

2019 October

软件架构与**设计模式** 实现文档

指导教师：侯捷 冯巾松

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

## 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。**

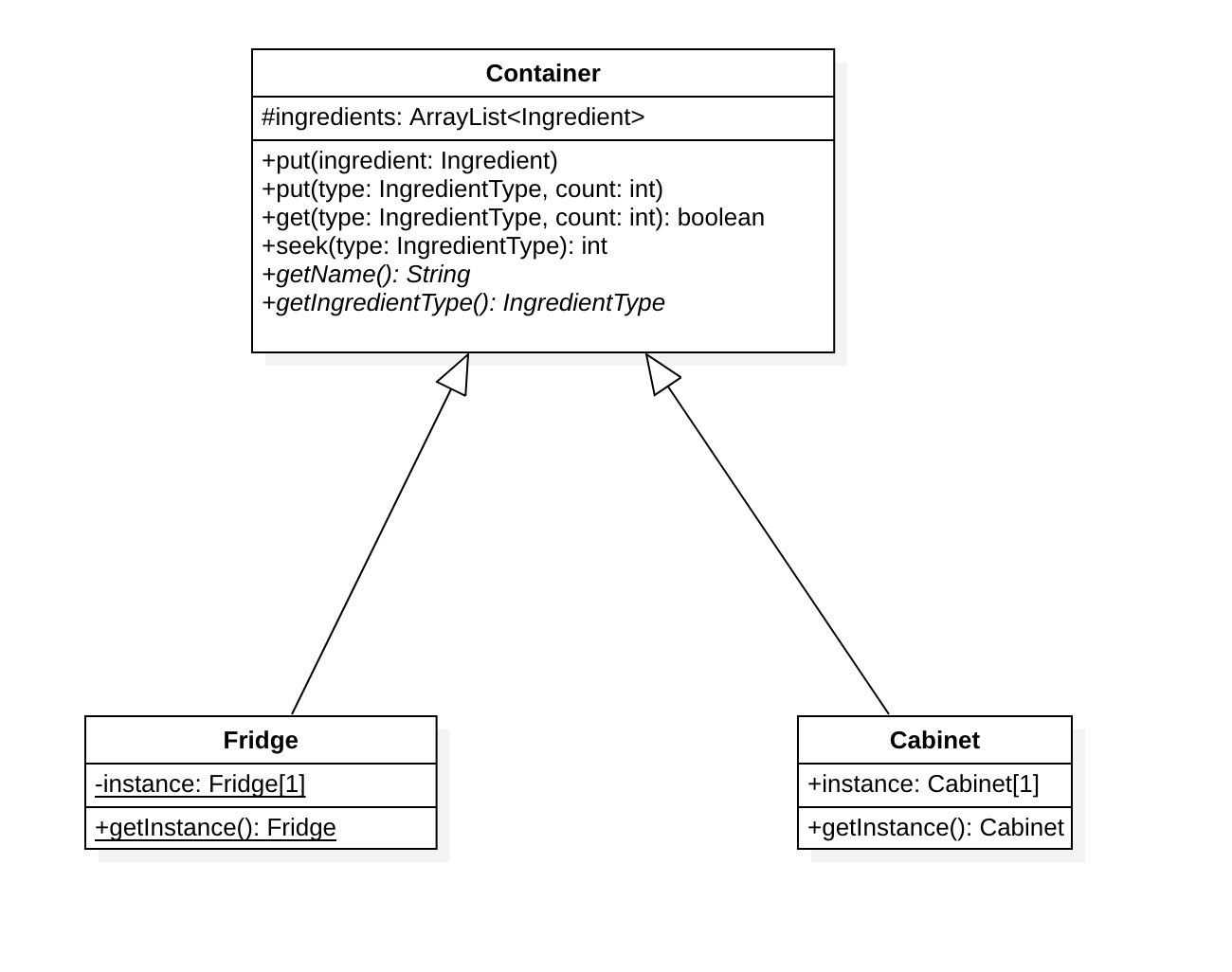
我们的项目由七个 package 组成：

* + 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

文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本

* + 类 Staff
* **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

文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本

* + 类 Ingredient
* **Package** Cooker

文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本

* + 类 Cooker
* **Package** Merch

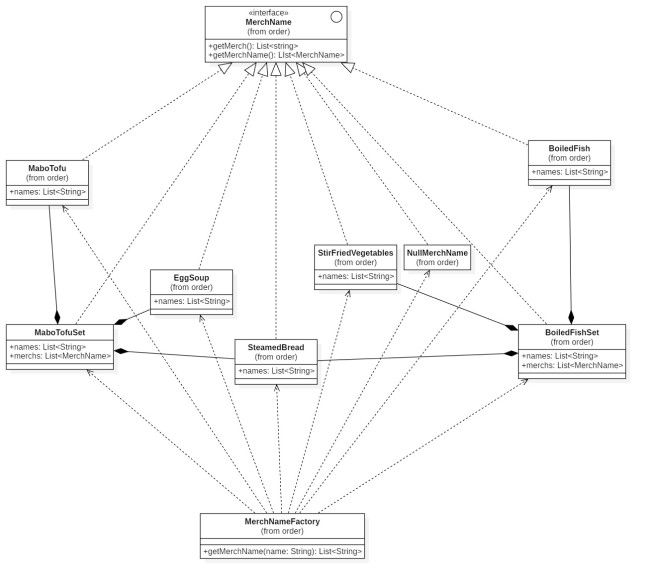
文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本

* + 类 Merch
* **Package** Order

**Package** MerchName

在开始时，客人需要根据菜单挑选商品，商品包括单品或套餐，套餐由若干单品组合而成，这里就体现了组合模式，在订单生成之前，客人选择的商品以菜名的形式存在，然后会通过一个菜名处理工厂把单品和套餐转换成各种单品的序列，这就体现了工厂模式。同时，如果客人误点了厨房中没有的商品，则会产生一个“空菜名”，这就体现了空对象模式。

* + 类 Order
  + 类 MerchName



* **Package** Time

文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本文本

* + 类 Time

## Design Pattern 详述

### 4.1 单例模式 Singleton

#### 4.1.1 模式简介

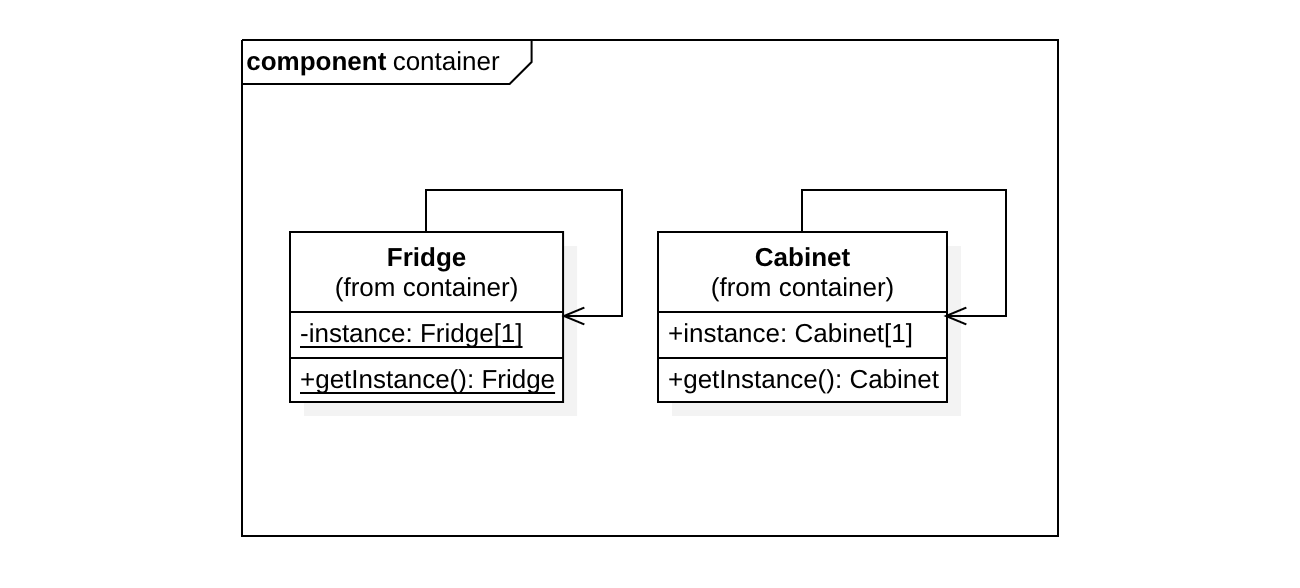
单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

#### 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

#### 少chef、waiter的图



#### 4.1.4 API描述

* **Chef 厨师**

文本

Code here

* **Waiter 服务员**

文本

Code here

* **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

#### 4.2.3 类图

#### 4.2.4 API描述

文本

Code

### 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

#### 4.3.3 类图

#### 4.3.4 API描述

文本

Code

### 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

文本

Code

### 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

#### 4.5.3 类图

#### 4.5.4 API描述

文本

Code

### 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

#### 4.6.3 类图

#### 4.6.4 API描述

文本

Code

### 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

#### 4.7.3 类图

#### 4.7.4 API描述

文本

Code

### 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

#### 4.8.3 类图

#### 4.8.4 API描述

文本

Code

### 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

菜品装盘

#### 4.9.3 类图

#### 4.9.4 API描述

文本

Code

### 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

#### 4.10.3 类图

#### 4.10.4 API描述

文本

Code

### 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

#### 4.11.3 类图

#### 4.11.4 API描述

文本

Code

### 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

#### 4.12.3 类图

#### 4.12.4 API描述

文本

Code

### 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

#### 4.13.3 类图

#### 4.13.4 API描述

文本

Code

### 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

#### 4.14.3 类图

#### 4.14.4 API描述

文本

Code

### 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

#### 4.15.3 类图

#### 4.15.4 API描述

文本

Code

### 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

#### 4.16.3 类图

#### 4.16.4 API描述

文本

Code

### 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

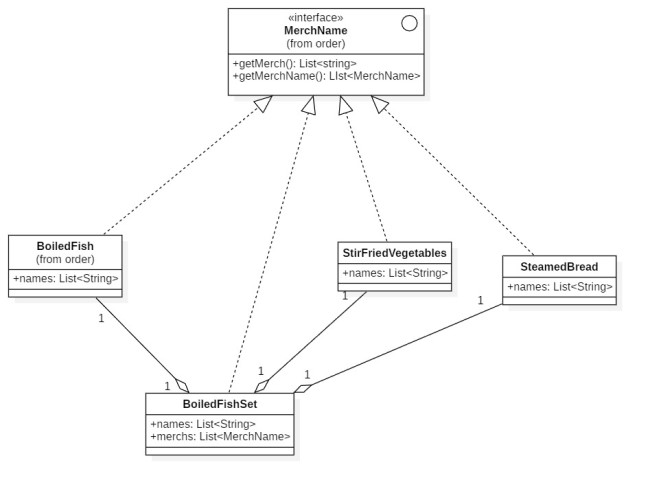
#### 4.18.1 模式简介

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

#### 4.19.3 类图

#### 4.19.4 API描述

文本

Code

### 4.20 适配器模式 Adapter

#### 4.20.1 模式简介

适配器模式（Adapter Pattern）作为两个不兼容的接口之间的桥梁，它是一种结构型模式，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。

#### 4.20.2 应用场景

对于两种饮料，需要不同的类进行生成，两个不同的类通过适配器模式实现兼容，实现制作不同种类饮料的功能。

#### 4.20.3 类图

#### 4.20.4 API描述

文本

Code

### 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

#### 4.21.3 类图

#### 4.21.4 API描述

文本

Code

### 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

#### 4.22.3 类图

#### 4.22.4 API描述

文本

Code

### 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

#### 4.23.3 类图

#### 4.23.4 API描述

文本

Code

### 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

#### 4.24.3 类图

#### 4.24.4 API描述

文本

Code

[下载视频](javascript:;)