



第六讲 面向对象设计与UML建模 (下)

饶元

社会智能与复杂数据处理实验室
西安交通大学软件学院

2017年4月

主要内容

- 面向对象设计基础
- 面向对象的建模思想与方法
- **面向对象的设计目标与原则**
- 基于模式的事件处理机制
- 软件重构与设计优化

主要参考文献

- 面向对象设计讲义2012版，饶元；
- 软件设计原则培训，浪潮公司；
- CMU面向对象设计课程讲义

设计的总体目标

- **为用户而设计(design for users)**
 - 定义用户需求，确定系统边界
- **为分工而设计 (for division of labor)**
 - 更小的接口，更易于区分
- **为变化而设计 (design for change)**
 - 减少依赖使得变化更容易
 - 变化导致代码臭味的传播 (rippling effect)
- **为易理解而设计 (Design for understandability)**
 - 依赖导致代码复杂，难理解
- **为重用而设计 (for reuse)**



软件可维护性设计

• 可维护性设计目标

- 可扩展性extensibility;
加入新模块，不影响原有模块
- 灵活性flexibility;
修改一个模块，不影响其他模块
- 可插入性plug ability
可以很容易的去掉一个模块，更换一个模块，加入一个新模块，而不影响其他模块。

1、单一职责原则（SRP）

- 一个类，最好只做一件事，只有一个引起它变化的原因。
- There should never be more than one reason for a class to change
- 单一职责，强调的是职责的分离，在某种程度上对职责的理解，构成了不同类之间耦合关系的设计关键，因此单一职责原则或多或少成为设计过程中一个必须考虑的基础性原则。
- 所谓职责，我们可以理解为功能，就是设计的这个类功能应该只有一个，而不是两个或更多。也可以理解为引用变化的原因，当你发现有两个变化会要求我们修改这个类，那么你就考虑撤分这个类了。因为职责是变化的一个轴线，当需求变化时，该变化会反映类的职责的变化。

面向对象设计原则

面向对象设计易用性原则：

- 1、易维护性原则
- 2、可扩展性原则
- 3、可重用性原则

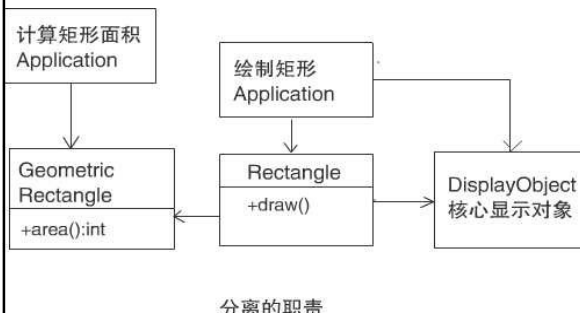


面向对象设计基本原则：

- 1、单一职责原则--SRP
- 2、开放封闭原则--OCP
- 3、接口隔离原则--ISP
- 4、依赖倒置原则--DIP
- 5、Liskov替换原则—LSP
- 6、合成/聚合复用原则
- 7、迪米特法则

示例

- Rectangle类具有两方法，一个方法把矩形绘制在屏幕上，另一个方法计算矩形的面积。



示例

- Rectangle类具有两方法, 一个方法把矩形绘制在屏幕上, 另一个方法计算矩形的面积。

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}

public void drawShaps() {
    ....
}

public void AreaCalculator() {
    double area = 0;
    area = Width* Height;
}

}

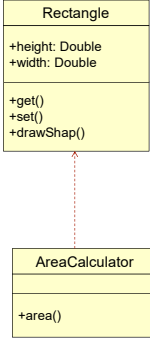
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}

public class AreaCalculator
{
    public double Area(Rectangle[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Width*shape.Height;
        }
        return area;
    }
}
```

示例

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}

public class AreaCalculator
{
    public double Area(Rectangle[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Width*shape.Height;
        }
        return area;
    }
}
```



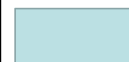
示例

- Rectangle类具有两方法, 一个方法把矩形绘制在屏幕上, 另一个方法计算矩形的面积。

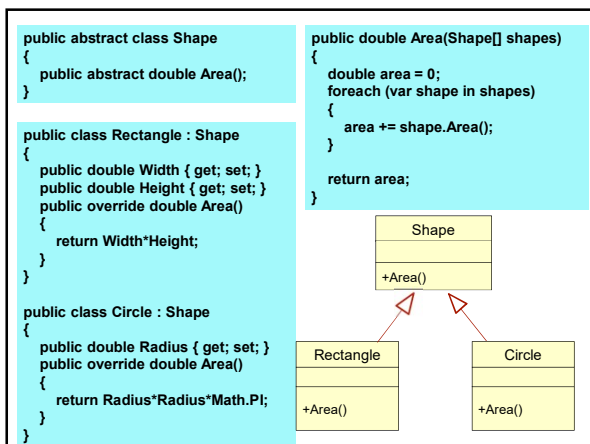
```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }

    public void drawShaps() {
        ....
    }
    public void AreaCalculator() {
        double area = 0;
        area = Width* Height;
    }
}
```

| Rectangle |
|--|
| +height: Double +width: Double |
| +get() +set() +drawShap() +areaCalculator() |



```
public double AreaCalculator(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Math.PI;
        }
    }
    return area;
}
```



小练习

```

interface IPHONE{
    public dial(pno:String):void;
    public hangup():void;
    public send(c:Char):void;
    public recv():void;
}

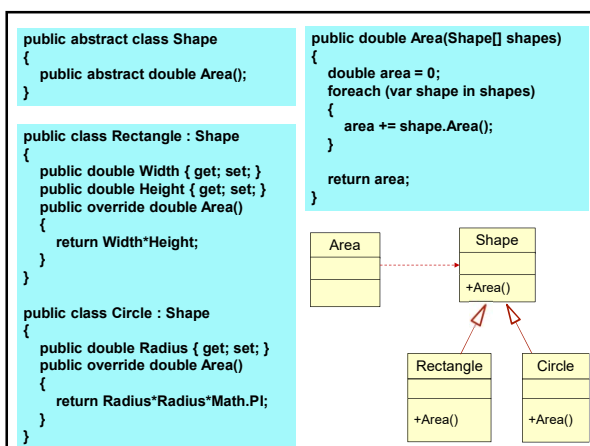
```

该接口却显示出了两个职责:

- 一个是连接管理 (dial+hangup)
- 第二个是数据通信 (send+recv) chat

单一职责原则有什么好处:

- 1、类的复杂度降低，实现什么职责都有清晰明确的定义；
- 2、可读性提高；复杂度降低，那当然可读性提高了；
- 3、可维护性提高，可读性提高，那就更容易维护了；
- 4、变更引起的风险降低；也就是说提高了扩展性和可维护性。



2、开-闭原则 (OCP)

- 一个软件实体，应当对扩展开放，对修改关闭。

设计一个模块时，应当使该模块在未被修改前提下被扩展；即不修改源代码，而改变该模块的行为。

- 满足开-闭原则的设计的优越性:

- 1、具备适应性和灵活性；
- 2、稳定性和延续性；

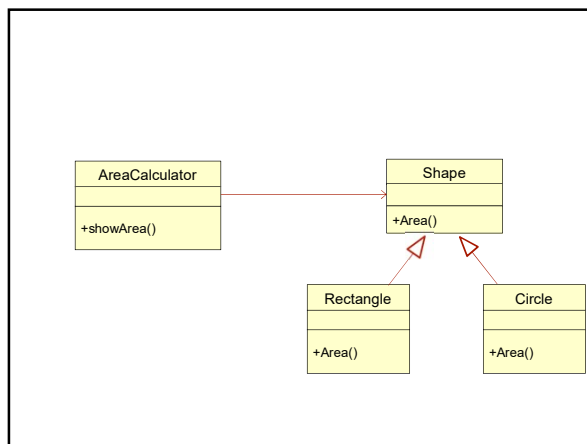
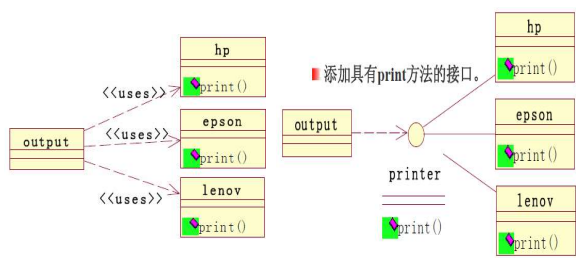
实现该原则，是在更高层次上，实现了复用的、易于维护的系统。

关键点：抽象
对可变性的封装

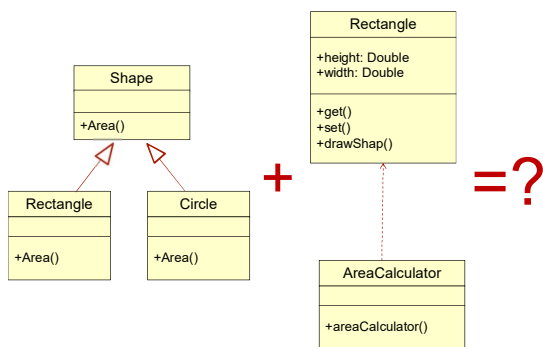
抽象设计：给系统定义出一个一劳永逸、不再修改的抽象设计。允许此设计有“无穷无尽”的行为在实现层被实现。

开闭原则示例

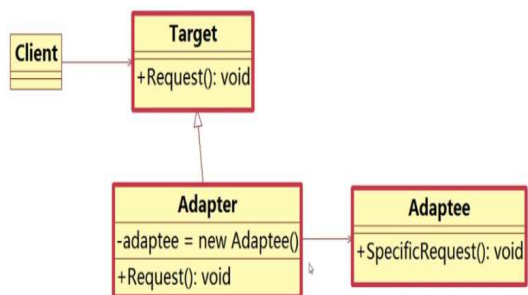
- 考虑设计中什么可能会发生变化，将其封装起来，考虑允许什么发生而不让这一变化导致重新设计。
- 声明的变量的类型、函数的参数类型、函数的返回类型等要尽量使用抽象类和接口。



思考问题??

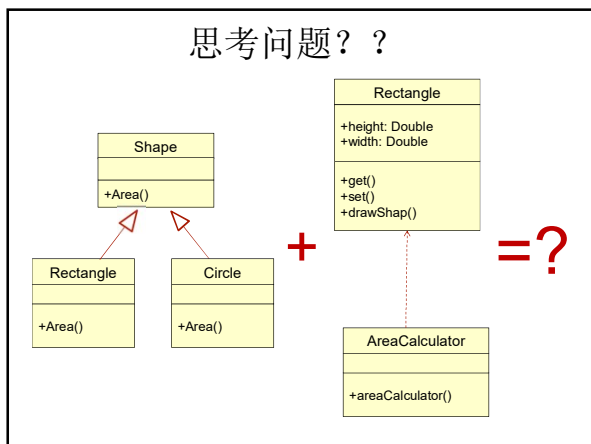


示例：适配器模式



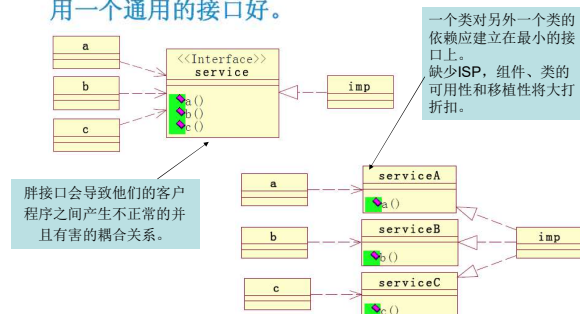
把“被适配器”作为一个对象组合到适配器类中，以修改目标接口包装被适配器。

思考问题??

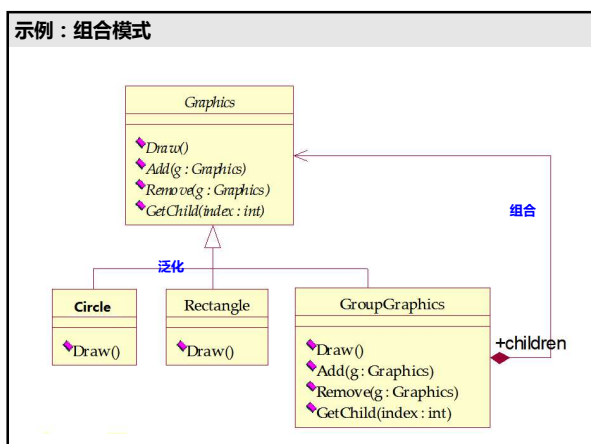


3、接口分离的原则 (ISP)

◆ 设计时采用多个与特定客户类有关的接口比采用一个通用的接口好。

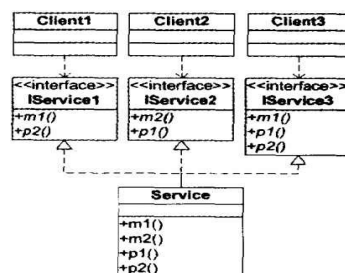


示例：组合模式



接口隔离原则

- 使用多个专门的接口，比使用单一的总的接口要好。即：一个类对另一个类的依赖性，应当建立在最小的接口上。



- 为同一个角色提供宽窄不同的接口，以对付不同的客户端，即定制服务。
- 有一个角色service以及3个不同的客户端；每一个JAVA接口都仅仅将客户端需要的行为暴露给客户端，而没有将客户端不需要的行为放到接口中。

接口隔离原则

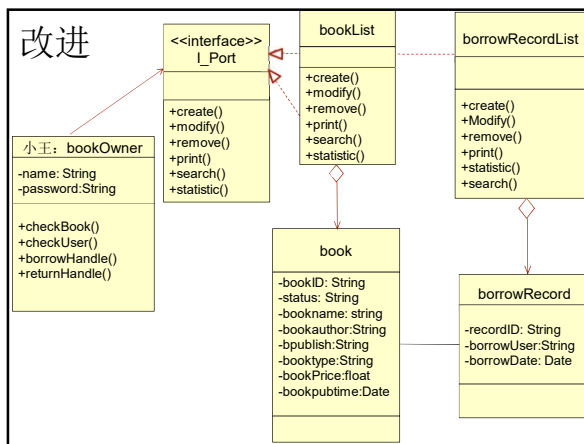
接口污染：过于臃肿的接口就是对接口的污染。

- 每一个接口都代表一个角色，实现一个接口的对象，在整个生命周期中都扮演该角色。
- 一个符合逻辑的推断，不应当将几个不同的角色交给同一个接口，而应当交给不同的接口。
- 将没有关系的接口合并在一起，形成一个大的臃肿的接口，是对角色和接口的污染。

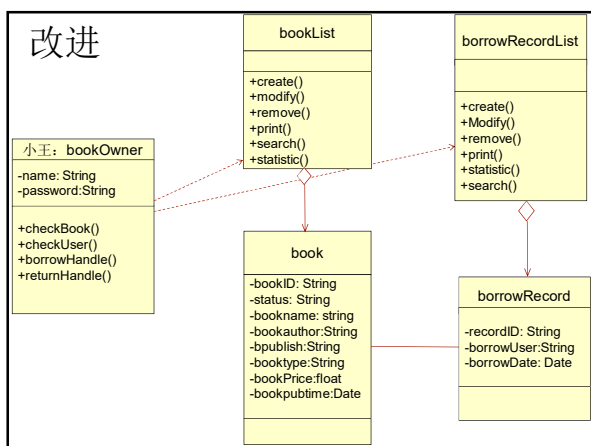
与迪米特法则的关系

- 迪米特法则要求任何一个软件实体，除非绝对必要，不然不要与外界通信。即使必须进行通信，也应当尽量限制通信的广度和深度。
- 定制服务原则拒绝向客户提供不需要的行为，符合迪米特法则。

改进

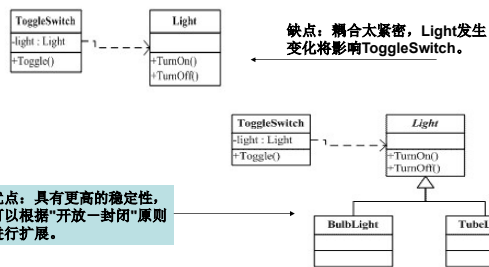


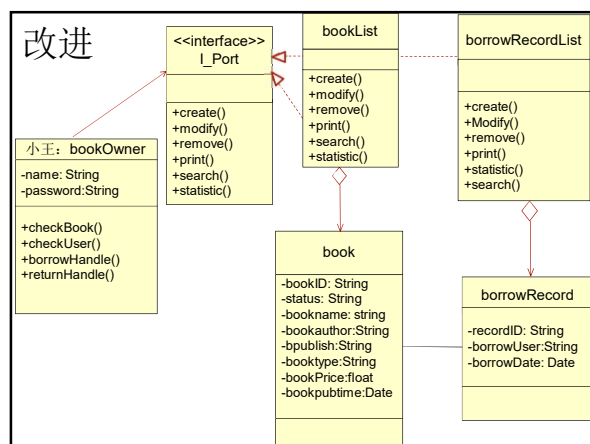
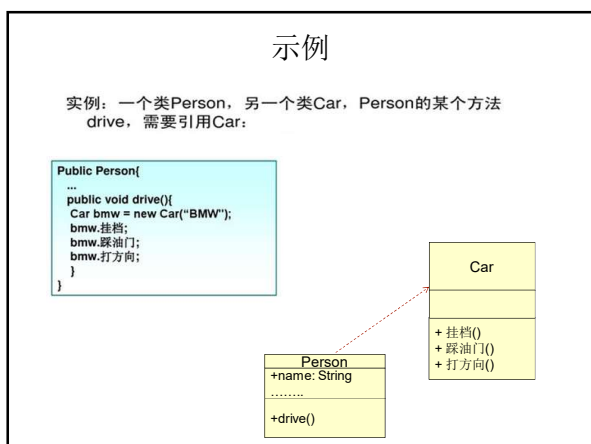
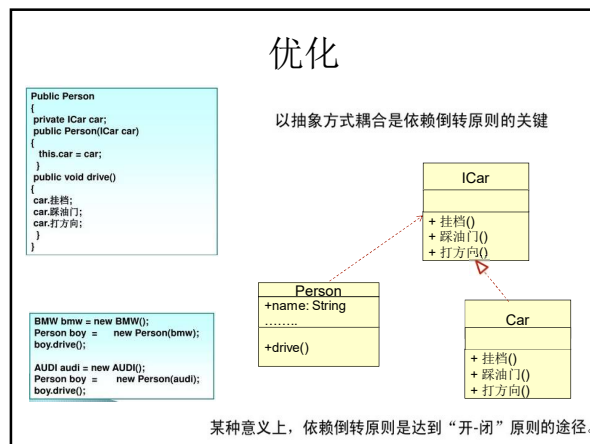
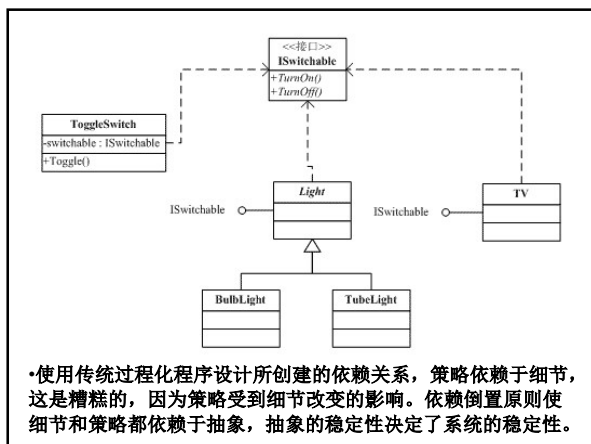
改进



4、依赖倒置原则（DIP）

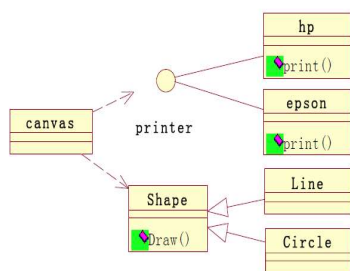
- 1、高层模块不应该依赖于低层模块，二者都应该依赖于抽象。
- 2、抽象不应该依赖于细节，细节应该依赖于抽象。





依赖性说明:

- ◆ 依赖关系尽量依赖接口（或抽象类），而不是依赖具体类。

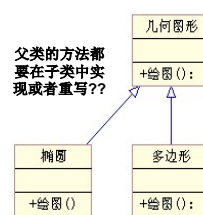


- 抽象反映高层策略
- 仔细分析需求，先找出那些业务规则，然后把它们抽象出来形成你的接口。
- 层次化你的设计，常见的方式就是划分出显示层，业务层，持久层，再在每层做抽象。
- 在实现的时候始终遵循前面提到的原则：只依赖于接口。
- 谁也无法在开始就做到最好，因此要不断迭代，精化设计。

◆ 子类可以替代父类出现在任何父类出现的地方

- 1、保证系统或子系统有良好的扩展性。
- 2、实现运行期内绑定，即保证了面向对象多态性的顺利进行。
- 3、有利于实现契约式编程。

父类的方法都要在子类中实现或者重写??



```
Var g: Graph; //图形
var p: Polygon := Polygon.New
//多边形
var o: Oval := Oval.New //椭圆形
...
if user says OK then g:= p
else g:= o
End if
...
g.getArea; //g可能表示Polygon或Oval对象
...
```

<http://hi.baidu.com/feipeng/blog/item/078f21a4560ce5f79052ee91.html>

5、Liskov 替换原则（LSP）

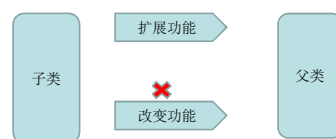
- 名字的由来（Liskov Substitution Principle, LSP）
- 里氏替换原则以Barbara Liskov（芭芭拉·利斯科夫）教授的姓氏命名。最早是在1988年，由麻省理工学院的女教授（芭芭拉·利斯科夫）提出来的。

- 芭芭拉·利斯科夫：美国计算机科学家，2008年图灵奖得主，2004年约翰·冯诺依曼奖得主，美国工程院院士，美国艺术与科学院院士，美国计算机协会会士，麻省理工学院电子电气与计算机科学系教授，美国第一位计算机科学女博士。



里氏替换原则的定义

- 1、里氏替换原则通俗的来讲就是：子类可以扩展父类的功能，但不能改变父类原有的功能。



- 2、里氏替换原则告诉我们，在软件中将一个基类对象替换成它的子类对象，程序将不会产生任何错误和异常，反过来则不成立，如果一个软件实体使用的是一个子类对象的话，那么它不一定能够使用基类对象。

里氏替换原则是继承复用的基石，**只有当衍生类可以替换掉基类，软件单元的功能不受影响时，基类才能真正被复用。**

JAVA对里氏替换原则的支持：

基类Base，对象b；子类Sub，对象d；
一般有method（Base b），则可以method（d）

问题：

基类Base，实现函数public run（）；
子类Sub，是否可以实现函数private run（）；

分析：

里氏替换原则要求，凡是基类适用的地方，子类一定适用。
因此，子类必须具备基类的所有接口，可以更宽，但不能少。

如果method调用了b的public run（），则同样应该可以调用d的run（）。
但如果Sub中的run定义成了private，不能被调用。此时，就会出错。

所以，从里氏替换原则的角度看，Sub中的run不可能被定义为private类型。

问题

```
public class C {
    public int func(int a, int b){
        return a+b;
    }
}

public class C1 extends C{
    @Override
    public int func(int a, int b) {
        return a-b;
    }
}

public class Client{
    public static void main(String[] args) {
        C c = new C1();
        System.out.println("2+1=" + c.func(2, 1));
    }
}
```

```
public class C {
    public int func(int a, int b){
        return a+b;
    }
}

public class C1 extends C{
    @Override
    public int func2(int a, int b) {
        return a-b;
    }
}

public class Client{
    public static void main(String[] args) {
        C1 c = new C1();
        System.out.println("2+1=" + c.func2(2, 1));
    }
}
```

子类C1继承父类C时，可以添加新方法完成新增功能，尽量不要重写父类C的方法。否则可能带来难以预料的风险。

在继承父类属性和方法的同时，每个子类也都可以有自己的个性，在父类的基础上扩展自己的功能。当功能扩展时，子类尽量不要重写父类的方法，而是另写一个方法，所以对上面的代码加以更改，使其符合里氏替换原则，

3、里氏替换原则是实现开闭原则的重要方式之一，由于**使用基类对象的地方都可以使用子类对象**，因此在程序中**尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象。**

说明：

1、子类可以实现父类的抽象方法，但是不能覆盖父类的非抽象方法

在我们做系统设计时，经常会设计接口或抽象类，然后由子类来实现抽象方法，这里使用的其实就是里氏替换原则。子类可以实现父类的抽象方法很好理解，事实上，子类也必须完全实现父类的抽象方法，哪怕写一个空方法，否则会编译报错。

父类中凡是已经实现好的方法，实际上是在设定一系列的规范和契约，虽然它不强制要求所有的子类必须遵从这些规范，但是如果子类对这些非抽象方法任意修改，就会对整个继承体系造成破坏。而里氏替换原则就是表达了这一层含义。

四、在使用里氏替换原则时需要注意如下几个问题

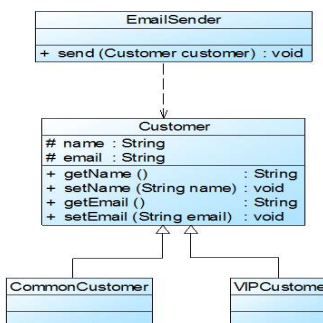
1、**子类的所有方法必须在父类中声明，或子类必须实现父类中声明的所有方法。**根据里氏替换原则，为了保证系统的扩展性，在程序中通常使用父类来进行定义，如果一个方法只存在于子类中，在父类中不提供相应的声明，则无法在以父类定义的对象中使用该方法。

四、在使用里氏替换原则时需要注意如下几个问题

2、我们在运用里氏替换原则时，**尽量把父类设计为抽象类或者接口**，让子类继承父类或实现父接口，并实现在父类中**声明的方法**，运行时，子类实例替换父类实例，我们可以很方便地扩展系统的功能，同时无须修改原有子类的代码，增加新的功能可以通过增加一个新的子类来实现。里氏替换原则是开闭原则的具体实现手段之一。

3、在系统设计时，遵循里氏替换原则，**尽量避免子类重写父类的方法**，可以有效降低代码出错的可能性。

可以增加一个新的抽象客户类Customer，而将CommonCustomer和VIPCustomer类作为其子类，邮件发送类EmailSender类针对抽象客户类Customer编程，根据里氏替换原则，**优化**后的结构如下图：



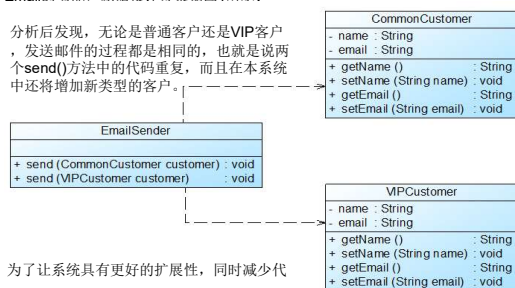
知识点：
1、尽量把父类设计为抽象类或者接口

2、能够接受基类对象的地方必然能够接受子类对象

示例：

在Sunny软件公司开发的CRM系统中，客户(Customer)可以分为VIP客户(VIPCustomer)和普通客户(CommonCustomer)两类，系统需要提供一个发送Email的功能，原始设计方案如图1所示：

分析后发现，无论是普通客户还是VIP客户，发送邮件的过程都是相同的，也就是说两个send()方法中的代码重复，而且在本系统中还将增加新类型的客户。



为了让系统具有更好的扩展性，同时减少代码重复，使用里氏替换原则对其进行重构。

6、迪米特与合成/聚合复用原则

迪米特法则 (Law of Demeter, LoD) 也称为最少知识原则 (Least Knowledge Principle, LKP)，一个对象应该对其他对象有最少的了解。

```

public class Teacher {
    public void command(GroupLeader groupLeader) {
        List<Student> listStudent = new ArrayList<Student>();

        for (int i = 0; i < 20; i++) {
            listStudent.add(new Student());
        }

        groupLeader.countStudent(listStudent);
    }
}
  
```

6、迪米特与合成/聚合复用原则

迪米特法则 (Law of Demeter, LoD) 也称为最少知识原则 (Least Knowledge Principle, LKP), 一个对象应该对其他对象有最少的了解。

```
public class Teacher {
    public void commond(GroupLeader groupLeader) {
        groupLeader.countStudent();
    }
}

public class GroupLeader {
    private List<Student> listStudent;

    public GroupLeader(List<Student> _listStudent) {
        this.listStudent = _listStudent;
    }

    public void countStudent() {
        System.out.println("学生数量是: " + listStudent.size());
    }
}
```

合成/聚合复用原则

- **继承复用**: 子类继承基类, 从而继承基类的方法。
- 优点:
 - 新的实现较为容易, 基类的大部分功能可以通过继承关系自动进入子类;
 - 修改或扩展继承而来的实现较为容易。
- 缺点
 - 继承复用破坏包装, 基类的内部细节对子类常常是透明的, 继承会将超类的实现细节暴露给子类, 白箱复用。
 - 如果基类发生变化, 子类的实现也不得不发生改变。当基类发生变化时, 会像石子引起的水波一样, 将变化一圈一圈的传导到一级又一级更大范围的子类中, 使得程序员不得不相应的修改这些子类, 工作量大。
 - 从基类继承而来的实现是静态的, 不可能在运行期间发生改变, 缺乏灵活性。

| | |
|--|------------------------|
| <pre>public class Teacher { public void commond(GroupLeader groupLeader) { List<Student> listStudent = new ArrayList<Student>(); for (int i = 0; i < 20; i++) { listStudent.add(new Student()); } groupLeader.countStudent(listStudent); } }</pre> | <p>面向对象的设计强调“黑箱”调用</p> |
| <p>信息隐藏的越深越好;</p> <p>信息只需要暴露需要提供的信息就可以了;</p> | |
| <pre>public class Teacher { public void commond(GroupLeader groupLeader) { groupLeader.countStudent(); } } public class GroupLeader { private List<Student> listStudent; public GroupLeader(List<Student> _listStudent) { this.listStudent = _listStudent; } public void countStudent() { System.out.println("学生数量是: " + listStudent.size()); } }</pre> | |

7、合成/聚合复用原则

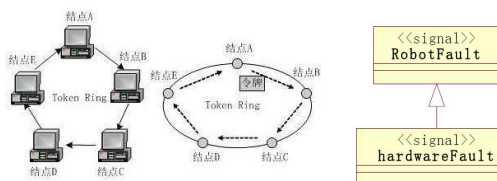
- 该原则的简述: **要尽量使用合成/聚合, 尽量不要使用继承。**
- **合成/聚合复用原则, 也称合成复用原则**: 就是在一个新的对象里面使用一些已有的老对象, 使之成为新对象的一部分; 新的对象通过向这些老对象的委派, 达到复用已有功能的目的。
- **合成**:
 - 合成关系中, 新对象在整体角度, 对其组成部分拥有完全的支配权, 包括他们的创建和湮灭。即组合而成的新对象对组成部分的内存分配、内存释放有绝对的责任。
 - 一个合成的多重性不能超过1, 即, 一个合成关系中的成分对象不能与另外一个合成关系共享。一个合成关系湮灭了, 则所有的成分对象在同一时间内都会被湮灭。

主要内容

- 面向对象设计基础
- 面向对象的建模思想与方法
- 面向对象的设计目标与原则
- **基于模式的事件处理机制**
- 软件重构与设计优化

◆ 信号 (signer) 事件

- 所谓信号,是指由一个对象异步地发送、并由另外一个对象接收的一个已命名的对象。
- 信号事件表示对象接收到某个信号。
- 信号可以作为状态机中一个状态转换的动作而被发送,也可作为交互中一个的消息发送



事件

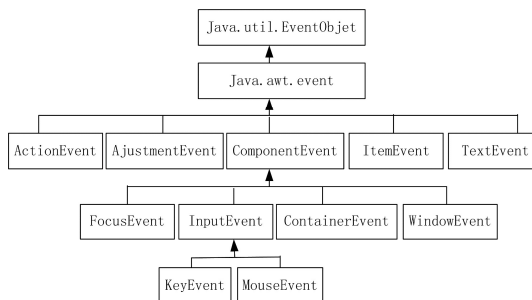
◆ Event

- 是对一个时间和空间上占有一定位置的有意义的事情的规格说明。
- 事件触发状态的转移

◆ 四类主要事件

- 信号事件
- 调用事件
- 变化事件
- 时间事件

事件类型

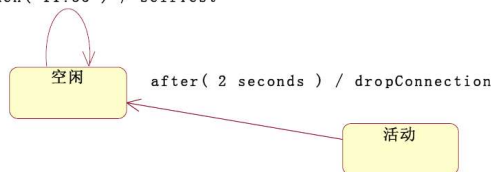


2、时间事件

◆ 时间 (time) 事件

- 满足某一时间表达式的情况的出现，例如到达某一时间或经过了某一时间段。用关键字After或When表示。

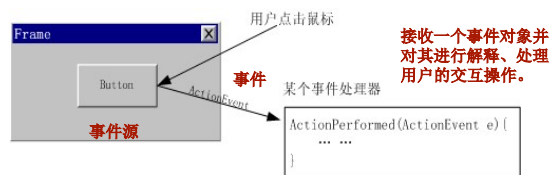
when(11:35) / selfTest



4、调用事件

◆ 调用 (call) 事件

- 表示一个操作的调度。请求调用另一个对象的操作。
- 一般是同步调用。



问题：不同事件源上发生的事件种类是不同的，若希望事件源上引发的事件被程序处理，那么怎么办呢？

3、变化事件

◆ 变化 (change) 事件

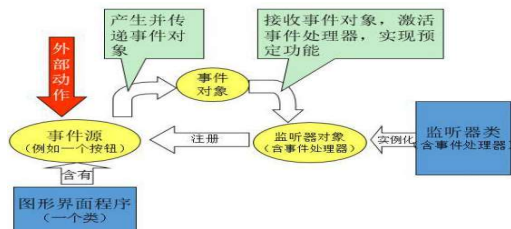
- 用关键字When，后面跟布尔表达式。

When(temperature > 120) / alarm()



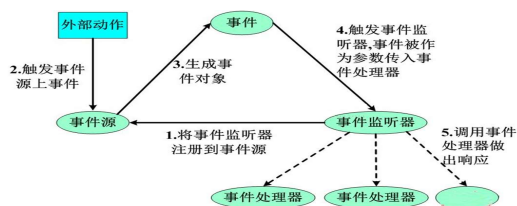
监听器

- JDK使用委托模型，事件被发送给发起该事件的组件，但它依赖于每个组件来将该事件传播到一个或者更多的已经注册的监听器。



监听器

- 监听器包含接收和处理该事件的事件处理器，它具有监听和处理某类事件的功能。
- 事件对象只向已注册的监听器报告。对于没有注册监听器的组件，来自这些组件的事件不被传播。



监听器分类

| 事件 | 监听器接口 | 监听器适配器类 | 事件处理方法 |
|------------------|---------------------|--------------------|---|
| KeyEvent | KeyListener | KeyAdapter | keyPressed keyReleased keyTyped |
| MouseEvent | MouseListener | MouseAdapter | mouseClicked mouseEntered mouseExited mousePressed mouseReleased |
| MouseMotionEvent | MouseMotionListener | MouseMotionAdapter | mouseDragged mouseMoved |
| TextEvent | TextListener | 无 | textValueChanged |
| WindowEvent | WindowListener | WindowAdapter | windowActivated windowClosed windowClosing windowDeactivated windowDeiconified windowIconified windowOpened |

监听器分类

- 不同事件源上发生的事件种类是不同的，若希望事件源上引发的事件被程序处理，则需要将事件源注册给能够处理该事件类型的监听器。

| 事件 | 监听器接口 | 监听器适配器类 | 事件处理方法 |
|-----------------|--------------------|------------------|---|
| ActionEvent | ActionListener | 无 | actionPerformed |
| AdjustmentEvent | AdjustmentListener | 无 | adjustmentValueChanged |
| ComponentEvent | ComponentListener | ComponentAdapter | componentHidden componentMoved componentResized componentShown |
| ContainerEvent | ContainerListener | ContainerAdapter | componentAdded componentRemoved |
| FocusEvent | FocusListener | FocusAdapter | focusGained focusLost |
| ItemEvent | ItemListener | 无 | itemStateChanged |

实例：为“Button”按钮添加事件的处理

1. 实现监听器接口

实现ActionListener接口

```
class FirstFrame extends JFrame implements ActionListener{
    .....
}
```

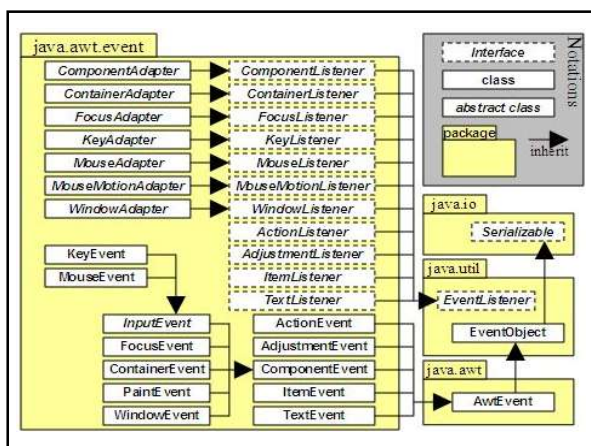
2. 为按钮注册监听器

```
btn1.addActionListener(this);
```

ActionEvent事件，这里指鼠标点击按钮这个事件

3. 实现监听器接口的所有方法

```
public void actionPerformed(ActionEvent e){
    JOptionPane.showMessageDialog(this,"登陆成功");
}
```



示例1

```

class Program
{
    static void Main(string[] args)
    {
        // ...
    }
}

class Car
{
    public delegate void Notify(int value);
    public event Notify notifier;

    petrol = value; // (petrol < 10) // (petrol的值小于10时，由实例报
    if (petrol < 10)
    {
        if (notifier != null)
        {
            notifier.Invoke(Petrol);
        }
    }
}

class Alerter
{
    public void Notify()
    {
        // ...
    }
}

Car.Notify()
{
    // ...
}

public void NotEnoughPetrol(int value)
{
    Console.ForegroundColor =
    value.ToString() + " gallon petrol left!");
    Console.ResetColor();
}

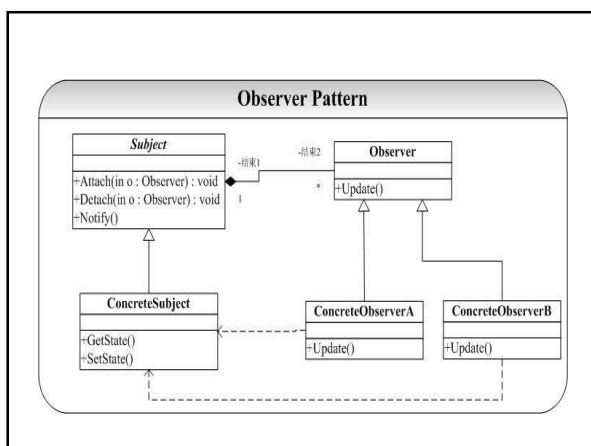
public void Run(int speed)
{
    int distance = 0;
    // ...
}

```

问题1: 为什么不在public int Petrol中直接调用Alerter.NotEnoughPetrol呢?

问题2: 关于事件?

问题3: 观察者模式 (观察者模式又称Source/Listener模式的实现)



```

public class Demo1 {
    public static void main(String[] args) {
        Frame f = new Frame(); // 代表windows窗口
        f.setSize(400, 400);
        f.setVisible(true);

        // 把监听器注册到事件源上
        f.addWindowListener(new MyListener());
    }

    // 编写一个监听器
    class MyListener implements WindowListener {
        @Override
        public void windowOpened(WindowEvent e) {
            // TODO Auto-generated method stub
        }

        /* 当window窗体关闭时，MyListener这个监听器
        就会监听到，
        * 监听器就会调用windowClosing方法处理
        window窗体关闭时的动作 */
        @Override
        public void windowClosing(WindowEvent e) {
            Frame f = (Frame) e.getSource(); // 拿到事件源
            f.dispose(); // 关闭窗口
        }

        @Override
        public void windowClosed(WindowEvent e) {
            // TODO Auto-generated method stub
        }

        // window窗口从正常状态变为最小化状态时调用
        @Override
        public void windowIconified(WindowEvent e) {
            System.out.println("hahahaha!!!");
        }

        @Override
        public void windowDeiconified(WindowEvent e) {
            // TODO Auto-generated method stub
        }

        @Override
        public void windowActivated(WindowEvent e) {
            // TODO Auto-generated method stub
        }

        @Override
        public void windowDeactivated(WindowEvent e) {
            // TODO Auto-generated method stub
        }
    }
}

```


示例2

```

<head>
<title>JSDemo</title>
</head>
<body>
<center>
<form name="form1" method="post" action="">
  请输入姓名:<input name="text1" type="text" value="" size="10">
  <input name="but1" type="button" value="检测" onclick="checkRealName()">
</form>
<script type="text/javascript">
  function checkRealName(){
    var str=form1.text1.value;
    if(str==""){
      alert("请输入姓名!");
      form1.text1.focus();
      return;
    }else{
      var objExp=/[u4E00-\u9FA5]{2,}/;
      if(objExp.test(str)==true){
        alert("您输入的姓名正确");return;
      }else{
        alert("您输入的姓名不合法");return;
      }
    }
  }
</script>
</center>
</body>

```

示例3

ServletContext, HttpSession和ServletRequest

```

1 package com.bw.listener;
2
3 import javax.servlet.http.HttpSessionEvent;
4 import javax.servlet.http.HttpSessionListener;
5
6 public class ListenerTest implements HttpSessionListener {
7
8     public void sessionCreated(HttpSessionEvent se) {
9         //在线人数+1
10        CountUtils.add();
11    }
12
13     public void sessionDestroyed(HttpSessionEvent se) {
14         //在线人数-1
15        CountUtils.subtract();
16    }
17
18 }

```

示例3

ServletContext, HttpSession和ServletRequest

```

1 public class MyHttpSessionListener implements HttpSessionListener {
2
3     @Override
4     public void sessionCreated(HttpSessionEvent se) {
5         System.out.println(se.getSession() + "被创建了!!!");
6         System.out.println("创建好的HttpSession的id是:" + se.getSession().getId());
7     }
8
9     @Override
10    public void sessionDestroyed(HttpSessionEvent se) {
11        System.out.println("session被销毁了!!!");
12    }
13
14 }

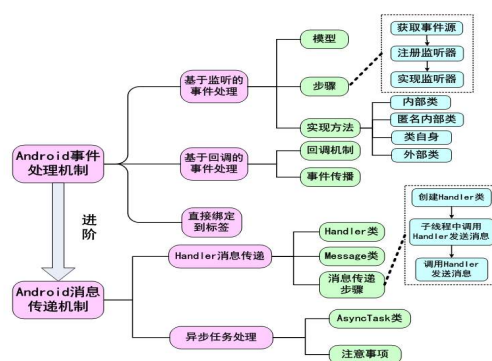
```

```

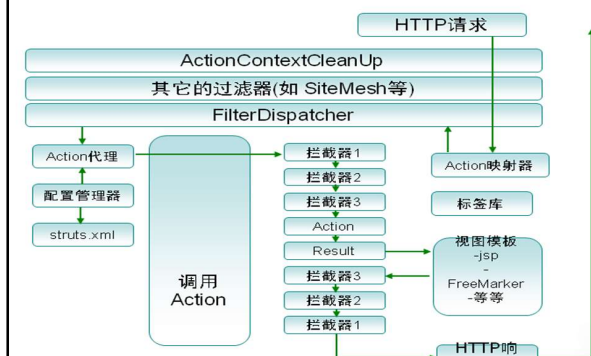
1 <listener>
2 <listener-class>cn.itcast.web.listener.MyHttpSessionListener</listener-class>
3 </listener>
4 <!-- 配置HttpSession对象的销毁时机 -->
5 <session-config>
6 <!-- 配置HttpSession对象1分钟之后销毁 -->
7 <session-timeout>1</session-timeout>
8 </session-config>

```

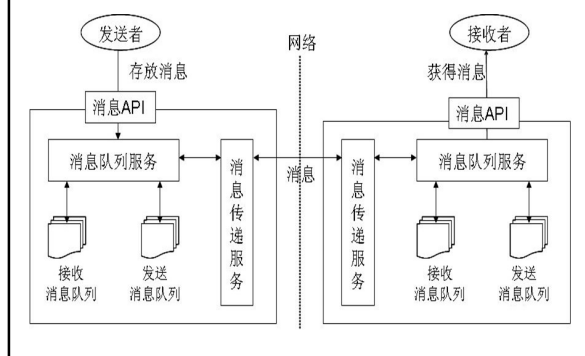
扩展1: Android事件处理机制



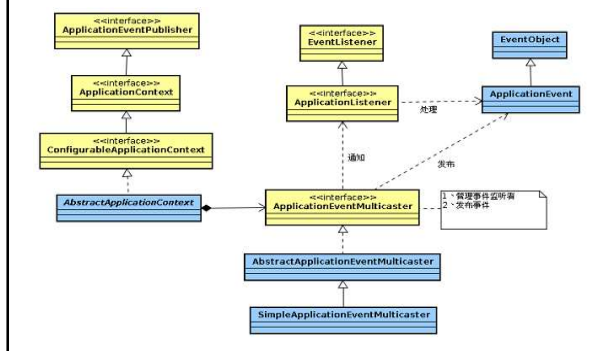
扩展2: Struts 请求处理机制



扩展4: 消息中间件



扩展3: Spring 事件处理机制



主要内容

- 面向对象设计基础
- 面向对象的建模思想与方法
- 面向对象的设计目标与原则
- 基于模式的事件处理机制
- **软件重构与设计优化**

软件重用

- 1968年，NATO软件工程会议上，McIlroy的论文“大量生产的软件构件”首次提出：

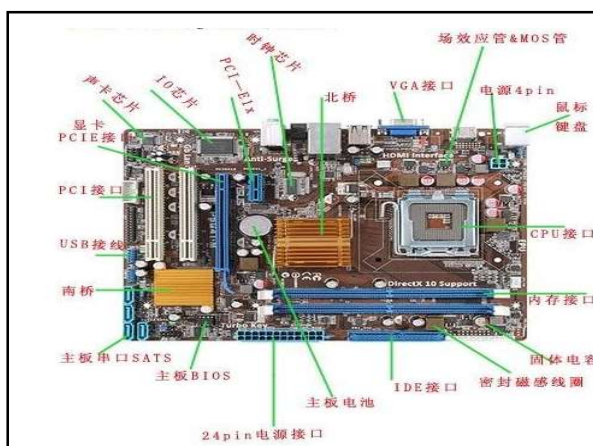
Doug McIlroy on Software Components, 1968



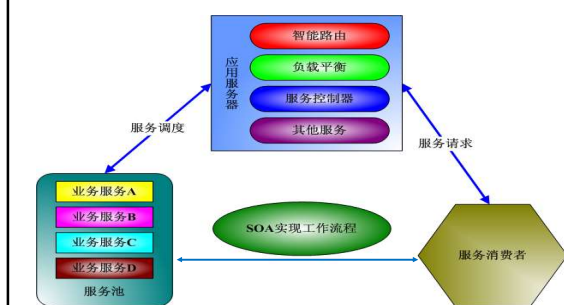
软件重用形式

软件重用（Software Reuse，又称软件复用或软件再用）就是将已有的软件成分用于构造新的软件系统。

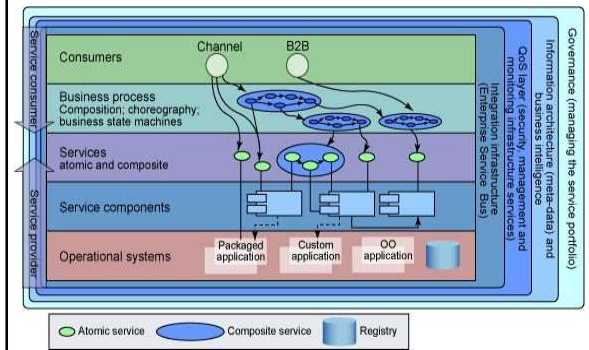
1. 源代码模块或者类一级的重用。
2. 二进制形式的重用。如组件重用。
3. 组装式重用。例如，要建立一个门户网站应用，登陆用户既可以查询天气情况，又可以查看股市行情，还可以在线购物。
4. 分析级别重用。
5. 设计级别重用。
6. 软件文档重用。



基于SOA的服务重用



基于SOA的服务重用



• 联系方式:

– Tel: 82663000转8002

13572238819

– Email: yuanrao@163.com

raoyuan@mail.xjtu.edu.cn

微信: iroyals



社会智能与复杂数据处理

谢谢大家

总结：对象设计基本概念

