**Design pattern in action**

切勿要求胜利，只应要求一往无前的勇气。

因为从坚忍不拔的奋斗中，你将为自己迎来荣誉。

# Chapter 1 Introduction

# **Chapter2 Observer Pattern**

## 观察者模式的定义

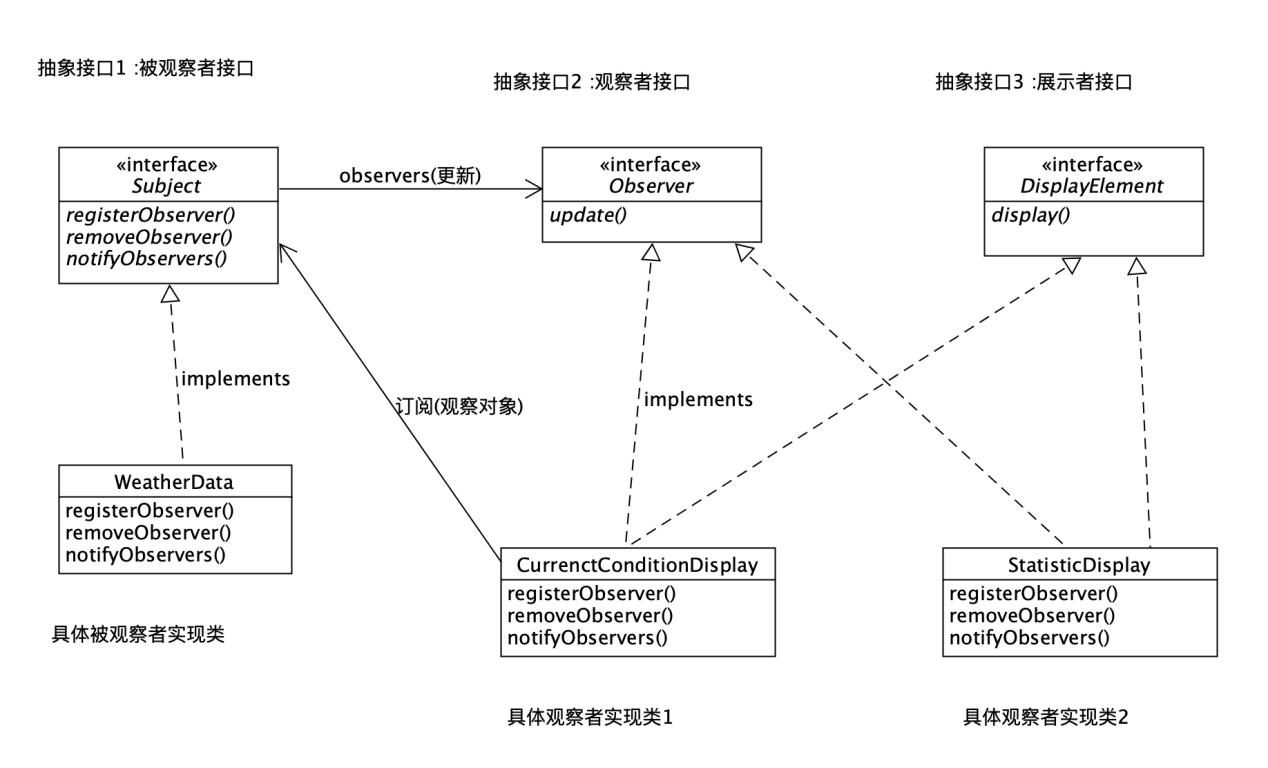
观察者模式的定义如下：

|  |
| --- |
| **The Observer Pattern**  Defines a one-to-many dependency between objects so that when one object changes state,all of its depentends are notified and updated automatically. |

其实就是发行者-订阅者模式。发行方一旦有新的内容，就能马上更新到订阅者那里。这个更新动作被抽象出来，放到一个接口中。这样，发行者、订阅者就能解耦了。

## 实际场景

我们实践一下书本中的实践场景案例，类图如下：



这个设计模式的好处就是，被观察者和观察者之间解耦了。一旦被观察者发生了变化，只要操作观察者抽象接口接口就行了，无需关心观察者具体是啥。

这符合开闭原则：一旦观察者的逻辑发生了变动，不会影响被观察者。被观察者甚至无需重新编译。

## 设计原则汇总

1. 各个交互对象之间尽量解耦
2. 编程的时候尽量采用接口，少用实际的对象
3. 尽量用组合(compotition)的方式，少用继承(inheritance)

## 总结

观察者模式有什么优势？适应什么场景？我们总结一下。

其实就是原来被观察者要自己去引入各个观察者，并且被观察者发生了变化，要一个个去通知各观察者。

在新的模式下，被观察者通过Subject接口，提供了注册、注销、更新等标准接口方法，观察者可以通过调用这些接口主动注册、注销自己；被观察者发生了变化，也可以通过标准接口通知到各个观察者。

这个设计模式使得观察者和被观察者解耦。

# **Chapter3 Decorator Pattern**

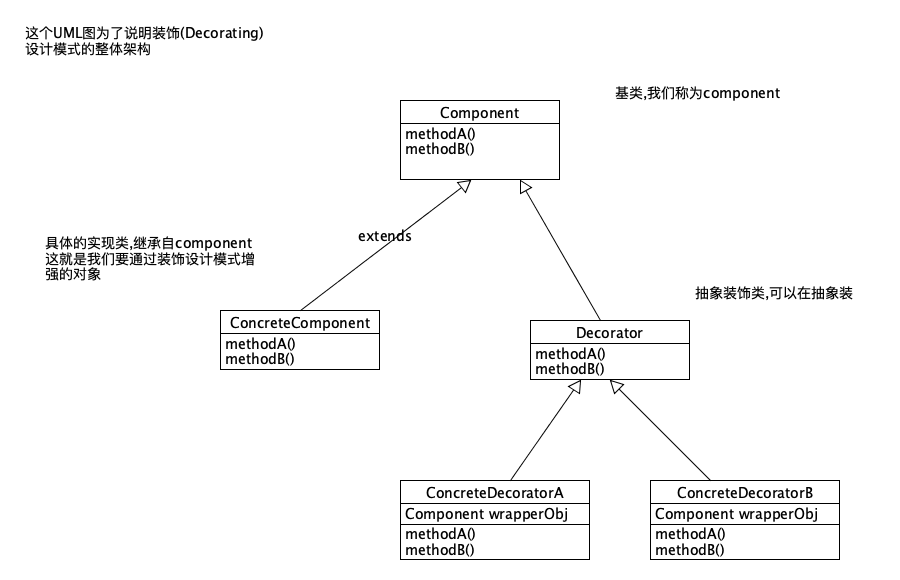
装饰模式定义如下

|  |
| --- |
| **The Decorator pattern**  Attaches additional responsibilities to an object dynamically.Decorators provides a flexible altinative to subclassing for extending functionality. |

    Class diagram如下图所示。我们可以看到，通过Decorator，我们不断地在原有的类上添加新的方法或者特性。

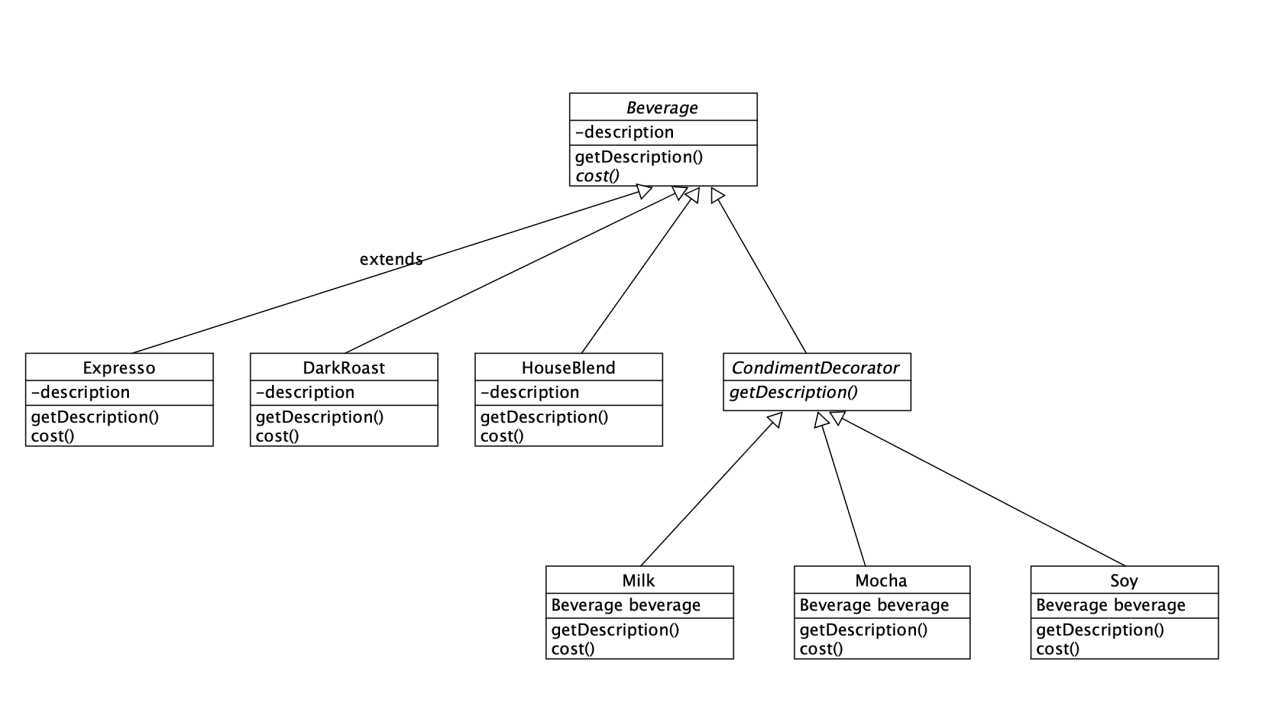
## 装饰模式UML

装饰模式UML图如下所示：



## 实例代码

书中的实例是咖啡馆，UML如下：



有这么几层含义：

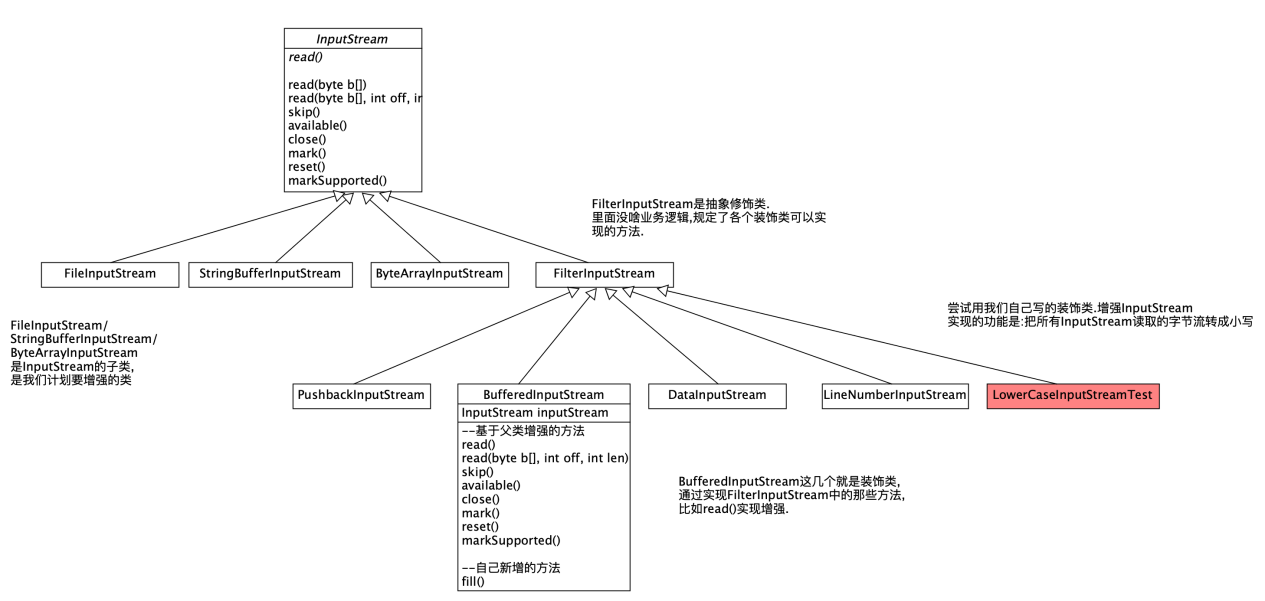
1. Beverage是基类，就是我们所说的Component
2. Expresso、DarkRoast、HouseBlend这些具体的类继承抽象类，是我们要装饰的对象；
3. CondimentDecorator作为抽象修饰类，虽然对于父类没有什么增强，但是调整父类的方法，比如CondimentDecorator就把父类中的getDescription()方法变成了抽象方法，要求各个装饰类(CondimentDecorator的子类)实现getDescription()方法；
4. Milk/Mocha/Soy是实体修饰类。继承抽象修饰类CondimentDecorator。最大的特点，就是包含了Beverage Component对象。通过包含Beverage类，可以实现对Beverage进行增强。这也是Decorating设计模式最大的特点：特性增强。

## Java中的例子

Java中使用了Decorating设计模式的例子就是BufferedInputStream，BufferedInputStream通过引入InputStream类，可以对InputStream几个具体实现类进行了增强：

(FileInputStream/StringBufferInputStream/ByteArrayInputStream)

从整体上，提升了InputStream的整体效率：



备注：通过BufferedInputStream读取文件的例子，参考FileTest.java

本书非常有意思，模拟BufferedInputStream装饰类的写法，还实现了一个自定义的装饰类：

|  |
| --- |
| LowerCaseInputStream.java |

LowerCaseInputStream实现的功能是:把所有InputStream读取的字节流转成小写

具体用法参考：LowerCaseInputStreamTest.java

通过BufferedInputStream、LowerCaseInputStreamTest这两个例子，进一步加深了我们对于装饰设计模式的理解。

## 装饰模式的优缺点讨论

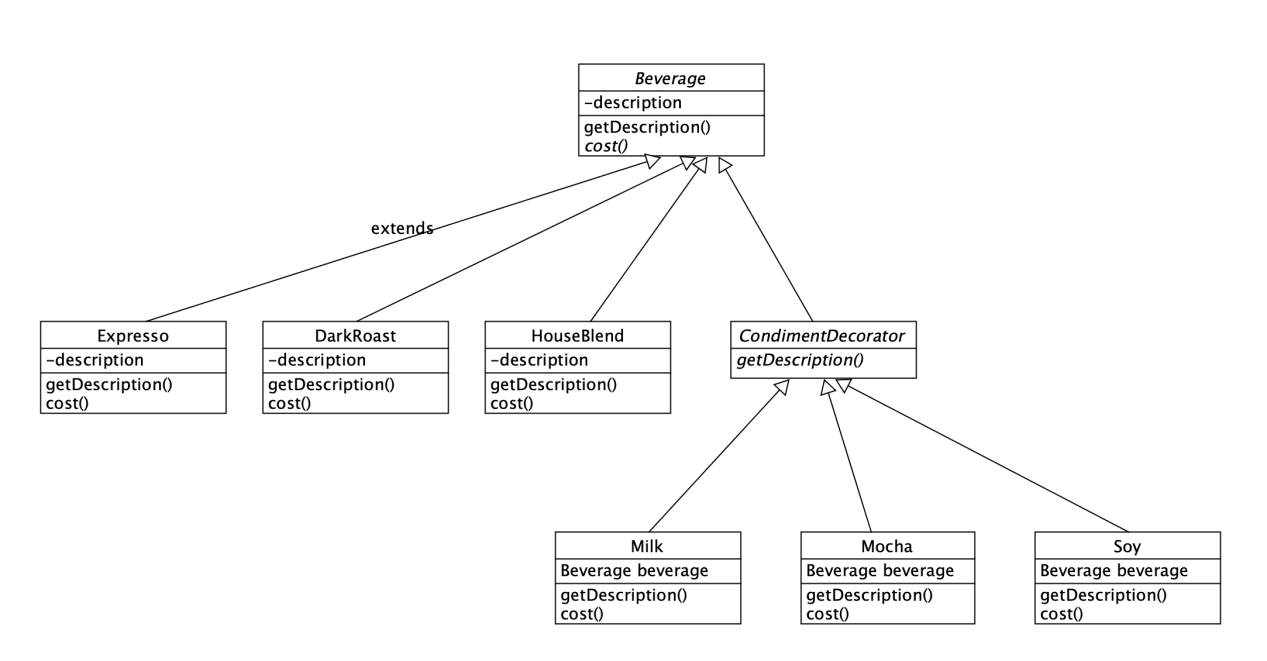
### 装饰模式的优点

1. 可以通过抽象装饰类AbstractDecorator规定具体的装饰类需要实现哪些方法；
2. 可以层层装饰；

### 装饰模式的缺点：

1. 具体装饰类通过**extends**抽象装饰类的方式实现，这就很有局限性，因为某个具体装饰类只能继承一个抽象装饰类。
2. 抽象装饰类只能够通过abstract method的方式，强制要求装饰类实现某些方法，远没有interface清晰。
3. 经过具体装饰之后，会丢失被装饰对象的细节。这个参考Beverage.toString()方法的注释。

我们还是要展开讨论一下缺点2，这个问题是这样的，我们看Beverage的UML：



通过UML可以看到，虽然我们通过Milk对Beverage下的几个具体实现类(Expresso/DarkRoast/HouseBlend)进行了装饰。如下图代码所示：

|  |
| --- |
| Beverage beverage3 = new HouseBlend(); beverage3 = new Milk(beverage3); |

这段代码看起来beverage3是HouseBlend对象，经过Milk装饰之后，cost()/getDescription()这两个方法得到了增强。那么，问题来了，经过Milk装饰之后，beverage3到底是什么对象？是HouseBlend对象还是Milk对象？

我们看代码、调试代码之后会发现，经过Milk装饰之后，beverage3已经由最初的HouseBlend对象，变成了Milk对象。这会有什么问题吗？这有很大的问题。比如，最初beverage3对象的description属性是”HouseBlend”，但是经过Milk装饰之后,beverage3对象的description属性变成了“Unknown Beverage”。这是因为经过Milk装饰之后，Milk仅对getDescription()这个方法进行了增强(在HouseBlend对象description属性基础上增强)，但是对于description属性本身并未进行更新。这就导致在经过Milk装饰之后beverage3对象的description属性和通过beverage3.getDescription()的内容不一致。

所以，在经过Milk装饰之后，beverage3看起来还是Beverage，但已经不是原来那个HouseBlend对象了。

Milk仅仅只是根据HouseBlend对象装饰了cost()/getDescription()这两个方法。至于其他属性、方法，则没有进行改动。

这意味着什么呢？这意味着某个对象经过装饰之后，有可能丢失这个对象原本的属性。

这个问题深层次的原因就是，我们在写装饰类的时候，往往只关注我们在装饰类中对原有类进行增强的那部分方法，至于其他方法、属性，我们没法全部都照顾到。更深层次的原因则是装饰模式是通过composition的模式，引入了被装饰对象，这个模式的优势当然是能够增强被装饰对象的实例数据，缺点是增强好可能会遗漏部分数据，不太可能做到面面俱到。

对于这个问题有解决方案吗？很遗憾，没有好的解决办法。我们设想这样一种场景，比如具体实现类，比如HouseBlend，新增了属性和方法，比如size/getSize()，那么，经过Milk装饰之后，必然丢失size属性。因为HouseBlend新增了属性和方法之后根本不会通知Milk这些装饰类进行同步修改。因为我们只是想通过装饰类增强特定的方法，至于你具体的实现类其他的属性和方法，关我装饰类什么事呢？

所以，装饰设计模式的局限性也就在这里：经过装饰之后，会丢失原对象的属性。所以，通过动态代理这种模式，可能是更好的选择。

正因为这个问题存在，装饰设计模式只适合那些小型的、功能相对较为单一的对象，比如本书例子中的InputStream。InputStream聚焦于read()等少数方法，因此我们可以在装饰类中对read()这个功能进行增强。

## 总结

我们实践了一下装饰模式，装饰模式通过extends/composition的方式，对某个类以及该类的所有子类，都进行了增强。

# **Chapter4 Factory Pattern**

## 工厂模式定义

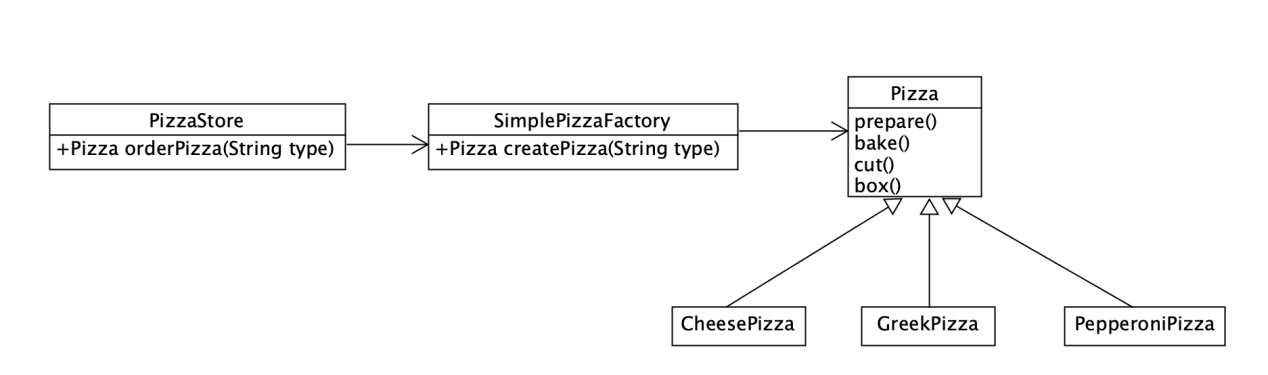
工厂模式定义如下：

|  |
| --- |
| **The Factory Method Pattern**  Defines an interface for creating an object ,but lets subclasses decide which class to instantiate.Factory Methods lets a class defer instantiation to subclasses. |

|  |
| --- |
| **The abstract Factory Pattern**  Provides an interface for creating families of related or dependent objects without specifying their contrete classes. |

## 实例代码

### 实例1 SimpleFactory

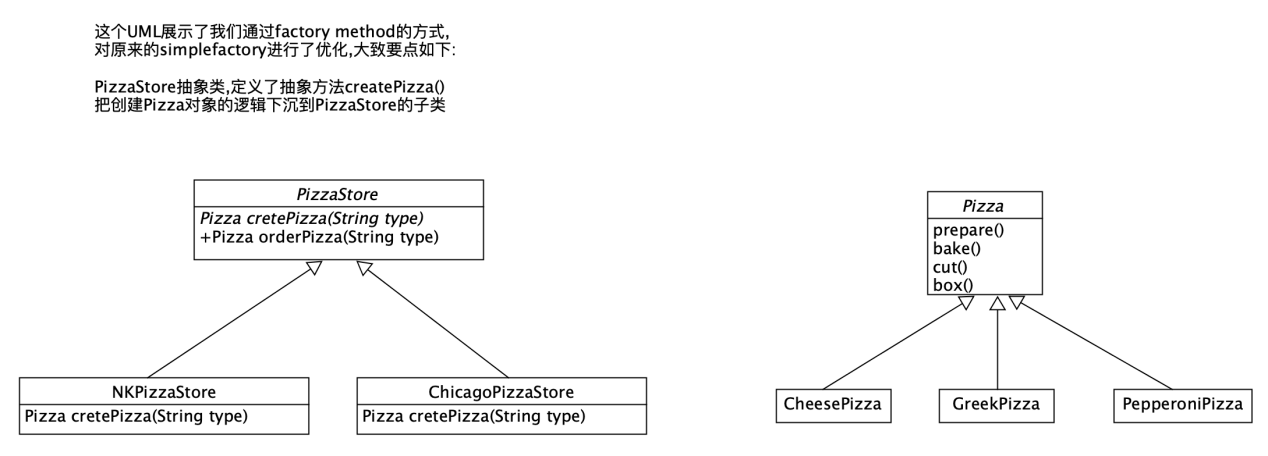


实例1是最简单的，就是把创建Pizza实例的代码抽象出来，由SimplePizzaFactory.createPizza()实现。

参考如下package的代码：

|  |
| --- |
| com.nbcb.design.factory.simpleFactory |

### 实例2 factory method



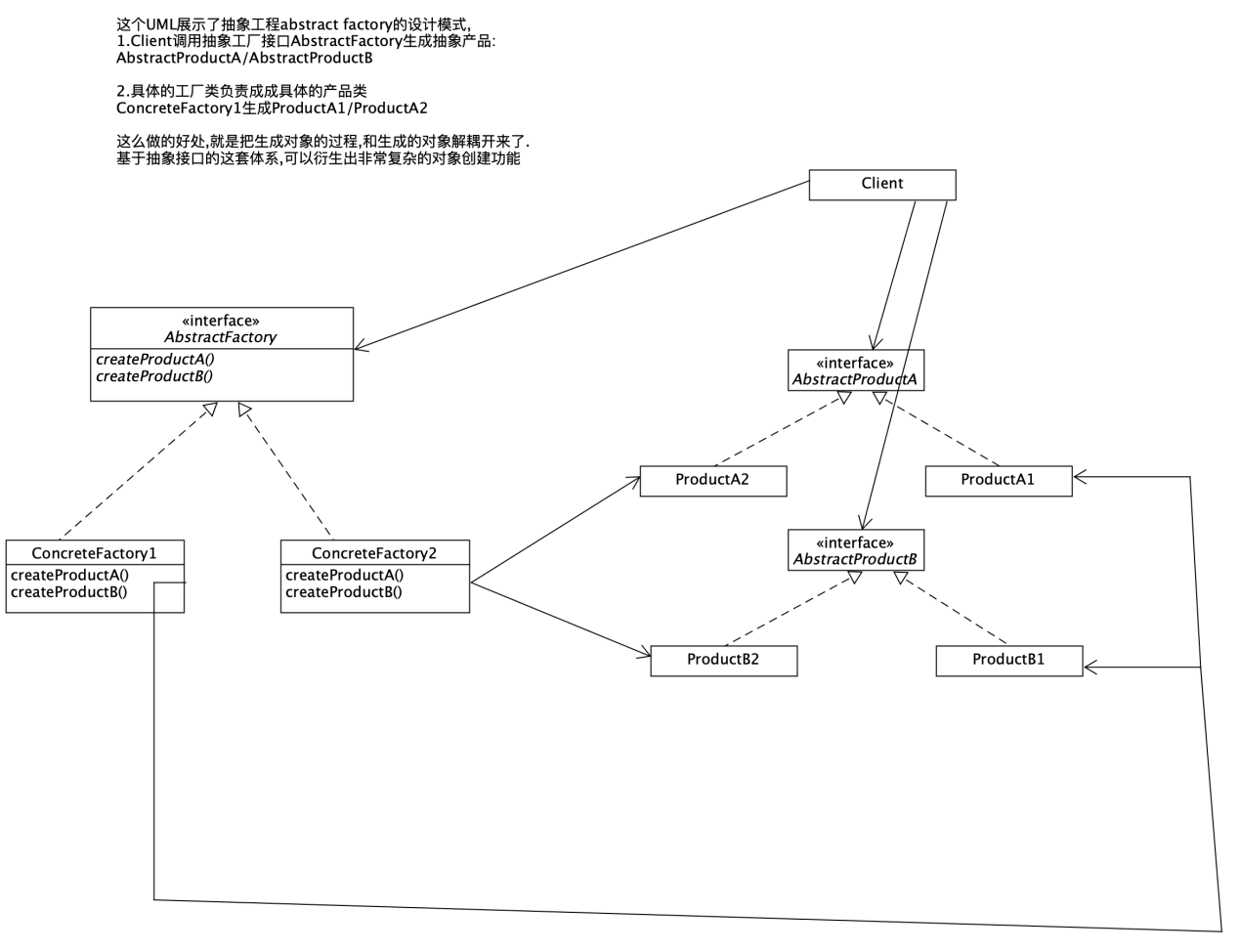
参考如下package的代码：

|  |
| --- |
| com.nbcb.design.factory.factoryMethod |

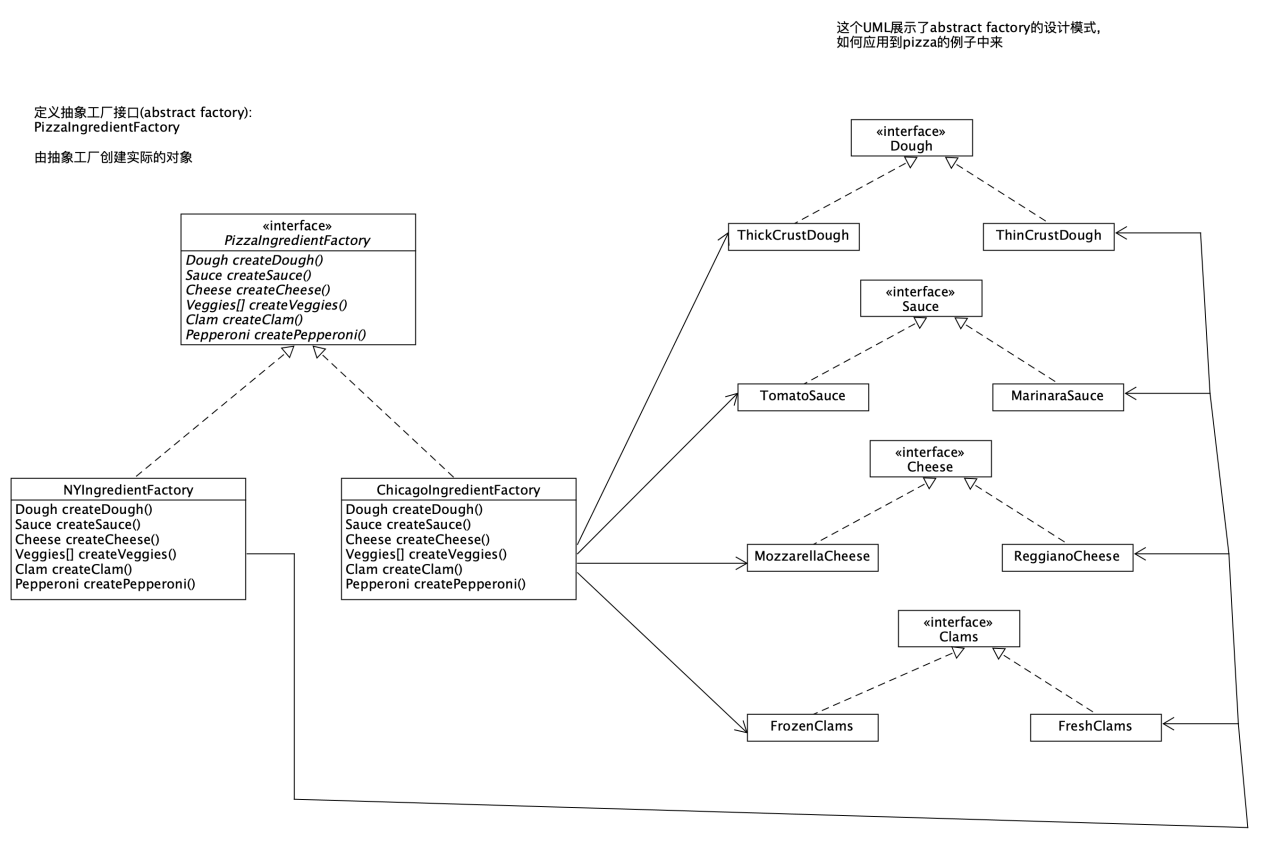
### 实例3 abstract factory

实例3引入了抽象工厂的概念

抽象工厂概念的UML图如下：



抽象工程概念在Pizza中的实例：



## 设计原则

依赖抽象(抽象类、抽象接口)，不要依赖具体的类。

## 总结

本章通过三个案例：simple factory/factory method/abstract factory，生动解释了工厂设计模式是怎么一回事。

最后的abstract factory再次强调了如何通过抽象接口，将”生产产品的动作”和”产品”本身进行解耦，从而达到提升扩展性的目标。

# **Chapter5 Singleton Pattern**

## 单例模式定义

单例模式定义如下：

|  |
| --- |
| **The Singleton Pattern**  Ensures a class has only one instance,and provides a global point of access to it. |

类图如下：

OLE-object

## 代码演进说明

类图已经很清楚地说明了Singleton这个设计模式的意思了。用实际代码展示就是：

|  |
| --- |
| **private static** Singleton *mySingleton*;  **public static** Singleton getInstance(){  **if**(*mySingleton* == **null**){  *mySingleton* = **new** Singleton();      }  **return** *mySingleton*;  } |

代码1 Singleton设计模式 v1.0

v1.0存在的问题是，在多线程模式下会有问题。多个线程同时调用getInstance()方法的时候，会导致mySingleton被多次实例化的，导致Singleton模式失败。

如何验证这个现象呢，我们通过ThreadTest.java就能验证。

解决方案1：

|  |
| --- |
| **private static** Singleton *mySingleton*;  **public static synchronized** Singleton getInstance(){  **if**(*mySingleton* == **null**){  *mySingleton* = **new** Singleton();      }  **return** *mySingleton*;  } |

代码2 Singleton设计模式 v1.1

v1.1通过给getInstance()加上synchronized关键字，防止并发线程同时调用getInstance()方法，规避了以上的问题。

v1.1的问题是效率比较低。为啥效率低呢？因为我们看到getInstance()方法被加上synchronized关键字后，每次获取实例都加上了synchronized关键字，这样其实是不合理的。因为需要new Singleton对象的场景非常少。

后续针对效率低的问题，提出了2个解决方案：

|  |
| --- |
| **private static** ChocolateBoiler *chocolateBoiler* =  **new** ChocolateBoiler();  **public static synchronized** ChocolateBoiler getInstance(){  **return** *chocolateBoiler*;  } |

代码3 Singleton设计模式 v1.2 解决方案1

|  |
| --- |
| **private volatile static** ChocolateBoiler *chocolateBoiler*;  *// static getInstance()*  **public static** ChocolateBoiler getInstance(){  **if**(*chocolateBoiler* == **null**){  **synchronized** (ChocolateBoiler.**class**){  **if**(*chocolateBoiler* == **null**){  *chocolateBoiler* = **new** ChocolateBoiler();              }          }      }  **return** *chocolateBoiler*;  } |

代码4 Singleton设计模式 v1.3 解决方案2

## 总结

单例模式的思路比较清楚，要点如下：  
 1.constructor设置为private ，private的意思是只有当前类可以初始化，别的类没法通过new XXX初始化对象。这就保证了对象的唯一性；

2.通过static getInstantce()方法统一获取(唯一的)对象；

3.在getInstantce()方法中，通过synchronized关键字保证并发线程在初始化对象过程中，保持同步；

4.volatile关键字，当car对象在初始化的时候，如果有并发线程在访问，能够保证car对象实例的唯一性。如果不加这个关键字，在多并发场景下可能会创建多个car实例。

书本中在讨论单例模式的时候，非常有价值的一块内容就是对于并发线程的

讨论，核心思想就是只有在真正需要同步的时候，才加入synchronize关键字。

# **Chapter6 Command Pattern**

|  |
| --- |
| **The Command Pattern**  Encapsulates a request as an object, thereby letting you parameterize other objects with different requests,queue or log requests, and support undoable operations. |

## 实例代码

示例代码主要分为4部分，分别放在4个package下：

1. simple

用最精简的代码，说明了Command设计模式是怎么回事，UML参考《command-simpleRemoteControl.uxf》

1. remotecontrol

介绍了完整了Command设计模式的一个场景：遥控器(remote control) 。整体来说就是通过Command接口，抽象出LightOnCommand/LightOffCommand/StereoOnWithCDCommand/StereoOffWithCDCommand/CeilingFanOnCommand/CeilingFanOffCommand这几个具体的Command命令。然后交给RemoteControl统一处理。

具体UML参考：《command-RemoteControl.uxf》

1. ceilingfan

介绍了ceilingfan 代码，说明了Undo功能在command设计模式中是怎么实现的。所谓的undo功能，就是具体的Command类实现Command接口中的undo()方法。然后通过RemoteControl统一实现。

具体UML参考：《command-ceilingfan.uxf》

1. macro

介绍了如何批量执行command的，具体参考《command-macro-command.uxf》

## 总结

我们看到，所谓的Command设计模式，其实就是A要调用B，不是直接调用B对象的某个方法，而是由B实现Command接口，然后通过一个控制类，去操作这些Command接口。这样就把对象A调用对象B的动作解耦合了。这么做有什么好处呢？就是可以通过控制类，统一管控要操作的各种命令，实现命令队列管理、命令日志记录、命令回退等各种高级功能。

后续碰到这种场景，就可以用这个设计模式。Command设计模式的核心，还是通过抽象接口的方式，将”调用某个对象”这个动作解耦开来，有利于扩展性。

# **Chapter7 Adpter Pattern**

## 设计模式定义

**这章介绍了两种设计模式：**

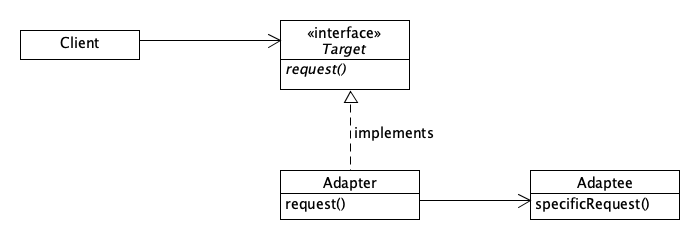
**Adapter/facade**

|  |
| --- |
| **The Adapter Pattern**  Converts the interface of a class into another interface the clients expect.Adapter lets classes work together that couldn’t otherwise because of incompatible interface. |

|  |
| --- |
| **The Facade Pattern**  Provide a unified interface to a set of interfaces in a subsystem.Facede defines a higher-level interface that makes the subsystem easier to use. |

## Adapter适配器设计模式

Adapter适配器的UML如下：



Adapter设计模式的意思.其实就是Client想要操作Target对象,但是我现在只有Adaptee对象,那我就通过一个Adapter对象中转一下.怎么做呢：

1.Adapter对象依赖Adaptee对象

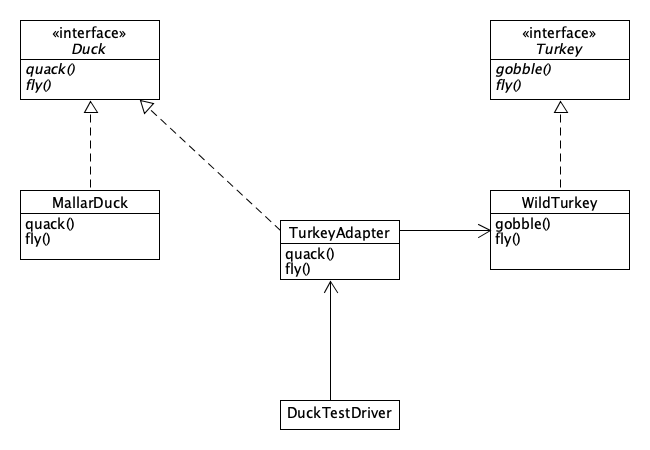
2.Adapter对象实现Target对象的接口

2.但是Adapter对象对象实现Target对象接口的实现方法中, 都是由Adaptee对象完成的

有两个实例：

### 实例1 Duck

UML图如下：

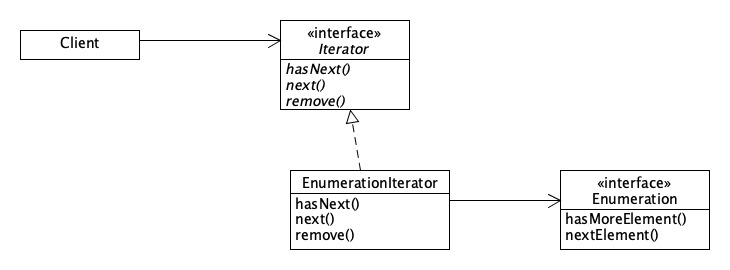


参考代码：

|  |
| --- |
| com.nbcb.design.adapter.duck |

### 实例2 EnumerationIterator

UML图如下：



参考代码：

|  |
| --- |
| com.nbcb.design.adapter.enumeration |

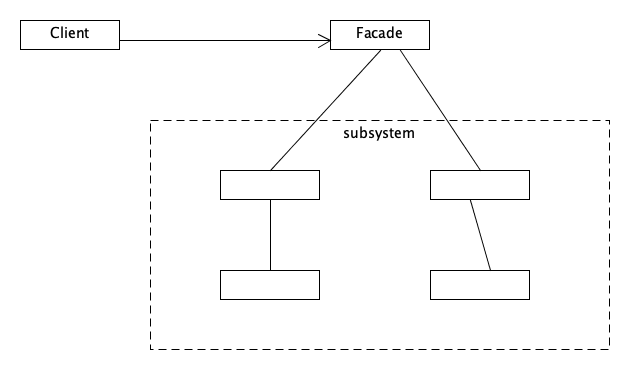
## Facade门面设计模式

门面模式意思就是通过一个门面类:Facace

封装若干个类,然后客户端Client只要和Facade打交道就行了

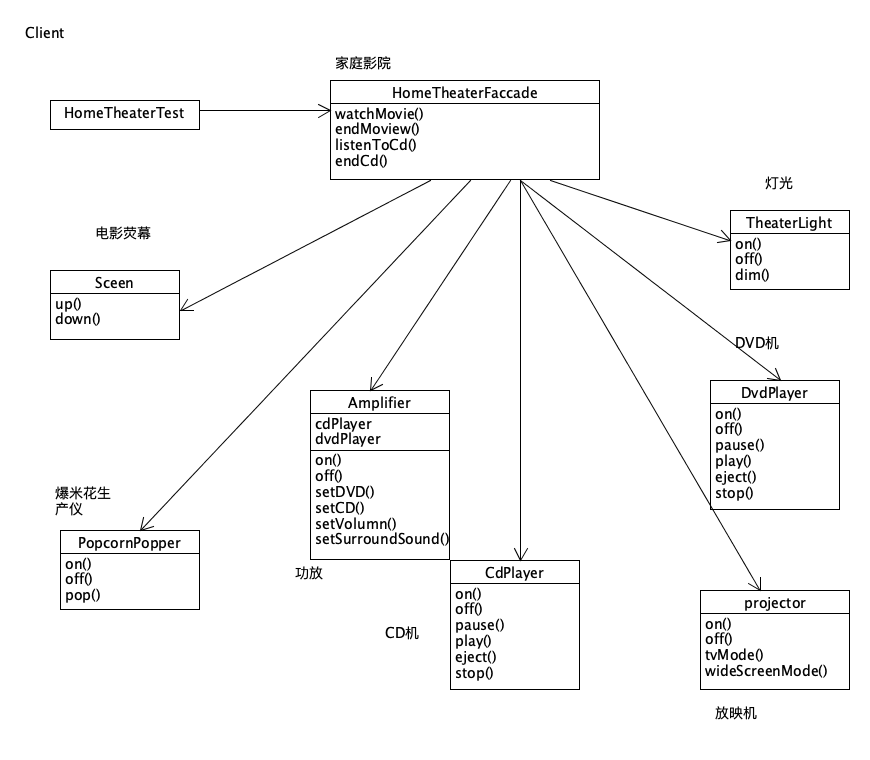
无需知道subsystem中其他类的实现细节

UML图如下：



### 实例1 HomeTheaterFaccade

UML图如下：



这个UML是为了说明Facade设计模式的一个实例场景。虽然整个UML看起来很复杂,但是核心思路非常简单。其实就是HomeTheaterFaccade把家庭影院中各个模块封装起来了。我只要通过HomeTheaterFaccade.watchMovie()就能够实现看电影的目标。无需和具体的模块打交道.

参考代码：

|  |
| --- |
| com.nbcb.design.adapter.enumeration |

### 实例2 tomcat HttpServlet

大致代码如下：RequestFacade就是体现了门面的设计模式。RequestFacade封装了Request类，也就是把Request的一些实现细节封装起来了。servlet开发人员只能够通过访问RequestFacade提供的各个方法访问Request中的信息，无法直接访问Request对象本身。这么做的好处就是防止servlet开发人员破坏Request中的信息。

备注：在RequestFacade中虽然依赖了Request对象，但是Request对象是private的servlet开发人员没法直接访问Request。



## 设计原则

Least knowledge原则：只和你最直接的朋友对话。

## 总结

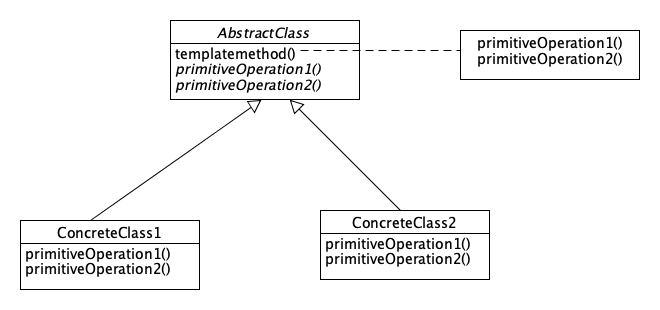
# **Chapter8 Template Method Pattern**

## 设计模式定义

|  |
| --- |
| **The Template Method Pattern**  Defines the skeleton of an algorithm in a method ,deffering some steps to subclasses.Template Method lets subclasses fedefine certain steps of an algorithm without changing the algorithm’s structure. |

有一个父类，这个父类中部分方法是定义在父类中，另外一部分是抽象类。需要在子类中定义。

UML图如下：



这个UML是为了说明TemplateMethod设计模式。所谓的template method,就是在某个抽象类中,定义一个template method,这个方法会调用若干抽象方法(primitiveOperation1()/primitiveOperation2())，这些抽象方法由具体的子类去实现.

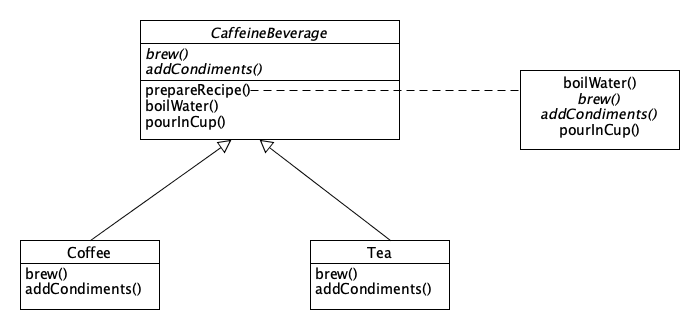
这么做的好处是,template method可以专注于某个算法的实现。而不用关心具体的子类是是啥.

最典型的场景就是Arrays.sort(),排序的时候,无需关心array中元素的真正类型是啥

## 实例代码

### 例子1 Tea and Coffee

UML图如下：



这个UML是为了说明TemplateMethod设计模式的一个景:CaffeineBeverage。CaffeineBeverage中定义一个template method:prepareRecipe()，这个模板方法调用

1.若干抽象方法:

brew()/addCondiments()

这些抽象方法由具体的子类(Coffee/Tea)去实现.

2.若干实际方法:

boilWater()/pourInCup()

这些具体方法定义在抽象类CaffeineBeverage内部

这么做的好处是,template method(prepareRecipe())可以专注于算法的实现:通过各个步骤准备饮料.而不用关心具体的子类是是啥.template method中,一部分是各子类实现逻辑都一样的通用方法,那就定义在抽象类中;一部分是各子类个性化的实现逻辑,由子类自己去实现.

具体代码参考：

|  |
| --- |
| com.nbcb.design.templatemethod.coffeine |

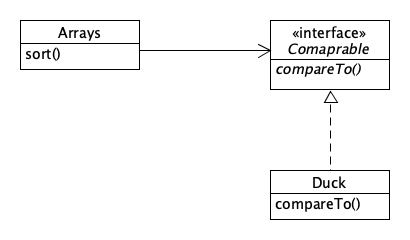
|  |
| --- |
| 父类：  Coffeine.java  子类：  Tea.java  Coffee.java  抽象方法放在父类中，特色的方法定义在子类中 |

### 例子2 CompareTo方法

|  |
| --- |
| Duck.java  DuckSortDrive.java  Duck.java实现了Comarable接口，实现了compareTo()方法  Comparable接口中只是声明了compareTo()的抽象方法。  具体的比较逻辑定义在Duck.java类中。 |

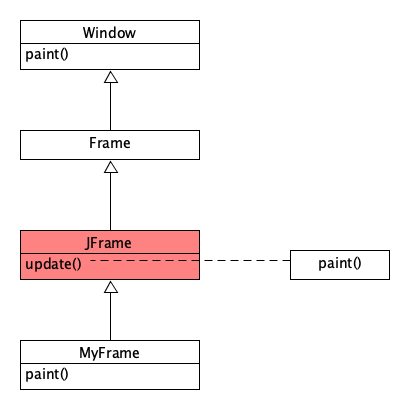
template method设计模式一个典型的应用场景是Arrays.sort()。那些需要通过Arrays.sort()进行排序的类必须实现Comparable接口，并且实现中CompareTo()方法的比较逻辑。Arrays.sort()排序的时候会调用Comparable接口。

UML图如下



### 例子3 MyFrame

模板方法设计模式的第三个应用场景是MyFrame，UML图如下：



这个UML是tempalte method的一个应用场景，在这个场景中, JFrame.update()方法就是一个tempalte method。JFrame.update()方法会调用paint()方法对UI进行一个重绘的动作：

1.如果我们在子类(MyFrame)中没有重写paint()方法,

那么JFrame就会调用父类Window.paint()这个默认的方法

2.如果我们在子类(MyFrame)中重写了paint()方法,

就按照我们自定义的逻辑来展示UI.

具体代码参考：

|  |
| --- |
| com.nbcb.design.templatemethod.myframe |

|  |
| --- |
| MyFrame.java  MyFrame.java继承了JFrame，重载了paint方法。 |

## 设计原则

好莱坞原则：dont call me ,we call you

意思就是顶层的抽象类在合适的时候，会调用子类实现的方法。

## 总结

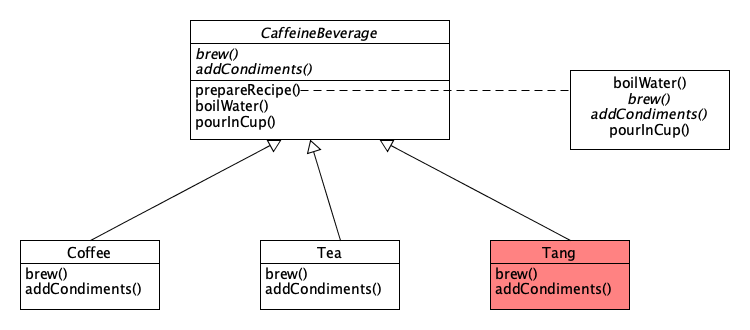
从整体来看，template method的意思是，整合了整个hiarachy中各个类的一些方法，形成一个算法。template method汇总了很多方法，部分定义在抽象父类中，部分下沉到子类中去。

其实本质上来说，就是把template method中的逻辑抽提出来，形成一个算法。这个算法不依赖于具体的子类类型，而是依赖于某个业务逻辑。这样，这个算法就和具体的子类解耦了。

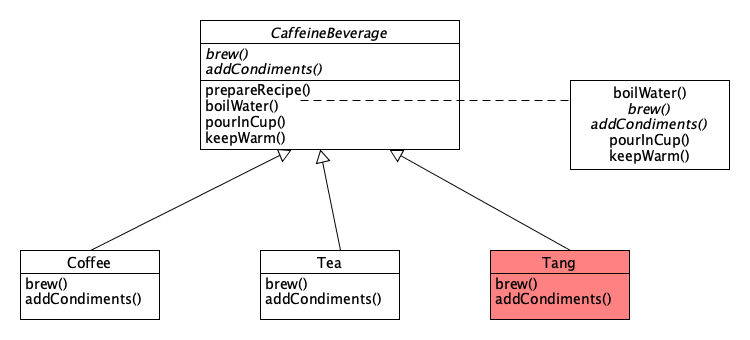
### Good Part

这么做有啥好处呢？

1. 这么做体现了OO设计中依赖倒置的思想。客户端只要和抽象父类打交道就行了，具体是什么子类其实不用怎么关心。
2. 这么做体现了OO设计中解耦的思想。一旦将算法和具体的子类解耦，就有利于扩展。比如我们加入了一个新的饮料，比如果珍(Tang)，只要继承CaffeineBeverage抽象父类，实现brew()/addCondiments()两个抽象方法就行了。整体的准备饮料的动作(prepareRecipe()方法)逻辑是无需改动的。



又比如我们希望调整算法，比如在准备饮料的动作(prepareRecipe()方法)中加入一个新的步骤：保温(keepWarm())，我们只要在父类中新增keepWarm()方法，并放到prepareRecipe()中就行了，无需改动各个子类:



### Bad Part

这个设计模式的一个问题，就是extends。抽象父类把一些方法下沉到子类，由子类去实现。由于在Java中，子类只能有一个父类，因此子类本身的可扩展性就会差一些。

当然，Arrays.sort()提出了一个较好的解决方案，Arrays.sort()作为一个template method，把compareTo()方法的实现逻辑下沉到数组中各个具体的元素，由各个元素去实现compareTo()。compareTo()方法并非通过extends的方式实现的，而是通过各个element实现Comparable接口去实现，这就比较灵活。因为element类既可以实现Comparable接口,在其他场景中也可以实现其他什么XX接口。所以，我们要清楚看到template method设计模式的这个缺点，后续尽量通过实现接口的方式，把业务逻辑下沉。尽量避免通过extends的方式，把业务逻辑下沉到子类。

# **Chapter9 The Iterator and Composite Patterns**

## 设计模式定义

### Iterator

|  |
| --- |
| **The Iterator Pattern**  Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. |

       我们来总结一下Iterator这个设计模式：有一个集合，我们通过Iterator的形式，将遍历这个集合的动作抽象出来。

       这样做有什么好处呢？我们遍历这个集合对象的时候，不用考虑这个对象的内部实现，即便集合对象的内部实现有了变化，遍历的动作也无须改变。

       示例代码如下：

|  |
| --- |
| MenuItem.java  DinerMenu.java  PancakeHouseMenu.java  DinerMenuIterator.java  PancakeHouseMenuIterator.java  Waitress.java  Iterator.java   // interface  Menu.java       // interface  MenuTestDrive.java // 测试程序入口 |

### Composite

|  |
| --- |
| **The Composite Pattern**  Allows you to compose objects into tree structures to represent part-whole hierarchies.Composite lets clients treat individual objects and compositions of objects uniformly. |

       怎么解释composite pattern呢？例子还是之前的menu，之前我们在menu中只能有menuitem，如下图所示：

    现在我们要把Menu也作为tree hiarachy中的一个node，如下图：

   说白了，就是能够将Menu和MenuItem融合起来，无差别地进行访问(当然细微的差别还是有的)。

   类比一下，就是Menu相当于Windows中的目录(dir)，MenuItem相当于文件(file)，之前我们只有一层目录，现在我们可以通过Composite，实现目录嵌套。

    怎么实现呢？实现代码汇总：

|  |
| --- |
| MenuComponent.java  // 抽象类  MenuNode.java // menu节点 是MenuComponent的子类  MenuItem.java // menu中每个具体的菜单项 是MenuComponent的子类  WaitressNew.java  // 打印MenuComponent组成的tree hierachy  MenuTestDriveNew.java // 新的驱动类 |

    看看代码当然是看不出什么意思来的，还是总结一下吧。

    无差别地访问Menu和MenuItem？

    听起来是不是有点熟悉？对了，之前我们通过Iterator实现对象遍历的时候也是这个意思：

    无差别地访问集合对象，无论集合对象是什么。

    Composite和Iterator其实有异曲同工之妙，都是不管你集合或者tree hierachy中的对象是什么，都能无差别地访问。

    之前Iterator的实现方式是，集合对象实现Iterator接口(hasNex() and next())

    Composite的实现方式是，tree hierachy中的node只要继承了MenuComponent抽象类(add()/remove()/getChild()/print()等方法)，就能无缝集成到我们tree hierachy体系中，进行访问。

**Composite和Iterator结合**

    前面两结我们通过Iterator访问MenuItem，通过Comonent实现了MenuNode和MenuItem融合在Tree hierachy架构下统一访问。现在我们将Comosite和Iterator结合：通过Iterator的方式嵌套访问MenuNode下的MenuItem信息。

**总结**

    我们总结一下，这一章的代码。最关键的是2个代码文件：

|  |
| --- |
| MyIterator.java  MenuComponent.java  CompositeIterator.java |

    其中MyIterator.java是一个interface，类似java.util.Iterator。我们通过实现这个自定义的Iterator，实现对Menu下MenuItem的遍历。

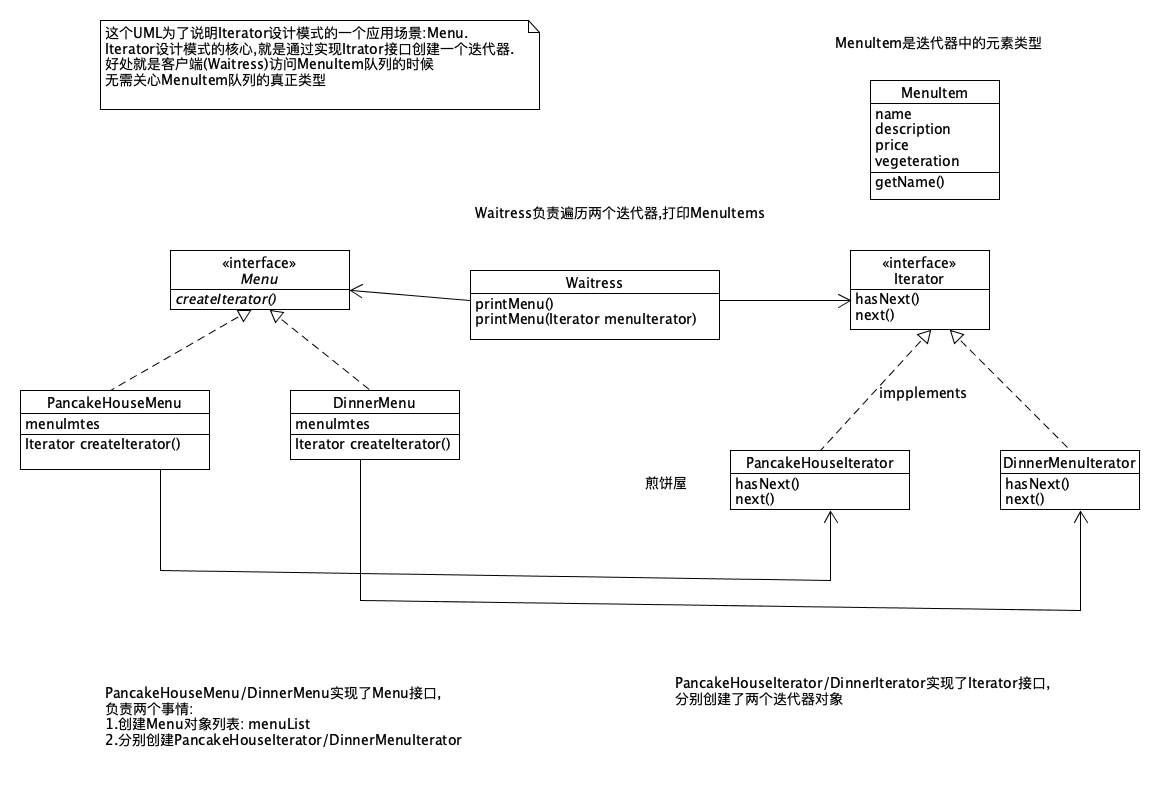
  其中MenuComponent.java是抽象类，通过集成这个抽象类，MenuItem和MenuNode可以无差别地进行访问。

  那么如何实现以Iterator的方式访问MenuComponent通过对象呢？那就要用到CompositeIterator.java了。

## 实例代码

### 例子1 Menu

这个例子的UML图如下：



这个UML为了说明Iterator设计模式的一个应用场景:Menu。Iterator设计模式的核心,就是通过实现Itrator接口创建一个迭代器.好处就是客户端(Waitress)访问MenuItem队列的时候，无需关心MenuItem队列中元素的真正类型。

这相当于是把某个队列中”各个元素的类型”和”遍历这个队列”的动作实施了一个解耦。这么做的好处也是显而易见的，就是增加了可扩展性。比如：

1. 我们把队列中元素的类型换一下，就无需调整”遍历这个队列”的动作相关的代码；
2. 或者我们把”遍历这个队列”的动作进行了扩展，比如我们希望新增一个remove()操作，那也无需修改队列元素的代码。
3. 甚至我们调整一下队列的类型，比如从ArrayList换成HashSet，对于客户端来说也是无感的，因为客户端是按照Iterator的方式遍历队列的。

### 例子2

## 设计原则

A class should have only one reason to change

单一职责原则，一个类只负责一件事情

## 一个BUG说明

**注意点**

    让人大跌眼镜的是，这一章居然还有一个代码bug:

|  |
| --- |
| CompositeIterator.java  next()方法：  *stack.push(component.createIterator());*  ------>  **stack**.push(currentNode.**menucomponents**.iterator()); |

    仔细想想，也不难理解，stack.push()的意思是，将需要遍历的MenuItem或者MenuNode下的MenuItem节点以Iterator的形式，放到stack中。

    而*stack.push(component.createIterator())*中，*component.createIterator()*虽然是返回*Iterator*，但是我们看对于*MenuNode*，*createIterator()*方法是这样的：

|  |
| --- |
| **public** Iterator createIterator(){  **return new** CompositeIterator(**menucomponents**.iterator());  } |

看出问题来了吗：这里是创建了一个新的*CompositeIterator*实例，那么在 *CompositeIterator*的*constructor*中，

|  |
| --- |
| **public** CompositeIterator(Iterator iterator){  **stack**.push(iterator);  } |

又执行了一次*stack.push()*，这样不是*stack*就重复*push*了吗？其实正确的做法是将当前*MenuNode*下的*MenuItem*节点以*Iterator*的形式，直接*push()*到*stack*中：

|  |
| --- |
| **stack**.push(currentNode.**menucomponents**.iterator()); |

# **Chapter10 The State of Things**

|  |
| --- |
| **The State Pattern**  Allows an object to alter its behavior when its internal state changes. The object will appear to change its class. |

     这章的例子是gumball machine，gumball machine的各个状态如下：

Has Quarter

Out of

ball

投币

按上按钮，放球

No Quarter

退币

Gumball = 0

球全部卖光了

Gumball sold

Gumball >0

放球后，回到硬币清空的状态

    作为对比，这章主要有两个代码：

1.GumbalMachine.java

2.GumbalMachineState.java

       其中GumbalMachine.java是老的思路：面向动作：

   在每个动作中，都要先判断当前*gumball machine*的状态，然后选择相应的动作，类似*: if(state == SOLD\_OUT){ ... } else if(){ ... }*

    GumbalMachineState.java给我们带来了新的思路：

和之前的*GumbalMachine.java*不同的是，这次我们每次要做动作的时候，不需要判断当前状态了。为什么呢？因为我们有一个变量：*state*。这个变量保存的就是当前的状态。这个类实现的思路是：在当前状态下，如果要做某个动作，会发生什么。至于不同状态下*,*做某个动作，需要做什么是，逻辑都放在各个状态的实现类中。

    新的思路和状态机有点类似：在某给状态下，如果施加某个动作，会做什么事情，事情做完后，会转到下一个动作。

# **Chapter11 The Proxy Pattern**

|  |
| --- |
| **The Proxy Pattern**  Provides a surrogate or placeholder for another object to control access to it |

## 1.Gumball例子

Gumball是一个普通的java代码示例，为后续说明如何使用动态代理进行铺垫。大致代码如下：

|  |
| --- |
| 服务端：  GumballMachine.java  客户端：  GumballMonitor.java  客户端GumballMonitor.report()，获取服务端GumballMachine对象的状态。 |

具体代码参考如下package：

|  |
| --- |
| com.nbcb.design.proxy.gumballmachine |

## 2.RMI例子

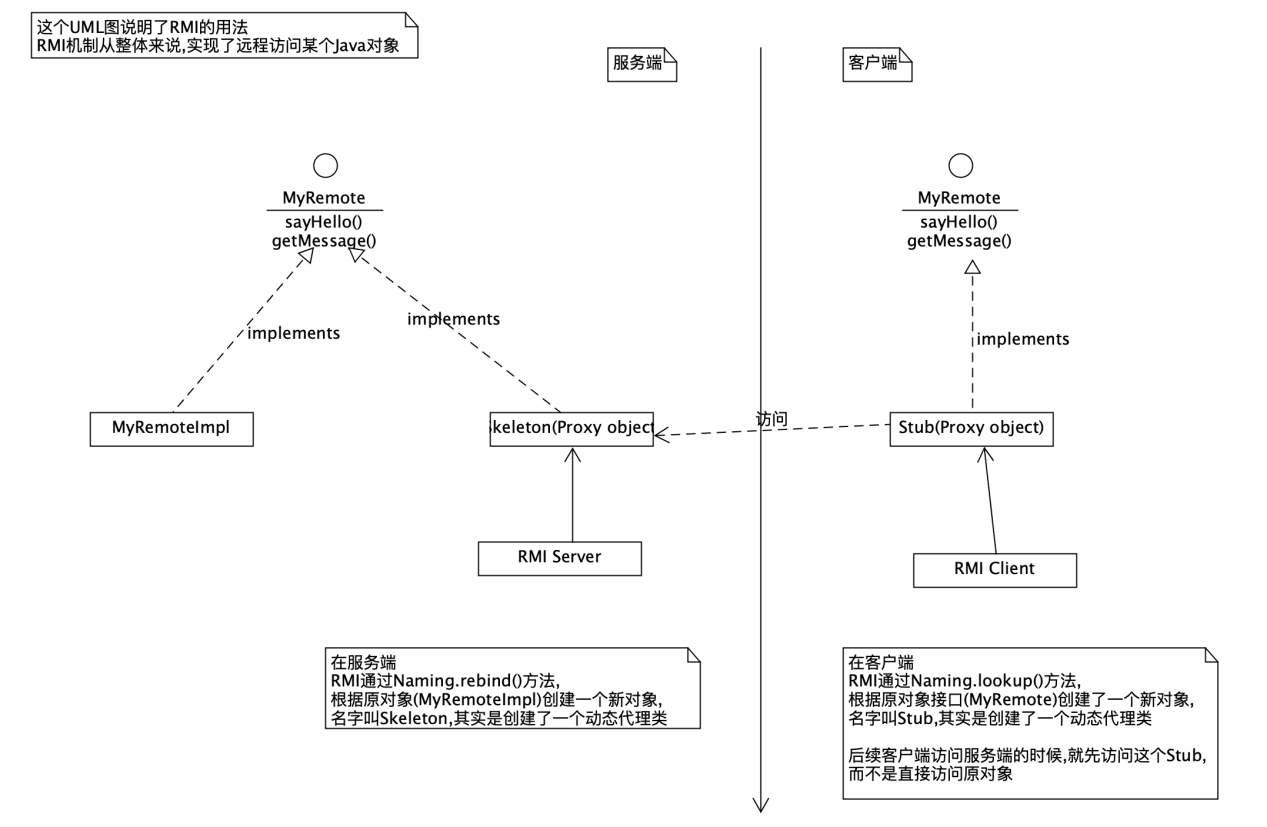
我们通过这个小节的例子，说明一下java RMI的用法。

### RMI功能说明

所谓的RMI，其实就是java提供的一种远程方法调用的方式。通过RMI，能够跨JVM，甚至跨服务器，调用远程>的java方法。

### RMI整体架构

我们先梳理一下RMI整体架构：



接下来，我们整理一下java RMI通讯的详细步骤：

### 服务端：

1. 定义一个RMI接口：

|  |
| --- |
| interface MyRemote extends Remote { ... } |

其中定义的接口方法，供客户端调用

1. 定义RMI实现类：

|  |
| --- |
| public class MyRemoteImpl  extends UnicastRemoteObject         implements MyRemote { ... } |

1. 定义RMI Server

比如:

|  |
| --- |
| LocateRegistry.createRegistry(1900); Naming.rebind("rmi://localhost:1900/remoteSayHello",myRemote); |

### 客户端：

|  |
| --- |
| MyRemote myRemote  = (MyRemote)Naming.lookup("rmi://localhost:1900/remoteSayHello"); myRemote.sayHello() |

### 特别备注：

1. RMI接口中定义的接口方法，必须throws RemoteException 否则启动RmiServer的时候会抛错：

2.如果RMI Server和client之间需要传递对象，这个对象

必须实现Serializable接口 用于序列化、反序列化 参考：Message.java

### 具体代码

具体代码参考：

|  |
| --- |
| com.nbcb.design.proxy.rmi |

### RMI总结

我们总结一下RMI的用法，其实要实现RMI通讯，核心是定义一个客户端、服务端都能够接受的Remote接口。然后RMI会通过动态代理的方式，分别在服务端创建Skeleton、客户端创建Stub(Skeleton/Stub都实现了Remote接口，其实Skeleton/Stub都是动态代理类)。

后续client/Server通过Skeleton/Stub进行通讯。

## 3.Gumball结合RMI

Gumball和RMI结合，就能够实现远程调用

为了实现跨JVM访问GumballMachine信息的目标，我们要做哪些事情呢？  
我们记录一下步骤：

### **1.定义RMI Remote接口类**

GumballMachineRemote  
  
2.改造GumballMachine

我们原来的GumballMachine类要改造一下，实现  
新定义的Remote接口：GumballMachineRemote  
  
GumballMachine  
 extends UnicastRemoteObject  
 implements GumballMachineRemote {  
 ...  
 }  
  
3.创建RMI Server  
参考：GumballMachineServer.java  
这个Server的功能就是把我们新创建的RMI Remote接口类发布出去

4.创建RMI Client  
参考：  
GumballMachineClient.java  
GumballMonitor.java  
  
整理总结一下  
我们看到，要实现某个对象的远程访问，  
必须通过一个定义好的RMI Remote接口  
无论是RMI服务端还是RMI客户端，都通过这个RMI Remote接口进行交互

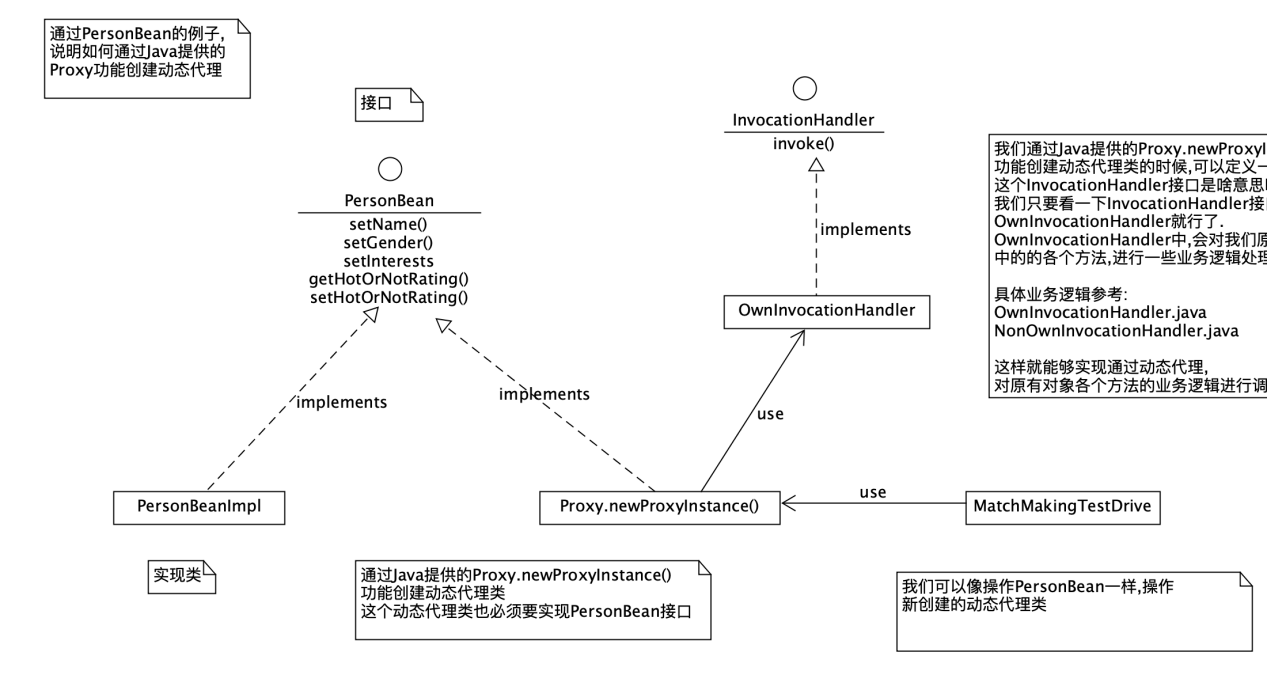
具体代码参考：

|  |
| --- |
| com.nbcb.design.proxy.remotegumball |

## 4.CDViewer

## 5.PersonBean

通过PersonBean的例子,说明如何通过Java提供的Proxy功能创建动态代理

大致架构如下：

## 总结

我们通过各种动态代理的案例，来说明动态代理的功能。这章更多的是介绍动态代理是什么，并没有解释动态代理的原理。

我们从Java自带的Proxy.newProxyInstance()的用法，也能对动态代理的实现方式窥探一二：

|  |
| --- |
| return (PersonBean) Proxy.newProxyInstance(  personBean.getClass().getClassLoader(),  personBean.getClass().getInterfaces(),  new OwnInvocationHandler(personBean)); |

# **Chapter12 Comound Patterns:Patterns of Patterns**

这章介绍如何把前几章的设计模式组合起来。有两个实例：

1. Duck
2. MVC

其中Duck例子综合运用了adapter/decorator/factory/observer等设计模式；

MVC例子就非常著名了，大量使用在BS系统中，MVC用到了

## 实例代码

### 代码1 Duck

### 代码2 MVC

### 代码3 CellPhone

## 总结

# **Chapter13 Patterns in the Real World**

## 实例代码

## 总结

# Chapter14 Appendix: Leftover Patterns

其它设计模式还有很多。

# 设计模式汇总

我们把所有本书出现的设计模式都梳理一遍，整理在《设计模式汇总.xlsx》中

|  |  |  |
| --- | --- | --- |
| 序号 | 设计模式名称(英文) | 设计模式名称(中文) |
| 1 | Observer Pattern | 观察者模式 |
| 2 | Decorator | 装饰者模式 |
| 3 | Factory | 工厂模式 |
| 4 | Singleton | 单例模式 |
| 5 | Command | 命令模式 |
| 6 | Adpter | 适配器模式 |
| 7 | Template Method | 模板方法 |
| 8 | Iterator and Composite Patterns | 遍历者模式 |
| 9 | State | 状态模式 |
| 10 | Proxy | 代理模式 |
| 11 | Bridge | 桥接模式 |
| 12 | Builder |  |
| 13 | Chain of Responsibility | 责任链模式 |
| 14 | Flyweight |  |
| 15 | Interpreper |  |
| 16 | Mediator |  |
| 17 | Memento |  |
| 18 | Prototype |  |
| 19 | Visitor |  |

其中1-10是作者详细介绍的，后面9种是简略介绍。

# 总结

## 本书的优点

1. 及其丰富的案例、图例，能够使得我们以最小的成本掌握设计模式的知识；而且这些案例都是我们生活中的案例，非常有意思
2. 从纷繁复杂的设计模式中，总结出OO设计的一些原则，比如解耦合什么的；
3. UML图非常专业；
4. 除了具体场景的示例代码，还结合了Java API中对应的代码，非常不错。

## 本书缺点

1. 各个案例规模较小，和那些开源组件的规模相比，还是完全不在一个量级上。所以一方面，我们要考虑在较大规模工程中应用设计模式的可行性，也要考虑在大型项目中，如何综合应用多个设计模式；

## 后续计划

1. 将书中各个设计模式通过UML图展示出来；
2. 调研一下在开源工程中，都用到了哪些设计模式；
3. 把设计模式应用到日常项目中去；
4. 学习更多的设计模式相关书籍；

比如下面这些豆瓣高分的书籍，本地都有电子书：



1. 还有一个很好的学习来源是极客学院

地址为：

<https://www.jikexueyuan.com/course/750f1003/d7925729/f4c017ad>

“Chapter14 Appendix: Leftover Patterns”中那些书中没有详细解释的设计模式，在极客学院有详细的视频说明。

# 阅读本书的一些感受

1. 我们这轮学习设计模式，主要是为了配合UML的学习。在这轮学习中，我们体会到，先画UML，再动手写代码，是一个比较好的习惯；一般来说，UML体现了整体架构和整体的设计思路。一旦思路确定了，后续编程实现就相对顺利一些；
2. 一种设计模式，是为了解决一种特定的场景。后续我们碰到相关场景，就用对应的一个设计模式去解决。
3. 我们总结一下各个设计模式，设计模式虽然千变万化，但是核心都是相同的：

核心一

遵循OO的几个设计原则，比如：开闭原则、依赖倒置原则，等等；

核心二

整体目标就是实现可扩展性、解耦，等等。

核心三：

通常的做法就是实现某个接口，通过接口，将各个component解耦开来。