文章已收录我的仓库：Java学习笔记与免费书籍分享

设计意图

为了将复杂对象的构建与它的表示分离，使得对象可以通过不同的表示创建出来。

例如对一个迷宫可能有墙、房间和门，并且数量不计。迷宫可能仅由一堵墙构成，也可能由两堵墙构成，也可能由2个房间加一扇门构成...如果采用重载的方式生产迷宫，代码量是难以计数的、无比庞大的。

针对一个对象拥有大量的组件（迷宫可以拥有很多墙或房间或门），而构造这个对象对其组件的使用又是不确定的这一问题（使用墙、房间、门的数量是不确定的），想要精细的控制构建过程，此时可以采用建造者模式解决问题。

建造者模式的意图是为了构造对象，因此它属于创建型模式。

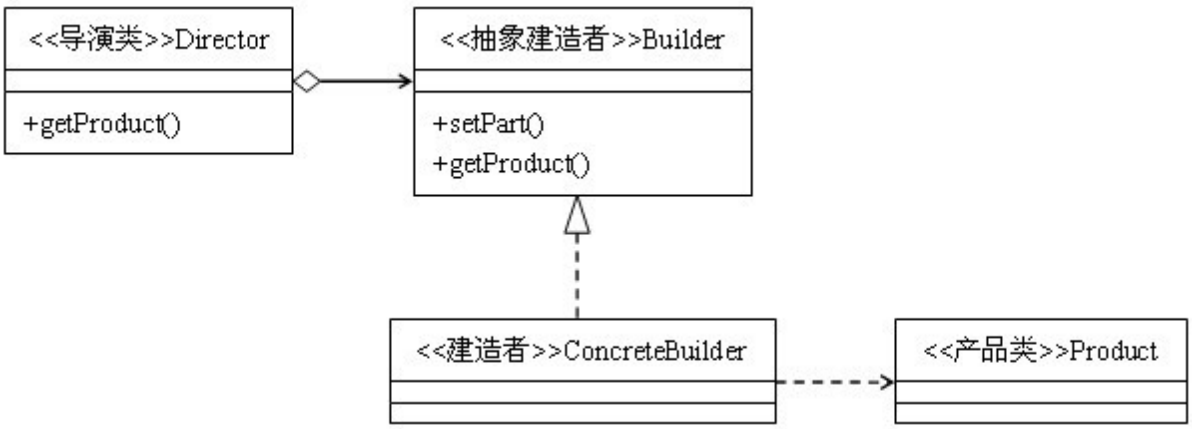
适用场景

- 构造过程中，被构造的对象具有不同的表示。
- 一些基本部件不会变，而其组合经常变化的时候。
- 需要生成的对象内部属性本身相互依赖。

设计

指派一个建造者交给总指挥，由总指挥设计如何建造，由建造者实际建造，最终建造者将产品交给总指挥，用户从总指挥手中完成交付。

这样的流程是非常人性化的，例如我们实际的房子装修也是如此，我们请一些工人们，再请一个工人头目设计建造思路，指导这些工人该如何装修，我们接下来只与工人头目打交道，由工人头目将装修好的房子交付给我们。



代码示例

考虑上述的迷宫问题：

```
//定义迷宫类
class Maze {
    public void setRoom(int x, int y, int roomId) {
        System.out.println("在[" + x + "," + y + "]处建立一座编号为" +
roomId + "的房间");
        //保存相关信息
    }

    public void setDoor(int Id1, int Id2) {
        System.out.println("在组件编号为" + Id1 + "和组件编号为" + Id2 + "之
间建立一扇门");
        //保存相关信息
    }

    public void setBarrier(int x, int y, int barrierId) {
        System.out.println("在[" + x + "," + y + "]处建立一座编号为" +
barrierId + "的障碍物");
        //保存相关信息
    }
}

//定义迷宫建造者接口
interface MazeBuilder {
    void buildRoom(int x, int y, int roomId);//建造房间
    void buildDoor(int Id1, int Id2);//建造门
    void buildBarrier(int x, int y, int barrierId);//建造障碍物
    Maze getMaze();//获取最终结果
}

//迷宫建造团队A，每个建造团队的建造方式是不一样的
class MazeBuilderA implements MazeBuilder {
    //建造团队将逐步完善这个迷宫，并交付给客户
    private Maze maze = new Maze();
```

```
    @Override
    public void buildRoom(int x, int y, int roomId) {
        maze.setRoom(x, y, roomId);
    }

    @Override
    public void buildDoor(int Id1, int Id2) {
        maze.setDoor(Id1, Id2);
    }

    @Override
    public void buildBarrier(int x, int y, int barrierId) {
        maze.setBarrier(x, y, barrierId);
    }

    @Override
    public Maze getMaze() {
        return maze;
    }
}

//迷宫指挥者类
class MazeDirecotr {
    MazeBuilder mazeBuilder;

    public MazeDirecotr(MazeBuilder mazeBuilder) {
        this.mazeBuilder = mazeBuilder;
    }

    //将工作交给指挥，用户只需要等待建筑完成取走实例
    public Maze getMaze() {
        //具体的设计思想在这里产生
        //在这里我们可以一步一步按照我们的想法构造出复杂的模型
        //例如这里我想要建造两个房间和中间的一扇门
        mazeBuilder.buildRoom(0, 0, 0);
        mazeBuilder.buildRoom(0, 1, 1);
        mazeBuilder.buildDoor(0, 1);

        //实际的工作是由工人们建造的
        //总指挥从工人们手中获取最终产品，由总指挥完成与用户的交接
        return mazeBuilder.getMaze();
    }
}

//测试类
public class Test {
    public static void main(String[] args) {
        //请工人
        MazeBuilder mazeBuilder = new MazeBuilderA();

        //请指挥，并将工人交给其指挥
        MazeDirecotr mazeDirecotr = new MazeDirecotr(mazeBuilder);

        //最终与总指挥完成产品交付
        Maze maze = mazeDirecotr.getMaze();
```

```
    }
}
//输出
/*
在[0,0]处建立一座编号为0的房间
在[0,1]处建立一座编号为1的房间
在组件编号为0和组件编号为1之间建立一扇门
*/
```

通过建造者模式，我们不再需要重载大量函数。通过设计导演类而一步一步的生成我们期望的对象，可以更加精细的控制创建的过程。实际开发中，导演类由用户自己编写定义，一个优化的技巧是采用链式编程，我们可以定义建造者接口如下：

```
//定义迷宫建造者接口
interface MazeBuilder {
    MazeBuilder buildRoom(int x, int y, int roomId);//建造房间
    MazeBuilder buildDoor(int Id1, int Id2);//建造门
    MazeBuilder buildBarrier(int x, int y, int barrierId);//建造障碍物
    Maze getMaze();//获取最终结果
}
```

那么我们就可以在导演类中这样编写代码：

```
                                                mazeBuilder.buildRoom(0, 0,
0).buildRoom(0,1,1).buildDoor(0, 1);
```

优缺点总结

优点：

1、建造者独立，采用接口方式，易扩展。

2、便于控制细节风险。

3、将设计与使用解耦，利于扩展与维护。

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。

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。