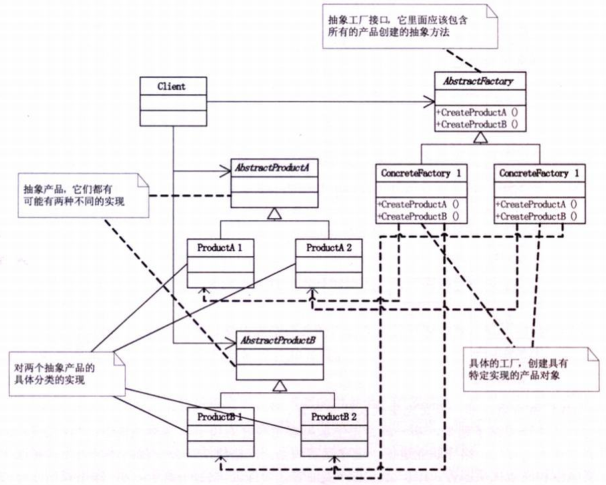
# 1创建型：

## 1.1抽象工厂（abstractFactory）

### 概念

提供个创建一系列或相关依赖对象的接口，而无需指定他们具体的类。

### 结构图

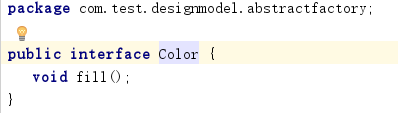


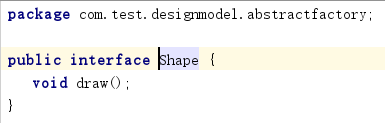
### 代码

#### 客户端



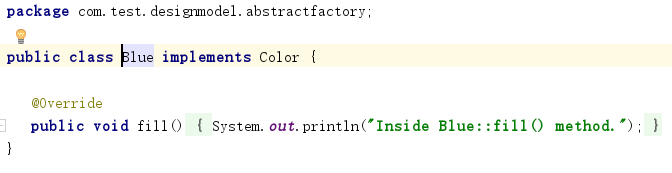
#### 产品类

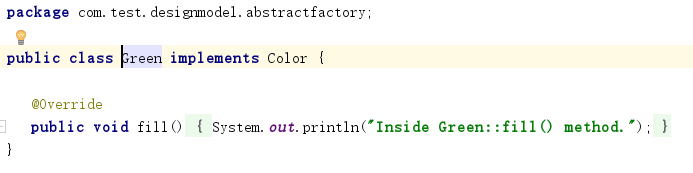


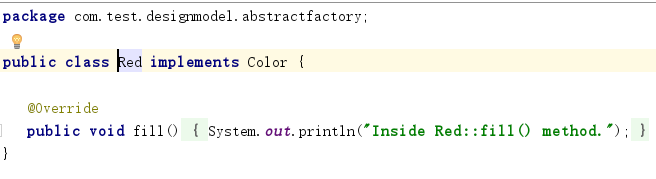


#### 产品具体实现

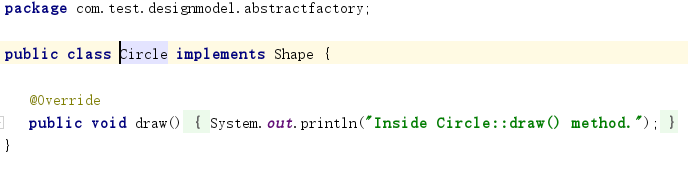
##### Color

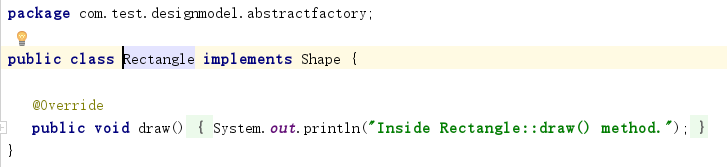


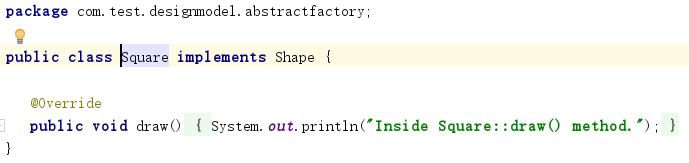




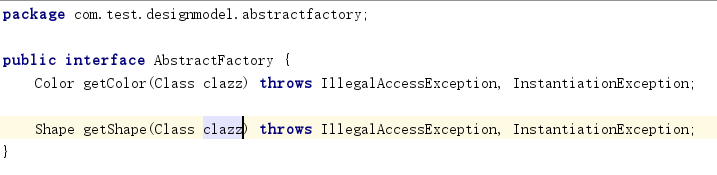
##### Shape



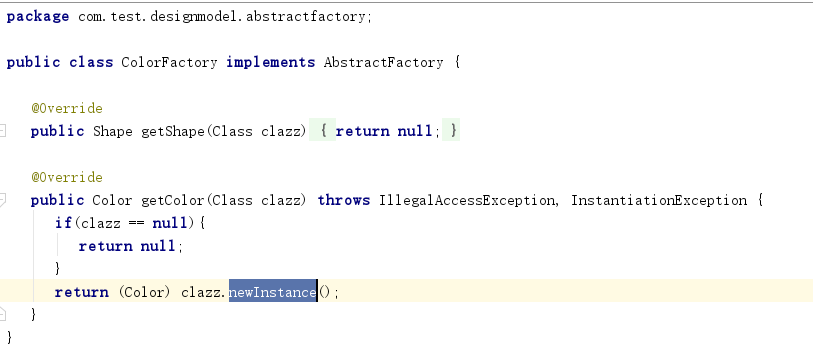


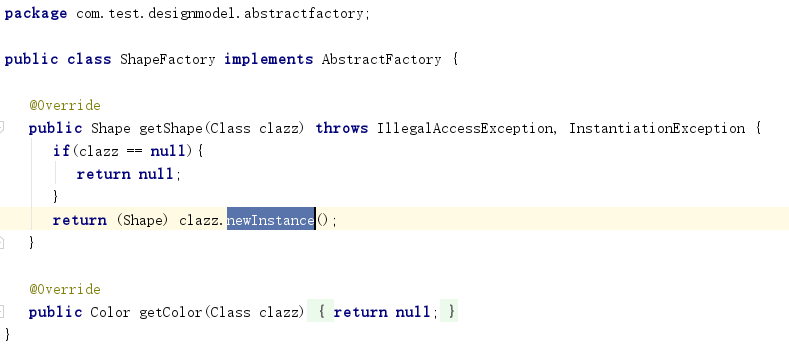


#### 抽象工厂接口

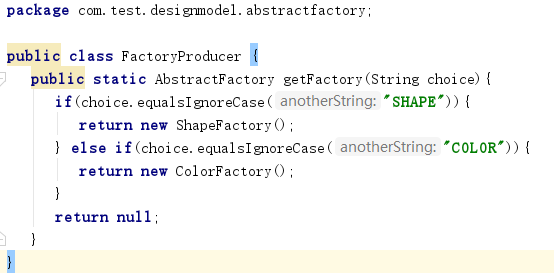


#### 具体工厂





#### 简单工厂生成指定具体工厂

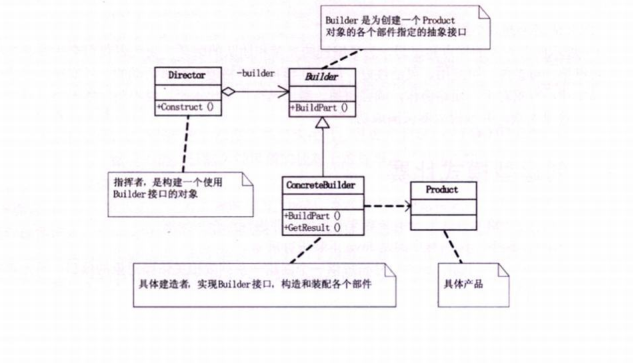


## 1.2建造者（builder）

### 概念

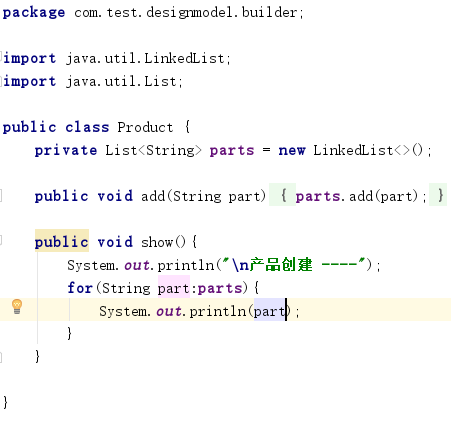
将一个复杂对象的结构与它的表示分离，使得同样的构建过程可以创建不同的表示。

### 结构图

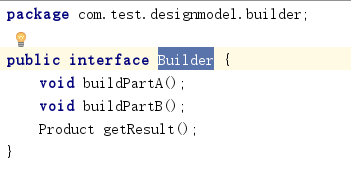


### 代码

#### 具体产品

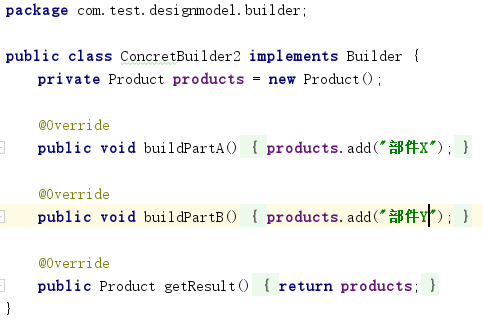


#### 创建产品的接口

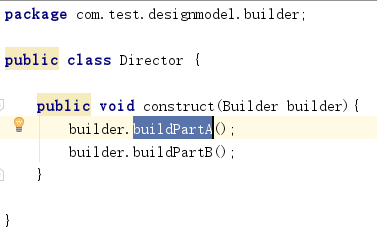


#### 具体创建者

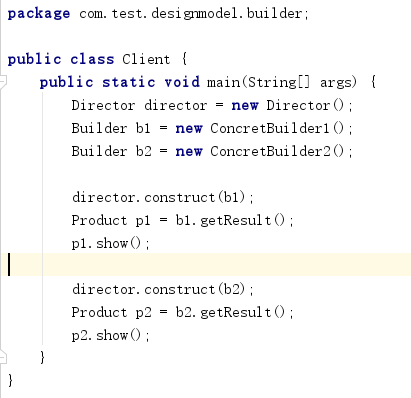




#### 指挥者



#### 客户端

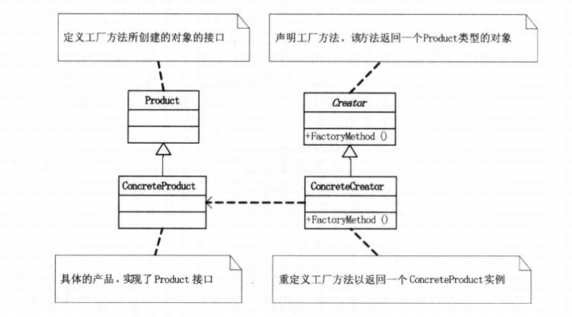


## 1.3工厂方法（factoty method）

### 概念

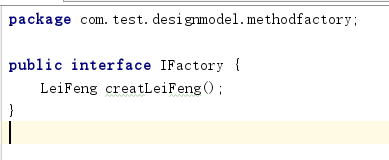
定义一用于创建对象的接口，让子类决定实例化哪一个类，工厂模式使一个类的实例化延迟到其子类。

### 结构图

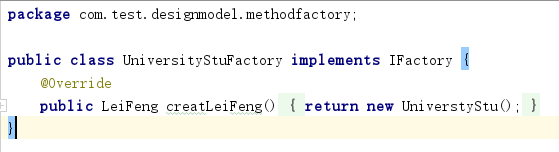


### 代码

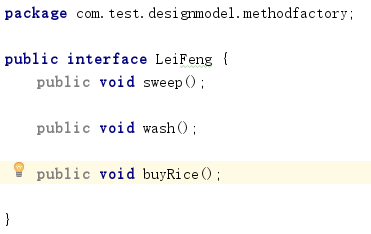
#### 工厂方法接口



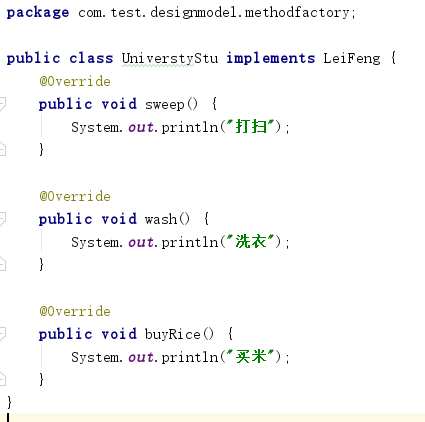
#### 具体工厂



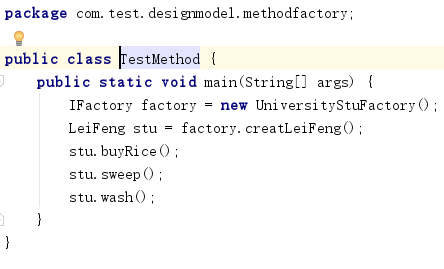
#### 产品的接口



#### 具体产品



客户端

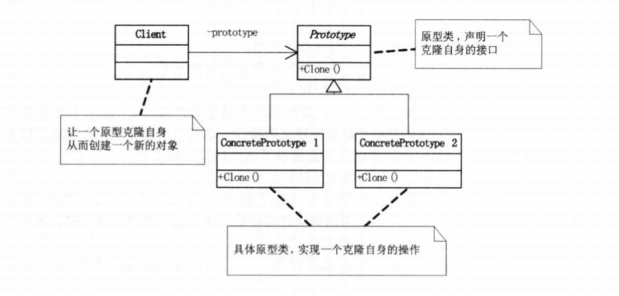


## 1.4原型（prototype）

### 概念

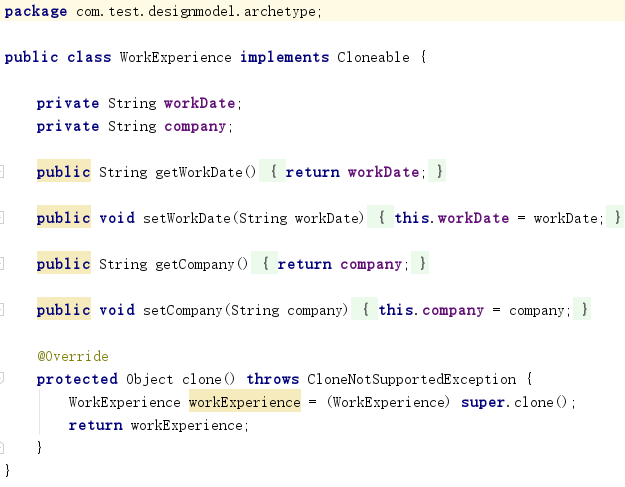
用原型实例指定创建对象的种类，并通过拷贝这些原型创建新的对象。

### 结构图



### 代码

#### 原型类





#### 客户端



## 1.5单例（singleton）

### 概念

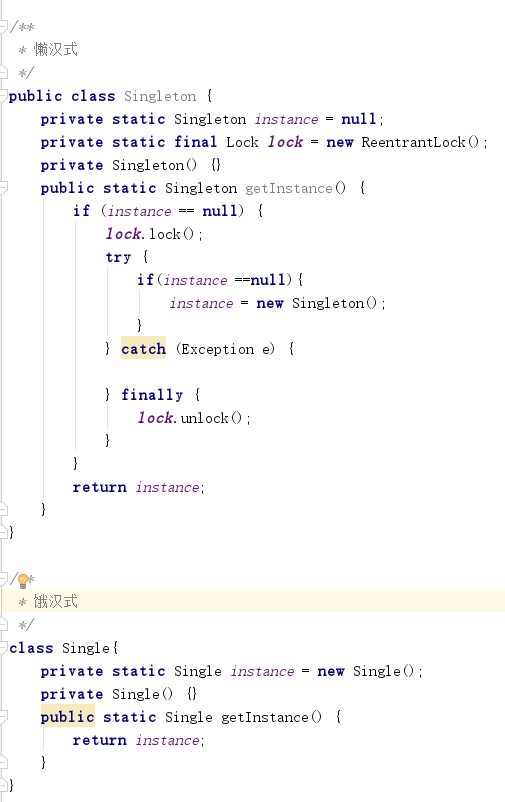
保证一个类仅有一个实例，并提供一个访问它的全局访问点。

### 结构图

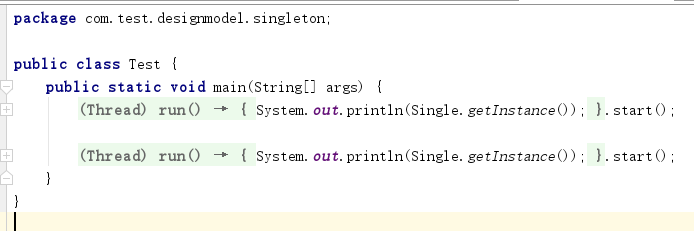


### 代码

#### 两种方式



#### 客户端



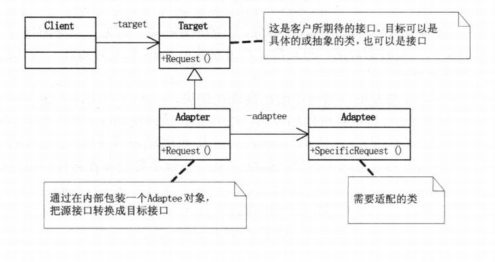
# 2.结构型：

## 2.1适配器（adapter）

### 概念

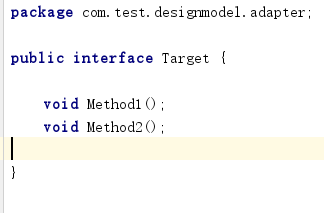
将一个类的接口转换成客户希望的另一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

### 结构图

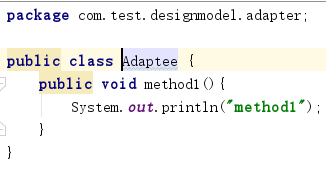


### 代码

#### 目标接口



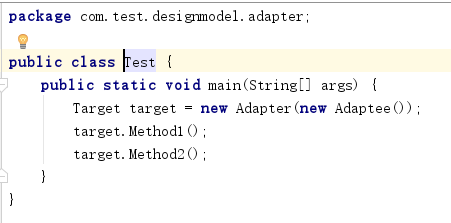
#### 待适配的目标



#### 适配器



#### 客户端

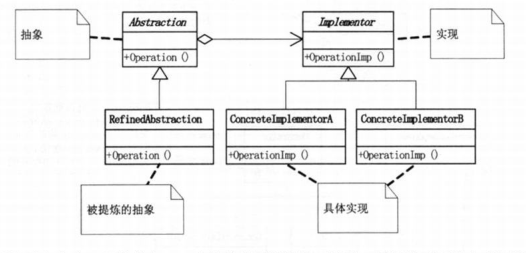


## 2.2桥接（bridge）

### 概念

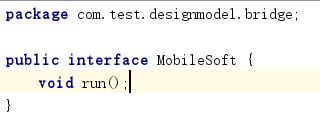
将抽象部分与它的实现部分分离，是它们都可以独立的变化。

### 结构图

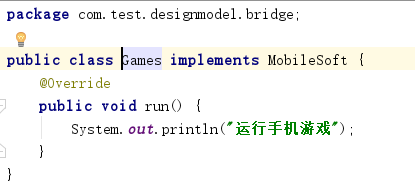


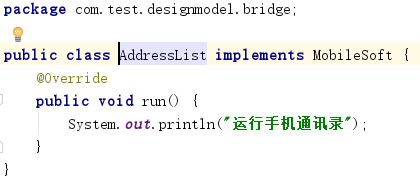
### 代码

#### 桥接接口

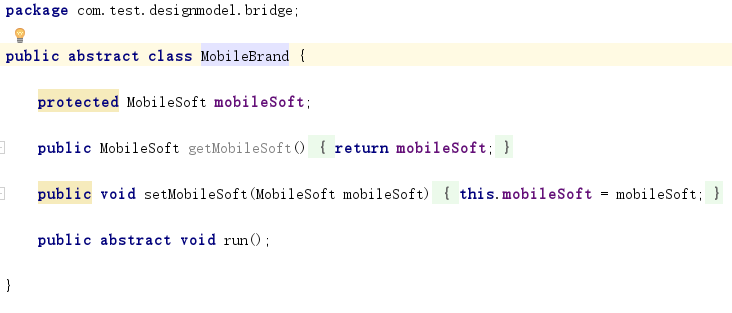


#### 具体桥接类

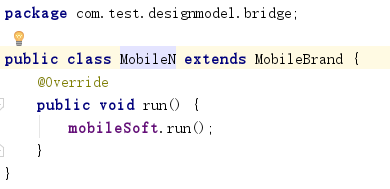


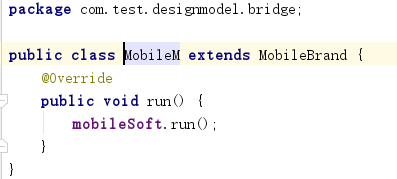


#### 抽象类



#### 抽象实现类



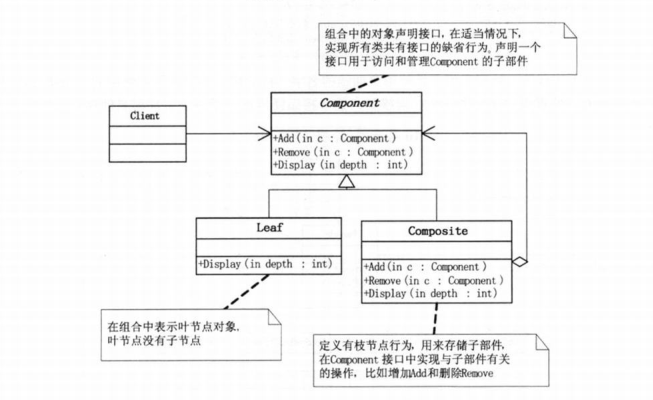


## 2.3组合（composite）

### 概念

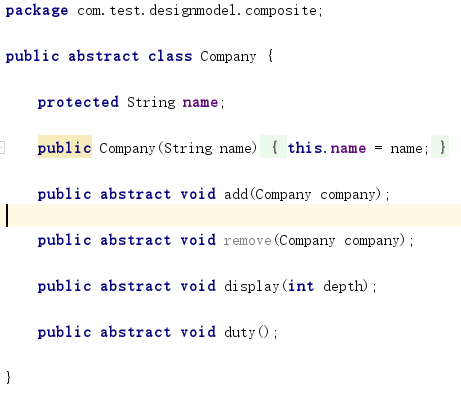
将对象组合成树形结构以表示“部分-整体”额层次结构，组合模式使得用户对单个对象和组合对象的使用具有一致性。

### 结构图



### 代码

#### 声明抽象类



#### 叶子结点





#### 树枝

#### 客户端

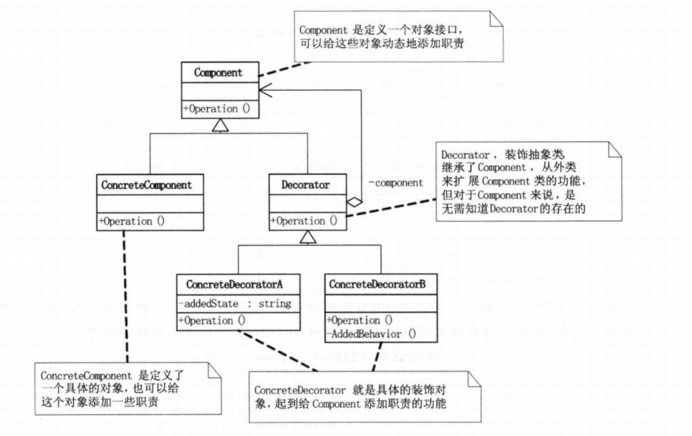


## 2.4装饰（decorator）

### 概念

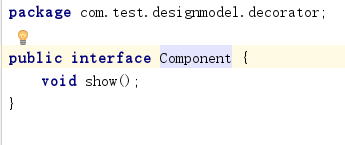
动态的给一个对象添加一些额外的职责。就添加功能来说，装饰模式相比生成子类更加灵活。

### 结构图

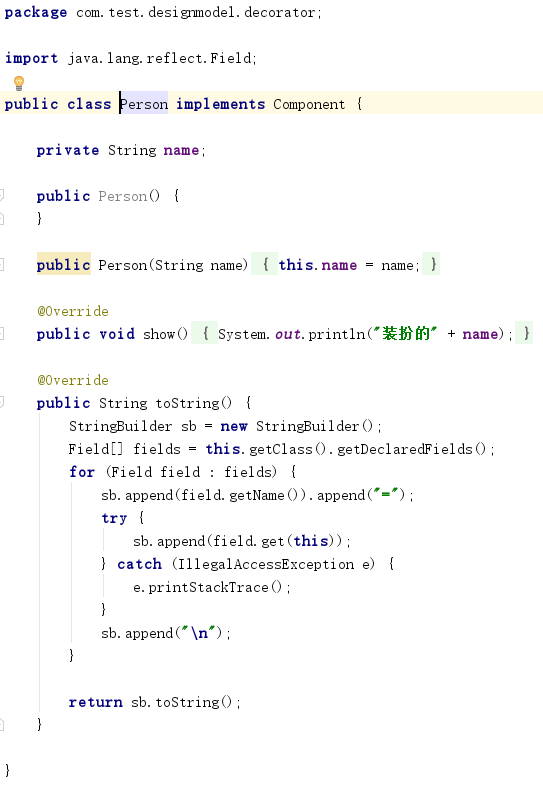


### 代码

#### 对象接口

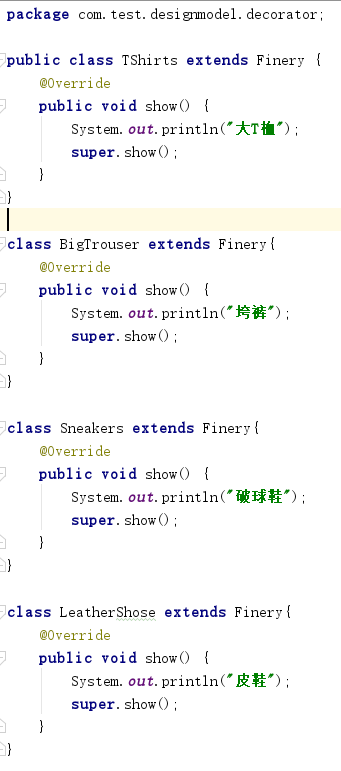


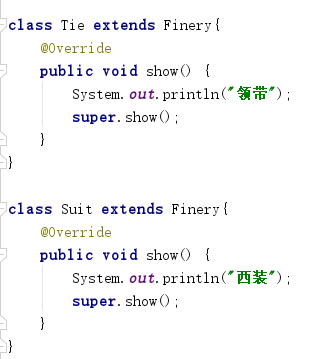
#### 具体对象



#### 装饰器







#### 客户端



## 2.5外观或门面（facade）

### 概念

为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

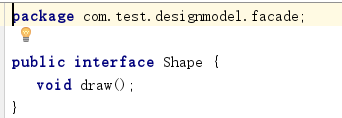
### 结构图



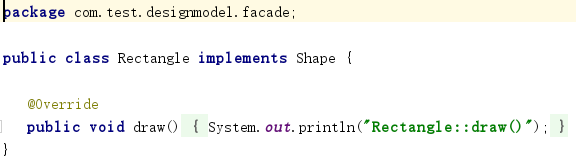
### 代码

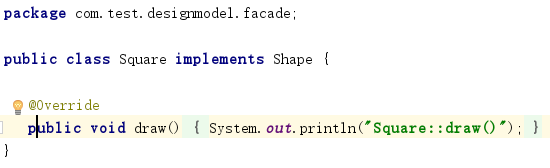
#### 子系统

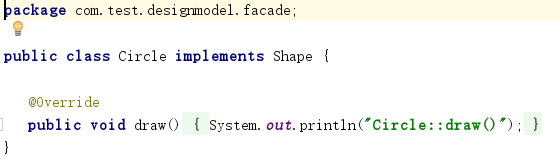
##### 接口



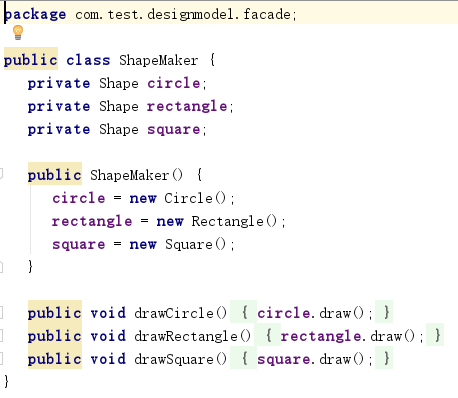
##### 实现类



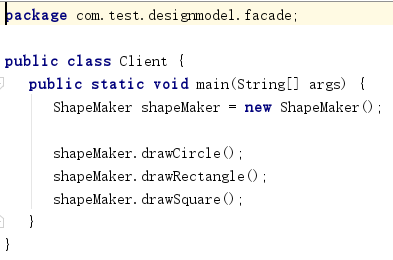




#### 外观类



#### 客户端

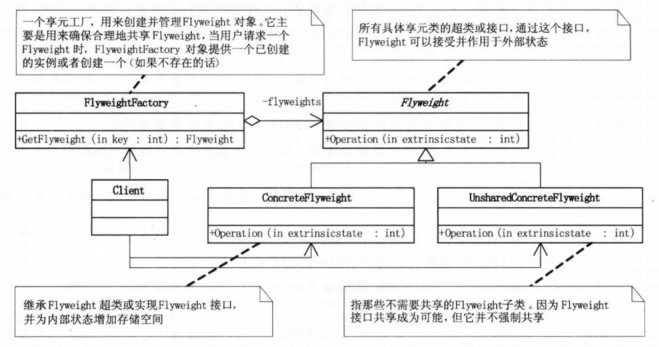


## 2.6享元（flyweight）

### 概念

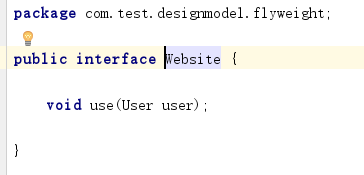
为运用共享技术有效地支持大量细粒度的对象。

### 结构图

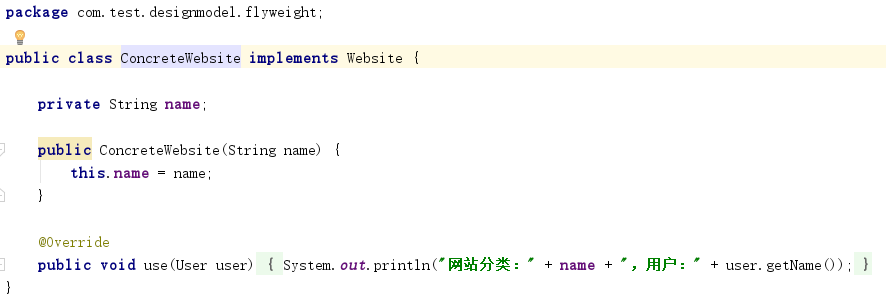


### 代码

#### 接口



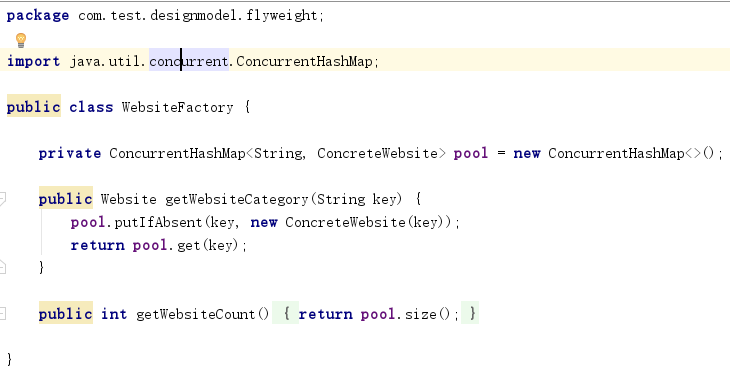
#### 实现类



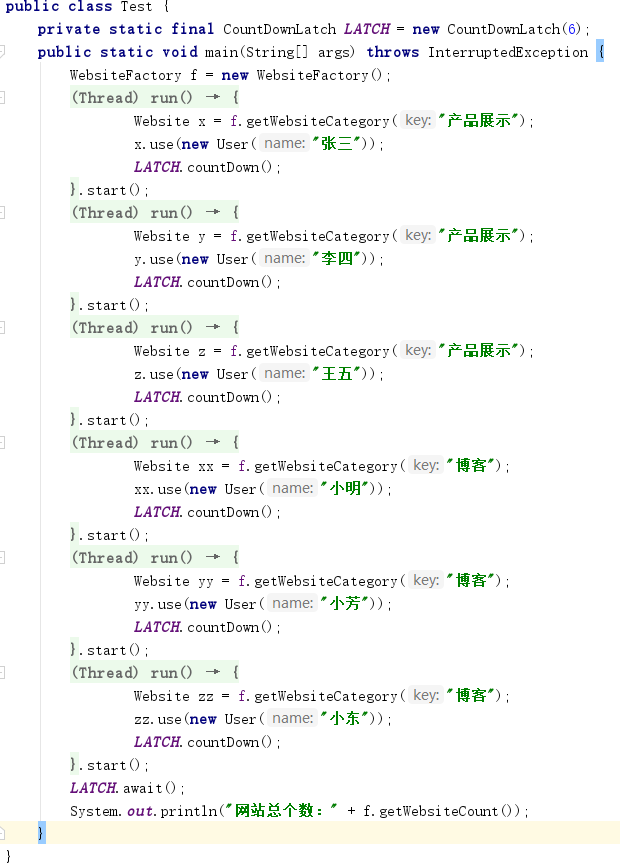
#### 外部



#### 享元工厂



客户端

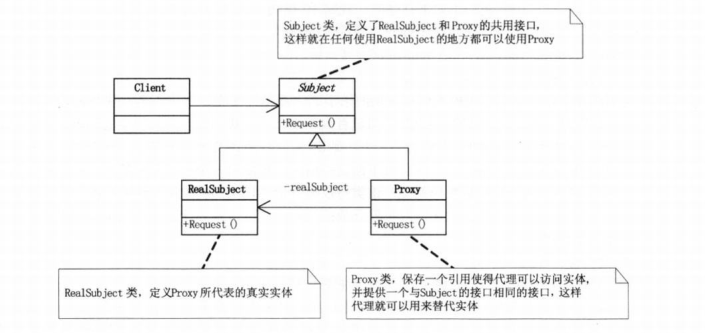


## 2.7代理（proxy）

### 概念

为其他对象提供一种代理以控制对这个对象的访问。（动态【需要用到反射里面Proxy类和InvocationHandler】和静态代理）

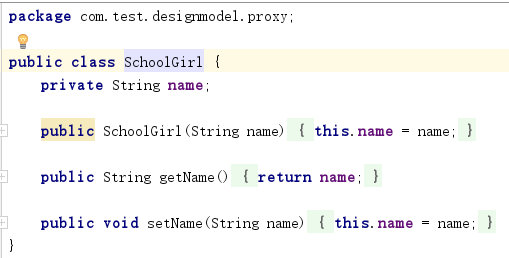
### 结构图



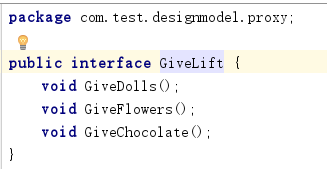
### 代码

#### 静态代理

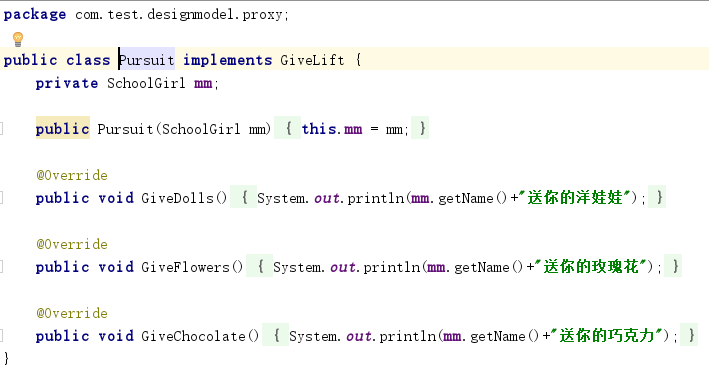
##### 目标对象



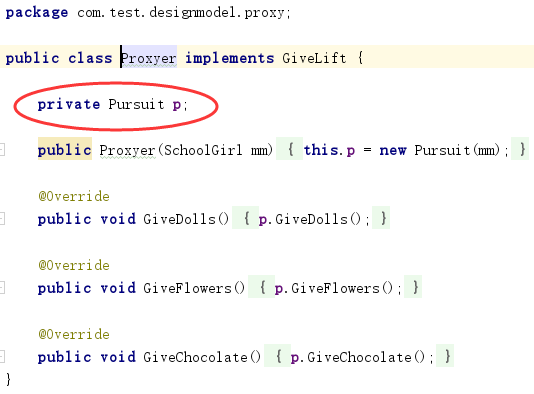
##### 代理主题



##### 被代理者



##### 代理人

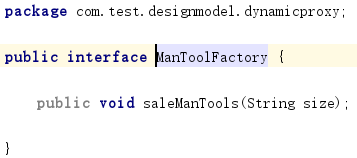


##### 客户端



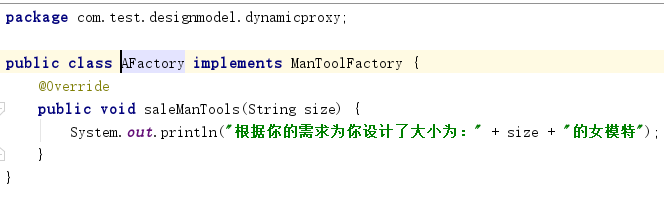
#### 动态代理

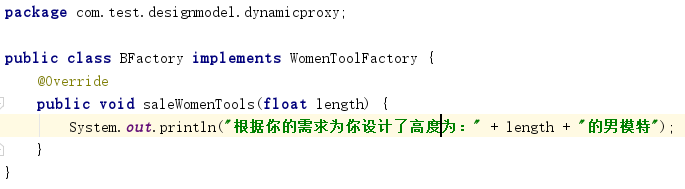
##### 代理工厂





##### 具体工厂实现类





##### 代理人



##### 客户端



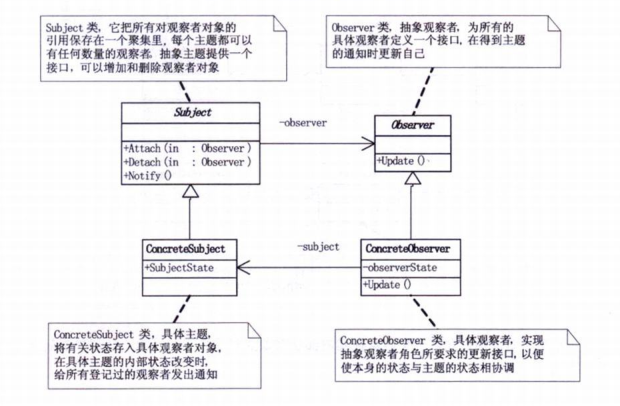
# 3.行为型：

## 3.1观察者（Observer）

### 概念

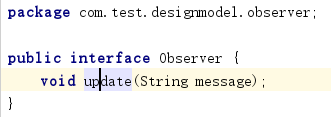
定义对象中的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖它的对象都得到通知并自动更新。

### 结构图

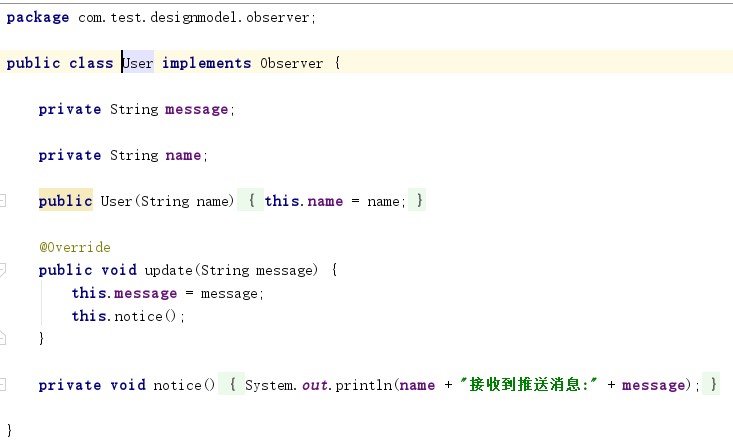


### 代码

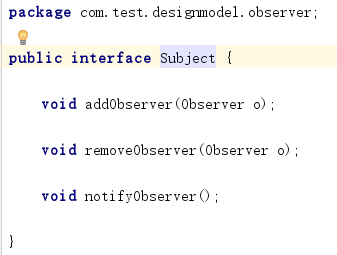
#### 抽象观察类



#### 实际观察者



#### 主题类



#### 具体主题类



#### 客户端

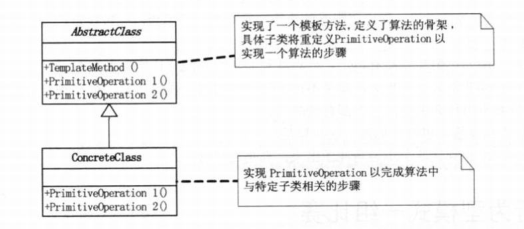


## 3.2模板方法（template method）

### 概念

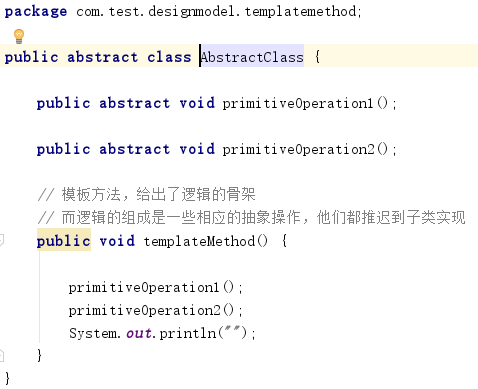
定义一个操作的算法骨架，而将一些步骤延迟到子类中，模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

### 结构图



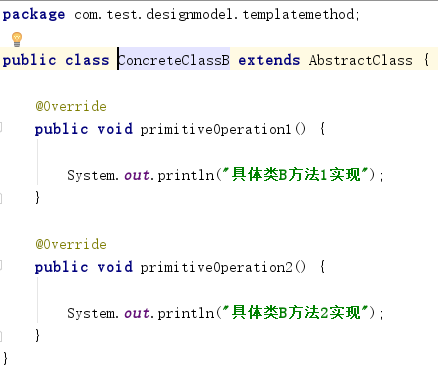
### 代码

#### 抽象类

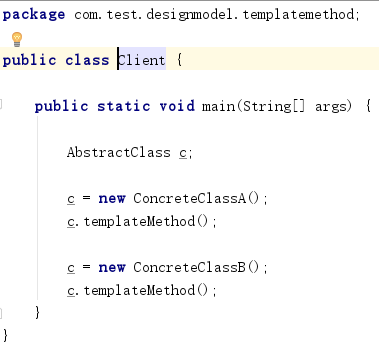


#### 子类





#### 客户端

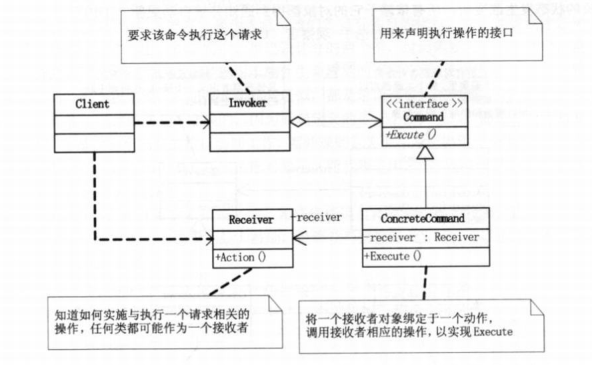


## 3.3命令（command）

### 概念

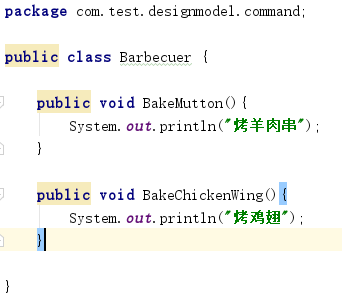
将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化：可以对请求排队或记录请求日志，以及支持可撤销的操作。

### 结构图

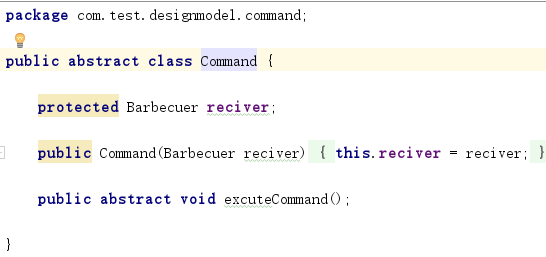


### 代码

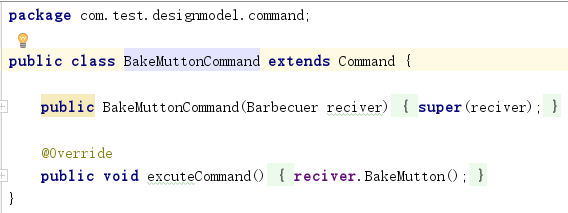
#### 实施者

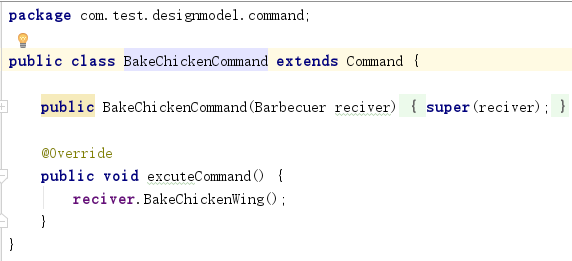


#### 命令接口



#### 具体实施





#### 指挥者



#### 客户端

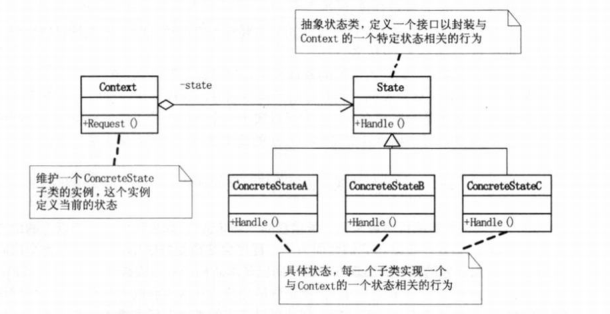


## 3.4状态（state）

### 概念

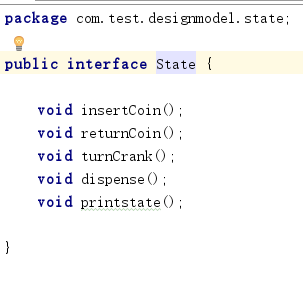
允许一个对象在其内部状态改变时改变它的行为，让对象看起来似乎修改了它的类。

### 结构图



### 代码

#### 状态接口



#### 具体状态

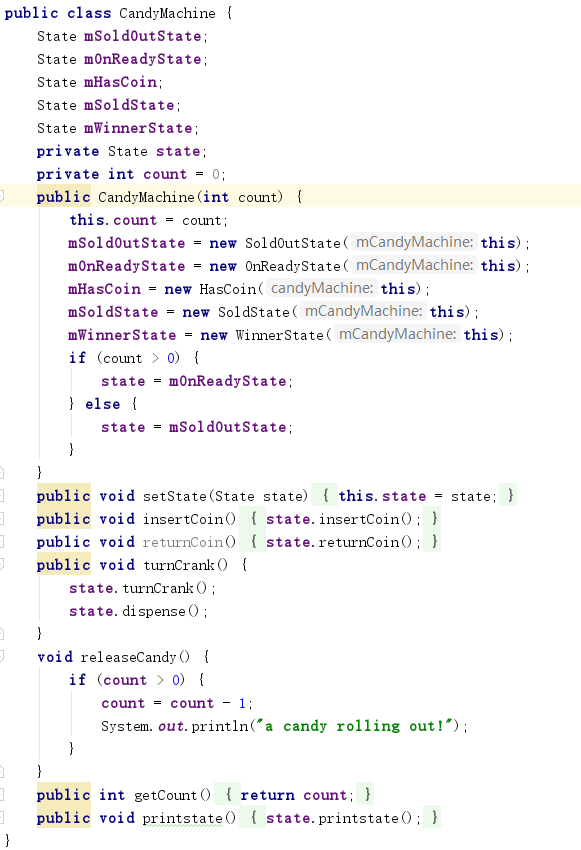












#### 客户端



## 3.5责任链（chain of responsibility）

### 概念

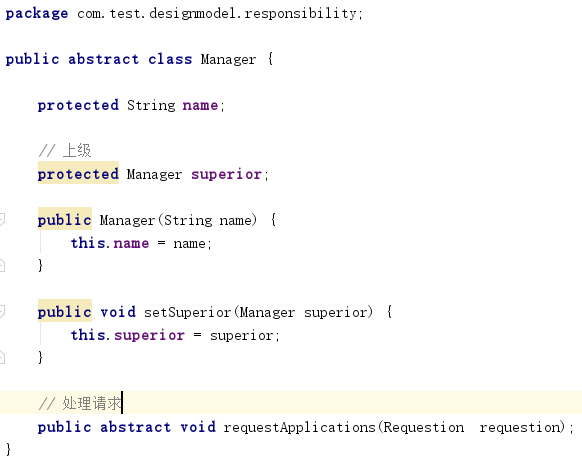
使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连城一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

### 结构图



### 代码

#### 抽象类



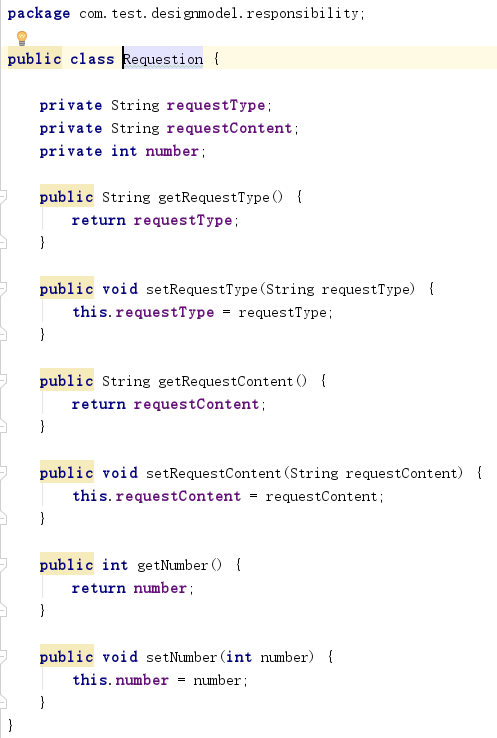
#### 具体处理者



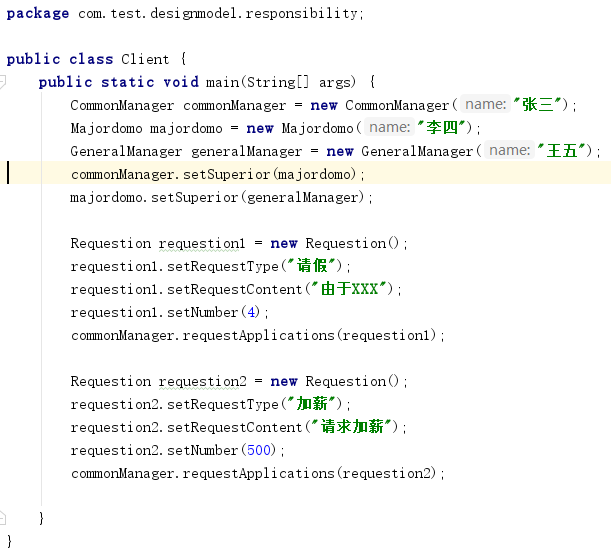




#### 具体请求



#### 客户端

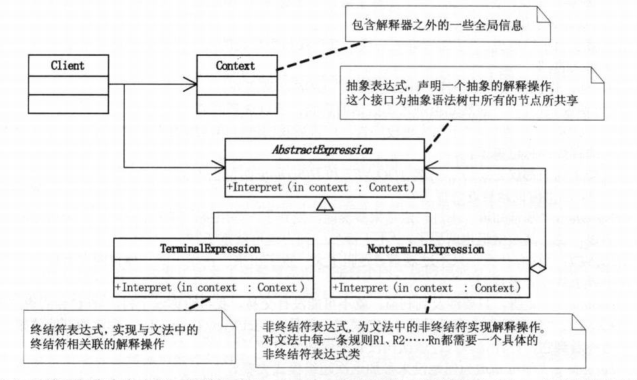


## 3.6解释器（interpreter）

### 概念

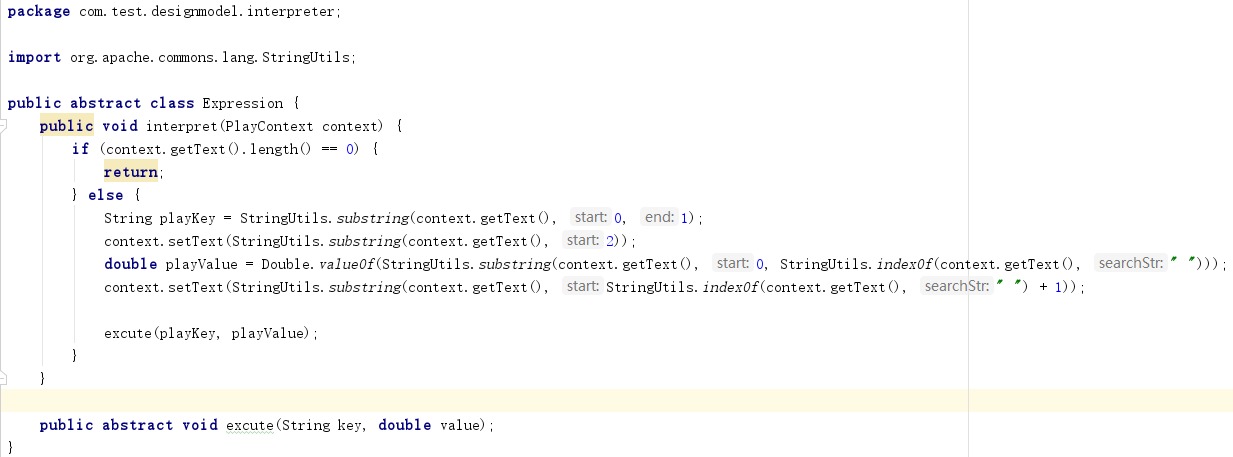
给定一种语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

### 结构图

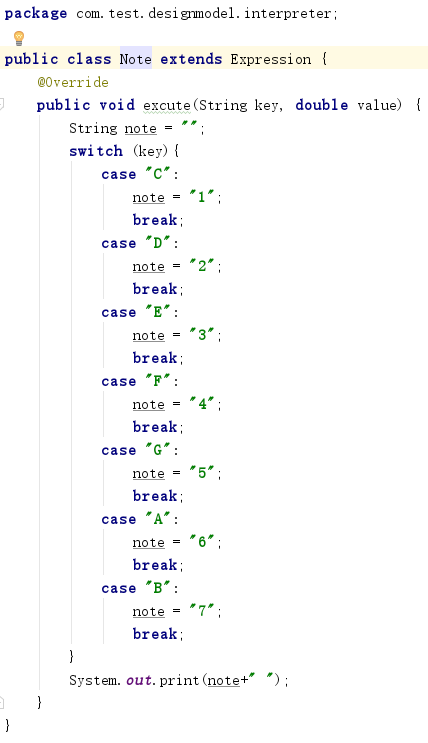


### 代码

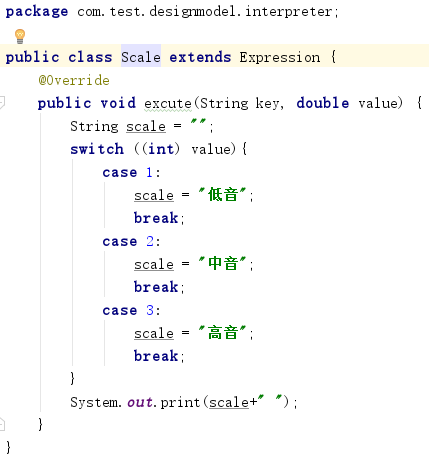
#### 抽象表达式



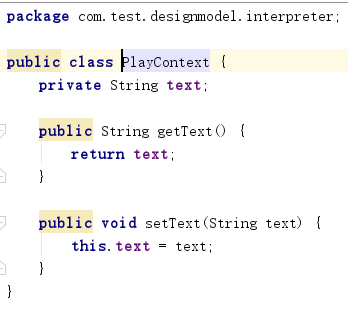
#### 非终结符表达式



终结符表达式



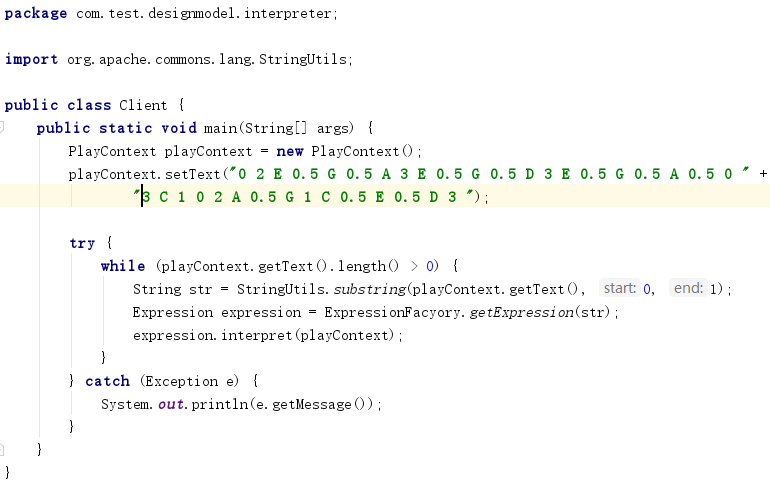
#### 解释的内容



#### 抽象表达式工厂



#### 客户端

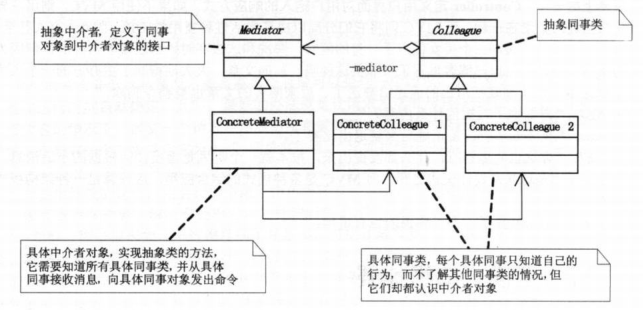


## 3.7中介者（mediator）

### 概念

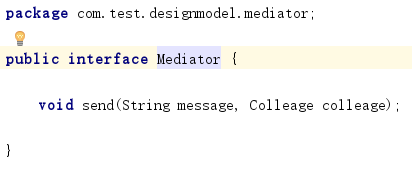
用一个中介对象来封装一系列的对象交互。中介者使各个对象不需要显示的相互引用，从而使其耦合松散，而且可以独立地改变他们之间的交互。

### 结构图

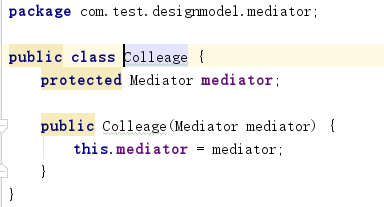


### 代码

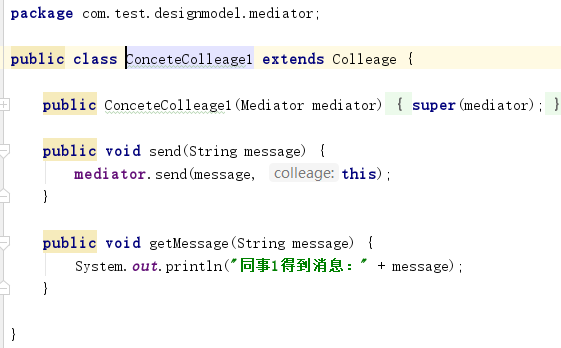
#### 中介

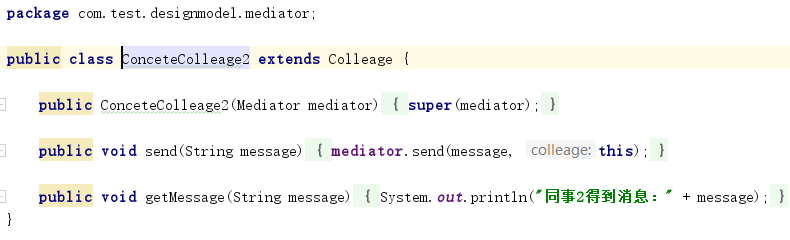


#### 抽象同事类



#### 具体同事类

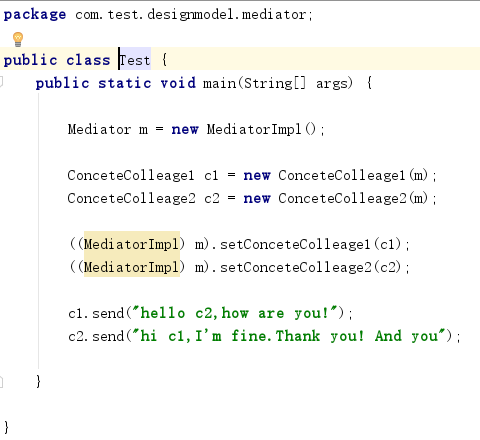




#### 具体中介



#### 客户端

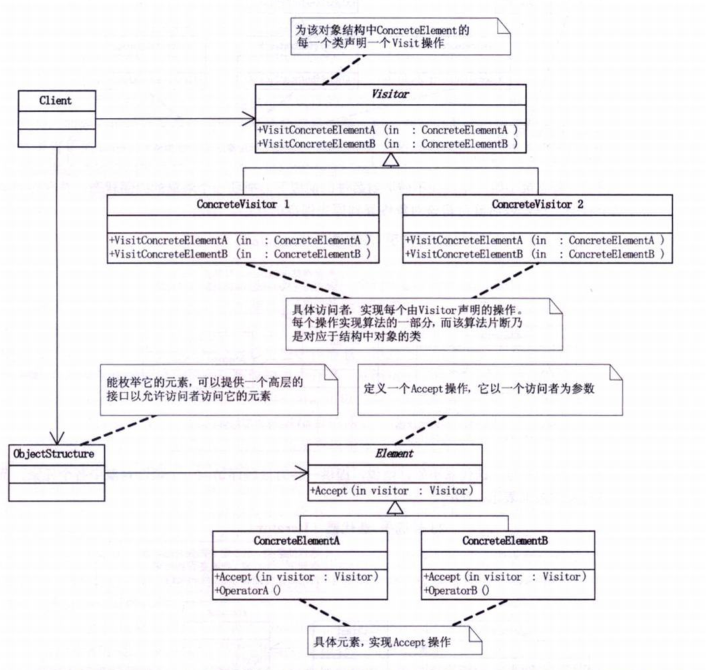


## 3.8访问者（visitor）

### 概念

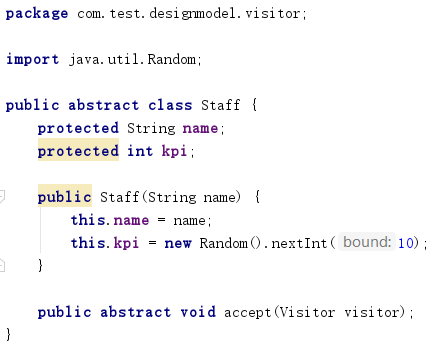
表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变元素的类的前提下定义作用于这些元素的新操作。

### 结构图

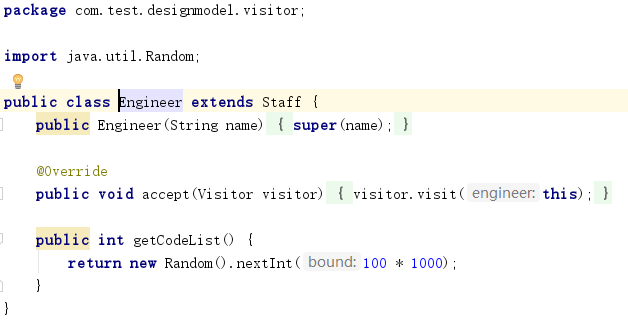


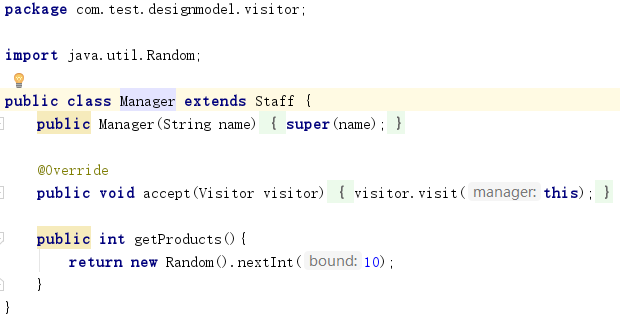
### 代码

#### 抽象元素类

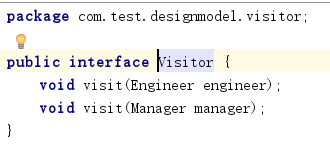


#### 具体元素类

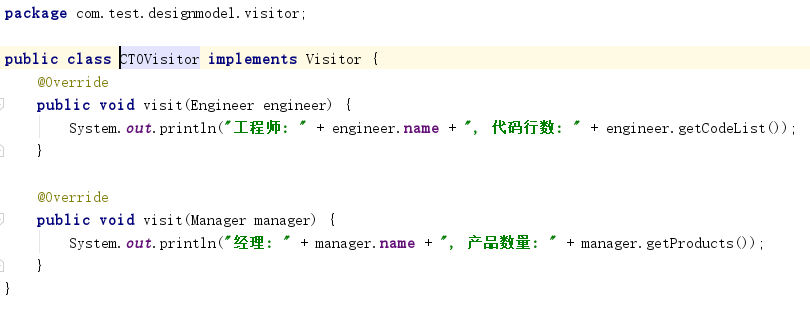


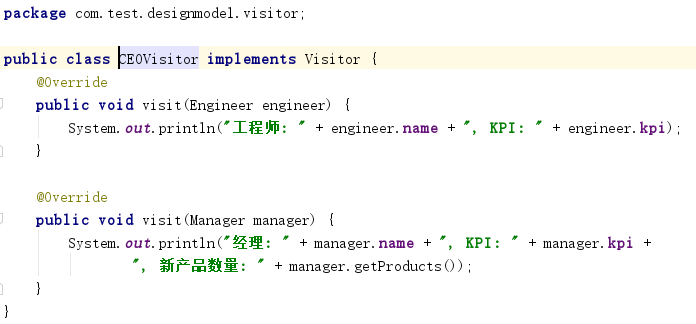


#### 访问者接口

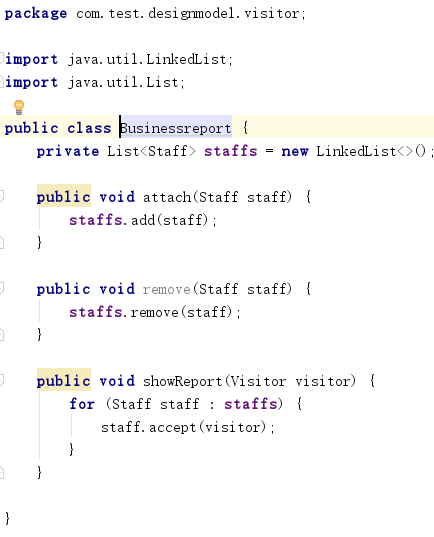


#### 具体访问者





#### 数据结构类



#### 客户端

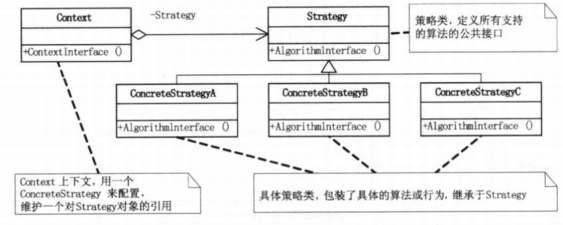


## 3.9策略（strategy）

### 概念

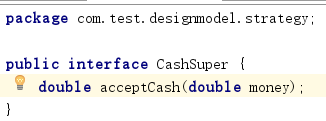
定义一些列的算法，把它们一个个封装起来，并且可以使它们相互替换。本模式使得算法可独立于使用它的客户而变化。

### 结构图

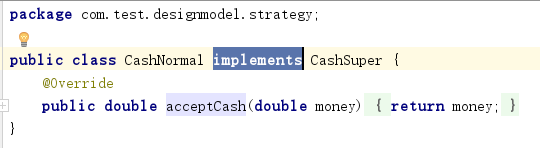


### 代码

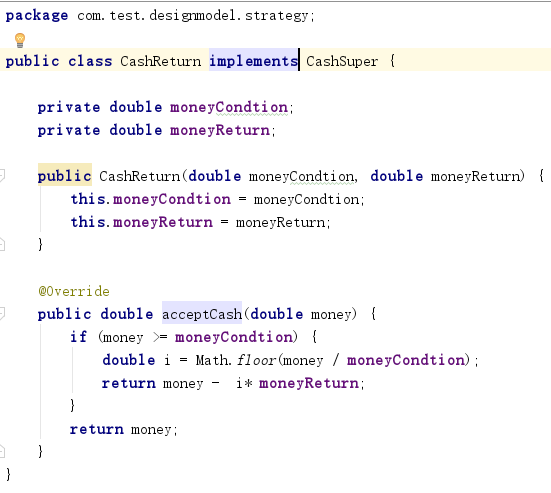
#### 策略类



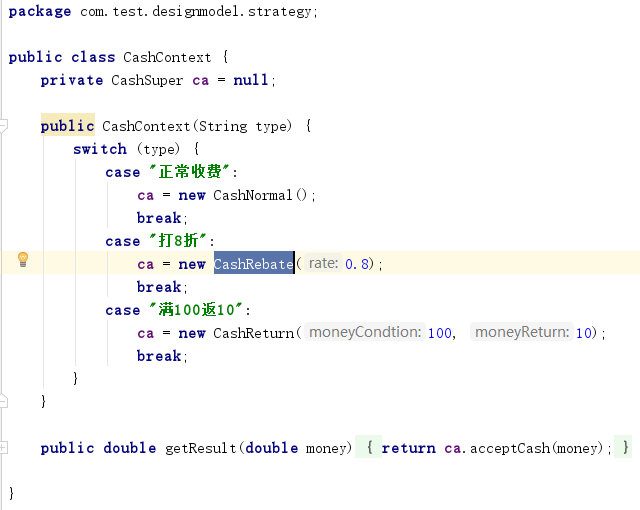
#### 具体策略类



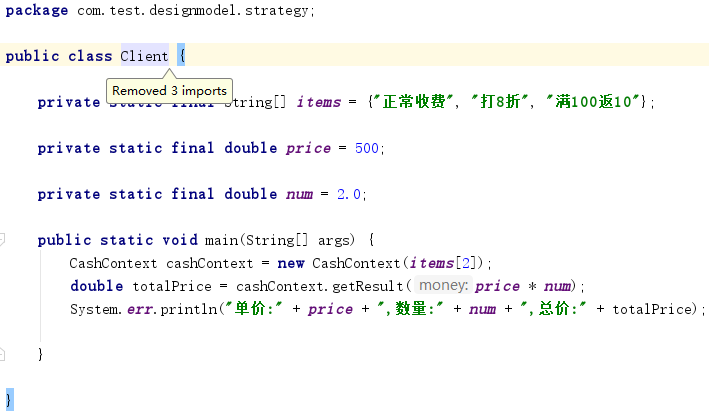




#### 配置类



#### 客户端

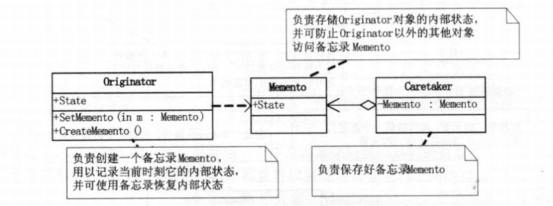


## 3.10备忘录（memento）

### 概念

在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

### 结构图

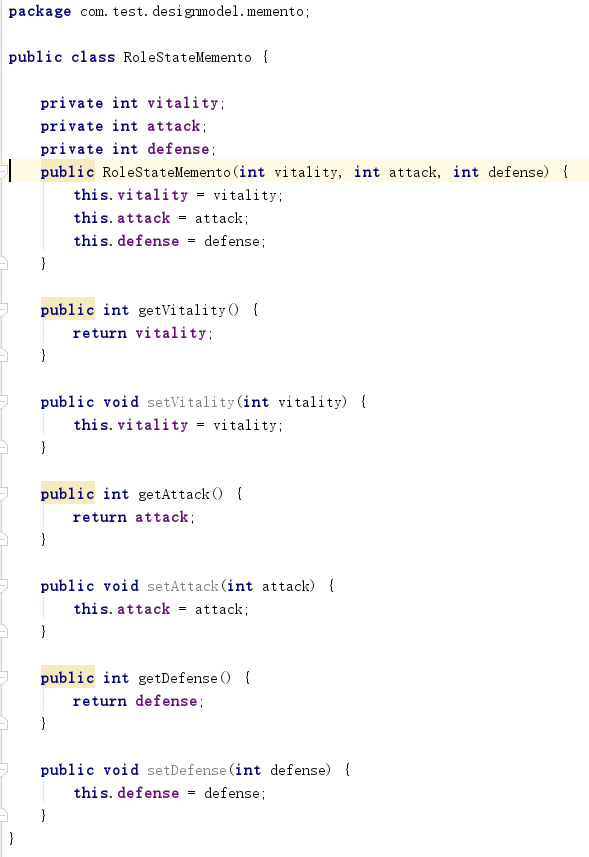


### 代码

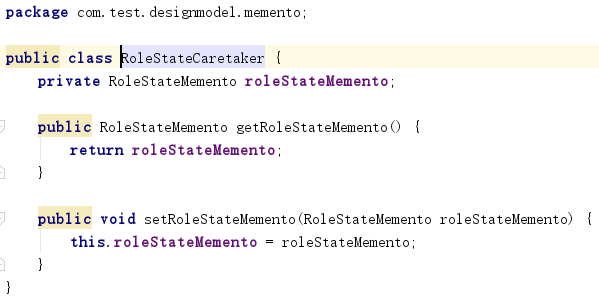
#### 原始类



#### 状态备忘录



#### 备份



#### 客户端

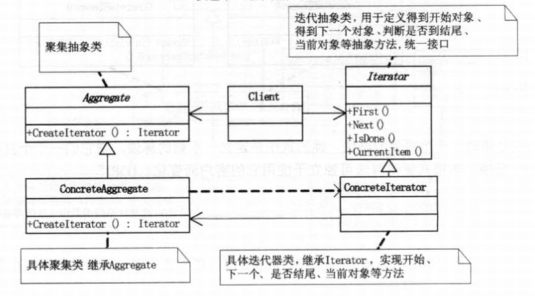


## 3.11迭代器（iterator）

### 概念

提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露其内部的表示。

### 结构图



# 设计原则

## 单一职责原则

一个类只有一个引起他变化的原因，相关性很高的函数、数据封装到一个类中。

原因：避免某块函数或数据的变化影响其他类，降低耦合度。

## 开放-封闭原则（开闭原则）

一个实体（类、函数、模块等）应该对外扩展开放，对内修改封闭。某实体应该易于扩展，在扩展某类的功能时应该通过添加新的代码来实现而不是修改其内部的代码。

原因：新功能经常会有，应该易于添加。但是添加新功能不能修改旧的已有功能，容易引起众多BUG。

实现：提供一个固有的接口，让所有可能扩展的类实现该接口，需要新功能时添加接口方法。

## 里氏替换原则

子类必须可以替换掉它们的父类型。子类对象能够替换父类对象，而程序逻辑不变。

说明：也就是说，利用多态后子类强转为父类对象，子类必须能正常使用父类中的所有函数。多态让一个类（父类）拥有多种形态（子类），任何父类出现的地方都可以由子类替换并且不会出现问题。

实现：抽象类的设计。抽象类中包含抽象方法，子类继承抽象父类必须实现父类的抽象方法，这样无论哪个子类都可以执行父类中定义的抽象方法。

## 依赖倒置原则

细节应该依赖于抽象，而抽象不应依赖细节。在 Java 中，抽象一般指抽象类和接口，细节就是实现类，实现接口或继承抽象类产生的类就是细节。也就是说：简单的说就是要求对抽象进行编程，不要对实现进行编程，这样就降低了客户与实现模块间的耦合。

## 接口隔离原则

使用多个专门功能的接口，而不是单一的总接口。

原因：一个总接口体验是非常糟糕的，实现总接口总会又很多用不到的函数。

## 合成复用原则

在一个新对象里使用一些已有的对象，使之成为新对象的一部分。

原因：新对象想使用某些对象中的功能，使用一个单独的类的功能可以继承，但是多个就不可继承了。倒不如使一些对象成为新对象的一部分，就可以通过委派的形式复用现有的某些对象的功能。

## 最少知识原则（迪米特法则）

一个模块或对象应该尽量少地与其他模块或对象发生相互作用，这样当一个模块修改时不会影响太多。

使用：A 有一个功能，B 想要使用，那么 A 只需要暴露一个函数或返回一个结果给 B 使用，而不是了解 A 如何实现的以及 A 包含的数据细节