

# 建造者模式

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。

一个 Builder 类会一步一步构造最终的对象。该 Builder 类是独立于其他对象的。

## 介绍

**意图：**将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

**主要解决：**主要解决在软件系统中，有时候面临着"一个复杂对象"的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。

**何时使用：**一些基本部件不会变，而其组合经常变化的时候。

**如何解决：**将变与不变分离开。

**关键代码：**建造者：创建和提供实例，导演：管理建造出来的实例的依赖关系。

**应用实例：**1、去肯德基，汉堡、可乐、薯条、炸鸡翅等是不变的，而其组合是经常变化的，生成出所谓的"套餐"。2、JAVA 中的 StringBuilder。

**优点：**1、建造者独立，易扩展。2、便于控制细节风险。

**缺点：**1、产品必须有共同点，范围有限制。2、如内部变化复杂，会有很多的建造类。

**使用场景：**1、需要生成的对象具有复杂的内部结构。2、需要生成的对象内部属性本身相互依赖。

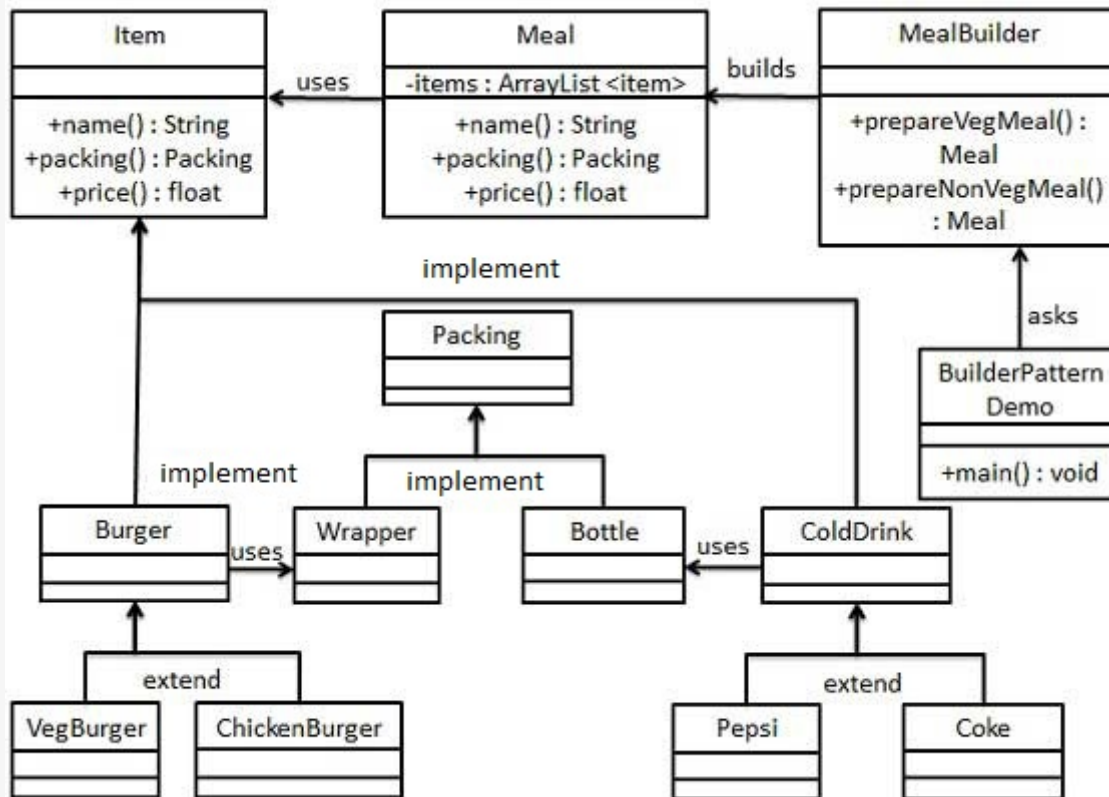
**注意事项：**与工厂模式的区别是：建造者模式更加关注与零件装配的顺序。

## 实现

我们假设一个快餐店的商业案例，其中，一个典型的套餐可以是一个汉堡 ( Burger ) 和一杯冷饮 ( Cold drink )。汉堡 ( Burger ) 可以是素食汉堡 ( Veg Burger ) 或鸡肉汉堡 ( Chicken Burger )，它们是包在纸盒中。冷饮 ( Cold drink ) 可以是可口可乐 ( coke ) 或百事可乐 ( pepsi )，它们是装在瓶子中。

我们将创建一个表示食物条目 ( 比如汉堡和冷饮 ) 的 *Item* 接口和实现 *Item* 接口的实体类，以及一个表示食物包装的 *Packing* 接口和实现 *Packing* 接口的实体类，汉堡是包在纸盒中，冷饮是装在瓶子中。

然后我们创建一个 *Meal* 类，带有 *Item* 的 *ArrayList* 和一个通过结合 *Item* 来创建不同类型的 *Meal* 对象的 *MealBuilder*。 *BuilderPatternDemo*，我们的演示类使用 *MealBuilder* 来创建一个 *Meal*。



## 步骤 1

创建一个表示食物条目和食物包装的接口。

## Item.java

```
public interface Item {
    public String name();
    public Packing packing();
    public float price();
}
```

## Packing.java

```
public interface Packing {
    public String pack();
}
```

## 步骤 2

创建实现 Packing 接口的实体类。

## Wrapper.java

```
public class Wrapper implements Packing {
    @Override
    public String pack() {
        return "Wrapper";
    }
}
```

**Bottle.java**

```
public class Bottle implements Packing {  
    @Override  
    public String pack() {  
        return "Bottle";  
    }  
}
```

**步骤 3**

创建实现 Item 接口的抽象类，该类提供了默认的功能。

**Burger.java**

```
public abstract class Burger implements Item {  
    @Override  
    public Packing packing() {  
        return new Wrapper();  
    }  
    @Override  
    public abstract float price();  
}
```

**ColdDrink.java**

```
public abstract class ColdDrink implements Item {  
    @Override  
    public Packing packing() {  
        return new Bottle();  
    }  
    @Override  
    public abstract float price();  
}
```

**步骤 4**

创建扩展了 Burger 和 ColdDrink 的实体类。

**VegBurger.java**

```
public class VegBurger extends Burger {  
    @Override  
    public float price() {  
        return 25.0f;  
    }  
    @Override  
    public String name() {  
        return "Veg Burger";  
    }  
}
```

**ChickenBurger.java**

```
public class ChickenBurger extends Burger {  
    @Override  
    public float price() {  
        return 50.5f;  
    }  
    @Override  
    public String name() {  
        return "Chicken Burger";  
    }  
}
```

### Coke.java

```
public class Coke extends ColdDrink {  
    @Override  
    public float price() {  
        return 30.0f;  
    }  
    @Override  
    public String name() {  
        return "Coke";  
    }  
}
```

### Pepsi.java

```
public class Pepsi extends ColdDrink {  
    @Override  
    public float price() {  
        return 35.0f;  
    }  
    @Override  
    public String name() {  
        return "Pepsi";  
    }  
}
```

## 步骤 5

创建一个 Meal 类，带有上面定义的 Item 对象。

### Meal.java

```
import java.util.ArrayList;  
import java.util.List;  
public class Meal {  
    private List<Item> items = new ArrayList<Item>();  
    public void addItem(Item item){  
        items.add(item);  
    }  
    public float getCost(){  
        float cost = 0.0f;  
        for (Item item : items) {  
            cost += item.price();  
        }  
    }  
}
```

```
}  
return cost;  
}  
public void showItems(){  
for (Item item : items) {  
System.out.print("Item : "+item.name());  
System.out.print(", Packing : "+item.packing().pack());  
System.out.println(", Price : "+item.price());  
}  
}  
}
```

## 步骤 6

创建一个 MealBuilder 类，实际的 builder 类负责创建 Meal 对象。

### MealBuilder.java

```
public class MealBuilder {  
public Meal prepareVegMeal () {  
Meal meal = new Meal();  
meal.addItem(new VegBurger());  
meal.addItem(new Coke());  
return meal;  
}  
public Meal prepareNonVegMeal () {  
Meal meal = new Meal();  
meal.addItem(new ChickenBurger());  
meal.addItem(new Pepsi());  
return meal;  
}  
}
```

## 步骤 7

BuilderPatternDemo 使用 MealBuilder 来演示建造者模式 ( Builder Pattern )。

### BuilderPatternDemo.java

```
public class BuilderPatternDemo {  
public static void main(String[] args) {  
MealBuilder mealBuilder = new MealBuilder();  
Meal vegMeal = mealBuilder.prepareVegMeal();  
System.out.println("Veg Meal");  
vegMeal.showItems();  
System.out.println("Total Cost: " + vegMeal.getCost());  
Meal nonVegMeal = mealBuilder.prepareNonVegMeal();  
System.out.println("\n\nNon-Veg Meal");  
nonVegMeal.showItems();  
System.out.println("Total Cost: " + nonVegMeal.getCost());  
}  
}
```

## 步骤 8

执行程序，输出结果：

```
Veg Meal
Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
Total Cost: 55.0

Non-Veg Meal
Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
Total Cost: 85.5
```

[← 单例模式](#)[原型模式 →](#)

## 2 篇笔记

[📝 写笔记](#)

建造者模式举例：去肯德基点餐，我们可以认为点餐就属于一个建造订单的过程。我们点餐的顺序是无关的，点什么东西也是没有要求的，可以单点，也可以点套餐，也可以套餐加单点，但是最后一定要点确认来完成订单。

```
public class OrderBuilder{
    private Burger mBurger;
    private Suit mSuit;

    //单点汉堡,num为数量
    public OrderBuilder burger(Burger burger, int num){
        mBurger = burger;
    }

    //点套餐，实际中套餐也可以点多份
    public OrderBuilder suit(Suit suit, int num){
        mSuit = suit;
    }

    //完成订单
    public Order build(){
        Order order = new Order();
        order.setBurger(mBurger);
        order.setSuit(mSuit);
        return order;
    }
}
```

另外适用于快速失败，在 build 时可以做校验，如果不满足必要条件，则可以直接抛出创建异常，在 OkHttpClient 中的 Request.Builder 中就是这样用的。

```
public Request build() {  
    if (url == null) throw new IllegalStateException("url == null");  
    return new Request(this);  
}
```

例如订单要求价格至少达到 30 块:

```
//完成订单  
public Order build(){  
    Order order = new Order();  
    order.setBurger(mBurger);  
    order.setSuit(mSuit);  
    if(order.getPrice() < 30){  
        throw new BuildException("订单金额未达到30元");  
    }  
    return order;  
}
```

另外，在构建时如果有必传参数和可选参数，可以为 Builder 类添加构造函数来保证必传参数不会遗漏，例如在构建一个 http 请求时, url 是必传的:

```
public class RequestBuilder{  
    private final String mUrl;  
    private Map<String, String> mHeaders = new HashMap<String, String>();  
  
    private RequestBuilder(String url){  
        mUrl = url;  
    }  
  
    public static RequestBuilder newBuilder(String url){  
        return new RequestBuilder(url);  
    }  
  
    public RequestBuilder addHeader(String key, String value){  
        mHeaders.put(key, value);  
    }  
  
    public Request build(){  
        Request request = new Request();  
        request.setUrl(mUrl);  
        request.setHeaders(mHeaders);  
        return request;  
    }  
}
```

牟春平 8个月前 (07-26)



建造者模式，又称生成器模式：将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

三个角色：**建造者**、**具体的建造者**、**监工**、**使用者**（严格来说不算）

- **建造者角色**：定义生成实例所需要的所有方法；
- **具体的建造者角色**：实现生成实例所需要的所有方法，并且定义获取最终生成实例的方法；
- **监工角色**：定义使用建造者角色中的方法来生成实例的方法；
- **使用者**：使用建造者模式。

**注意**：定义中“**将一个复杂的构建过程与其表示相分离**”，表示并不是由建造者负责一切，而是由监工负责控制（定义）一个复杂的构建过程，由各个不同的建造者分别负责实现构建过程中所用到的所有构建步骤。不然，就无法做到“使得同样的构建过程可以创建不同的表示”这一目标。

建造者角色：

```
public abstract class Builder {  
    public abstract void buildPart1();  
    public abstract void buildPart2();  
    public abstract void buildPart3();  
}
```

监工角色：

```
public class Director {    // 将一个复杂的构建过程与其表示相分离  
    private Builder builder;    // 针对接口编程，而不是针对实现编程  
    public Director(Builder builder) {  
        this.builder = builder;  
    }  
    public void setBuilder(Builder builder) {  
        this.builder = builder;  
    }  
  
    public void construct() {    // 控制（定义）一个复杂的构建过程  
        builder.buildPart1();  
        for (int i = 0; i < 5; i++) {    // 提示：如果想在运行过程中替换构建算法，可以考虑结合策略模式。  
            builder.buildPart2();  
        }  
        builder.buildPart3();  
    }  
}
```

具体的建造者角色：

```
/**  
 * 此处实现了建造纯文本文档的具体建造者。  
 * 可以考虑再实现一个建造HTML文档、XML文档，或者其它什么文档的具体建造者。  
 * 这样，就可以使得同样的构建过程可以创建不同的表示  
 */  
public class ConcreteBuilder1 extends Builder {
```



```
private StringBuffer buffer = new StringBuffer();//假设 buffer.toString() 就是最终生成的产品
```

```
@Override
public void buildPart1() { //实现构建最终实例需要的所有方法
    buffer.append("Builder1 : Part1\n");
}
```

```
@Override
public void buildPart2() {
    buffer.append("Builder1 : Part2\n");
}
```

```
@Override
public void buildPart3() {
    buffer.append("Builder1 : Part3\n");
}
```

```
public String getResult() { //定义获取最终生成实例的方法
    return buffer.toString();
}
}
```

客户角色：

```
public class Client {
    public void testBuilderPattern() {
        ConcreteBuilder1 b1 = new ConcreteBuilder1();//建造者
        Director director = new Director(b1);//监工
        director.construct();//建造实例(监工负责监督, 建造者实际建造)
        String result = b1.getResult();//获取最终生成结果
        System.out.printf("the result is :%n%s", result);
    }
}
```

jade 6个月前 (09-20)