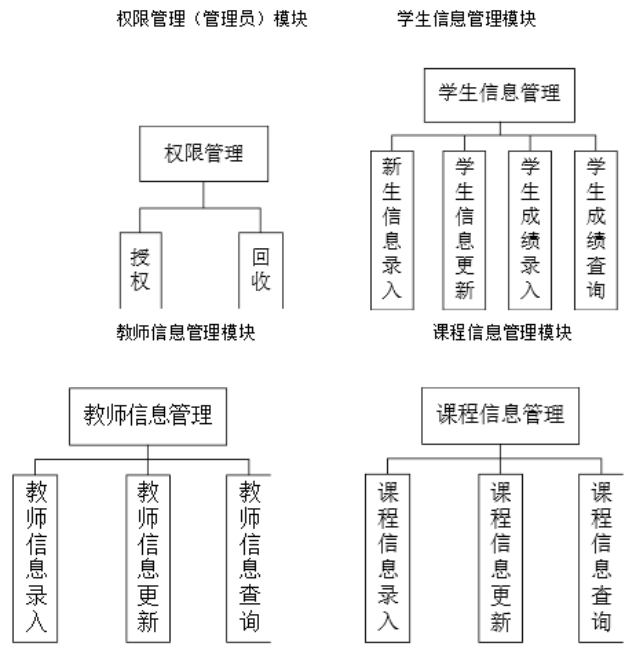
内容安排

详细设计要求

详细设计要求-系统结构关系

- 功能结构图：
- 描述模块、子模块的结构

每个模块应采用IPO（输入-处理-输出图）形式说明该模块具有的功能

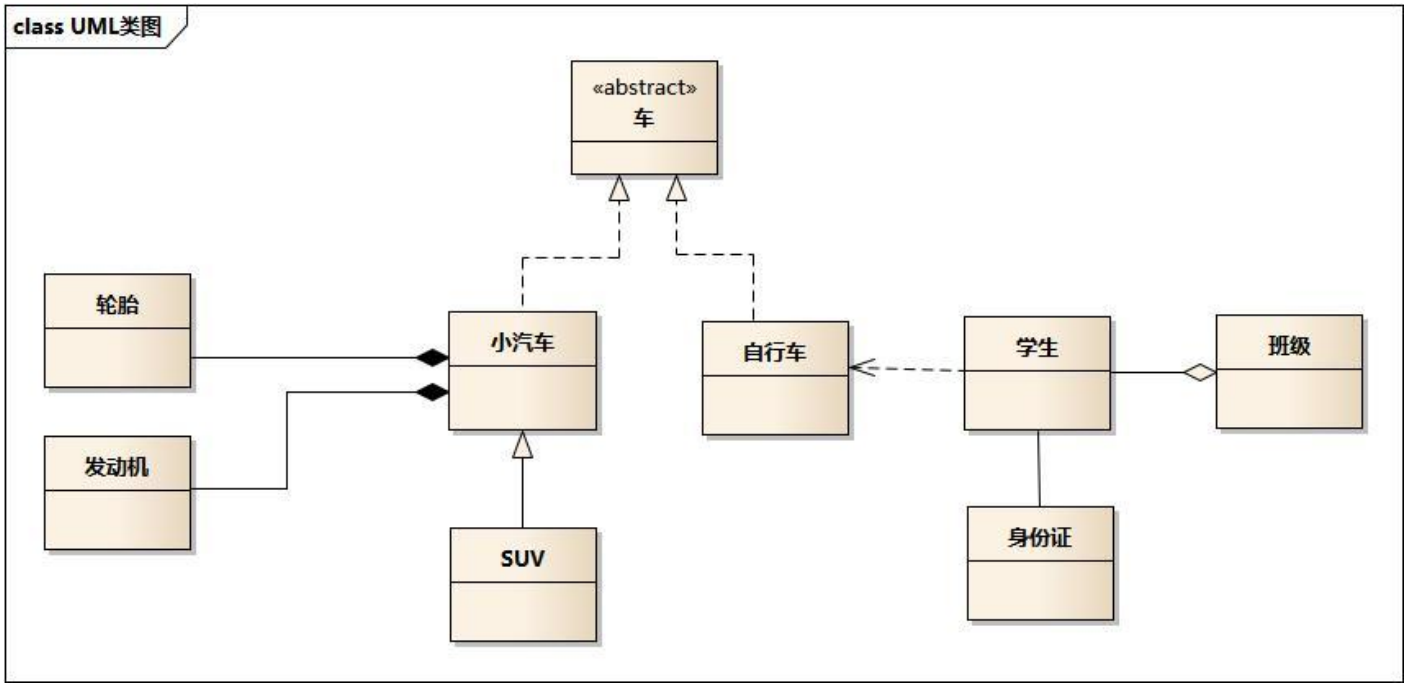


详细设计要求-输入输出模块

- 确定程序输入、输出
- 输入：类型、格式、输入方式、有效范围
- 输出：输出形式，数值格式范围等

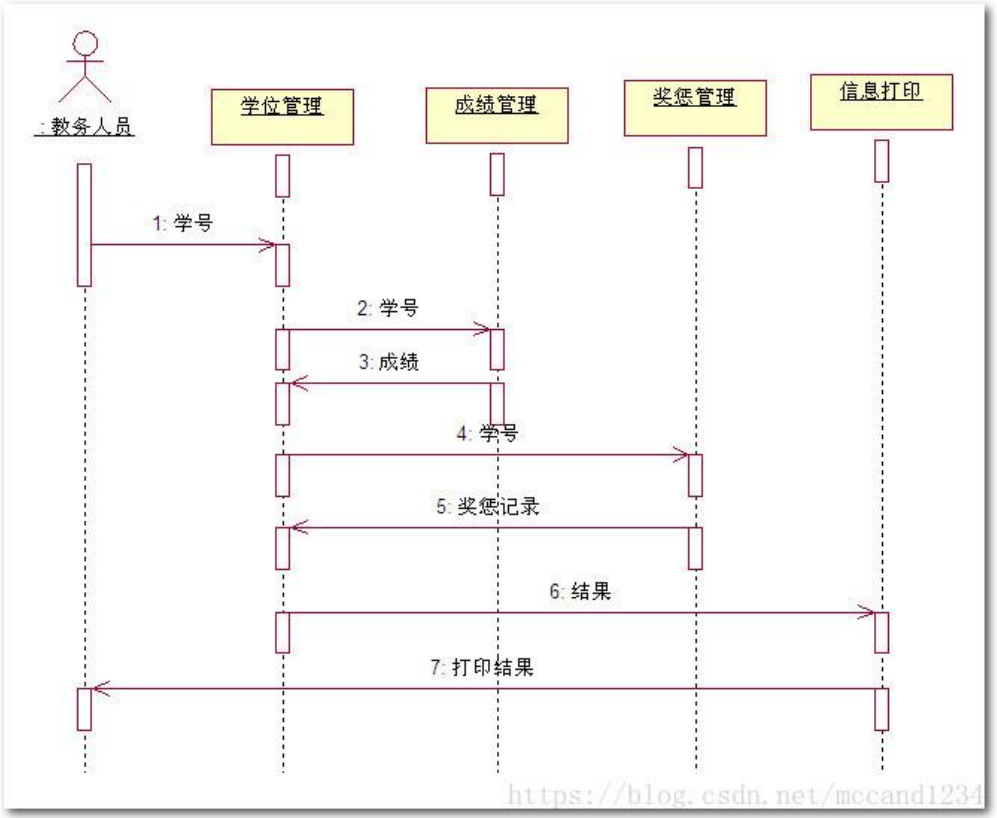
详细设计要求-类图设计

- 类图：
- 细化各个功能子模块



详细设计要求-复杂流程或算法描述

- 流程图、类序列图、
- 描述程序中所选用的算法或程序逻辑。



详细设计要求-接口设计

- 接口定义
- 程序内部各子模块之间参数类型和调用方式。
- 可采用图表形式进行说明。

接口描述：用户登陆成功后，或进入个人中心时会获取一次用户信息

URI	方法
/userinfo	GET

请求参数

名称	必填	备注
id	是	用户id

响应参数

名称	类型	备注
id	String	用户id
name	String	姓名，例：张三
age	String	年龄，例：20

详细设计要求-测试设计

- 功能测试
- 对各个功能子模块设计其测试用例

[权限管理系统]项目测试用例

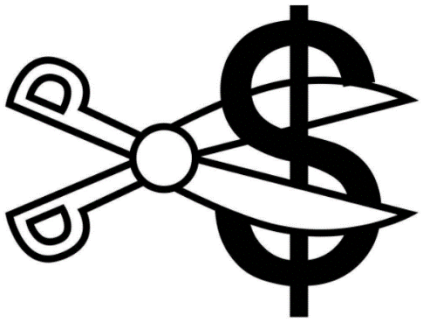
系统模块	功能点	用例编号	用例说明	前置条件	输入	预期结果	测试结果	失败原因
	2.1 增加用户	2.1.1	新增用户动作	系统管理员已登录系统	系统管理员在用户管理页面中单击“新增用户”按钮	成功进入到“新增用户”页面		
		2.1.2	必填项	系统管理员已登录系统，并进入到“新增用户”页面	令某项必填数据为空，其余数据正常填写，提交（例如，将用户名置空，提交）	提示“请输入×××”（例如：请输入用户名）		
		2.1.3	数据有效性	系统管理员已登录系统，并进入到“新增用户”页面	在本框中输入无效数据，提交（例如，输入非法E-mail：cxw521816.hxf999.com,提交）	提示“请输入合法的××”(例如：请输入合法的E-mail地址)		
		2.1.4	全部置空	系统管理员已登录系统，并进入到“新增用户”页面	令所有文本框为空，提交	提示“请输入用户名”		
		2.1.5	输入范围	系统管理员已登录系统，并进入到“新增用户”页面	在文本框中输入长度为999的文本提交（例如，在用户名文本框中输入长度为999的文本）	输入框本身应有输入范围限制，多出“最大输入位数”部分应自动舍弃		
		2.1.6	提交新增用户信息	系统管理员已登录系统	在“新增用户”页面各项文本框中输入有效数据，提交	系统提示“新增用户成功”，成功添加系统用户		
	2.2 删除用户	2.2.1	删除用户动作	系统管理员已登录系统，并选择要删除的用户	系统管理员单击“删除”按钮	弹出系统提示框“是否真的要删除该用户？”，并出现两个按钮“是”和“否”		
		2.2.2	取消删除动作	系统管理员已登录系统，并单击“删除”按钮	在弹出的系统提示框中，单击按钮“否”	删除用户动作取消		

编码规范

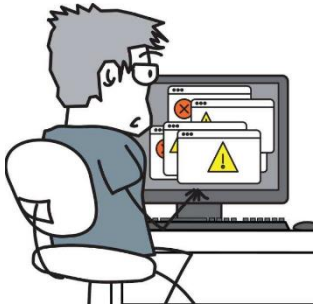
编码规范的重要性

- 规范的代码可以促进团队合作

- 规范的代码可以降低维护成本



- 规范的代码可以减少bug处理



- 规范的代码有助于代码审查

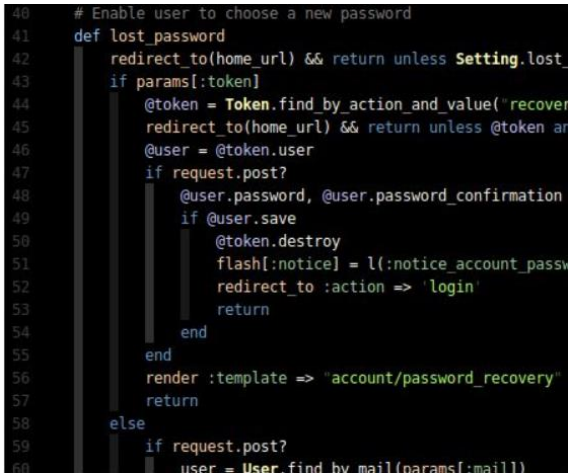
- 养成代码规范的习惯非常重要

如何做到代码规范？

- 排版
- 注释
- 标识符命名
- 可读性
- 变量、结构
- 函数、过程

排版

- 1.程序块要采用缩进对齐



```
# Enable user to choose a new password
def lost_password
    redirect_to(home_url) && return unless Setting.lost_
    if params[:token]
        @token = Token.find_by_action_and_value("recover
        redirect_to(home_url) && return unless @token an
        @user = @token.user
        if request.post?
            @user.password, @user.password_confirmation
            if @user.save
                @token.destroy
                flash[:notice] = l(:notice_account_passw
                redirect_to :action => 'login'
                return
            end
        end
        render :template => "account/password_recovery"
        return
    else
        if request.post?
            user = User.find_by_mail(params[:mail])
```

- 2.不要一行写多条语句

如 rect.length = 0, rect.width = 0;

改为：rect.length = 0;

rect.width = 0;

- 3.相对独立的代码块之间要加空行

```
if(!valid_ni(ni))
{
    … // program code block A
}

repssn_ni = ssn_data[index].ni;
index = index + 1;
// code block B
```

- 4. 操作符前后加空格

```
a = b + c;
b = c ^ 2;
```

注释

- 注释的原则是有助于对程序的理解，在该加的地方加，注释语言必须准确、易懂、简洁

- TODO注释

- 用代码来阐述

```
//check to see if the employee is eligible
    for full benefits
if(employee.flags & HOURLY_FLAG)
    && (employee.age >65))
```

还是这个？

```
if(employee.isEligibleForFullBenefits())
```

能用函数和变量名说明时就不要用注释

标识符命名

- 标识符的命名要清晰、明了、有明确的含义

- 名副其实

```
int d; // 消逝的时间, 以日计
```

名称d什么也没有说明，依赖于注释解释。我们应选择如下的名称：

```
int   elapsedTimeInDays;
int   daysSinceCreation;
int   daysSinceModification;
int   fileAgeIndays;
```

- 做有意义的区分

```
public static void copyChars(char a1[], char a2[])
{
    for (int i =0；i < a1.length; i++)
    {
        a2[i] = a1[i];
    }
}
```

这里a1改为source，a2改为destination会更好

可读性

- 注意运算符的优先级，并用括号说明正确的表达顺序，避免使用默认的优先级产生误读。

word = high < < 8 | low ⟶ word = (high < < 8) | low

if (a | b && a & c) ⟶ if ((a | b) && (a & c))

- 避免使用不易解释的数字，用有意义的标识来代替

```
if(Trunk[index]. trunk_state == 0）{
    Trunk[index].trunk_state = 1;
}
```

改为：

```
#define  TRUNK_IDLE   0
#define   TRUNK_BUSY   1
if(Trunk[index]. trunk_state == TRUNK_IDEL）{
    Trunk[index].trunk_state =  TRUNK_BUSY;
}
```

- 避免使用难懂的技巧性高的语句

*stat_poi ++ += 1; ⟶

```
*stat_poi + = 1;
stat_poi ++ ;
```

变量、结构

- 去掉没有必要的公共变量，降低耦合度

- 构造仅有一个模块或函数可以修改、创建、而其余有关模块或函数只访问的公共变量，防止多处创建、修改同一公共变量

函数、过程

- 防止函数的参数作为工作变量

```
 void sum_data( unsigned int num, int *data, int* sum)
{
    unsigned int count;
    *sum=0;
    for (count = 0; counter < num; counter ++)
    {
          *sum + = data[count]; //sum 成为工作变量，不太好
    }
}
```

有可能错误改变地址参数sum中的内容

小结

- 介绍编码的基本规范
  - ✓ 排版
  - ✓ 注释
  - ✓ 标识符命名
  - ✓ 可读性
  - ✓ 变量、结构
  - ✓ 函数、过程

关于设计模式的应用

什么是设计模式?

每一个设计模式描述一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次一次地使用该方案而不必做重复劳动。

什么是软件设计模式?

- ✓ 广义讲，软件设计模式是可解决一类软件问题并能重复使用的软件设计方案
- ✓ 狭义讲，设计模式是对被用来在特定场景下解决一般设计问题的类和相互通信的对象的描述。是在类和对象的层次描述的可重复使用的软件设计问题的解决方案
- ✓ 模式体现的是程序整体的构思，所以有时候它也会出现在分析或者是概要设计阶段
- ✓ 模式的核心思想是通过增加抽象层，把变化部分从那些不变部分里分离出来

设计模式的起源

软件领域的设计模式起源于建筑学。

1977年，建筑大师Alexander出版了《A Pattern Language：Towns, Building, Construction》一书。受Alexander著作的影响 ，Kent Beck和Ward Cunningham在1987年举行的一次面向对象的会议上发表了论文：《在面向对象编程中使用模式》。

软件领域的设计模式著作：GOF

设计模式领域最具影响力著作是**Erich Gamma**、**Richard Helm**、**Ralph Johnson**和**John Vlissides**在1994年合作出版的著作：《Design Patterns：Elements of Reusable Object-Oriented Software》（中译本**《设计模式：可复用面向对象软件的基础》**），该书被广大喜爱者昵称为GOF（Gang of Four）之书，被认为是学习设计模式的必读著作、奠基之作。

https://item.jd.com/10057319.html

秦小波. 设计模式之禅（第2版）. 机械工业出版社, ISBN: 9787111437871, 2014.

https://item.jd.com/11414555.html

模式的基本要素

- ✓ **模式名称(Pattern Name)**
- ✓ **问题(Problem)**：描述应该在何时使用模式。解释了设计问题和问题存在的前因后果,可能还描述模式须满足的先决条件
- ✓ **解决方案(Solution)**：描述了设计的组成成分、相互关系及各自的职责和协作方式。模式就像一个模板，可应用于多种场合，所以解决方案并不描述一个具体的设计或实现，而是提供设计问题的抽象描述和解决问题所采用的元素组合（类和对象）
- ✓ **效果(consequences)**：描述模式的应用效果及使用模式应权衡的问题

如何描述设计模式

- **模式名和分类**
- **意图**：设计模式是做什么的？它的基本原理和意图是什么？它解决的是什么样的特定设计问题？
- **动机**：说明一个设计问题以及如何用模式中的类、对象来解决该问题的特定情景
- **适用性**：什么情况下可以使用该设计模式？该模式可用来改进哪些不良设计？如何识别这些情况？
- **结构**：采用对象建模技术对模式中的类进行图形描述

描述设计模式（续）

◆**参与者**：指设计模式中的类 和/或 对象以及它们各自的职责

◆**协作**：模式的参与者如何协作以实现其职责

◆**效果**：模式如何支持其目标？使用模式的效果和所需做的权衡取舍？系统结构的哪些方面可以独立改变？

◆**实现**：实现模式时需了解的一些提示、技术要点及应避免的缺陷，以及是否存在某些特定于实现语言的问题

◆**代码示例**：用来说明怎样实现该模式的代码片段

◆**相关模式**：与这个模式紧密相关的模式有哪些？其不同之处是什么？这个模式应与哪些其他模式一起使用？

单件（Singleton）模式的由来

- 单件模式的实例较为普遍：
  - 系统中只能有一个窗口管理器
  - 系统中只能有一个文件系统
  - 一个数字滤波器只能有一个A/D转换器
  - 一个会计系统只能专用于一个公司

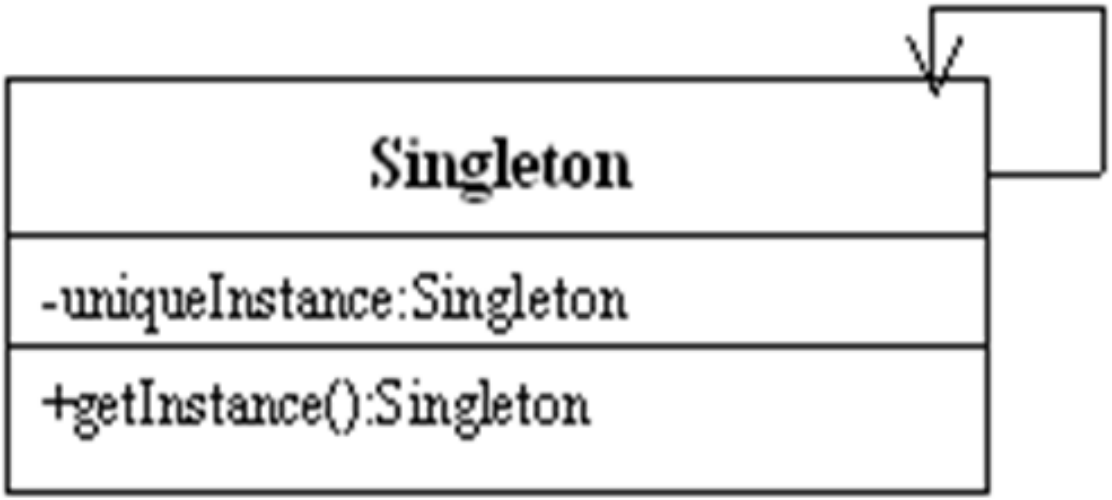
单件模式的由来

- 如何才能保证一个类只有一个实例并且这个实例易于被访问呢？一个全局变量使得一个对象可以被访问，但它不能防止你实例化多个对象
- 一个更好的办法是，让类自身负责保存它的唯一实例。这个类可以保证没有其他实例可以被创建，并且它可以提供一个访问该实例的方法

单件模式的意图和适用性

- **意图**：保证一个类仅有一个实例,并提供一个全局访问点
- **适用场合**：
  - 当类只能有一个实例而且用户可以从一个众所周知的访问点访问它时
  - 当这个唯一实例是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展实例时

单件(单例)模式的结构


Singleton
-uniqueInstance:Singleton
+getInstance():Singleton

模式的结构的描述与使用

1. 单件类（Singleton）：Moon.java

```
public class Moon{
      private static Moon  uniqueMoon;
      double radius;
      double distanceToEarth;
      private Moon(){
            uniqueMoon=this;
            radius=1738;
            distanceToEarth=363300;
      }
      public static synchronized Moon getMoon(){
             if(uniqueMoon==null){
                   uniqueMoon=new Moon();
             }
             return uniqueMoon;
      }
      public String show(){
            String s="月亮的半径是"+radius+"km,距地球是"+distanceToEarth+"km";
            return s;
      }
}
```

模式的结构的描述与使用

2. 应用 Application.java

```
import javax.swing.*;
import java.awt.*;
public class Application{
    public static void main(String args[]){
          MyFrame f1=new MyFrame("张三看月亮");
          MyFrame f2=new MyFrame( "李四看月亮");
          f1.setBounds(10,10,360,150);
          f2.setBounds(370,10,360,150);
          f1.validate();
          f2.validate();
    }
}
class  MyFrame extends JFrame{
    String str;
    MyFrame(String title){
        setTitle(title);
        Moon moon=Moon.getMoon();
        str=moon.show();
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        setVisible(true);
        repaint();
   }
    public void paint(Graphics g){
          super.paint(g);
          g.setFont(new Font("宋体",Font.BOLD,14));
          g.drawString(str,5,100);
    }
}
```

抽象工厂模式的由来

- 在软件系统中，经常面临“一系列相互依赖对象”的创建工作，由于需求变化，这“一系列相互依赖的对象”也要改变，如何应对这种变化呢？如何像工厂模式一样绕过常规的”new”，提供一种“封装机制”来避免客户程序和这种“多系列具体对象创建工作”的紧耦合？
- 一种说法：可以将这些对象一个个通过工厂模式来创建。但是，既然是一系列相互依赖的对象，它们是有联系的，每个对象都这样解决，如何保证他们的联系呢？

抽象工厂模式的由来

- 实例：Windows桌面主题，当更换一个桌面主题的时候，系统的开始按钮、任务栏、菜单栏、工具栏等都变了，而且是一起变的，他们的色调都很一致，类似这样的问题如何解决呢？

- 应用抽象工厂模式，是一种有效的解决途径

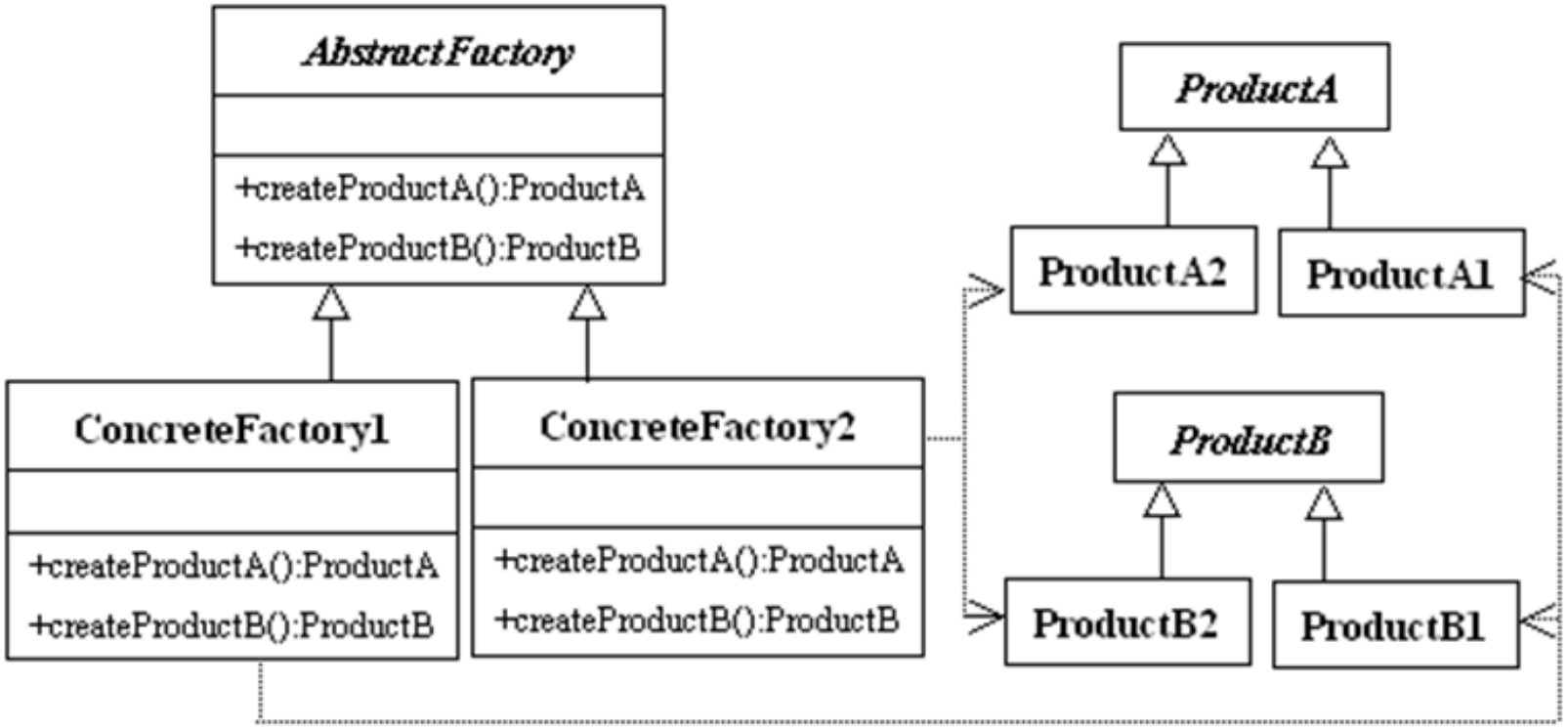
抽象工厂模式的意图

◆**意图：**提供一个创建一系列相关或相互依赖对象的接口，而无需指定他们具体的类

◆**适用场合**

- ◆一个系统独立于其产品创建、组合和表示时
- ◆一个系统由多个产品系列中的一个来配置时
- ◆强调一系列相关产品对象的设计以便进行联合时
- ◆提供一个产品类库，只想显示其接口而非实现时

抽象工厂模式的结构


AbstractFactory
+createProductA():ProductA
+createProductB():ProductB
ConcreteFactory1
+createProductA():ProductA
+createProductB():ProductB
ConcreteFactory2
+createProductA():ProductA
+createProductB():ProductB
ProductA
ProductA2
ProductA1
ProductB
ProductB2
ProductB1

模式的结构的描述与使用

1. 抽象产品（Product）：

UpperClothes.java

```
public abstract class UpperClothes{
    public abstract int getChestSize();
    public abstract int getHeight();
    public abstract String getName();
}
```

Trousers.java

```
public abstract class Trousers{
    public abstract int getWaistSize();
    public abstract int getHeight();
    public abstract String getName();
}
```

模式的结构的描述与使用

2. 具体产品（ConcreteProduct）_1:

WesternUpperClothes.java

```
public class WesternUpperClothes extends UpperClothes{
   private int chestSize;
   private int height;
   private String name;
   WesternUpperClothes(String name,int chestSize,int height){
        this.name=name;
        this.chestSize=chestSize;
        this.height=height;
   }
   public int getChestSize(){
        return chestSize;
   }
   public int getHeight(){
        return height;
   }
   public String getName(){
        return name;
   }
}
```

模式的结构的描述与使用

2. 具体产品（ConcreteProduct）_2:

CowboyUpperClothes.java

```
public class CowboyUpperClothes extends UpperClothes{
   private int chestSize;
   private int height;
   private String name;
   CowboyUpperClothes(String name,int chestSize,int height){
        this.name=name;
        this.chestSize=chestSize;
        this.height=height;
   }
   public int getChestSize(){
        return chestSize;
   }
   public int getHeight(){
        return height;
   }
   public String getName(){
        return name;
   }
}
```

模式的结构的描述与使用

2. 具体产品（ConcreteProduct）_3:

WesternTrousers.java

```
public class WesternTrousers extends Trousers{
   private int waistSize;
   private int height;
   private String name;
   WesternTrousers(String name,int waistSize,int height){
       this.name=name;
       this.waistSize=waistSize;
       this.height=height;
   }
    public int getWaistSize(){
       return waistSize;
   }
   public int getHeight(){
       return height;
   }
   public String getName(){
       return name;
   }
}
```

模式的结构的描述与使用

2. 具体产品（ConcreteProduct）_4: CowboyTrousers.java

```
public class CowboyTrousers extends Trousers{
   private int waistSize;
   private int height;
   private String name;
   CowboyTrousers(String name,int waistSize,int height){
        this.name=name;
        this.waistSize=waistSize;
        this.height=height;
   }
    public int getWaistSize(){
        return waistSize;
   }
   public int getHeight(){
        return height;
   }
   public String getName(){
        return name;
   }
}
```

模式的结构的描述与使用

3．抽象工厂（AbstractFactory）:ClothesFactory.java

```
public abstract class ClothesFactory{
    public abstract UpperClothes createUpperClothes(int chestSize,int height);
    public abstract Trousers createTrousers(int waistSize,int height);
}
```

模式的结构的描述与使用

4. 具体工厂（ConcreteFactory）：

BeijingClothesFactory.java

```
public class BeijingClothesFactory extends ClothesFactory {
    public UpperClothes createUpperClothes(int chestSize,int height){
        return new WesternUpperClothes("北京牌西服上衣",chestSize,height);
    }
    public Trousers createTrousers(int waistSize,int height){
        return new WesternTrousers("北京牌西服裤子",waistSize,height);
    }
}
```

ShanghaiClothesFactory.java

```
public class ShanghaiClothesFactory extends ClothesFactory {
    public UpperClothes createUpperClothes(int chestSize,int height){
        return new WesternUpperClothes("上海牌牛仔上衣",chestSize,height);
    }
    public Trousers createTrousers(int waistSize,int height){
        return new WesternTrousers("上海牌牛仔裤",waistSize,height);
    }
```

模式的结构的描述与使用

5. 应用_1: Shop.java

```
public class Shop{
    UpperClothes cloth;
    Trousers trouser;
    public void giveSuit(ClothesFactory factory,int chestSize,int waistSize,int height){
       cloth=factory.createUpperClothes(chestSize,height);
       trouser=factory.createTrousers(waistSize,height);
       showMess();
    }
    private void showMess(){
       System.out.println("<套装信息>");
       System.out.println(cloth.getName()+":");
       System.out.print("胸围:"+cloth.getChestSize());
       System.out.println("身高:"+cloth.getHeight());
       System.out.println(trouser.getName()+":");
       System.out.print("腰围:"+trouser.getWaistSize());
       System.out.println("身高:"+trouser.getHeight());
    }
}
```

模式的结构的描述与使用

5. 应用_2: Application.java

```
public class Application{
    public static void main(String args[]){
        Shop shop=new Shop();
        ClothesFactory factory=new BeijingClothesFactory();
        shop.giveSuit(factory,110,82,170);
        factory=new ShanghaiClothesFactory();
        shop.giveSuit(factory,120,88,180);
    }
}
```

抽象工厂模式的优点

- 抽象工厂模式可以为用户创建一系列相关的对象，使得用户和创建这些对象的类脱耦。
- 使用抽象工厂模式可以方便的为用户配置一系列对象。用户使用不同的具体工厂就能得到一组相关的对象，同时也能避免用户混用不同系列中的对象。
- 在抽象工厂模式中，可以随时增加“具体工厂”为用户提供一组相关的对象。