同济大学 TONGJI UNIVERSITY

November 2, 2018
版本: 1.0

系统设计与实现文档

软件架构与设计模式

指导教师: 侯捷 冯巾松

阿思亘 *(1652819)*
林钰涵 *(1652723)*
何卓坤 *(1552743)*
杨明荟 *(1652703)*
李天阳 *(1652726)*
温　悦 *(1652695)*
汪涵秋 *(1652704)*
沈枭昂 *(1652672)*
王嘉诚 *(1652673)*
马鸣啸 *(1652727)*

题材综述

上海是一座繁华的城市，在这座魔幻之都里，一切皆有可能。比如，就有这样一家欢乐餐厅，它只售卖汉堡和饮料，却隐含着设计模式的迷幻奥秘。

当顾客来到餐厅时，他可以下单一份汉堡或饮料，如果是大胃王的话，点上几个汉堡或是几个饮料也没什么不可能。当顾客付钱、厨师接到订单时，他会按照顾客的要求呈现美味，聪明地他也会无视订单中各种商品的下单顺序，用他的智慧完成烹饪的过程。如果需要制作一个汉堡，厨师会从壁橱中拿出汉堡坯，然后根据不同的汉堡种类做出不同的反应——烤一块牛肉饼做成牛肉汉堡，烤一片培根做成培根汉堡，或是两个都来，做成双拼豪华汉堡，享受双份美味。肉被厨师从冰箱中取出，放上烤架，烤熟后放入操作台上的汉堡坯中，别忘了加上生菜和西红柿，做成的汉堡就可以放到托盘里了。

啊呀？汉堡肉烤糊了？没关系，神奇的欢乐餐厅总有办法，哔哔哔叭叭哔唠！肉就复原到生的状态了！

诶？没有牛肉饼/培根/汉堡坯/西红柿/生菜了？没关系，欢乐餐厅里的厨师说话好听又厉害，他可以去进货，也可以暂时创建一个缺少材料的“空汉堡”，原料到了再加进去。

当顾客想要的所有美味都进入托盘后，顾客就可以享用了。享用美味，就在神奇的欢乐餐厅！

Design Pattern 汇总表

编号	Design pattern name	实现个（套）数	sample programs个数	备注
1	单例模式 Singleton	3	2	
2	抽象工厂模式 Abstract Factory	1	2	
3	工厂模式 Factory	2	2	
4	模板模式 Template	1	1	
5	观察者模式 Observer	1	2	
6	迭代器模式 Iterator	2	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	2	
12	桥接模式 Bridge	1	1	
13	外观模式 Facade	1	2	
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	2	1	
18	组合模式 Composite	1	1	
19	访问者模式 Visitor	1	1	

项目结构 Package Structure

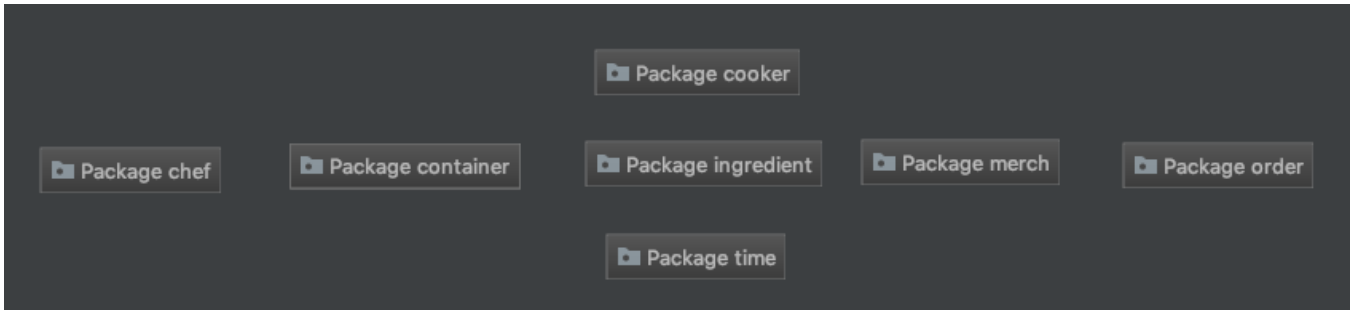
- **总览**

  由于项目是由 Java 编写，包结构则自然是项目架构的重中之重。我们的设计遵循了 Java 包结构设计的统一标准：

  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。同一个包中的类名字是不同的，不同的包中的类的名字是可以相同的，当同时调用两个不同包中相同类名的类时，应该加上包名加以区别。因此，包可以避免名字冲突。

  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。

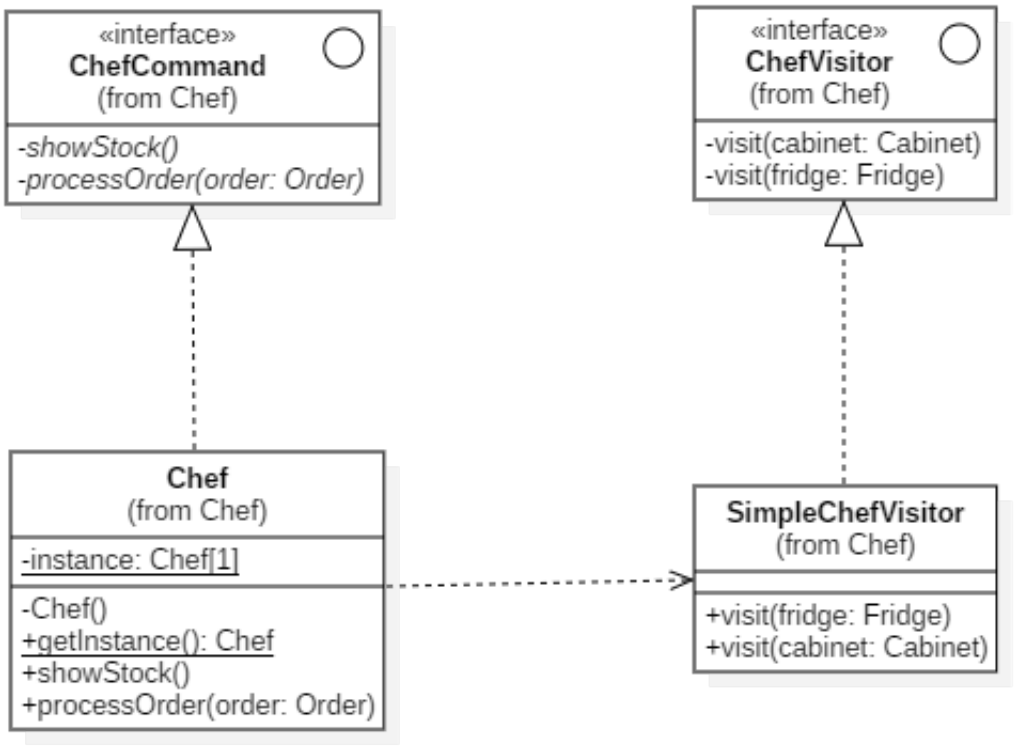
  

  由图可以得知，我们的项目由七个 package 组成：

- 厨师 `Chef`
- 容器 `Container`
- 原料 `Ingredient`
- 厨具 `Cooker`
- 商品 `Merch`
- 时间控制器 `Time`
- 订单 `Order`

- **Package** `Chef`

  我们的魔幻餐厅中只有一个大厨，所以这里理所应当的使用了单例模式。大厨在整个工作过程中，实际上是对不同的实体（食材，订单）发出了不同的命令，所以这里使用了命令模式。同时，大厨也会对容器里的原料列表进行访问，所以这里使用了访问者模式。

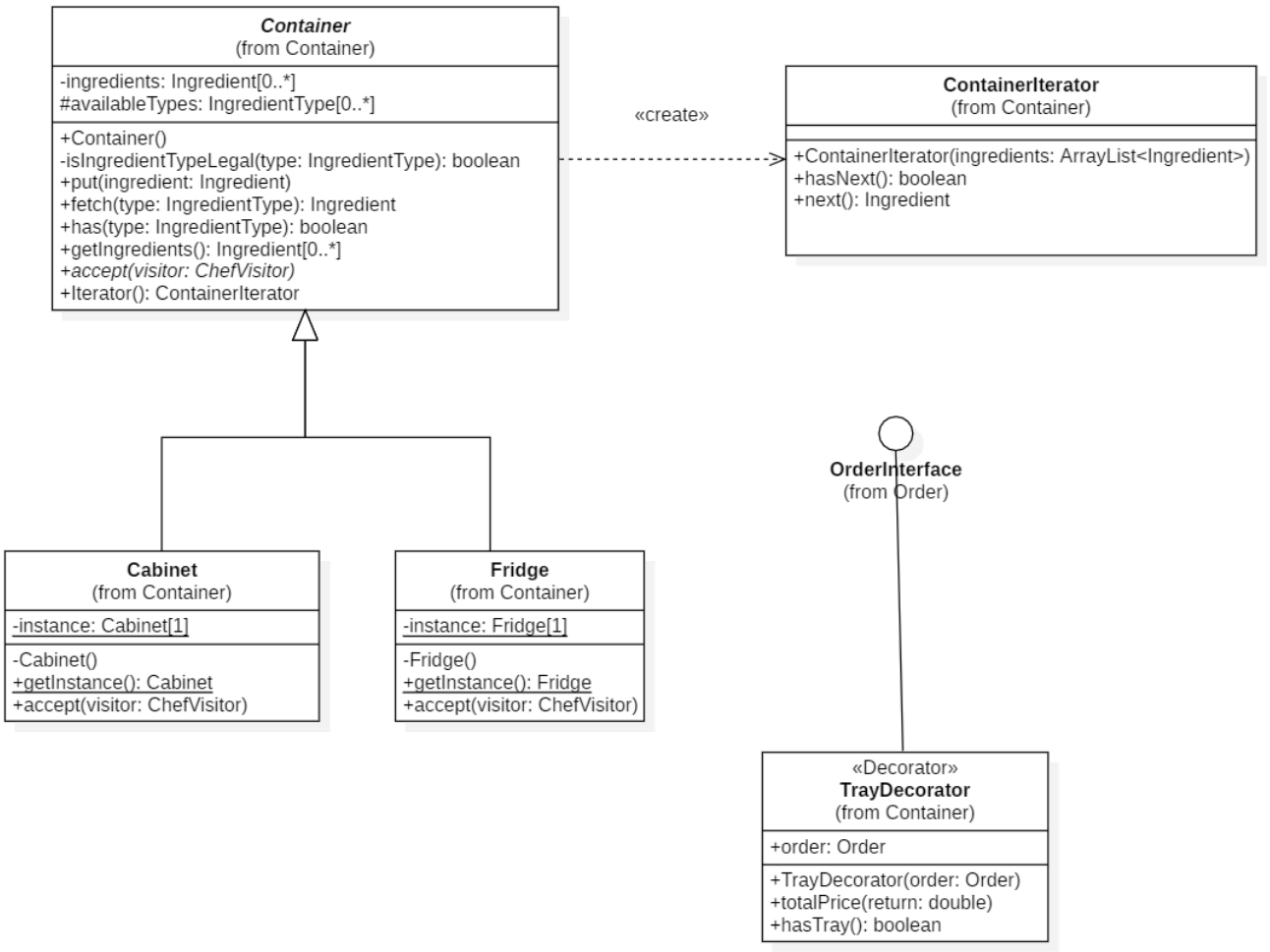


  - 厨师类 `Chef`
  - 厨师命令接口 `ChefCommand`
  - 厨师观察者接口 `ChefVisitor`
  - 厨师观察者实例 `SimpleChefVisitor`

- **Package** `Container`

  我们的厨房中冰箱和橱柜，由于是魔幻厨房嘛，所以他们的容量都是无限的，所以每一样我们只需要一台，这里也用到了单例模式。同时，在订单中的菜品被做好之后，订单

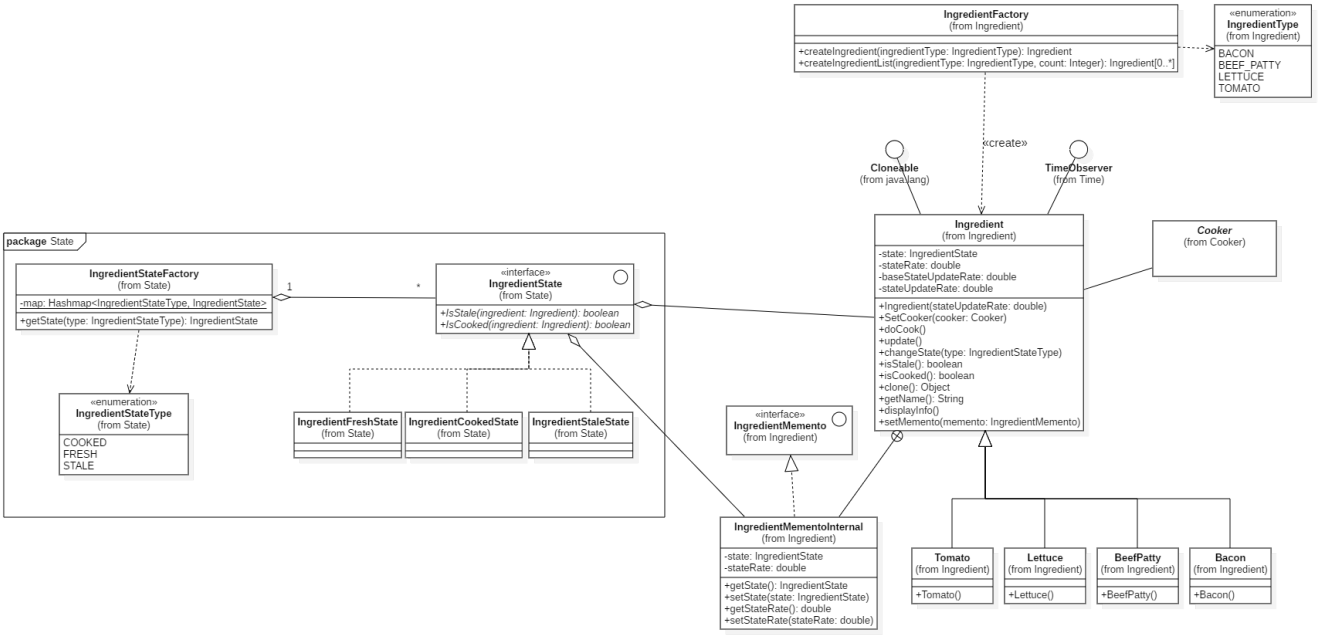
将会被放入托盘中，标志着订单完全完成。这里用到了装饰器模式。



- 容器类 `Container`
- 橱柜类 `Cabinet`
- 冰箱类 `Fridge`
- 托盘装饰器类 `TrayDecorator`

- **Package** `Ingredient`

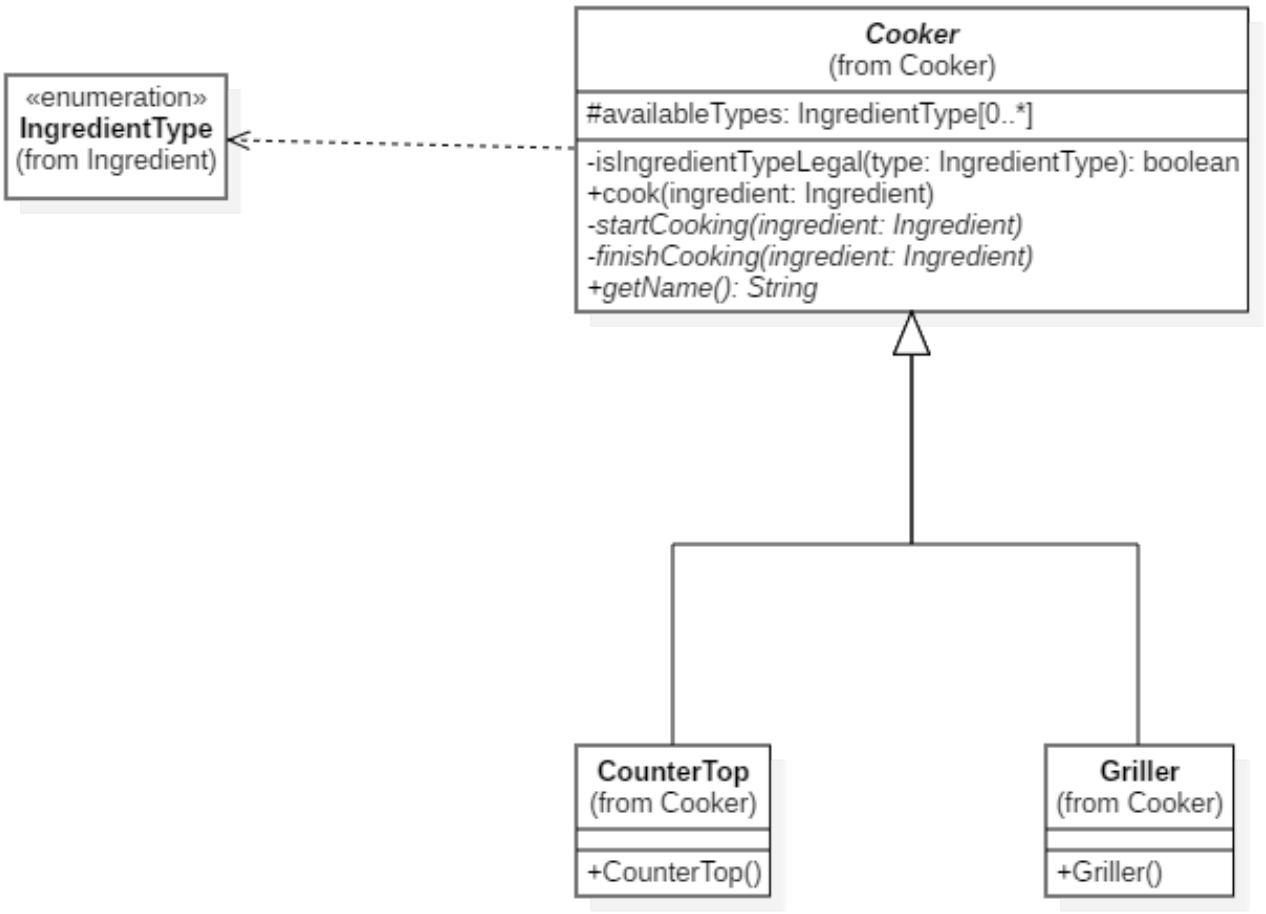
为了制作汉堡，我们当然需要各种各样的食材啦。由于店铺刚开张，我们只提供两种肉类供选择，培根与牛肉，而蔬菜方面则汉堡都标配了生菜和西红柿。创建食材的时候我们使用了工厂模式。我们的魔幻厨房虽然功能完备，但是依然阻止不了食材的过期，所以这里首先通过状态模式为新鲜的以及过期了的食材配备了不同的操作，其次通过观察者模式监测了食材的新鲜程度。此外，在处理食材的过程中，需要制定处理食材的厨具类型，这里应用了策略模式。同时，食材可能在多处被调用，例如它被某个汉堡确定为将要使用的食材之后还可能被厨具拿去处理，这里就要用到享元模式。在食材处理过程中，可能会出现处理失误的情况，比如牛肉烤糊了，此处我们使用了备忘录模式来记录了食材的状态变化，从而可以达成时空逆转，将牛肉复原到最初的状态。当食材快要耗尽的时候，魔幻厨房也可以出发它独有的技能，复制当前材料，从而补充库存，这是原型模式的体现。



- 原料类 `Ingredient`
- 原料工厂类 `IngredientFactory`
- 原料类型枚举类 `IngredientType`
- 番茄类 `Tomato`
- 生菜类 `Lettuce`
- 牛肉饼类 `BeefPatty`
- 培根类 `Bacon`
- 原料状态接口 `IngredientState`
- 原料状态工厂类 `IngredientStateFactory`
- 原料新鲜状态类 `IngredientFreshState`
- 原料已加工状态类 `IngredientCookedState`
- 原料已过期状态类 `IngredientStaleState`
- 原料状态类型枚举类 `IngredientStateType`
- 原料备忘录接口 `IngredientMemento`
- 原料备忘录内部类 `IngredientMementoInternal`

- **Package** `Cooker`

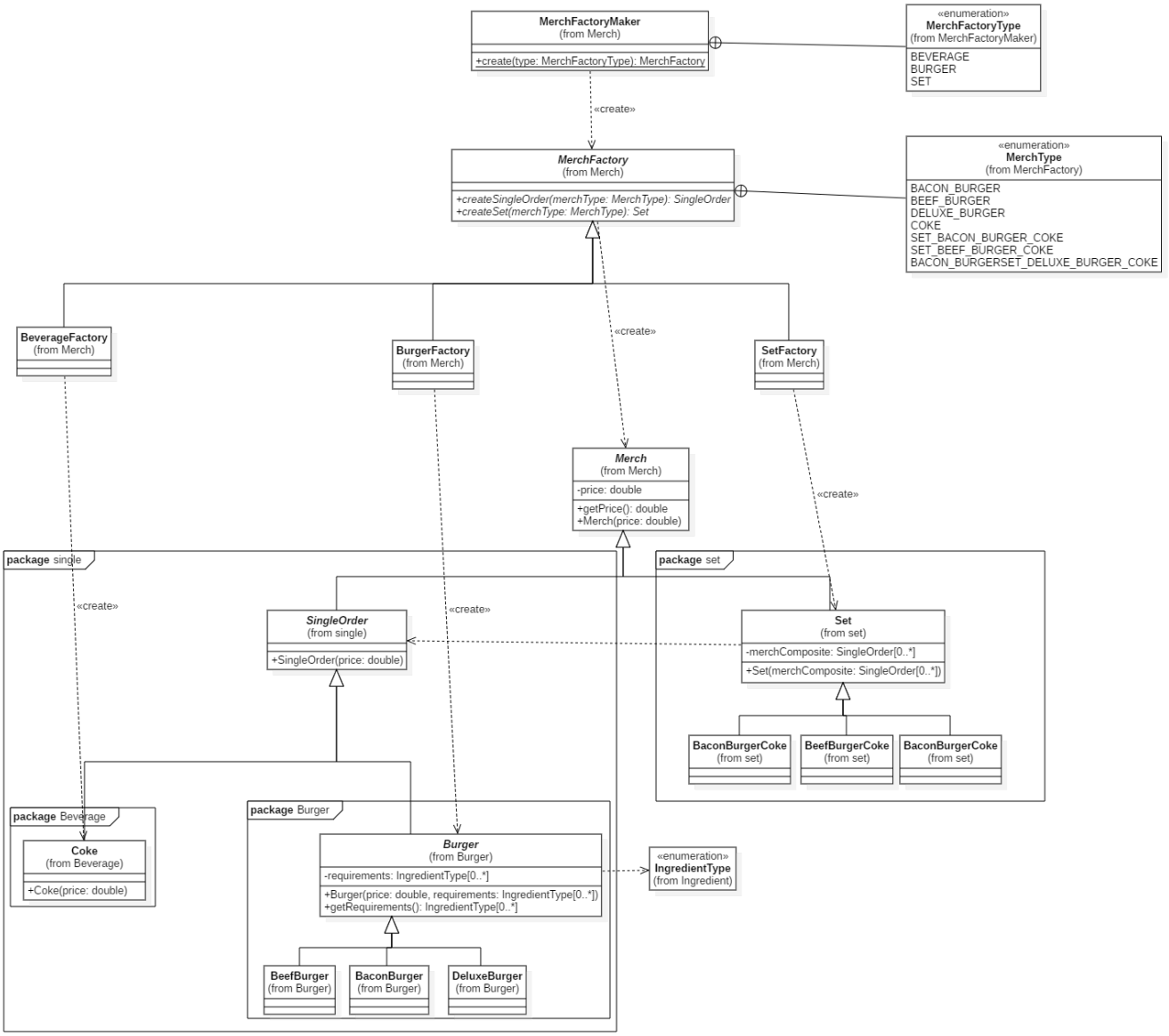
  为了处理不同的食材，我们需要不同的厨具。在奇幻厨房中，处理生菜以及番茄的是操作台，处理培根以及牛肉的则是烤架。这里首先使用模版模式，统一了两种厨具的操作。



  - 厨具类 `Cooker`
  - 橱柜类 `CounterTop`
  - 烤架类 `Griller`

- **Package** `Merch`

  在我们的奇幻餐厅中，我们将所有带有标价可以售卖的实体统称为商品。商品分为三种，汉堡、饮料以及由一杯饮料和一个汉堡所组成的套餐。这里显而易见，使用了组合模式。汉堡与饮料都通过各自的工厂创建，使用了工厂模式。而我们在创造套餐的时候，则是先通过套餐工厂创造出汉堡工厂与饮料工厂，而汉堡工厂又创造出原料工厂，这是抽象工厂模式的体现。而在构建汉堡的过程中，汉堡与饮料的制作顺序不影响最终产出，构建汉堡的过程中，对不同食材的处理顺序也不影响最终产出，这就是建造者模式的体现。此外，在创建商品的时候可能出现创建失败的情况，此时系统将生成一个空商品从而保证后续工作能够继续运行，这里使用了空对象模式。

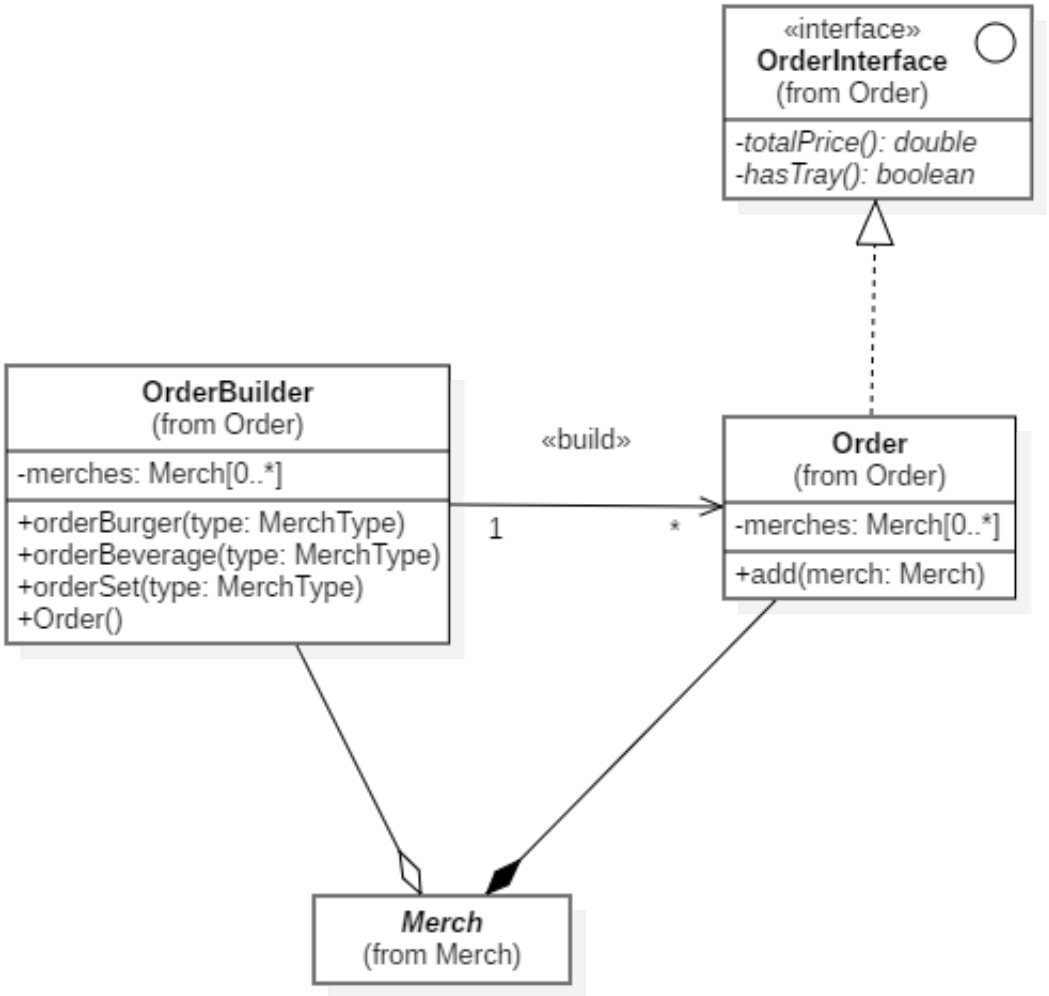


- 商品类 `Merch`
- 商品类型枚举类 `MerchType`
- 单点类 `SingleOrder`
- 汉堡类 `Burger`
- 饮品类 `Beverage`
- 可乐类 `Coke`
- 牛肉汉堡类 `BeefBurger`
- 培根汉堡类 `BaconBurger`
- 双拼豪华汉堡类 `DeluxeBurger`
- 套餐类 `Set`
- 培根汉堡可乐套餐类 `BaconBurgerCoke`
- 牛肉汉堡可乐套餐类 `BeefBurgerCoke`
- 豪华汉堡可乐套餐类 `DeluxeBurgerCoke`
- 商品工厂类 `MerchFactory`
- 商品工厂类型枚举类 `MerchFactoryType`

- 饮品工厂类 `BeverageFactory`
- 汉堡工厂类 `BurgerFactory`
- 套餐工厂类 `SetFactory`

- **Package** `Order`

  订单可以视作商品的列表，它的属性很多也是汇总了商品的属性。比如价格，在计算订单的价格的时候实际上是利用迭代器模式，将其中的商品列表进行遍历，从而算出总价。同时处理订单是一个非常复杂的过程，它涉及到了对订单中的所有商品进行相应的处理，此处我们使用外观模式，将处理订单作为一个外部调用函数，内部的逻辑不暴露给类的使用者。

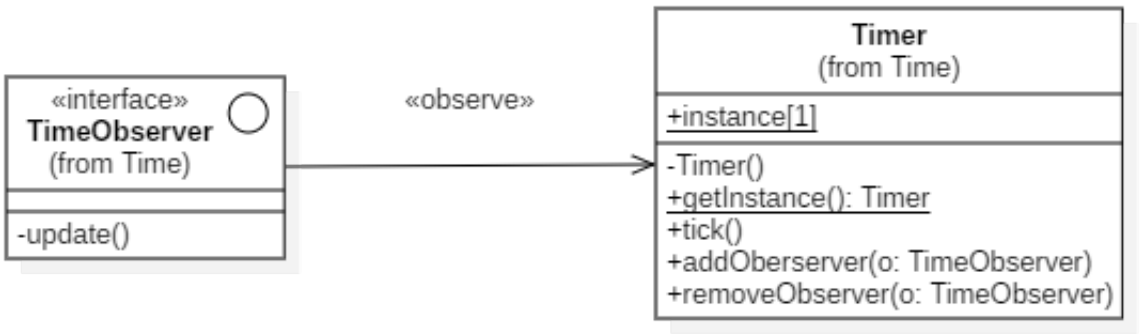


  - 订单类 `Order`
  - 订单接口 `OrderInterface`
  - 订单建造者 `OrderBuilder`

- **Package** `Time`

  由于食材可能会变质，我们需要一个时间处理的模块来监视食材的情况。这里使用了观

察者模式。



- 计时器类 `Timer`
- 时间观察者类 `TimeObserver`

Design Pattern 详述

4.1 单例模式 Singleton

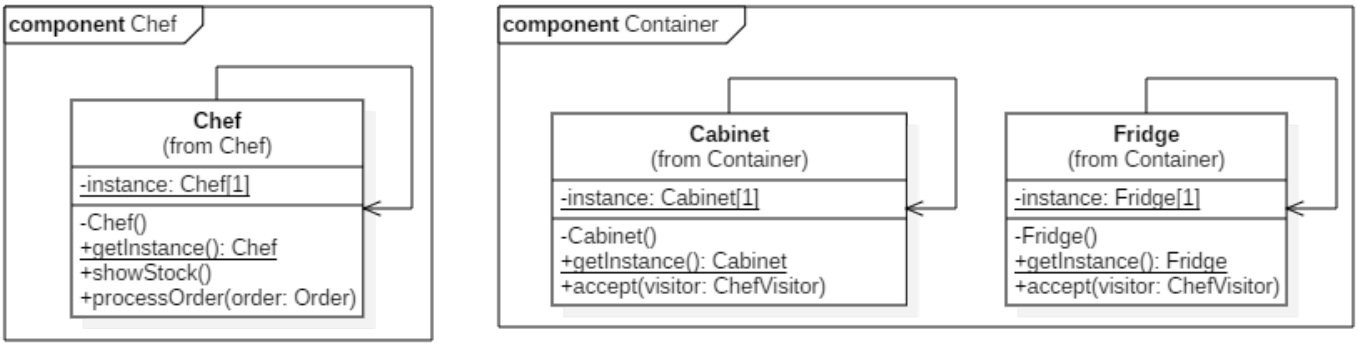
4.1.1 模式简介

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

在欢乐厨房中，有且仅有一个厨师，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。厨房中的容器（包括冰箱与壁橱）也符合这一模式。

4.1.3 Class diagram



4.1.4 API描述

- **Chef 厨师**

  为了保证仅有一个厨师，`Chef` 类具有私有构造函数和本身的一个静态实例，并提供了 `getInstance()` 方法，供外界进行访问。

```
public final class Chef {
    private Chef() {
        if (instance == null) {
            instance = this;
        } else {
            throw new IllegalStateException("Already initialized.");
        }
    }
    private static Chef instance;
    public static synchronized Chef getInstance() {
        if (instance == null) {
            instance = new Chef();
        }
        return instance;
    }
}
```

- **Container 容器**

  `Fridge` 类与 `Cabinet` 类扩展自 `Container` 类，也采用了 Singleton 模式，同样具有自身的私有构造函数与静态实例，提供了 `getInstance()` 方法。

4.2 抽象工厂模式 Abstract Factory

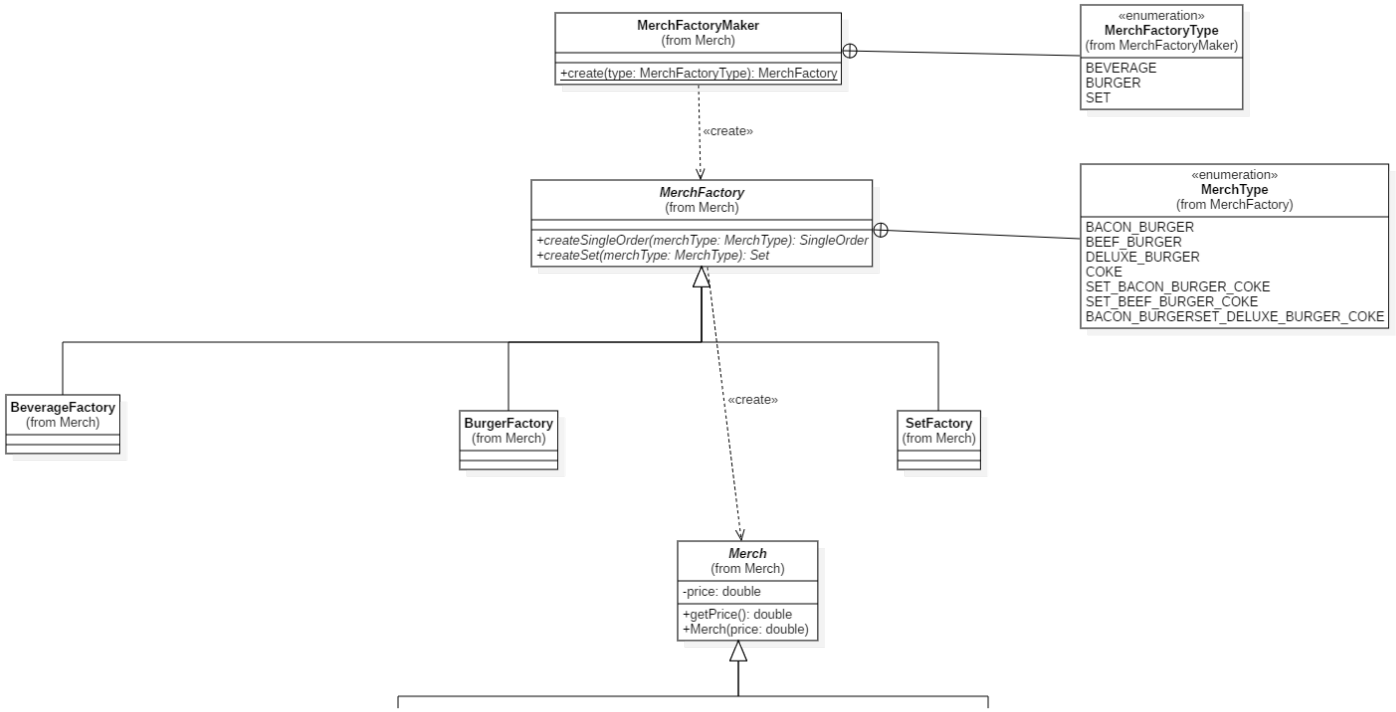
4.2.1 模式简介

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

商品分为单点、套餐两类。创建商品时，通过抽象的商品工厂生成单点（或套餐）的工厂，再通过该工厂生成具体的商品。

4.2.3 Class diagram



4.2.4 API描述

创建了抽象工厂类 `MerchFactory` ，工厂类 `BurgerFactory` 、`BeverageFactory` 等都是继承自 `MerchFactory` 。然后创建了一个工厂创造器类 `MerchFactoryMaker` ，通过传递type信息来获取工厂类型。

```
public class MerchFactoryMaker {
    public enum MerchFactoryType {
        BEVERAGE,
        BURGER,
        SET,
        NULL
    }
```

```
    public static MerchFactory create(MerchFactoryType type) {
        switch (type) {
            case SET:
                return new SetFactory();
            case BURGER:
                return new BurgerFactory();
            case BEVERAGE:
                return new BeverageFactory();
            default:
                return new NullMerchFactory();
        }
    }
}
```

4.3 工厂模式 Factory

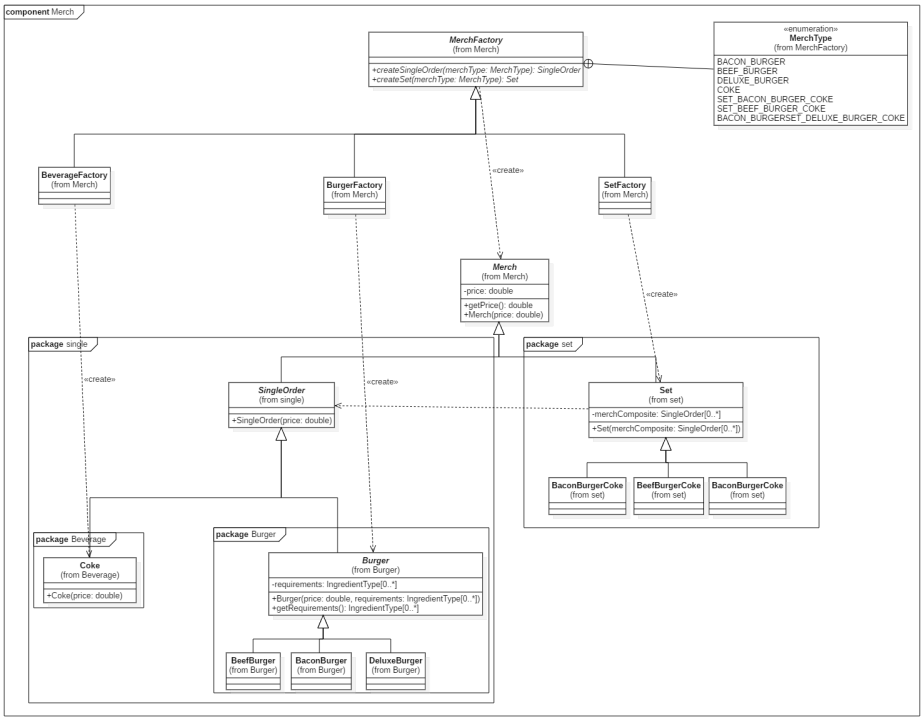
4.3.1 模式简介

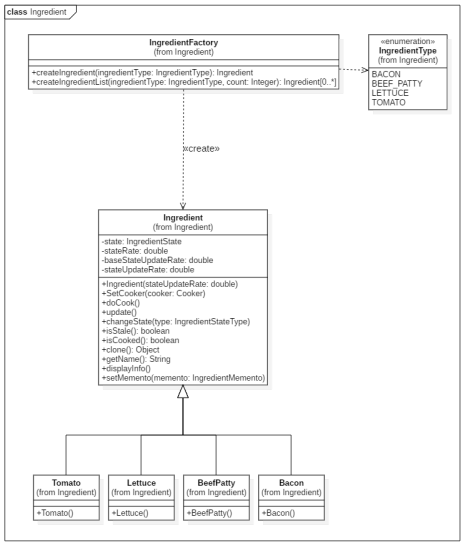
工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

4.3.2 应用场景

生成汉堡、套餐等商品时，根据提供的商品类型生成对象。

4.3.3 Class diagram





4.3.4 API描述

- **汉堡工厂/饮品工厂/套餐工厂**

  创建了工厂类 `BurgerFactory` 、`BeverageFactory` 等，生成基于 `MerchType` 信息的实体类的对象

```
class BurgerFactory extends MerchFactory {
  @Override
  public SingleOrder createSingleOrder(MerchType merchType) {
      switch (merchType) {
          case BEEF_BURGER:
              return new BeefBurger(
                      new ArrayList<>(Arrays.asList(
                              IngredientType.BEEF_PATTY,
                              IngredientType.LETTUCE,
                              IngredientType.TOMATO)),
                      5);
          case BACON_BURGER:
              return new BaconBurger(
                      new ArrayList<>(Arrays.asList(
                              IngredientType.BACON,
                              IngredientType.LETTUCE,
                              IngredientType.TOMATO)),
                      4);
          case DELUXE_BURGER:
              return new DeluxeBurger(
                      new ArrayList<>(Arrays.asList(
                              IngredientType.BEEF_PATTY,
                              IngredientType.BACON,
                              IngredientType.LETTUCE,
                              IngredientType.TOMATO)),
                      8.8);
          default:
              throw new IllegalArgumentException("no such burger in f
actory");
      }
  }
  @Override
  public Set createSet(MerchType merchType) {
      return null;
  }
}
```

- **原料工厂**

  创建了工厂类 `IngredientFactory` ，生成基于 `IngredientType` 信息的实体类的对象。

4.4 模板模式 Template

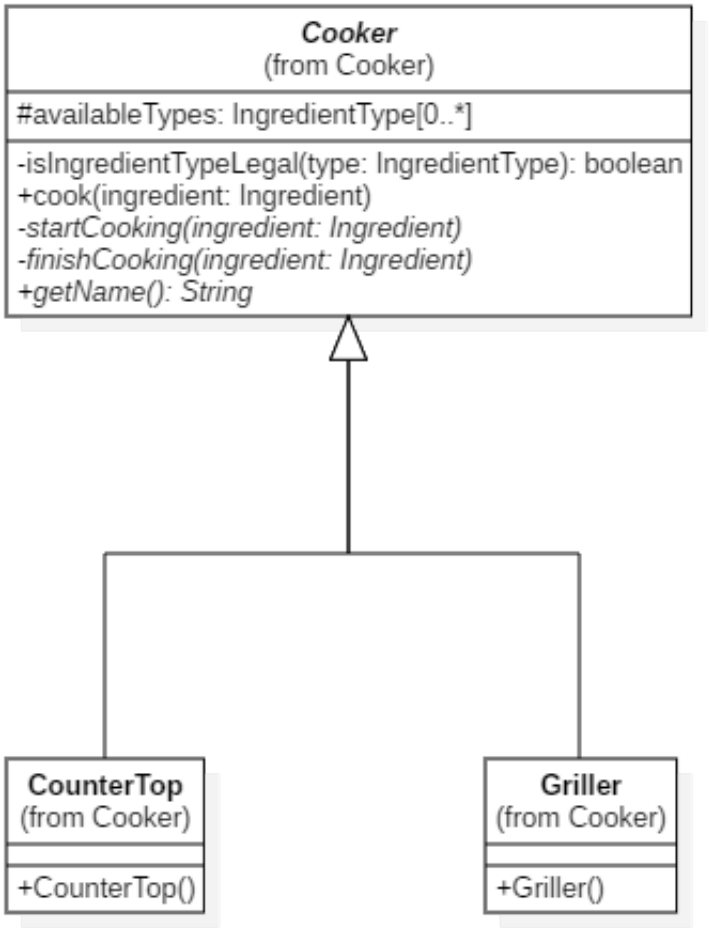
4.4.1 模式简介

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

在烹饪过程中，具体的烹饪方法将根据不同的厨具类型来确定，因此我们采用了模板模式来将制作步骤延迟到子类中。

4.4.3 Class diagram



4.4.4 API描述

`CounterTop` 类和 `Griller` 类是扩展了 `Cooker` 的实体类，它们重写了抽象类的方法 `startCooking()` 和 `finishCooking()` 。

```
public abstract class Cooker {
```

```
    protected ArrayList<IngredientType> availableTypes = new ArrayList<Ingre
dientType>();

    private boolean isIngredientTypeLegal(IngredientType type) {
        return availableTypes.contains(type);
    }

    public final void cook(Ingredient ingredient) {
        if (!(this.isIngredientTypeLegal(ingredient.getIngredientType()))) {
            throw new IllegalArgumentException("this ingredient doesn't belo
ng here");
        }
        startCooking(ingredient);
        finishCooking(ingredient);
    }

    abstract void startCooking(Ingredient ingredient);

    abstract void finishCooking(Ingredient ingredient);

    public abstract String getName();
}
```

4.5 观察者模式 Observer

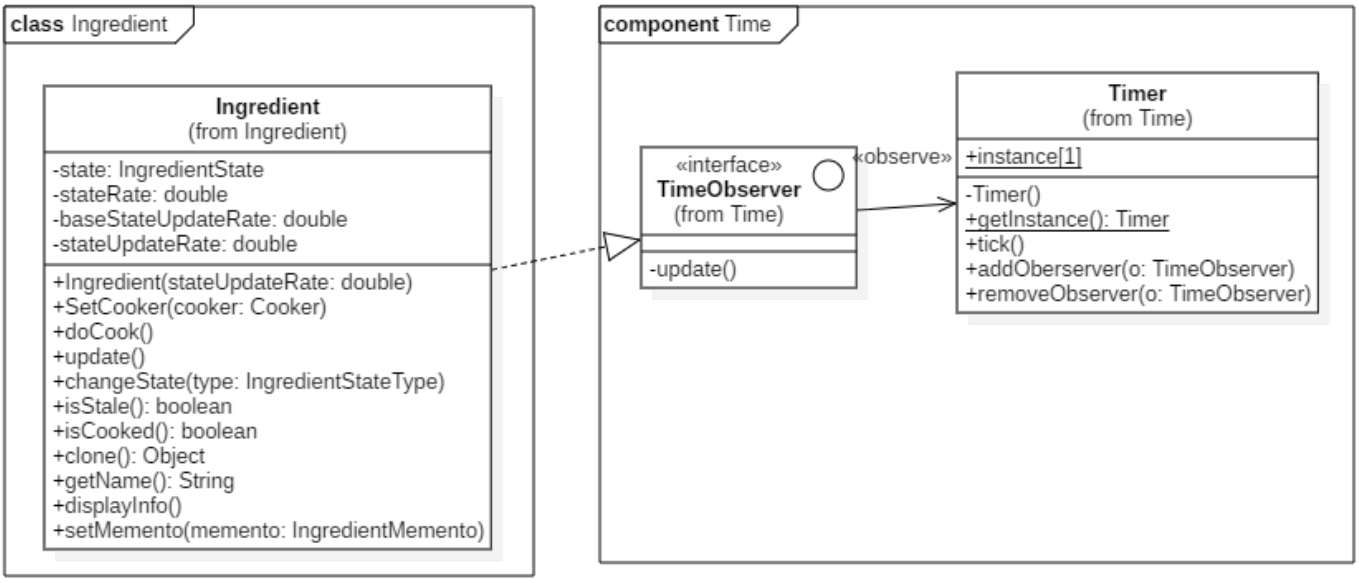
4.5.1 模式简介

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

在欢乐厨房中，当系统时间发生变化时，会被食材观测到，从而影响食材的新鲜程度。

4.5.3 Class diagram



4.5.4 API描述

```
private ArrayList<TimeObserver> observers = new ArrayList<>();

/**
 * 更新内部时间，加快历史的进程
 */

public void tick() {
    for (TimeObserver o : observers) {
        o.update();
    }
}

public void addObserver(TimeObserver o) {
    observers.add(o);
}

public void removeObserver(TimeObserver o) {
    observers.remove(o);
}
```

4.6 迭代器模式 Iterator

4.6.1 模式简介

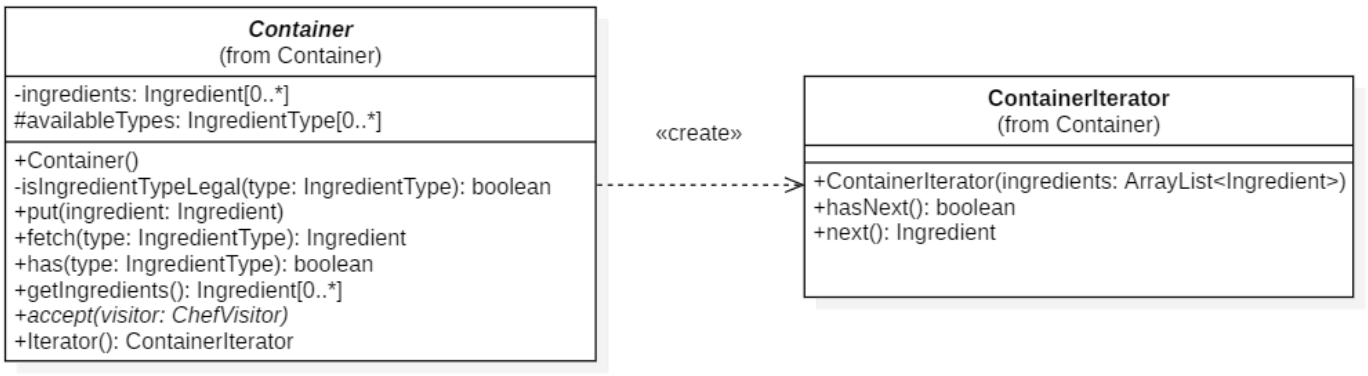
迭代器模式（Iterator Pattern）是 Java 和 .Net 编程环境中非常常用的设计模式。这种模式

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

我们使用了迭代器模式来提供方法遍历容器中的食材。此外，订单总价实际上为订单内部所有汉堡或套餐价格相加的总价，也采用了迭代器模式。

4.6.3 Class diagram



4.6.4 API描述

容器的遍历器，实现了 `java.util.Iterator` 中的方法。

```
public class ContainerIterator implements Iterator<Ingredient> {
    private ArrayDeque<Ingredient> queue = new ArrayDeque<>();

    ContainerIterator(ArrayList<Ingredient> ingredients) {
        for (Ingredient ingredient: ingredients) {
            queue.push(ingredient);
        }
    }

    public boolean hasNext() {
        return !queue.isEmpty();
    }

    public Ingredient next() throws NoSuchElementException {
        if (queue.isEmpty()) {
            throw new NoSuchElementException();
        }
```

```
            return queue.pollFirst();
        }
    }
```

获取遍历器。

```
public ContainerIterator Iterator() {
    return new ContainerIterator(ingredients);
}
```

4.7 状态模式 State

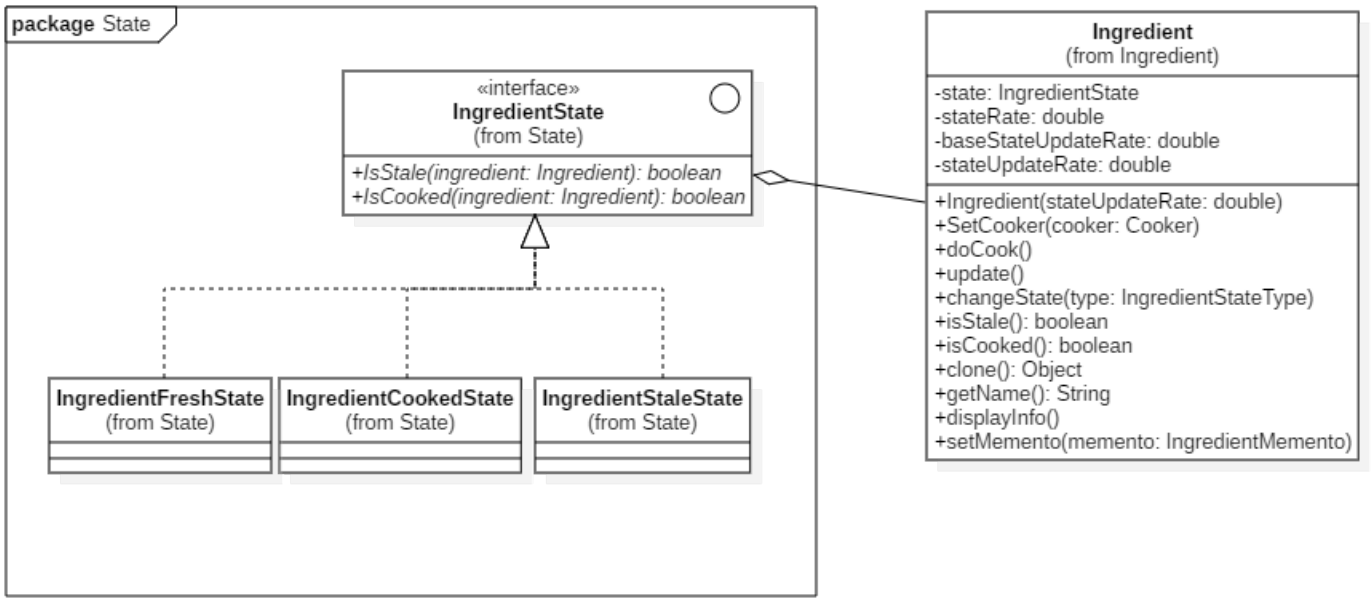
4.7.1 模式简介

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

食材的状态（生、熟、腐烂）会影响它的行为，决定它是否可以被加入菜品。

4.7.3 Class diagram



4.7.4 API描述

创建了IngredientState接口和实现了接口的实体类 `IngredientCookedState` 、`IngredientFreshState` 等。

```
public interface IngredientState {
    public abstract boolean isStale(Ingredient ingredient);

    public abstract boolean isCooked(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

厨师可以对食材和菜品进行操作，包括处理食材、制作菜品等等。

4.8.3 Class diagram

4.8.4 API描述

创建了作为命令的接口 `ChefCommand` ，而实体类 `Chef` 实现了 `ChefCommand` 接口。

```
public interface ChefCommand {
    void showStock();
    void processOrder(Order order);
}
```

4.9 装饰器模式 Decorator

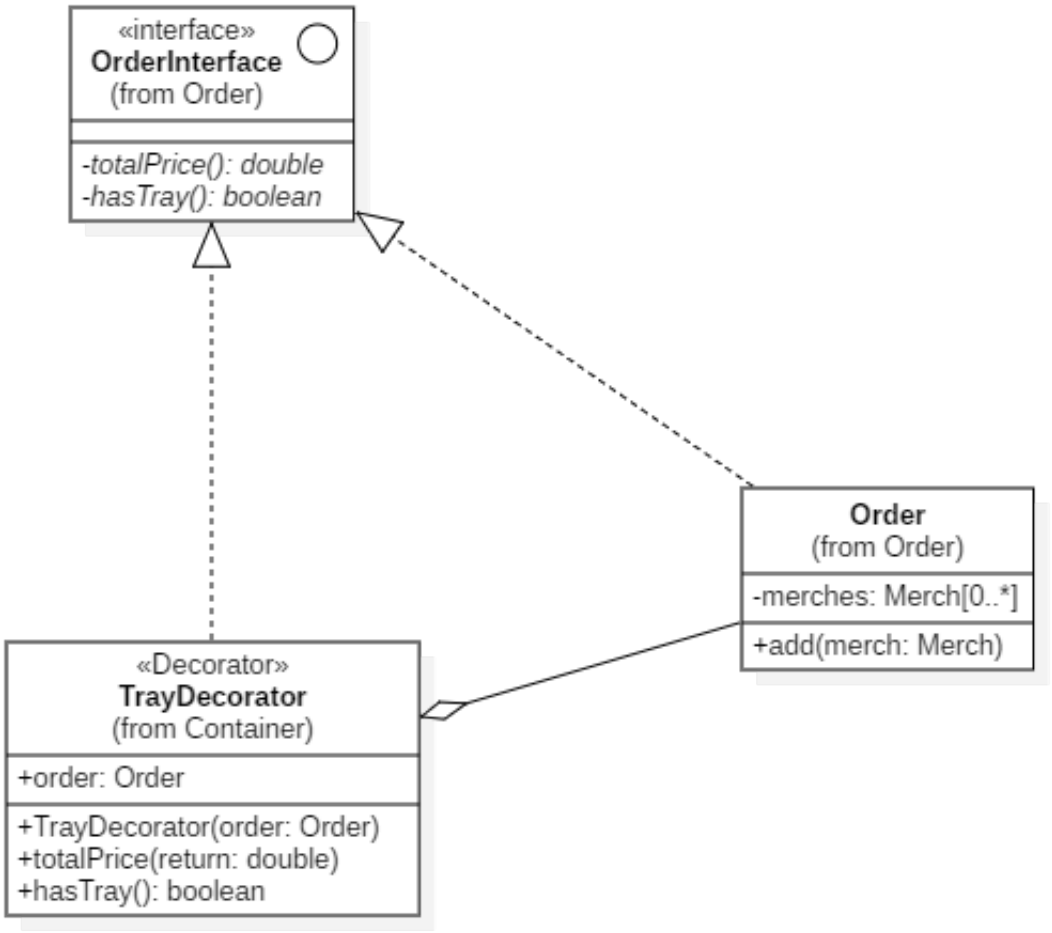
4.9.1 模式简介

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 Class diagram



4.9.4 API描述

这种模式创建了一个装饰类TrayDecorator，用来实现接口 OrderInterface，

```
public class TrayDecorator implements OrderInterface {
    private Order order;

    public TrayDecorator(Order order) {
        this.order = order;
    }

    @Override
    public double totalPrice() {
        return order.totalPrice();
    }
```

```
    @Override
    public boolean hasTray() {
        return true;
    }
}
```

4.10 策略模式 Strategy

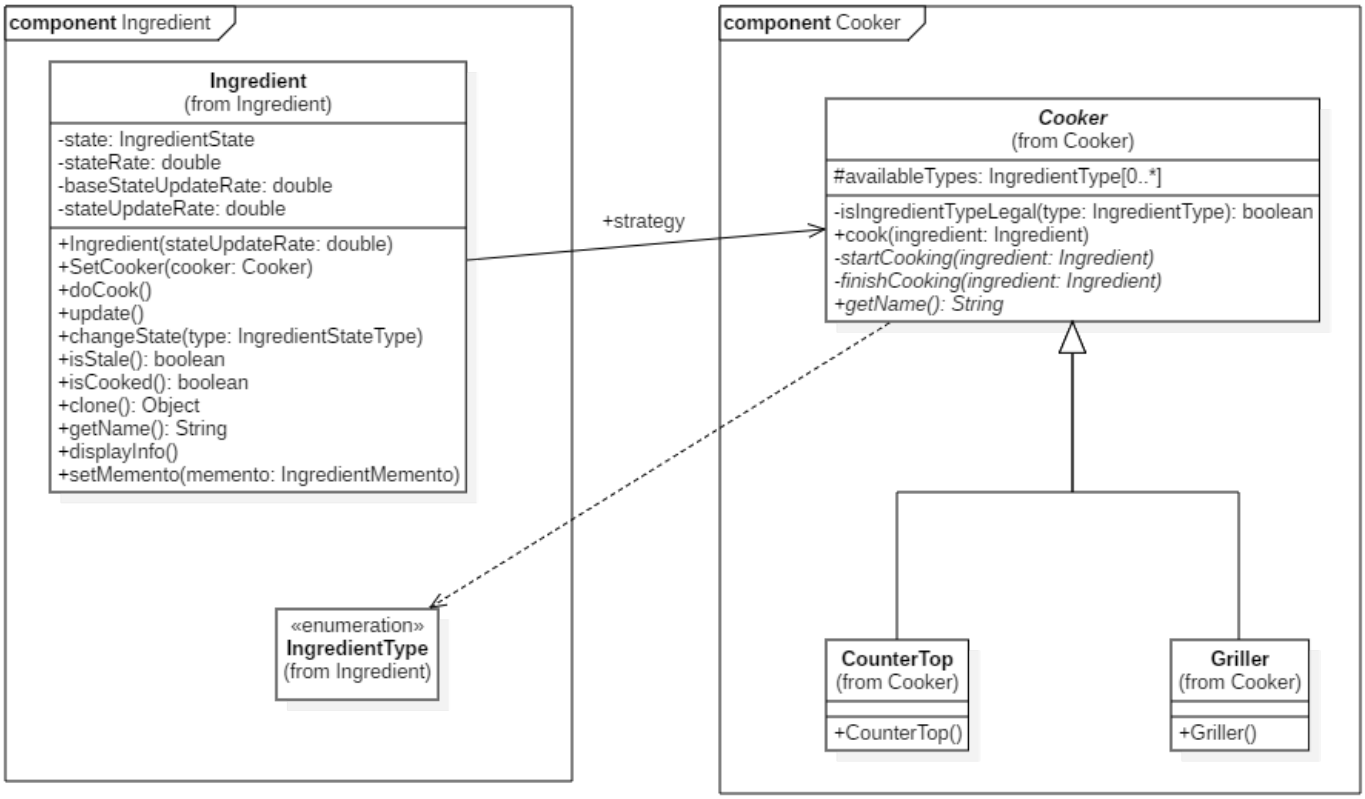
4.10.1 模式简介

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

处理原材料的时候，不同材料需要使用不同的厨具（Strategy），从而烹饪的方法也不同。

4.10.3 Class diagram



4.10.4 API描述

设置处理食材的厨具

```
 protected void setCooker(Cooker cooker) {
        this.cooker = cooker;
}
```

4.11 建造者模式 Builder

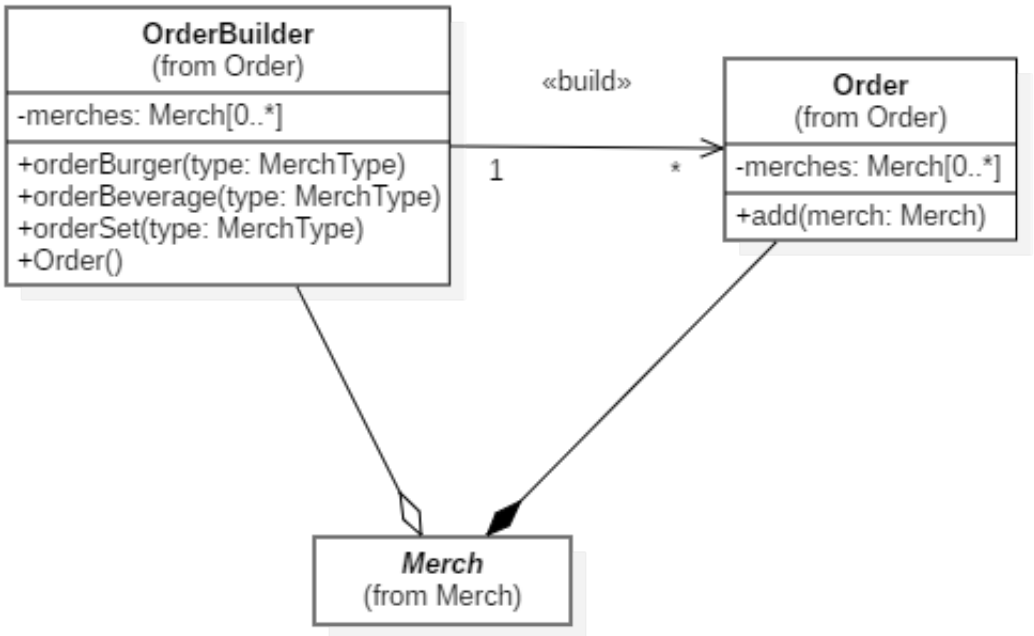
4.11.1 模式简介

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

在创建订单的过程中，汉堡、饮料或套餐的选择顺序对订单的最终产出没有影响。

4.11.3 Class diagram



4.11.4 API描述

创建了 `OrderBuilder` 类，其中包含带有 `Merch` 的 `ArrayList` 以及根据 `MerchType` 创建不同类型的 `Order` 对象。

- `orderBurger(MerchType type)`

  点汉堡

  ```
  protected void setCooker(Cooker cooker) {
        this.cooker = cooker;
  }
  ```

- `orderBeverage(MerchType type)`

  点饮料

  ```
  public void orderBeverage(MerchType type) {
    try {
        MerchFactory factory = MerchFactoryMaker.create(MerchFactoryMaker.
  MerchFactoryType.BEVERAGE);
        merches.add(factory.createSingleOrder(type));
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
  }
  ```

- `orderSet(MerchType type)`

  点套餐

  ```
  public void orderSet(MerchType type) {
    try {
        MerchFactory factory = MerchFactoryMaker.create(MerchFactoryMak
  er.MerchFactoryType.SET);
        merches.add(factory.createSet(type));
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
  }
  ```

- `Order order()`

  下订单

```
public Order order() {
  Order order = new Order();
  for (Merch merch : merches) {
      order.add(merch);
  }
  return order;
}
```

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

当原料盛放在容器中时，容器在检测原料的可食用状态变化时实际上使用的是原料的内部方法；此外在订单部分，我们在计算订单总价的时候也使用了订单内部食物内部的抽象方法来获取食物的单价。

4.12.3 Class diagram

4.12.4 API描述

OrderInterface接口作为桥接实现，订单价格根据内部商品计算。

```
public interface OrderInterface {

    double totalPrice();

    boolean hasTray();

    void displayMerches();

}

public double totalPrice() {
        double price = 0;
```

```
        for (Merch merch : merches) {
            price += merch.getPrice();
        }
        return price;
}
```

4.13 外观模式 Facade

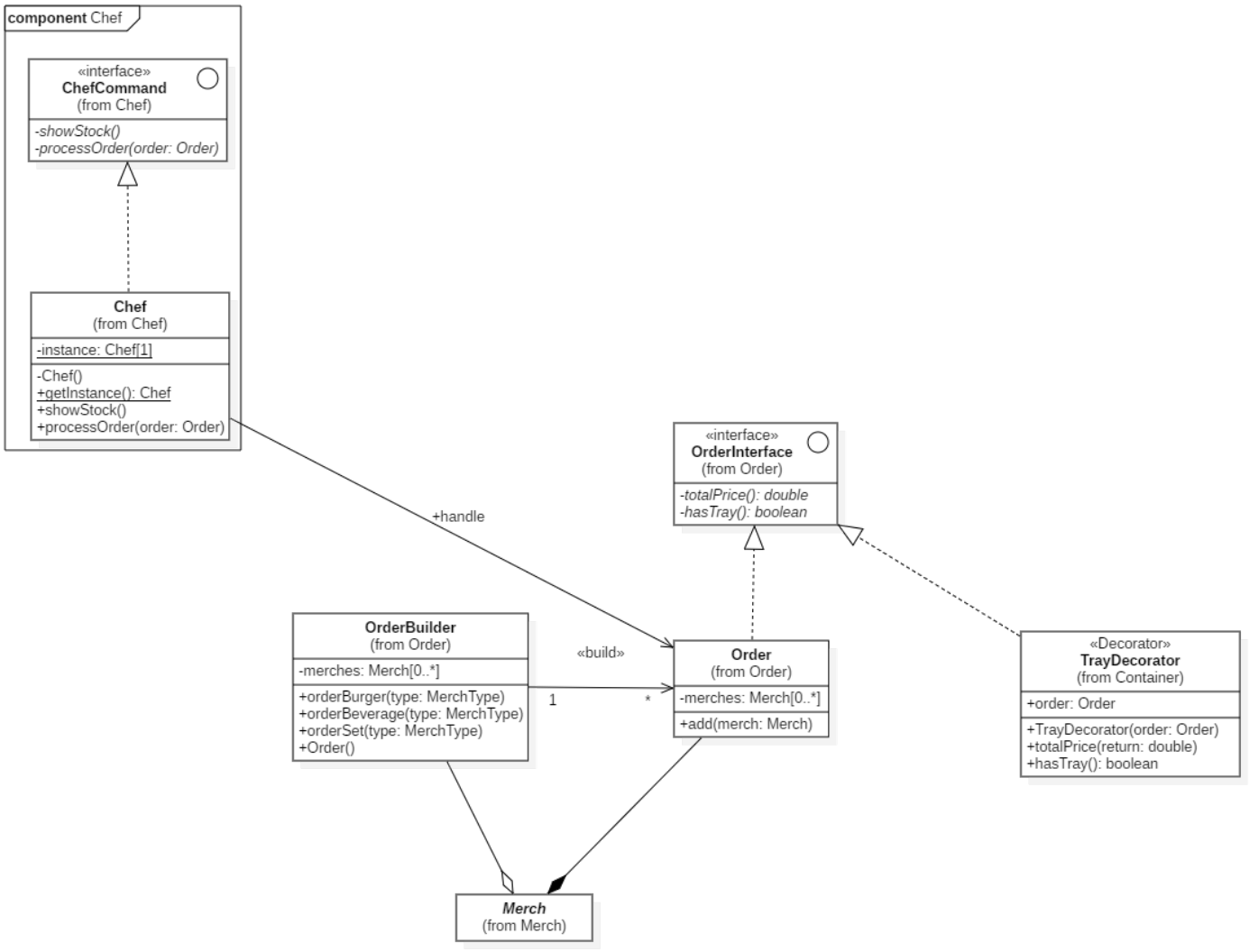
4.13.1 模式简介

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

汉堡的制作是一个极为复杂的过程，我们这里想对顾客隐藏后厨的制作细节，因此我们在这里使用“制作”这个接口来隐藏具体实现。

4.13.3 Class diagram



4.13.4 API描述

对外只提供处理订单接口，内部实现处理的逻辑是取食材、加工、整合。

```
public void processOrder(Order order) {
    System.out.println("大厨，你刚刚接到了新订单！");
    order.displayMerches();
    order.handle();
    TrayDecorator decorator = new TrayDecorator(order);
    decorator.displayMerches();
    System.out.println("\n订单完成啦！");
}
```

4.14 享元模式 Flyweight

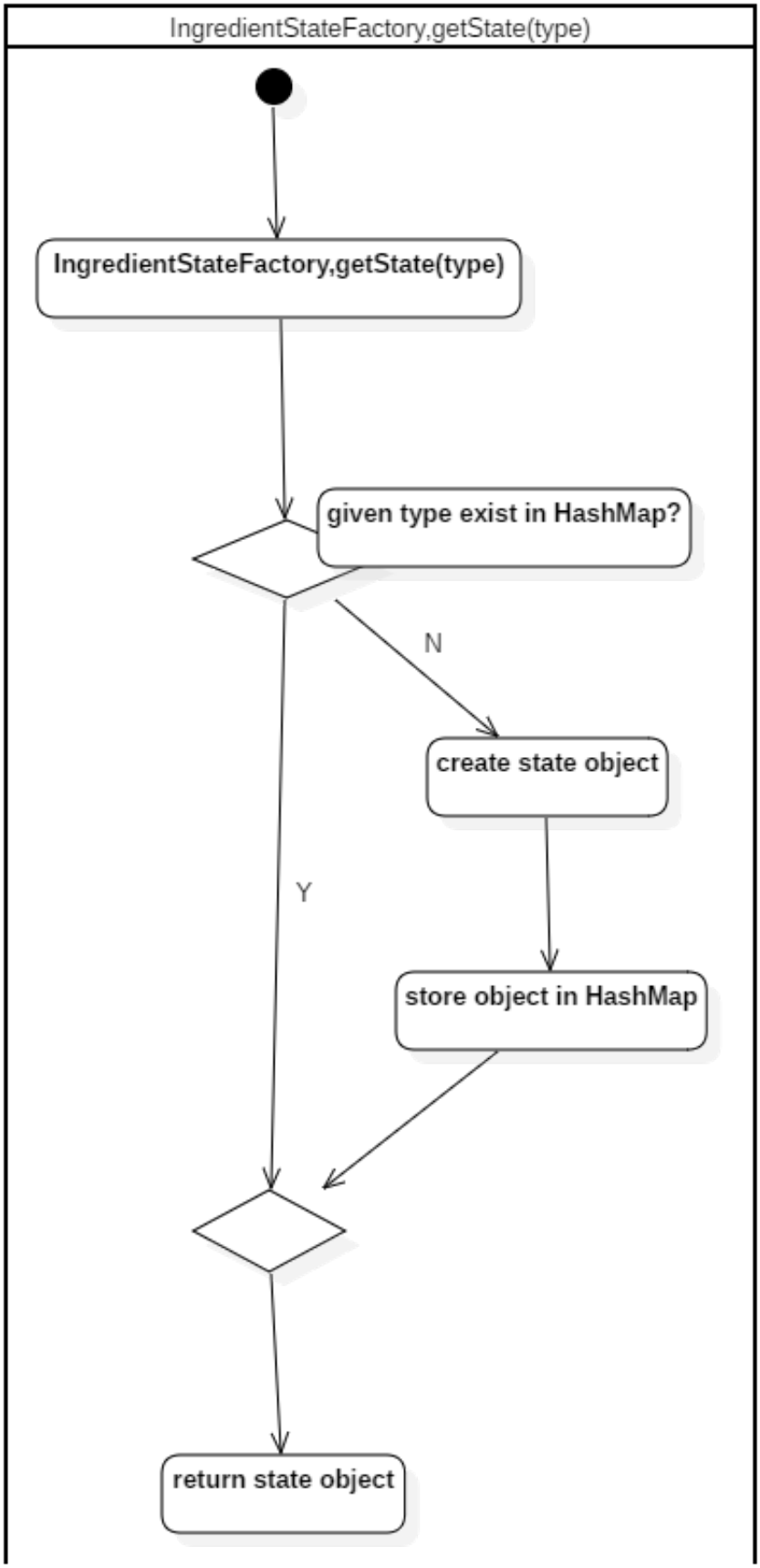
4.14.1 模式简介

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

欢乐厨房里食材的状态对象（State）只需要创建一次，以后会被多处共享同一个对象，因此我们在这里使用享元模式来实现共享。

4.14.3 Class diagram



4.14.4 API描述

定义了工厂类 `IngredientStateFactory` 来获取 `IngredientState` 对象，向

`IngredientStateFactory` 传递信息以便获取它所需状态对象。

```
public class IngredientStateFactory {
    private static HashMap<IngredientStateType, IngredientState> map = new H
ashMap<>();

    public static IngredientState getState(IngredientStateType type) {
        IngredientState state = map.get(type);
        if (state == null) {
            switch (type) {
                case FRESH:
                    state = new IngredientFreshState();
                    map.put(type, state);
                    break;
                case STALE:
                    state = new IngredientStaleState();
                    map.put(type, state);
                    break;
                case COOKED:
                    state = new IngredientCookedState();
                    map.put(type, state);
                    break;
            }
        }
        return state;
    }
}
```

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

当欢乐厨房中的原料用尽时，厨师可以补充原料，因此为了简化模型复杂度我们这里使用原型模式对原有原料进行克隆。

4.15.3 Class diagram

4.15.4 API描述

```
public Object clone() throws CloneNotSupportedException {
    Object clone = null;
    try {
        clone = super.clone();
        Timer.getInstance().addObserver((TimeObserver) clone);
    } catch (CloneNotSupportedException e) {
        System.out.println(e.getMessage());
    }
    return clone;
}
```

4.16 备忘录模式 Memento

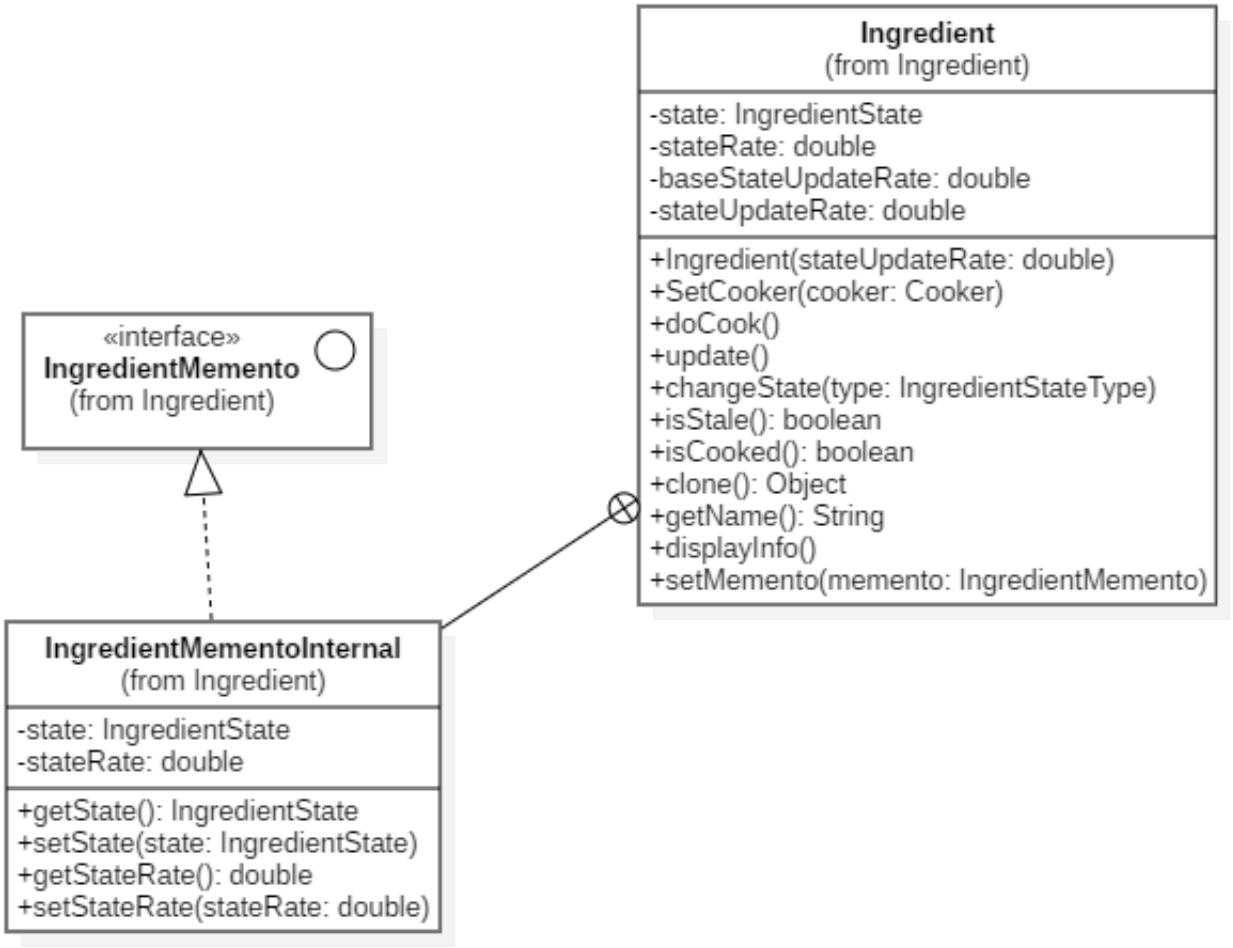
4.16.1 模式简介

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

食材处理过程可能出现意外，需要备忘录机制来进行处理。食材的状态和新鲜程度保存在备忘录中，处理食材之前可以先保存当前状态。

4.16.3 Class diagram



4.16.4 API描述

用于保存食材的状态和新鲜程度。

```
public static class IngredientMementoInternal implements IngredientMemento {

    private IngredientState state;

    private double stateRate;

    public IngredientState getState() {
        return state;
    }

    public void setState(IngredientState state) {
        this.state = state;
    }

    public double getStateRate() {
        return stateRate;
```

```
    }

    public void setStateRate(double stateRate) {
        this.stateRate = stateRate;
    }

}
```

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

在创建订单时，有可能出现指定的商品不存在，直接返回一个空商品（价格为0）。

4.17.3 Class diagram

4.17.4 API描述

创建订单时，可能出现单品或套餐为空的情况，这时直接返回一个空的单品订单或套餐订单。

```
public class NullMerchFactory extends MerchFactory {
    @Override
    public SingleOrder createSingleOrder(MerchType merchType) {
        return new NullSingleOrder();
    }

    @Override
    public Set createSet(MerchType merchType) {
        return new NullSet();
    }
}
```

4.18 组合模式 Composite

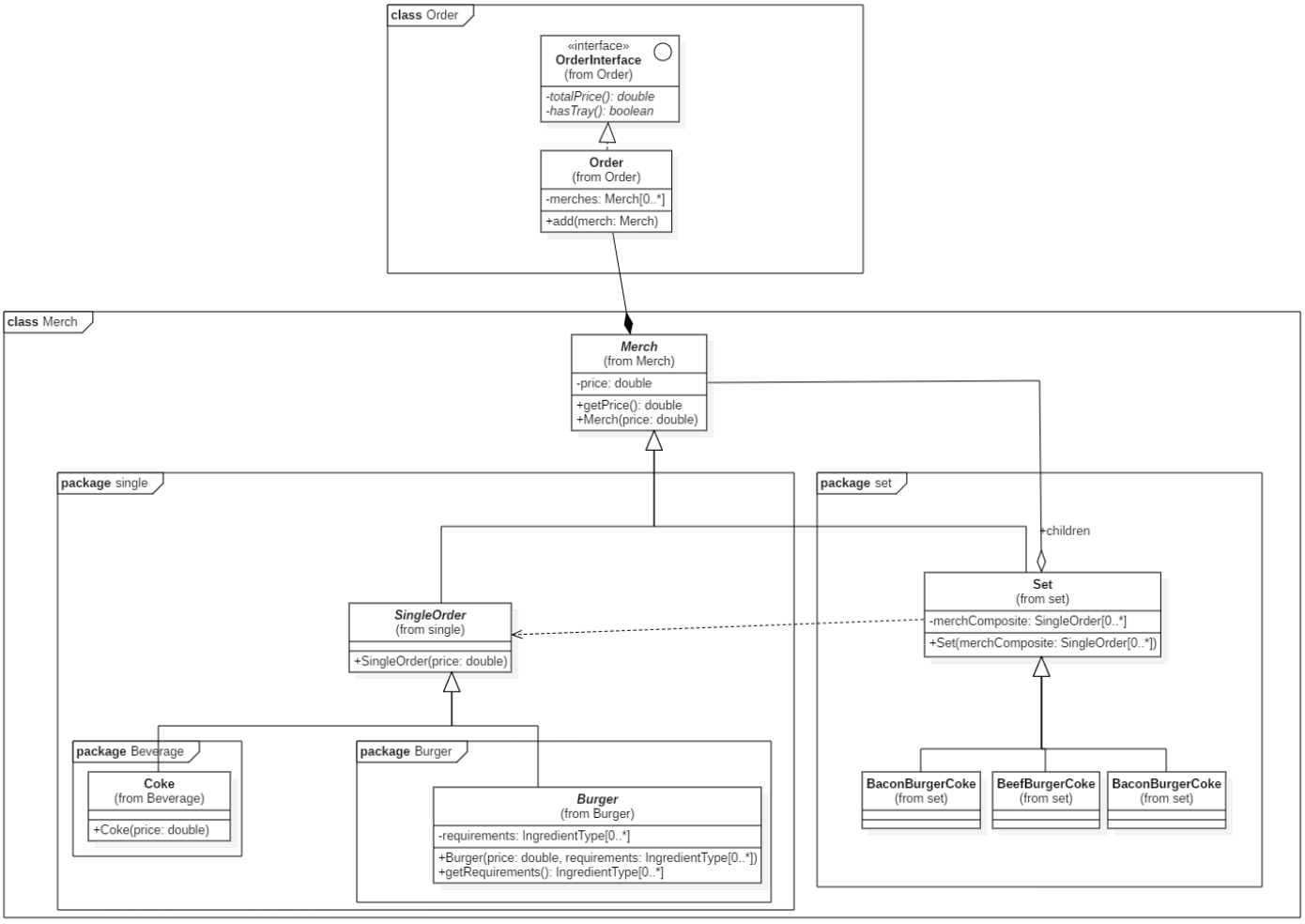
4.18.1 模式简介

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

订单中既可以包含食品、又可以包含食品组成的套餐。

4.18.3 Class diagram



4.18.4 API描述

- `add(Merch merch)`

  订单的行为，向订单中增加商品，商品类型是 `Merch` ， `Merch` 又有 `Coke` 和 `Burger` 两个子类。

```
public void add(Merch merch) {
      merches.add(merch);
}
```

4.19 访问者模式 Visitor

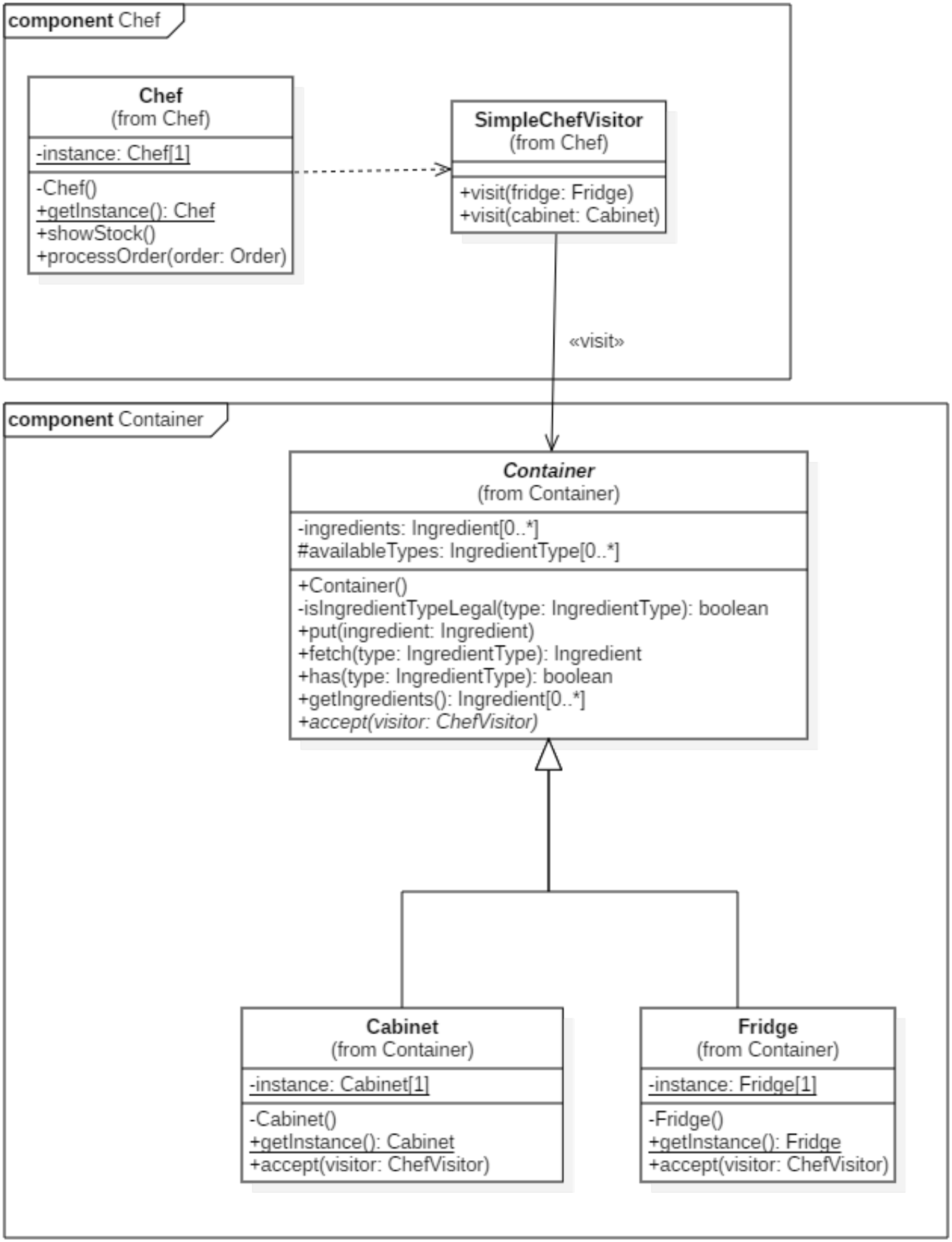
4.19.1 模式简介

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

厨师查看冰箱、橱柜的东西时，不需要在冰箱、橱柜中设计遍历的方法，可以设计一个外部的厨师的 `visitor` 类来访问。

4.19.3 Class diagram



4.19.4 API描述

- `visit(Container container)`

访问冰箱或是橱柜中的食材

```
public void visit(Container container) {
  System.out.println(container.getName() + "里装有: ");
  for (Ingredient ingredient : container.getIngredients()) {
      ingredient.displayInfo();
  }
  System.out.println();
}
```