

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： 创建型模式

实验、实践地点： 行勉A105

专业班级： 软件2121班 学号： 2020005731

学生姓名： 张帅

指导教师： 曹若琛

2023年 3 月 21 日

目录

[一、实验目的和要求 3](#_Toc13905)

[二、实验平台 3](#_Toc15439)

[三、主要实验内容及结果 3](#_Toc17723)

[1. 抽象工厂模式的运用 3](#_Toc24204)

[1.1 实验内容 4](#_Toc10696)

[1.2 代码实现 4](#_Toc10300)

[1.3 运行结果 7](#_Toc23386)

[2.建造者模式的运用 8](#_Toc2495)

[2.1 实验内容 8](#_Toc28098)

[2.2 代码实现 9](#_Toc24783)

[2.3 运行结果 14](#_Toc9303)

[3. 单例模式的运用 14](#_Toc21987)

[3.1 实验内容 14](#_Toc5702)

[3.2 代码实现 15](#_Toc14780)

[3.3 运行结果 16](#_Toc13928)

[四、心得体会 16](#_Toc3179)

**一、实验目的和要求**

（1）实验目的

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。

（2）实验要求

1．独立完成实验。

2．书写实验报告书。

**二、实验平台**

使用JDK-11.0.1，IntelliJ IDEA 2020.3.1x64，笔记本电脑。

**三、主要实验内容及结果**

1. **抽象工厂模式的运用**

**1.1 实验内容**

（1）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）

（2）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。

（3）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。

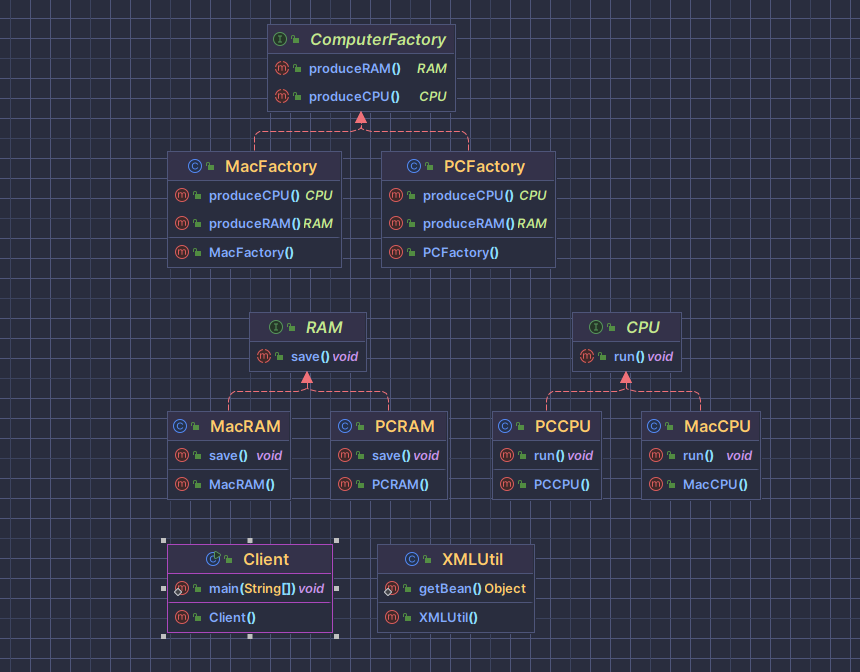
 系统中有多于一个的产品族，但每次只使用其中某一产品族。

 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。

**1.2 代码实现**

绘制类图：



**抽象产品类CPU：**

package edu.zhshio.abstractfactory.lab.support;/\*\*

/\*\*

\* @description:

\* @author: zs

\* @time: 2024/3/20 20:31

\*/

public interface CPU {

void run();

}

**抽象产品类 RAM：**

package edu.zhshio.abstractfactory.lab.support;

/\*\*

\* @description:

\* @author: zs

\* @time: 2024/3/20 20:31

\*/

public interface RAM {

void save();

}

**具体产品类 MacCPU：**

package edu.zhshio.abstractfactory.lab.products;

import edu.zhshio.abstractfactory.lab.support.CPU;

/\*\*

\* @description:

\* @author: zs

\* @time: 2024/3/20 20:34

\*/

public class MacCPU implements CPU {

@Override

public void run() {

System.out.println("Mac cpu is running");

}

}

**具体产品类 PCCPU：**

package edu.zhshio.abstractfactory.lab.products;/

import edu.zhshio.abstractfactory.lab.support.CPU;

/\*\*

\* @description:

\* @author: zs

\* @time: 2024/3/20 20:33

\*/

public class PCCPU implements CPU {

@Override

public void run() {

System.out.println("PC CPU is running...");

}

}

**具体产品类 MacRAM：**

package edu.zhshio.abstractfactory.lab.products;

import edu.zhshio.abstractfactory.lab.support.RAM;

/\*\*

\* @description:

\* @author: zs

\* @time: 2024/3/20 20:35

\*/

public class MacRAM implements RAM {

@Override

public void save() {

System.out.println("Mac RAM is saving");

}

}

**具体产品类 PCRAM：**

package edu.zhshio.abstractfactory.lab.products;

import edu.zhshio.abstractfactory.lab.support.RAM;

/\*\*

\* @description:

\* @author: zs

\* @time: 2024/3/20 20:34

\*/

public class PCRAM implements RAM {

@Override

public void save() {

System.out.println("PC RAM is saving......");

}

}

**抽象工厂类 ComputerFactory:**

package edu.zhshio.abstractfactory.lab.support;

/\*\*

\* @description:

\* @author: zs

\* @time: 2024/3/20 20:32

\*/

public interface ComputerFactory {

CPU produceCPU();

RAM produceRAM();

}

**具体工厂类 MacFactory：**

package edu.zhshio.abstractfactory.lab.facotry;

import edu.zhshio.abstractfactory.lab.products.MacCPU;

import edu.zhshio.abstractfactory.lab.products.MacRAM;

import edu.zhshio.abstractfactory.lab.support.CPU;

import edu.zhshio.abstractfactory.lab.support.ComputerFactory;

import edu.zhshio.abstractfactory.lab.support.RAM;

/\*\*

\* @description:

\* @author: zs

\* @time: 2024/3/20 20:36

\*/

public class MacFactory implements ComputerFactory {

@Override

public CPU produceCPU() {

return new MacCPU();

}

@Override

public RAM produceRAM() {

return new MacRAM();

}

}

**具体工厂类 PCFactory：**

package edu.zhshio.abstractfactory.lab.facotry;

import edu.zhshio.abstractfactory.lab.products.PCCPU;

import edu.zhshio.abstractfactory.lab.products.PCRAM;

import edu.zhshio.abstractfactory.lab.support.CPU;

import edu.zhshio.abstractfactory.lab.support.ComputerFactory;

import edu.zhshio.abstractfactory.lab.support.RAM;

/\*\*

\* @description:

\* @author: zs

\* @time: 2024/3/20 20:36

\*/

public class PCFactory implements ComputerFactory {

@Override

public CPU produceCPU() {

return new PCCPU();

}

@Override

public RAM produceRAM() {

return new PCRAM();

}

}

**辅助文件 XMLUtil：**

package edu.zhshio.abstractfactory.lab.util;

import org.w3c.dom.Document;

import org.w3c.dom.Node;

import org.w3c.dom.NodeList;

import org.xml.sax.SAXException;

import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.DocumentBuilderFactory;

import javax.xml.parsers.ParserConfigurationException;

import java.io.File;

import java.io.IOException;

/\*\*

\* @description:

\* @author: zs

\* @time: 2024/3/7 11:15

\*/

public class XMLUtil {

public static Object getBean() {

DocumentBuilderFactory documentBuilderFactory = DocumentBuilderFactory.newInstance();

try {

DocumentBuilder documentBuilder = documentBuilderFactory.newDocumentBuilder();

Document document = documentBuilder.parse(new File("src/main/java/edu/zhshio/abstractfactory/lab/resource/beanconfig.xml"));

NodeList mealBuilderNames = document.getElementsByTagName("beanName");

Node mealBuilderName = mealBuilderNames.item(0).getFirstChild();

String mealBuilerNameVal = mealBuilderName.getNodeValue().trim();

Class<?> aClass = Class.forName(mealBuilerNameVal);

return aClass.newInstance();

} catch (ParserConfigurationException e) {

throw new RuntimeException(e);

} catch (IOException e) {

throw new RuntimeException(e);

} catch (SAXException e) {

throw new RuntimeException(e);

} catch (ClassNotFoundException e) {

throw new RuntimeException(e);

} catch (InstantiationException e) {

throw new RuntimeException(e);

} catch (IllegalAccessException e) {

throw new RuntimeException(e);

}

}

}

**配置文件 beanconfig.xml:**

<?xml version="1.0"?>

<config>

<beanName>edu.zhshio.abstractfactory.lab.facotry.PCFactory</beanName>

</config>

**用户测试类 Client:**

package edu.zhshio.abstractfactory.lab;  
  
import edu.zhshio.abstractfactory.lab.support.CPU;  
import edu.zhshio.abstractfactory.lab.support.ComputerFactory;  
import edu.zhshio.abstractfactory.lab.support.RAM;  
import edu.zhshio.abstractfactory.lab.util.XMLUtil;  
  
/\*\*  
 \* @description:  
 \* @author: zs  
 \* @time: 2024/4/14 下午8:09  
 \*/  
  
public class Client {  
 public static void main(String[] args) {  
 ComputerFactory computerFactory = (ComputerFactory) XMLUtil.getBean();  
 CPU cpu = computerFactory.produceCPU();  
 RAM ram = computerFactory.produceRAM();  
  
 cpu.run();  
 ram.save();  
 }  
}

**1.3 运行结果**



**2.建造者模式的运用**

**2.1 实验内容**

（1）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。

（2）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。

（3）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。

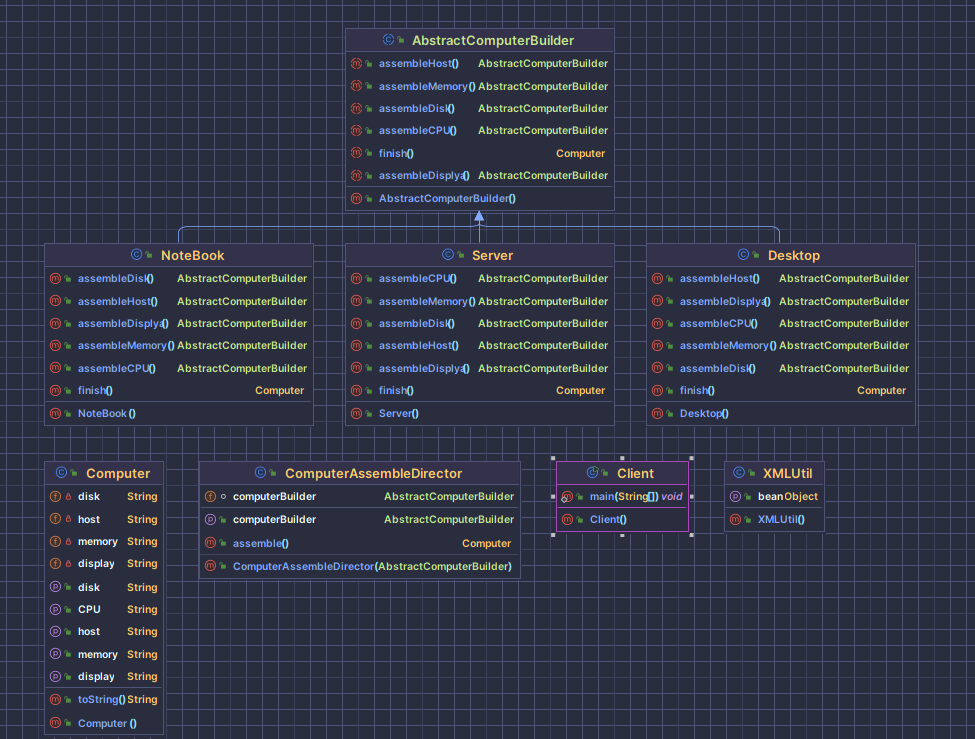
 需要生成的产品对象的属性相互依赖，需要指定其生成顺序。

 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。

**2.2 代码实现**

绘制类图：



**复合产品类 Computer：**

package edu.zhshio.builder.lab.support;/\*\*

\* @description:

\* @author: zs

\* @time: 2024/3/7 11:31

\*/

public class Computer {

private String cpu;

private String memory;

private String disk;

private String host;

private String display;

public String getCPU() {

return cpu;

}

public void setCPU(String cpu) {

this.cpu = cpu;

}

public String getMemory() {

return memory;

}

public void setMemory(String memory) {

this.memory = memory;

}

public String getDisk() {

return disk;

}

public void setDisk(String disk) {

this.disk = disk;

}

public String getHost() {

return host;

}

public void setHost(String host) {

this.host = host;

}

public String getDisplay() {

return display;

}

public void setDisplay(String display) {

this.display = display;

}

@Override

public String toString() {

return "Computer{" +

"cpu='" + cpu + '\'' +

", memory='" + memory + '\'' +

", disk='" + disk + '\'' +

", host='" + host + '\'' +

", display='" + display + '\'' +

'}';

}

}

**抽象建造者类** AbstractComputerBuilder**：**

package edu.zhshio.builder.lab.support;

/\*\*

\* @description:

\* @author: zs

\* @time: 2024/3/7 11:31

\*/

public abstract class AbstractComputerBuilder {

protected Computer computer = new Computer();

public abstract AbstractComputerBuilder assembleCPU();

public abstract AbstractComputerBuilder assembleMemory();

public abstract AbstractComputerBuilder assembleDisk();

public abstract AbstractComputerBuilder assembleHost();

public abstract AbstractComputerBuilder assembleDisplya();

public abstract Computer finish();

}

**具体建造者类 Notebook：**

package edu.zhshio.builder.lab.conceretebuilder;

import edu.zhshio.builder.lab.support.AbstractComputerBuilder;

import edu.zhshio.builder.lab.support.Computer;

/\*\*

\* @description:

\* @author: zs

\* @time: 2024/3/7 11:37

\*/

public class NoteBook extends AbstractComputerBuilder{

String concereteCategory = "笔记本";

@Override

public AbstractComputerBuilder assembleCPU() {

this.computer.setCPU(concereteCategory + "CPU");

return this;

}

@Override

public AbstractComputerBuilder assembleMemory() {

this.computer.setMemory(concereteCategory + "内存");

return this;

}

@Override

public AbstractComputerBuilder assembleDisk() {

this.computer.setDisk(concereteCategory + "硬盘");

return this;

}

@Override

public AbstractComputerBuilder assembleHost() {

this.computer.setHost(concereteCategory + "主机箱");

return this;

}

@Override

public AbstractComputerBuilder assembleDisplya() {

this.computer.setDisplay(concereteCategory + "显示器");

return this;

}

@Override

public Computer finish() {

return this.computer;

}

}

**具体建造者类 Desktop：**

package edu.zhshio.builder.lab.conceretebuilder;  
import edu.zhshio.builder.lab.support.AbstractComputerBuilder;  
import edu.zhshio.builder.lab.support.Computer;  
  
/\*\*  
 \* @description:   
 \* @author: zs  
 \* @time: 2024/3/7 11:35  
 \*/  
   
public class Desktop extends AbstractComputerBuilder {  
  
 String concereteCategory = "台式机";  
  
 @Override  
 public AbstractComputerBuilder assembleCPU() {  
 this.computer.setCPU(concereteCategory + "CPU");  
 return this;  
 }  
  
 @Override  
 public AbstractComputerBuilder assembleMemory() {  
 this.computer.setMemory(concereteCategory + "内存");  
 return this;  
 }  
  
 @Override  
 public AbstractComputerBuilder assembleDisk() {  
 this.computer.setDisk(concereteCategory + "硬盘");  
 return this;  
 }  
  
 @Override  
 public AbstractComputerBuilder assembleHost() {  
 this.computer.setHost(concereteCategory + "主机箱");  
 return this;  
 }  
  
 @Override  
 public AbstractComputerBuilder assembleDisplya() {  
 this.computer.setDisplay(concereteCategory + "显示器");  
 return this;  
 }  
  
 @Override  
 public Computer finish() {  
 return this.computer;  
 }  
}

**具体建造者类 Server：**

package edu.zhshio.builder.lab.conceretebuilder;  
  
import edu.zhshio.builder.lab.support.AbstractComputerBuilder;  
import edu.zhshio.builder.lab.support.Computer;  
  
/\*\*  
 \* @description:  
 \* @author: zs  
 \* @time: 2024/4/10 下午8:02  
 \*/  
  
public class Server extends AbstractComputerBuilder {  
  
 String concereteCategory = "服务器";  
  
 @Override  
 public AbstractComputerBuilder assembleCPU() {  
 this.computer.setCPU(concereteCategory + "CPU");  
 return this;  
 }  
  
 @Override  
 public AbstractComputerBuilder assembleMemory() {  
 this.computer.setMemory(concereteCategory + "内存");  
 return this;  
 }  
  
 @Override  
 public AbstractComputerBuilder assembleDisk() {  
 this.computer.setDisk(concereteCategory + "硬盘");  
 return this;  
 }  
  
 @Override  
 public AbstractComputerBuilder assembleHost() {  
 this.computer.setHost(concereteCategory + "主机箱");  
 return this;  
 }  
  
 @Override  
 public AbstractComputerBuilder assembleDisplya() {  
 this.computer.setDisplay(concereteCategory + "显示器");  
 return this;  
 }  
  
 @Override  
 public Computer finish() {  
 return this.computer;  
 }  
}

**指挥者类 ComputerAssembleDirector：**

package edu.zhshio.builder.lab.director;  
  
import edu.zhshio.builder.lab.support.Computer;  
import edu.zhshio.builder.lab.support.AbstractComputerBuilder;  
  
/\*\*  
 \* @description:   
 \* @author: zs  
 \* @time: 2024/3/7 11:38  
 \*/  
   
public class ComputerAssembleDirector {  
  
 AbstractComputerBuilder computerBuilder;  
  
 public Computer assemble() {  
 return computerBuilder  
 .assembleCPU()  
 .assembleMemory()  
 .assembleDisk()  
 .assembleHost()  
 .assembleDisplya()  
 .finish();  
 }  
  
 public ComputerAssembleDirector(AbstractComputerBuilder computerBuilder) {  
 this.computerBuilder = computerBuilder;  
 }  
  
 public void setComputerBuilder(AbstractComputerBuilder computerBuilder) {  
 this.computerBuilder = computerBuilder;  
 }  
}

**辅助类 XMLUtil：**

package edu.zhshio.builder.lab.util;  
  
import org.w3c.dom.Document;  
import org.w3c.dom.Node;  
import org.w3c.dom.NodeList;  
import org.xml.sax.SAXException;  
  
import javax.xml.parsers.DocumentBuilder;  
import javax.xml.parsers.DocumentBuilderFactory;  
import javax.xml.parsers.ParserConfigurationException;  
import java.io.File;  
import java.io.IOException;  
  
/\*\*  
 \* @description:  
 \* @author: zs  
 \* @time: 2024/3/7 11:15  
 \*/  
  
public class XMLUtil {  
  
 public static Object getBean() {  
  
 DocumentBuilderFactory documentBuilderFactory = DocumentBuilderFactory.newDefaultInstance();  
 try {  
 DocumentBuilder documentBuilder = documentBuilderFactory.newDocumentBuilder();  
 Document document = documentBuilder.parse(new File("src/main/java/edu/zhshio/builder/lab/resource/beanconfig.xml"));  
  
 NodeList mealBuilderNames = document.getElementsByTagName("beanName");  
 Node mealBuilderName = mealBuilderNames.item(0).getFirstChild();  
 String mealBuilerNameVal = mealBuilderName.getNodeValue().trim();  
  
 Class<?> aClass = Class.forName(mealBuilerNameVal);  
 return aClass.newInstance();  
  
 } catch (ParserConfigurationException e) {  
 throw new RuntimeException(e);  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 } catch (SAXException e) {  
 throw new RuntimeException(e);  
 } catch (ClassNotFoundException e) {  
 throw new RuntimeException(e);  
 } catch (InstantiationException e) {  
 throw new RuntimeException(e);  
 } catch (IllegalAccessException e) {  
 throw new RuntimeException(e);  
 }  
 }  
}

**配置文件 beanconfig.xml:**

<?xml version="1.0"?>  
<config>  
 <beanName>edu.zhshio.builder.lab.conceretebuilder.NoteBook</beanName>  
</config>

**用户测试类 Client：**

package edu.zhshio.builder.lab;  
import edu.zhshio.builder.lab.director.ComputerAssembleDirector;  
import edu.zhshio.builder.lab.support.AbstractComputerBuilder;  
import edu.zhshio.builder.lab.support.Computer;  
import edu.zhshio.builder.lab.util.XMLUtil;  
  
/\*\*  
 \* @description:   
 \* @author: zs  
 \* @time: 2024/3/7 11:41  
 \*/  
   
public class Client {  
 public static void main(String[] args) {  
 AbstractComputerBuilder computerBuilder = (AbstractComputerBuilder) XMLUtil.getBean();  
 Computer computer = new ComputerAssembleDirector(computerBuilder).assemble();  
 System.out.println(computer);  
 }  
}

**2.3 运行结果**



**3. 单例模式的运用**

**3.1 实验内容**

（1）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。

（2）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（见下一页）。

 根据类图，实现多例模式的代码。

（3）案例总结：

单例模式主要适用于：

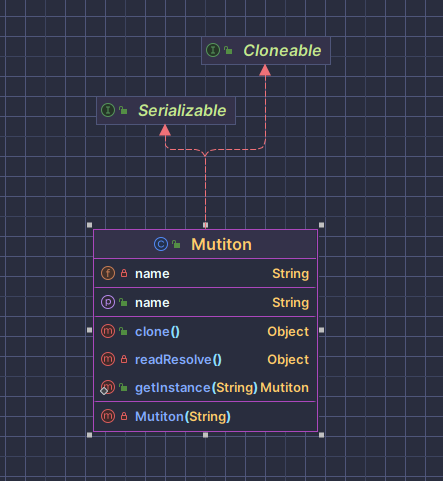
 系统只需要一个实例对象，或者因为资源消耗太大而只允许创建一个对象。

 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。

**3.2 代码实现**

绘制类图：



**多例类 Multiton:**

package edu.zhshio.singleton.lab.multiton;  
  
import java.io.ObjectStreamException;  
import java.io.Serializable;  
import java.util.Map;  
import java.util.concurrent.ConcurrentHashMap;  
  
  
  
/\*\*  
 \* @description:  
 \* @author: zs  
 \* @time: 2024/4/10 下午8:24  
 \*/  
  
public class Mutiton implements Cloneable, Serializable {  
  
 // volatile保持可见性, ConcurrentHashMap保证并发下Map内部状态一致性  
 private static volatile Map<String, Mutiton> instanceCache = new ConcurrentHashMap<>();  
 // 观察调用了几次构造方法  
 // private static AtomicInteger count = new AtomicInteger(0);  
 private String name;  
  
 private Mutiton(String name) {  
  
 // 假如instanceCache中不存在实例, 反射可以创建对象, 在创建后仍然会放到instanceCache中, 不会破坏单例模式  
 // 主要是预防反射重复创建对象  
 if (!instanceCache.isEmpty() && instanceCache.containsKey(name)) {  
 // 防止反射破坏单例模式  
 throw new RuntimeException("Mutiton 被设计为单例 不可重复实例化");  
 }  
 // 不会产生死锁, 因为synchronized是可重入的  
 synchronized (Mutiton.class) {  
 if (!instanceCache.isEmpty() && instanceCache.containsKey(name)) {  
 // 双重检查锁, 防止并发场景下反射破坏单例模式  
 throw new RuntimeException("Mutiton 被设计为单例 不可重复实例化");  
 }  
 // 观察调用了几次构造方法  
 // System.out.println(count.addAndGet(1));  
 this.name = name;  
 instanceCache.put(name, this);  
 }  
  
 }  
  
  
 // 为什么重复写两段双重检查锁逻辑? 因为getInstance在发现已存在单例时是返回单例, 而构造方法则是直接抛出异常, 逻辑不可重用  
 public static Mutiton getInstance(String name) {  
 // 双重检查锁, 防止并发场景下破坏单例模式  
 if (instanceCache.isEmpty() || !instanceCache.containsKey(name)) {  
 synchronized (Mutiton.class) {  
 if (instanceCache.isEmpty() || !instanceCache.containsKey(name)) {  
 Mutiton mutiton = new Mutiton(name);  
 }  
 }  
 }  
 return instanceCache.get(name);  
 }  
  
  
  
 // 防止克隆破坏单例模式  
 @Override  
 public Object clone() throws CloneNotSupportedException {  
 throw new CloneNotSupportedException("Mutiton 被设计为单例, 不可以clone");  
 }  
  
 // 防止序列化破坏单例模式  
 private Object readResolve() throws ObjectStreamException {  
 return getInstance(this.name);  
 }  
  
 public String getName() {  
 return this.name;  
 }  
  
  
}

**用户测试类 Client：**

public class Client {  
 public static void main(String[] args) {  
 Mutiton instance1 = Mutiton.getInstance("实例1");  
 Mutiton instance2 = Mutiton.getInstance("实例2");  
 Mutiton instance3 = Mutiton.getInstance("实例3");  
  
 Mutiton instance11 = Mutiton.getInstance("实例1");  
 Mutiton instance22 = Mutiton.getInstance("实例2");  
 Mutiton instance33 = Mutiton.getInstance("实例3");  
  
 System.out.println(instance1 == instance11);  
 System.out.println(instance2 == instance22);  
 System.out.println(instance3 == instance33);  
  
 }  
}

**3.3 运行结果**



**四、心得体会**

本次创建型模式的实验中，整体上参考课本的例题没有太大的问题，就是在引入配置文件的时候，路径总是找不对，最后不断试验解决问题, 通过这次试验，让我对创建型模式有了一个更深刻的理解。

抽象工厂模式为创建一组相关或依赖的对象提供创建接口，它很好地满足了开闭原则，也就是说具体产品在应用层的代码被隔离，无需关心创建的细节，将一系列的产品统一到一起去创建，如果需要增加新的工厂，直接继承抽象工厂类即可。但是缺点也很明显，如果需要增加新的产品就意味着要彻底修改所有的工厂类对象，扩展是很困难的，增加了系统的抽象性和理解难度。

建造者模式中，需要生成的产品对象一般有复杂的内部结构，有许多的部件需要我们去组装，所以一个产品类要包含很多的成员变量，这些成员变量部件最终可组成不同的产品，需创建一个抽象建造者类，具体产品的建造者类再继承至该抽象建造者类，为了与客户端解耦，建造者模式又用了一个指挥者类，指挥者类负责创建抽象建造者对象，构建产品细节，把结果返回给客户端，使得外界看不到具体的操作过程。

用单例模式的思想实现多例模式，依据课本单例模式的例子，单例模式是对对象进行了一个判断，如果对象不存在则创建，如果已经有对象了就还是返回该对象，并不会再次创建一个新的对象，这保证了一个类只有一个实例对象，这里是通过判断某变量值是否为 null 实现的。那么如果换成多例模式，就是说在一个类中只允许存在有限个实例对象，同理，判断的时候让创建对象的次数小于某个值就可以，小于就创建新对象，如果超出设定个数就不会再创建。同时给出私有的无参构造方法，使外部不能通过new来实例化对象，即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。