# 综合性实验报告

华南农业大学信息学院

软件体系结构综合性实验报告

龙婷

201727020119

2020年1月完稿

目录

[综合性实验报告 1](#_Toc29504597)

[综合性实验内容 5](#_Toc29504598)

[综合性实验要求 6](#_Toc29504599)

[综合性实验的实现的总界面 6](#_Toc29504600)

[模式一：接口模式 7](#_Toc29504601)

[(1) 解决问题类型 7](#_Toc29504602)

[(2) 解决设计思路 7](#_Toc29504603)

[(3) 具体待解决的问题 7](#_Toc29504604)

[(4) UML设计 7](#_Toc29504605)

[(5) 核心代码 7](#_Toc29504606)

[(6) 实现效果 8](#_Toc29504607)

[(7) 模式的优点 8](#_Toc29504608)

[(8) 模式的不足 8](#_Toc29504609)

[模式二：单例模式 8](#_Toc29504610)

[(1) 解决问题类型 8](#_Toc29504611)

[(2) 解决设计思路 8](#_Toc29504612)

[(3) 具体待解决的问题 8](#_Toc29504613)

[(4) UML设计 9](#_Toc29504614)

[(5) 核心代码 9](#_Toc29504615)

[(6) 实现效果 10](#_Toc29504616)

[(7) 模式的优点 11](#_Toc29504617)

[(8) 模式的不足 11](#_Toc29504618)

[模式三：多例模式 11](#_Toc29504619)

[(1) 解决问题类型 11](#_Toc29504620)

[(2) 解决设计思路 12](#_Toc29504621)

[(3) 具体待解决的问题 12](#_Toc29504622)

[(4) UML设计 12](#_Toc29504623)

[(5) 核心代码 12](#_Toc29504624)

[(6) 实现效果 13](#_Toc29504625)

[(7) 模式的优点 14](#_Toc29504626)

[(8) 模式的不足 15](#_Toc29504627)

[模式四：简单工厂模式 15](#_Toc29504628)

[(1) 解决问题类型 15](#_Toc29504629)

[(2) 解决设计思路 15](#_Toc29504630)

[(3) 具体待解决的问题 15](#_Toc29504631)

[(4) UML设计 16](#_Toc29504632)

[(5) 核心代码 16](#_Toc29504633)

[(6) 实现效果 17](#_Toc29504634)

[(7) 模式的优点 17](#_Toc29504635)

[(8) 模式的不足 17](#_Toc29504636)

[模式五：工厂模式 18](#_Toc29504637)

[(1) 解决问题类型 18](#_Toc29504638)

[(2) 解决设计思路 18](#_Toc29504639)

[(3) 具体待解决的问题 18](#_Toc29504640)

[(4) UML设计 19](#_Toc29504641)

[(5) 核心代码 19](#_Toc29504642)

[(6) 实现效果 20](#_Toc29504643)

[(7) 模式的优点 21](#_Toc29504644)

[(8) 模式的不足 21](#_Toc29504645)

[模式六：抽象工厂模式 21](#_Toc29504646)

[(1) 解决问题类型 21](#_Toc29504647)

[(2) 解决设计思路 21](#_Toc29504648)

[(3) 具体待解决的问题 21](#_Toc29504649)

[(4) UML设计 22](#_Toc29504650)

[(5) 核心代码 22](#_Toc29504651)

[(6) 实现效果 24](#_Toc29504652)

[(7) 模式的优点 24](#_Toc29504653)

[(8) 模式的不足 24](#_Toc29504654)

[模式七：适配器模式（类、对象） 25](#_Toc29504655)

[(1) 解决问题类型 25](#_Toc29504656)

[(2) 解决设计思路 25](#_Toc29504657)

[(3) 具体待解决的问题 25](#_Toc29504658)

[(4) UML设计 25](#_Toc29504659)

[(5) 核心代码 27](#_Toc29504660)

[(6) 实现效果 28](#_Toc29504661)

[(7) 类适配器模式和对象适配器模式的区别 30](#_Toc29504662)

[(8)模式的优点 30](#_Toc29504663)

[(9) 模式的不足 30](#_Toc29504664)

[模式八：安全组合模式 31](#_Toc29504665)

[(1) 解决问题类型 31](#_Toc29504666)

[(2) 解决设计思路 31](#_Toc29504667)

[(3) 具体待解决的问题 31](#_Toc29504668)

[(4) UML设计 32](#_Toc29504669)

[(5) 核心代码 32](#_Toc29504670)

[(6) 实现效果 34](#_Toc29504671)

[(7) 模式的优点 35](#_Toc29504672)

[(8) 模式的不足 35](#_Toc29504673)

[模式九：一致性组合模式 35](#_Toc29504674)

[(1) 解决问题类型 35](#_Toc29504675)

[(2) 解决设计思路 35](#_Toc29504676)

[(3) 具体待解决的问题 36](#_Toc29504677)

[(4) UML设计 37](#_Toc29504678)

[(5) 核心代码 37](#_Toc29504679)

[(6) 实现效果 39](#_Toc29504680)

[(7) 模式的优点 40](#_Toc29504681)

[(8) 模式的不足 40](#_Toc29504682)

[模式十：基于DI、ORM模式，使用Spring集成Hibernate框架，实现学生指导学生毕业设计 40](#_Toc29504683)

[(1) 具体待解决的问题 40](#_Toc29504684)

[(2) 核心代码 41](#_Toc29504685)

[(3) 实现效果 49](#_Toc29504686)

[综合性实验的心得体会 54](#_Toc29504687)

## 综合性实验内容

利用一个Java Fx主界面，描述本学期所实验的设计设计模式，每一个按钮对应一个设计模式，点击该按钮，打开利用该模式的实例运行界面。

综合性实验主界面



点击不同的实验后显示各个设计模式的实例



## 综合性实验要求

（1）需要完成单例模式、多例模式、简单工厂方法模式、工厂模式、抽象工厂模式、对象适配器模式、类适配器模式、安全型组合模式、一致型组合模式，完成以上模式的同学可以最高得分90分；利用基于DI、ORM模式，使用Spring集成Hibernate框架，完成“学生指导学生毕业设计”实验的最高可得100分。

（2）代码请按照规范提交，路径名，项目名，报名，类名要符合规范，要确保clone下来后能直接运行，尤其注意JDK设置，项目的编码设置(UTF-8)，类路径、jar包等，同学们最好在非自己私人电脑上测试一下，以免出现上述的情况。

（3）所有的图都要使用visio画，并且复制粘贴到实验报告中，不允许截屏，图片大小要合适，美观大方。

（4）不需要所有的代码，有核心的表示设计思想的代码片段即可。

（5）标题按照本实验文档的格式设置，正文和代码一律使用中文宋体，英语times new roma字体，大小是五号。代码使用外框框住。

（6）项目名称、包名、word文件名等等是我课堂上强调的命名规范，不能自由发挥。

（7）报告需要打印纸质文件。

（8）附件是实验的实验报告模板，请大家按照模板编写实验报告

## 综合性实验的实现的总界面



## 模式一：接口模式

### (1) 解决问题类型

一个实体对象在逻辑上有多种子类，当需要调用所有子类的相同方法，但是这些方法在不同子类有不同的处理操作时，为了使代码不冗余以及保证其灵活性，使用接口编程。

### (2) 解决设计思路

核心思想是使用接口回调，即接口变量存放实现该接口的类的对象的引用，从而接口变量就可以回调类实现的接口方法。

### (3) 具体待解决的问题

大学生本科毕业后存在两种学生子类，分别为应届毕业生和在读研究生，以接口模式的设计模式，调用这两种学生的learn() 方法，输出对应的学习动作。

### (4) UML设计



### (5) 核心代码

**public class InterProgrammingClient {**

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

**Student graduateStudent = new GraduateStudent();**

**graduateStudent.learning();**

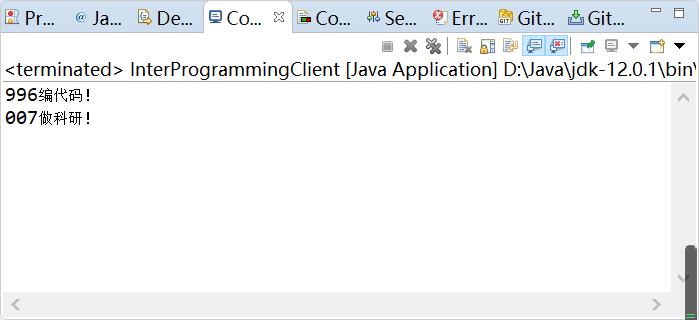
**Student undergraduateStudent = new UndergraduateStudent();**

**undergraduateStudent.learning();**

**}**

**}**

### (6) 实现效果



### (7) 模式的优点

* 降低了程序的耦合性
* 易拓展，易维护
* 实现了多态

### (8) 模式的不足

* 增加了设计的复杂度，不佳的接口的设计会对所有使用接口的层都有影响，并且并不是所有的程序的都需要使用接口，接口只有在系统的复杂性达到一定程度才能体现出它的优势，否则只是纯粹的增加工作量，当然选择接口是不会错的，这需要自己的衡量
* 可能会降低代码的可复用性
* 可能会降低程序的执行效率

## 模式二：单例模式

### (1) 解决问题类型

一个类仅能用来产生一个唯一对象，并且提供一个全局的访问点。

### (2) 解决设计思路

1. 构造方法私有，不能在外面被创建
2. 公有的、静态的获取实例的方法
3. 获取实例是判定实例是否为空，如果没有，就新建，否则就返回这个存在的实例
4. 有一个静态私有的实例，保存创建的实例

### (3) 具体待解决的问题

1. 包名、类名等要求和前面一致。客户端类名为SingletonClient.
2. 建立一个JavaFX界面，在界面中有一个创建主席(Chairman)名称的输入框、一个创建按钮和一个显示创建成功信息的Label。
3. 输入框输入你需要创建的主席的名称，例如输入“毛泽东”。
4. 点击创建按钮，在Label中显示创建对象的名称和对象(默认就是类似Chairman@234ed34这样的输出)。
5. 再次或者多次输入主席名称，点击创建，在前面显示创建结果的下一行显示本次创建对象的名称和对象。

### (4) UML设计



（因为单例和多例是组合在一个界面显示的，所以上面的UML图左半边是单例，右半边是多例）

### (5) 核心代码

public class Chairman {

private static Chairman chairman;

private String name;

private Chairman(String name) { this.name=name; }

public static Chairman getInstance(String name) {

if(chairman == null) {

chairman = new Chairman(name);

return chairman;

}else { return chairman; }

}

-----getter() 和 setter()---------

}

### (6) 实现效果

主界面



输入要创建的主席的名称，点击“创建”，可生成相应的对象实例



再次点击“创建”，因为无法创建新的对象，输出相应的提示



### (7) 模式的优点

* 减少内存开销，尤其是频繁的创建和销毁实例
* 避免对资源对过多占用。

### (8) 模式的不足

* 没有抽象层，不能继承扩展很难。
* 违背了“单一职责原则”，一个类只重视内部关系，而忽略外部关系。
* 不适用于变化对象。
* 滥用单例会出现一些负面问题，如为节省资源将数据库连接池对象设计为单例，可能会导致共享连接池对象对程序过多而出现连接池溢出。如果实例化的对象长时间不被利用，系统会认为是垃圾而被回收，这样将导致对象状态丢失。

## 模式三：多例模式

### (1) 解决问题类型

当存在这样的类，只需要有限数量实例的情况下，可以使用多例。

### (2) 解决设计思路

1. 构造方法私有，不能在外面被创建
2. 公有的、静态的获取实例的方法
3. 获取实例是否已经达到了指定数量，如果没有，就新建，否则就返回现有的集合
4. 有一个静态私有的实例数组，保存创建的实例

### (3) 具体待解决的问题

在上面单例模式的问题的基础上，使用单例模式的思想实现多例模式，确保系统中某个类的对象只能存在有限个，如中华人民共和国十大元帅(Marshal)，设计并实现。界面的基本要求和前面一致。

### (4) UML设计



（因为单例和多例是组合在一个界面显示的，所以上面的UML图左半边是单例，右半边是多例）

### (5) 核心代码

public class Marshal {

private static int NUM = 10;

private static ArrayList<Marshal> marshals = new ArrayList<>(NUM);

private String name;

private Marshal(String name) { this.name = name; }

public static Marshal getInstance(String name) {

if(marshals.size() < NUM) {

marshals.add(new Marshal(name));

return marshals.get(marshals.size()-1);

}else {

return marshals.get(NUM-1);

}

}

-------------getter() 和 setter() --------------

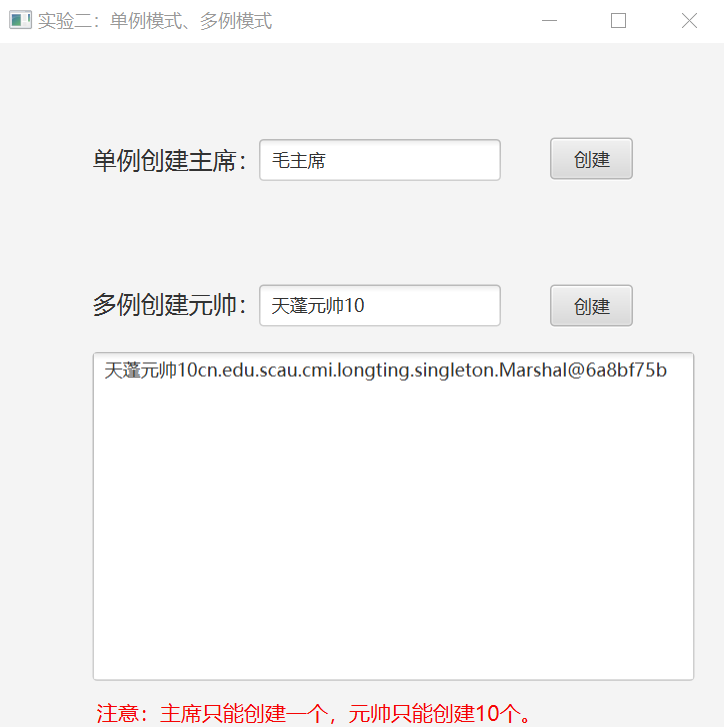
}

### (6) 实现效果

主界面



输入要创建的元帅的名称，点击“创建”，可生成相应的元帅对象实例



多次点击（超过10次）“创建”，因为只能创建十个新的对象，输出相应的提示



### (7) 模式的优点

可以控制类的实例数量

### (8) 模式的不足

提高了类实例维护复杂度

## 模式四：简单工厂模式

### (1) 解决问题类型

工厂类负责创建的对象比较少：由于创建的对象较少，不会造成工厂方法中的业务逻辑太过复杂。客户端只知道传入工厂类的参数，对于如何创建对象不关心：客户端既不需要关心创建细节，甚至连类名都不需要记住，只需要知道类型所对应的参数。

### (2) 解决设计思路



（简单工厂方法模式类图）

上面类图每一部分的意义如下：

1. Creator : 是简单工厂方法模式的核心，包含应用程序所需要的业务逻辑，当客户类Client需要的时候，委托工厂类创建产品类的对象。
2. Product：可以是Java接口或者Java抽象类，是具体子类的超类或者共同接口。
3. ConcreateProduct：实现Product接口，或者继承抽象类Product.

### (3) 具体待解决的问题

肉厂(MeatFactory)可以生产猪肉(Pork)，牛肉(Beef)，鸡肉(Chicken)等肉类(Meat)，使用简单工厂方法实现肉类的生产。

在界面上选择某一种肉类，在输出框中显示获得的对象的行为。

### (4) UML设计



### (5) 核心代码

public class MeatFactory {

public static Meat getMeat(String meatTypeString) {

Meat meat = null;

switch (meatTypeString) {

case "beef":

meat = new Beef();break;

case "pork":

meat = new Pork();break;

case "chicken":

meat = new Chicken();break;

}

return meat;

}

}

public abstract class Meat {

public abstract String cook();

}

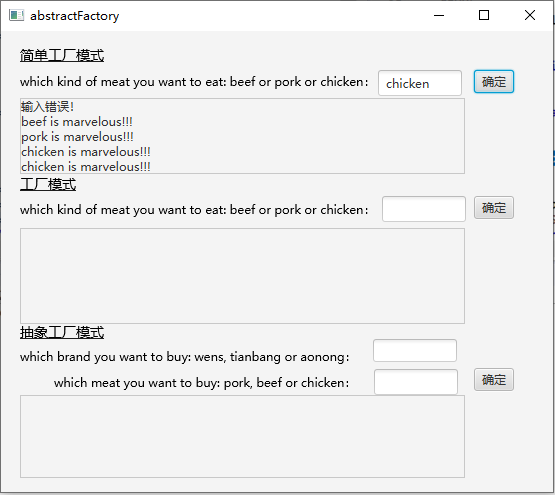
public class Chicken extends Meat{

private String name;

public String cook() { return "chicken is marvelous!!!"; }

}

### (6) 实现效果



### (7) 模式的优点

* 工厂方法包含从一个类的结构中选择初始类的业务逻辑
* 客户类不直接创建产品类的对象，客户类只作为对象的消费者
* 简单工厂方法模式实现了责任分离
* 客户类不负有创建类的对象的责任，因此如果有新的电子产品的加入，不必修改已有的客户类代码（前提是客户类暂时不需要使用新的产品子类对象）
* 因为工厂类中包含了对象生成的必要的逻辑判断，根据客户端的选择条件动态实例化相关的类，所以客户端类中没有创建对象所需要的条件语句。

### (8) 模式的不足

* 由于工厂类集中了所有实例的创建逻辑，违反了开闭原则，将全部创建逻辑集中到了一个工厂类中；
* 它所能创建的类只能是事先考虑到的，如果需要添加新的类，则就需要改变工厂类了。
* 当系统中的具体产品类不断增多时候，可能会出现要求工厂类根据不同条件创建不同实例的需求．
* 这种对条件的判断和对具体产品类型的判断交错在一起，很难避免模块功能的蔓延，对系统的维护和扩展非常不利；

## 模式五：工厂模式

### (1) 解决问题类型

符合工厂模式的情况，多由于解耦

### (2) 解决设计思路



（工厂方法模式类图）

说明：每一个产品类对应于一个工厂类。该工厂类只负责创建相应的产品类的对象。

### (3) 具体待解决的问题

不同类型的肉厂生产不同的肉类，PorkMeatFactory生产猪肉，BeefMeatFactory生产牛肉，ChickenMeatFactory生产鸡肉。使用工厂模式实现这一功能。

### (4) UML设计



### (5) 核心代码

public interface MeatFactory {

Meat getMeat();

}

public class PorkMeatFactory implements MeatFactory {

@Override

public Meat getMeat() {

return new Pork();

}

}

public class FactoryClient {

public static void main(String[] args) {

System.out.println("Please select which kind of meat you want to eat: beef or pork or chicken???");

Scanner scanner = new Scanner(System.in);

String meatString = scanner.next();

MeatFactory meatFactory = null;

switch (meatString) {

case "beef":

meatFactory = new BeefMeatFactory();

break;

case "pork":

meatFactory = new PorkMeatFactory();

break;

case "chicken":

meatFactory = new ChickenMeatFactory();

break;

default:

System.out.println("sorry, your input is not correct!");

break;

}

if(meatFactory != null) {

Meat meat = meatFactory.getMeat();

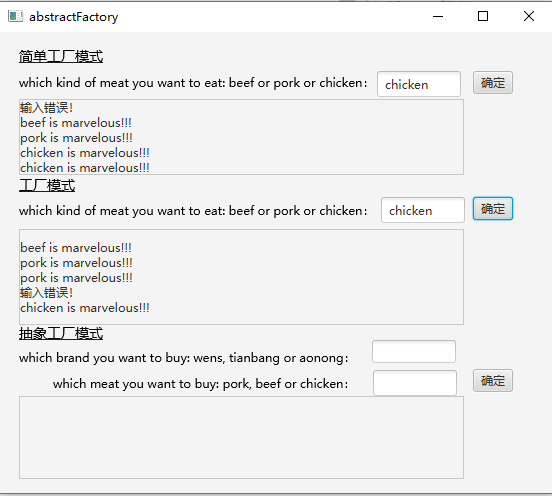
System.out.println(meat.cook());;

}

}

}

### (6) 实现效果



### (7) 模式的优点

* 工厂方法模式将创建对象的逻辑与任务交给了工厂类
* 工厂方法模式支持开闭原则

### (8) 模式的不足

* 相对于简单工厂模式来说，每增加一个产品，相应的也要增加一个子工厂，加大了额外的开发量。

## 模式六：抽象工厂模式

### (1) 解决问题类型

客户对象要从一个相关的产品组中创建一个对象，为没有必要知道到底创建的是哪个对象。

### (2) 解决设计思路



（抽象工厂方法模式设计类图-两种产品类的情况）

### (3) 具体待解决的问题

温氏(Wens)集团、天邦(Tianbang)股份、傲农(Aonong)生物是我国三家大型的养殖企业，可以养猪，养牛，养鸡。利用抽象工厂模式实现用户购买不同品牌(Brand)的不同肉类(Meat)产品。

### (4) UML设计



### (5) 核心代码

public interface Wens extends Brand {

public String NAME="Wens";

}

public abstract class Pork extends Meat{

String TYPE="pork";

}

public class WensPork extends Pork implements Wens{

@Override

public String getMeat() {

return (super.TYPE+" has benn produced by "+Wens.NAME);

}

}

public class WensFactory extends AbstractFactory {

@Override

public Meat getBeef() {

return new WensBeef();

}

@Override

public Meat getChicken() {

return new WensChicken();

}

@Override

public Meat getPork() {

return new WensPork();

}

}

public abstract class AbstractFactory {

public static AbstractFactory getFactory(String brand) {

switch (brand) {

case "Wens":

return new WensFactory();

case "Aonong":

return new AonongFactory();

case "Tianbang":

return new TianbangFactory();

}

return null;

}

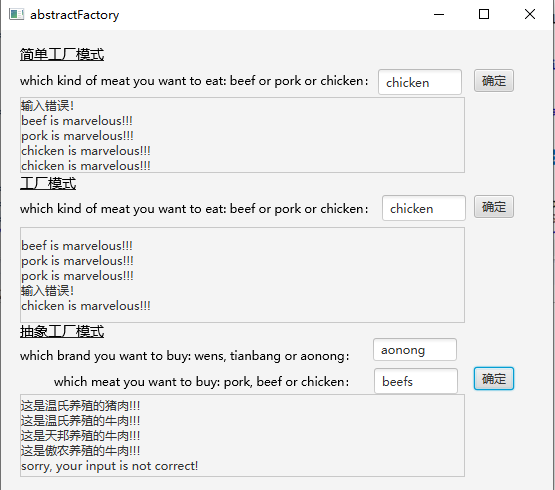
public abstract Meat getBeef();

public abstract Meat getChicken();

public abstract Meat getPork();

}

### (6) 实现效果



### (7) 模式的优点

* 抽象工厂模式隔离了具体类的生产，使得客户并不需要知道什么被创建。
* 当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。
* 增加新的具体工厂和产品类很方便，无须修改已有系统，符合“开闭原则”

### (8) 模式的不足

* 增加新的产品等级结构很复杂，需要修改抽象工厂和所有的具体工厂类
* 对“开闭原则”的支持呈现倾斜性

## 模式七：适配器模式（类、对象）

### (1) 解决问题类型

将一个类的接口转换成用户希望的另一个接口。使得原来由于接口不兼容而不能在一起工作的那些类可以在一起工作。

### (2) 解决设计思路

Interface

Class

Adapter

Interface

Class

Adapter

实现

继承

实现

调用

类适配器模式

对象适配器模式

所期望实现的Java接口

被继承的Java类

被调用的Java类

所期望实现的Java接口

适配器模式分为两种：类适配器模式和对象适配器模式

上图各组成部分的说明：

1. Target：即所期望的Java接口
2. Adaptee：被继承的Java类
3. Adapter：将Adaptee类转化到增加了新功能的Target接口（起了桥梁的作用）

### (3) 具体待解决的问题

类SortUtil可以把整数排序List sortInt(List<int> intList)，这个方法请自己实现；学生有学号(int类型)，姓名(String类型)这两个域。在客户端输入学生的学号和姓名后，及时显示按照学号从小到大排序了的学生名单。排序功能使用类适配器和对象适配器两种方式实现。

### (4) UML设计

**类适配模式的UML图：**



**对象适配模式的UML图：**



### (5) 核心代码

**类适配器模式的核心代码：**

**Sort.java**

public interface Sort {

public List<Student> sortStudents(List<Student> students);

}

**SortClassAdapter.java**

public class SortClassAdapter extends SortUtil implements Sort{

@Override

public List<Student> sortStudents(List<Student> students) {

List<Student> newStudents = new ArrayList<Student>();

List<Integer> sortIntList = new ArrayList<Integer>();

for(Student student : students) {

sortIntList.add(student.getNumber());

}

sortIntList = this.sortInt(sortIntList);

int i,num;

for(int k=0;k < sortIntList.size(); k++) {

num = sortIntList.get(k);

i=0;

while(i<students.size()) {

if(num == students.get(i).getNumber()) {

newStudents.add(students.get(i));

break;

}

i++;

}

}

return newStudents;

}

}

**对象适配器的核心代码：**

**Sort.java**

public interface Sort {

public List<Student> sortStudents(List<Student> students);

}

**SortObjectAdapter.java**

public class SortObjectAdapter implements Sort{

private SortUtil sortUtil;

@Override

public List<Student> sortStudents(List<Student> students) {

sortUtil = new SortUtil();

List<Student> newStudents = new ArrayList<Student>();

List<Integer> sortIntList = new ArrayList<Integer>();

for(Student student : students) {

sortIntList.add(student.getNumber());

}

sortIntList = sortUtil.sortInt(sortIntList);

int i;

for(Integer num : sortIntList) {

i=0;

while(i<students.size()) {

if(num == students.get(i).getNumber()) {

newStudents.add(students.get(i));

break;

}

i++;

}

}

return newStudents;

}

}

### (6) 实现效果

**类适配器模式的实现效果图：**

1. 起始页面：



1. 增加学生信息后，所有的学生在右边的表格中按照学号从小到大进行排序显示：



**对象适配器模式的实现效果图：**

1. 起始页面：



1. 增加学生信息后，所有的学生在右边的表格中按照学号从小到大进行排序显示：



### (7) 类适配器模式和对象适配器模式的区别

在java语言中，使用对象适配器模式可以把多种不同的源类都适配到同一个Target接口，而使用类适配模式是做不到这一点的。

如果一个被适配源类中有大量的方法，使用类适配模式比较容易，只需要让Adapter类继承被适配的源类即可。而此时使用对象适配器模式则要在Adapter类中明确写出Target角色中的每个方法，而且在不同方法中要一一调用被适配的源类中的相应方法。

### (8)模式的优点

* 将目标类和适配者类解耦
* 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，对于客户端来说是透明的，而且提高了适配者的复用性
* 灵活性和拓展性都很好，符合开闭原则
* **类适配器还有优点**： 由于适配器类是适配者类的子类，因此可以在适配器类中置换一些适配者的方法，使得适配器的灵活性更强。
* **对象适配器还有优点**：把多个不同的适配者适配到同一个目标，也就是说，同一个适配器可以把适配者类和他的子类都适配到目标接口。

### (9) 模式的不足

* **类适配器的缺点**：对于Java、C#等不支持多重继承的语言，一次最多只能适配一个适配者类，而且目标抽象类只能为接口，不能为类，其使用有一定的局限性，不能将一个适配者类和他的子类同时适配到目标接口。
* **对象适配器的缺点**：与类适配器模式相比，要想置换适配者类的方法就不容易。

## 模式八：安全组合模式

### (1) 解决问题类型

不加区别地对待部分与整体的关系，处理树形数据，在非叶节点中声明所有用来管理子类对象的方法，而叶节点不包含管理子类对象的方法。

### (2) 解决设计思路



（组合模式的安全形式）

上图个组合部分的说明：

1. Component：为组合模式中的对象声明接口，在适当情况下，实现所有类共有接口的默认行为；声明一个接口用于访问和管理其子组件；在递归结构中定义一个接口，用于访问一个父组件，并在合适的情况下实现。
2. Leaf：在组合模式总表示叶节点对象，叶节点对象没有子节点，实现Component的所有方法。
3. Componsite :表示组合部件（注意部件带有子部件），实现操纵子部件的所有方法；实现所有在Component的操作。
4. Client : 通过Component 接口操纵组合部件的对象。

### (3) 具体待解决的问题

个人(Person)与团队(Team)可以形成一个组织(Organization)：组织有两种：个人组织和团队组织，多个个人可以组合成一个团队，不同的个人与团队可以组合成一个更大的团队。

使用控制台**或者**JavaFx界面完成一下功能：

控制台功能：创建个人，创建团队，每创建一个团队，每次团队成员有变化，就递归输出所有的成员。

下面是最基本的控制台操作过程，同学们可以在最基本功能上做适当修改。

请输入创建的个人的名字：张三

请输入创建的个人的名字：李四

请输入包含这两个人的组织的名字：第一个团队

输出：

第一个团队

张三

李四

javaFx界面的基本界面如下，同学们在此基础上适当修改

 左边是创建对象的树状结构展示，右边是选中某一个节点后的操作。例如，如果选中T12。然后就可以输入T121，输入后直接刷新树状结构。

### (4) UML设计



### (5) 核心代码

public abstract class SafeComponent {

protected String name;

String doSomething() {

return "I am safe component!!!";

}

SafeComponent getParent() { return null; }

abstract String getName();

}

public class SafeComposite extends SafeComponent{

private Set<SafeComponent> children = new HashSet<SafeComponent>();

public SafeComposite() {}

public SafeComposite(String name) { this.name = name; }

public String doSomething() {

return "我是安全性组合模式中的组合组件！！！";

}

public SafeComponent getParent() {

return this.getParent();

}

public SafeComponent addChild(SafeComponent child) {

children.add(child);

return this;

}

public SafeComponent removeChild(SafeComponent child) {

children.remove(child);

return this;

}

public Set<SafeComponent> getChildren(){

return this.children;

}

public void setChildren(Set<SafeComponent> children) {

this.children=children;

}

public String getName() { return name; }

}

public class SafeLeaf extends SafeComponent{

public SafeLeaf() { }

public SafeLeaf(String name) { this.name = name; }

public String doSomething() {

return "我是安全性组合模式里的叶子节点组件！";

}

public SafeComponent getParent() {

return this.getParent();

}

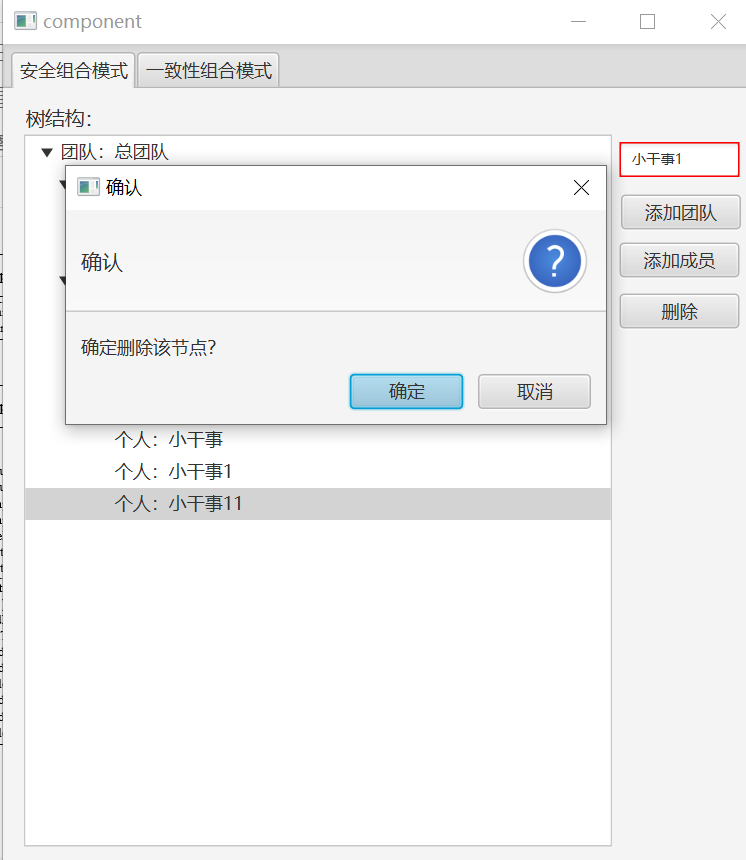
public String getName() { return this.name; }

public void setName(String name) { this.name = name; }

}

### (6) 实现效果





### (7) 模式的优点

* 定义了包含基本对象和组合对象的类层次结构，基本对象可以被组合成更复杂的组合对象，而这个组合对象又可以被组合。
* 简化了客户代码。
* 使得更加容易增加新类型的组件。
* 使设计变得更通用。
* 在Component中不去声明Add和Remove方法，那么子类的Leaf就不需要实现它，而是在Composit声明所有用来管理子类对象的方法。

### (8) 模式的不足

* Component层次结构类的超类与组合子类的接口不一致
* 叶节点无需在实现Add与Remove这样的方法，但是对于客户端来说，必须对叶节点和枝节点进行判定，为客户端的使用带来不便。

## 模式九：一致性组合模式

### (1) 解决问题类型

不加区别地对待部分与整体的关系，处理树形数据，在叶节点和非叶节点所继承的父类中声明所有用来管理子类对象的方法，即非叶节点和叶节点都包含管理子类对象的方法。

### (2) 解决设计思路



（组合模式的一致性形式）

上图个组合部分的说明：

1. Component：为组合模式中的对象声明接口，在适当情况下，实现所有类共有接口的默认行为；声明一个接口用于访问和管理其子组件；在递归结构中定义一个接口，用于访问一个父组件，并在合适的情况下实现。
2. Leaf：在组合模式总表示叶节点对象，叶节点对象没有子节点，实现Component的所有方法。
3. Componsite :表示组合部件（注意部件带有子部件），实现操纵子部件的所有方法；实现所有在Component的操作。
4. Client : 通过Component 接口操纵组合部件的对象。

### (3) 具体待解决的问题

个人(Person)与团队(Team)可以形成一个组织(Organization)：组织有两种：个人组织和团队组织，多个个人可以组合成一个团队，不同的个人与团队可以组合成一个更大的团队。

使用控制台**或者**JavaFx界面完成一下功能：

控制台功能：创建个人，创建团队，每创建一个团队，每次团队成员有变化，就递归输出所有的成员。

下面是最基本的控制台操作过程，同学们可以在最基本功能上做适当修改。

请输入创建的个人的名字：张三

请输入创建的个人的名字：李四

请输入包含这两个人的组织的名字：第一个团队

输出：

第一个团队

张三

李四

javaFx界面的基本界面如下，同学们在此基础上适当修改

 左边是创建对象的树状结构展示，右边是选中某一个节点后的操作。例如，如果选中T12。然后就可以输入T121，输入后直接刷新树状结构。

### (4) UML设计



### (5) 核心代码

public abstract class ConsistentComponent {

protected String name;

ConsistentComponent parent = null;

Set<ConsistentComponent> children = new HashSet<ConsistentComponent>();

abstract String doSomething();

abstract ConsistentComponent getParent();

public abstract ConsistentComponent addChild(ConsistentComponent child);

public abstract ConsistentComponent deleteChild(ConsistentComponent child);

public abstract Set<ConsistentComponent> getChildren();

public abstract String getName();

}

public class ConsistentComposite extends ConsistentComponent{

public ConsistentComposite() { }

public ConsistentComposite(String name) {

this.setName(name);

}

String doSomething() {

System.out.println("这是一致性组件中的非叶子节点！");

return null;

}

ConsistentComponent getParent() {

return this;

}

public Set<ConsistentComponent> getChildren() {

return this.children;

}

public ConsistentComponent addChild(ConsistentComponent child) {

System.out.println("在组合节点中添加子节点");

this.children.add(child);

return this;

}

@Override

public ConsistentComponent deleteChild(ConsistentComponent child) {

children.remove(child);

return this;

}

@Override

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

public class ConsistentLeaf extends ConsistentComponent{

public ConsistentLeaf() {

super();

}

public ConsistentLeaf(String name) {

super();

this.setName(name);

}

@Override

String doSomething() {

System.out.println("这是一致性模式中的叶节点，不能再包含子节点了！");

return "这是一致性模式中的原子节点，不能再包含子节点了！";

}

@Override

ConsistentComponent getParent() {

return this;

}

@Override

public ConsistentComponent addChild(ConsistentComponent child) {

System.out.println("对不起，该节点是叶节点，不能添加子节点！");

return this;

}

@Override

public ConsistentComponent deleteChild(ConsistentComponent child) {

System.out.println("对不起，该节点是叶节点，没有子节点可以删除！");

return this;

}

@Override

public Set<ConsistentComponent> getChildren() {

System.out.println("对不起，该节点是叶节点，没有子节点！");

return null;

}

@Override

public String getName() {

return name;

}

public void setName(String name) { this.name = name; }

}

### (6) 实现效果

插入一些节点后的显示界面：



使用“删除”后的显示界面：



### (7) 模式的优点

* 定义了包含基本对象和组合对象的类层次结构，基本对象可以被组合成更复杂的组合对象，而这个组合对象又可以被组合。
* 简化了客户代码。
* 使得更加容易增加新类型的组件。
* 使设计变得更通用。
* **使得所有的构件类都有相同的接口**

### (8) 模式的不足

* 客户端对叶节点和枝节点是一致的，但叶节点并不具备Add和Remove的功能，因而对它们的实现是没有意义的

## 模式十：基于DI、ORM模式，使用Spring集成Hibernate框架，实现学生指导学生毕业设计

### 具体待解决的问题

具体学生和老师的关系：

学生（id和name），教师（id和name），学生可以选择一个教师做毕业设计指导教师(tutor)，教师可以指导多个学生毕业设计。

功能1：添加学生

功能2：修改学生

功能3：删除学生

功能4：添加教师

功能5：修改教师

功能6：删除教师

功能7：学生选择指导教师

功能8：学生修改指导教师

功能9：教师选择被指导的学生

功能10：教师修改被指导的学生

（1）功能达到上面的要求；

（2）利用JavaFx完成界面；

（3）利用Hibernate完成数据的CRUD。

### (2) 核心代码

1. **Spring配置文件**

<beans ~>

<bean id="teacherDao" class="cn.edu.scau.cmi.longting.hibernate.dao.TeacherDao"/>

<bean id="studentDao" class="cn.edu.scau.cmi.longting.hibernate.dao.StudentDao"/>

</beans>

1. **Hibernate的数据库配置文件**

hibernate.cfg.xml

<hibernate-configuration>

<session-factory>

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

<property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/softwarearchitecture?characterEncoding=UTF-8 </property>

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password"></property>

<property name="hibernate.show\_sql">true</property>

<property name="hibernate.format\_sql">true</property>

<property name="hibernate.hbm2ddl.auto">update</property>

<!-- 配置映射文件 -->

<mapping resource="cn/edu/scau/cmi/longting/hibernate/domain/Student.hbm.xml" />

<mapping resource="cn/edu/scau/cmi/longting/hibernate/domain/Teacher.hbm.xml" />

</session-factory>

</hibernate-configuration>

1. **Hibernate的ORM映射文件**

hibernateApplicationContext.xml

<beans ~>

<!-- 访问数据库的各种dao -->

<bean id=*"studentDAO"* class=*"cn.deu.scau.cmi.longting.hibernate.dao.StudentDao"/*>

<bean id=*"teacherDAO"* class=*"cn.deu.scau.cmi.longting.hibernate.dao.TeacherDAO"/*>

</beans>

Student.hbm.xml

<hibernate-mapping>

<class name=*"cn.edu.scau.cmi.longting.hibernate.domain.Student"* table=*"student"* catalog=*"softwarearchitecture"*>

<id name=*"id"* type=*"long"*>

<column name=*"id"*/>

<generator class=*"identity"*/>

</id>

<many-to-one name=*"teacher"* class=*"cn.edu.scau.cmi.longting.hibernate.domain.Teacher"* fetch=*"select"*>

<column name=*"tutor"*/>

</many-to-one>

<property name=*"name"* type=*"string"*>

<column name=*"name"* not-null=*"true"* unique=*"true"*/>

</property>

</class>

</hibernate-mapping>

Teacher.hbm.xml

<hibernate-mapping>

<class name=*"cn.edu.scau.cmi.longting.hibernate.domain.Teacher"* table=*"teacher"* catalog=*"softwarearchitecture"*>

<id name=*"id"* type=*"long"*>

<column name=*"id"*/>

<generator class=*"identity"*/>

</id>

<property name=*"name"* type=*"string"*>

<column name=*"name"* not-null=*"true"*/>

</property>

<set name=*"students"* inverse=*"true"*>

<key> <column name=*"tutor"*/> </key>

<one-to-many class=*"cn.edu.scau.cmi.longting.hibernate.domain.Student"* />

</set>

</class>

</hibernate-mapping>

1. **实体类**

Student.java

public class Student implements Serializable{

private Long id;

private String name;

private Teacher teacher;

public Student() { }

public Student(Long id, String name) {

this.id = id;

this.name = name;

}

public Student(Long id, String name, Teacher teacher) {

this.id = id;

this.name = name;

this.teacher = teacher;

}

----------getter() 和 setter() ----------

}

Teacher.java

public class Teacher implements Serializable{

private Long id;

private String name;

private Set students = new HashSet(0);

public Teacher() { }

public Teacher(Long id, String name) {

this.id = id;

this.name = name;

}

----------getter() 和 setter() ----------

}

1. **其他类**

StudentDAO.java

public class StudentDao {

private static final Logger log = LoggerFactory.getLogger(StudentDao.class);

private static final String NAME = "name";

private Session session;

public void save(Student student) {

try {

this.getSession();

Transaction transaction = session.beginTransaction();

session.save(student);

transaction.commit();

} catch (RuntimeException re) {

throw re;

}

}

public void delete(Student student) {

log.debug("deleting Student instance");

try {

this.getSession();

Transaction transaction = session.beginTransaction();

session.delete(student);

transaction.commit();

log.debug("delete successful");

} catch (Exception e) {

log.error("delete faild", e);

throw e;

}

}

public Student findById(Long id) {

log.debug("getting Student instance with id: " + id);

try {

Student instance =

(Student) session.get("cn.edu.scau.cmi.longting.hibernate.domain.Student", id);

return instance;

} catch (RuntimeException re) {

log.error("get failed", re);

throw re;

}

}

public List findByExample(Student student) {

log.debug("finding Student instance by example");

try {

List results =

getSession().createCriteria("cn.edu.scau.cmi.longting.hibernate.domain.Student")

.add(Example.create(student)).list();

log.debug("find by example successful, result size: " + results.size());

return results;

} catch (RuntimeException re) {

log.error("find by example failed", re);

throw re;

}

}

public List findByProperty(String propertyName, Object value) {

log.debug("finding Student instance with property: " + propertyName + ", value: " + value);

try {

String queryString = "from Student as model where model."+propertyName+"= ?";

Query queryObject = getSession().createQuery(queryString);

queryObject.setParameter(0, value);

return queryObject.list();

} catch (RuntimeException re) {

log.error("find by property name failed", re);

throw re;

}

}

public List findByName(Object name) {

return findByProperty(NAME, name);

}

public List findAll() {

log.debug("finding all Student instances");

try {

String queryString = "from Student";

Query queryObject = getSession().createQuery(queryString);

return queryObject.list();

} catch (RuntimeException re) {

log.error("find all failed", re);

throw re;

}

}

public Student merge(Student student) {

log.debug("merging Student instance");

try {

this.getSession();

Transaction transaction = session.beginTransaction();

Student result = (Student) session.merge(student);

transaction.commit();

log.debug("merge successful");

return result;

} catch (RuntimeException re) {

log.error("merge failed", re);

throw re;

}

}

public void attachDirty(Student student) {

log.debug("attaching dirty Student instance");

try {

this.getSession();

Transaction transaction = session.beginTransaction();

session.saveOrUpdate(student);

transaction.commit();

log.debug("attach successful");

} catch (RuntimeException re) {

log.error("attach failed", re);

throw re;

}

}

public void attachClean(Student student) {

log.debug("attaching clean Student instance");

try {

this.getSession();

Transaction transaction = session.beginTransaction();

session.buildLockRequest(LockOptions.NONE).lock(student);

transaction.commit();

log.debug("attach successful");

} catch (RuntimeException re) {

log.error("attach failed", re);

throw re;

}

}

public Session getSession() {

session = ScauCmiHibernateSessionFactoryUtil.getSession();

return session;

}

}

TeacherDAO.java

public class TeacherDao {

private static final Logger log = LoggerFactory.getLogger(TeacherDao.class);

private static final String NAME = "name";

private Session session;

public void save(Teacher teacher) {

this.getSession();

Transaction Transaction = session.beginTransaction();

session.save(teacher);

Transaction.commit();

}

public void delete(Teacher teacher) {

log.debug("deleting Student instance");

try {

this.getSession();

Transaction transaction = session.beginTransaction();

session.delete(teacher);

transaction.commit();

log.debug("delete successful");

} catch (Exception e) {

log.error("delete faild", e);

throw e;

}

}

public Teacher findById(Long id) {

log.debug("getting Teacher instance with id: " + id);

try {

Teacher instance = (Teacher) getSession().get("cn.edu.scau.cmi.longting.hibernate.domain.Teacher", id);

return instance;

} catch (RuntimeException re) {

log.error("get failed", re);

throw re;

}

}

public List findByExample(Teacher teacher) {

log.debug("finding Teacher instance by example");

try {

List results = getSession().createCriteria("cn.edu.scau.cmi.longting.hibernate.domain.Teacher")

.add(Example.create(teacher)).list();

log.debug("find by example successful, result size: " + results.size());

return results;

} catch (RuntimeException re) {

log.error("find by example failed", re);

throw re;

}

}

public List findByProperty(String propertyName, Object value) {

log.debug("finding Teacher instance with property: " + propertyName + ", value: " + value);

try {

String queryString = "from Teacher as model where model."+propertyName+"= ?";

Query queryObject = getSession().createQuery(queryString);

queryObject.setParameter(0, value);

return queryObject.list();

} catch (RuntimeException re) {

log.error("find by property name failed", re);

throw re;

}

}

public List findByName(Object name) {

return findByProperty(NAME, name);

}

public List findAll() {

log.debug("finding all Teacher instances");

try {

String queryString = "from Teacher";

Query queryObject = getSession().createQuery(queryString);

return queryObject.list();

} catch (RuntimeException re) {

log.error("find all failed", re);

throw re;

}

}

public Teacher merge(Teacher teacher) {

log.debug("merging Teacher instance");

try {

this.getSession();

Transaction transaction = session.beginTransaction();

Teacher result = (Teacher) session.merge(teacher);

transaction.commit();

log.debug("merge successful");

return result;

} catch (RuntimeException re) {

log.error("merge failed", re);

throw re;

}

}

public void attachDirty(Teacher teacher) {

log.debug("attaching dirty Student instance");

try {

this.getSession();

Transaction transaction = session.beginTransaction();

session.saveOrUpdate(teacher);

transaction.commit();

log.debug("attach successful");

} catch (RuntimeException re) {

log.error("attach failed", re);

throw re;

}

}

public void attachClean(Teacher teacher) {

log.debug("attaching clean Student instance");

try {

this.getSession();

Transaction transaction = session.beginTransaction();

session.buildLockRequest(LockOptions.NONE).lock(teacher);

transaction.commit();

log.debug("attach successful");

} catch (RuntimeException re) {

log.error("attach failed", re);

throw re;

}

}

public Session getSession() {

session = ScauCmiHibernateSessionFactoryUtil.getSession();

return session;

}

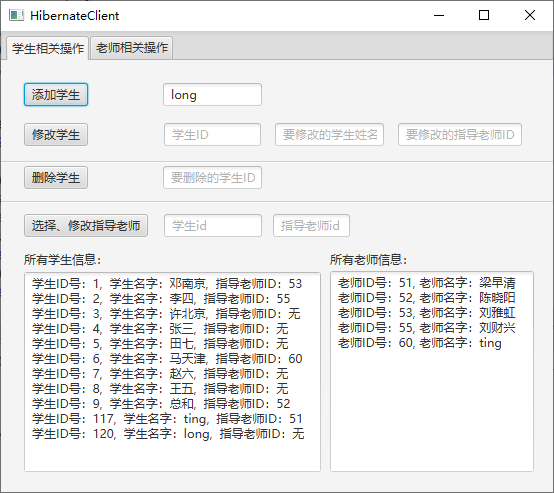
}

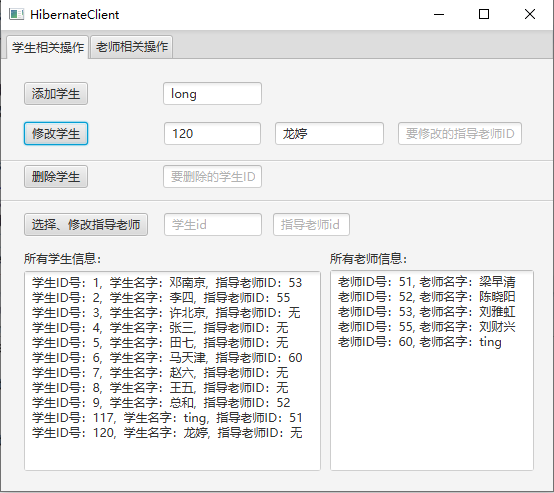
### (3) 实现效果

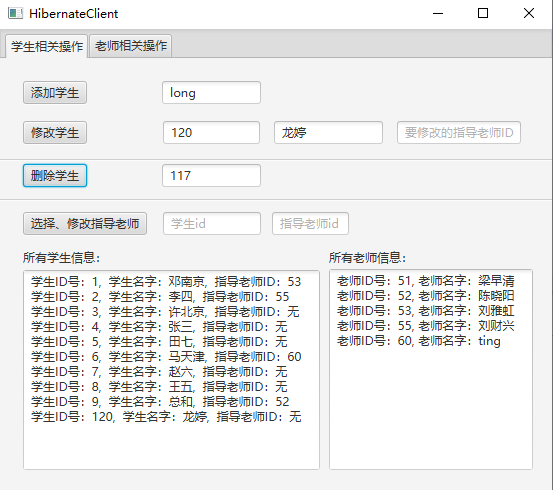
1. **初始界面，显示数据库中所有的学生、老师信息**

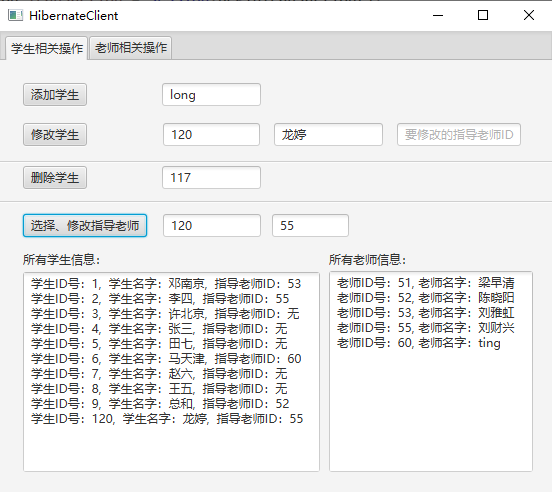


1. **下面依次展示增、删、改学生信息，以及修改学生对应的指导老师**

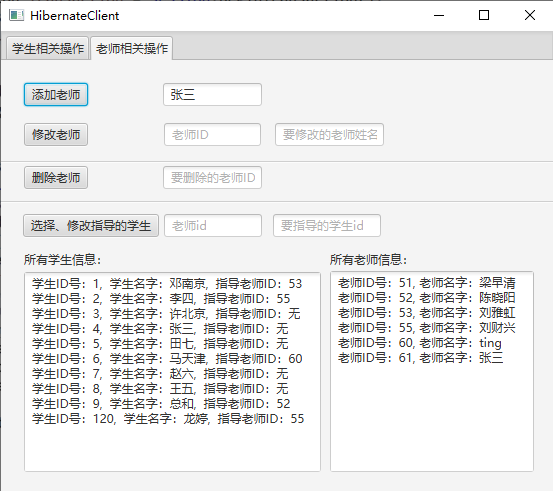


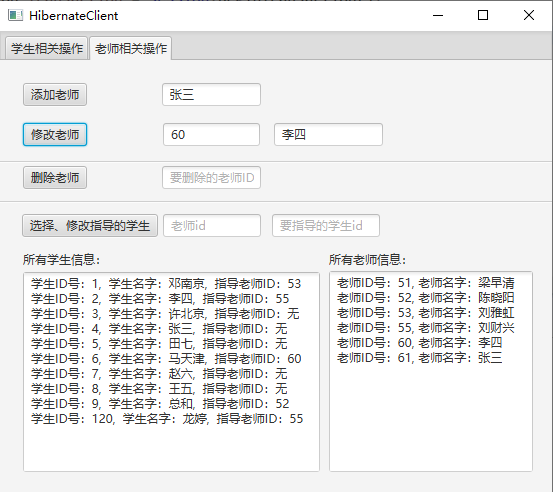


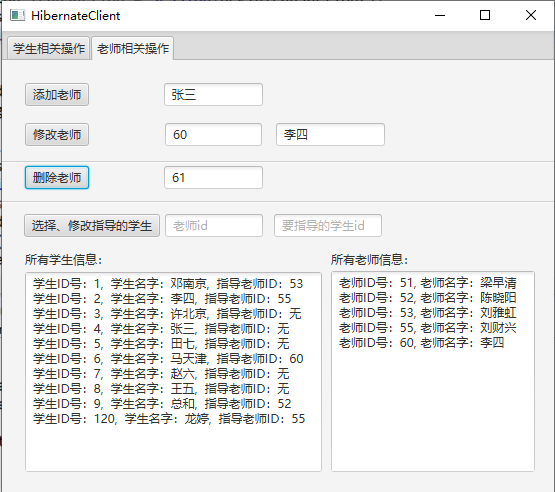


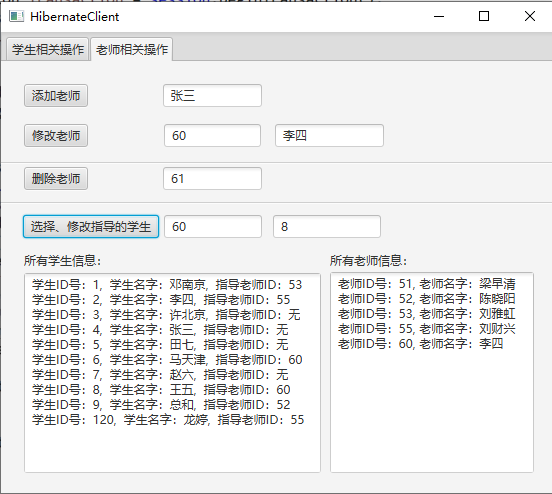


1. **下面依次展示增、删、改老师信息，以及修改老师对应指导的学生**

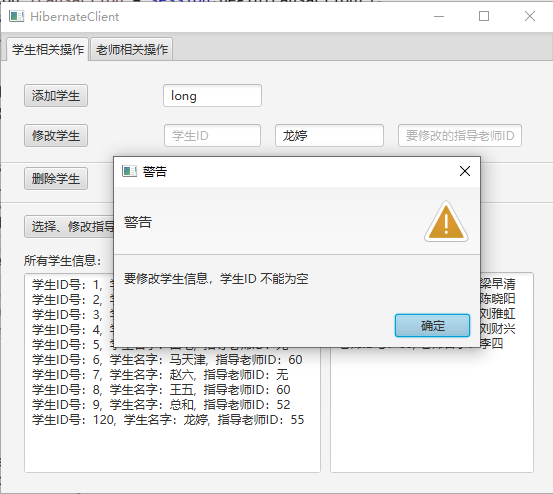








1. **具体操作时，会适当有一些出错提示：**



## 综合性实验的心得体会

本次的综合性实验是整个学期的实验1~7的内容的整合，在设计整合界面的时候，没有很多的问题，但是这次的综合性实验报告编写完后，让我有很大的收获。

本综合性实验报告中整合了主要的设计模式：接口模式、单例模式、多例模式、简单工厂模式、工厂模式、抽象工厂模式、类适配器模式、对象适配器模式、安全组合模式、一致性组合模式和基于DI、ORM模式的使用Spring 集成Hibernate框架，实现学生指导学生毕业设计。在每一个模式中，分点阐述了该模式解决问题的类型、解决设计思路、该模式的优缺点以及举例了该模式典型的例题，对该例题进行UML设计、核心代码和实现的效果。

其中在整合每一个模式的解决问题类型和优缺点的时候，让我更加深入理解和掌握每个模式的使用场景和利弊。让我更加知道了设计模式的强大，这是前人解决问题的经验总结，同时这也是一个难点，我以后要注意运用和学习这些宝贵的经验。