



软件工程
第五章 面向对象分析与设计
5-6 面向对象的设计

刘铭

2025年6月13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. 软件部署
 5. 数据库设计
 6. 对象设计
 7. 面向对象设计总结

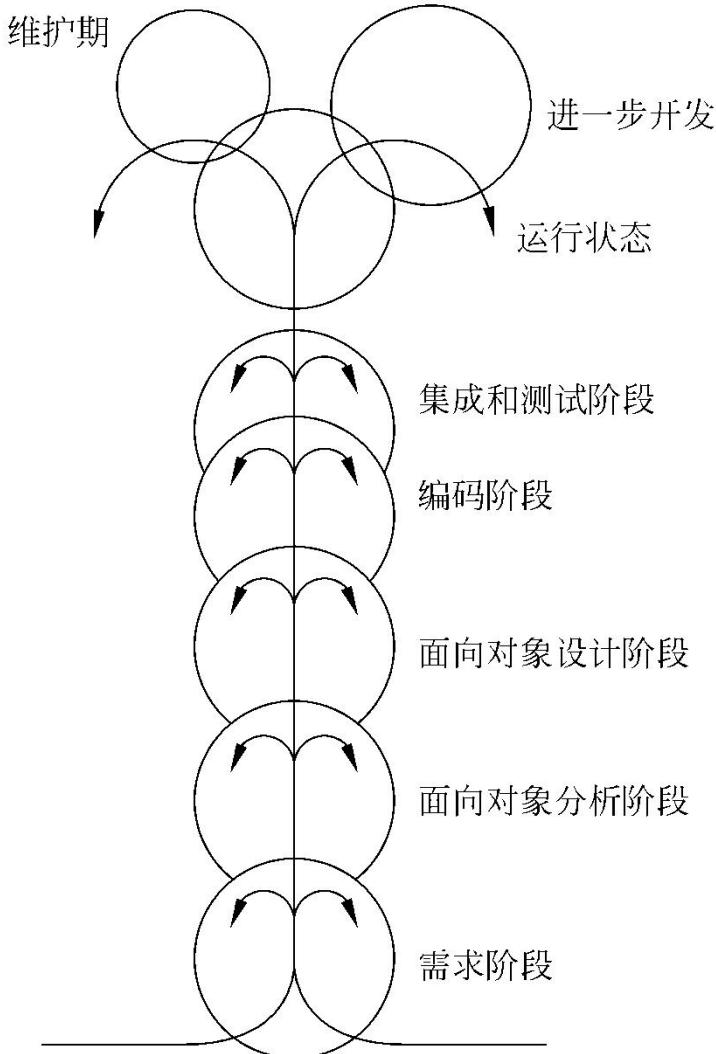


1. 面向对象设计概述



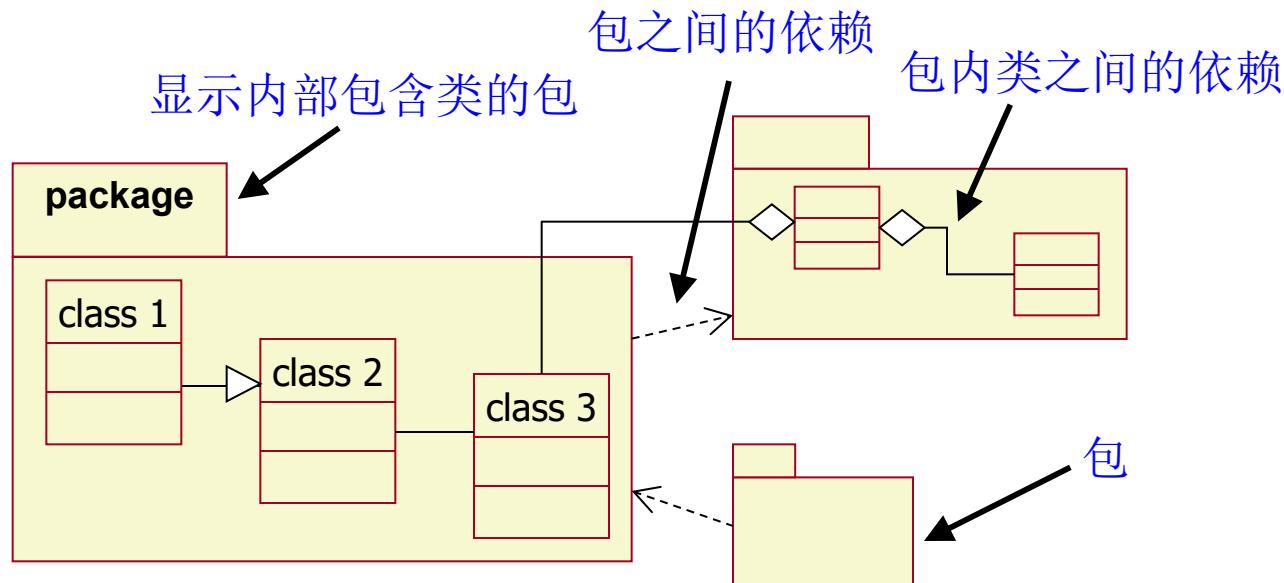
面向对象的设计

- 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。
- 面向对象的分析与设计(**OOA&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。



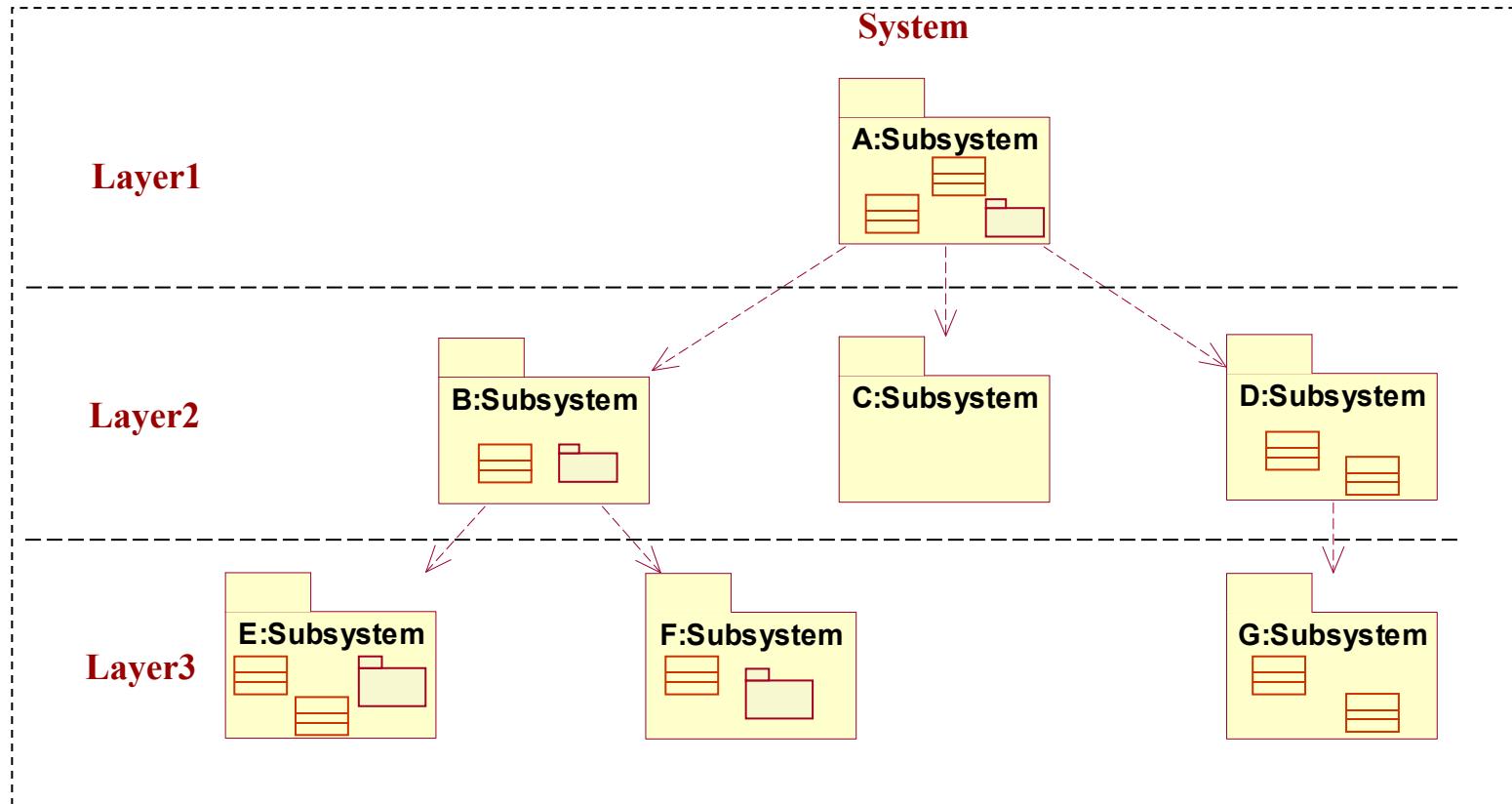
面向对象设计中的基本元素

- 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

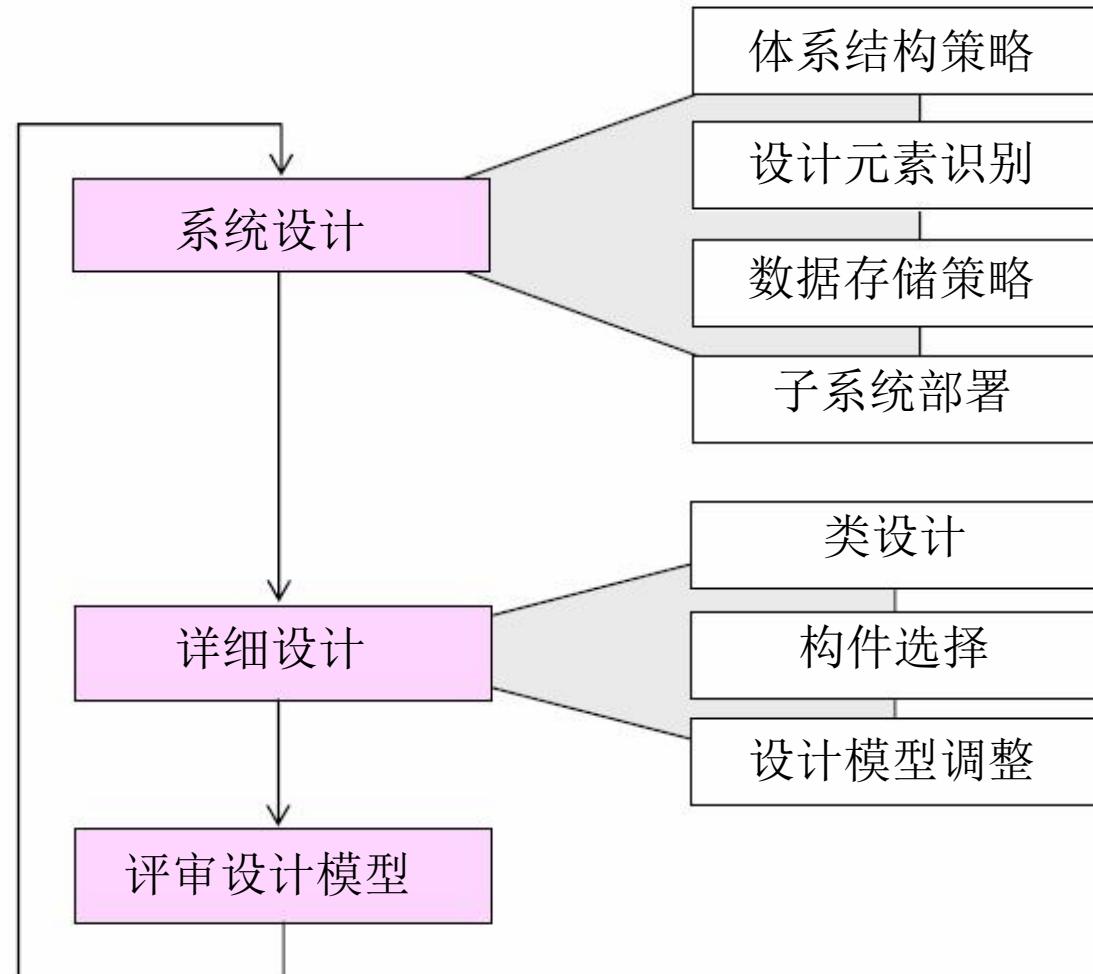
- **系统设计(System Design)**

- 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





2. 系统设计



系统设计概述

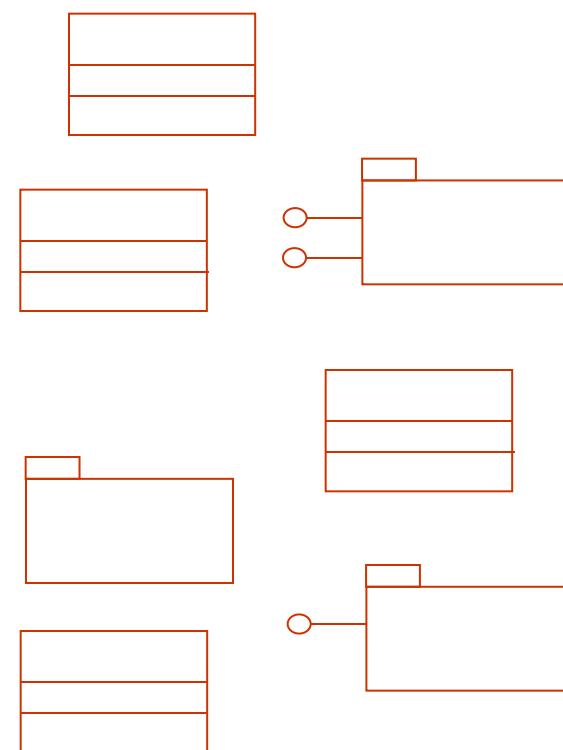
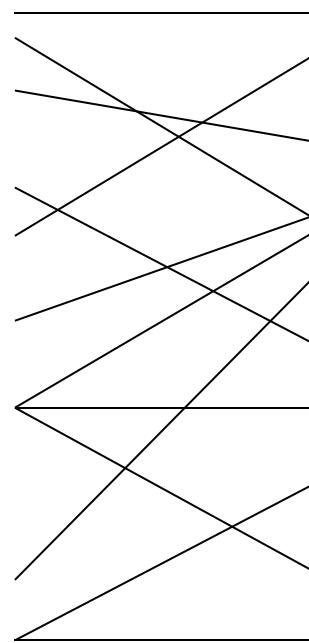
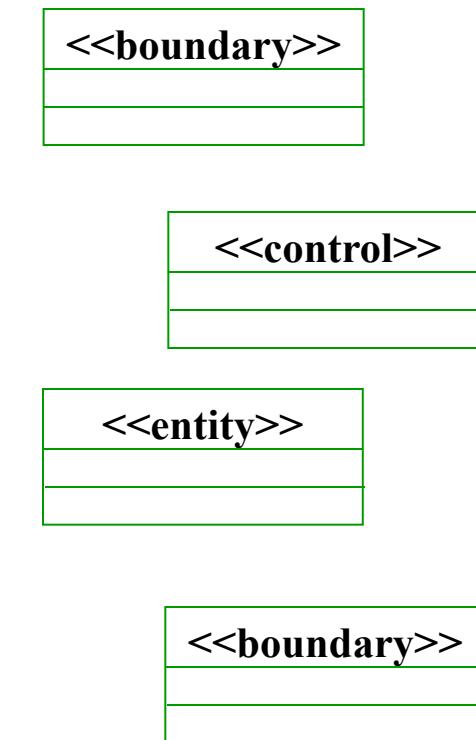
- 设计系统的体系结构 --**详见5-3和5-4**
 - 选择合适的分层体系结构策略，建立系统的总体结构：分几层？每层的功能分别是什么？
- 识别设计元素 --**详见后续章节**
 - 识别“设计类”(design class)、“包”(package)、“子系统”(sub-system)
- 部署子系统 --**详见后续章节**
 - 选择硬件配置和系统平台，将子系统分配到相应的物理节点，绘制部署图(deployment diagram)
- 定义数据的存储策略 --**详见后续章节**
- 检查系统设计



3. 包的设计：从逻辑角度



识别设计元素



多对多映射

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

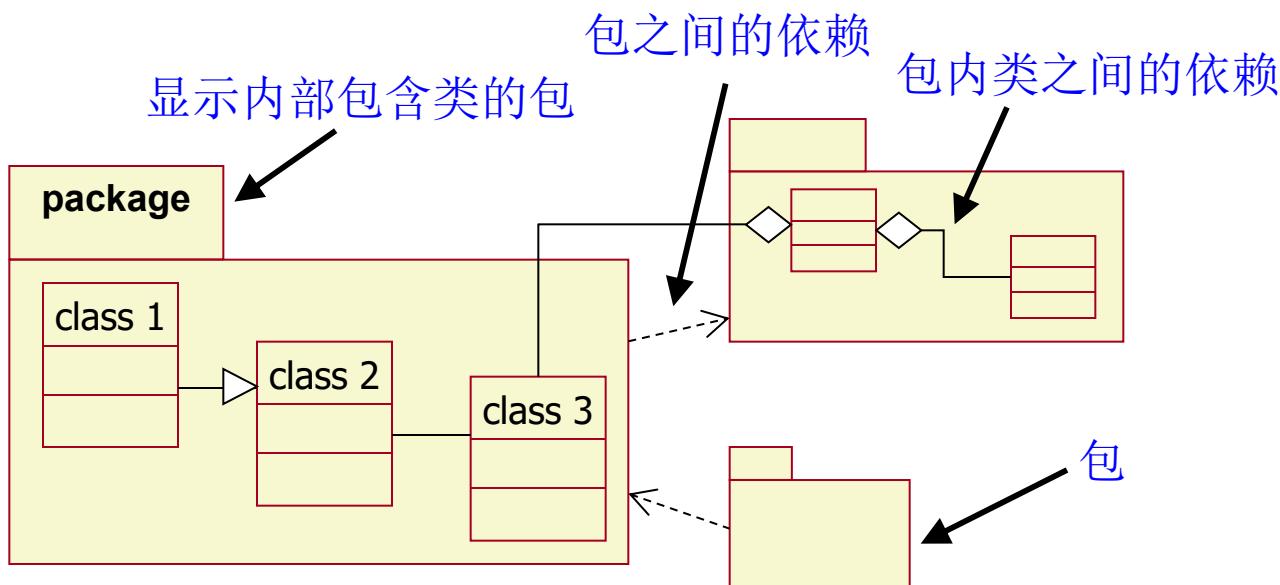
类型	分析类	设计元素
	<i>LoginForm</i>	“设计类” <i>LoginForm</i>
	<i>BrowseForm</i>	“设计类” <i>BrowseForm</i>

	<i>MailSystem</i>	“子系统接口” <i>IMailSystem</i>
	<i>BrowseControl</i>	“设计类” <i>BrowseControl</i>
	<i>MakeReservationControl</i>	“设计类” <i>MakeReservationControl</i>

	<i>BorrowerInfo</i>	“设计类” <i>BorrowerInfo</i>
	<i>Loan</i>	“设计类” <i>Loan</i>

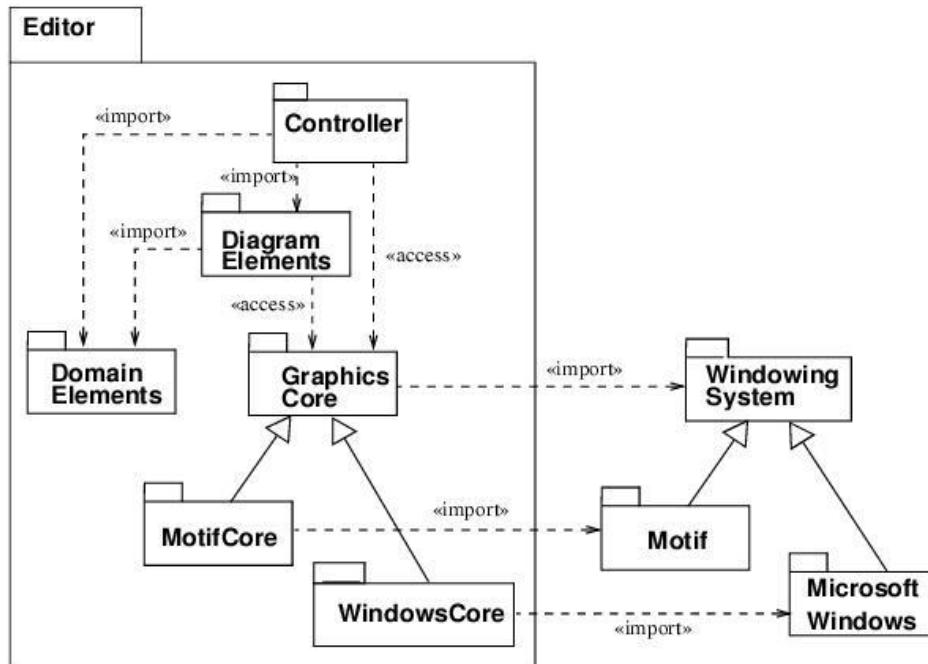
绘制包图 (package diagram)

- 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)；



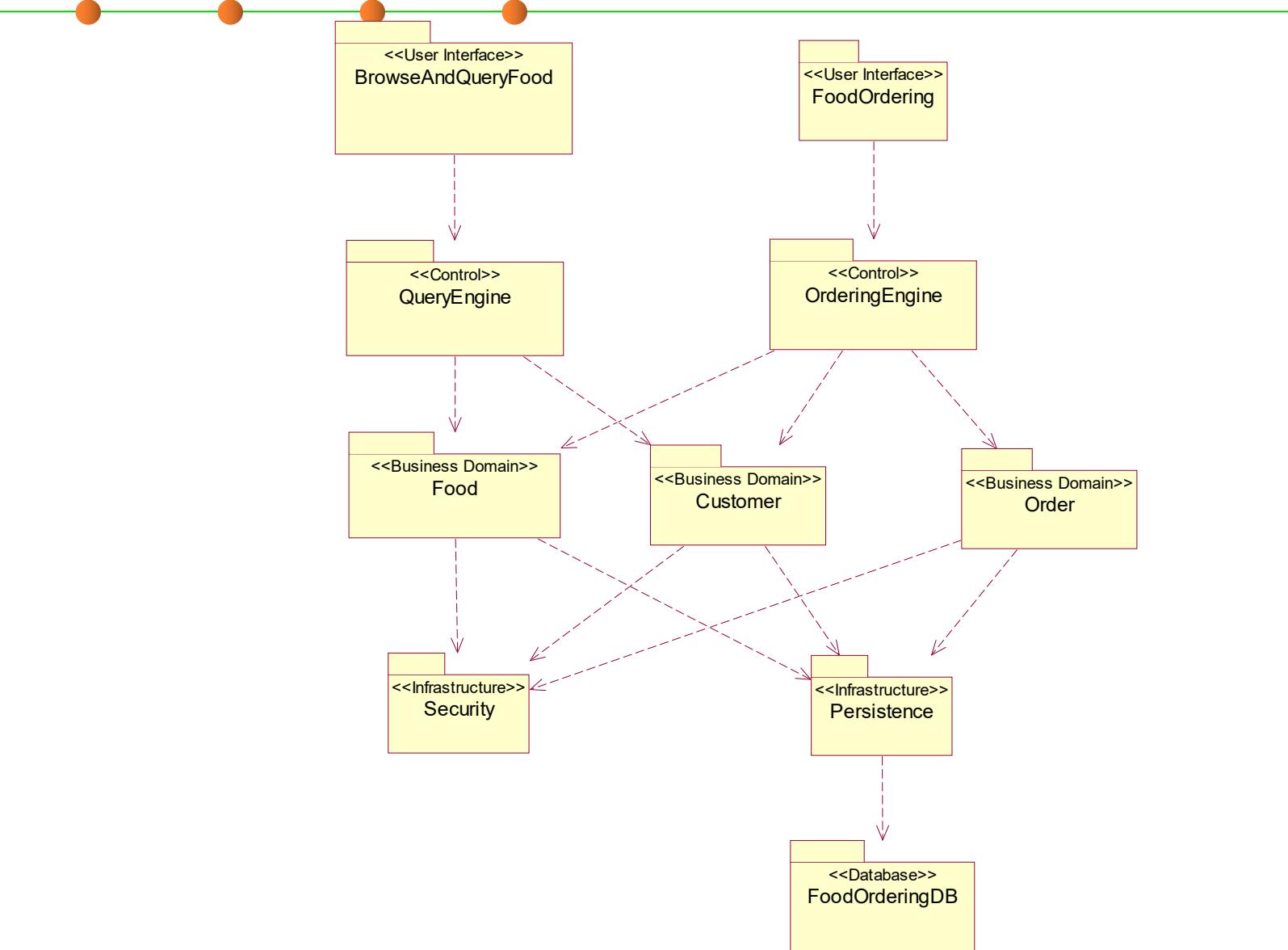
模型管理视图—包图

- 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。

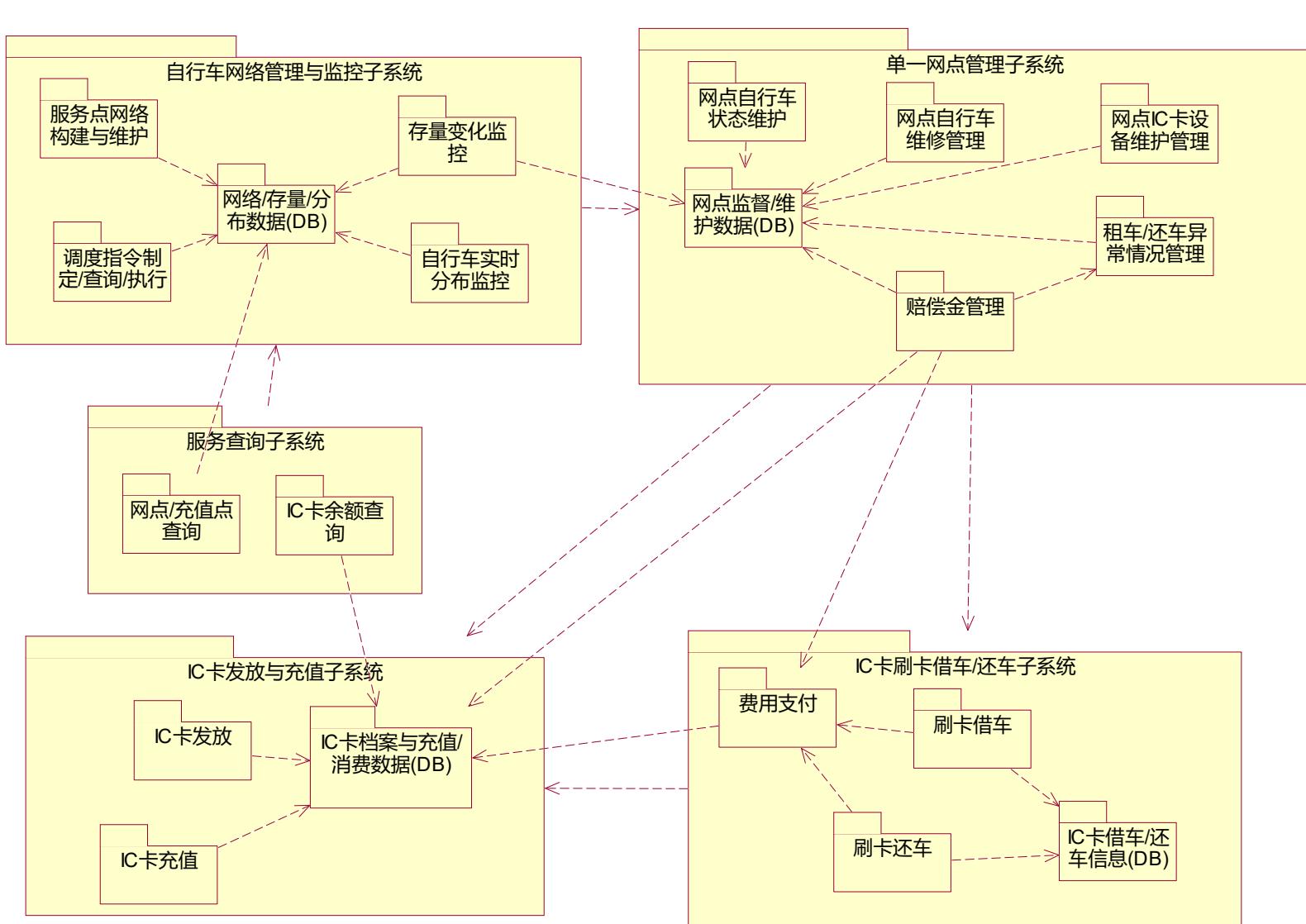
绘制包图的方法

- 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。

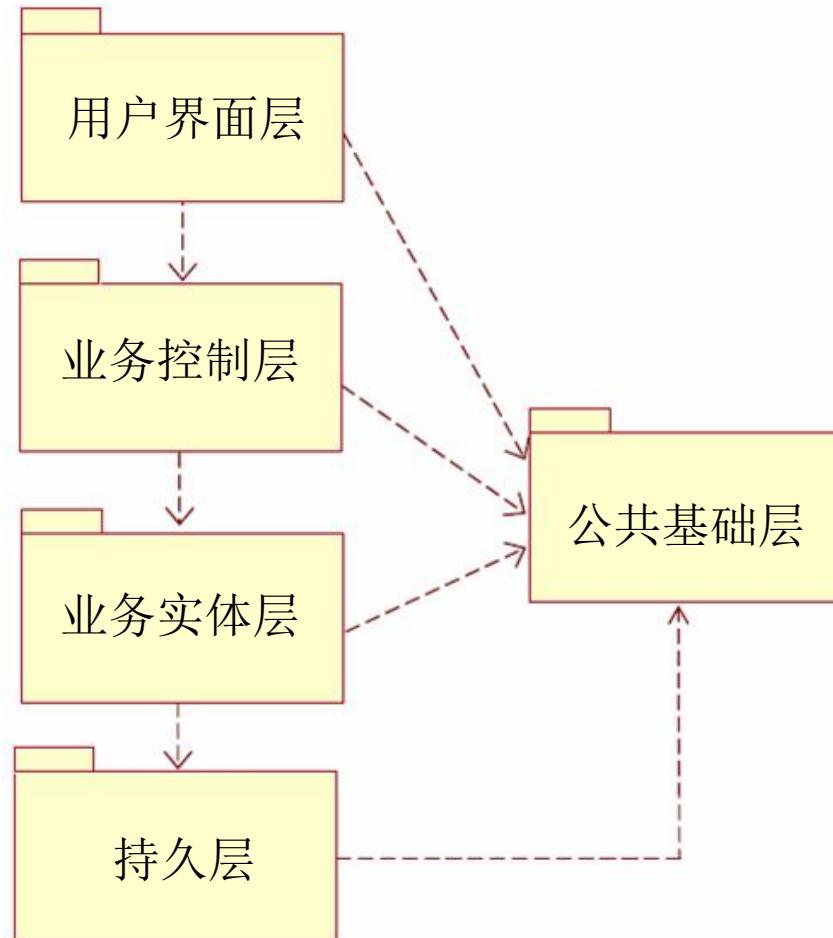
包图示例1



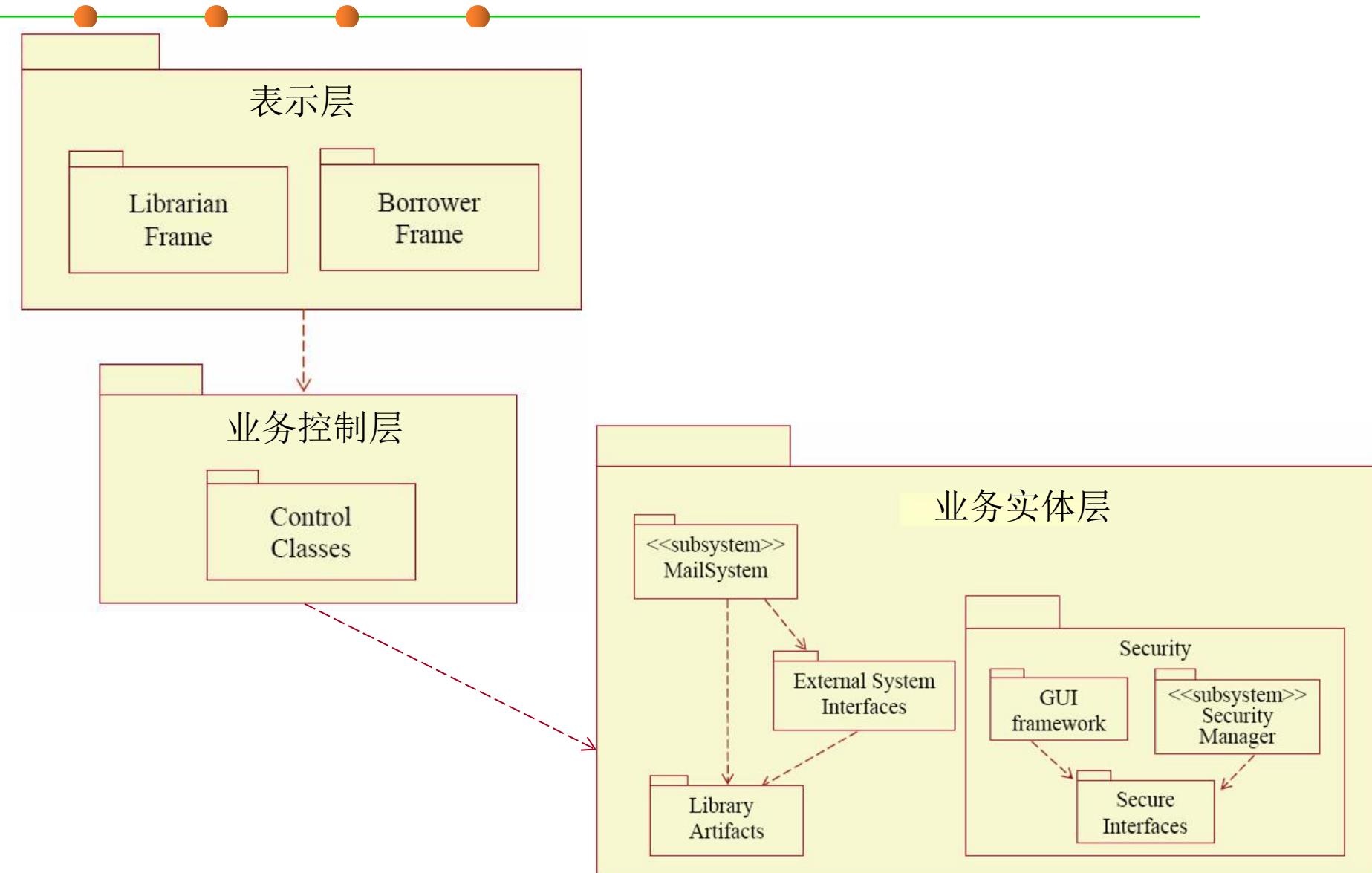
包图示例2



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;
```

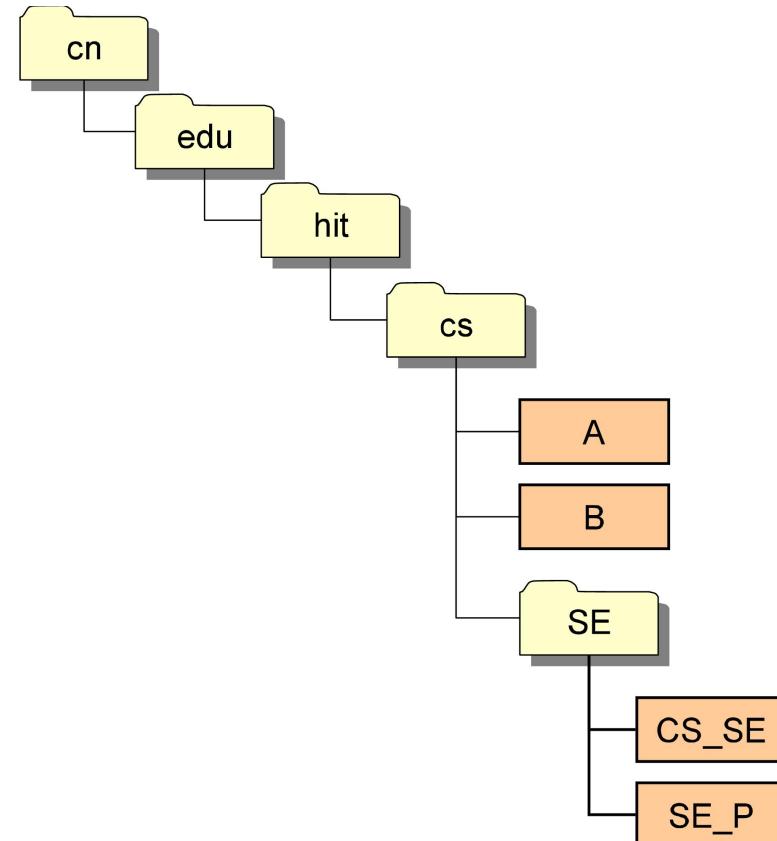
```
public class B{}
```

```
package cn.edu.hit.cs.se;
```

```
public class CS_SE{}
```

```
package cn.edu.hit.cs.se;
```

```
public class CS_SE_P{}
```



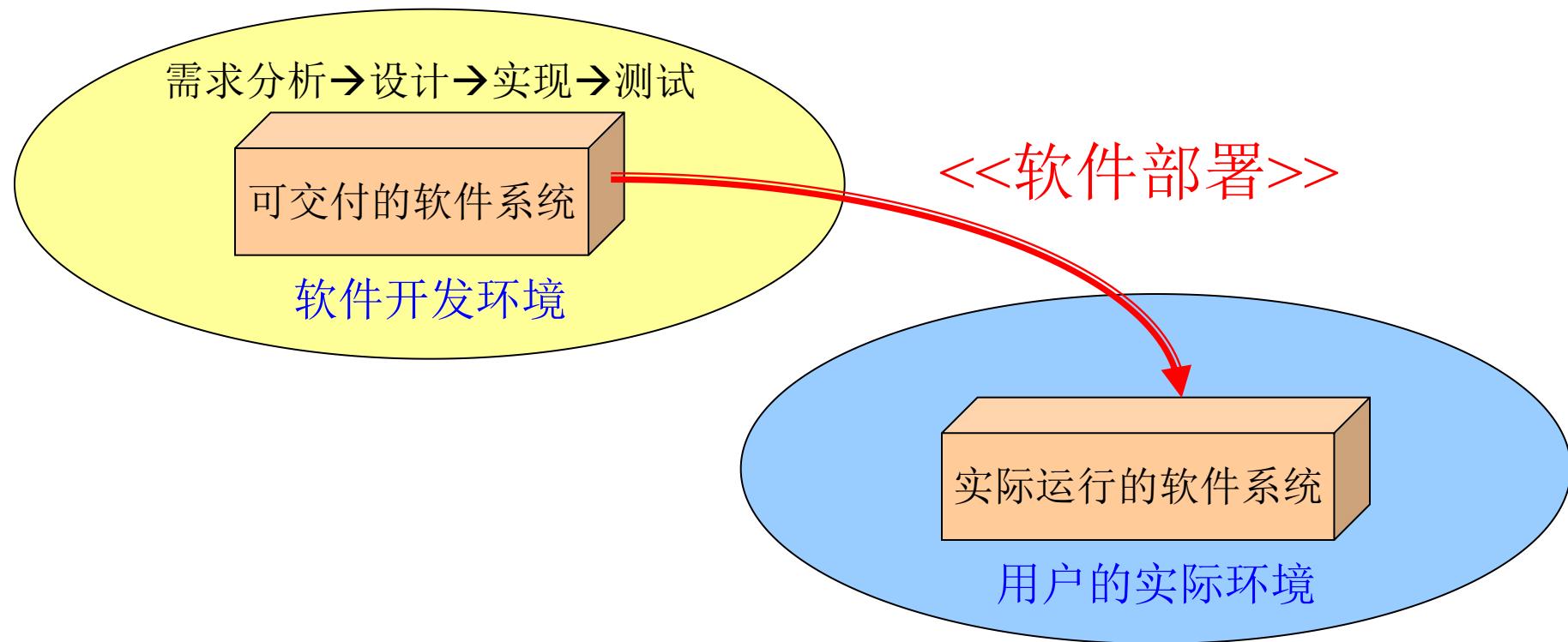


4. 软件部署



何谓“软件部署”

- 软件部署与实施(Software Deployment & Implementation): 将系统设计方案与软件系统转换成实际运行系统的全过程。

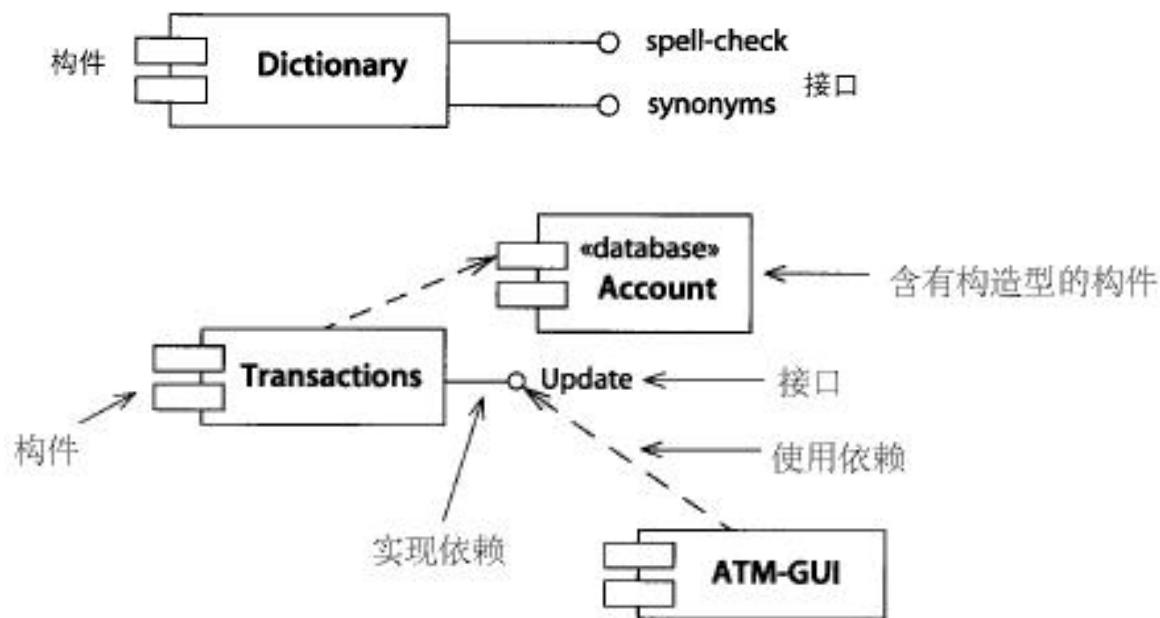


软件部署模型 (Deployment Model)

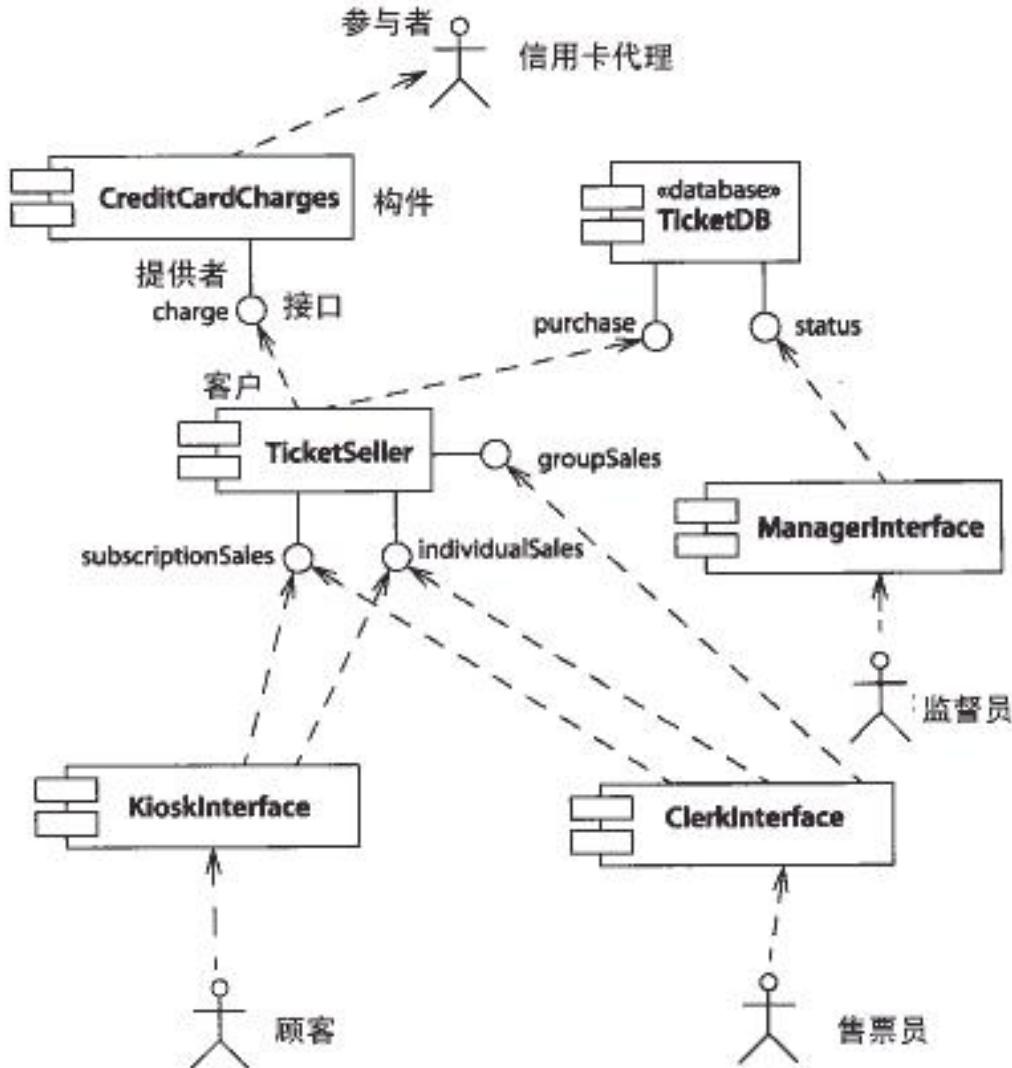
- 为系统选择硬件配置和运行平台；
 - 将类、包、子系统分配到各个硬件节点上。
-
- UML物理视图为系统的实现结构建模，例如系统的构件组织和建立在运行节点上的配置。这类视图提供了将系统中的类映射成物理构件和节点的机制。
 - 物理视图有两种：实现视图和部署视图。
 - 实现视图为系统的构件建立模型，还包括各构件之间的依赖关系，以便通过这些依赖关系来估计对系统构件的修改给系统可能带来的影响。**实现视图用“构件图” (component diagram) 来表达。**
 - 部署视图描述位于节点实例上的运行构件实例的安排。节点是一组运行资源，如计算机、设备或存储器。该视图用来评估分配结果和资源分配。**部署视图用“部署图” (deployment diagram) 来表达。**

构件图 (component diagram)

- 将系统中可重用的块包装成具有可替代性的物理单元，这些单元被称为构件。
- 构件是定义了良好接口的物理实现单元，它是系统中可替换的部分
- 每个构件体现了系统设计中特定类的实现。
- 良好定义的构件不直接依赖于其他构件而依赖于构件所支持的接口。

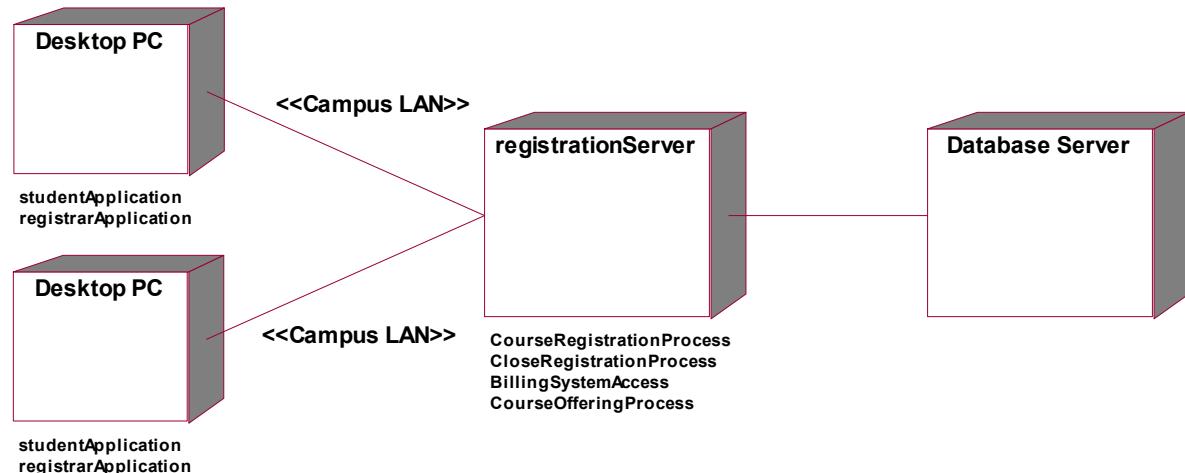


构件图示例



部署图 (Deployment diagram)

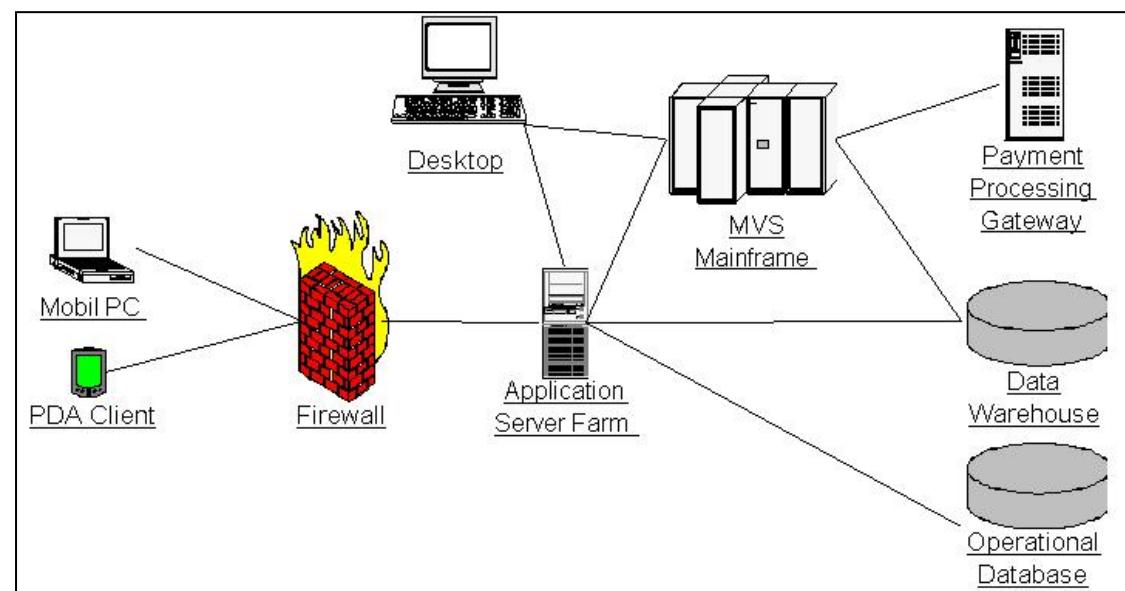
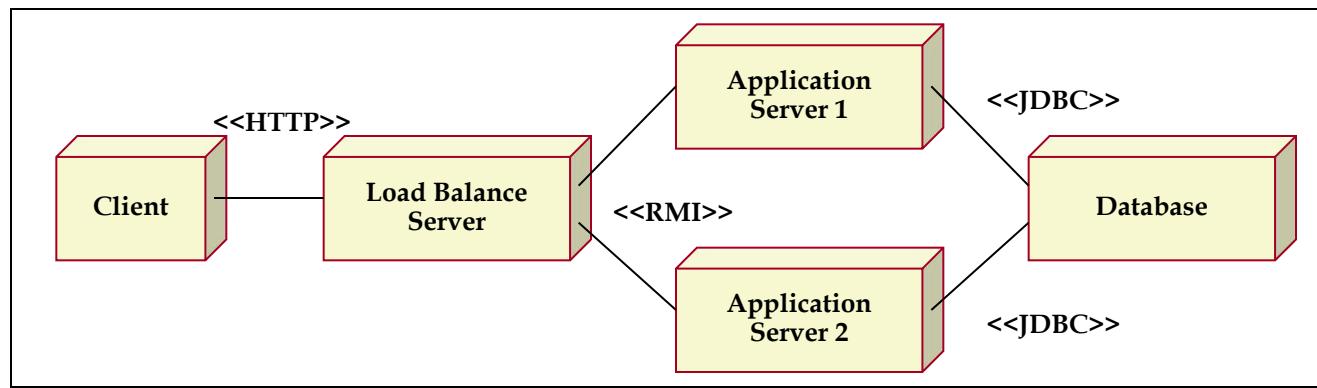
- 系统通常使用分布式的多台硬件设备，通过UML的部署图来描述：
 - 部署图反映了系统中软件和硬件的物理架构，表示系统运行时的处理节点以及节点中对象/子系统的分布与配置；
 - 描述系统中各硬件之间、软件实体之间的物理通讯关系；
 - 部署对系统的性能和复杂度具有较大影响，需要在设计初期就要完成。
- 目的：
 - 明确构件的分布
 - 描述软硬件实体之间的物理通讯关系
 - 找出性能的瓶颈



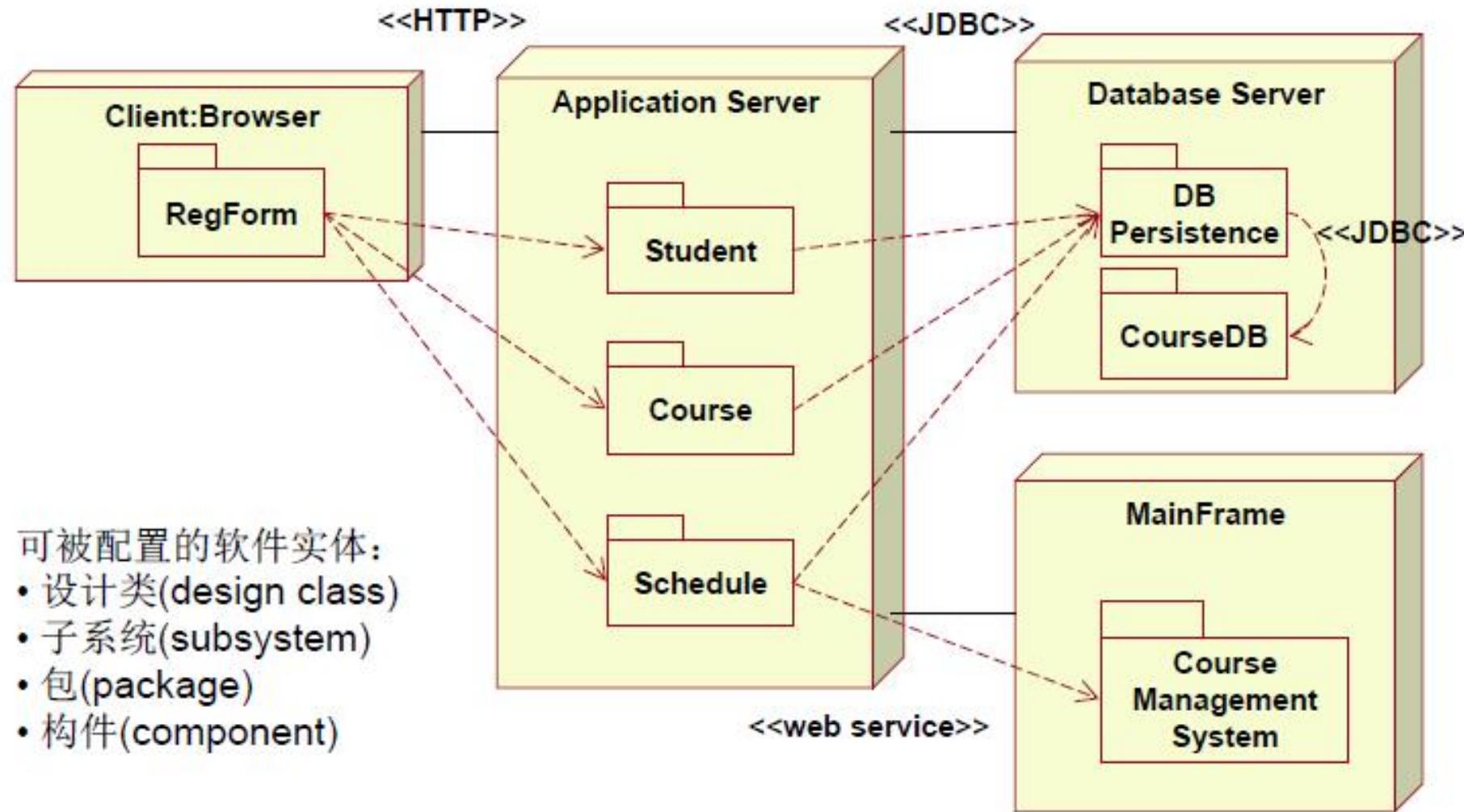
关于部署图

- 部署图(deployment diagram):
 - 节点(node): 一组运行资源，如计算机、设备或存储器等。每个节点用一个立方体来表示。
 - 节点的命名: client、Application Server、Database Server、Mainframe等较通用的名字。
 - 节点立方体之间的连接表示这些节点之间的通信关系，通常有以下类型：异步、同步；HTTP、SOAP；JDBC、ODBC；RMI、RPC等等。
- 部署图在两个层面的作用：
 - High-level: 描述系统中各硬件之间的物理通讯关系；
 - Low-level: 描述各软件实体被配置到哪个具体硬件上、这些软件实体之间的物理通讯关系。
- 由架构师、网络工程师和系统工程师开发

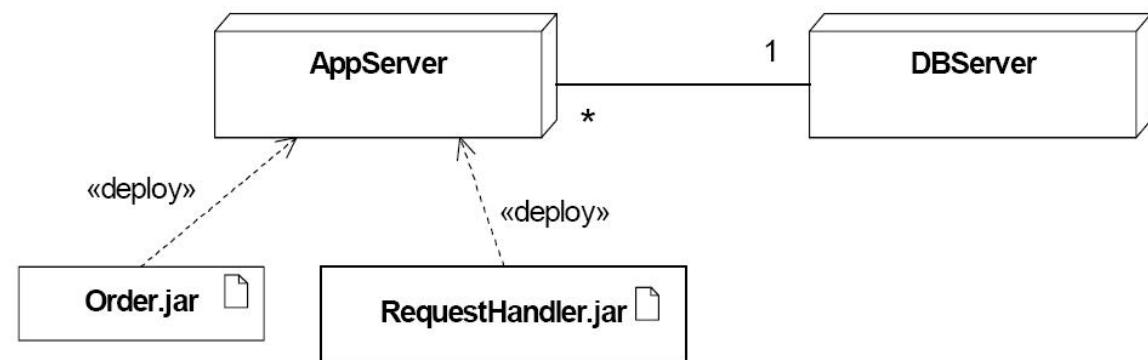
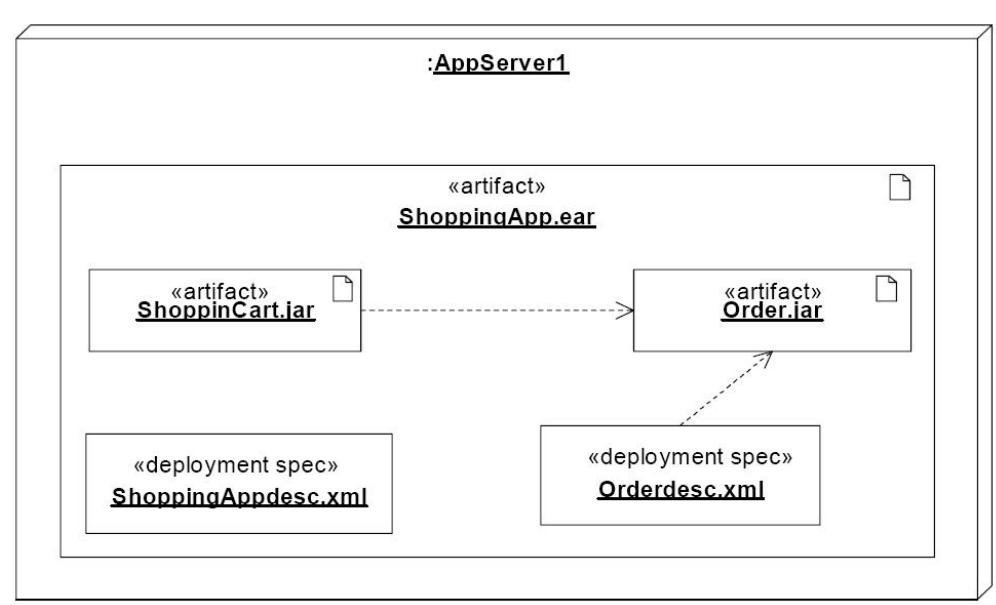
High-level Deployment Diagram



Low - level Deployment Diagram: 课程注册系统



Low-level Deployment Diagram



绘制部署图(deployment diagram)

- 确定“节点(node)”：
 - 标识系统中的硬件设备，包括大型主机、服务器、前端机、网络设备、输入/输出设备等。
 - 一个处理机(processor)是一个节点，它具有处理功能，能够执行一个组件；
 - 一个设备(device)也是一个节点，它没有处理功能，但它是系统和外部世界的接口。
- 对节点加上必要的“构造型(stereotype)”
 - 使用UML的标准构造型或自定义新的构造型，说明节点的性质。
- 确定“连接(connection)”
 - 把系统的包、类、子系统、数据库等内容分配到节点上，并确定节点与节点之间、节点与内容之间、内容与内容之间的联系，以及它们的性质。
- 绘制配置图(deployment diagram)

软件架构与软件部署

- 逻辑架构：只考虑如何分层、每个层次中的模块、层次内模块的关系、层次间模块的关系。主要是给开发者提供指南。
- 物理架构：考虑的是实际硬件/网络环境，以及如何将逻辑架构映射到硬件/网络环境上去。主要是给实施人员提供指南。
 - 通常，逻辑分层可以1:1映射到物理分层上；
 - 某些时候，多个逻辑层次可以部署在同一个物理层次上(n:1)；
 - 某些时候，同一个逻辑层次可以拆分为多个物理设施上(1:n)；
- 物理架构的设计思路：
 - 从逻辑架构入手，分别考虑每个逻辑层次在物理环境下是如何实现的；
 - 从简单到复杂，考虑每项设计决策对物理设施的要求，逐渐扩充物理架构。

软件部署的原则

- **最小化原则**

- 只需安装、部署和配置支撑软件运行和服务提供的最少软硬件要素，以提高软件系统和运行环境的精简性，提升目标软件系统的运行效率，减低运行和维护成本。

- **相关性原则**

- 部署的运行环境和软件系统要素均与系统建设相关联，剔除不想关的软硬件要素，防止将无关的软件要素部署到计算平台之中，以简化软件系统的部署和配置，降低软件运行和维护的复杂度。

- **适应性原则**

- 当软件系统的运行环境发生变化时，目标软件系统的部署也要随之发生变化，以确保目标软件系统部署的灵活性，提高目标软件系统的健壮性，提升软件部署和运维的自动化程度。

单机部署方式

- 将软件的各个要素（如可运行软构件、数据、文档等）集中部署到某个**单一的计算设备上**
 - 软件的运行环境只依赖于单一的计算设施
 - 不同软构件之间不存在网络通讯
 - 计算设施不仅仅是指各种计算机，如个人计算机、笔记本电脑或服务器等，还包括智能手机、智能手环等嵌入式计算设施
- 典型示例
 - 手机便签、闹钟和时钟、光盘刻录软件、扫雷游戏软件

分布式部署

- 将软件的各个要素（如可运行的软构件、数据和文档等）分散部署在多个计算设备上的部署方式
 - 基于C/S的部署方式
 - 基于客户端-应用服务器-数据库服务器的部署方式
 - 基于互联网的软件部署方式
- 典型示例
 - 智能驾驶、Google搜索引擎、淘宝和中国铁路“12306”

软件部署方法

基于中间件、开发框架和容器部署的软件

基于操作系统部署的软件

软件中间件

软件开发框架

软件容器

操作系统层 (如Windows、Linux、MacOS等计算机操作系统，及
Android、IOS、鸿蒙等移动智能终端操作系统)

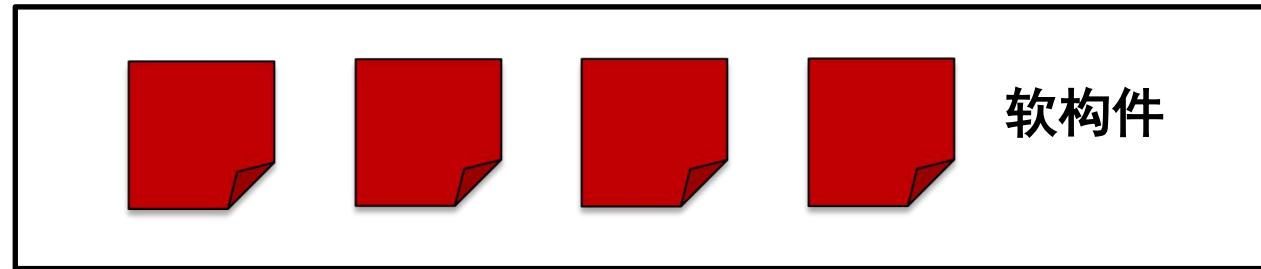
软件部署的目标平台 (如计算机、服务器、移动智能终端等)

软件部署方法：基于操作系统的部署

- 软件系统直接部署在目标计算设施的操作系统之上运行
 - 软件的运行仅依赖于计算设施上的操作系统及其提供的基础服务（如文件操作和管理、窗口界面的创建和操作等）
 - 操作系统构成了软件运行的上下文环境
- 典型示例
 - 移动端APP软件部署在移动端的操作系统之上，如智能手机、平板电脑等中的Android、IOS等
 - 桌面端软件主要部署在计算机上，常见的操作系统有Windows、Linux、MacOS等

软件部署方法：基于软件开发框架的部署

待部署的
目标软件系统

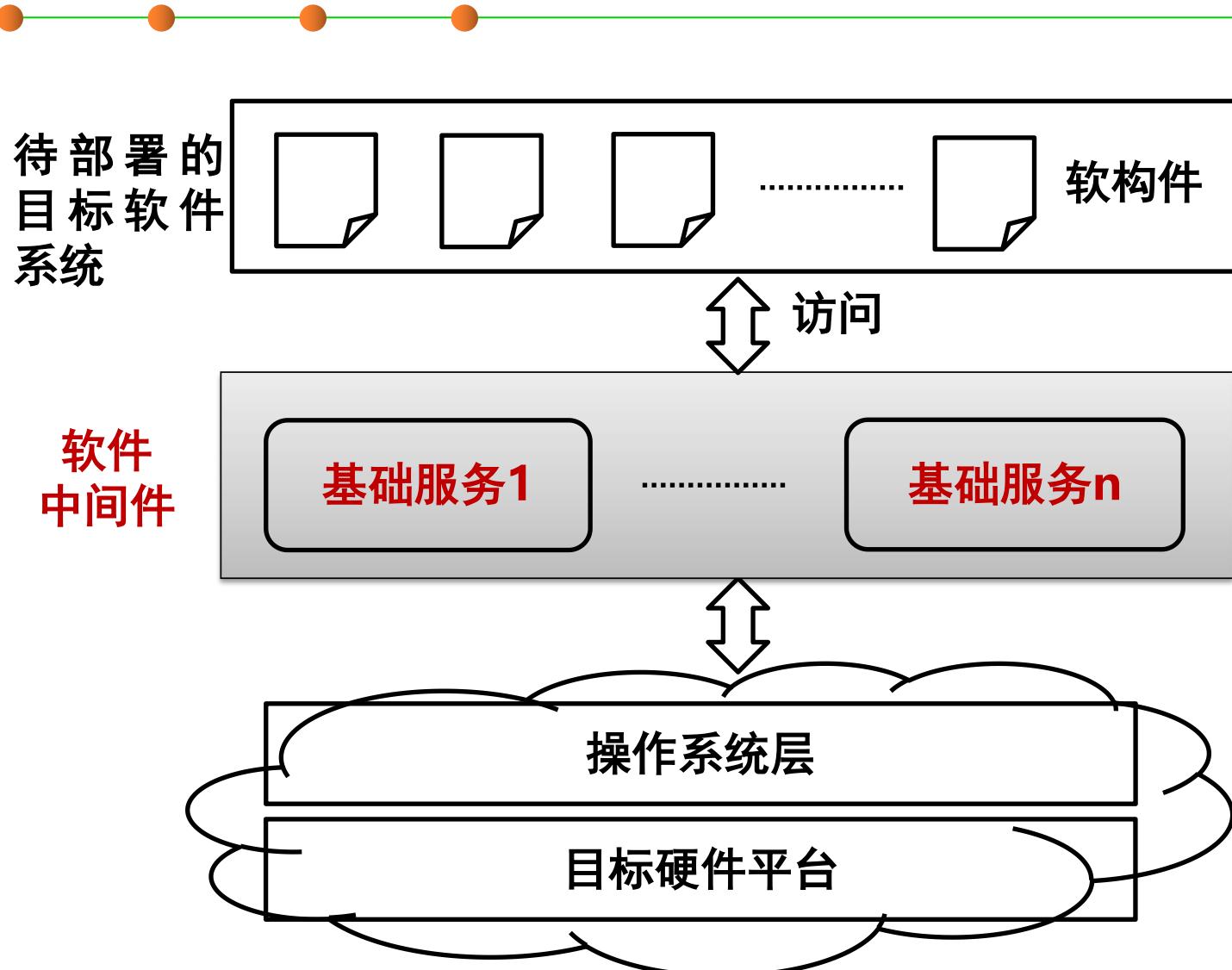


软件开发
框架

操作系统层

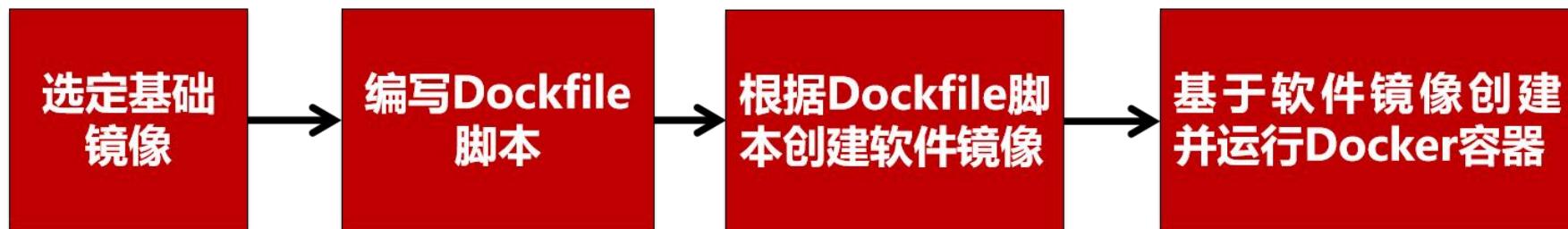
目标硬件平台

软件部署方法：基于软件中间件的部署



软件部署方法：基于容器和镜像的部署

- 容器是一个视图隔离、资源可限制、具有独立文件系统的进程集合
 - 容器作为软件运行的上下文
 - 容器具有独立文件系统，包括二进制文件、配置文件以及依赖
- 容器运行时所需要的所有文件集合称之为容器镜像，又叫做rootfs
 - 对于容器镜像而言，它打包的不仅仅是应用代码，而且还包括应用运行所需要的所有依赖文件
 - 容器镜像就是容器的文件系统，有了容器镜像，就可以构建该镜像的多个容器实例



SaaS软件的部署

- 三种选择：
 - 本机(主要用于开发环境)
 - 自己搭建服务器(组织内部使用)
 - 公共的服务器——[云](#)

什么是Cloud?

- 这是一种新的计算方式和共享基础架构的方法，IT相关的计算能力被作为“服务”，通过Internet向外部客户提供，但客户不需了解这些计算能力的物理来源及其分布。
- 目标：使IT计算能力(存储和计算)可以向电能一样提供给客户。



“自己造电厂”

VS



“订购电能”http://www.mipis.com

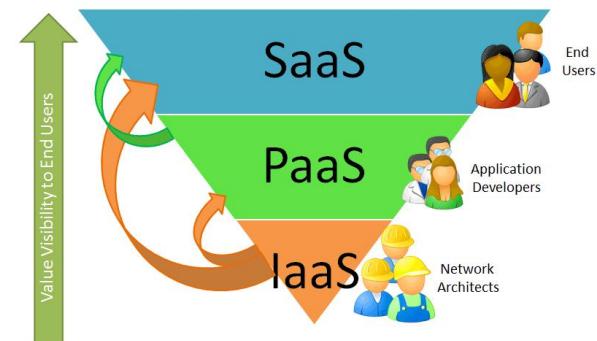
Cloud所能提供的三种典型服务

- **IaaS (Infrastructure as a Service, 基础架构即服务)**

- 通过互联网提供了数据中心、基础架构硬件，可以提供服务器、操作系统、磁盘存储、数据库和/或信息资源。
- 简单的说，IaaS可看作物理服务器(裸机，CPU+内存+磁盘)的虚拟化；用户可在上面安装操作系统、运行环境、装载数据，再在上面部署应用系统。

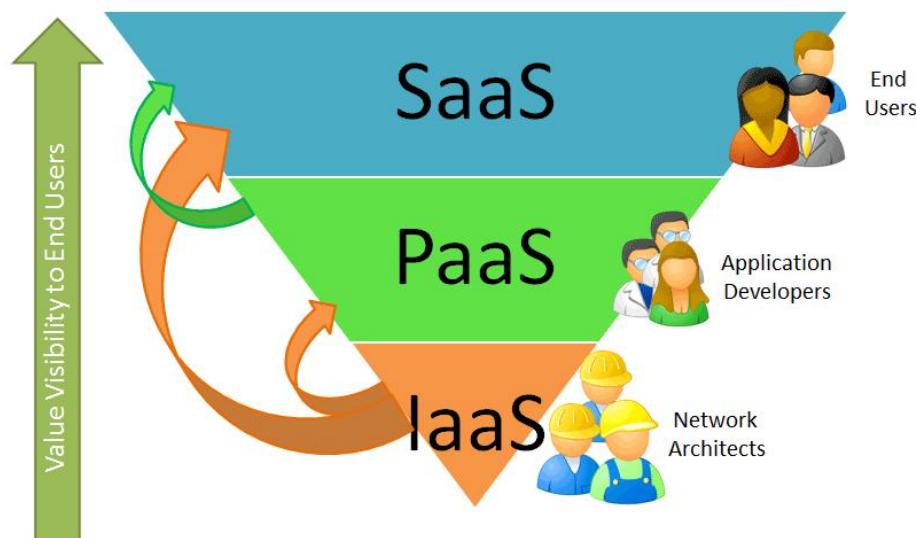
- **PaaS (Platform as a Service, 平台即服务)**

- 提供了软件基础架构，软件开发者可以在这个基础架构之上建设新的应用，或者扩展已有的应用。
- 在IaaS基础上，有了完整的运行环境和基础服务支持(例如OS、DB、应用服务器、MVC、Message等)。
- 将中间件环境作为了服务，向用户提供；
- 按照平台要求将程序部署到上面去。



Cloud所能提供的三种典型服务

- **SaaS (Software-as-a-service, 软件即服务)**
 - 一种通过Internet提供软件的模式，用户不用再购买软件，而改用向提供商租用基于web的软件，来管理企业经营活动，且无需对软件进行维护，服务提供商负责软件的可用性(软件维护、可扩展性、灾难恢复等)管理与支持；
- 企业采用**SaaS**服务模式，就像使用自来水和电能一样方便，从而大幅度降低了组织中应用先进技术的门槛与风险。
- 关键词：On-demand licensing and use



代表性云服务商

- 代表性云服务:

- 百度云
- 阿里云
- 腾讯云
- 华为云
- Amazon
- Google
- Microsoft
- OpenStack
- VMWare
- Eucalyptus



5. 数据库设计



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

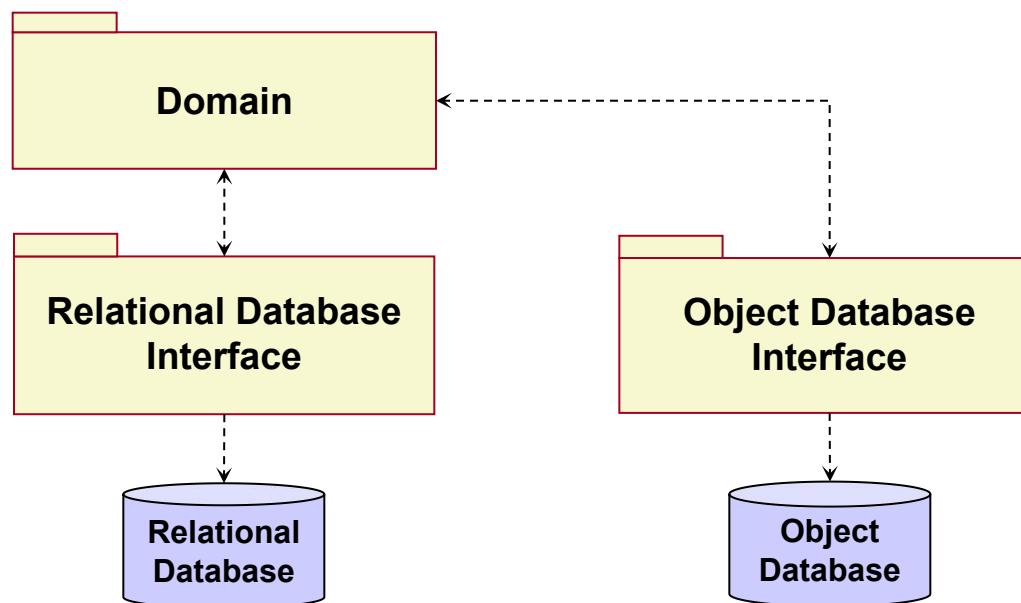
- “对象”只是存储在内存当中，而某些对象则需要永久性的存储起来。
 - 持久性数据(**persistent data**)
- 数据存储策略
 - **数据文件**: 由操作系统提供的存储形式，应用系统将数据按字节顺序存储，并定义如何以及何时检索数据。
 - **关系数据库**: 数据是以表的形式存储在预先定义好的称为Schema 的类型中。
 - **面向对象数据库**: 将对象和关系作为数据一起存储。
 - **存储策略的选择**: 取决于非功能性的需求。

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。



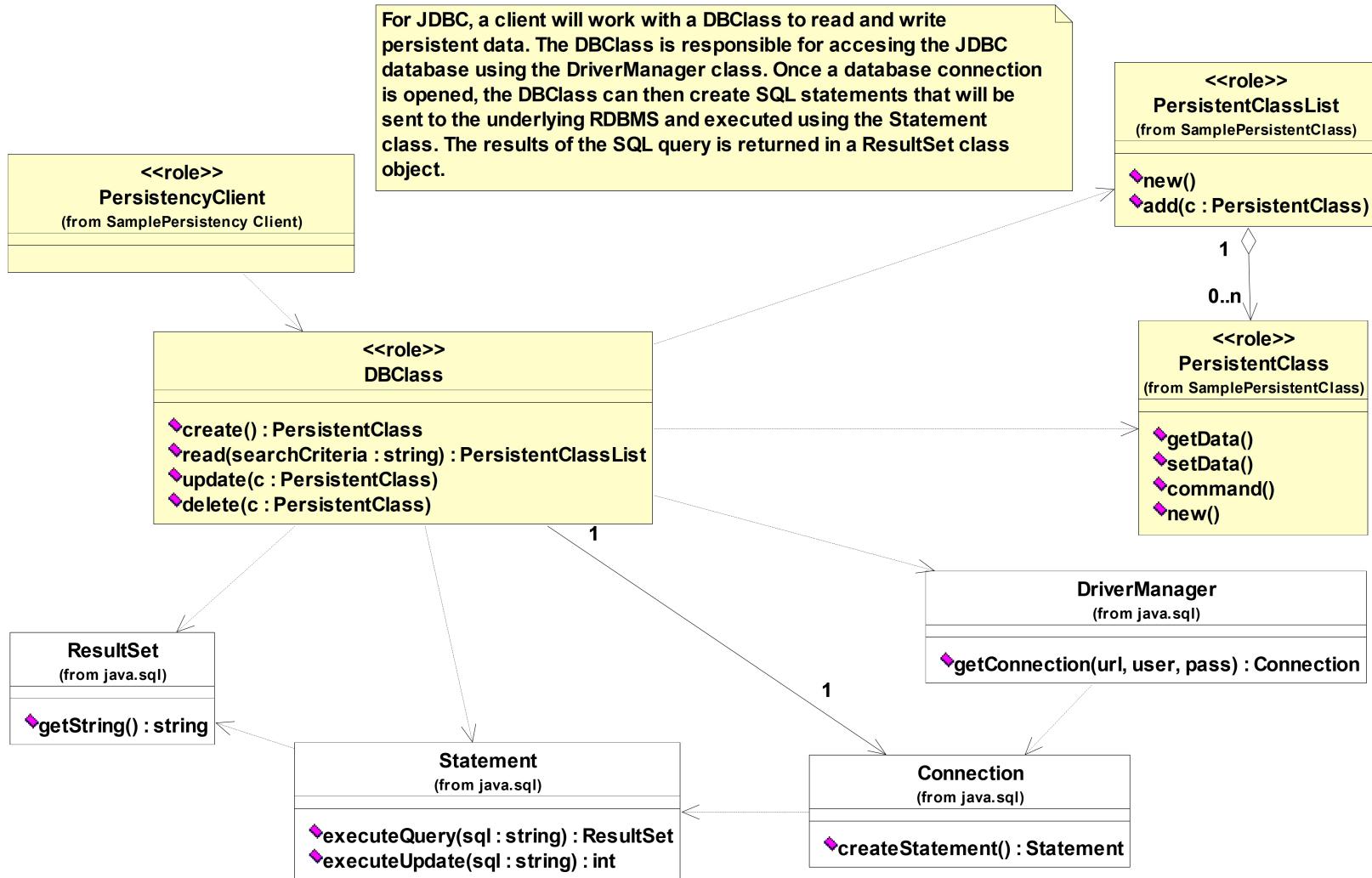
OO设计中的数据库设计

- 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
 - Mybatis
 - Spring Data JPA
 - Oracle TopLink
 - ...

Example: Persistence: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)。

Object
Model

Person	
name	
address	

Table
Model

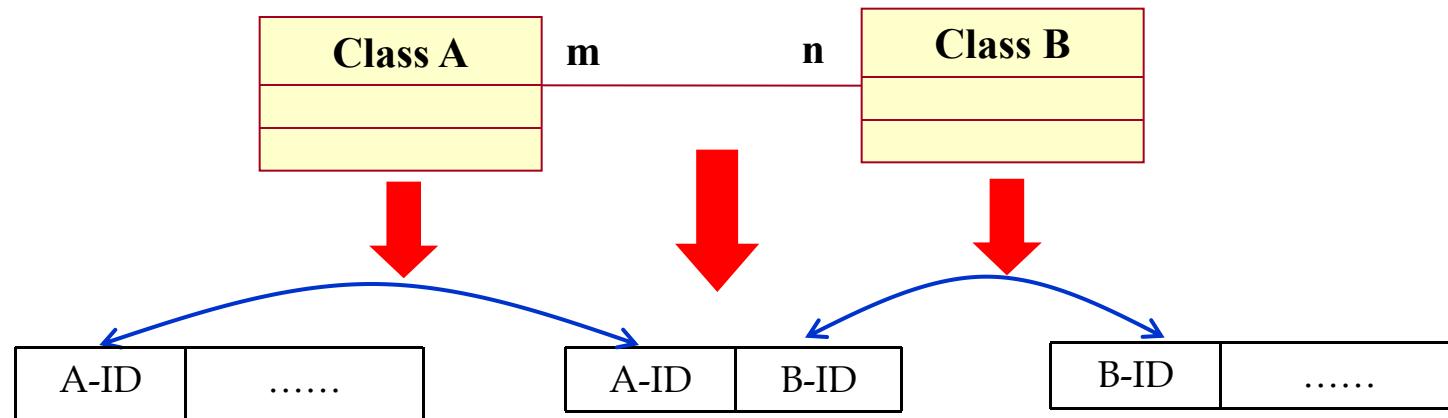
Attribute Name	nulls	Datatype
Person_ID	N	int
Person_name	N	Char(20)
address	Y	Char(50)

SQL
Code

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

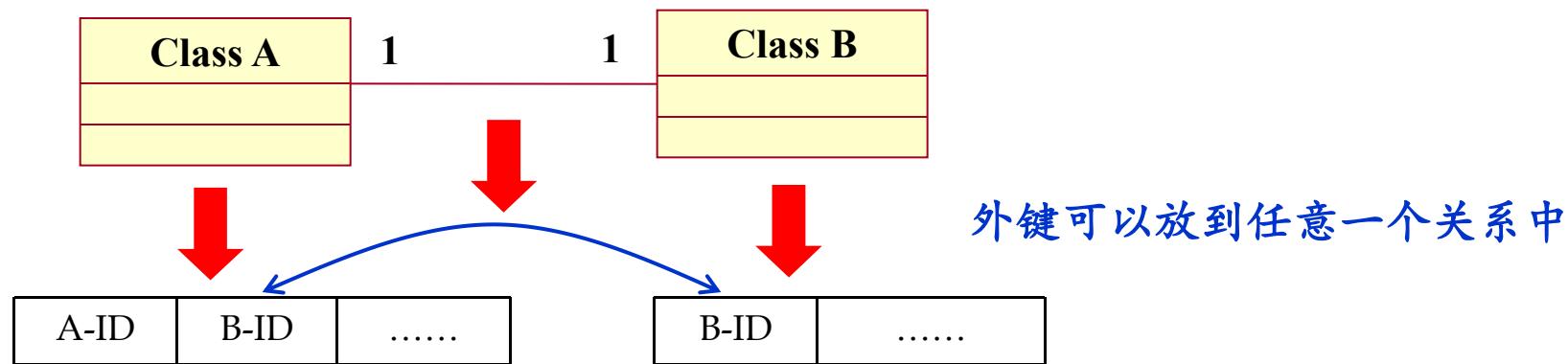
将关联映射到关系数据库(统一的、简单的途径)

- 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。

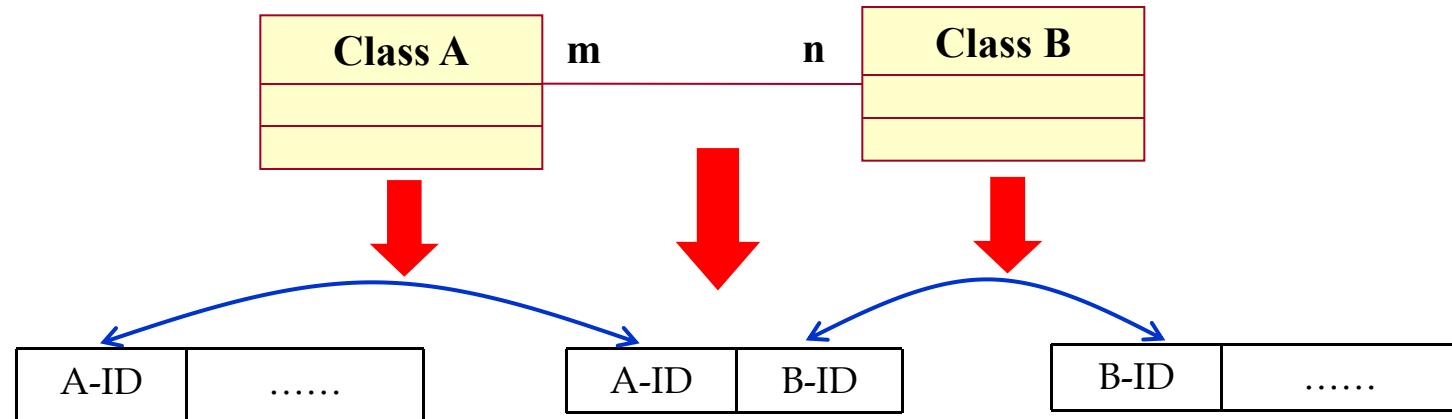


将关联映射到关系数据库(1:1和m:n的关联关系)

- Implementing 1 to 1

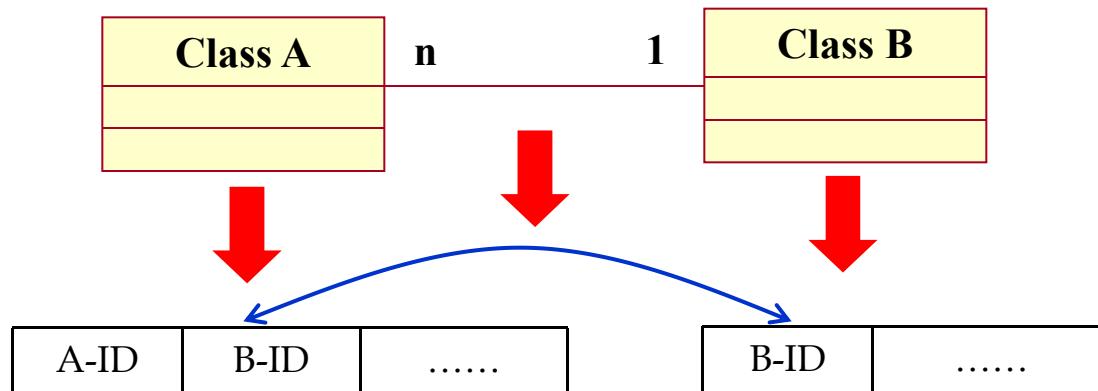
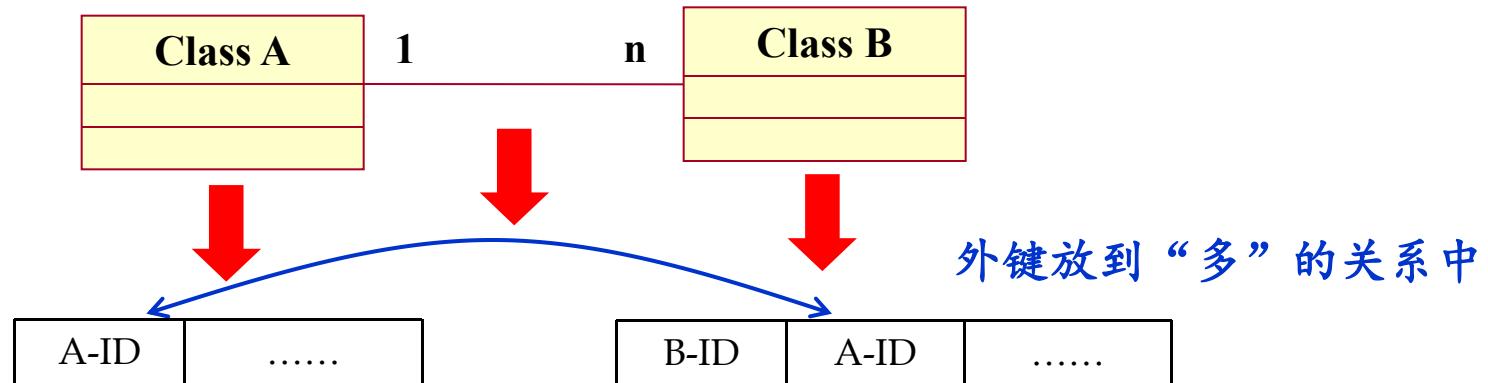


- Implementing m:n

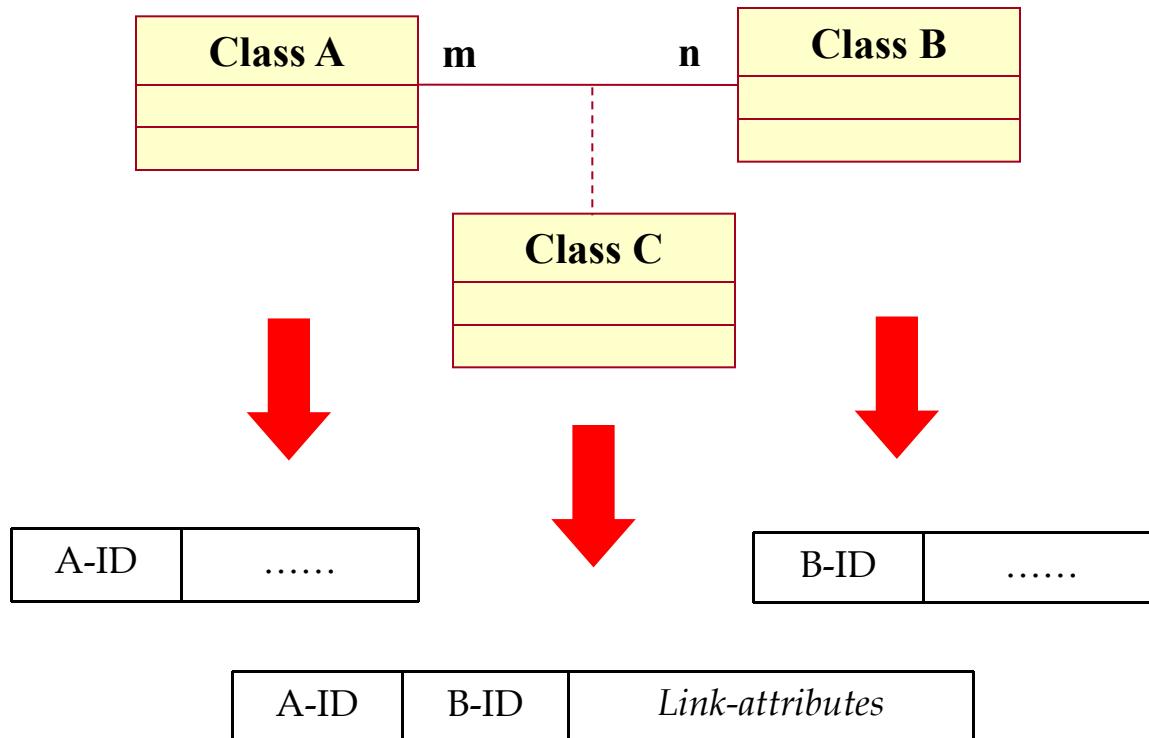


将关联映射到关系数据库(1:n的关联关系)

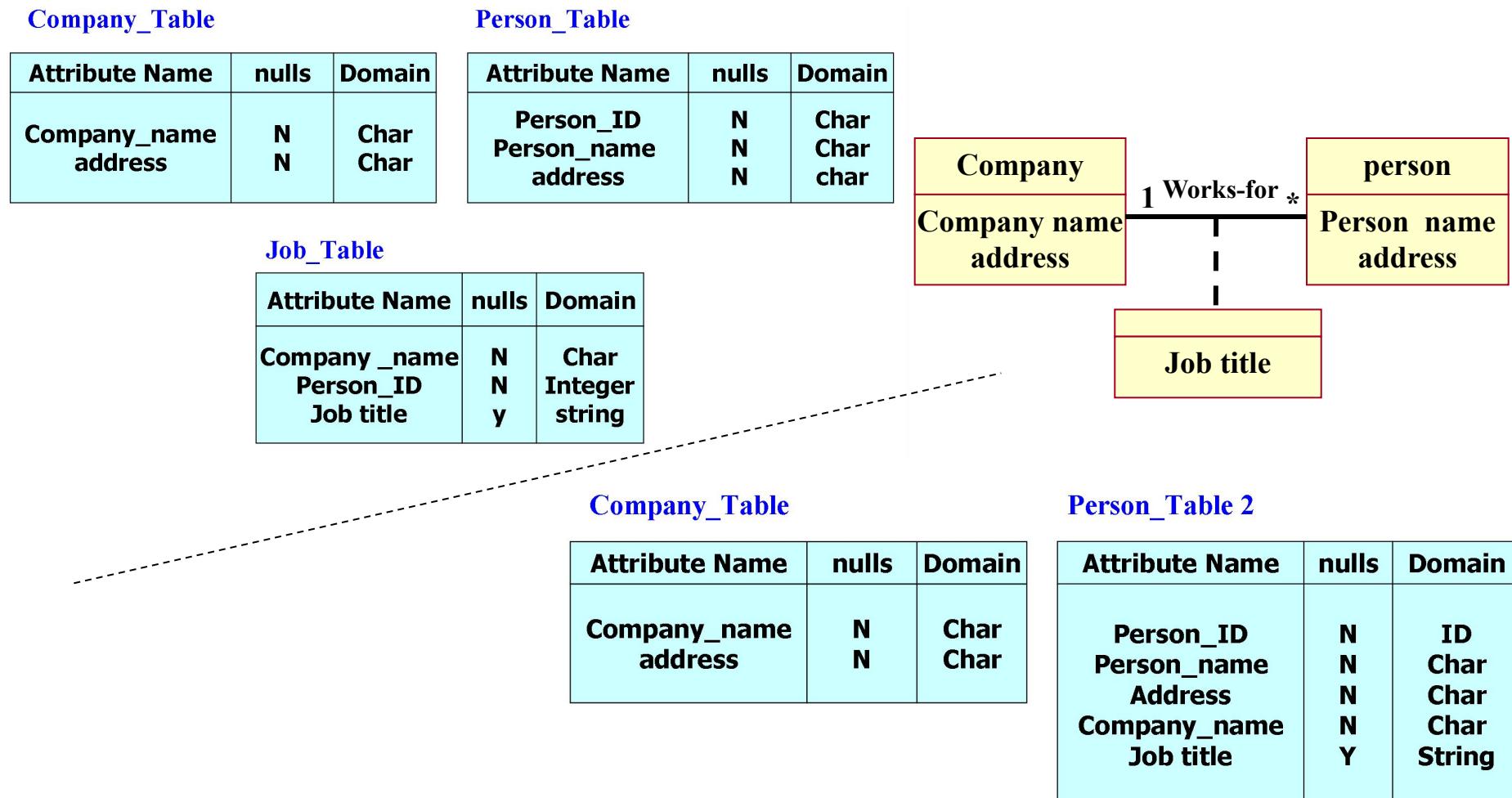
- Implementing 1:n



将关联映射到关系数据库(基于关联类的关联关系)

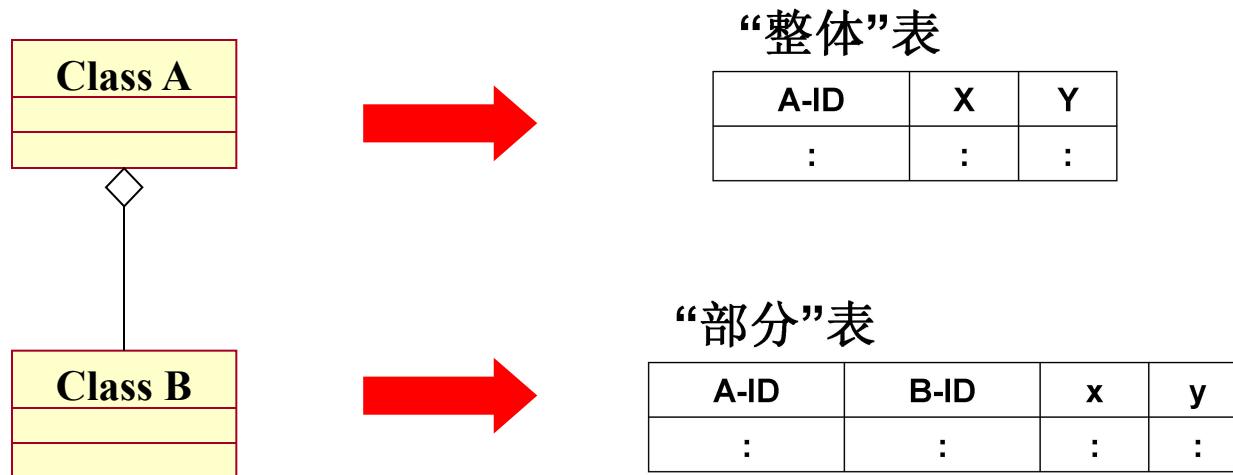


将关联映射到关系数据库(基于关联类的关联关系)



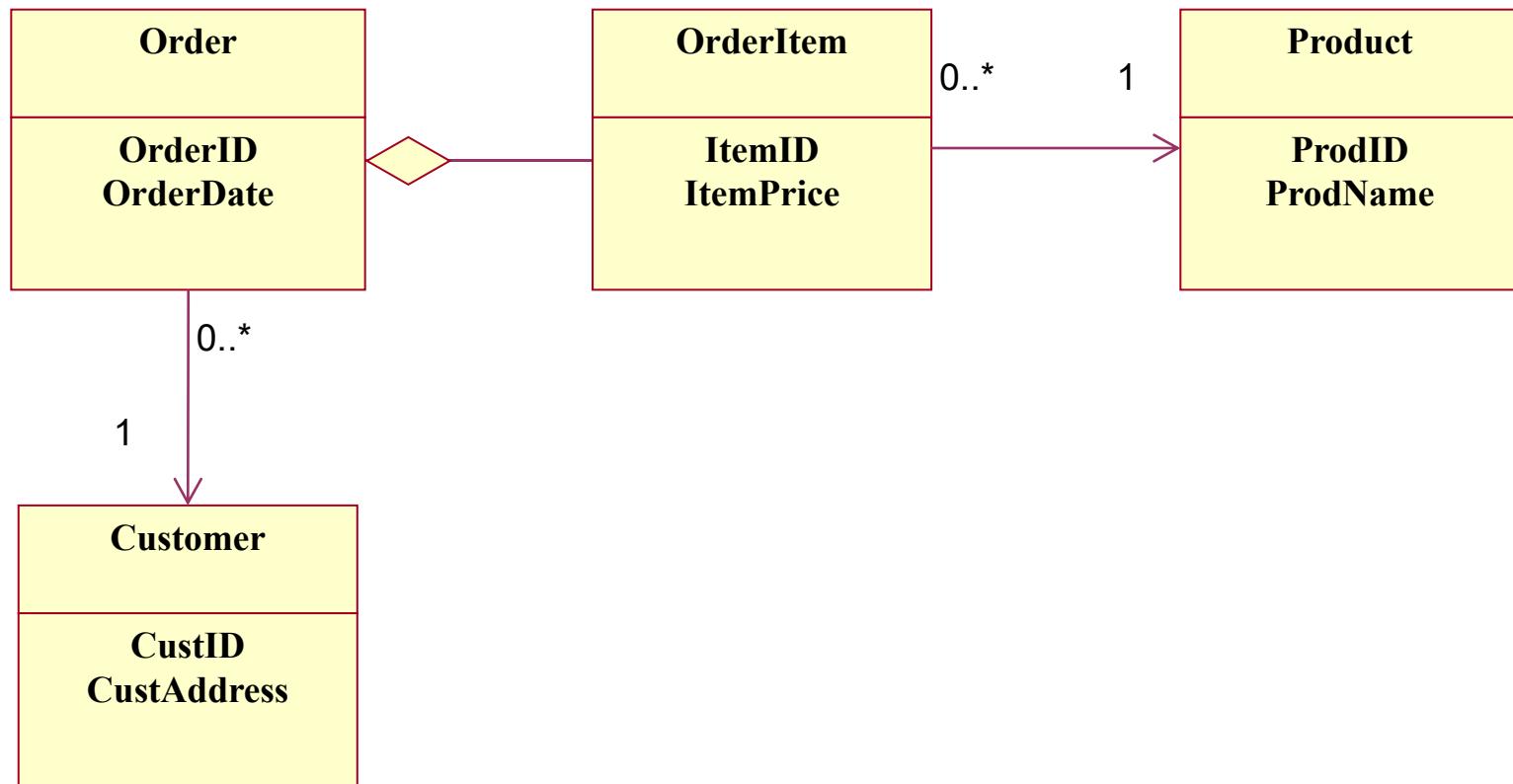
将组合/聚合关系映射到关系数据库

- 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



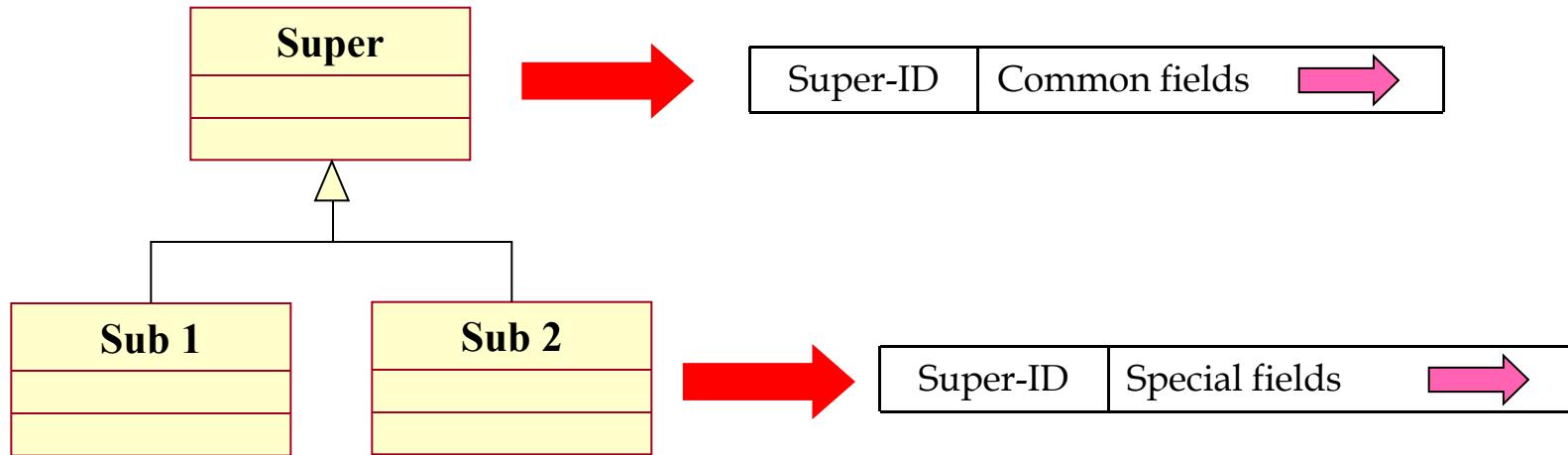
OO到DB的映射

- 为以下类图设计关系数据表



将继承关系映射到关系数据库

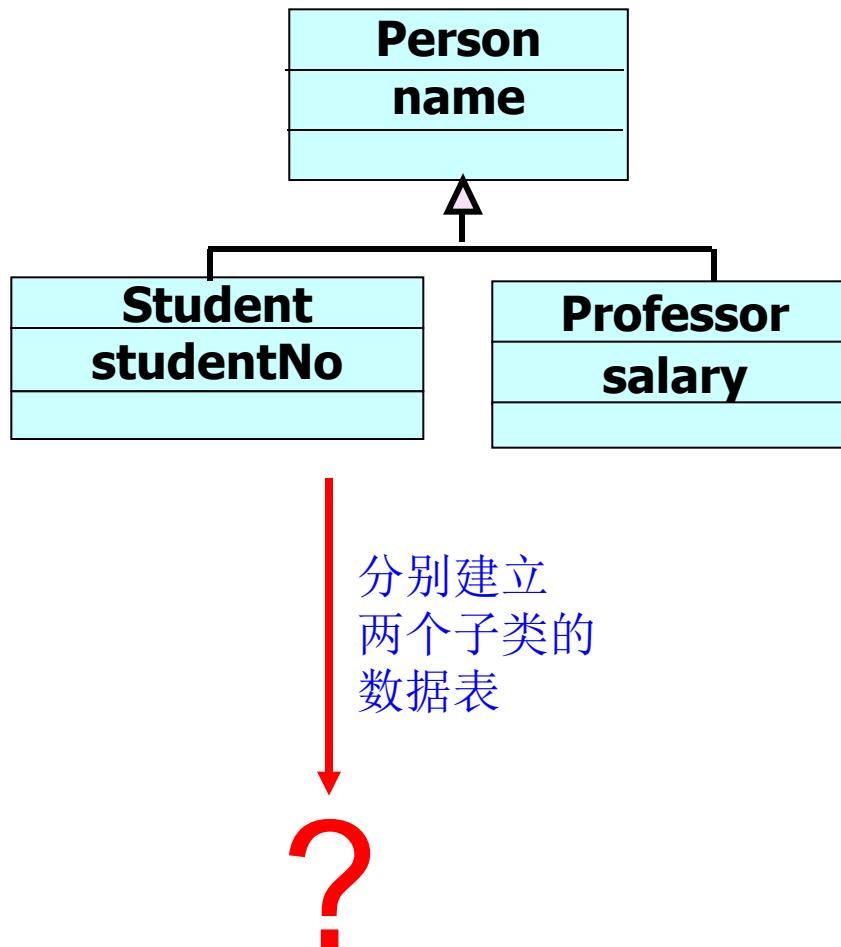
- 策略1：分别建立父类和子类的三张数据表



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



只建立父类的一个数据表



分别建立
两个子类的
数据表



分别建立
父类和子类的
三张数据表



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？



6. 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

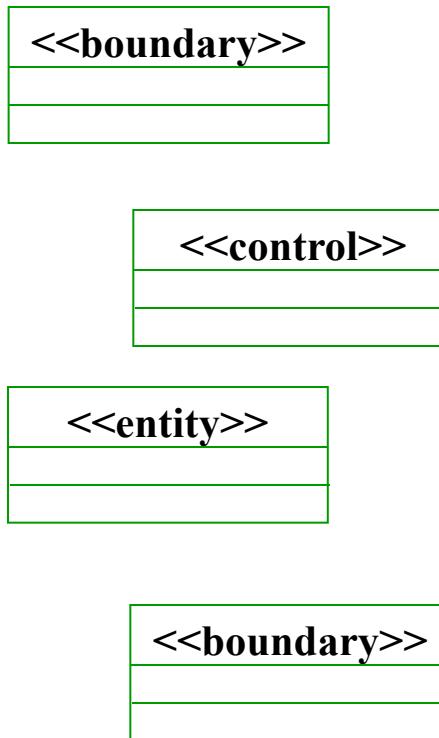
对象设计的基本步骤

- 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

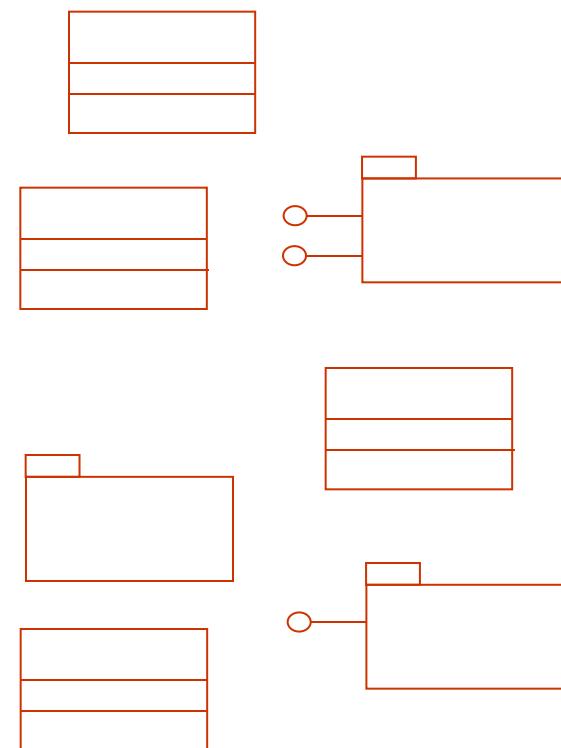


1. 创建初始的设计类

分析类



设计元素



多对多映射

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

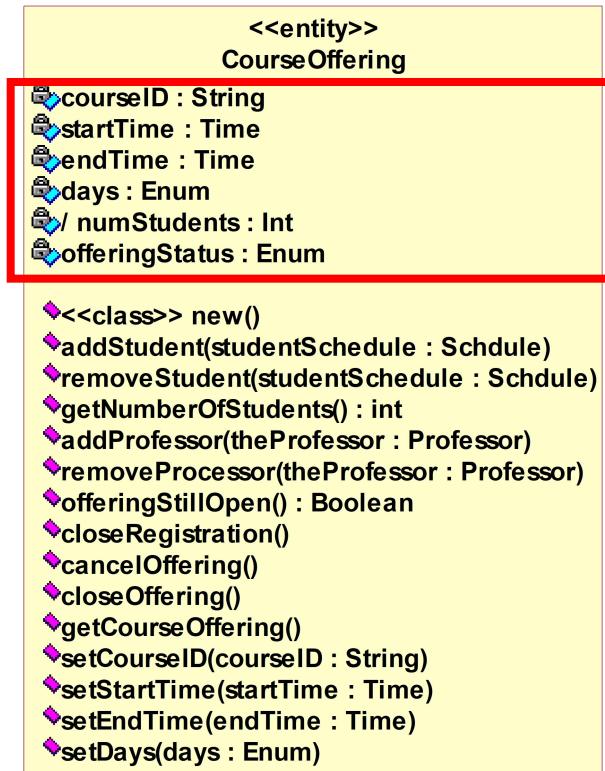
visibility attributeName : Type = Default

- 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

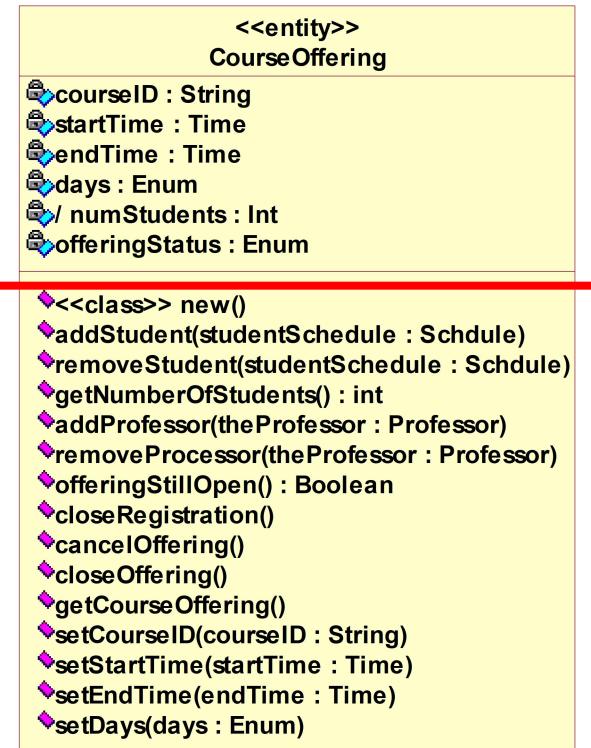
addProfessor (theProfessor:Professor)

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

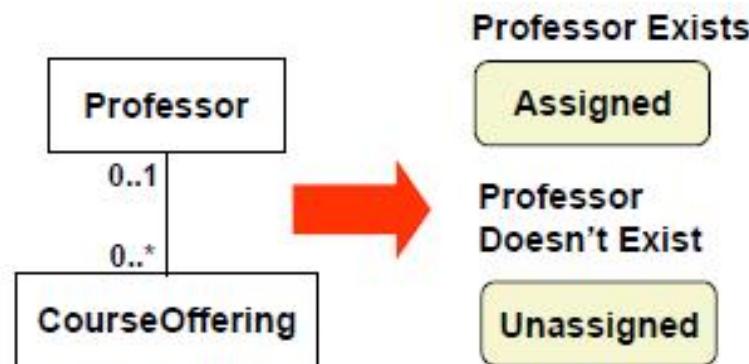
- 目的:
 - 设计一个对象的状态是如何影响其行为;
 - 绘制对象状态图。
- 需要考虑的要素:
 - 哪些对象有状态?
 - 如何发现对象的所有状态?
 - 如何绘制对象状态图?
- Example: CourseOffering

`numStudents < maximum`

Open

`numStudents >= maximum`

Closed



Professor Exists

Assigned

Professor Doesn't Exist

Unassigned

状态图 (Statechart Diagram)



- UML的状态图
 - 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。
 - 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。

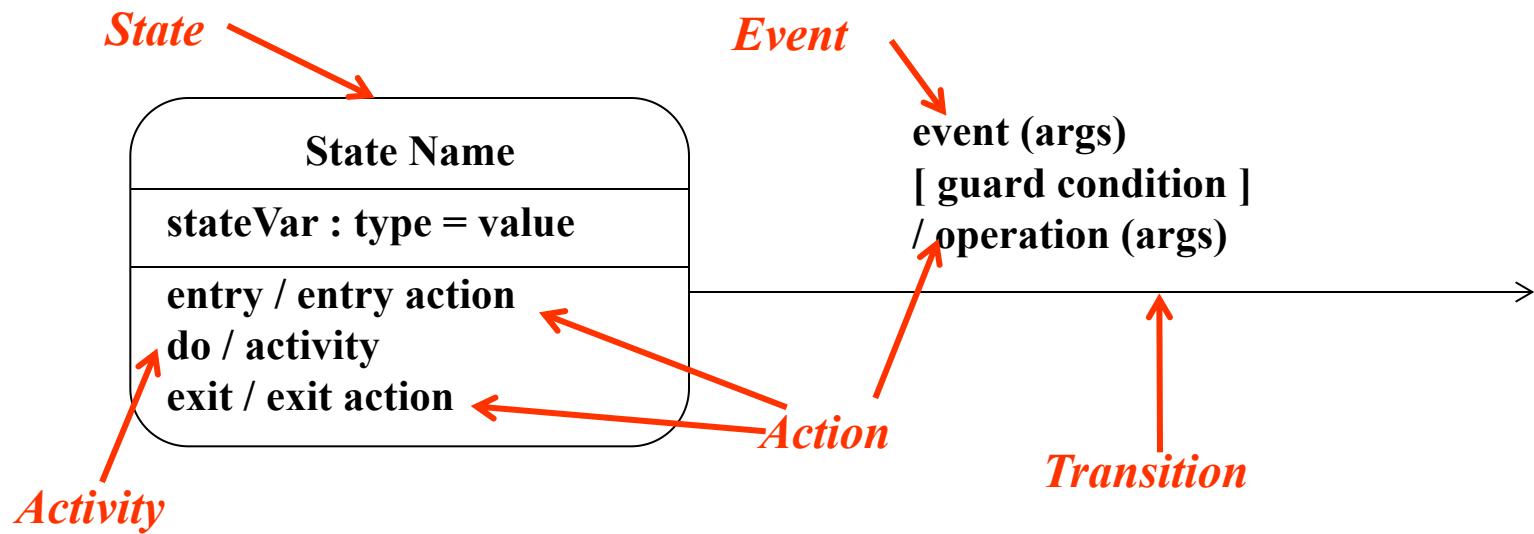
状态图使用原则

- 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。

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)



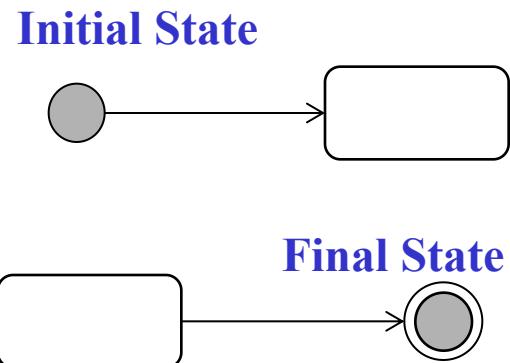
状态图：状态



- 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。

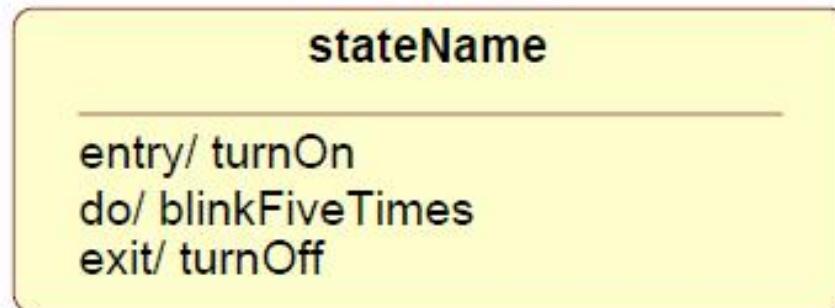
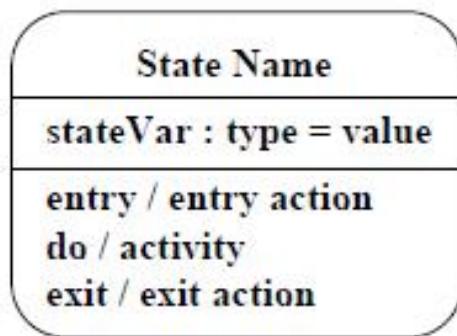
■ 状态图中定义的状态

- 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

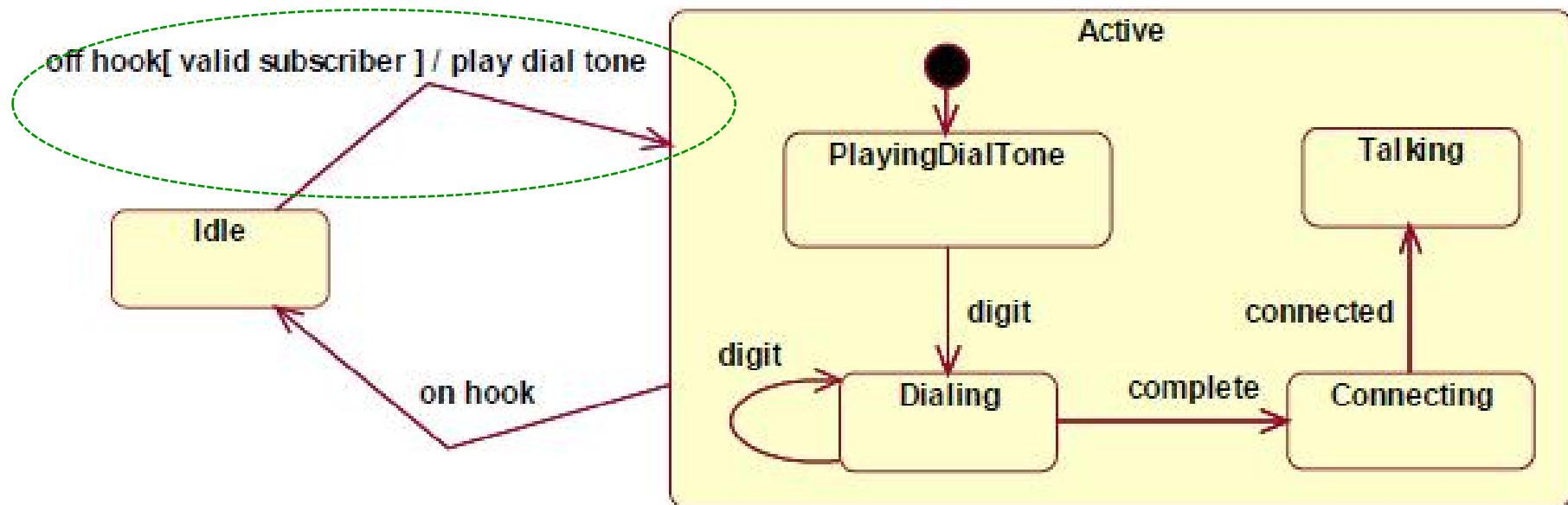
- 中间状态包括两个区域：名字域和内部转移域，如下图所示，其中内部转移域是可选的。
 - 入口动作表达式，**entry/action _ expression**：状态到达/进入时执行的活动。
 - 出口动作表达式，**exit/action _ expression**：状态退出时执行的活动。
 - 内部动作表达式，**do/activity _ expression**：表示某对象处在某状态下的全部或部分持续时间内执行的活动。（如：复印机卡纸状态下闪烁5次灯）



状态图：转换

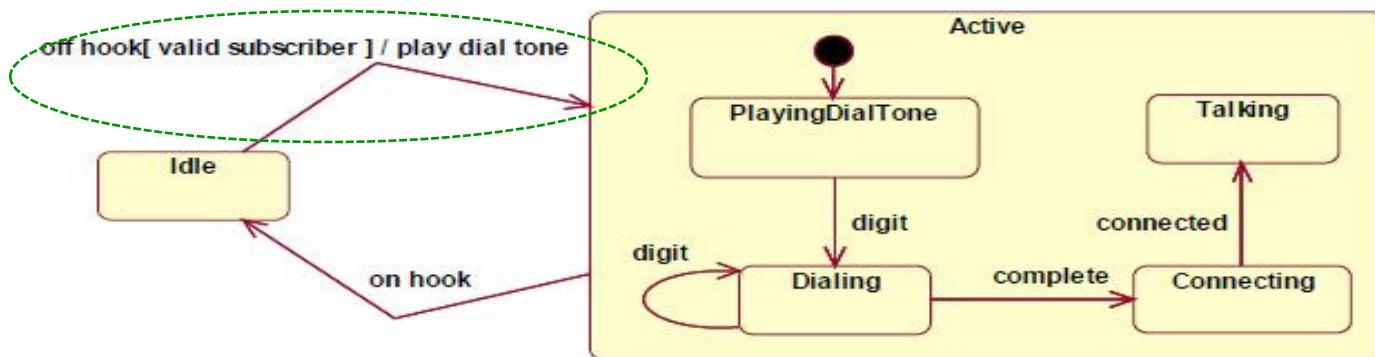
- 转换/转移/迁移(Transition)

- 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。

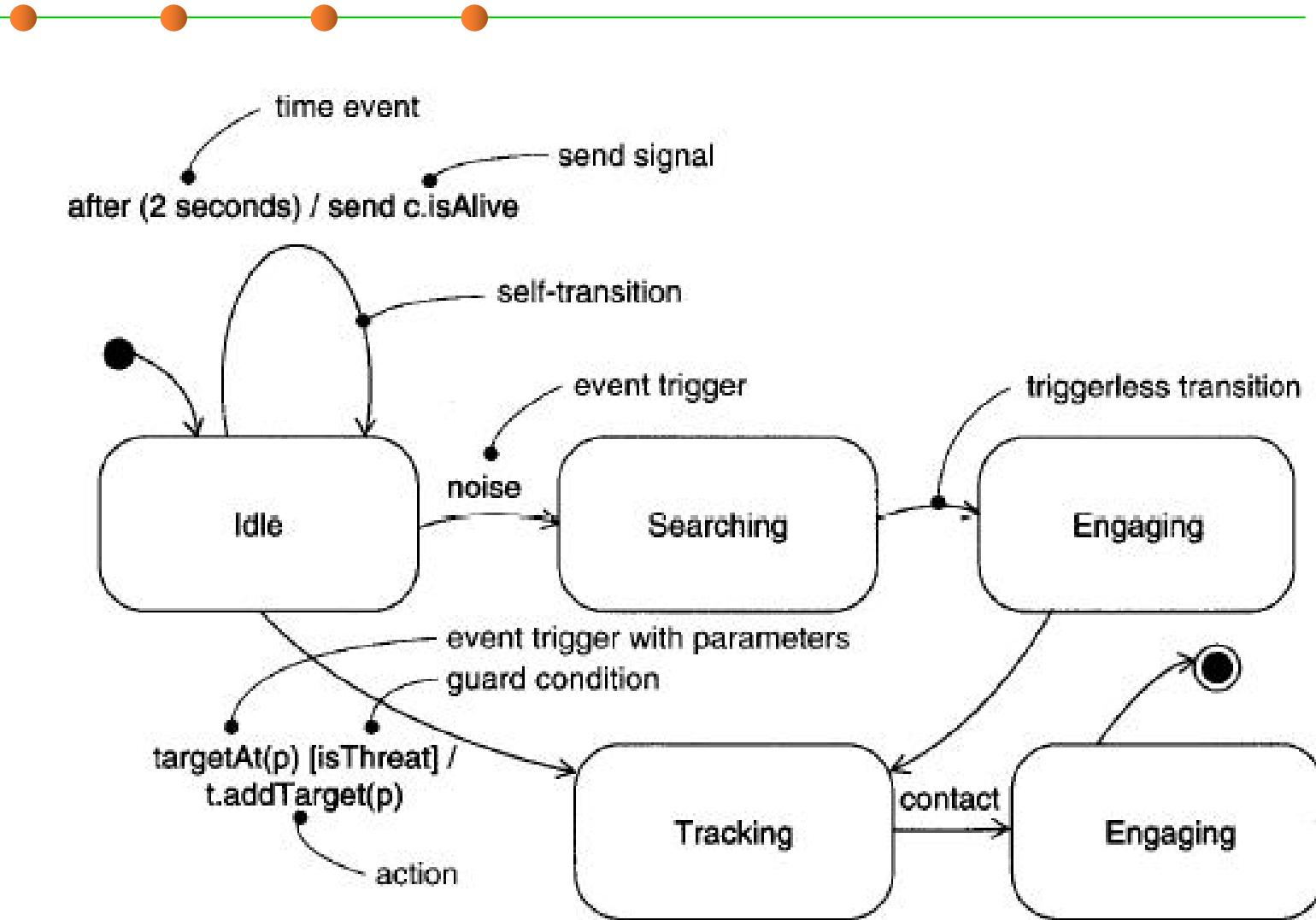


状态图：转换

- 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

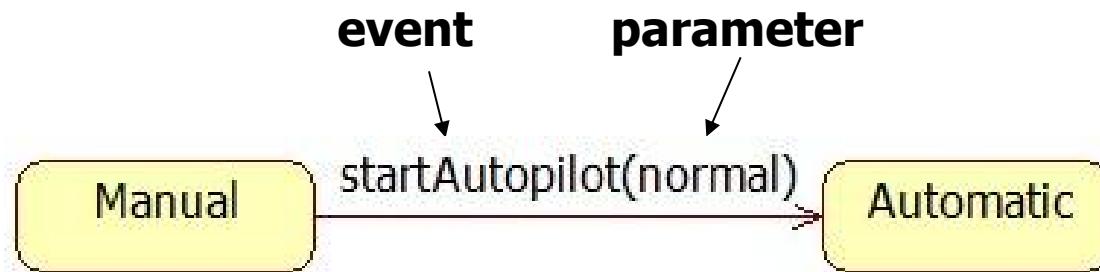


- UML中事件的分类

- Call event (调用事件)
 - Change event (变化事件)
 - Time event (时间事件)
 - Signal event (信号事件)

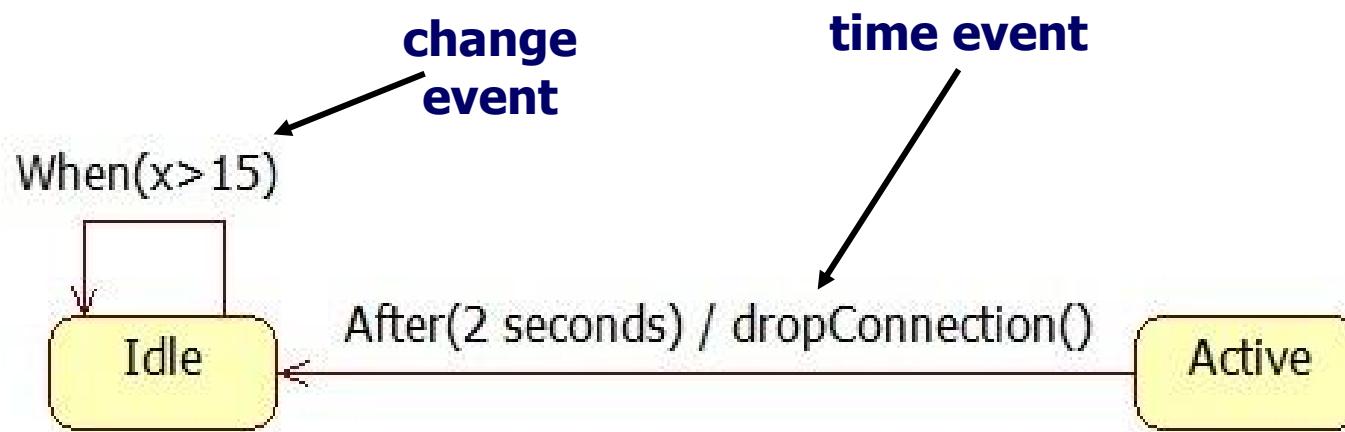
调用事件

- 调用事件(Call event)



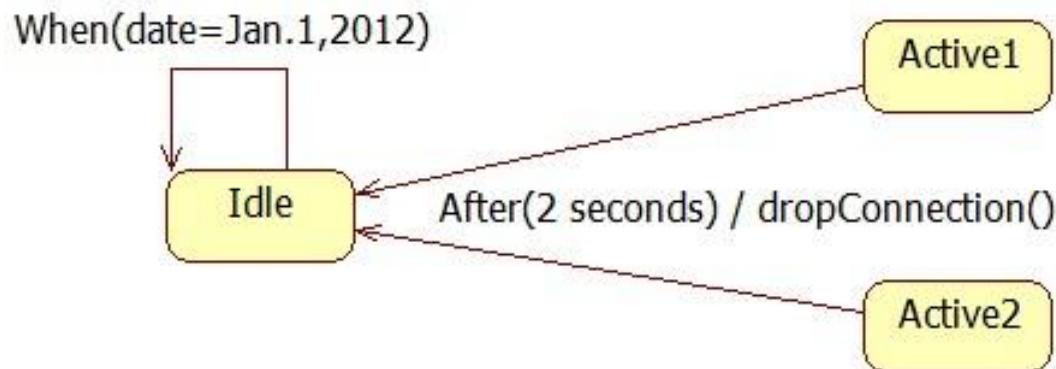
变化事件

- 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

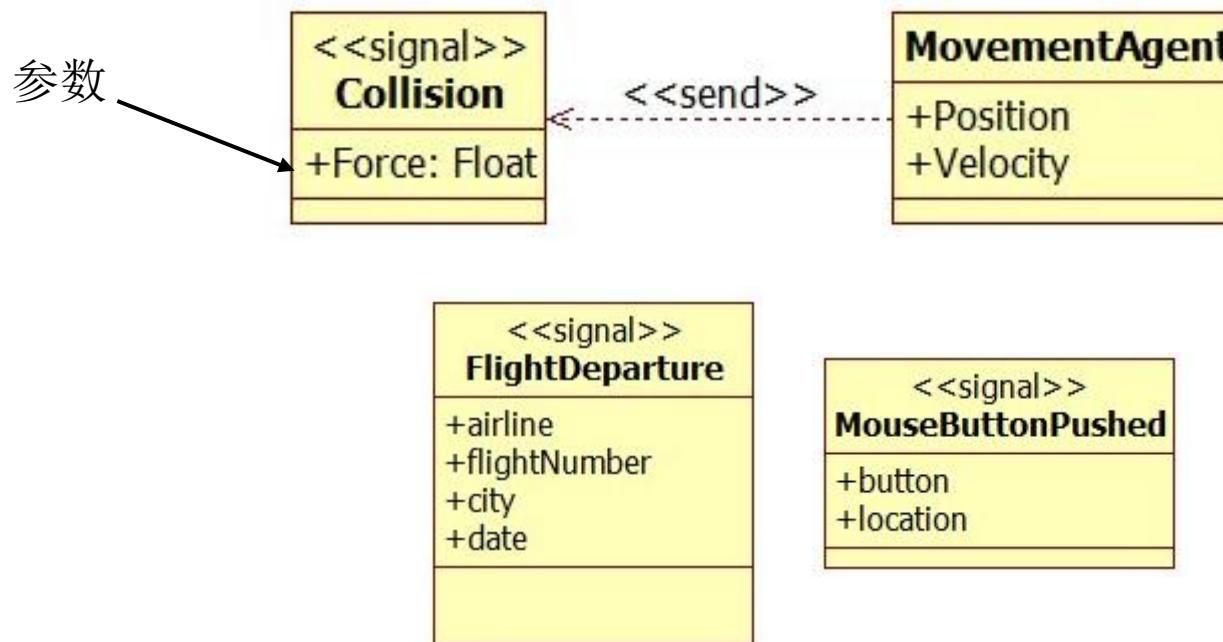
- 时间事件(Time event)：在绝对时间上或在某个时间间隔内发生的事情所引起的事件。
 - 绝对时间用关键字When表示，后面括号中为包含时间的表达式；
 - 时间段采用关键词After表示，后面括号中为计算时间间隔的表达式



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

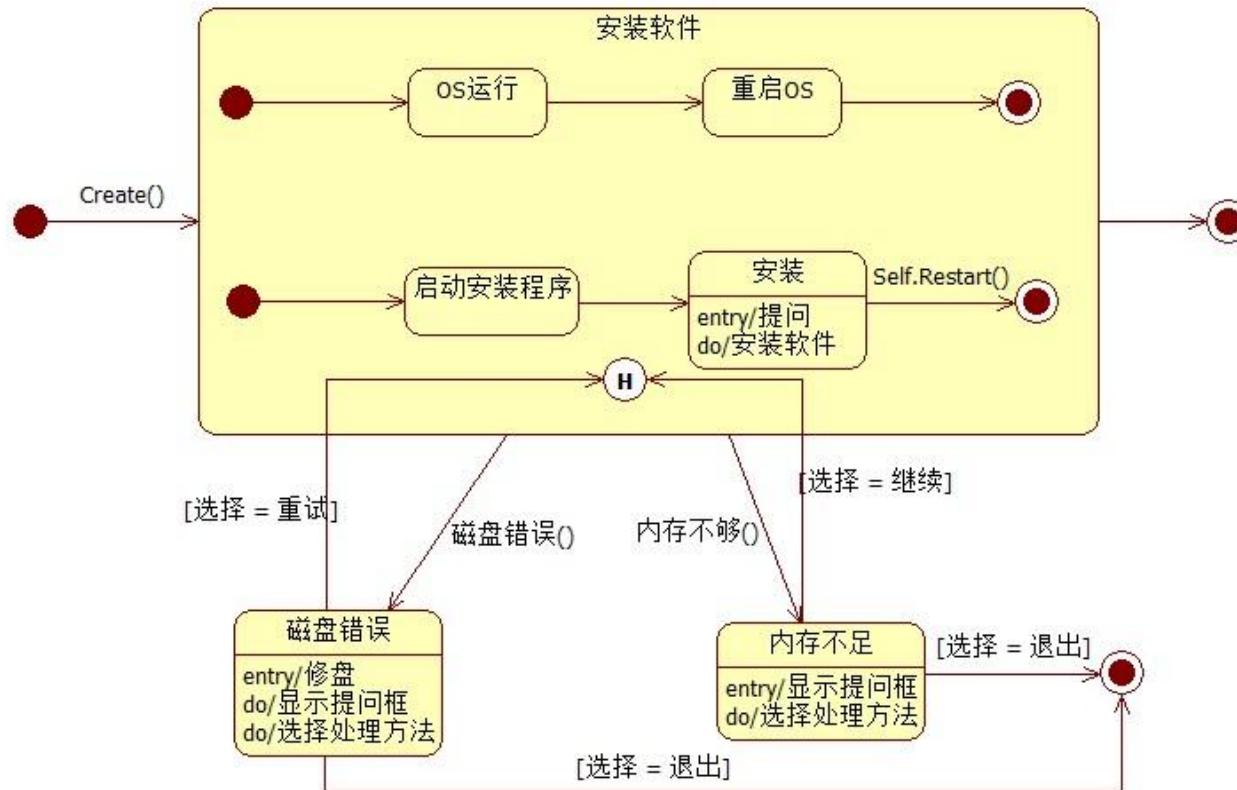


- 历史指示器被用来存储内部状态。
 - 当对象处于某一状态，经过一段时间后可能会返回到该状态，则可以用历史指示器来保存该状态。
 - 可以将历史指示器应用到状态区。如果到历史指示器的状态转移被激活，则对象恢复到在该区域内的原来的状态。
 - 历史指示器用空心圆中方一个“H”表示。可以有多个历史指示器的状态转移，但没有从历史指示器开始的状态转移。

状态图：历史指示器

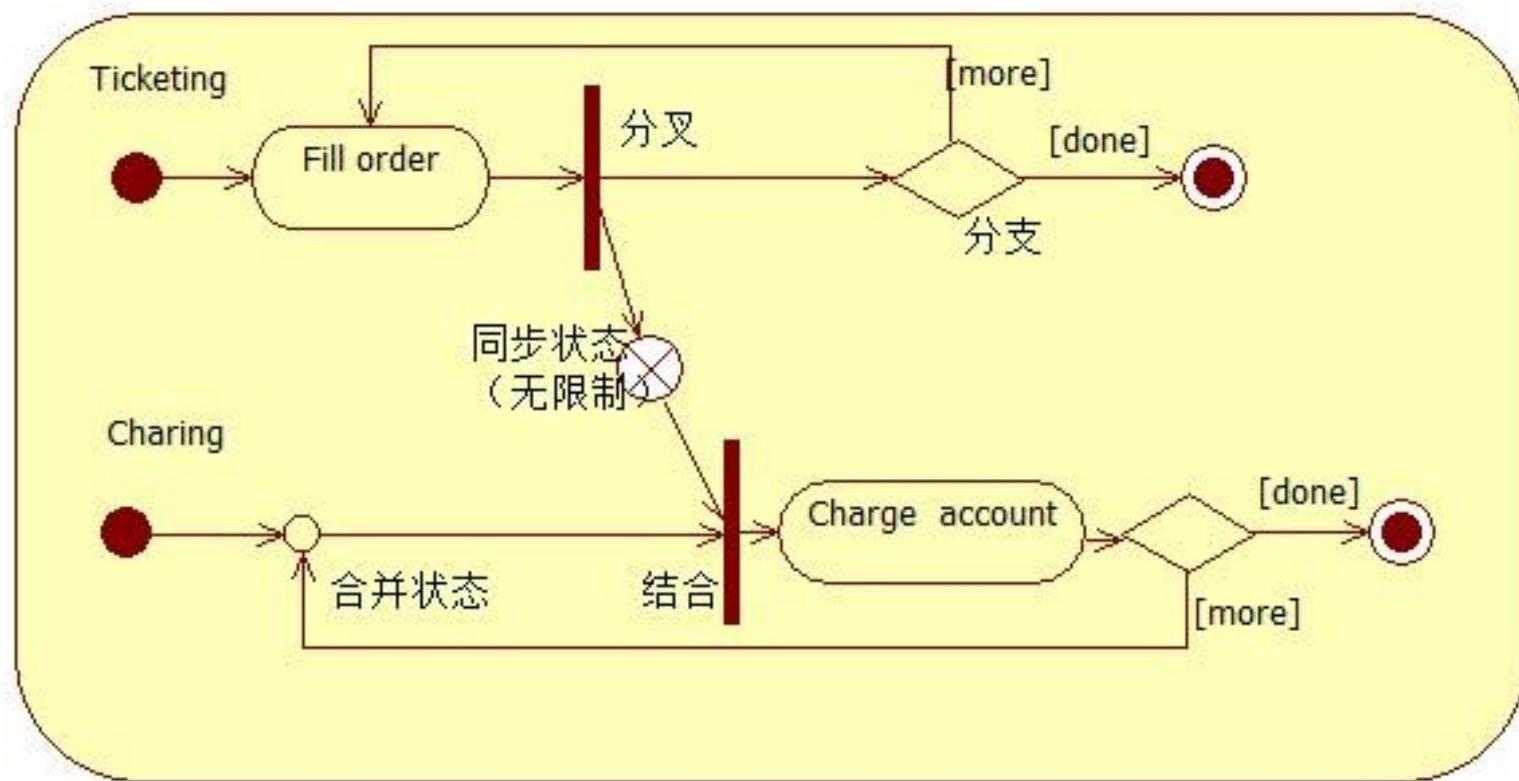
- 例：软件安装程序

- 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。



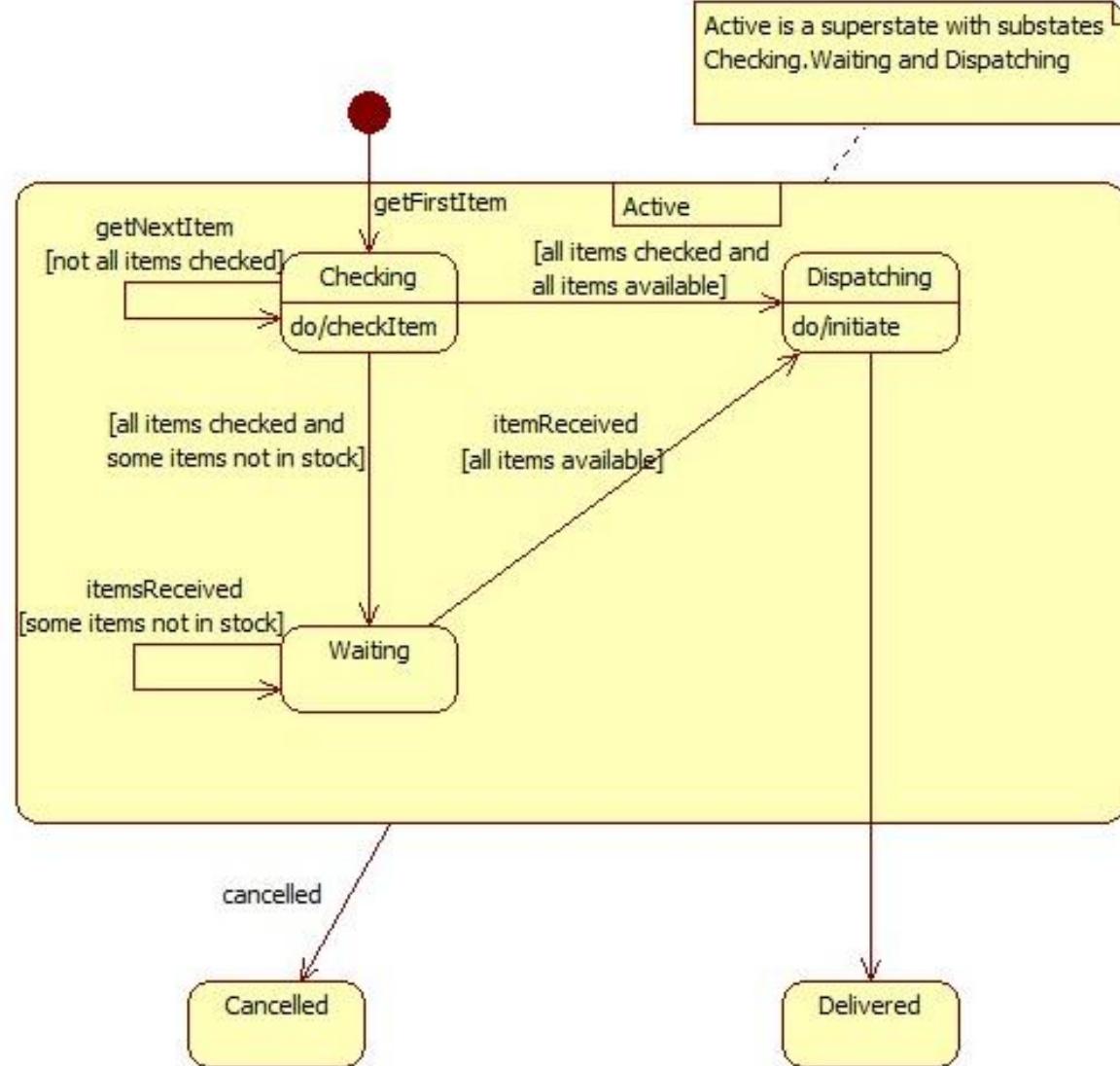
状态图：同步状态

- 同步状态



状态图 (Statechart Diagram)

组合状态

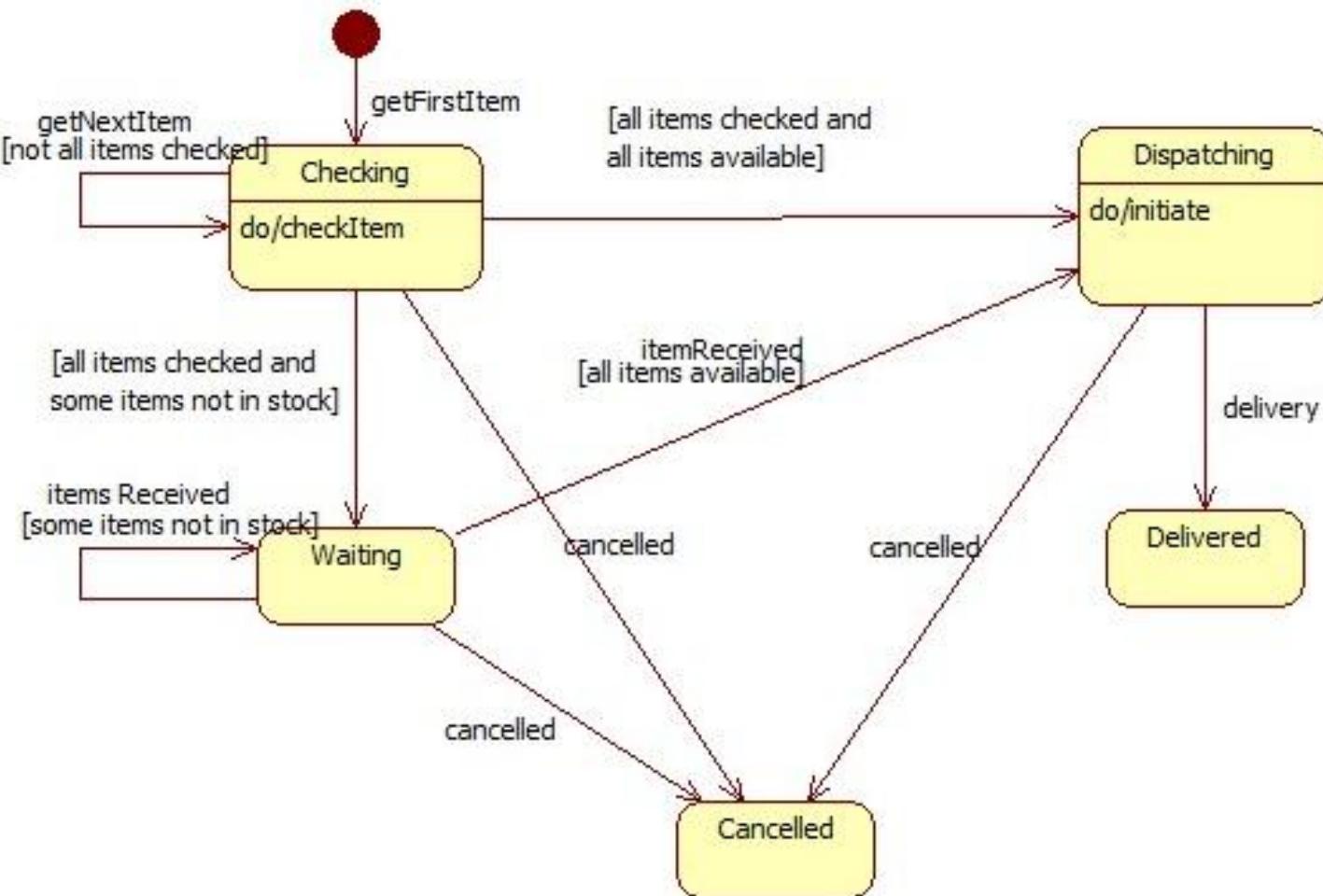


状态图绘制方法

- 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。

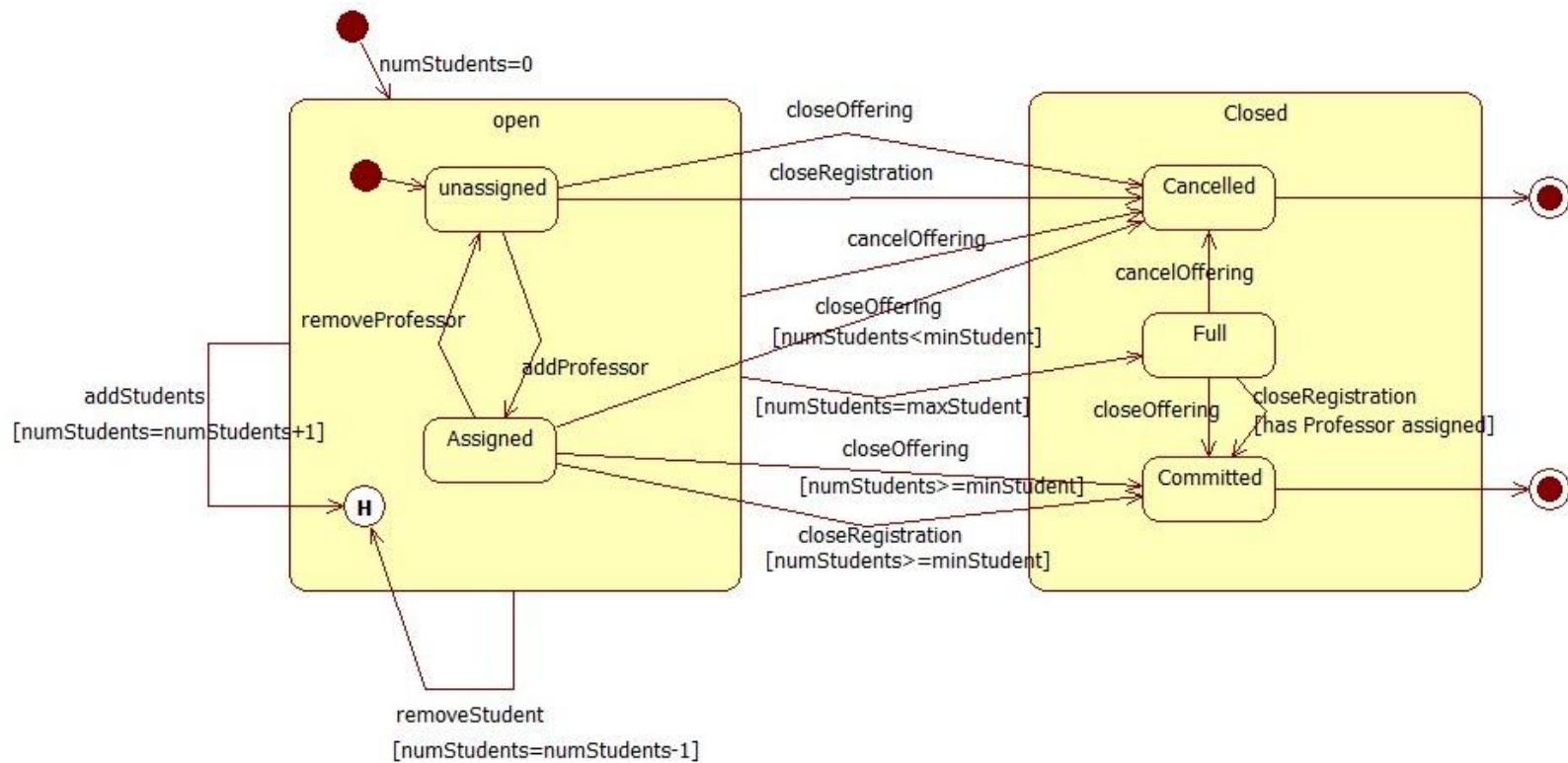
状态图 (Statechart Diagram)

- “订单”对象的状态图

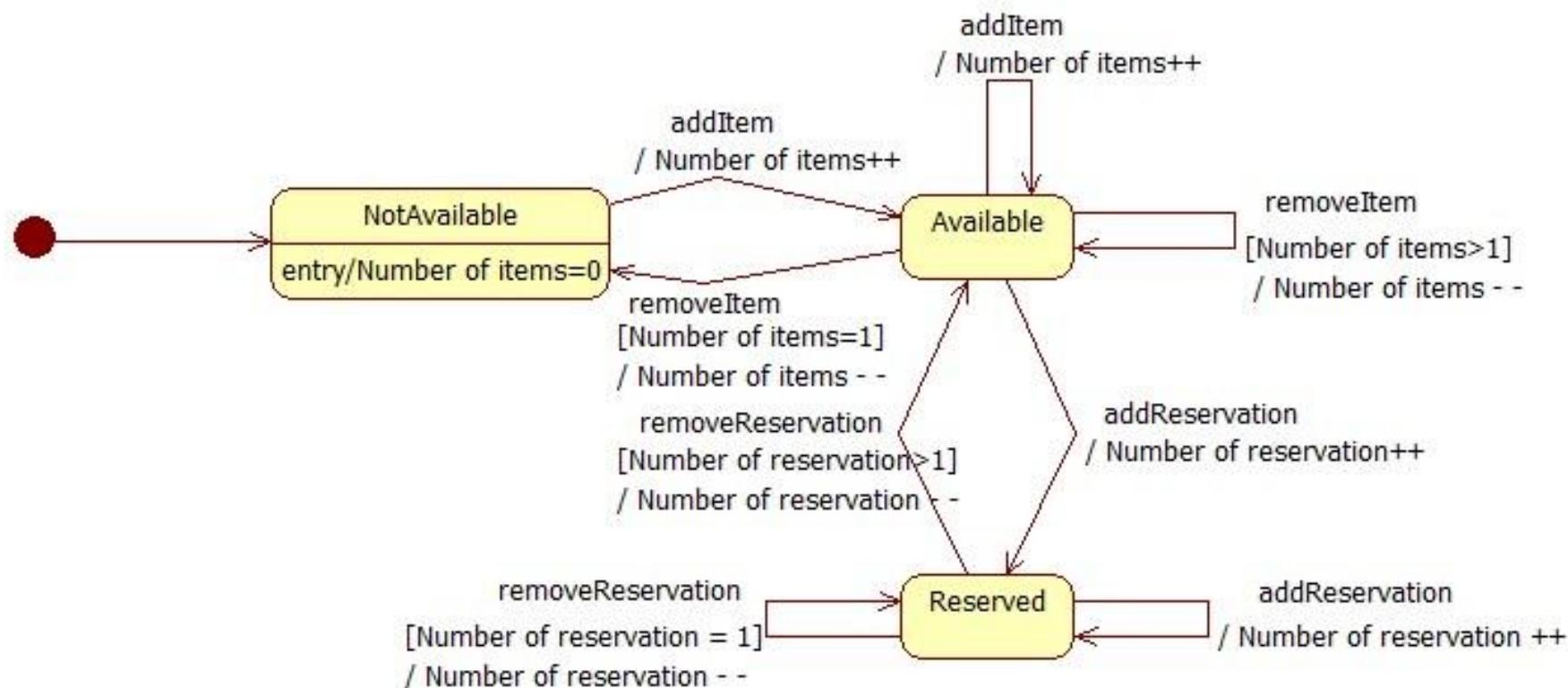


状态图 (Statechart Diagram)

- “选课”对象的状态图



状态图 (Statechart Diagram)



5. 细化类之间的关系

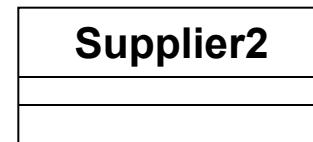
- 细化关系：关联关系、依赖关系、继承关系、组合和聚合关系
- “继承”关系很清楚；
- 在对象设计阶段，需要进一步确定详细的关联关系、依赖关系和组合/聚合关系等。

- 不同对象之间的可能连接：四种情况

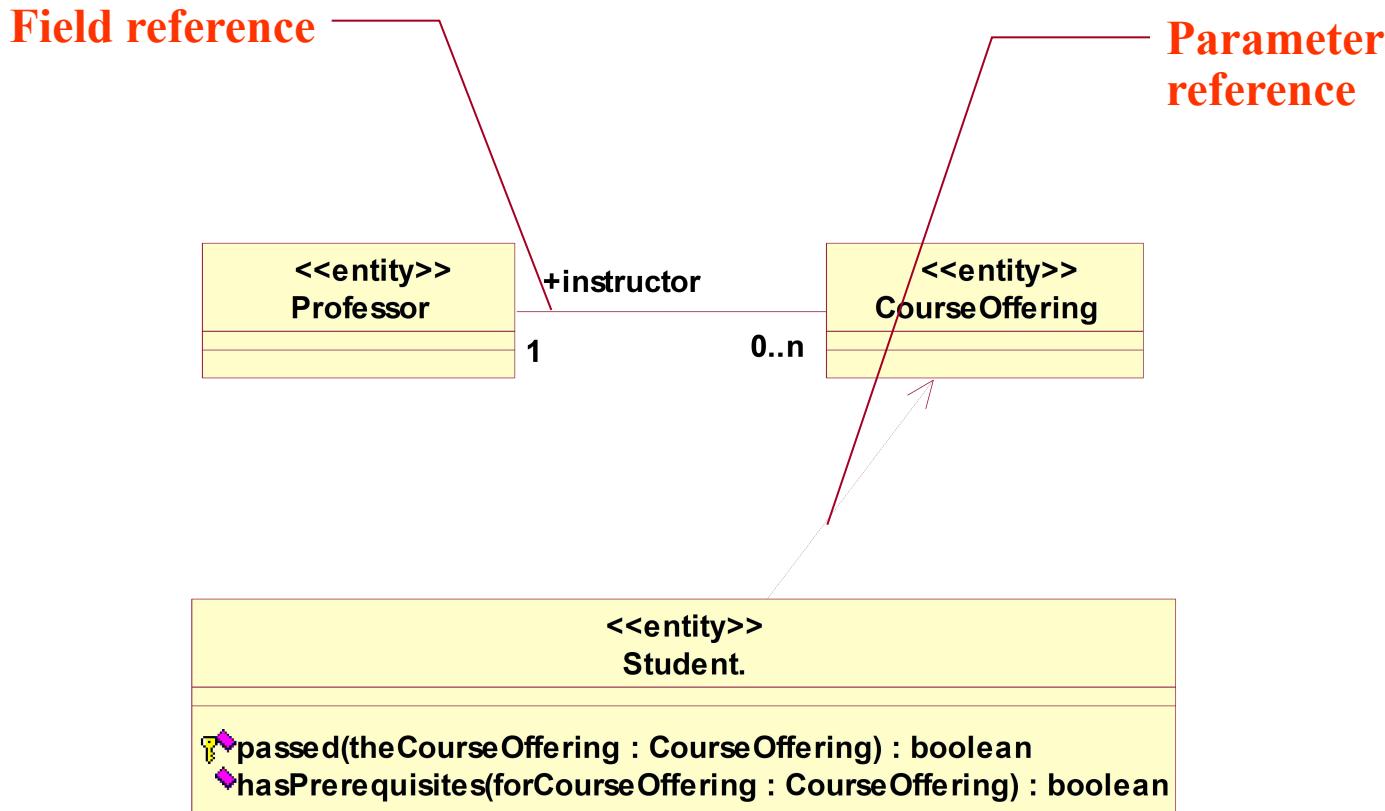
- 全局(*Global*)
- 参数(*Parameter*)
- 局部(*Local*)
- 域(*Field*)

Dependency

*Association/Aggregation/
Composition*

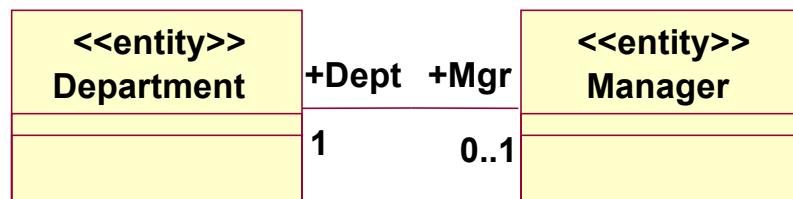


定义类之间的关系：示例



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

- 根据“多重性”进行设计(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。



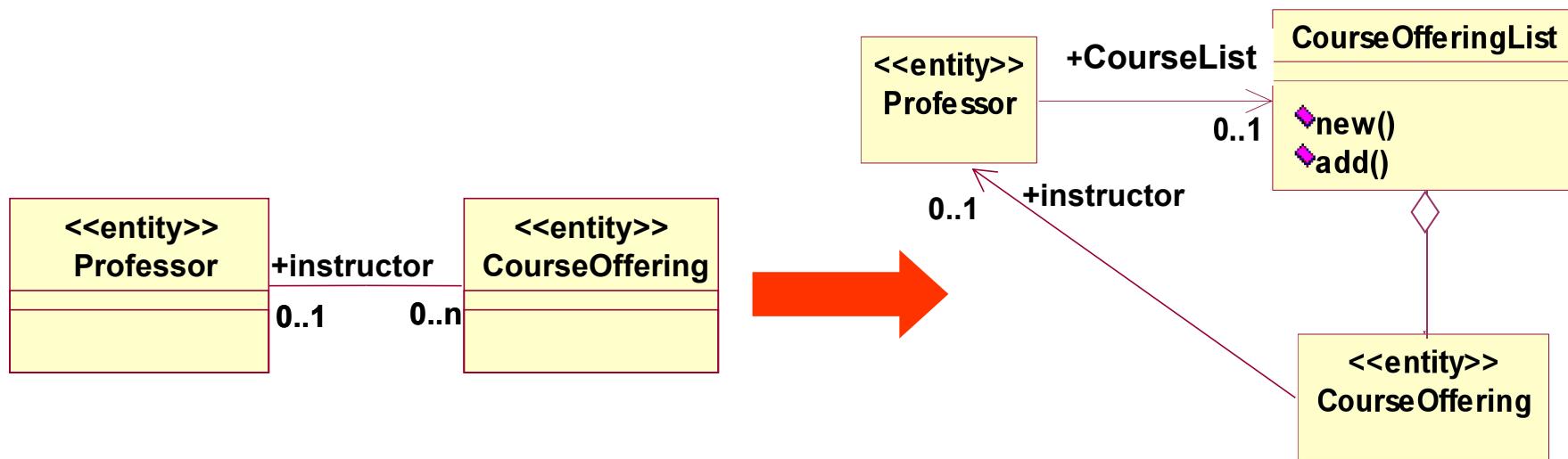
设计阶段需要将这种
“关联属性”增加到属性列表中，并更新操作列表

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

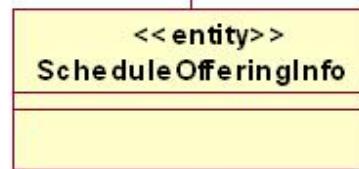
(Association/Composition/Aggregation)

- 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)

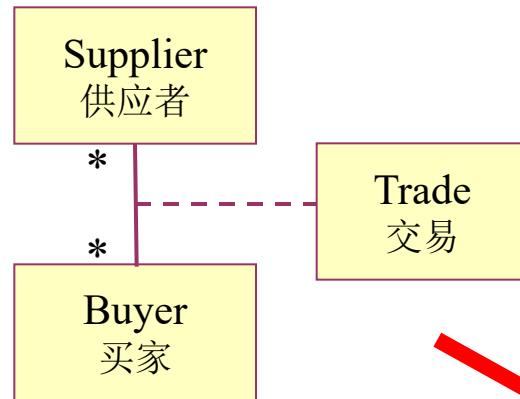
- 情况3：有些情况下，关联关系本身也可能具有属性，可以使用“关联类”将这种关系建模。
- 举例：选课表Schedule与开设课程CourseOffering



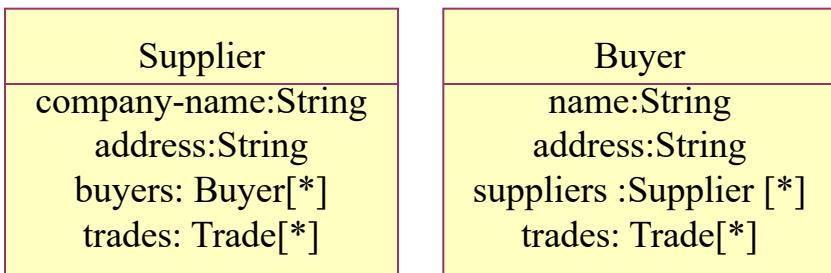
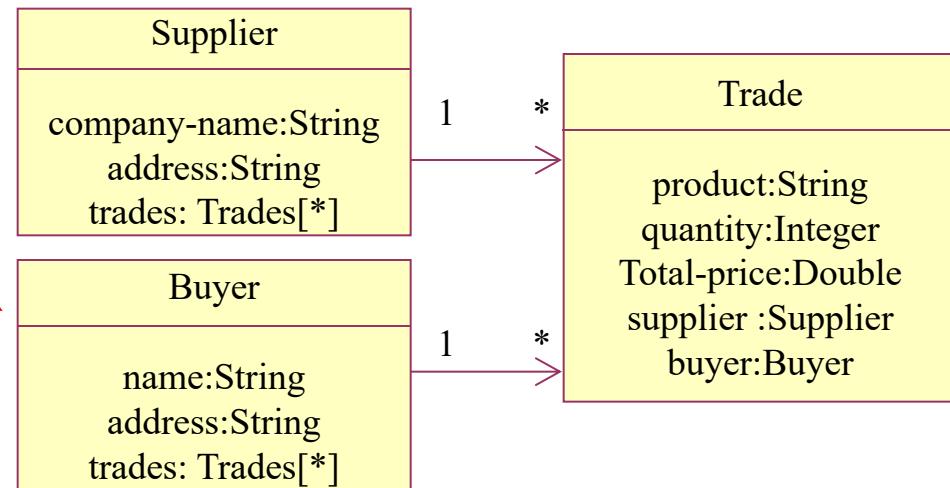
Design Decisions



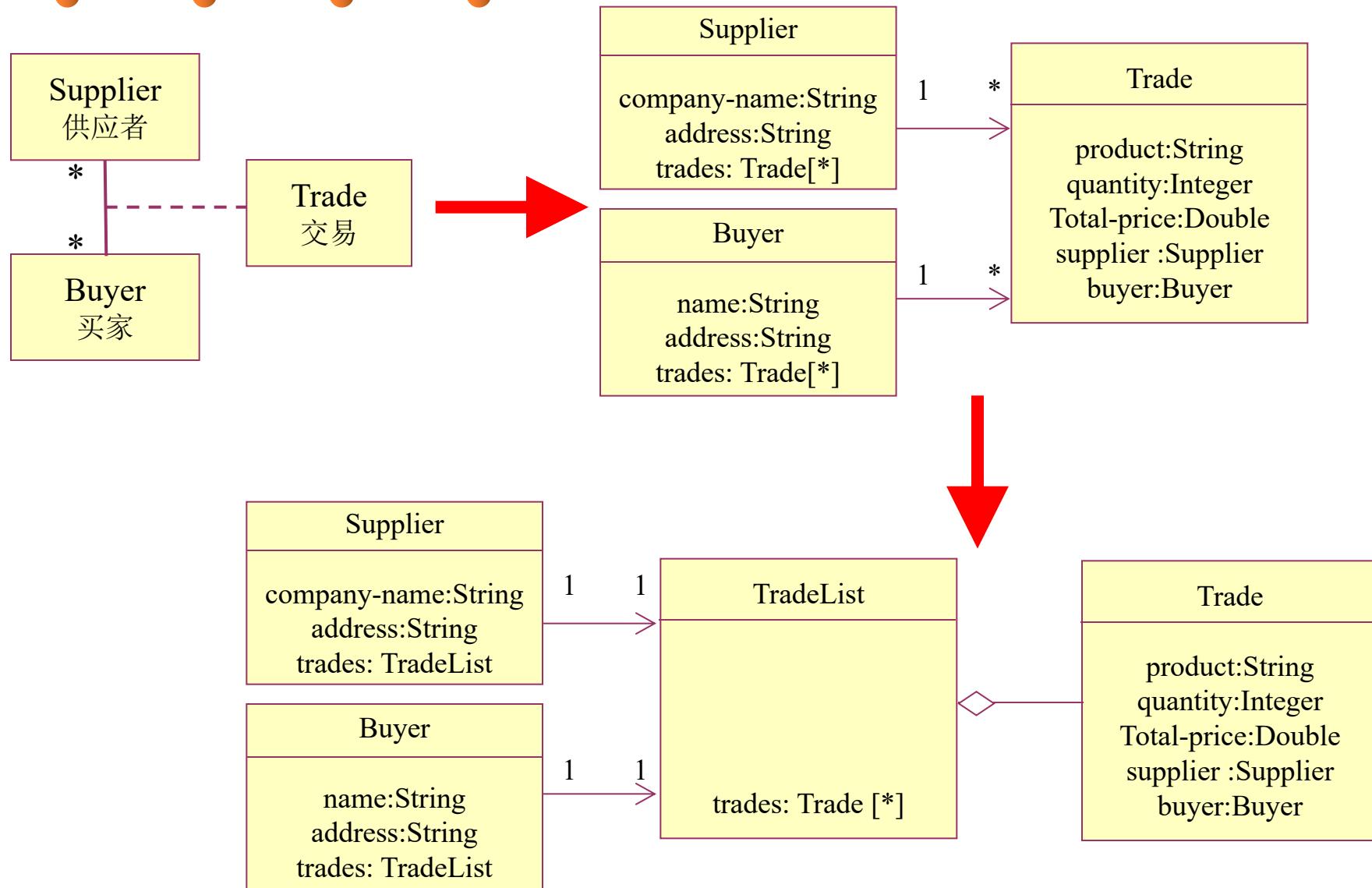
定义关联关系 (Association/Composition/Aggregation)



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)



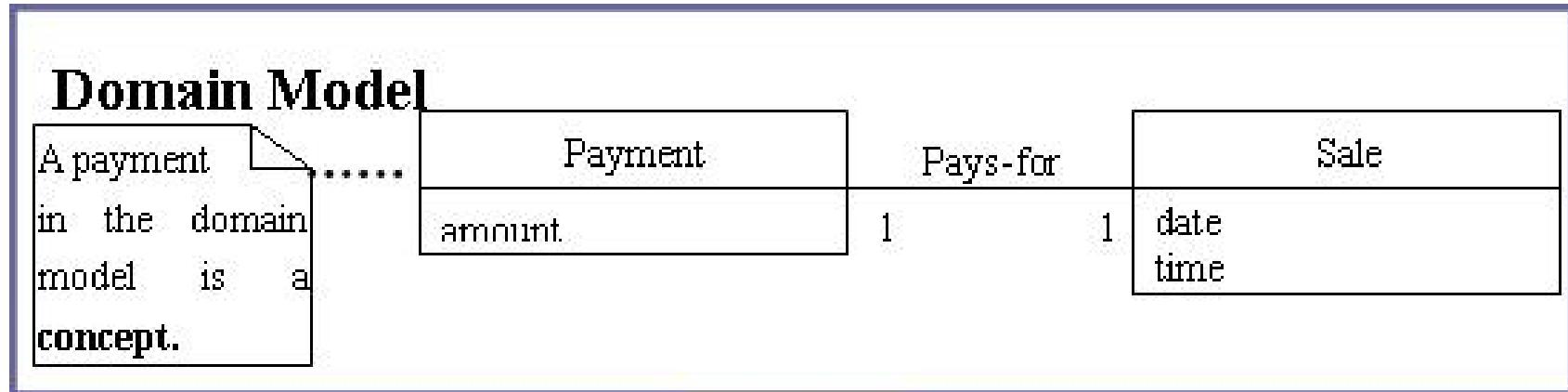
引入“辅助类”简化类的内部结构

- 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)。

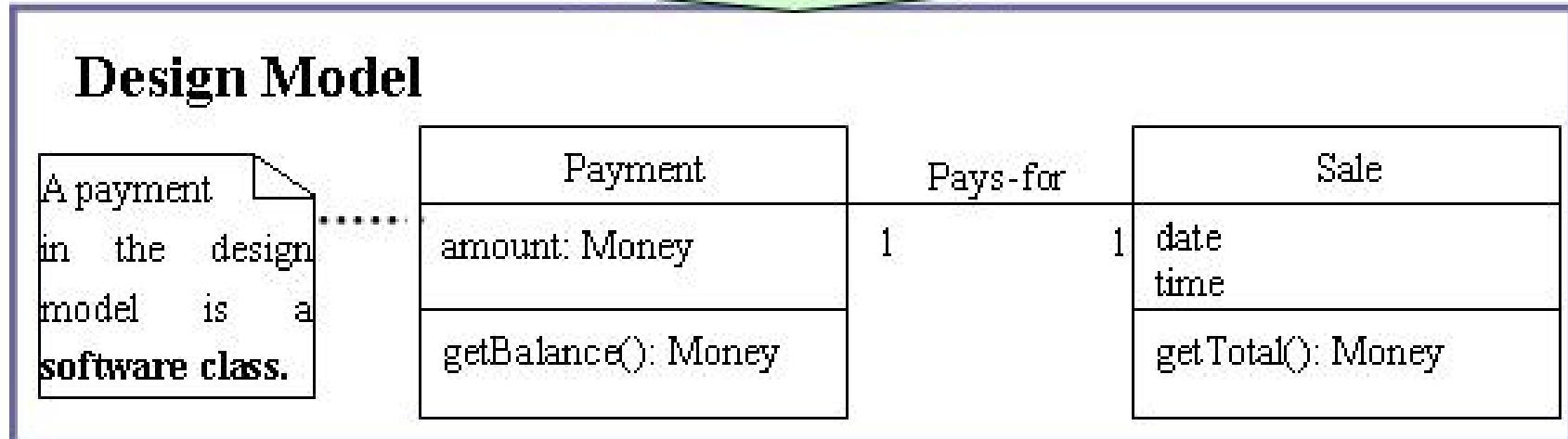
引入“辅助类”简化类的内部结构

- 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



模块化 / OOP/ 设计模式, etc!





7. 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



结束

2025年6月13日

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 (Object Modeling Technique)方法，由James Rumbaugh提出；
 - Booch方法，由Grady Booch提出；
 - OOSE方法，由Ivar Jacobson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。