自由扩展你的项目——Bulider模式

1.Builder模式介绍

Bulider模式是一步一步创建一个复杂对象的创建一个复杂的创建型模式，它允许用户在不知道内部构建细节的情况下，可以更精细地控制对象的构造流程。该模式是为了将构建复杂对象的过程和它的部件解耦，使得构建过程和部件的表示隔离开来。
因为一个复杂的对象有很多大量组成部分，如汽车，有车轮、方向盘、发动机，还有各种小零件等，如何将这些部分装配成一辆汽车，这个装配过程很漫长，也很复杂，对于这种情况，为了在构建过程中对外部隐藏实现细节，就可以使用Biulder模式将部件和组装过程分离，使得构建过程和部件都可以自由扩展，两者之间的耦合也降到最低。

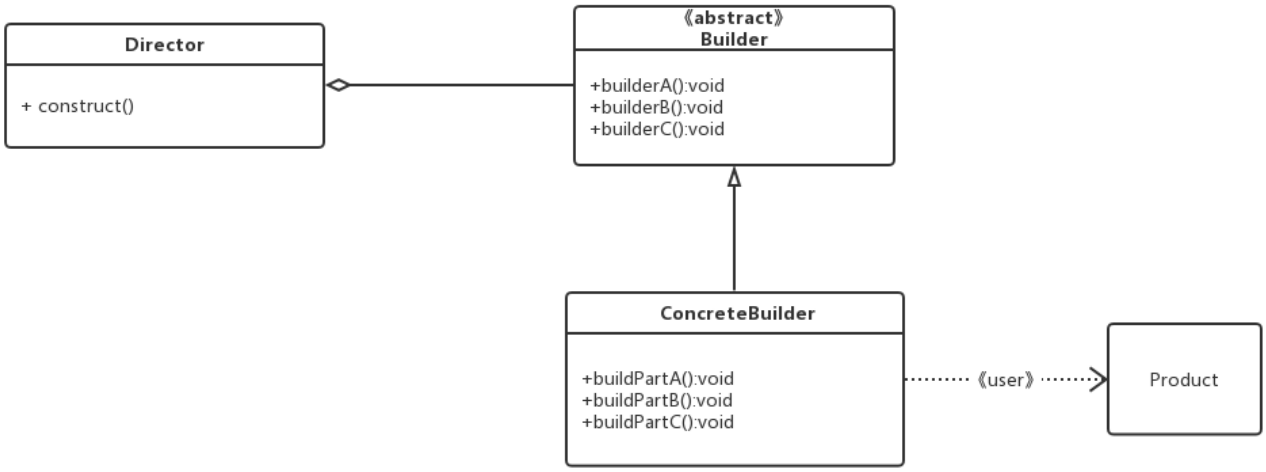
2.Builder模式的定义

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。

3.Builder模式的使用场景

1. 相同的方法，不同的执行顺序，产生不同的事件结果时；
2. 多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同时。
3. 产品类非常复杂，或者产品类中的调用顺序不同产生了不同的作用，这个时候使用建造者模式非常合适。
4. 当初始化一个对象特别复杂，如参数多，且很多参数都具有默认值时。

4.Builder模式的UML类图



角色介绍：

- Product产品类：产品的抽象类(一般来说，一个系统终会有多于一个产品类，而且这些产品类并不一定有共同的接口，而完全可以是不相关)；
- Builder：抽象Builder类，规范产品的组建，一般是由子类实现具体的组建过程（一般而言，以接口独立于应用程序的商业逻辑。模式中直接创建产品对象的是具体的建造者< ConcreteBuilder >角色。具体建造者类必须实现这个接口所要求的两种方法：**一种是建造方法；另一种是结果返还方法。**一般来说，产品所包含的部件数目与建造方法的数目相符。换言之，有多少部件，就有多少相应的建造方法）；
- ConcreteBuilder：具体的Builder类(担任这个角色的是与应用程序紧密相关的一些类，它们在应用程序调用下创建产品的实例。这个角色要完成的任务包括：实现抽象建造者Builder所声明的接口，给出一步一步地完成创建产品实例的操作。在创建过程完成后，提供产品的实例)；
- Director：统一组装过程(担任这个角色的类调用具体建造者角色以创建产品对象。导演者角色并没有产品类的具体知识，真正拥有产品类的具体知识的是具体建造者角色)。

5.Builder模式的简单实现

计算机的组装过程较为复杂，并且组装顺序是不固定的，为了易于理解，我们把计算机组装的过程简化为构建主机、设置操作系统、设置显示器3个部分,然后通过Director和具体的Builder来构建计算机对象。请看下看的示例：

```
1  //计算机抽象类
2  public abstract class Computer {
3
4          protected String mBoard;
5          protected String mDisplay;
6          protected String mOS;
7
8          protected Computer() {
9
10         }
11
12         public void setBoard(String board) {
13                 this.mBoard = board;
14         }
15
```

```
        public void setDisplay(String display) {
                this.mDisplay = display;
        }

        public abstract void setOS();

        @Override
        public String toString() {
                return "Computer [mBoard=" + mBoard + ",mDisplay=" + mDisplay + ",mOS="
                                + mOS + "]";
        }

}

```

具体的Computer类，Macbook

```
public class MacBook extends Computer {

        @Override
        public void setOS() {
                mOS = "Mas OS X 10.10";
        }

        protected MacBook() {

        }

}
```

//抽象Builder类

```
//抽象Builder类
public abstract class Builder {
        // 设置主机
        public abstract void buildBoard(String board);

        // 设置显示器
        public abstract void buildDisplay(String display);

        // 设置操作系统
        public abstract void buildOS();

        // 创建Computer
        public abstract Computer create();

}

```

具体的Builder类，Macbookbuilder

```
public class MacbookBuilder extends Builder {

        private Computer mComputer = new MacBook();
```

```
4
5           @Override
6           public void buildBoard(String board) {
7                   mComputer.setBoard(board);
8           }
9
10          @Override
11          public void buildDisplay(String display) {
12                  mComputer.setDisplay(display);
13          }
14
15          @Override
16          public void buildOS() {
17                  mComputer.setOS();
18          }
19
20          @Override
21          public Computer create() {
22                  return mComputer;
23          }
24
25  }
26
```

Director类，负责构造Computer

```
1   //Director类，负责构造Computer
2   public class Director {
3           Builder mBuilder = null;
4
5           public Director(Builder builder) {
6                   this.mBuilder = builder;
7           }
8
9           public void construct(String board, String display) {
10                  mBuilder.buildBoard(board);
11                  mBuilder.buildDisplay(display);
12                  mBuilder.buildOS();
13          }
14
15  }
```

测试代码

```
1   public class Test {
2           public static void main(String[] args) {
3                   // 构造器
4                   Builder builder = new MacbookBuilder();
5                   // Director
6                   Director pcDirector = new Director(builder);
7                   // 封装构建过程，4核，内存2G、Mac系统
8                   pcDirector.construct("英特尔主板", "Retina显示器");
9                   // 构建计算机，输出相关信息
10                  System.out.println("Computer Info:" + builder.create().toString());
11
```

```
12          }
    }
```

输出：

```
Computer Info:Computer [mBoard=英特尔主板,mDisplay=Retina显示器,mOS=Mas OS X 10.10]
```

上述示例中，通过具体的MacbookBuilder来构建的Macbook对象，而Director封装了构建复杂产品对象的过程，对外隐藏构建细节。Builder与Director一起将一个复杂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对象。
值得注意的是，在现实开发中，Director角色经常会被省略。而直接使用一个Builder来进行对象的组装，这个Builder通常为链式调用，它的关键点是每个setter方法都返回自身，也就是return this，这样就使得setter方法可以链式调用，代码大致如下：

```
new TestBuilder.setA("A").setB("B").setC("C").create();
```

通过这种形式不仅去除了Director角色吗整个结构也变简单了，特能对Product对象的组装过程有更精细的控制。

6.总结

Builder模式在Android开发中也较为常用，
优点

- 良好的封装性，使用建造者模式可以使客户端不必知道产品内部组成的细节。
- 构建者独立，容易扩展。
  缺点
- 会产生多余的Builder对象以及Director对象，消耗内存。

——摘录《Android 源码设计模式解析与实战 第三章》

xyzso1z
原创文章 80　获赞 40　访问量 2万+