仅以这个文档纪念我在深圳的破出租屋里的孤单寂寞的岁月，多少的青春岁月已经随时间远去，留下的只有那些略微模糊的记忆。 —张永祥

1.设计模式入门之策略模式

问题：鸭子超类，里面定义了所有鸭子的操作；所有具体的鸭子继承这个超类，现在需要有的鸭子会飞，那么将 fly()加入到超类中；此时所有的鸭子都具有了 fly 的行为，包括不会飞的鸭子；可以在不会飞的鸭子类中覆盖这个方法，什么都不做，但是后续每增加一种鸭子，都覆盖这个方法，会很麻烦；此时可以将 fly 方法拿出来放到一个单独的 flyable 接口中，只有会飞的鸭子才实现这个接口；那么实现这个接口，都要实现这个方法，可能飞翔的行为都是一样的，那么会造成代码复用性下降。

设计原则：找出应用中可能需要变化之处，把他们独立出来；不要与那些不需要变化的代码混在一起；在变化的代码封装起来，让其他部分不会受到影响。

解决方案：现在要把 fly 方法从鸭子类中分离出来，因为这是变化的部分，fly 有一个接口，定义所有的飞行行为实现类或者什么都不做，鸭子类中含有这个飞行行为接口，在运行时注入进具体的飞行行为来决定怎么飞；这就是针对接口编程而不是实现编程；

这个解决方案就是策略模式，定义：定义了算法族，分别封装起来，让他们之间可以相互替换，此模式让算法的变化独立于使用算法的客户。

2.观察者模式

JDK 中使用最多的模式。

问题：气象站（获取气象数据的）、WestherData 对象（封装好的气象数据）、布告板（用于展示关于温度、湿度等的信息）；初识的 WeatherData 对象的操作如下：

```
public interface WeatherData {
    float getTemperature();
    float getHumidity();
    float getPressure();
    void measurementsChanged();
}
```

有 3 个获取温度、湿度、气压的 getter 函数以及当数值改变时调用的 measurementsChanged 函数，需要自己实现布告板并且保证布告板的可扩展性；不建议直接在函数内部调用布告板的实现完成布告板的更新，因为就造成了耦合，是对实现编程，不利于后续扩展，后续需要改变的是布告板，而不是 WeatherData，所以没有能够封装变化；布告板太随意，没有共同的抽象接口。

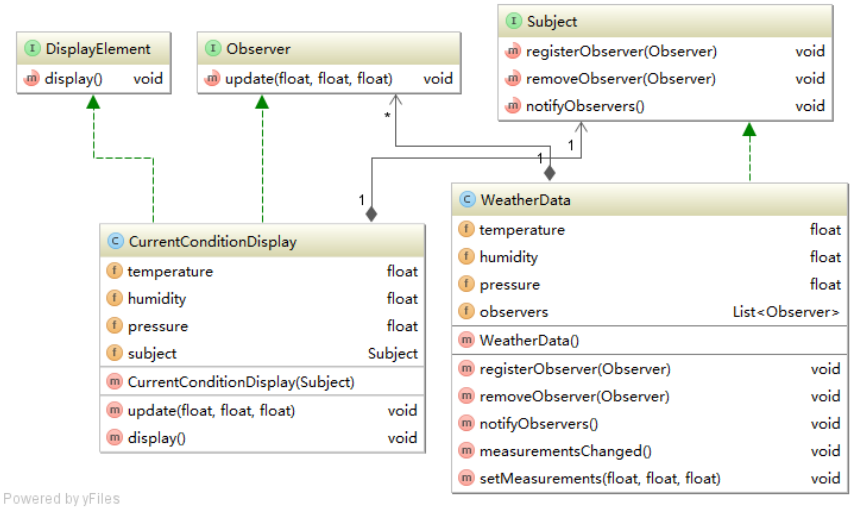
解决方案：观察者模式；简要介绍：就是订阅者与出版者（主题），当主

题变化时，订阅该主题（依赖该主题的状态）的订阅者就会收到通知，做出改变。可以订阅或者取消订阅。

观察者模式的定义：定义了对象之间的一对多依赖，当一个对象改变时，它的所有依赖者都会收到通知并自动更新；

实际上观察者模式解决的是 2 个平等对象之间的依赖关系的问题；这种关系的一个特点就是他们都要互相持有对象的组合，这表明了互相依赖，但是这个是一对多的依赖，多的部分实现共同的接口，解耦了依赖关系，让实现部分可以自由定义。就是将依赖关系有相互之间的接口联系起来来解耦。

气象站实现的类图如下：



Java 为观察者模式内定义了一些类，比如 Observable（Subject）与 Observer 接口等，这些可能是 JDK 在实现时的需要，此处可以直接拿来借用；他们在包 java.util 下；前面使用的是主题更新数据更新观察者，观察者也可以自己去主题这里主动获取数据；

3 装饰者模式

问题：星巴克订单系统，Beverage 饮料抽象类。这是一个基类。

```
public abstract class Beverage {
    private String description;//每种饮料的描述
    public String getDescription(){
        return description;
    }
    //每种饮料的价钱
    public abstract double cost();
}
```

每一种饮料继承这个基类饮料，每一种饮料中还要加入各种调料。Cost()方法返回的价钱就是饮料+调料的价钱；这种继承带来的后果就是每一种饮料都要

加入一个类，每一种饮料放入不同的调料就是一种饮料，会造成类爆炸，而且子类中加入这种调料的计算，重复代码量多。

改进方案 1：将调料抽取出来，因为是公有的，放入父类中，在 Beverage 基类中加入每一种调料标志，代表是否加入了这个调料。

```
public abstract class Beverage {
    private String description;//每种饮料的描述
    private boolean milk;
    private boolean soy;
    public String getDescription(){
        return description;
    }
    //每种饮料的价钱
    public double cost(){
        double tiaoliao=0;
        if(milk){
            tiaoliao+=TiaoLiao.MILK_PRICE;
        }
        if(soy){
            tiaoliao+=TiaoLiao.SOY_PRICE;
        }
        return tiaoliao;
    }
    public boolean hasMilh(){
        return milk;
    }
    public boolean hasSoy(){
        return soy;
    }
    public void setMilk(boolean milk){
        this.milk=milk;
    }
    public void setSoy(boolean soy){
        this.soy=soy;
    }
}
```

此时 cost 写了方法，返回调料的价格，子类计算价钱的后需要先调用父类，将调料单独加入到父类中，减少了类的数量；带来的问题是，新出现了调料，要改父类代码，一些调料可能不适用与有的饮料，客户需要双倍调料时，无法解决，调料价钱是在类的常量中实现的，以后需要更改，很麻烦。

总结：继承不能实现最有弹性与最好维护的设计，继承是在编译时静态决定的子类行为，是统一的，组合与委托也能达到继承的效果，同时是在运行时动态扩展对象的行为，动态的组合对象，可以写新的代码，添加新的功能，无需修改现有的代码，可以在设计超类时没有想到的职责加在子类中，类应该对外扩展开放，对内部修改关闭，加入新的行为扩展类，原有代码都测试了，不能随便修改啊。这样是弹性设计，可以应对新的需求与功能。

遵循开放-关闭原则会引入新的抽象层次，不是每块都需要这么做，只需要在最有可能改变的地方这样做就可以了。

改进方案 2：装饰者模式，以饮料为主体，在运行时以调料来装饰饮料；比如摩卡和奶泡咖啡，先拿一个咖啡对象，以摩卡对象装饰他，以奶泡对象装饰他，调用 cost 方法，并依赖委托将调料价钱加上去。

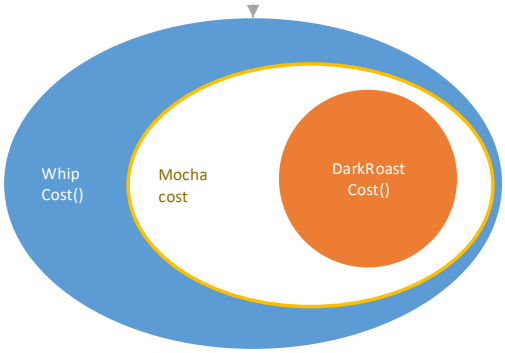
先创建 DarkRoast 对象，继承 Beverage：

```java
public class DarkRoast extends Beverage {

    @Override
    public double cost() {
        // TODO 自动生成的方法存根
        return YinLiao.DARKROST_PRICE;
    }

}
```

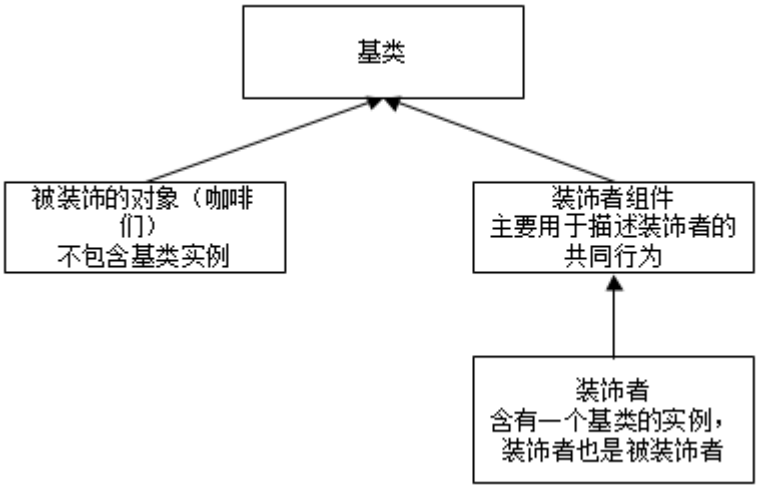
创建 Mocha 对象。也是 Beverage 对象，是装饰者与被装饰者的类型一致，因为继承 Beverage，可以把调料、饮料包括自己，都当成 Beverage。Whip 也是同样的。

```java
public class Mocha extends Beverage {
    private Beverage beverage;


    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }


    @Override
    public double cost() {
        return beverage.cost()+TiaoLiao.MOCHA_PRICE;
    }

}
```

外部调用整个层级就实现了动态扩展层级。形象的表达是：



装饰者与被装饰者有相同的超类型，可以用多个装饰者装饰对象，可以用装饰者代替被装饰者，装饰者可以在委托的行为前后加上自己的行为。



这里也用到了继承，但是只是用继承达到了类型匹配，不是获得行为；行为来自装饰者自定义，如果是继承，就需要修改基类才能获得行为，因为行为是编译器静态确定的。

实际上来说，组合（调料组件与饮料组件）问题，不要用继承，因为这不是继承可以应对的方案，装饰者模式说白了就是组合问题，但是一个比较好的思想是：组合的组件继承与同一个基类，被装饰者是获得基类的行为的，装饰者只是为了获得基类的类型，因为装饰者组件是不能独立存在的，必须依附于一个被装饰者，所以本质来说，他的类型是对的，正确的，符合继承的思想的，里面继承的行为也是必须的，但是继承行为的前提是有被装饰者实例。

Java I/O 就是装饰者与被装饰者的例子。

4.工厂模式

问题：在编程时，虽然变量定义的是接口，但是在 new 时还是 new 的具体的实现类，在多态情况下，在代码中 new 时，需要判断 new 哪个子类；然后写一堆 if else 逻辑等；一旦要加入新的类，那么就要修改这块代码，这违反了，对修改关闭，对扩展开放；针对具体的类编码，就不是对修改关闭了（找出会变化的方面，把他们从不变的部分分离出来）。

案例：披萨店，买披萨；建立的披萨类如下：

```
public abstract class Pizza {
    protected String type="unknow pizza";
    public abstract void setType(String type);
    public void prepare(){
        System.out.println("材料准备");
    }
    public void bake(){
        System.out.println("烘培");
    }
    public void cut(){
        System.out.println("切割");
    }
    public void box(){
        System.out.println("装箱");
    }
}
```

一个抽象类，常规的披萨流程。披萨的订单处理方法如下：

```
Pizza orderPizza(){
    Pizza pizza=new Pizza() {
        @Override
        public void setType(String type) {
            this.type=type;
        }
    };
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

Pizza 是一种抽象的概念，所以弄了一个抽象类；加入需要更多的披萨类型，那么要建立更多的类型的披萨类，还要改订单的方法：

```
Pizza orderPizza(String type){
    Pizza pizza=null;
    if(type.equals("cheese")){
        pizza=new CheesePizza();
    }else if(type.equals("greek")){
        pizza=new GreekPizza();
    }else if(type.equals("pepperoni")){
        pizza=new PepperoniPizza();
    }
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

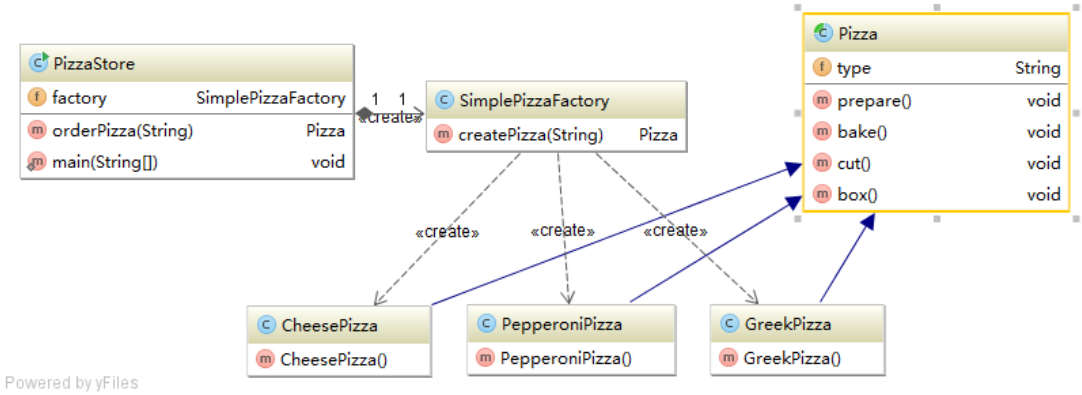
如果再次新增披萨类型，这个方法就要不断的加入修改，所以把修改的地方提炼从不需要修改的地方提炼出来，就是把 new 披萨类型的这段代码抽取出来放到另一个对象中，因为这是变化的，后面的操作是不变的。这个新的对象叫做工厂。orderPizza()就是这个工厂的客户，需要披萨时，就叫工厂做一个，做的细节，由工厂负责，方法只需要得到披萨就可以。

解决方案 1：一个简单的工厂，简单的工厂就是把方法内的代码直接移到工

厂对象的生产方法内。

```
public class SimplePizzaFactory {
    public Pizza createPizza(String type){
        Pizza pizza=null;
        if(type.equals("cheese")){
            pizza=new CheesePizza();
        }else if(type.equals("greek")){
            pizza=new GreekPizza();
        }else if(type.equals("pepperoni")){
            pizza=new PepperoniPizza();
        }
        return pizza;
    }
}
```

这种实现方式将对象的创建与逻辑分离，后面修改代码至修改这一块，不用静态工厂的原因是，静态工厂不能通过继承来改变生产的行为。这种模式其实不是设计模式，是一种编程习惯。类图表示如下：



新的问题出现了：上面的问题解决了 Pizza 种类不同的生产问题，加入，PizzaStore 有多个加盟，处在全球各地，每个店生产的同一种披萨还存在调料差异该怎么办呢？比如纽约店希望的披萨：薄饼、少量芝士；芝加哥厚饼，多芝士；

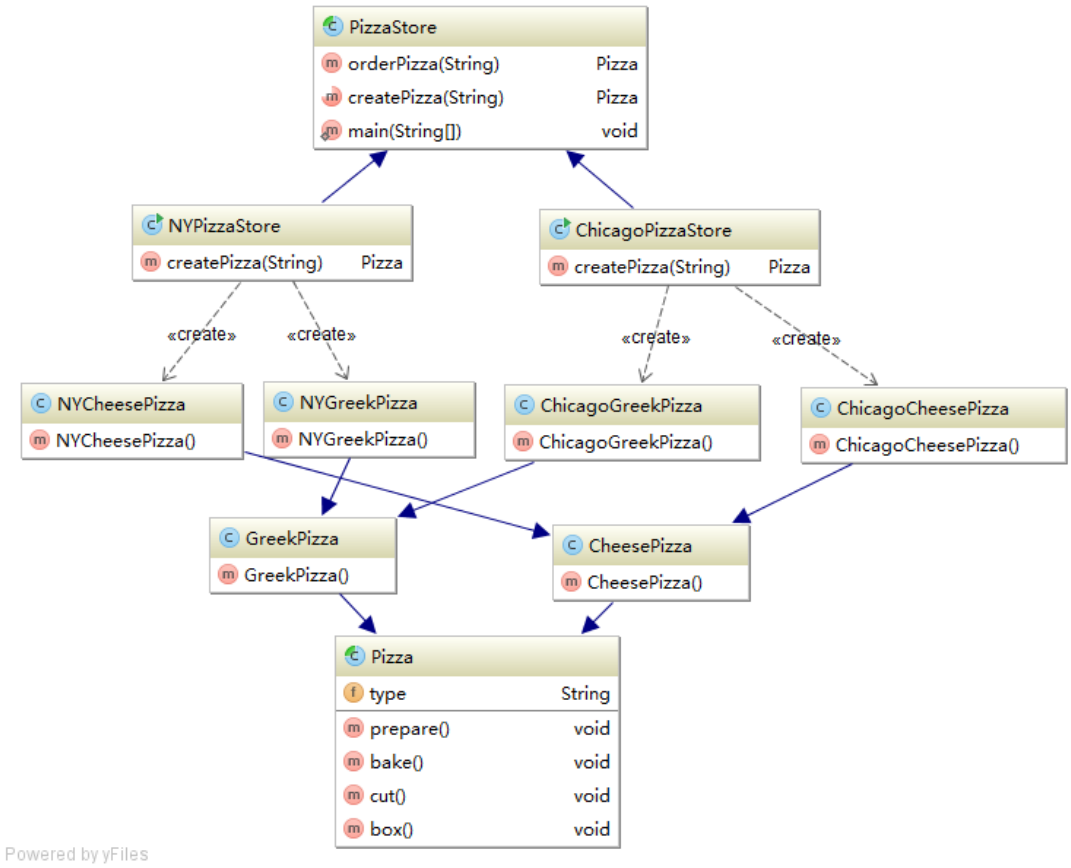
解决方案 1：扩展 SimplePizzaFactory，每个加盟店一个工厂，NYPizzaFactory 生产纽约风味的，ChicagoPizzaFactory 生产芝加哥的，创建纽约店时，创建 NYPizzaFactory；

问题：地域特色的加盟店与工厂是相互分离的，工厂继承于简单工厂，完成的制作披萨过程是固定的，如果披萨店想要在披萨制作上加入额外的操作，就不能实现？

解决方案 2：使用工厂方法：将创建 pizza 的活动加入到披萨店中，Pizza 店父类，只定义接口，具体的操作由加盟店自己做决定。

```
public abstract class PizzaStore{
    SimplePizzaFactory factory=new SimplePizzaFactory();
    Pizza orderPizza(String type){
        Pizza pizza=factory.createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
    public abstract Pizza createPizza(String type);
```

每个分店继承 PizzaStore 后实现自作披萨的方法，自己决定如何制作披萨。最终的工厂方法 UML 图如下：



实际上，就是根据面向对象的分析，将工厂对象的操作移到了每个加盟店的方法中，这个方法就是工厂方法。OrderPizza()，定义在父类中，是因为它不关注具体的 pizza 类型，他是所有 pizza 的操作。工厂方法用来处理对象的创建，将这样的行为封装在子类中，这样超类的代码和子类对象创建代码解耦了。

工厂模式都是用来封装对象的创建，工厂方法模式通过让子类决定该创建的对象是什么来达到将创建对象封装的目的。

工厂方法模式：定义一个创建对象的接口，但由子类决定要实例化的对象，工厂方法让类把实例化推迟到子类。

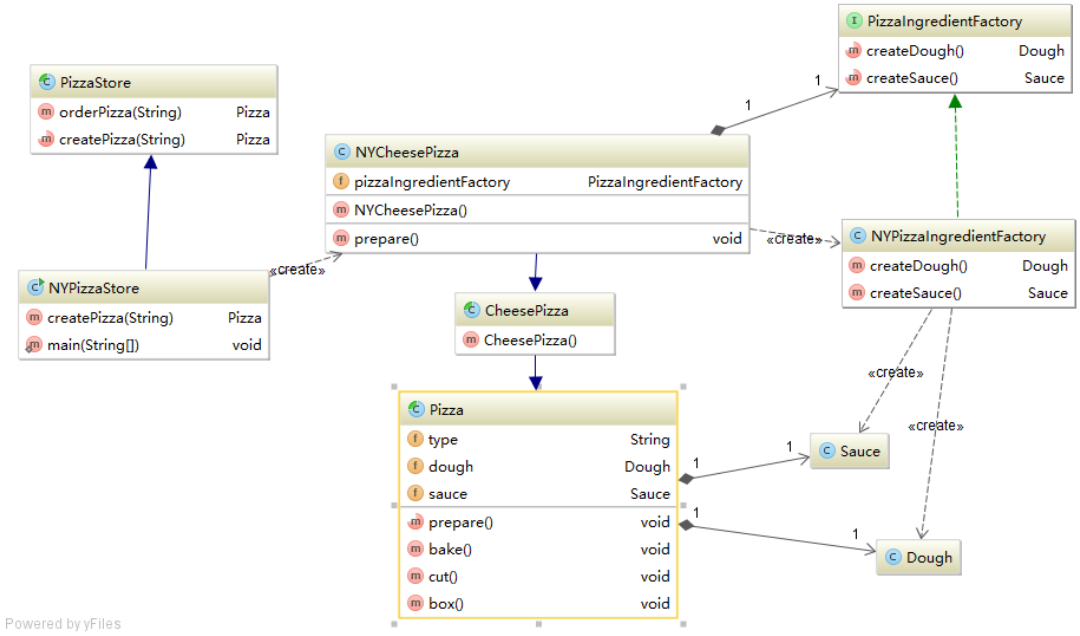
本质：工厂方法也可以叫抽象工厂方法，是对实现的一种行为定义。应用于

继承的多态行为机制上。

依赖倒置原则：依赖抽象，不要依赖具体类；调用者还有被调用者都应该是抽象的形式，而不能是具体的组件，比如：披萨店就是抽象的，店内的披萨也是抽象的，具体的 new 类是在工厂中实现；披萨这个抽象类变为了依赖披萨店，依赖倒置了。变量不可以持有具体类的引用（工厂），不要让类派生自具体类（依赖具体类，变为抽象），不要覆盖基类中已实现的方法。

问题：加入加入了调料问题呢？每个区域的调料的组成都是不同的，如何调配调料呢？加入一个加盟店就建一个调料工厂实现吗？披萨总店负责定义调料工厂生产哪些调料？

解决方案：披萨总店先为调料工厂设定接口。每个区域实现的调料工厂都有共同的行为，调料主要与 Pizza 的 prepare()方法有关，所以 prepare 是变化的部分，抽象出来，定义为抽象类，具体的地域风味的披萨实现时，需要根据自身以及地域的调料工厂实现这个方法。类图：



抽象工厂为家族提供接口，主要是为对象的组件提供接口，这部分的组件是对于对象内来说是变化的，不同的。只能定义行为，而不能定义具体实现；

定义：抽象工厂模式提供一个接口，用于创建相关的或者依赖对象的家族，问题 1 是继承相关的问题，同一类对象具体的行为不同，所以用到了抽象工厂方法，因为耦合度比较高同时变化又比较大；问题 2 是组合相关的问题，Pizza 与调料是不同的组件，所以用抽象工厂实现。

5.单件模式

单件模式就是整个程序这个对象只能有一个实例，在生活中，这样的对象有

很多。简单的单例模式如下：

```
*/
public class Singleton {
    private static Singleton singleton;
    public static Singleton getInstance(){
        if(singleton==null)singleton=new Singleton();
        return singleton;
    }
    private Singleton() {
    }
}
```

构造函数私有化，保证只能在本类内调用，但是没有实例产生就不能调用方法，所以有了静态的 Singleton 属性与 getInstance()方法。这种实现方式在多线程的情况下，容易返回多个实例对象。

解决方案 1：同步 synchronized。

```
public class Singleton {
    private static Singleton singleton;
    public static synchronized Singleton getInstance(){
        if(singleton==null)singleton=new Singleton();
        return singleton;
    }
    private Singleton() {
    }
}
```

简单粗暴，但是每次获取对象时都要检查下同步，影响性能。

解决方案 2：不用延迟初始化，在程序编译时，就初始化单例，那么 getInstance()直接返回就可以了。

解决方案 3：双重检查枷锁，既使用了同步又减少了同步的次数。

```
public class Singleton {
    private static volatile Singleton singleton;
    public static  Singleton getInstance(){
        if(singleton==null) {
            synchronized (Singleton.class) {
                if(singleton==null)singleton = new Singleton();
            }
        }
        return singleton;
    }
    private Singleton() {
    }
}
```

6.命令模式（封装调用）

命令模式将动作的请求者与动作的执行者解耦。

问题 1：餐厅的工作流程：顾客（产生订单）->招待（放到柜台）->厨师（制作）->产品；顾客（createOrder）->招待（takeOrder,orderUp）->厨师（make）->ouput；在这个流程中，招待与厨师是解耦的，招待只需要把订单往柜台放就可以了，厨师自然会根据订单生产佳肴，这就是招待这个服务请求者与厨师这个服务执行者进行了解耦，各自执行，互不相关；招待通过发出一个命令（订单）给厨师就可

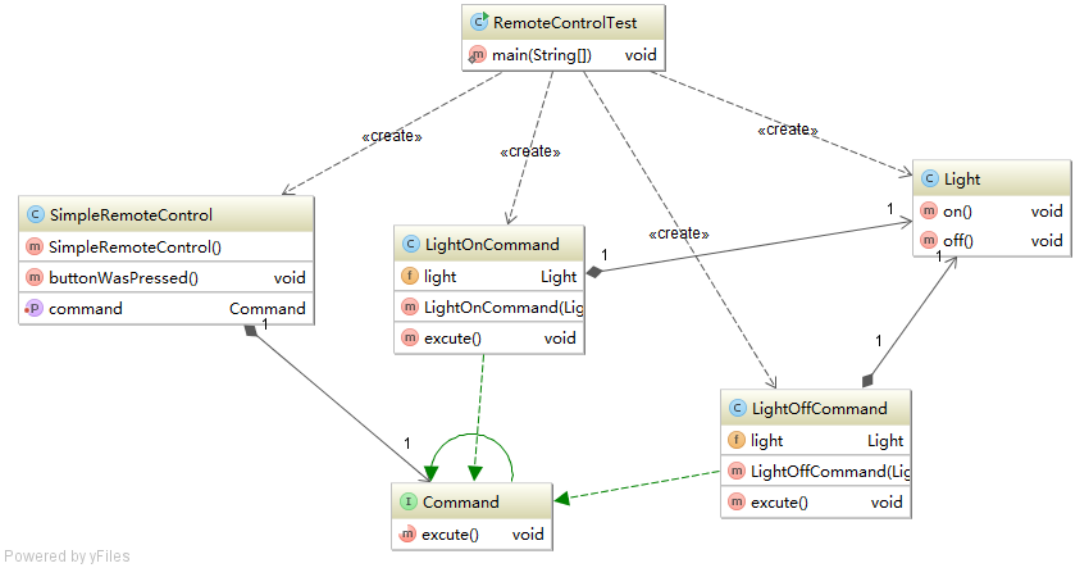
以了，这就是命令模式；就是 2 个实体件的交互，怎么交互呢？命令模式。

模式：Client(createCommandObject())->Command(里面会有一些命令规定的行为)->setCommand(注入命令对象)->excute()->Receiver。

问题 2：遥控器，7 个插槽，每个插槽有 2 个控制健，每个插槽控制一个设备。

解决方案 1：设置命令接口，每一种命令实现该接口，向命令里注入执行者，接口操作实现里调用执行者的对应操作，在服务请求者处，注入命令，然后执行操作。

类图如下：



RemoteCrontolTest 类就是人了。

```
public class RemoteControlTest {
    public static void main(String[] args){
        SimpleRemoteControl simpleRemoteControl=new SimpleRemoteControl();
        Light light=new Light();
        LightOnCommand onCommand=new LightOnCommand(light);
        LightOffCommand offCommand=new LightOffCommand(light);
        simpleRemoteControl.setCommand(onCommand);
        simpleRemoteControl.buttonWasPressed();
        simpleRemoteControl.setCommand(offCommand);
        simpleRemoteControl.buttonWasPressed();
    }
```

类的内容完美体现了餐厅的工作流程；命令内部是封装了服务执行者的，服务请求者只需要注入命令就可以，发出自己需要的操作，那么为什么服务请求者内部为什么不直接注入服务执行者，然后定义多个方法，每个方法内部使用服务执行者执行对应的操作呢，因为这样就将服务请求者与服务执行者耦合在一起，服务执行这都是相同的类型还好说，如果是不同的类型呢？就要新增代码来实现，所以通过增加一个间接层来实现服务器请求者与服务执行者的解耦。并将本来应该在服务请求者内的方法操作，抽取出来，每个操作形成一个命令对象。将服务

执行者的操作放入这个命令对象中，实际上来说，代码是没有减少的；实际来说就是 2 个不同实体的交互映射问题，每一种映射关系就建立一个中间层对象。

问题 2：加入撤销功能。

解决方案：在 Command 接口定义方法 undo，每个实现的具体命令都实现 undo 方法，在 SimpleRemoteControl 类中定一个 undoCommand 的属性记录上次执行的命令。代码如下：

```
public class SimpleRemoteControl {
    Command onCommand;
    Command offCommand;
    Command undoCommand;

    public SimpleRemoteControl() {
    }

    public void setCommand(Command onCommand,Command offCommand) {
        this.onCommand = onCommand;
        this.offCommand = offCommand;
    }
    public void onButtonWasPressed(){
        onCommand.excute();
        undoCommand=onCommand;
    }
    public void offButtonWasPressed(){
        offCommand.excute();
        undoCommand=offCommand;
    }
    public void undoButtonWasPressed(){
        undoCommand.undo();
    }
```

Undo 的行为是在服务请求者层面的，所以他会在服务请求者类中定义一个属性记录前一个命令。假如 undo 的操作涉及到服务者的细节，需要在对应的映射命令中记录一些状态。

问题 3：一个命令执行多个操作。因为每个命令对应一个映射关系，需要建立一个信息的映射关系包含所有这些映射关系。没了让所有映射关系的行为保持一致（都是映射），他们继承的命令接口是相同的。

```
public class MacroCommand implements  Command {
    Command[] commands;

    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    @Override
    public void excute() {
        for(Command command:commands){
            command.excute();
        }
    }

    @Override
    public void undo() {
        for(Command command:commands){
            command.undo();
        }
    }
```

代码如上图。

问题 4：为何命令对象不直接实现 excute 的细节呢？我觉得完全可以，但是不符合面向对象的思想，服务执行者与命令是 2 个不同的实体，如果结合到一起，职责不清，同时造成了命令与服务执行者之间的完全耦合，导致这个命令不能再次应用到别的服务执行者。

命令模式的用途：队列请求，可以将命令对象放入队列中，线程从队列中取命令对象执行 excute 方法；

7.适配器模式与外观模式

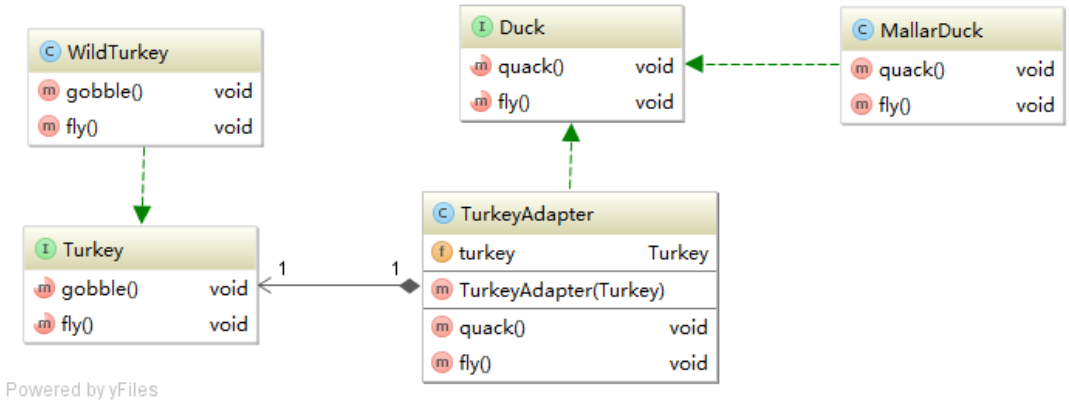
场景：美国制造的笔记本在欧洲国家使用，由于电源插头不一样，所以需要中间一个插头转换器才能接入电源，这个转换器改变了插座的接口，这个就是适配器。面向对象的适配器，就是客户端需要一个接口，这个接口可能是客户端定义的，现在需要为客户端接入新的底层部件，但是这个部件的接口不是客户端需要的接口，那么需要定义一个类实现客户端需要的接口，接口内部完成新部件的接口调用，那么新部件通过适配器，接口就变为了客户端需要的样子。是不同组件之间的交互问题，与命令模式有什么区别呢？实际都是完成 2 个组件之间的映射，但是命令模式是一个映射就会建立一个中间对象，适配器模式是 2 个组件之间只建立一个映射，映射内部每个方法代表了一个命令。

问题 1：鸭子，火鸡，现在用火鸡来冒充鸭子，因为火鸡与鸭子的接口不同，所喜需要适配器。

```
*/
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    @Override
    public void quack() {
        turkey.gobble();
    }

    @Override
    public void fly() {
        turkey.fly();
    }
}
```

适配器代码，需要实现客户端预定义的接口。传入火鸡对象，因为火鸡对象是实际的对象。

适配器一般纸包装一个底层接口，如果需要包装多个底层接口，这种模式是外观模式。



适配器模式定义：将一个类的接口，转换成客户希望的另一个接口，适配器让原本不兼容的类可以合作无间。

适配器的中间层让客户端接口与其实现解耦，底层实现不论怎么变化对于客户端来说都是透明的。

上面的适配器是对象适配器模式：因为是组合模式，底层接口是通过组合传递到适配器中的，这种模式下，适配器就是适配器完成转换操作，底层接口就是底层接口，它们 2 个分离的。还有类适配器模式，类适配器简而言之就是适配器继承了底层接口实现的对象又继承了客户端规定接口的行为，这是一种多重继承，实际就是将适配器与底层接口实现融合在一起，因为此时适配器继承底层对象，就是变为了它的一个子类，所以就是形成了一个新的产品，当适配器与底层对象关联度很大时，就需要类适配器模式。

```java
public class TurkeyClassAdapter extends WildTurkey implements Duck {
    @Override
    public void quack() {
        this.gobble();
    }
    @Override
    public void fly() {
        super.fly();
    }
}
```

因为 fly 方法在 2 个接口中的定义相同，所以用 super 调用。

优劣势：对象适配器可以组合任意的底层接口的子类，弹性更大；类适配器类与某个具体的底层对象耦合在一起，因为继承，可以覆盖底层接口对象的行为，因为适配器也是底层对象的子类了，可以覆写行为。

有时客户端固定的接口的行为更多，而底层接口对象往往不能通过适配器完成规定的行为，此时可以通过抛出异常来解决。

装饰者模式与适配器模式差异：差异很大，装饰者的在进行装饰时，是不允许改变接口的行为，从一定意义上讲，每一层的装饰的行为都是统一的，不是简单的行为伪装。而适配器模式改变了行为，继承后，内部的行为发生了严重的变

化，实际上来说，装饰模式是对一个对象进行装饰，装饰品与被装饰的对象是依附关系；适配器器模式是 2 个几乎没有关系不同对象的关联。

外观模式：它将一个或者数个类的复杂的一切隐藏在背后，只显露出一个干净美好的外观。

问题：组建一个家庭影院系统，由不同的组件组成。现在想观看一步影片（抽象的一个行为）；需要打开爆米花机、打开投影机、打开 DVD.....一堆的行为。如果你没有一个外观类，你就自己需要亲自一步一步做这些事情，现在如果有一个外观类接口，里面包含了家庭影院的所有组件，就是家庭影院，并向外提供一个外观（遥控器）点击播放电影，这些操作都有外观类来完成，你就不用亲自做这些了，还不容易出错或者忘掉某步。

外观只是提供了一个抽象操作的简化接口，方便调用，是的客户端与底层的组件解耦，互不影响。

外观模式的定义：提供一个统一的接口，用来访问子系统中的一群接口，外观定义了一个高层接口，让子系统更容易使用。

最少知识原则：减少对象之间的关联与交互，只与最亲密的交谈。实际就是间谍组织的单线联系原则，一条线坏了，不影响其他线。就是自顶向下的逐层分解，不能跨层级交互。

在对象的方法内，只应该调用以下范围的内容：该对象本身、参数、方法内创建的局部对象、对象的任何组件。就是每个对象只管他本身及它的直属下属，只会向直属下属分配任务，别人的下属不要管，也不能分配任务。在方法内调用其他方法返回的对象，这种属于管理下属的下属，这种是禁止的。

缺点：会制造更多的中间管理人员用来沟通。为神马与现实这么类似。

8.模板方法模式（封装算法）

问题 1:冲泡咖啡：把水煮沸、用沸水冲泡咖啡、把咖啡倒进杯子里、加糖与牛奶；冲泡茶：把水煮沸、用沸水浸泡茶叶、把茶倒进杯子里、加柠檬。建立一个比较好的模型。

解决方案 1：咖啡与茶分别建立一个类。因为茶与咖啡的一些操作是相同的，如果在每个类中写，无疑出现了重复的代码，后续如果是温水煮，怎么办呢？就要一个一个来修改。

解决方案 2：茶与咖啡都是饮料，可以将固定的步骤抽取到父类中，变化的部分转移到子类中去。

```
public abstract class CoffeineBeverage {
    abstract  void prepareRecipe();
    void boilWater(){
        System.out.println("把水烧开");
    }
    void pourInCup(){
        System.out.println("倒进杯子里");
    }
}
```

这是父类，共有方法写到父类，还有共同的操作，但是具体实现不同的prepareRecipe()方法设为抽象方法。

```
public class Coffee extends CoffeineBeverage{
    void prepareRecipe(){
        boilWater();
        brewCoffeeGrind();
        pourInCup();
        addSugarAndMilk();
    }

    public void brewCoffeeGrind(){
        System.out.println("冲泡咖啡");
    }
    public void addSugarAndMilk(){
        System.out.println("加糖与牛奶");
    }
}
```

```
public class Tea extends CoffeineBeverage {
    void prepareRecipe(){
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

    public void steepTeaBag(){
        System.out.println("冲泡茶");
    }
    public void addLemon(){
        System.out.println("加柠檬");
    }
}
```

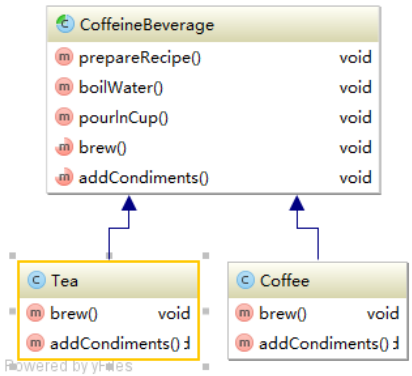
对比茶与咖啡的 prepareRecipe()方法，他们的操作步骤几乎是相同的，冲泡茶与冲泡咖啡进一步抽象都是冲泡饮料，加糖与牛奶与加柠檬都是加调料。所以这个不同的 2 个步骤可以抽象到父类中，具体的操作有子类去实现，prepareRecipe()就是执行 4 个步骤的分装，在父类定义好。这个就不是变化的了。

解决方案 3：将固定步骤组合的方法写到父类，作为不变的部分，其中每一个步骤，共有的则在父类中，不同的则设为抽象方法，由子类去实现。

父类如下：

```java
public abstract class CoffeineBeverage {
    void prepareRecipe(){
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }
    void boilWater(){
        System.out.println("把水烧开");
    }
    void pourInCup(){
        System.out.println("倒进杯子里");
    }
    abstract void brew();
    abstract  void addCondiments();
}
```

类图如下：



方案 3 就是模板方法模式，prrepareRecipe()就是模板方法，这个方法内是一系列共同操作的组合，而且是不变的。方法操作就是遵守这样一个流程。模板方法内的每个步骤细节可能是不同的。模板方法模式就是将一些列有相同父类对象的共同的行为抽象到父类中，父类定义了一个行为准则（不是抽象方法）的具体实现，实现的细节是抽象的，有子类实现。

模板方法的定义：在一个方法中定义算法的骨架，而一些步骤延迟到子类中，模板方法可以使得子类在不改变算法结构的情况下，重新定义算法中的某些步骤。这使得算法本身与算法的细节解耦了。

模板方法所在的父类中也可以有一些方法，里面为空或者执行一些默认的操作，这写方法也是模板方法里面的步骤，这种方法叫做钩子，因为你可以通过继承时覆盖改变这些方法，进而影响算法的执行步骤。如果继承时，不需要改变默认的行为，就可以不覆盖这个钩子方法。钩子是算法中可选的部分，所以一个步骤定义成钩子方法就代表这个步骤是可选的。

新的设计原则：好莱坞原则，别调用（打电话给）我们，我们会调用(打电话给)你。实际就是高层组建决定什么时候和怎样使用底层组件。模板方法就是这个原则的体现，父类的 prepareRecipe()方法就是高层组件，他是调用子类的步骤细节的。

模板方法模式有时并不一定需要继承算法所在的类，只要方法（算法）内用到了某些操作，这些操作要抽象出来由具体的子类去实现，就可以是模板方法模式，这个子类与算法所在类可能没有任何关系。Arrays.sort(Object[])算法内，用来比较对象时，需要传入的数据对象继承 Compareble 接口，实现里面的操作。只要是算法内步骤与具体实现分离的都是模板方法模式。

Swing 的 JFrame 类中的 paint()方法就是钩子方法；

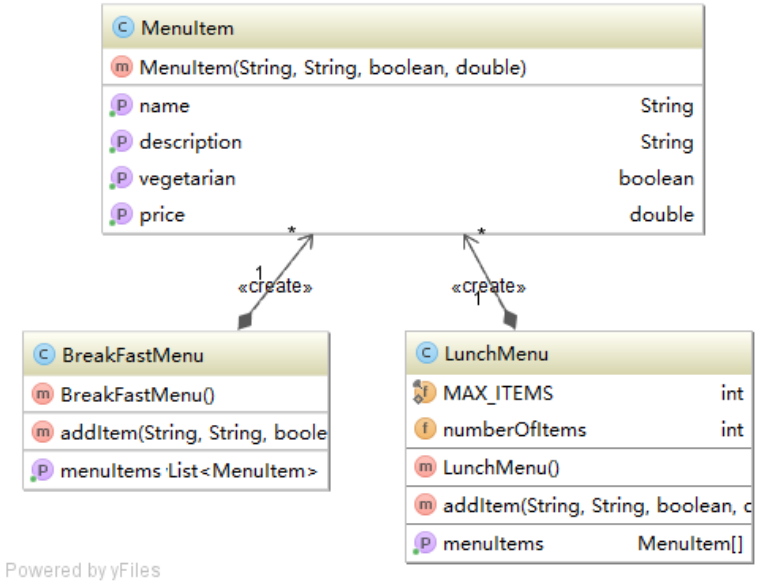
模板方法是一个算法内的步骤细节不同，策略模式通过组合使用不同的算法，总的来说模板方法控制的粒度更细，代码复用度更高；但是策略模式将会更有弹性，可以使用不同的算法相互替换（算法步骤不同）。

9.迭代器与组合模式（管理良好的集合）

场景问题 1：早餐餐厅与午餐餐厅合并，每个餐厅都有自己的菜单实现，早餐餐厅使用 ArrayList 实现菜单，午餐餐厅使用数组实现餐厅；类如下：

```
public class BreakFastMenu {
    ArrayList<MenuItem> menuItems;

    public BreakFastMenu() {
        this.menuItems =new ArrayList<>();
        addItem("建饼","一种很大的饼",true,3.12);
        addItem("包子","带馅儿的馒头",true,1.50);
    }
    public void addItem(String name, String description, boolean vegetarian, double price){
        MenuItem menuItem=new MenuItem(name,description,vegetarian, price);
        menuItems.add(menuItem);
    }
    public ArrayList<MenuItem> getMenuItems() {
        return menuItems;
    }
}
```

午餐类的代码跟上述类似。



问题 1：现在餐厅合并了，顾客需要一个女招待能够提供完整的菜单，那么怎么实现呢？一种办法就是创建一个女招待的对象，在方法里遍历 2 个菜单的项，打印出来。

```
public class Waitress {
    BreakFastMenu breakFastMenu=new BreakFastMenu();
    LunchMenu lunchMenu=new LunchMenu();
    void printMemu(){
        printBreakFastMemu();
        printLunchMenu();
    }
    void printBreakFastMemu(){
        List<MenuItem> breakFastMenus=breakFastMenu.getMenuItems();
        for(int i=0;i<breakFastMenus.size();i++){
            MenuItem menuItem=breakFastMenus.get(i);
            System.out.println(menuItem.getName()+",");
            System.out.println(menuItem.getDescription()+",");
            System.out.println(menuItem.getPrice());
        }
    }
    void printLunchMenu(){
        MenuItem[] lunchMenus=lunchMenu.getMenuItems();
        for(int i=0;i<lunchMenus.length;i++){
            MenuItem menuItem=lunchMenus[i];
            System.out.println(menuItem.getName()+",");
            System.out.println(menuItem.getDescription()+",");
            System.out.println(menuItem.getPrice());
        }
    }
```

每个菜单的内部实现方式都不相同，所以需要每个菜单都处理一下，得到合并的结果，如果新加入一个餐厅，还要再加入类似的代码，无疑这是不利于扩展的。女招待对象针对了具体实现编码，还有女招待知道每个餐厅内部菜单的存储方式，这也违反了封装。

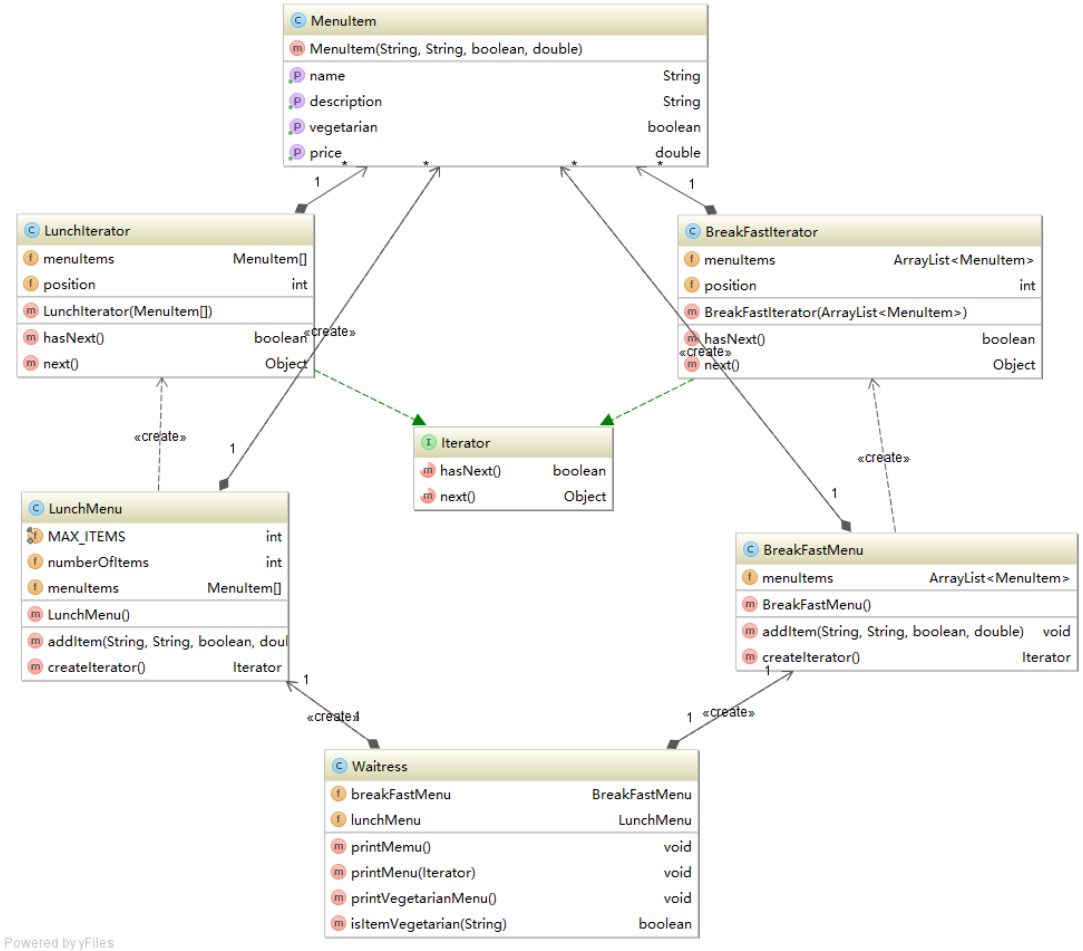
解决方案 1：如果每个菜单的实现都继承相同的接口，那么他们对外的表现行为就是一致的，女招待也就不需要知道其内部存储，这样满足了封装性要求，

但是对外表现一致的行为，女招待对象还是要知道每个餐厅的所有菜单项，这又涉及到了实现的细节，实现细节是不同的，暴漏细节也是不好的，所以我们要封装变化的部分，不同的部分就是每个餐厅菜单项的存储方式以及遍历方式不一致；但是他们遍历的行为大体是一致的，考虑用迭代器代替这种便利操作；

```
Iterator iterator=breakFastMenu.createIterator();
While(iterator.hasNext()){
    MenuItem menu=iterator.next();
}
```

因为 ArrayList 与数组从根本上来说都是集合类，也是列表，其共同点还是很多的。

这个就是迭代器模式，通常都要集成一个共同的接口，Iterator；实现这个接口时，我们内部定义了其具体存储内容的属性，这个就完成了对于具体细节的一个封装。定义好的女招待代码如下：



实质上，迭代器模式是对象的细节访问统一提供对外的访问接口。避免了直接暴漏其细节，统一了多个类似对象的访问细节行为。

上面的女招待类中，女招待的构造方法仍然是传入了 2 个具体类的实现，这

是耦合的，不利于拓展的，所以要将他们解耦，从实现类中抽象出他们需要的接口的行为，createIterator 就可以了。

迭代器模式定义：提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴漏其内部实现。

迭代器分为内部迭代器与外部迭代期，外部迭代器是由客户端代码决定遍历出来的元素做什么操作，内部迭代器是将操作传递到迭代器内部，遍历的时候，由迭代器调用操作。

单一责任原则，一个类应该只有一个引起变化的原因。这也是内聚的体现，一个模块与类要设计成只支持一组相关的功能，如果功能不相关，就是低内聚，要尽力做到高内聚。

Java 的集合类都实现了迭代器的接口，返回 iterator 的方法定义在 Collection 接口里，继承这个接口的 ArrayList、Vector、LinkedList、Stack 与 PriorityQueue 等集合类都实现了这个接口；java 5 中新增了 for/in 语法来支持遍历集合。

但是只要并入新的餐厅，女招待的打印菜单的方法就需要加入新的遍历，仍然需要进一步的解耦，并且，假如每个餐厅都提供了一份自己的甜点菜单，此时现有的菜单结构无法满足要求。因为甜点菜单项与目前菜单的菜单项是无法兼容的。现在需要某种树形结构，可以容纳菜单、子菜单和菜单项，并且可以遍历所有的菜单项及部分菜单项。

解决方案 1：组合模式解决，因为纯粹的迭代器已经解决不了现在问题的复杂维度；

组合模式的定义：允许你将对象组合成树形结构来表现“整体/部分”层次结构，组合能够让客户以一致的方式处理个别对象以及对象组合。这其实就是一个树型结构；一个节点可以包含叶节点，也可以包含其他的非叶节点，每个节点都是同一个类。这个就是组合模式。为了对叶子节点与非叶子节点的统一处理的的考虑，为他们建立一个共同的操作接口。

```
*/
public abstract class MenuComponent {
    public  void add(MenuComponent menuComponent){
        throw new UnsupportedOperationException();
    }
    public  void remove(MenuComponent menuComponent){
        throw new UnsupportedOperationException();
    }
    public  MenuComponent getChild(int i){
        throw new UnsupportedOperationException();
    }
    public  String getName(){
        throw new UnsupportedOperationException();
    }
    public  String getDescription(){
        throw new UnsupportedOperationException();
    }
    public  String getPrice(){
        throw new UnsupportedOperationException();
    }
    public  String isVegetarian(){
        throw new UnsupportedOperationException();
    }
    public  void print(){
        throw new UnsupportedOperationException();
    }
}
```

接着实现菜单项类也就是叶子节点类与非叶节点的菜单类，菜单类中包含一个 ArrayList 统一包含子菜单与子菜单项，print 方法如下：

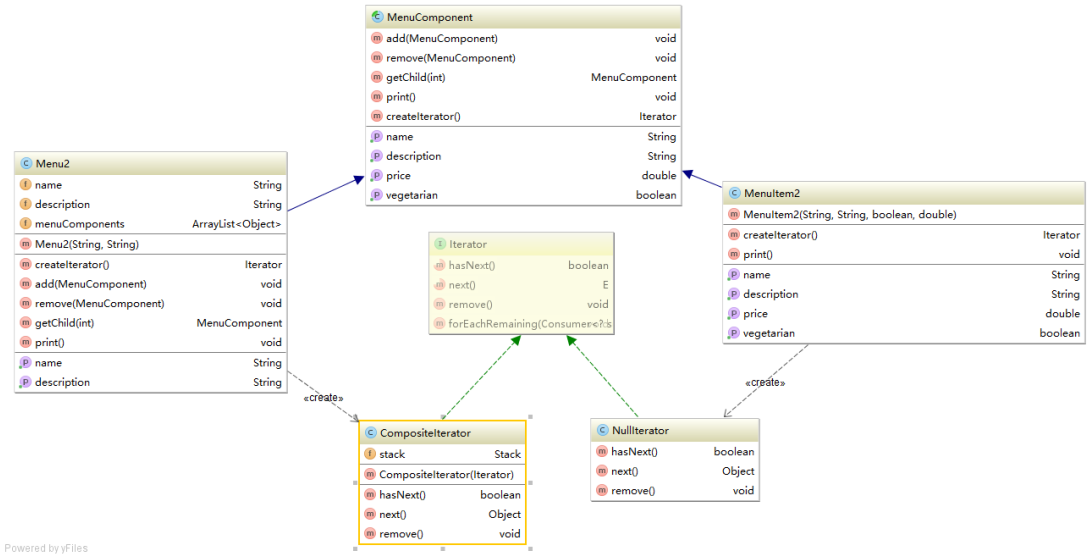
```
public void print() {
    System.out.println("\n        "+getName());
    System.out.println(","+getDescription());
    System.out.println("---------------------------------------");
    java.util.Iterator iterator=menuComponents.iterator();
    while(iterator.hasNext()){
        MenuComponent menuComponent= (MenuComponent) iterator.next();
        menuComponent.print();
    }
}
```

这里面使用了前面介绍的集合的迭代器。然后将顶层的节点传入到女招待的类中，直接掉用 print 就可以打印出所用的方法。这种遍历实际上是一种内部迭代器模式，因为在类中的 print 方法中以及实现了迭代以及所要进行的操作，这些都是在类内部实现的。也可以实现一个外部的迭代器，但是因为菜单项是一个单一的元素，所以不能返回一个真实存在的迭代器，可以返回 null，那么客户端代码就要判断返回值是否是 null，也可以返回一个迭代器，hasNext()方法永远返回 false；至于菜单类的迭代器，根据外部迭代器愿意，其内部要包含存储的树形结构数据，但是属性结构数据本身就是菜单的对象表示形式了，为了纯粹的存储数据而不是存储遍历的逻辑，采用了栈的数据结构。

```java
public class NullIterator implements java.util.Iterator {
    @Override
    public boolean hasNext() {
        return false;
    }

    @Override
    public Object next() {
        return null;
    }

    @Override
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

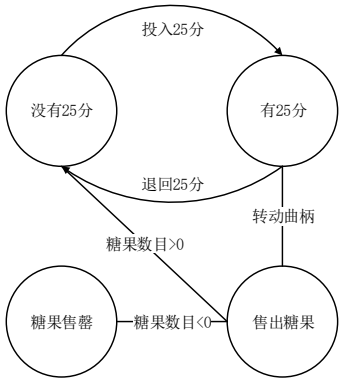
```java
public class CompositeIterator implements java.util.Iterator {
    Stack stack=new Stack();

    public CompositeIterator(Iterator menuComponent) {
        stack.push(menuComponent);
    }
    @Override
    public boolean hasNext() {
        if(stack.isEmpty()){
            return false;
        }else{
            Iterator iterator= (Iterator) stack.peek();
            if(!iterator.hasNext()){
                stack.pop();
                return hasNext();
            }else{
                return true;
            }
        }
    }

    @Override
    public Object next() {
        if(hasNext()){
            Iterator iterator= (Iterator) stack.peek();
            MenuComponent component= (MenuComponent) iterator.next();
            if(component instanceof  Menu2){
                stack.push(component.createIterator());
            }
            return component;
        }else{
            return null;
        }
    }

    @Override
    public void remove() {

    }
}
```

结构化的类图如下：



10 状态模式（事物的状态）

问题 1：糖果机的状态图，用代码实现。



这是一个状态机，里面有状态有操作。转化成代码：

1.找出所有的状态，2.创建一个实例变量持有目前的状态，3.发生的动作整合起来。建立一个类，里面用属性表示没种状态，没种行为化为一个方法，方法里对状态进行判断并到达下一个方法。

问题 2：假如，转动曲柄发出糖果时，有 10%的几率发出 2 颗，此时还用上面的方法写代码时，就需要更多的逻辑判断，导致代码臃肿。

解决方案：封装变化，把状态的行为局部化，每个状态一个类，封装他自己的动作，这样也会方便添加新的状态；首先，定义一个 state 接口，接口内定义所有的糖果机动作，为每个状态实现状态类，并书写对应状态下的动作，将动作委托到状态类。

建立一个总的状态机对象，里面只负责状态的转变。

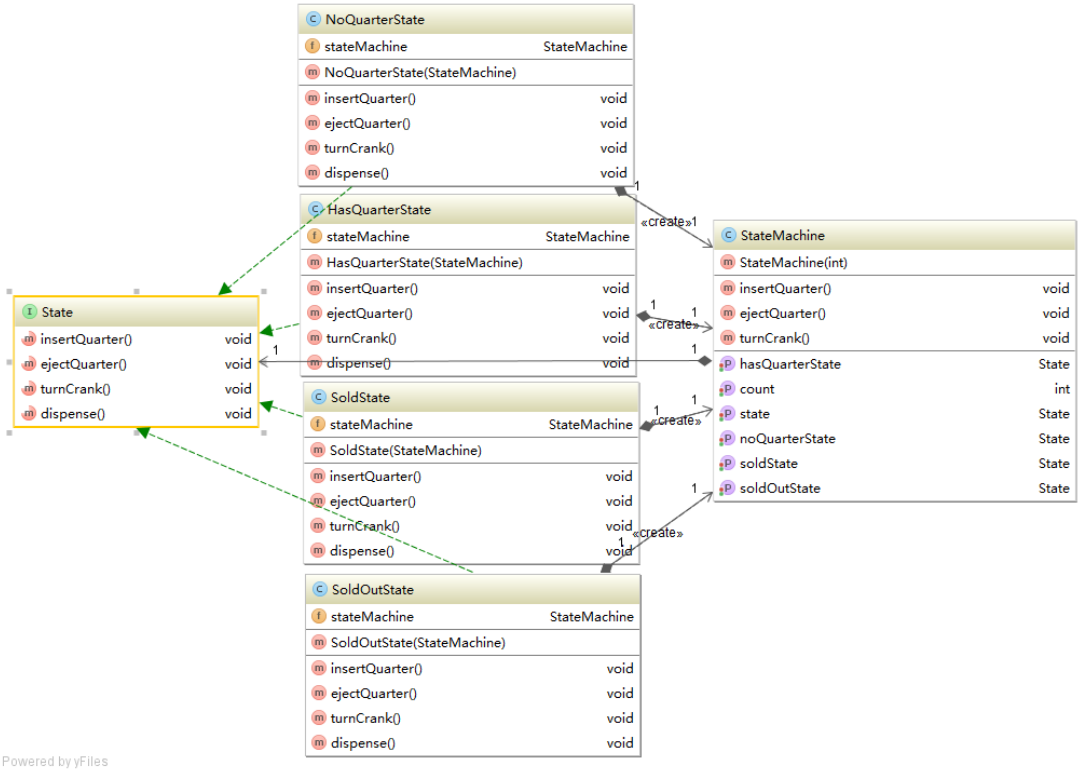
其中一个状态机的内部实现如下图：

```java
public class StateMachine {
    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State state=soldOutState;
    int count=0;

    public StateMachine(int count) {
        this.soldOutState =new SoldOutState(this);
        this.noQuarterState = new NoQuarterState(this);
        this.hasQuarterState = new HasQuarterState(this);
        this.soldState = new SoldState(this);
        this.count = count;
    }
    public void insertQuarter(){
        state.insertQuarter();
    }
    //退回25分钱的动作
    public void ejectQuarter(){
        state.ejectQuarter();
    }
    //转动曲柄
    public void turnCrank(){
        state.turnCrank();
    }
    //发放糖果
    public void dispense(){
        state.dispense();
    }
    public void setSoldOutState(State soldOutState) {
        this.soldOutState = soldOutState;
    }
```

状态机内部存储了所有的状态及当前的状态，在构造器内创建所有的状态，并将状态机本身传入到状态中，方便更改状态机的状态。

```java
public class NoQuarterState implements State {
    StateMachine stateMachine;

    public NoQuarterState(StateMachine stateMachine) {
        this.stateMachine = stateMachine;
    }

    @Override
    public void insertQuarter() {
        System.out.println("已经投入了20分钱，请转动把手，拿出糖果");
        stateMachine.setSoldState(stateMachine.getHasQuarterState());
    }

    @Override
    public void ejectQuarter() {
        System.out.println("没有投入钱币，不能执行退回操作");
    }

    @Override
    public void turnCrank() {
        System.out.println("没有投入钱币，不能转动把手");
    }

    @Override
    public void dispense() {
        System.out.println("没有投入钱币，不能发出糖果");
    }
}
```

单一类的不同状态下不同行为的模式，将所有的状态抽象出来，每个状态代表当前状态下的状态机，所有状态内部包含了状态机这个属性；同时状态机有抽象的包含其所拥有的所有的状态，但是状态机每时每刻只能处于一个状态中，这与现实的对象是符合的。



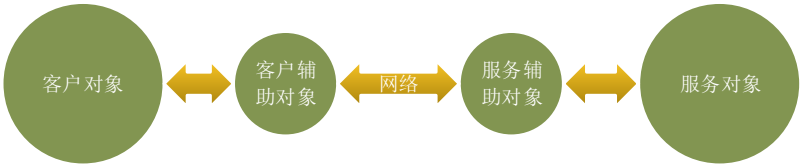
状态机模式的定义：允许对象在内部状态改变时改变它的行为，对象看起来好像修改了他的类。

11.代理模式(控制对象的访问)

问题 1：糖果机有很多台，分布在全国各地，公司需要在某事某刻知道糖果机的售卖状态。

解决方案 1：创建一个类，糖果监视器类；用于产生糖果机的状态报告，但是这种方式糖果机与监视器运行在一个 JVM 上，监视器不能监视其他地方的糖果机；所以要创建基于网络的糖果机监视器；这就需要代理，在糖果机监视器内部有远程的糖果机对象的代理，对本地糖果机对象的访问都将通过网络访问到真实的糖果机对象，代理就是代表某个真实的对象；代理对象与真实对象往往运行在不同机器的不同地址空间内；Java 已经内置了 RMI 工具可以实现远程代理。

RMI 简单的介绍：



远程调用需要 2 个辅助对象完成底层的网络传输调用等，使得上层的调用运行就像在本地一样，辅助对象已经由 RMI 工具本身提供，客户辅助对象与服务对象会由相同方法，用于代理访问；RMI 也提供查找服务的功能，远程调用会发生一些网络异常，需要在调用的方法上加入相关的异常处理。

RMI 将客户辅助对象称为 stub（桩），服务辅助对象称为 skeleton（骨架）。

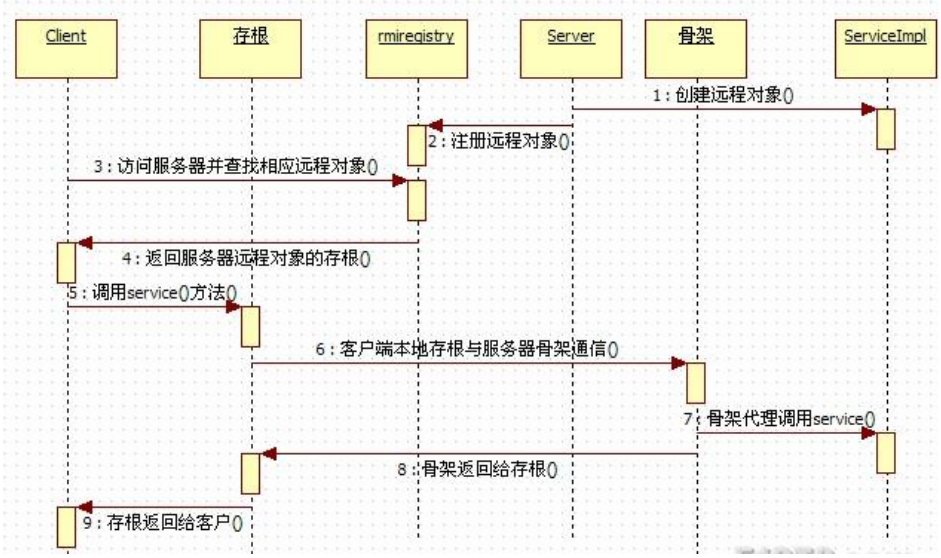
步骤一：制作远程接口（stub 与服务对象实现此接口），一般都要继承 Remote 类远程方法的变量与返回值都要是可序列化的；

步骤二：制作远程实现（服务对象实现），要使对象具有远程功能，最简单的是继承 UnicastRemoteObject 类，这样才能保证客户端访问获得远程对象时，该远程对象将会把自身的一个拷贝以 Socket 的形式传输给客户端，此时客户端所获得的这个拷贝称为“存根”，而服务器端本身已存在的远程对象则称之为“骨架”。其实此时的存根是客户端的一个代理，用于与服务器端的通信，而骨架也可认为是服务器端的一个代理，用于接收客户端的请求之后调用远程方法来响应客户端的请求。；

步骤三：使用 rmic 产生 stub 与 skeleton（生成辅助对象）；

步骤四：启动 RMI registry（rmiregistry）可以查找到客户辅助对象；

步骤五：服务对象运行，就会在 RMI registry 注册一个代理。



代理模式的定义：为另一个对象提供一个替身或占位符以控制对这个对象的访问；代理的几种应用场景：

1.远程代理控制访问远程对象；

2.虚拟代理控制访问创建开销大的资源；

3.保护代理基于权限控制对资源的访问。

代理与被代理对象往往实现了同样的接口，这种是接口代理；代理内部往往含有被代理对象的引用。

虚拟代理：直到真正需要一个对象的时候才创建，没有时，由虚拟代理扮演对象的替身。

代理模式是装饰者模式的一个子集，但是在网络上，他们又不同。

12 复合模式（设计模式组合）

复合模式具有一般性，以解决一般或者重复发生的问题；例子是鸭子的模式广泛使用的一个复合模式是 MVC；

13 真实世界中的模式

模式就是在某种情境下，针对某问题的某种解决方案。

根据模式的目标的不同分为 3 类：创建型、行为型、结构型。

创建型模式就是实例化对象、行为型模式是有关于对象之间的交互与职责分配、结构型就是类的组织结构与行为型不同的是这种组织是分层次的，行为型是平等的。

14 其他设计模式

桥接模式：将实现与抽象放到 2 个不同的类层次中使他们可以独立改变。

生成器模式：封装一个产品的构造过程，并允许按步骤构造。

责任链模式：顺序处理请求。

蝇量模式：

解释器模式：

中介者模式：添加一个中介者，用来为多个对象之间的交互增加一个调度机制。

备忘录模式：

原型模式：

访问者模式：