



第二部分：如何运用UML建模

## 第六章 结构模型

# 提纲

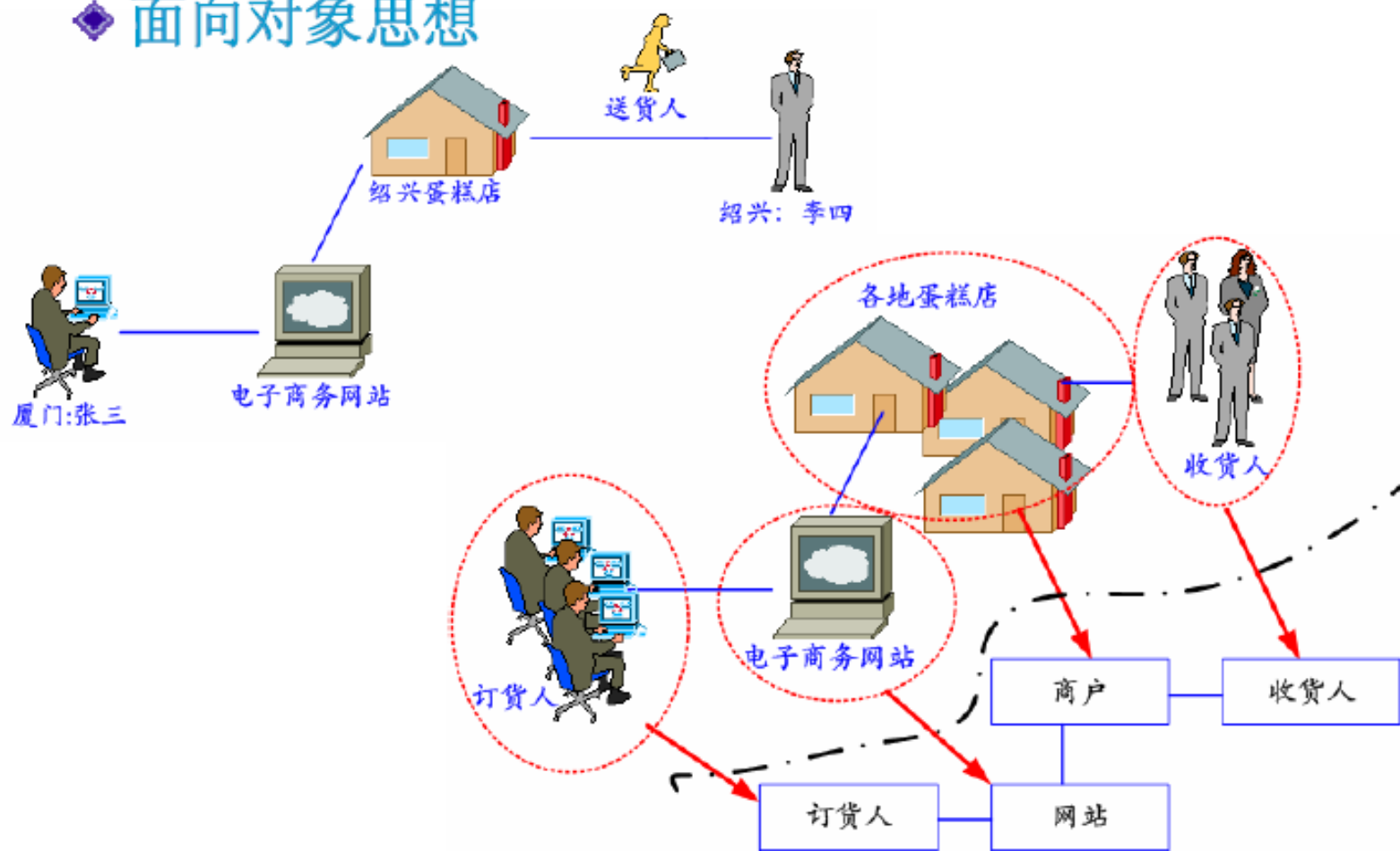


- 概述
- 类图
- 对象图
- 组件图
- 部署图
- 小结

- 结构模型
  - 对系统的静态方面进行可视化、详述、构造和文档化
  - 由类、接口、协作、组件和节点等结构事物的布局构成
  - 包含
    - 类图
    - 对象图
    - 组件图
    - 部署图

# 类图

## ◆ 面向对象思想



# 类

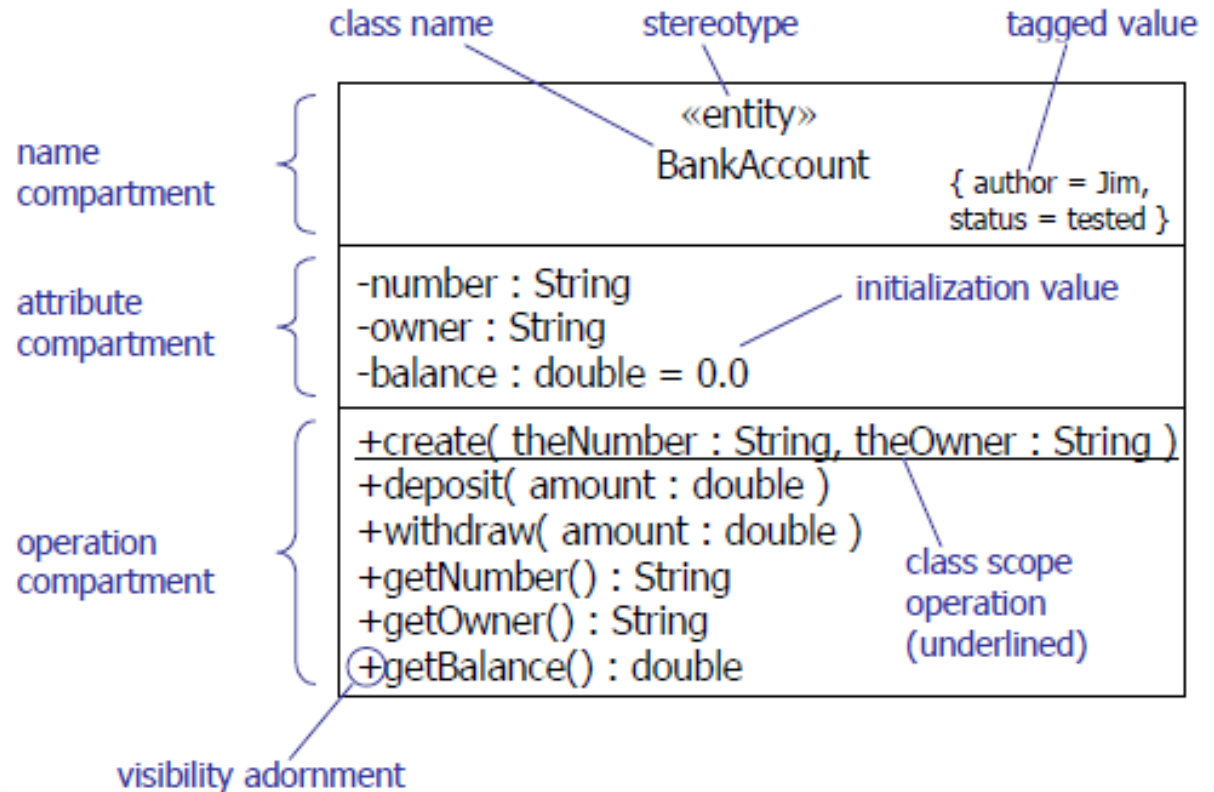
- 什么是类？

- 具有相似结构、行为和关系的一组对象的描述

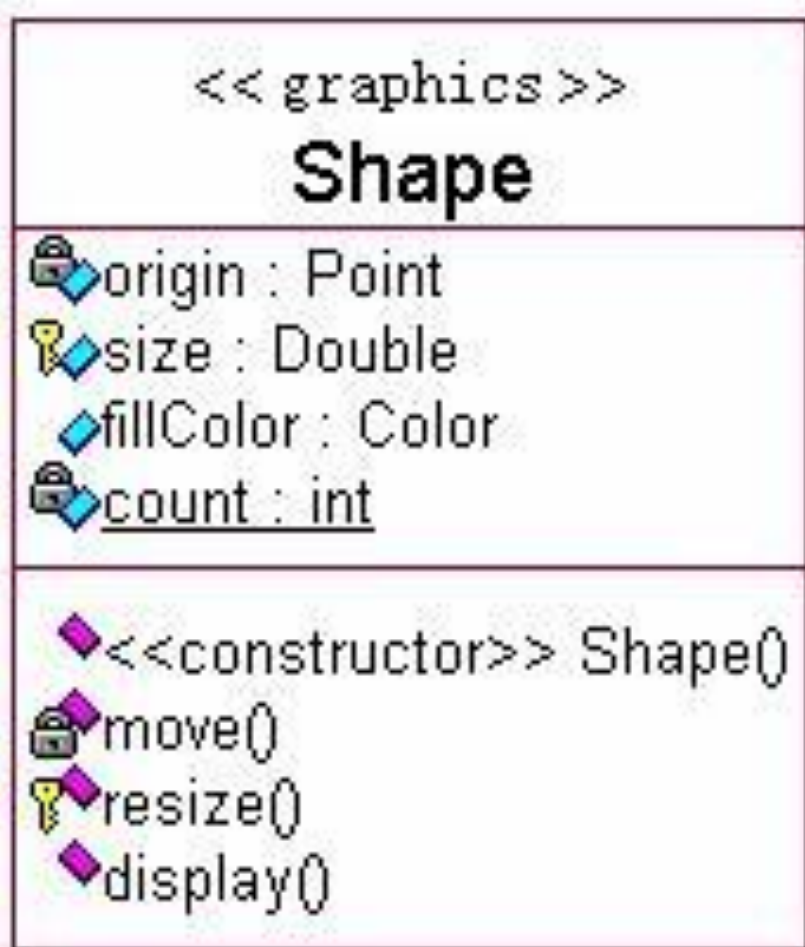
——Rumbaugh

- 组成

- 类名
    - 属性
    - 操作



# 类的定义



**类名** : Shape

四个**属性** : origin、size、fillColor、count（静态属性）。

四个**方法** : Shape()、move()、resize()、display()。

方法Shape()的**构造型**为<<constructor>>, 表示该方法是构造方法。

Shape类是一个**构造型**为<<graphics>>的类。

# 类的定义——建模的角度

- 类是什么：
- 可从4个角度来看类：建模、设计、实现、编译器。
- (1) 从建模的角度：
  - 1、类将特征（属性）定义为数据变量。
  - 2、类将服务（行为）定义为操作（包括接口与实现）。
  - 3、类将规则和策略定义为应用的约束。
  - 4、类定义了对象间的关系（对等或层次）；也可有泛化/特化关系。

# 类的定义——设计的角度

- 类是什么：
- 可从4个角度来看类：建模、设计、实现、编译器。
- (2) 从设计的角度：

- 1、类是一种特殊类型的对象。
- 2、类是它所创建（实例化）的所有对象的集合。
- 3、类用来创建并销毁属于它的集合的对象。
- 4、类可以为这个对象集合保存数据并提供服务。



# 类的定义——实现的角度

- 类是什么：
- 可从4个角度来看类：建模、设计、实现、编译器。
- (3) 从实现的角度：
  - 1、类是一个具有类数据成员和类服务的“全局”对象，应用可以使用类名访问类的服务。
  - 2、应用可以用类的构造函数“实例化”对象，以此来创建对象。

# 类的定义——编译器的角度

- 类是什么：
- 可从4个角度来看类：建模、设计、实现、编译器。
- (4) 从编译器的角度：
  - 1、类是程序员定义的数据型；
  - 2、类是定义并创建运行对象的一种机制。

# 类的组成



- 类的组成

- 属性

- [可见性] 属性名[: 类型] [ ‘[’ 多重性[次序] ‘]’ ]  
[=初始值][{特性}]
    - 其中，{特性}是用户对该属性性质的一个约束说明。
    - 例：– balance: double = 0.0

- 操作




- [可见性] 操作名[ (参数列表) ][:返回类型][{特性}]
    - 例：– getBalance ( ): double

# 类的组成



- 属性和操作

- 可见性：模型元素在所含空间外的可见度

- 公共 (Public)      +      
    - 私有 (Private)      -      
    - 保护 (Protected)      #      

- 范围：表示属性/操作的所有者范围

- 实例范围
    - 类范围

# 类的属性

- 属性格式:

[可见性] 属性名 [:类型] [ '[' 多重性 [次序] ']' ] [=初始值] [{特性}]

- 其中, {特性} 是用户对该属性性质的一个约束说明。
- 例如, {只读} 这样的特性, 说明该属性的值不能被修改。

# 类的属性

- 属性：
- 属性的选取应考虑以下因素：
  - 原则上来说，类的属性应能描述并区分每个特定的对象；
  - 只有系统感兴趣的特征才包含在类的属性中；
  - 系统建模的目的。根据类图的详细程度，每条属性可以包括属性的可见性、属性名称、类型、多重性、初始值和约束特性。

# 类的属性

- 例：一些属性声明的例子：
- `+size: Area = (100,100)`
- `#visibility: Boolean = false`
- `+default-size: Rectangle`
- `#maximum-size: Rectangle`
- `-xptr: XwindowPtr`
- `~colors : Color [3]`
- `points : Point [2..* ordered]`
- `name : String [0..1]`

# 类的属性

- 例：一些属性声明的例子：

```
public class B
{
    Public +size: Area = (100, 100) theArea;
    /**
     * @roseuid 3DAFBF0F01A2
     */
    public B()
    {

    }
}
```



# 类的操作

- 操作格式:

[可见性] 操作名 [ (参数列表) ] [ :返回类型 ] [ {特性} ]

- 其中, **{特性}** 是一个文字串, 说明该操作的一些有关信息。
- 例如, **{query}** 这样的特性, 说明该操作不会修改系统的状态。

# 类的操作

- 操作：
- 操作的选取应考虑以下因素：
  - 类的操作 (Operation)：操作用于修改、检索类的属性或执行某些动作，操作通常也被称为功能，但是它们被约束在类的内部，只能作用到该类的对象上；
  - 操作名、参数列表和返回类型组成操作接口；

# 类的操作

- 例：一些操作声明的例子：
- `+display () : Location`
- `~hide ()`
- `#create ()`
- `-attachXWindow(xwin: XwindowPtr)`

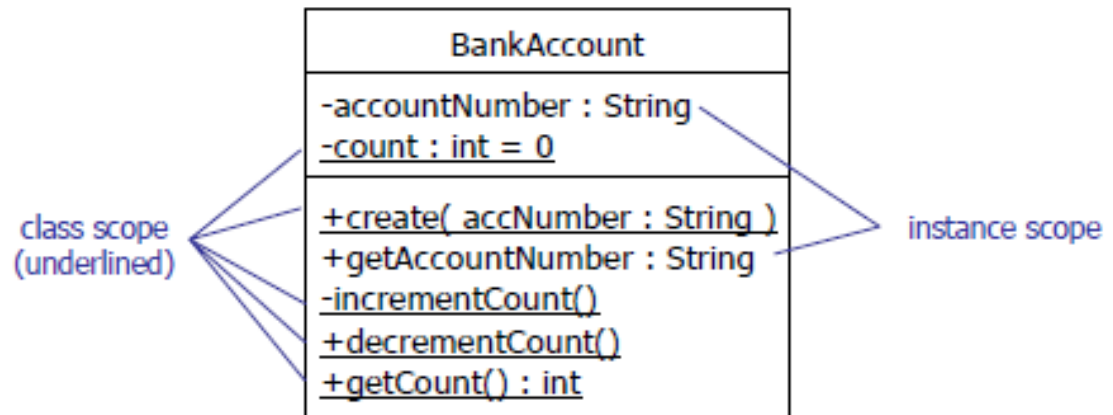
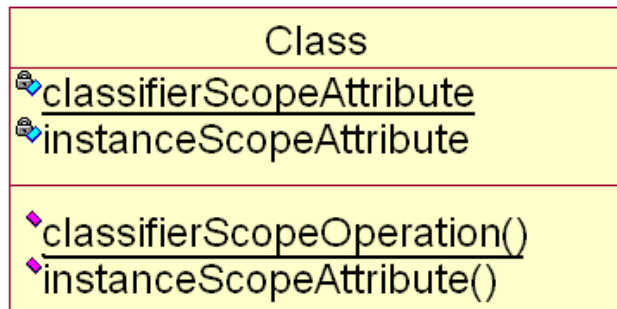
# 类的属性

- 例：一些操作声明的例子：

```
public class B
{
    Public +size: Area = (100,100) theArea;
    /**
     * @roseuid 3DAFBF0F01A2
     */
    public B( )
    {
    }
    public +display ( ): Location
    {
    }
}
```

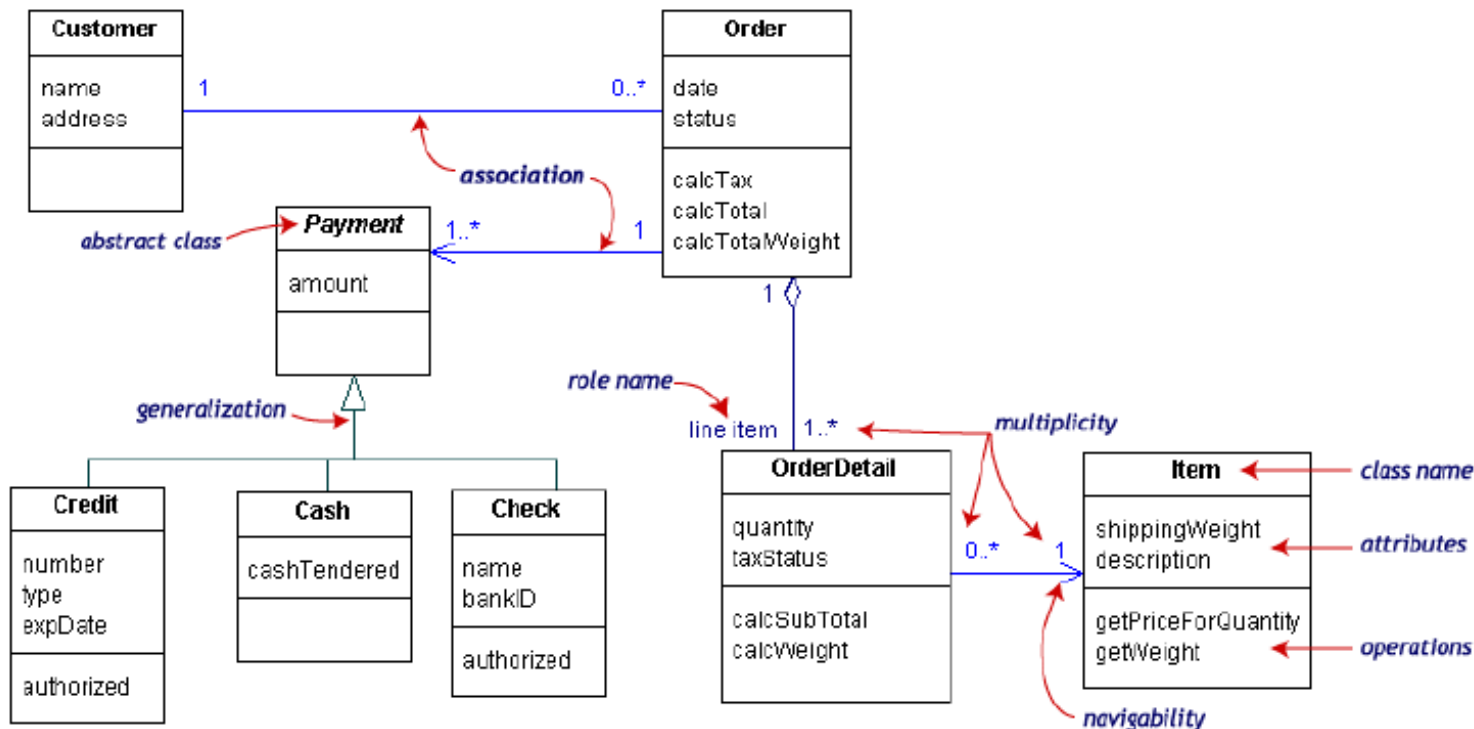
# 类的范围

- 范围
  - 实例范围
    - 不同对象具有不同属性值
  - 类范围
    - 为类的所有对象提供全局特征
    - 为每个对象定义具有单一、共享值的属性



- 类图

- 展示一组类、接口、协作及其之间的关系



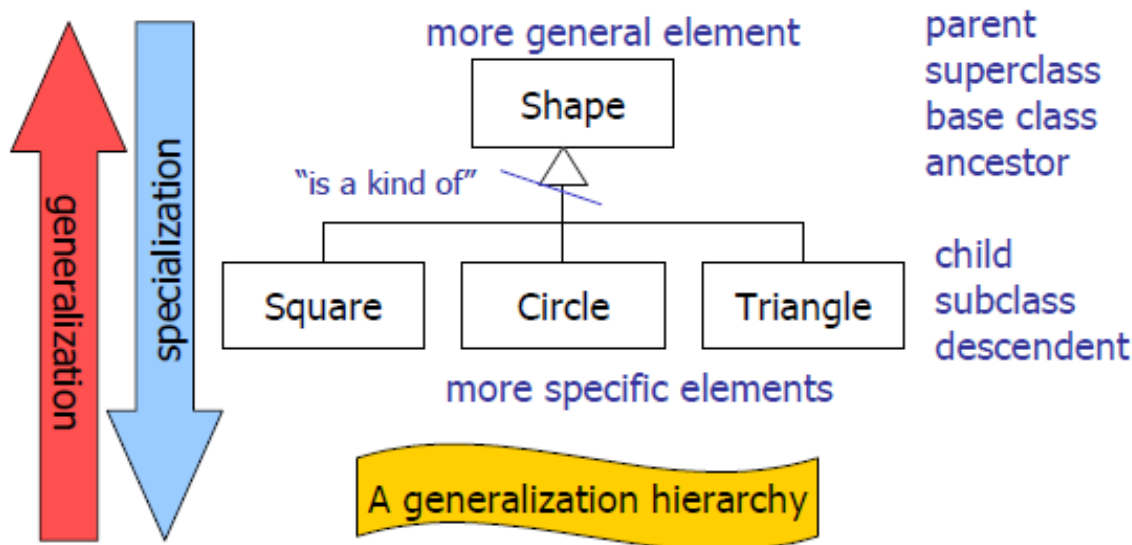
- 类图
  - 通常包含
    - 结构事物： 类、接口、协作
    - 关系： 依赖、关联、泛化、实现
  - 一般用法
    - 命名和建模系统中的概念
    - 对简单协作建模
    - 建模逻辑数据库模式

# 泛化——关系



- 泛化关系

- 一种特殊（子元素）/一般（父元素）关系
- 子元素共享父元素的结构和行为
- 子元素的对象可替代父元素的对象





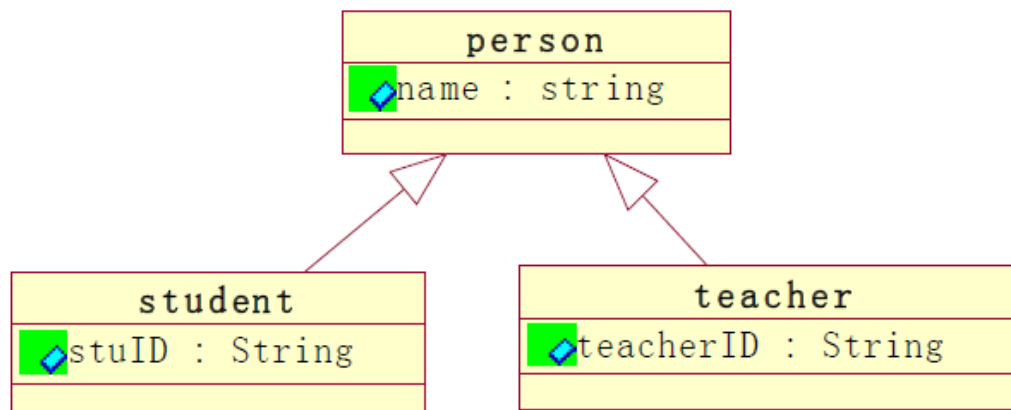
# 泛化——关系



- 泛化关系

- 用于建模面向对象中的继承

- 给定一组类，寻找两个及以上类之间公共的职责、属性和操作
    - 将公共的职责、属性与操作抽象为一个父类
    - 指明子类到父类间的继承关系



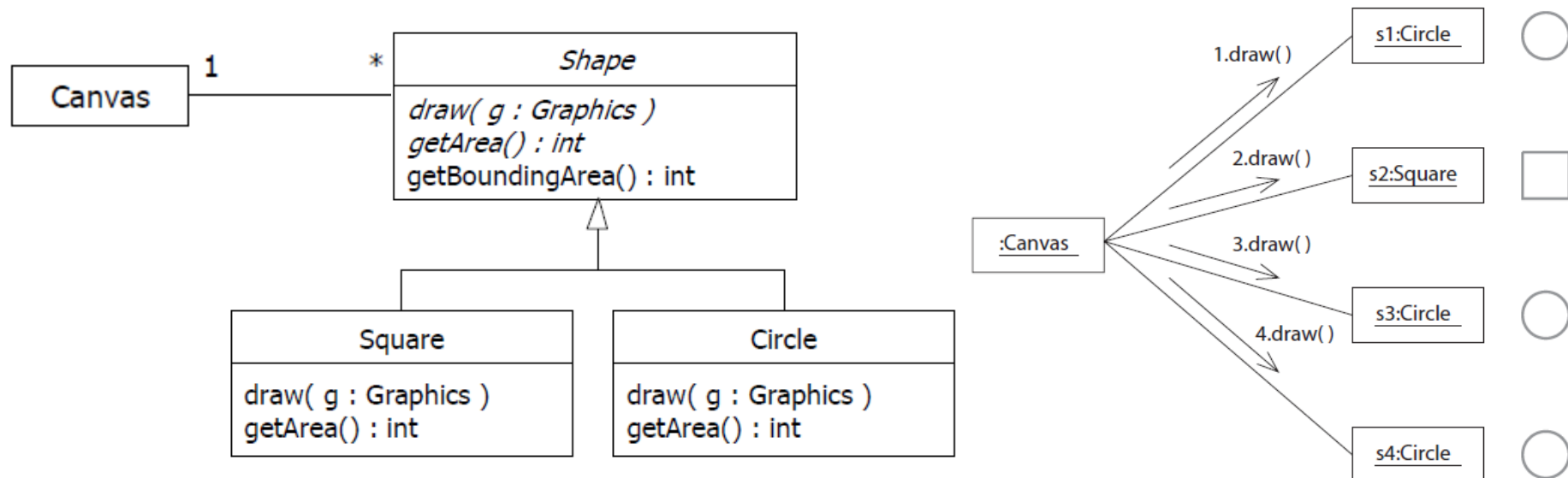
# 泛化——关系



- 泛化关系

- 用于实现面向对象中的多态

- 覆写：子类重新定义从父类继承来的操作
    - 多态：继承了同一父类的多个子类的同名操作具有各自不同的形态



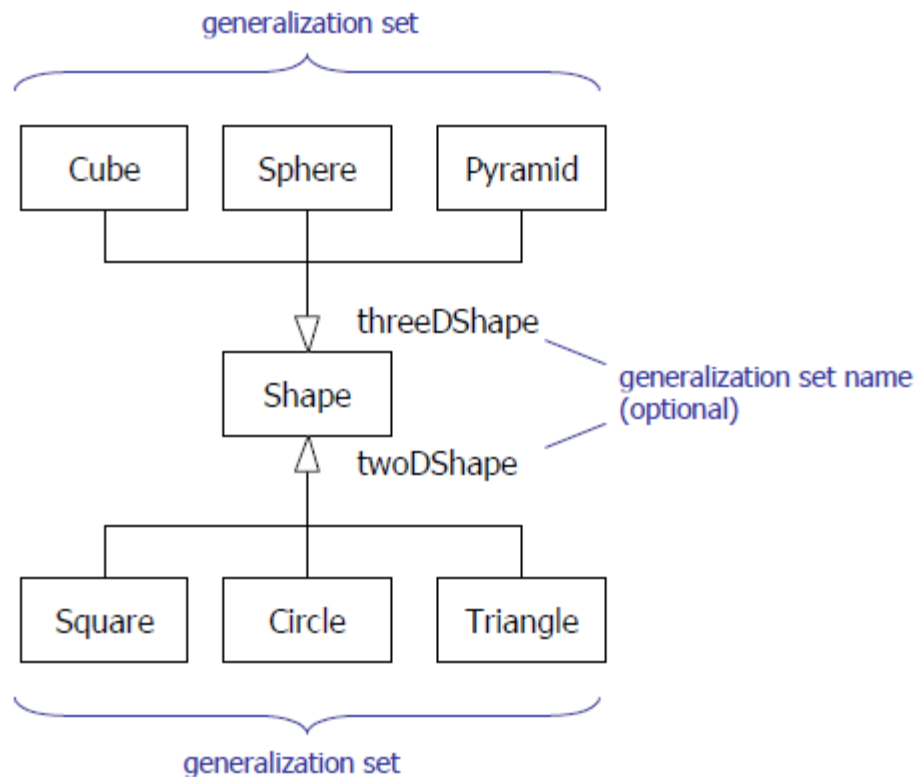
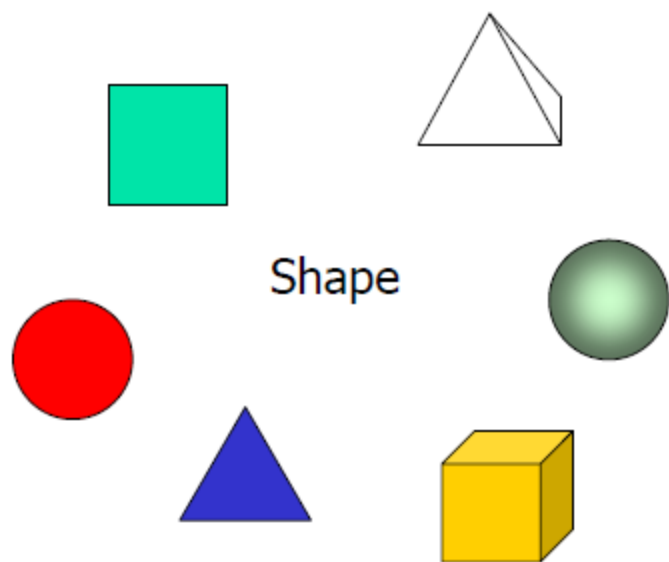
# 抽象类——泛化



- 抽象类 (Abstract class)
  - 不能产生实例的类
    - 在UML中用斜体书写表示
    - 类中的操作只有声明，没有实现
    - 泛化关系中所有的基类都应该是抽象类
  - 优点
    - 抽象操作定义了一组子类必须实现的“契约”
    - 可替换性原则：任意子类都可代替父类

# 泛化——关系

- 泛化的约束
  - 泛化集合：由一个超类的若干子类构成



# 泛化——关系



- 泛化的约束

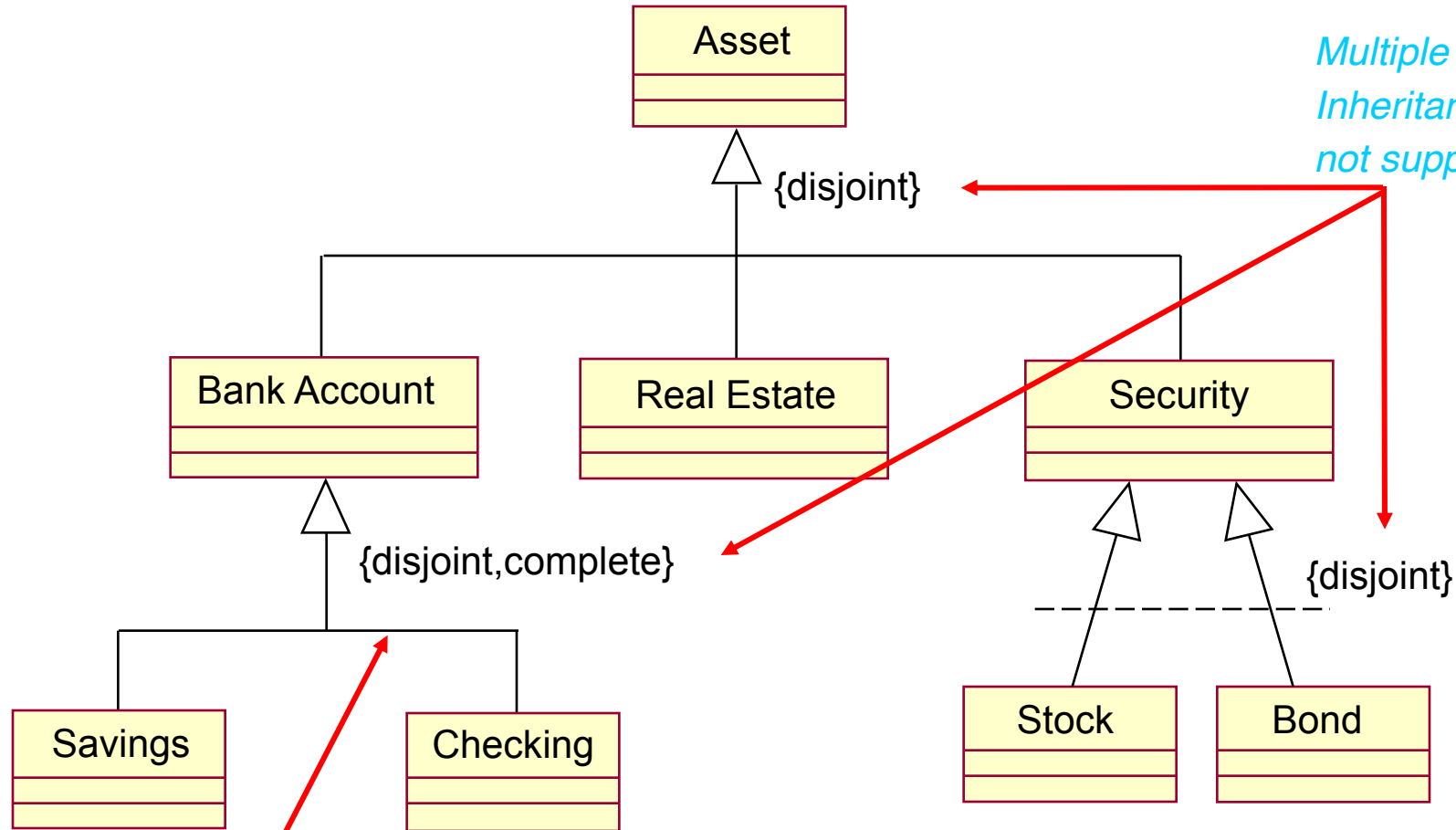
- 四种约束

- 完全 (complete): 泛化集合中子类覆盖所有可能性
    - 不完全 (incomplete): 泛化集合中还存在其他子类
    - 互斥 (disjoint): 对象仅可能是一个子类的实例
    - 重叠 (overlapping): 对象可以是多个子类的实例

- 作用

- 完全/不完全: 指明是否到达继承层次的终点
    - 互斥/重叠: 指明是否支持多继承

# 泛化——关系

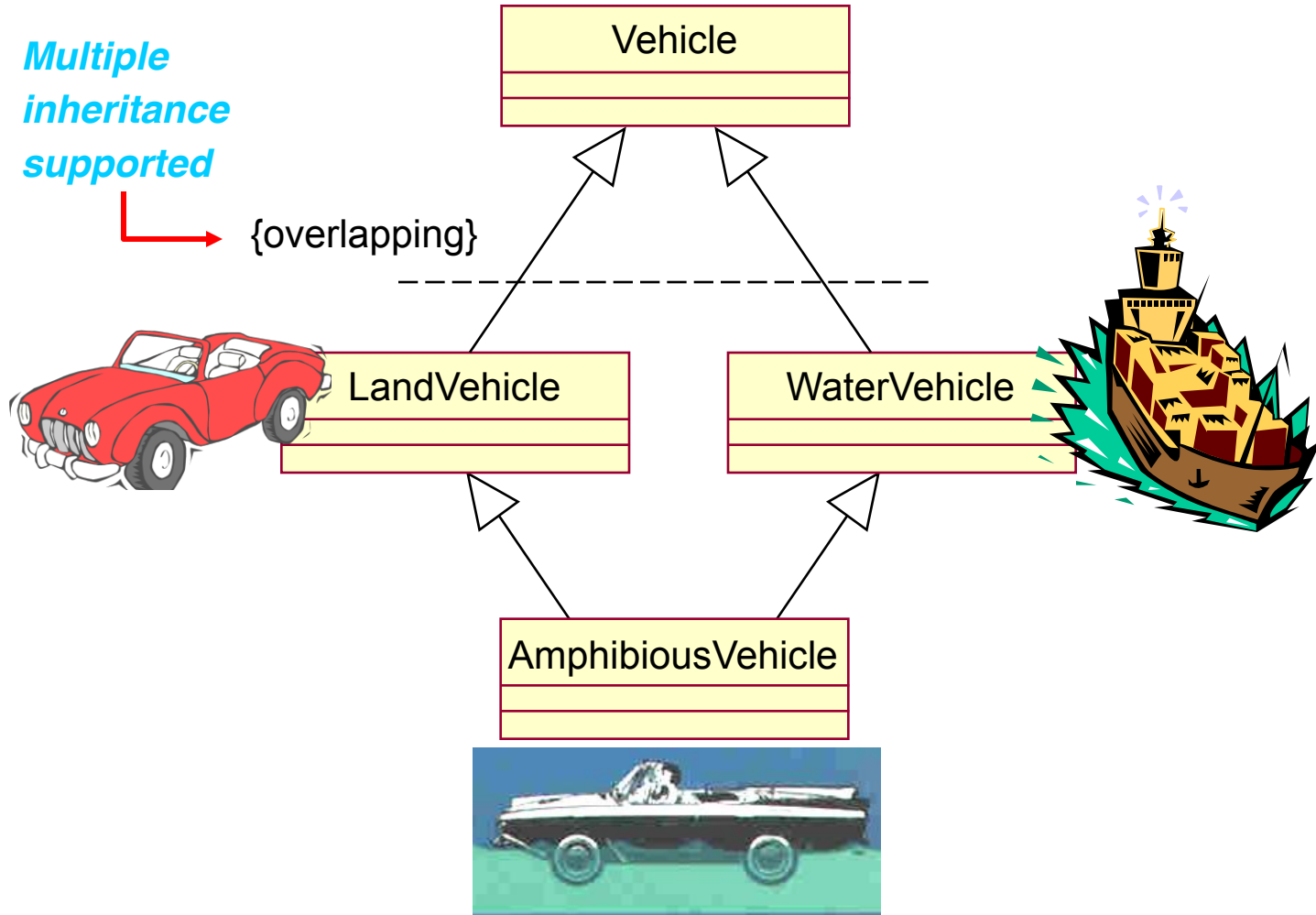


End of inheritance hierarchy

# 泛化——关系

*Multiple  
inheritance  
supported*

└─ {overlapping}





# Boston Duck Tour



EXPERIENCE THE  
RIDE OF *your* LIFE

SAVE  
**\$2<sup>00</sup>**  
per adult\*  
Expires 12/31/16

**BOSTON DUCK TOURS**

Present this ad at any Boston Duck Tours ticket booth to receive your discount. Not to be combined with any other discount or offers. Not valid on phone or email reservations. BDT Code: C-CTM

▲ Prudential 



▲ Kendall/MIT 

**BOSTON DUCK TOURS**

**HOP ON & OFF ALL DAY**

**UPPER DECK TROLLEY TOURS**

**SUPER TOURS**

**BOSTON DUCK TOURS**

[www.bostonsupertours.com](http://www.bostonsupertours.com)  
or contact us at 877.34.ducks (38257)

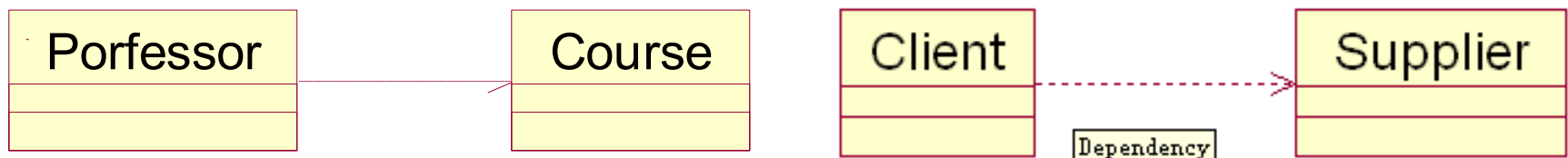
▲ Aquarium  & Park Street  



# 依赖——关系



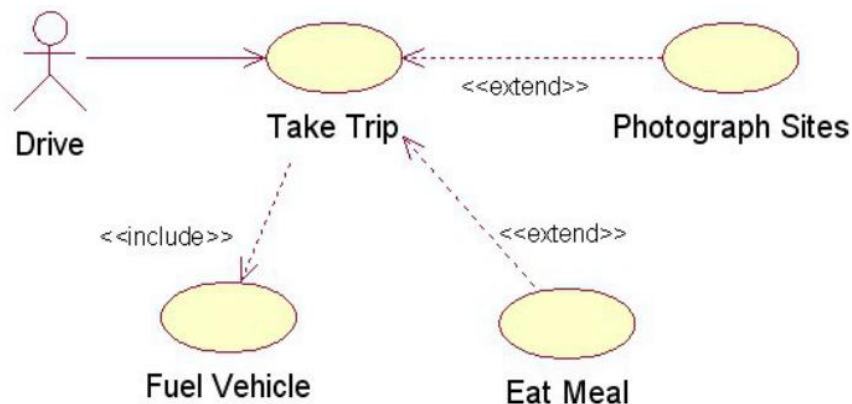
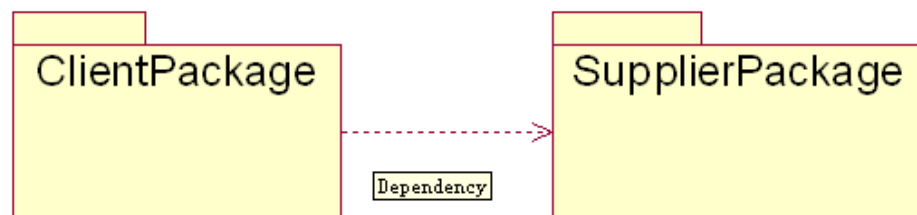
- 依赖关系
  - 描述类元间的使用关系
  - 一个元素(b)的改变会影响另外一个元素(a), 则称存在依赖关系 (a依赖于b)
  - 一个元素的变化可能影响或提供信息给使用它的另一个元素, 反之不然



# 依赖——关系

## • 依赖关系的标准构造型

- 应用于包之间
  - 引入 (import)
  - 访问 (access)
- 应用于用例之间
  - 扩展 (extend)
  - 包含 (include)
- 应用于对象之间
  - 发送 (send)
- .....



# 依赖——关系

## • 依赖关系的标准构造型

- ◆ 抽象(abstraction): 从一个对象中提取一些特性, 并用类方法表示。
- ◆ 绑定(binding): 为模板参数指定值, 以定义一个新的模板元素。
- ◆ 组合(combination): 对不同类或包进行性质相似融合。
- ◆ 许可(permission): 允许另一个对象对本对象的访问。
- ◆ 使用(usage): 声明使用一个模型元素需要用到已存在的另一个模型元素, 这样才能正确实现使用者的功能(包括调用、实例化、参数、发送)。
- ◆ 调用(call): 声明一个类调用其他类的操作的方法。
- ◆ 导出(derive): 声明一个实例可从另一个实例导出。
- ◆ 实例(instantiation): 关于一个类的方法创建了另一个类的实例声明。
- ◆ 参数(parameter): 一个操作和它参数之间的关系。
- ◆ 精化(refine): 声明具有两个不同语义层次上的元素之间的映射。

# 依赖——关系

## • 常见依赖关系与JAVA实现

依赖构造型	含义	示例程序
<<create>>	目标对象是由源对象创建的，目标对象创建后将传递给系统其他部分	<pre>public class ClassA {     public ClassB createB() {         return new ClassB();     } }</pre>
<<local>>或<<call>>	源类对象创建目标类对象实例，并将该实例包含在一个局部变量中	<pre>public class ClassA {     public void testMethod() {         ClassB test=new ClassB();     } }</pre>
<<parameter>>	源类对象通过它的某个成员函数的参数得以访问目标类对象实例	<pre>public class ClassA {     public void     testMethod(ClassB test) { } }</pre>

# 关联——关系



- 关联关系
  - 描述两个或多个类间的结构关系
  - 两个类间的关联表明
    - 一个类的对象与另一个的对象相联系
    - 实例：对象间的链接（link）
    - 修饰：名称，角色，导航性，多重性
    - 约束



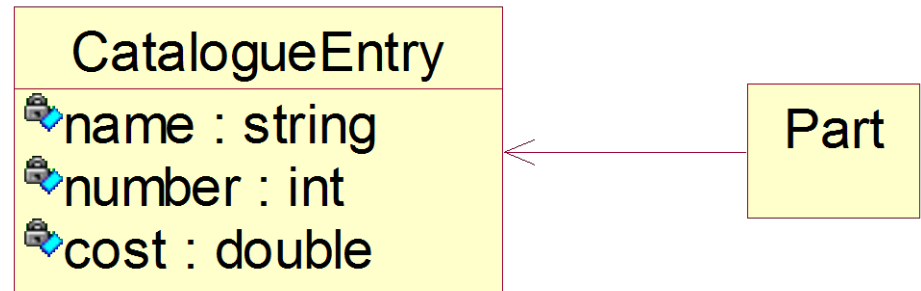
# 关联——关系



## • 关联关系举例

```
class CatalogueEntry {  
    private string name;  
    private int number;  
    private double cost;  
    public double getCost() {  
        return cost;  
    }  
}
```

```
class Part {  
    private CatalogueEntry entry;  
    public double cost() {  
        return entry.getCost();  
    }  
}
```



# 关联——关系

- 关联关系的修饰

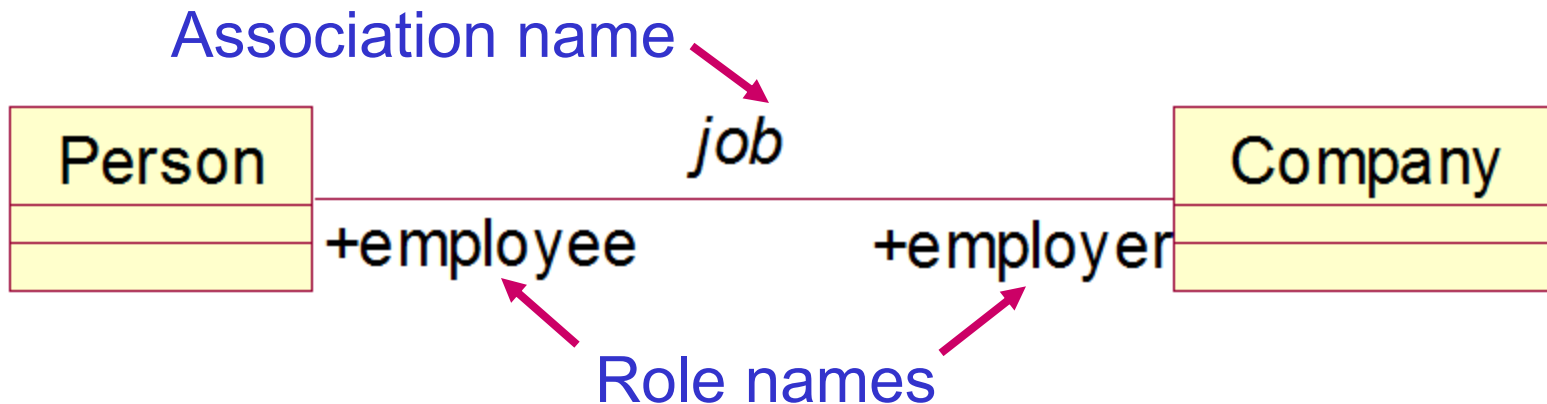
- 关联名称

- 角色名称

- 表明关联的一端类的对象所扮演的角色

- 名词（短语）

- 对角色名称添加可见性符号描述关联端点的可见性



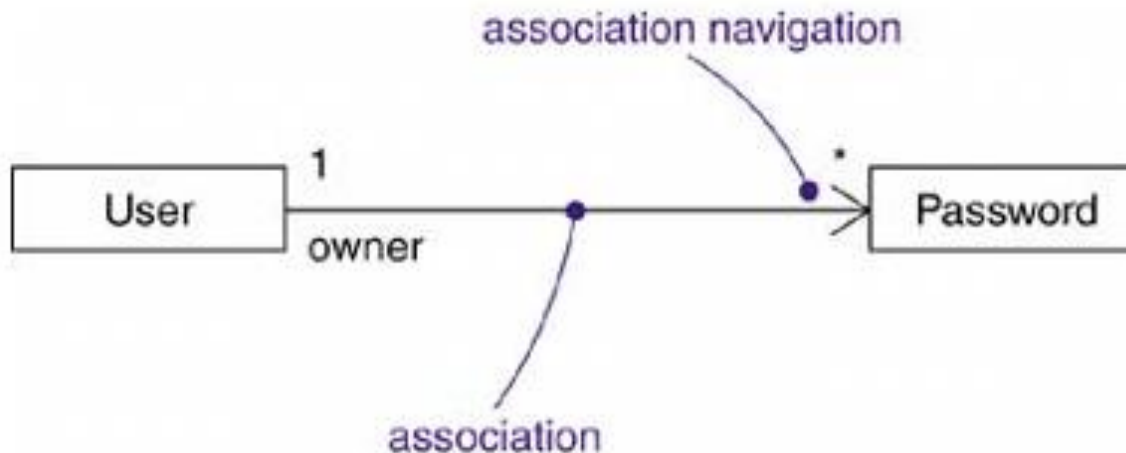
# 关联——关系



- 关联关系的修饰

- 导航性

- 限制关联的方向是单向的
    - 表明可从源类的一个对象遍历目标类的对象
    - 最小化类间耦合





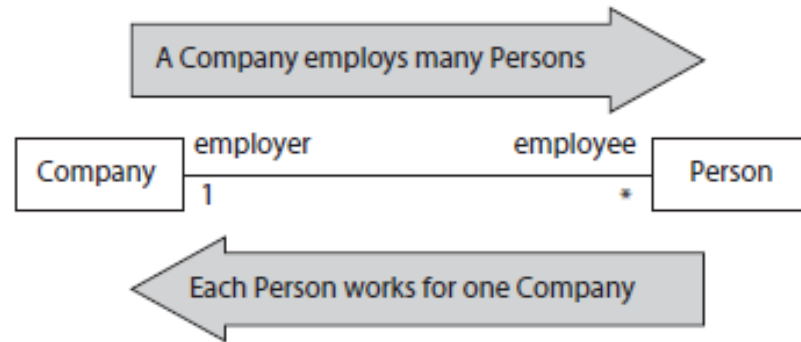
# 关联——关系



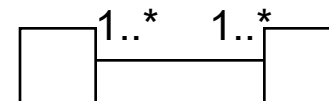
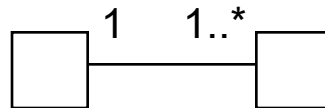
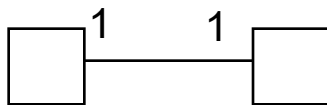
- 关联关系的修饰

- 多重性

- 说明一个关联的实例中有多少个链接的对象



- 三种映射：一对一，一对多，多对多



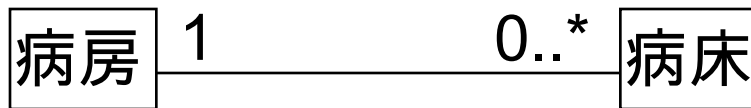
# 关联——关系



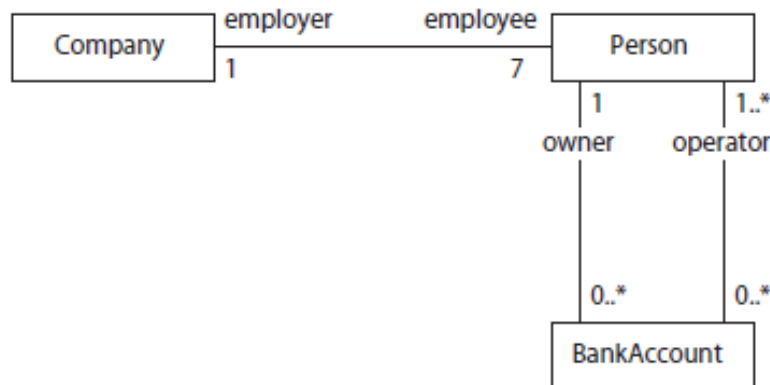
- 关联关系的修饰

- 多重性

- 条件关联：“0”指明某些对象不参与关联



- 描述业务规则、需求和约束



Unspecified	
Exactly One	1
Zero or More	0..*
Zero or More	*
One or More	1..*
Zero or One (optional scalar role)	0..1
Specified Range	2..4
Multiple, Disjoint Ranges	2, 4..6

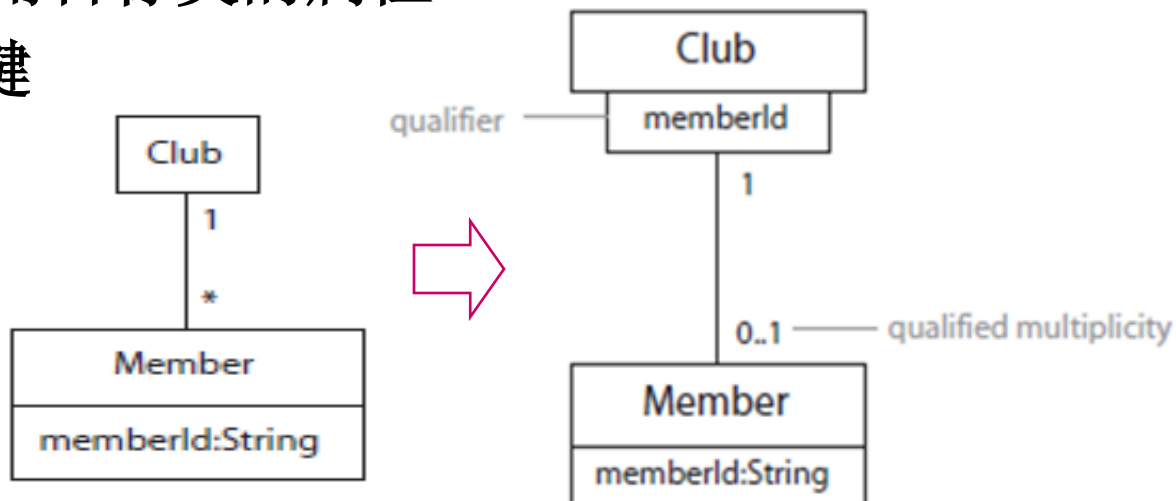
# 关联——关系

- 关联关系的修饰

- 受限关联 (qualified association)

- 用于一对多/多对多的关联
    - 目的：区分“多”端的对象集合
    - 使用唯一键从对象集合中选定特殊对象
    - 限定符：引用目标类的属性

指定唯一键



# 关联——关系



- 关联关系的约束

- 详述关联含义的细微差别

- UML提供的5种约束

- 有序 (ordered)

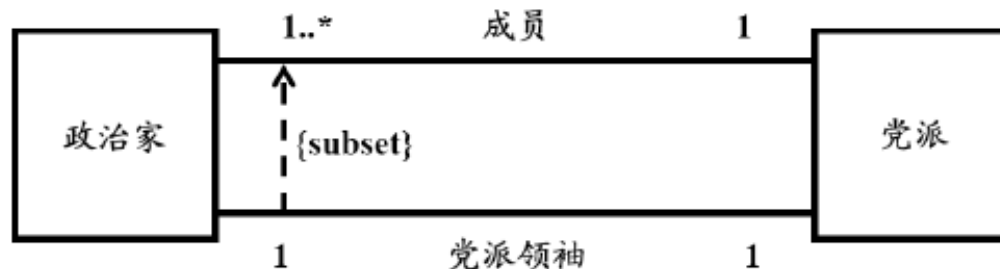
- 有序集合 (ordered set)

- 集合 (set)

- 表 (list) 或序列 (sequence)

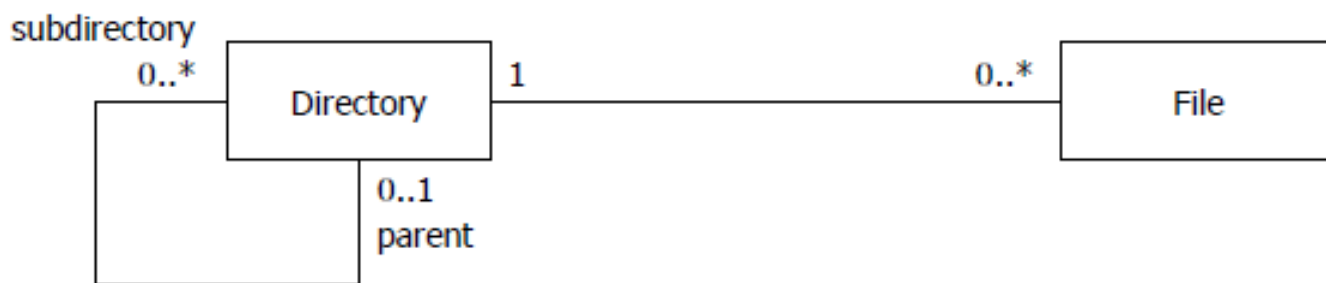
- 袋 (bag)

- 自定义的约束

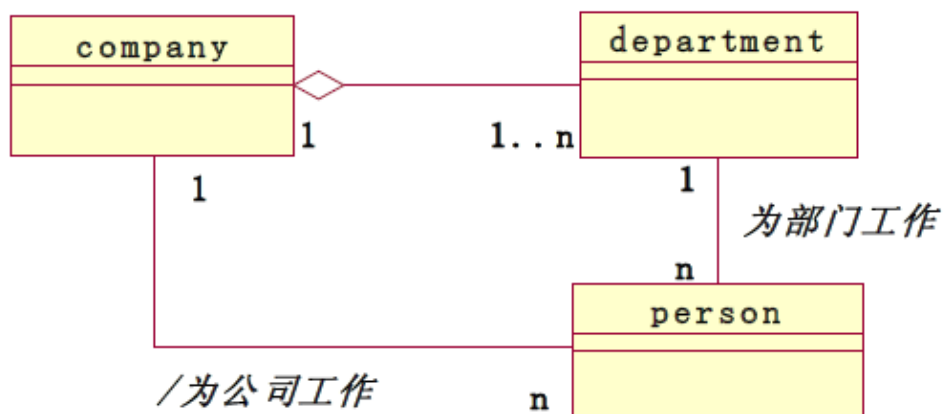


# 关联——关系

- 自反关联： 同一个类的对象间存在链接



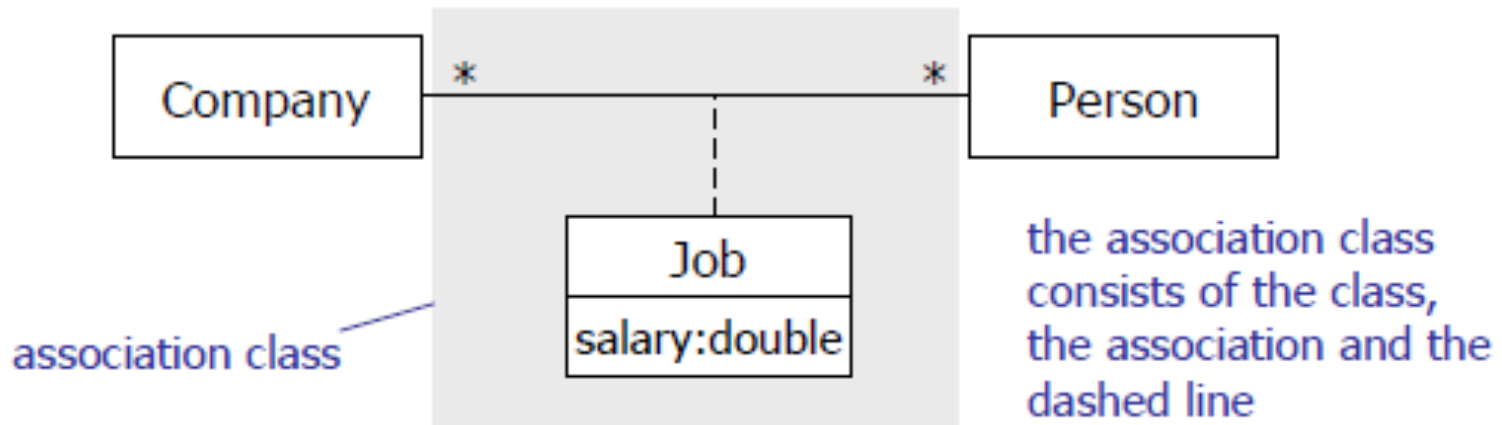
- 派生关联： 关联间的传递关系



# 关联类——关联

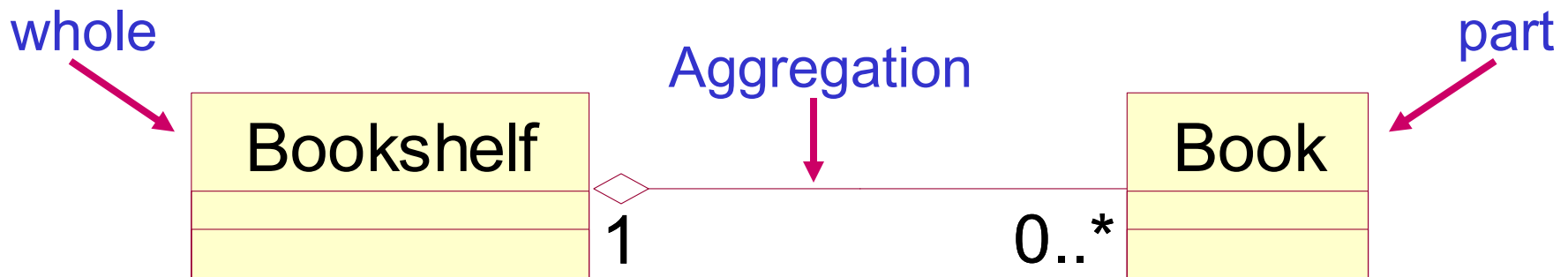


- 关联类
  - 表达一组属于关联关系本身的特征
  - 包含关联连线、悬挂线和类方框
  - 实例：具有属性和操作的链接



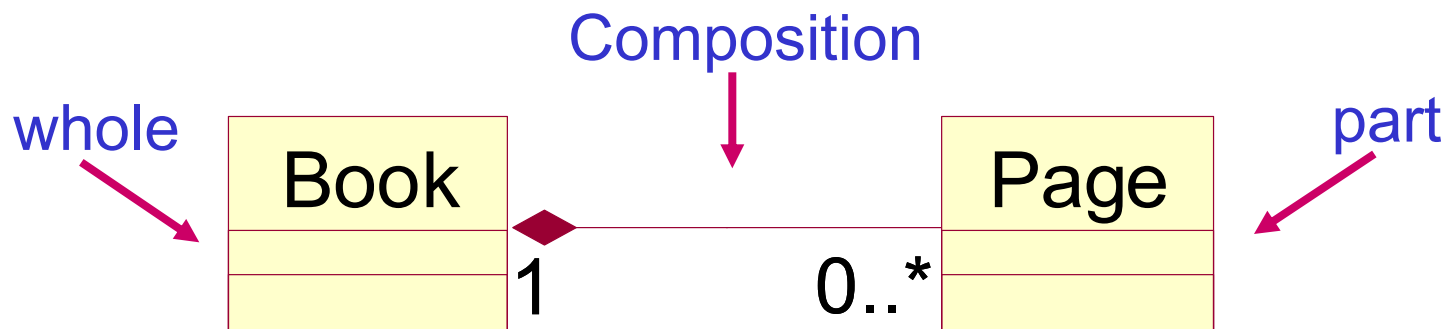
# 聚集——关联

- 聚集 (Aggregation)
  - 一种特定的关联关系
  - 建模“整体-部分 (whole-part)”关系



# 组合——关联

- 组合 (Composition)
  - 更强形式的聚合
  - part无法离开whole单独存在





# 聚集——组合



- 聚集VS组合

- 聚集

- 概念上区分整体与部分
    - 与整体和部分的生命周期无关



Some objects are weakly related, like a computer and its peripherals

- 组合

- 加强的拥有一关系
    - 整体与部分的生命周期一致



Some objects are strongly related, like a tree and its leaves

# 关联——依赖



- 关联VS依赖

- 依赖

- 使用关系;相对临时的对应关系

- 关联

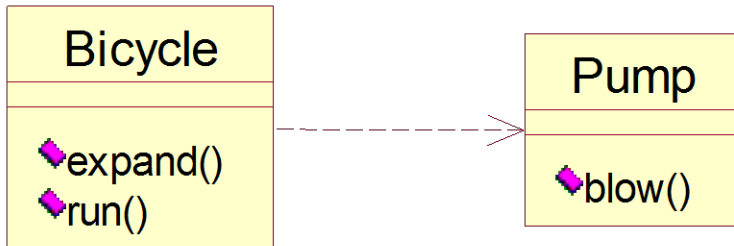
- 描述类的对象间相互作用的结构路径;相对固定的对应关系

数据驱动	对于每一对类，若需要从一个类的对象到另一个类的对象导航，就要在这两个类之间说明一个关联
行为驱动	对于每一对类，若一个类的对象要与另一个类中不作为其操作参数的对象交互，就要在这两个类之间说明一个关联

# 关联——依赖

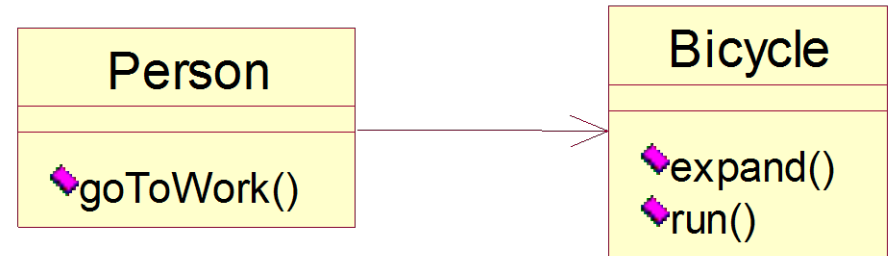


## • 关联VS依赖



```
Public class Bicycle{
public void expand(Pump pump){
pump.blow(); }
}
```

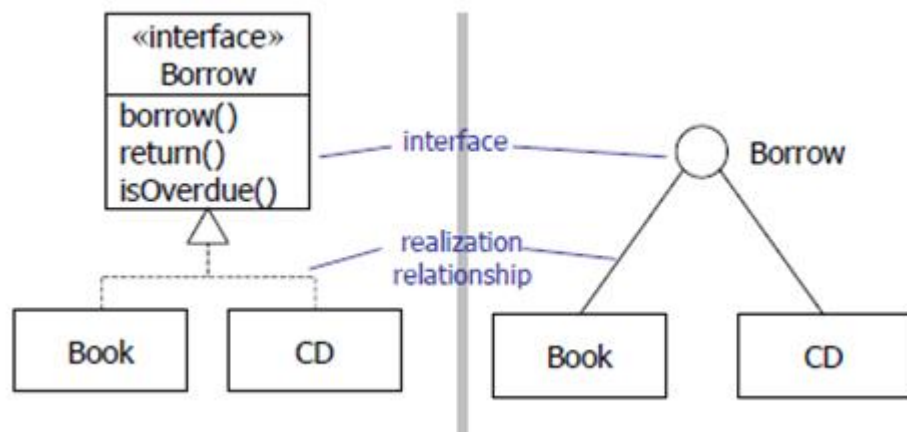
```
myBic.expand(pumpRepairShed1);
myBic.expand(pumpRepairShed2);
```



```
public class Person{
private Bicycle bicycle;
public Bicycle getBicycle(){
return bicycle; }
public void setBicycle(Bicycle bicycle){
this.bicycle=bicycle; }
public void goToWork(){
bicycle.run(); }
}
```

# 接口

- 接口
  - 类的一个构造型
    - 一组操作集合
    - 不包含任何实现，不能被实例化
  - 供接口 (provided interface)
    - 由类元实现的接口
    - 描述类元承诺提供的一组服务



# 接口——抽象类



- 接口与抽象类

- 共同点

- 不能实例化
    - 有未实现的方法声明

- 不同点

- 抽象类可以有属性及非抽象的方法
    - 接口中不能有实现方法

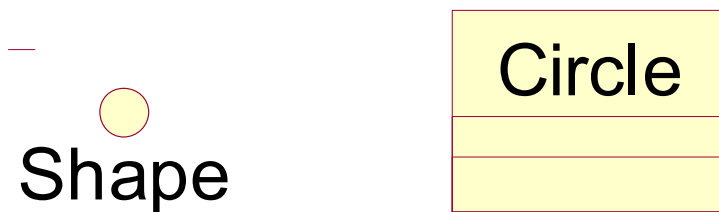
- 本质区别

- 接口: “like a”; 抽象类: “is a”

# 实现——关系



- 实现关系
  - 类元间的语义关系
  - 一个类元保证实现另一个定义的契约



```
interface Shape{  
    public void draw();  
}  
class Circle implements Shape {  
    public void draw() {.....}  
    .....  
}
```

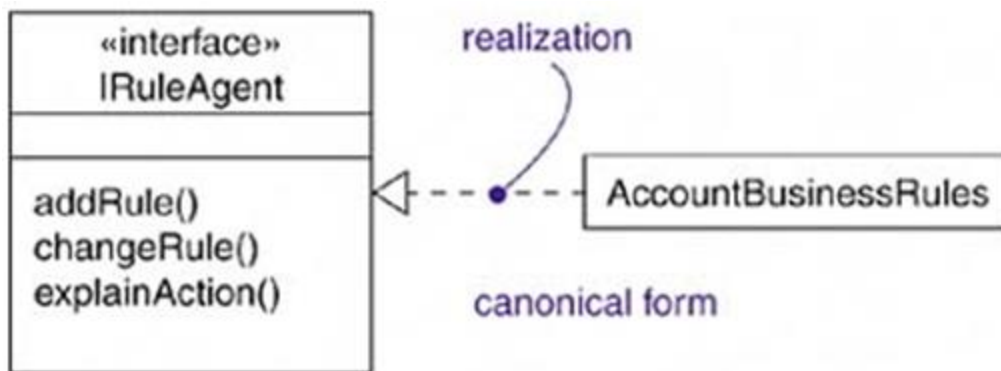
# 实现——关系



- 实现关系

- 表示方法

- 规范方式: <<interface>>



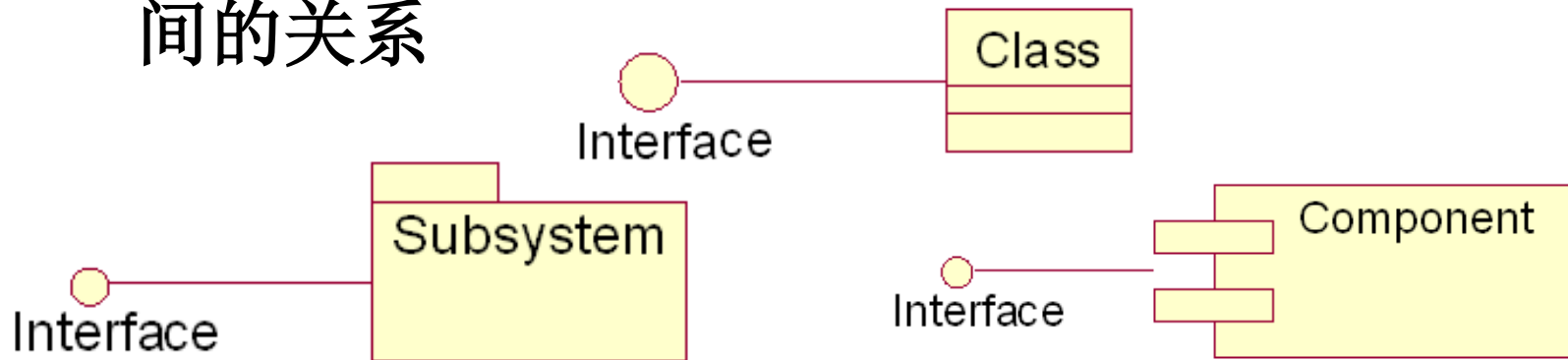
- 省略方式: 棒棒糖表示法



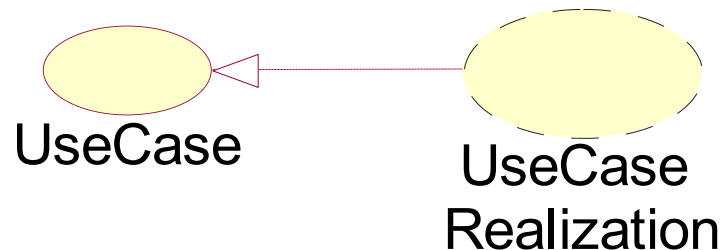
# 实现——关系

- 实现关系

- 描述接口与为其提供操作的类/组件/子系统间的关系



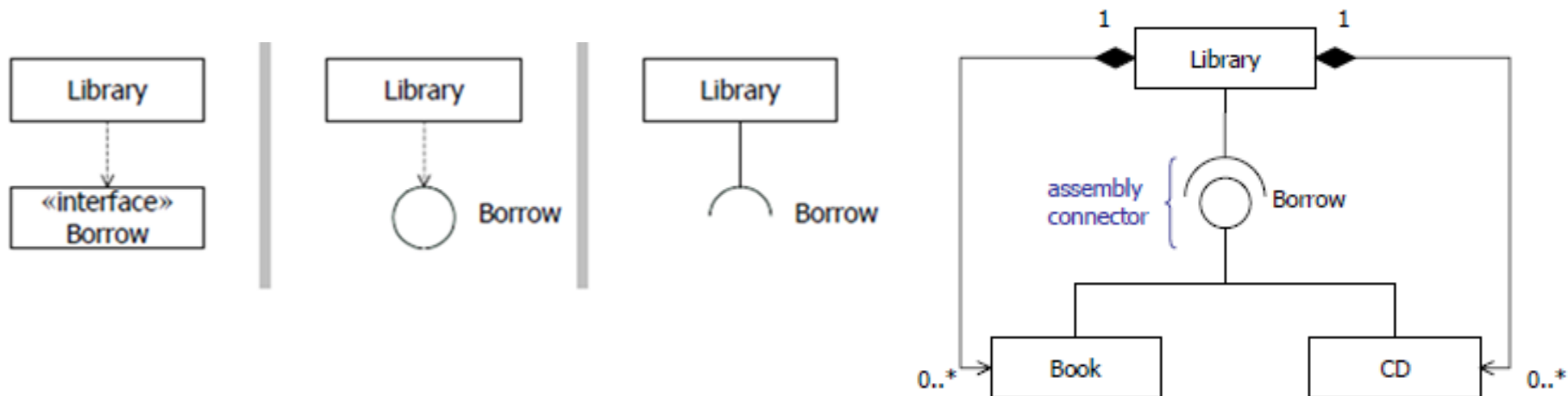
- 描述用况与其实实现间的关系





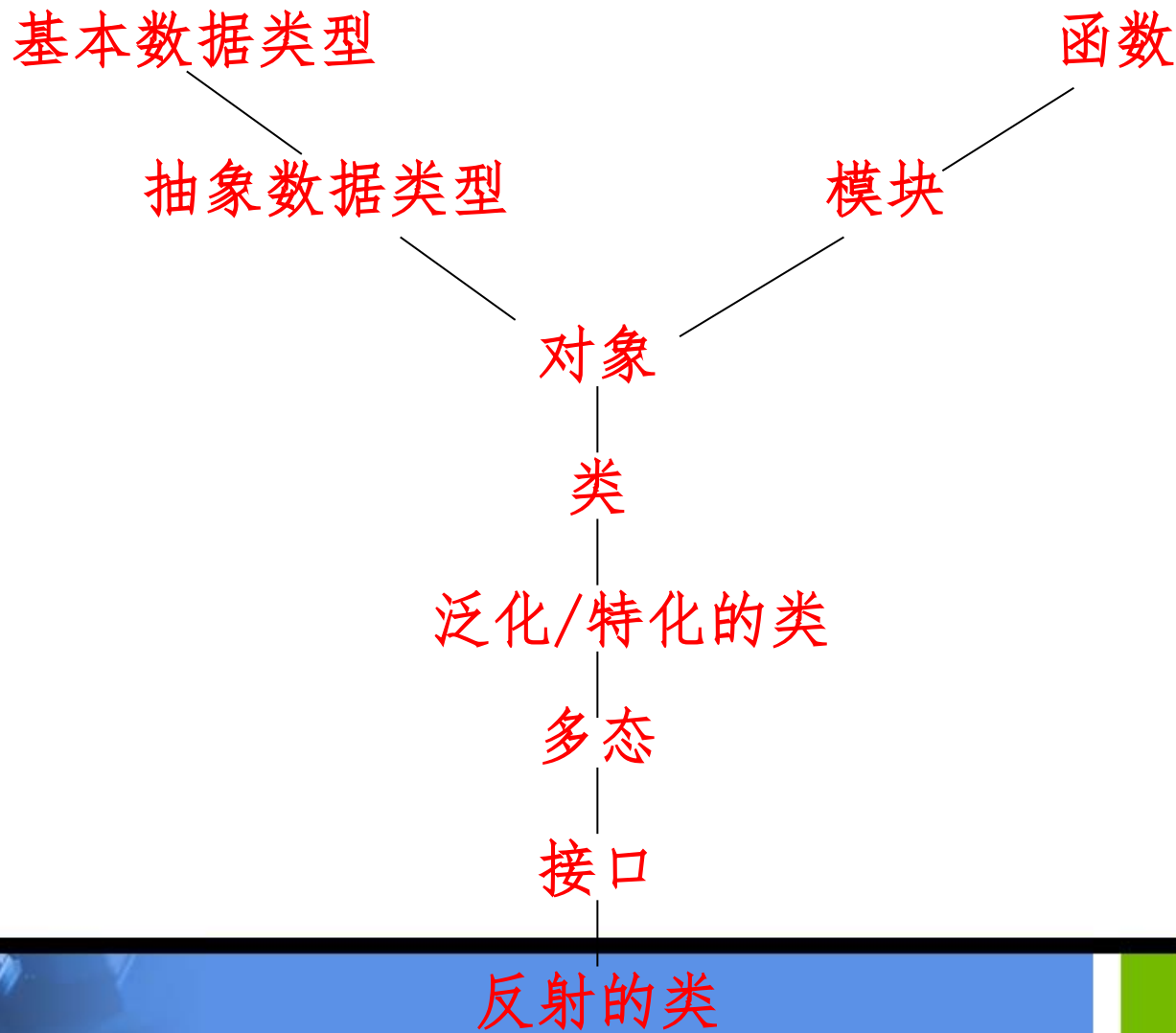
# 实现——接口

- 接口
  - 需接口 (required interface)
    - 需要被实现的接口
    - 理解并需要由接口定义的特定服务集合



# 抽象类——接口

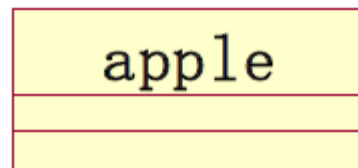
- 数据抽象的演化:



- 类图的抽象层次

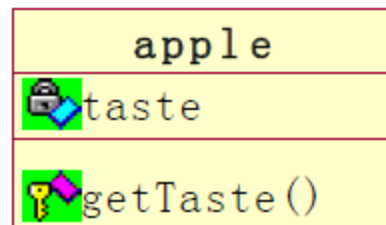
- 概念层类图

- 描述应有领域中的概念，仅包含类名，不考虑细节
    - 位于业务建模阶段



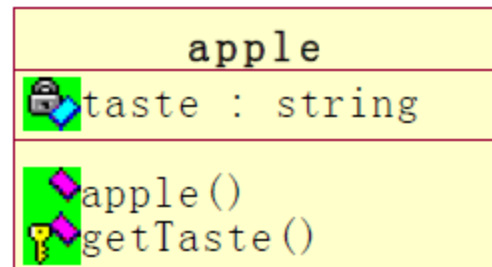
- 说明层类图

- 不针对具体语言，包含一些细节特性
    - 以分析类和分析模型表示



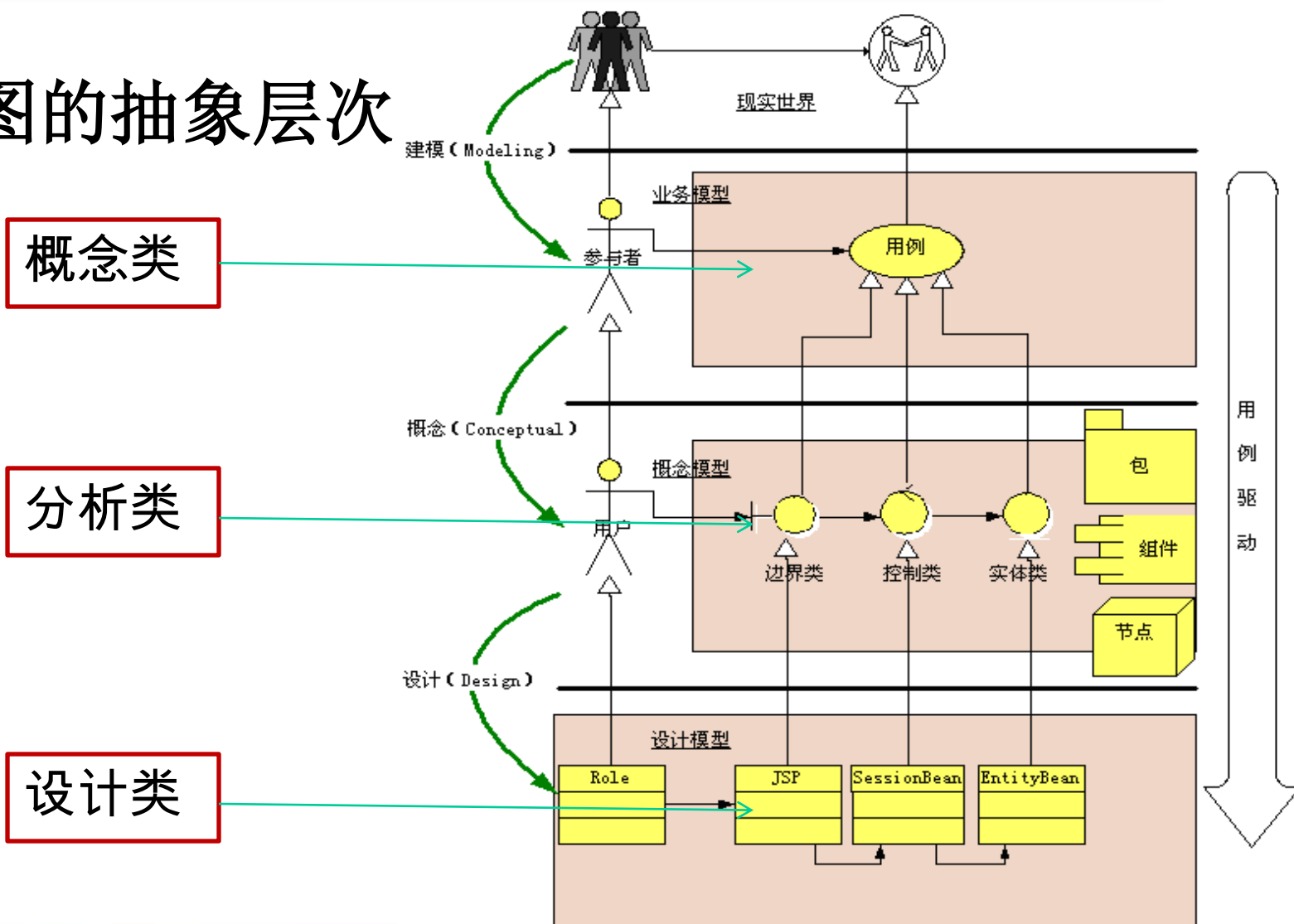
- 实现层类图

- 针对具体语言，考虑类的实现细节



# 类图

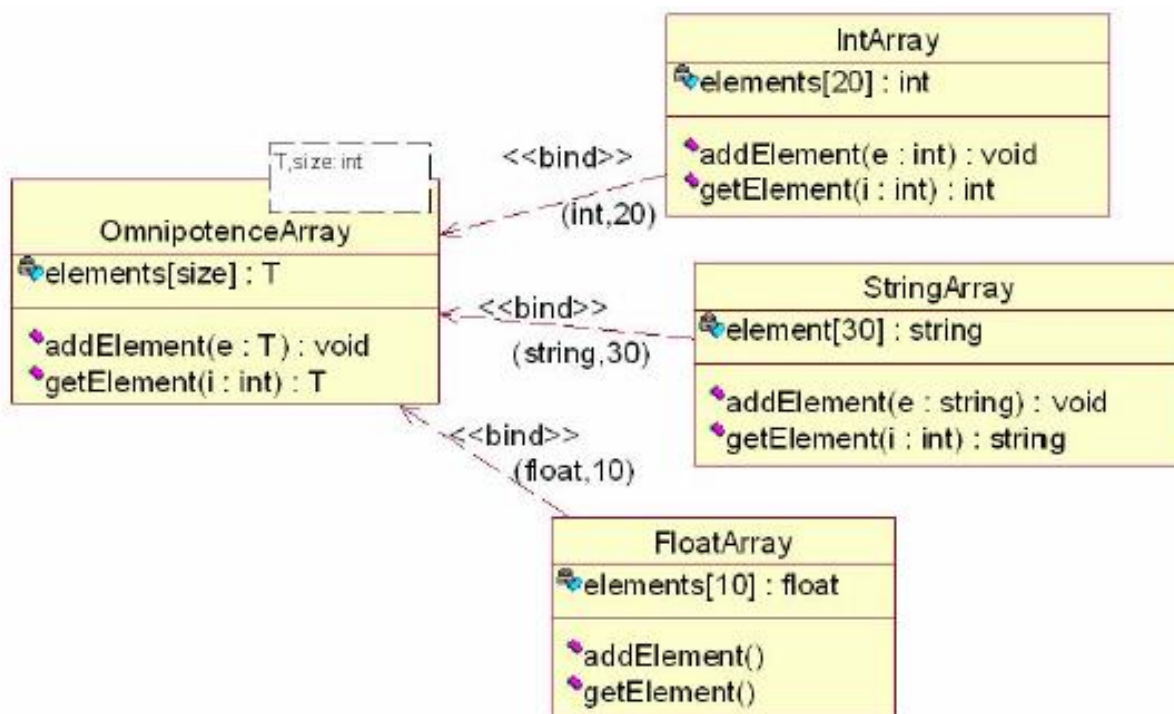
## • 类图的抽象层次



# 类图

- 高级类

- 关联类：即是关联也是类
- 模板类：根据占位符或参数来定义类



- 高级类

- 主动类

- 主动类的实例称为主动对象，一个主动对象拥有一个控制线程并且能发起控制活动；它不在别的线程、堆栈或状态机内运行，具有独立的控制期

- 嵌套类

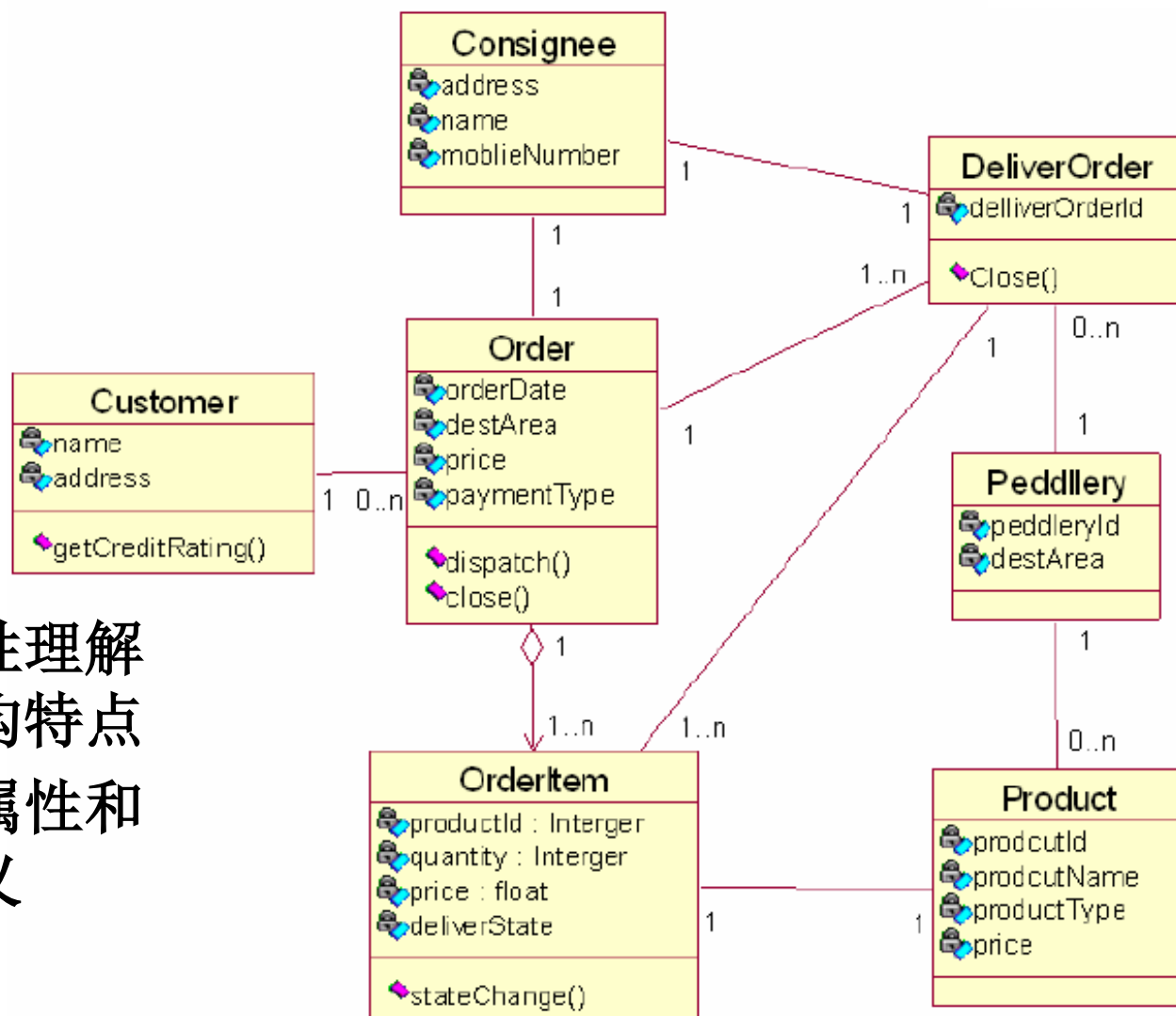
- 将一个类的定义放在另一个类定义的内部（在诸如Java的语言中）
    - 嵌套类是声明在它的外层类中的，因此只能够通过外层类或外层类的对象对它进行访问

# 类图

## • 阅读类图

### - 步骤

- 读出类
- 读出类间的关系
- 结合多重性理解类图的结构特点
- 理解各个属性和操作的含义



- 阅读类图

- 读出类

- Order、OrderItem、Customer、Consignee、Deliver--Order、Peddlery、Prodcut

- 读出类间的关系

- 从图中关系最复杂（线最密集）的类开始阅读
    - OrderItem和Order之间是组合关系
    - 根据箭头方向可知Order包含OrderItem。Order类和Customer、Consignee、DeliverOrder是关联关系



# 类图



## • 阅读类图

– 多重性：说明关联的两个类之间的数量关系

源类及多重性	目标类及多重性	分析	
Customer(1)	Order(0...n)	订单是属于某个客户的，网站的客户可以有0个或多个订单	
Order(1)	Consignee(1)	每个订单只能够有一个收货人	
Order(1)	OrderItem(1...n)	订单是由订单项组成的，至少要有有一个订单项，最多可以有n个	
Order(1)	DeliverOrder(1...n)	一个订单有一个或多个送货单	说明：系统根据订单项的产品所属的商户，将其分发给商户，拆成了多个送货单！
DeliverOrder(1)	OrderItem(1...n)	一张送货单对应订单中的一到多个订单项	
DeliverOrder(1)	Consignee(1)	每张送货单都对应着一个收货人	
Peddler(1)	DeliverOrder(0...n)	每个商户可以有相关的0个或多个送货单	
OrderItem(1)	Product(1)	每个订单项中都包含着唯一的一个产品	
Peddler(1)	Product(0...n)	产品是属于某个商户的，可以注册0到多个产品	

- 阅读类图

- 理解属性和操作

- Order类:

- 四个属性: orderDate, destArea, price, pamentType

- 两个方法: dispatch()和close(), 从名字中可以猜出它们分别实现“分拆订单生成送货单”和“完成订单”

- DeliverOrder类:

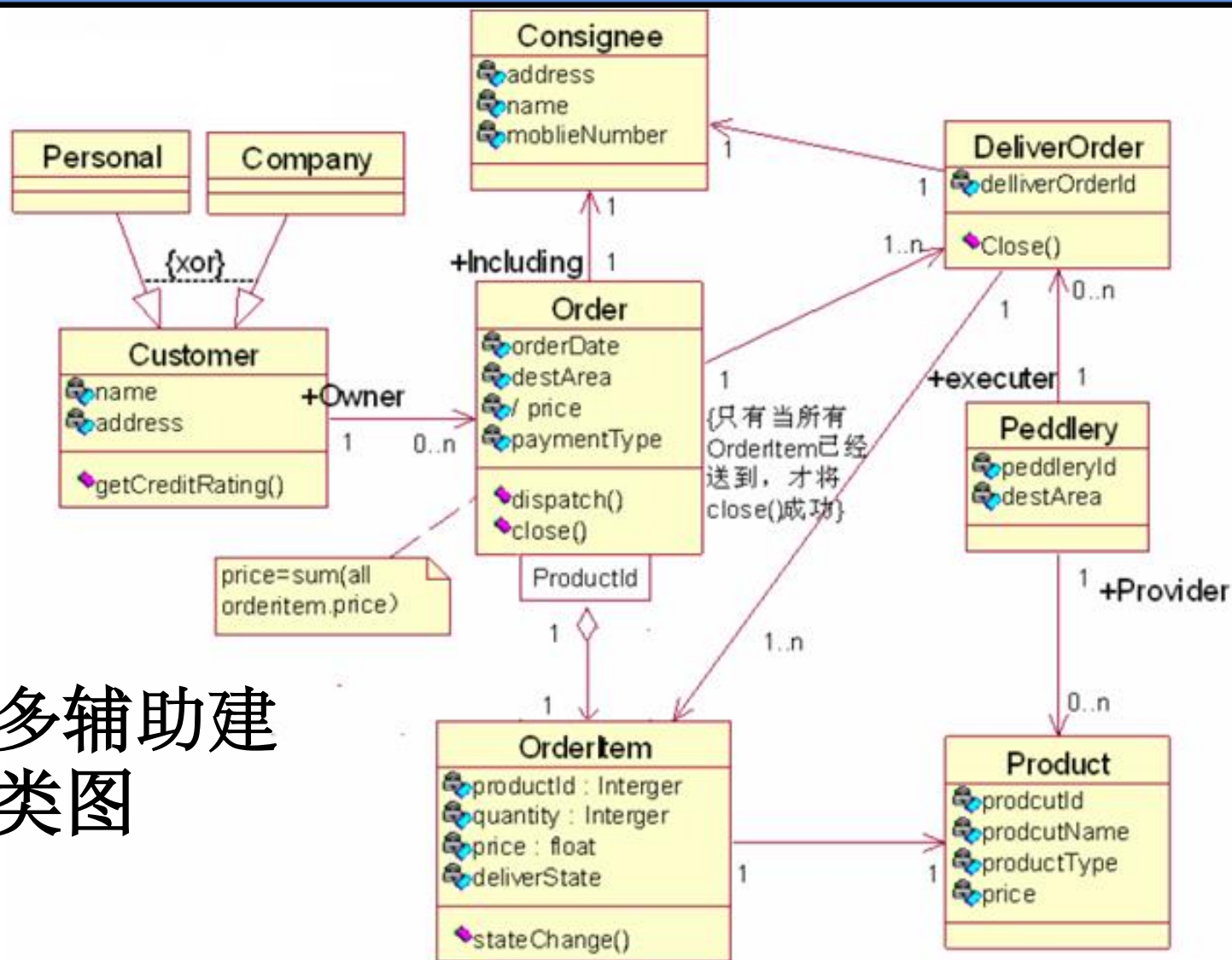
- 一个属性: deliverOrderID

- 一个方法: close(), 表示“完成送货”

- OrderItem类:

- 一个方法: stateChange(), 改变其“是否交给收货人”标志位

# 类图



- 使用了更多辅助建模元素的类图

# 对象图



- 对象

- 代表一个单独的、可确认的单元或实体

- 状态

- 所有属性（通常是静态的）

- 属性的当前值（通常是动态的）

- 行为

- 一个对象根据它的状态改变和消息传送所采取的行动和所做出的反应



- 对象VS类

- 类元/实例关系

- 对象是一个存在于时间和空间中的具体实体，而类仅代表一个抽象，抽象出对象的“本质”
    - 类是共享一个公用结构和一个公共行为对象集合
    - 类是静态的，对象是动态的
    - 类是一般化，对象是个性化；类是定义，对象是实例；类是抽象、对象是具体

# 对象图



- 对象的表示法

- 对象标识

- 对象名: 类名

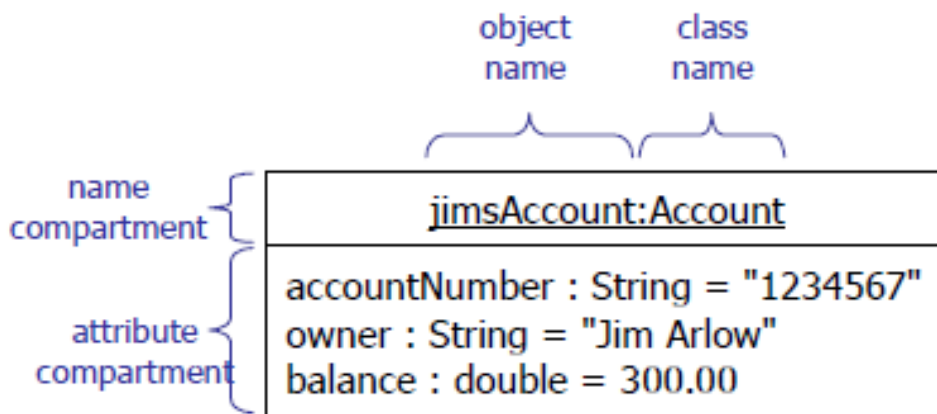
- 对象属性

- 所有属性值都已经确定，通常会在后面列出其值

uml :  
course

: course

UML

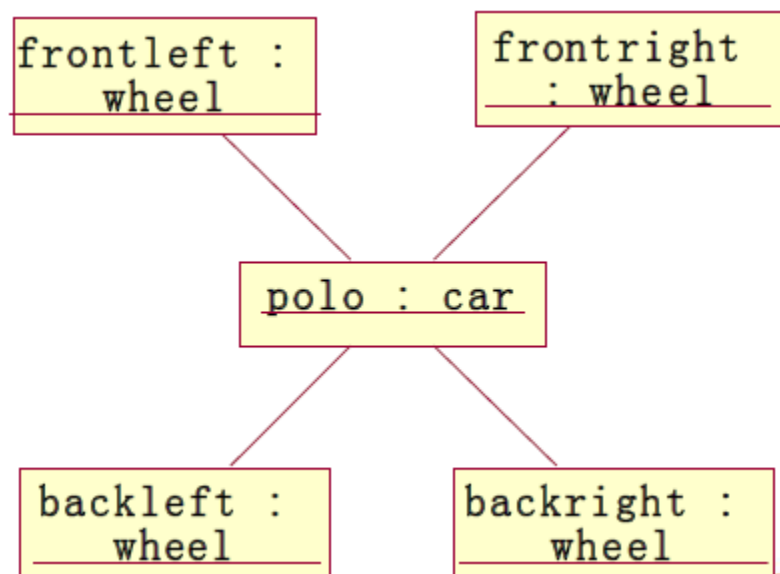


- 对象图
  - 描述某一时间点上的一组对象及其联系
    - 对象
    - 链 (link)
    - 注解/约束
  - 链
    - 类间关联关系的实例
    - 对象间的通信路径

# 对象图



- 对象图
  - 表达了交互的静态部分
    - 由协作的对象组成
    - 不包含传递的任何消息





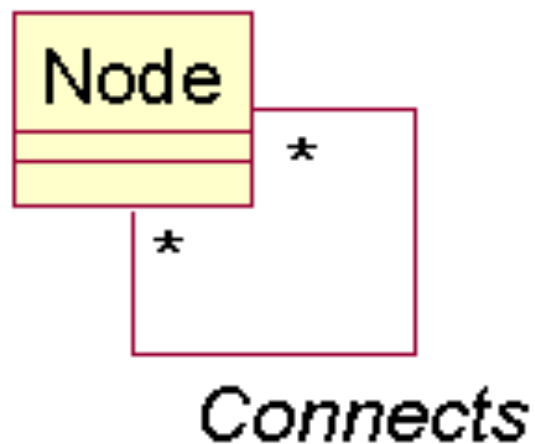
- 对象图的作用
  - 论证类模型的设计
    - 通过对象图来模拟出一个运行时的状态
    - 研究在运行时设计的合理性
  - 分析和说明源代码
    - 在分析源代码时，可以通过对象图来细化分析
    - 对于逻辑较复杂的类交互时，可以画出一些对象图来做补充说明

# 对象图

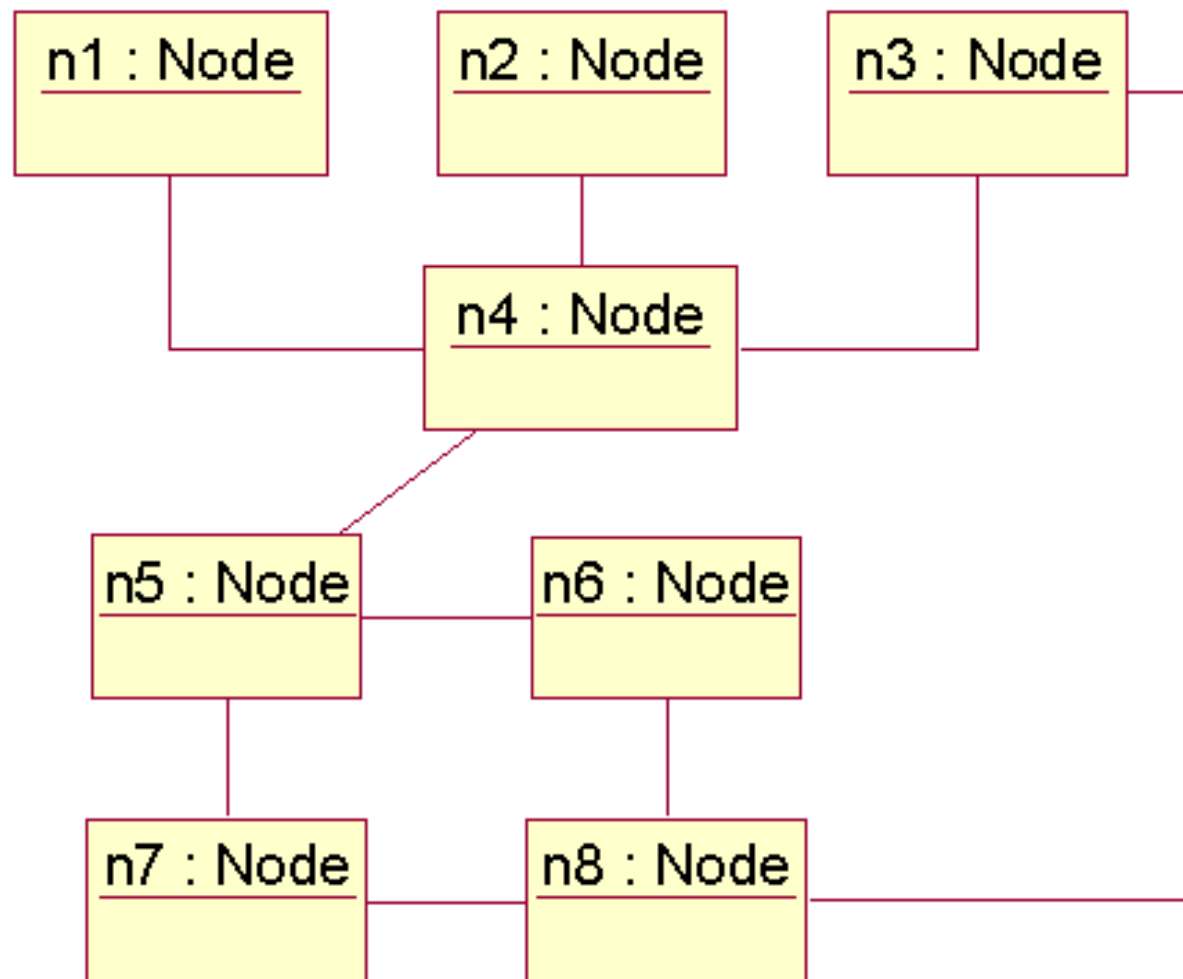
- **对象图：**
  - 表示一组对象和它们之间的联系。
  - 对象图是一个系统的详细状态在某一时刻的快照。
  - 常用于表示复杂的类图的一个实例。

# 对象图

- 例：表示网络间节点关系的类图及其一个对象图的例子：



类图



# 对象图

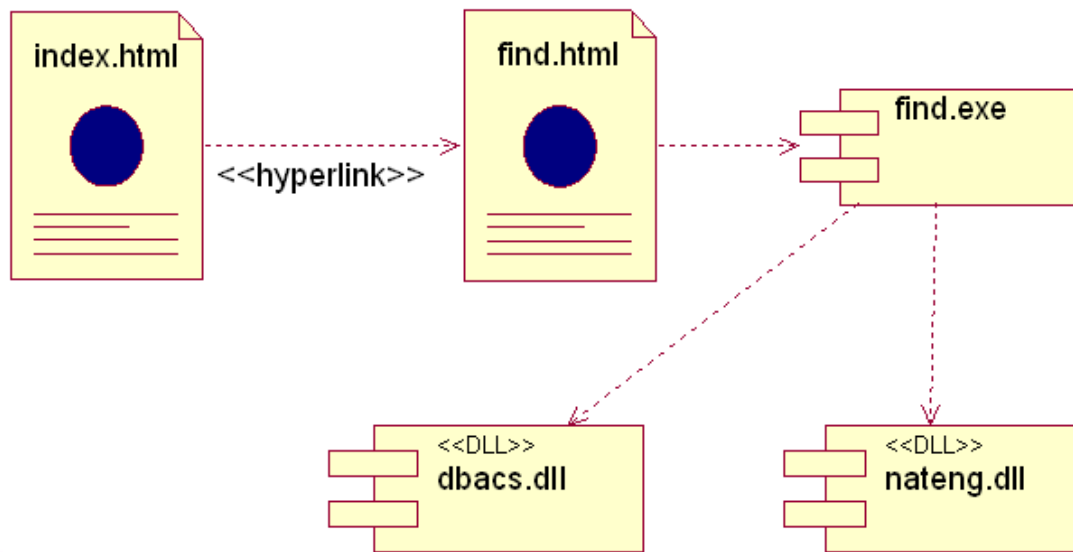
- 总结:

- 对象图的模型元素有对象和链(link)。对象是类的实例；对象之间的链是类之间的关联的实例。
- 对象与类的图形表示相似，UML中对象图与类图具有相同的表示形式。
- 对象图实质上是类图的实例。
- 对象图的使用相当有限，主要用于表达数据结构的示例，以及了解系统在某个特定时刻的具体情况。

# 组件和组件图

- 定义:

- **组件**是系统中遵从一组接口且提供其实现的物理的、可替换的部分。
- **组件图**则显示一组组件以及它们之间的相互关系。包括编译、链接、执行时组件之间的依赖关系。



# 组件图



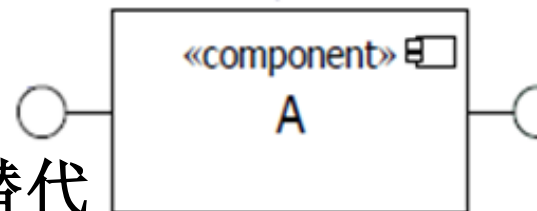
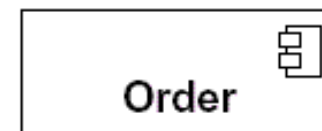
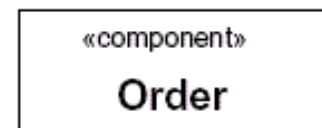
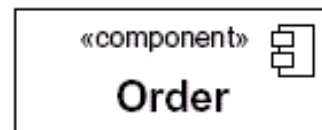
- 组件

- 组件的定义

- 系统中逻辑的可替换部分
    - 遵循并提供对一组接口的实现
    - 实现视图的基本单元

- “黑盒”

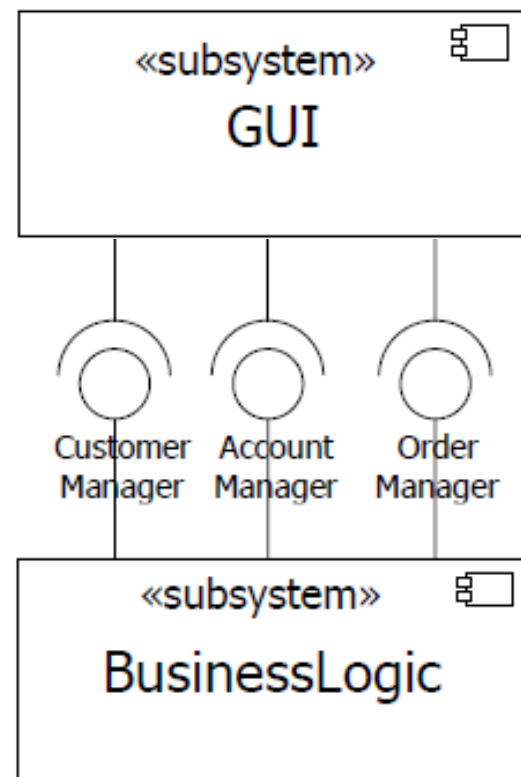
- 外部行为由接口定义
    - 拥有只能通过其接口访问的操作
    - 可被支持相同协议的另一个组件替代



# 组件图

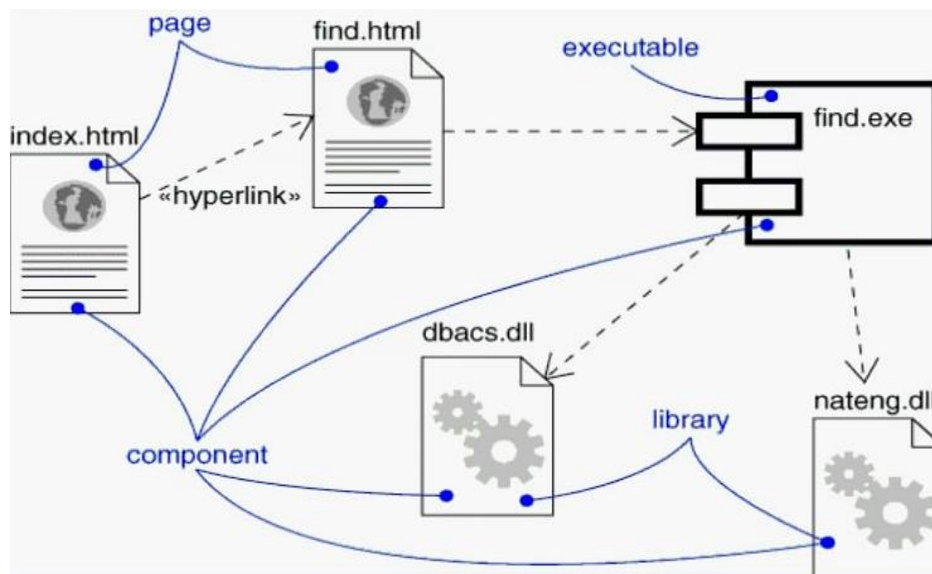


- 组件类型
  - 表示纯粹的逻辑构建
  - 子系统
    - 系统分解的单元
    - 带有构造型<<subsystem>>的组件
  - 基于组件的开发 (Component-based development)



# 组件图

- 组件类型
  - 软件代码及其等价物
  - 类型
    - 部署组件
    - 工作产品组件
    - 执行组件





# 组件和组件图

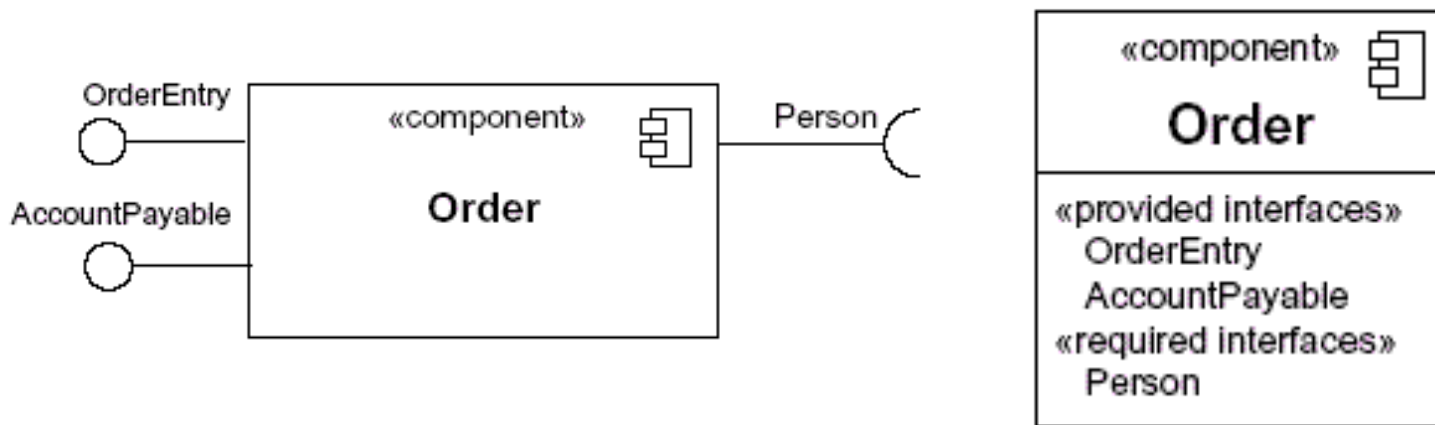
- **组件的类型:**

- 一般说来, 组件就是一个实际文件, 可以有以下几种类型:
  - (1) **deployment component部署组件**, 运行系统需要配置的组件, 如Java虚拟机等。
  - (2) **work product component工作产品组件**, 如源代码文件, 数据文件等, 这些组件可以用来产生部署组件。如Java类, JAR文件, dll文件, 数据库表等。
  - (3) **execution component执行组件**, 系统执行后得到的组件。如EJB, 动态Web页, exe文件, COM+对象, CORBA对象等。

# 组件图



- 组件与接口
  - 接口是组件的粘合剂
  - 接口定义组件的外部行为
    - 供接口：向其他组件提供的服务
    - 需接口：向其他组件请求的服务

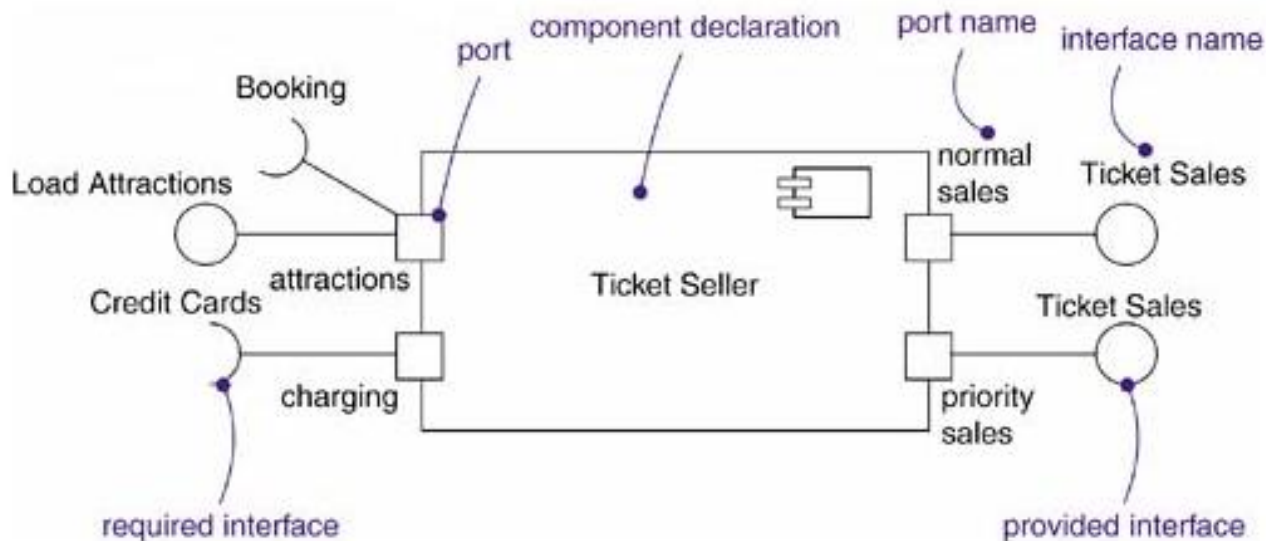


# 组件图

- 组件与接口

- 端口

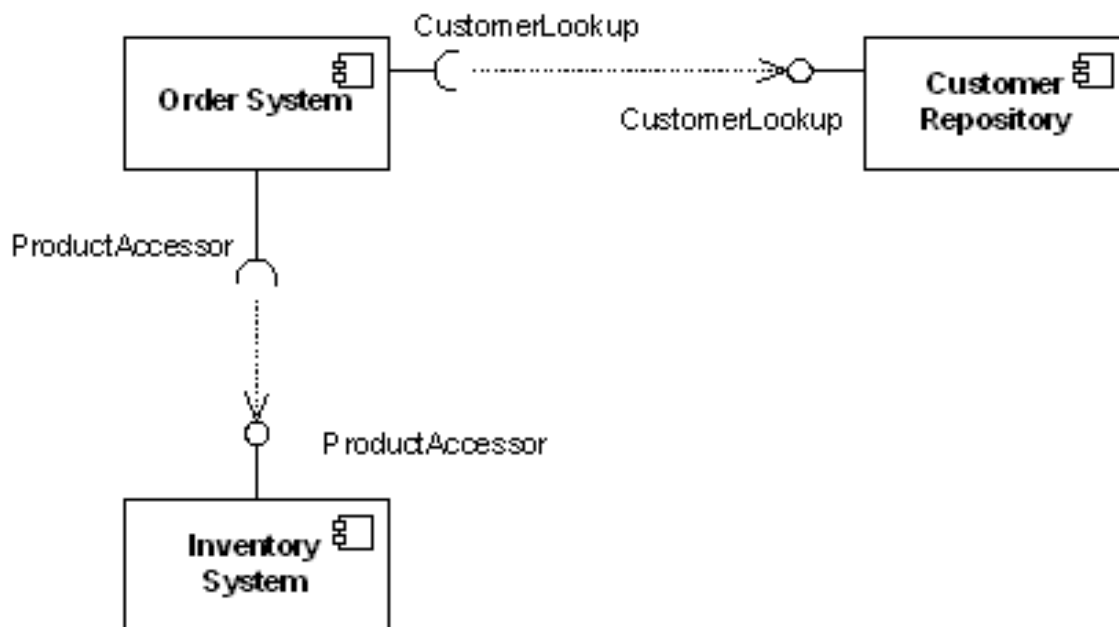
- 划分和组织一个组件的所有接口
    - 被封装的组件的对外窗口



# 组件图



- 组件与接口
  - 通过接口连接组件
    - 建模组件与组件之间的关系
    - 显示组件如何通过接口依赖于其他组件



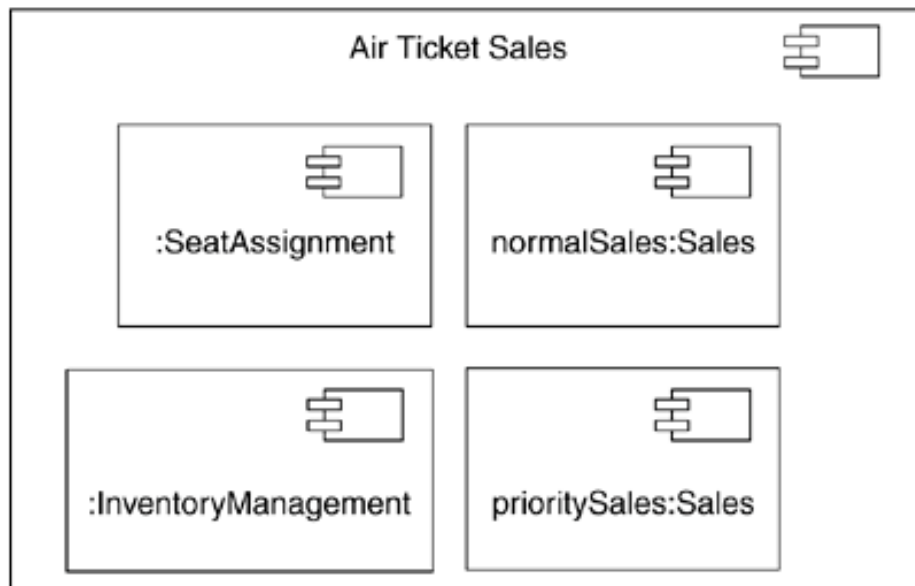
# 组件图



- 组件的内部结构

- 部件 (part)

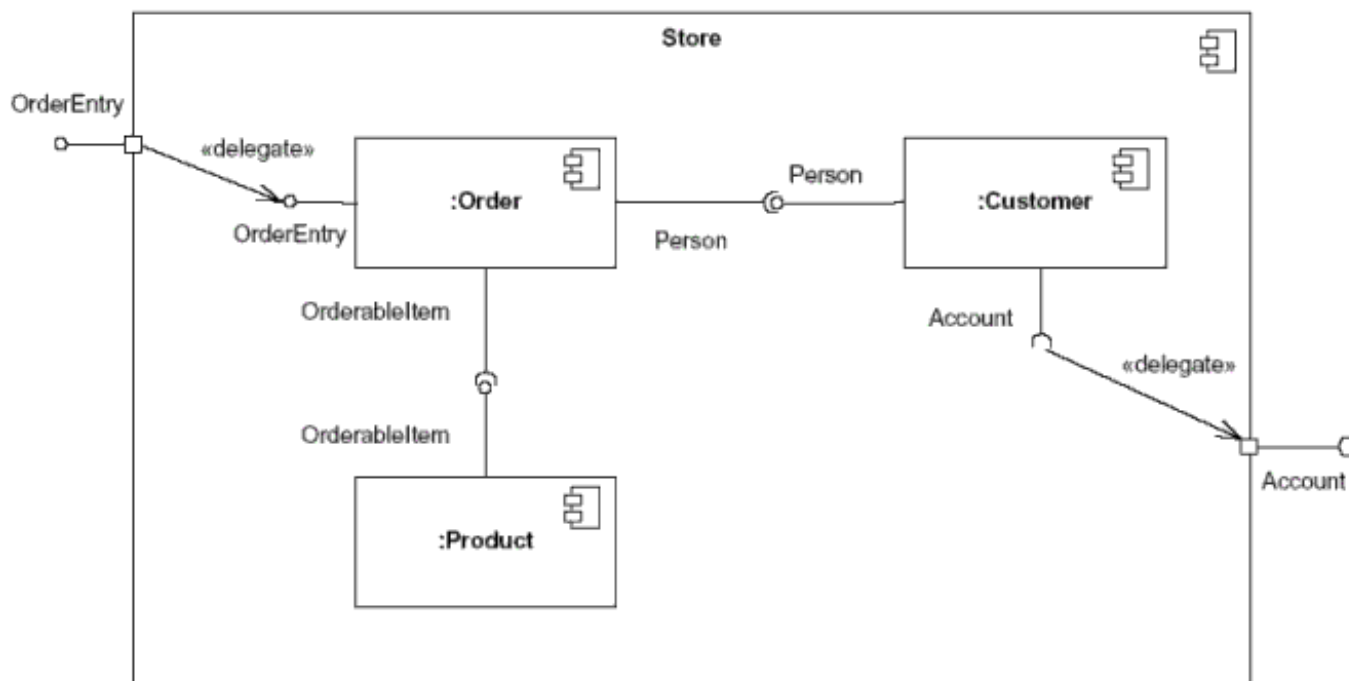
- 组件的实现单元
    - 名字和类型     +partName : TypeName
    - 多重性: 多个部件  
有相同的组件类型



# 组件图



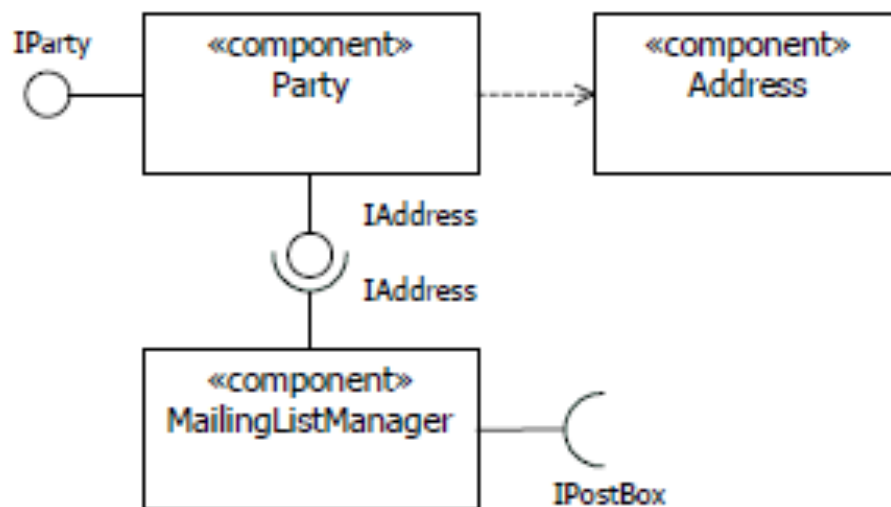
- 组件的内部结构
  - 用相互连接的一些部件表示组件的实现
  - 委托关系：父组件端口与部件的接口间



# 组件图



- 组件图
  - 展现组件及其间的组织与依赖关系
    - 捕获系统实现的物理结构
    - 体系结构规范的组成部分
  - 包括
    - 组件
    - 接口
    - 依赖关系
    - 组件包



# 组件图



- 组件间的关系

- 依赖关系

- 通过接口进行连接
    - 两个组件的类间存在泛化/依赖关系
    - 软件代码及其等价物之间

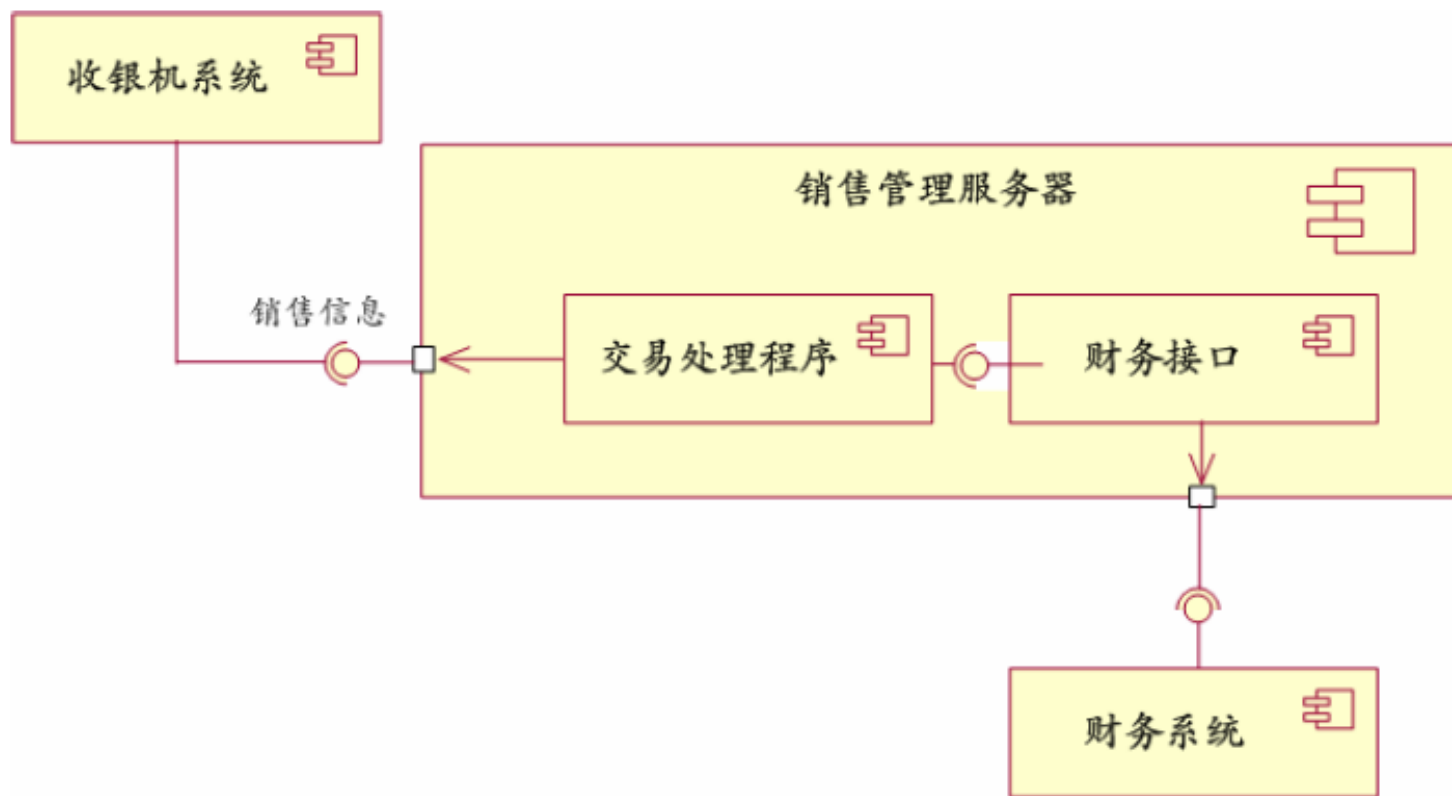


- 实现关系



# 组件图

- 组件图示例



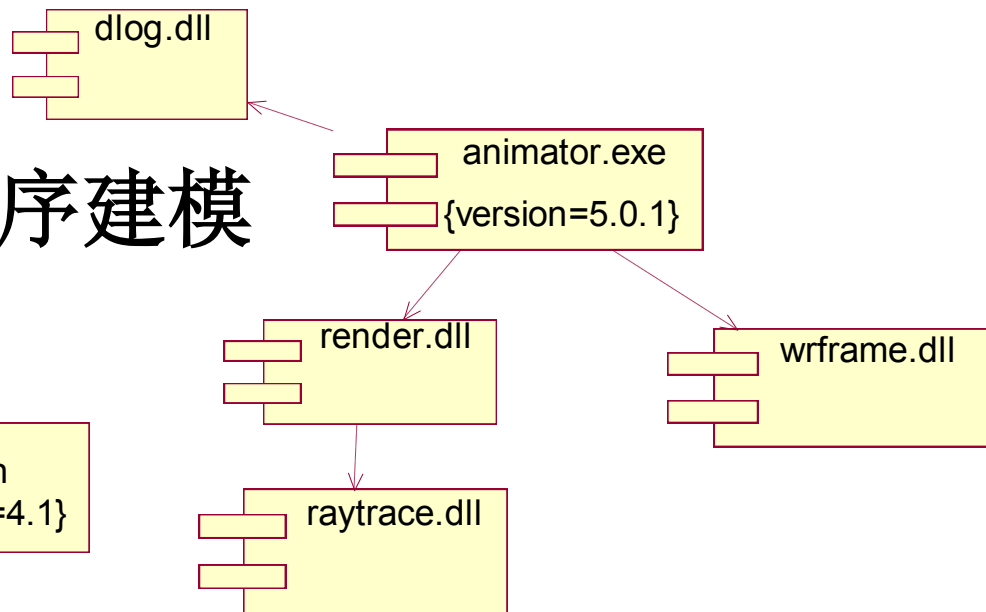
# 组件图

- 组件图的作用
  - 通过显示组件的接口来展示组件外部可见的行为
  - 通过显示组件的内部结构来展示组件的实现
  - 提供当前模型的物理视图
  - 显示软件代码及其等价物间的组织依赖关系
    - 对可执行程序的结构建模
    - 对源代码建模

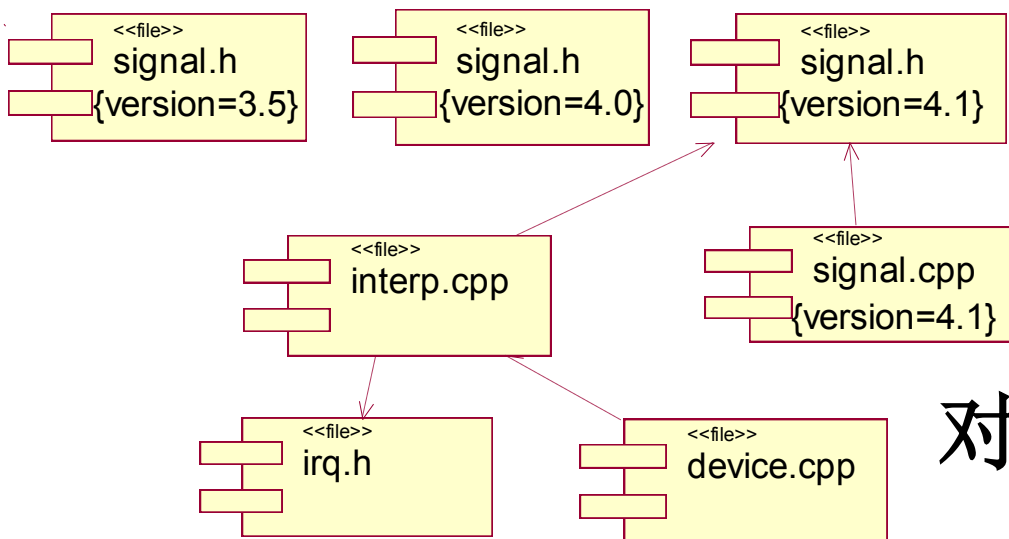
# 组件图

组件图可以对以下几个方面建模：

## 对可执行程序建模

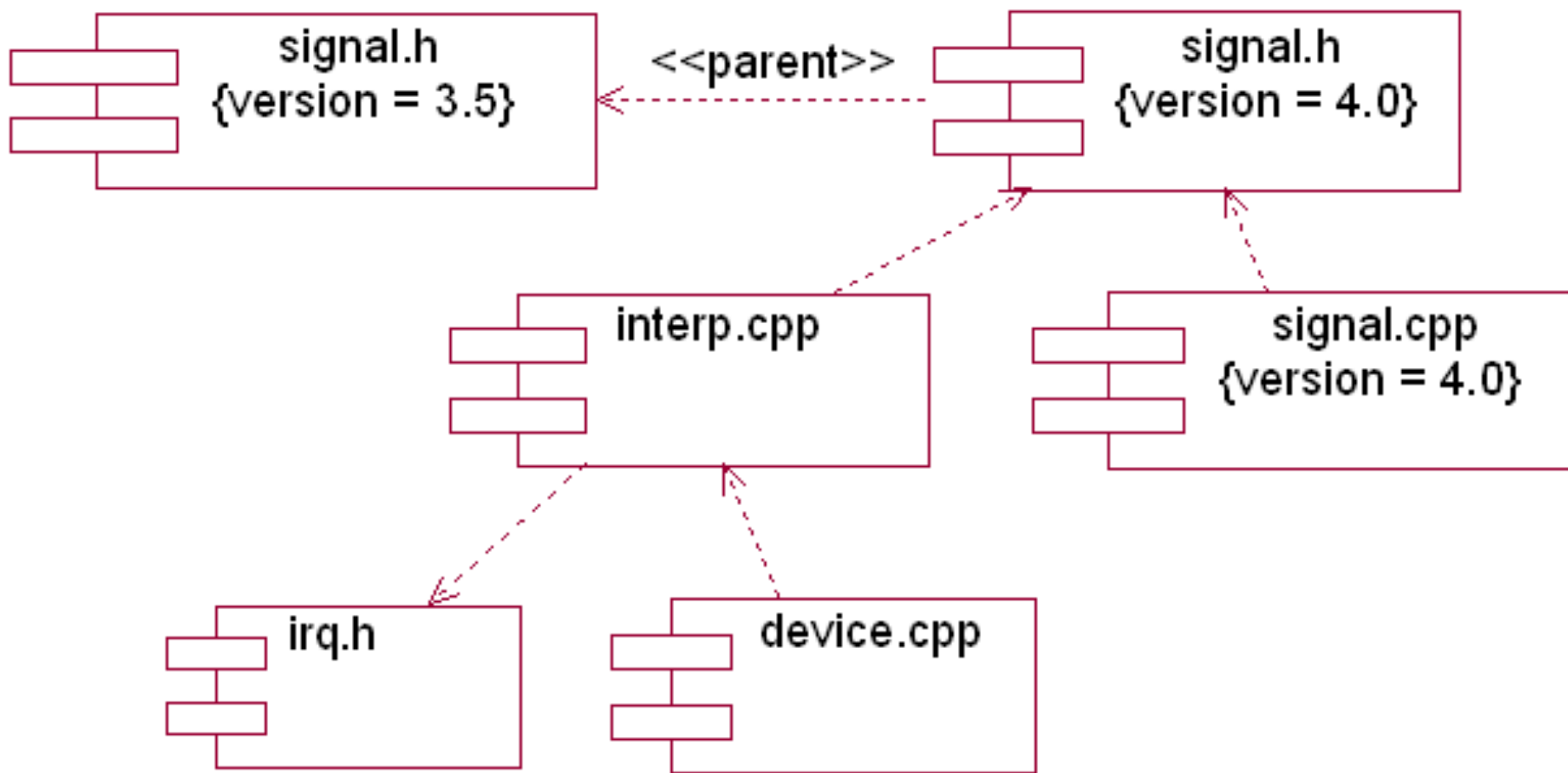


## 对源代码建模



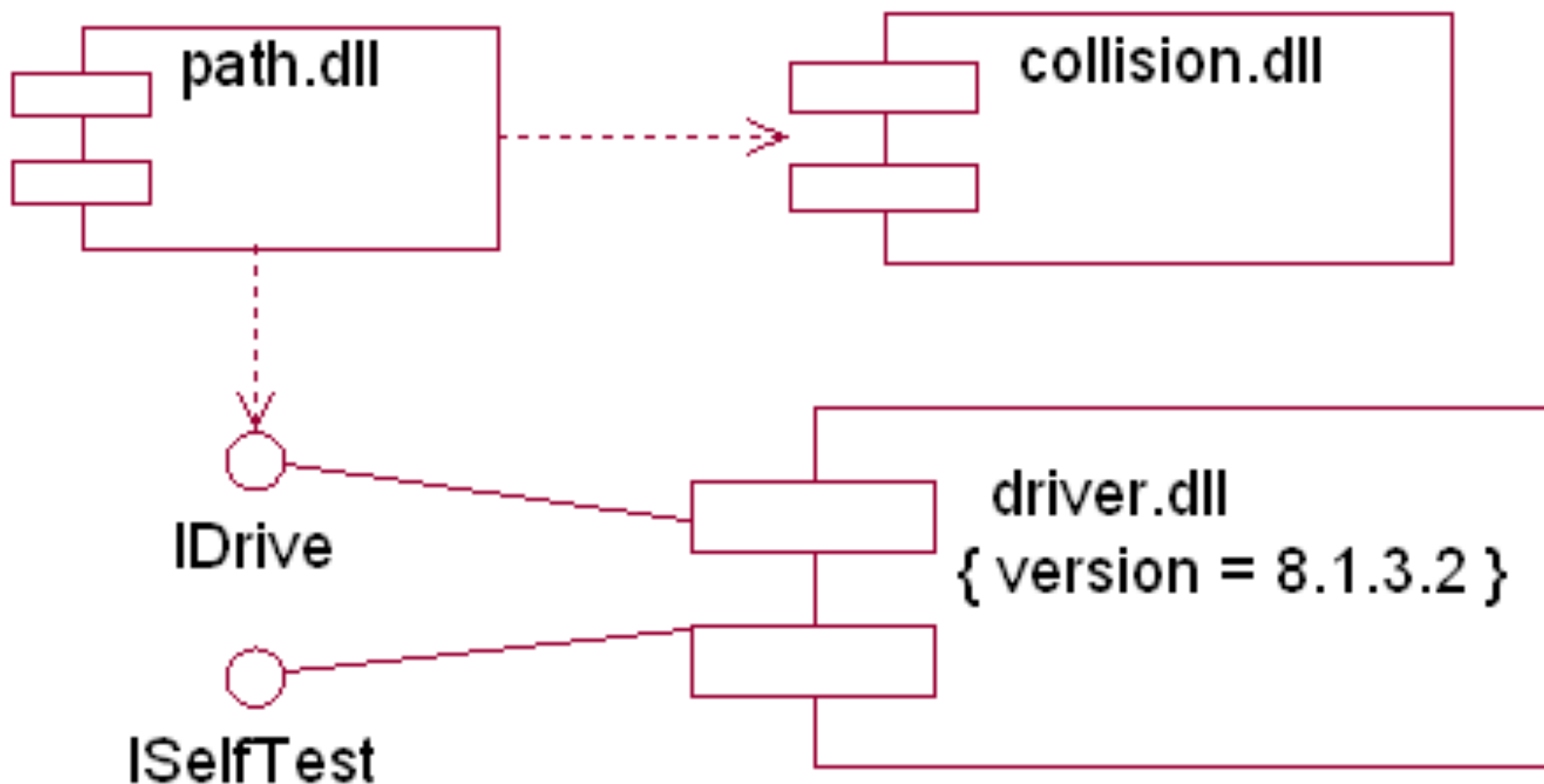
# 组件图的作用

- 组件图可以对以下几个方面建模：
- 对源代码文件之间的相互关系建模。



# 组件图的作用

- 组件图可以对以下几个方面建模：
- 对可执行文件之间的相互关系建模。



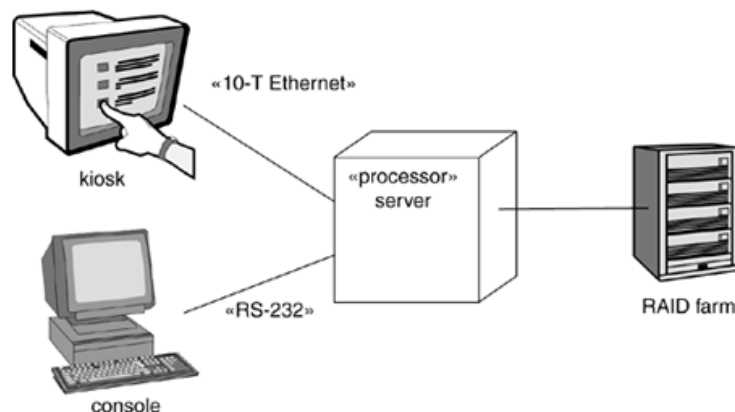
- 组件与类
  - 组件与类中很多方面类似
    - 类元/实例关系；包含元素的可视性
    - 内部结构表示
  - 实质性区别
    - 组件表示逻辑/物理抽象，类表示逻辑抽象
    - 类可以有属性和操作；组件通常只有操作，而且这些操作只能通过组件的接口才能使用。
    - 组件可用于配置图的节点中，类不行
  - 组件是类的载体，类依存于组件

# 部署图



- 部署图

- 对系统的静态部署视图建模
- 表示运行时进行处理的节点和节点上组件的配置
  - 捕获系统硬件的拓扑结构
  - 系统硬件：软件执行在其上的处理器和设备
- 是系统体系结构规格的一部分
  - 开发人员：体系结构设计师、系统工程师、网络工



# 部署图



- 部署图的组成

- 节点

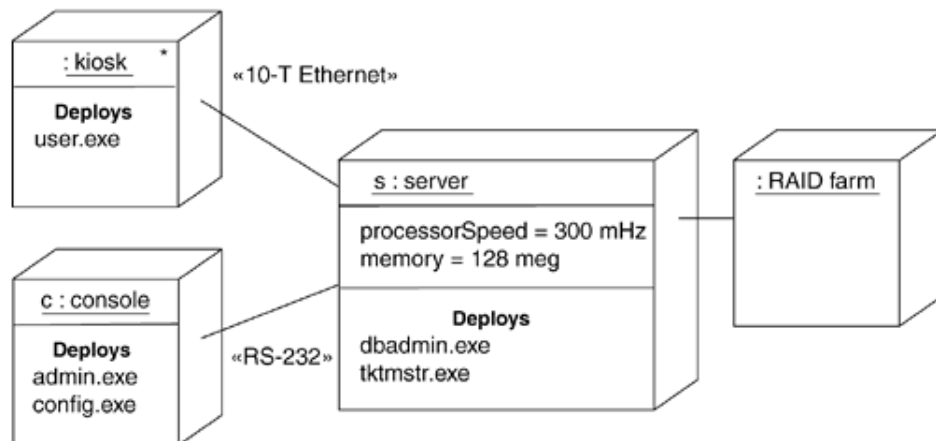
- 运行时存在的物理元素；可计算资源
    - 分类：处理器(process)和设备(device)

- 连接

- 节点间的某种硬件耦合关系
    - 包括：依赖和关联

- 还可包含

- 制品
    - 包或子系统





# 部署图



- 节点

- 占有内存；有处理能力
- 组件和制品的载体
- 处理器



塔式机



笔记本

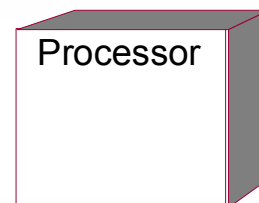


小型机



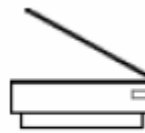
大型机

- 能够执行程序硬件部件
- PC，服务器等



- 设备

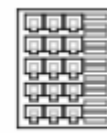
- 没有计算能力的硬件部件
- 打印机，终端显示设备等



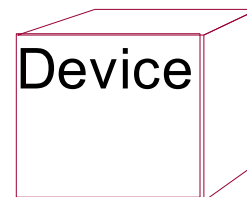
扫描仪



打印机



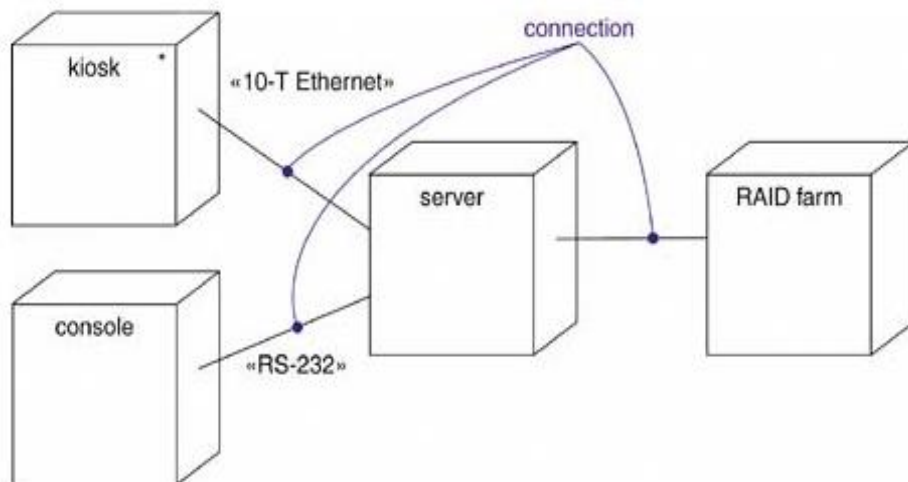
集线器



- 连接(connection)
  - 关联
    - 通常用于表达节点间信息传递的通信渠道
    - 直接/间接
    - 通常是双向的
  - 连接形式
    - 物理媒体: RS232线缆
    - 通信机制: GPRS
    - 软件协议: HTTP

# 部署图

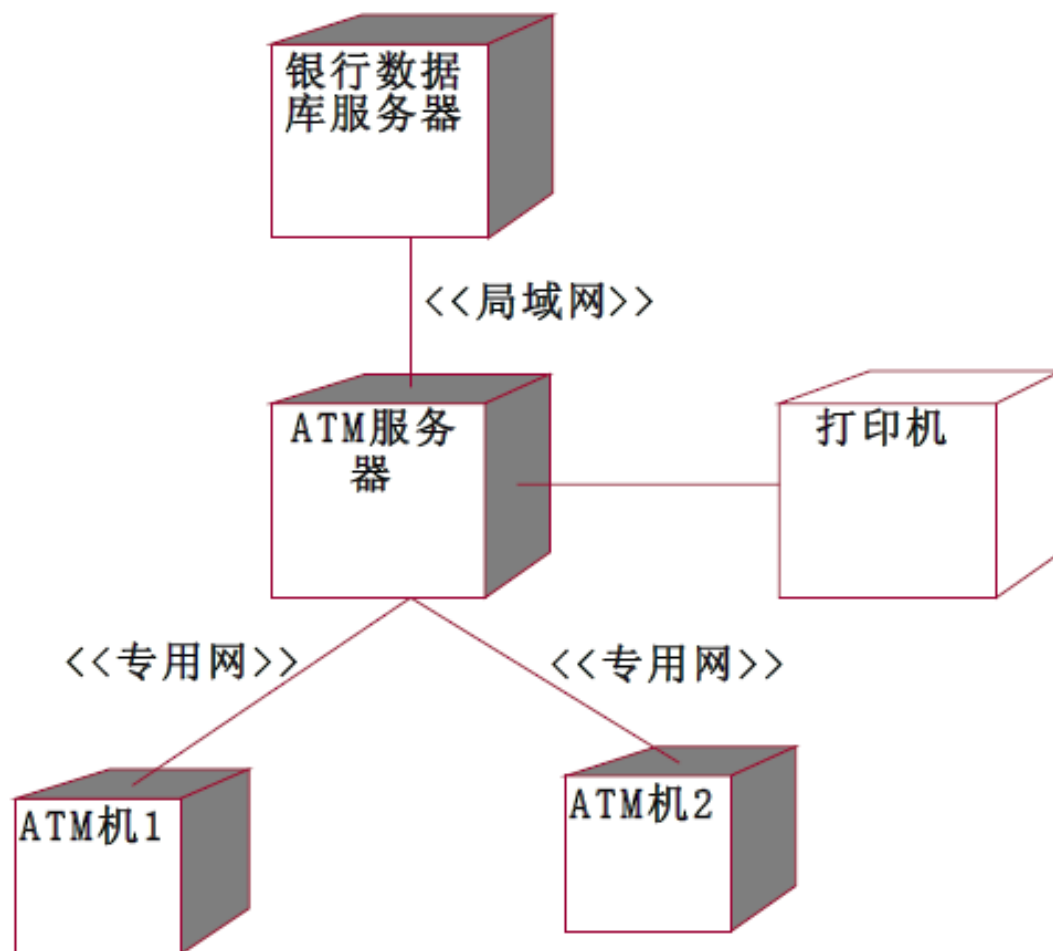
- 连接
  - 通过添加构造型对连接进行描述



- 通过“约束”对连接进行描述

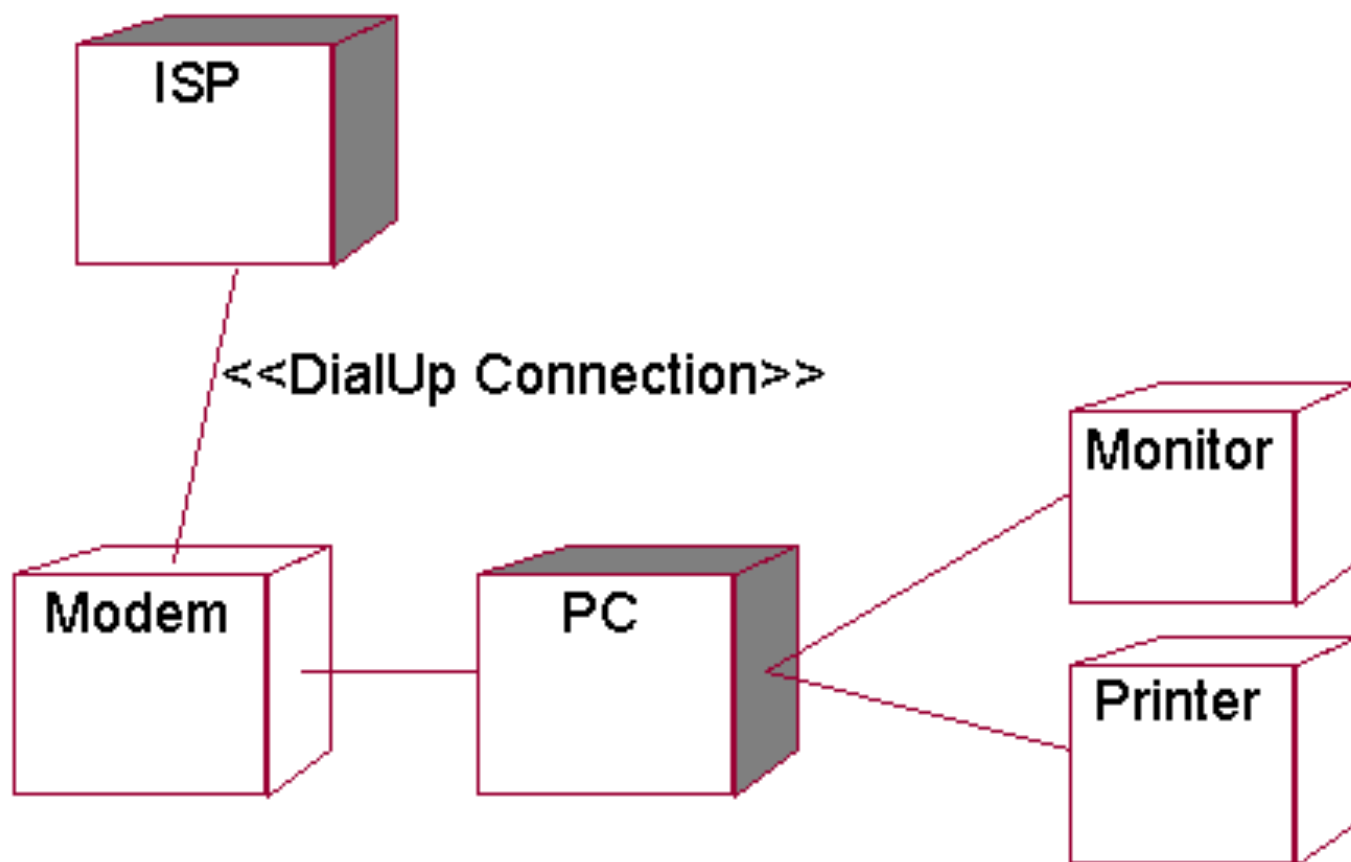
- 部署图的用法
  - 对嵌入式系统建模
    - 识别并区分所有处理器和设备
    - 建模节点间、制品和节点间的关系
  - 对C/S系统建模
    - 由构造型提供可视化提示
    - 建模节点间拓扑关系
  - 对分布式系统建模
    - 对通信设备建模
    - 使用包对节点进行逻辑分组

# 部署图的例子



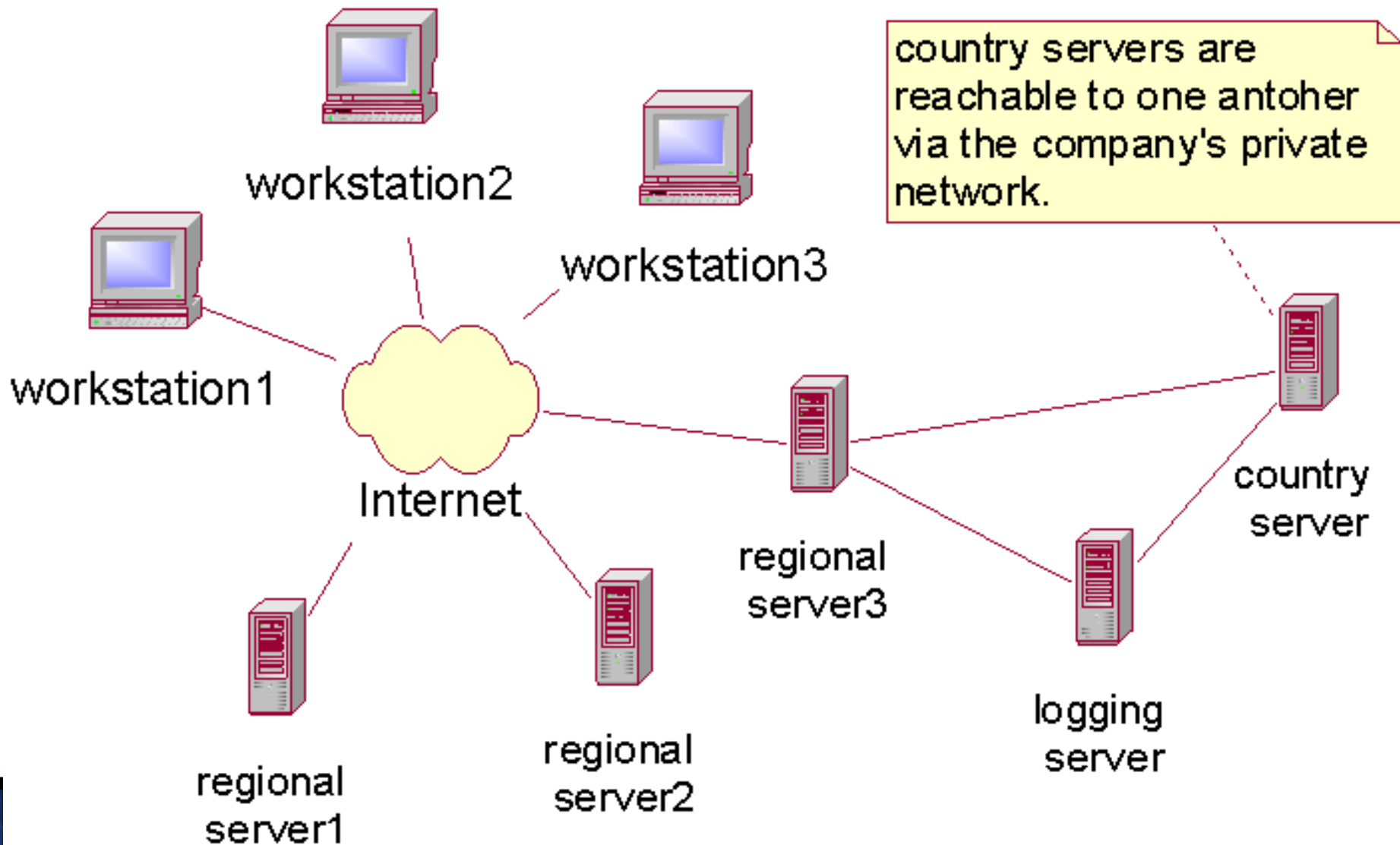
# 部署图的例子

- 例：PC和外设及ISP的连接的部署图的例子。



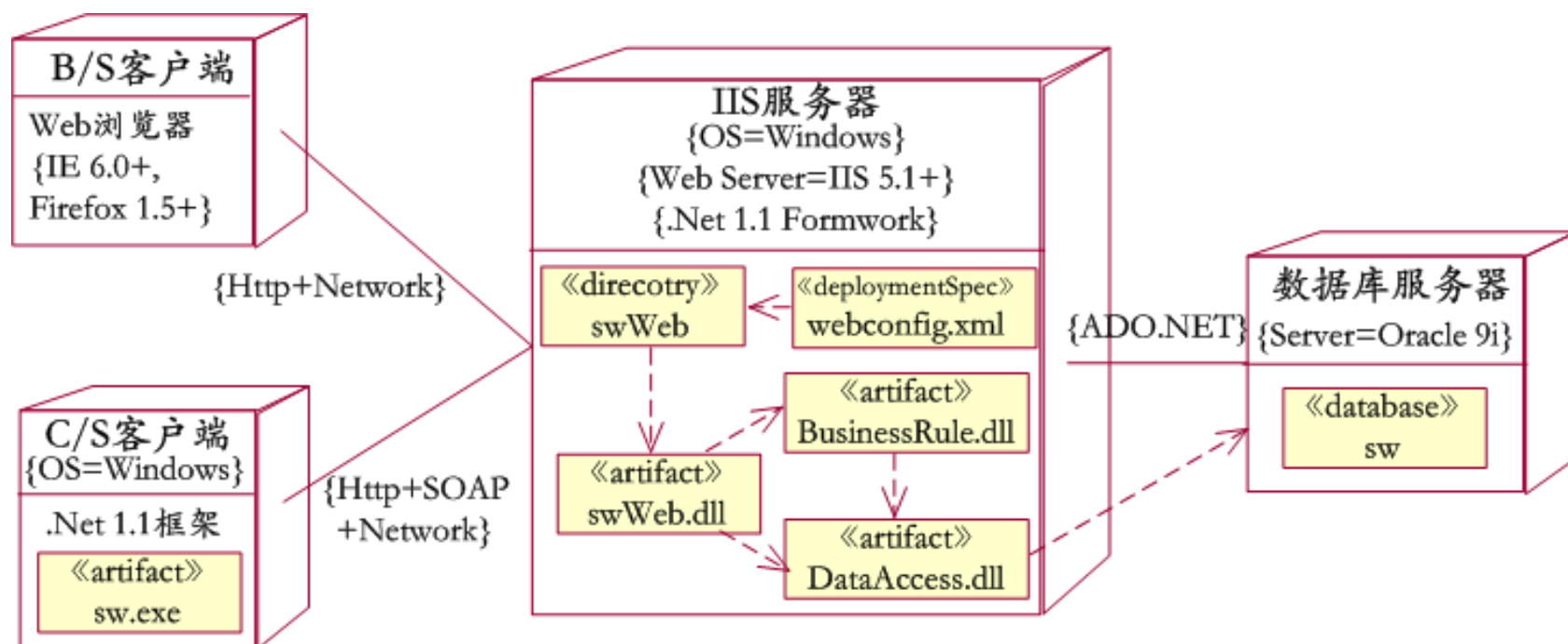
# 部署图的例子

- 例：对fully分布式系统建模的例子。



# 部署图

- 阅读部署图
  - 识别节点
  - 识别连接





- 阅读部署图

- 为了更好地表示两个节点之间的关系，可以通过“约束”来对连接进行描述

源节点	目标节点	约束	含义
B/S客户端	IIS服务器	{HTTP+Network}	网络连接，使用HTTP协议
C/S客户端	IIS服务器	{HTTP+SOAP+Network}	网络连接，通过Web Service访问服务
IIS服务器	数据库服务器	{ADO.NET}	.NET提供的数据库访问解决方案

- 节点和组件
  - 有许多相同之处
    - 都可以被嵌套，可以有实例
    - 可以参与交互
  - 本质区别
    - 组件是参与系统执行的事物，表示逻辑元素的物理打包
    - 节点是执行构件的事物，表示构件的物理部署
  - 节点是组件的载体

# 小结



- 重点
  - 类图
  - 接口
  - 类图中的关系
  - 类图的阅读与绘制
  - 组件图和节点图的阅读