式开店？

兔小白：好的，我试试把加盟的品牌方添加进来。

## 13.3 加盟开店，受益颇丰

兔小白：我在程序中增加了品牌方类BrandSideCompany，它整合了开店所有对接方的行为，只不过行为实现都是调用相应的对接方来实现。

public class BrandSideCompany {  
    private DecorationCompany decorationCompany = new DecorationCompany();  
    private GovernmentOffice governmentOffice = new GovernmentOffice();  
    private RecruitingWebsite recruitingWebsite = new RecruitingWebsite();  
    private Supplier supplier = new Supplier();  
​  
    public void decorate(){  
        decorationCompany.decorate();  
   }  
​  
    public void applyBusinessLicense(){  
        governmentOffice.applyBusinessLicense();  
   }  
​  
    public void hireTalent(){  
        recruitingWebsite.hireTalent();  
   }  
​  
    public void orderMeat(){  
        supplier.orderMeat();  
   }  
​  
    public void orderVegetables(){  
        supplier.orderVegetables();  
   }  
}

品牌方就像一个代理，他负责对接所有开店需要的对接方。想要开店的人只需对接品牌方。People类现在只依赖BrandSideCompany一个类，依赖关系简单多了。

public class People {  
    public void openRestaurant(){  
        BrandSideCompany brandSideCompany = new BrandSideCompany();  
​  
        brandSideCompany.decorate();  
        brandSideCompany.applyBusinessLicense();  
        brandSideCompany.hireTalent();  
        brandSideCompany.orderMeat();  
        brandSideCompany.orderVegetables();  
​  
        System.out.println("饭店成功开业！");  
   }  
}

嗯 ......不过我发现一个问题，虽然People类的依赖关系简单了，但BrandSideCompany承接了原来People类的所有依赖。程序整体的依赖关系并没有减少啊？

熊小猫：这是因为还没有开发新需求。如果黄小汪要开发一个投资公司Company类，也要实现开饭店的功能。当没有品牌方BrandSideCompany类时，你需要怎样写代码？

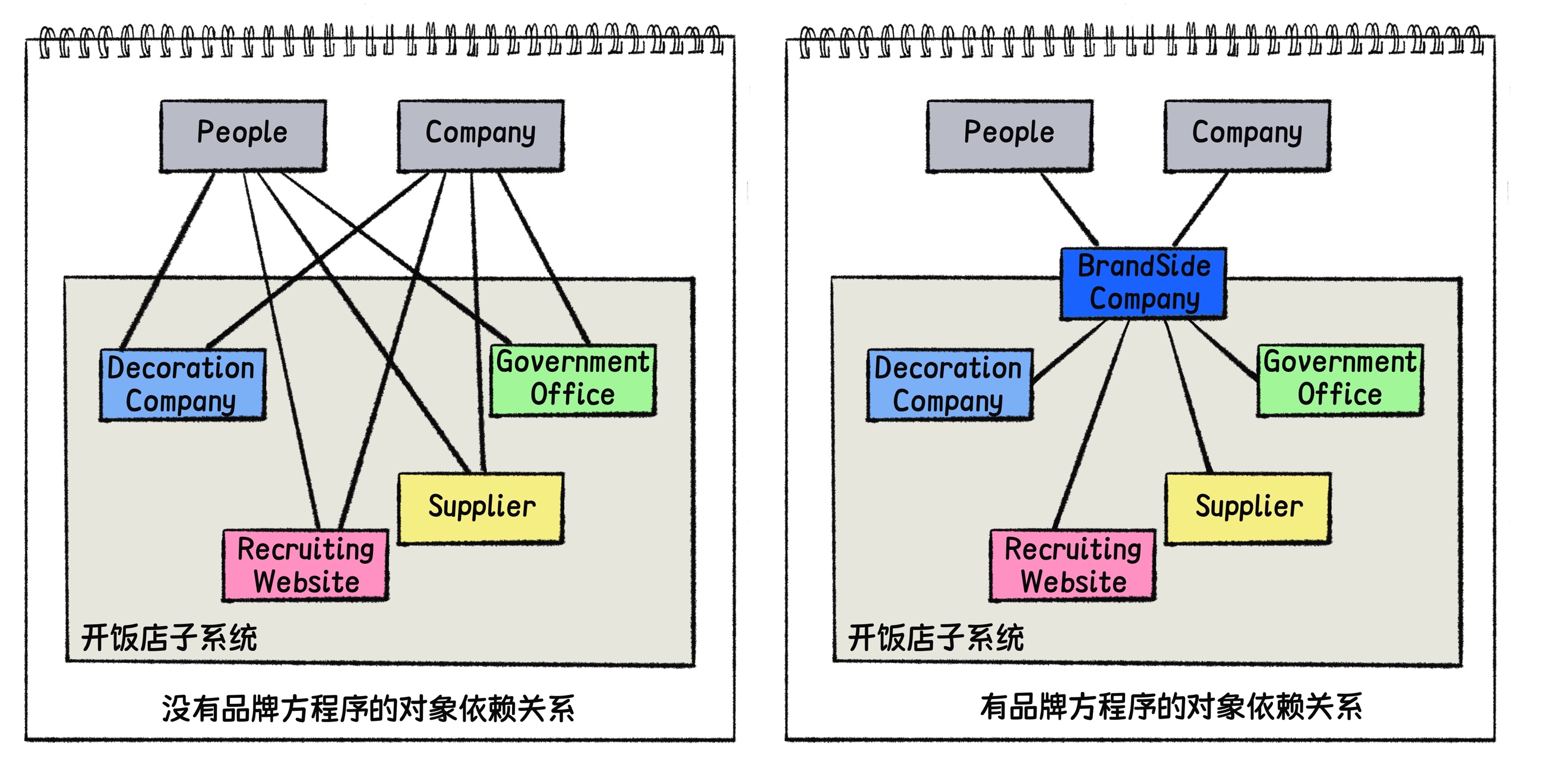
兔小白：Company类需要按照上一版People类的实现方式，依次调用各个对接方的方法。

熊小猫：你看，这个时候就能凸显品牌方的作用了。引入品牌方BrandSideCompany类后，Company类也只需要依赖BrandSideCompany类，就可以实现开饭店。

public class Company {  
    public void openRestaurant(){  
        BrandSideCompany brandSideCompany = new BrandSideCompany();  
​  
        brandSideCompany.decorate();  
        brandSideCompany.applyBusinessLicense();  
        brandSideCompany.hireTalent();  
        brandSideCompany.orderMeet();  
        brandSideCompany.orderVegetables();  
​  
        System.out.println("饭店成功开业！");  
   }  
}

兔小白：有道理，既然作为子系统，使用它的客户类肯定不会是一个。

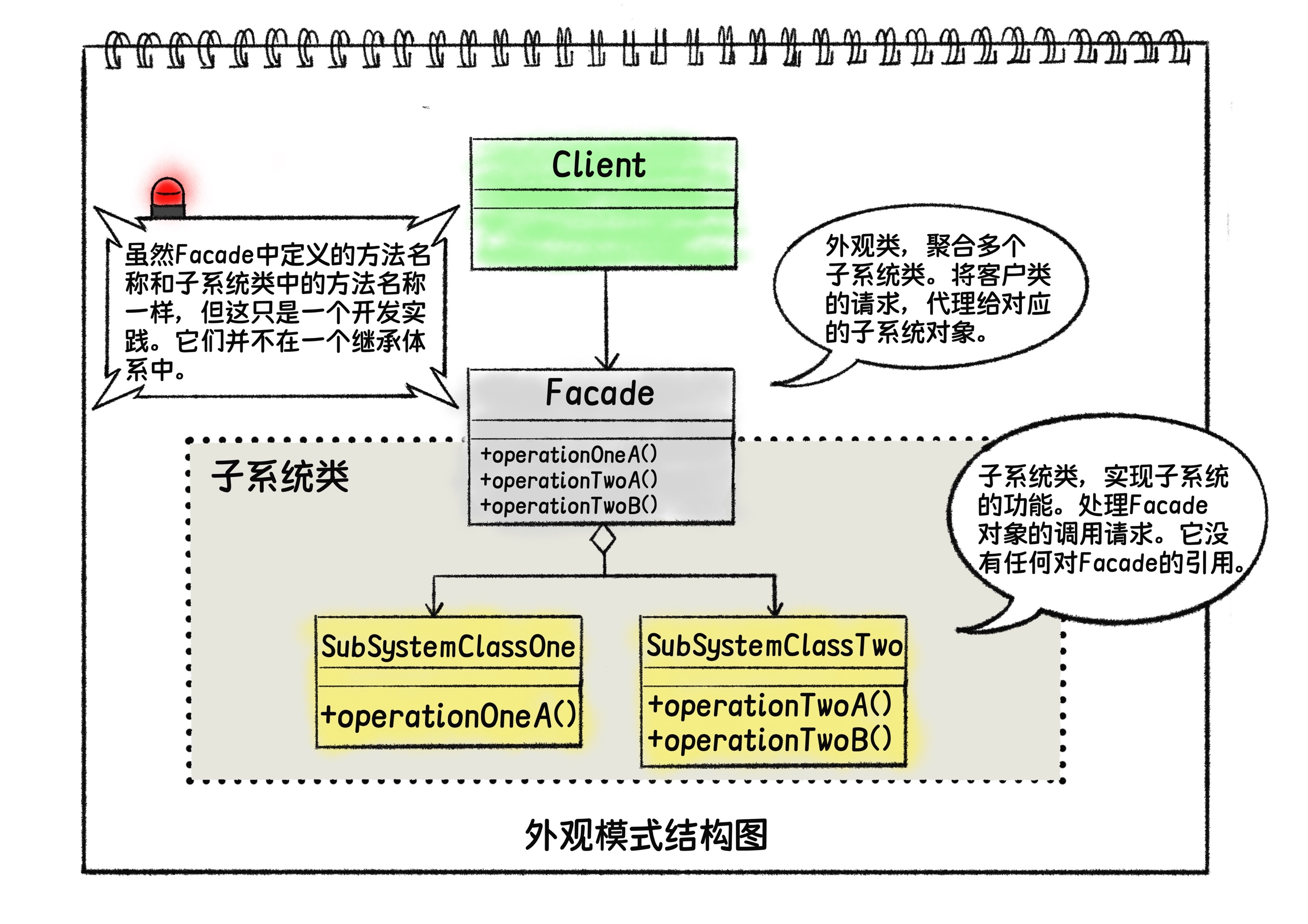
熊小猫：BrandSideCompany类属于子系统，所以他依赖子系统内部类是合理的，这是子系统类之间的依赖。它将子系统和客户类隔离开，子系统变化的影响被局限在其内部。



外观模式在开发中经常会被用到，我们来看看外观模式适合的场景。

## 13.4 外观模式适合的场景

熊小猫：下面是外观模式的结构图。



外观类Facade是子系统的外观。所谓外观，其实就是子系统想呈现给客户类的样子。换句话说，外观类是子系统对外暴露的接口集合。外观模式通过外观类，将子系统内部与客户类隔离开。客户类只需要和外观类打交道。原则上外观类不做任何业务操作。**外观类提供的核心价值是对客户类屏蔽子系统内部，将子系统内部服务封装成统一的接口，对外提供服务。**它和例子中的连锁加盟品牌方作用类似。

外观模式有着如下优点。

1. 降低客户类和子系统内部类的耦合度。通过外观类，隔离子系统内部类和客户类。子系统内部类和客户类都只依赖外观类。
2. 降低客户类开发难度。由于外观类对子系统内部进行了封装，所以当你需要对接子系统时，可以先去了解外观类，而不需要了解子系统内部所有的类。另外，外观类统一子系统对外的接口标准，让开发者不再由于子系统类的差异而感到困惑。加盟开饭店的例子中，品牌方使得加盟商开店难度大大下降。

外观模式能够降低程序复杂度。那么，程序首先需要具备一定的复杂度，外观模式才能发挥它的优势。否则引入外观类，反而会提升系统复杂度，降低灵活度。因为多了一层外观类，开发时不但要修改子系统内部类，还需要修改外观类。外观模式比较适合以下这些场景。

1. 内部结构复杂的子系统。子系统的内部复杂有两种原因，一种是业务需求本身就很复杂，所以导致子系统在设计之初就呈现出复杂性。这种场景，我们在设计阶段就应当采用外观模式，引入外观类。还有一种情况是子系统最初并不复杂，也没有采用外观模式。但随着系统的不断演进，子系统变得越来越复杂。当有一天，你在实现新需求的时候，发现客户端类已经和多个子系统内部类紧密耦合。此时，已经到了用外观模式对子系统进行重构的时候。
2. 复杂的层次结构系统。层次架构是较为常见的架构设计。使用外观模式，可以收紧各层之间的入口点，简化层次间的依赖关系。
3. 封装复杂子系统的内部实现。当你不想将一个复杂子系统的内部设计和实现暴露出来的时候，可以利用外观类将子系统封装起来。子系统作为一个整体，通过外观类对外提供服务。这样可以让子系统保持独立性和可移植性。

兔小白：听你说了这么多外观模式的优点，我倒想提出一些反面的质疑。我觉得外观类的职责不够单一呀，他整合了子系统的大部分职责，这不是违反了单一职责原则吗？

熊小猫：这个问题提的很好！外观类虽然对外提供整个子系统的功能，但其实他并没有做任何业务处理。其实他的职责很单一，仅仅负责子系统接口的对外呈现。所以外观类是符合单一职责原则的。

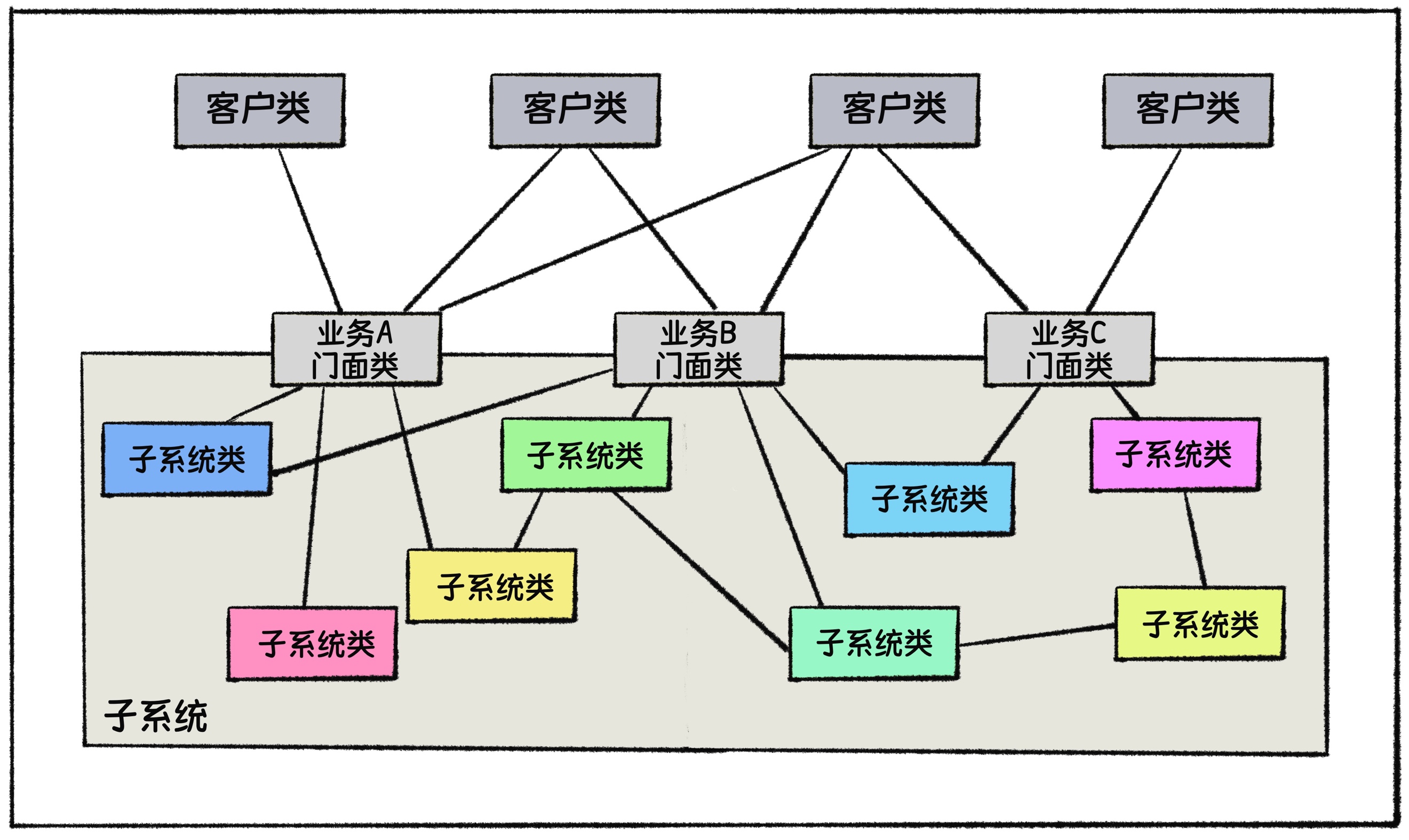
兔小白：我还发现一个问题，外观类不符合开闭原则。子系统内部类的接口修改，会导致外观类的代码修改。

熊小猫：这个问题你提到了点子上。但这也是由外观模式的特性所决定。由于外观类自身不承载任何业务，所以不存在扩展新的外观类实现新需求的情况。因此外观类不需要具备扩展性。另外，外观类的改变一般是被动修改。子系统内部类变化后，如果外观类不做相应修改，程序是要报错的！因此它也只能对修改开放。

兔小白：有道理。而且外观类的职责简单，修改起来很容易。违反开闭原则的弊端也可以接受。

熊小猫：设计原则用来指导软件设计，但不能束缚软件设计。具体情况具体分析，外观模式虽然有悖于开闭原则，但也情有可原。

一个子系统一般只有一个外观类。但是当子系统庞大到一定程度，外观类也可以按照业务模块进行拆分。拆分后的外观类之间，不应该有业务交集，也不应该相互依赖。下图展示了在一个复杂子系统中，类的依赖关系。



兔小白：这样看来，复杂的子系统用外观模式准没错！

熊小猫：你的结论没有方向性的错误。但是什么样的子系统算复杂？是对子系统局部使用外观模式，还是整体使用外观模式？使用一个还是多个外观类？这些问题都需要你的经验积累，才能给出合理的解答。勤学、多用、多问、多思考，才能真正掌握设计模式。