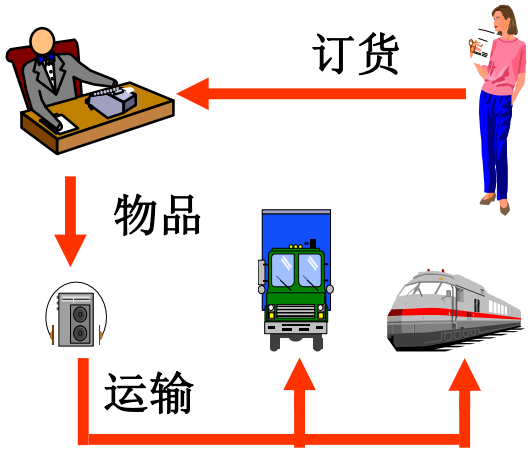
UML

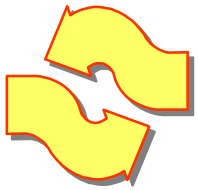
可视化建模

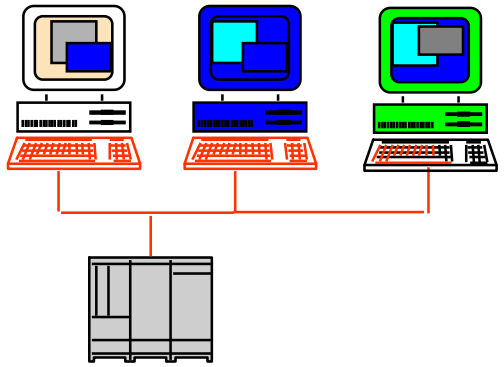


商务过程

建模是为了捕捉、描述系统的核心.

James Rumbaugh





计算机系统

可视化建模就是用标准的图示化方法来进行建模工作

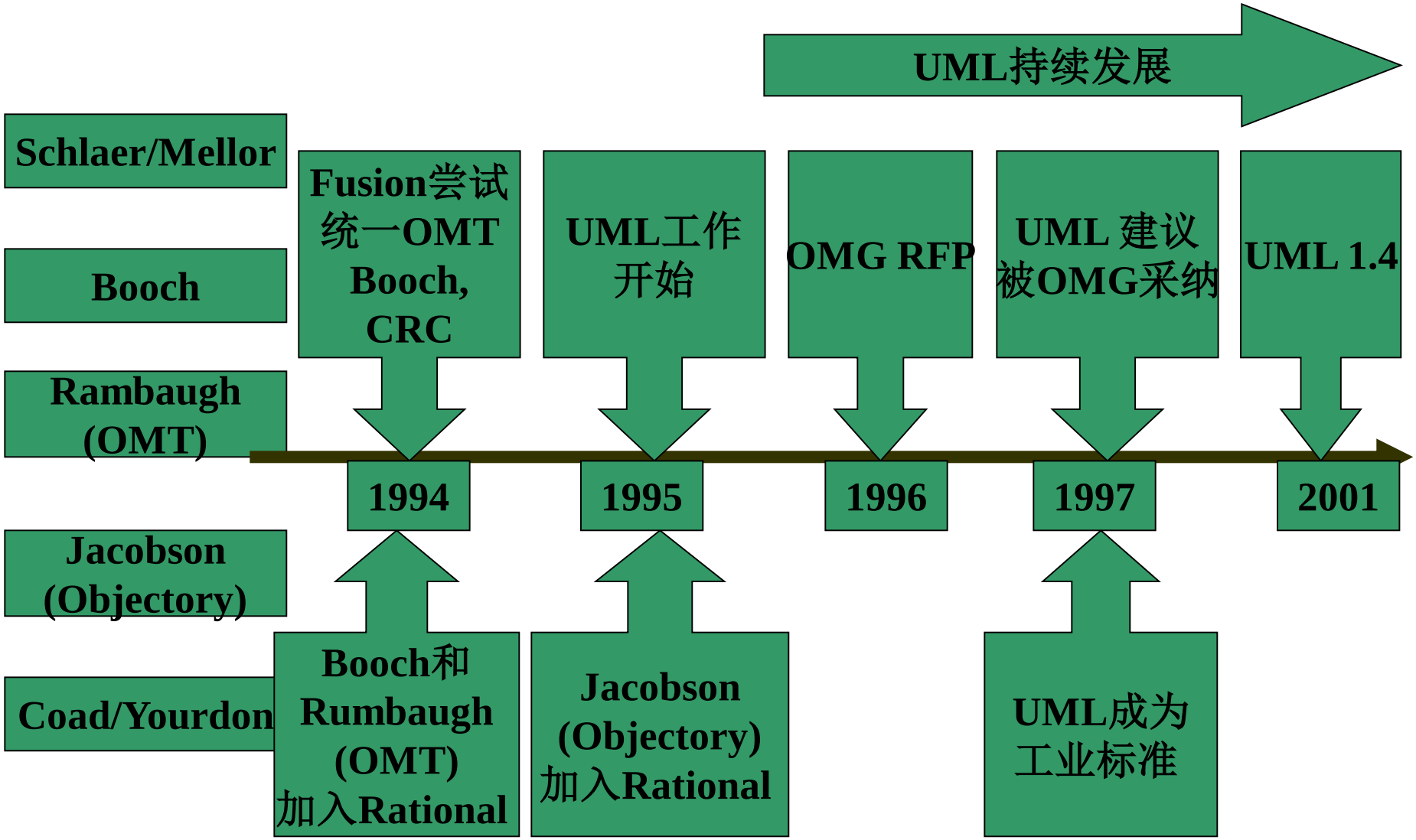
UML

- Unified Modeling Language 统一建模语言
- UML是一个适合于面向对象技术的、开放的、可扩展的、可视化建模语言工业标准
- UML由OMG(Object Management Group)所批准

UML产生与发展

面向对象的分析与设计(OOA＆D)方法在80年代末至90年代中出现了一个高潮， UML是这个高潮的产物。它统一了Booch、Rumbaugh和Jacobson的表示方法，并对其作了进一步的发展，并最终统一为大众所接受的标准建模语言。

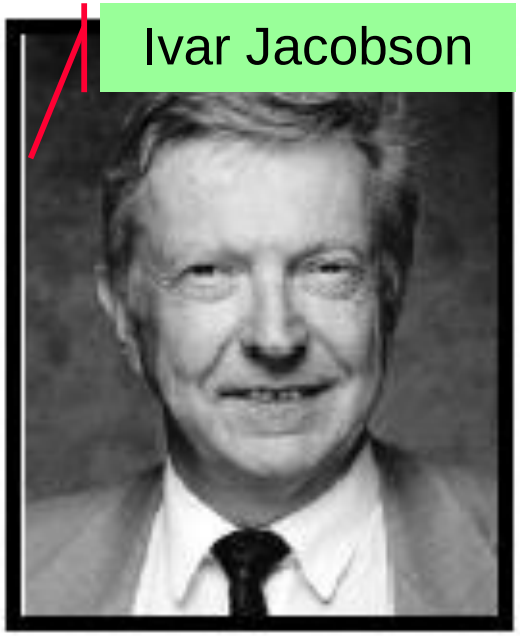
UML的进化过程


UML持续发展
Schlaer/Mellor
Booch
Rambaugh (OMT)
Jacobson (Objectory)
Coad/Yourdon
Fusion尝试统一OMT Booch, CRC
UML工作开始
OMG RFP
UML 建议被OMG采纳
UML 1.4
1994
1995
1996
1997
2001
Booch和Rumbaugh (OMT)加入Rational
Jacobson (Objectory)加入Rational
UML成为工业标准

技术发展背景— Rational三剑客

Jim Rumbaugh

Ivar Jacobson



Grady Booch

UML的目标

- 最重要目标：UML是所有建模人员可以使用的通用建模语言。它包含主流建模方法的概念，从而可以替代现有的软件分析和设计方法，比如：OMT,Booch,OOSE等。
- UML不是完整的开发方法，它不包括逐步的开发流程，但它提供所有必要的概念，具备足够的表达能力。
- UML的另一个目标是：能尽量简洁地表达系统的模型。

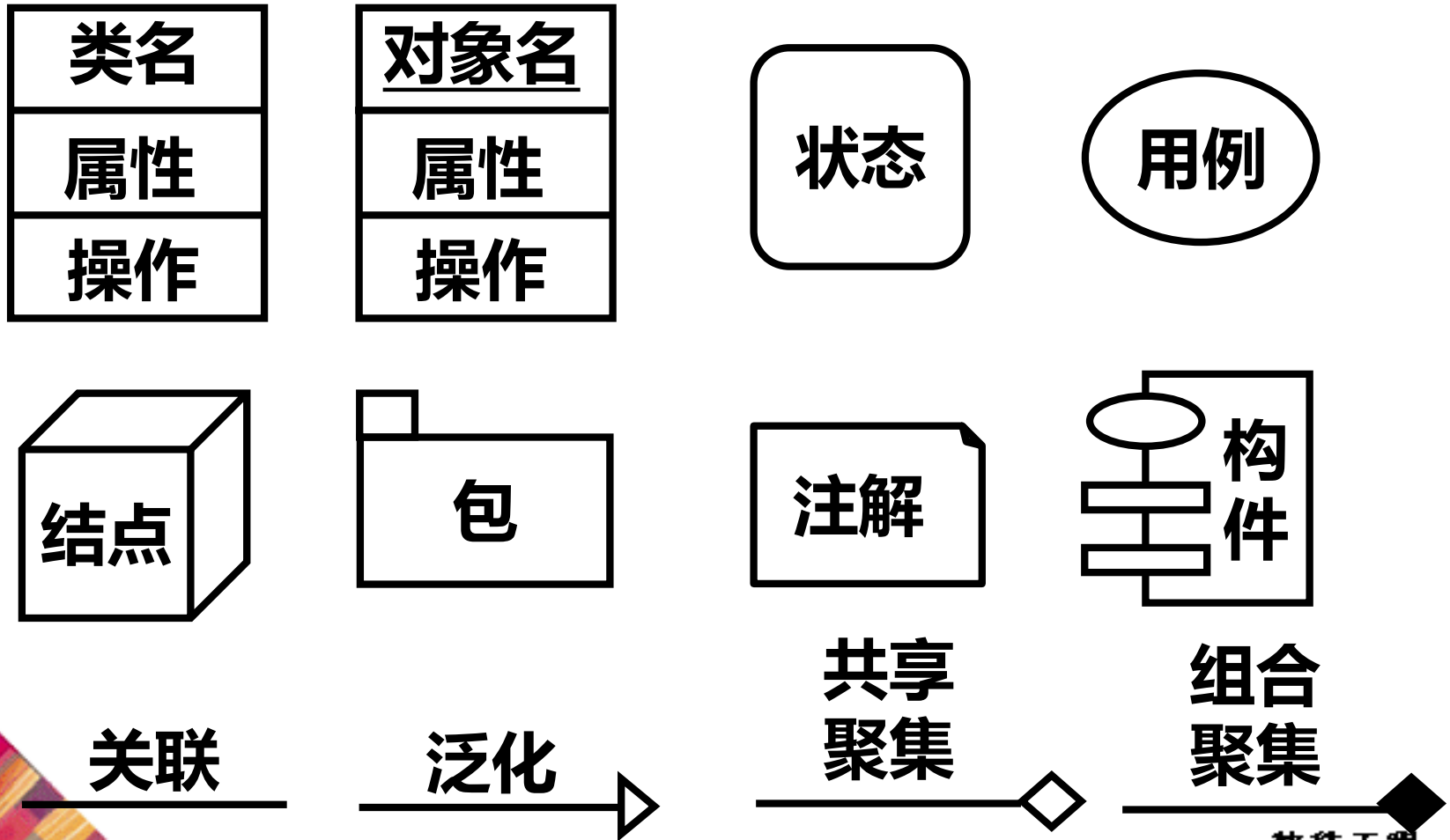
- UML语言：用一组由语法规则、语义规则和语用规则控制的建模符号来表示分析模型。
  - 用户模型视图：从用户的角度来表示系统，即从用户的使用的角度来描述使用场景；
  - 结构模型视图：从系统内部观察数据和功能，即对静态结构（类、对象、关系）建模；
  - 行为模型视图：描述各种结构元素间的交互或协作；
  - 实现模型视图：描述系统的结构和行为；
  - 环境模型视图：表示系统环境的结构和行为方面的属性；

UML各系统视图的图示及适用对象

系统视图	使用图示	适用对象
用户视图	使用实例图，活动图	用户，设计者，实现者，测试者
结构视图	类和对象图，状态图，时序图，协作图，活动图	设计者，实现者
行为视图	状态图，时序图，协作图，协作图，构件图，配置图	实现者，组装者
实现视图	构件图	实现者
环境视图	配置图	实现者，组装者，测试者

- 模型元素

模型中的实体以及实体间相互连接的关系



- **图（diagram）**

  用例图、类图、对象图、状态图、时序图（顺序图）、协作图、活动图、构件图、部署图

- **视图（view）**

  用例视图、逻辑视图、构件视图、并发视图、部署视图

① 用例（use-case）图：展示各类外部行为者与系统所提供的用例之间的连接。

② 类（class）图：展示系统中类的静态结构，即类与类之间的相互联系。

③ 对象（object）图：显示类图的一些实例。

④ 状态（state）图：状态图通常是对类描述的补充，它说明该类的对象所有可能的状态以及哪些事件将导致状态的改变。

⑤ 时序（sequence）图：它主要是用来显示对象之间发送消息的顺序，以及对象之间的交互。

⑥ 协作（collaboration）图：展示消息的交互，对象以及它们之间的关系。

⑦ 活动（activity）图：展示连续的活动流。活动图通常用来描述完成一个操作所需要的活动。

⑧ 构件（component）图：展示以代码构件为单位的代码的物理结构。

⑨ 部署（deployment）图：展示系统中硬件和软件的物理结构。

用例建模

1992年由Jacobson提出了Use case 的概念及可视化的表示方法—Use case图。用例驱动的系统分析与设计方法已成为面向对象的系统分析与设计方法的主流。

用例模型(Use case model)

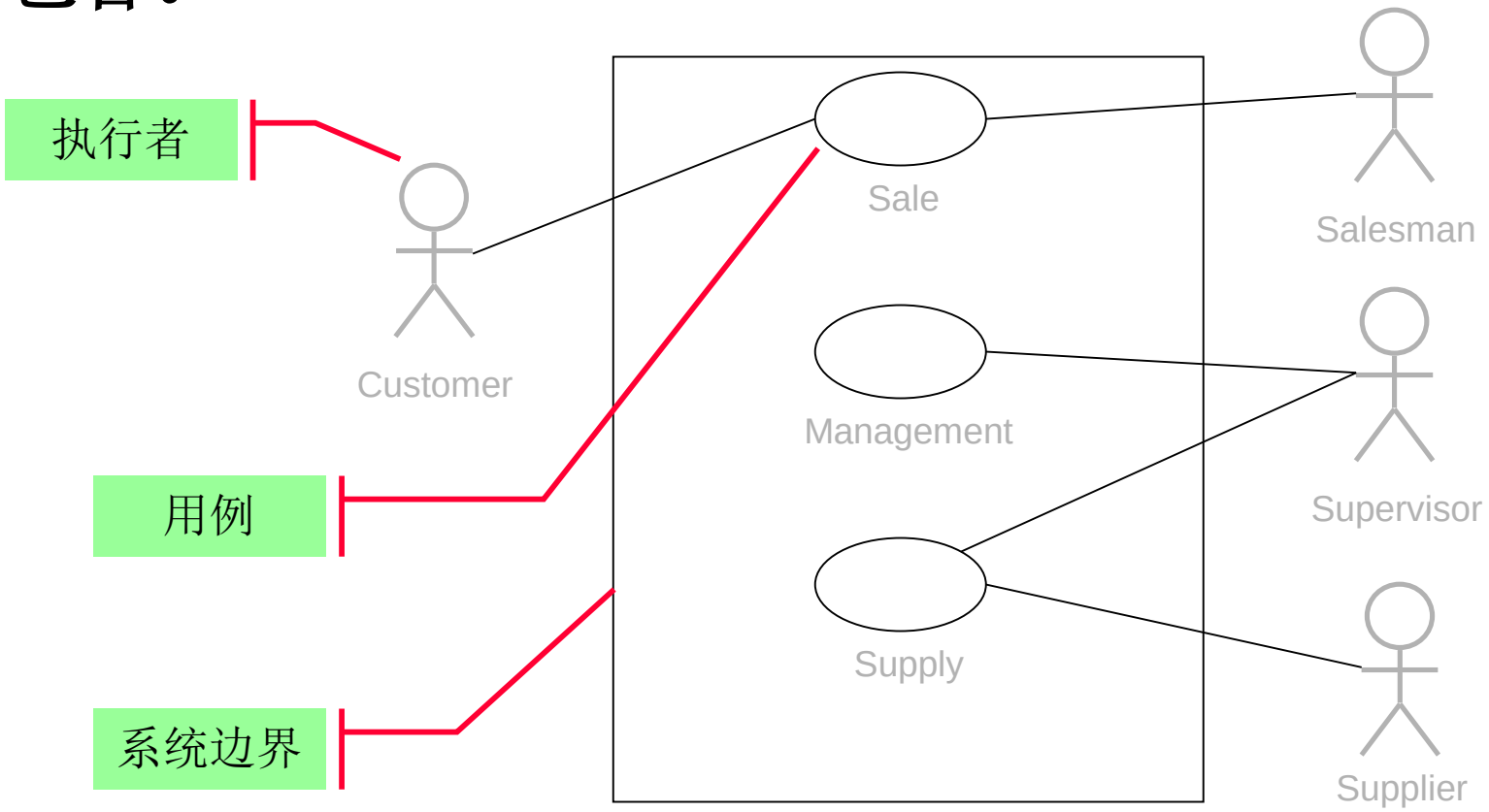
用例模型由若干个用例图构成，用例图中主要描述执行者和用例之间的关系。在UML中,构成用例图的主要元素是用例和执行者及其它们之间的联系。

创建用例模型的工作包括：

定义系统、确定执行者和用例、描述用例、定义用例间的关系、确认模型。

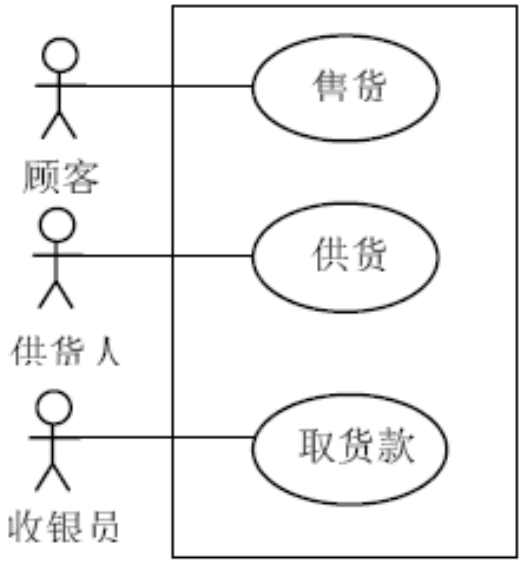
用例图

- 用例可以参与多种关系：关联、扩展、泛化和包含。



一、执行者(Actor)

执行者是指用户在系统中所扮演的角色。执行者在用例图中是用类似人的图形来表示，但执行者可以是人，也可以是一个外界系统。



二、用例(use case)

从本质上讲,一个用例是用户与计算机之间的一次典型交互作用。在UML中,用例被定义成系统执行的一系列动作（功能）。

用例有以下特点:

用例捕获某些用户可见的需求,实现一个具体的用户目标。

用例由执行者激活,并将结果值反馈给执行者。

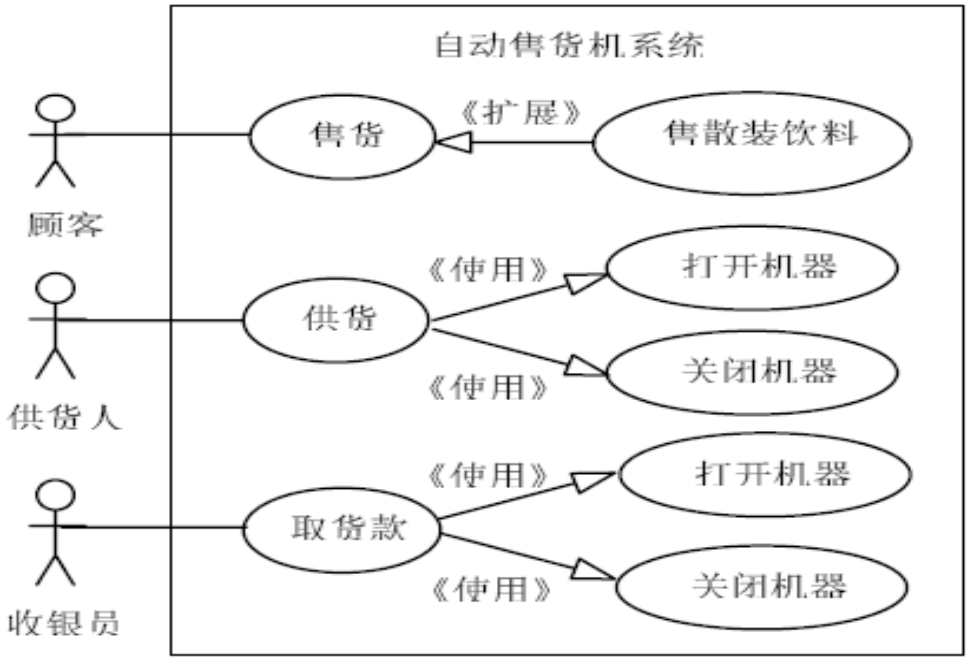
用例必须具有功能上的完整描述。

用例图

《Use》用例图描述了系统的功能需求，它是从执行者的角度来理解系统，由“执行者”、“用例”和“用例之间的关系”3类模型元素构成。

表示一个用例使用另一个用例。(《include》)

《Extend》通过向被扩展的用例添加动作来扩展用例。



识别用例

执行者使用这个系统达到什么目标

语法测试：【执行者】使用系统来【用例】

识别用例

这些用例形式上对了，内容呢？

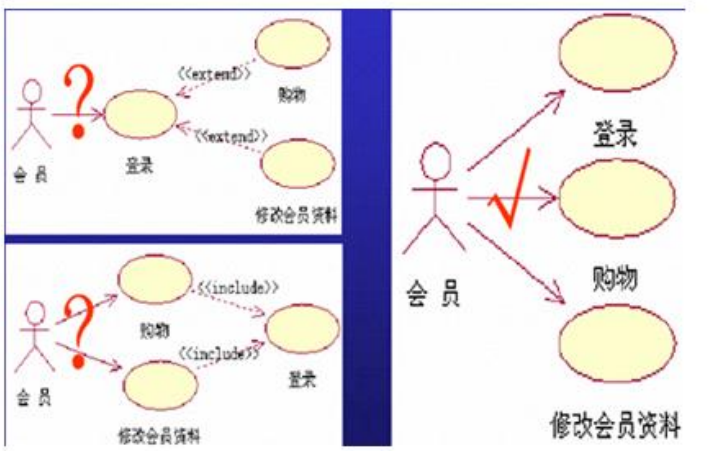
	用例	用例	用例	用例	用例	用例	用例	用例	用例	用例
目标		●	●	●			●		●	
目标		●								
目标					●	●				
目标			●							
目标				●						

是不是所有愿景目标都有用例？

是不是所有用例都有愿景目标？

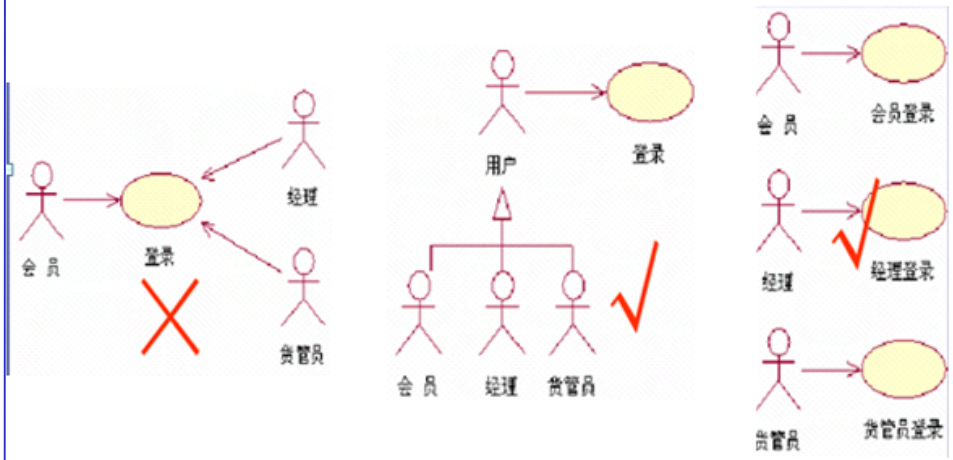
识别用例

——讨论（1）：登录怎么处理



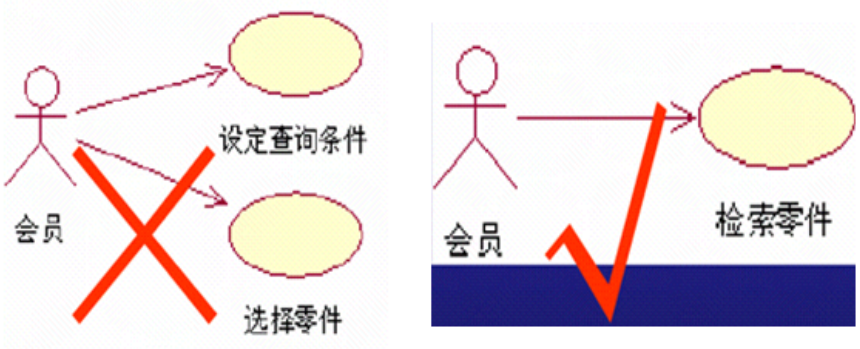
识别用例

——讨论（2）：几个登录？



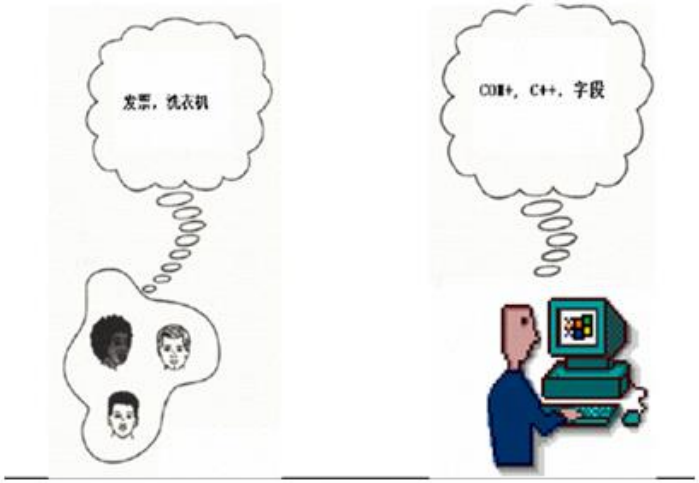
识别用例

—有意义的目标



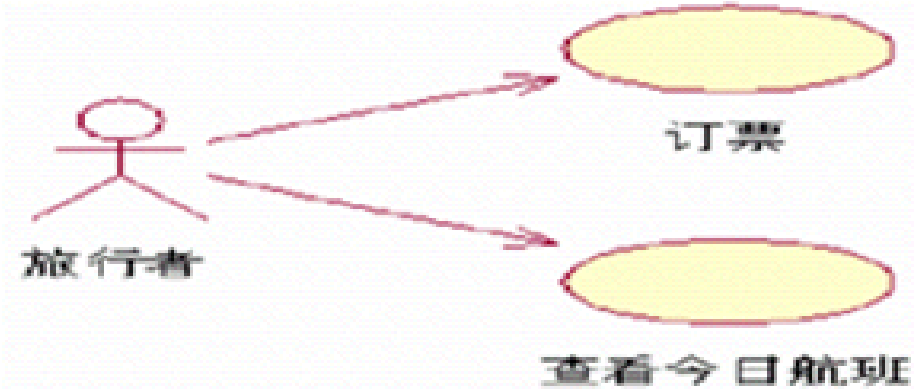
识别用例

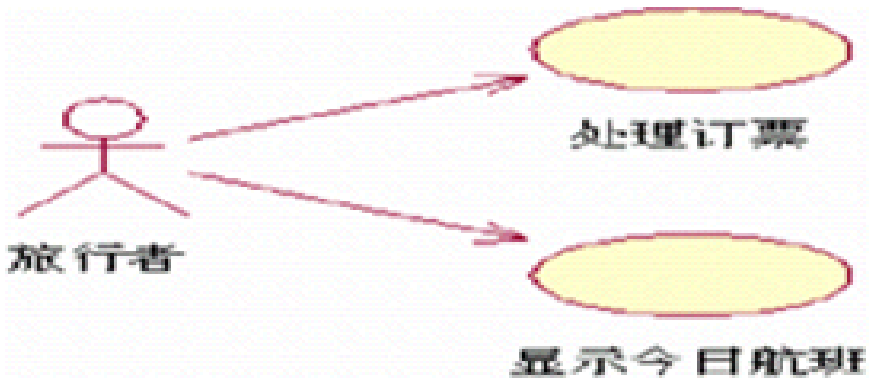
—业务语言而非技术语言



识别用例

—用户观点而非系统观点

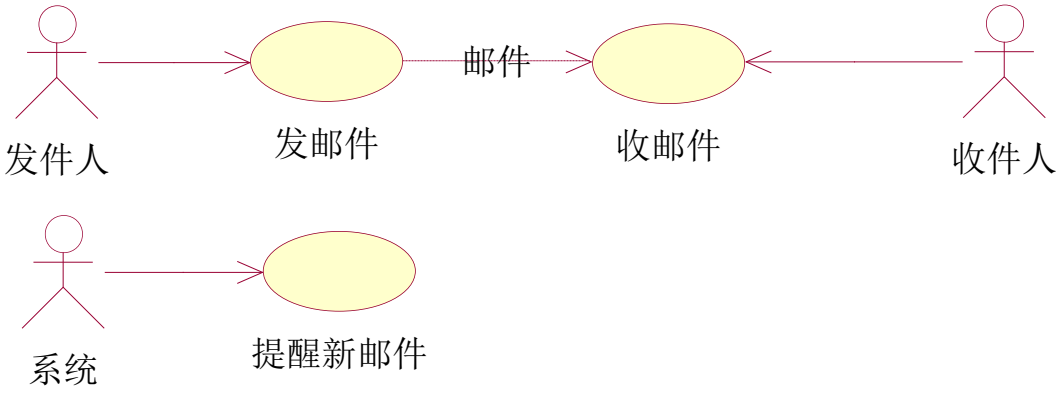


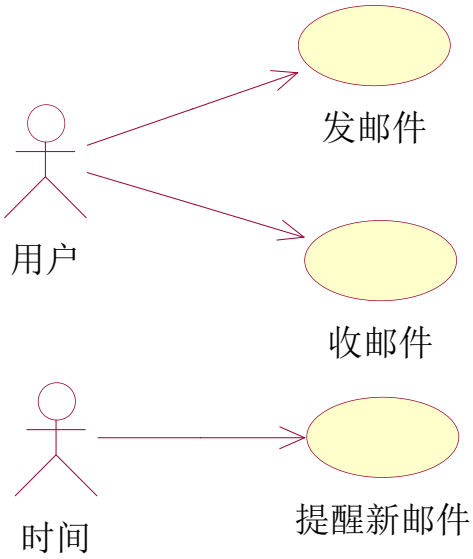


思考：识别用例-1

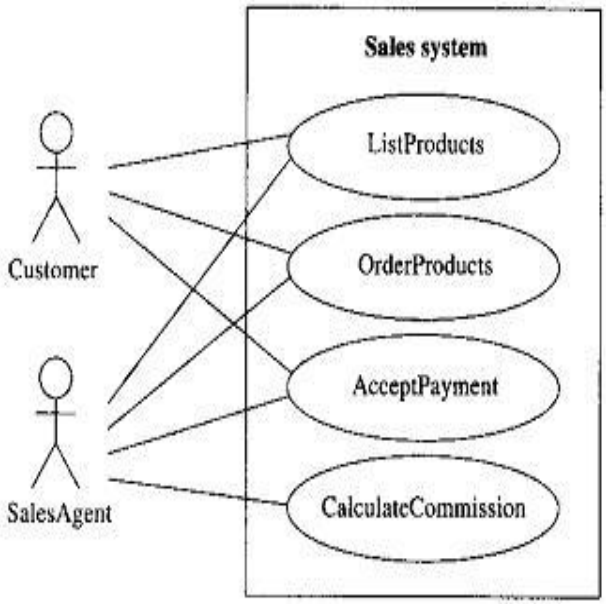
- Email客户端（如：outlook express），A在北京发邮件给上海的B，系统提醒B你有“新邮件”，B收邮件

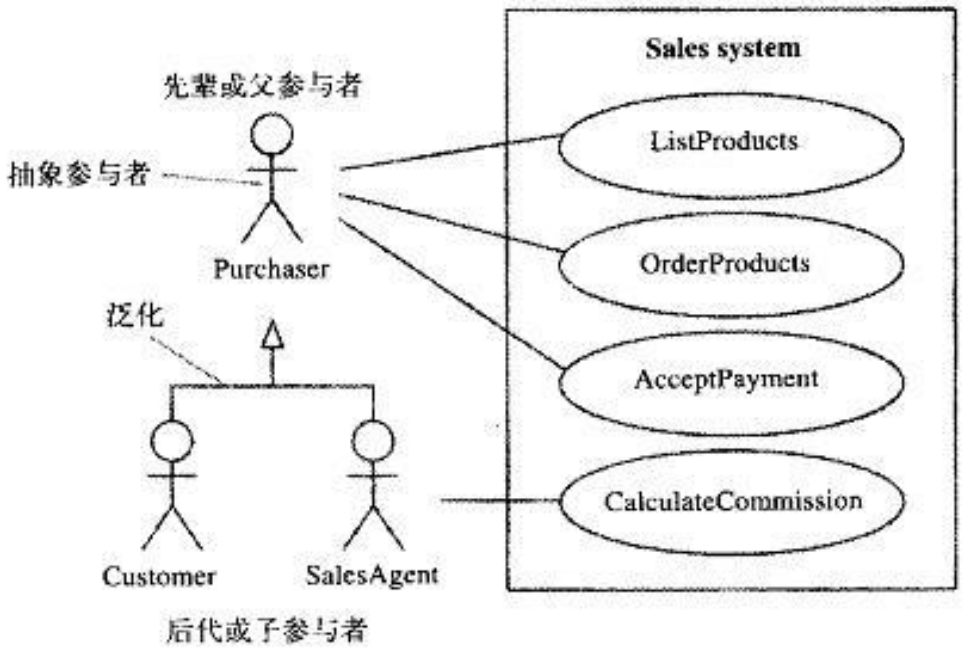
思考：识别用例-2





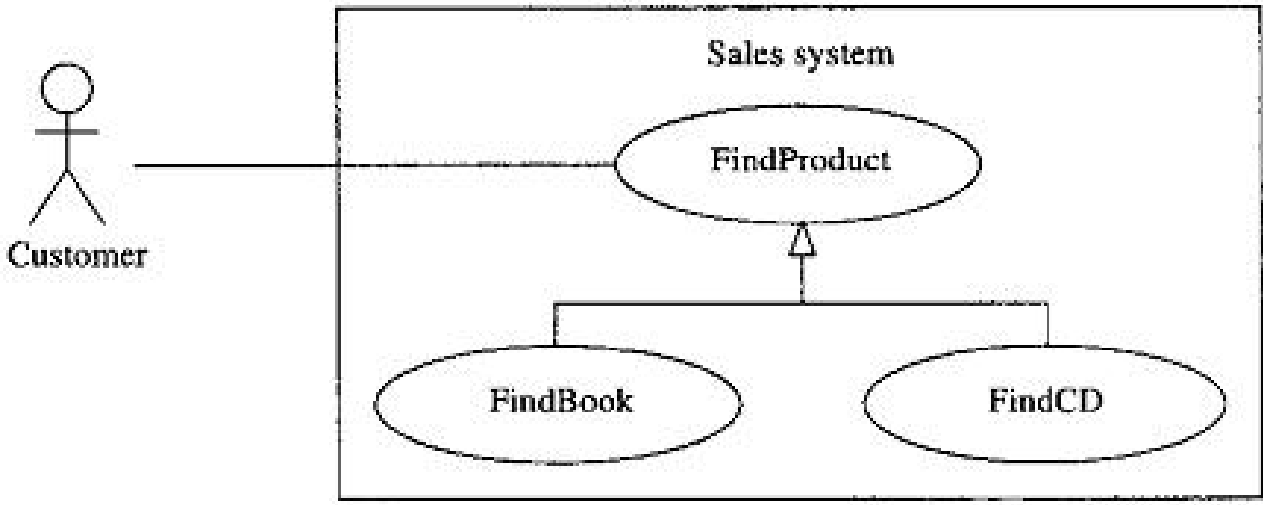
参与者泛化





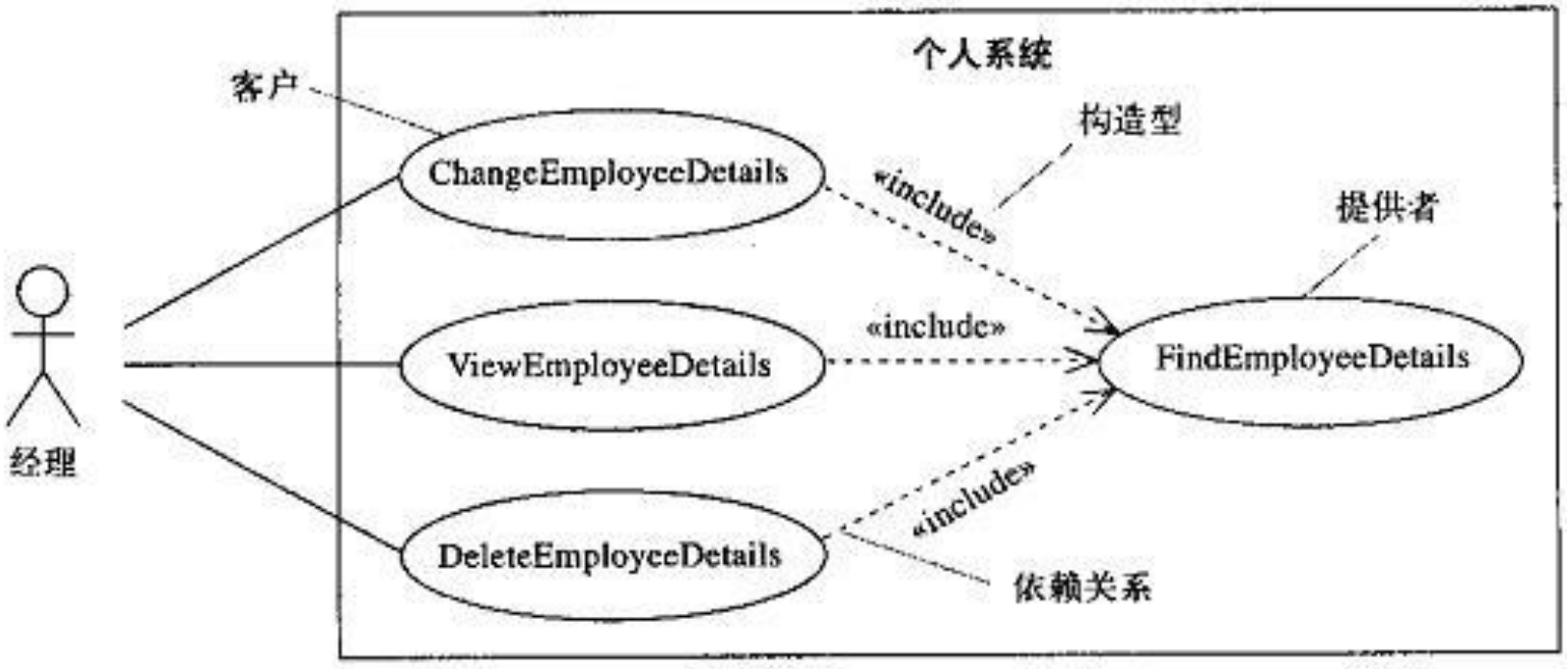
- 使用参与者泛化分解这个共性行为

用例泛化



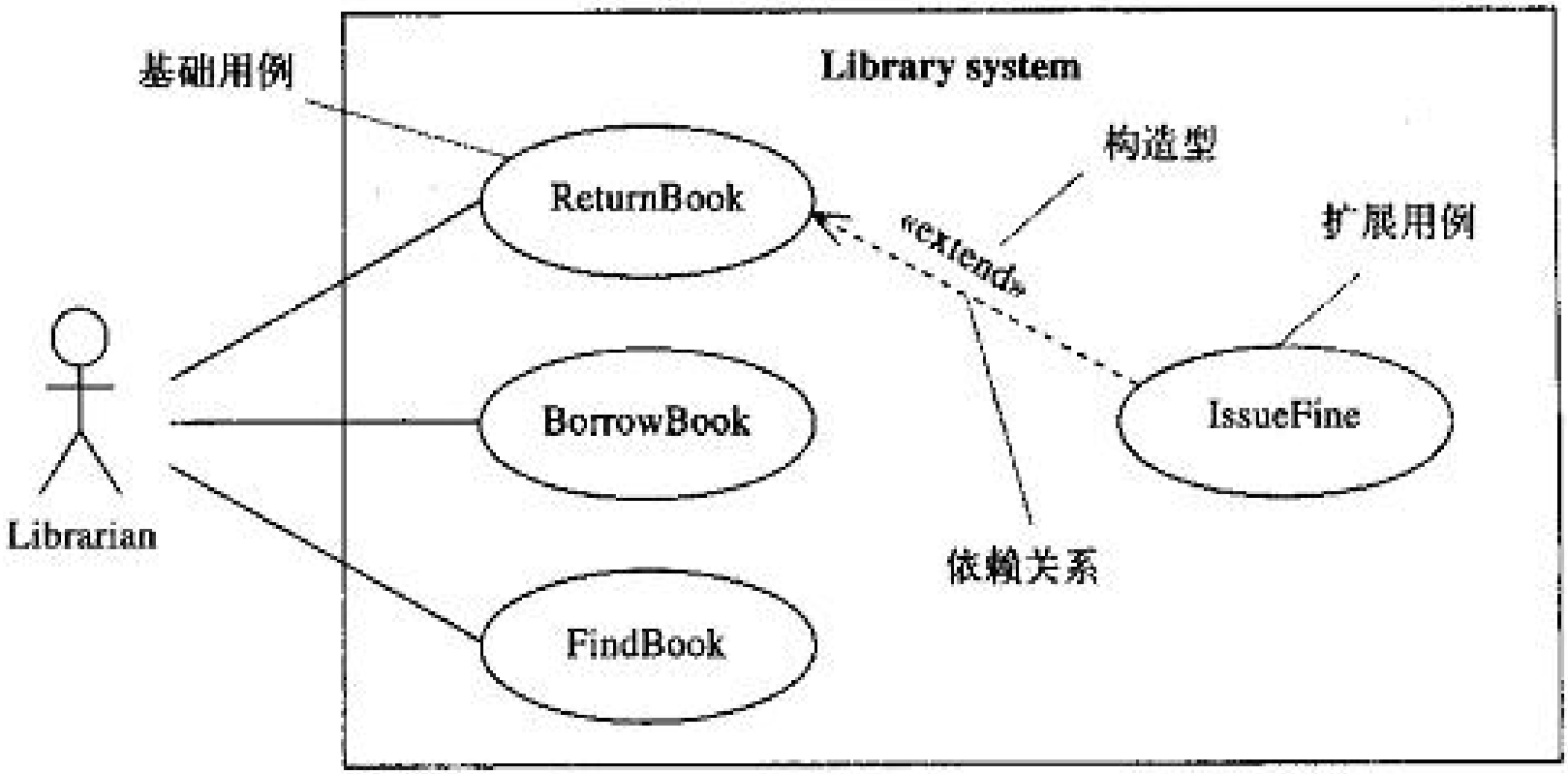
- 继承特征
- 添加新的特征
- 覆盖(改变)已继承的特征

《include》



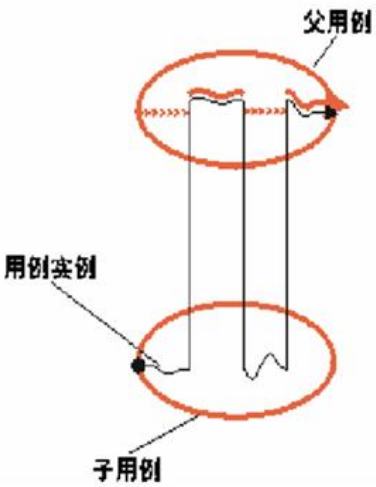
- 客户用例执行到包含点，然后执行传递给提供者用例。当提供者用例完成时，控制再次返回客户用例。

«extend»


基础用例
Library system
构造型
ReturnBook
«extend»
扩展用例
BorrowBook
IssueFine
Librarian
依赖关系
FindBook

识别用例的关系

——泛化

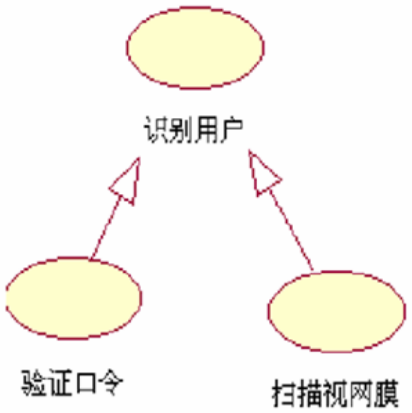


子用例继承父用例的一切：

- 关系
- 前置条件
- 后置条件
- 路径
- 步骤

识别用例的关系

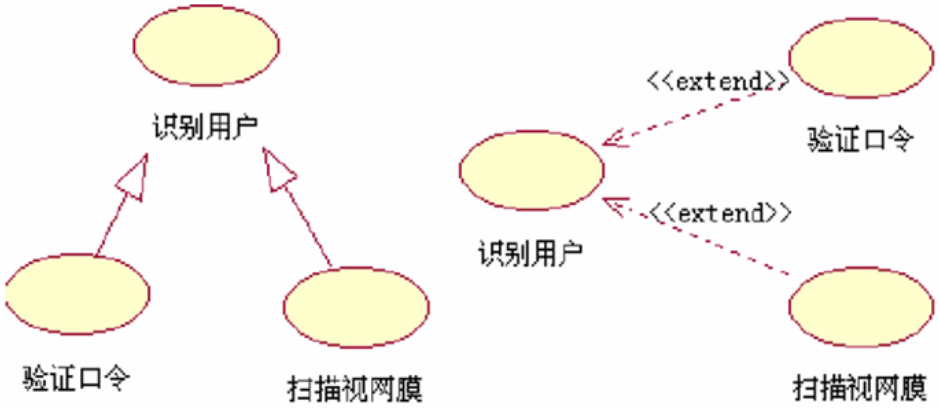
——泛化关系举例



同一业务目的不同技术实现

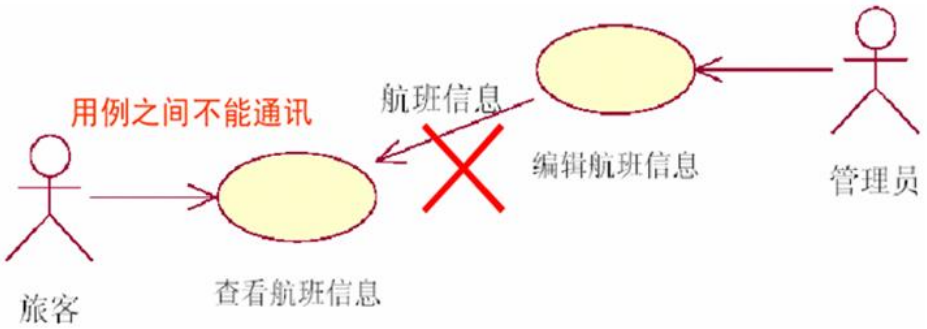
识别用例的关系

——泛化？扩展？



采用关系不同，文档结构不同

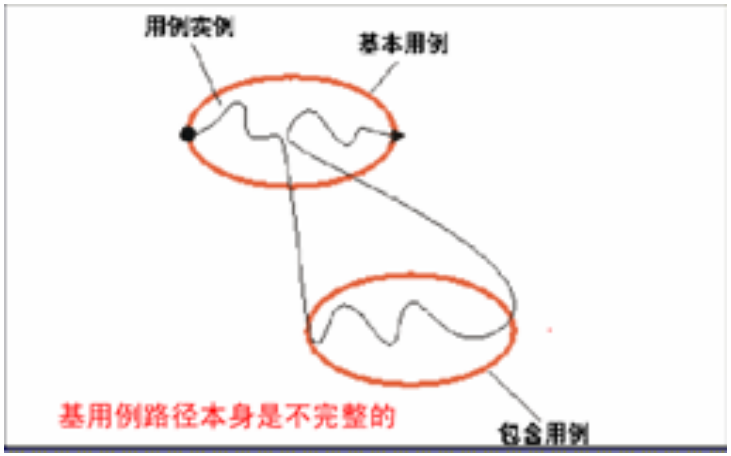
识别用例的关系



除此之外，不能有别的关系

识别用例的关系

——包含



识别用例的关系

——何时使用包含关系

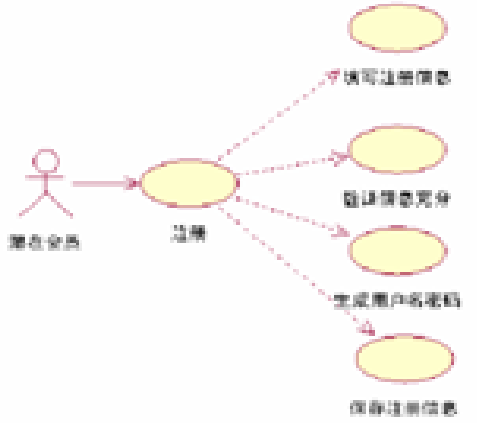
- 某些步骤在多个用例重复出现，且单独形成价值
- 用例的步骤较多时，可以用Include简化（慎用）

识别用例的关系

——包含关系举例

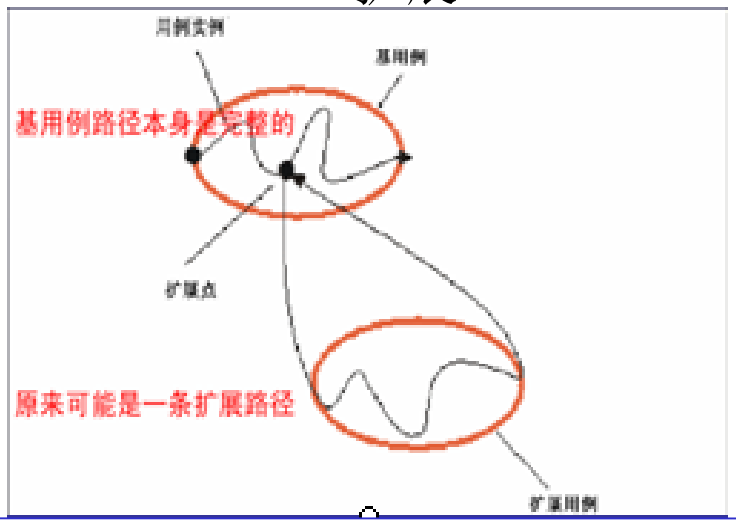
识别用例的关系

——包含关系误用



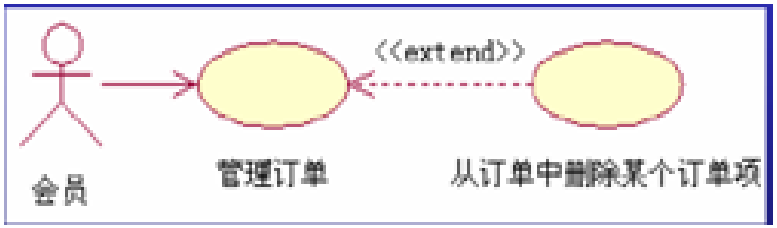
识别用例的关系

——扩展



- 识别用例的关系

——扩展关系举例



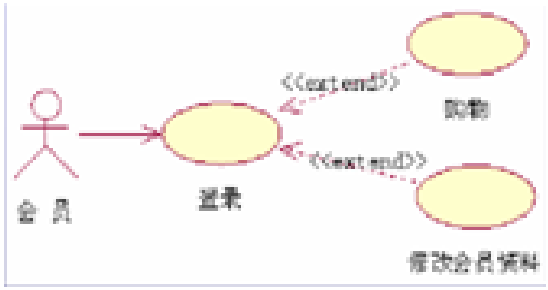
- 识别用例的关系

——何时使用扩展关系

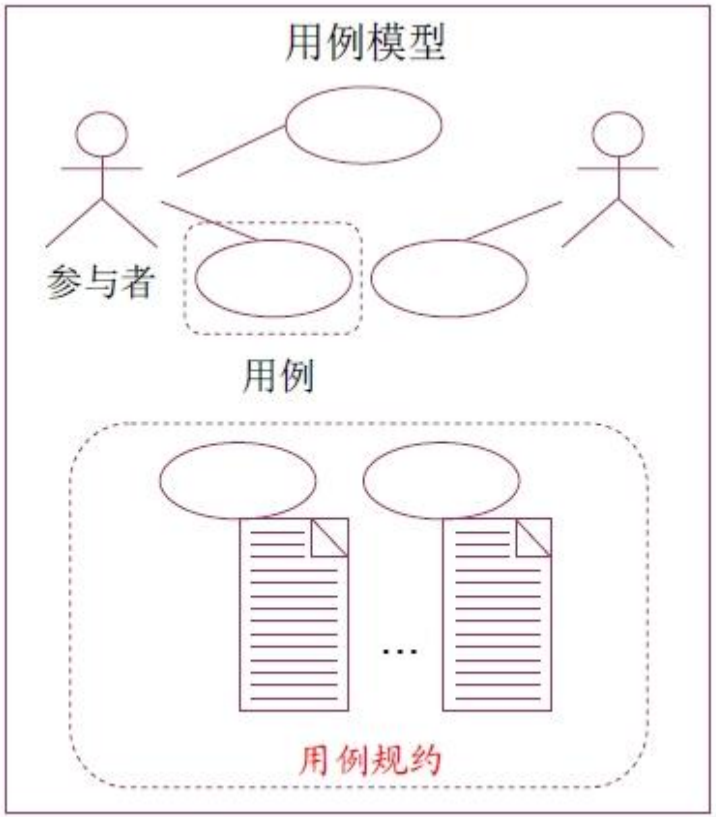
- 拓展路径步骤多
- 拓展路径内部还存有扩展点——扩展之扩展
- 拓展路径未定或容易变化 ——分离以“冻结”基用例

- 识别用例的关系

——扩展关系的误用



用例规约



Name of the Use Case (用例的名字)

Description (描述)
Actor(s) (参与者)
Flow of events(事件流)
  Basic flow(常规流)
    Event 1 (事件)
    Event 2
  ……
  Alternate flow(备选流)
Pre-conditions (前置条件)
Post-conditions (后置条件)
……

静态建模

任何建模语言都以静态建模机制为基础,标准建模语言UML也不例外。所谓静态建模是指对象之间通过属性互相联系，而这些关系不随时间而转移。

类和对象的建模，是UML建模的基础。

类图(Class diagram)

对象图(Object diagram)

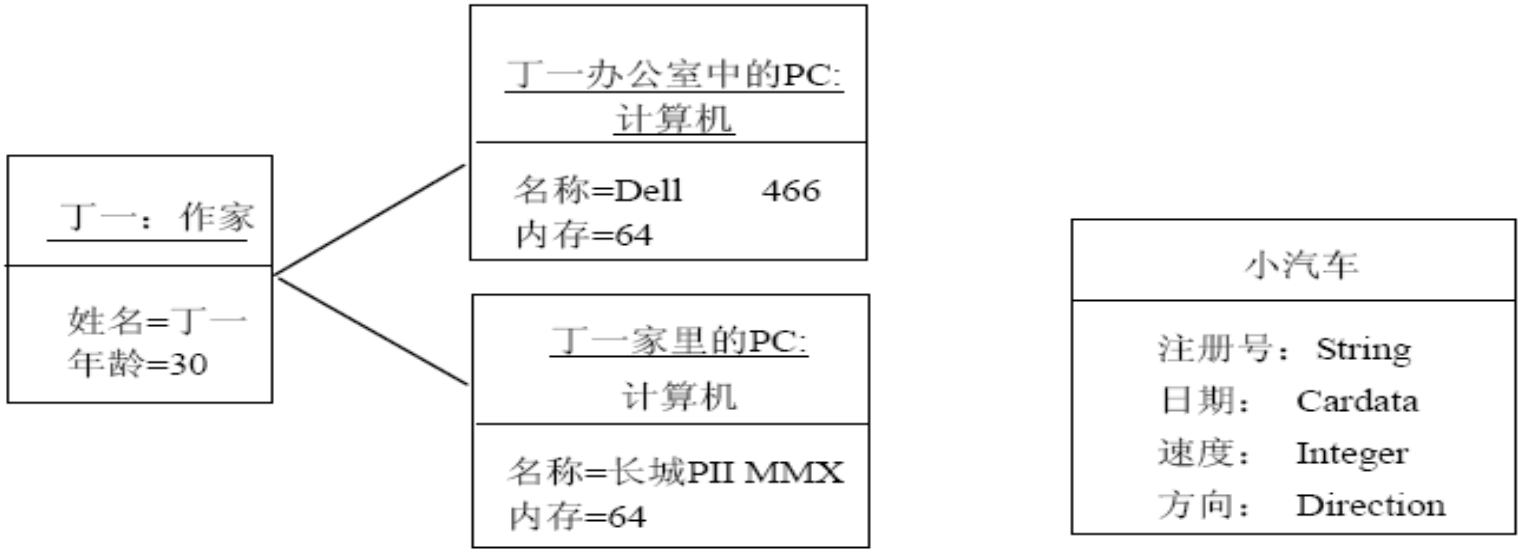
包图(Package diagram)

构件图(Component diagram)

配置图(Deployment diagram)

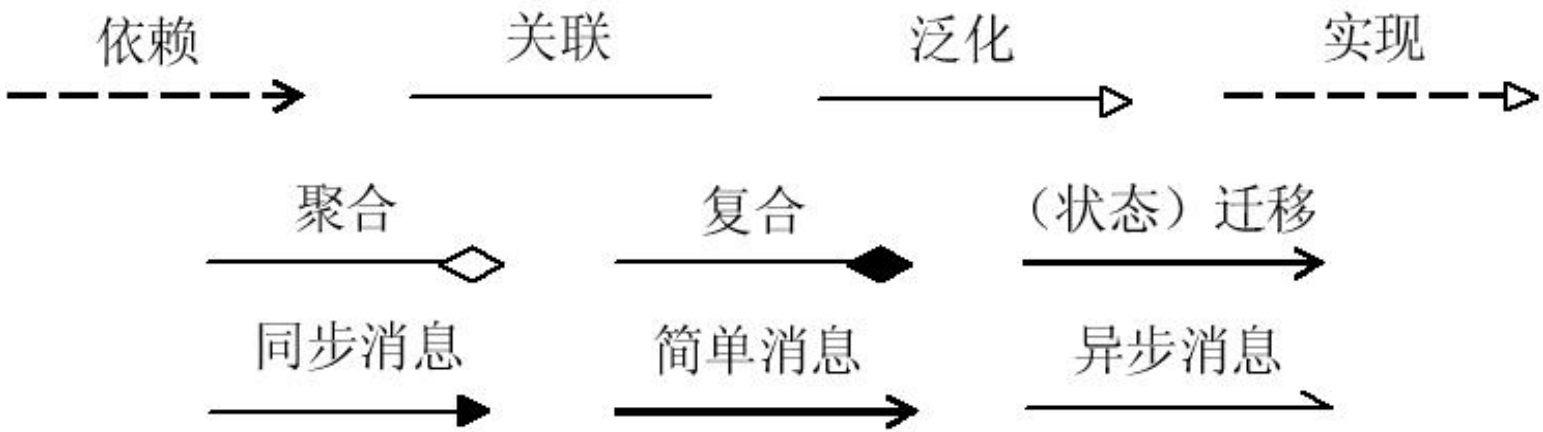

类名
类名
属性：类型
操作
对象名
对象名:类名
属性
操作

(a) (b)


丁一：作家
姓名=丁一
年龄=30
丁一办公室中的PC:
计算机
名称=Dell 466
内存=64
丁一家里的PC:
计算机
名称=长城PII MMX
内存=64
小汽车
注册号： String
日期： Cardata
速度： Integer
方向： Direction

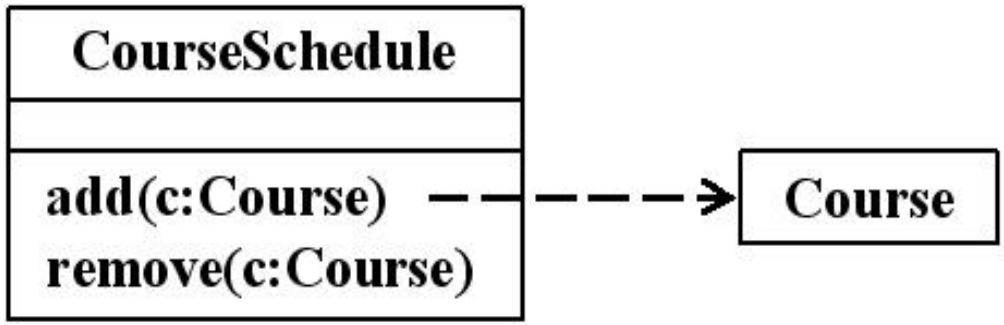
(a) (b)

UML的关系



依赖关系

- 依赖(Dependency)是两个事物之间的语义关系，其中一个事物发生变化会影响到另一个事物的语义，它用一个虚线箭头表示。
- 虚线箭头的方向从源事物指向目标事物，表示源事物依赖于目标事物。

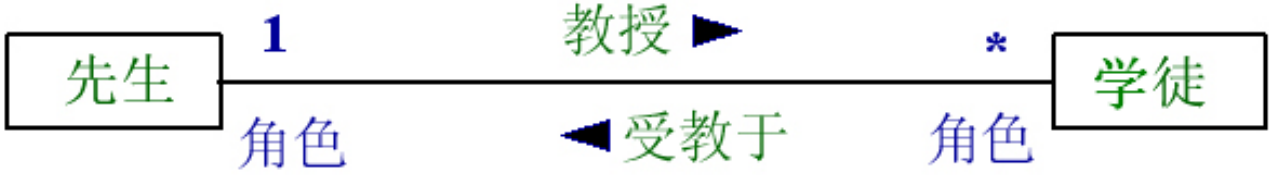


关联关系

- 关联(association)是一种结构关系，它描述了两个或多个类的实例之间的连接关系，是一种特殊的依赖。
- 关联分为普通关联、限定关联、关联类，以及聚合与复合。

关联关系——普通关联

- 普通关联是最常见的关联关系，只要类与类之间存在连接关系就可以用普通关联表示。普通关联又分为二元关联和多元关联。
- 二元关联描述两个类之间的关联，用两个类之间的一条直线来表示，直线上可写上关联名。



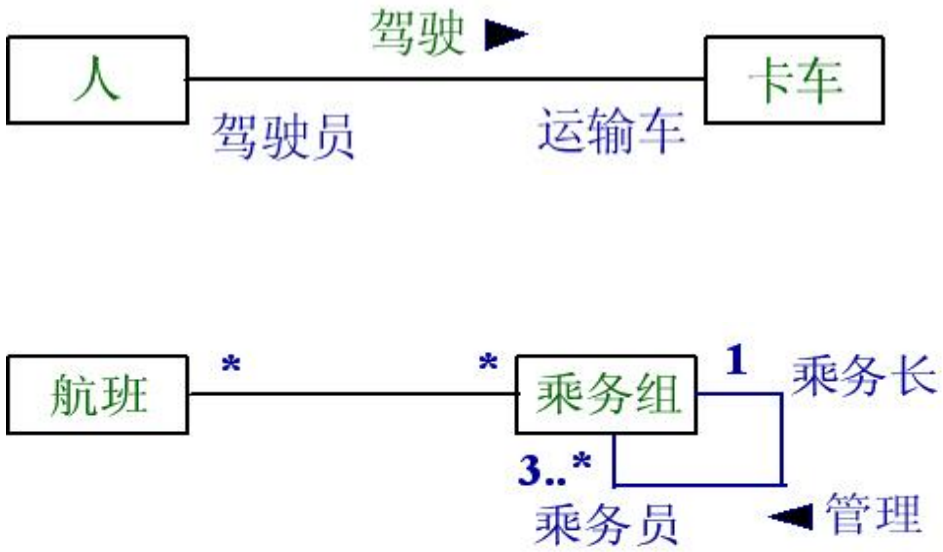
关联关系——普通关联

- 多重性（multiplicity）：多重性表明在一个关联的两端连接的类实例个数的对应关系，即一端的类的多少个实例对象可以与另一端的类的一个实例相关。
- 如果图中没有明确标出关联的多重性，则默认的多重性为1。

1	—— 1 个实例
0..1	—— 0 到 1 个实例
0..* 或 *	—— 0 到多个实例
1+ 或 1..*	—— 1 到多个实例

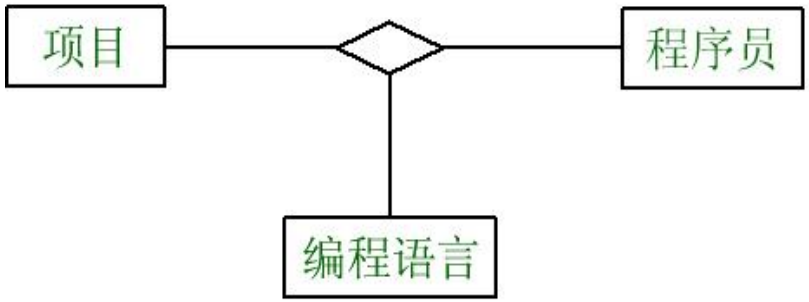
关联关系——普通关联

- 角色：关联端点上还可以附加角色名，表示类的实例在这个关联中扮演的角色。UML还允许一个类与它自身关联。



关联关系——普通关联

- 多元关联：多元关联是指3个或3个以上类之间的关联。
- 多元关联由一个菱形，以及由菱形引出的通向各个相关类的直线组成，关联名可标在菱形的旁边，在关联的端点也可以标上多重性等信息。

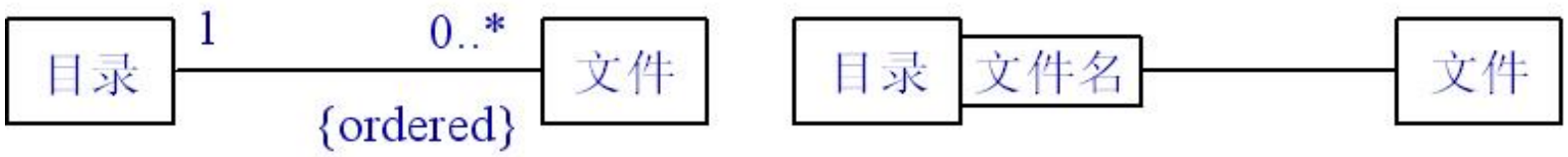


关联的链

程序员	编程语言	项目
宫力	C++	CAD
周斌	Java	网站
陈健	C++	CAD
陈健	C++	MIS
.....		

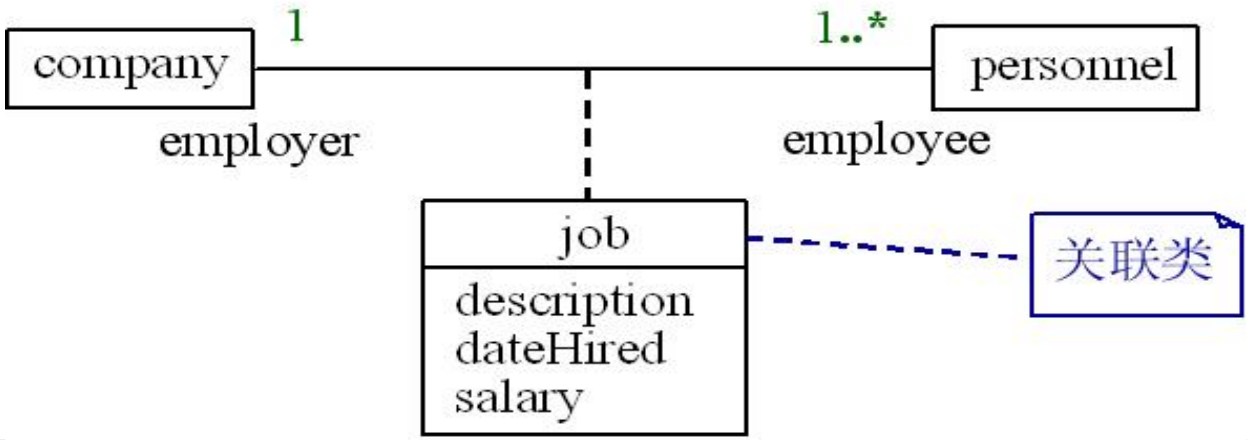
关联关系——限定关联

- 限定关联通常用在一对多或多对多的关联关系中，可以把模型中的多重性从一对多变成一对一，或将多对多简化成多对一。
- 在类图中把限定词（qualifier）放在关联关系末端的一个小方框内。



关联关系——关联类

- 在关联关系比较简单的情况下，关联关系的语义用关联关系的名字来概括。
- 但在某些情况下，需要对关联关系的语义做详细的定义、存储和访问，为此可以建立关联类（association class），用来描述关联的属性。
- 关联中的每个链与关联类的一个实例相联系。关联类通过一条虚线与关联连接。



关联关系——聚合

- 聚合（Aggregation）也称为聚集，是一种特殊的关联。它描述了整体和部分之间的结构关系。
- 两种特殊的聚合关系：共享聚合（shared aggregation）和复合聚合（composition aggregation）。
- 如果在聚合关系中处于部分方的实例可同时参与多个处于整体方实例的构成，则该聚合称为共享聚合。

共享聚合

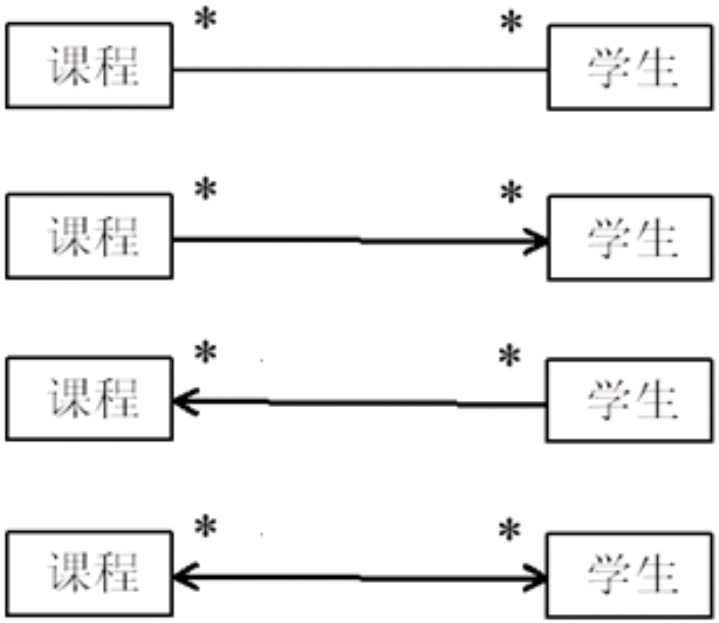
关联关系——聚合

- 如果部分类完全隶属于整体类，部分类需要与整体类共存，一旦整体类不存在了，则部分类也会随之消失，或失去存在价值，则这种聚合称为复合聚合。

复合聚合

关联关系——导航

- 导航（navigability）是关联关系的一种特性，它通过在关联的一个端点上加箭头来表示导航的方向。



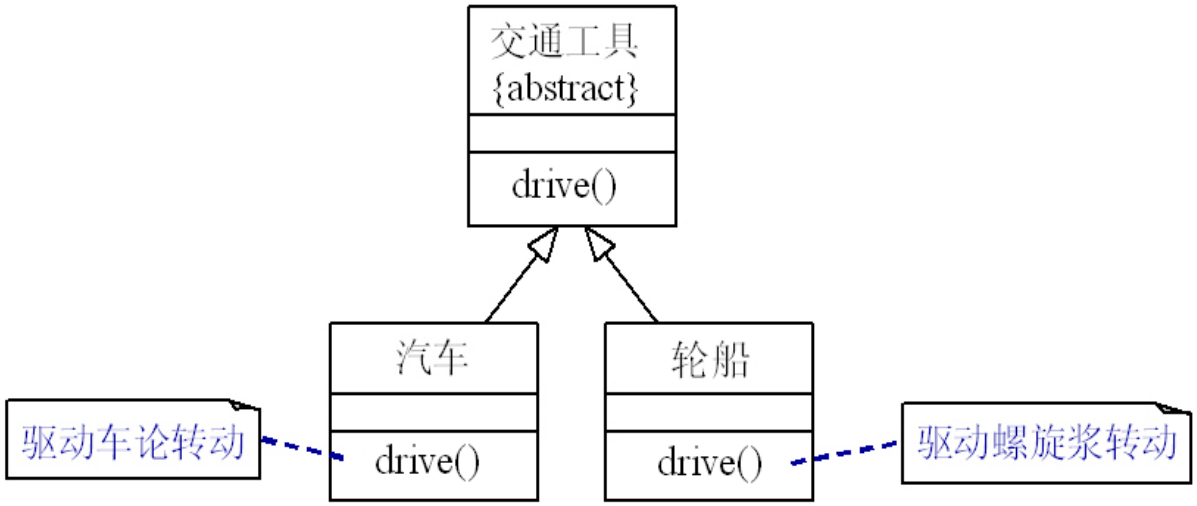
导航

泛化关系

- 泛化(generalization)关系就是一般类和特殊类之间的继承关系。
- 在UML中，一般类亦称泛化类，特殊类亦称特化类。
- 泛化针对类型而不针对实例，因为一个类可以继承另一个类，但一个对象不能继承另一个对象。

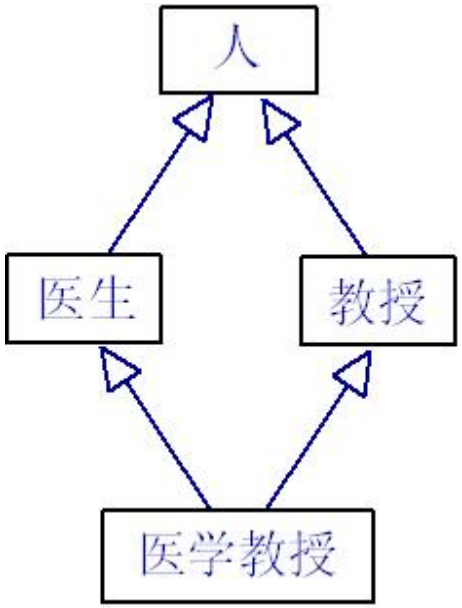
泛化关系——普通泛化

- 普通泛化与前面讲过的继承基本相同。但在泛化关系中常遇到抽象类。
- 一般称没有具体对象的类为抽象类。抽象类通常作为父类，用于描述其他类（子类）的公共属性和行为。

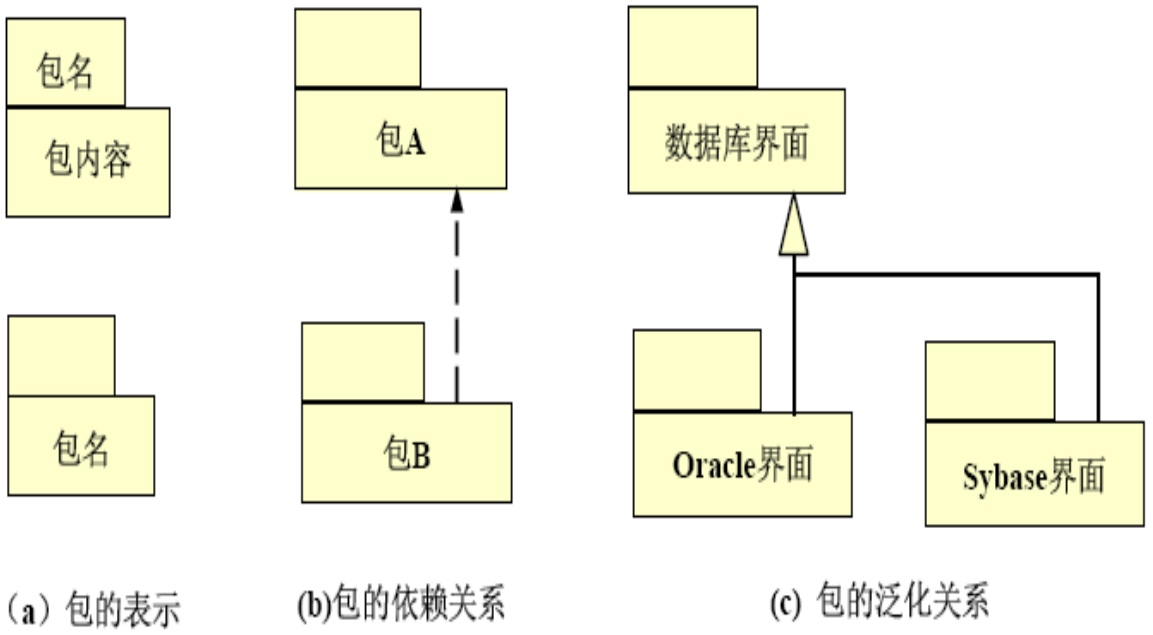


泛化关系——普通泛化

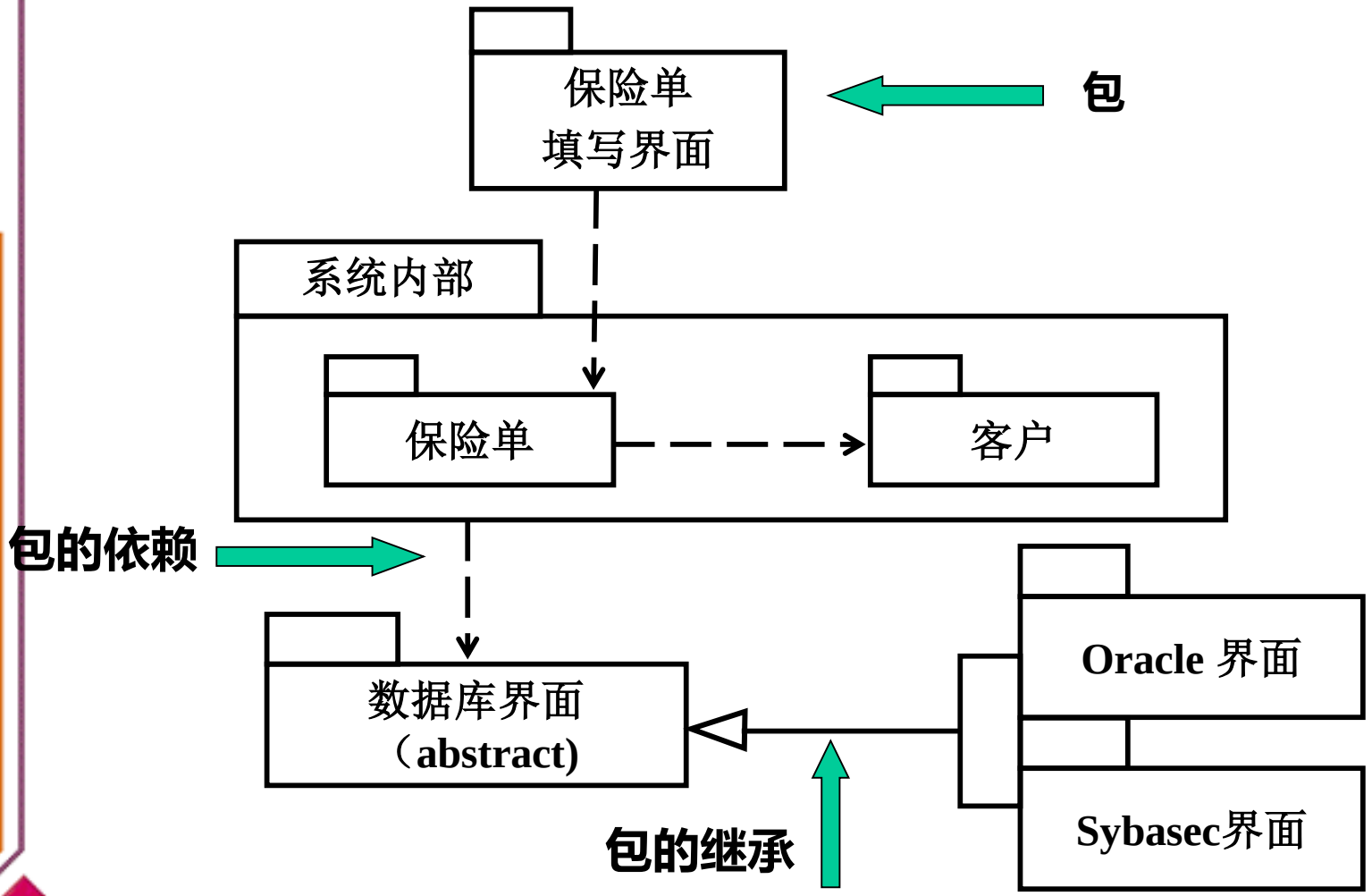
- 普通泛化可以分为多重继承和单继承。多重继承是指一个子类可同时继承多个上层父类。



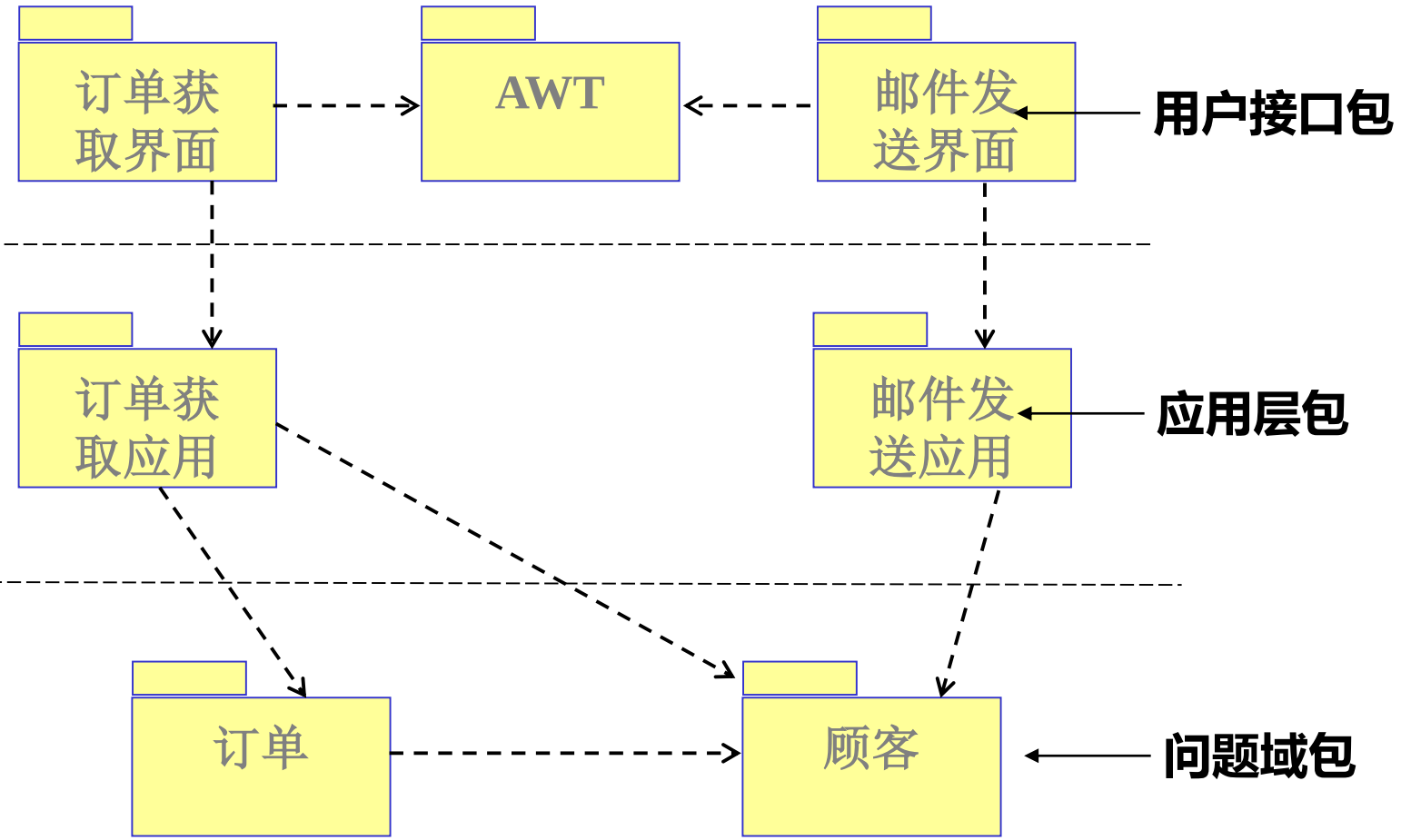
包图



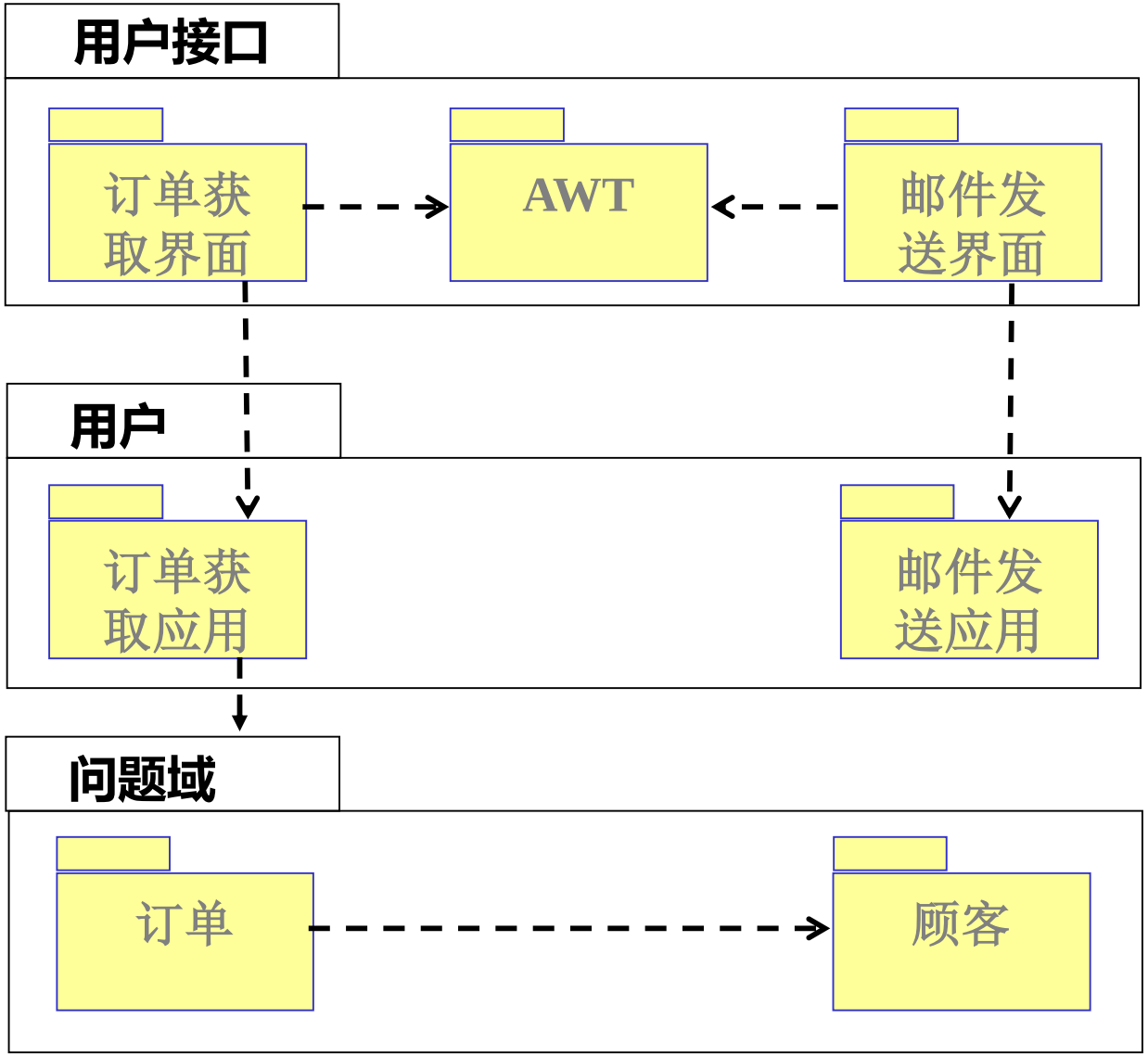
（a）包的表示　(b)包的依赖关系　(c) 包的泛化关系


保险单
填写界面
包
系统内部
保险单
客户
包的依赖
数据库界面
（abstract)
Oracle 界面
Sybasec界面
包的继承

包图举例

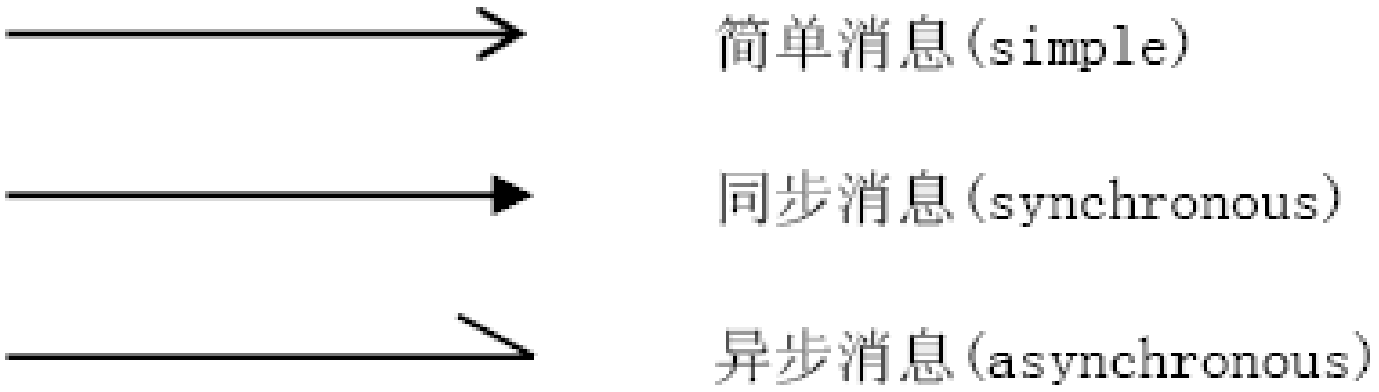


包图另一种划分方式



动态建模

动态模型主要描述系统的动态行为和控制结构。在动态模型中,对象间的交互是通过对象间消息的传递来完成的。



动态模型

状态图、活动图、顺序图、合作图。

状态图(state diagram)：状态图用来描述对象，子系统，系统的生命周期。

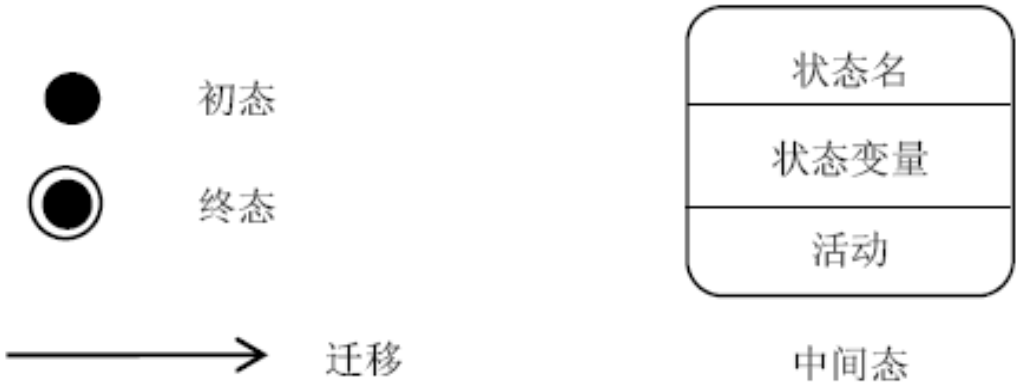
活动图(activity diagram)：着重描述操作实现中完成的工作以及用例实例或对象中的活动，活动图是状态图的一个变种。

顺序图(sequence diagram)：是一种交互图，主要描述对象之间的动态合作关系以及合作过程中的行为次序，常用来描述一个用例的行为。

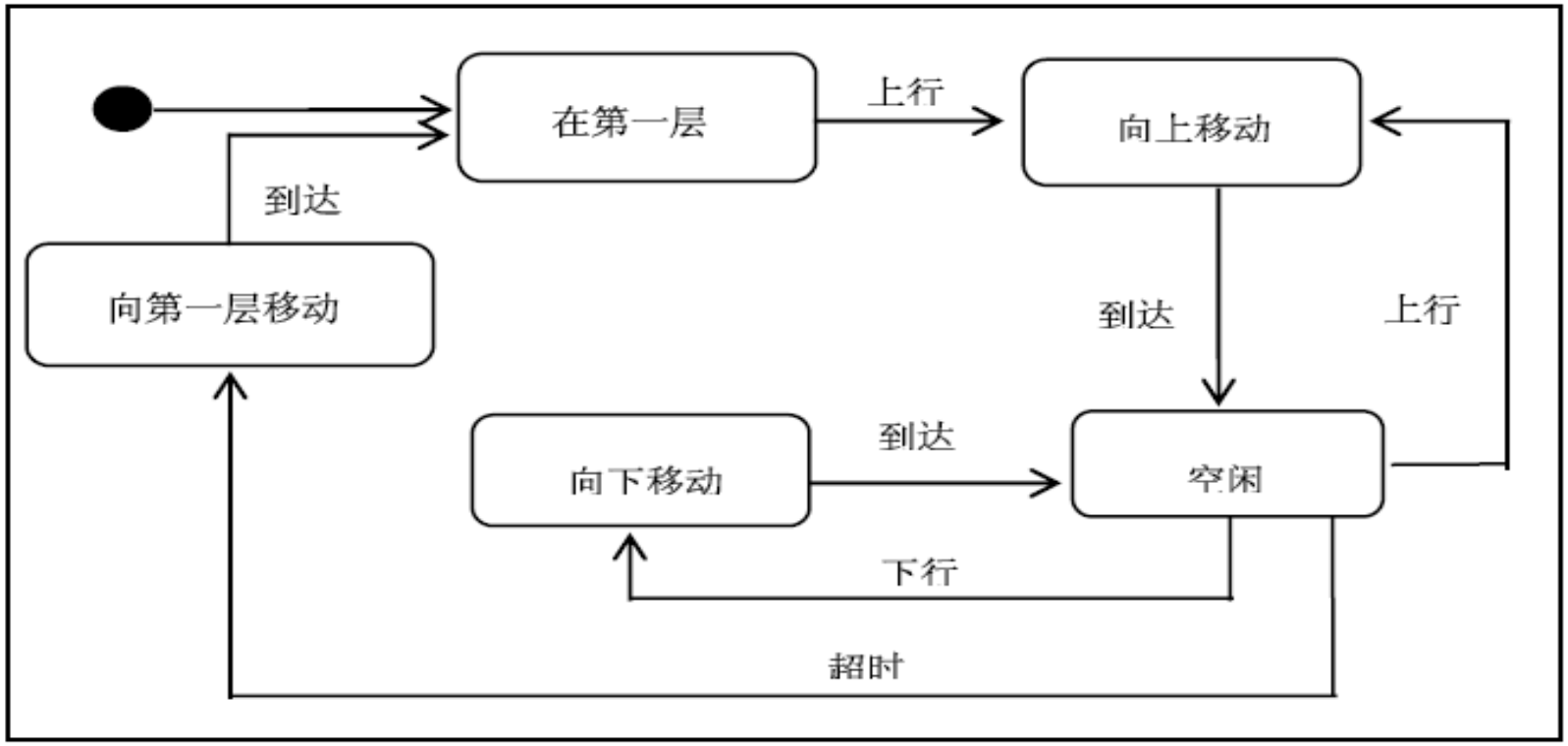
合作图(collaboration diagram)：用于描述相互合作的对象间的交互关系，它描述的交互关系是对象间的消息连接关系。

状态图

状态图(State Diagram)用来描述一个特定对象的所有可能的状态及其引起状态转移的事件。一个状态图包括一系列的状态以及状态之间的转移。

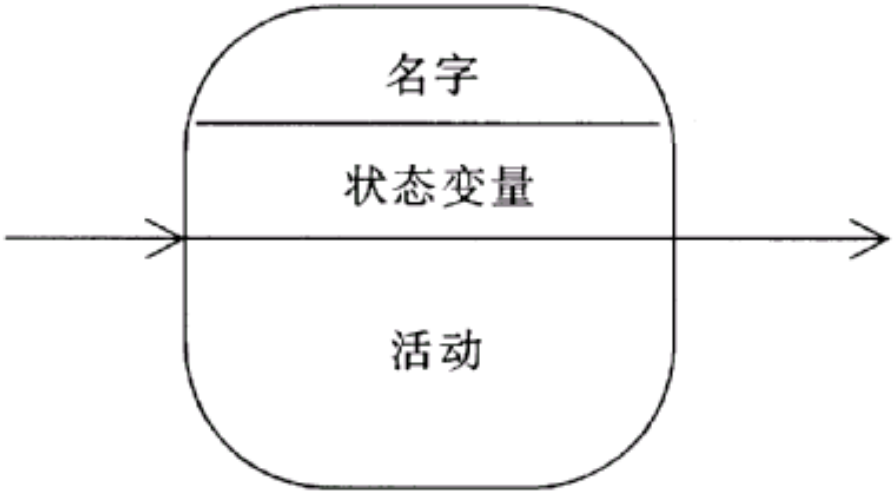


状态迁移　一个对象的状态的变迁称为状态迁移。通常是由事件触发的，此时应标出触发转移的事件表达式。如果转移上未标明事件，则表示在源状态的内部活动执行完毕后自动触发转移。



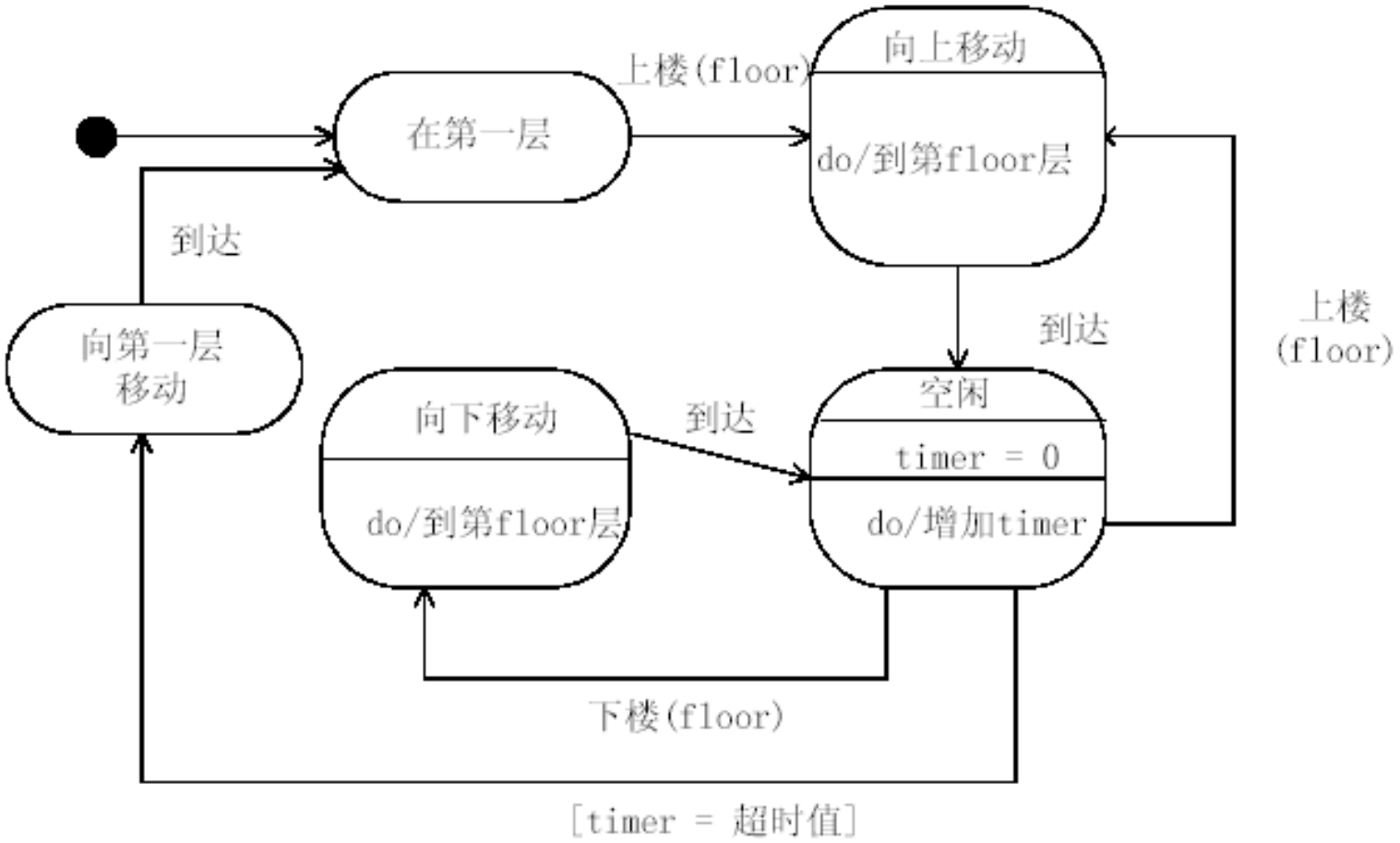
状态图

- 中间状态用圆角矩形表示，可能包含三个部分，第一部分为状态的名称；第二部分为状态变量的名字和值，这部分是可选的；第三部分是活动表，这部分也是可选的。



中间状态

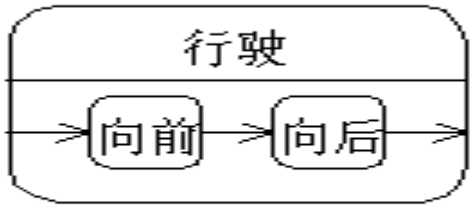
在“空闲”状态，将属性timer的值置0，然后连续递增timer的值，直到“上楼”或“下楼”事件发生,或守卫条件“timer=超时值”。



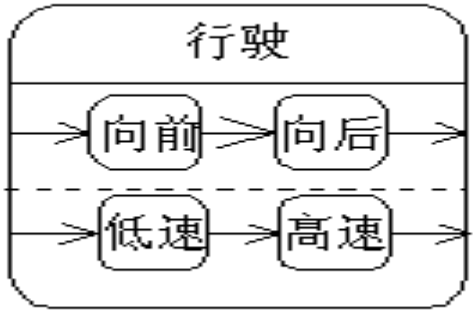
加上属性的状态转换

嵌套状态图

状态图可能有嵌套的子状态图，且子状态图可以是另一个状态图。子状态又可分为两种：“与”子状态和“或”子状态



或关系的子状态

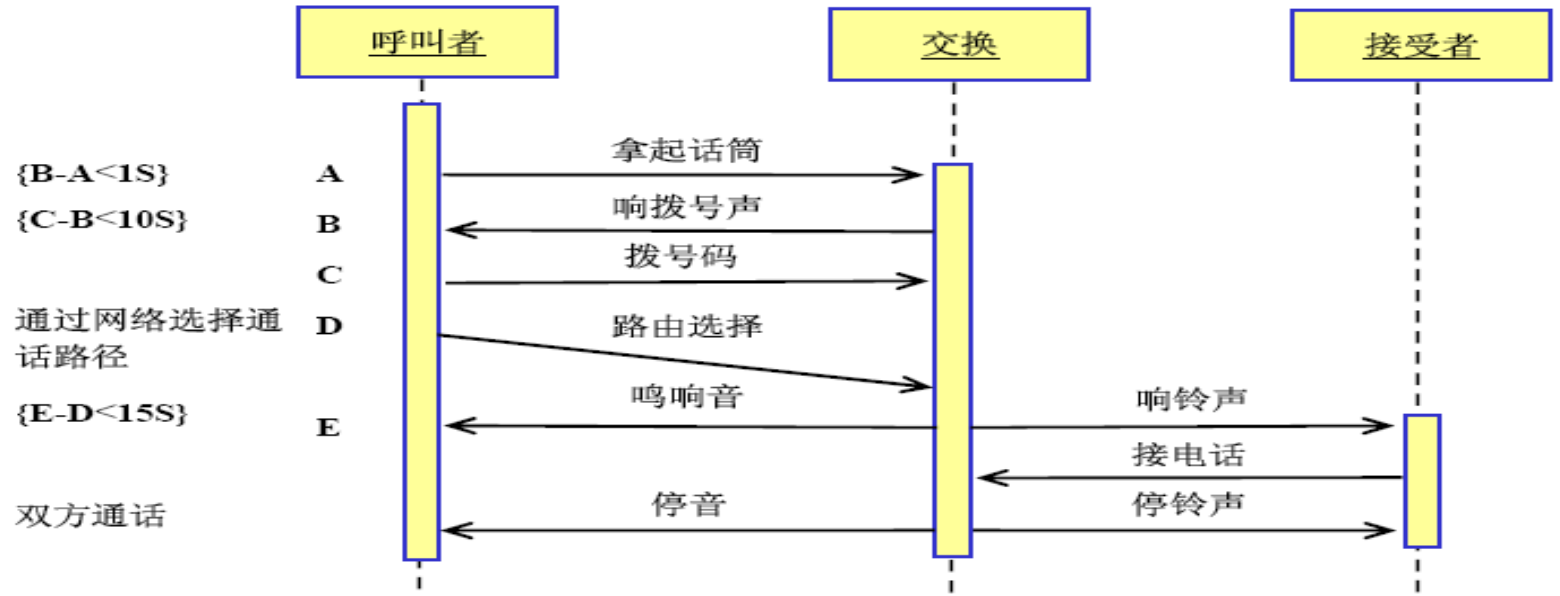


与子状态及或子状态

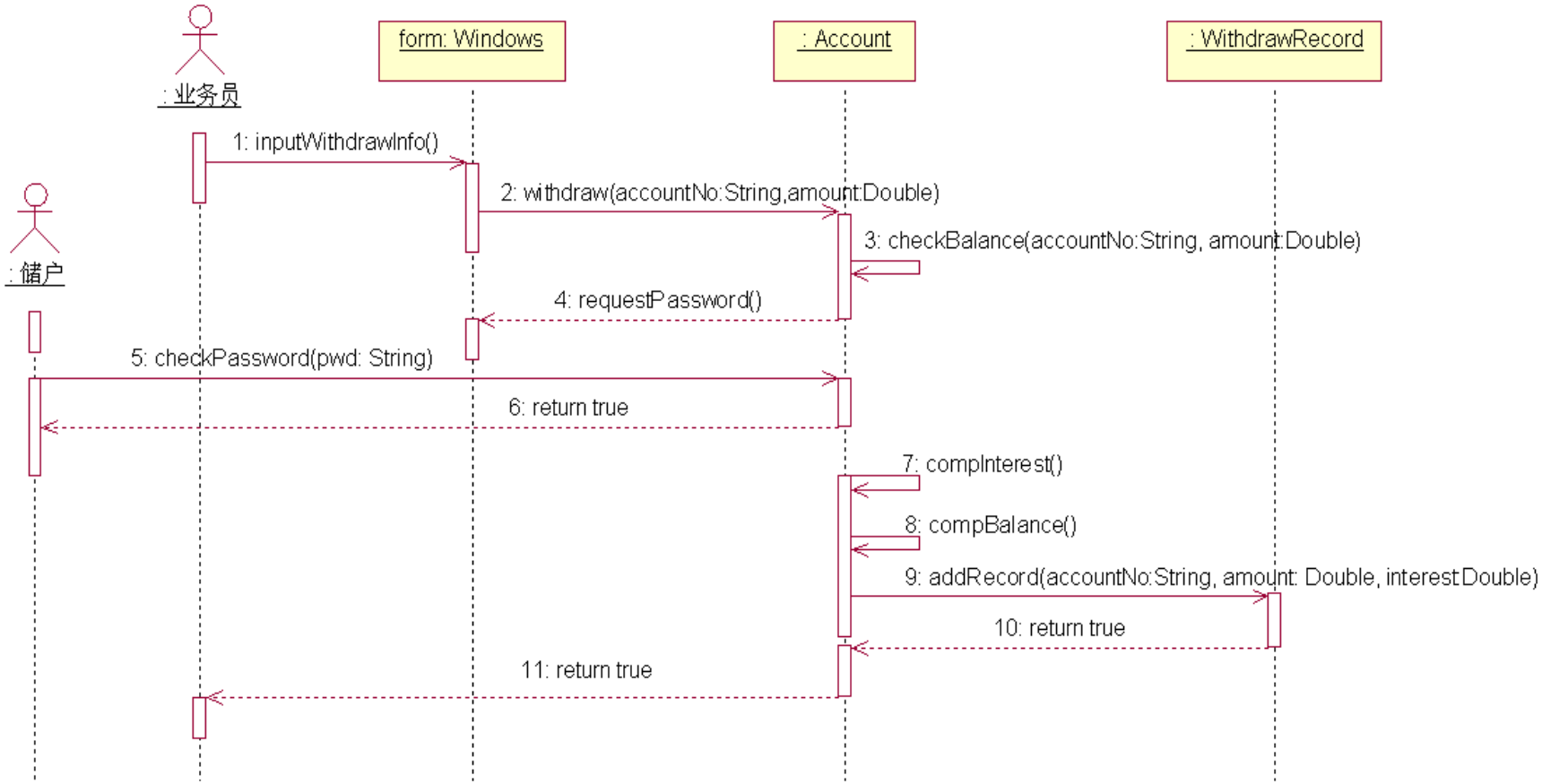
顺序图

顺序图(Sequence Diagram)用来描述对象之间动态的交互行为，着重体现对象间消息传递的时间顺序。

顺序图存在两个轴:水平轴表示一组对象,垂直轴表示时间。



取款用例的顺序图

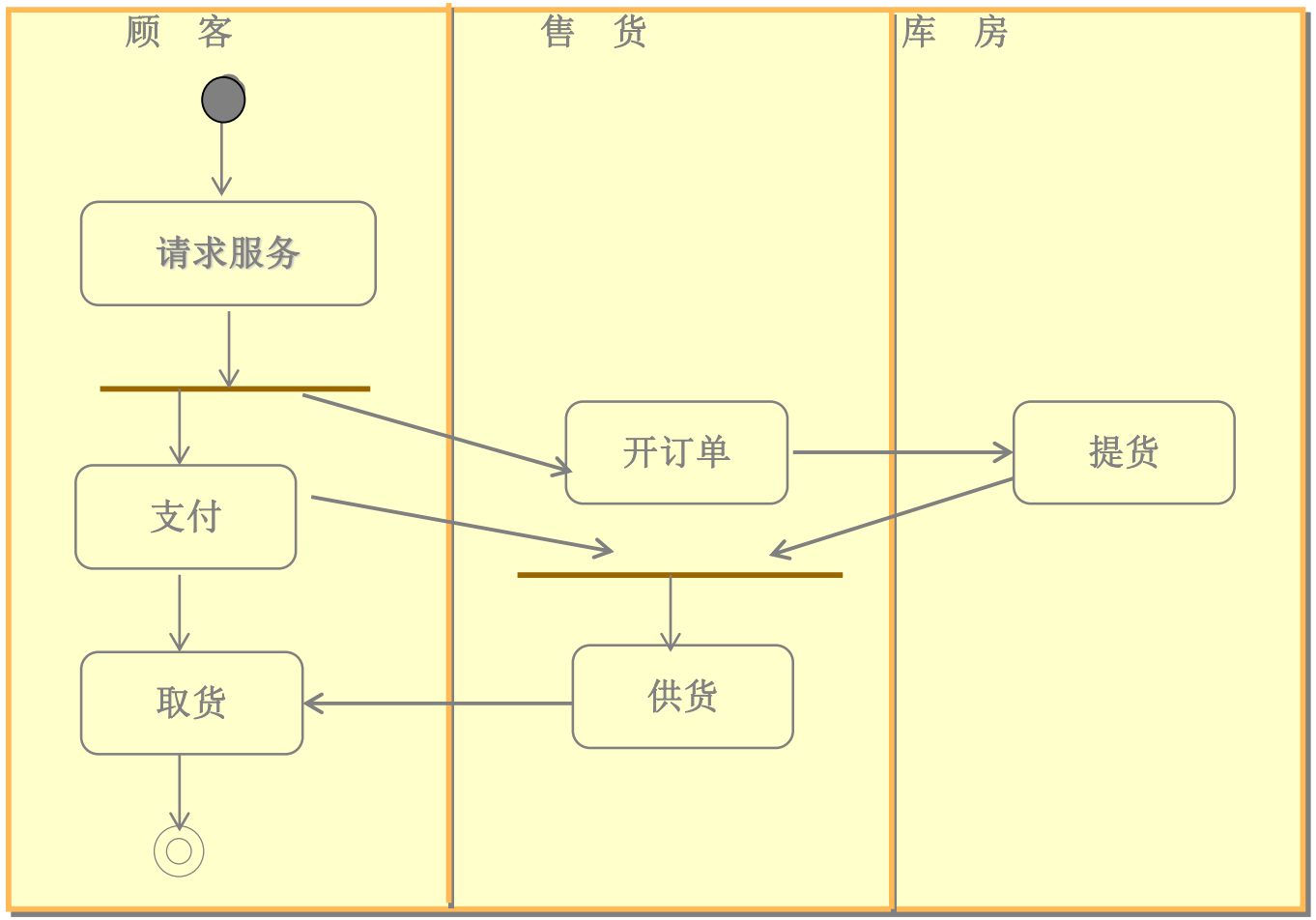

: 业务员
form: Windows
: Account
: WithdrawRecord
: 储户
1: inputWithdrawInfo()
2: withdraw(accountNo:String,amount:Double)
3: checkBalance(accountNo:String, amount:Double)
4: requestPassword()
5: checkPassword(pwd: String)
6: return true
7: compInterest()
8: compBalance()
9: addRecord(accountNo:String, amount: Double, interest:Double)
10: return true
11: return true

活动图

活动图(Activity Diagram)的应用非常广泛,它既可用来描述操作(类的方法)的行为,也可以描述用例和对象内部的工作过程,并可用于表示并行过程。

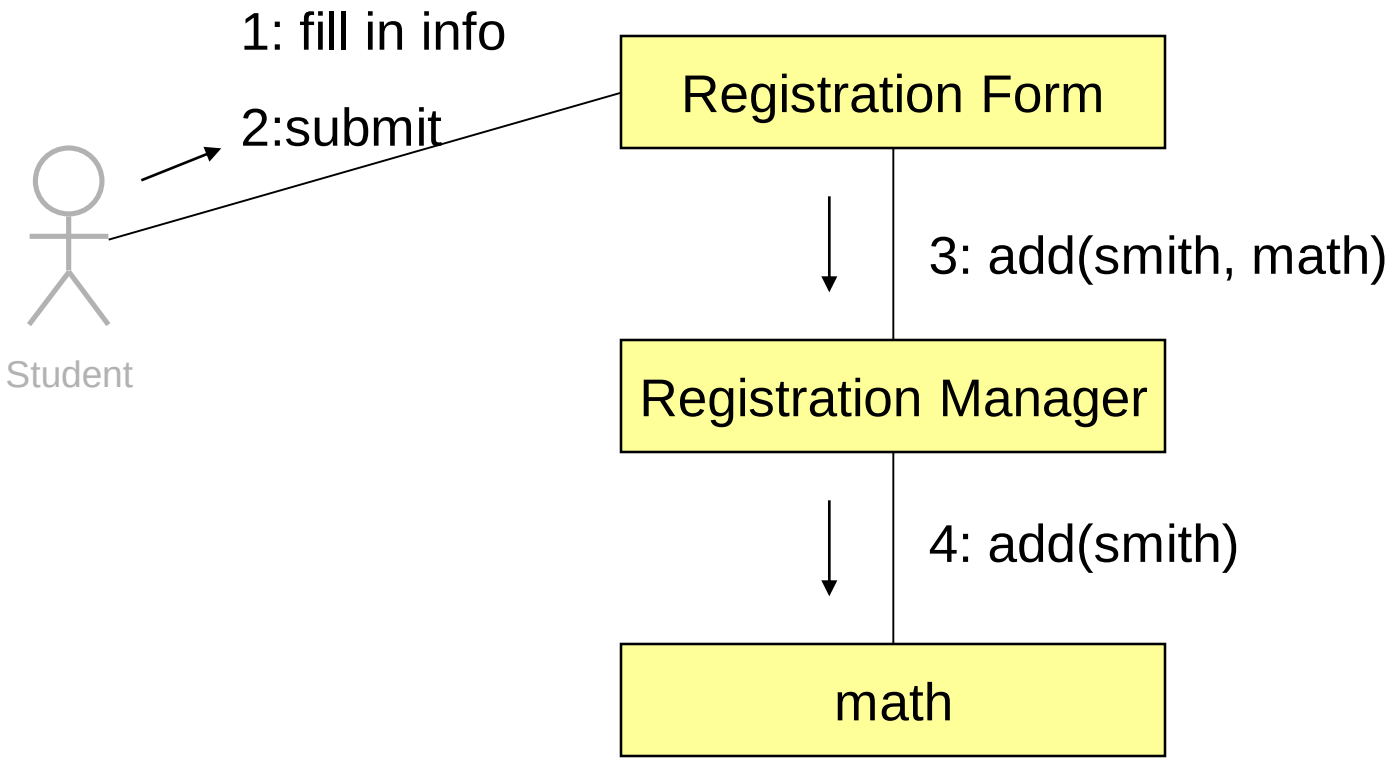
活动图是由状态图变化而来的,它们各自用于不同的目的。活动图描述了系统中各种活动的执行的顺序。刻化一个方法中所要进行的各项活动的执行流程。

构成活动图的模型元素有：活动、转移、对象、信号、泳道等。



合作图

合作图(Collaboration Diagram),也称为协作图，用于描述相互合作的对象间的交互关系和链接(Link)关系。

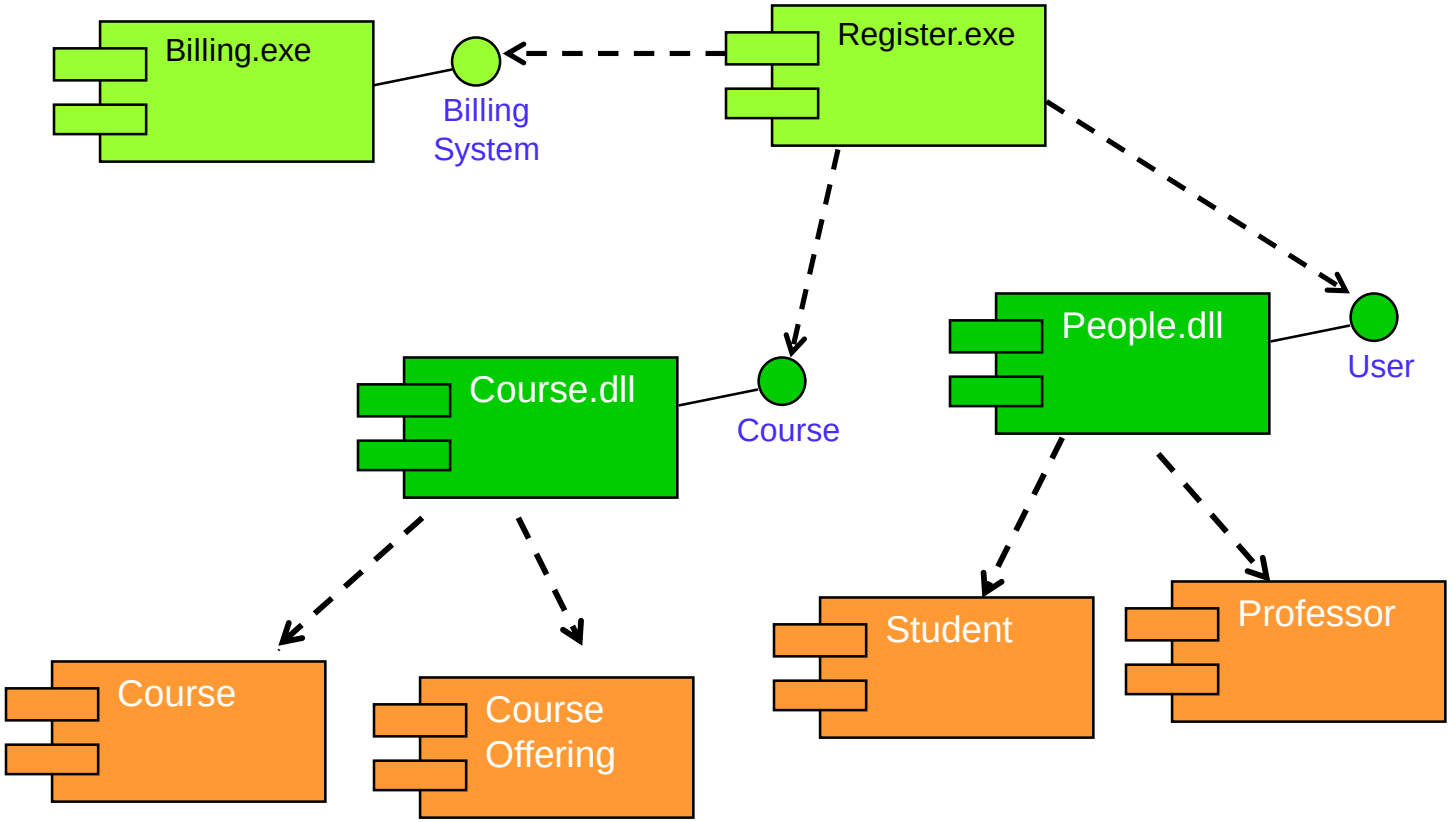


物理架构

- 实现视图
- 部署视图

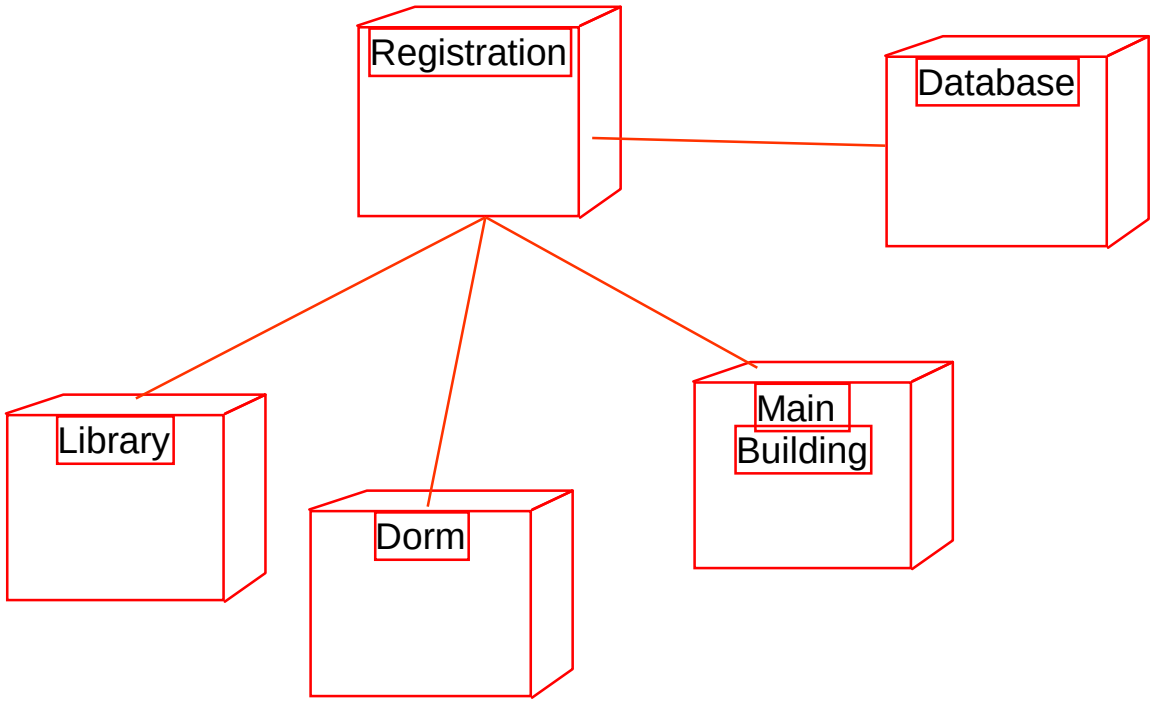
物理架构—实现视图

- 实现视图描述可重用的系统组件以及组件之间的依赖。

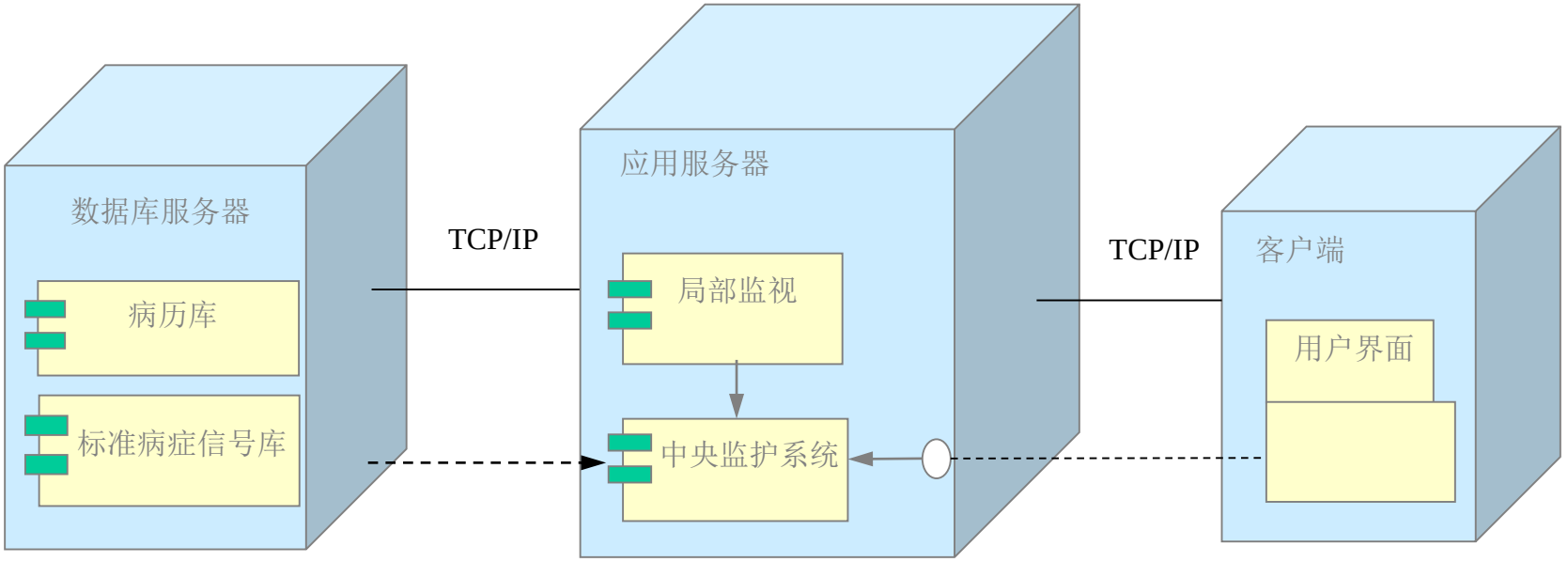


物理架构—部署视图

- 部署视图描述系统资源在运行时的物理分布，系统资源成为结点。



医院诊疗系统的配置图



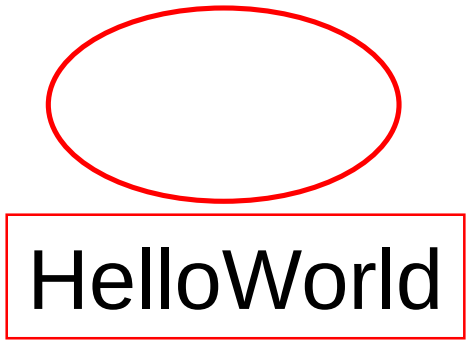
实例－Hello World

- 第一个程序就是Hello world，一个在屏幕上简单地打印出“Hello world!”语句的例子。
- 在java中一个在浏览器中显示“Hello World!”的Applet的代码如下：

```
import  java.awt.Graphics;
class HelloWorld extends java.applet.Applet{
      public  void paint( Graphics g ){
       g.drawString("Hello World!",10,10 );
      }
}
```

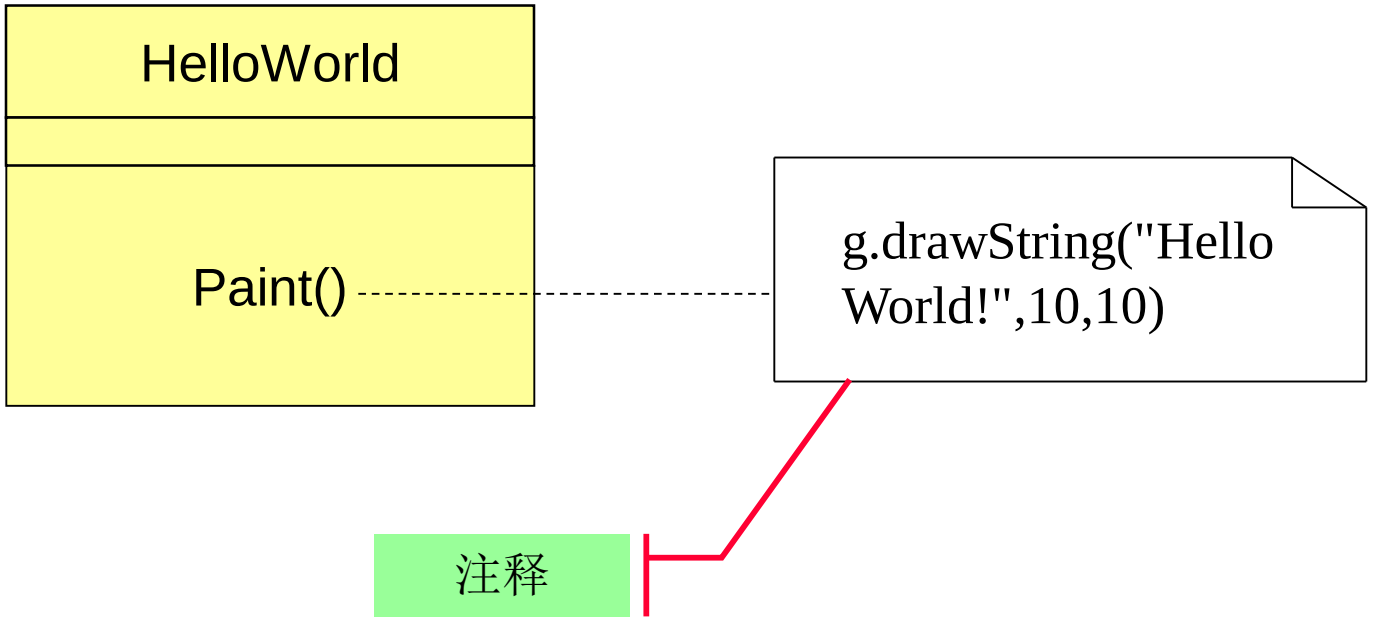
实例－Hello World

- 用例图



实例－Hello World

- HelloWorld类



实例－Hello World

- 类图