

2019 NOVEMBER 5

软件架构与**设计模式** 实现文档

指导教师：侯捷 冯巾松

Version 2.3.6

谢康 1753402

谢尚汝 1754188

贺思奇  1751324

吉筠柏  1753294

吕海军  1751074

卢朋艺  1751043

宋亮  1753424

李航  1752026

## 成员与分工

| **姓名** | **学号** | **分工** |
| --- | --- | --- |
| 谢康 | 1753402 | 整体架构 + 类概要设计 |
| 谢尚汝 | 1754188 | 整体架构 + 类概要设计 |
| 贺思奇 | 1751324 | 设计模式应用 + 类详细设计 |
| 吉筠柏 | 1753294 | 设计模式应用 + 类详细设计 |
| 吕海军 | 1751074 | 接口设计 +类详细设计 |
| 卢朋艺 | 1751043 | 接口设计 +类详细设计 |
| 宋亮 | 1753424 | 工作整合 + 类详细设计 |
| 李航 | 1752026 | 工作整合 + 类详细设计 |

## 综述

生活在嘉定，程序员们的胃口总是得不到满足，于是他们幻想出了这样一家中餐厅，它并不起眼，但在程序员的眼中，它却有着无穷的魅力。

当客人进入餐厅，他只需要查看菜单，选出几个他想要的菜或者套餐，又或者各种搭配的菜+套餐，便只需付款等待上菜就可以了。而勤劳的餐厅服务员，会马上把订单进行一番“翻译”并交给厨师，厨师拿到订单后，会聪明的完成烹饪过程，而这一过程并不依赖客人的下单顺序。

当然了，我们的中餐厅也会不可避免的出现一些突发情况。比如厨师想要拿两个鸡蛋，但发现蛋不够了，这时厨师可以托人购买一定数量的鸡蛋，很快货就会补上；又比如客人一次性点了非常多的菜，需要非常多的原料，这时不要担心，我们的冰箱和橱柜可以容纳无限多的原料，并且不会在取出时造成麻烦。

## Design Pattern 汇总表

| **编号** | **Design pattern name** | **实现个数** | **sample programs个数** | **备注** |
| --- | --- | --- | --- | --- |
| 1 | 单例模式 Singleton | 3 | 3 |  |
| 2 | 抽象工厂模式 Abstract Factory | 1 | 1 |  |
| 3 | 工厂模式 Factory | 1 | 1 |  |
| 4 | 模板模式 Template | 1 | 1 |  |
| 5 | 观察者模式 Observer | 1 | 1 |  |
| 6 | 迭代器模式 Iterator | 1 | 1 |  |
| 7 | 状态模式 State | 1 | 1 |  |
| 8 | 命令模式 Command | 1 | 1 |  |
| 9 | 装饰器模式 Decorator | 1 | 1 |  |
| 10 | 策略模式 Strategy | 1 | 1 |  |
| 11 | 建造者模式 Builder | 1 | 1 |  |
| 12 | 桥接模式 Bridge | 1 | 1 |  |
| 13 | 外观模式 Facade | 1 | 1 |  |
| 14 | 享元模式 Flyweight | 1 | 1 |  |
| 15 | 原型模式 Prototype | 1 | 1 |  |
| 16 | 备忘录模式 Memento | 1 | 1 |  |
| 17 | 空对象模式 Null Object | 1 | 1 |  |
| 18 | 组合模式 Composite | 1 | 1 |  |
| 19 | 访问者模式 Visitor | 1 | 1 |  |
| 20 | 适配器模式 Adaptor | 1 | 1 |  |
| 21 | 中介者模式 Mediator | 1 | 1 |  |
| 22 | 代理模式 Proxy | 1 | 1 |  |
| 23 | 责任链模式 Responsibility Chian | 1 | 1 |  |
| 24 | 解释器模式 Interpreter | 1 | 1 |  |

## 项目结构 Package Structure

* **总览**

由于本项目是采用 Java 语言，因此我们的设计遵循了 Java 包结构设计的统一标准，这样带来的好处是：

**1. 把功能相似或相关的类或接口组织在同一个包中，方便类的查找和使用。**

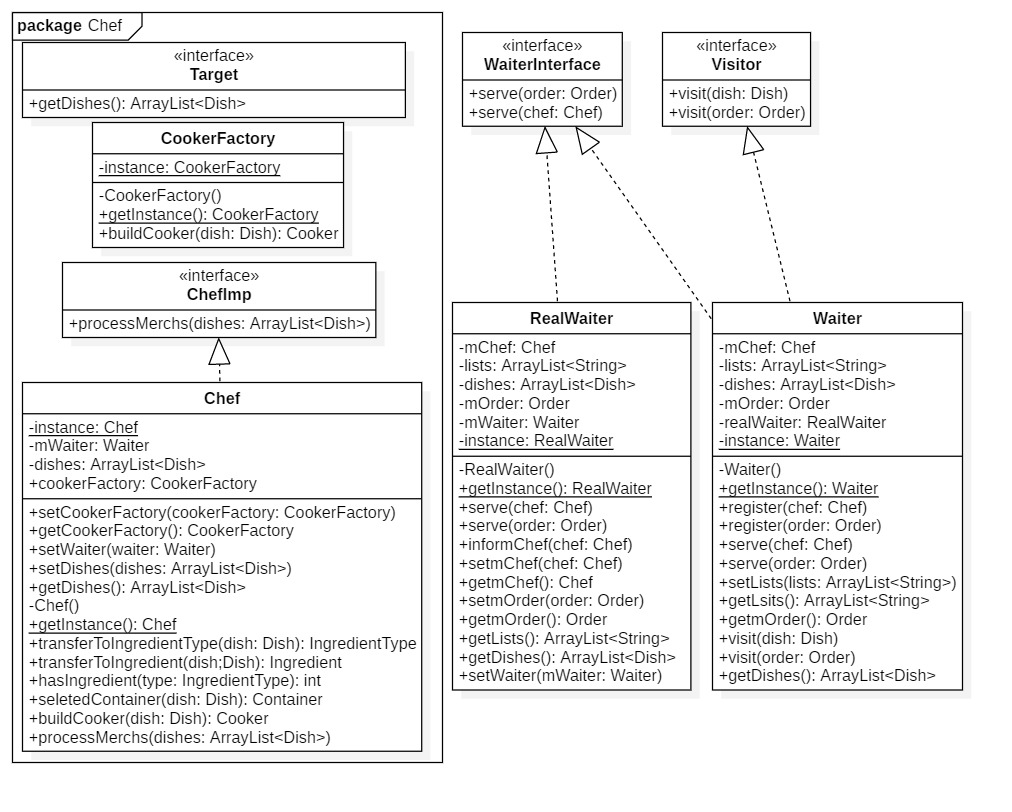
**2. 如同文件夹一样，包也采用了树形目录的存储方式。由于包创建了新的命名空间，所以不会跟其他包中的任何名字产生命名冲突。使用包这种机制，更容易实现访问控制，并且让定位相关类更加简单**

**3. 包也限定了访问权限，拥有包访问权限的类才能访问某个包中的类。**

我们的项目由七个 package 组成：

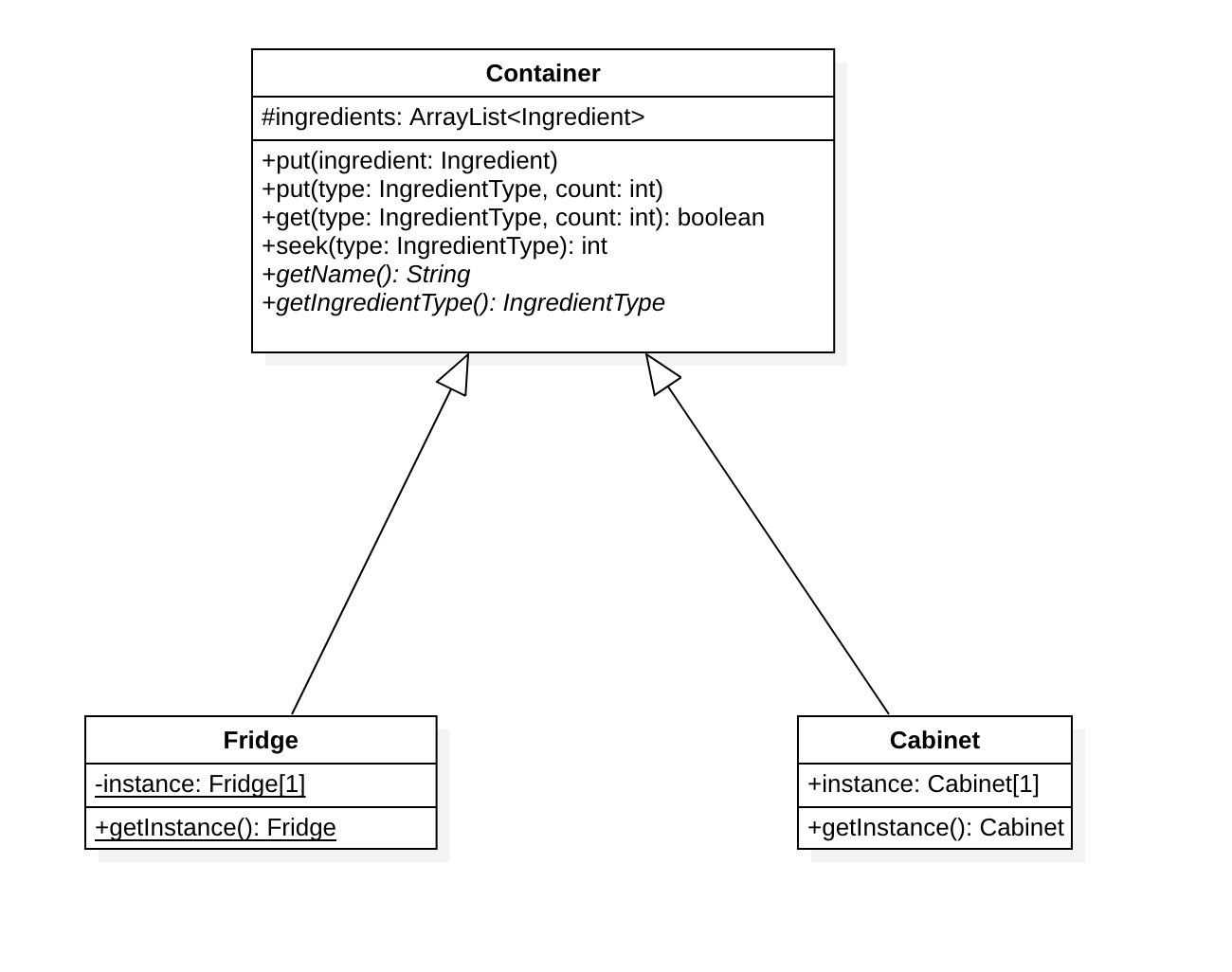
* + 员工 Staff
  + 容器 Container
  + 原料 Ingredient
  + 厨具 Cooker
  + 商品 Merch
  + 订单 Order
  + 时间控制器 Time
* **Package** Staff

我们厨房的员工只有一位厨师和服务员，这里使用了单例模式，他们的能力很强，能同时处理好若干菜单，厨师对外只是处理菜单，实际上则是进行了从菜单解析，到原材料的获取，和厨具的使用，以及烹饪，所以这里用到了命令模式；选择厨具时，对厨具进行遍历选择，用时长的优先选择，这里使用了责任链模式。



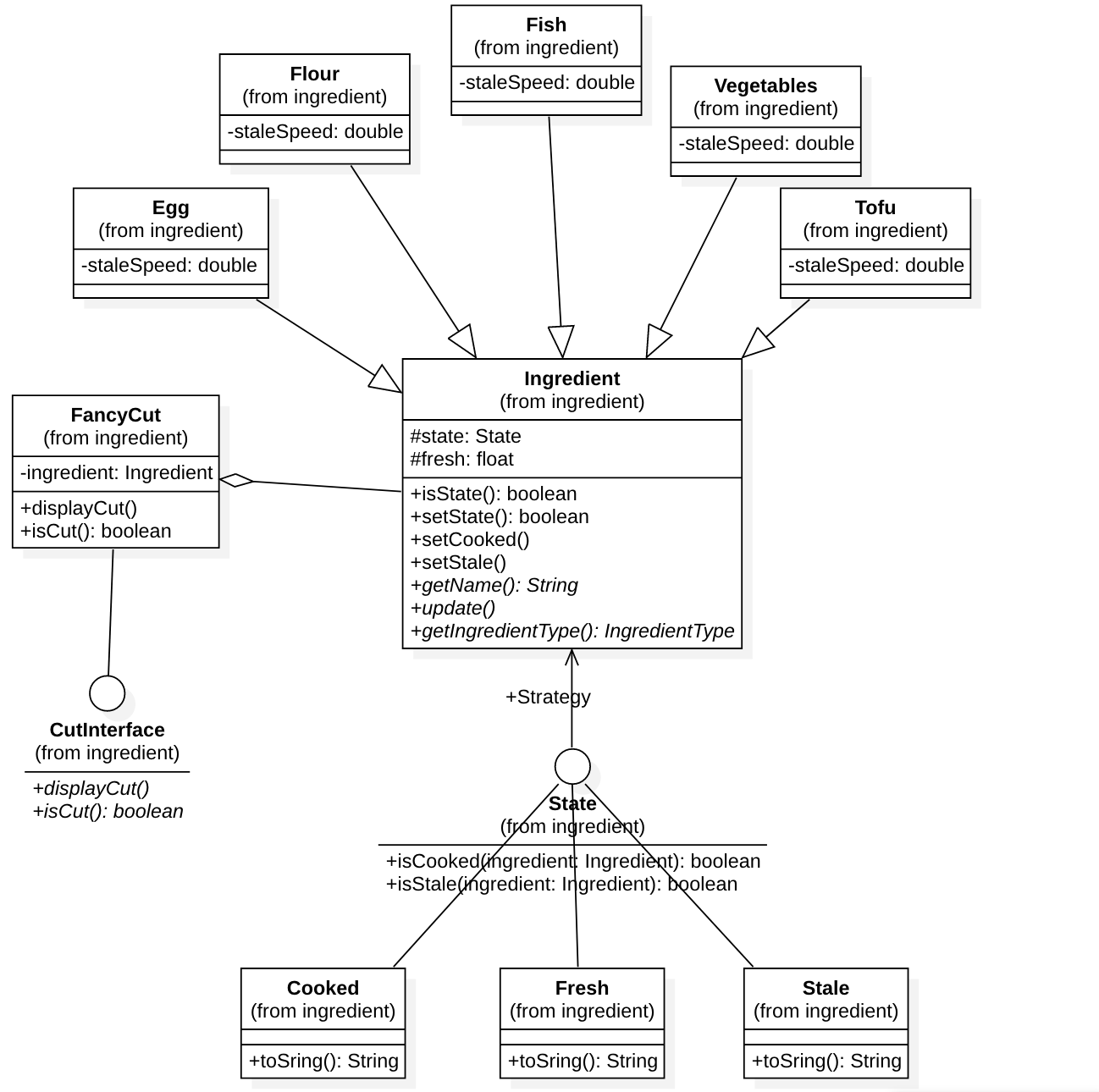
* + 厨师类 Chef
  + 厨师类接口 ChefImp
  + 厨具工厂类 CookerFactory
  + 解释器接口 Target
  + 服务员接口 WaiterInterface
  + 访问者接口 Visitor
  + 服务员代理类 Waiter
  + 真正的服务员类 RealWaiter
* **Package** Container

我们厨房的容器只有冰箱和橱柜，由于它只在程序员的幻想中，所以他们的容量都是无限的，所以每一样我们只需要一台，这里用到了单例模式；当我们在容器中查找某一个原料的时候，会用到迭代器，这里用到了迭代器模式。



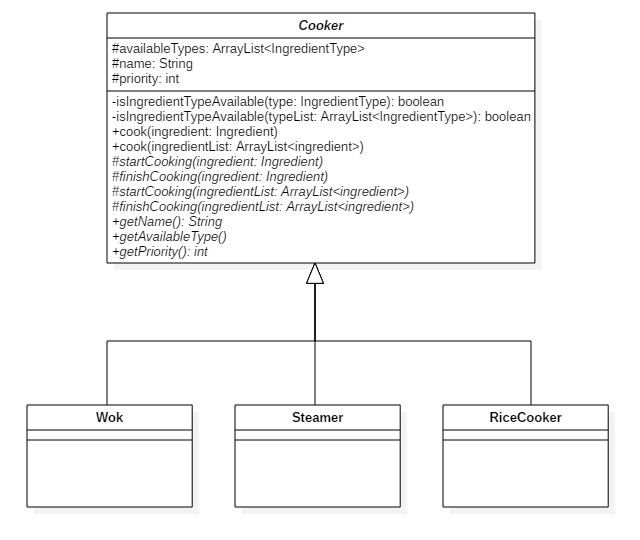
* + 容器类 Container
  + 橱柜类 Cabinet
  + 冰箱类 Fridge
  + 容器迭代器类 ContainerIterator
* **Package** Ingredient

为了烹饪出各种各样的菜，我们自然需要多种原料，而原料具有相当多的属性，比如它们会腐败，这里用到了观察者模式；又比如原料腐败到一定程度或者食材被用掉了，这时食材的状态会发生变化，这里用到了策略模式；原料被厨师处理的手法不同，这便里用到了装饰器模式。



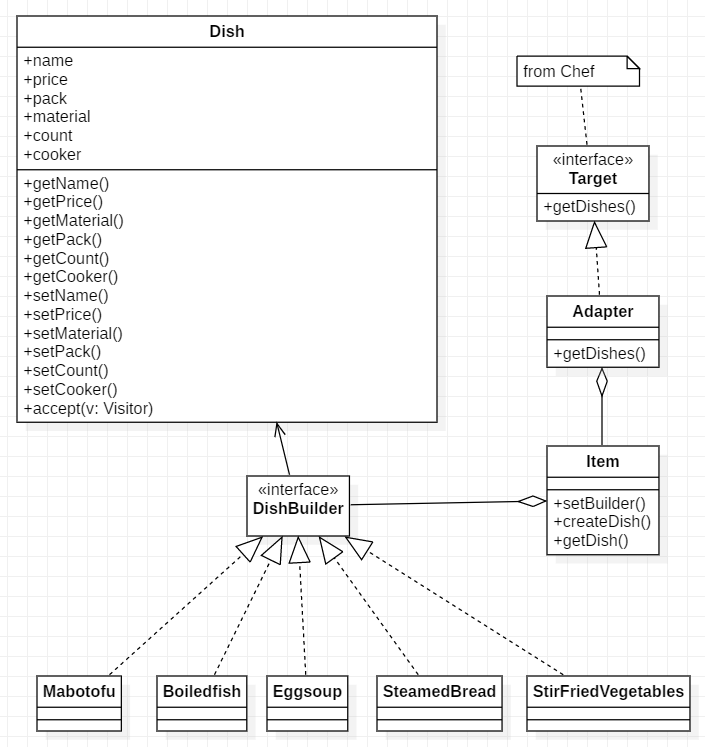
* + 原料类 Ingredient
  + 鸡蛋类 Egg
  + 面粉类 Flour
  + 豆腐类 Tofu
  + 蔬菜类 Vegetables
  + 鱼类 Fish
  + 原料处理接口 CutInterface
  + 原料处理装饰类 FancyCut
  + 原料种类类 IngredientType
  + 状态类 State
  + 新鲜类 Fresh
  + 腐败类 Stale
  + 已用类 Cooked
* **Package** Cooker

厨房中有很多厨具，我们是个中餐厅，当然电饭煲、炒锅、蒸笼是不可或缺的。它们都派生于厨具抽象类，因为每种厨具都会包含开始烹饪、结束烹饪这一流程，这里用到了模板模式；也用到了责任链模式。



* + 厨具类 Cooker
  + 炒锅类 Wok
  + 电饭煲类 RiceCooker
  + 蒸笼类 Steamer
* **Package** Merch

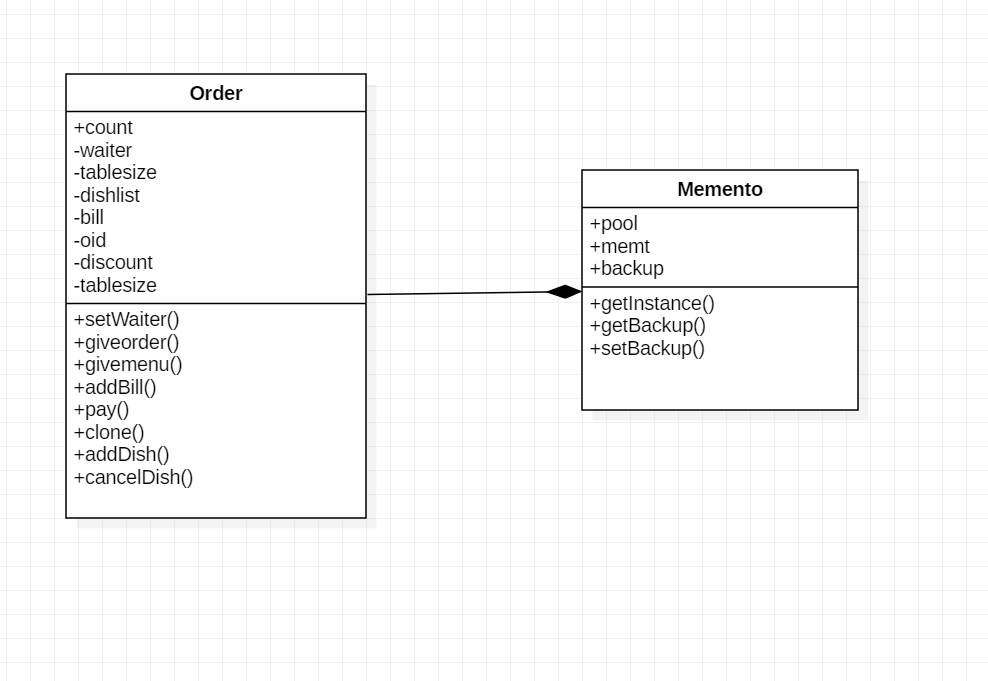
**在我们的厨房中，有着多样的商品，每种商品品都拥有着各自不同的名称、价格等等，商品分为五种：麻婆豆腐、水煮鱼、炒青菜、鸡蛋汤和馒头。在这里，各种商品的属性一致，所以使用建造者模式生成，同时商品的准备过程只由接口实现，步对外暴露过程，这里使用到了外观模式。**



* + 菜品类 Dish
  + 麻婆豆腐类 Mabotofu
  + 水煮鱼类 BoiledFish
  + 鸡蛋汤类 EggSoup
  + 馒头类 SteamedBread
  + 炒青菜类 StirFriedVegetables
  + 菜品生成接口DishBuilder
  + 适配器类 Adapter
  + 商品类 Item
* **Package** Order

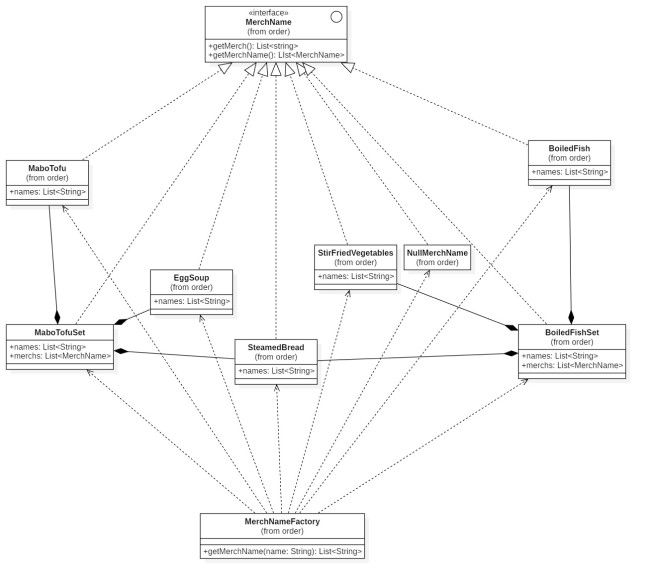
订单是面向顾客，直接和顾客进行交互的类，订单包中封装了订单的需要用到的菜单类；用于接收顾客的输入，判断菜品是否存在的菜单生成类；包含菜单信息的基本订单类，和用于存储订单操作历史的订单历史类。orderform包中，订单的生成应用了原型模式，订单历史的实现应用了单例模式、备忘录模式和享元模式；merchname包中，菜品包括单品和套餐，套餐由若干单品组合而成，其实现运用了组合模式的方法，从顾客输入到菜单实例的生成运用了工厂模式；对于不存在的菜品，使用了空对象模式的方法来表达。

**Package** orderform



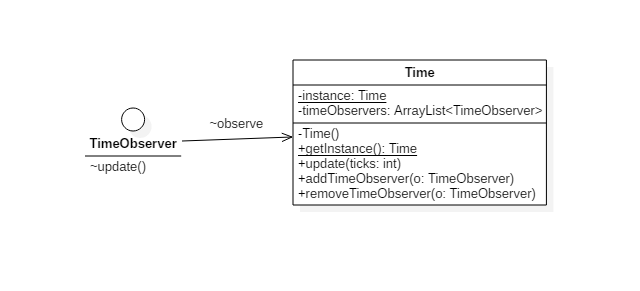
* + 订单类 Order
  + 订单历史类 Memento

**Package** MerchName



* **Package** Time

食材的腐败与时间息息相关，时间包用来处理时间的流动，这里使用了观察者模式；另外因为只有一个时间实例来处理所有观察者，这里也用到了单例模式。



* + 类 Time
  + 接口 TimeObserver

## Design Pattern 详述

### 4.1 单例模式 Singleton

#### 4.1.1 模式简介

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

#### 4.1.2 应用场景

在中餐厅中，我们只有一个厨师与一个服务员，而作为容器冰箱、橱柜也都只有一个，因此我们采用了单例模式来创建实例，并向其它对象提供这一实例。

#### 4.1.3 类图

#### 

#### 

#### 4.1.4 API描述

* **Chef 厨师**

Chef类采用了单例模式，具有自身的私有构造函数与静态实例，提供静态方法：getInstance()获取Chef单例

private static Chef instance;

private Chef() {

if (instance == null) {

instance = this;

} else {

throw new IllegalStateException("Already initialized.");

}

}

public static synchronized Chef getInstance() {

if (instance == null) {

instance = new Chef();

}

return instance;

}

* **Waiter 服务员**

Waiter类采用了单例模式，具有自身的私有构造函数与静态实例，提供静态方法：getInstance()获取Waiter单例

private static Waiter instance = new Waiter();

private Waiter(){

if (instance == null) {

instance = this;

} else {

throw new IllegalStateException("Already initialized.");

}

}

public static Waiter getInstance(){

return instance;

}

* **Container 容器**

Fridge 类与 Cabinet 类继承自 Container 类，也采用了单例模式，同样具有自身的私有构造函数与静态实例，提供 getInstance() 方法。

public class Fridge extends Container {

public Fridge() {

if (instance == null) {

instance = this;

availableTypes = new ArrayList<>(Arrays.asList(

IngredientType.TOFU,

IngredientType.FISH));

} else {

throw new IllegalStateException("Already initialized.");

}

}

private static Fridge instance;

public static synchronized Fridge getInstance() {

if (instance == null) {

instance = new Fridge();

}

return instance;

}

@Override

public String getName()

{

return("冰箱");

}

}

### 4.2 抽象工厂模式 Abstract Factory

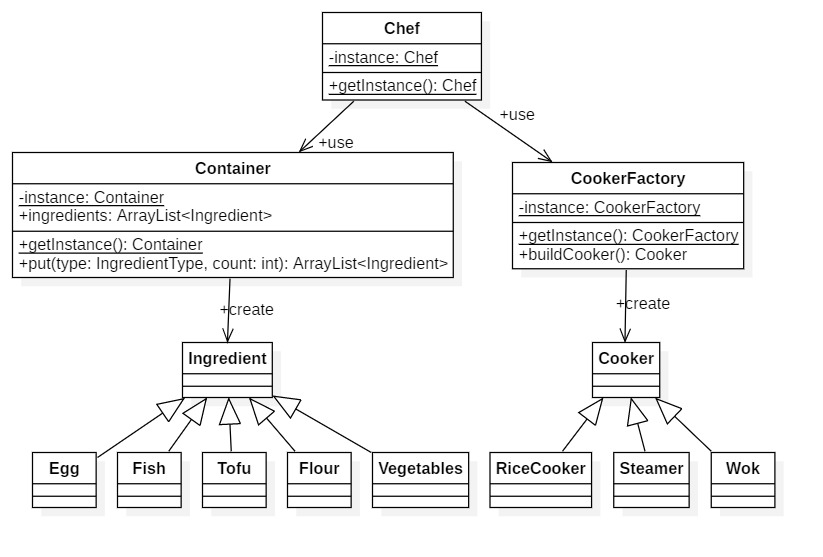
#### 4.2.1 模式简介

抽象工厂模式（Abstract Factory Pattern）是围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。在抽象工厂模式中，接口是负责创建一个相关对象的工厂，不需要显式指定它们的类。每个生成的工厂都能按照工厂模式提供对象。

#### 4.2.2 应用场景

在我们的厨房里，厨师（Chef）使用容器(Container)生成原料，使用厨具工厂(CookerFactory)生成厨具(Cooker)，这本质上就是一个抽象工厂。

#### 4.2.3 类图



#### 4.2.4 API描述

for(Dish dish:dishes){

IngredientType type=chef.transferToIngredientType(dish);

Ingredient ingredient=chef.transferToIngredient(dish);

Container container=chef.selectedContainer(dish);

priorityCount++;

//取原料及其数量并进行消耗

int count=dish.getCount();

boolean isTrue= container.get(type, count);

CookerFactory cookerFactory=CookerFactory.getInstance();

chef.setCookerFactory(cookerFactory);

Cooker newCooker= chef.getCookerFactory().buildCooker(dish);

if(isTrue){

System.out.println("厨师成功从"+"【"+container.getName()+"】"+"获得"

+"【"+type.toString()+"】") ;

mDishes.put(newCooker.getPriority()+priorityCount,dish);

}

else{

System.out.println("厨师获得"+"【"+type.toString()+"】"+"失败");

}

}

### 4.3 工厂模式 Factory

#### 4.3.1 模式简介

工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

#### 4.3.2 应用场景

在订单生成前获取菜名时，根据顾客输入的内容生成菜名列表

#### 4.3.3 类图

#### 工厂

#### 4.3.4 API描述

public class MerchNameFactory {

    /\*\*

     \* 根据客人的输入，将套餐和单品名转换为单品名的List

     \*

     \* @param name 客人输入的商品名或编号

     \* @return 客人输入的内容对应的商品名称的List

     \*/

    static public List<String> getMerchName(String name){

        MerchName merchName=null;

        switch(name){

            case "1":

            merchName=new BoiledFish();

            break;

            case "水煮鱼":

            merchName=new BoiledFish();

            break;

            case "2":

            merchName=new EggSoup();

            break;

            case "鸡蛋汤":

            merchName=new EggSoup();

            break;

            case "3":

            merchName=new MaboTofu();

            break;

            case "麻婆豆腐":

            merchName=new MaboTofu();

            break;

            case "4":

            merchName=new StirFriedVegetables();

            break;

            case "炒青菜":

            merchName=new StirFriedVegetables();

            break;

            case "5":

            merchName=new SteamedBread();

            break;

            case "馒头":

            merchName=new SteamedBread();

            break;

            case "6":

            merchName=new BoiledFishSet();

            break;

            case "水煮鱼套餐":

            merchName=new BoiledFishSet();

            break;

            case "7":

            merchName=new MaboTofuSet();

            break;

            case "麻婆豆腐套餐":

            merchName=new MaboTofuSet();

            break;

            default:

            merchName=new NullMerchName();

            break;

        }

        return merchName.getMerchName();

    }

}

### 4.4 模板模式 Template

#### 4.4.1 模式简介

在模板模式（Template Pattern）中，一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。这种类型的设计模式属于行为型模式。

#### 4.4.2 应用场景

#### 

#### 4.4.3 类图

#### 4.4.4 API描述

Wok类、Steamer类以及RiceCooker类继承了抽象类Cooker并重写了startCooking()和endCooking()方法。

**public abstract class** Cooker {  
**public static int** FIRST=1;  
 **public static int** SECOND=2;  
 **public static int** THIRD=3;  
 **protected int** priority;   
**public int** getPriority(){  
 **return this**.priority;  
 }  
**public** Cooker() {  
 }  
**protected** ArrayList<IngredientType> availableTypes;  
**protected** String name;  
**private boolean** isIngredientTypeAvailable(IngredientType type) {  
 **return** availableTypes.contains(type);  
 }  
**private boolean** isIngredientTypeAvailable(ArrayList<IngredientType> typeList) {  
 **for** (IngredientType itype:typeList) {  
 **if**(!availableTypes.contains(itype))**return false**;  
 }  
 **return true**;  
 }  
**public final void** cook(Ingredient ingredient) {  
 **if**(!isIngredientTypeAvailable(ingredient.getIngredientType())){  
 System.out.println(**"不能用该厨具烹饪该食材！"**);  
 **return**;  
 }  
 startCooking(ingredient);  
 finishCooking(ingredient);  
 }  
  
**public final void** cook(ArrayList<Ingredient> ingredientList) {  
 **for** (Ingredient ingredient:ingredientList) {  
 **if**(!isIngredientTypeAvailable(ingredient.getIngredientType())){  
 System.out.println(**"不能用该厨具烹饪这些食材！"**);  
 **return**;  
 }  
 }  
 startCooking(ingredientList);  
 finishCooking(ingredientList);  
 }  
  
**protected abstract void** startCooking(Ingredient ingredient);  
**protected abstract void** finishCooking(Ingredient ingredient);  
**protected abstract void** startCooking(ArrayList<Ingredient> ingredientList);  
**protected abstract void** finishCooking(ArrayList<Ingredient> ingredientList);  
**public** String getName() {  
 **return** name;  
 }  
**public abstract void** getAvailableType();  
  
}

### 4.5 观察者模式 Observer

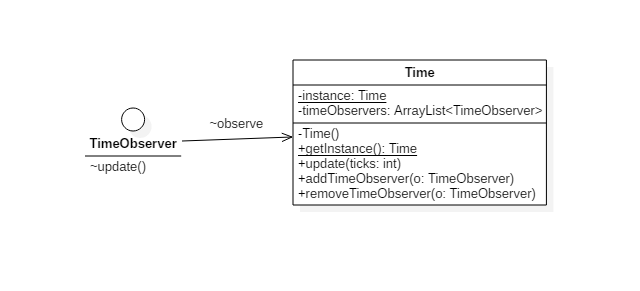
#### 4.5.1 模式简介

当对象间存在一对多关系时，则使用观察者模式（Observer Pattern）。比如，当一个对象被修改时，则会自动通知它的依赖对象。观察者模式属于行为型模式。

#### 4.5.2 应用场景

当时间流动时，食材的新鲜程度发生变化，即时间通知时间的观察者——食材。

#### 4.5.3 类图



#### 4.5.4 API描述

**public class** Time {  
   
 **private** Time() {  
 }  
   
 **private static** Time instance;  
   
 **private** ArrayList<TimeObserver> timeObservers = **new** ArrayList<>();  
   
 **public static synchronized** Time getInstance() {  
 **if** (instance == **null**) instance = **new** Time();  
  
 **return** instance;  
 }  
  
 **public void** update(**int** ticks) {  
 **while**(ticks>0)  
 {  
 **for** (TimeObserver o : timeObservers) o.update();  
 ticks--;  
 }  
  
 }  
   
 **public void** addTimeObserver(TimeObserver o) {  
 timeObservers.add(o);  
 }  
   
 **public void** removeTimeObserver(TimeObserver o) {  
 timeObservers.remove(o);  
 }  
  
}

### 4.6 迭代器模式 Iterator

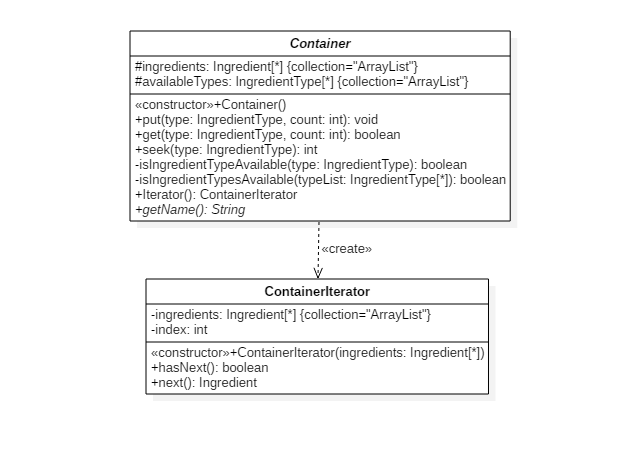
#### 4.6.1 模式简介

迭代器模式（Iterator Pattern）是 Java 和 .Net 编程环境中非常常用的设计模式。这种模式用于顺序访问集合对象的元素，不需要知道集合对象的底层表示。迭代器模式属于行为型模式。

#### 4.6.2 应用场景

当我们想查看容器里有多少东西（比如冰箱），使用迭代器模式可以提供一种更简易的遍历方式。

#### 4.6.3 类图



#### 4.6.4 API描述

**public class** ContainerIterator **implements** Iterator<Ingredient>{  
**private** ArrayList<Ingredient> ingredients;  
**private int** index;  
**public** ContainerIterator(ArrayList<Ingredient> ingredients) {  
 **this**.ingredients = ingredients;  
 index = 0;  
 }  
**public boolean** hasNext() {  
 **return** index < ingredients.size();  
 }  
**public** Ingredient next() {  
 **return** ingredients.get(index++);  
 }  
  
}

### 4.7 状态模式 State

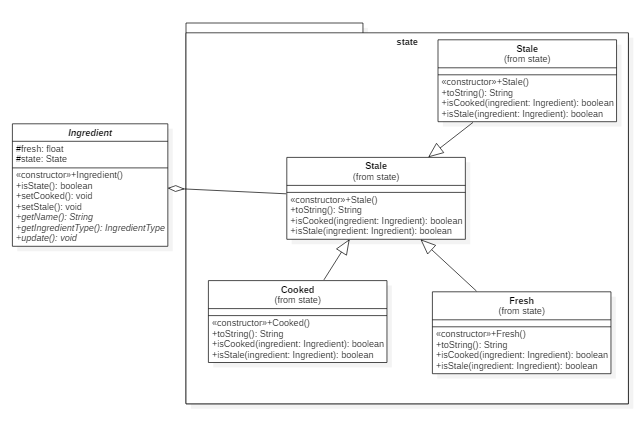
#### 4.7.1 模式简介

在状态模式（State Pattern）中，类的行为是基于它的状态改变的。这种类型的设计模式属于行为型模式。在状态模式中，我们创建表示各种状态的对象和一个行为随着状态对象改变而改变的 context 对象。

#### 4.7.2 应用场景

食物的多种状态会产生多种行为，为了减少if…else…的使用，在食物的状态上使用了状态模式。

#### 4.7.3 类图



#### 4.7.4 API描述

**public interface** State {  
**public abstract boolean** isCooked(Ingredient ingredient);  
**public abstract boolean** isStale(Ingredient ingredient);  
  
}

#### 4.8 命令模式 Command

#### 4.8.1 模式简介

命令模式（Command Pattern）是一种数据驱动的设计模式，它属于行为型模式。请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令。

#### 4.8.2 应用场景

#### 厨师处理订单的操作。

#### 4.8.3 API描述

创建了接口ChefImp，而Chef实现了此接口。

/\*\*

 \* 命令模式

 \*/

public interface ChefImp{

    void processMerchs(ArrayList<Dish> dishs);

}

### 4.9 装饰器模式 Decorator

#### 4.9.1 模式简介

装饰器模式（Decorator Pattern）允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。

#### 4.9.2 应用场景

原料处理方法。

#### 4.9.3 类图

#### 4.9.4 API描述

此模式创建一个装饰类FancyCut，用以实现接口CutInterface。

public class FancyCut implements CutInterface  {

    private Ingredient ingredient;

    public FancyCut(Ingredient obj)

    {

        ingredient = obj;

    }

    @Override

    public void displayCut() {

        if(ingredient.getIngredientType() == IngredientType.EGG)

        {

            System.out.println("厨师使用神来一击，"+ingredient.getName()+"完成了打碎以及搅拌。");

        }

        else if(ingredient.getIngredientType() == IngredientTypeFLOUR

        {

            System.out.println("厨师使用推拿掌法，"+ingredient.getName()+"和成活面。");

        }

        else

        {

            System.out.println("厨师使用刀法飞龙在天，"+ingredient.getName()+"完成了切片。");

        }

    }

    @Override

    public boolean isCut() {

        return true;

    }

}

### 4.10 策略模式 Strategy

#### 4.10.1 模式简介

在策略模式（Strategy Pattern）中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 context 对象。策略对象改变 context 对象的执行算法。

#### 4.10.2 应用场景

原料在不同的时间，在被处理前后，对应的状态需要转换。

#### 4.10.3 类图

#### 4.10.4 API描述

设置食材的状态。

public abstract class Ingredient implements TimeObserver {

   public Ingredient() {

        state = new Fresh();

    }

    public void setCooked() {

        state = new Cooked();

    }

    public void setStale() {

        state = new Stale();

    }

}

### 4.11 建造者模式 Builder

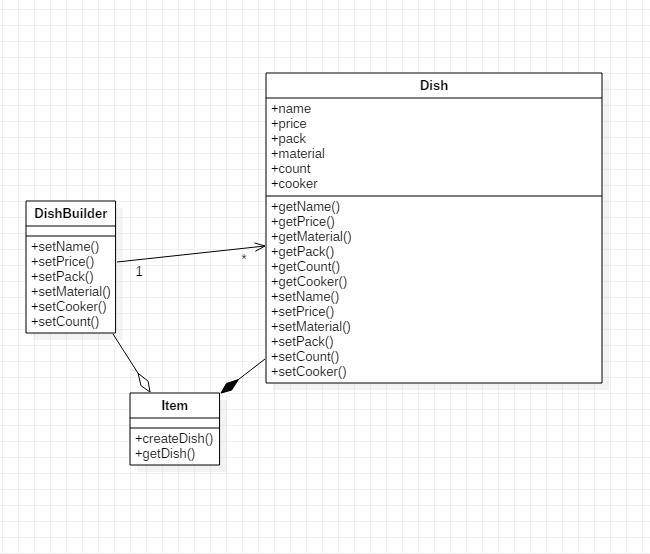
#### 4.11.1 模式简介

建造者模式（Builder Pattern）使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

#### 4.11.2 应用场景

#### 在从接收到的订单中创建新的商品时，CreateItem类通过商品类调用builder来生成新的菜品。

#### 4.11.3 类图



#### 4.11.4 API描述

创建了Dishbuilder接口，在执行商品的生成时，调用builder来生成不同的菜品类。

preMerch(ArrayList<Integer> a)

根据菜单创建商品

preMerch(ArrayList<Integer> a){

if(a.get(i) == 1) item.setbuilder(new MaboTofu());

else if(a.get(i) == 2) item.setbuilder(new BoiledFish());

else if(a.get(i) == 3) item.setbuilder(new EggSoup());

else if(a.get(i) == 4) item.setbuilder(new StirFriedVegetables());

else item.setbuilder(new SteamedBread());

item.createDish();

Dish dish = item.getDish();

}

### public interface DishBuilder

### 建造者模式：

public interface DishBuilder {

//创建一个抽象的菜品生成过程的Builder类

void setName();

void setPack();

void setPrice();

void setMaterial();

void setCooker();

void setCount();

Dish getDish();

void accept(Visitor visitor);

}

### 4.12 桥接模式 Bridge

#### 4.12.1 模式简介

桥接模式（Bridge）是软件设计模式中较为复杂的模式之一，用于把抽象化和现实化解耦，使二者可以独立变化，这种类型的设计模式属于结构型模式，它通过提供抽象化和现实化之间的桥接结构来实现二者的解耦。这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类。这两种类型的类可被结构化改变而互不影响。

#### 4.12.2 应用场景

#### 在订单部分，我们在计算订单的价格的时候使用了商品的内部方法来获取商品的价格，从而获取订单的总价

#### 4.12.3 类图

#### 4.12.4 API描述

调用的是dish类中的getPrice函数用于计算订单的价格。

/\*\*

\* 服务员visit订单，获取内容并计算总价

\* @param order

\*/

@Override

public void visit(Order order) {

ArrayList<String> list = order.getOrderList();

for (String string: list) {

switch (string) {

case "BoiledFish":

totalPrice += this.visit(new BoiledFish());

break;

case "EggSoup":

totalPrice += this.visit(new EggSoup());

break;

case "MaboTofu":

totalPrice += this.visit(new MaboTofu());

break;

case "SteamedBread":

totalPrice += this.visit(new SteamedBread());

break;

case "StirFriedVegetables":

totalPrice += this.visit(new StirFriedVegetables());

break;

default:break;

}

double discount = order.getDiscount();

System.out.println("共消费：" + totalPrice \* discount + "元");

}

}

### 4.13 外观模式 Facade

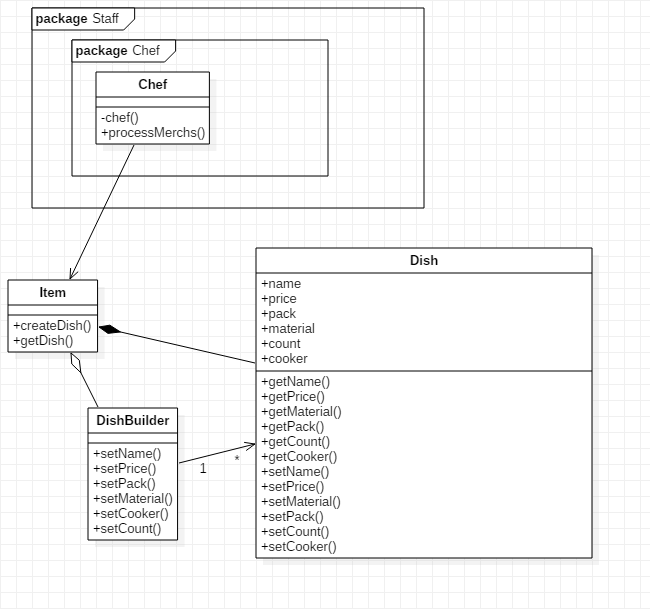
#### 4.13.1 模式简介

外观模式（Facade Pattern）隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。这种类型的设计模式属于结构型模式，它向现有的系统添加一个接口，来隐藏系统的复杂性。这种模式涉及到一个单一的类，该类提供了客户端请求的简化方法和对现有系统类方法的委托调用。外观模式多用于为复杂模块提供外部访问接口，提高子系统的相对独立性。

#### 4.13.2 应用场景

#### 商品的制作是一个复杂的过程，我们对顾客隐藏做菜的细节，只暴露一个接口函数processMerchs来隐藏具体实现。

#### 4.13.3 类图



#### 4.13.4 API描述

对外只有准备商品这一接口，内部逻辑是取食材、取厨具、加工、完成制作。

public void preMerch(ArrayList<Integer> a) {

Item item = new Item();

for(int i=0; i<a.size(); i++){

//判断点的是哪一道菜

if(a.get(i) == 1) item.setbuilder(new MaboTofu());

else if(a.get(i) == 2) item.setbuilder(new BoiledFish());

else if(a.get(i) == 3) item.setbuilder(new EggSoup());

else if(a.get(i) == 4) item.setbuilder(new StirFriedVegetables());

else item.setbuilder(new SteamedBread());

item.createDish();

Dish dish = item.getDish();

totalprice += dish.getPrice();

ArrayList<Dish> allmerch = new ArrayList<>();

allmerch.add(dish);

}

System.out.print("总价格是:"+getprice()+"元\n");

//调用厨师包的函数进行菜品的生成

Chef chef = Chef.getInstance();

chef.processMerchs(allmerch);

}

### 4.14 享元模式 Flyweight

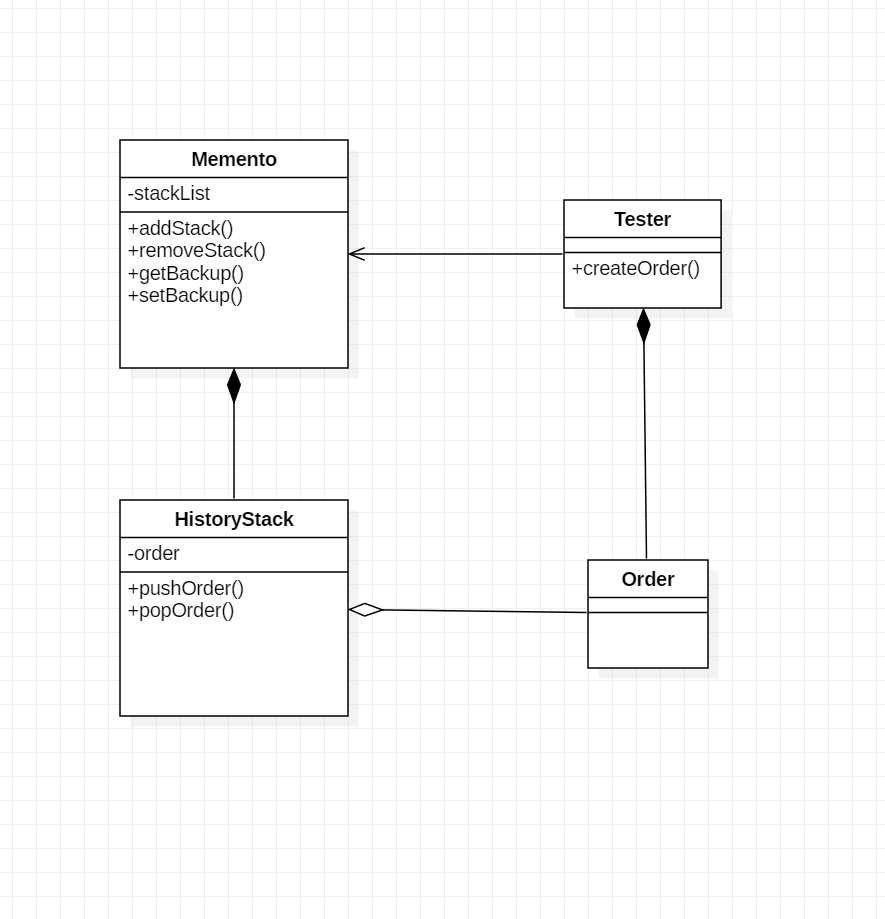
#### 4.14.1 模式简介

享元模式（Flyweight Pattern）主要用于减少创建对象的数量，以减少内存占用和提高性能。这种类型的设计模式属于结构型模式，它提供了减少对象数量从而改善应用所需的对象结构的方式。享元模式尝试重用现有的同类对象，如果未找到匹配的对象，则创建新对象。享元模式大多用在系统中有大量相似的对象或者需要缓冲池的场景。

#### 4.14.2 应用场景

#### 对于每一个订单，使用一个栈来存储其历史信息。对这个栈的维护，可以使用享元模式。即对于相同订单号的订单，共享一个历史栈；只有对于新建的订单，才会生成一个新的历史栈。

#### 4.14.3 类图



#### 4.14.4 API描述

public void setBackup( Order o ) throws CloneNotSupportedException {

Order backup = (Order)o.clone();

if( pool.containsKey(o.getOid()) ) {

pool.get(o.getOid()).push(o);

}

else{

Stack<Order> s = new Stack<Order>();

s.push(o);

pool.put(o.getOid(), s);

}

}

### 4.15 原型模式 Prototype

#### 4.15.1 模式简介

原型模式（Prototype Pattern）是用于创建重复的对象，同时又能保证性能。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。该模式实现了一个原型接口，该接口用于创建当前对象的克隆。当直接创建对象的代价比较大时，则采用这种模式。

#### 4.15.2 应用场景

#### 在Order类中实现Cloneable接口的clone()函数，可以使用一个现有订单的clone()函数来生成一个新的订单，可以用于推荐套餐的生成，在Memento模式生成历史记录时也可用于生成历史实例。

#### 4.15.3 类图

#### 

#### 4.15.4 API描述

Order类提供的clone()需要实现深拷贝，即对于指针类型的成员变量，不能直接赋值，而是要分配新的空间，填入指针指向的类型的值（如果再遇到指针则重复分配空间，进行上上述操作，直至没有指针变量为止），再返回新的指针。

@Override

public Object clone() throws CloneNotSupportedException {

Order o = (Order)super.clone();

ArrayList<String> dl = new ArrayList<String>();

o.tablesize = this.tablesize;

o.discount = this.discount;

o.bill = this.bill;

o.waiter = this.waiter;

for(String s:this.dishlist){

dl.add(s);

}

o.dishlist = dl;

return o;

}

### 4.16 备忘录模式 Memento

#### 4.16.1 模式简介

备忘录模式（Memento Pattern）保存一个对象的某个状态，以便在适当的时候恢复对象。备忘录模式属于对象行为型模式。备忘录对象是一个用来存储另外一个对象内部状态的快照的对象。备忘录模式的用意是在不破坏封装的条件下，捕获一个对象的状态捕捉，并外部化，存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。

#### 4.16.2 应用场景

#### 在每一次顾客对订单进行修改时，生成一个修改前订单的备份，所有的备份形成一份订单的历史记录，顾客可以随时回退到之前某个订单状态。

#### 4.16.3 类图

#### 4.16.4 API描述

Memento中，存储的数据类型实际上是HashTable<Integer, Stack<Order>>，HashTable的key值对应订单的编号，相同订单号的不同订单备份形成一个历史订单栈。这个数据的维护方法如下：

public Order getBackup( Order o ) {

Stack<Order> backup = (Stack<Order>)pool.get(o.getOid());

if(backup == null || backup.empty()) {

System.out.println("此订单不存在备份。");

return o;

}

return backup.pop();

}

public void setBackup( Order o ) throws CloneNotSupportedException {

Order backup = (Order)o.clone();

if( pool.containsKey(o.getOid()) ) {

pool.get(o.getOid()).push(o);

}

else{

Stack<Order> s = new Stack<Order>();

s.push(o);

pool.put(o.getOid(), s);

}

}

### 4.17 空对象模式 Null Object

#### 4.17.1 模式简介

空对象模式（Null Object Pattern），用一个空对象取代 NULL 对象实例的检查。Null 对象不是检查空值，而是反应一个不做任何动作的关系。这样的 Null 对象也可以在数据不可用的时候提供默认的行为。

#### 4.17.2 应用场景

#### 在客人进行点单时，可能会点到菜单中没有的商品。

#### 4.17.3 类图

#### 空对象

#### 4.17.4 API描述

在点单时，如果点到不存在的商品，就会生成一个空对象，空对象内部没有商品名，随后会提醒客人点单失败

public class NullMerchName implements MerchName {

    @Override

    public List<MerchName> getMerch() {

        return null;

    }

    @Override

    public List<String> getMerchName() {

        return null;

    }

}

### 4.18 组合模式 Composite

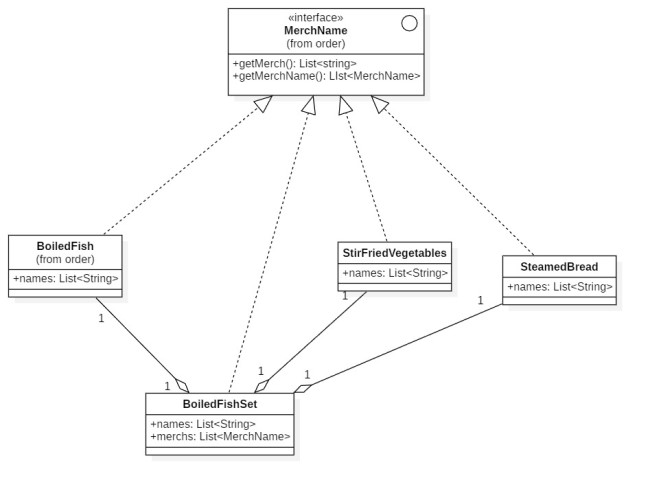
#### 4.18.1 模式简介

组合模式（Composite Pattern），又叫部分整体模式，是用于把一组相似的对象当作一个单一的对象。组合模式依据树形结构来组合对象，用来表示部分以及整体层次。这种类型的设计模式属于结构型模式，它创建了对象组的树形结构。这种模式创建了一个包含自己对象组的类。该类提供了修改相同对象组的方式。

#### 4.18.2 应用场景

#### 菜单中的商品不只有单品，还有套餐

#### 4.18.3 类图



#### 4.18.4 API描述

套餐和单品都实现同一个接口，但套餐类中有一个List，其中包含了套餐内的所有单品

Code

public class BoiledFishSet implements MerchName {

    public final static List<String> names= List.of("水煮鱼","炒青菜","馒头");

public final static List<MerchName> merchs=List.of(

new BoiledFish(),

new StirFriedVegetables(),

new SteamedBread());

    @Override

    public List<MerchName> getMerch() {

        return merchs;

    }

    @Override

    public List<String> getMerchName() {

        return names;

    }

}

### 4.19 访问者模式 Visitor

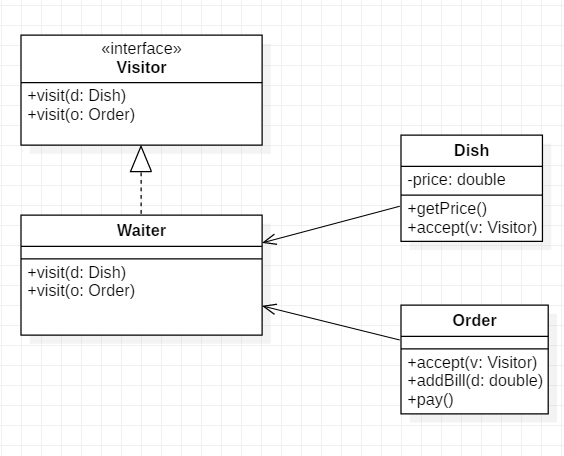
#### 4.19.1 模式简介

在访问者模式（Visitor Pattern）中，我们使用了一个访问者类，它改变了元素类的执行算法。通过这种方式，元素的执行算法可以随着访问者改变而改变。这种类型的设计模式属于行为型模式。根据模式，元素对象已接受访问者对象，这样访问者对象就可以处理元素对象上的操作。

#### 4.19.2 应用场景

服务员访问一个菜的时候可以看到它的价格，访问一个订单的时候，算出订单中所有菜的总价。

#### 4.19.3 类图



#### 4.19.4 API描述

**Visitor 抽象类**

public interface Visitor {

void visit(Dish dish);

void visit(Order order);

}

**Waiter继承Visitor**

@Override

public void visit(Dish dish) {

System.out.println("visit " + dish.getName() + ":" + dish.getPrice() + "元");

}

@Override

public void visit(Order order) {

System.out.println("visit order");

//测试waiter时注释以下部分

for (Dish dish: dishes) {

this.visit(dish);

mOrder.addBill(dish.getPrice());

}

mOrder.pay();

}

**DishBuilder接口**

void accept(Visitor visitor);

**Dish类**

public void accept(Visitor visitor){

visitor.visit(this);

}

**BoiledFish类（举一例）**

@Override

public void accept(Visitor visitor){

this.getDish().accept(visitor);

}

**Order类**

public void addBill(double money){

this.bill += money;

}

public void pay(){

System.out.println("顾客共消费：" + (this.bill \* this.discount) + "元。");

}

public void accept(Visitor visitor){

visitor.visit(this);

}

**Main中**

public static void main(String[] args) throws CloneNotSupportedException {

Waiter waiter = Waiter.getInstance();

Order order = new Order();

order.setDiscount(0.8);

order.adddish("水煮鱼");

order.adddish("鸡蛋汤");

order.adddish("馒头");

waiter.register(order);

waiter.serve(order);

System.out.println("--测试访问者模式--");

System.out.println("----访问菜品----");

waiter.getDishes().get(0).accept(waiter);

System.out.println("----访问订单----");

order.accept(waiter);

}

### 4.20 适配器模式 Adapter

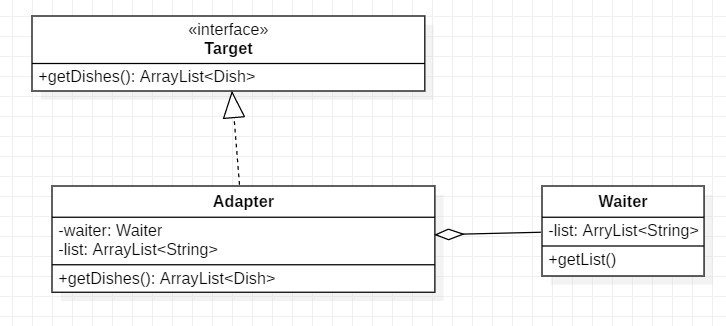
#### 4.20.1 模式简介

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。这种模式涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能。适配器分为类适配器和对象适配器，本例展示对象适配器。

#### 4.20.2 应用场景

服务员手里有一个字符串的列表，这样一个列表厨师无法直接使用。厨师发出一个请求（Target）要求拿到一个菜的列表。适配器接管了这个请求，取到服务员单例中的字符串列表进行处理，并返回一个菜的列表。

#### 4.20.3 类图



#### 4.20.4 API描述

**Target 厨师请求**

public interface Target {

ArrayList<Dish> getDishes();

}

**Adapter 适配器**

public class Adapter implements Target {

private Waiter waiter = Waiter.getInstance();

private ArrayList<String> list = waiter.getLists();

@Override

public ArrayList<Dish> getDishes() {

System.out.println("--使用适配者模式--");

ArrayList<Dish> dishes = new ArrayList<>();

for (String string: list){

Item item = new Item();

int flag = 1;

switch (string){

case "水煮鱼":

item.setbuilder(new BoiledFish());

break;

case "鸡蛋汤":

item.setbuilder(new EggSoup());

break;

case "麻婆豆腐":

item.setbuilder(new MaboTofu());

break;

case "馒头":

item.setbuilder(new SteamedBread());

break;

case "炒青菜":

item.setbuilder(new StirFriedVegetables());

break;

default:flag = 0;

}

if (flag == 1){

item.createDish();

Dish dish = item.getDish();

dishes.add(dish);

}

else {

System.out.println("菜单有误，请检测");

return null;

}

}

return dishes;

}

}

### 4.21 中介者模式 Mediator

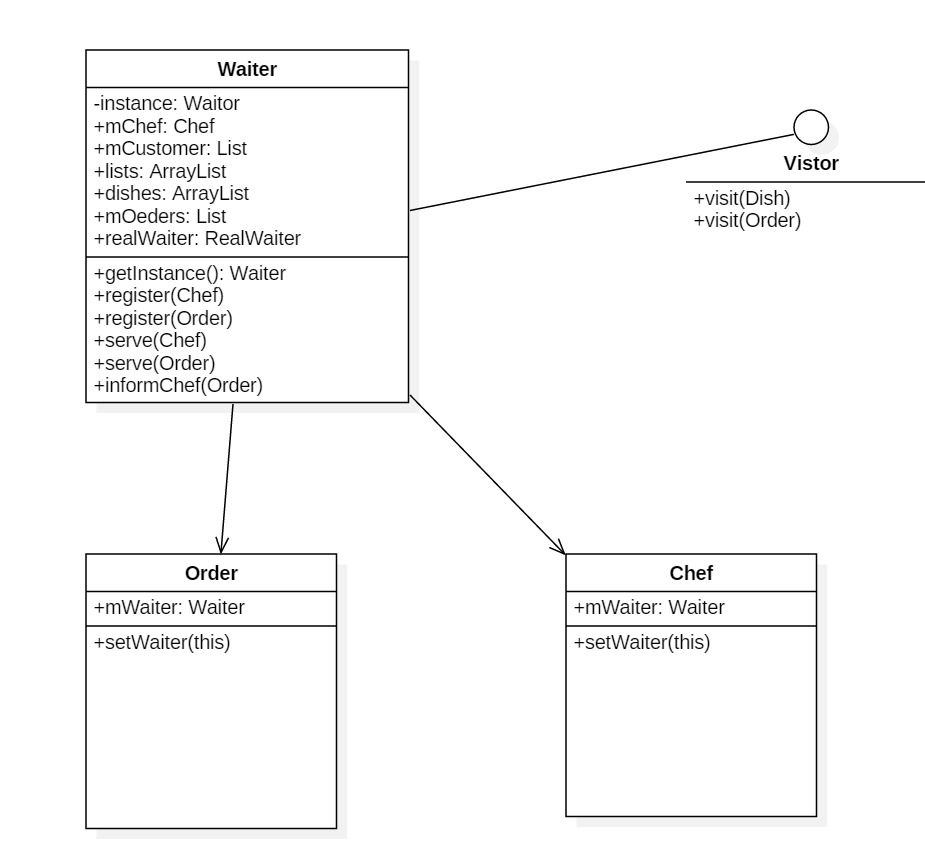
#### 4.21.1 模式简介

中介者模式（Mediator Pattern）用来降低多个对象和类之间的通信复杂性,其属于行为型模式。它提供了一个中介类，该类通常处理不同类之间的通信，并支持松耦合，使代码易于维护。

#### 4.21.2 应用场景

#### 在创建好了订单之后，订单并不直接传递给厨师，而是以服务员为中介者对订单里的数据进行处理后再将处理好的数据传递给厨师

#### 4.21.3 类图



#### 4.21.4 API描述

在厨师类或订单类完成了自己的工作后需要相互传递数据时，调用Waiter的serve方法来传递（实际使用realwaiter的serve函数实现具体功能）

//为厨师服务的方法

//厨师做好菜后使用

@Override

public void serve(Chef chef){

System.out.println("代理模式开始，真实服务员服务厨师");

realWaiter.serve(chef);

System.out.println("真实服务员服务厨师结束，代理模式结束");

visit(mOrder);

}

//为客人服务的方法

//客人点单完成后使用

@Override

public void serve(Order order){

System.out.println("代理服务订单开始");

realWaiter.serve(order);

System.out.println("代理服务订单结束");

lists = realWaiter.getLists();

dishes = realWaiter.getDishes();

}

### 4.22 代理模式 Proxy

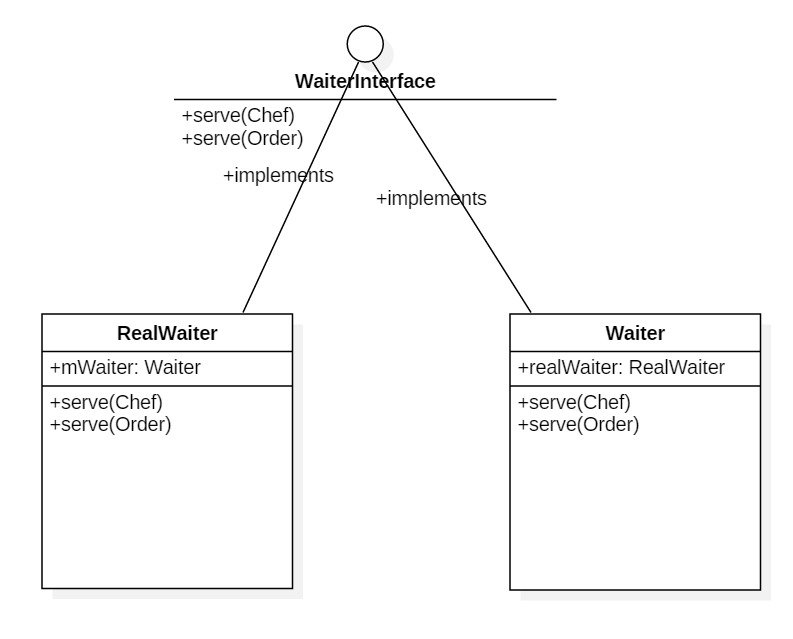
#### 4.22.1 模式简介

代理模式（Proxy Pattern）是一种结构型模式，它设计一个类代表另一个类的功能，创建具有现有对象的对象，利用中间层，以便向外界提供功能接口。

#### 4.22.2 应用场景

#### 当需要使用真实服务员的功能时，并不直接使用真实服务员来操作，而是通过使用代理服务员来先进行预处理在调用真实服务员的功能

#### 4.22.3 类图



#### 4.22.4 API描述

当Waiter类中的serve函数被调用时，实际上会起作用的是RealWaiter中的serve函数

//为厨师服务的方法

//厨师做好菜后使用

@Override

public void serve(Chef chef){

System.out.println("服务员已经拿到菜了");

}

//为客人服务的方法

//客人点单完成后使用

@Override

public void serve(Order order){

lists = order.givemenu();

if(lists != null) {

Adapter adapter = new Adapter();

dishes = adapter.getDishes();

//此步可将订单进一步处理再传递给厨师，测试visitor的时候注释掉

System.out.println("订单开始传递给厨师");

informChef();

}

else

System.out.println("菜单有误");

}

### 4.23 责任链模式 Chain of Responsibility

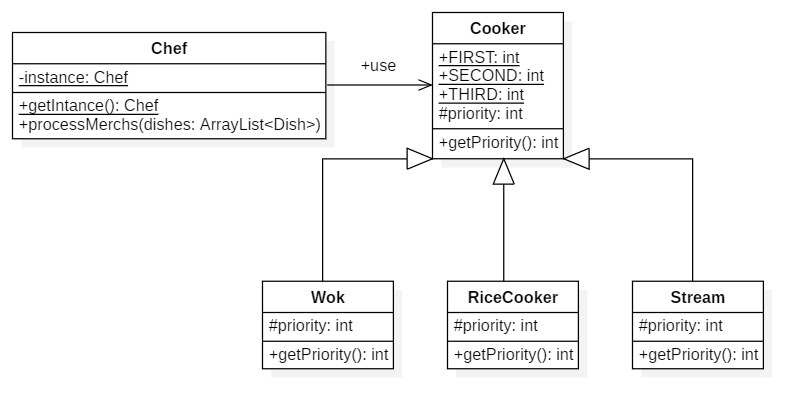
#### 4.23.1 模式简介

责任链模式（Chain of Responsibility Pattern）为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。这种类型的设计模式属于行为型模式。在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。

#### 4.23.2 应用场景

在我们的厨房中，厨师可以使用不同的厨具进行，根据烹饪时间的长短，进行厨具选择使用，采用HashMap实现厨具的动态选择，实现责任链。

#### 4.23.3 类图



#### 4.23.4 API描述

对外只是厨师使用适当的厨具完成了烹饪，实际上是对厨具的责任链进行了选择

priorityCount=1;  
Iterator<Map.Entry<Integer, Dish>> it = mDishes.entrySet().iterator();  
while(it.hasNext()){  
 Map.Entry<Integer,Dish> mEntry = it.next();  
 int key = mEntry.getKey();  
 Dish mDish=mEntry.getValue();  
 Ingredient mIngredient=chef.transferToIngredient(mDish);  
 Cooker mNewCooker= chef.getCookerFactory().buildCooker(mDish);  
 System.*out*.println("【"+priorityCount+"】： ");  
 mNewCooker.cook(mIngredient);  
 priorityCount++;  
}

### 4.24 解释器模式 Interpreter

#### 4.24.1 模式简介

解释器模式（Interpreter Pattern）提供了评估语言的语法或表达式的方式，它是一种行为型模式，其实现了一个表达式接口，该接口解释一个特定的上下文。

#### 4.24.2 应用场景

#### 在订单交付给厨师前，接受商品的名字(string)并通过解释器将其实例化

#### 4.24.3 类图

#### 4.24.4 API描述

通过解释器由string的list生成关于dish的list。

for (String string: list){

Item item = new Item();

int flag = 1;

switch (string){

case "BoiledFish":

item.setbuilder(new BoiledFish());

break;

case "EggSoup":

item.setbuilder(new EggSoup());

break;

case "MaboTofu":

item.setbuilder(new MaboTofu());

break;

case "SteamedBread":

item.setbuilder(new SteamedBread());

break;

case "StirFriedVegetables":

item.setbuilder(new StirFriedVegetables());

break;

default:flag = 0;

}

if (flag == 1){

item.createDish();

Dish dish = item.getDish();

dishes.add(dish);

}

else {

System.out.println("菜单有误，请检测");

return null;

}[下载视频](javascript:;)