6-4 面向对象的设计

2019年5月26日 22:37

5 对象设计

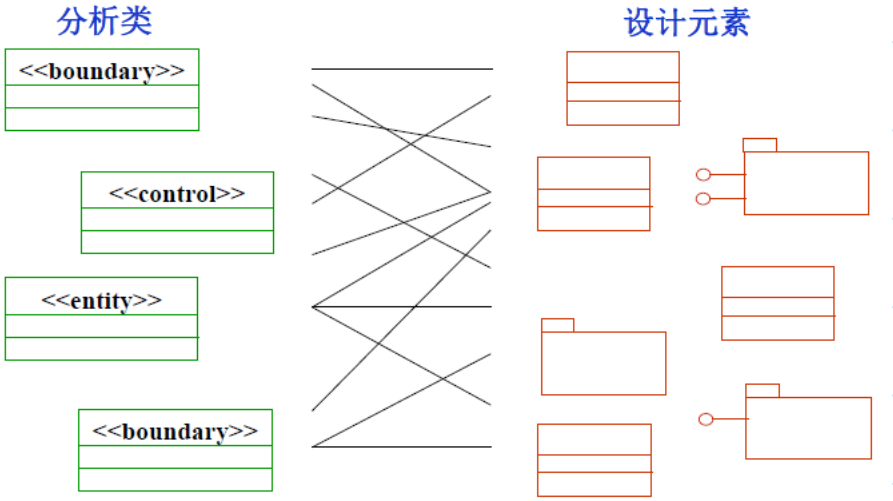
对象设计概述

- 对象设计
  - 细化需求分析和系统设计阶段产生的模型
  - 确定新的设计对象
  - 消除问题域与实现域之间的差距
- 对象设计的主要任务
  - 精化类的属性和操作
    - 明确定义操作的参数和基本的实现逻辑
    - 明确定义属性的类型和可见性
  - 明确类之间的关系
  - 整理和优化设计模型

对象设计的基本步骤

1. 创建初始的设计类
2. 细化属性
3. 细化操作
4. *定义状态
5. 细化类之间的关联/依赖/泛化关系

创建初始的设计类



细化属性

- 细化属性
- 具体说明属性的名称、类型、缺省值、可见性等visibility attributeName : Type = Default
  - Public: ‘+’;
  - Private: ‘-’
  - Protected: ‘#’
- 属性的来源:
  - 类所代表的现实实体的基本信息;
  - 描述状态的信息;
  - 描述该类与其他类之间关联的信息;
  - 派生属性(derived attribute):该类属性的值需要通过计算其他属性的值才能得到,属性前面加“/”表示。
  - 例如:类CourseOffering中的“学生数目”/numStudents:int
- 基本原则
  - 属性命名命名符合规范(名词组合,首字母小写)
  - 尽可能将所有属性的可见性设置为private;
  - 仅通过set方法更新属性;
  - 仅通过get方法访问属性;
  - 在属性的set方法中,实现简单的有效性验证,而在独立的验证方法中实现复杂的逻辑验证。

关于状态属性

- 状态用实体类的一个或多个属性来表示。

    - 对“订单”类来说，可以设定一个状态属性“订单状态”，取值为enum{未付款、取消、已付款未发货、已发货、已确认收货未评价、买方已评价、双方已评价、...}。
    - 也可以有多个属性来表示状态：是否已付款、是否已发货、是否已确认、买方是否已评价、卖方是否已评价、等。每个状态属性的类型均为boolean。
- 大部分状态属性，可以由该类的其他属性的值进行逻辑判断得到；
    - 若订单处于“未付款”状态，则该订单的“订单变迁记录”中一定不会包含有付款信息；若它处于“买家已评价”状态，则它的“买家评价”属性一定不为空。
- 从一个状态值到另一个状态值的变迁，一定是由该实体类的某个操作所导致的；
    - 订单从不存在到变为“未付款”，是由new操作导致的状态变化；
    - 订单从“已发货”到“已确认”的变化，是由“确认收货”操作所导致的状态变化；
    - 根据这一原则，可以判断你为实体类所设计的操作是否完整。

关于关联属性

- 两个类之间有association关系，意味着需要永久管理对方的信息，需要在程序中能够从类1的object“导航”(navigate)到类2的object。
    - 例如：“订单”类与“买家”类产生双向关联，意即一个订单对象中需要能够找到相应的“买家”对象，反之买家对象需要知道自己拥有哪些订单对象。
        - 订单类中有一个关联属性buyer，其数据类型是“买家”类；
        - 买家类中有一个关联属性OrderList，其数据类型是“订单”类构成的序列；
- 关联属性不只是一个ID，而是一个或多个完整的对方类的对象。
- 务必与数据库中的“外键”区分开：
    - 订单table中有一个外键：买家ID(字符串类型)，靠它与买家table联系起来；
- 不要用关系数据模式的设计思想来构造类的属性。
- 关联属性的目标：在程序运行空间内实现object之间的导航，而无需经过数据库层的存取。

细化操作

- 找出满足基本逻辑要求的操作：针对不同的actor，分别思考需要类的哪些操

作；

- 补充必要的辅助操作：
  - 初始化类的实例、销毁类的实例——Student(…)、~Student()；
  - 验证两个实例是否等同——equals()；
  - 获取属性值(get操作)、设定属性值(set操作)——getXXX()、setXXX(…)；
  - 将对象转换为字符串——toString()；
  - 复制对象实例——clone()；
  - 用于测试类的内部代码的操作——main()；
  - 支持对象进行状态转换的操作；
- 细化操作时，要充分考虑类的“属性”与“状态”是否被充分利用：
  - 对属性进行CRUD；
  - 对状态进行各种变更；
- 给出完整的操作描述：
  - 确定操作的名称、参数、返回值、可见性等；
  - 应该遵从程序设计语言的命名规则(动词+名词，首字母小写)
- 详细说明操作的内部实现逻辑。
  - 复杂的操作过程采用伪代码或者绘制程序流程图/活动图方式
- 在给出内部实现逻辑之后，可能需要：
  - 将各个操作中公共部分提取出来，形成独立的新操作；
- 操作的形式：
  - **操作的形式：**

    visibility opName (param : type = default, …) : returnType
  - **一个例子: CourseOffering**
    - **Constructor**

      <<class>>new ( )
    - **Set attributes**

      setCourseID (courseID:String)

      setStartTime (startTime:Time)

      setEndTime (endTime:Time)

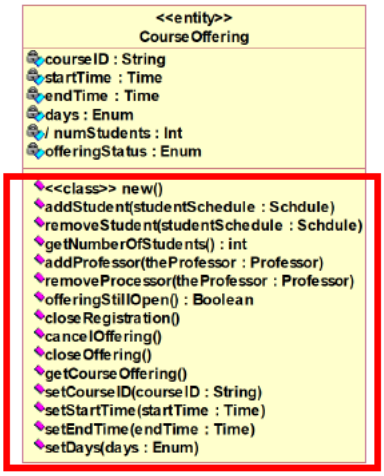
      setDays (days:Enum)
    - **Others**

      addProfessor (theProfessor:Professor)

      removeProfessor (theProfessor:Professor)

      offeringStillOpen ( ) : Boolean

      getNumberOfStudents ( ) : int



- 一个例子：*BorrowerInfo*类
  - 构造函数

    <<class>> + new ( )
  - 属性赋值

    + setName(name:String)

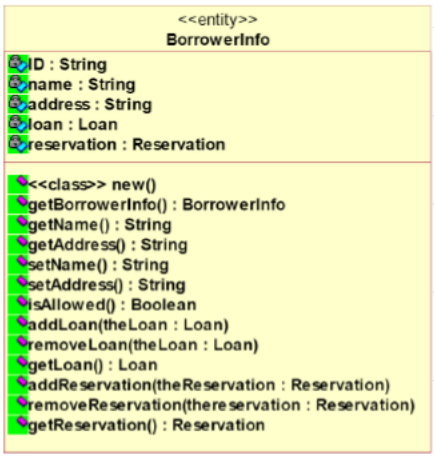
    + setAddress(address:String)
  - 其他

    + addLoan(theLoan:Loan)

    + removeLoan(theLoan:Loan)

    + isAllowed( ) : Boolean

    … …



new ( )

```
offeringStatus := unassigned;
numStudents := 0;
```

addProfessor (theProfessor : Professor)

```
if offeringStatus = unassigned {
   offeringStatus := assigned;
   courseInstructor := theProfessor;
}
else errorState ( );
```

removeProfessor (theProfessor : Professor)

```
if offeringStatus = assigned {
   offeringStatus := unassigned;
   courseInstructor := NULL;
}
else errorState ( );
```

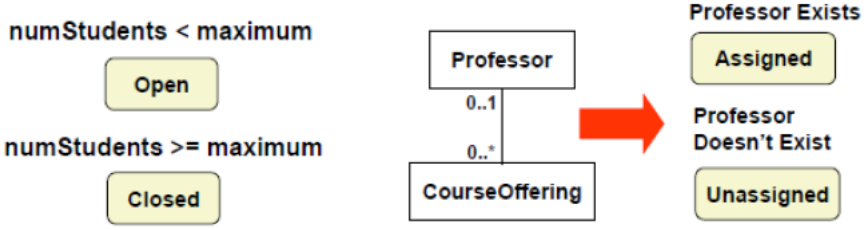
closeOffering ( )

```
switch ( offeringStatus ) {
 case unassigned:
       cancelOffering ( );
       offeringStatus := cancelled;
         break;
 case assigned:
       if ( numStudents < minStudents )
          cancelOffering ( );
          offeringStatus := cancelled;
        else
          offeringStatus := committed;
        break;
 default: errorState ( );
}
```

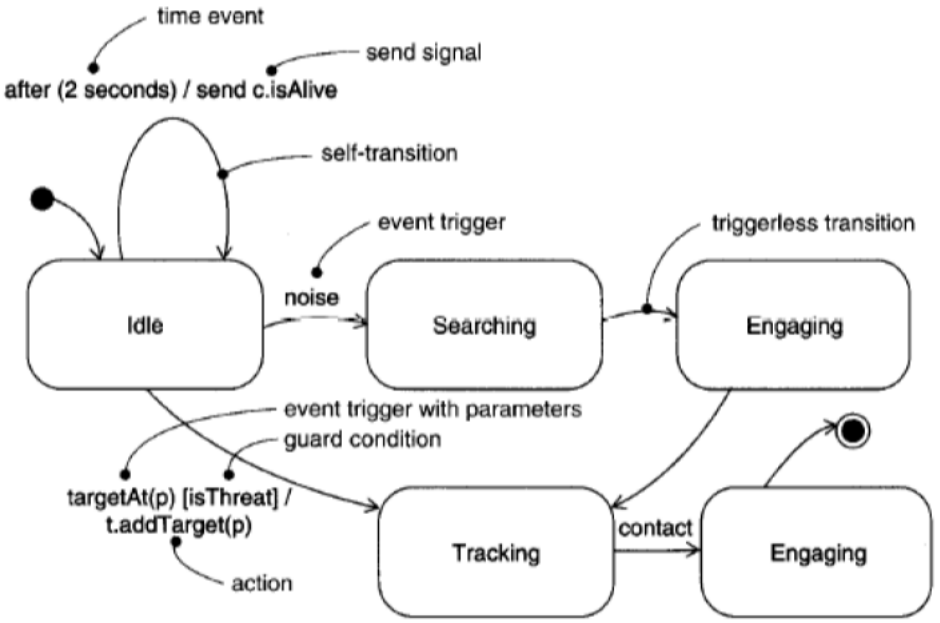
*定义状态

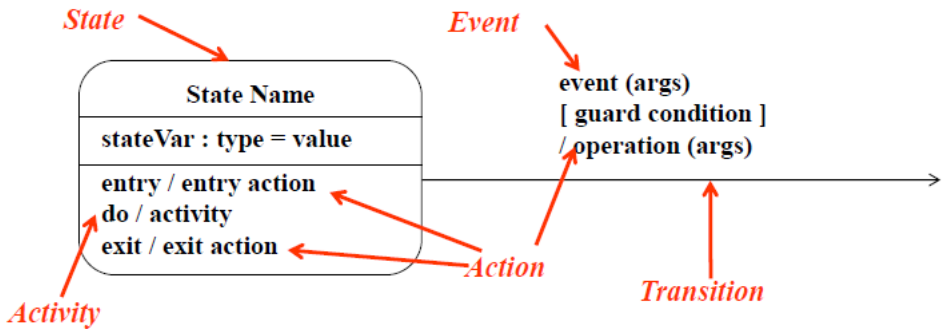
- 目的：
  - 设计一个对象的状态是如何影响其行为；
  - 绘制对象状态图；
- 需要考虑的要素：
  - 哪些对象有状态?
  - 如何发现对象的所有状态?
  - 如何绘制对象状态图?

Example: CourseOffering



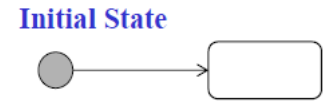
状态图(Statechart Diagram)

- 

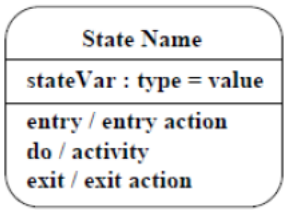
- 状态图描述了一个特定对象的所有可能状态以及由于各种事件的发生而引起的状态之间的转移。
- UML的状态图
  - 主要用于建立类的一个对象在其生存期间的动态行为，表现一个对象所经历的状态序列，引起状态转移的事件(Event)，以及因状态转移而伴随的动作(Action)。
- 状态图适合于描述跨越多个用例的单个对象的行为，而不适合描述多个对象之间的行为协作，因此，常常将状态图与其它技术组合使用。
- 状态图的基本概念
  - 状态(State)
  - 转换(Transition)
  - 事件(Event)
  - 动作(Action)
  - 

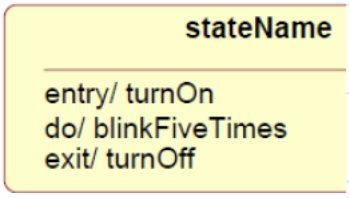
状态(State)

- 定义

    - 一个对象在生命期中满足某些条件、执行一些行为或者等待一个事件时的存在条件。
- 所有对象都具有状态，状态是对象执行了一系列活动的结果。当某个事件发生后，对象的状态将发生变化。
- 事件和状态间具有某种对称性，事件表示时间点，状态表示时间段，状态对应着对象接收的两次事件之间的时间间隔。
- 状态图中定义的状态
    - 初态、终态
    - 中间状态、组合状态、历史状态
    - 一个状态图只能有一个初态，而终态可以有多个
    - 

    - 

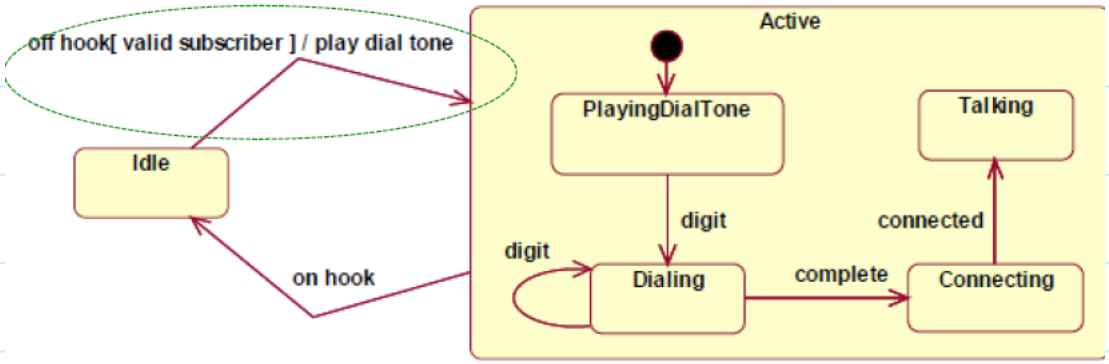
- 中间状态包括两个区域：名字域和内部转移域，如下图所示，其中内部转移域是可选的。
    - 入口动作表达式，entry/action_expression：状态到达/进入时执行的活动。
    - 出口动作表达式，exit/action_expression：状态退出时执行的活动
    - 内部动作表达式，do/action_expression：表示某对象处在某状态下的全部或部分持续时间内执行的活动。（如：复印机卡纸状态下闪烁5次灯）
    - 

    - 

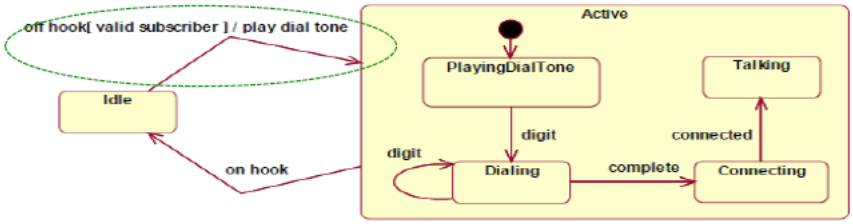
转换(Transition)

- 转换/转移/迁移
    - 转换是状态图的一个组成部分，表示一个状态到另一个状态的移动。
    - 状态之间的转换通常是由事件触发的，此时应在转移上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。

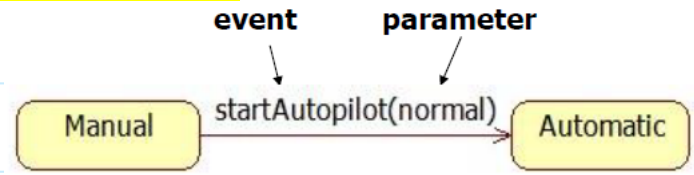


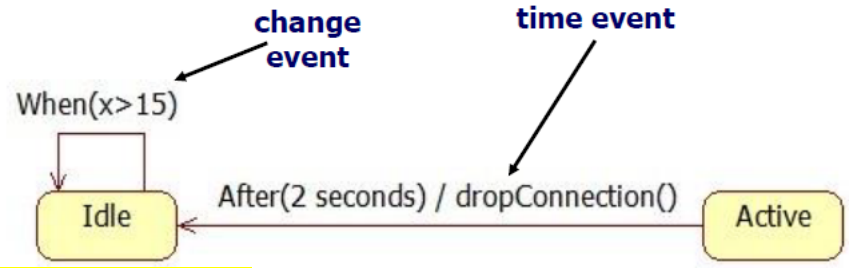
- 转换格式：event - signature[guard - condition]/action
- event - signature格式：
  event - name(comma - separated - parameter - list)
- 保护条件(Guard condition)：可选
  - 一个true或false测试表明是否需要转换
  - 当事件发生时，只有在保护条件为真时才发生进行转换
- “/action”
  - 表示转换发生时执行的动作
  - 同在目标状态的“entry/”动作中进行表达效果相同

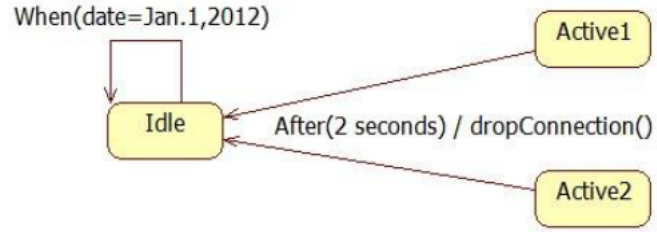


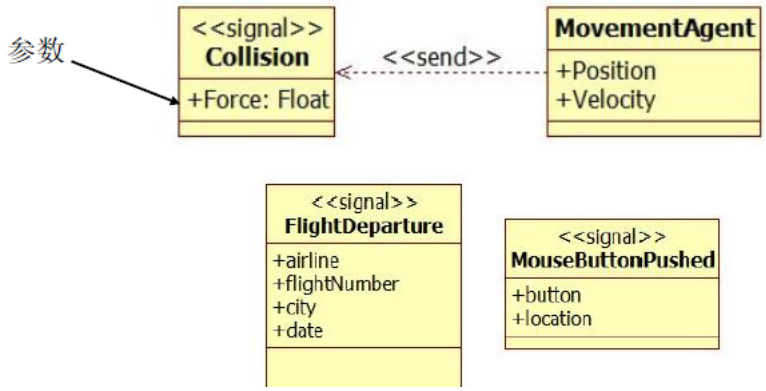
事件(Event)

- 事件是一件有意义、值得关注的现象。
- 事件具有的性质
  - 一个事件的发生是在某一时刻，无持续性；
  - 两个事件是可以相继发生的，也可以是共存的，也可以互不相关的；
  - 多个事件可以组成事件类；
  - 事件具有触发事件动作的对象；
  - 事件在对象间传递信息；
  - 事件包括错误状态(马达被卡住、超时等)和普通事件，二者只是称呼不同
- UML中事件的分类
  - 调用事件(Call event)



- 变化事件(Change event)
  - 由满足布尔表达式而引起的事件。
  - 变化事件意味着要不断测试表达式（实际中并不会连续检测变化事件，但必须有足够频繁的检查）
  - UML采用关键词When，后面跟着用括号括起来的表达式
  - 

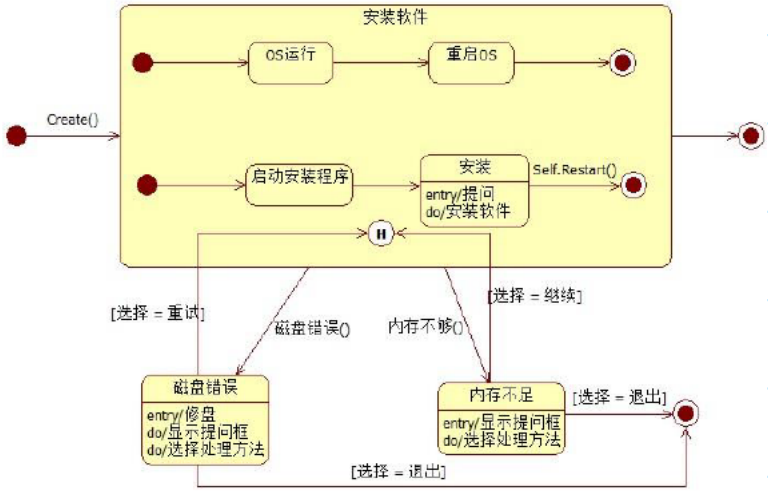
- 时间事件(Time event)
  - 在绝对时间上或在某个时间间隔内发生的事情所引起的事件。
  - 绝对时间用关键字When表示，后面括号中为包含时间的表达式；
  - 时间段采用关键词After表示，后面括号中为计算时间间隔的表达式
  - 

- 信号事件(Signalevent)
  - 发送或者接收信号的事件
  - 更关心信号的接收过程，因为会对接收对象产生影响
  - 一般是异步事件(调用事件一般是同步事件)
  - 信号与信号事件的差别：信号是对象间的消息，信号事件是某时刻发生的事情
  - 采用信号类来表示公共的结构和行为
  - 

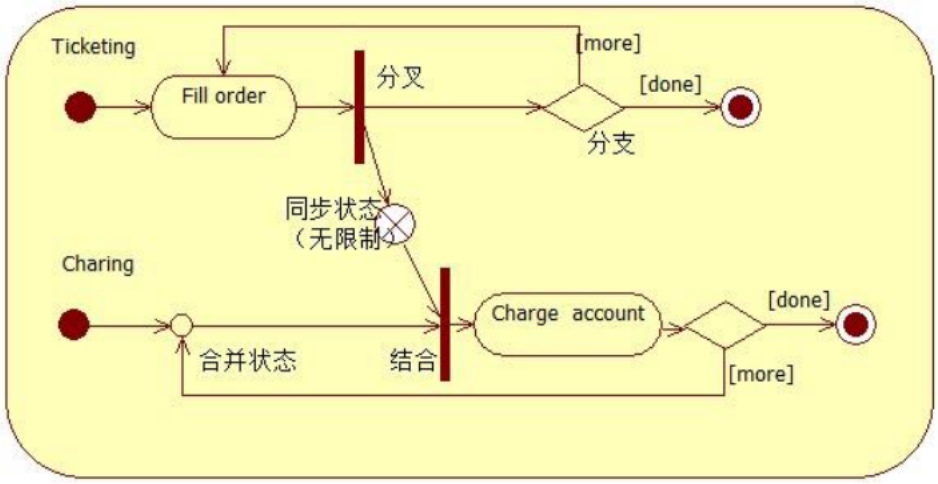
动作(Action)

- 在一个特定状态下对象执行的行为
- 动作是原子的，不可被中断的，其执行时间可忽略不计的。
- 两个特殊的action:entryaction和exitaction
    - Entry动作：进入状态时执行的活动
    - Exit动作：退出状态时执行的活动

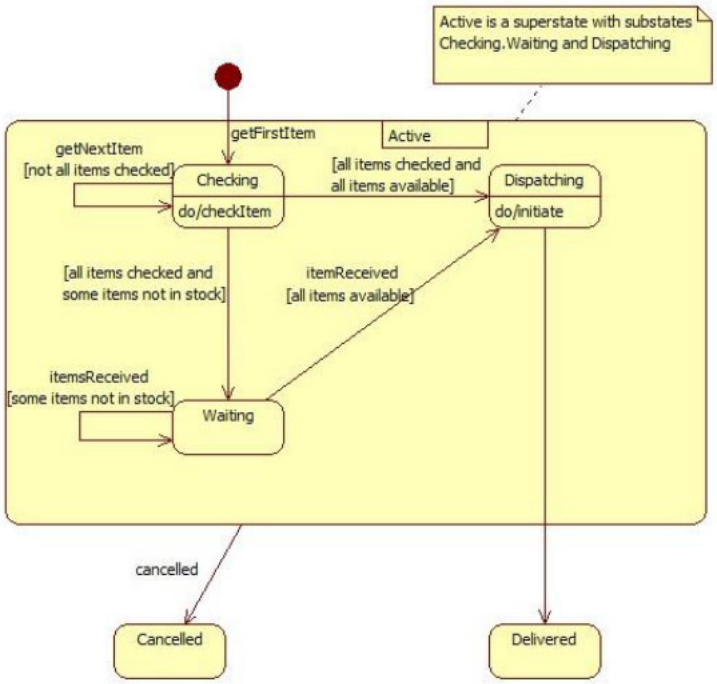
状态图：历史指示器

- 历史指示器被用来存储内部状态。
    - 当对象处于某一状态，经过一段时间后可能会返回到该状态，则可以用历史指示器来保存该状态。
    - 可以将历史指示器应用到状态区。如果到历史指示器的状态转移被激活，则对象恢复到在该区域内的原来的状态。
    - 历史指示器用空心圆中方一个“H”表示。可以有多个历史指示器的状态转移，但没有从历史指示器开始的状态转移。
- 例：软件安装程序
    - 历史指示器被用来处理错误，如“内存溢出”，“磁盘错误”等。
    - 当错误状态被处理完后，用历史指示器返回到错误之前的状态。
    - 

状态图：同步状态

- 

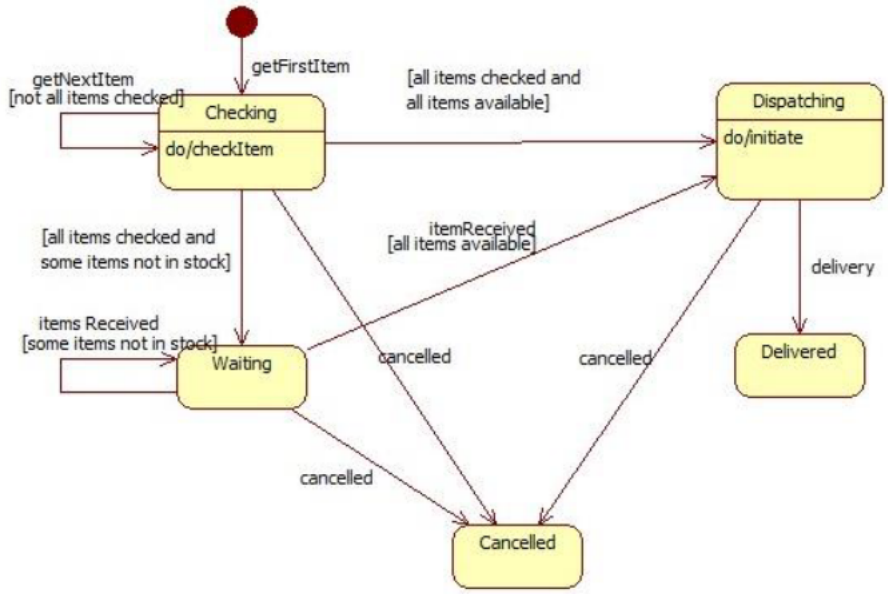
状态图:组合状态

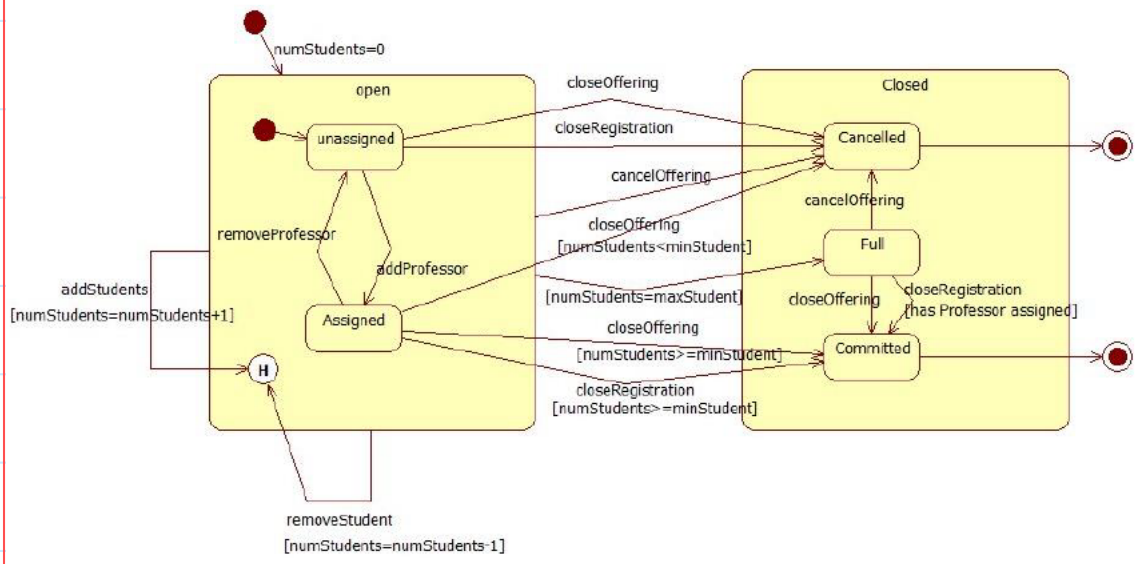
- 

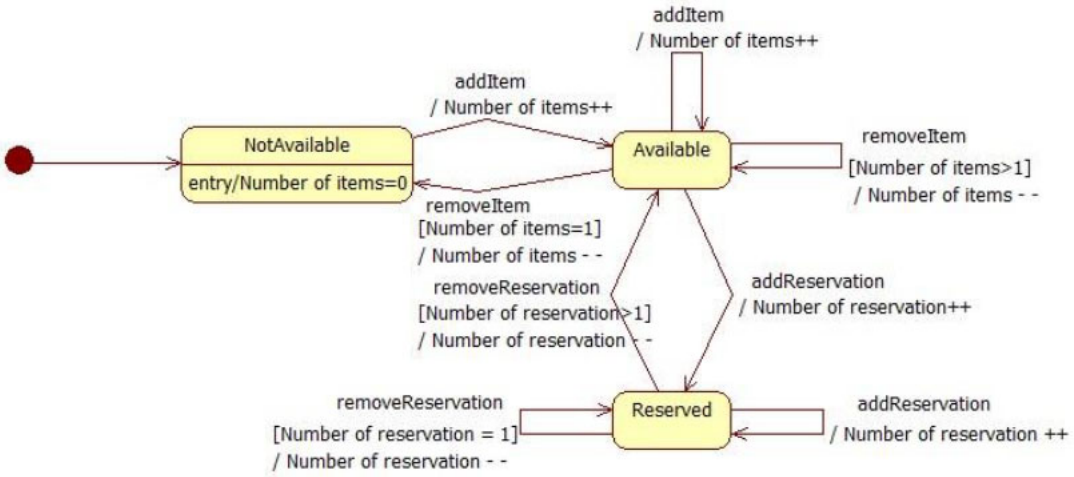
状态图绘制方法

- 检查类图，选择出需要状态图的类(用况或过程、系统、窗口、控制者、事物、设备、突变类型)
  - 状态由属性值表示
  - 通过查看属性的边界值来寻找状态
- 确定状态的转移
  - 确定尽可能多的状态，然后寻找转换。
  - 转换可能是方法调用的结果，经常会反映业务规则。
- 复杂的状态，会有子状态存在。
- 状态图常用于实时系统，记录复杂的类，还会揭示潜在错误条件。

“订单”对象的状态图





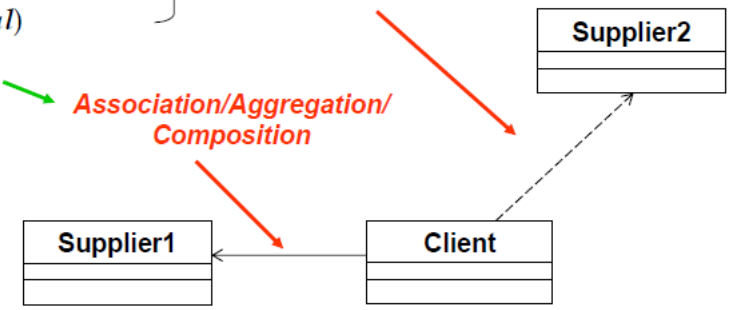


细化类之间的关联/依赖/泛化关系

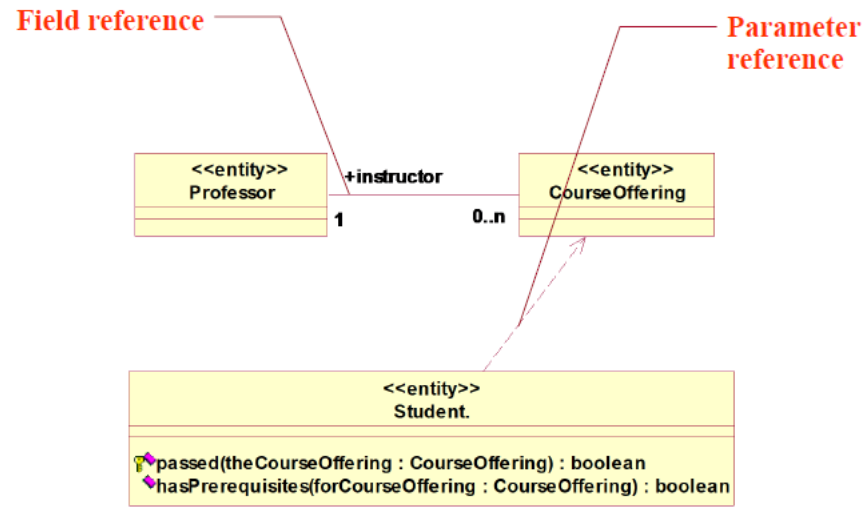
- 细化关系：关联关系、依赖关系、继承关系、组合和聚合关系
- “继承”关系很清楚；
- 在对象设计阶段，需要进一步确定详细的关联关系、依赖关系和组合/聚合关系等。
- 不同对象之间的可能连接：四种情况
  - 全局(*Global*)
  - 参数(*Parameter*)
  - 局部(*Local*)
  - 域(*Field*)

  Dependency

  Association/Aggregation/ Composition

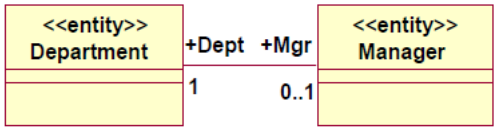


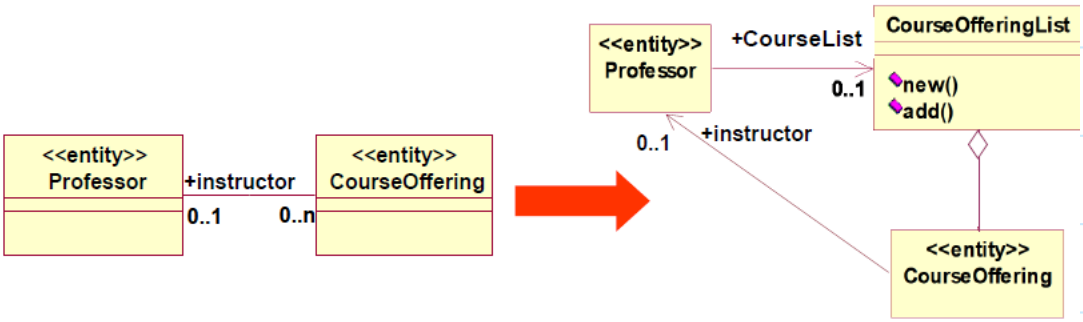
定义类之间的关系：示例

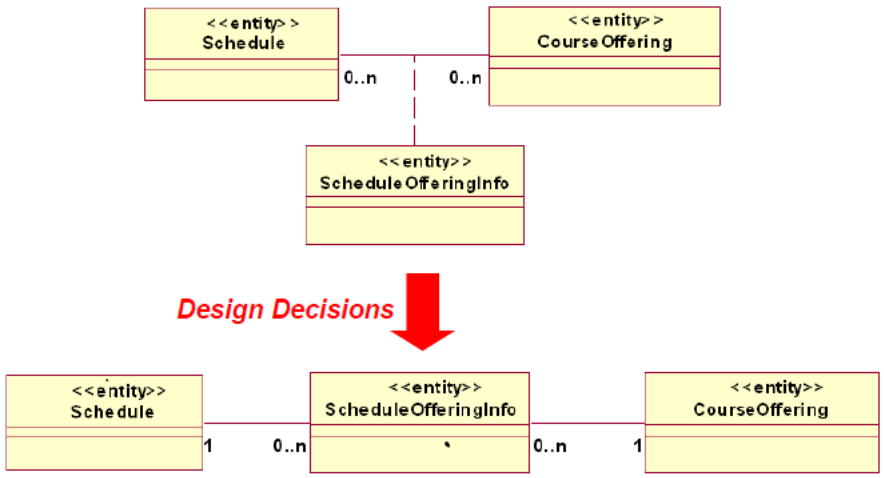


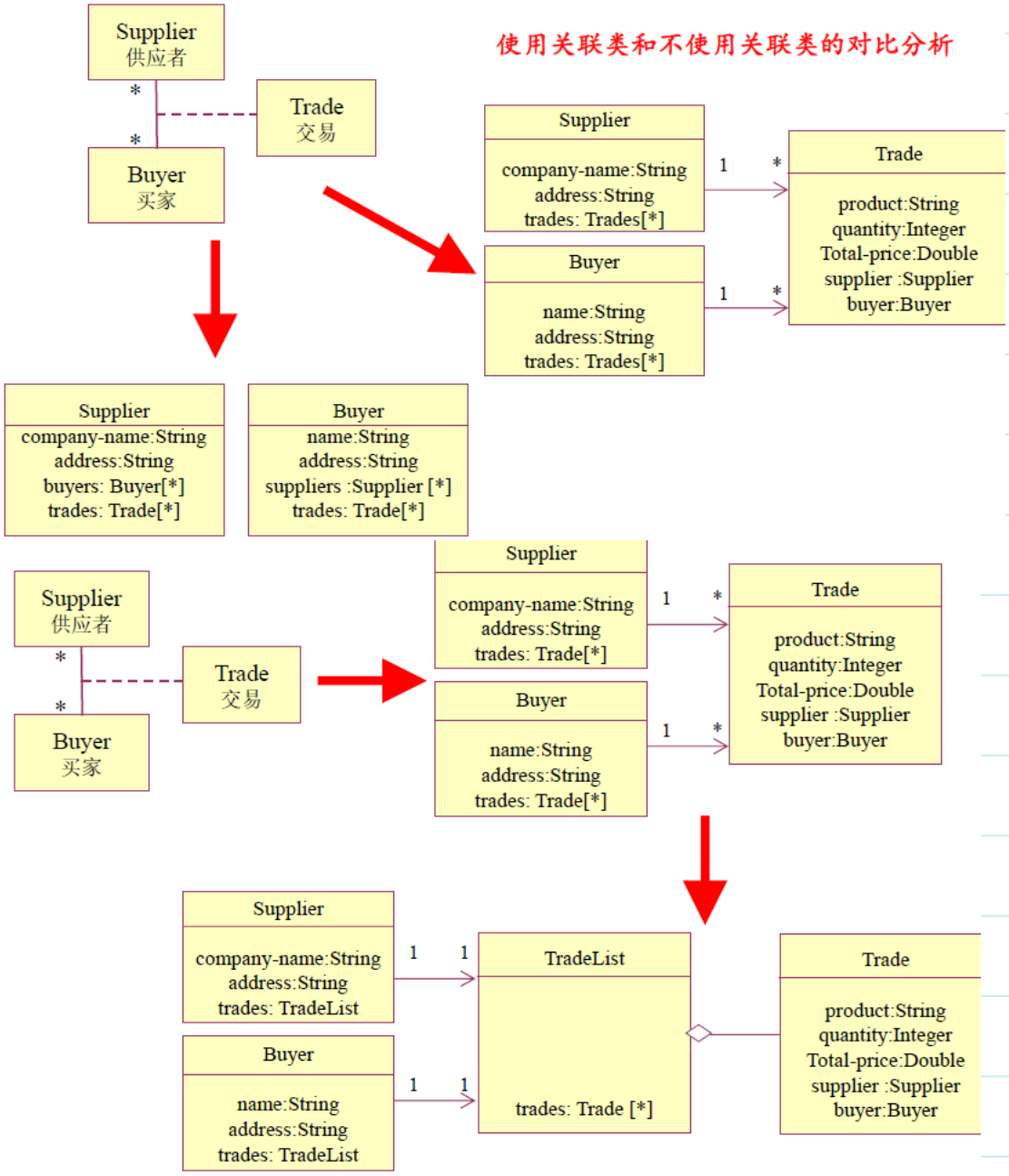
定义关联关系(Association/Composition/Aggregation)

- 根据“多重性”进行设计(multiplicityorienteddesign)
- 情况1：Multiplicity=1或Multiplicity=01
  - 可以直接用一个单一属性/指针加以实现，无需再作设计；
  - 若是双向关联：
    - Department类中有一个属性：+Mgr:Manager
    - Manager类中有一个属性：+Dept:Department
  - 若是单向关联：
    - 只在关联关系发出的类中增加关联属性。

- 

  - 设计阶段需要将这种“关联属性”增加到属性列表中，并更新操作列表
  - 分析阶段则不需要在属性列表中加入“关联属性”
- 根据“多重性”进行设计(multiplicitydesign)
- 情况2：Multiplicity>1
  - 无法用单一属性/指针来实现，需要引入新的设计类或能够存储多个对象的复杂数据结构(例如链表、数组等)。
  - 将1:n转化为若干个1:1。
  - 

- 情况3：有些情况下，关联关系本身也可能具有属性，可以使用“关联类”将这种关系建模。
  - 举例：选课表Schedule与开设课程CourseOffering
  - 

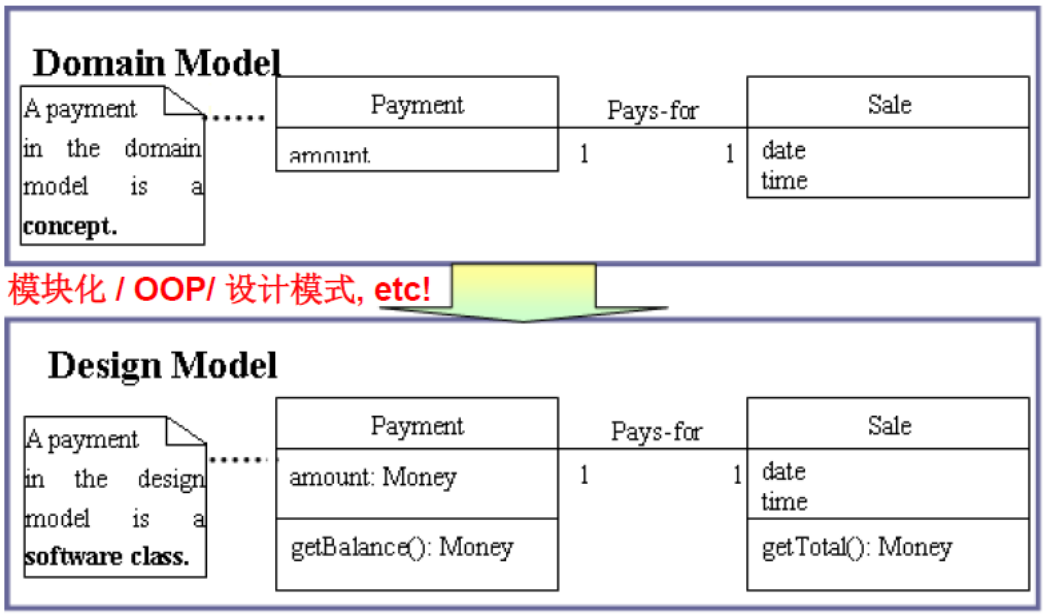


引入“辅助类”简化类的内部结构

- 辅助实体类，是对从用例中识别出的核心实体的补充描述，目的是使每个实体类的属性均为简单数据类型：
  - 何谓“简单数据类型”？编程语言提供的基本数据类型(int,double,char,string,boolean,list,vector等，以及其他实体类)；
  - 目的：使用起来更容易。
- 例如对“订单”类来说，需要维护收货地址相关属性，而收货地址又由多个小粒度属性构成(收货人、联系电话、地址、邮编、送货时间)：
  - 办法1：这五个小粒度属性直接作为订单类的五个属性；——实际上，这五个属性通常总是在一起使用，该办法会导致后续使用的麻烦；
  - 办法2：构造一个辅助类“收货地址”，订单类只保留一个属性，其类型为

该辅助类；
    - 在淘宝系统中，恰好还有用例是“增加、删除、修改收货地址”，故而设置这样一个辅助类是合适的。
    - 这两个实体类之间形成聚合关系(收货地址可以独立于订单而存在)。
- 仍以订单类为例，订单的物流记录是一个非常复杂的结构体，由多行构成，每行又包含“时间(datetime)、流转记录(text)、操作人(text)”等属性，故而可以将“订单物流记录”作为一个实体类，包含着三个简单属性，而订单类中维护一个“订单流转记录(list)”属性，该属性是一个集合体，其成员元素的类型是“订单流转记录”这个实体类。
- 这两个实体类之间形成组合关系(没有订单，就没有物流记录)。
- 当查询订单的流转记录时，使用getXXX操作获得这个list属性，然后遍历每个要素，从中分别取出这三个基本属性即可。

领域类图/分析类图设计类图

- 

6 面向对象设计总结

- 系统设计
    - *包图(packagediagram)逻辑设计
    - 部署图(deploymentdiagram)物理设计
- 对象设计
    - 类图(classdiagram)更新分析阶段的类图，对各个类给出详细的设计说明
    - *状态图(statechartdiagram)
    - *时序图(sequencediagram)使用设计类来更新分析阶段的次序图
    - 关系数据库设计方案(RDBMSdesign)

    - 用户界面设计方案(UIdesign)

关于UML

- 到目前为止，课程所学的OO模型(分析与设计阶段)均采用UML表示；
    - 用例图usecasediagram
    - 活动图activitydiagram
    - 类图classdiagram
    - 时序图sequencediagram
    - 协作图*collaborationdiagram
    - 状态图*statechartdiagram
    - 部署图deploymentdiagram
    - 包图packagediagram
    - 构件图*componentdiagram

状态图使用原则

- 考虑为具有复杂行为的状态依赖对象而不是状态无关对象建立状态图
    - 状态无关对象：对于所有事件，对象的响应总是相同的
    - 状态依赖对象：对事件的响应根据对象的状态或模式而不同。
- 应用领域
    - 一般而言，业务信息系统通常只有少数几个复杂的状态依赖类，状态图使用较少；
    - 过程控制、设备控制、协议处理和通信等领域通常有较多的状态依赖对象，状态图使用较多
- 在建模工具中，状态图不生成代码，但状态图在检查、调试和描述类的动态行为时非常有用。

课堂讨论（需提前准备）：UML的三大源头

- 在UML出现之前，软件工程界对OO分析与设计方法形成了三种不同的流派
    - OMT(ObjectModelingTechnique)方法，由JamesRumbaugh提出
    - Booch方法，由GradyBooch提出；
    - OOSE方法，由IvarJacobson提出；
- 多种方法的并存，导致了各自模型存在差异，沟通变得不畅；
- 1994年开始，三方尝试着三种方法融合在一起，并最终于1997年形成了UML；
    - 统一了Booch、Rumbaugh和Jacobson的表示方法，而且对其作了进一步

的发展，并最终统一为大众所接受的标准OO建模语言。

- 通过查阅资料，向大家分享你对这三种初始方法的认识，分析它们之间存在哪些异同，最终的UML中分别包含了三者的哪些思想。