




哈爾濱工業大學  
HARBIN INSTITUTE OF TECHNOLOGY

软件工程  
第六章 OO分析与设计  
6-2 面向对象的分析

王忠杰  
rainy@hit.edu.cn

2017年10月31日

# 主要内容

- 
- 1 面向对象的分析方法概述
  - 2 静态结构模型
  - 3 动态行为模型
  - 4 面向对象分析的案例



# 1 面向对象的分析方法



# 面向对象的分析

- 分析业务领域，找出问题解决方案，发现对象，分析对象的内部构成和外部关系，建立软件系统的对象模型。
- 着重分析业务领域和系统责任，建立独立于实现的OOA模型，暂时忽略与系统实现有关的问题。
- 主要使用5种图描述完整的系统需求：
  - 用例图
  - 类图
  - 时序图
  - 协作图
  - 状态图



# 面向对象的分析

## ■ 面向对象的分析模型由三个独立的模型构成：

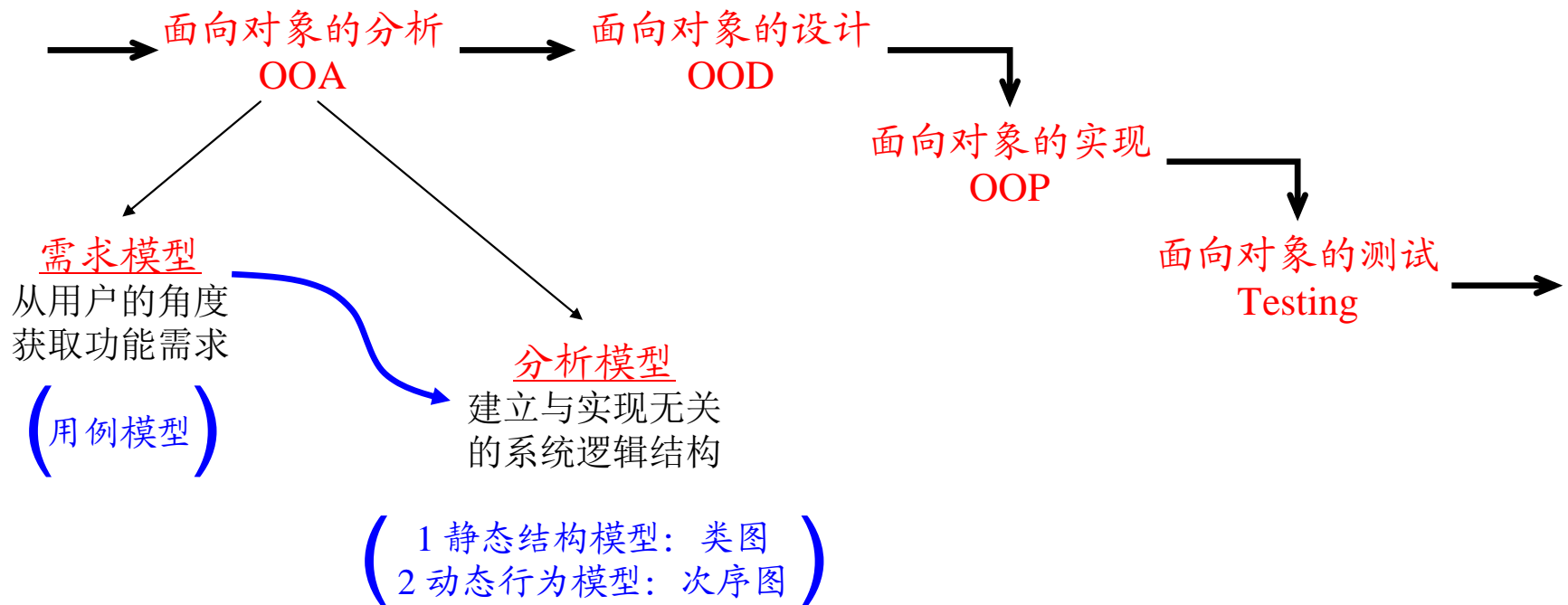
— 功能模型：从用户的角度获取功能需求，由用例模型表示(已在上堂课学习过)；

— 静态结构模型(分析对象模型)：描述系统的概念实体，由类图表示；

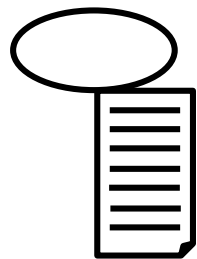
— 动态行为模型：描述对象之间的交互行为，由状态图和顺序图表示。

建立与实现  
技术无关的  
系统逻辑结构

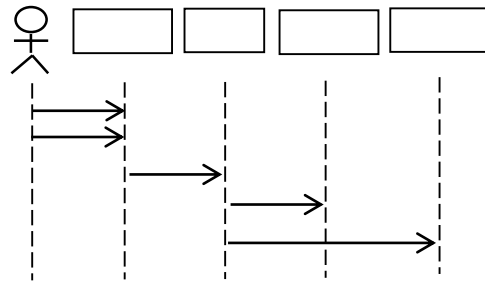
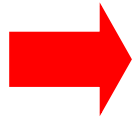
# 面向对象的分析



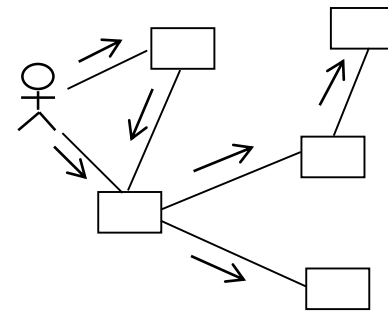
# 面向对象的分析



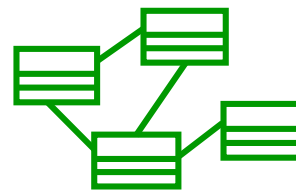
Use Case



Sequence Diagrams



Collaboration Diagrams



Class Diagrams

# 面向对象分析的过程

## ■ 第一阶段：业务领域分析

- 分析应用领域的业务范围、业务规则和业务处理过程，确定系统的责任、范围和边界，确定系统的需求。
- 在分析中需要着重对系统与外部的用户和其他系统的交互进行分析，确定交互的内容、步骤和顺序。

——用例模型



# 面向对象分析的过程

## ■ 第二阶段：发现和定义对象和类

- 识别对象和类，确定它们的内部特征：属性与服务操作。
- 这是一个从现实世界到概念模型的抽象过程，而抽象是面向对象分析的基本原则。

## ■ 第三阶段：识别对象的外部联系

- 在发现和定义对象和类的过程中，需要同时识别对象与对象、类与类之间的各种外部联系，如一般与特殊、整体与部分、实例连接(关联)、消息连接等联系。
- 对象和类是现实世界中的事物的抽象，它们之间的联系也要从分析现实世界事物的各种真实的联系中获得。

# 面向对象分析的过程

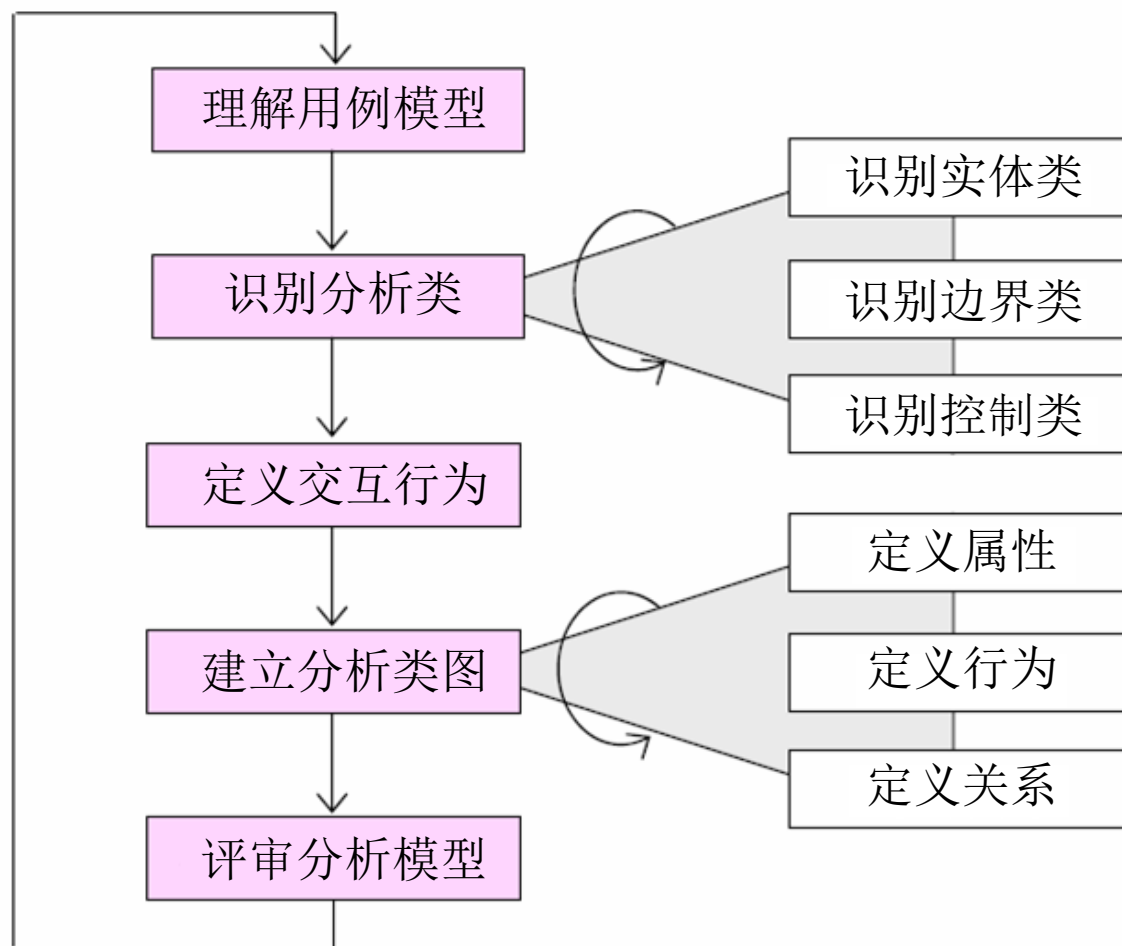
## ■ 第四阶段：建立系统的静态结构模型

- 分析系统的行为，建立系统的静态结构模型，并将其用图形和文字说明表示出来，如绘制类图、对象图、系统与子系统结构图等，编制相应的说明文档。

## ■ 第五阶段：建立系统的动态行为模型

- 分析系统的行为，建立系统的动态行为模型，并将其用图形和文字说明表示出来，如绘制用例图、交互图、活动图、状态图等，编制相应的说明文档。

# 面向对象的分析过程





## 2 建立静态结构模型



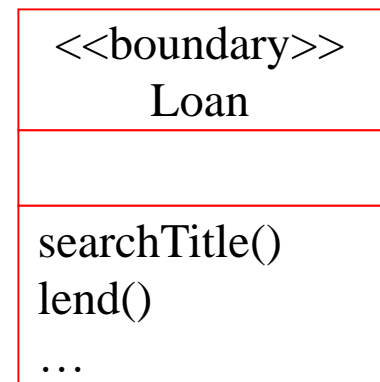
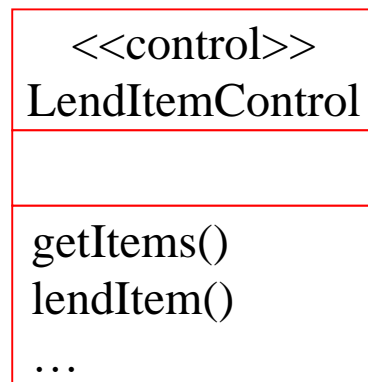
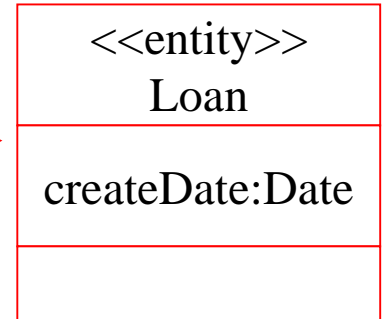
# 建立静态结构模型

- 基本的分析过程：
  - Step 1: 从用例模型入手，识别分析类；
  - Step 2: 描述各个类的属性；
  - Step 3: 定义各个类的操作；
  - Step 4: 建立类之间的关系；
  - Step 5: 绘制类图(class diagram)

# 什么是“分析类”？

- 分析类是概念层次上的内容，用于描述系统中较高层次的对象。
- 分析类直接与应用逻辑相关，而不关注于技术实现的问题。
- 分析类的类型

- 实体类：表示系统存储和管理的永久信息
- 边界类：表示参与者与系统之间的交互
- 控制类：表示系统在运行过程中的业务控制逻辑



# 边界类(Boundary Class)

- 边界类:

- 描述外部的参与者与系统之间的交互
- 目的: 将用例的内部逻辑与外部环境进行隔离, 使得外界的变化不会影响到内部的逻辑部分。
- 类型: 用户界面、系统接口、设备接口

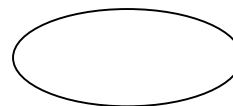
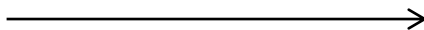
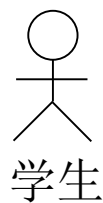
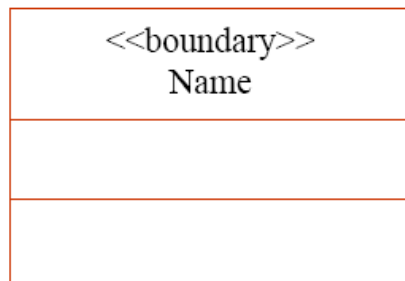
- 对用户界面来说:

- 描述用户与系统的交互信息(传入哪些信息/指令, 传出哪些信息/指令), 而不是用户界面的显示形式(如按钮、菜单等);

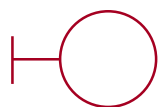
- 对系统接口/设备接口来说:

- 描述通信协议, 但不必说明协议如何实现的。

# 边界类(Boundary Class)



课程注册



“学生注册课程”的界面

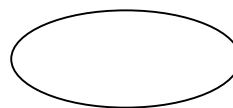
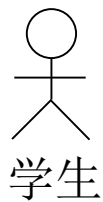
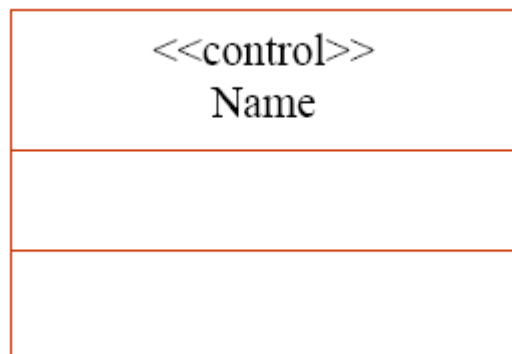


# 控制类(Control Class)

## ■ 控制类

- 描述一个用例所具有的事件流的控制行为，本身并不处理具体的任务，而是调度其他类来完成具体的任务；
- 实现了对用例行为的封装，将用例的执行逻辑与边界和实体进行隔离，使得边界类和实体类具有较好的通用性。

# 控制类(Control Class)



课程注册



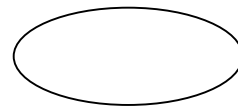
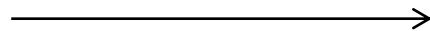
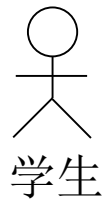
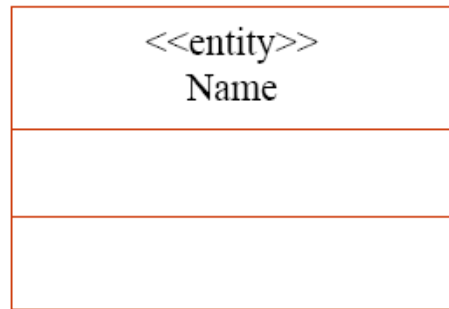
“课程注册” 控制类

# 实体类(Entity Class)

- 实体类

- 描述必须存贮的信息及其相关行为
- 对系统的核心信息建模，通常这些信息需要长久的保存
- 通常对应现实世界中的“事物”

# 实体类(Entity Class)



“课程” 实体类



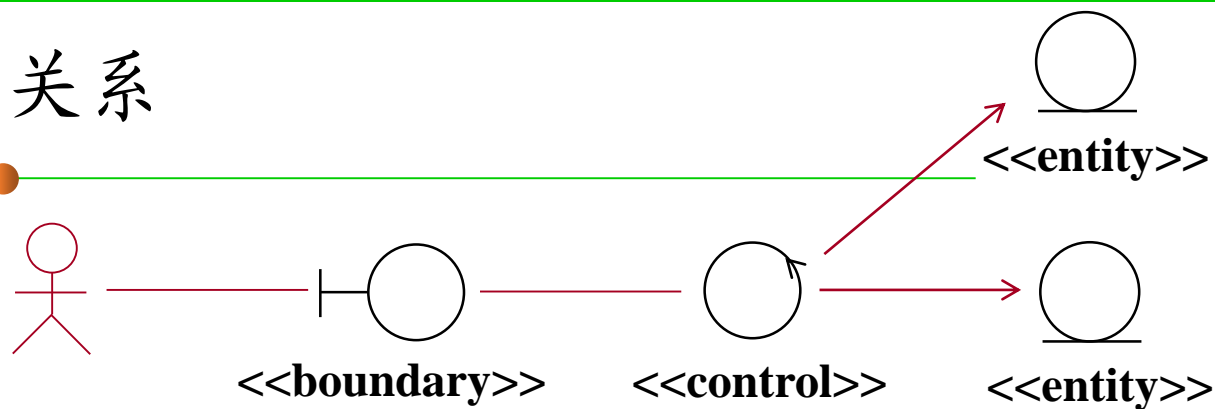
“学生” 实体类



“课程注册信息” 实体类

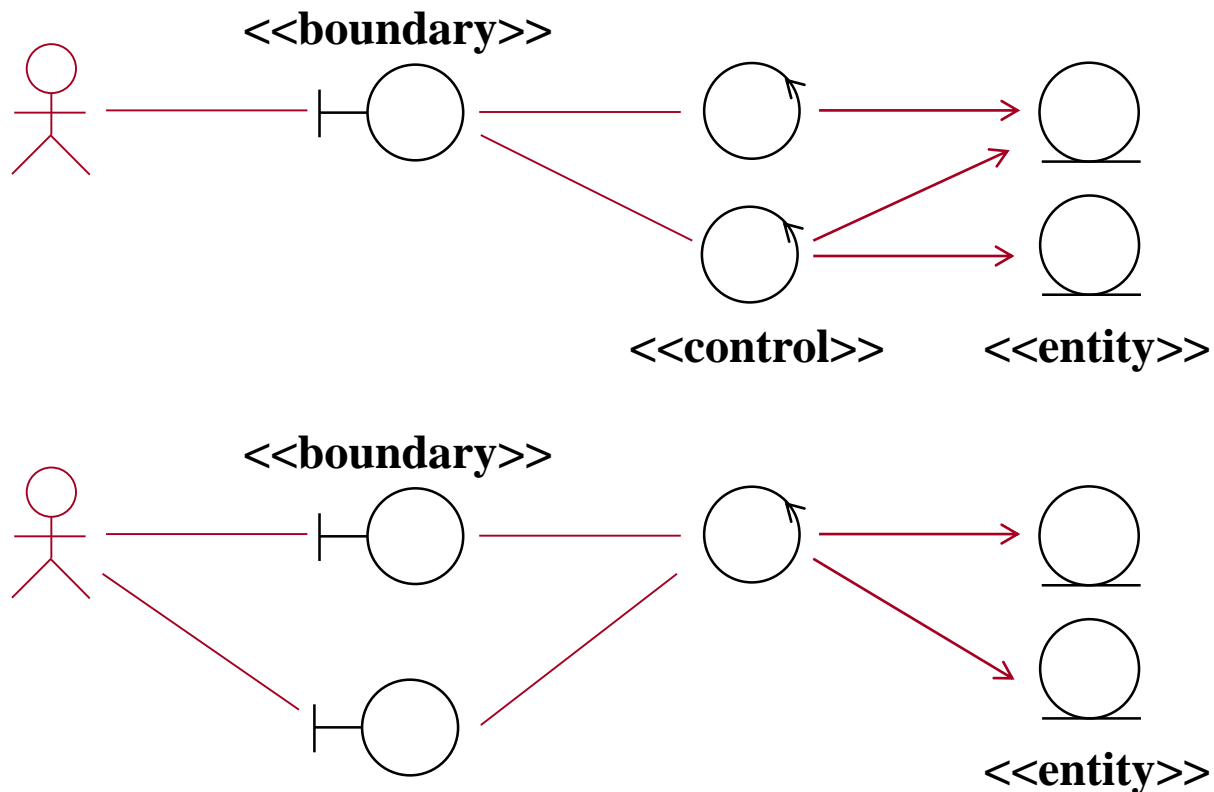
# 三种分析类之间的关系

- 三种分析类之间是“关联”关系 (association);



- 三种分析类之间是多对多(m:n)关系:

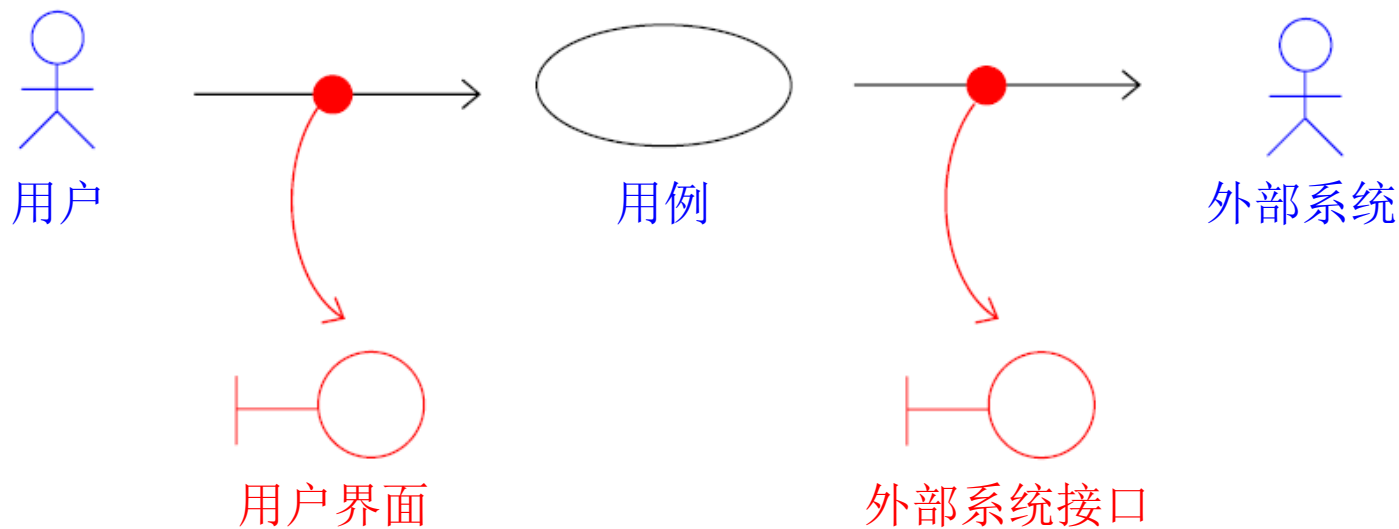
- 一个边界类可以与多个控制类相关联;
- 一个控制类可以与多个边界类相关联、与多个实体类关联;
- 一个实体类可以与多个控制类相关联。



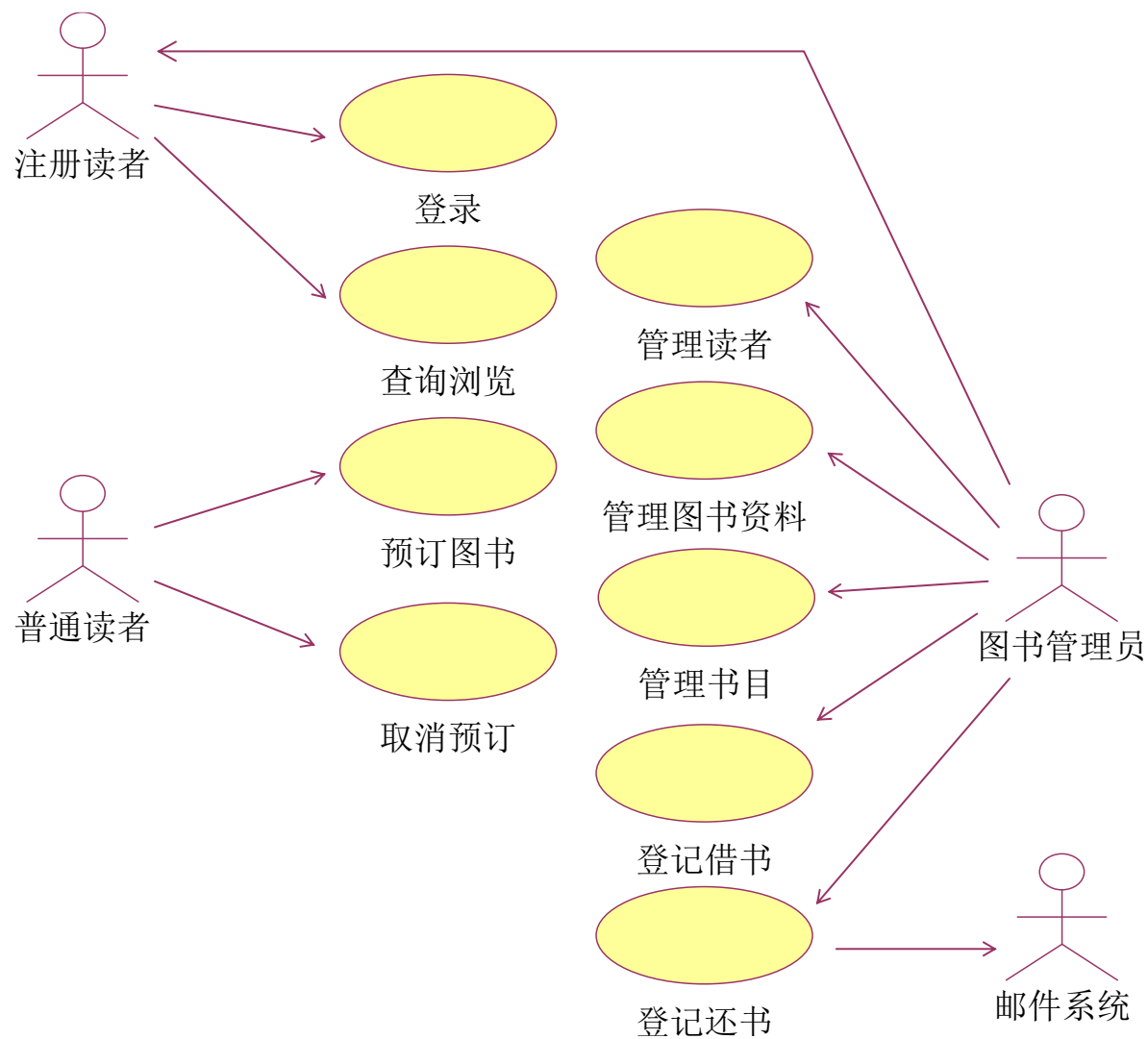
# Step 1: 识别分析类之边界类

## ■ 识别边界类











- 通常，一个参与者与一个用例之间的交互或通信关联对应一个边界类。



# [案例]图书管理系统的用例图



## [案例]图书管理系统的边界类

边界类		说明
LoginForm		注册用户进行登录的操作界面
BrowseForm		注册用户进行查询浏览的操作界面
MakeReservationForm		普通读者预定图书的操作界面
RemoveReservationForm		普通读者取消预定的操作界面
ManageBrowsersForm		图书管理员管理读者的操作界面
ManageTitlesForm		图书管理员管理图书资料的操作界面
ManageItemsForm		图书管理员管理书目的操作界面
LendItemForm		图书管理员登记借书的操作界面
ReturnItemForm		图书管理员登记还书的操作界面
MailSystem		与邮件系统的接口



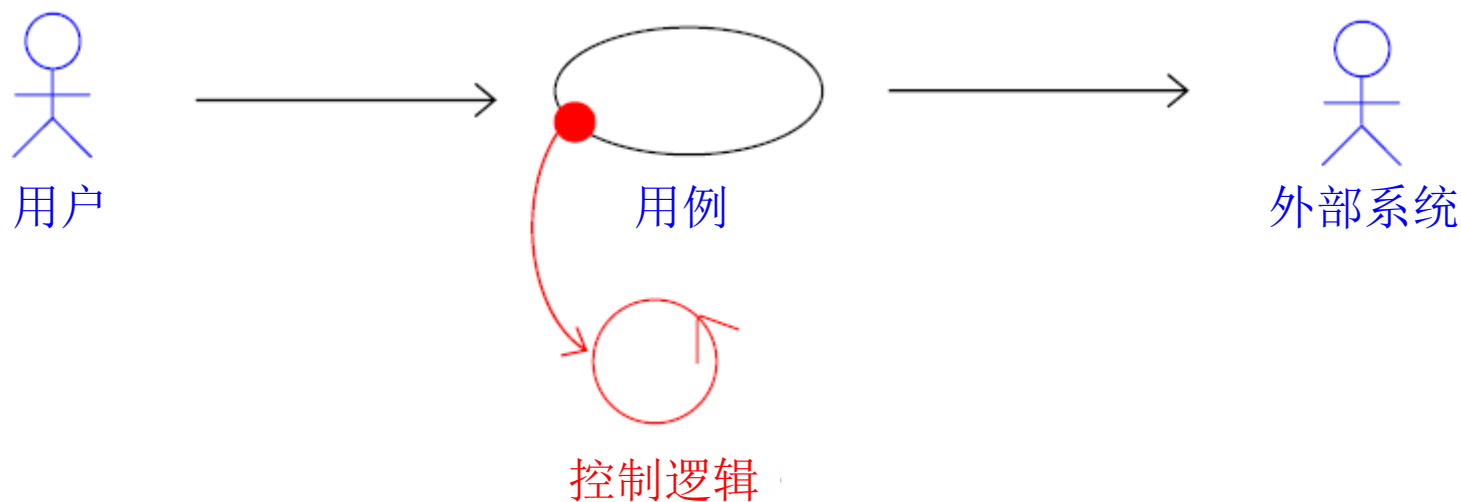
## 识别边界类应当注意的问题

- 边界类应关注于参与者与用例之间交互的信息或者响应的事件，不要描述窗口组件等界面的组成元素；
- 在分析阶段，力求使用用户的术语描述界面；
- 边界类实例的生命周期并不仅限于用例的事件流，如果两个用例同时与一个参与者交互，那么它们有可能会共用一个边界类，以便增加边界类的复用性。

# Step 1: 识别分析类之控制类

## ■ 识别控制类

- 控制类负责协调边界类和实体类，通常在现实世界中没有对应的事物；
- 负责接收边界类的信息，并将其分发给实体类；
- 控制类与用例存在着密切的关系，它在使用例开始执行时创建，在使用例结束时取消。一般来说，一个用例对应一个控制类。



## [案例]图书管理系统的控制类

控制类		说明
BrowseControl		负责执行注册用户的查询浏览
MakeReservationControl		负责执行普通读者的预定图书
RemoveReservationControl		负责执行普通读者的取消预定
ManageBrowsersControl		负责执行图书管理员对读者的管理
ManageTitlesControl		负责执行图书管理员对图书资料的管理
ManageItemsControl		负责执行图书管理员对书目的管理
LendItemControl		负责执行图书管理员登记借书
ReturnItemControl		负责执行图书管理员登记还书

## 识别控制类应当注意的问题

- 当用例比较复杂时，特别是产生分支事件流的情况下，一个用例可以有多个控制类。
- 在有些情况下，用例事件流的逻辑结构十分简单，这时没有必要使用控制类，边界类可以实现用例的行为。
  - 例如：图书管理系统中的“登录”用例
- 如果不同用例包含的任务之间存在着比较密切的联系，则这些用例可以使用一个控制类，其目的是复用相似部分以便降低复杂性。
  - 通常情况下，应该按照一个用例对应一个控制类的方法识别出多个控制类，再分析这些控制类找出它们之间的共同之处。

# Step 1: 识别分析类之实体类

- 如何从用户的需求陈述中找到“实体类”？

## ——名词驱动的认识方法

- 实体类通常是用例中的参与对象，对应着现实世界中的“事物”
- 对用户需求陈述进行“语法分析”，找出所有的名词或名词短语，对其标注下划线；
- 判断一个“名词”是否为实体类，其标准是：
  - 系统是否需要管理该名词所拥有的信息？
  - 系统是否需要管理该名词所能发出或接受的动作？
- 合并同义词；
- 将最终得到的每一个名词映射为一个实体类；
- 动词映射为类的操作，形容词/名词映射为类的属性。

# [案例]图书管理系统的实体类

边界类		说明
BrowserInfo	○	普通读者的基本信息
Loan	○	普通读者的借书记录
Reservation	○	普通读者的预定信息
Title	○	图书资料的基本信息
Item	○	书目
(由于图书资料中包括书籍和杂志等类型，因此可以进一步划分子类)		
BookItem	○	书籍的基本信息
MagazineItem	○	杂志的基本信息

## 识别实体类应当注意的问题

- 实体类的识别质量在很大程度上取决于分析人员书写文档的风格和质量；
- 自然语言是不精确的，因此在分析自然语言描述时应该规范化描述文档中的一些措辞，尽量弥补这种不足；
- 在自然语言描述中，名词可以对应类、属性或同义词等多种类型，开发人员需要花费大量的时间进行筛选。

## 注意：actor == 实体类？

- “角色” (actor)是否一定是实体类？—— NO
- 除非系统需要在各用例中管理和维护该角色的信息(不是指ID和密码)，否则只需将其作为actor，无需作为实体类。
- 例如：
  - 选课系统中：教学秘书仅属于actor，无需作为实体类；但学生和老师既是actor也是实体类，因为课程/课表/选课单等实体中均需要使用学生和老师的信息；
  - 淘宝系统中：支付宝/物流系统/淘宝平台管理员均为actor。
- 对用户(actor)的权限管理，是任何系统均具备的通用功能，无需在实体类中考虑。将来放在基础设施层实现即可。



## Step 2: 描述分析类的属性之边界类、控制类

- **对UI类型的边界类:**

- 需要actor输入的各数据;
- 系统反馈给actor的各数据;
- 需要临时保存的、用于在边界类和控制类之间传递的临时数据;

- **对API类型的边界类:**

- 需要向外部系统(软件/硬件)传递的数据;
- 需要从外部系统(软件/硬件)接收的数据;

- **控制类的属性:**

- 从UI接收的数据;
- 为进行事件流执行所需的临时数据;
- 需要调用的实体类;
- 经过计算之后、需要发送给UI的数据。

## Step 2: 描述分析类的属性之实体类

- 基本属性：
  - 按照一般常识，找出实体类的基本属性；
  - 考虑对象需要系统保存的信息，找出对象的相应属性；
- 状态属性：识别对象需要区别的状态，考虑是否需要增加一个属性来区别这些状态；
- 关联属性：确定属性表示整体与部分结构和实例连接；
- 派生属性：通过计算其他属性的值所得到的新属性。

# 细化属性

## ■ 细化属性

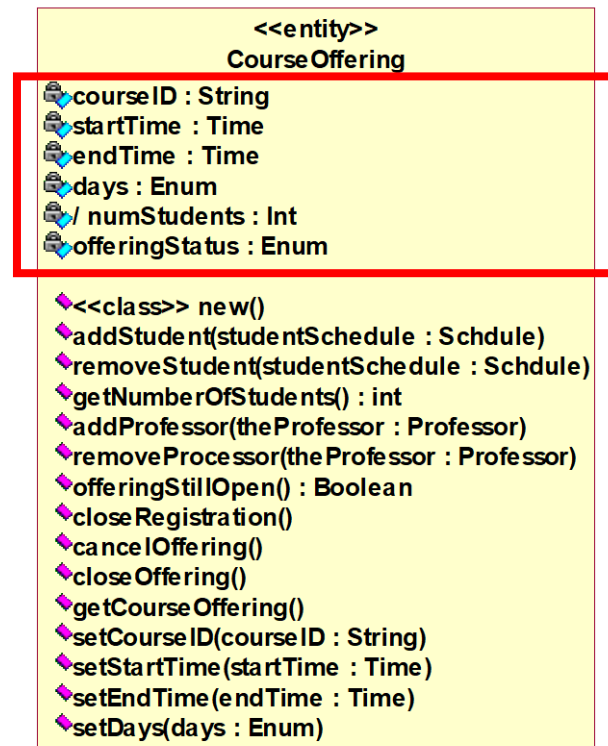
- 具体说明属性的名称、类型、缺省值、可见性等

**visibility attributeName : Type = Default**

- Public: '+';
- Private: '-'
- Protected: '#'

## ■ 属性的类别:

- 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。



# 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“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。

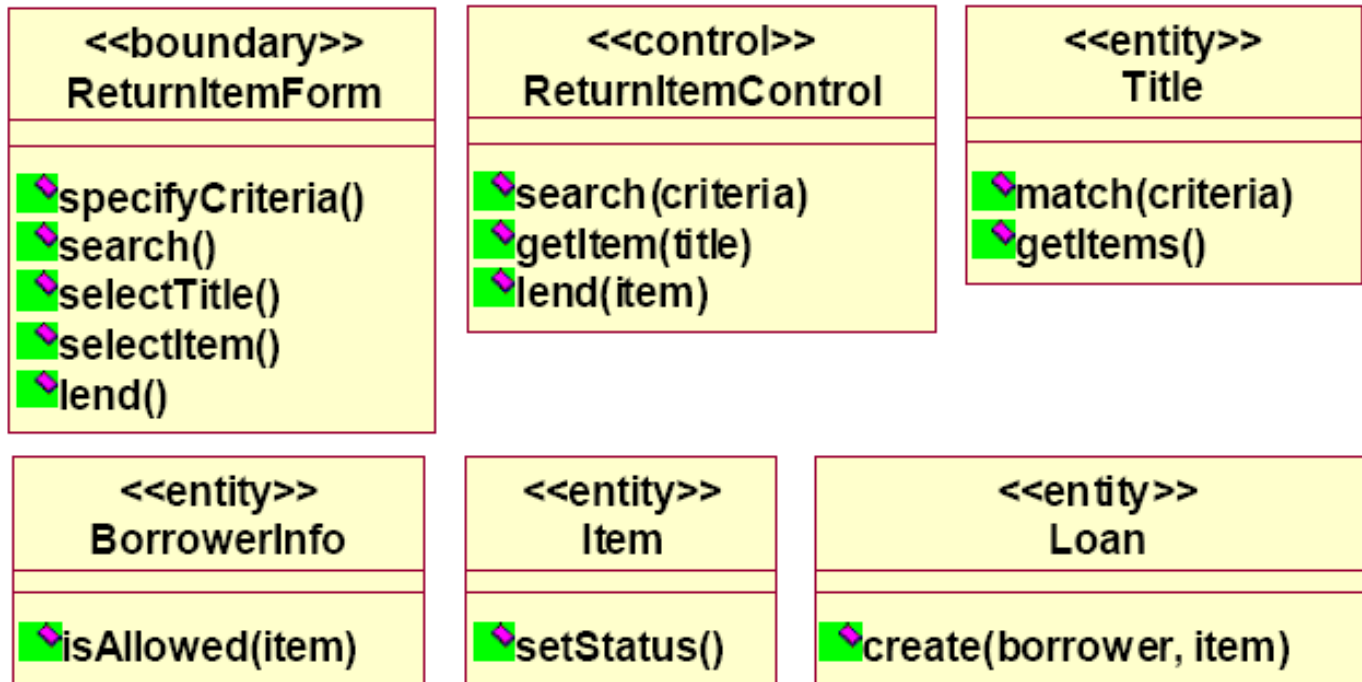
# 细化属性

## ■ 基本原则

- 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。

## Step 3: 定义分析类的操作

- 将用例行为分配到相应的分析类之后，系统的一些分析类具有相应的职责。



## Step 3: 定义分析类的操作

### ■ 边界类的操作:

- 提供给用户的、可在UI上进行的各类操作;
- 对从控制类返回的数据进行各类临时处理而进行的操作;
- 提供给其他系统的API;

### ■ 控制类的操作:

- 对从边界类接收到的数据进行各类临时处理而进行的操作;
- 向实体类所发出的调用操作;
- 对从实体类接收到的数据进行临时处理而进行的操作;

### ■ 实体类的操作:

- 对属性进行CRUD的操作;
- 对状态进行更新的操作;
- 辅助操作。



## 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，充分考虑类的“属性”与“状态”是否被充分利用起来：
  - 对属性进行CRUD；
  - 对状态进行各种变更；

# 细化操作

- 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；
- 详细说明操作的内部实现逻辑。
  - 采用伪代码或程序流程图的方式；
- 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；

# 细化操作

## ■ 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

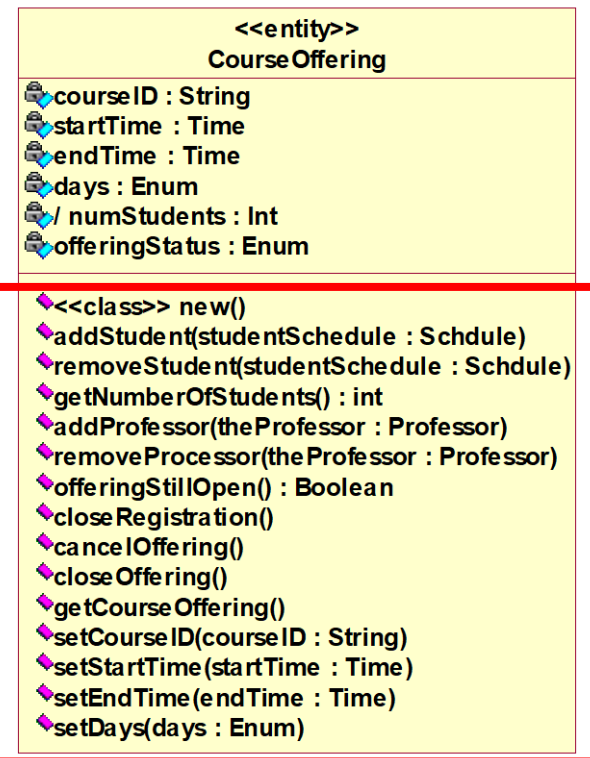
*addProfessor ( theProfessor:Professor )*

*removeProfessor ( theProfessor:Professor )*

*offeringStillOpen ( ) : Boolean*

*getNumberOfStudents ( ) : int*

... ..



# 细化操作

## ■ 一个例子: *BorrowerInfo*类

### — 构造函数

**<<class>> + new ( )**

### — 属性赋值

**+ setName( name:String)**

**+ setAddress( address:String)**

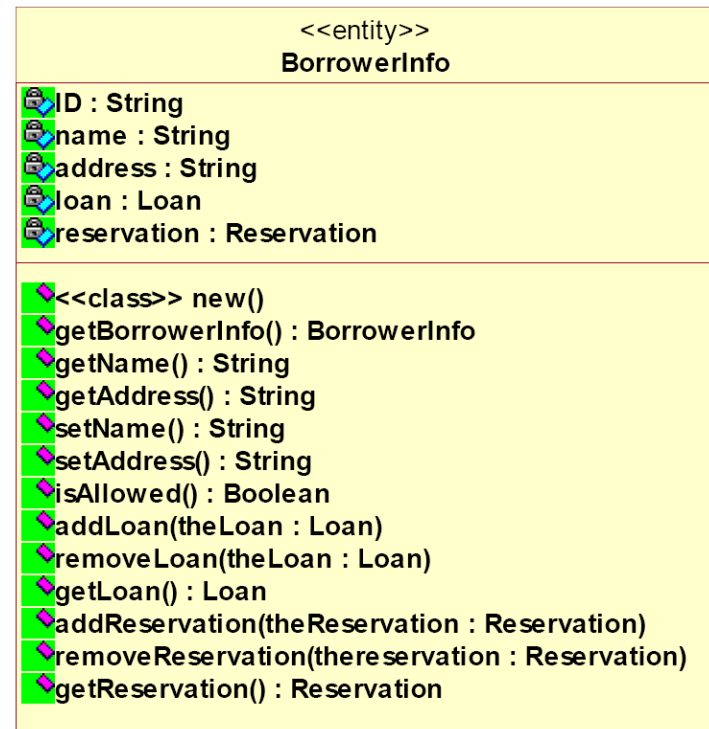
### — 其他

**+ addLoan( theLoan:Loan)**

**+ removeLoan( theLoan:Loan)**

**+ isAllowed( ) : Boolean**

... ..



# 细化操作

*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 ( );  
}
```

## 从用例出发，识别类的操作

- 用例的事件流描述了一系列的“动作”：actor做的、系统自身做的。
- 如何把这些动作映射到分析类的操作中？简要归纳一下原则：
  - actor自己做的动作，都是边界类的操作，与控制类和实体类无关；
  - 边界类的操作会调用控制类的操作，并进一步被分解为对各个不同实体类的操作的调用序列。
  - actor发出的动作通常都比较大，往往涉及到对多个实体类的属性的增删改查(CRUD)，不能粗暴的将它直接放置在某个实体类中。
- 实体类的操作分为两类：
  - 对该实体的生命周期的操作：如new()、destroy()；
  - 对该实体内部的一个或多个属性的CRUD操作；——注意，是自己的属性，不是其他实体的！
- 因此，对用例中actor所发出的每个动作，需要仔细分析它对哪些实体类的哪些属性做CRUD。

# 从用例出发，识别类的操作

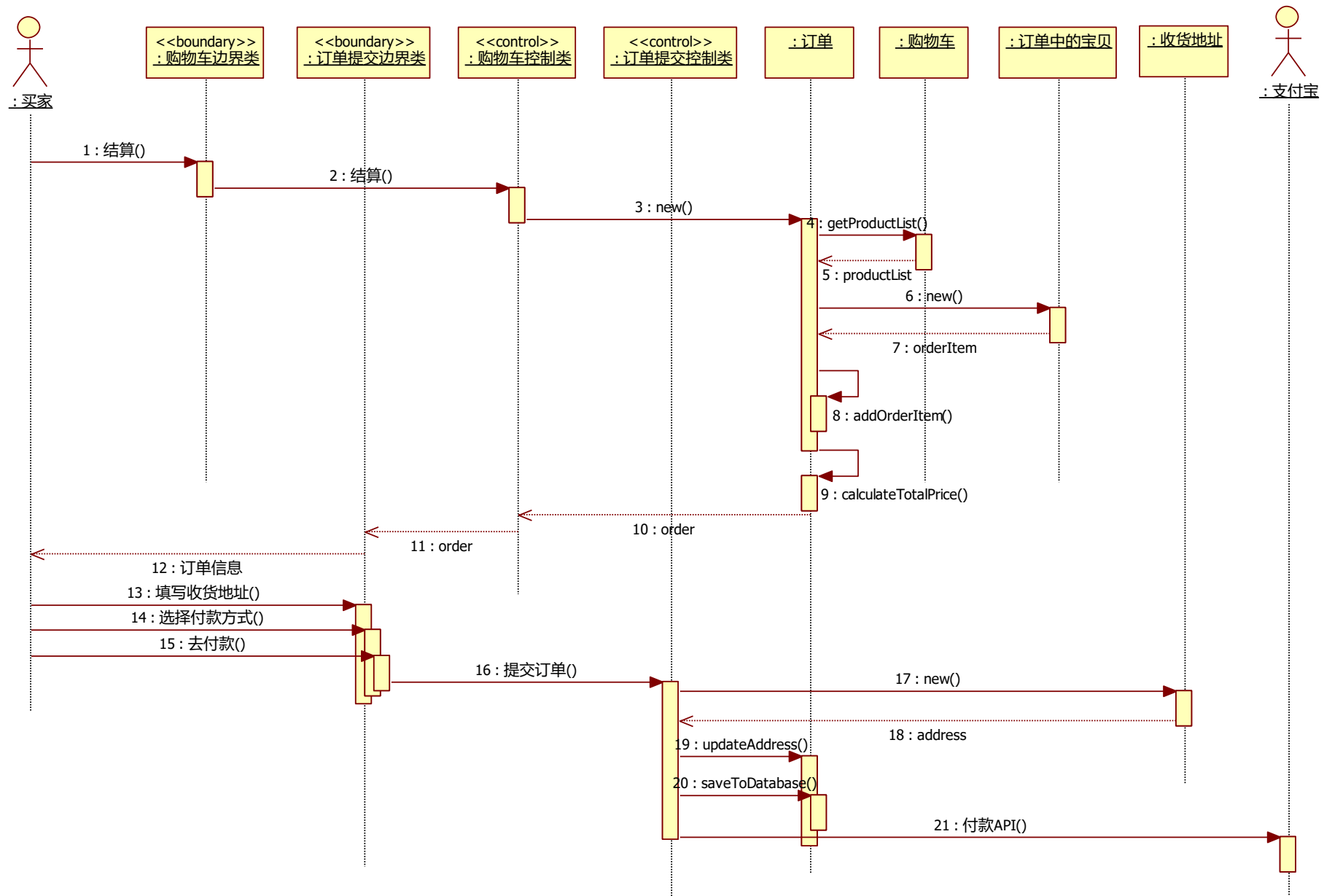
## ■ 例如：用例“从购物车生成订单”

- 事件流：1 用户点击“去结算”；系统根据当前购物车对象中包含的宝贝信息，生成一张订单；2 用户填写订单的收货地址、付款方式等信息，点击“去付款”，系统永久存储这张订单，引导用户到达支付宝页面。
- 用例中涉及的实体类有：购物车、宝贝、收货地址、订单、订单项；
- 为了完成上述事件流，“从购物车生成订单”控制类逐步调用这些实体类的多个操作 (此刻程序运行空间中必然已经有一个“购物车”类的示例对象cart)：
  - “订单”类的新操作：Order newOrder = new Order(cart)，它从cart对象中读取到包含的宝贝列表，形成一个新的订单对象newOrder；
  - 为此，“购物车”对象需要提供一个操作供订单类的新操作所使用：getProductList()，返回购物车对象中包含的所有产品(来自购物车类的关联属性ProductList，这是一个数组，由0到多个“宝贝”类构成)；
  - 获得产品信息之后，订单对象还需调用多次“订单项”类的新操作，构造多个订单项对象，并与自己聚合起来(需要一个操作addOrderItem)；

## 从用例出发，识别类的操作

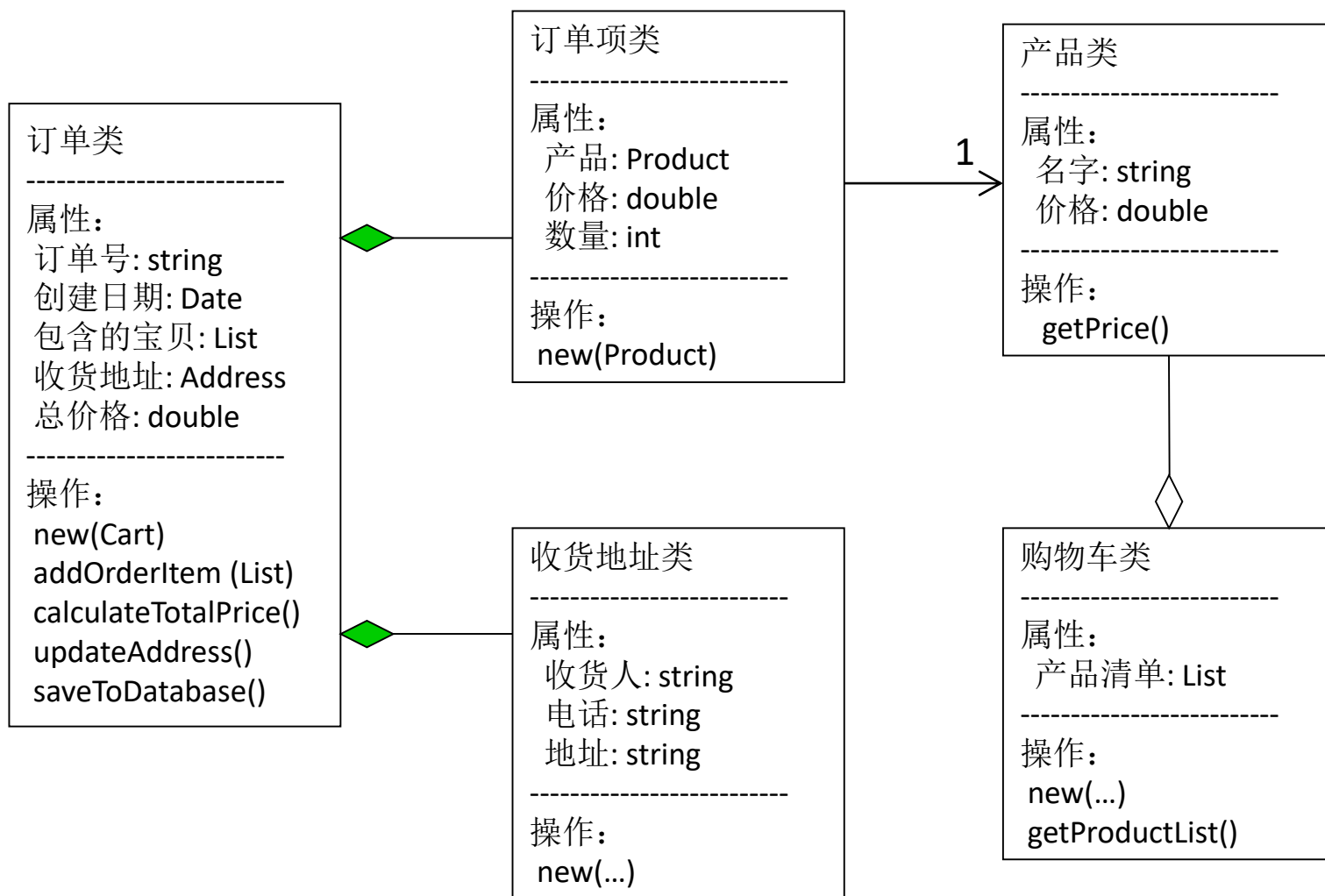
- 接下来，控制类还需要根据订单包含的产品，计算总价格并展示。总价格是“订单”类的一个派生属性，订单类需要一个操作`calculateTotalPrice()`，该操作会读取订单对象中聚合的多个订单项对象并加以计算，将结果更新到订单类的“总价格”属性中。
- 经过上述步骤之后，控制类才算真正构造完一个订单对象；
- 接下来，当用户填写完收货地址后，需要首先调用“收货地址”类的`new`操作，根据用户填写的信息构造一个收货地址对象`address`；
- 调用“订单”对象的操作：`updateAddress(address)`，将刚才构造的`address`传递进去，更新订单对象`order`的属性`address`(其类型就是“收货地址”类)；
- 当用户点击付款时，意味着需要将该订单进行持久化存储，故需要调用“订单”类的另一个操作：`saveToDatabase()`，该操作将对象中保存的所有属性信息更新到关系数据表中；
- 同时，系统调用支付宝的API，触发支付宝的行为。该行为就不需要刻画了，因为它属于外部系统，通过API调用即可。





# 从用例出发，识别类的操作

- 汇总一下，从这个用例中得到的实体类操作和属性是这样的：



# 从用例出发，识别类的操作

- 但是，这几个实体类的属性和操作并不完整。
  - 继续考虑其他每一个用例，在其他用例中，类似的会发现其他所需要的属性和操作，继续加入到实体类中。
  - 当考虑完所有的用例之后，实体类的属性集和操作集就完整了。
- 判断类的“操作”设计是否合理的标准：
  - 对该实体类的任何一个属性的CRUD，均有特定的操作加以完成（而不是将属性的可见性设定为public，由使用者直接读取和更新操作的值）；
  - 若一个操作CRUD的对象不是该实体类的属性，那么该操作不该出现在这里，把它移到相应的实体类中去；
  - 若你发现某个操作op会CRUD多个实体类的属性值，通常意味着这个操作需要分解为多个小粒度操作( $op_1, op_2, \dots$ )，每个 $op_i$ 放在它专门的实体类中。此时，op本身就不会成为某个实体类的操作，而通常将它放在某个控制类中，由该控制类分别来调用 $op_1, op_2, \dots$ 。
    - “从购物车生成订单”这个操作，就分解为了“订单”、“购物车”、“订单项”、“收货地址”这四个实体类的多个小操作。

## 从用例出发，识别类的操作

- 有一类用例中的操作，没办法分解为小粒度的op，尤其以查询类操作为甚。
- 例如：“用户通过关键字查询宝贝”，返回的是一组宝贝。这个操作绝不会是“宝贝”类的操作，因为任何一个“宝贝”对象都不可能包含淘宝上的全体宝贝。
- 所以，这个操作只能由控制类加以完成：
  - 根据用户提供的关键字，直接到数据库中检索出满足条件的宝贝清单；
  - 基于数据库中的每条宝贝记录，使用“宝贝”类的new操作构建一个“宝贝”对象，形成一个list，返回给相应的边界类，边界类对其加以展示。
- 另一个办法：在领域模型中增加一个实体类“宝贝清单”，其中包含了所有的“宝贝”对象。这样的话，“通过关键字查询”就成了该实体类的一个操作。
  - 问题在于：需要在系统初始化时就将该实体类构造出来，否则无法被使用。这明显是不可能的，占用内存空间太大。

## \* 定义状态

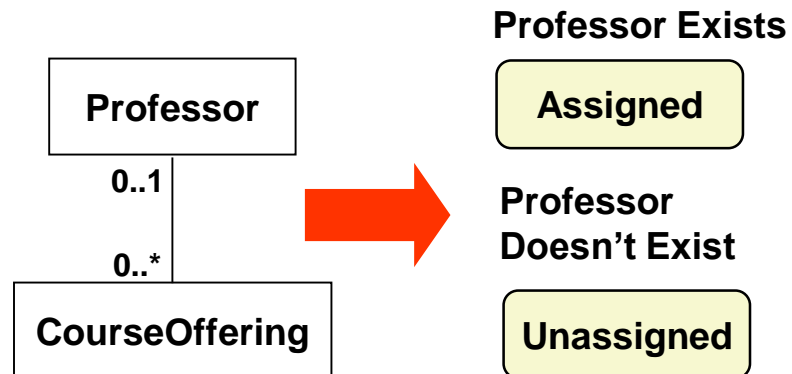
- 目的：
  - 设计一个对象的状态是如何影响其行为；
  - 绘制对象状态图；
- 需要考虑的要素：
  - 哪些对象有状态？
  - 如何发现对象的所有状态？
  - 如何绘制对象状态图？
- Example: CourseOffering

**numStudents < maximum**

Open

**numStudents >= maximum**

Closed



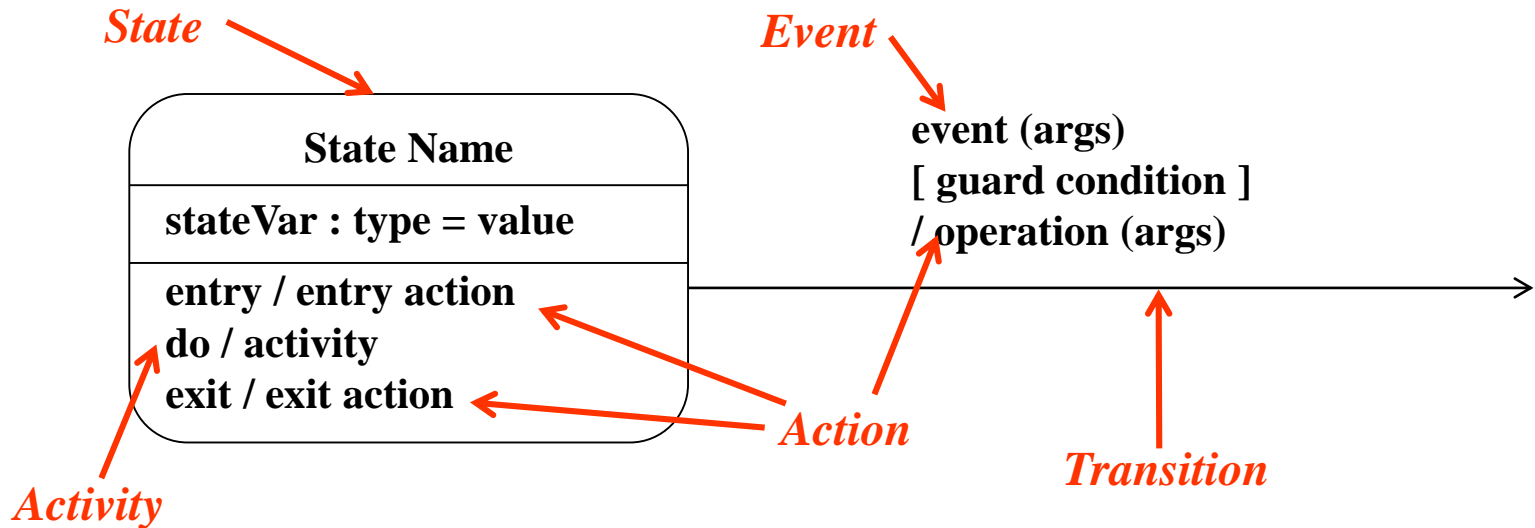
# 状态图(Statechart Diagram)

- 状态图(Statechart Diagram)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(Statechart Diagram)

## ■ 状态图的基本概念

- State(状态)
- Event(事件)
- Transition(转移)
- Action(动作)



# 状态图(Statechart Diagram)

## ■ 状态(State)

— 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。

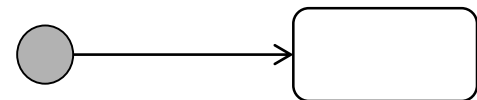
## ■ 状态图中定义的状态

— 初态、终态

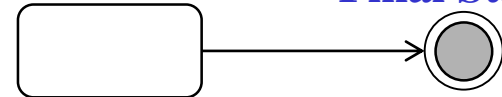
— 中间状态、组合状态、历史状态

— 一个状态图只能有一个初态，  
而终态可以有多个

**Initial State**



**Final State**





# 状态图(Statechart Diagram)

## ■ 转移(Transition)

- 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。
- 状态之间的转移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。

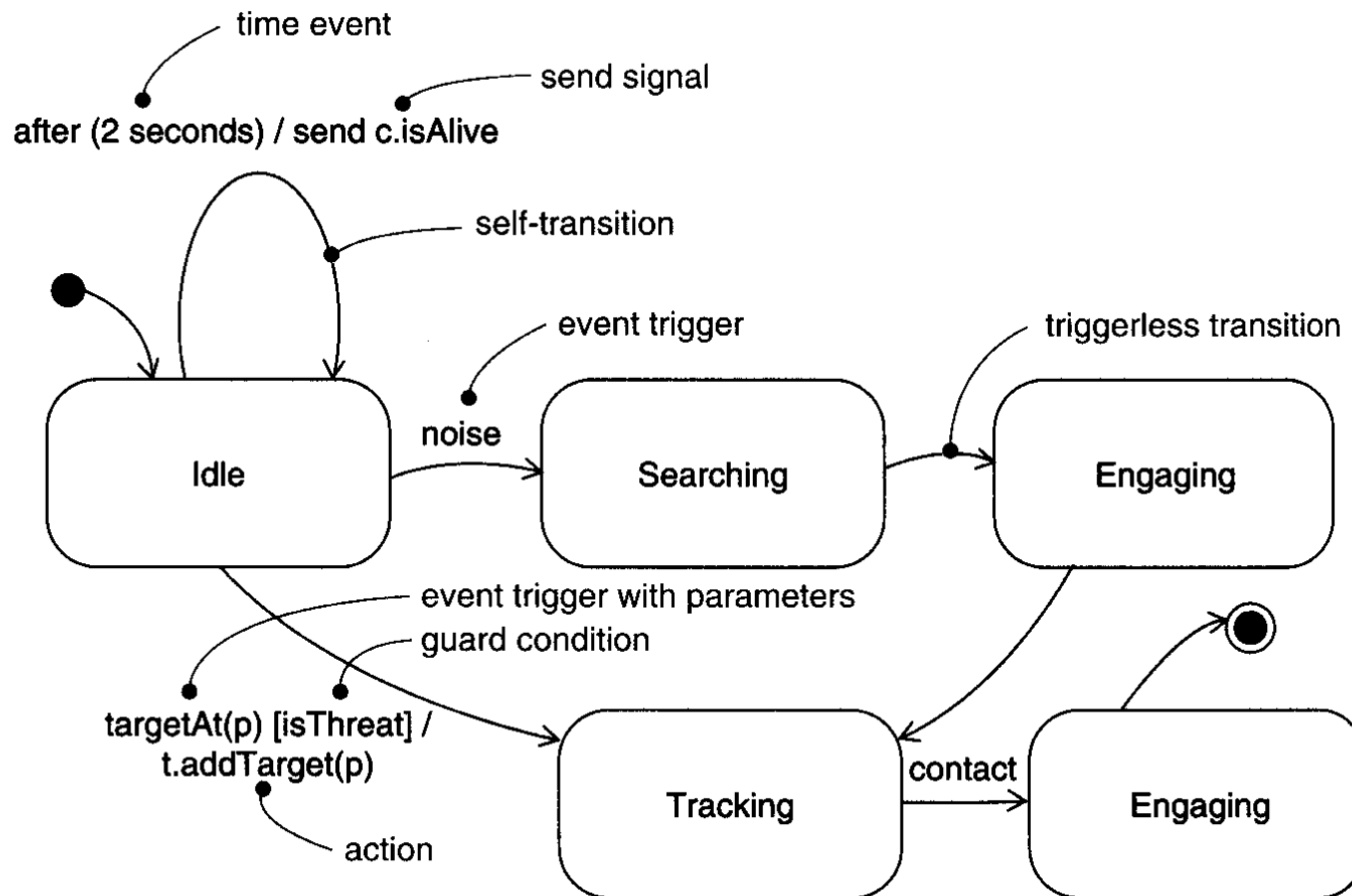
## ■ 转移格式: event-signature '['guard-condition ']'/'action

## ■ event-signature格式: event-name '('comma-separated-parameter-list ')'

## ■ 保护条件(Guards): 可选

- 一个true或false测试表明是否需要进行转换
- 当事件发生时，只有在保护条件为真时才发生转换

# 状态图(Statechart Diagram)



# 状态图(Statechart Diagram)

## ■ 事件(Event)

- An event is the specification of a noteworthy occurrence that has a location in time and space.
- 事件是一件有意义、值得关注的现象。

## ■ UML中事件分为四类

- Call Event (调用某个操作)
- Change Event: when (temperature > 120)
- Signal event (触发事件)
- Time event: after 5 seconds

# 状态图(Statechart Diagram)

- 动作(Action)

- An action is an executable atomic computation.
- 在一个特定状态下对象执行的行为

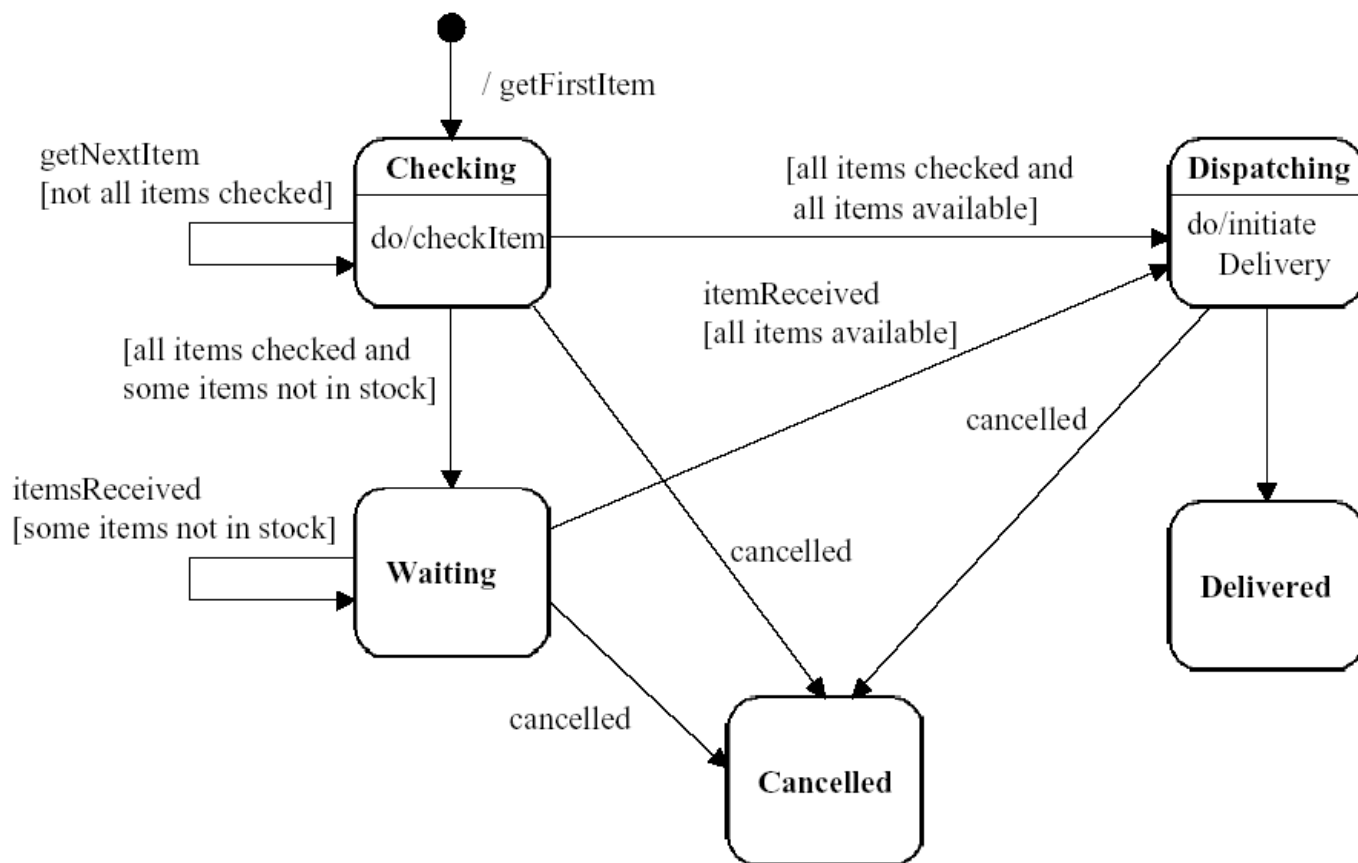
- 动作是原子的，不可被中断的，其执行时间可忽略不计的。

- 两个特殊的action: **entry action**和**exit action**

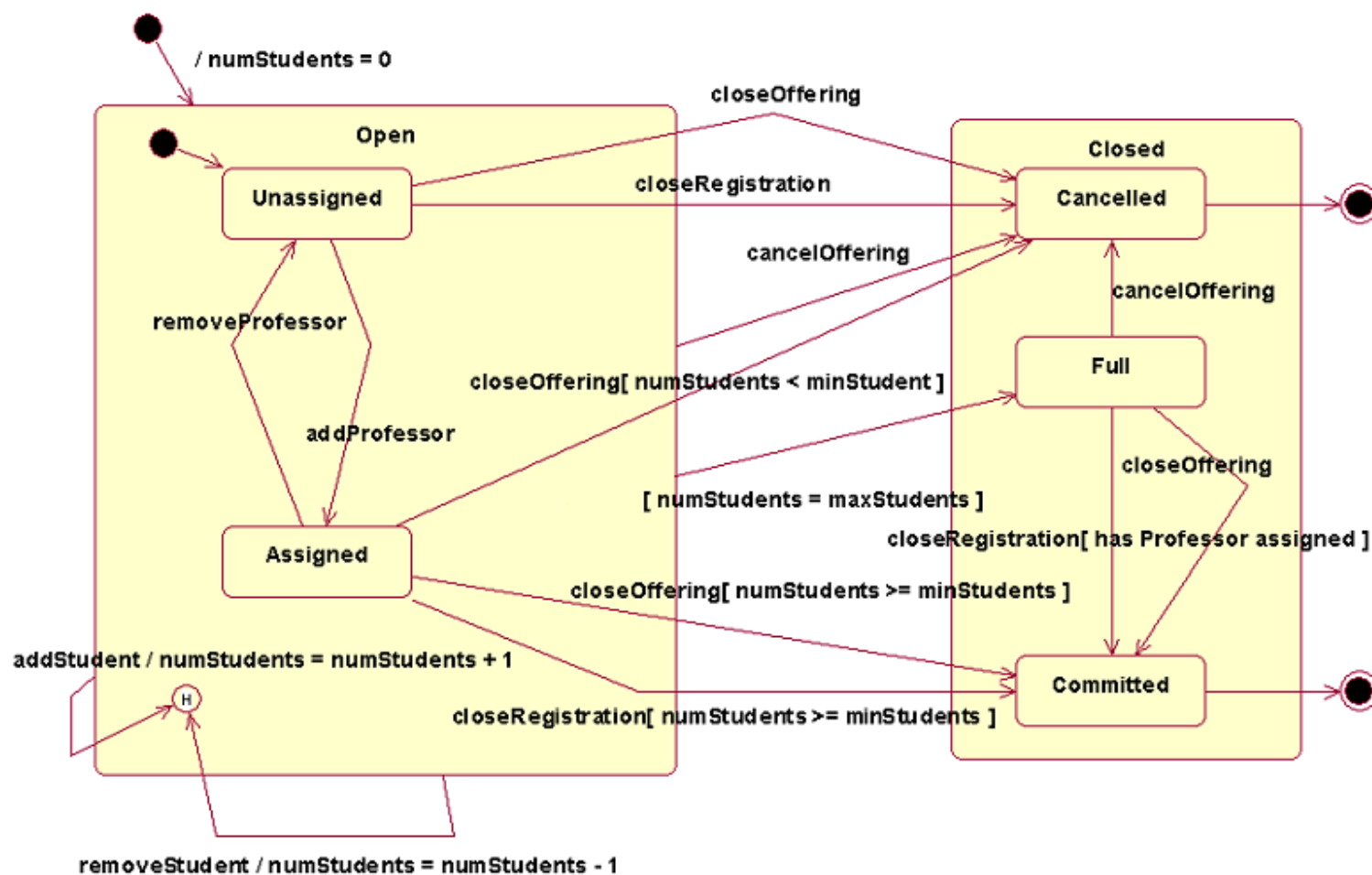
- Entry动作：进入状态时执行的活动
- Exit动作：退出状态时执行的活动

# 状态图(Statechart Diagram)

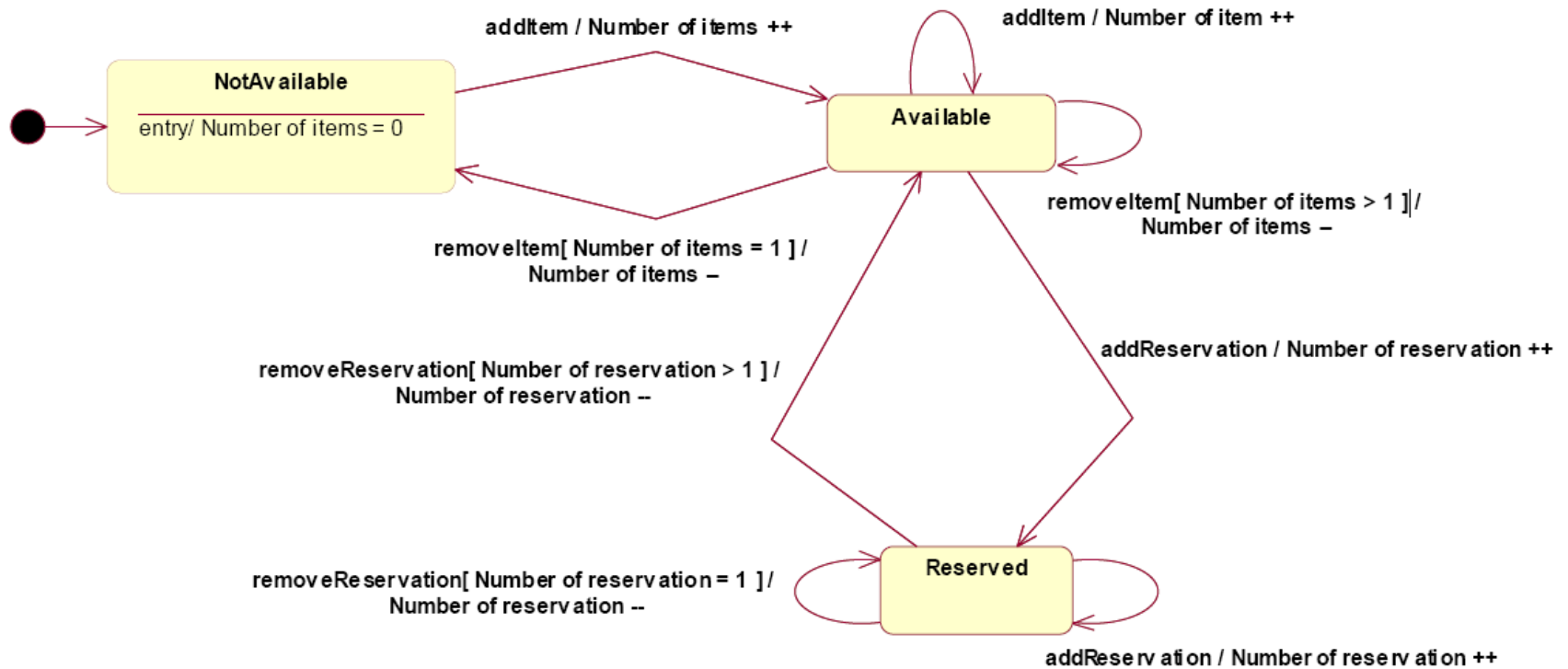
## “订单”对象的状态图



# 状态图(Statechart Diagram)



# 状态图(Statechart Diagram)



## Step 4: 建立类之间的关系

- 五种关系:

- 泛化(generalization)
- 关联(association)
- 组合(composition)
- 聚合(aggregation)
- 依赖(dependency)

- 例如:

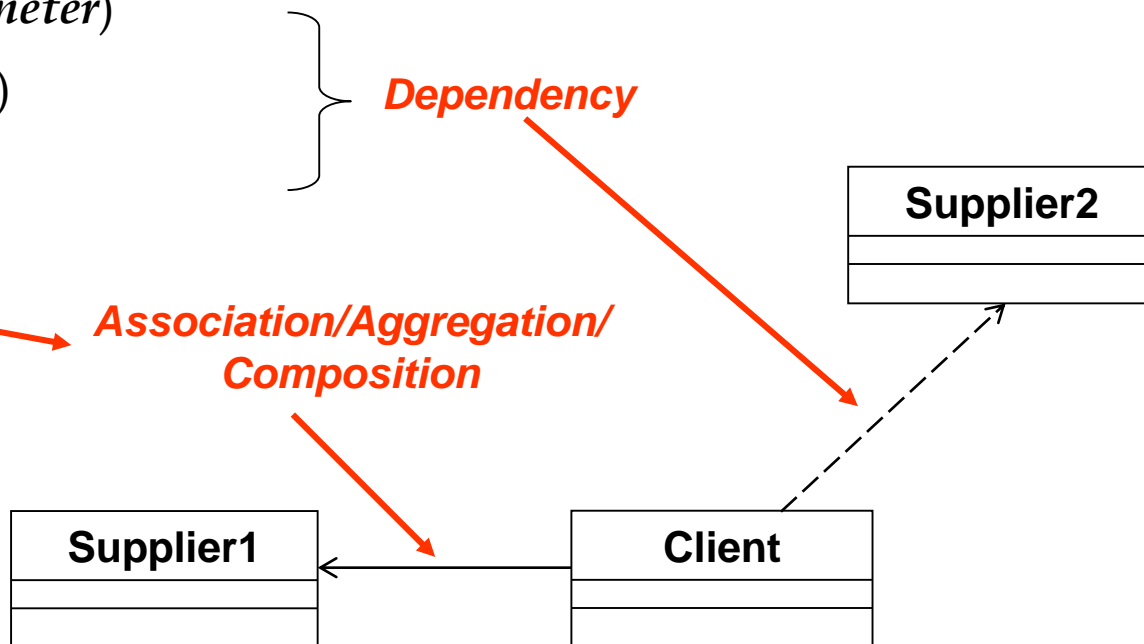
- “读者”类与“预定”类之间有1:n的关联关系;
- “读者”类与“借书记录”类之间有1:n的关联关系;
- “图书资料”类与“借书记录”类之间有1:1的关联关系;
- “书目”类与“书籍”类、“杂志”类之间是泛化关系。



## 细化类之间的关系

- “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

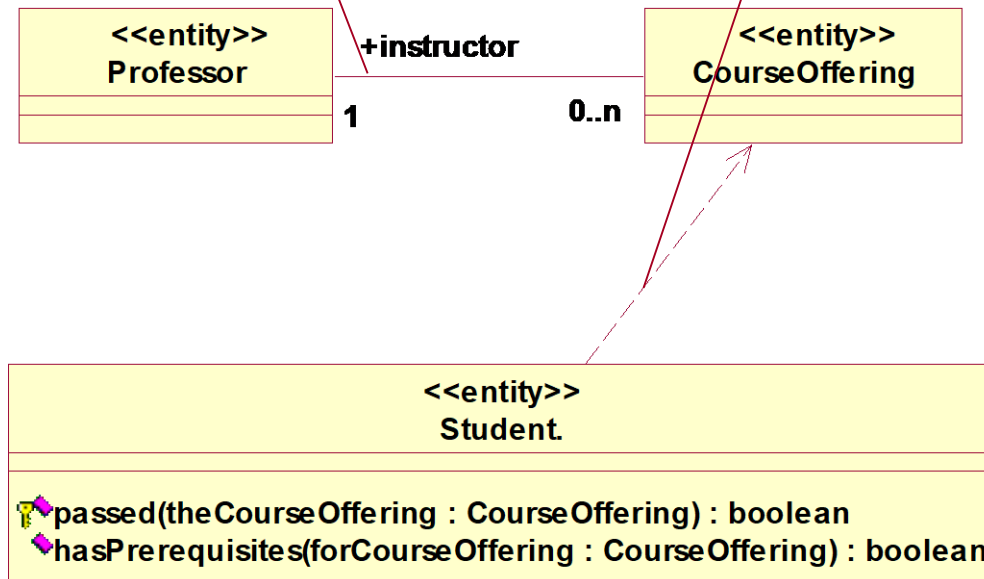
- 全局(*Global*)
- 参数(*Parameter*)
- 局部(*Local*)
- 域(*Field*)



# 定义类之间的关系： 示例

Field reference

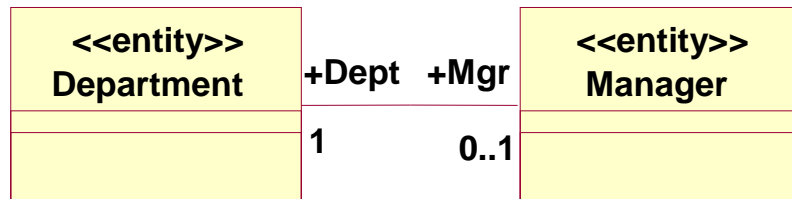
Parameter reference



# 定义关联关系

## (Association/Composition/Aggregation)

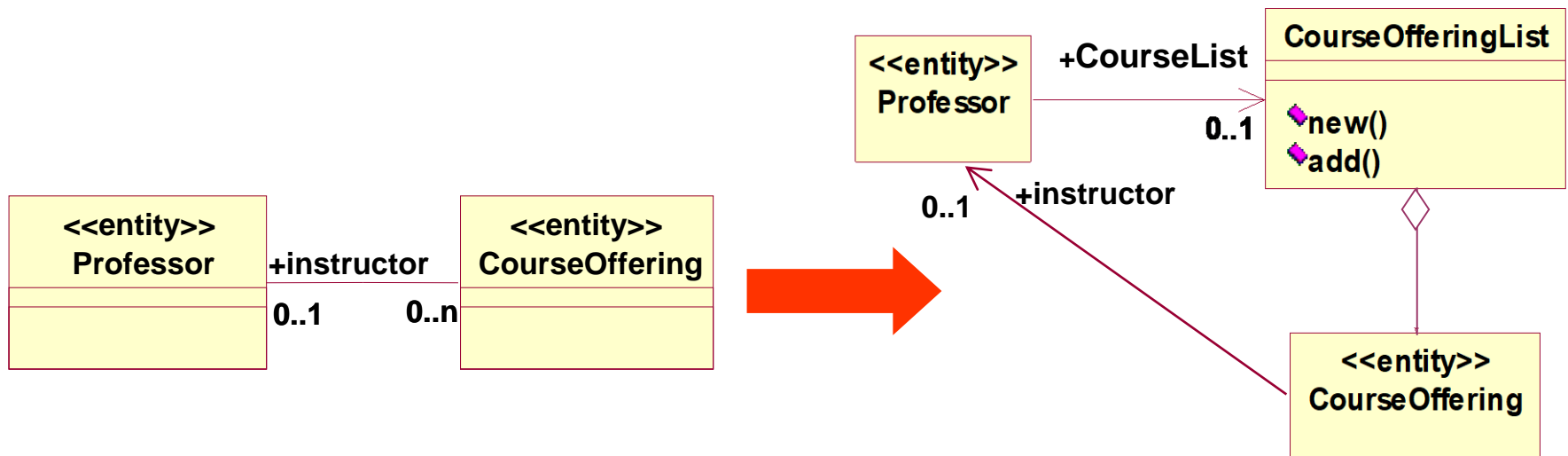
- 根据“多重性”进行设计(multiplicity-oriented design)
- 情况1: Multiplicity = 1或Multiplicity = 0..1
  - 可以直接用一个单一属性/指针加以实现，无需再作设计；
  - 若是双向关联：
    - Department类中有一个属性：+Mgr: Manager
    - Manager类中有一个属性：+Dept: Department
  - 若是单向关联：
    - 只在关联关系发出的类中增加关联属性。



需要将这种“关联属性”增加到属性列表中，并更新操作列表

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

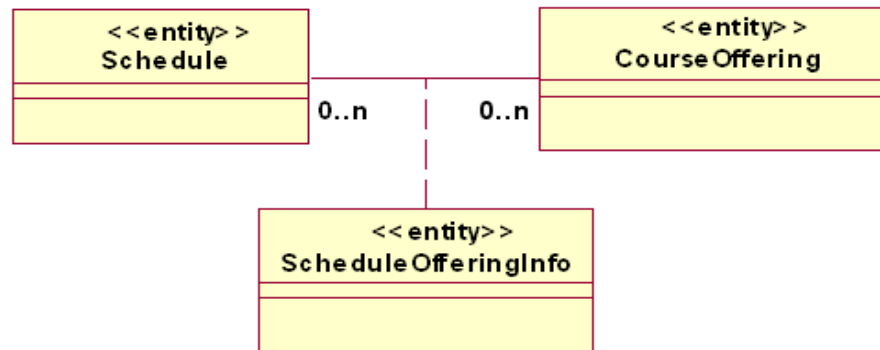
- 根据“多重性”进行设计(multiplicity design)
- 情况2: Multiplicity > 1
  - 无法用单一属性/指针来实现, 需要引入新的设计类或能够存储多个对象的复杂数据结构(例如链表、数组等)。
  - 将1:n转化为若干个1:1。



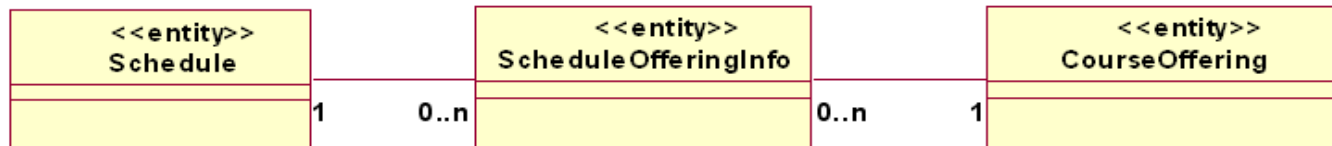
# 定义关联关系

## (Association/Composition/Aggregation)

- 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



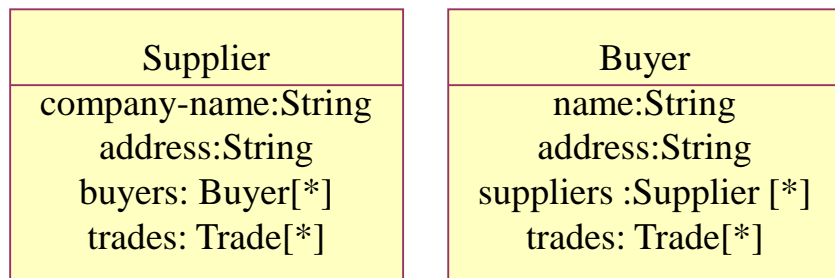
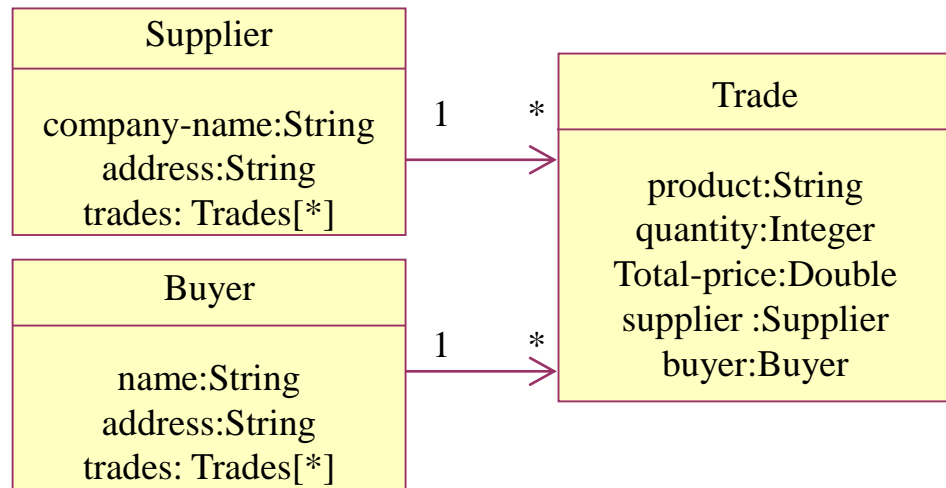
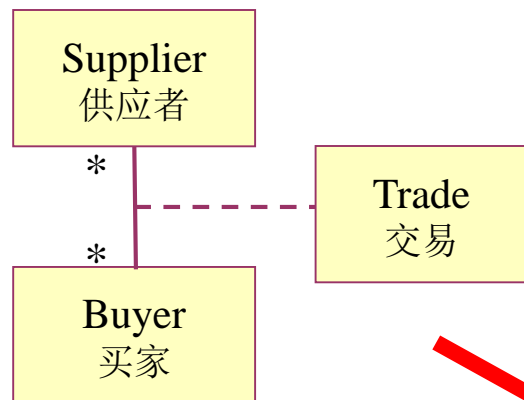
*Design Decisions*



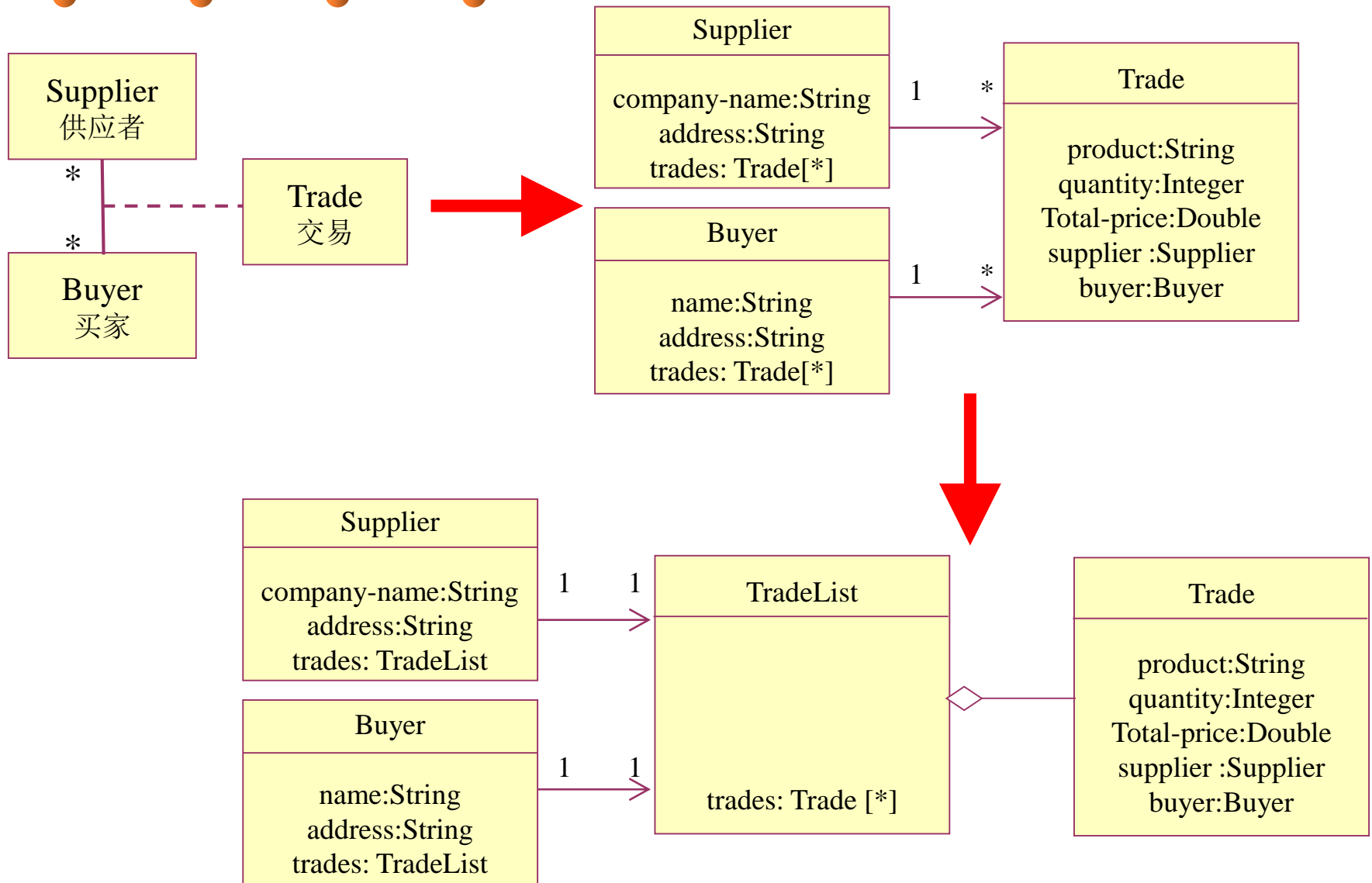
## 定义关联关系

## (Association/Composition/Aggregation)

使用关联类和不使用关联类的对比分析



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



## 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)。

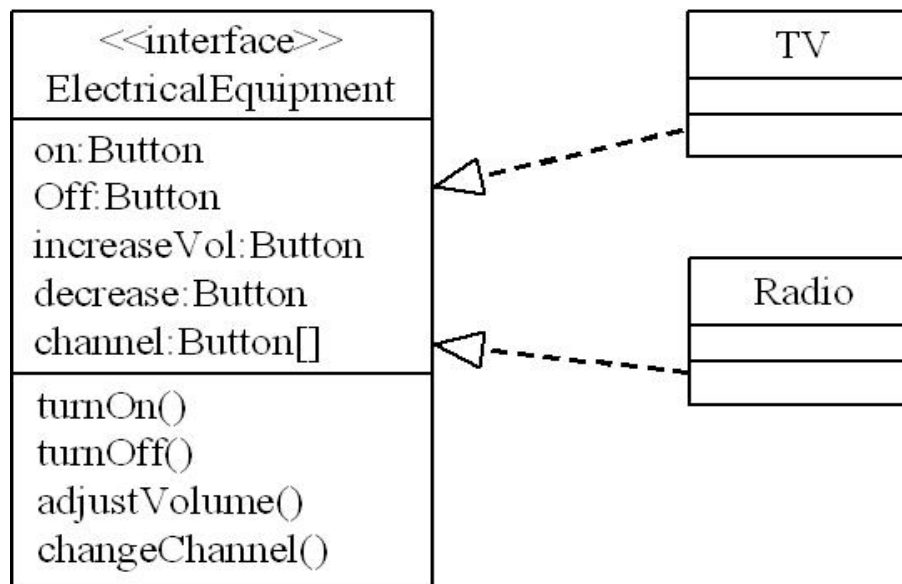


## 引入“辅助类”简化类的内部结构

- 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)、流转纪录(text)、操作人(text)”等属性，故而可以将“订单物流纪录”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，就没有物流纪录)。
- 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。

## 引入接口实现对“变化”的封装

- 实现关系(realization): 是泛化关系和依赖关系的结合, 通常用以描述一个接口和实现它们的类之间的关系;
- 是“棒棒糖”的另一种形式。



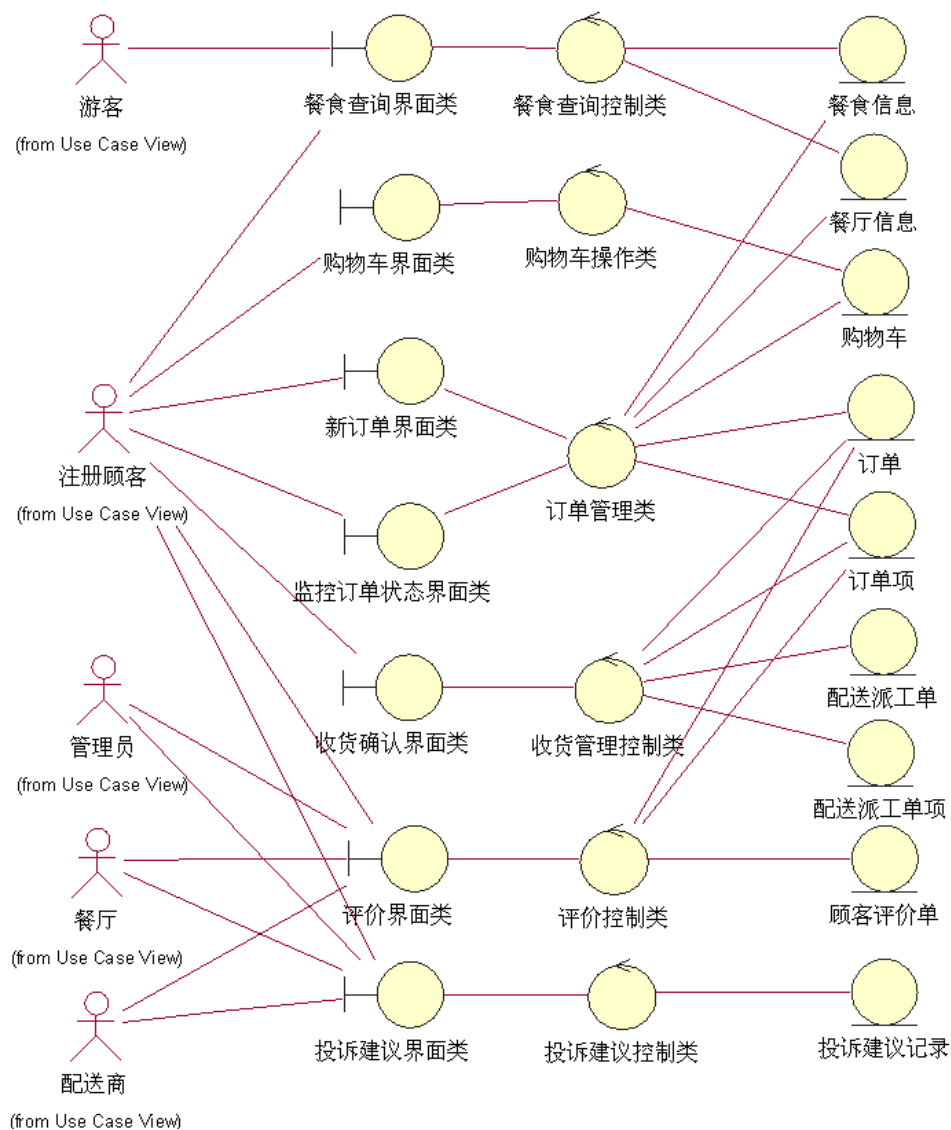
## Step 5: 绘制类图

- 两种类图:

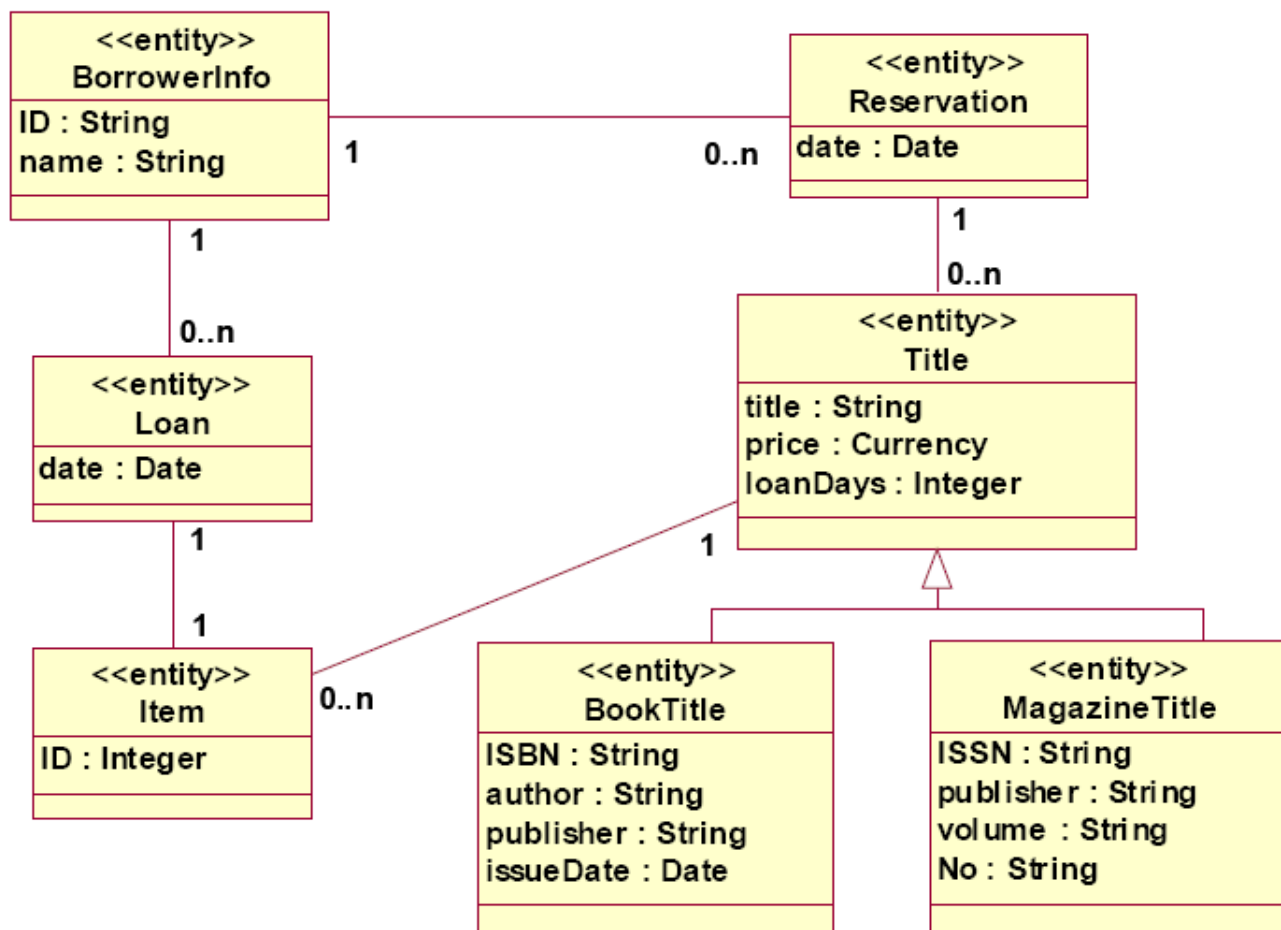
- 分析类图: 描述各边界类、实体类、控制类之间的关联关系, 无需刻画属性与操作集;
- 领域类图: 可以不包含边界类与控制类, 侧重描述各实体类之间的五种关系, 需要给出详细的属性与操作集合。

- 在不加说明的情况下, 领域类图更有价值。

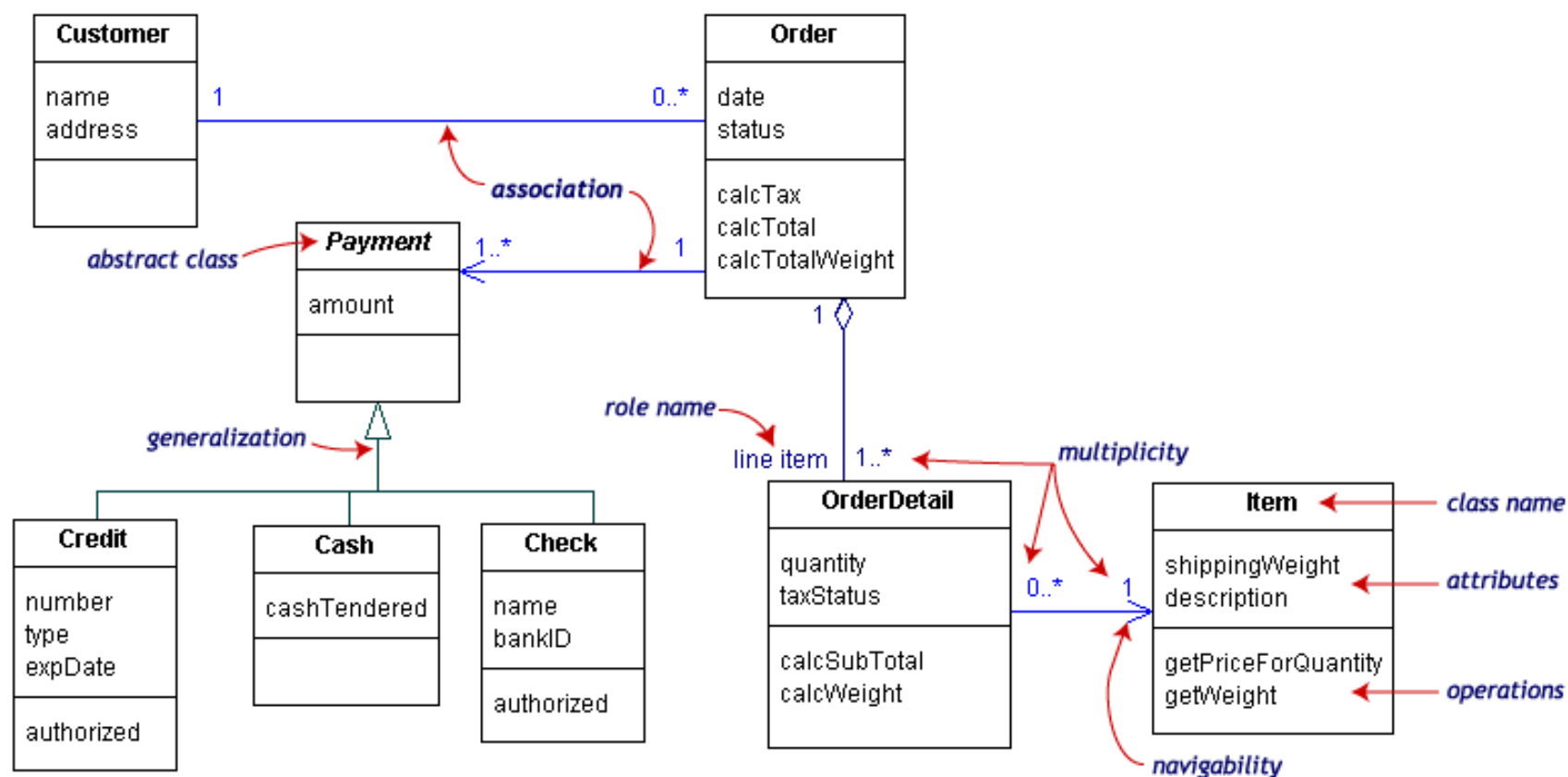
# 分析类图实例



# 领域类图实例(1)



# 领域类图实例(2)





### 3 建立动态行为模型

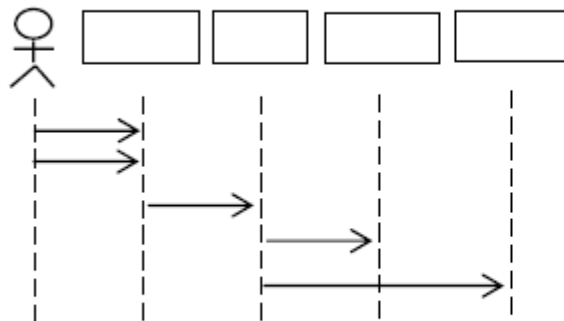


# 建立动态行为模型

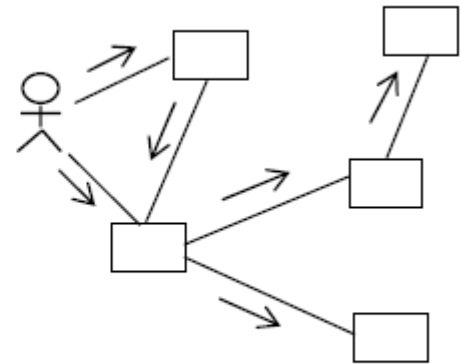
- 动态行为模型可用两个新视图描述：
  - 时序图(Sequence Diagram)
  - 协作图(Collaboration Diagram)



用例



时序图



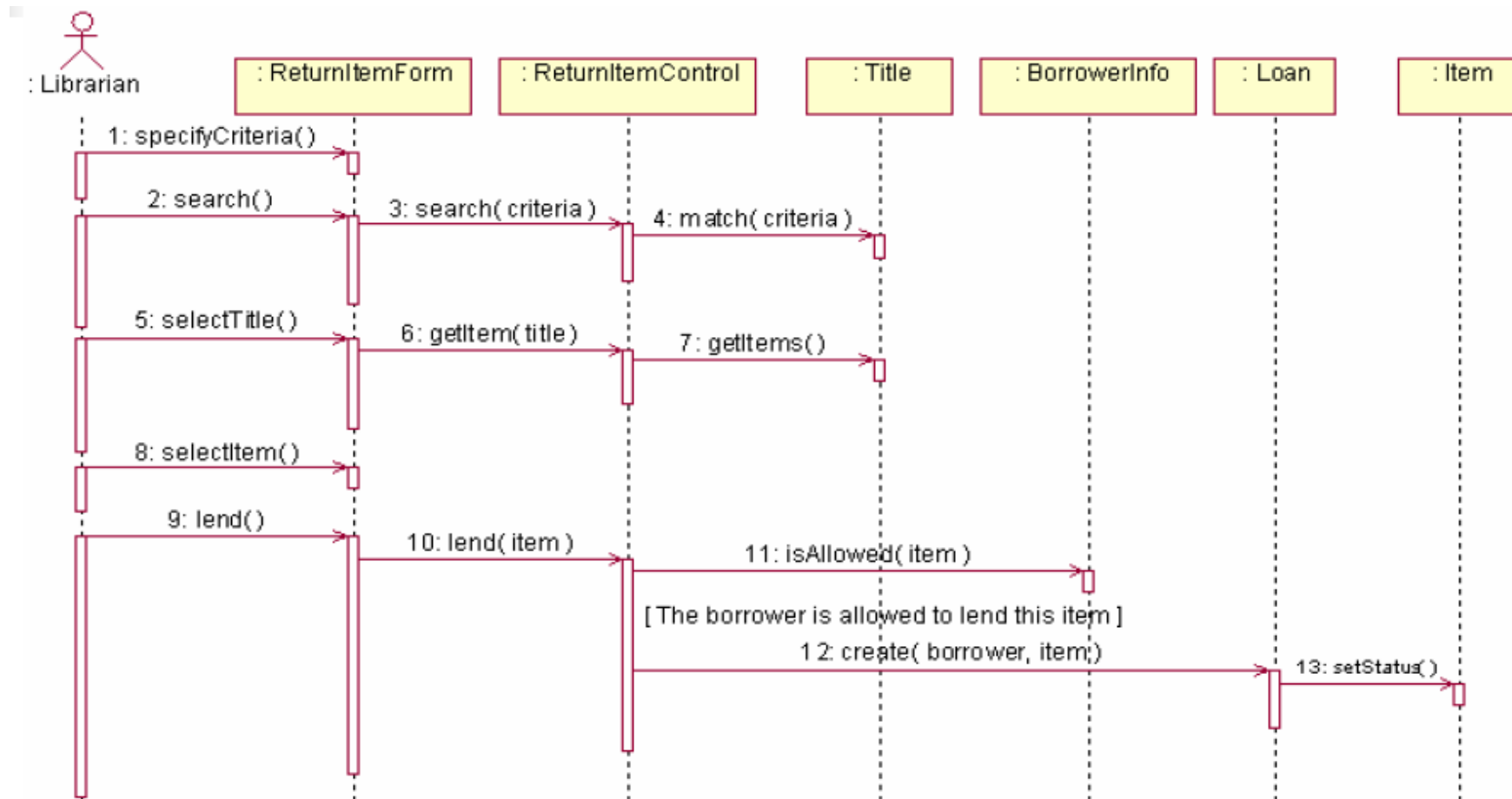
协作图



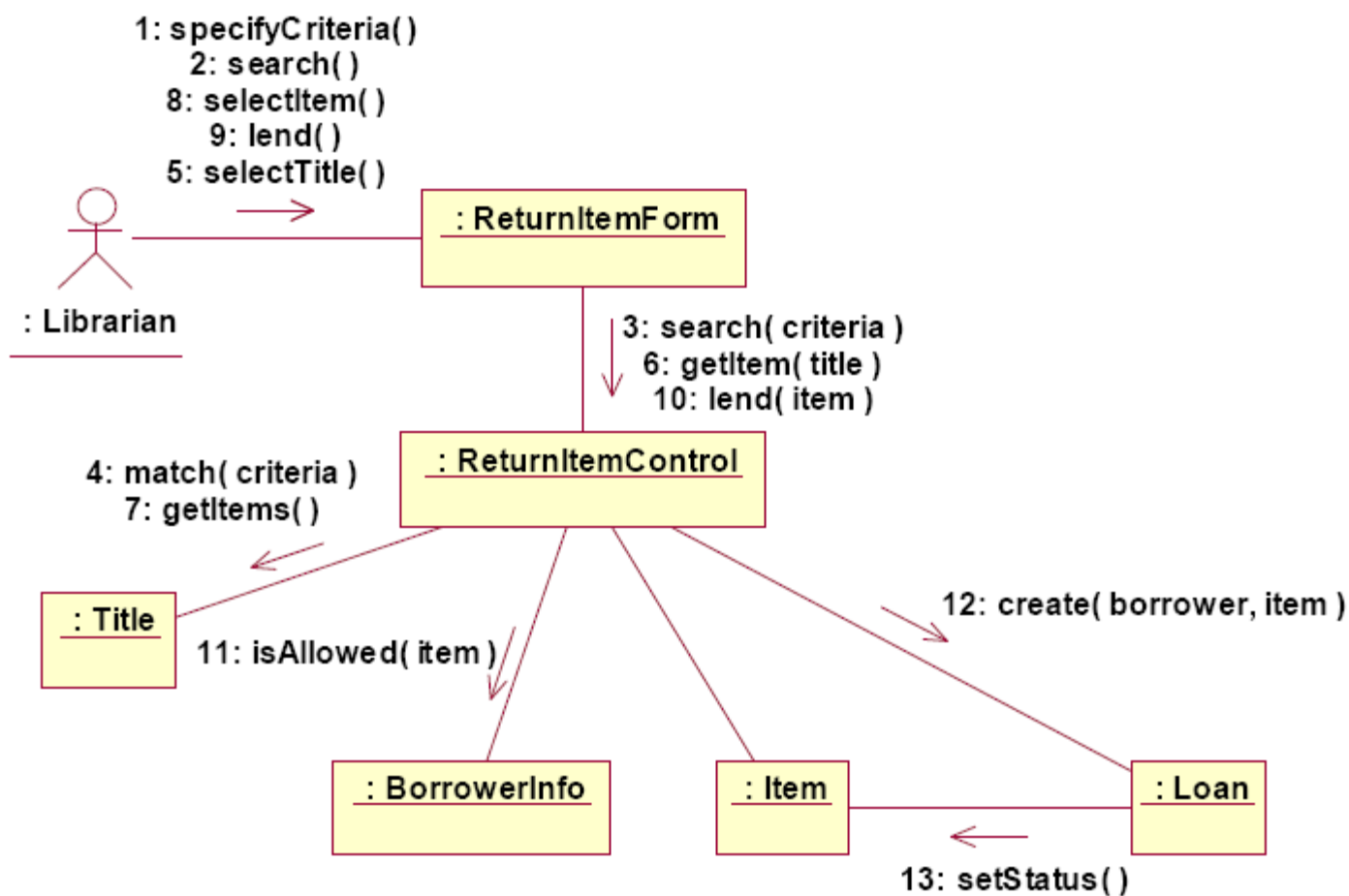
# 时序图

- **时序图(Sequence Diagram):** 将用户与分析类结合在一起, 实现将用例的行为分配到所识别的分析类中;
- **绘制步骤:**
  - 列出启动该用例的参与者;
  - 列出启动用例时参与者使用的边界对象;
  - 列出管理该用例的控制对象;
  - 根据用例描述的流程, 按时间顺序列出分析类之间进行消息访问的序列。
- **说明:**
  - 每个用例对应一张时序图;
  - 时序图描述的消息序列需要与用例的事件流保持一致;
  - 时序图中出现的每个箭头代表一个操作, 需要包含在箭头所指向的分析类中。

# [案例]图书管理系统的时序图



# [案例]图书管理系统的协作图

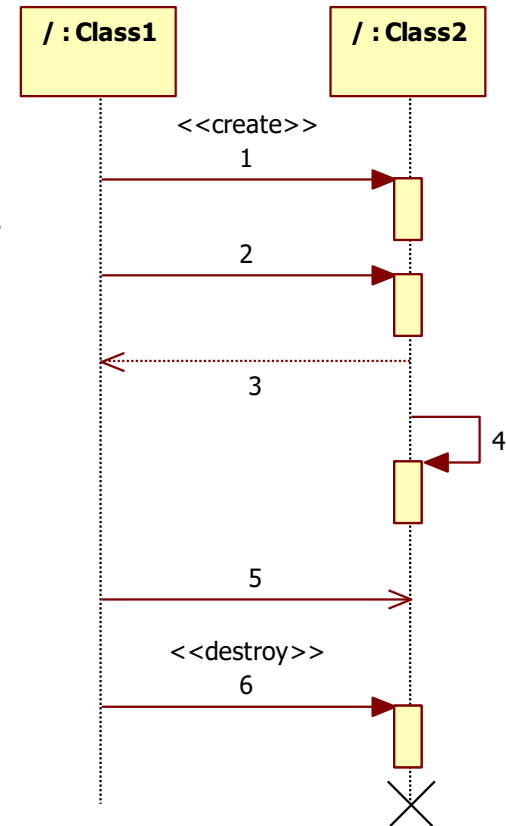


# 关于时序模型与用例模型的关系

- 用例事件流中：
  - 用户请求XX；
  - 系统响应XX；
- 在时序图中：
  - 前者对应于actor向边界类发出的指令；
  - 后者对应于边界类向控制类、实体类发出的指令、执行、反馈结果。
- 用例事件流描述用自然语言，但时序图将其转化为类之间的函数调用与消息传递，是对事件流的“可实现化的描述”。
- 对比：
  - 用例的事件流更粗放，是给用户看的；
  - 时序模型的事件流更细致，是给程序员看的；
  - 若二者完全一致了，说明你设计得不够细节。

# 时序模型

- 时序模型中的每一列：本质上是object，每个出现的object，其相应的class必须都在分析类模型/领域模型中已经定义过了。
  - 这些object的排列次序无所谓，但按照惯例，一张时序图从左到右应该是：actor(用户)、边界类、控制类、实体类、actor(外部系统)。
- 出现在时序图中的箭头，UML中分了六种类型：
  - Create: obj1调用Class2的new操作，创建出obj2实例
  - Call: obj1调用obj2的某个操作
  - Return: obj2被调用的操作执行结束后，将结果返回obj1
  - Self-call: obj2自己调用自己内部的某个操作；
  - Send: obj1向obj2发送消息来触发(而不直接调用其操作)
  - Destroy: obj1调用Class2的destroy操作，销毁obj2实例。
  - 图形表示方式分别如右图中的1~6箭头所示。



## 关于时序模型

- 任何类型的箭头(create、call、return、self-call、send、destroy), 均代表着该箭头需要触发obj2的某个操作。
- 这就意味着, 箭头上标识的操作, 必须在Class2的操作集合中有所定义。
- 所以, 如果在分析类和领域建模阶段没有想全, 建立时序模型时可以通过一致性检查发现缺失。



## 4 面向对象分析的小案例



## 案例：学生选课系统

- 教学秘书需要录入可选课程信息、任课教师信息、学分政策，并从学籍管理系统中导入学生信息；
- 教师登录进入系统，查询本学期所开设课程清单，并选择自己所承担的课程；
- 学生登录进入系统，查询本学期可选课程的清单，并创建自己的选课单，将某些课程加入到选课单中；学生可对选课单进行维护，包括加入其他课程、删除已选课程等；
- 学生也可对选课单中包含的数据进行学分政策验证，判断所选课程是否满足学校要求；
- 在规定时间之前，学生将选课单做正式提交；
- 教学秘书检查每个学生的选课单，若不符合学分政策，退回重选。否则，根据所有学生提交的选课单，生成课表和每门课程的学生清单；
- 教师可查看自己承担课程的课表与学生清单，学生可查询自己的课表。



# OO分析的步骤：回顾

- Step 1: 角色识别
- Step 2: 用例识别
- Step 3: 建立用例模型
- Step 4: 对用例图进行精化(include, extend, generalization)
- 针对每个用例：
  - Step 5: 撰写用例描述
  - Step 6: 建立用例的活动(泳道)图
  - Step 7: 识别分析类(边界类、控制类、实体类)
  - Step 8: 识别每个类的属性和方法
  - Step 9: 建立分析类图
  - Step 10: 建立领域类图
  - Step 11: 建立时序图

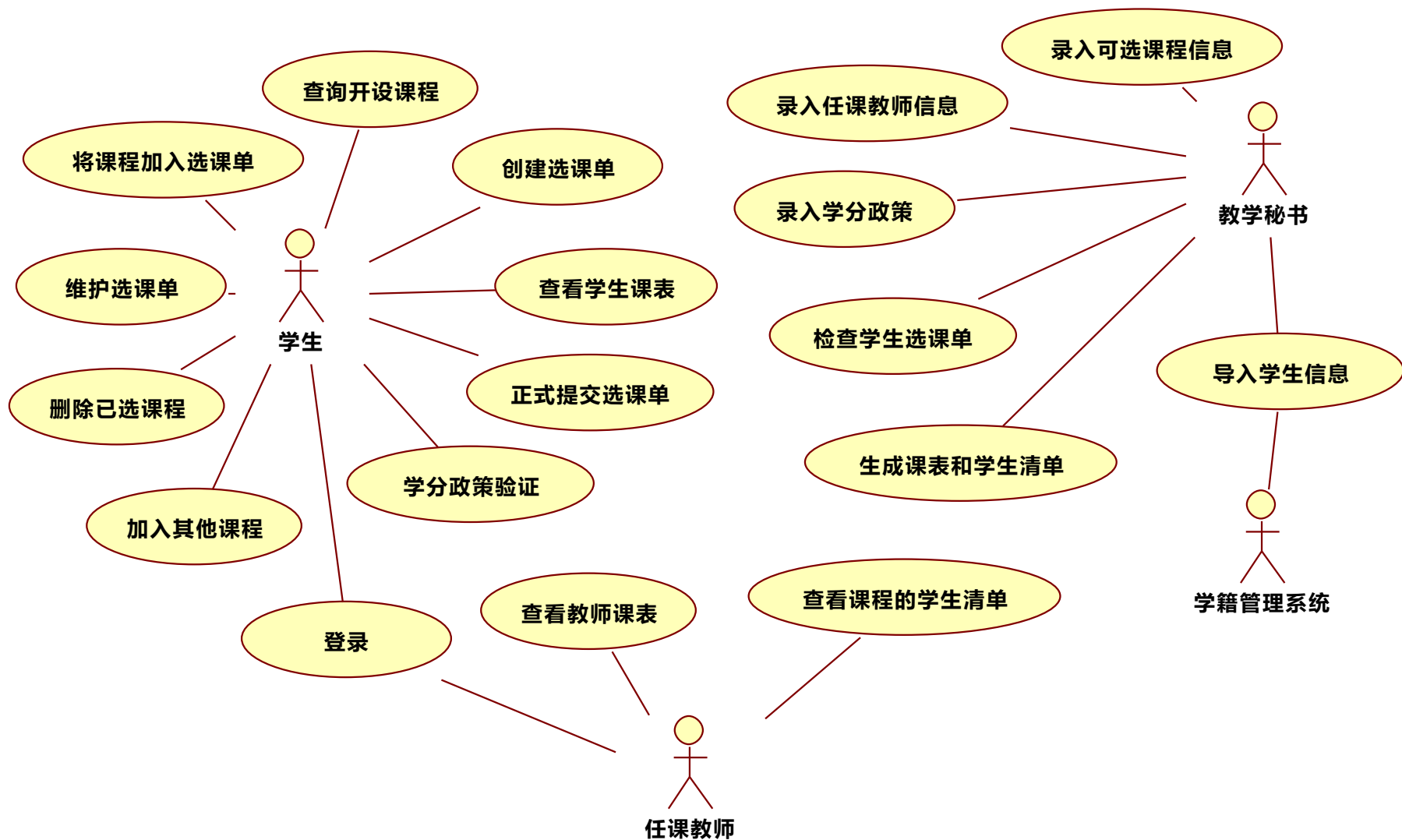
## 步骤1: 角色识别

- 教学秘书需要录入可选课程信息、任课教师信息、学分政策，并从学籍管理系统中导入学生信息；
- 教师登录进入系统，查询本学期所开设课程清单，并选择自己所承担的课程；
- 学生登录进入系统，查询本学期可选课程的清单，并创建自己的选课单，将某些课程加入到选课单中；学生可对选课单进行维护，包括加入其他课程、删除已选课程等；
- 学生也可对选课单中包含的数据进行学分政策验证，判断所选课程是否满足学校要求；
- 在规定时间之前，学生将选课单做正式提交；
- 教学秘书检查每个学生的选课单，若不符合学分政策，退回重选。否则，根据所有学生提交的选课单，生成课表和每门课程的学生清单；
- 教师可查看自己承担课程的课表与学生清单，学生可查询自己的课表。

## 步骤2: 用例识别

- 教学秘书需要录入可选课程信息、任课教师信息、学分政策，并从学籍管理系统中导入学生信息；
- 教师登录进入系统，查询本学期所开设课程清单，并选择自己所承担的课程；
- 学生登录进入系统，查询本学期可选课程的清单，并创建自己的选课单，将某些课程加入到选课单中；学生可对选课单进行维护，包括加入其他课程、删除已选课程等；
- 学生也可对选课单中包含的数据进行学分政策验证，判断所选课程是否满足学校要求；
- 在规定时间之前，学生将选课单做正式提交；
- 教学秘书检查每个学生的选课单，若不符合学分政策，退回重选。否则，根据所有学生提交的选课单，生成课表和每门课程的学生清单；
- 教师可查看自己承担课程的课表与学生清单，学生可查询自己的课表。

## 步骤3: 绘制用例图



## 步骤4：对用例图进行精化

### ■ 问题1：

- 用例“将课程加入选课单”是否是一个独立交互？——不是，需要先查询，再将课程加入选课单。
- 它与“查询开设课程”之间的关系是什么？——include? extend?

### ■ 如何修改？

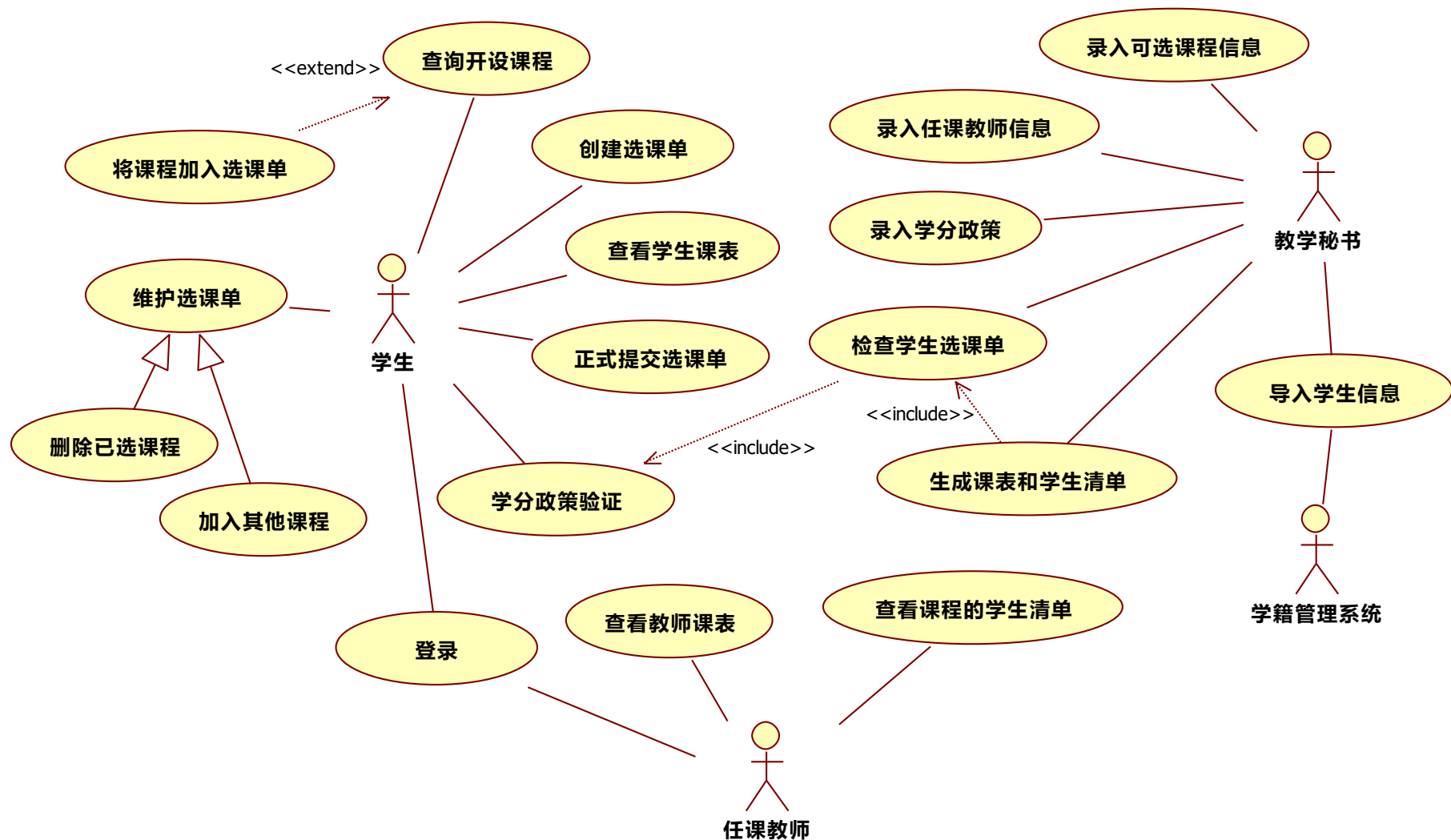
- 前者→<extend>→后者？
- 后者→<include>→前者？——哪种更恰当？为什么？

### ■ 问题2：“维护选课单”与“加入其他课程”、“删除已选课程”之间是什么关系？——generalization

### ■ 问题3：“检查学生选课单”是否有必要独立存在？它与“生成课表和学生清单”是什么关系？——通常不会单独检查，而是在生成之前检查；

### ■ 问题4：“检查学生选课单”与“学分政策验证”是什么关系？——前者调用后者。

## 步骤4: 对用例图进行精化



## 步骤5: 撰写用例描述

### 用例1: 查询开设课程

#### 1. 目标:

本用例允许学生查询本学期所开设课程, 进而选择课程加入到选课单中;

#### 2 事件流:

##### 2.1 常规流程

- (1) 学生输入课程查询条件(无条件意味着列出全部), 点击查询;
- (2) 系统查询出所有可选课程, 以列表形式展示; 随后可执行下面的任一步骤:
- (3) 学生可对课程列表按开课时间/任课教师排序,
- (4) 选择某一门课程查看详细信息;
- (5) 学生选择某一门课程, 可进入用例2“将课程加入选课单”。该步骤可重复多次;
- (6) 学生选择退出。

##### 2.2 备选流程

无

#### 3 前提条件: 用例开始前, 学生须在系统登录成功;

#### 4 后置条件: 如果用例执行成功, 学生可以看到满足条件的全部可选课程。若学生选课, 则所有被选课程将加入到选课单中, 否则选课单保持不变。

## 步骤5：撰写用例描述

### 用例2：将课程加入选课单

#### 1. 目标：

本用例允许学生将一门课程加入到选课单中；

#### 2 事件流：

##### 2.1 常规流程

- (1) 学生选择一门课程，点击“加入选课单”；
- (2) 系统将该课程加入到选课单，并弹出窗口，展示目前已选课程；

##### 2.2 备选流程

在(2)中，若系统发现该课程已经在选课单中，则提示“该课已选”，选课单状态不更新。

**3 前提条件：**用例开始前，学生须在系统登录成功并已经查询到课程清单；

**4 后置条件：**如果用例执行成功，被选课程将加入到选课单中，否则选课单保持不变。



## 步骤5: 撰写用例描述

### 用例3: 导入学生信息

#### 1. 目标:

本用例允许教学秘书手工将已注册学生的信息从外部的学籍管理系统中导入到本系统;

#### 2 事件流:

##### 2.1 常规流程

- (1) 教学秘书输入要导入的班级号码, 点击“导入学生数据”;
- (2) 系统将系统内该班级的学生信息清空, 同时生成调用指令, 调用学籍管理系统所提供的API接口;
- (3) 学籍管理系统返回该班级中已注册的学生名单;
- (4) 系统接收学生名单, 将其保存起来;
- (5) 系统将学生名单显示给教学秘书;

##### 2.2 备选流程

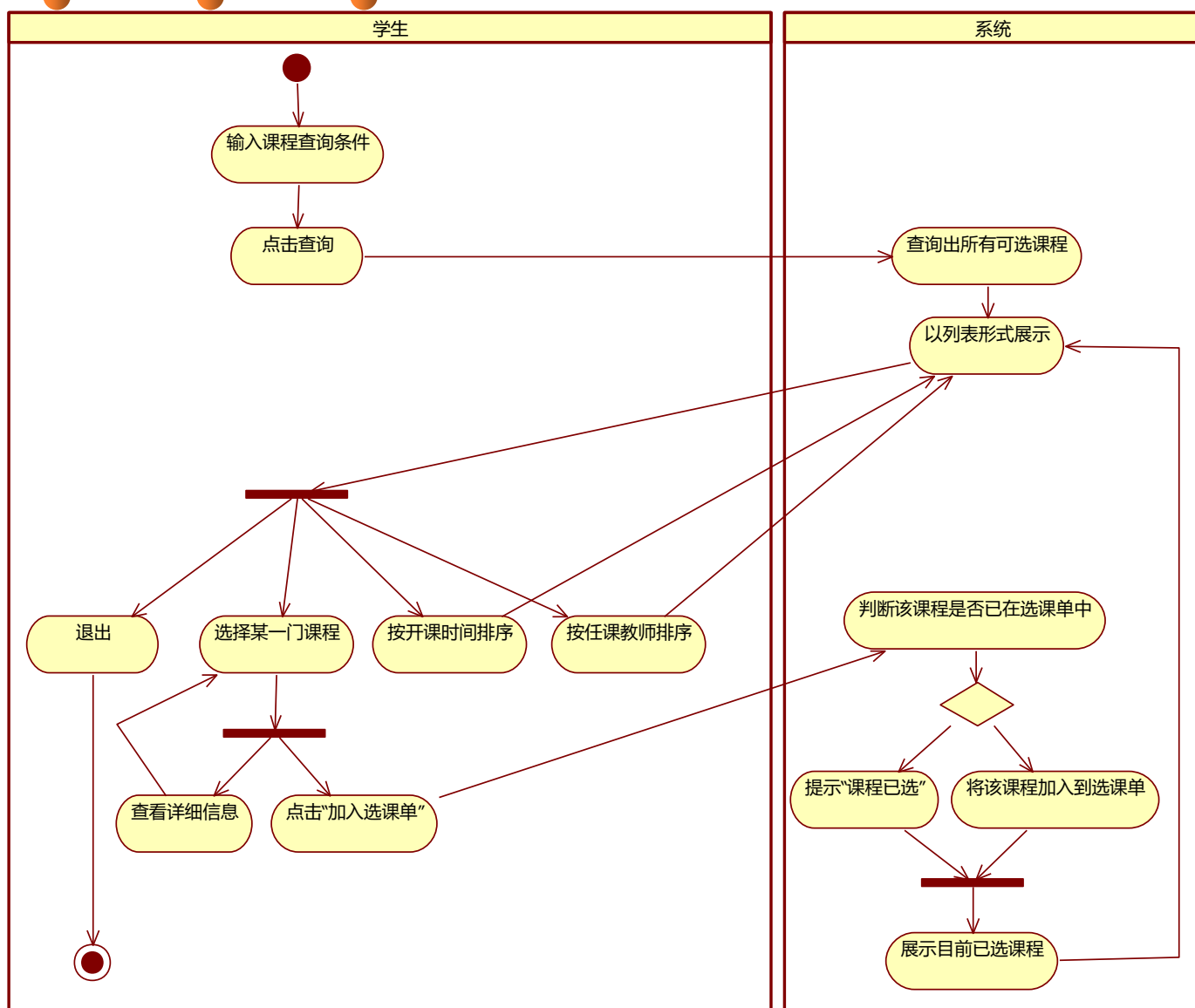
在步骤(2)中, 若系统发现该班号不存在, 则提示“输入班号错误”, 返回重新输入。

在步骤(4)中, 若系统发现学生名单为空, 则提示“该班尚未有学生注册”, 直接退出用例;

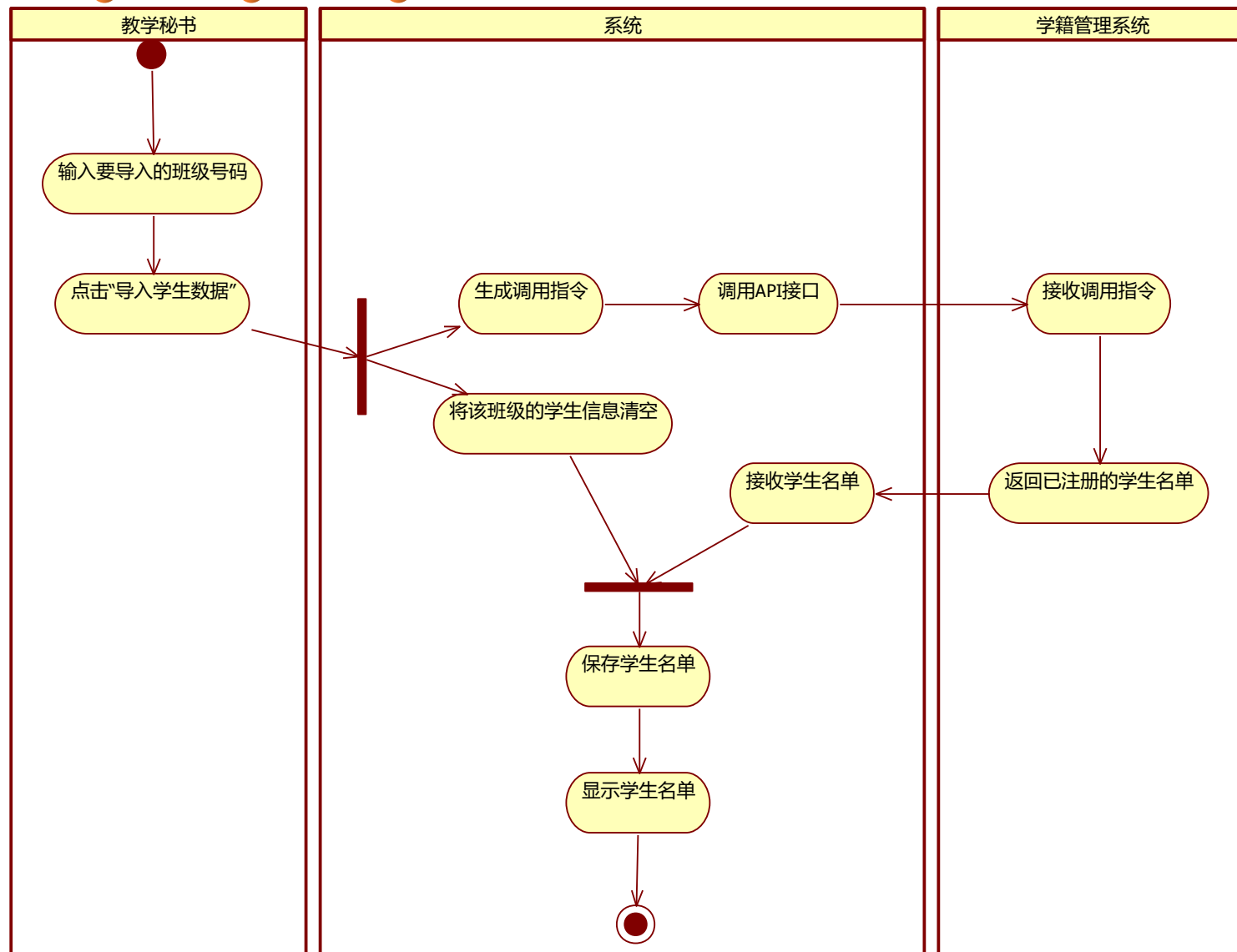
#### 3 前提条件: 无;

#### 4 后置条件: 如果用例执行成功, 系统将某班级的学生信息更新为学籍系统中的信息。

## Step 6: 绘制用例的活动(泳道)图

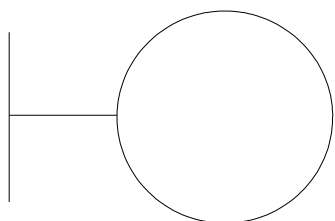
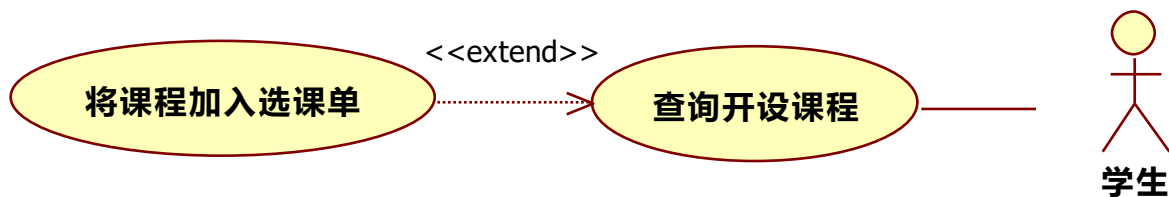


## Step 6: 绘制用例的活动(泳道)图

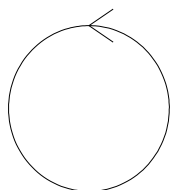


## Step 7: 识别分析类(边界类、控制类、实体类)

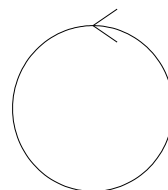
- 针对“查询开设课程”、“将课程加入选课单”这两个用例的分析类:



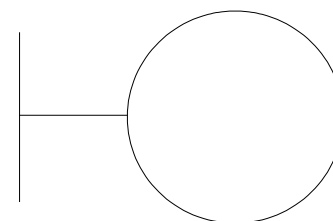
显示已选课程界面类



将课程加入选课单控制类



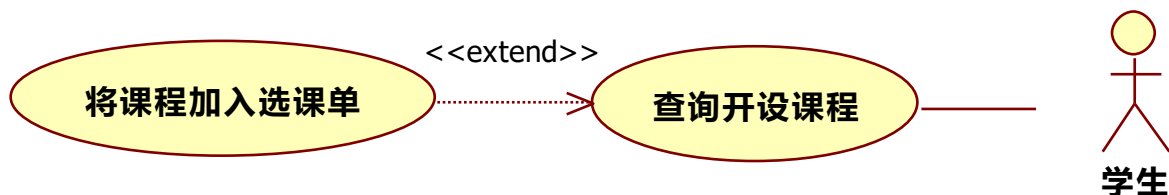
查询开设课程控制类



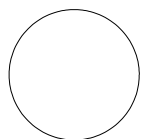
查询开设课程界面类

## Step 7: 识别分析类(边界类、控制类、实体类)

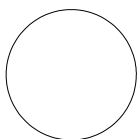
- 针对“查询开设课程”、“将课程加入选课单”这两个用例的分析类:



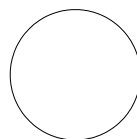
- 
- (1) 学生输入课程查询条件(无条件意味着列出全部), 点击查询;
  - (2) 系统查询出所有可选课程, 以列表形式展示;
  - (3) 学生可对课程列表按开课时间/任课教师排序, 或选择某一门课程查看详细信息;
  - (4) 学生选择某一门课程, 可进入用例2“将课程加入选课单”。该步骤可重复多次;
    - (1) 学生选择一门课程, 点击“加入选课单”;
    - (2) 系统将该课程加入到选课单, 并弹出窗口, 展示目前已选课程;
  - (5) 学生选择退出。



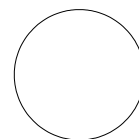
学生



课程



选课单



任课教师

# Step 8: 识别每个类的属性和方法

<<boundary>> 显示已选课程界面类
+已选课程清单
+刷新已选课程清单()

<<control>> 查询开设课程控制类
+查询开设课程() +查询课程详细信息() +将课程加入选课单()

<<control>> 将课程加入选课单控制类
+将课程加入选课单() +判断课程是否已在清单中()

<<boundary>> 查询开设课程界面类
+查询条件1(课程名) +查询条件2(教师名) +查询得到的课程清单 +当前所选课程
+查询开设课程() +显示课程清单() +按开课时间排序() +按任课教师排序() +从列表中选择课程() +查看课程详细信息() +将课程加入选课单() +退出()

<<entity>> 任课教师
+工号 +姓名

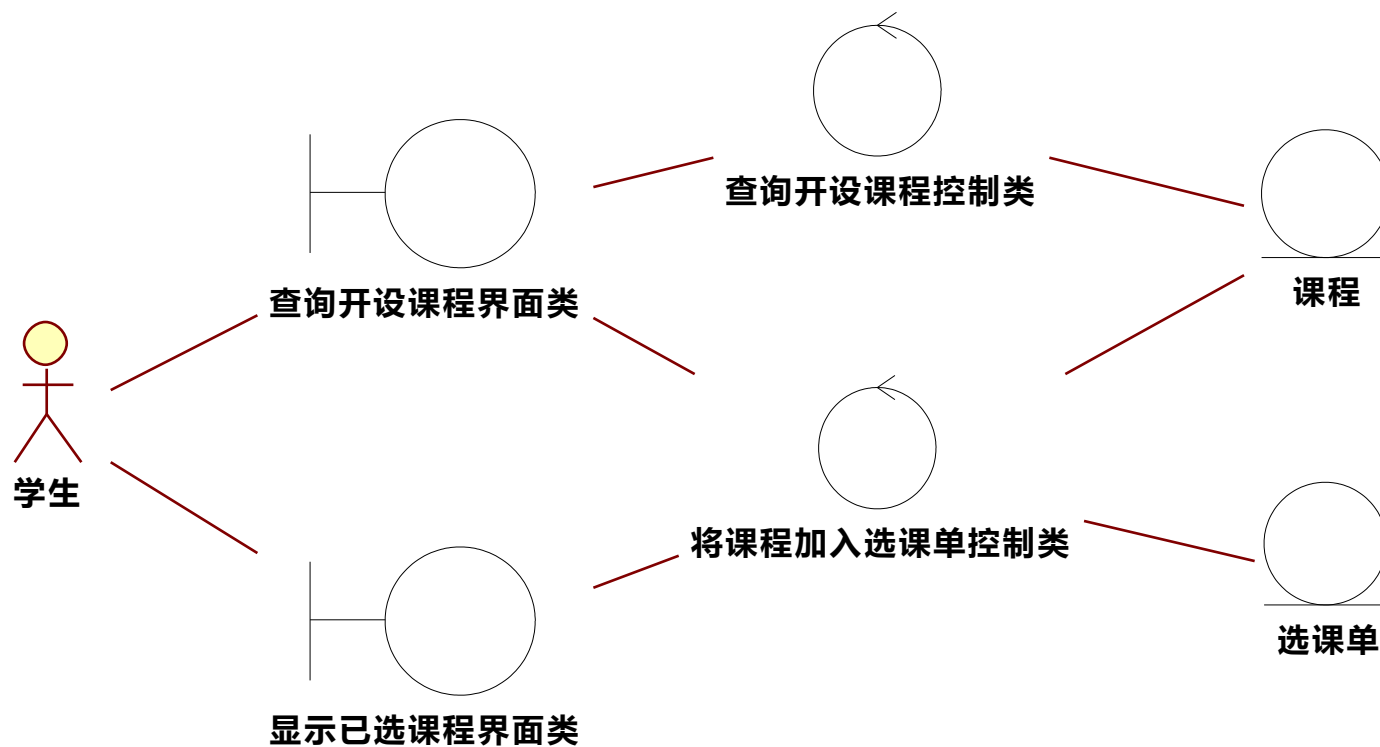
<<entity>> 学生
+学号 +姓名

<<entity>> 课程
+课程号 +开课时间 +周数 +每周课时数 +任课教师 +学分
+查询课程详细信息()

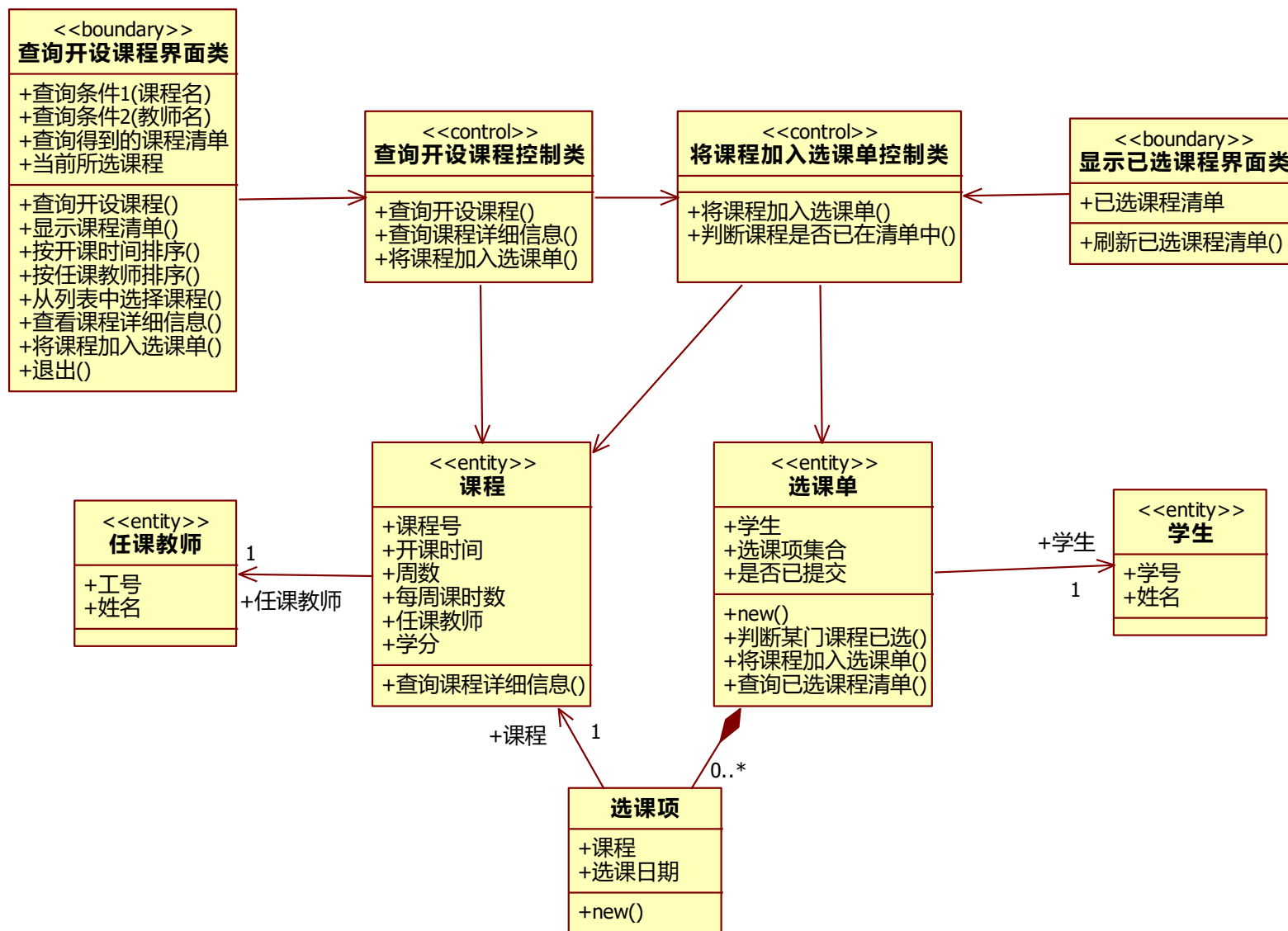
<<entity>> 选课单
+学生 +选课项集合 +是否已提交
+new() +判断某门课程已选() +将课程加入选课单() +查询已选课程清单()

选课项
+课程 +选课日期 +new()

## Step 9: 绘制分析类图

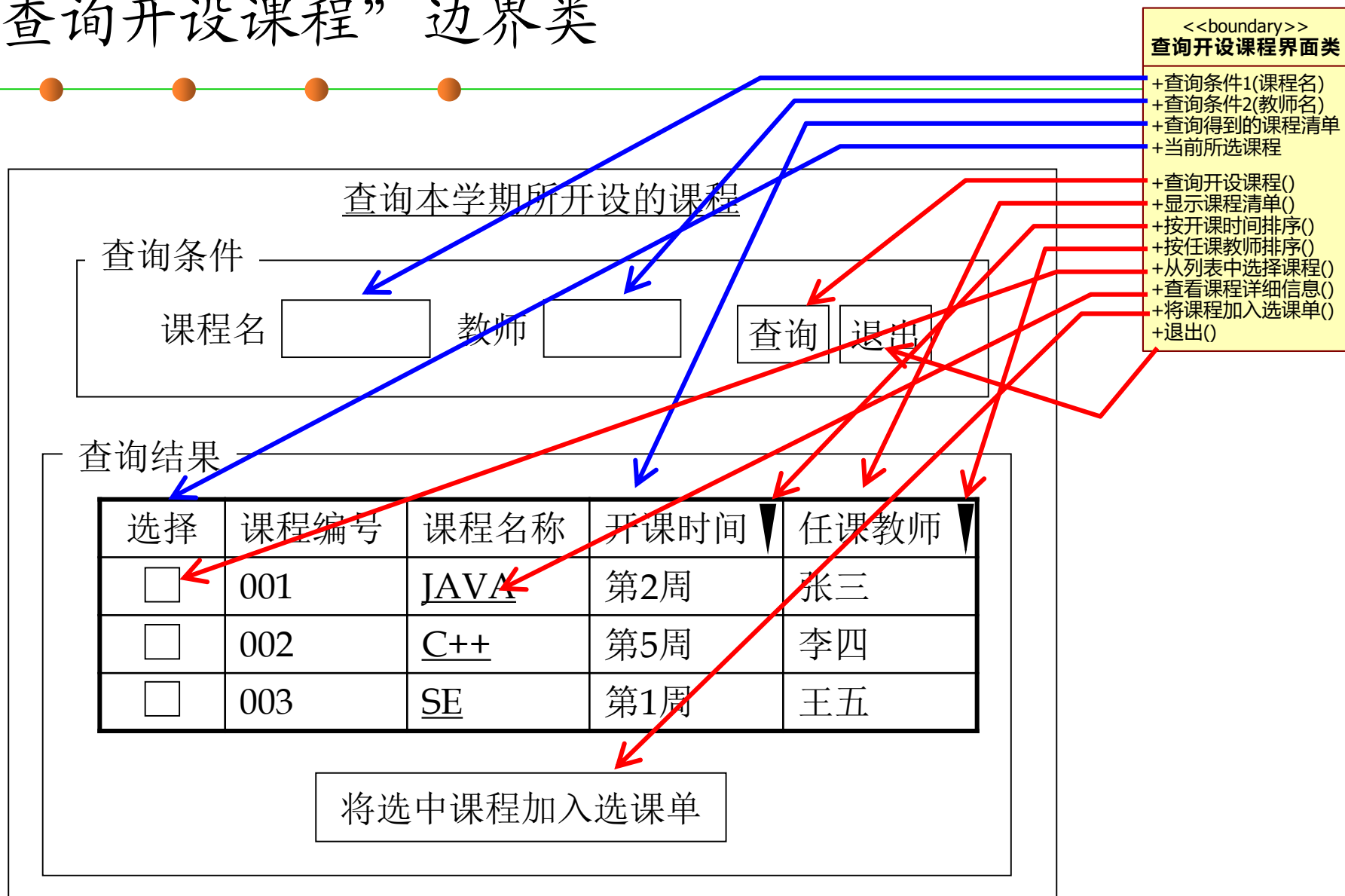


# Step 10: 绘制领域类图





# “查询开设课程” 边界类



## “查询开设课程” 控制类

<<control>>  
查询开设课程控制类

+查询开设课程()  
+查询课程详细信息()  
+将课程加入选课单()

### ■ 查询开设课程()

```
List QueryCourses(String TName, String CName) {
```

    连接数据库，从数据库中读取所有满足条件的课程；

    构造List，返回UI；

```
}
```

### ■ 将课程加入选课单()

```
void AddCourseToForm(Course c, Student s) {
```

    构造“将课程加入选课单”控制类addCourseCtrl;

    addCourseCtrl.AddCourseToForm (s, c);

```
}
```

## “将课程加入选课单” 控制类

<<control>>  
将课程加入选课单控制类

+将课程加入选课单()  
+判断课程是否已在清单中()

### ■ 将课程加入选课单()

```
Void AddCourseToForm (Student s, Course c) {
```

从数据库中查询学生s的选课单信息;

根据这些信息, 调用选课单类的new()操作, 构造选课单对象form;

调用form的“将课程加入选课单”操作, 输入参数为c;

```
}
```

## “选课单” 实体类

### ■ 判断某门课程已选()

```
Boolean isCourseSelected (Course course) {  
    循环从选课项集合中逐一取出每一项item {  
        if (item.课程 == course)  
            return true;  
    }  
    return false;  
}
```

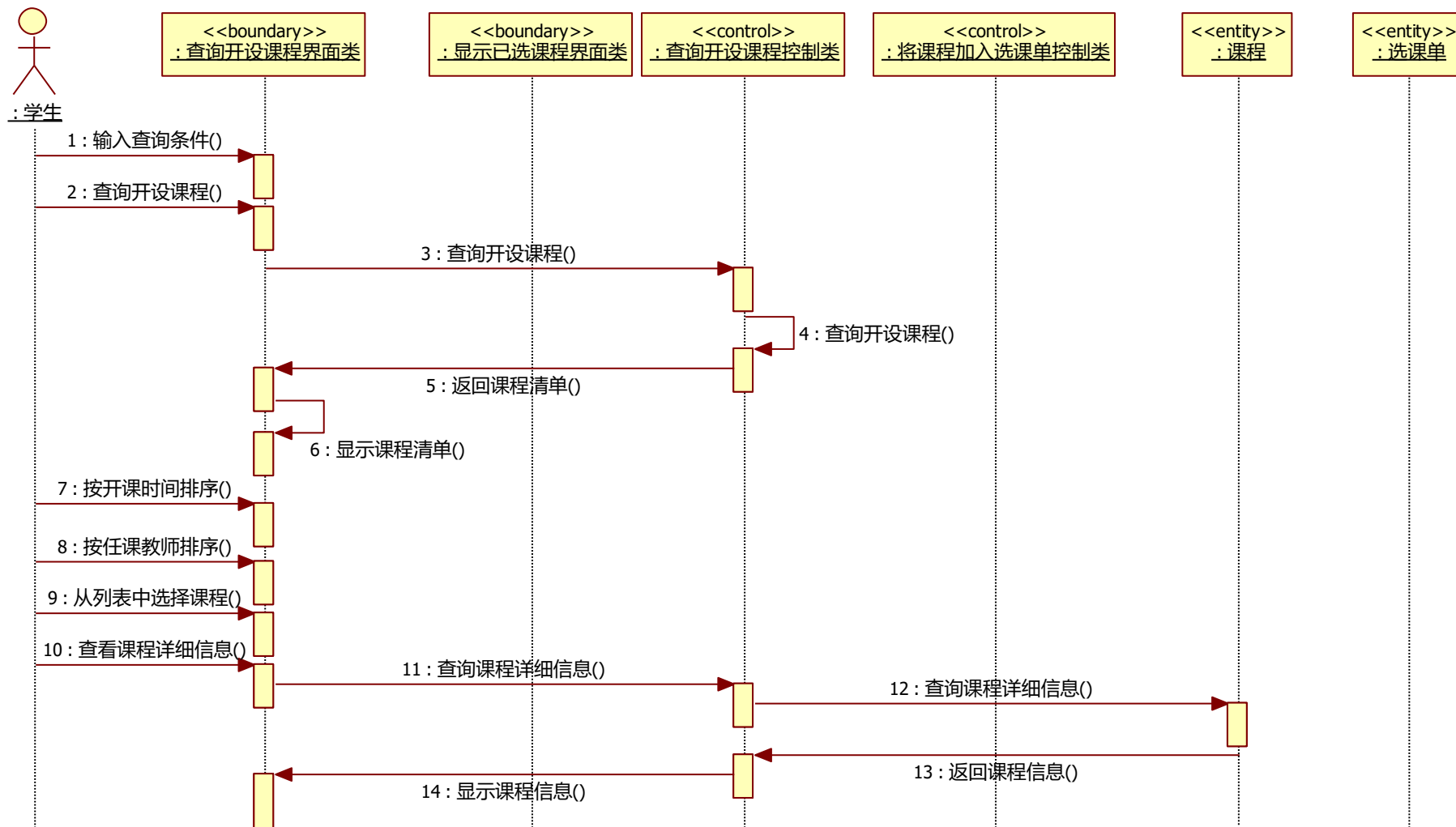
### ■ 将课程加入选课单()

```
Void AddNewCourse(Course course) {  
    Item item = new Item(course, 当前日期);  
    将item加入到self.选课项集合中;  
}
```

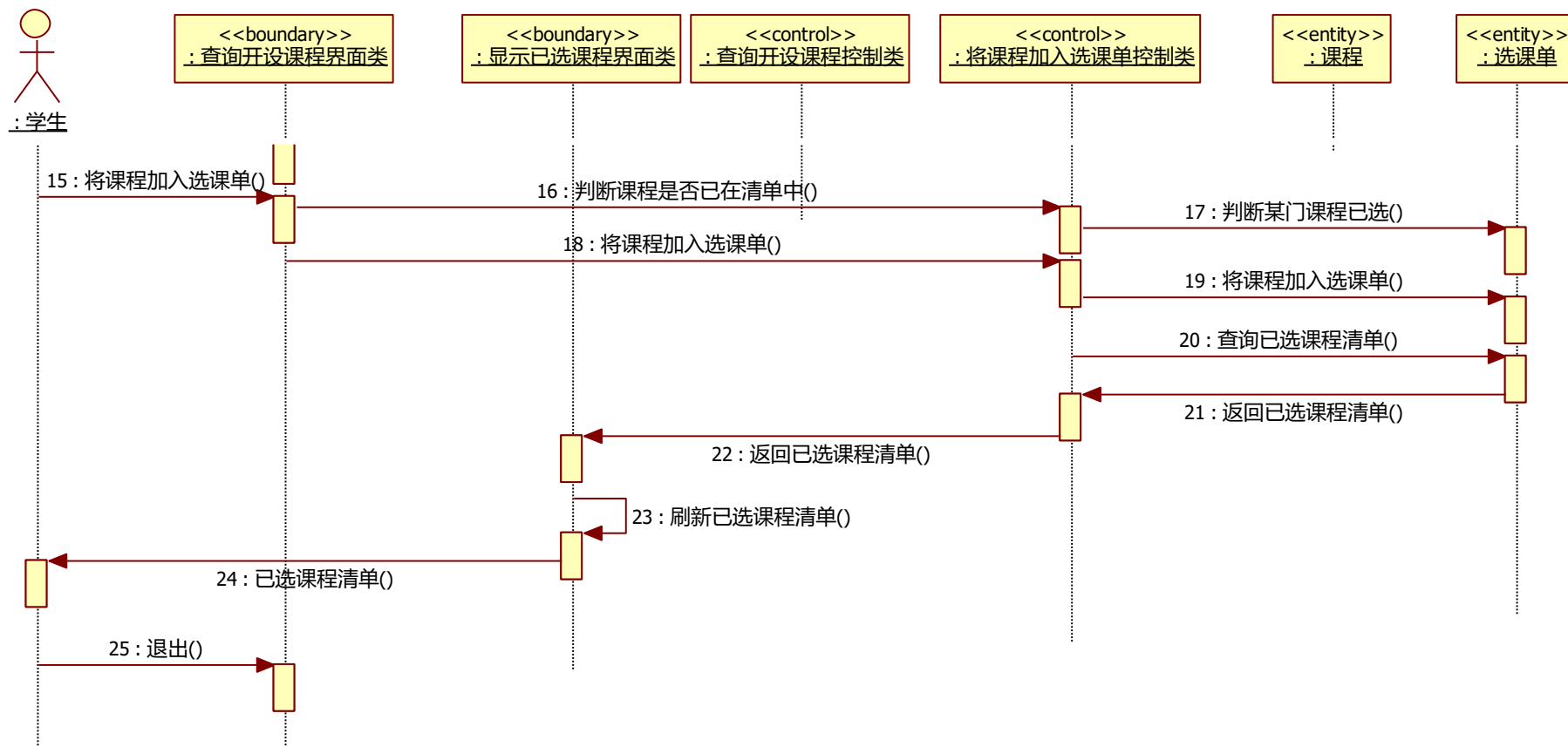
<<entity>>  
选课单

+学生  
+选课项集合  
+是否已提交  
+是否符合学分政策  
  
+new()  
+判断某门课程已选()  
+将课程加入选课单()  
+查询已选课程清单()  
+学分政策验证()  
+提交选课单()

# Step 11: 绘制时序图



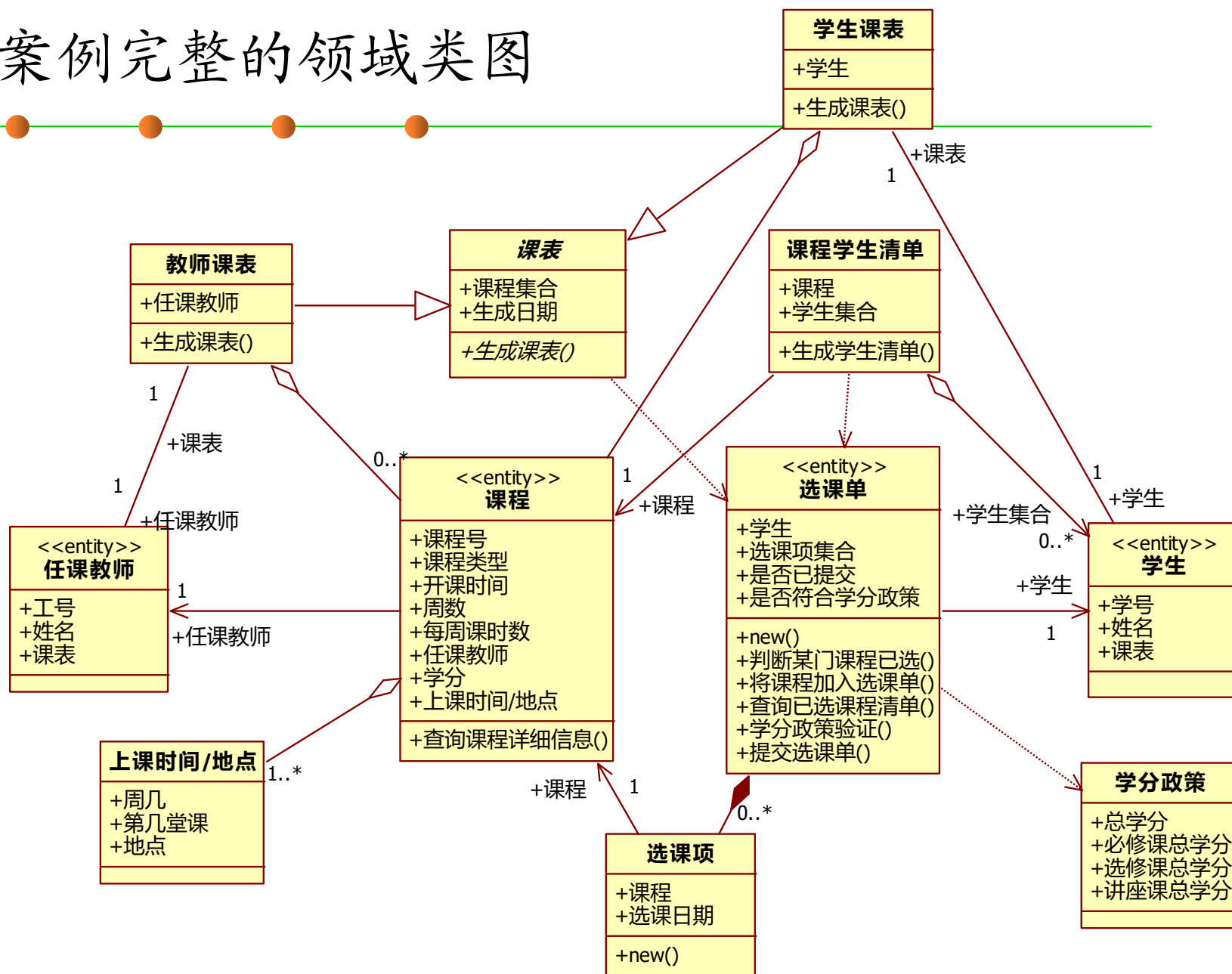
# Step 11: 绘制时序图



## 完整的实体类识别

- 教学秘书需要录入可选课程信息、任课教师信息、学分政策，并从学籍管理系统中导入学生信息；
- 教师登录进入系统，查询本学期所开设课程清单，并选择自己所承担的课程；
- 学生登录进入系统，查询本学期可选课程的清单，并创建自己的选课单，将某些课程加入到选课单中；学生可对选课单进行维护，包括加入其他课程、删除已选课程等；
- 学生也可对选课单中包含的数据进行学分政策验证，判断所选课程是否满足学校要求；
- 在规定的规定时间之前，学生将选课单做正式提交；
- 教学秘书检查每个学生的选课单，若不符合学分政策，退回重选。否则，根据所有学生提交的选课单，生成课表和每门课程的学生清单；
- 教师可查看自己承担课程的课表与学生清单，学生可查询自己的课表。

# 本案例完整的领域类图







結束

2017年10月31日