软件构造实验报告二

实验名称：生成器模式与原型模式编程实现

实验时间：2019.4.10

学号：E21614061

姓名：徐奕

所在院系：计算机科学与技术学院

所在专业：软件工程

【实验目的和要求】

a）熟悉并理解生成器模式与原型模式的原理与方法

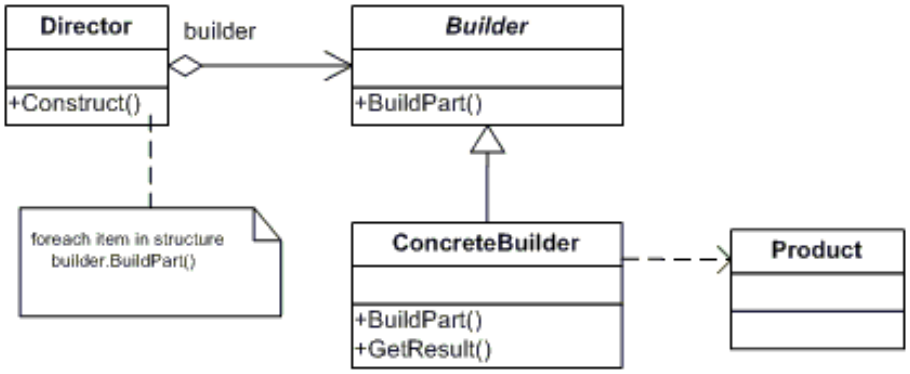
b）熟练掌握生成器模式与原型模式的代码与方法

【实验原理】

1.生成器模式

生成器模式可以将一个产品的内部表象与产品的生成过程分割开来，从而可以使一个建造过程生成具有不同的内部表象的产品对象。

Builder 模式的结构：



建造者（Builder）角色：给出一个抽象接口，以规范产品对象的各个组成成分的建造。一般而言，此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建造者（ConcreteBuilder）角色。具体建造者类必须实现这个接口所要求的方法：一个是建造方法，另一个是结果返还方法。

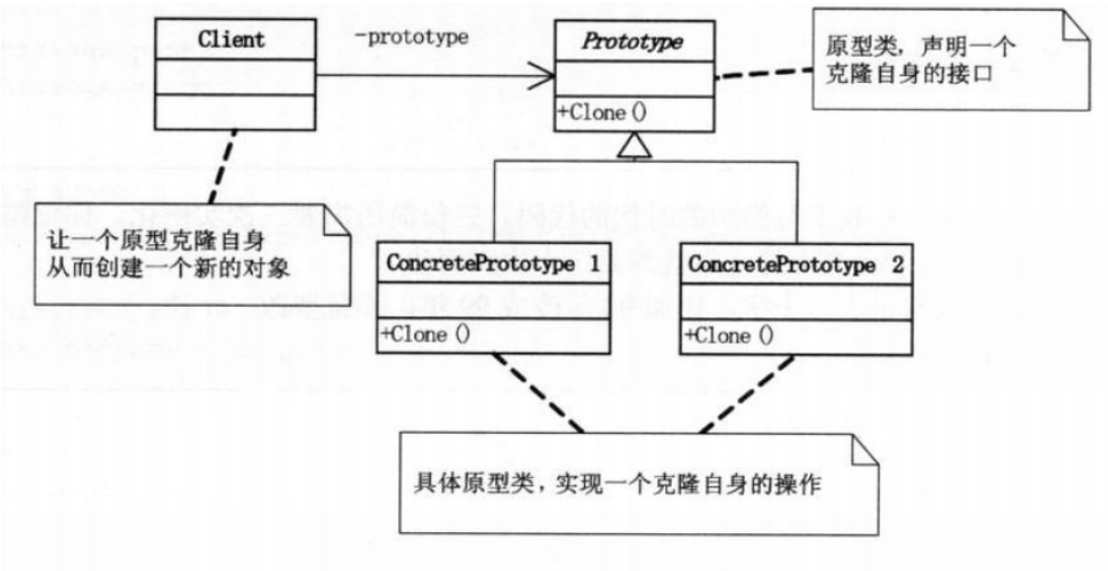
具体建造者（Concrete Builder）角色：担任这个角色的是于应用程序紧密相关的类，它们在应用程序调用下创建产品实例。这个角色主要完成的任务包括：实现 Builder 角色提供的接口，一步一步完成创建产品实例的过程。
在建造过程完成后，提供产品的实例。

指导者（Director）角色：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对象。

产品（Product）角色：产品便是建造中的复杂对象。

2.原型模式

在原型模式中，所发动创建的对象通过请求原型对象来拷贝原型对象自己来实现创建过程，当然所发动创建的对象需要知道原型对象的类型。这里也就是说所发动创建的对象只需要知道原型对象的类型就可以获得更多的原型实例对象，至于这些原型对象时如何创建的根本不需要关心。



浅拷贝：使用一个已知实例对新创建实例的成员变量逐个赋值，这个方式被称为浅拷贝。

深拷贝：当一个类的拷贝构造方法，不仅要复制对象的所有非引用成员变量值，还要为引用类型的成员变量创建新的实例，并且初始化为形式参数实例值。

【实验内容】

使用生成器模式模拟实现 IBM 电脑的生产，其中 IBM 电脑的主要结构用如下表示：

```
class IBM{
  string monitor=”IBM 的显示器”;
  string keyboard=”IBM 的键盘”;
  string mouse=”IBM 的鼠标”;
  Motherboard* MB;
  void display();
}
```

其中 MB 是一个主板类，其主要结构如下：

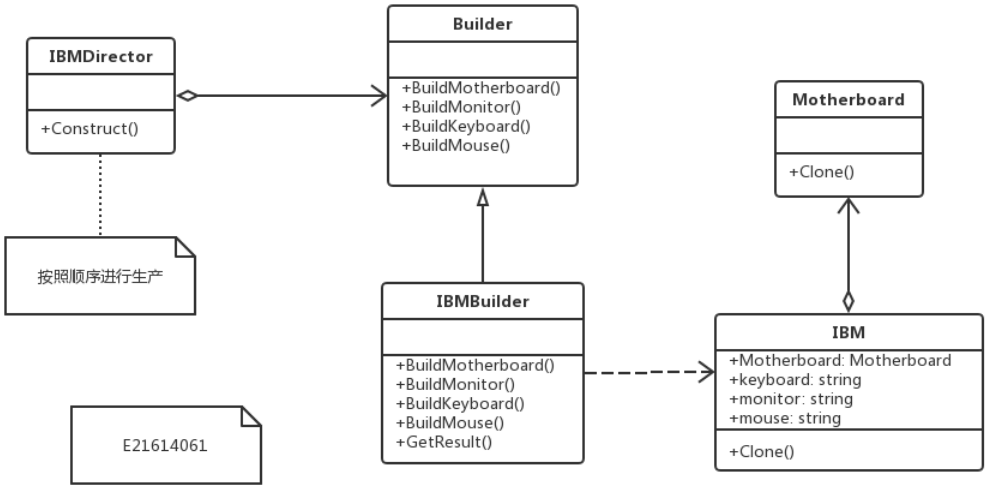
```
class Motherboard{
string CPU;
string RAM;
}
```

即主板包含 CPU 和 RAM。display 是一个打印各个组件的函数，主要用于检查是否生产正确。

建造顺序为先生产主板，再依次生产显示器、键盘和鼠标。

使用生成器模式生产出第一台 IBM 电脑后，利用原型模式，将该电脑再复制两台。

【实验 UML 图】



【实验代码与函数】

```
#include<iostream>
using namespace std;

/*
使用生成器模式模拟实现 IBM 电脑的生产，其中 IBM 电脑的主要结构用如下表示：
class IBM{
  string monitor=”IBM 的显示器”;
  string keyboard=”IBM 的键盘”;
  string mouse=”IBM 的鼠标”;
  Motherboard* MB;
  void display();
}
其中 MB 是一个主板类，其主要结构如下：
class Motherboard{
string CPU;
string RAM;
}
即主板包含 CPU 和 RAM。display 是一个打印各个组件的函数，主要用于检查是否生产正确。
建造顺序为先生产主板，再依次生产显示器、键盘和鼠标。
使用生成器模式生产出第一台 IBM 电脑后，利用原型模式，将该电脑再复制两台。
*/

class Motherboard{
public:
    string CPU;
    string RAM;
    Motherboard* Clone(){
        return new Motherboard(*this);
    }
};

class IBM{
public:
    IBM(){
        MB = new Motherboard();
    }
    string monitor;
    string keyboard;
    string mouse;
    Motherboard* MB;
    void display(){
```

```
        cout<<"-------------------------"<<endl;
        cout<<"IBM 电脑配置："<<endl;
        cout<<"主板："<<MB->CPU<<" "<<MB->RAM<<endl;
        cout<<"显示器："<<monitor<<endl;
        cout<<"键盘："<<keyboard<<endl;
        cout<<"鼠标："<<mouse<<endl;
        cout<<"-------------------------"<<endl<<endl;
    }

    void SetMotherboard(string CPU, string RAM){
        this->MB->CPU = CPU;
        this->MB->RAM = RAM;
    }

    void SetMonitor(string monitor){
        this->monitor = monitor;
    }

    void SetKeyboard(string keyboard){
        this->keyboard = keyboard;
    }

    void SetMouse(string mouse){
        this->mouse = mouse;
    }

    IBM* Clone(){
        IBM* obj = new IBM();
        obj->MB = MB->Clone();
        obj->monitor = monitor;
        obj->keyboard = keyboard;
        obj->mouse = mouse;
        return obj;
    }
private:
    /*IBM(Motherboard* mo){
        this->MB = mo->Clone();
    }*/
};

class Builder{
public:
    virtual void BuildMotherboard() = 0;
    virtual void BuildMonitor() = 0;
```

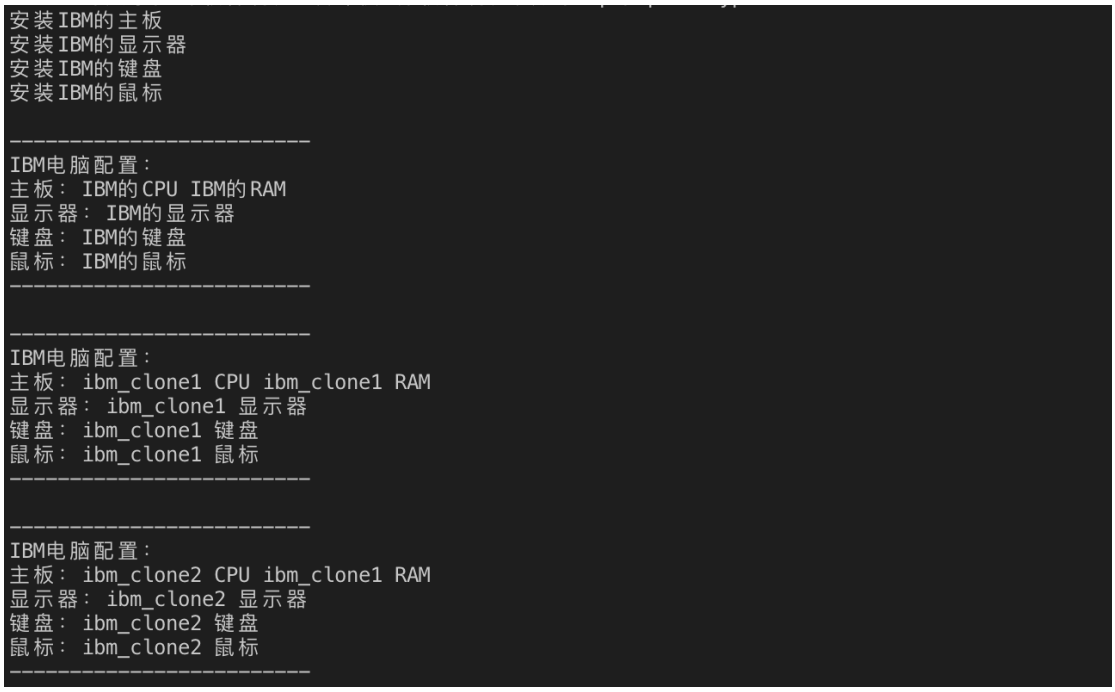
```
    virtual void BuildKeyboard() = 0;
    virtual void BuildMouse() = 0;
    virtual IBM* GetResult(){
        return NULL;
    }
};

class IBMBuilder: public Builder{
public:
    IBMBuilder(){
        _IBM = new IBM();
    }
    void BuildMotherboard(){
        _IBM->MB->CPU = "IBM 的 CPU";
        _IBM->MB->RAM = "IBM 的 RAM";
        cout<<"安装"<<"IBM 的主板"<<endl;
    }
    void BuildMonitor(){
        _IBM->monitor = "IBM 的显示器";
        cout<<"安装"<<_IBM->monitor<<endl;
    }
    void BuildKeyboard(){
        _IBM->keyboard = "IBM 的键盘";
        cout<<"安装"<<_IBM->keyboard<<endl;
    }
    void BuildMouse(){
        _IBM->mouse = "IBM 的鼠标";
        cout<<"安装"<<_IBM->mouse<<endl<<endl;
    }
    IBM* GetResult(){
        return _IBM;
    }
private:
    IBM* _IBM;
};

class IBMDirector{
public:
    void Construct(Builder* builder){
        builder->BuildMotherboard();
        builder->BuildMonitor();
        builder->BuildKeyboard();
        builder->BuildMouse();
    }
```

```
};


int main(){
    cout<<endl<<endl;
    IBMDirector* director = new IBMDirector();
    Builder* b1 = new IBMBuilder();
    director->Construct(b1);
    IBM* ibm1 = b1->GetResult();



    IBM* ibm_clone1 = ibm1->Clone();
    ibm_clone1->SetMotherboard("ibm_clone1 CPU", "ibm_clone1 RAM");
    ibm_clone1->SetMonitor("ibm_clone1 显示器");
    ibm_clone1->SetKeyboard("ibm_clone1 键盘");
    ibm_clone1->SetMouse("ibm_clone1 鼠标");

    IBM* ibm_clone2 = ibm1->Clone();
    ibm_clone2->SetMotherboard("ibm_clone2 CPU", "ibm_clone2 RAM");
    ibm_clone2->SetMonitor("ibm_clone2 显示器");
    ibm_clone2->SetKeyboard("ibm_clone2 键盘");
    ibm_clone2->SetMouse("ibm_clone2 鼠标");

    ibm1->display();
    ibm_clone1->display();
    ibm_clone2->display();
    return 0;
}
```

实验客户端中最后两个为克隆部分，为了验证是深拷贝，对 clone 出来的结果进行更改，最终验证结果正确。

【实验结果】

首先显示出安装的部分，接着第一部分为生产出的原型，接着最后两个为克隆的部分，克隆的部分为深拷贝更改了一些属性值。



【实验总结】

①在 new 一个 IBM 对象时，要注意，在类代码中一定要初始化一个 Motherboard（如下图所示）

```
IBM(){
    MB = new Motherboard();
}
string monitor;
string keyboard;
string mouse;
Motherboard* MB;
```

否则下面的语句在调用 MB（motherboard 的一个对象）时，会出现没有分配空间的情况，导致 Segmentation fault: 11 错误。

②在原型模式中，要有 Clone 函数，注意内嵌的 Motherboard 要一起克隆过来，否则会导致客户端不同的指针指向同一个对象，即深拷贝浅拷贝的区别。