# 背景

在软件开发中存在着这样复杂的对象，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。并且，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一定的顺序，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，建造者返回给客户端的是一个已经构建完毕的完整的产品对象，而用户无需关心该对象所包含的属性以及它们的组装方式，这就是建造者模式存在的背景。

# 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，又称为生成器模式。

# 结构

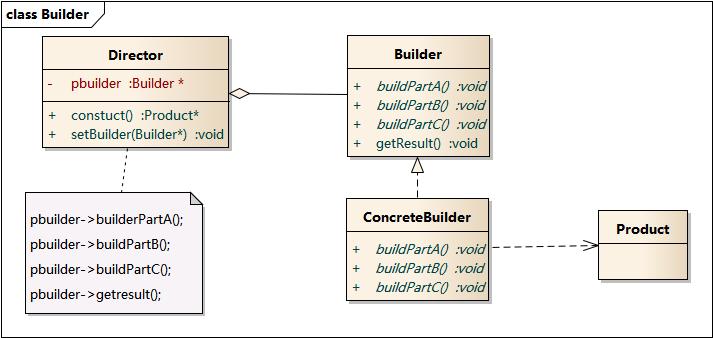
建造者模式包含以下角色：

Builder：抽象建造者

ConcreteBuilder：具体建造者

Director：指挥者

Product：产品角色



# 代码

#include <iostream>

#include "ConcreteBuilder.h"

#include "Director.h"

#include "Builder.h"

#include "Product.h"

using namespace std;

int main(int argc, char \*argv[])

{

ConcreteBuilder \* builder = new ConcreteBuilder();

Director director;

director.setBuilder(builder);

Product \* pd = director.constuct();

pd->show();

delete builder;

delete pd;

return 0;

}

///////////////////////////////////////////////////////////

// ConcreteBuilder.cpp

// Implementation of the Class ConcreteBuilder

// Created on: 02-十月-2014 15:57:03

// Original author: colin

///////////////////////////////////////////////////////////

#include "ConcreteBuilder.h"

ConcreteBuilder::ConcreteBuilder(){

}

ConcreteBuilder::~ConcreteBuilder(){

}

void ConcreteBuilder::buildPartA(){

m\_prod->setA("A Style "); //不同的建造者，可以实现不同产品的建造

}

void ConcreteBuilder::buildPartB(){

m\_prod->setB("B Style ");

}

void ConcreteBuilder::buildPartC(){

m\_prod->setC("C style ");

}

///////////////////////////////////////////////////////////

// Director.cpp

// Implementation of the Class Director

// Created on: 02-十月-2014 15:57:01

// Original author: colin

///////////////////////////////////////////////////////////

#include "Director.h"

Director::Director(){

}

Director::~Director(){

}

Product\* Director::constuct(){

m\_pbuilder->buildPartA();

m\_pbuilder->buildPartB();

m\_pbuilder->buildPartC();

return m\_pbuilder->getResult();

}

void Director::setBuilder(Builder\* buider){

m\_pbuilder = buider;

}

# 分析

抽象建造者类中定义了产品的创建方法和返回方法：

建造者模式的结构中还引入了一个指挥者类Director，该类的作用主要有两个：一方面它隔离了客户与生产过程；另一方面它负责控制产品的生成过程。指挥者针对抽象建造者编程，客户端只需要知道具体建造者的类型，即可通过指挥者类调用建造者的相关方法，返回一个完整的产品对象。

# 实例

# 特点

## 优点

1、在建造者模式中，客户端无需知道产品内部组装的细节，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、每一个具体创建者都相对独立，而与其他的具体创建者无关，因此可以很方便地替换具体创建者或增加新的具体创建者，用户使用不同的具体创建者即可得到不同的产品对象；

3、可以更加精细地控制产品的创建过程，将具体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；

4、增加具体创建者无需修改原有类库的代码，指挥者类针对具体建造者类编程，系统扩展方便，符合“开闭原则”。

## 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。

# 应用

建造者模式一般适用于以下的情况：

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；

2、需要生成的对象的属性相互依赖，需要指定其生成顺序；

3、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，将建造过程封装在指挥者类中，而不是在建造者类中；

4、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。

在很多游戏中，地图包括天空、地面、背景灯组成部分，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和人物。