建造者模式

2020年2月24日 12:10

1. 简介

1.1 定义

隐藏创建对象的建造过程 & 细节，使得用户在不知对象的建造过程 & 细节的情况下，就可直接创建复杂的对象

1. 用户只需要给出指定复杂对象的类型和内容；
2. 建造者模式负责按顺序创建复杂对象（把内部的建造过程和细节隐藏起来）- 可以看作把set过程隐藏了

1.2 作用

1. 降低创建复杂对象的复杂度
2. 隔离了创建对象的构建过程 & 表示
3. 方便用户创建复杂的对象（不需要知道实现过程）
4. 代码复用性 & 封装性（将对象构建过程和细节进行封装 & 复用）

   例子：造汽车 & 买汽车。
   1. 工厂（建造者模式）：负责制造汽车（组装过程和细节在工厂内）
   2. 汽车购买者（用户）：你只需要说出你需要的型号（对象的类型和内容），然后直接购买就可以使用了（不需要知道汽车是怎么组装的（车轮、车门、发动机、方向盘等等））

2. 模式原理

2.1 UML类图

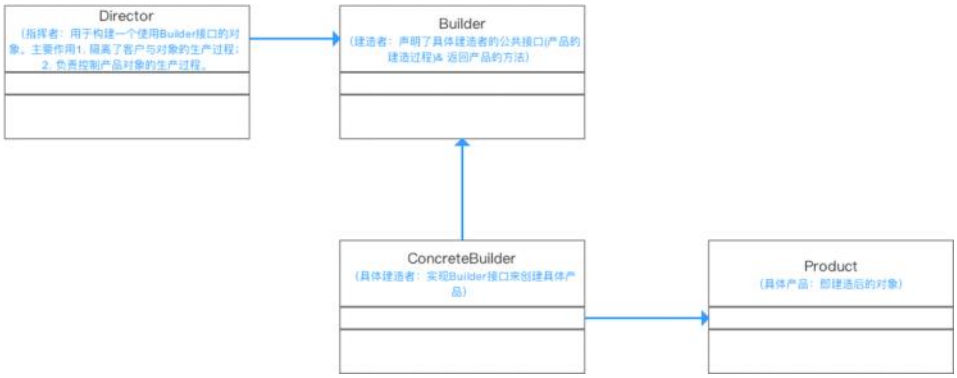

图2.1 建造者模式UML类图

1. 指挥者（Director）直接和客户（Client）进行需求沟通；
2. 沟通后指挥者将客户创建产品的需求划分为各个部件的建造请求（Builder）；
3. 将各个部件的建造请求委派到具体的建造者（ConcreteBuilder）；
4. 各个具体建造者负责进行产品部件的构建；
5. 最终构建成具体产品（Product）。

3. 实例讲解

3.1 实例概况

- 背景

  小成希望去电脑城买一台组装的台式主机

- 过程

1. 电脑城老板（Diretor）和小成（Client）进行需求沟通（买来打游戏？学习？看片？）
2. 了解需求后，电脑城老板将小成需要的主机划分为各个部件（Builder）的建造请求（CPU、主板blabla）
3. 指挥装机人员（ConcreteBuilder）去构建组件；
4. 将组件组装起来成小成需要的电脑（Product）

3.2 使用步骤

步骤1：定义组装的过程（Builder）：组装电脑的过程

```
public  abstract class Builder {
//第一步：装CPU//声明为抽象方法，具体由子类实现
    public abstract void  BuildCPU();
//第二步：装主板//声明为抽象方法，具体由子类实现
    public abstract void BuildMainboard () ;
//第三步：装硬盘//声明为抽象方法，具体由子类实现
    public abstract void BuildHD () ;
//返回产品的方法：获得组装好的电脑
    public abstract Computer GetComputer () ;
}
```

步骤2： 电脑城老板委派任务给装机人员（Director）

```
public class Director{
      //指挥装机人员组装电脑
      public void Construct(Builder builder){
      builder. BuildCPU();
      builder.BuildMainboard () ;
      builder. BuildHD () ;
  }
}
```

步骤3： 创建具体的建造者（ConcreteBuilder）:装机人员

```
//装机人员1
  public class ConcreteBuilder extend  Builder{
    //创建产品实例
    Computer computer = new Computer();
    //组装产品
    @Override
    public void  BuildCPU(){
       computer.Add("组装CPU")
    }
    @Override
    public void  BuildMainboard () {
       computer.Add("组装主板")
    }
    @Override
    public void  BuildHD () {
       computer.Add("组装主板")
    }
    //返回组装成功的电脑
     @Override
      public  Computer GetComputer () {
      return computer
    }
}
```

步骤4： 定义具体产品类（Product）：电脑

```
public class Computer{

    //电脑组件的集合
    private List<String> parts = new ArrayList<String>();
```

```
    //用于将组件组装到电脑里
    public void Add(String part){
        parts.add(part);
    }

    public void Show(){
        for (int i = 0;i<parts.size();i++){
        System.out.println("组件"+parts.get(i)+"装好了");
    }
        System.out.println("电脑组装完成，请验收");
}}
```

步骤5：客户端调用-小成到电脑城找老板买电脑

```
public class Builder Pattern{
    public static void main(String[] args){
    //逛了很久终于发现一家合适的电脑店//找到该店的老板和装机人员
    Director director = new Director();
    Builder builder = new ConcreteBuilder();
    //沟通需求后，老板叫装机人员去装电脑
    director.Construct(builder);
    //装完后，组装人员搬来组装好的电脑
    Computer computer = builder.GetComputer();//组装人员展示电脑给小成看
    computer.Show();
  }
}
```

结果输出

```
组件CUP装好了
组件主板装好了
组件硬盘装好了
电脑组装完成，请验收
```

4. 优/缺点

4.1 优点

1. 易于解耦。将产品本身与产品创建过程进行解耦，可以使用相同的创建过程来得到不同的产品。也就说细节依赖抽象。
2. 易于精确控制对象的创建。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
3. 易于拓展。增加新的具体建造者无需修改原有类库的代码，易于拓展，符合“开闭原则“。
4. 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。

4.2 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。

5. 应用场景

1. 需要生成的产品对象有复杂的内部结构，这些产品对象具备共性；
2. 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。

6. 参考资料

[1] https://www.jianshu.com/p/be290ccea05a