哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

软件工程
第六章 OO分析与设计
6-4 面向对象的设计

徐汉川
xhc@hit.edu.cn

2017年11月6日

主要内容

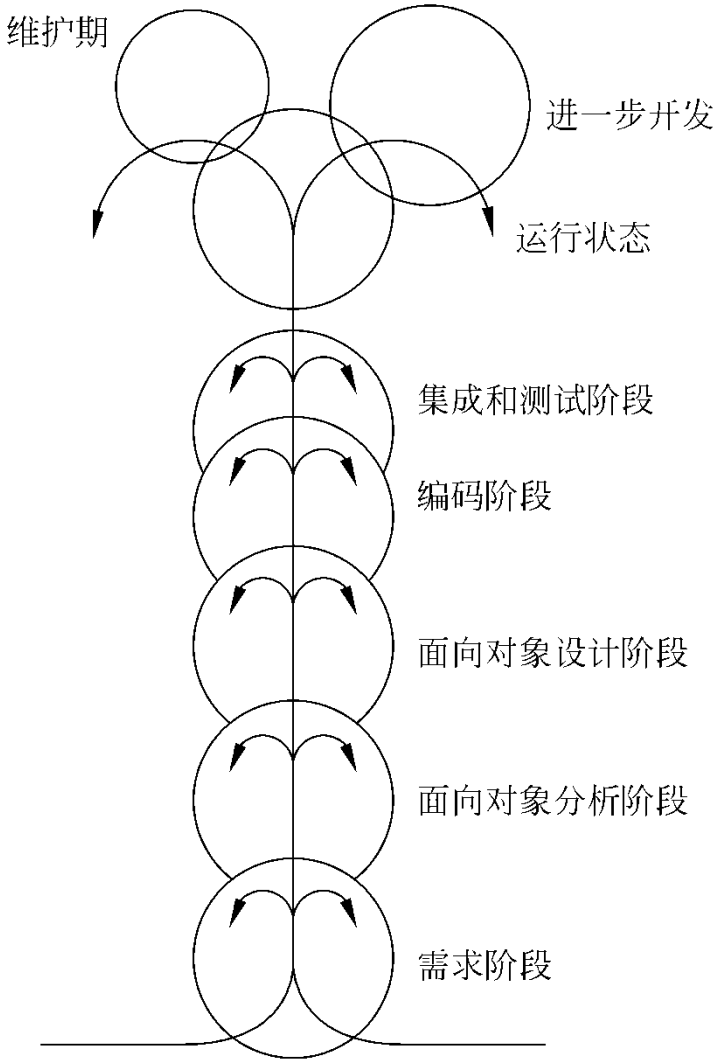
1. 面向对象设计概述
2. 系统设计
3. 包的设计
4. 数据库设计
5. 对象设计
6. 面向对象设计总结

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

1. 面向对象设计概述

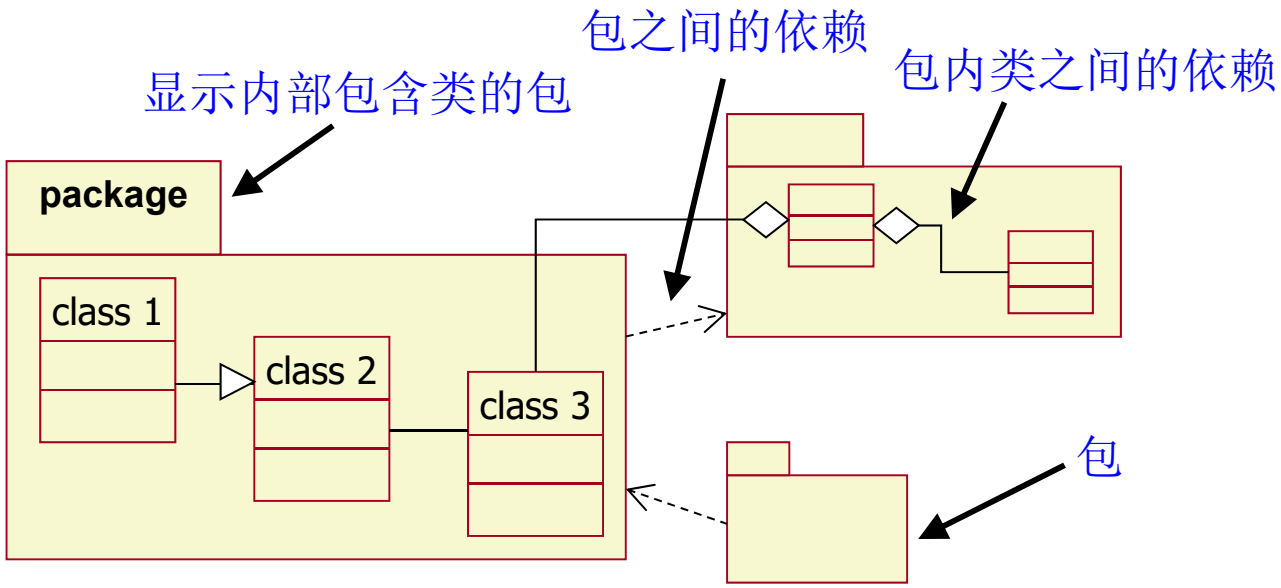
面向对象的设计

- 传统的结构化方法：分析阶段与设计阶段分得特别清楚，分别使用两套完全不同的建模符号和建模方法；
- 面向对象的设计(OOD)：OO各阶段均采用统一的“对象”概念，各阶段之间的区分变得不明显，形成“无缝”连接。
- 因此，OOD中仍然使用“类、属性、操作”等概念，是在OOA基础上的进一步细化，更加接近底层的技术实现。



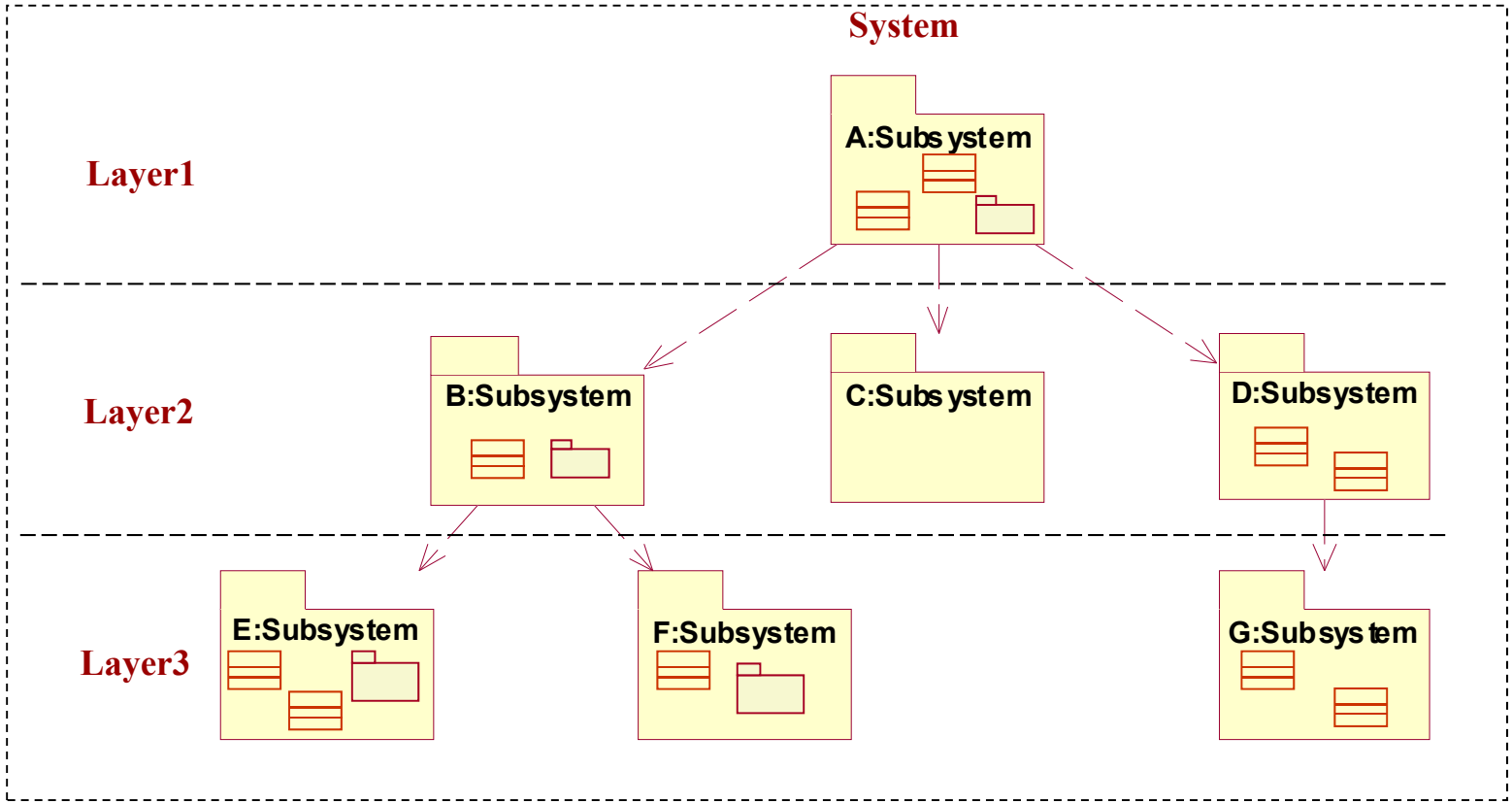
面向对象设计中的基本元素

- 基本单元：设计类(design class)　　—对应于OOA中的“分析类”
- 为了系统实现与维护过程中的方便性，将多个设计类按照彼此关联的紧密程度聚合到一起，形成大粒度的“包”(package)；



面向对象设计中的基本元素

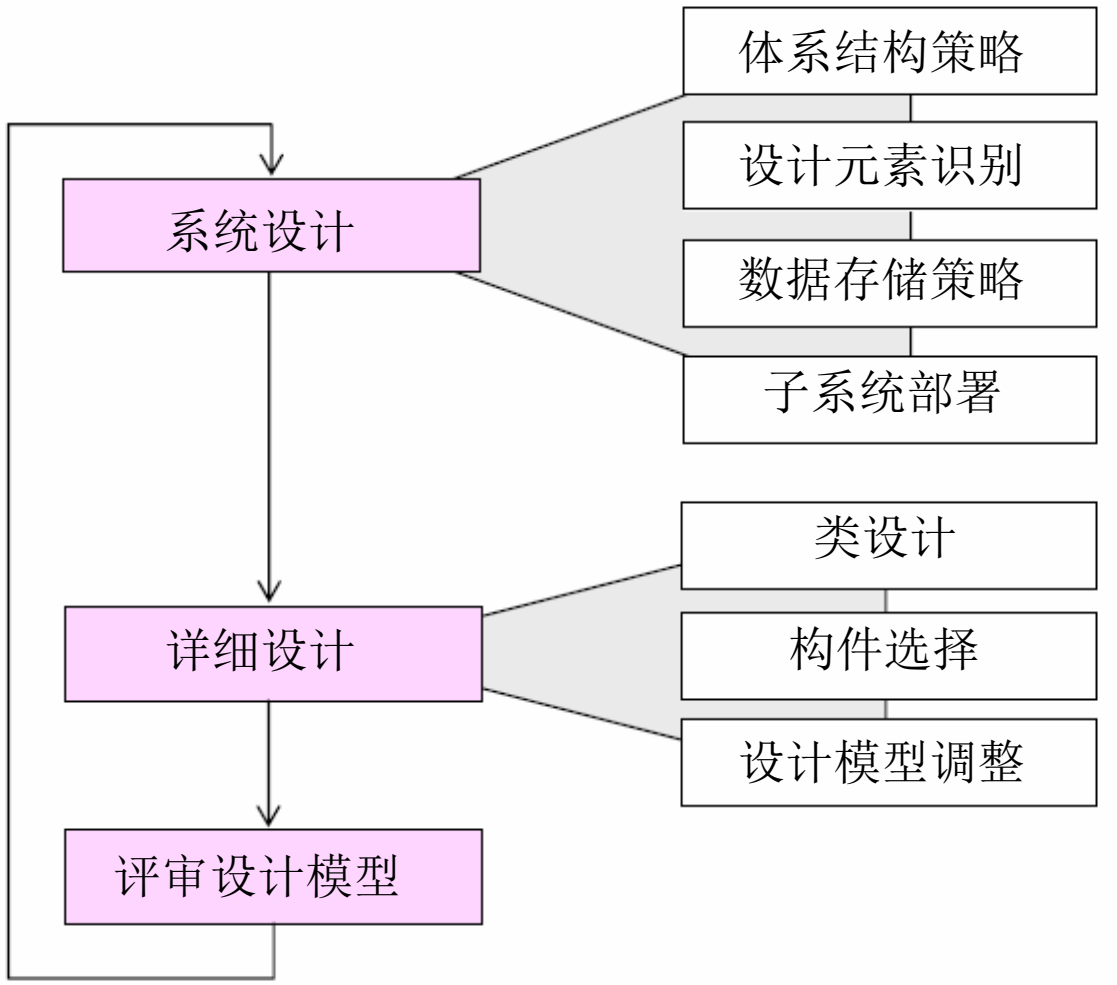
- 一个或多个包聚集在一起，形成“子系统”(sub-system)
- 多个子系统，构成完整的“系统”(system)



面向对象的设计的两个阶段

- **系统设计(System Design)**
  - 相当于概要设计（即设计系统的体系结构）；
  - 选择解决问题的基本途径；
  - 决策整个系统的结构与风格；
- **对象设计(Object Design)**
  - 相当于详细设计（即设计对象内部的具体实现）；
  - 细化需求分析模型；
  - 识别新的对象；
  - 在系统所需的应用对象与可复用的商业构件之间建立关联；
    - 识别系统中的应用对象；
    - 调整已有的构件；
    - 给出每个子系统/类的精确规格说明。

面向对象设计的过程


体系结构策略
设计元素识别
数据存储策略
子系统部署
系统设计
类设计
构件选择
设计模型调整
详细设计
评审设计模型

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2. 系统设计

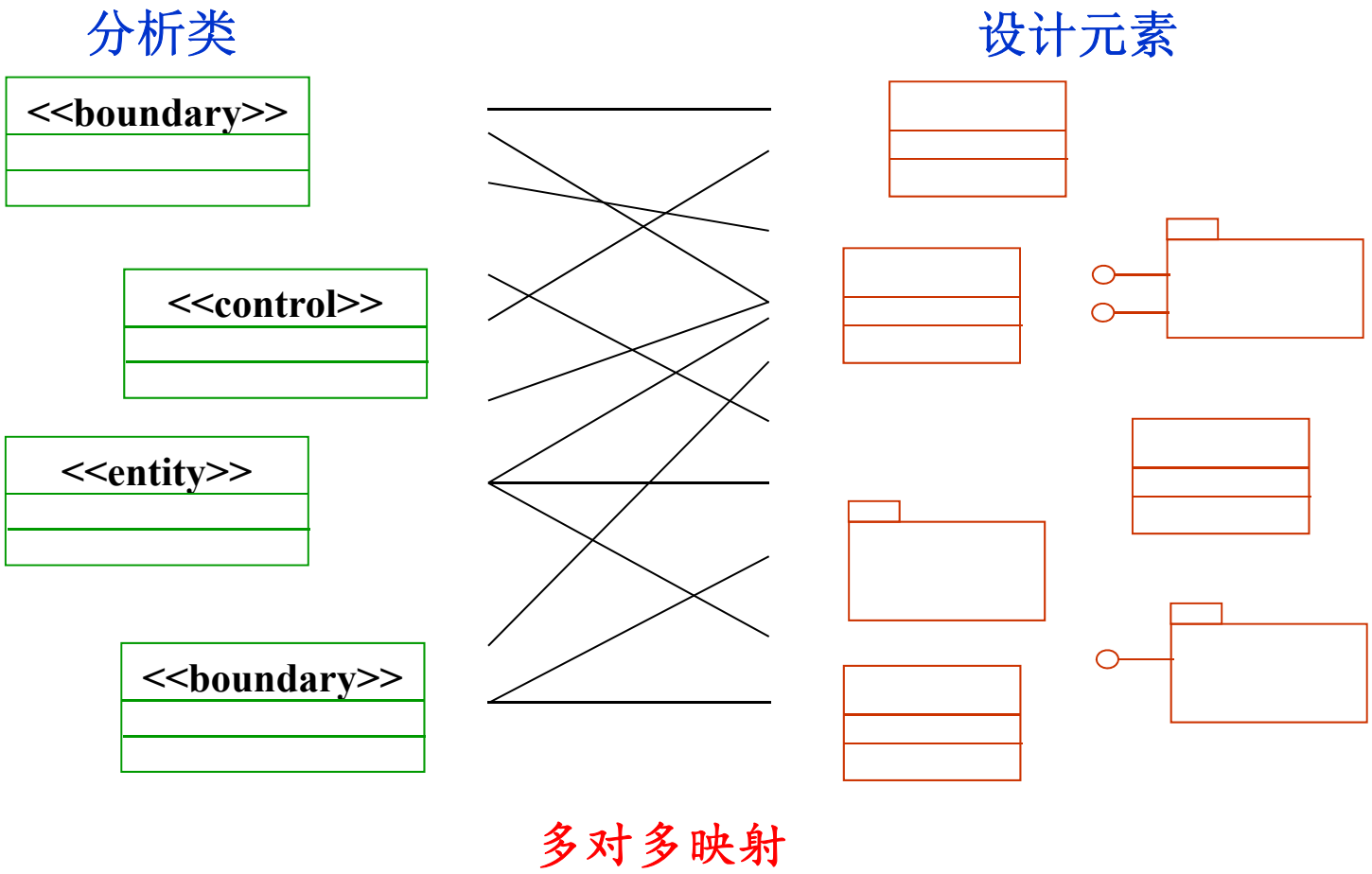
系统设计概述

- 设计系统的体系结构 --详见3-1和3-2
  - 选择合适的分层体系结构策略，建立系统的总体结构：分几层？每层的功能分别是什么？
- 识别设计元素 --详见后续章节
  - 识别“设计类”(design class)、“包”(package)、“子系统”(sub-system)
- 部署子系统 --详见3-3
  - 选择硬件配置和系统平台，将子系统分配到相应的物理节点，绘制部署图(deployment diagram)
- 定义数据的存储策略 --详见后续章节
- 检查系统设计

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

3. 包的设计：从逻辑角度

识别设计元素



确定设计元素的基本原则

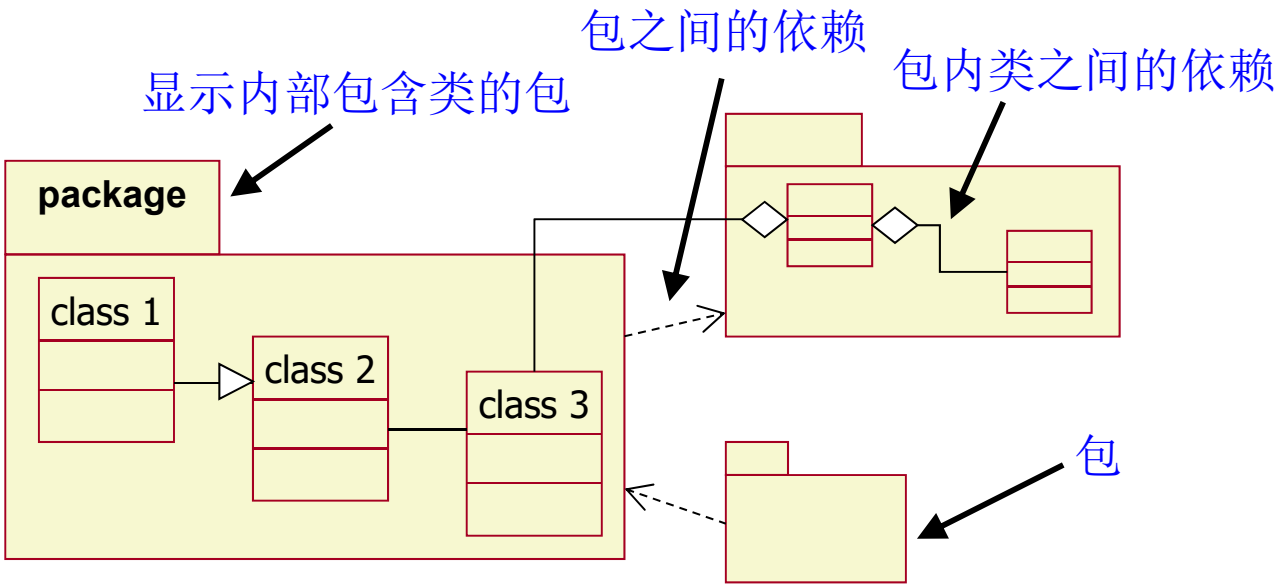
- 如果一个“分析类”比较简单，代表着单一的逻辑抽象，那么可以将其一对一的映射为“设计类”；
  - 通常，主动参与者对应的边界类、控制类和一般的实体类都可以直接映射成设计类。
- 如果“分析类”的职责比较复杂，很难由单个“设计类”承担，则应该将其分解为多个“设计类”，并映射成“包”或“子系统”；
- 将设计类分配到相应的“包”或“子系统”当中；
  - 子系统的划分应该符合高内聚低耦合的原则。

图书管理系统：识别设计元素

类型	分析类	设计元素
	***LoginForm***	“设计类” ***LoginForm***
	BrowseForm	“设计类” ***BrowseForm***
	……	……
	MailSystem	“子系统接口” ***IMailSystem***
	***BrowseControl***	“设计类” ***BrowseControl***
	MakeReservationControl	“设计类” ***MakeReservationControl***
	……	……
	***BorrowerInfo***	“设计类” ***BorrowerInfo***
	Loan	“设计类” ***Loan***
	……	……

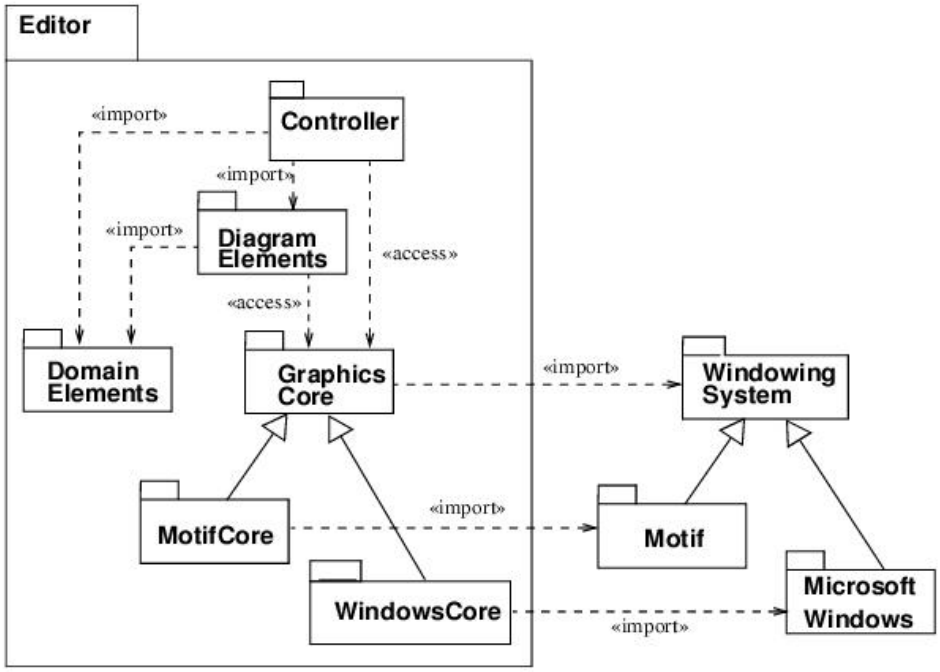
绘制包图(package diagram)

- 对一个复杂的软件系统，要使用大量的设计类，这时就必要把这些类分组进行组织；
- 把在语义上接近且倾向于一起变化的类组织在一起形成“包”，既可控制模型的复杂度，有助于理解，而且也有助于按组来控制类的可见性；
- 结构良好的包是松耦合、高内聚的，而且对其内容的访问具有严密的控制。



包之间的关系

- 类与类之间的存在的“聚合、组合、关联、依赖”关系导致包与包之间存在依赖关系，即“包的依赖”**(dependency)**；
- 类与类之间的存在的“继承”关系导致包与包之间存在继承关系，即“包的泛化”**(generalization)**；



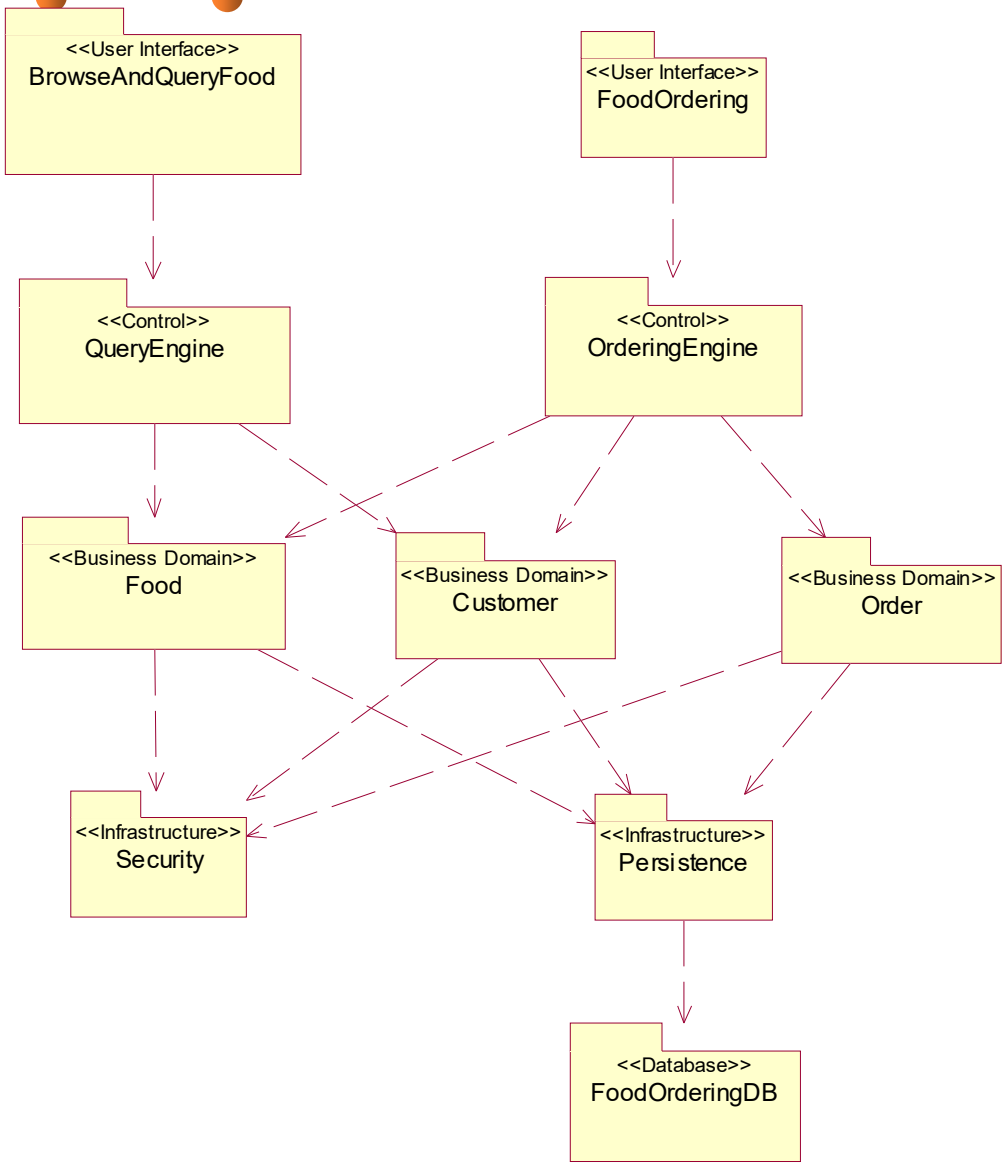
模型管理视图—包图

- 包图(**Package Diagram**)是在 **UML** 中用类似于文件夹的符号表示的模型元素的组合。
- **UML**包图提供了组织元素的方式，包能够组织任何事物：类、其它包、用例等，系统中的每个元素都只能为一个包所有，一个包可嵌套在另一个包中。注：**UML**中的包为广义概念，不等同于**Java**包的狭义概念。
  - 类
  - 接口
  - 组件
  - 节点
  - 协作
  - 用例图
  - 以及其他包
- 使用包图可以将相关元素归入一个系统。一个包中可包含附属包、图表或单个元素。

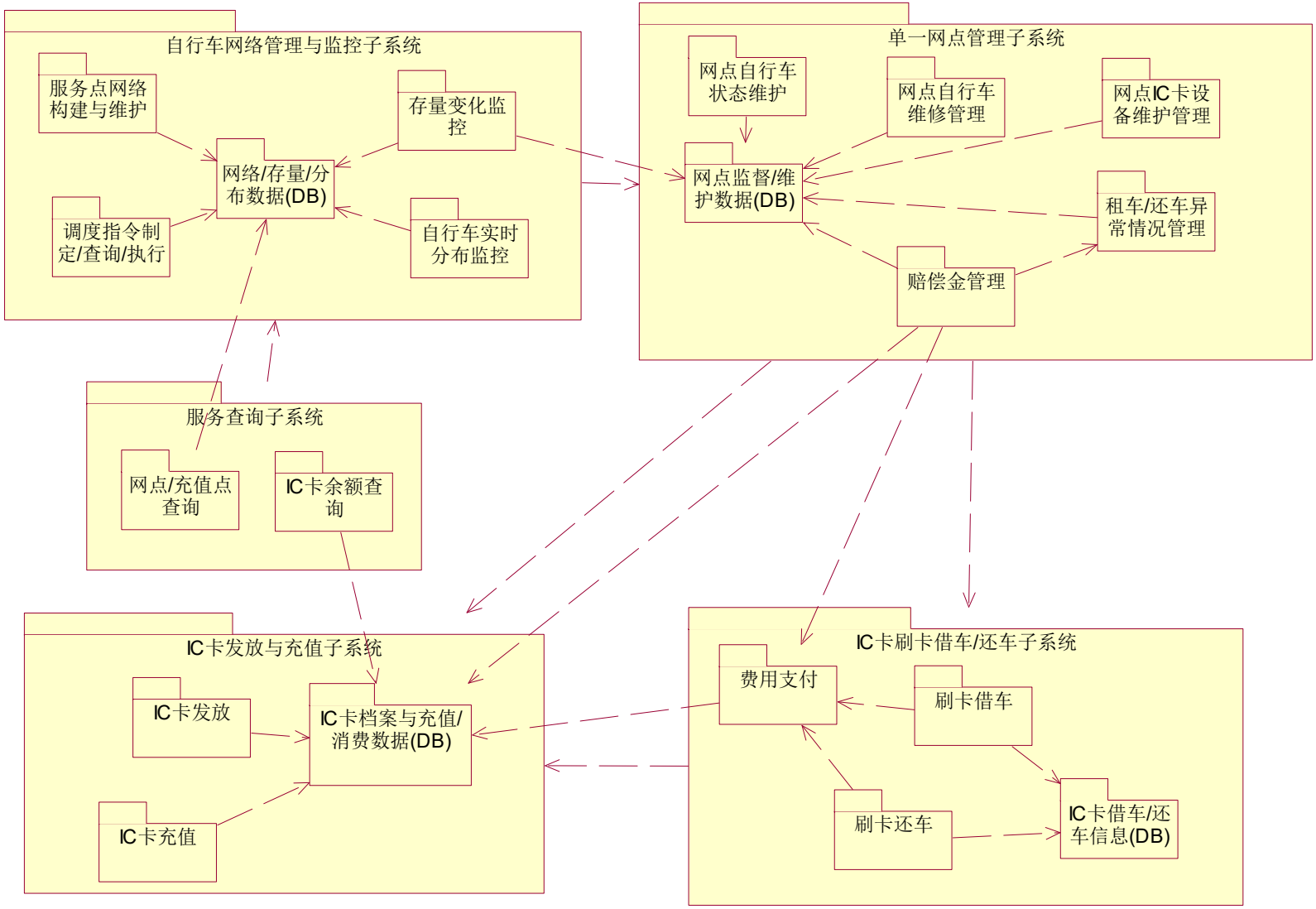
绘制包图的方法

- 分析设计类，把概念上或语义上相近的模型元素纳入一个包。
- 可以从类的功能相关性来确定纳入包中的类：
  - 如果一个类的行为和/或结构的变更要求另一个相应的变更，则这两个类是功能相关的。
  - 如果删除一个类后，另一个类便变成是多余的，则这两个类是功能相关的，这说明该剩余的类只为那个被删除的类所使用，它们之间有依赖关系。
  - 如果两个类之间大量的频繁交互或通信，则这两个类是功能相关的。
  - 如果两个类之间有一般/特殊关系，则这两个类是功能相关的。
  - 如果一个类激发创建另一个类的对象，则这两个类是功能相关的。
- 确定包与包之间的依赖关系(<<import>>、<<access>>等)；
- 确定包与包之间的泛化关系；
- 绘制包图。

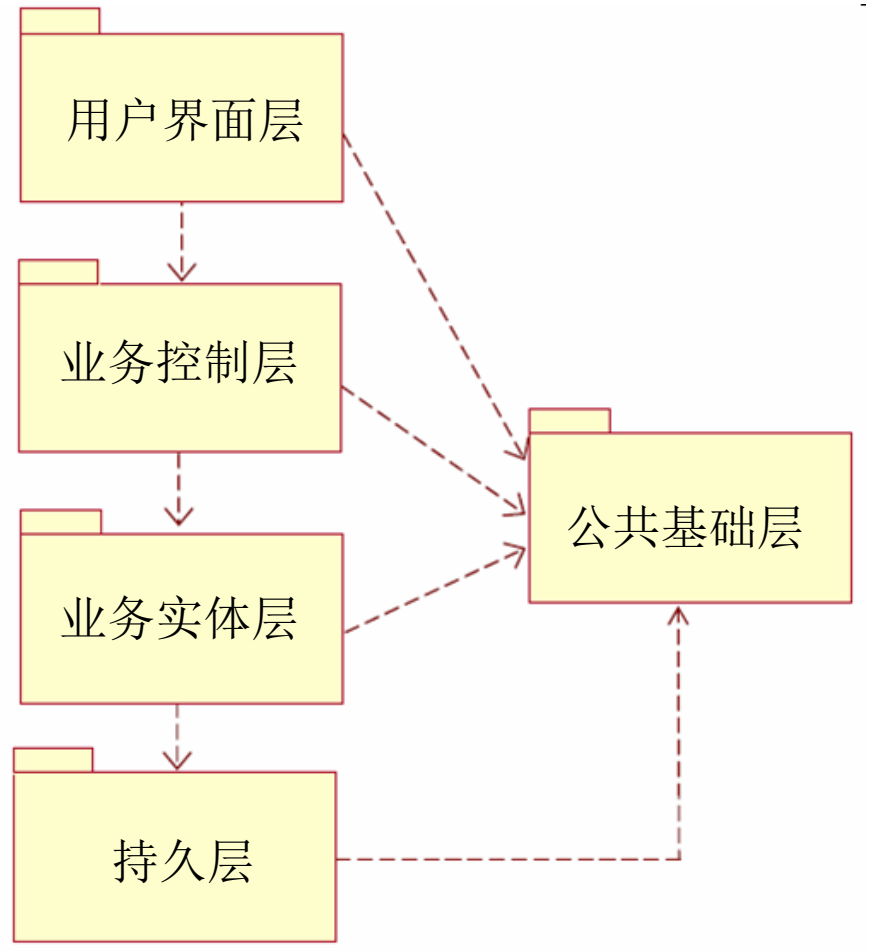
包图示例1


<<User Interface>>
BrowseAndQueryFood
<<User Interface>>
FoodOrdering
<<Control>>
QueryEngine
<<Control>>
OrderingEngine
<<Business Domain>>
Food
<<Business Domain>>
Customer
<<Business Domain>>
Order
<<Infrastructure>>
Security
<<Infrastructure>>
Persistence
<<Database>>
FoodOrderingDB

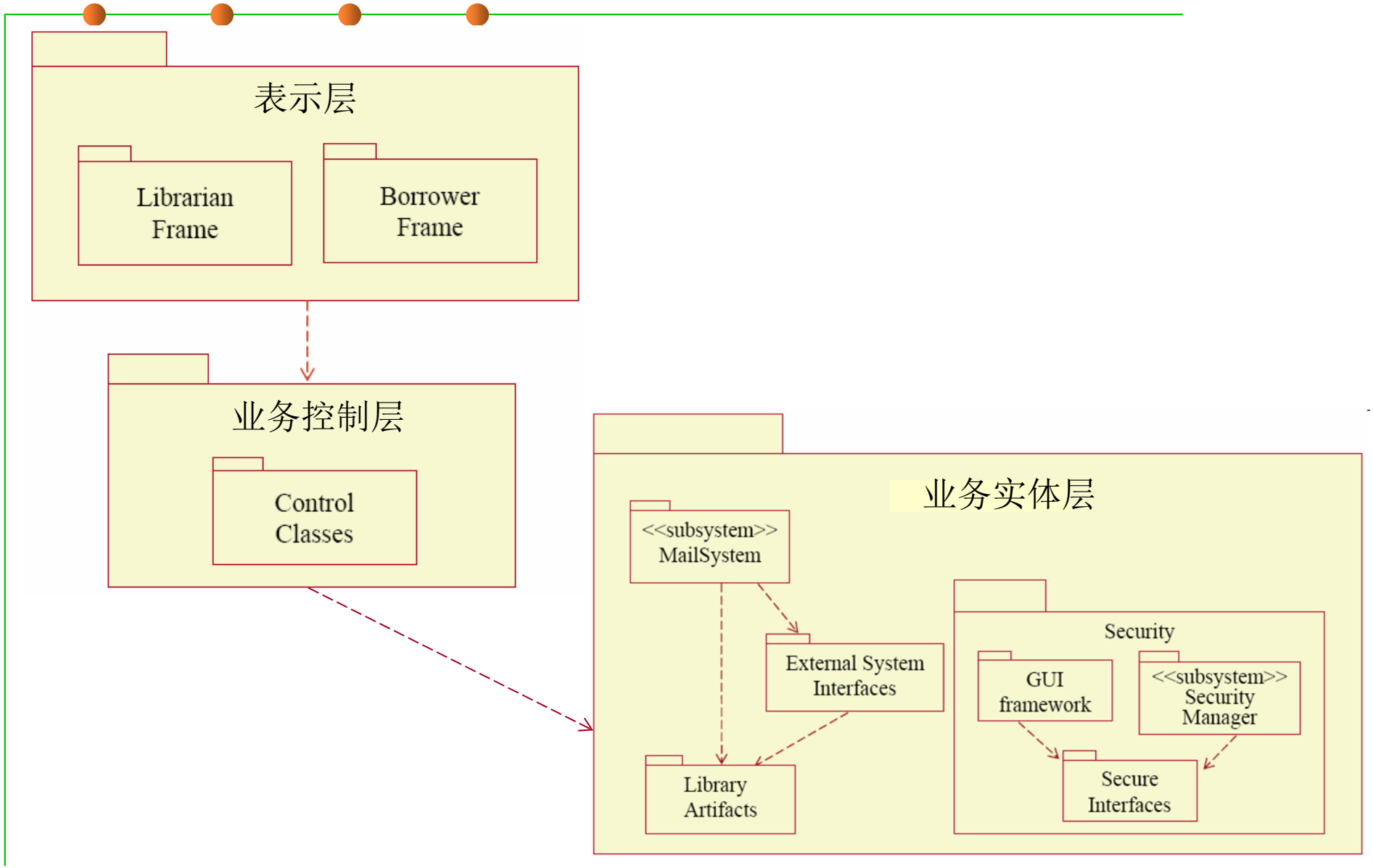
包图示例2


自行车网络管理与监控子系统
服务点网络构建与维护
存量变化监控
网络/存量/分布数据(DB)
调度指令制定/查询/执行
自行车实时分布监控
单一网点管理子系统
网点自行车状态维护
网点自行车维修管理
网点IC卡设备维护管理
网点监督/维护数据(DB)
租车/还车异常情况管理
赔偿金管理
服务查询子系统
网点/充值点查询
IC卡余额查询
IC卡发放与充值子系统
IC卡发放
IC卡档案与充值/消费数据(DB)
IC卡充值
IC卡刷卡借车/还车子系统
费用支付
刷卡借车
刷卡还车
IC卡借车/还车信息(DB)

按实现技术划分包的分层结构



图书管理系统：软件体系结构



JAVA中的“package”

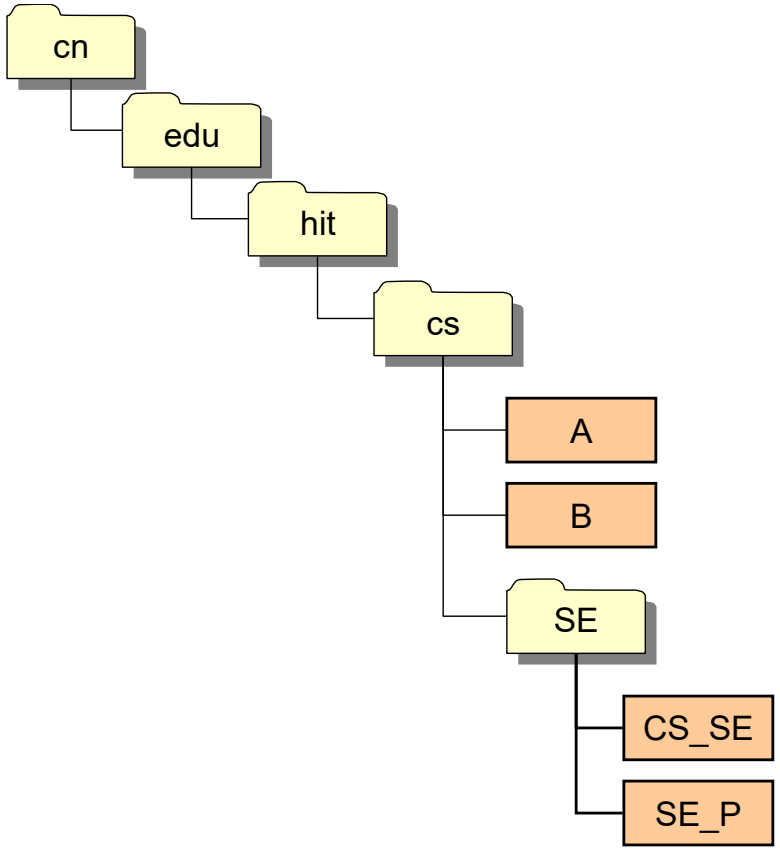
```
java.io.InputStream is = java.lang.System.in;
java.io.InputStreamReader isr= new java.io.InputStreamReader(is);
java.io.BufferedReader br = new java.io.BufferedReader(isr);
```

```
import java.lang.System;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.BufferedReader;

InputStream = System.in;
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
```

JAVA中的“package”

```
package cn.edu.hit.cs;
public class A{}
-----------------------------------
package cn.edu.hit.cs;
public class B{}
-----------------------------------
package cn.edu.hit.cs.se;
public class CS_SE{}
-----------------------------------
package cn.edu.hit.cs.se;
public class CS_SE_P{}
```



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

4. 数据库设计

课堂讨论

- 先从数据库设计入手，还是先从OO设计入手？ 1组

类和关系数据表的关系

- **OOP以class为基本单位，所有的object都是运行在内存空间当中；**
- **若某些object的信息需要持久化存储，那么就需要用到database，将object的属性信息写入关系数据表；**
  - 假如淘宝没有“保存购物车内容”的功能（意即若不购买，下次进入之后购物车中的内容就被清空），那么“购物车”这个实体的属性就不需要关系数据表。
- **在系统执行某些功能的时候，需要首先从database中将信息取出，使用各实体类的new操作构造相应的object，在程序运行空间中使用(充分利用继承/组合/聚合/关联/依赖关系在各object之间相互导航)。**
- **在进行OO分析和设计时完全可以将database忘掉，后续再加以考虑。**

数据存储设计

- “对象”只是存储在内存当中，而某些对象则需要永久性的存储起来；

  ——持久性数据(persistent data)

- 数据存储策略
  - **数据文件：**由操作系统提供的存储形式，应用系统将数据按字节顺序存储，并定义如何以及何时检索数据。
  - **关系数据库：**数据是以表的形式存储在预先定义好的成为Schema 的类型中。
  - **面向对象数据库：**将对象和关系作为数据一起存储。

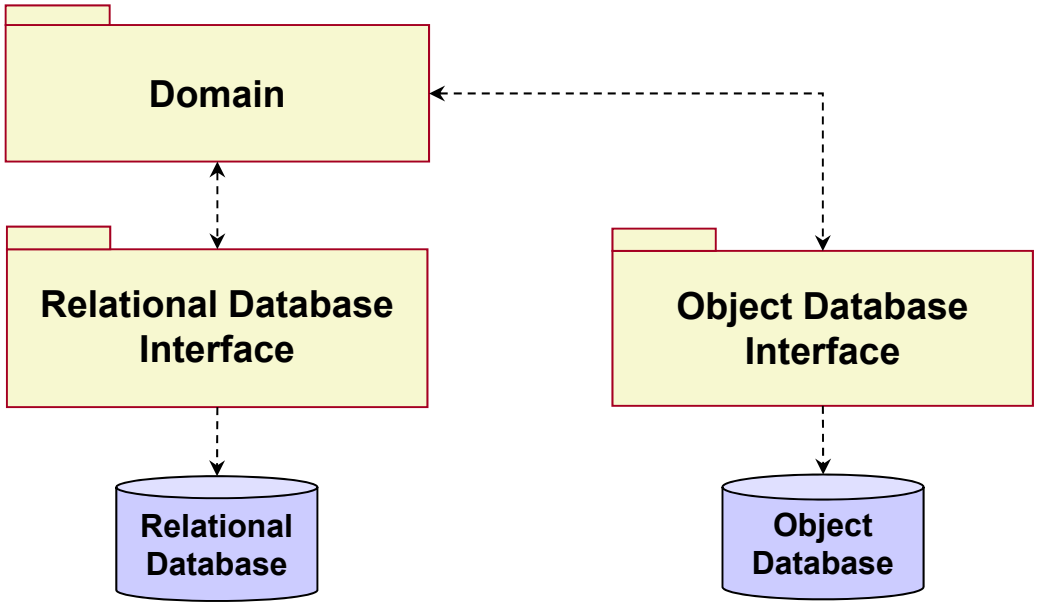
  - **存储策略的选择：**取决于非功能性的需求；

数据存储策略的tradeoff

- 何时选择文件？
  - 存储大容量数据、临时数据、低信息密度数据
- 何时选择数据库？
  - 并发访问要求高、系统跨平台、多个应用程序使用相同数据
- 何时选择关系数据库？
  - 复杂的数据查询
  - 数据集规模大
- 何时选择面向对象数据库？
  - 数据集处于中等规模
  - 频繁使用对象间联系来读取数据

数据存储策略

- 如果使用OO数据库，那么数据库系统应提供一个接口供应用系统访问数据；
- 如果使用关系数据库，那么需要一个子系统来完成应用系统中的对象和数据库中数据的映射与转换。



OO设计中的数据库设计

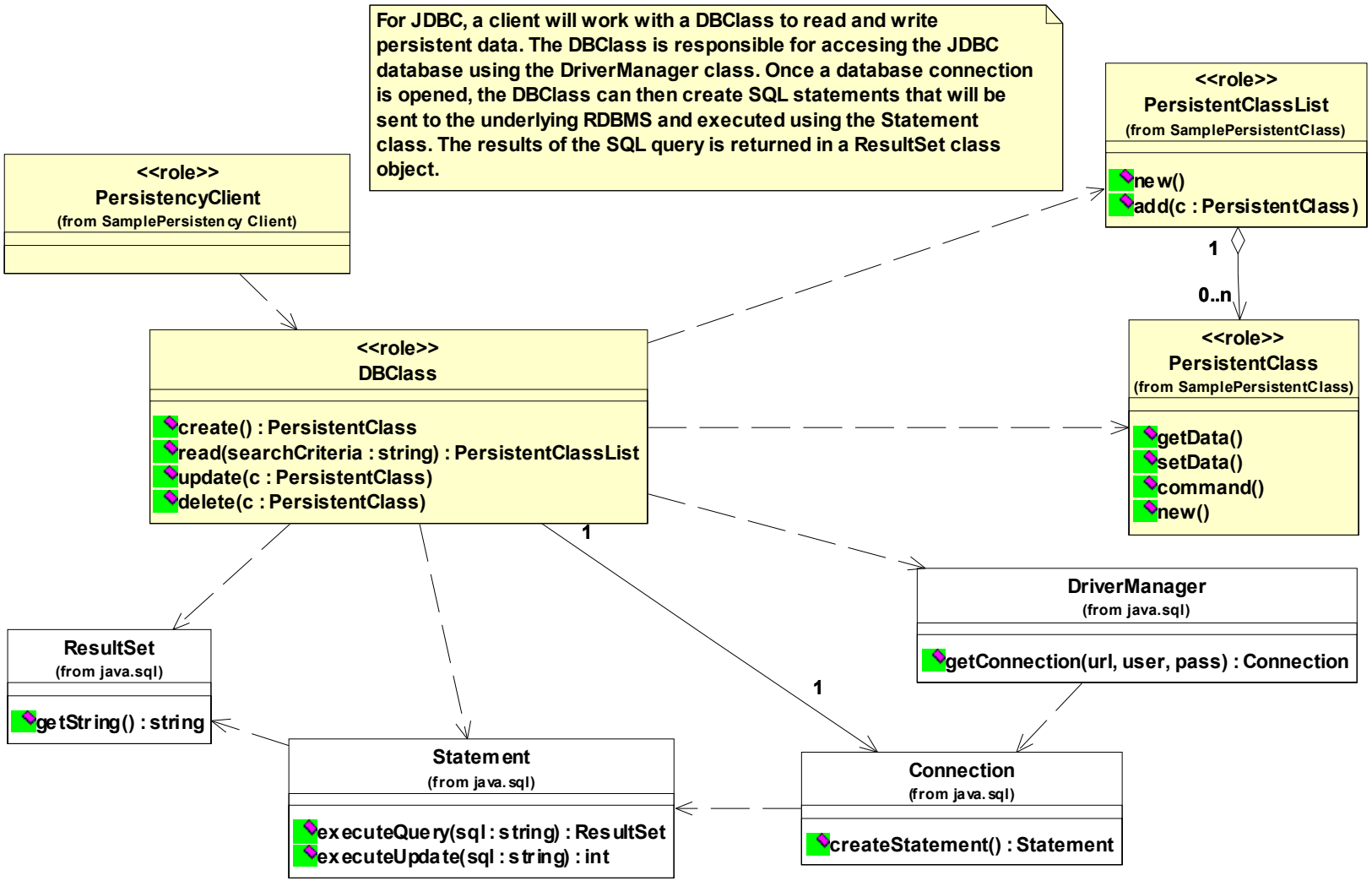
- 核心问题：对那些需要永久性存储的数据，如何将UML类图映射为数据库模型。
- 本质：把每一个类、类之间的关系分别映射到一张表或多张表；
- **UML class diagram → Relation DataBase (RDB)**
- 两个方面：
  - 将类(class)映射到表(table)
  - 将关联关系(association)映射到表(table)

对象关系映射(ORM)

- **对象关系映射(Object Relational Mapping, ORM)：**
  - 为了**解决面向对象与关系数据库存在的互不匹配**的现象；
  - 通过使用描述对象和数据库之间映射的元数据，将OO系统中的对象自动持久化到关系数据库中。

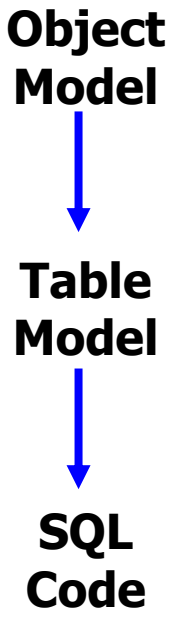
- **目前流行的ORM产品：**
  - Apache OJB
  - Hibernate
  - JPA(Java Persistence API)
  - Oracle TopLink
  - …

Example: Persistency:RDBMS:JDBC



将对象映射到关系数据库

- 最简单的映射策略——“一类一表”：表中的字段对应于类的属性，表中的每一行数据记录对应类的实例(即对象)。



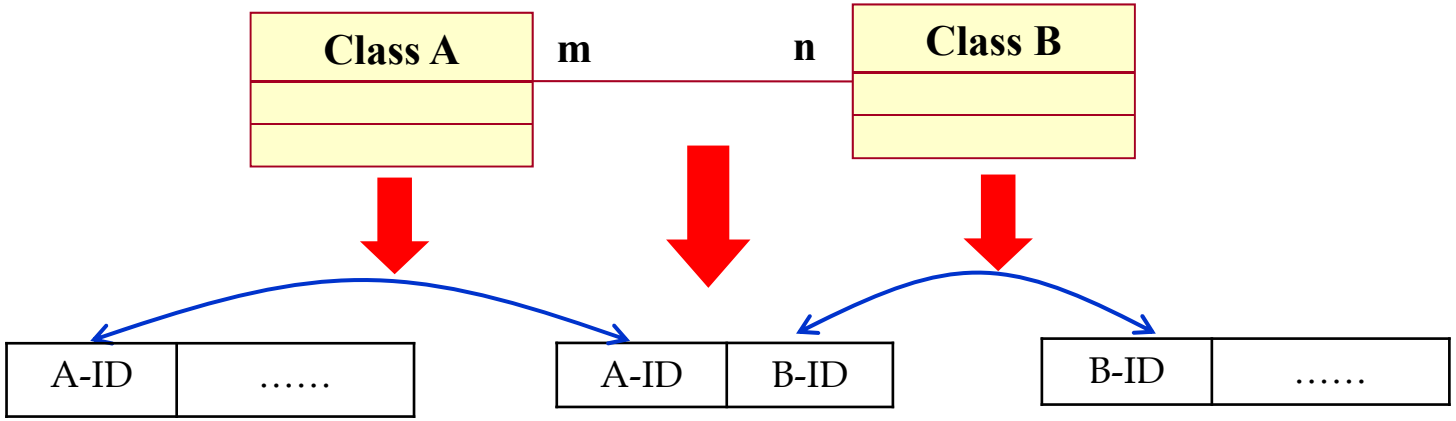
Person
name address

Attribute Name	nulls	Datatype
Person_ID Person_name address	N N Y	int Char(20) Char(50)

```
CREATE TABLE Person (person_ID int not null;
person_name char(20) notnull; address char(50);
PRIMARY KEY Person_ID);
```

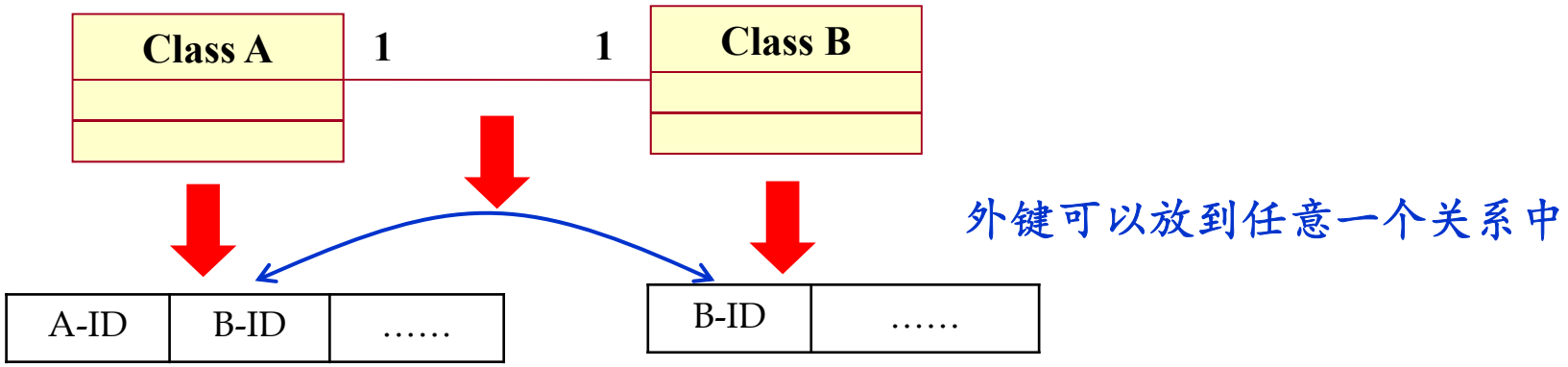
将关联映射到关系数据库(统一的、简单的途径)

- **不管是1:1、1:n还是m:n的关联关系，均可以采用以下途径映射为关系数据表：**
  - A与B分别映射为独立的数据表，然后再加入一张新表来存储二者之间的关联；



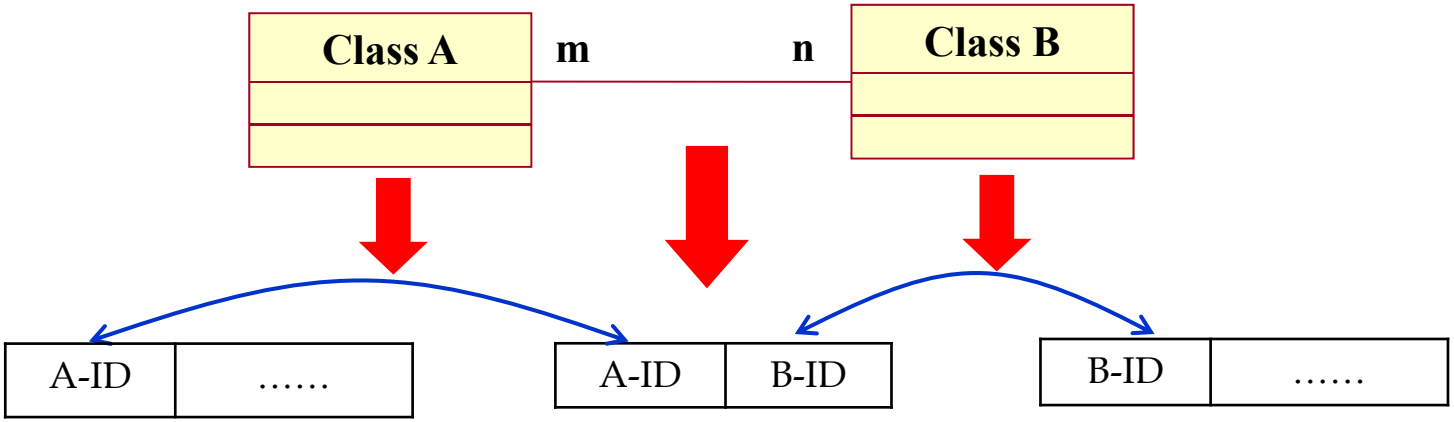
将关联映射到关系数据库(1:1和m:n的关联关系)

- **Implementing 1 to 1**



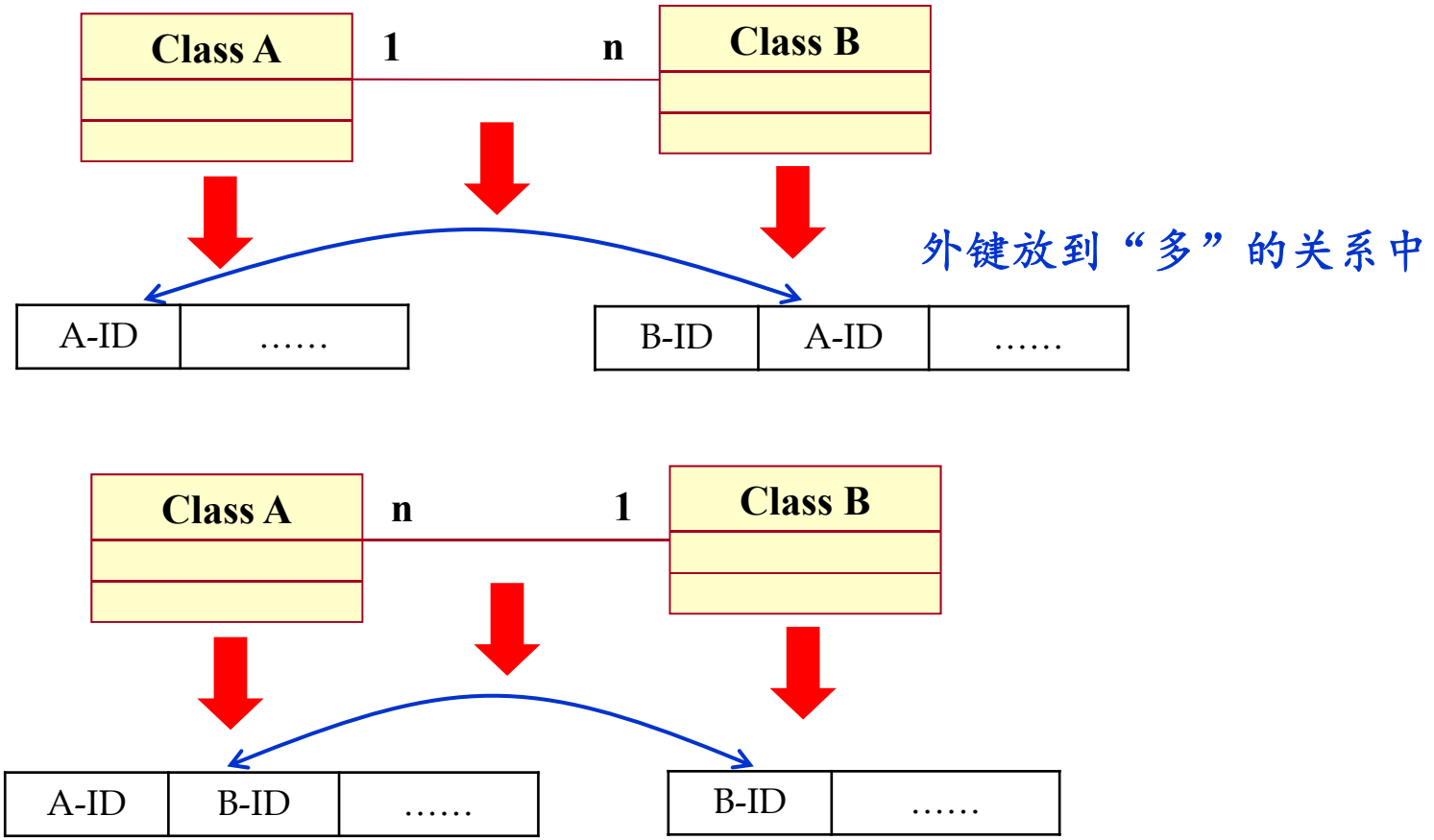
外键可以放到任意一个关系中

- **Implementing m:n**

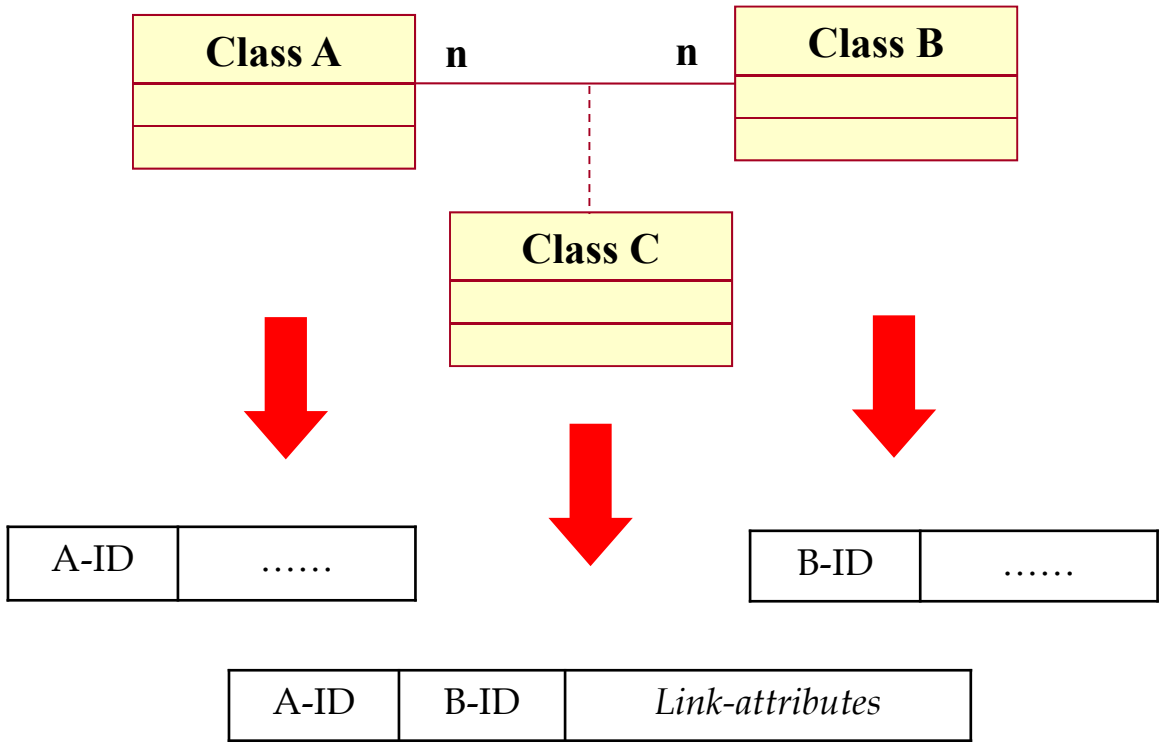


将关联映射到关系数据库(1:n的关联关系)

- **Implementing 1:n**



将关联映射到关系数据库(基于关联类的关联关系)



将关联映射到关系数据库(基于关联类的关联关系)

Company_Table

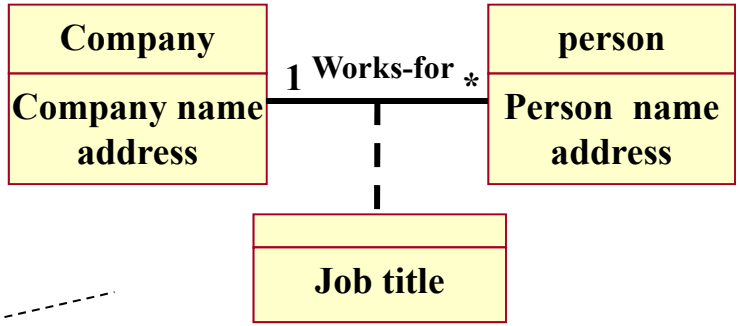
Attribute Name	nulls	Domain
Company_name address	N N	Char Char

Person_Table

Attribute Name	nulls	Domain
Person_ID Person_name address	N N N	Char Char char

Job_Table

Attribute Name	nulls	Domain
Company _name Person_ID Job title	N N y	Char Integer string



Company_Table

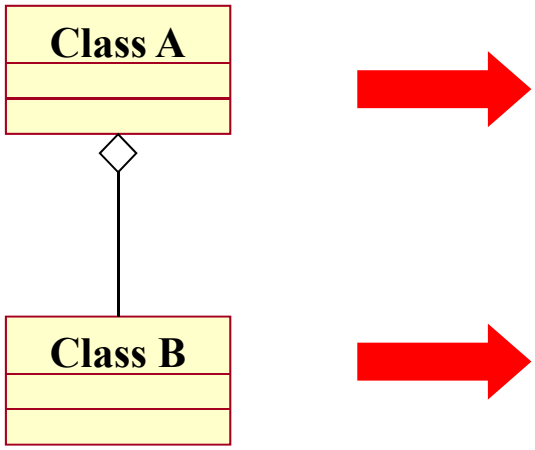
Attribute Name	nulls	Domain
Company_name address	N N	Char Char

Person_Table 2

Attribute Name	nulls	Domain
Person_ID Person_name Address Company_name Job title	N N N N Y	ID Char Char Char String

将组合/聚合关系映射到关系数据库

- **实现方法：类似于1:n的关联关系**
  - 建立“整体”表
  - 建立“部分”表，其关键字是两个表关键字的组合



“整体”表

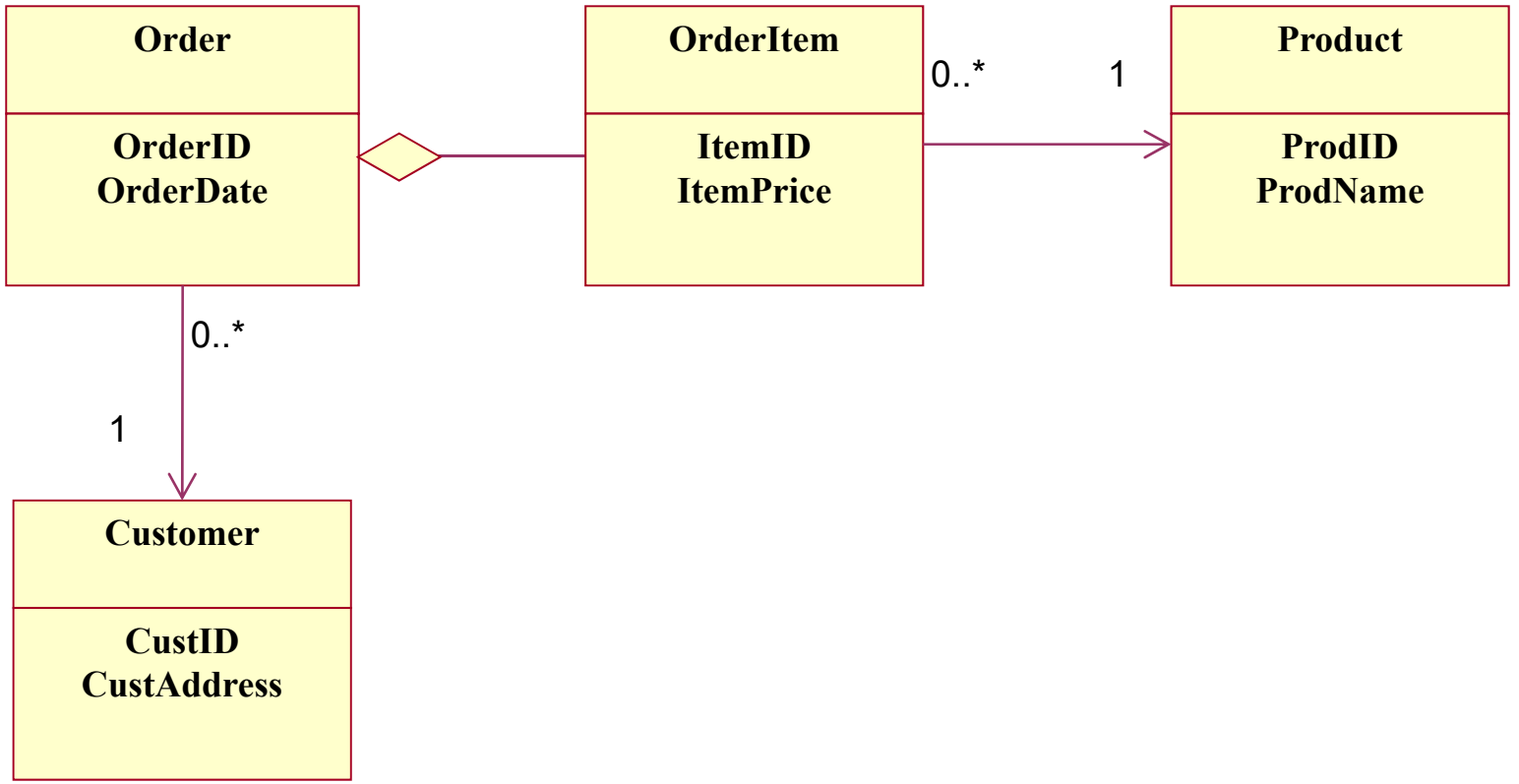
A-ID	X	Y
:	:	:

“部分”表

A-ID	B-ID	x	y
:	:	:	:

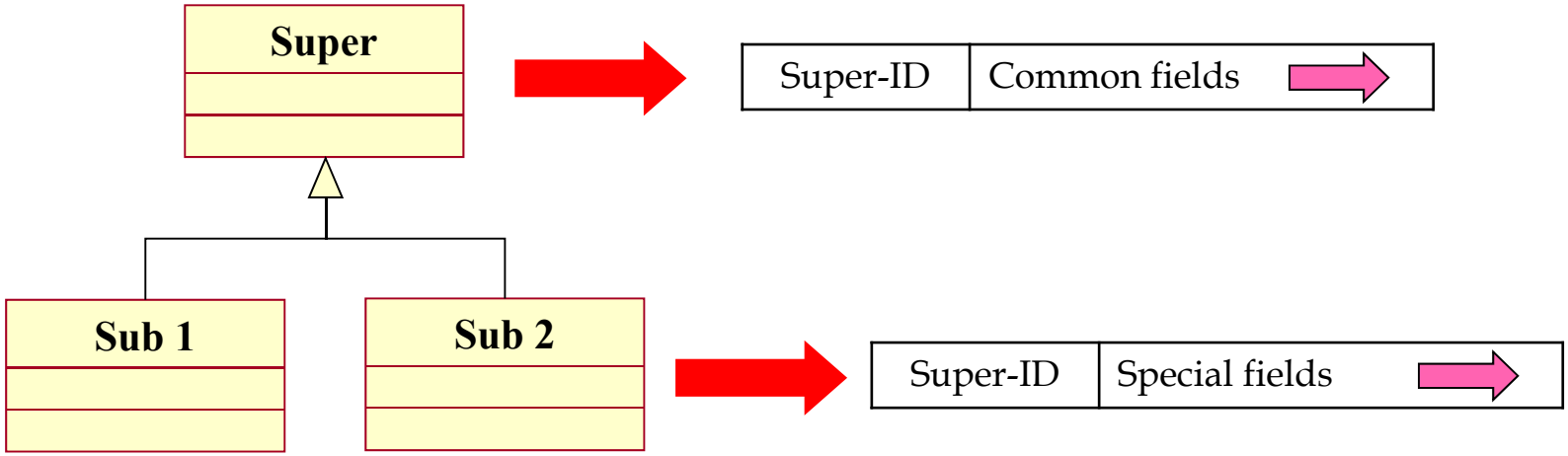
OO到DB的映射

- 为以下类图设计关系数据表　　讨论



将继承关系映射到关系数据库

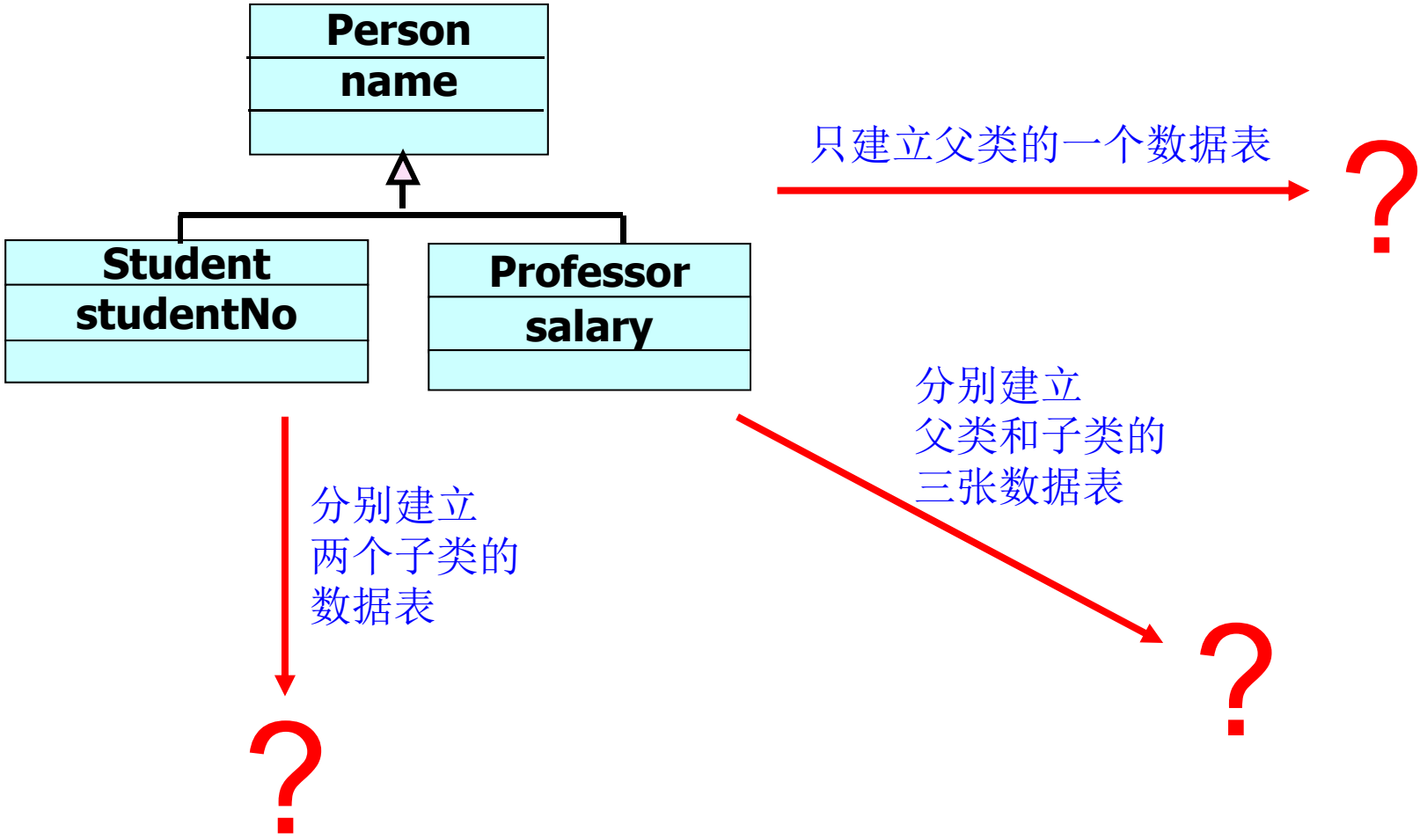
- 策略1：分别建立父类和子类的三张数据表



Requires a join to get the object

- 策略2：将子类的属性上移到父类所对应的数据表中，该表包括父类的属性、各子类的全部属性；
- 策略3：将父类的属性下移到各个子类所对应的数据表中

OO到DB的映射


Person
name
Student
studentNo
Professor
salary
只建立父类的一个数据表
?
分别建立
父类和子类的
三张数据表
?
分别建立
两个子类的
数据表
?

检查系统设计

- 检查“正确性”
  - 每个子系统都能追溯到一个用例或一个非功能需求吗？
  - 每一个用例都能映射到一个子系统吗？
  - 系统设计模型中是否提到了所有的非功能需求？
  - 每一个参与者都有合适的访问权限吗？
  - 系统设计是否与安全性需求一致？
- 检查“一致性”
  - 是否将冲突的设计目标进行了排序？
  - 是否有设计目标违背了非功能需求？
  - 是否存在多个子系统或类重名？

检查系统设计

- **检查“完整性”**
  - 是否处理边界条件？
  - 是否有用例走查来确定系统设计遗漏的功能？
  - 是否涉及到系统设计的所有方面(如硬件部署、数据存储、访问控制、遗留系统、边界条件)？
  - 是否定义了所有的子系统？
- **检查“可行性”**
  - 系统中是否使用了新的技术或组件？是否对这些技术或组件进行了可行性研究？
  - 在子系统分解环境中检查性能和可靠性需求了吗？
  - 考虑并发问题了吗？

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

5. 对象设计

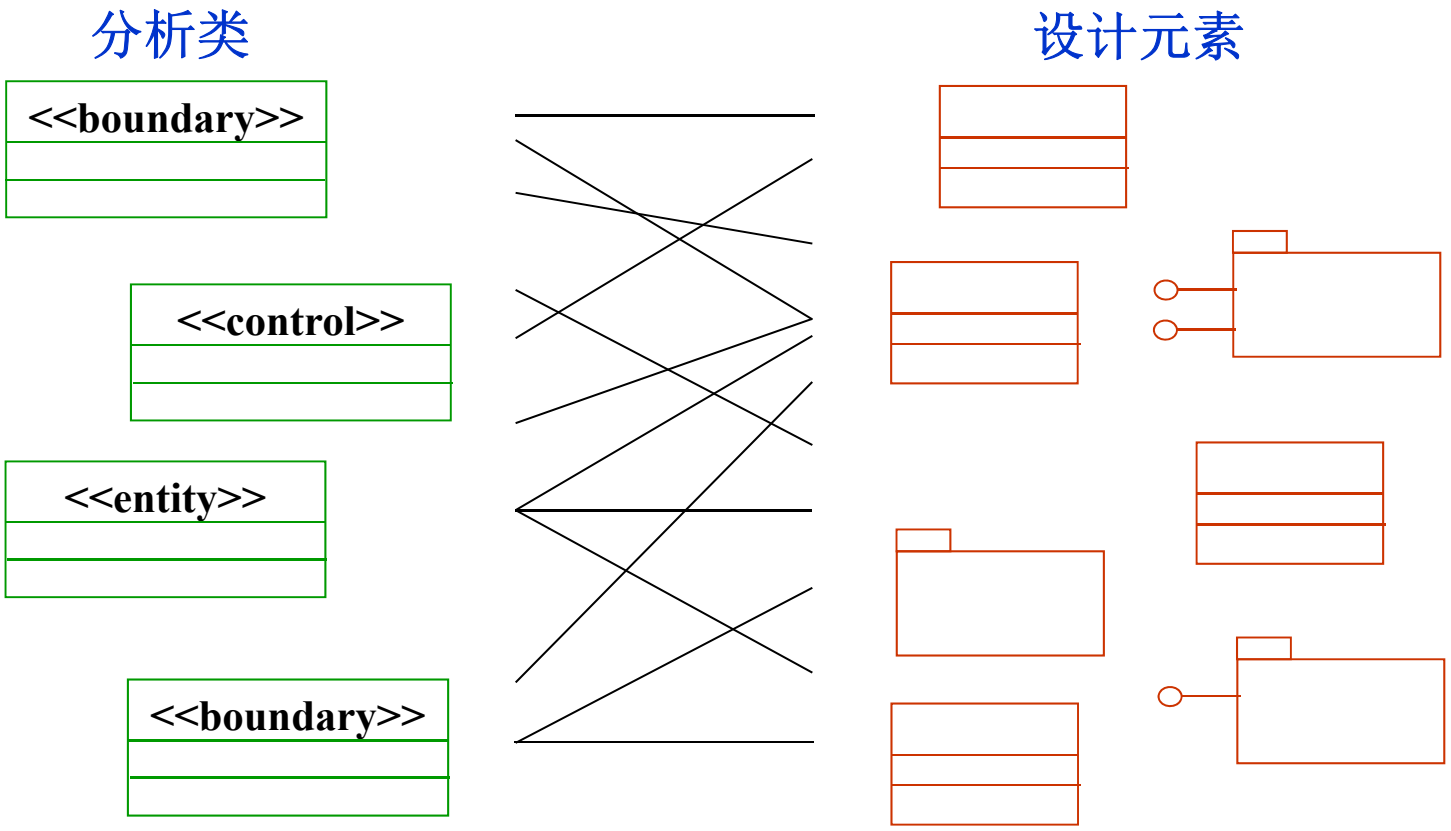
对象设计概述

- **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. 细化依赖关系
- 6. 细化关联关系
- 7. 细化泛化关系

1. 创建初始的设计类

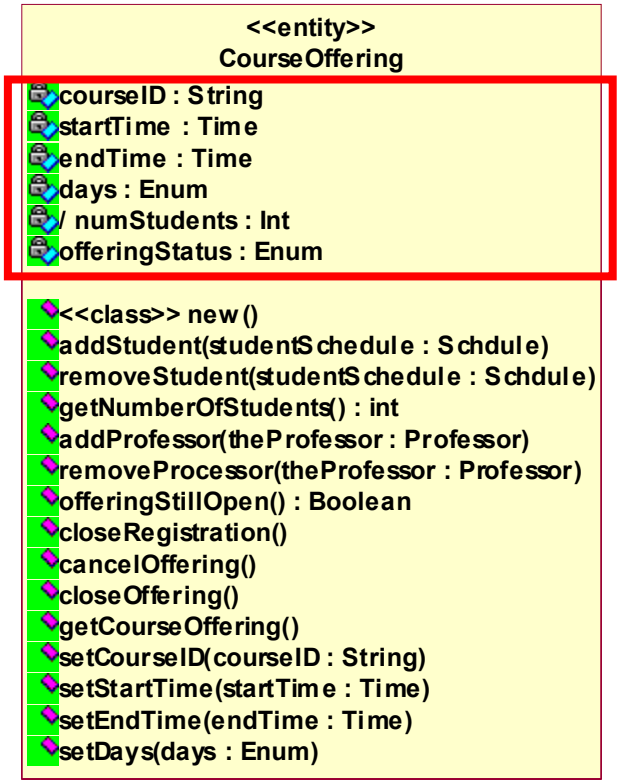


2. 细化属性

- 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



细化属性

- 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。**

3. 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；

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(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；

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**

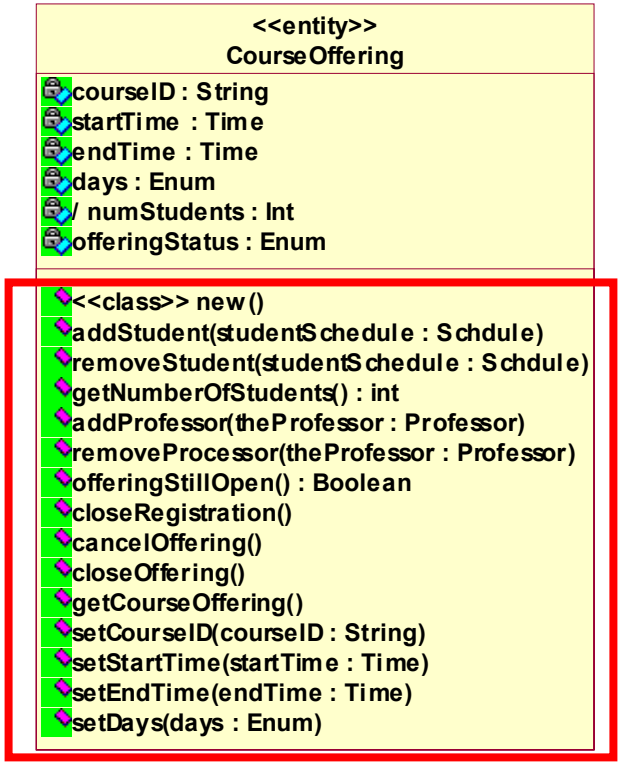
    addProfessor (theProfessor:Professor)

    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)

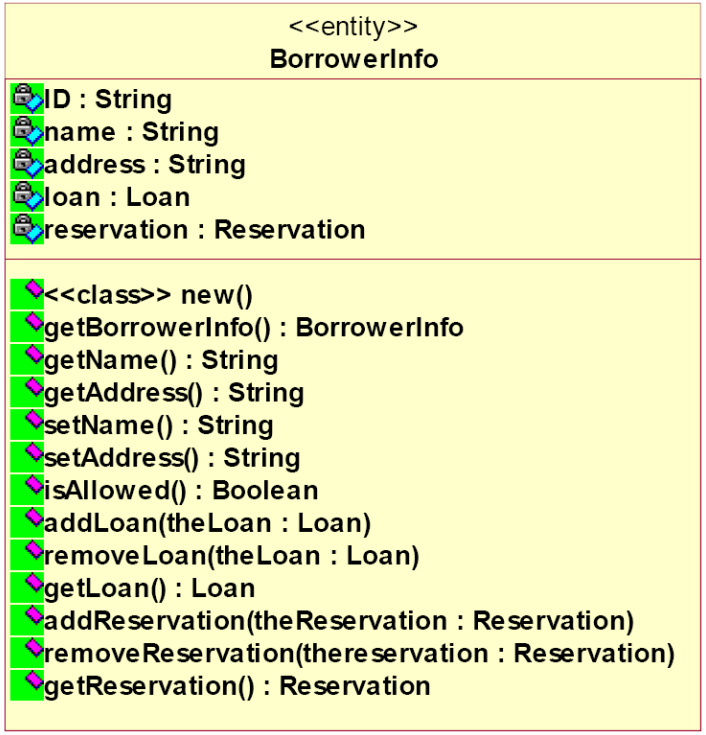
    + setAddress(address:String)
  - 其他

    + 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)

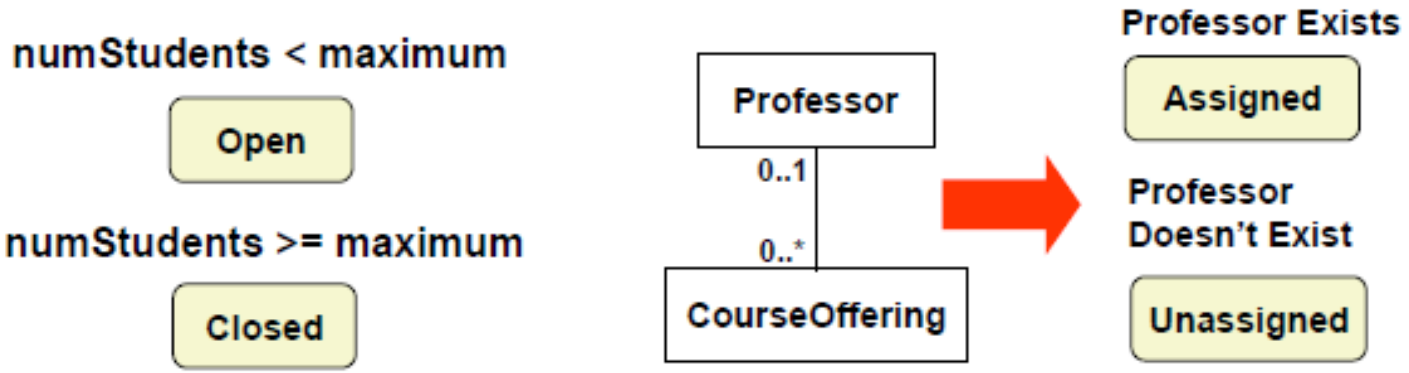
```
if offeringStatus = assigned {
   offeringStatus := unassigned;
   courseInstructor := NULL;
}
else errorState ( );
```

closeOffering ( )

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

4* 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**

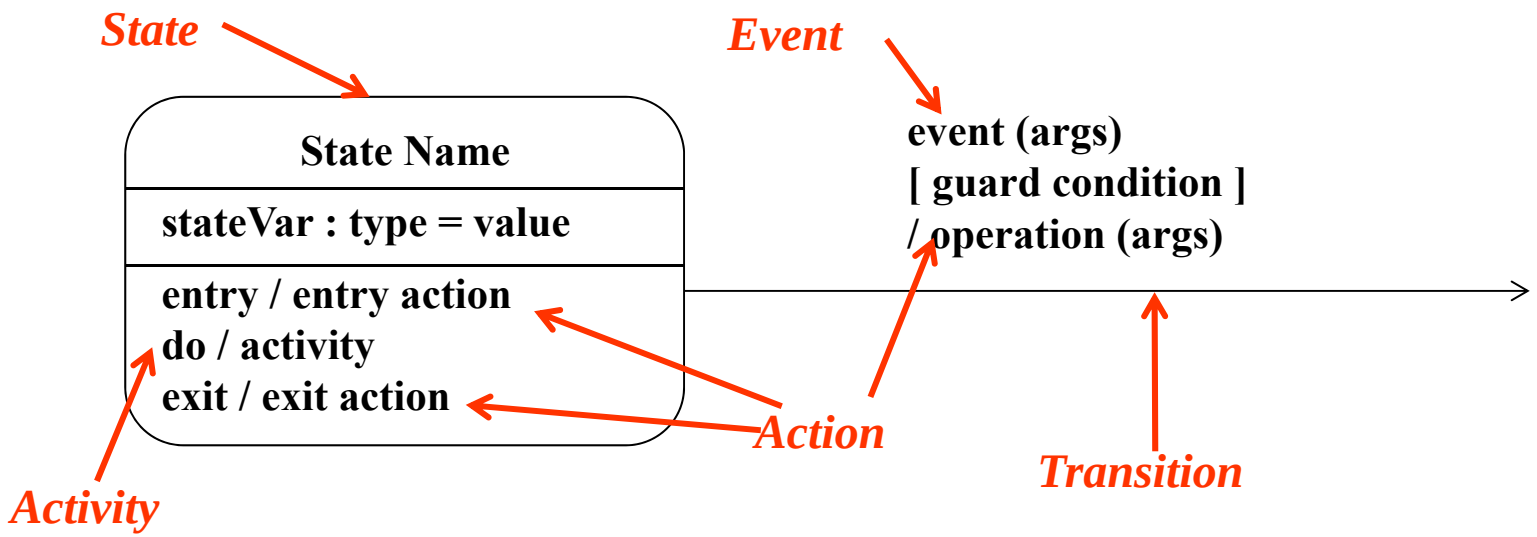


状态图(Statechart Diagram)

- 状态图(**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)

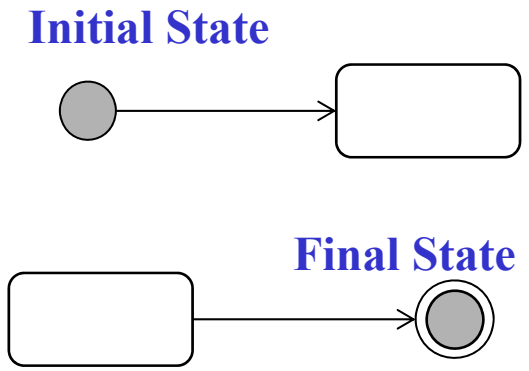
- 状态图的基本概念
  - State(状态)
  - Event(事件)
  - Transition(转移)
  - Action(动作)



状态图：状态

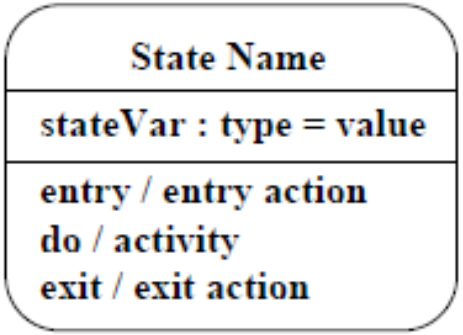
- **状态(State)**
  - 一个对象在生命期中满足某些条件、执行一些行为或者等待一个事件时的存在条件。
- **所有对象都具有状态，状态是对象执行了一系列活动的结果。当某个事件发生后，对象的状态将发生变化。**
- **事件和状态间具有某种对称性，事件表示时间点，状态表示时间段，状态对应着对象接收的两次事件之间的时间间隔。**

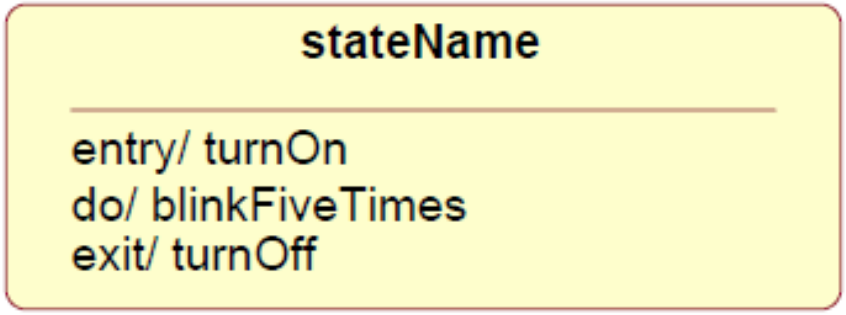
- **状态图中定义的状态**
  - 初态、终态
  - 中间状态、组合状态、历史状态
  - 一个状态图只能有一个初态，而终态可以有多个



状态图：状态

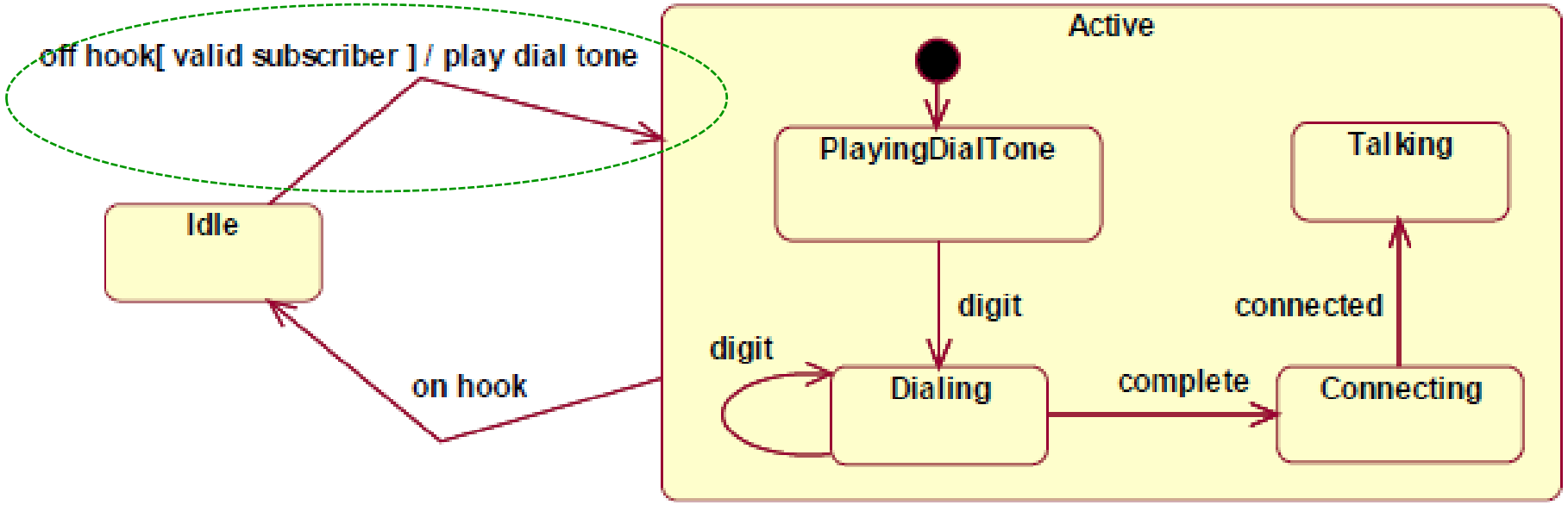
- 中间状态包括两个区域：名字域和内部转移域，如下图所示，其中内部转移域是可选的。
  - 入口动作表达式，**entry/action _ expression**：状态到达/进入时执行的活动。
  - 出口动作表达式，**exit/action _ expression** ：状态退出时执行的活动。
  - 内部动作表达式，**do/action _ expression** ：表示某对象处在某状态下的全部或部分持续时间内执行的活动。（如：复印机卡纸状态下闪烁5次灯）





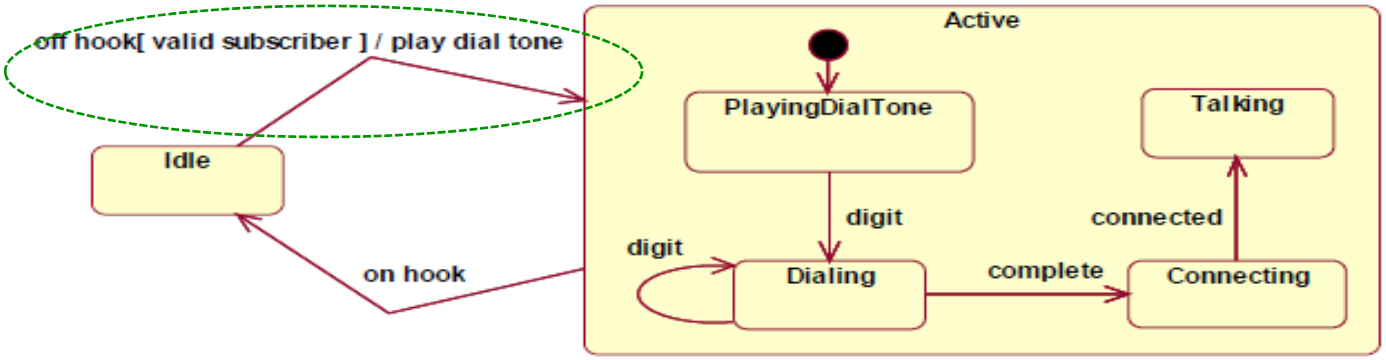
状态图：转换

- 转换/转移/迁移**(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。

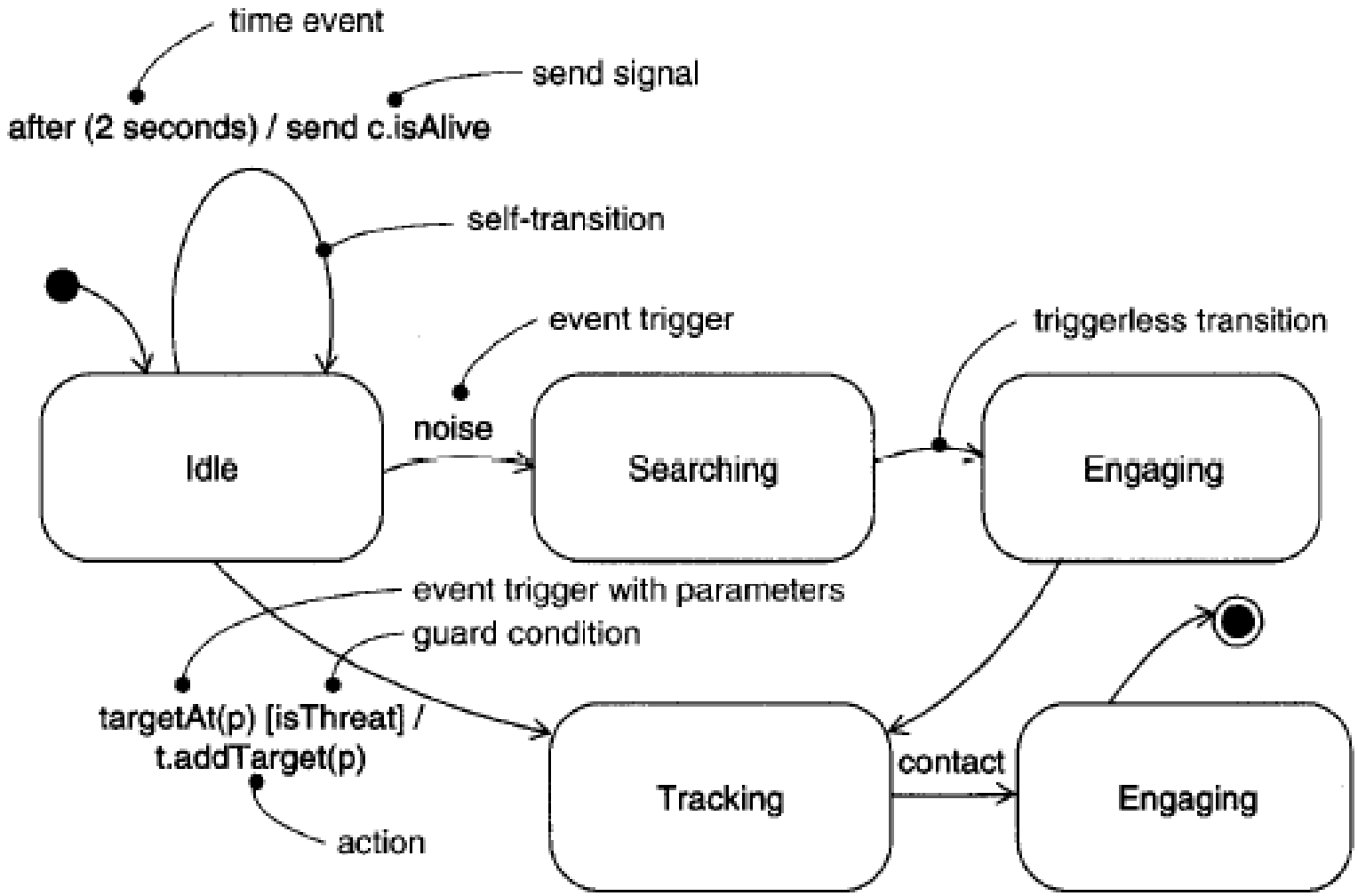


状态图：转换

- 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



状态图(Statechart Diagram)



状态图：事件

- 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。

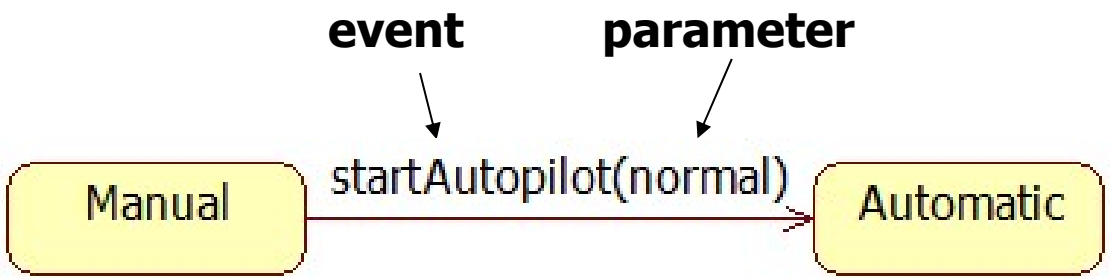
- **事件具有的性质**
  - 一个事件的发生是在某一时刻，无持续性；
  - 两个事件是可以相继发生的，也可以是共存的，也可以互不相关的；
  - 多个事件可以组成事件类；
  - 事件具有触发事件动作的对象；
  - 事件在对象间传递信息；
  - 事件包括错误状态(马达被卡住、超时等)和普通事件，二者只是称呼不同；

状态图：事件

- **UML中 事件的分类**
  - Call event (调用事件)
  - Change event (变化事件)
  - Time event (时间事件)
  - Signal event (信号事件)

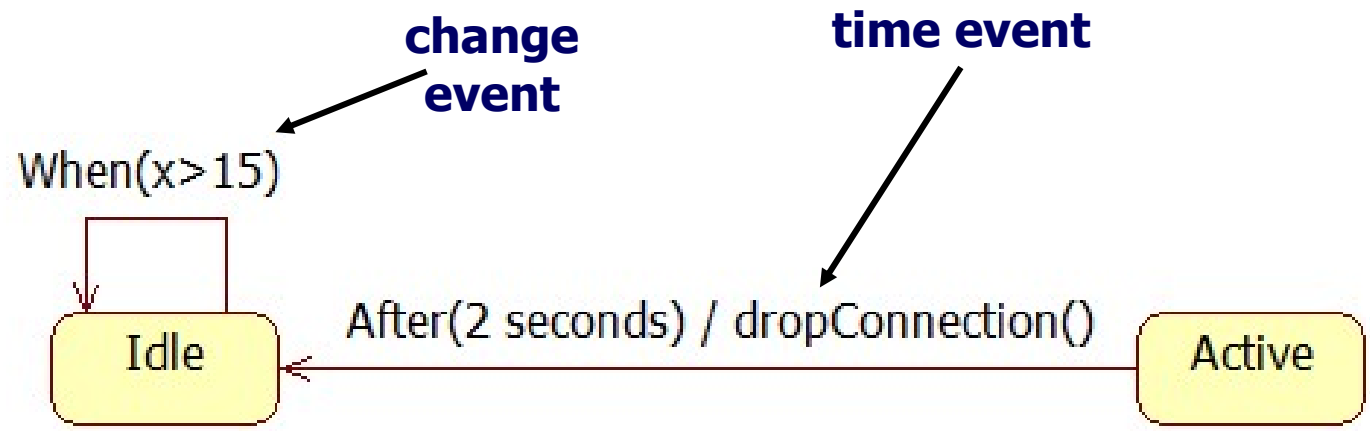
调用事件

- 调用事件(Call event)



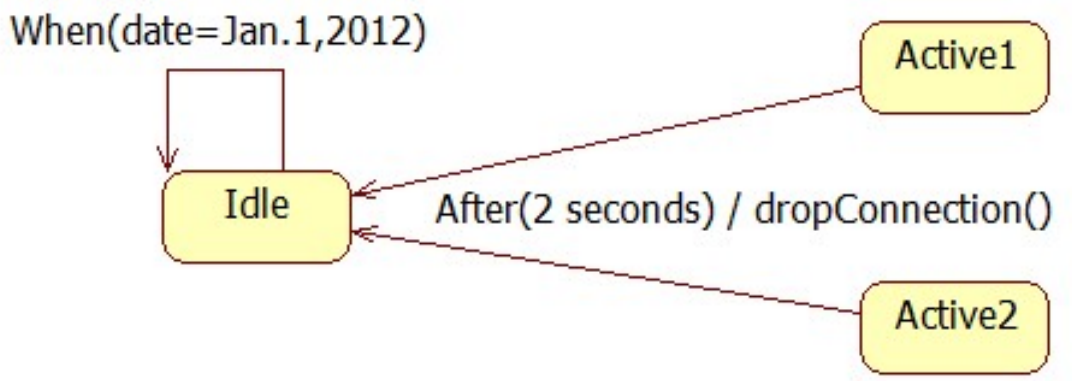
变化事件

- 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



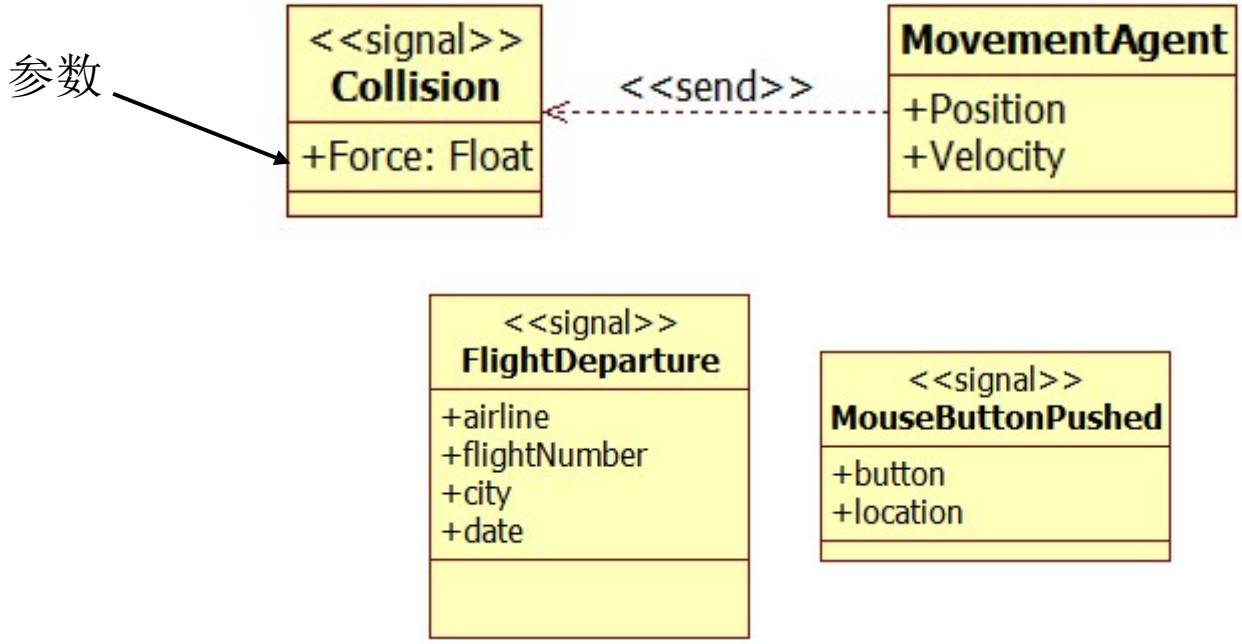
时间事件

- 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

- 信号事件(Signal event)：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



状态图：动作

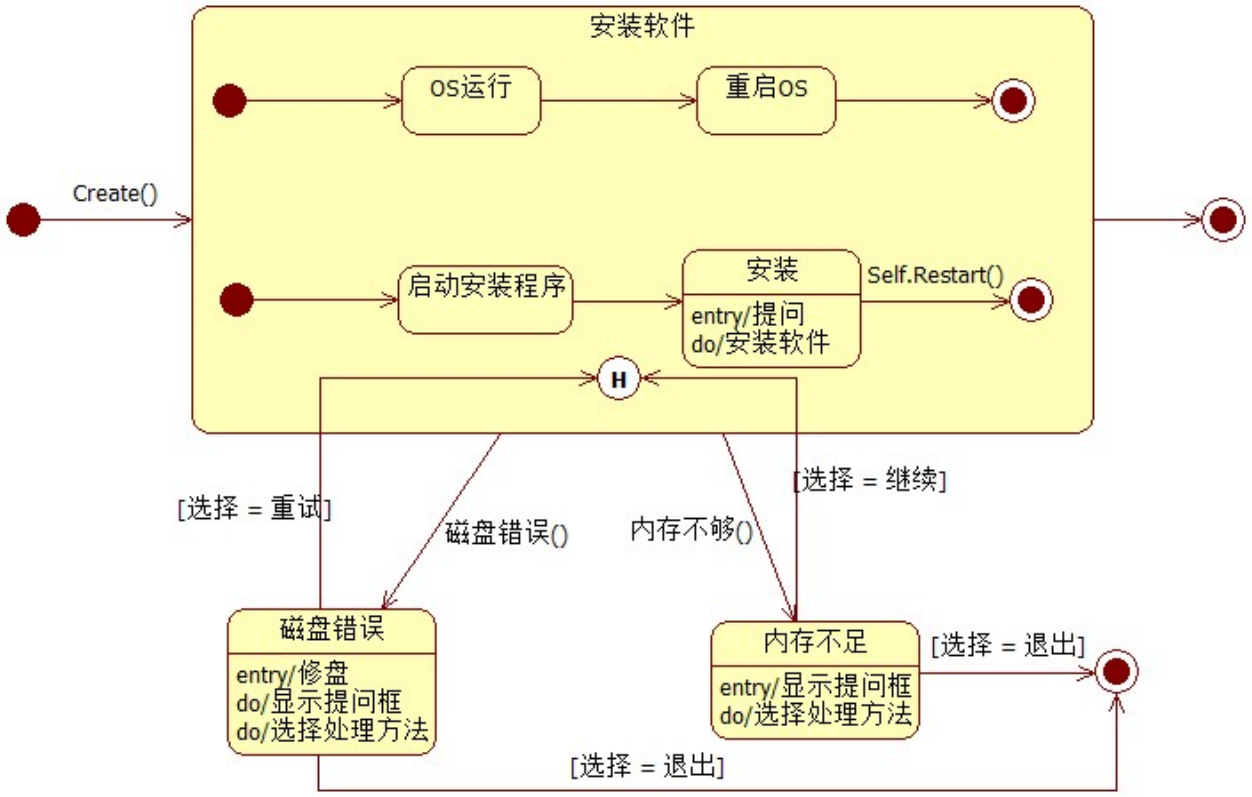
- **动作(Action)**
  - An action is an executable atomic computation.
  - 在一个特定状态下对象执行的行为
- **动作是原子的，不可被中断的，其执行时间可忽略不计的。**
- **两个特殊的action: entry action和exit action**
  - Entry动作：进入状态时执行的活动
  - Exit动作：退出状态时执行的活动

状态图：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。

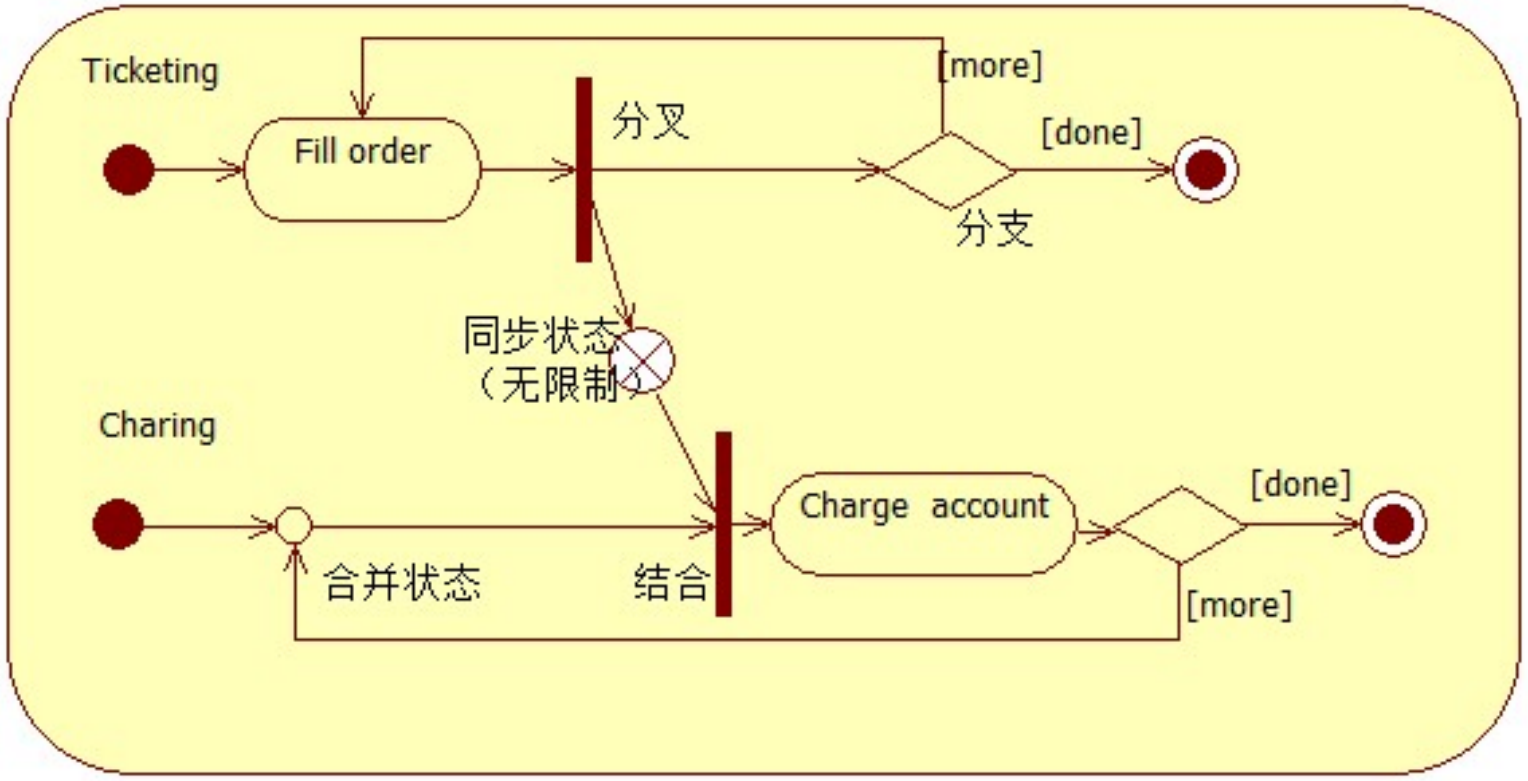
状态图：历史指示器

- 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。



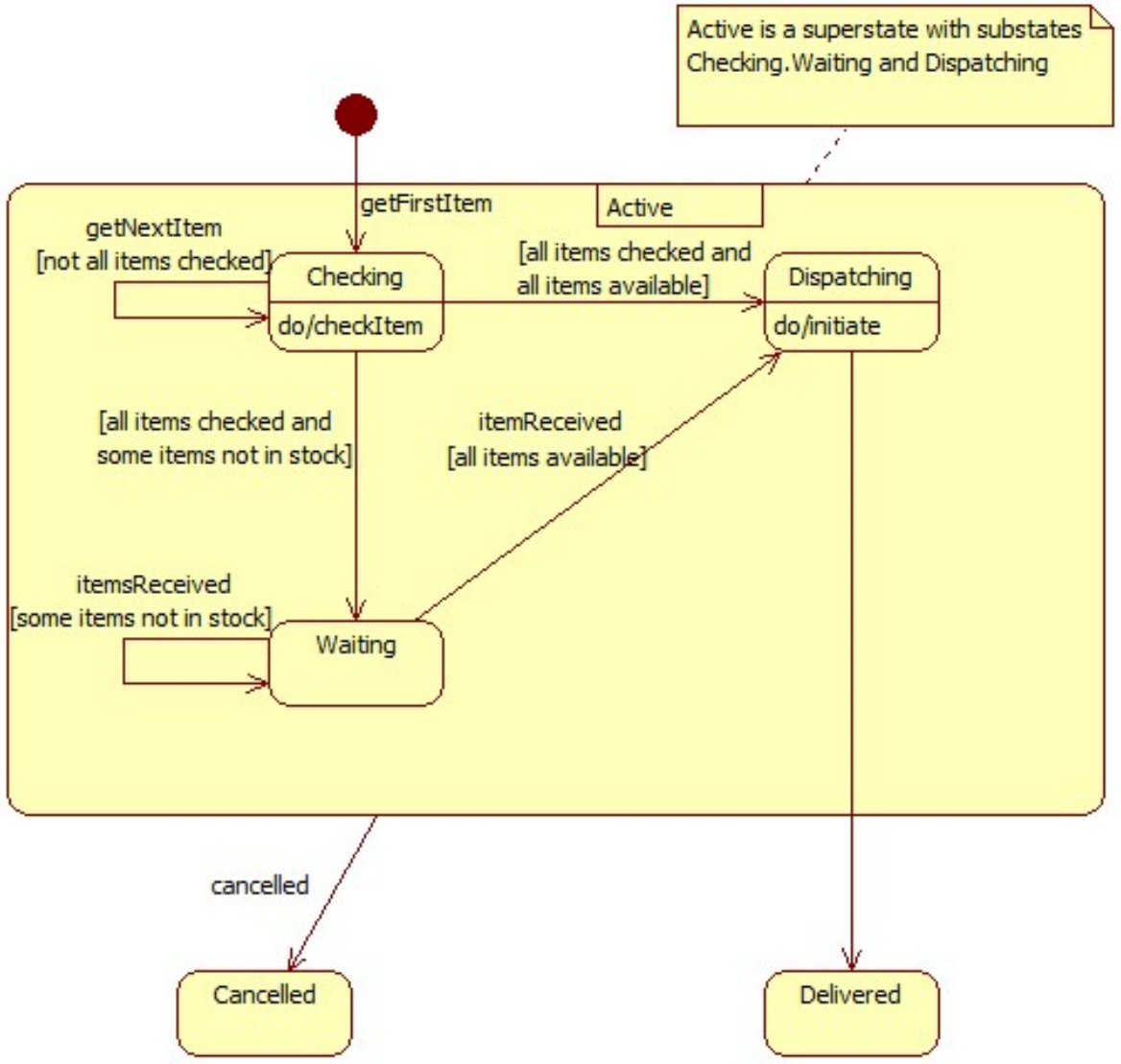
状态图：同步状态

- 同步状态



状态图(Statechart Diagram)

- 组合状态

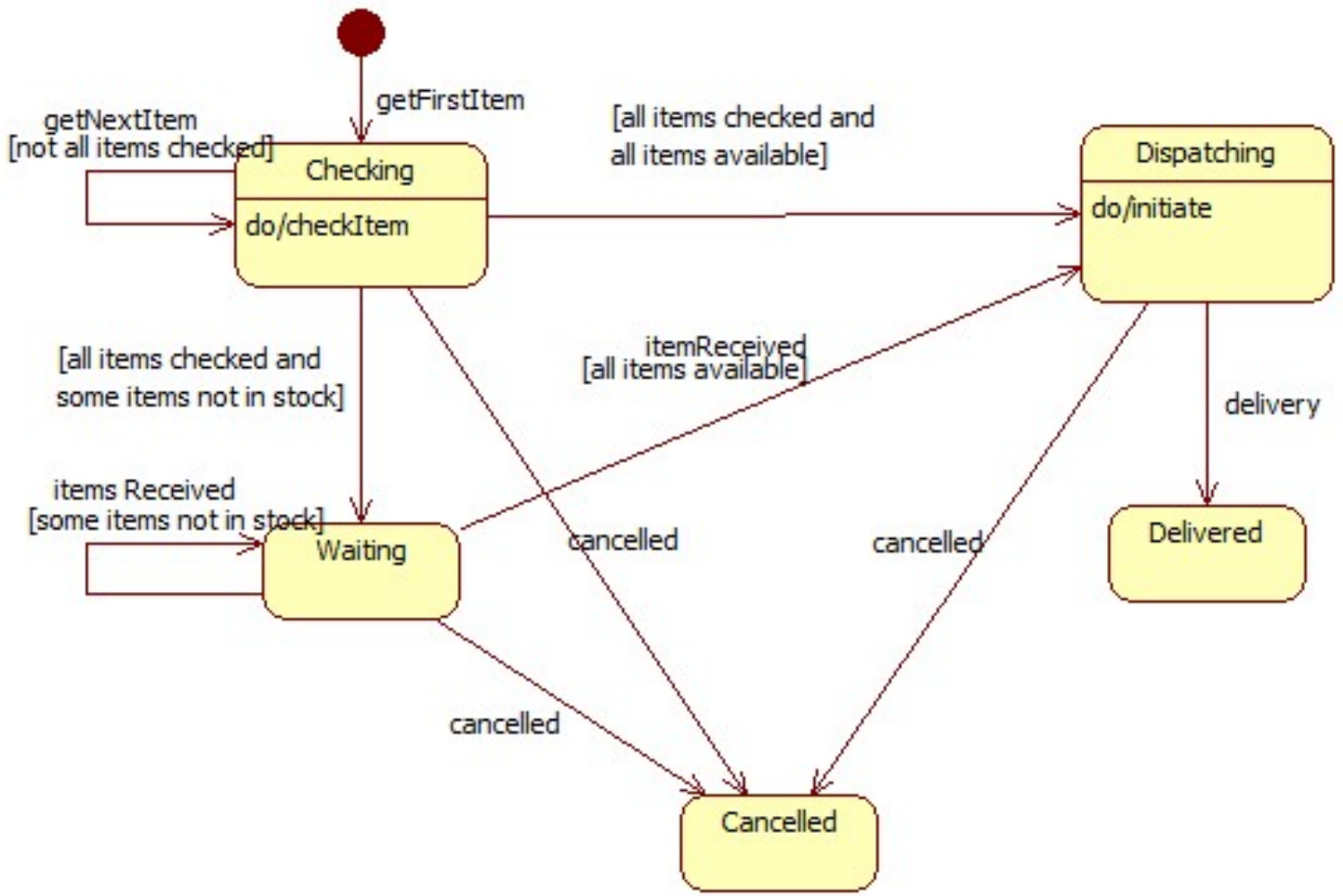


状态图绘制方法

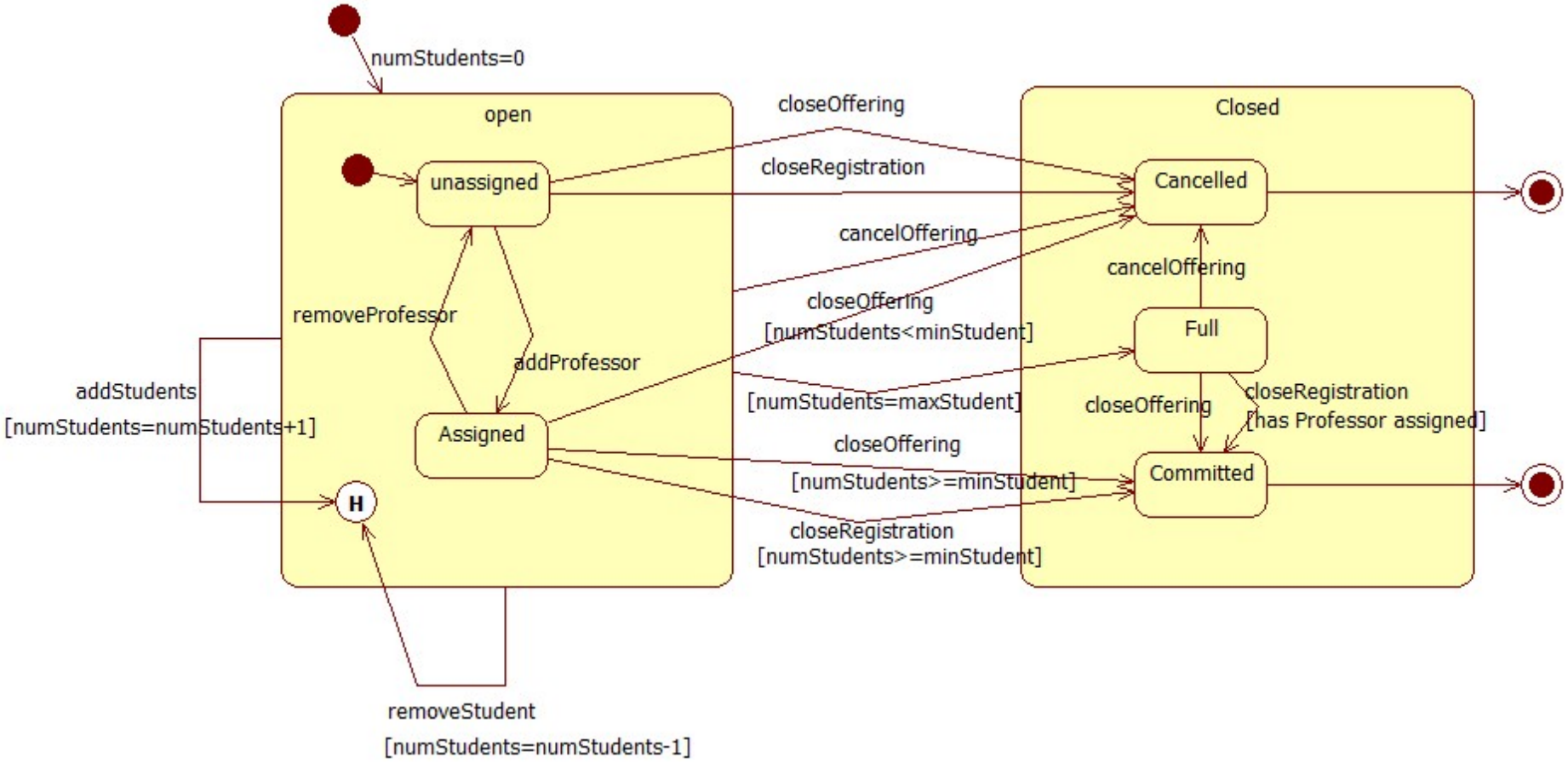
- 检查类图，选择出需要状态图的类(用况或过程、系统、窗口、控制者、事物、设备、突变类型)
  - 状态由属性值表示
  - 通过查看属性的边界值来寻找状态
- 确定状态的转移
  - 确定尽可能多的状态，然后寻找转换。
  - 转换可能是方法调用的结果，经常会反映业务规则。
- 复杂的状态，会有子状态存在。
- 状态图常用于实时系统，记录复杂的类，还会揭示潜在错误条件。

状态图(Statechart Diagram)

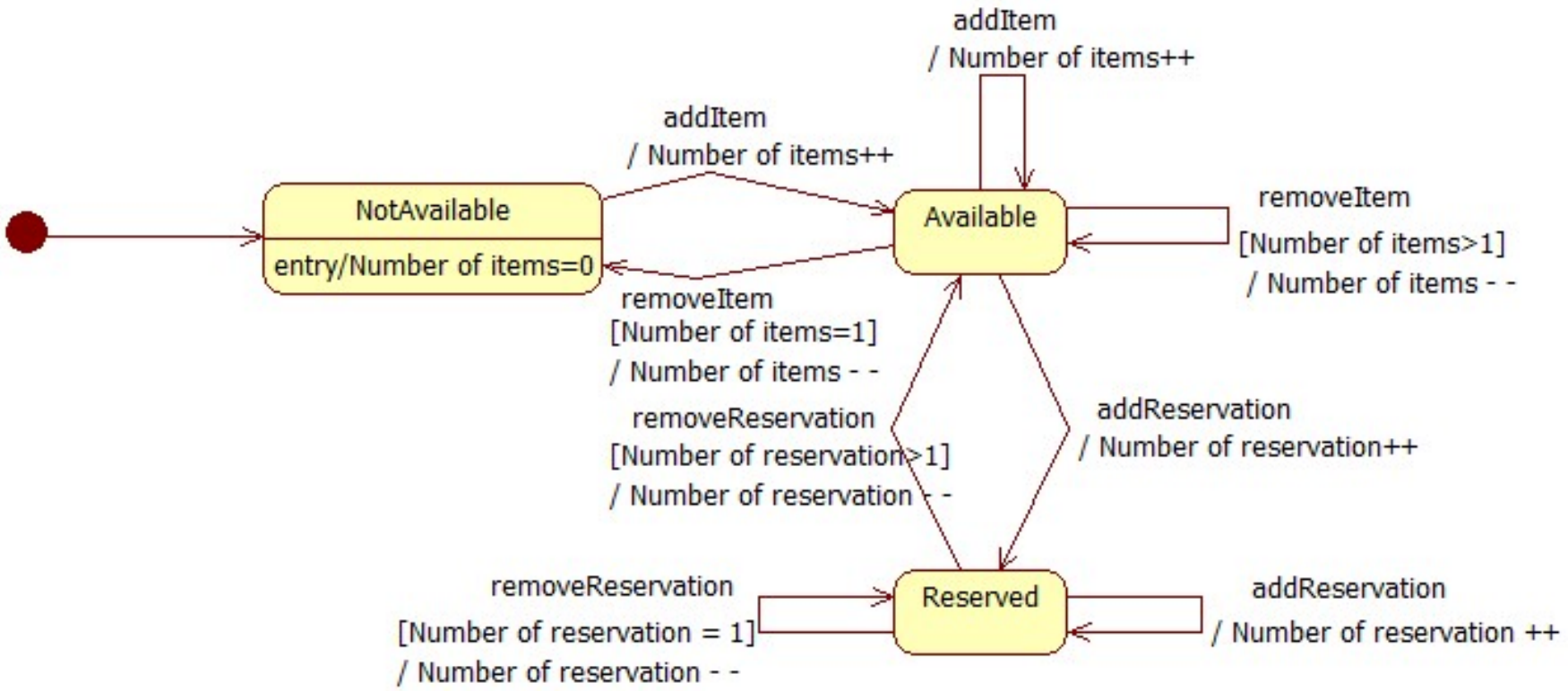
- “订单”对象的状态图



状态图(Statechart Diagram)

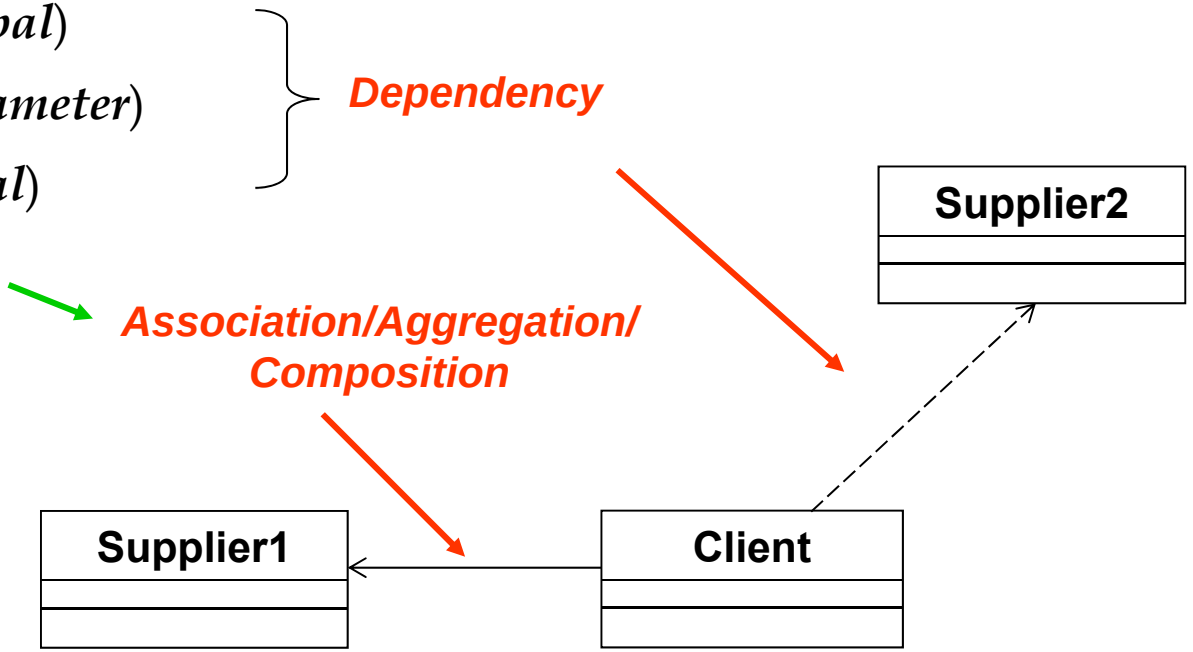


状态图(Statechart Diagram)

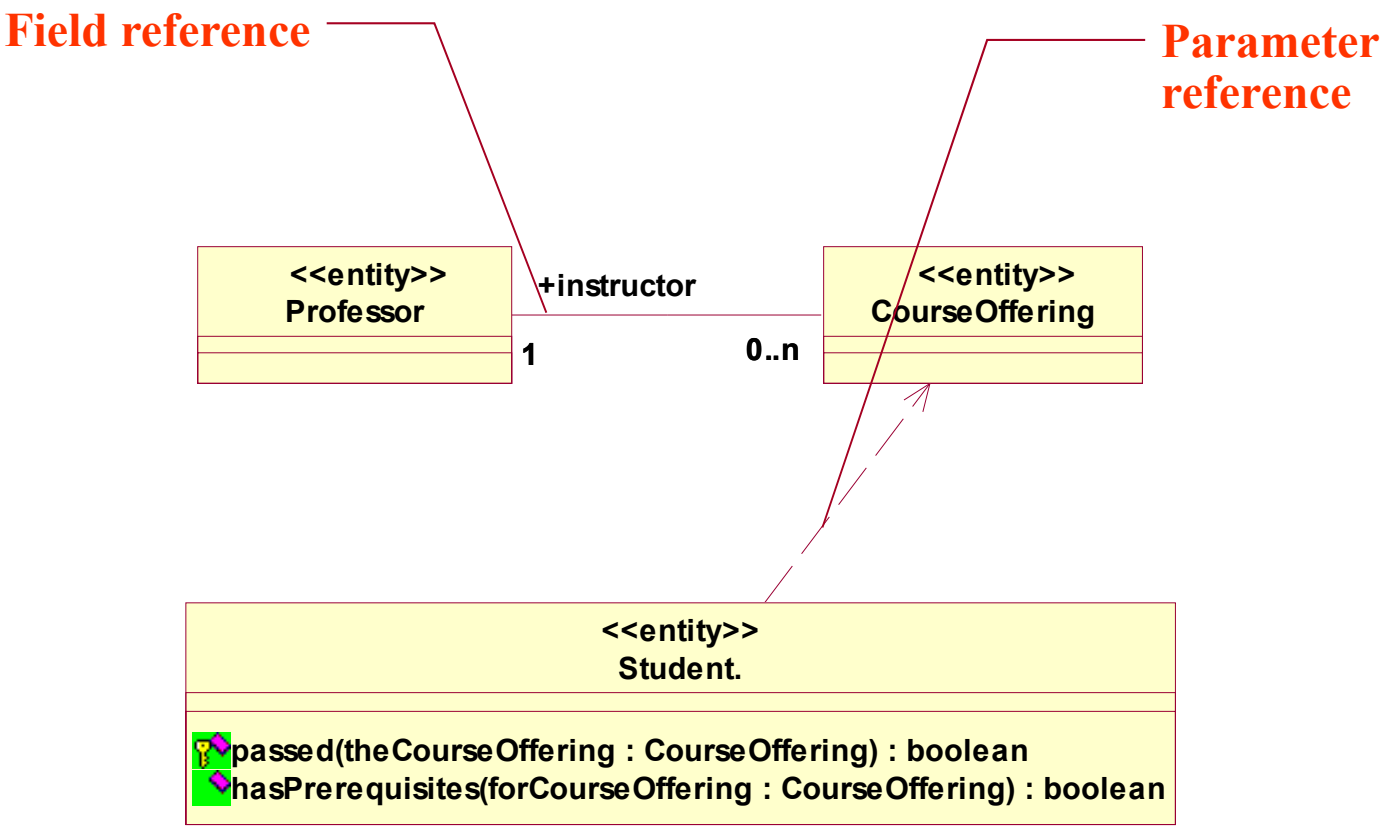


5. 细化类之间的关系

- 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*)

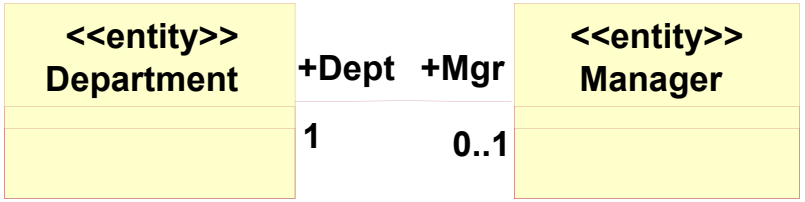


定义类之间的关系：示例


Field reference
Parameter reference
<<entity>>
Professor
+instructor
1
0..n
<<entity>>
CourseOffering
<<entity>>
Student.
passed(theCourseOffering : CourseOffering) : boolean
hasPrerequisites(forCourseOffering : CourseOffering) : boolean

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。

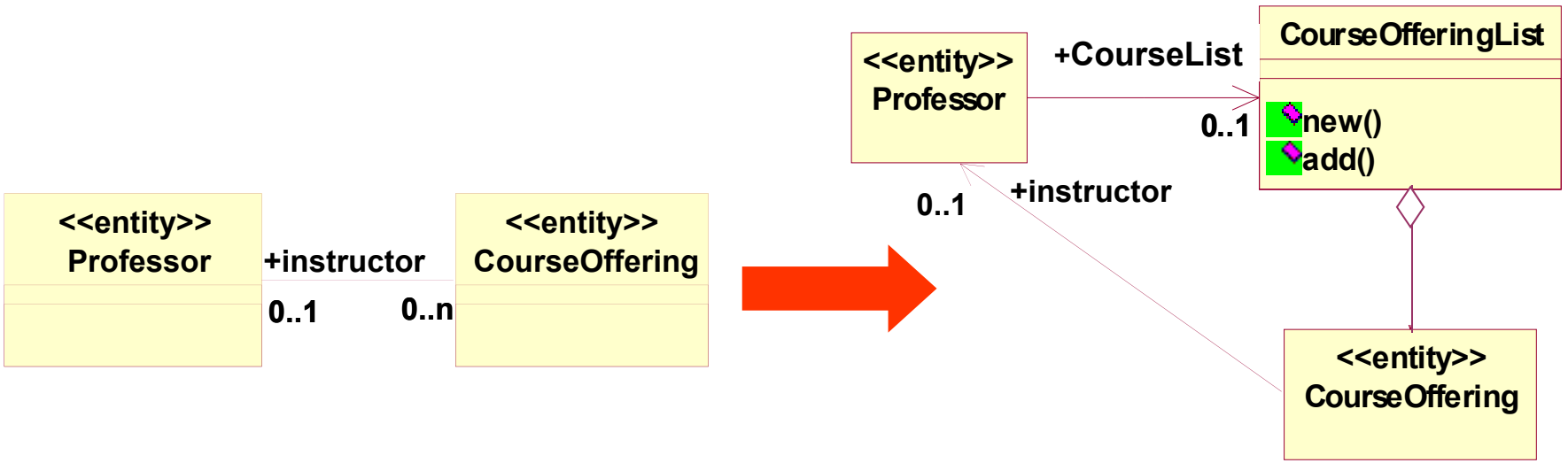


设计阶段需要将这种“关联属性”增加到属性列表中，并更新操作列表；

分析阶段则不需要在属性列表中加入“关联属性”

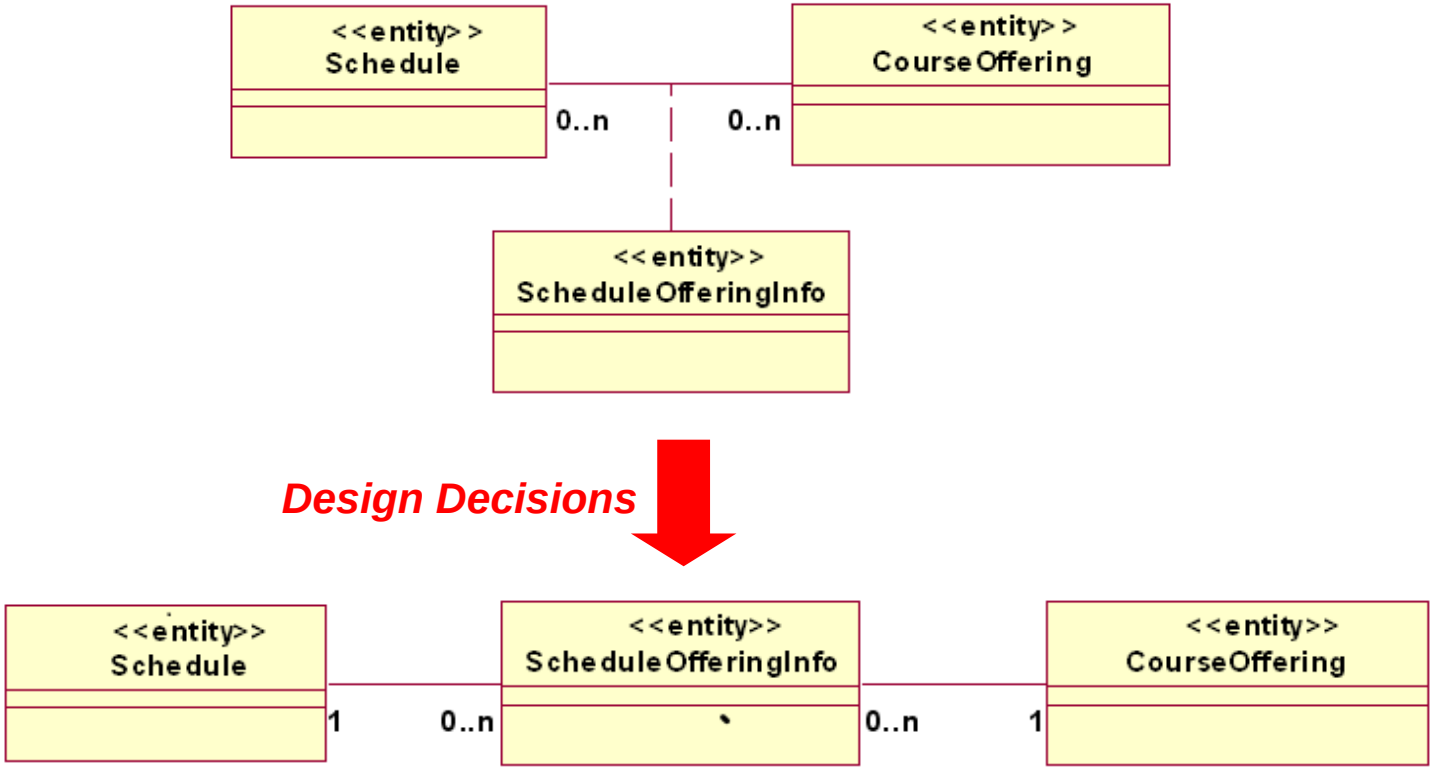
定义关联关系 (Association/Composition/Aggregation)

- 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。

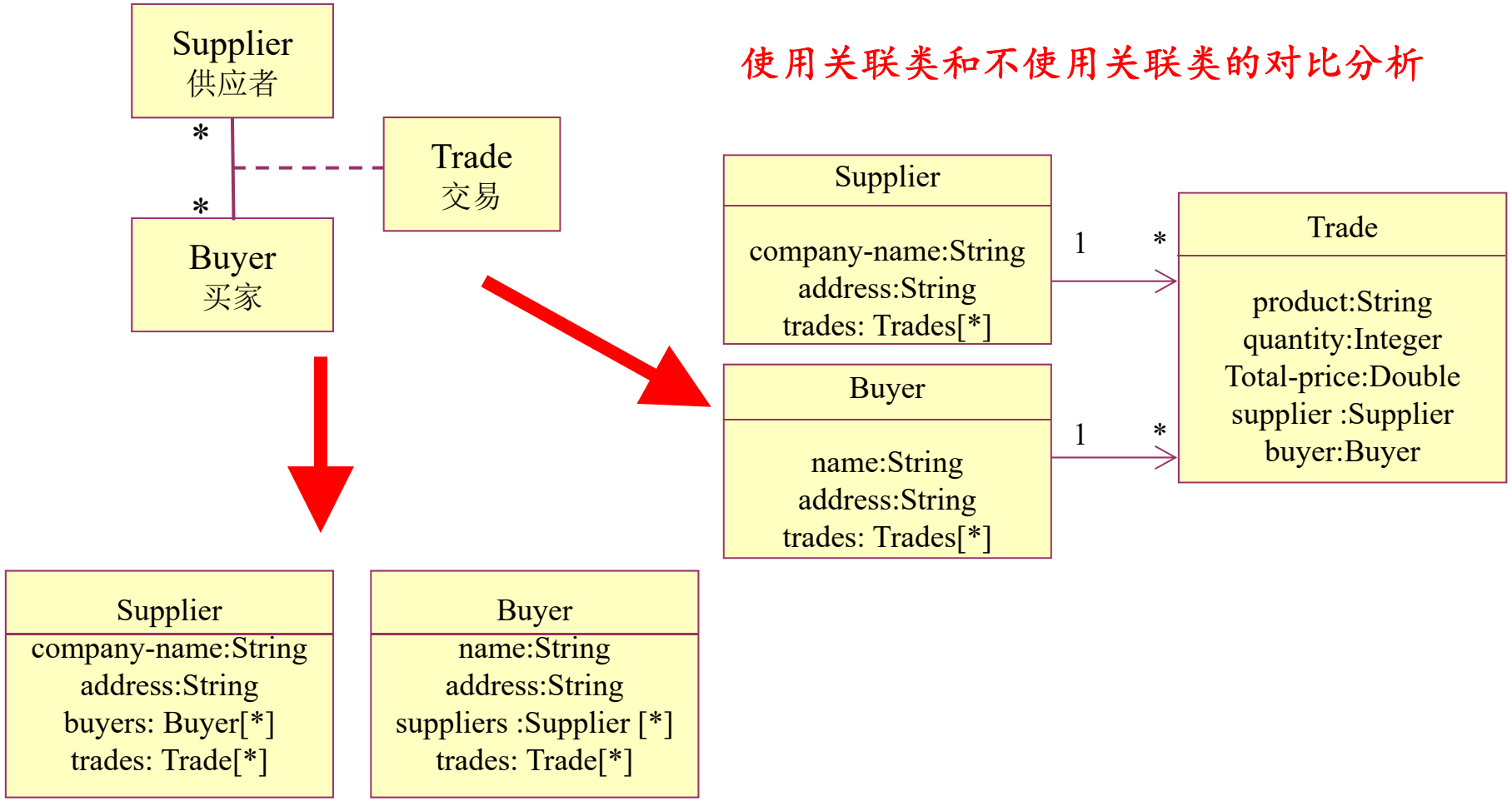


定义关联关系 (Association/Composition/Aggregation)

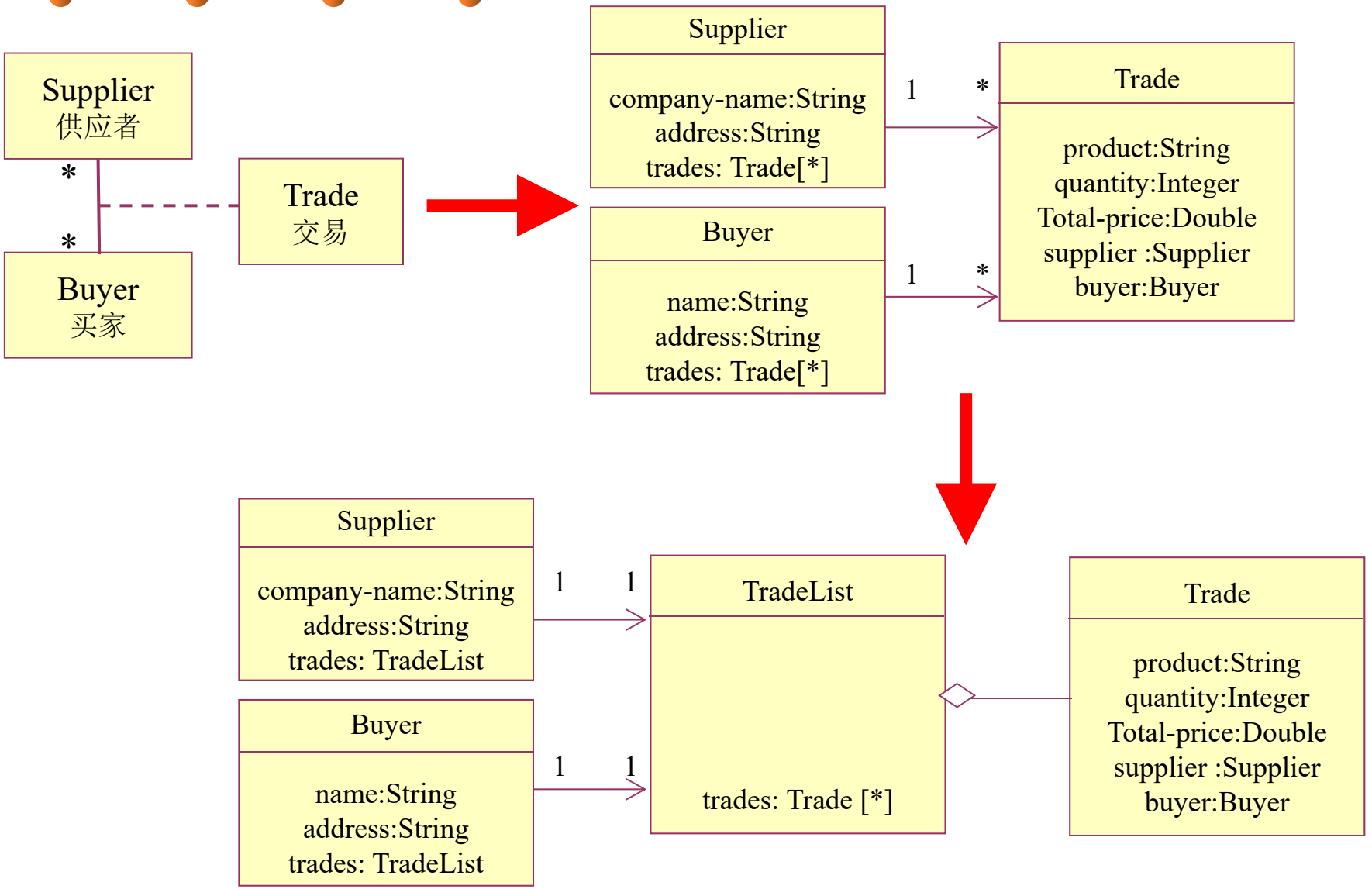
- 情况3：有些情况下，关联关系本身也可能具有属性，可以使用“关联类”将这种关系建模。
- 举例：选课表Schedule与开设课程CourseOffering



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



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



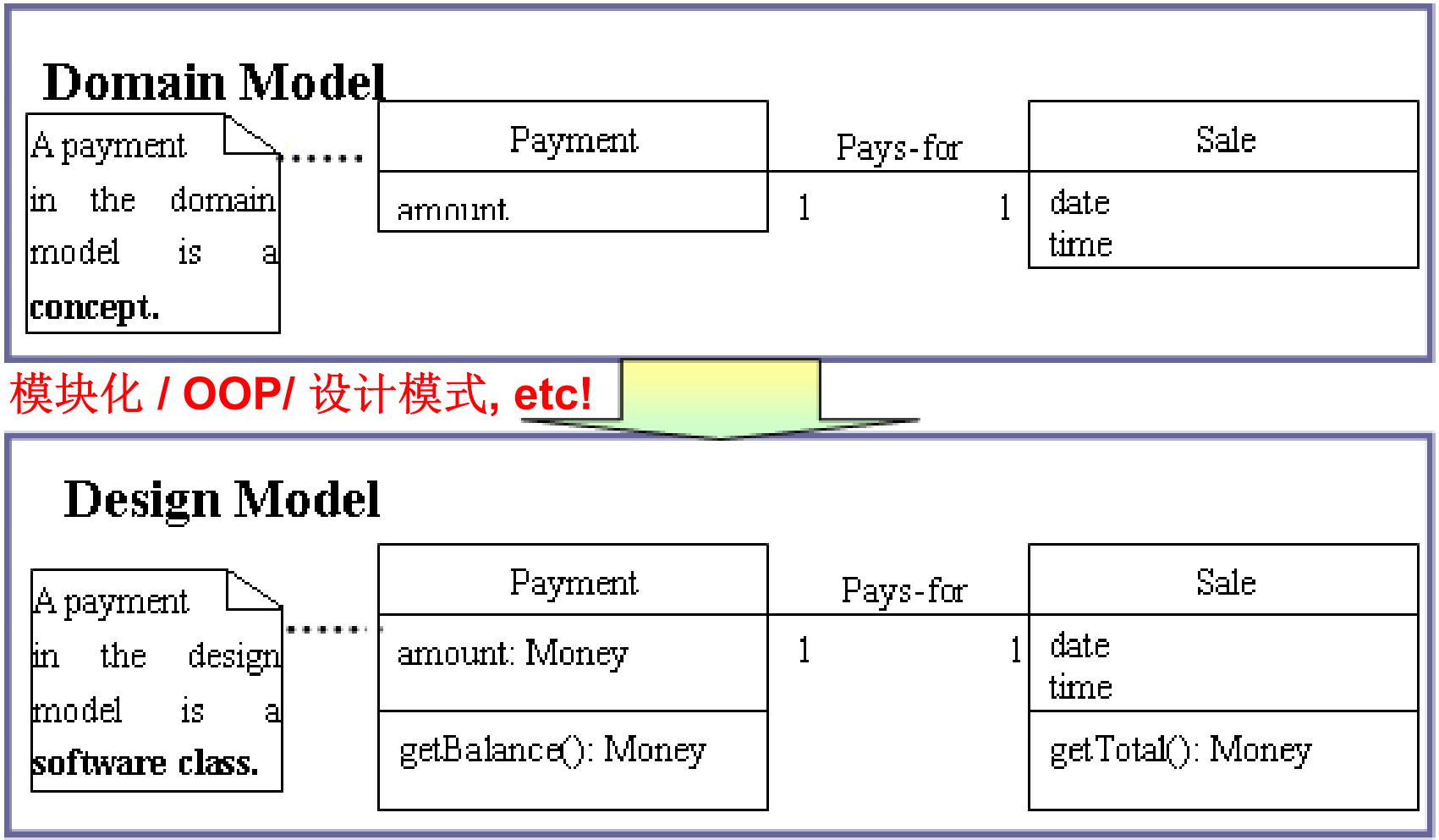
引入“辅助类”简化类的内部结构

- 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

面向对象设计总结

- **系统设计**
  - *包图(package diagram)→逻辑设计
  - 部署图(deployment diagram)→物理设计

- **对象设计**
  - 类图(class diagram)→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(statechart diagram)
  - *时序图(sequence diagram)→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(RDBMS design)
  - 用户界面设计方案(UI design)

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　　use case diagram
  - 活动图　　activity diagram
  - 类　图　　class diagram
  - 时序图　　sequence diagram
  - 协作图*　　collaboration diagram
  - 状态图*　　statechart diagram
  - 部署图　　deployment diagram
  - 包　图　　package diagram
  - 构件图*　　component diagram

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

课外阅读：Pressman教材第6-7、18-20章

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

结束

2017年11月6日