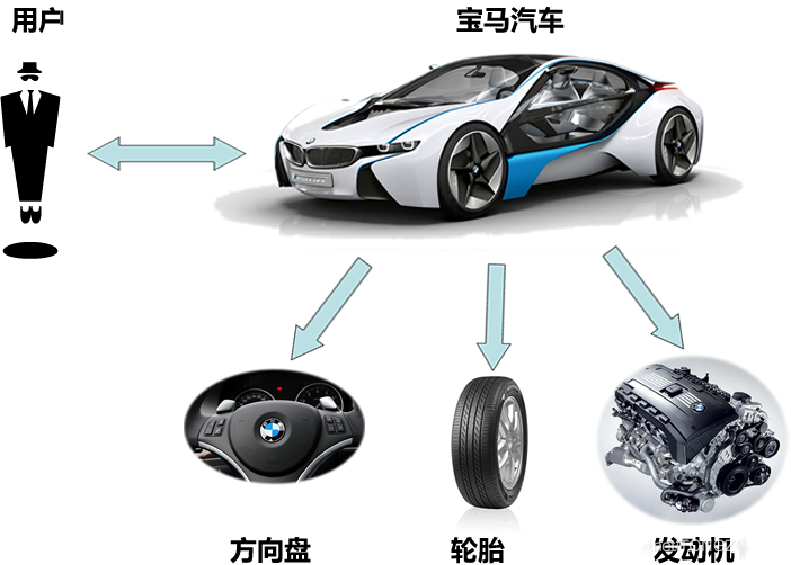
### **建造者模式**

#### **1、模式动机**

无论是在现实世界中还是在软件系统中，都存在一些复杂的对象，它们拥有多个组成部分，如汽车，它包括车轮、方向盘、发送机等各种部件。而对于大多数用户而言，无须知道这些部件的装配细节，也几乎不会使用单独某个部件，而是使用一辆完整的汽车，可以通过建造者模式对其进行设计与描述，建造者模式可以将部件和其组装过程分开，一步一步创建一个复杂的对象。用户只需要指定复杂对象的类型就可以得到该对象，而无须知道其内部的具体构造细节。



·在软件开发中，也存在大量类似汽车一样的复杂对象，它们拥有一系列成员属性，这些成员属性中有些是引用类型的成员对象。而且在这些复杂对象中，还可能存在一些限制条件，如某些属性没有赋值则复杂对象不能作为一个完整的产品使用；有些属性的赋值必须按照某个顺序，一个属性没有赋值之前，另一个属性可能无法赋值等。

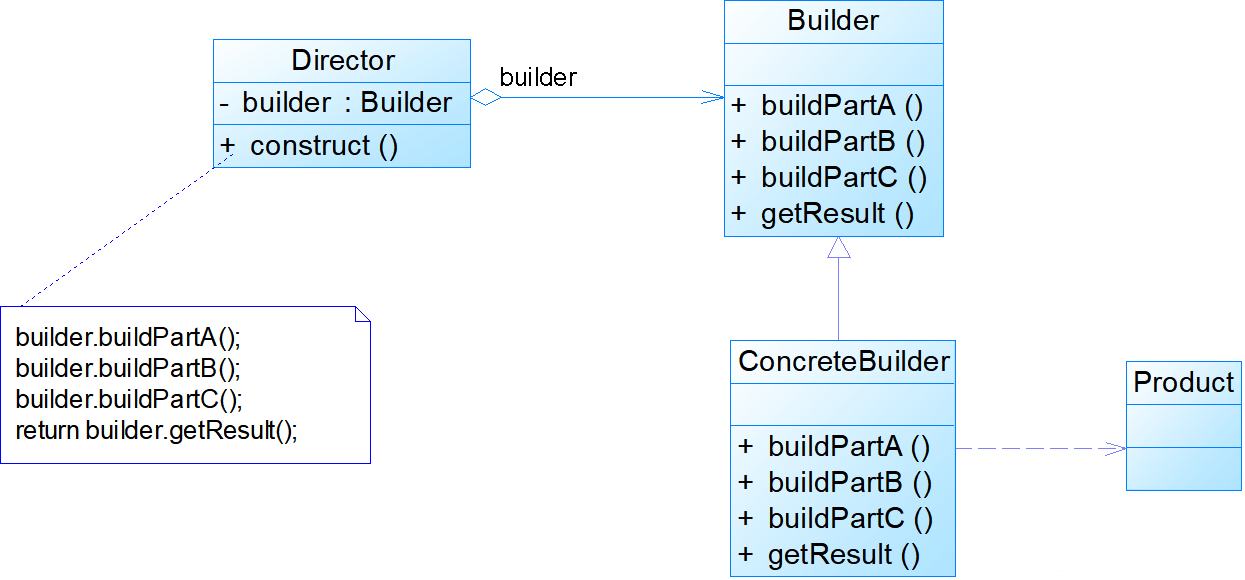
复杂对象相当于一辆有待建造的汽车，而对象的属性相当于汽车的部件，建造产品的过程就相当于组合部件的过程。由于组合部件的过程很复杂，因此，这些部件的组合过程往往被“外部化”到一个称作建造者的对象里，建造者返还给客户端的是一个已经建造完毕的完整产品对象，而用户无须关心该对象所包含的属性以及它们的组装方式，这就是建造者模式的模式动机。

#### **2、模式定义**

·建造者模式(Builder Pattern)：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

·建造者模式是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建它们，用户不需要知道内部的具体构建细节。建造者模式属于对象创建型模式。根据中文翻译的不同，建造者模式又可以称为生成器模式。

#### **3、模式结构**



##### **4、模式角色**

Builder：抽象建造者

ConcreteBuilder：具体建造者

Director：指挥者

Product：产品角色

package 建造者模式;

//产品角色

public class Pruduct {

private String partA;

private String partB;

private String partC;

public String getPartA() {

return partA;

}

public void setPartA(String partA) {

this.partA = partA;

}

public String getPartB() {

return partB;

}

public void setPartB(String partB) {

this.partB = partB;

}

public String getPartC() {

return partC;

}

public void setPartC(String partC) {

this.partC = partC;

}

}

package 建造者模式;

// 抽象建造者

public abstract class Builder {

protected Pruduct pruduct = new Pruduct();

public abstract void buildPartA();

public abstract void buildPartB();

public abstract void buildPartC();

public Pruduct getResult(){

return pruduct;

}

}

package 建造者模式;

//具体建造者

public class ConcreteBuilder extends Builder {

@Override

public void buildPartA() {

System.out.println("A");

}

@Override

public void buildPartB() {

System.out.println("B");

}

@Override

public void buildPartC() {

System.out.println("C");

}

}

* 指挥者类代码：

package 建造者模式;

//指挥者

public class Driector {

private Builder builder;

public Driector(Builder builder){

this.builder = builder;

}

public void setBuilder(Builder builder) {

this.builder = builder;

}

public Pruduct construct(){

builder.buildPartA();

builder.buildPartB();

builder.buildPartC();

return builder.getResult();

}

}

* 客户端代码

package 建造者模式;

public class Cilent {

public static void main(String[] args) {

//确定具体建造者的类型ConcreteBuilder即可

Builder builder = new ConcreteBuilder();

Driector driector = new Driector(builder);

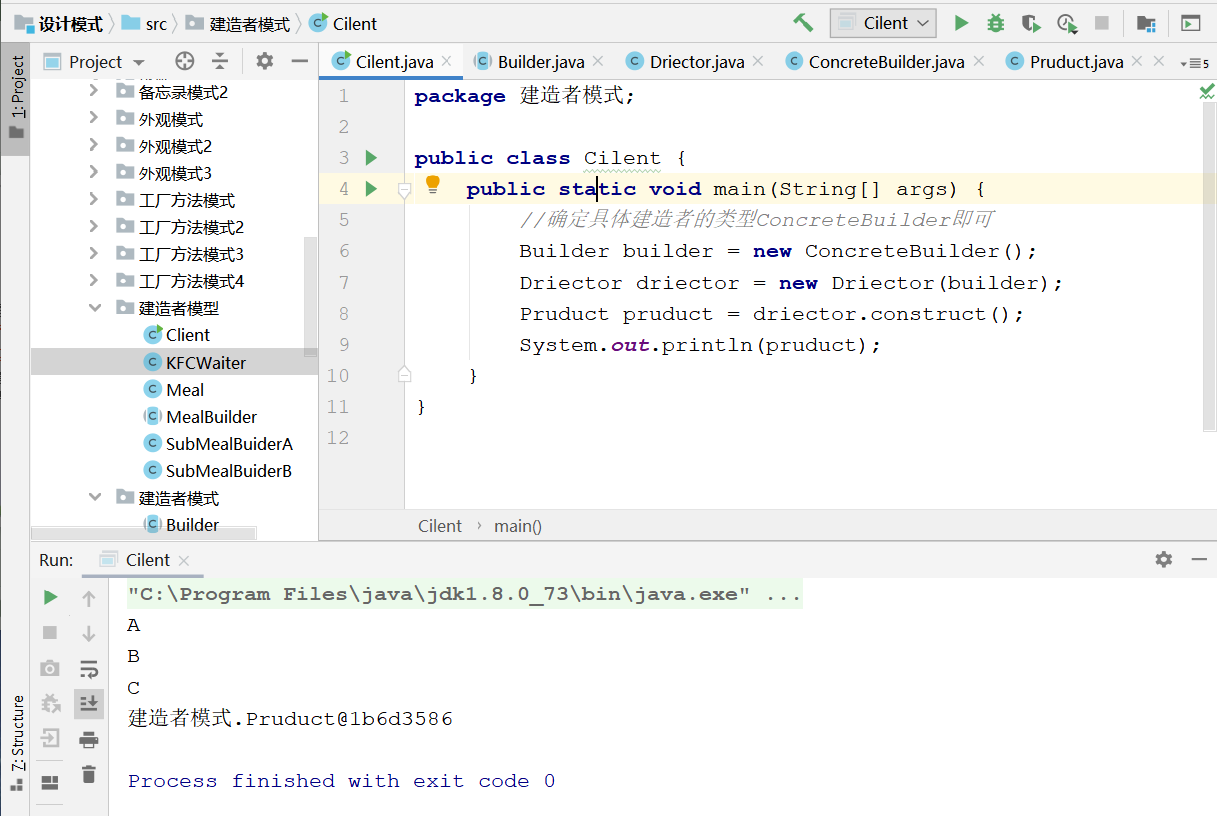
Pruduct pruduct = driector.construct();

System.out.println(pruduct);

}

}

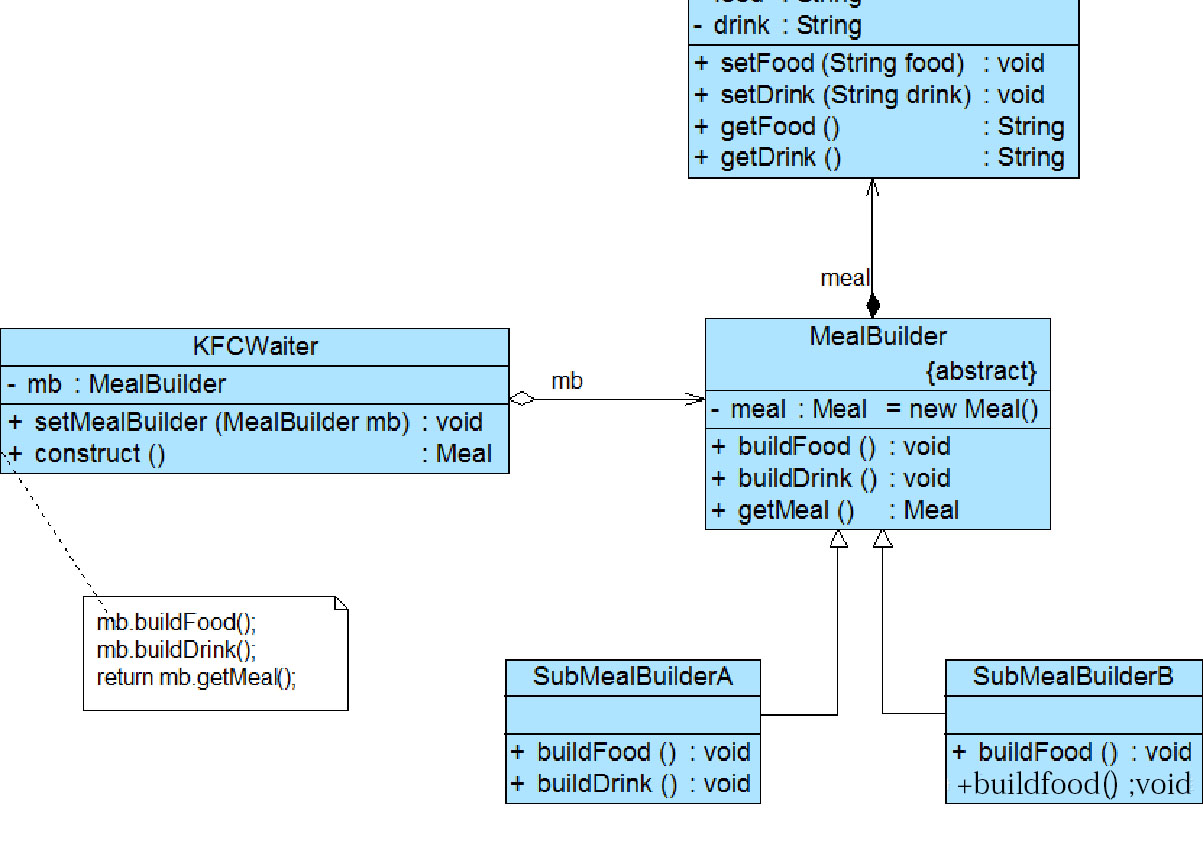
* 在客户端代码中，无须关心产品对象的具体组装过程，只需确定具体建造者的类型即可，建造者模式将复杂对象的构建与对象的表现分离开来，这样使得同样的构建过程可以创建出不同的表现
* 结构展示：



#### **5、模式实例与分析**

##### **KFC套餐**

建造者模式可以用于描述KFC如何创建套餐：套餐是一个复杂对象，它一般包含主食（如汉堡、鸡肉卷等）和饮料（如果汁、可乐等）等组成部分，不同的套餐有不同的组成部分，而KFC的服务员可以根据顾客的要求，一步一步装配这些组成部分，构造一份完整的套餐，然后返回给顾客。

* 代码实现

package 建造者模型;

//产品角色

public class Meal {

private String food;

private String drink;

public String getFood() {

return food;

}

public void setFood(String food) {

this.food = food;

}

public String getDrink() {

return drink;

}

public void setDrink(String drink) {

this.drink = drink;

}

}

package 建造者模型;

//抽象建造者

public abstract class MealBuilder {

protected Meal meal= new Meal();

public abstract void buildFood();

public abstract void buildDrink();

public Meal getMeal(){

return meal;

}

}

package 建造者模型;

// 具体建造者

public class SubMealBuiderA extends MealBuilder {

@Override

public void buildFood() {

meal.setFood("鸡腿汉堡");

}

@Override

public void buildDrink() {

meal.setDrink("冰镇可乐");

}

}

package 建造者模型;

// 具体建造者

public class SubMealBuiderB extends MealBuilder {

@Override

public void buildFood() {

meal.setFood("鸡肉卷");

}

@Override

public void buildDrink() {

meal.setDrink("雪碧");

}

}

package 建造者模型;

//指挥者

public class KFCWaiter {

private MealBuilder mb;

public void setMb(MealBuilder mb) {

this.mb = mb;

}

public Meal construct(){

mb.buildFood();

mb.buildDrink();

return mb.getMeal();

}

}

package 建造者模型;

import 工厂方法模式2.XMLUtil;

//客户端代码

public class Client {

public static void main(String[] args) {

// 确定套餐种类

MealBuilder mb = (MealBuilder) XMLUtil.getBean();

// 服务员是指挥者

KFCWaiter waiter = new KFCWaiter();

//服务员准备套餐

waiter.setMb(mb);

//客户获得套餐

Meal meal =waiter.construct();

System.out.println("套餐组成：");

System.out.println(meal.getFood());

System.out.println(meal.getDrink());

}

}

* config.xml配置：

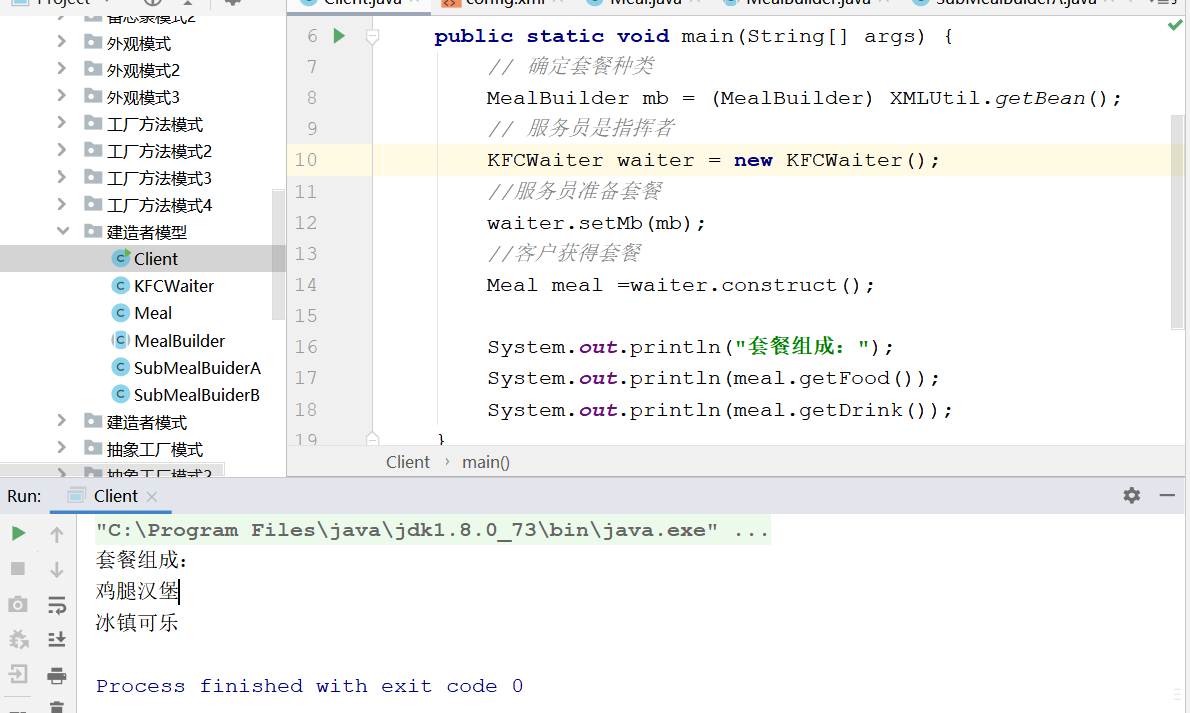
<?xml version="1.0"?>

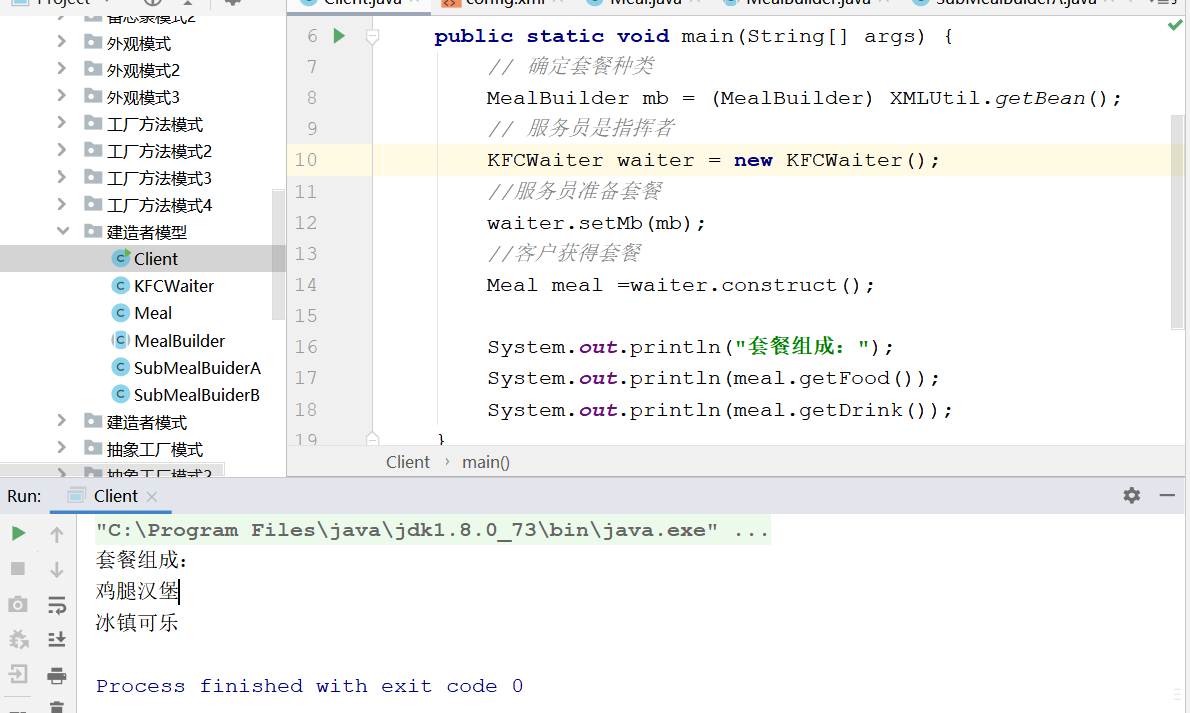
<config>

<className>建造者模型.SubMealBuiderA</className>

</config>

·结果展示：





6、模式的优缺点

优点：

1、在建造者模式中，客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象。

2、每一个具体建造者都相对独立，而与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者，用户使用不同的具体建造者即可得到不同的产品对象。增加新的具体建造者无须修改原有类库的代码，指挥者类针对抽象建造者类编程，系统扩展方便，符合“开闭原则”。

3、可以更加精细地控制产品的创建过程。将复杂产品的创建步骤分解在不同的方法中，使得创建过程更加清晰，也更方便使用程序来控制创建过程。

缺点：

1、建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，则不适合使用建造者模式，因此其使用范围受到一定的限制。

2、如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大。

7、模式的使用环境

使用环境：

1、需要生成的产品对象有复杂的内部结构，这些产品对象通常包含多个成员属性。

需要生成的产品对象的属性相互依赖，需要指定其生成顺序。

2、对象的创建过程独立于创建该对象的类。在建造者模式中引入了指挥者类，将创建过程封装在指挥者类中，而不在建造者类中。

3、隔离复杂对象的创建和使用，并使得相同的创建过程可以创建不同的产品。

#### **模式应用**

JavaMail



在很多游戏软件中，地图包括天空、地面、背景等组成部分，人物角色包括人体、服装、装备等组成部分，可以使用建造者模式对其进行设计，通过不同的具体建造者创建不同类型的地图或人物。

建造者模式的简化

1、省略抽象建造者角色：如果系统中只需要一个具体建造者的话，可以省略掉抽象建造者。

2、省略指挥者角色：在具体建造者只有一个的情况下，如果抽象建造者角色已经被省略掉，那么还可以省略指挥者角色，让Builder角色扮演指挥者与建造者双重角色。

建造者模式与抽象工厂模式的比较

1、与抽象工厂模式相比，建造者模式返回一个组装好的完整产品，而抽象工厂模式返回一系列相关的产品，这些产品位于不同的产品等级结构，构成了一个产品族。

2、在抽象工厂模式中，客户端实例化工厂类，然后调用工厂方法获取所需产品对象，而在建造者模式中，客户端可以不直接调用建造者的相关方法，而是通过指挥者类来指导如何生成对象，包括对象的组装过程和建造步骤，它侧重于一步步构造一个复杂对象，返回一个完整的对象。

3、如果将抽象工厂模式看成汽车配件生产工厂，生产一个产品族的产品，那么建造者模式就是一个汽车组装工厂，通过对部件的组装可以返回一辆完整的汽车。