实验三 抽象工厂模式的应用

1 实验目的

1) 掌握抽象工厂模式（Abstract Factory）的特点

2) 分析具体问题，使用抽象工厂模式进行设计。

2 实验内容和要求

麦当劳（McDonalds）和肯德基（KFC）快餐店都经营汉堡（Hamburg）和可乐（Cola），用C#或Java控制台应用程序实现这两个快餐店经营产品的抽象工厂模式。绘制该模式的UML图。

3 UML图

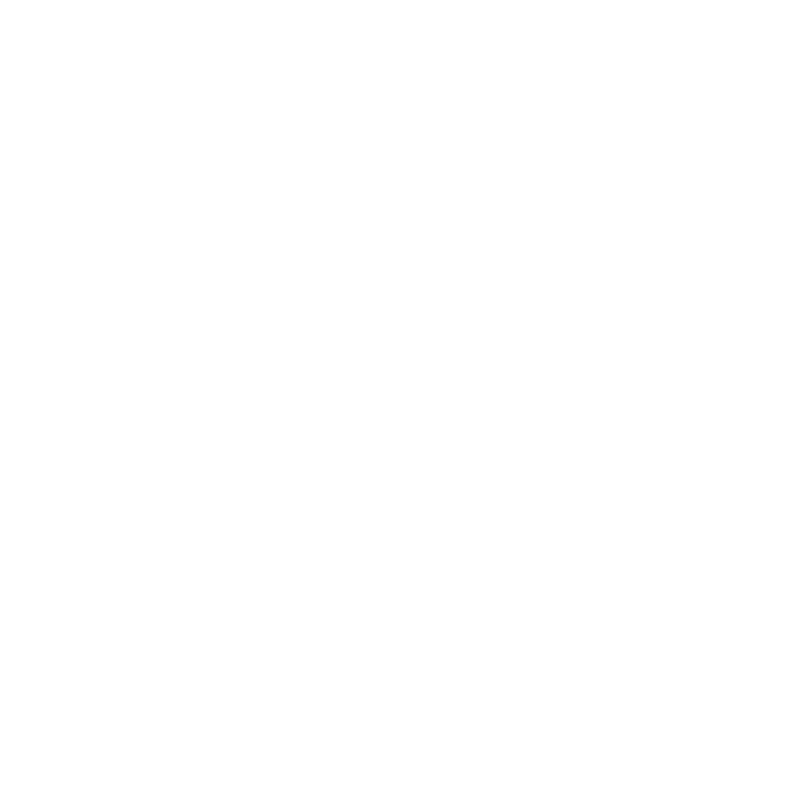


图3.3.1 UML类图

4 代码实现

interface Cola {  
 public void bubble();  
}

public class KFCCola implements Cola{  
 @Override  
 public void bubble() {  
 System.*out*.println("KFCCola Bubble");  
 }  
}

public class McDonaldsCola implements Cola{  
 @Override  
 public void bubble() {  
 System.*out*.println("McDonaldsCola Bubble");  
 }  
}

interface Hamburg {  
 public void steam();  
}

public class KFCHamburg implements Hamburg{  
 @Override  
 public void steam() {  
 System.*out*.println("KFCHamburg Steam");  
 }  
}

public class McDonaldsHamburg implements Hamburg{  
 @Override  
 public void steam() {  
 System.*out*.println("McDonaldsHamburg Steam");  
 }  
}

interface Factory {  
 public Cola createCola();  
 public Hamburg createHamburg();  
}

public class KFCFactory implements Factory{  
 @Override  
 public Cola createCola() {  
 System.*out*.println("KFC creates KFC Cola");  
 return new KFCCola();  
 }  
  
 @Override  
 public Hamburg createHamburg() {  
 System.*out*.println("KFC creates KFC Hamburg");  
 return new KFCHamburg();  
 }  
}

public class McDonaldsFactory implements Factory{  
 @Override  
 public Cola createCola() {  
 System.*out*.println("McDonalds creates McDonalds Cola");  
 return new McDonaldsCola();  
 }  
  
 @Override  
 public Hamburg createHamburg() {  
 System.*out*.println("McDonalds creates McDonalds Hamburg");  
 return new McDonaldsHamburg();  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 Factory factory=new KFCFactory();  
 Cola cola=factory.createCola();  
 cola.bubble();  
 Hamburg hamburg=factory.createHamburg();  
 hamburg.steam();  
 }  
}

**测试结果**

KFC creates KFC Cola

KFCCola Bubble

KFC creates KFC Hamburg

KFCHamburg Steam

5 要点总结

1. 抽象工厂模式在增加食品种类（披萨）时，不满足开放封闭原则
2. 抽象工厂模式在增加供应商（必胜客）时，则满足开放封闭原则

实验四 建造者模式的应用

1 实验目的

1) 掌握建造者模式（Builder）的特点

2) 分析具体问题，使用建造者模式进行设计。

2 实验内容和要求

建造者模式是一种创建型模式，它主要是应对项目中一些复杂对象的创建工作。所谓“复杂对象”，是指此对象中还含有其它的子对象。我们现在定义一个场景：汽车生产必须包含车轮(Wheel)、油箱(OilBox)和车身()，应用建造者模式，用C#或Java控制台应用程序实现该设计，构建BMW品牌和BenZ品牌汽车生产。绘制该模式的UML图。

3 UML图

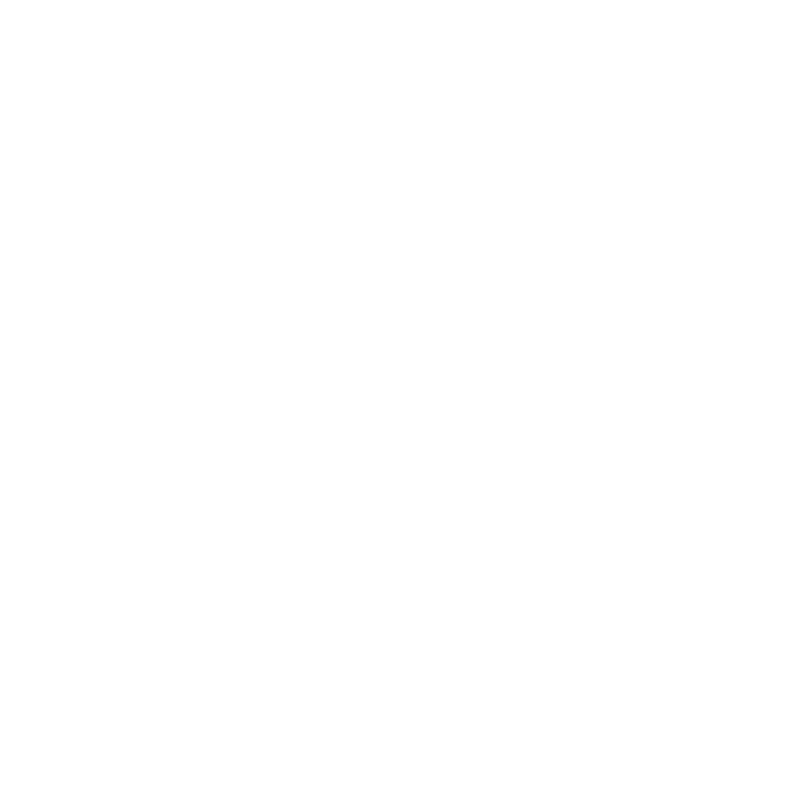


图4.3.1 UML类图

4 代码实现

public class Car {  
 String wheel;  
 String oilBox;  
 String body;  
  
 @Override  
 public String toString() {  
 return "Car{" +  
 "wheel='" + wheel + '\'' +  
 ", oilBox='" + oilBox + '\'' +  
 ", body='" + body + '\'' +  
 '}';  
 }  
  
 public String getWheel() {  
 return wheel;  
 }  
  
 public void setWheel(String wheel) {  
 this.wheel = wheel;  
 }  
  
 public String getOilBox() {  
 return oilBox;  
 }  
  
 public void setOilBox(String oilBox) {  
 this.oilBox = oilBox;  
 }  
  
 public String getBody() {  
 return body;  
 }  
  
 public void setBody(String body) {  
 this.body = body;  
 }  
}

abstract class Builder {  
 protected Car car = new Car();  
  
 public abstract void buildWheel();  
  
 public abstract void buildOilBox();  
  
 public abstract void buildBody();  
  
 public Car getCar() {  
 return car;  
 }  
}

public class BMWBuilder extends Builder {  
  
 @Override  
 public void buildWheel() {  
 car.setWheel("BMW Wheel");  
 }  
  
 @Override  
 public void buildOilBox() {  
 car.setOilBox("BMW OilBox");  
 }  
  
 @Override  
 public void buildBody() {  
 car.setBody("BMW Body");  
 }  
}

public class BenZBuilder extends Builder{  
 @Override  
 public void buildWheel() {  
 car.setWheel("Benz Wheel");  
 }  
  
 @Override  
 public void buildOilBox() {  
 car.setOilBox("Benz OilBox");  
 }  
  
 @Override  
 public void buildBody() {  
 car.setBody("Benz Body");  
 }  
}

public class Director{  
 private Builder builder;  
  
 public void setBuilder(Builder builder) {  
 this.builder = builder;  
 }  
  
 public Car buildCar(){  
 builder.buildBody();  
 builder.buildWheel();  
 builder.buildOilBox();  
 return builder.getCar();  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 Director director=new Director();  
 director.setBuilder(new BenZBuilder());  
 System.*out*.println(director.buildCar());  
 }  
}

**测试结果**

Car{wheel='Benz Wheel', oilBox='Benz OilBox', body='Benz Body'}

5 要点总结

1. 与工厂模式不同的是，建造者模式更加注重构建复杂对象的方法顺序；
2. 指挥者调用具体的建造者，来创建对象的各个部分，在调用者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建；使用创建者模式对于用户而言只需要制定需要建造的类就可以获得对象，建造过程及细节不需要了解。