

软件设计与体系结构实验报告



课 程： 软件设计与体系结构

指导老师： 徐本柱

学生姓名： 刘铭源

学 号： 2018214937

作业提交时间： 2020.11.19

目录

实验 1--创建型设计模式-抽象工厂模式	3
1、实验目的与内容	3
1.1 实验目的	3
1.2 实验内容	3
2、分析设计过程	3
2.1 模式分析	3
2.2 UML 类图	4
3、测试数据和程序运行的结果	5
3.1 运行结果:	5
3.2 整体框架	5
4、问题与总结	6
5、附录	6
5.1 英雄部分	6
5.2 装备部分	7
5.3 工厂部分	8
实验 2--结构型设计模式-适配器模式	9
1、实验目的与内容	9
1.1 实验目的	9
1.2 实验内容	9
2、分析设计过程	9
2.1 模式分析	9
2.2 UML 类图	10
3、测试数据和程序运行的结果	10
3.1 运行结果:	10
3.2 整体框架	11
4.问题与总结	11
5、附录	11
实验 3--行为型设计模式-观察者模式	13
1、实验目的与内容	13
1.1 实验目的	13
1.2 实验内容	13
2、分析设计过程	13
2.1 模式分析	13
2.2 UML 类图	14
3、测试数据和程序运行的结果	14
3.1 运行结果:	14
3.2 整体框架	15
4.问题与总结	15
5、附录	15
5.1 歹徒部分:	15
5.2 警察部分:	16
5.3 主函数部分:	17
实验 4--架构模式-面向对象 (MVC 设计模式为抽象工厂) 模式	18
1、实验目的与内容	18
1.1 实验目的	18
1.2 实验内容	18
2、分析设计过程	18
2.1 模式分析	18
2.2 UML 类图	19
3、测试数据和程序运行的结果	19
3.1 运行结果:	19
3.2 整体框架	20
4、问题与总结	20
5、附录	20
实验 5--架构模式-管道过滤器模式	26
1、实验目的与内容	26
1.1 实验目的	26
1.2 实验内容	26
2、分析设计过程	26
2.1 模式分析	26
2.2 UML 类图	26
3、测试数据和程序运行的结果	27
3.1 运行结果:	27
3.2 整体框架	27
4、问题与总结	27
5、附录	27

实验 1--创建型设计模式-抽象工厂模式

1、实验目的与内容

1.1 实验目的

加深所学理论知识的理解，掌握常见软件体系结构的知识使用方法；掌握抽象工厂设计模式的结构和适用问题，并提高解决实际问题的能力。

掌握抽象工厂模式（Abstract Factory）的特点
分析具体问题，使用抽象工厂模式进行设计。

1.2 实验内容

编程实现一个游戏，有英雄和装备，每个英雄有不同的属性，装备既有攻击型也有防御型。

2、分析设计过程

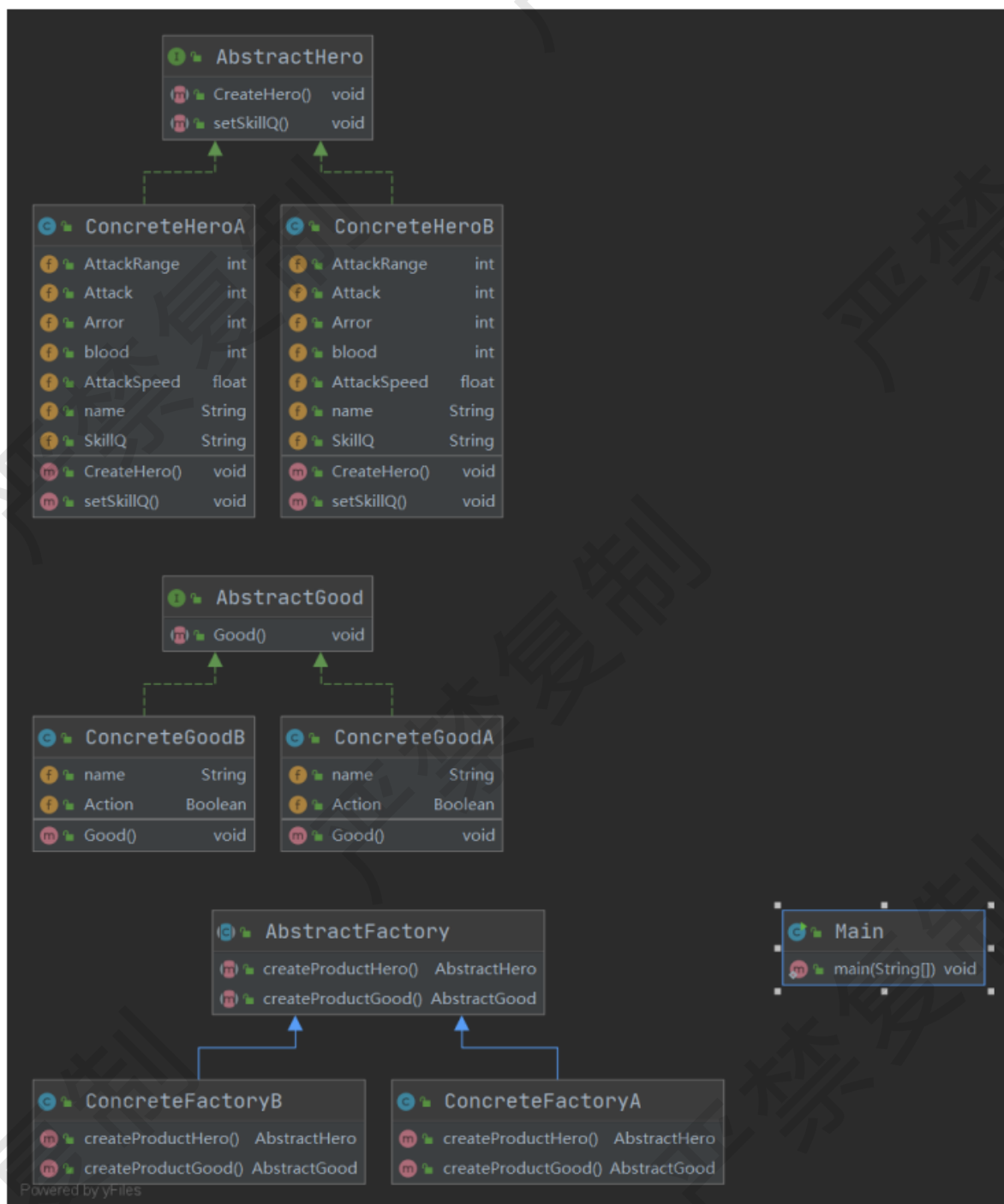
2.1 模式分析

本实验包含两种产品等级结构，英雄和装备，其中大苟类型的英雄和红尘英雄构成一个产品族，尚方宝剑装备和圆盾装备构成一个产品族。

经过分析，我觉得更加适合使用抽象工厂方法。理由如下：

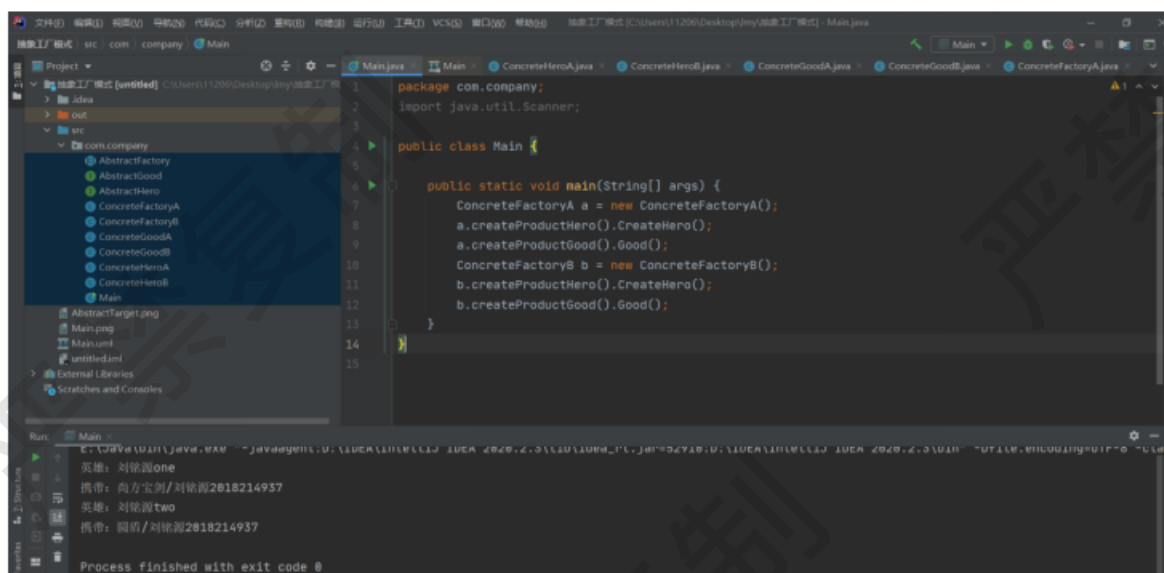
当使用工厂方法模式，你可以替换生成装备的工厂方法，就可以把装备从尚方宝剑换到圆盾。但是用了抽象工厂模式，你只要换家工厂，就可以同时替换英雄和它们所持有的装备。如果你英雄们不知持有一件装备，或者英雄不止一两个，那么使用抽象工厂全部一次替换掉最方便。如果说工厂模式就像一个二维的数，那么抽象工厂模式就是三维的数。工厂模式就像是抽象工厂模式中的一条生产线。

2.2 UML 类图



3、测试数据和程序运行的结果

3.1 运行结果：



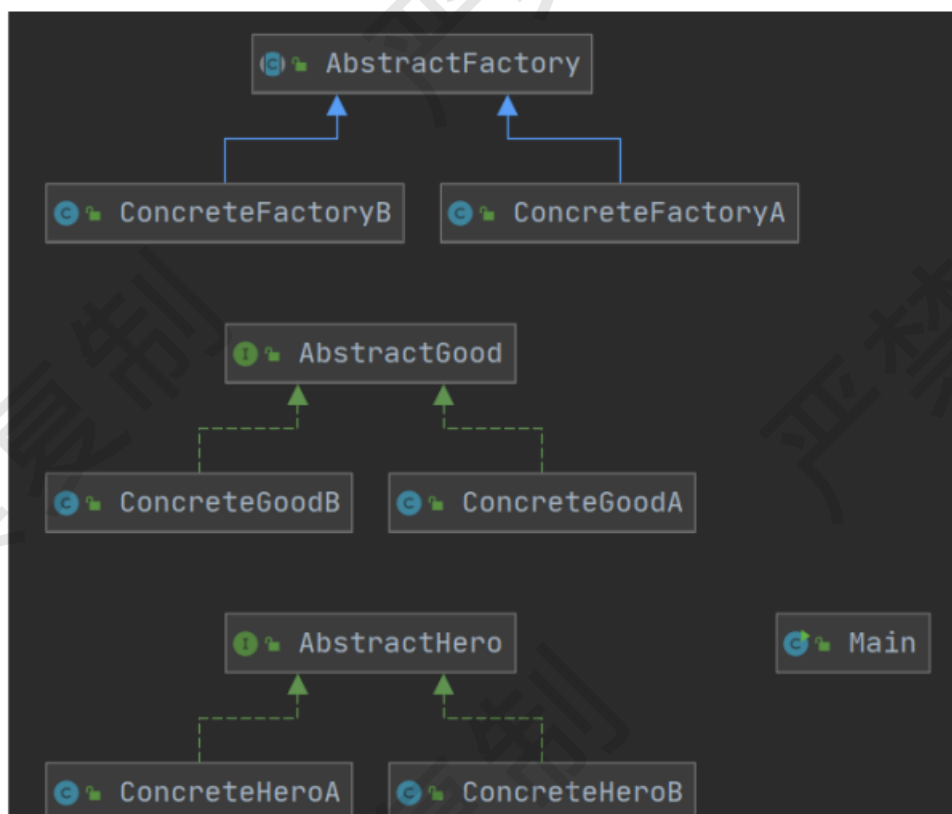
```
package com.company;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        ConcreteFactoryA a = new ConcreteFactoryA();
        a.createProductHero().CreateHero();
        a.createProductGood().Good();
        ConcreteFactoryB b = new ConcreteFactoryB();
        b.createProductHero().CreateHero();
        b.createProductGood().Good();
    }
}
```

Process finished with exit code 0

3.2 整体框架



4、 问题与总结

通过此次实验，我复习抽象工厂模式和 Java 语言，在做实验的过程中，我和同学交流了抽象工厂和工厂的实现，探讨了如何实现抽象工厂的具体实现；在实现工厂中，我一开始将抽象工厂和工厂搞混了，重新编写抽象工厂模式；温习抽象工厂模式，明白抽象工厂的缺点：不能灵活创建对象。

5、 附录

5.1 英雄部分

```
package com.company;

public class ConcreteHeroA implements AbstractHero{
    public int AttackRange;
    public int Attack;
    public int Armor;
    public int blood;
    public float AttackSpeed;

    public String name;
    public String SkillQ;

    public void CreateHero(){
        this.name = "刘铭源 one";
        this.Attack = 1000;
        this.Armor = 1000;
        this.AttackRange = 1000;
        this.AttackSpeed = 1000;
        this.blood = 1000;
        System.out.println("英雄： 刘铭源 one");
    };

    @Override
    public void setSkillQ() {
        this.SkillQ = "增加 50 护甲";
    }
}
```

```

    }
}
package com.company;

public class ConcreteHeroB implements AbstractHero{
    public int AttackRange;
    public int Attack;
    public int Arror;

    public int blood;
    public float AttackSpeed;

    public String name;
    public String SkillQ;

    public void CreateHero(){
        this.name = "刘铭源 two";
        this.Attack = 1000000;
        this.Arror = 1;

        this.AttackRange = 1000;
        this.AttackSpeed = 1000;
        this.blood = 1000;
        System.out.println("英雄: 刘铭源 two");
    };

    @Override
    public void setSkillQ() {
        this.SkillQ = "增加 100000000 攻速";
    }
}

```

5.2 装备部分

```

package com.company;

public class ConcreteGoodB implements AbstractGood{

```



```

    public String name = null;
    public Boolean Action = null;
    public void Good() {
        System.out.println("携带：圆盾/刘铭源 2018214937");
        this.name = "圆盾";
        this.Action = null;
    }
}

package com.company;

public class ConcreteGoodA implements AbstractGood{
    public String name = null;
    public Boolean Action = null;
    @Override
    public void Good() {
        System.out.println("携带：尚方宝剑/刘铭源 2018214937");
        this.name = "尚方宝剑";
        this.Action = true;
    }
}

```

5.3 工厂部分

```

package com.company;

public abstract class AbstractFactory
{
    public abstract AbstractHero createProductHero();
    public abstract AbstractGood createProductGood();
}

package com.company;

public interface AbstractGood {

```



```
void Good();  
}  
package com.company;  
  
public interface AbstractHero {  
  
    public void CreateHero();  
    public void setSkillQ();  
}
```

实验 2--结构型设计模式-适配器模式

1、实验目的与内容

1.1 实验目的

加深所学理论知识的理解，掌握常见软件体系结构的知识使用方法；掌握适配器工厂设计模式的结构和适用问题，并提高解决实际问题的能力。掌握适配器模式（Adapter Pattern）的特点分析具体问题，使用适配器模式进行设计。

1.2 实验内容

编程实现一个程序，使得不能使用家用充电插头的手机可以通过使用电源适配器来进行充电，其中电源适配器的电压为 5v，电源适配器的电压为 220v。

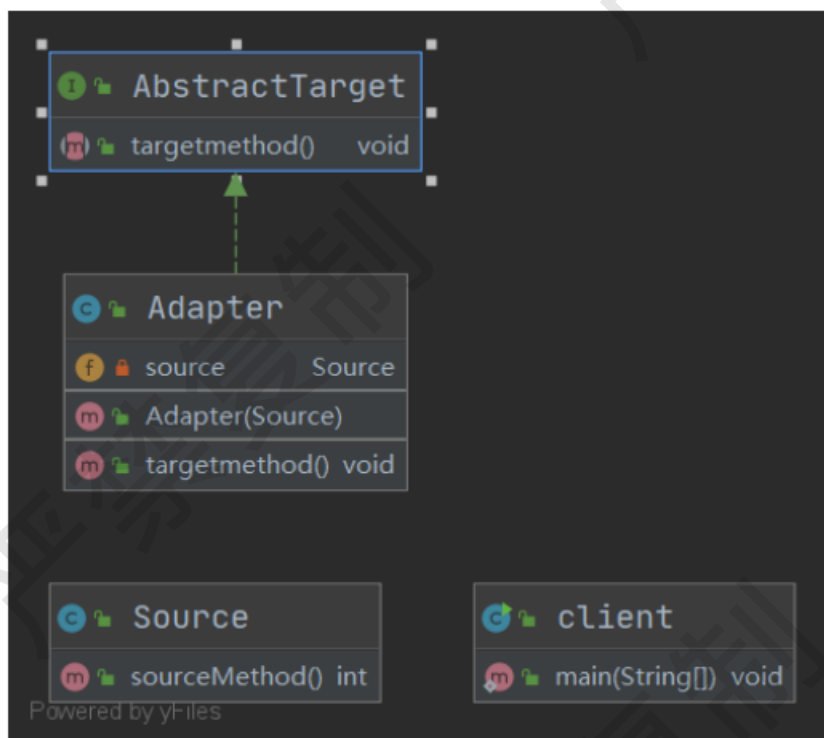
2、分析设计过程

2.1 模式分析

本实验包含三种物品，手机和插头、电源适配器。要使家用插头可以给手机充电，则需要把家用插头中的电源电压改为适配手机的电源电压。

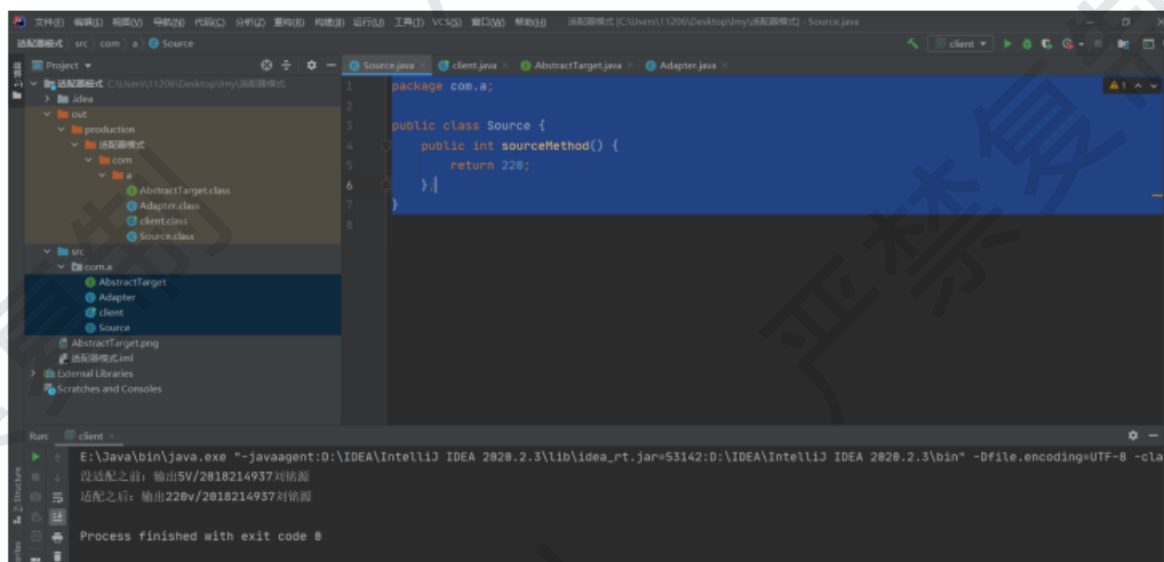
经过分析，我觉得更加适合使用适配器模式方法。理由如下：相比较装饰模式，最重要的是适配器模式只改变接口，不改变功能，在本次实验中只需要改变手机充电器的接口，将家用插口电源改为电源适配器即可，功能并不需要修改且适配器模式比静态继承更灵活，可以避免在层次结构高层的类有太多的特征。

2.2 UML 类图

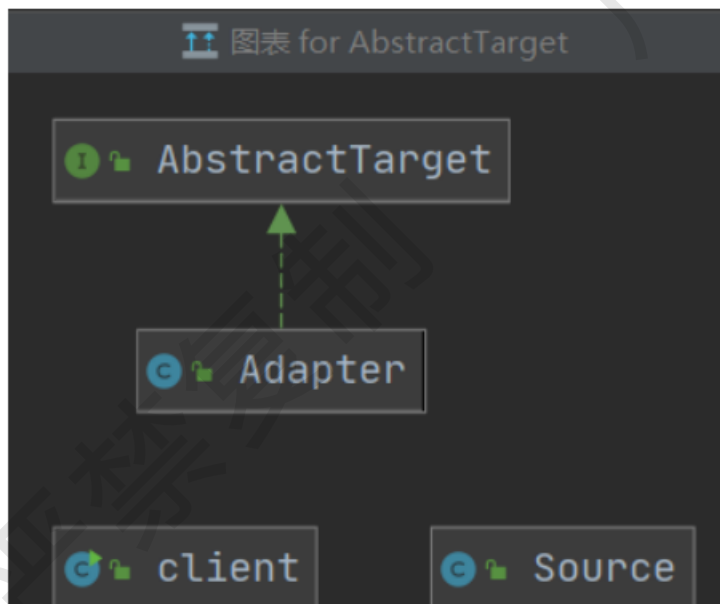


3、测试数据和程序运行的结果

3.1 运行结果：



3.2 整体框架



4.问题与总结

通过此次试验，我进一步了解了适配器模式，将适配器模式进行了实现，适配器模式具有以下特点：增加了类的透明性和复用性，将具体的实现封装在适配者类中，对于客户端类来说是透明的，而且提高了适配者的复用性，符合开闭原则。在实现过程中，我将上课的例子进行实现，和老师所讲例子完整匹配。

5、附录

```
package com.a;

public class Source {
    public int sourceMethod() {
        return 220;
    }
}

package com.a;

public class client {
    public static void main(String[] args) {
        Adapter adapter=new Adapter(new Source());
        adapter.targetmethod();
    }
}
```

```
}  
}  
package com.a;  
  
public class Adapter implements AbstractTarget{  
    private Source source;  
    //在构造方法中指定  
    public Adapter(Source source) {  
        super();  
        this.source = source;  
    }  
  
    @Override  
    public void targetmethod() {  
        System.out.println("没适配之前：输出 5V/2018214937 刘铭源");  
        int result=this.source.sourceMethod();  
        System.out.println("适配之后：输出"+result+"v/2018214937 刘铭源");  
    }  
}  
}  
package com.a;  
  
public interface AbstractTarget {  
    public void targetmethod();  
}
```

实验 3--行为型设计模式-观察者模式

1、实验目的与内容

1.1 实验目的

加深所学理论知识的理解，掌握常见软件体系结构的知识使用方法；掌握观察者设计模式的结构和适用问题，并提高解决实际问题的能力。掌握观察者模式（Observer Pattern）的特点分析具体问题，使用适配器模式进行设计。

1.2 实验内容

编程实现一个程序，实现将狙击手和突击手与歹徒绑定，当歹徒即将威胁到人质安全的时候，狙击手和突击手分别作出相应的动作来响应

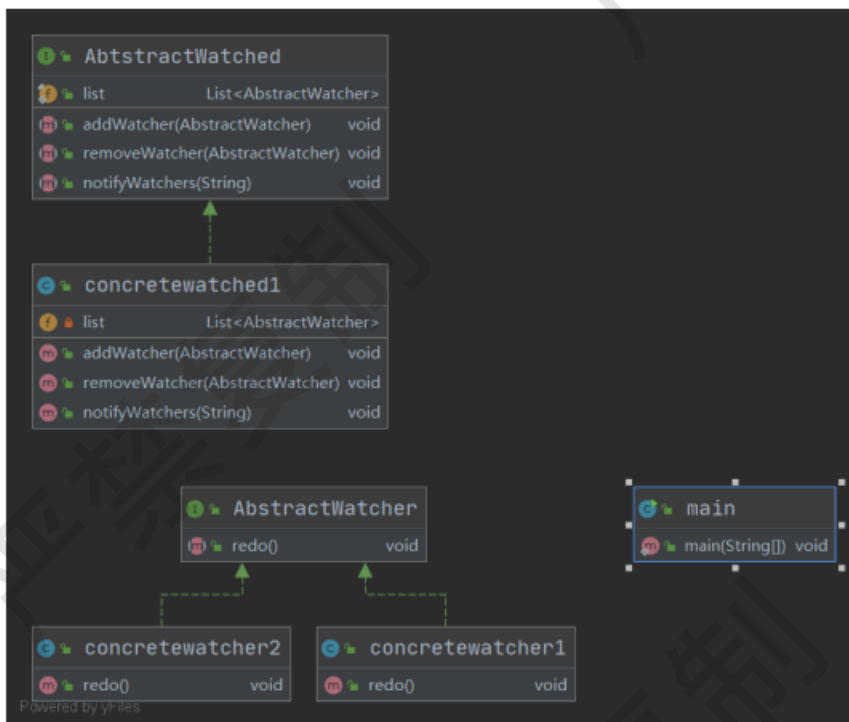
2、分析设计过程

2.1 模式分析

本实验包含三个对象，即：歹徒、狙击手、突击手，而狙击手和突击手要根据歹徒的行为作出相对应的行为。

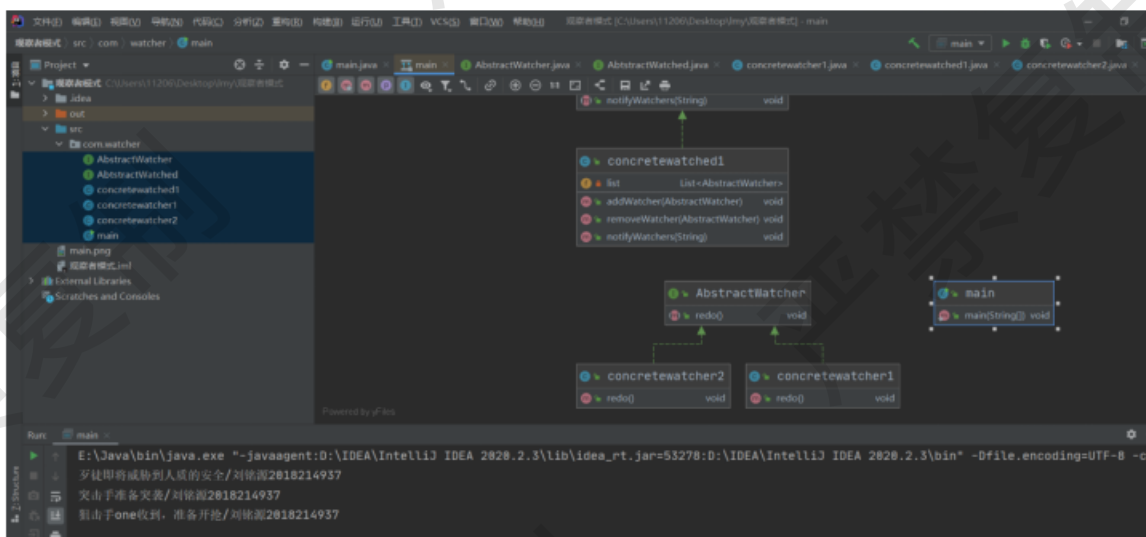
经过分析，我觉得更加适合使用观察者模式方法。因为本次实验所要求的功能与观察者模式相一致，狙击手和突击手充当观察者，歹徒充当被观察者。并且通过观察者模式可以独立的更改狙击手、突击手和歹徒。并且歹徒所做的每一个动作必须通知其它警卫，或者说其他警卫必须了解，而歹徒不可能知道那些警卫都是谁。

2.2 UML 类图

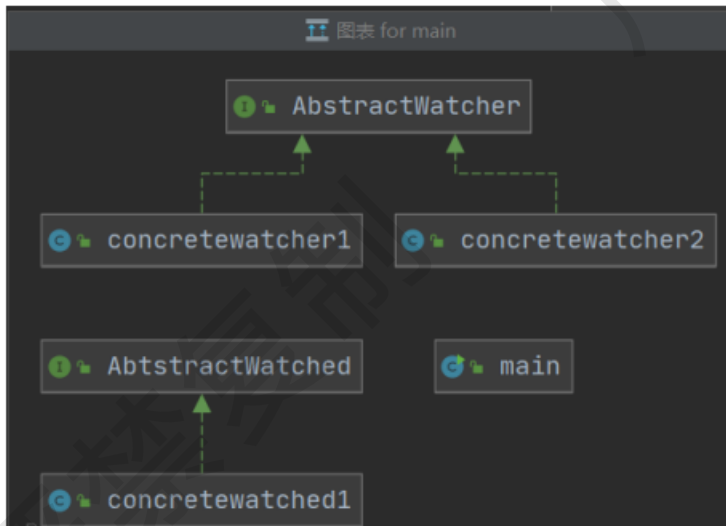


3、测试数据和程序运行的结果

3.1 运行结果:



3.2 整体框架



4.问题与总结

通过此次试验，我学到了许多观察者模式的的知识，在实验中充分利用自己的所学的内容强化了观察者模式的内容；利用 Java 将观察者模式进行创建，学到了观察者模型的内容：观察者模式在被观察者和观察者之间建立一个抽象的耦合；同时，被观察者角色所知道的只是一个具体观察者列表，每一个具体观察者都符合一个抽象观察者的接口。被观察者并不认识任何一个具体观察者，它只知道它们都有一个共同的接口。由于被观察者和观察者没有紧密地耦合在一起，因此它们可以属于不同的抽象化层次。如果被观察者和观察者都被扔到一起，那么这个对象必然跨越抽象化和具体化层次。

5、 附录

5.1 歹徒部分：

抽象歹徒：

```
package com.watcher;

import java.util.ArrayList;
import java.util.List;

public interface AbtstractWatched {
    List<AbstractWatcher> list = new ArrayList<AbstractWatcher>();
    // 添加观察者
    public void addWatcher(AbstractWatcher abstractWatcher);
}
```



```

// 移除观察者
public void removeWatcher(AbstractWatcher abstractWatcher);

// 通知观察者
public void notifyWatchers(String str);
}

```

具体歹徒

```

package com.watcher;

import java.util.ArrayList;
import java.util.List;

public class concretewatched1 implements AbtstractWatched {
    private List<AbstractWatcher> list = new
    ArrayList<AbstractWatcher>();

    public void addWatcher(AbstractWatcher abstractWatcher) {
        list.add(abstractWatcher);
    }

    public void removeWatcher(AbstractWatcher abstractWatcher) {
        list.remove(abstractWatcher);
    }

    public void notifyWatchers(String str) {
        System.out.println("歹徒即将威胁到人质的安全/刘铭源
2018214937");
        for (AbstractWatcher abstractWatcher : list) {
            abstractWatcher.redo();
        }
    }
}

```

5.2 警察部分:

抽象警察:

```
package com.watcher;
```

```
public class concretewatcher2 implements AbstractWatcher {
```

```
    public void redo(){
```

```
        System.out.println("狙击手 one 收到，准备开枪/刘铭源  
2018214937");
```

```
    };
```

```
}
```

```
package com.watcher;
```

```
public class concretewatcher1 implements AbstractWatcher {
```

```
    public void redo() {
```

```
        System.out.println("突击手准备突袭/刘铭源 2018214937");
```

```
    }
```

```
}
```

5.3 主函数部分：

```
package com.watcher;
```

```
public class main {
```

```
    public static void main(String[] args) {
```

```
        concretewatched1 a = new concretewatched1();
```

```
        concretewatcher1 b = new concretewatcher1();
```

```
        concretewatcher2 c = new concretewatcher2();
```

```
        a.addWatcher(b);
```

```
        a.addWatcher(c);
```

```
        a.notifyWatchers("被观察者变化/刘铭源 2018214937");
```

```
    }
```

```
}
```

实验 4--架构模式-面向对象(MVC 设计模式为 抽象工厂) 模式

1、实验目的与内容

1.1 实验目的

加深所学理论知识的理解，掌握常见软件体系结构的知识使用方法；掌握抽象工厂设计模式的结构和适用问题，并提高解决实际问题的能力。

掌握 MVC 模式的特点

分析具体问题，使用抽象工厂模式进行设计。

1.2 实验内容

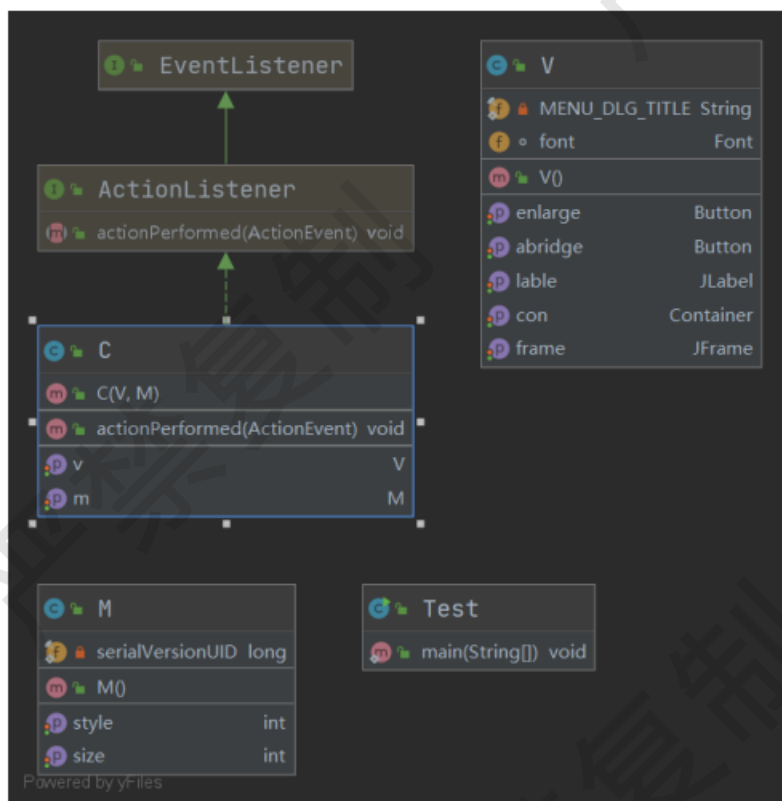
编程实现一个程序，程序包括界面，可以通过控制控件将显示的字体“test”进行放大和缩小，完成 MVC 模式的设计。

2、分析设计过程

2.1 模式分析

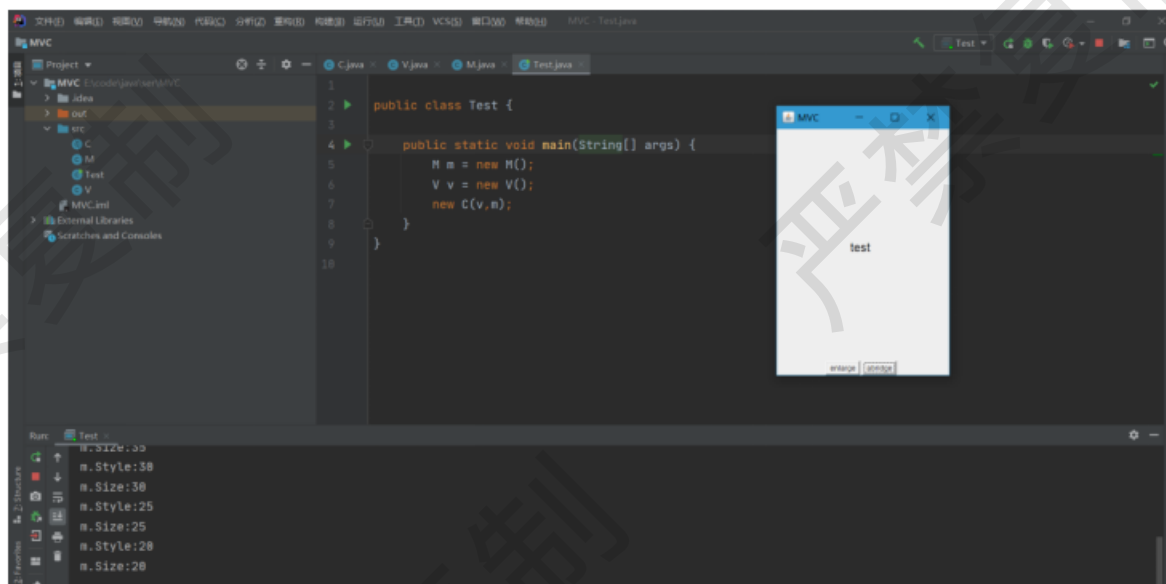
本次试验非常符合 MVC 架构的要求，即 model、view、control 模块，v 用于获取用户输入的模型，m 模块为产品，c 模块用于创建 um 选择视图，test 用于测试模型。

2.2 UML 类图

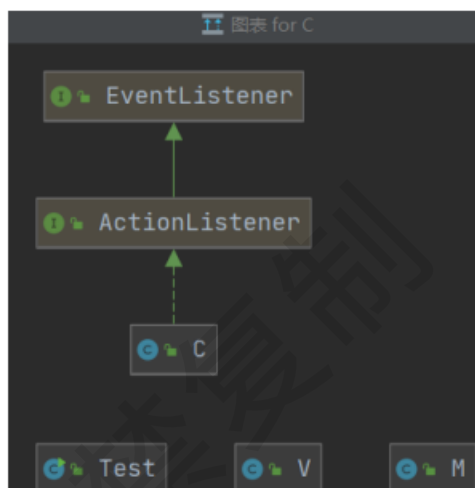


3、测试数据和程序运行的结果

3.1 运行结果：



3.2 整体框架



4、 问题与总结

通过此次试验，我结合 Java 课程中的 swing 编写面向对象的 MVC 模式，通过控制控件将字体进行放大和缩小；在 MVC 模式中，三个层各施其职，所以如果一旦哪一层的需求发生了变化，就只需要更改相应的层中的代码而不会影响到其它层中的代码。这样降低了软件的耦合性，提高了软件的内聚性。

5、 附录

C.java

```
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class C implements ActionListener{
    private V v;
    private M m;

    public C(V v,M m){
        this.v = v;
        this.m = m;
    }

    public void actionPerformed(ActionEvent e) {
```

```
if (e.getSource() == v.getEnlarge()) {  
    m.setSize(m.getSize() + 5);  
    m.setStyle(m.getStyle() + 5);  
}else{  
    m.setSize(m.getSize() - 5);  
    m.setStyle(m.getStyle() - 5);  
}
```

```
System.out.println("m.Style:" + m.getStyle());  
System.out.println("m.Size:" + m.getSize());  
Font font = new  
Font(v.getLabel().getText(),m.getStyle(),m.getSize());  
v.getLabel().setFont(font);  
  
}
```

```
public V getV() {
```

```
    return v;
```

```
}
```

```
public void setV(V v) {
```

```
    this.v = v;
```

```
}
```

```
public M getM() {
```

```
    return m;
```

```
}
```

```
public void setM(M m) {
```

```
    this.m = m;
```

```
}
```

```
}
```

M.java

```
public class M {  
  
    private static final long serialVersionUID = 1L;  
    private int size = 20;  
    private int style = 20;  
  
    public M() {  
    }  
}
```

```
    public int getSize() {  
        return size;  
    }  
}
```

```
    public void setSize(int size) {  
        this.size = size;  
    }  
}
```

```
    public int getStyle() {  
        return style;  
    }  
}
```

```
    public void setStyle(int style) {  
        this.style = style;  
    }  
}
```

```
}
```

V.java

```
import java.awt.Button;  
import java.awt.Container;  
import java.awt.Font;  
  
import javax.swing.JFrame;  
import javax.swing.JLabel;  
import javax.swing.SpringLayout;
```



```
public class V {

    // TITLE
    final private static String MENU_DLQ_TITLE = "MVC";

    // Frame
    private JFrame frame = new JFrame(MENU_DLQ_TITLE);

    // Container
    private Container con = frame.getContentPane();

    Font font = new Font("test", 20, 20);

    JLabel lable = new JLabel(font.getName());
    Button enlarge = new Button("enlarge");
    Button abridge = new Button("abridge");

    public V() {
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        SpringLayout lay = new SpringLayout();
        con.setLayout(lay);

        lay.putConstraint(SpringLayout.WEST, lable,
120, SpringLayout.WEST, con);
        lay.putConstraint(SpringLayout.NORTH, lable, 180,
SpringLayout.NORTH, con);

        lay.putConstraint(SpringLayout.WEST, enlarge, 80,
SpringLayout.WEST, con);
        lay.putConstraint(SpringLayout.NORTH, enlarge, 380,
SpringLayout.NORTH, con);

        lay.putConstraint(SpringLayout.WEST, abridge, 140,
SpringLayout.WEST, con);
        lay.putConstraint(SpringLayout.NORTH, abridge, 380,
```

```
SpringLayout.NORTH, con);
```

```
con.add(lable);
```

```
con.add(enlarge);
```

```
con.add(abridge);
```

```
C c = new C(this, new M());
```

```
enlarge.addActionListener(c);
```

```
abridge.addActionListener(c);
```

```
frame.setBounds(520, 80, 300, 450);
```

```
frame.setVisible(true);
```

```
}
```

```
public Button getEnlarge() {
```

```
    return enlarge;
```

```
}
```

```
public void setEnlarge(Button enlarge) {
```

```
    this.enlarge = enlarge;
```

```
}
```

```
public Button getAbridge() {
```

```
    return abridge;
```

```
}
```

```
public void setAbridge(Button abridge) {
```

```
    this.abridge = abridge;
```

```
}
```

```
public JFrame getFrame() {
```

```
    return frame;
```

```
}
```

```
public void setFrame(JFrame frame) {  
    this.frame = frame;  
}
```

```
public Container getCon() {  
    return con;  
}
```

```
public void setCon(Container con) {  
    this.con = con;  
}
```

```
public JLabel getLable() {  
    return lable;  
}
```

```
public void setLable(JLabel lable) {  
    this.lable = lable;  
}
```

Test.java

```
public class Test {  
  
    public static void main(String[] args) {  
        M m = new M();  
        V v = new V();  
        new C(v,m);  
    }  
}
```

实验 5--架构模式-管道过滤器模式

1、实验目的与内容

1.1 实验目的

加深所学理论知识的理解，掌握常见软件体系结构的知识使用方法；掌握抽象工厂设计模式的结构和适用问题，并提高解决实际问题的能力。

掌握管道过滤器模式的特点

分析具体问题，使用管道过滤器模式进行设计。

1.2 实验内容

我们将创建一个 **Person** 的对象、**Criteria** 的接口和实现该类接口的实体类，来过滤 **Person** 对象的列表，**CriteriaPatternDemo** 我们的演示类将使用 **Criteria** 的对象，基于各种标准和他们解和过滤 **Person** 的对象列表

2、分析设计过程

2.1 模式分析

第一步：我们创建一个类，在该类应用标准

第二步为标准（**criteria**）创建一个窗口

第三步创建实现 **Criteria** 接口和实体类

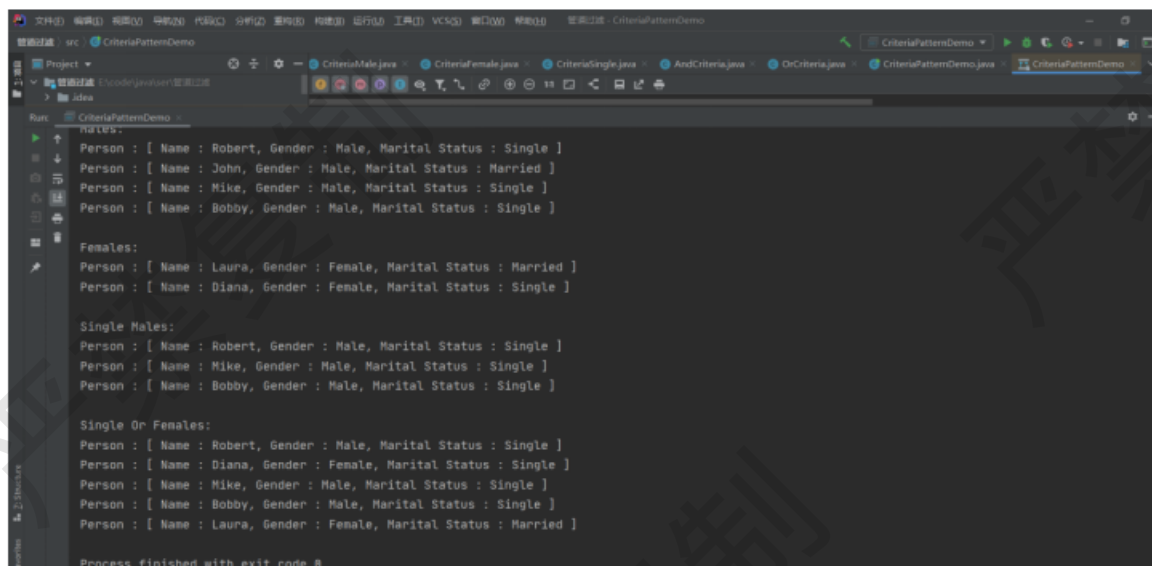
第四步使用不同的标准和他们结合过滤 **person** 对象列表

2.2 UML 类图



3、测试数据和程序运行的结果

3.1 运行结果：



```
Run: CriteriaPatternDemo
Person : [ Name : Robert, Gender : Male, Marital Status : Single ]
Person : [ Name : John, Gender : Male, Marital Status : Married ]
Person : [ Name : Mike, Gender : Male, Marital Status : Single ]
Person : [ Name : Bobby, Gender : Male, Marital Status : Single ]

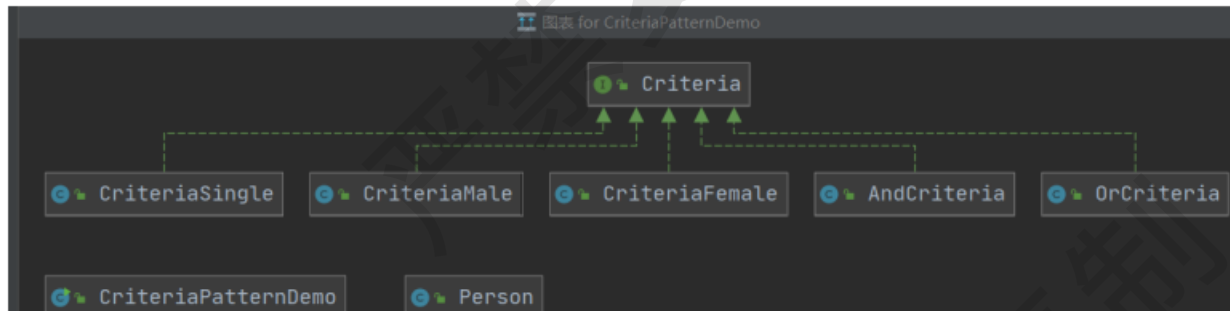
Females:
Person : [ Name : Laura, Gender : Female, Marital Status : Married ]
Person : [ Name : Diana, Gender : Female, Marital Status : Single ]

Single Males:
Person : [ Name : Robert, Gender : Male, Marital Status : Single ]
Person : [ Name : Mike, Gender : Male, Marital Status : Single ]
Person : [ Name : Bobby, Gender : Male, Marital Status : Single ]

Single Or Females:
Person : [ Name : Robert, Gender : Male, Marital Status : Single ]
Person : [ Name : Diana, Gender : Female, Marital Status : Single ]
Person : [ Name : Mike, Gender : Male, Marital Status : Single ]
Person : [ Name : Bobby, Gender : Male, Marital Status : Single ]
Person : [ Name : Laura, Gender : Female, Marital Status : Married ]

Process finished with exit code 0
```

3.2 整体框架



4、问题与总结

通过此次试验，我学到了许多关于管道过滤器模式的知识：允许将系统的输入和输出看作是各个过滤器行为的简单组合，独立的过滤器能够减小构件之间的耦合程度，由于各个独立模块的加入可以让新模块的加入变得非常简单，并且支持功能模块的重用；过滤器并不知道它的上游和下游的过滤器的特性，所以各个步骤之间是透明的，可以有效保证各个步骤之间的安全性，它的设计和实现不会对与它相连的过滤器加以限制。

5、附录



```
import java.util.List;
```

```
public class AndCriteria implements Criteria {

    private Criteria criteria;
    private Criteria otherCriteria;

    public AndCriteria(Criteria criteria, Criteria otherCriteria) {
        this.criteria = criteria;
        this.otherCriteria = otherCriteria;
    }

    @Override
    public List<Person> meetCriteria(List<Person> persons) {
        List<Person> firstCriteriaPersons = criteria.meetCriteria(persons);
        return otherCriteria.meetCriteria(firstCriteriaPersons);
    }
}

import java.util.List;

public interface Criteria {
    List<Person> meetCriteria(List<Person> persons);
}

import java.util.ArrayList;
import java.util.List;

public class CriteriaFemale implements Criteria {

    @Override
    public List<Person> meetCriteria(List<Person> persons) {
        List<Person> femalePersons = new ArrayList<Person>();
        for (Person person : persons) {
            if(person.getGender().equalsIgnoreCase("FEMALE")){
                femalePersons.add(person);
            }
        }
        return femalePersons;
    }
}
```

```

    }
}
import java.util.ArrayList;
import java.util.List;

public class CriteriaMale implements Criteria {

    @Override
    public List<Person> meetCriteria(List<Person> persons) {
        List<Person> malePersons = new ArrayList<Person>();
        for (Person person : persons) {
            if(person.getGender().equalsIgnoreCase("MALE")){
                malePersons.add(person);
            }
        }
        return malePersons;
    }
}

```

```

import java.util.ArrayList;
import java.util.List;

public class CriteriaPatternDemo {
    public static void main(String[] args) {
        List<Person> persons = new ArrayList<Person>();

        persons.add(new Person("Robert","Male", "Single"));
        persons.add(new Person("John","Male", "Married"));
        persons.add(new Person("Laura","Female", "Married"));
        persons.add(new Person("Diana","Female", "Single"));
        persons.add(new Person("Mike","Male", "Single"));
        persons.add(new Person("Bobby","Male", "Single"));

        Criteria male = new CriteriaMale();
        Criteria female = new CriteriaFemale();
        Criteria single = new CriteriaSingle();
    }
}

```



```
Criteria singleMale = new AndCriteria(single, male);
Criteria singleOrFemale = new OrCriteria(single, female);
```

```
System.out.println("Males: ");
printPersons(male.meetCriteria(persons));
```

```
System.out.println("\nFemales: ");
printPersons(female.meetCriteria(persons));
```

```
System.out.println("\nSingle Males: ");
printPersons(singleMale.meetCriteria(persons));
```

```
System.out.println("\nSingle Or Females: ");
printPersons(singleOrFemale.meetCriteria(persons));
```

```
}
```

```
public static void printPersons(List<Person> persons){
    for (Person person : persons) {
```

```
        System.out.println("Person : [ Name : " + person.getName()
            + ", Gender : " + person.getGender()
            + ", Marital Status : " + person.getMaritalStatus()
            + " ]");
```

```
    }
```

```
}
```

```
}
```

```
import java.util.ArrayList;
```

```
import java.util.List;
```

```
import java.util.ArrayList;
```

```
import java.util.List;
```

```
public class CriteriaSingle implements Criteria {
```

```
    @Override
```

```
    public List<Person> meetCriteria(List<Person> persons) {
```

```
List<Person> singlePersons = new ArrayList<Person>();  
for (Person person : persons) {  
    if(person.getMaritalStatus().equalsIgnoreCase("SINGLE")){  
        singlePersons.add(person);  
    }  
}  
return singlePersons;  
}
```

```
import java.util.List;
```

```
public class OrCriteria implements Criteria {
```

```
    private Criteria criteria;  
    private Criteria otherCriteria;
```

```
    public OrCriteria(Criteria criteria, Criteria otherCriteria) {  
        this.criteria = criteria;  
        this.otherCriteria = otherCriteria;  
    }
```

```
@Override
```

```
    public List<Person> meetCriteria(List<Person> persons) {  
        List<Person> firstCriteriaItems = criteria.meetCriteria(persons);  
        List<Person> otherCriteriaItems =  
otherCriteria.meetCriteria(persons);  
  
        for (Person person : otherCriteriaItems) {  
            if(!firstCriteriaItems.contains(person)){  
                firstCriteriaItems.add(person);  
            }  
        }  
        return firstCriteriaItems;  
    }  
}
```

```
public class Person {  
  
    private String name;  
    private String gender;  
    private String maritalStatus;  
  
    public Person(String name,String gender,String maritalStatus){  
        this.name = name;  
        this.gender = gender;  
        this.maritalStatus = maritalStatus;  
    }  
  
    public String getName() {  
        return name;  
    }  
    public String getGender() {  
        return gender;  
    }  
    public String getMaritalStatus() {  
        return maritalStatus;  
    }  
}
```