

实验报告☑ 实践报告□

课程名称： 软件详细设计R

实验、实践名称： 实验一 创建型设计模式实验

实验、实践地点： 软件学院C2实验室

专业班级： 软件2028班 学号： 2020007657

学生姓名： 常兆海

指导教师： 曹锐

2023年 3月 21日

目录

[1 实验目的和要求： 3](#_Toc131931464)

[1.1 目的： 3](#_Toc131931465)

[1.2 要求： 3](#_Toc131931466)

[2 实验平台: 4](#_Toc131931467)

[3 主要实验内容及结果: 5](#_Toc131931468)

[3.1 实验内容 5](#_Toc131931469)

[3.1.1 抽象工厂模式的运用 5](#_Toc131931470)

[3.1.2 建造者模式的运用 10](#_Toc131931471)

[3.1.3 单例模式的运用 18](#_Toc131931472)

[3.2 心得体会 21](#_Toc131931473)

# 实验目的和要求：

## 目的：

1．综合实例，熟练绘制常见的创建型设计模式结构图。

\2. 结合实例，熟练掌握不同创建型设计模式的特点并能够根据不同需求使用不同的创建型设计模式

\3. 通过实验，熟练掌握不同创建型设计模式代码的编写

## 要求：

* 独立完成实验
* 书写实验报告书

# 实验平台:

JDK-1.8 idea—2022.3

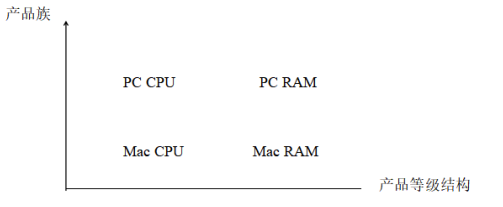
# 主要实验内容及结果:

## 实验内容

### 抽象工厂模式的运用

#### 案例背景：

计算机包含内存（RAM），CPU 等硬件设备，根据如图所示的“产品等级结构-产品族示意图”，使用 抽象工厂模式实现计算机设备创建过程并绘制类图（课本 105 页第二题）



#### 实现步骤：

* 根据题意，使用抽象工厂模式并画出类图，类图中应包含一个抽象工厂类 AbstractFactory， PcFactory 和 MacFactory 两个具体工厂，CPU，RAM 两个抽象产品类，PcCPU，MacCPU， PcRAM，MacRAM 四个具体产品类。
* 根据类图，实现上述类的具体代码以及用户类 Client 和辅助类 XMLUtil 以实现通过 XML 文 件来制造不同的产品
* 更改 XML 中的属性，观察用户类是否能使用不同的产品

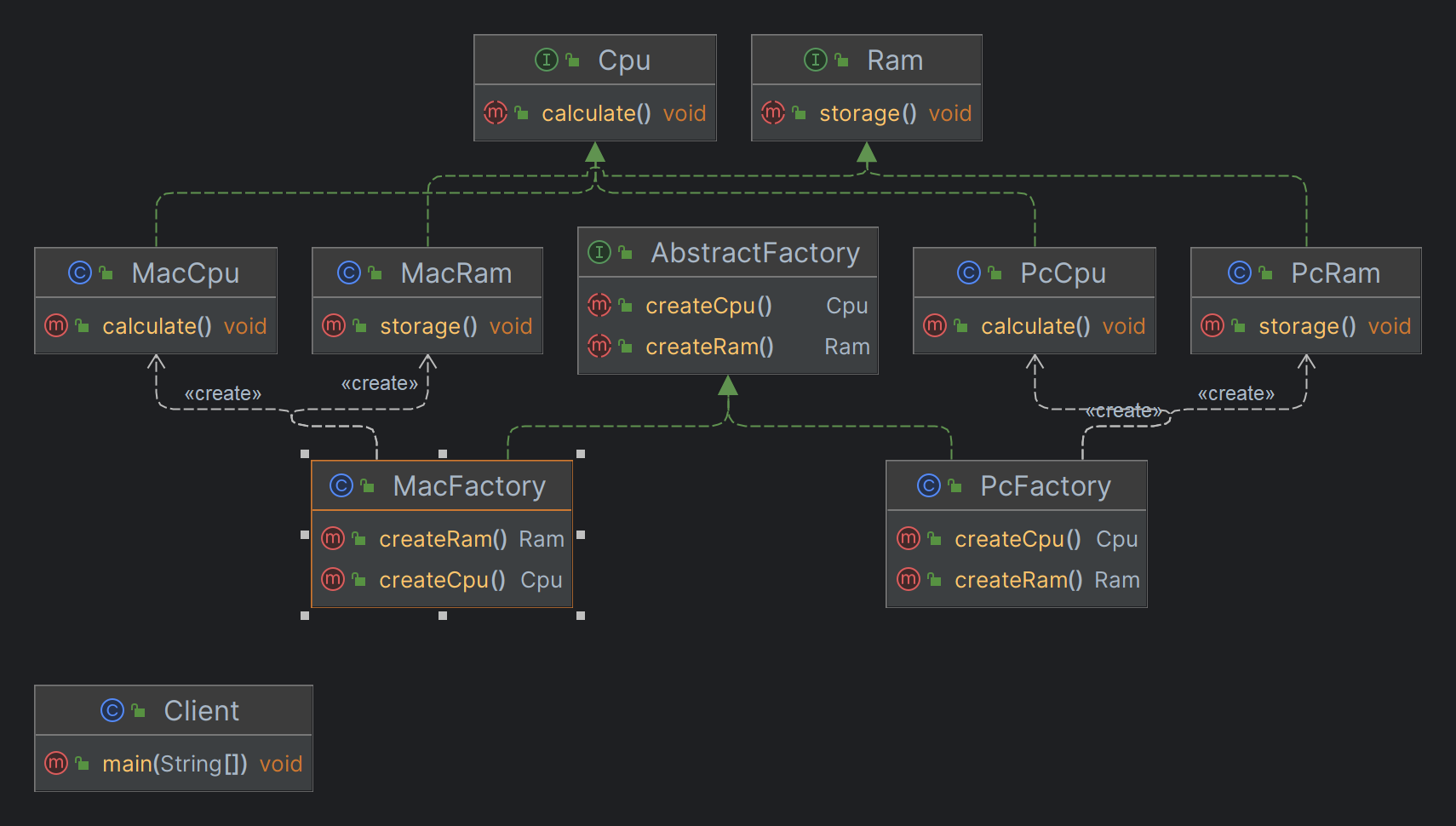
#### 案例总结：

抽象工厂模式主要适用于：

* 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节
* 系统中有多于一个的产品族，但每次只使用其中某一产品族
* 属于同一个产品族的产品将在一起使用，这一约束必须在系统的设计中体现出来
* 产品等级结构稳定，在设计完成之后不会向系统中增加新的产品等级结构或者删除已有的产品等级结构

#### 结果：

##### 类图



##### 代码实现

* AbstractFactory接口

public interface AbstractFactory {  
 Cpu createCpu();  
 Ram createRam();  
}

* PcFactory类

public class PcFactory implements AbstractFactory{  
 @Override  
 public Cpu createCpu() {  
 System.out.println("PcFactroy创建PC的CPU");  
 return new PcCpu();  
 }  
  
 @Override  
 public Ram createRam() {  
 System.out.println("PcFactroy创建PC的RAM");  
 return new PcRam();  
 }  
}

* MacFactory类

public class MacFactory implements AbstractFactory{  
 @Override  
 public Cpu createCpu() {  
 System.out.println("MacFactory创建Mac的CPU");  
 return new MacCpu();  
 }  
  
 @Override  
 public Ram createRam() {  
 System.out.println("MacFactory创建Mac的RAM");  
 return new MacRam();  
 }  
}

* Cpu接口

public interface Cpu {  
 void calculate();  
}

* MacCpu类

public class MacCpu implements Cpu {  
 @Override  
 public void calculate() {  
 System.out.println("常兆海在准备MacCpu calculate");  
 }  
}

* PcCpu 类

public class PcCpu implements Cpu {  
 @Override  
 public void calculate() {  
 System.out.println("pc cpu calculate");  
 }  
  
}

* Ram接口

public interface Ram {  
 void storage();  
}

* MacRam 类

public class MacRam implements Ram {  
 @Override  
 public void storage() {  
 System.out.println("准备MacRam storage");  
 }  
}

* PcRam 类

public class PcRam implements Ram {  
 @Override  
 public void storage() {  
 System.out.println(" Pc Ram storage");  
 }  
}

* XmlUtil工具类

public class XmlUtil {  
 public static Object getbBean(){  
 try {  
 DocumentBuilderFactory documentBuilderFactory= DocumentBuilderFactory.newInstance();  
 DocumentBuilder builder=documentBuilderFactory.newDocumentBuilder();  
 Document document;  
 document= builder.parse(new File("src/main/java/实验一一/config.xml"));  
 NodeList nodeList=null;  
 Node classNode=null;  
 String cNamebefore= "实验一一.";  
 String cName=null;  
 nodeList=document.getElementsByTagName("className");  
 classNode=nodeList.item(0).getFirstChild();  
 cName=classNode.getNodeValue();  
 Class c=Class.forName(cNamebefore+cName);  
 Object onj=c.newInstance();  
 return onj;  
 } catch (Exception e) {  
 e.printStackTrace();  
 return null;  
 }  
 }  
 }

* config.xml

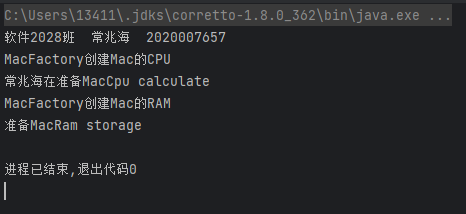
<?xml version="1.0" encoding="UTF-8"?>  
<config>  
 <className>MacFactory</className>  
</config>

* Client类

public class Client {  
 public static void main(String[] args) {  
 AbstractFactory factory;  
 Cpu cpu;  
 Ram ram;  
 factory = (AbstractFactory) XmlUtil.getbBean();  
 cpu = factory.createCpu();  
 cpu.calculate();  
 ram = factory.createRam();  
 ram.storage();  
 }  
}

##### 运行结果

软件2028班 常兆海 2020007657  
MacFactory创建Mac的CPU  
常兆海在准备MacCpu calculate  
MacFactory创建Mac的RAM  
准备MacRam storage  
  
进程已结束,退出代码0



### 建造者模式的运用

#### 案例背景：

计算机组装工厂可以将 CPU，内存，硬盘，主机，显示器等硬件设备组装在一起构成一台完整的计算 机，且构成的计算机可以是笔记本电脑，也可以是台式机，还可以是不提供显示器的服务器主机。对于用 户来言，无需关心计算机的组成设备和组装过程，工厂返回给用户的是完整的计算机对象。所以我们可以 使用建造者模式来实现计算机的组成过程，请绘制出类图并编程实现。

#### 实现步骤：

* 根据题意，使用建造者模式并画出类图。类图中应包含抽象建造者类 ComputerBuilder，复合产品 类 Computer，具体建造者类 Notebook，Desktop 和 Server，其中台式机和服务器主机使用相同的 CPU，内存，硬盘和主机，但是服务器不包含显示器，而笔记本使用自己独自的一套硬件设备。 此外还需要指挥者类 ComputerAssembleDirector，此类中应有将硬件设备组合在一起的建造方法 assemble()并返回用户需要的具体计算机
* 根据类图，实现上述类的具体代码以及用户类 Client 和辅助类 XMLUtil 以实现通过 XML 文件来 制造不同的计算机
* 更改 XML 中的属性，观察用户类是否能够获取到不同的计算机以及这些计算机的组装是否符合 要求

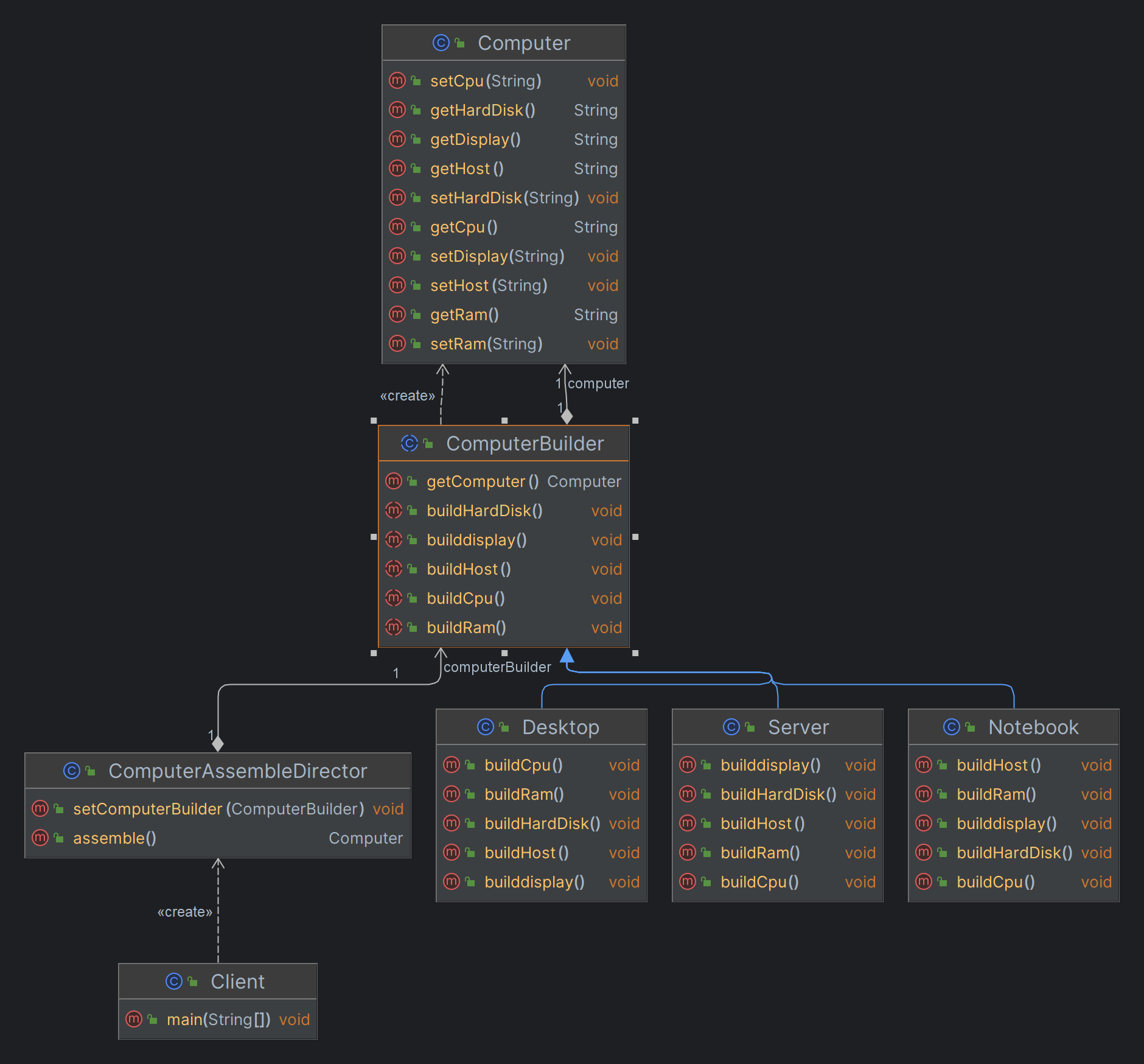
#### 案例总结：

建造者模式主要运用于：

* 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变量
* 需要生成的产品对象的属性相互依赖，需要指定其生成顺序
* 对象的创建过程独立于创建该对象的类。在建造者模式中通过引入了指挥者类，将创建过程封装 在指挥者类中，而不在建造者类和客户类中
* 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

#### 结果：

##### 类图



##### 代码实现

* Computer类

public class Computer {  
 private String cpu;  
 private String ram;  
 private String hardDisk;  
 private String host;  
 public String getDisplay() {  
 return display;  
 }  
 public void setDisplay(String display) {  
 this.display = display;  
 }  
 private String display;  
 public String getHost() {  
 return host;  
 }  
 public void setHost(String host) {  
 this.host = host;  
 }  
 public String getCpu() {  
 return cpu;  
 }  
 public void setCpu(String cpu) {  
 this.cpu = cpu;  
 }  
 public String getRam() {  
 return ram;  
 }  
 public void setRam(String ram) {  
 this.ram = ram;  
 }  
 public String getHardDisk() {  
 return hardDisk;  
 }  
 public void setHardDisk(String hardDisk) {  
 this.hardDisk = hardDisk;  
 }  
}

* ComputerBuilder抽象类

public abstract class ComputerBuilder {  
 protected Computer computer = new Computer();  
 public abstract void buildCpu();  
 public abstract void buildRam();  
 public abstract void buildHardDisk();  
 public abstract void buildHost();  
 public abstract void builddisplay() ;  
 public Computer getComputer() {  
 return computer;  
 }  
}

* ComputerAssembleDirector类

public class ComputerAssembleDirector {  
 private ComputerBuilder computerBuilder;  
 public ComputerAssembleDirector(ComputerBuilder computerBuilder) {  
 this.computerBuilder = computerBuilder;  
 }  
 public void setComputerBuilder(ComputerBuilder computerBuilder) {  
 this.computerBuilder = computerBuilder;  
 }  
 public Computer assemble(){  
 computerBuilder.buildCpu();  
 computerBuilder.buildRam();  
 computerBuilder.buildHardDisk();  
 computerBuilder.buildHost();  
 computerBuilder.builddisplay();  
 System.out.println("软件2028 常兆海 2020007657");  
 return computerBuilder.getComputer();  
 }  
}

* Desktop类

public class Desktop extends ComputerBuilder{  
 @Override  
 public void buildCpu() {  
 computer.setCpu("CPU");  
 }  
 @Override  
 public void buildRam() {  
 computer.setRam("RAM");  
 }  
 @Override  
 public void buildHardDisk() {  
 computer.setHardDisk("HardDisk");  
 }  
 @Override  
 public void buildHost() {  
 computer.setHost("Host");  
 }  
 @Override  
 public void builddisplay() {  
 computer.setDisplay("显示器");  
 }  
}

* Notebook类

public class Notebook extends ComputerBuilder {  
 @Override  
 public void buildCpu() {  
 computer.setCpu("Notebook CPU");  
 }  
 @Override  
 public void buildRam() {  
 computer.setRam("Notebook RAM");  
 }  
 @Override  
 public void buildHardDisk() {  
 computer.setHardDisk("Notebook HardDisk");  
 }  
 @Override  
 public void buildHost() {  
 computer.setHost("Notebook Host");  
 }  
 @Override  
 public void builddisplay() {  
 computer.setDisplay("Notebook 显示器");  
 }  
}

* Server 类

public class Server extends ComputerBuilder {  
 @Override  
 public void buildCpu() {  
 computer.setCpu("CPU");  
 }  
 @Override  
 public void buildRam() {  
 computer.setRam("RAM");  
 }  
 @Override  
 public void buildHardDisk() {  
 computer.setHardDisk("HardDisk");  
 }  
 @Override  
 public void buildHost() {  
 computer.setHost("Host");  
 }  
 @Override  
 public void builddisplay() {//空方法 返回为空  
 }  
}

* config.xml

<?xml version="1.0" encoding="UTF-8"?>  
<config>  
 <className>Desktop</className>  
</config>

* 工具类

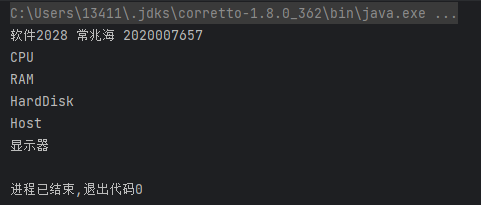
public class XmlUtil {   
   
   
 public static Object getbBean(){   
 try {   
 DocumentBuilderFactory documentBuilderFactory= DocumentBuilderFactory.newInstance();   
 DocumentBuilder builder=documentBuilderFactory.newDocumentBuilder();   
 Document document;   
 document= builder.parse(new File("src/main/java/实验一二/config.xml"));   
 NodeList nodeList=null;   
 Node classNode=null;   
 String cNamebefore= "实验一二.";   
 String cName=null;   
 nodeList=document.getElementsByTagName("className");   
 classNode=nodeList.item(0).getFirstChild();   
 cName=classNode.getNodeValue();   
 Class c=Class.forName(cNamebefore+cName);   
 Object onj=c.newInstance();   
 return onj;   
 } catch (Exception e) {   
 e.printStackTrace();   
 return null;   
 }   
 }   
 }

* 客户端类

public class Client {   
 public static void main(String[] args) {   
 ComputerBuilder builder= (ComputerBuilder) XmlUtil.getbBean();   
ComputerAssembleDirector director=new ComputerAssembleDirector(builder);   
 Computer computer=director.assemble();   
 System.out.println(computer.getCpu());   
 System.out.println(computer.getRam());   
 System.out.println(computer.getHardDisk());   
 System.out.println(computer.getHost());   
 System.out.println(computer.getDisplay()   
 );   
 }   
}

##### 运行结果

软件2028 常兆海 2020007657   
CPU   
RAM   
HardDisk   
Host   
显示器



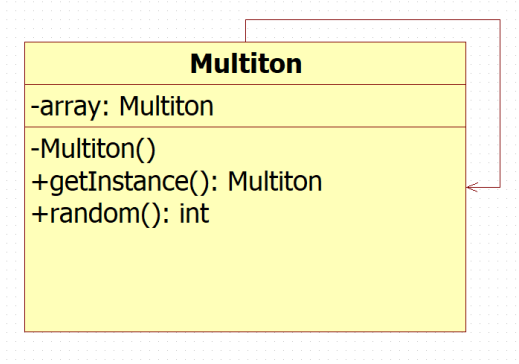
### 单例模式的运用

#### 案例背景：

在实际的运用中，我们有时一个类不止需要产生一个对象，可能需要两个或者三个。在课上我们讲过， 使用单例模式的思想可以实现多例模式，从而确保系统中某个类的对象只能存在有限个，请设计并实现代 码，从而实现多例模式

#### 实现步骤：

* 由于本题的实现较为复杂，所以我们直接给出参考类图
* 根据类图，实现多例模式的代码



#### 案例总结：

单例模式主要适用于：

* 系统只需要一个实例对象，或者因为资源消耗太大而只允许创建一个对象
* 客户调用类的单个实例只允许使用一个公共访问点，除了该公共访问点，不能通过其他途径访问该实例
* 根据不同的需求，也可以按照单例模式的思想来实现多例模式

#### 结果：

##### 代码实现

* Multiton类

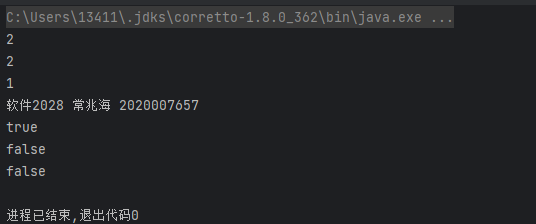
public class Multiton {   
 private volatile static Multiton multiton1=null;   
 private volatile static Multiton multiton2=null;   
private Multiton() {}   
 public static synchronized int random() {   
 //随机数的值为1-2   
 int random = (int) (Math.random() \*(2-1+1) + 1);//Math.random()\*(max-min+1)+min   
 System.out.println(random);   
 return random;   
}   
 public static Multiton getInstance(int random){ //不能乱改名字 这个是特定方法名   
 if (random==1){   
 if (multiton1==null)   
 synchronized (Multiton.class) {   
 if (multiton1==null)   
 multiton1 = new Multiton();   
 }   
 return multiton1;   
 }   
 else {   
 if (multiton2==null)   
 synchronized (Multiton.class) {   
 if (multiton2==null)   
 multiton2 = new Multiton();   
 }   
 return multiton2;   
 }   
 }   
}

* 客户端类

public class Client {   
 public static void main(String[] args) {   
   
 Multiton multiton1 = Multiton.getInstance( Multiton.random());   
 Multiton multiton2 = Multiton.getInstance( Multiton.random());   
 Multiton multiton3 = Multiton.getInstance( Multiton.random());   
 System.out.println("软件2028 常兆海 2020007657");   
 System.out.println(multiton1==multiton2);   
 System.out.println(multiton1==multiton3);   
 System.out.println(multiton2==multiton3);   
 }   
}

##### 运行结果

2   
2   
1   
软件2028 常兆海 2020007657   
true   
false   
false



## 心得体会

设计模式是前人结合实践总结出来的一套行之有效的理论，能够将实际业务需求转换为技术实现时，使系统更具有维护性、扩展性。在学习设计模式时，我不禁赞叹它巧妙的思路和优雅的方式。通过本此实验的学习，更好的理解了建造型设计模式，将理论学习中出现的不足在本次实践中得到了很好的补充。