




哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

软件工程
第六章 OO分析与设计
6-3 面向对象的设计与测试

王忠杰
rainy@hit.edu.cn

2017年11月6日

主要内容

- 
- 1 面向对象的设计
 - 2 包的设计
 - 3 数据库设计
 - 4 面向对象的测试
 - 5 OO分析、设计与测试小结

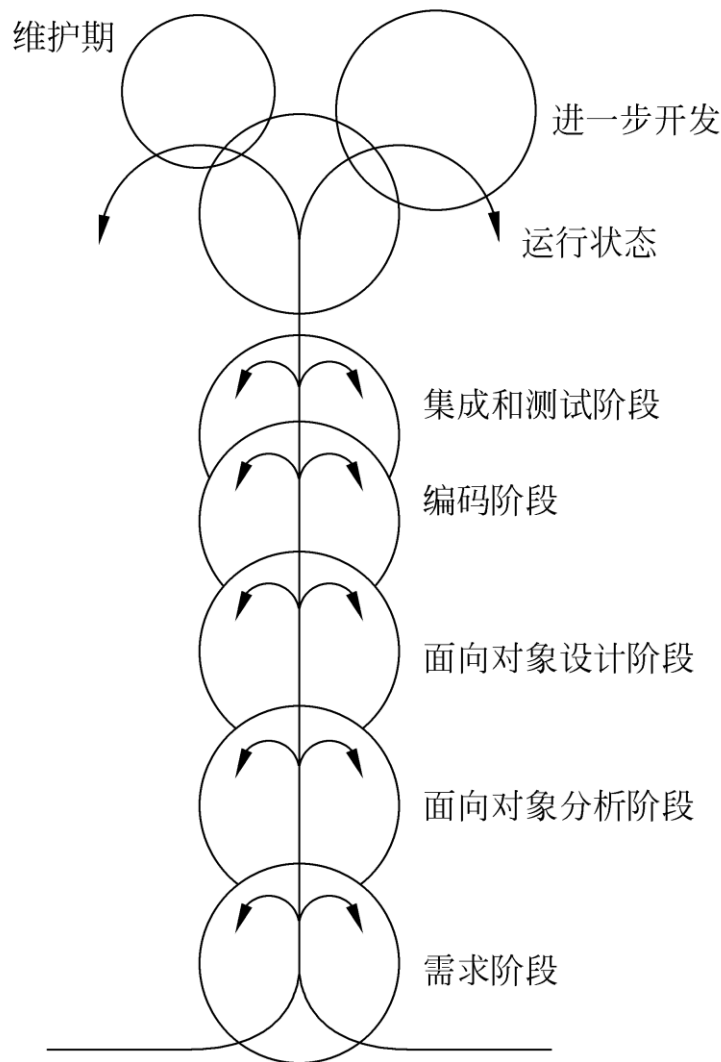


1 面向对象的设计



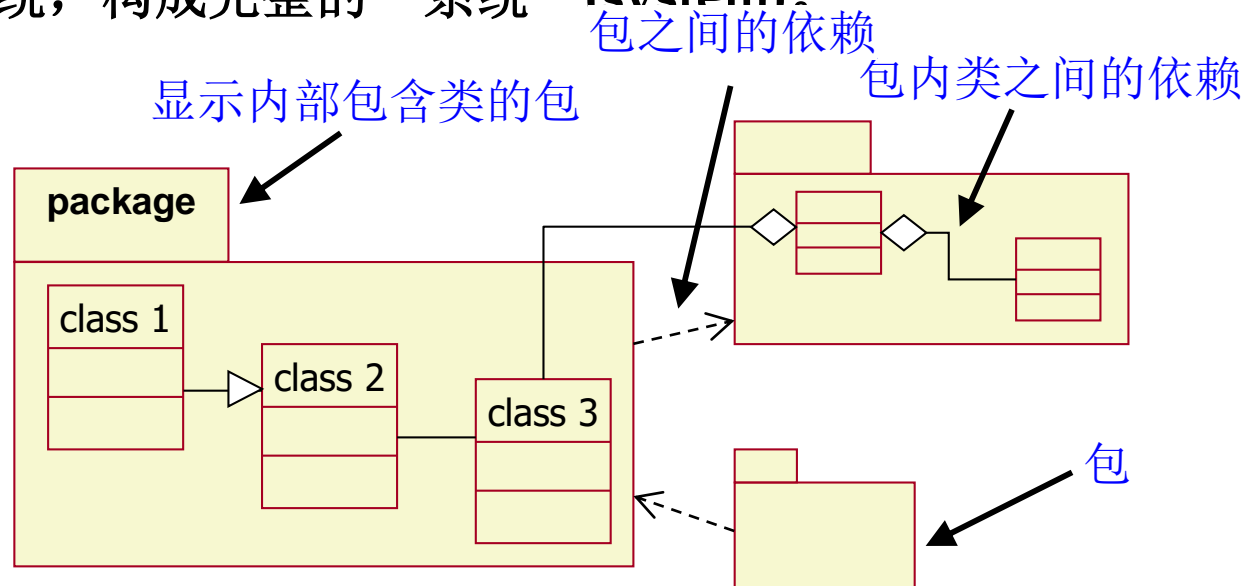
面向对象的设计

- 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。

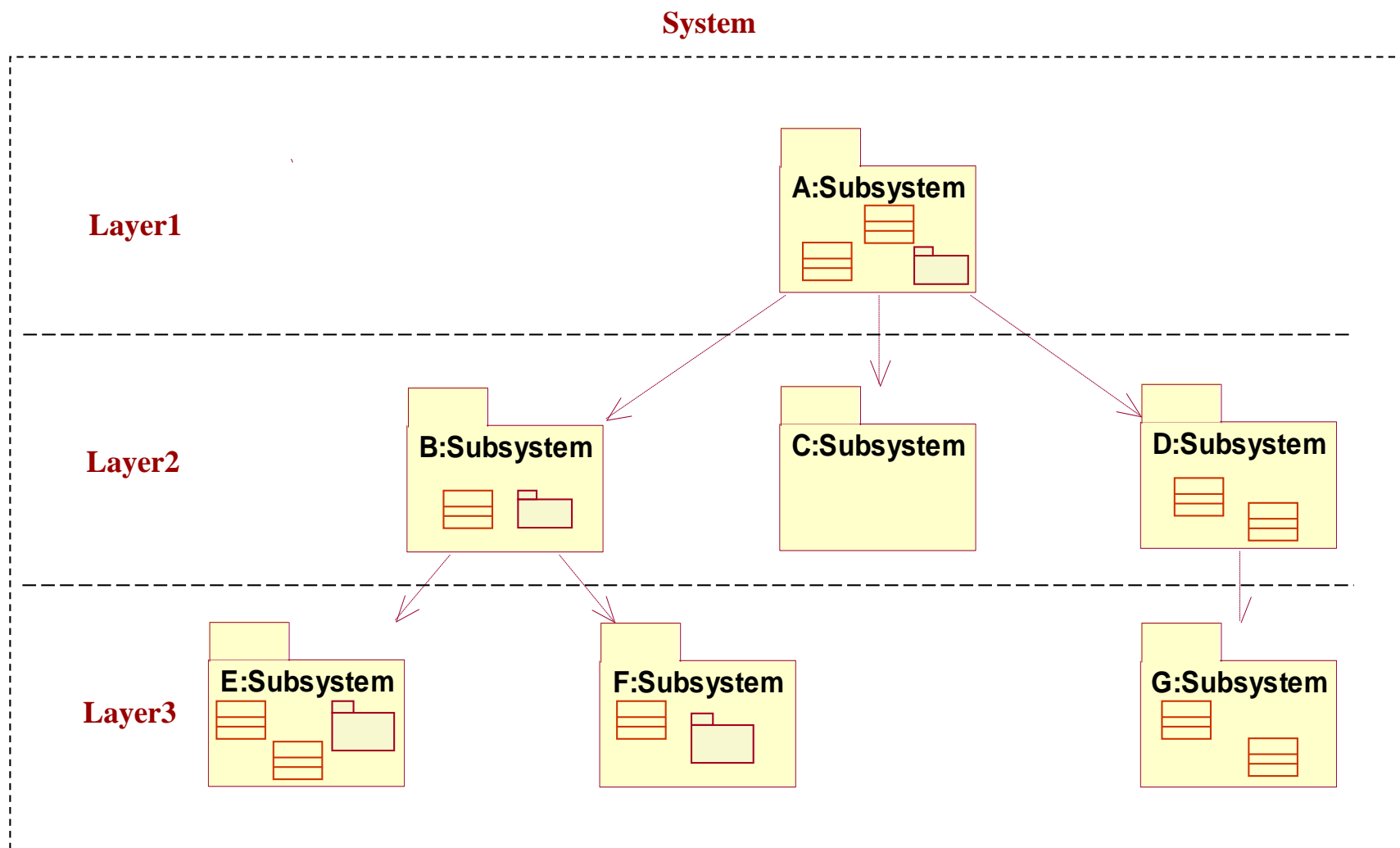


面向对象设计中的基本元素

- 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)；
- 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)。



面向对象设计中的基本元素



面向对象的设计的两个阶段

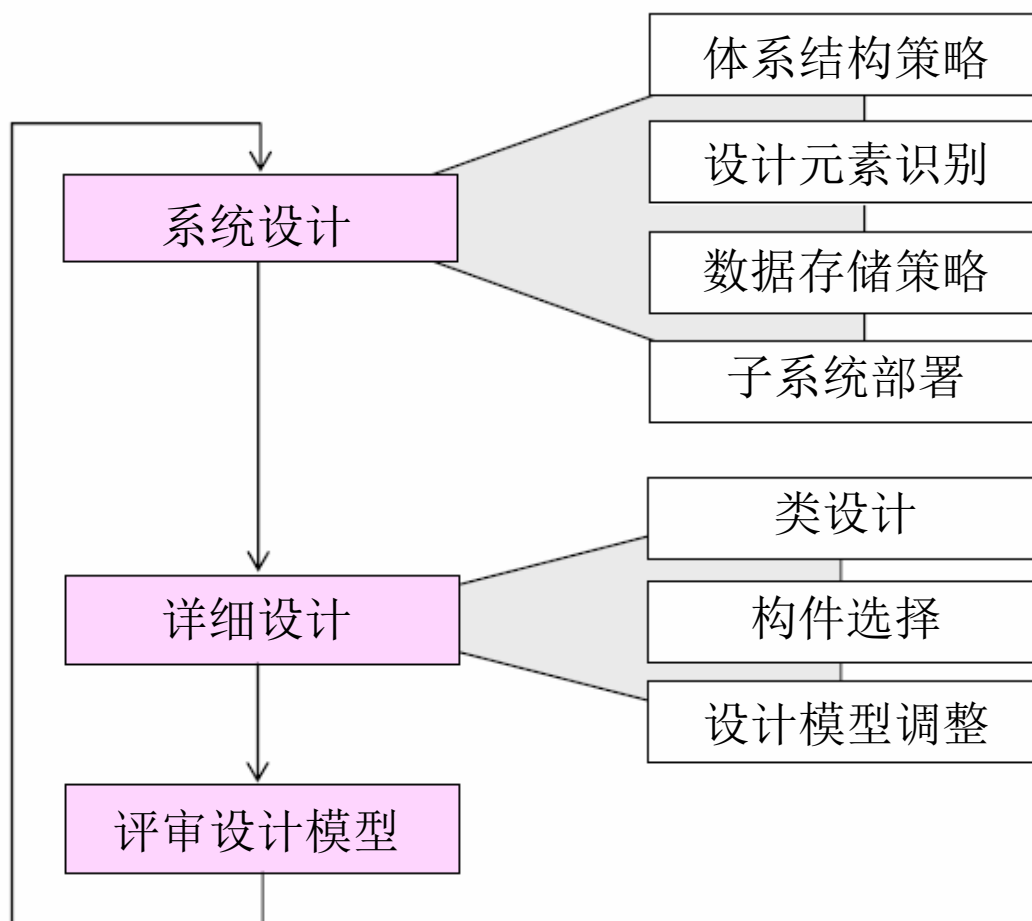
■ 系统设计(System Design)

- 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);
- 细化需求分析模型和系统体系结构设计模型;
- 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



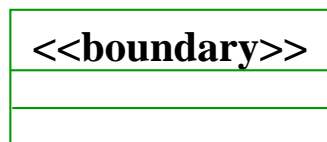
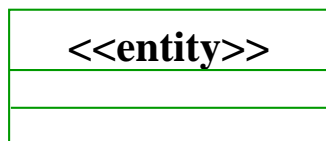
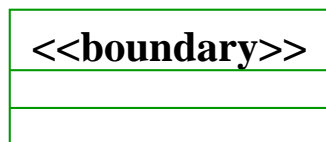


2 包的设计：从逻辑角度

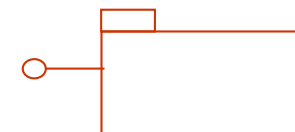
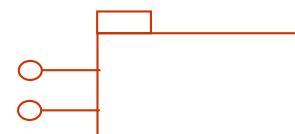
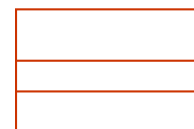


识别设计元素

分析类



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

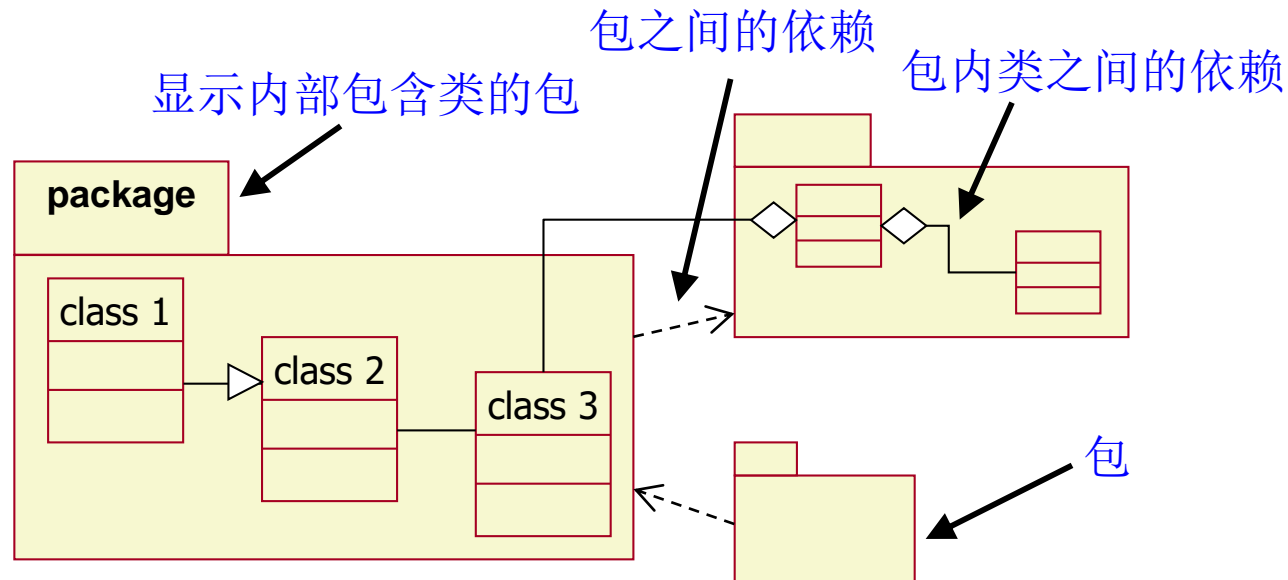
类型	分析类	设计元素
	<i>LoginForm</i>	“设计类” <i>LoginForm</i>
	<i>BrowseForm</i>	“设计类” <i>BrowseForm</i>

	<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>

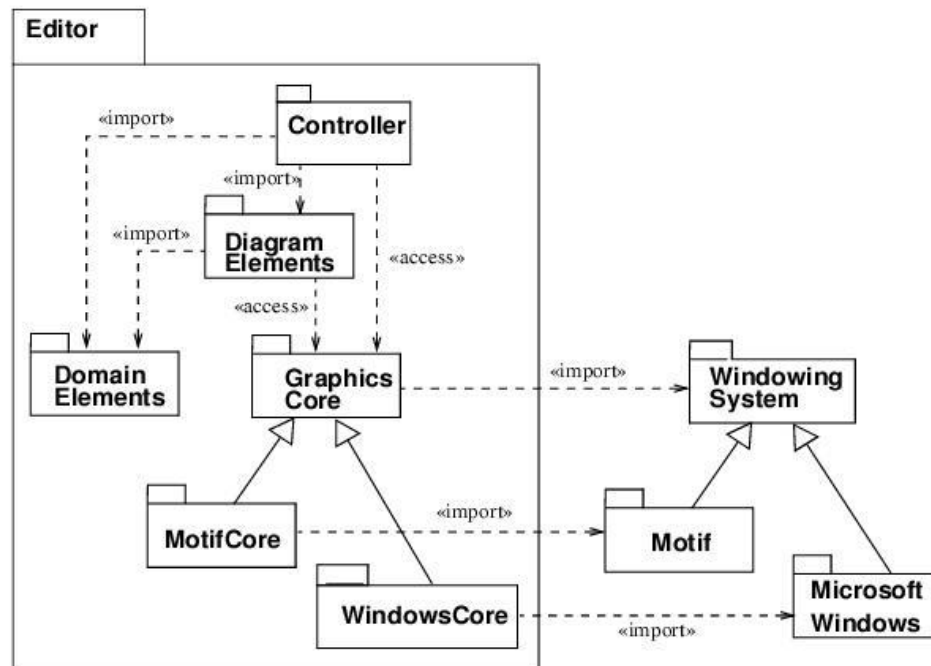
绘制包图(package diagram)

- 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

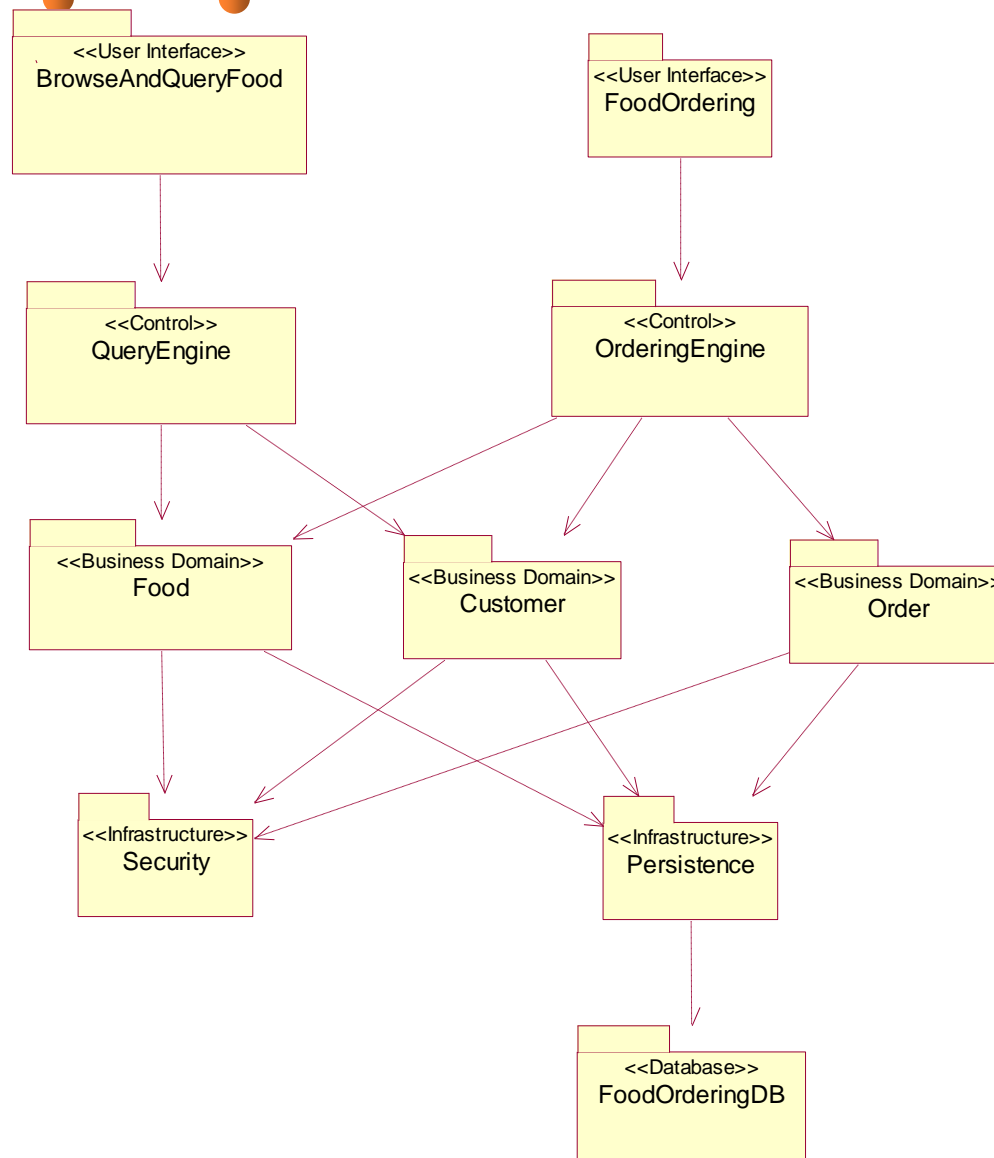
- 类与类之间的存在的“聚合、组合、关联、依赖”关系导致包与包之间存在依赖关系，即“包的依赖” (dependency);
- 类与类之间的存在的“继承”关系导致包与包之间存在继承关系，即“包的泛化” (generalization);



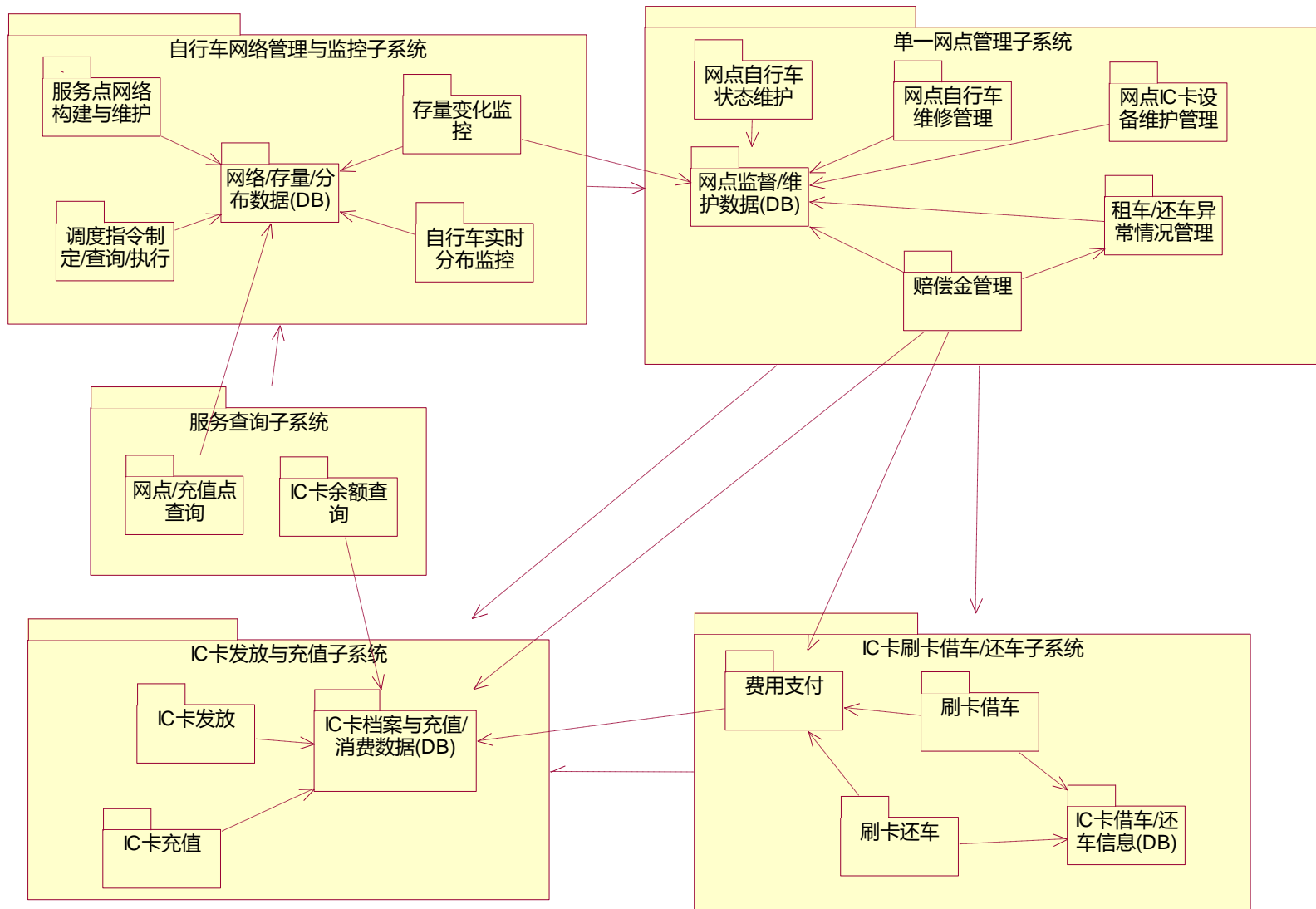
绘制包图(package diagram)的方法

- 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(package diagram): 示例



包图(package diagram): 示例



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;

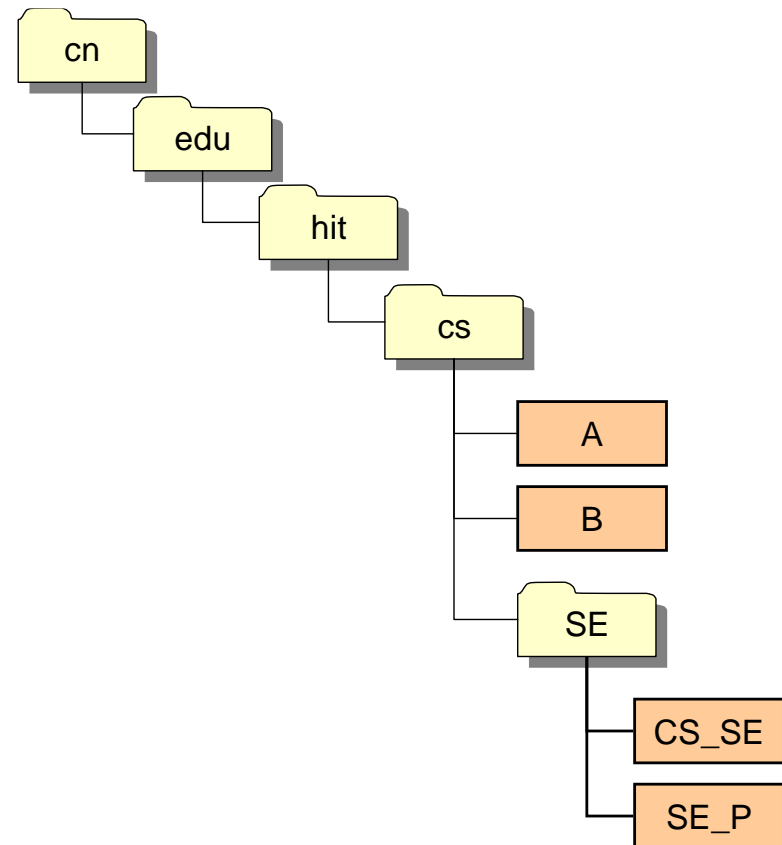
public class B{

package cn.edu.hit.cs.se;

public class CS_SE{

package cn.edu.hit.cs.se;

public class CS_SE_P{





3 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？

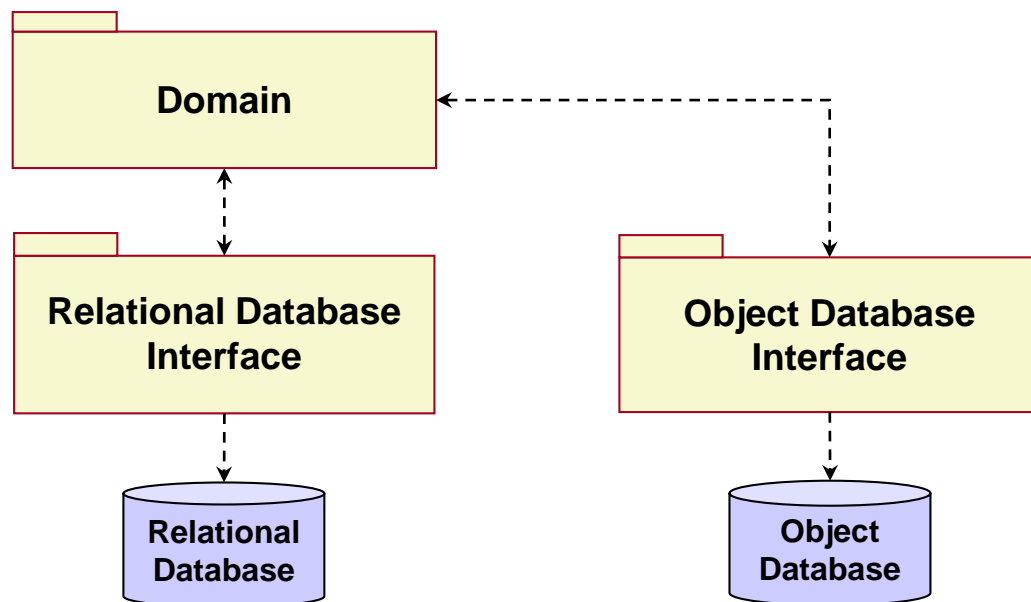
- 复杂的数据查询
- 数据集规模大

- 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；
- 如果使用非OO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

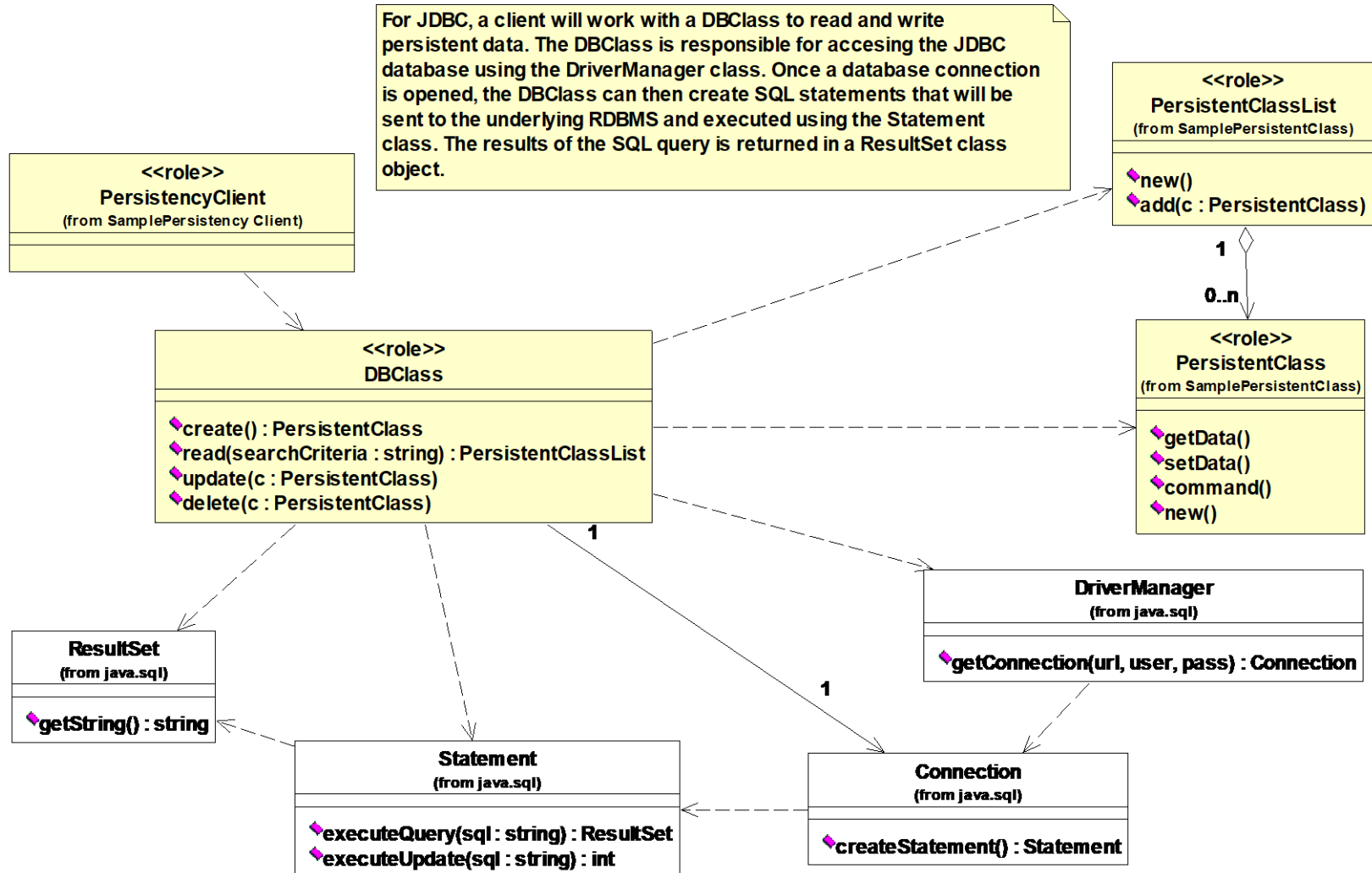
- 核心问题：对那些需要永久性存储的数据，如何将UML类图映射为数据库模型。
- 本质：把每一个类、类之间的关系分别映射到一张表或多张表；
- UML class diagram → Rational DataBase (RDB)
- 两个方面：
 - 将类(class)映射到表(table)
 - 将关联关系(association)映射到表(table)

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。

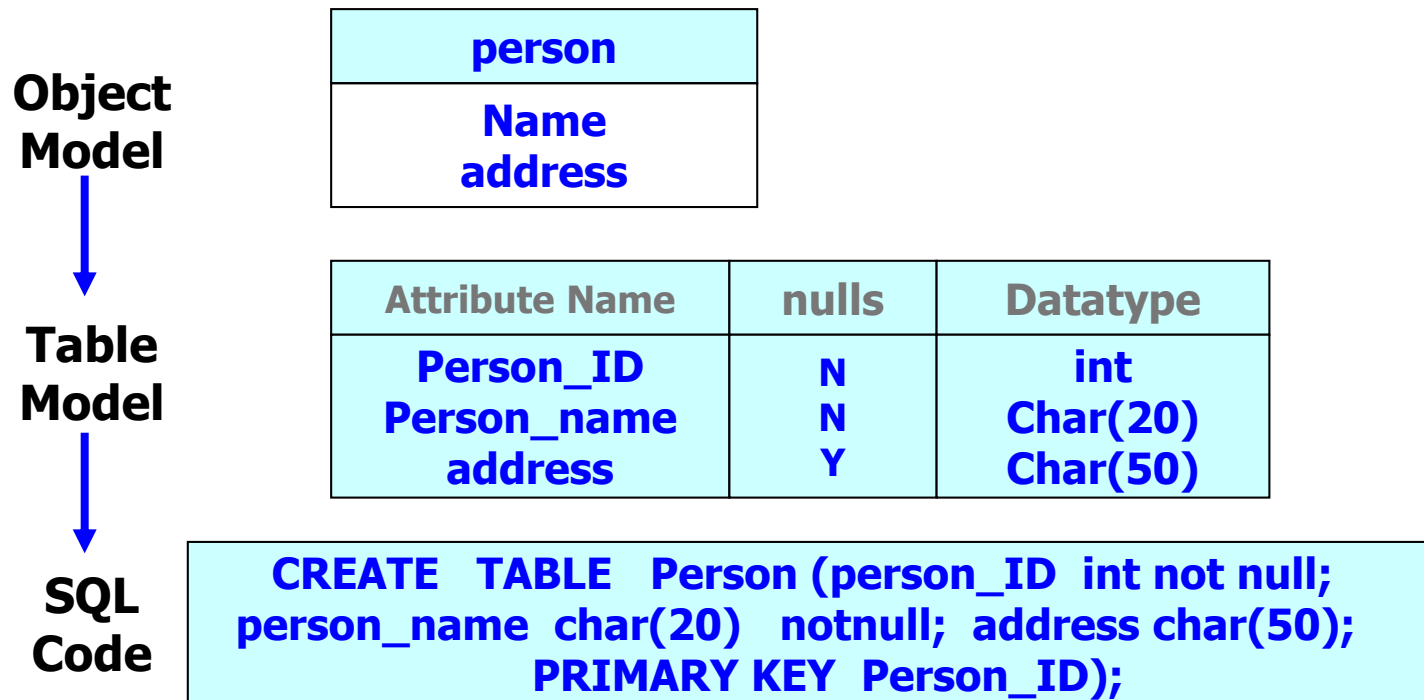
- **ORM产品:**
 - Apache OJB
 - Hibernate
 - Oracle TopLink
 - ...

Example: Persistency:RDBMS:JDBC



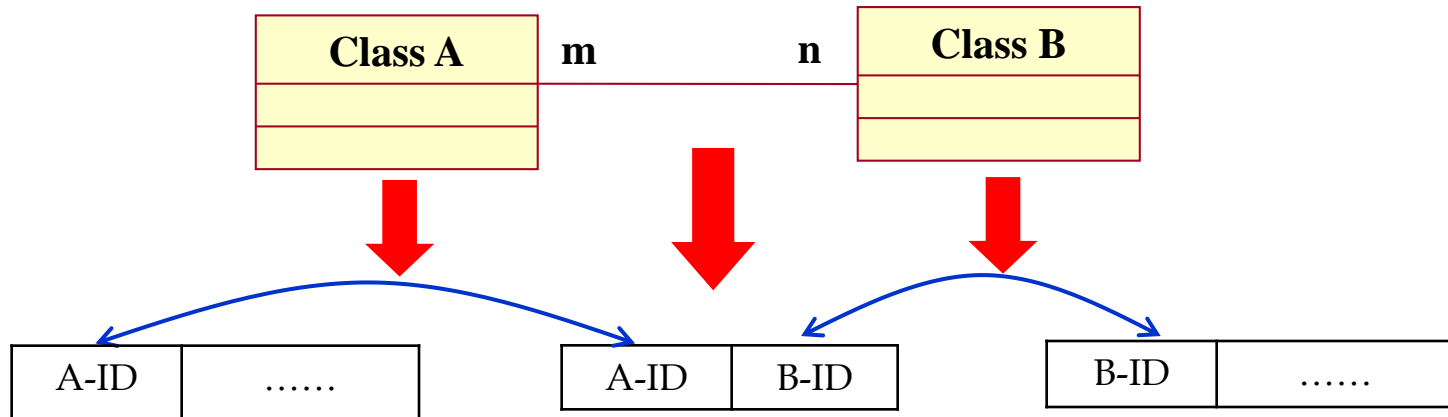
将对象映射到关系数据库

- 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)。



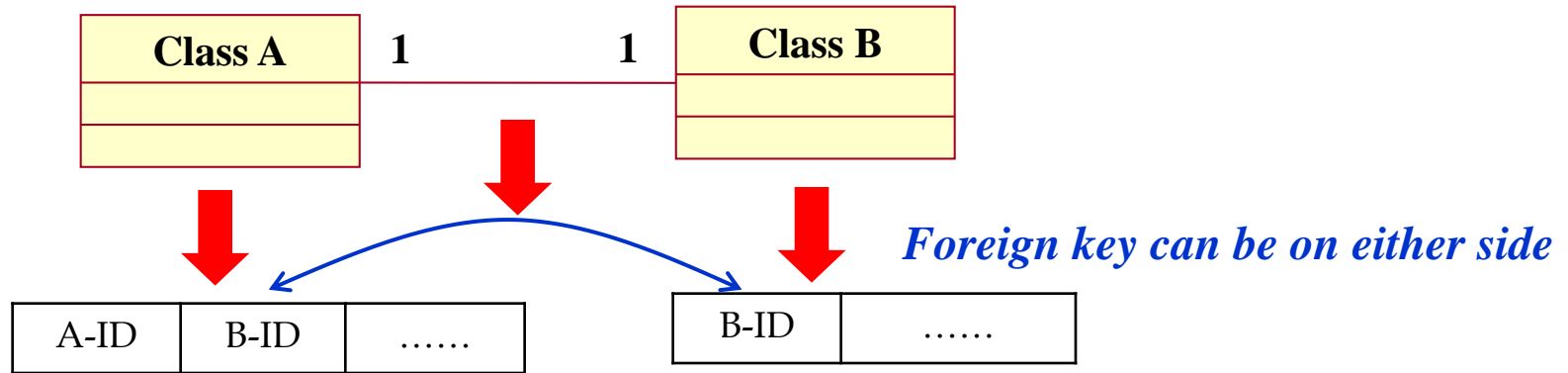
将关联映射到关系数据库(统一的、简单的途径)

- 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；

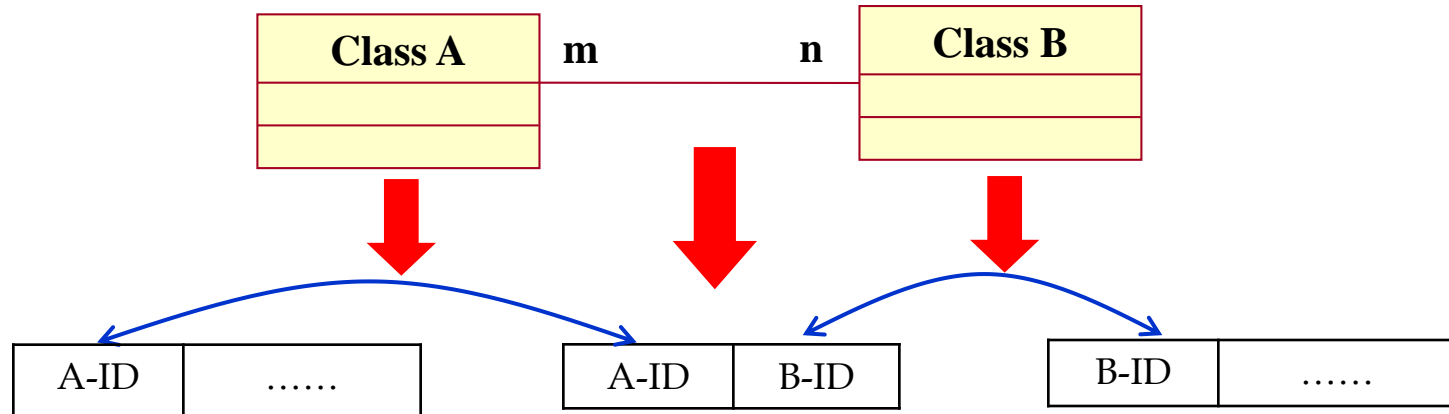


将关联映射到关系数据库(1:1和m:n的关联关系)

Implementing 1 to 1

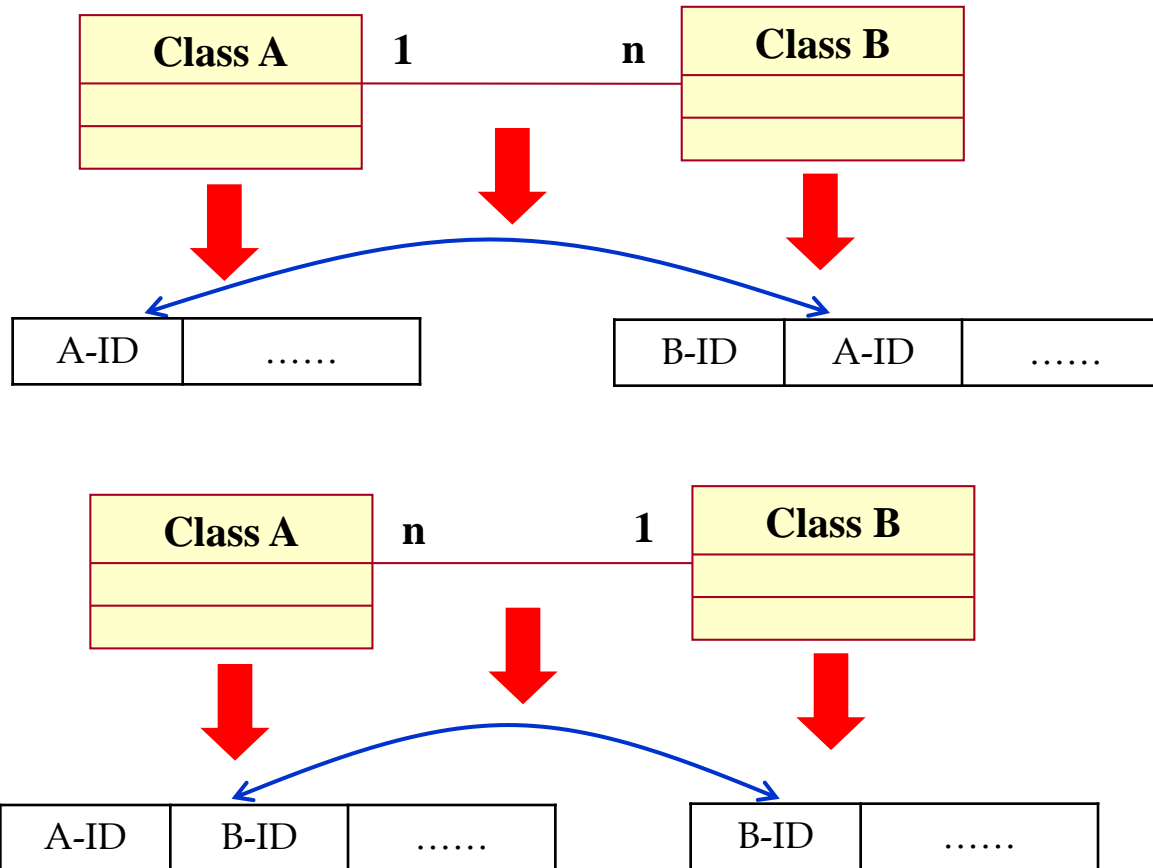


Implementing m:n

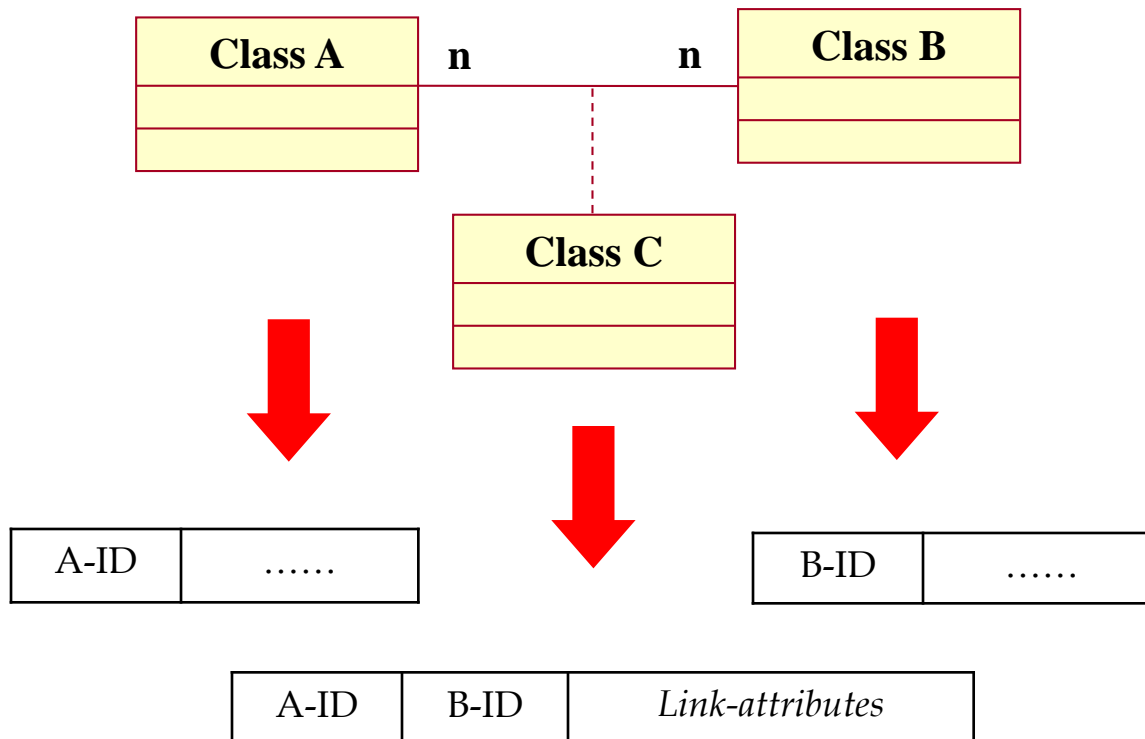


将关联映射到关系数据库(1:n的关联关系)

■ 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)

Company_Table

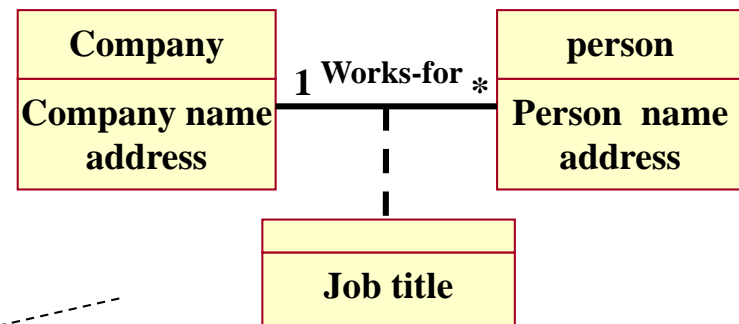
Attribute Name	nulls	Domain
Company_name	N	Char
address	N	Char

Person_Table

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

Job_Table

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



Company_Table

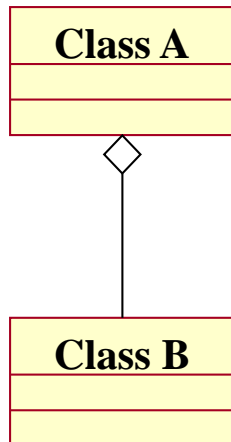
Attribute Name	nulls	Domain
Company_name	N	Char
address	N	Char

Person_Table 2

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

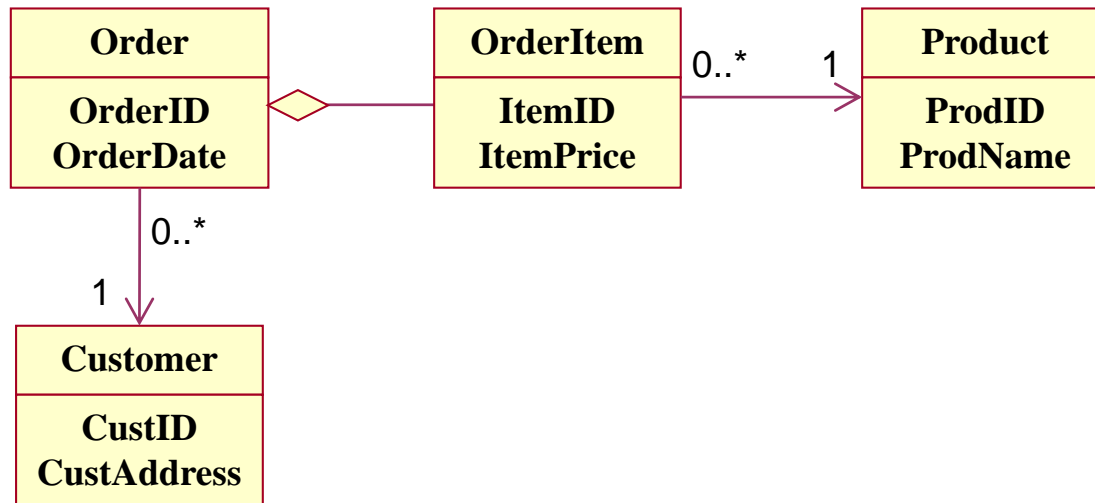
A-ID	X	Y
:	:	:

“部分”表

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

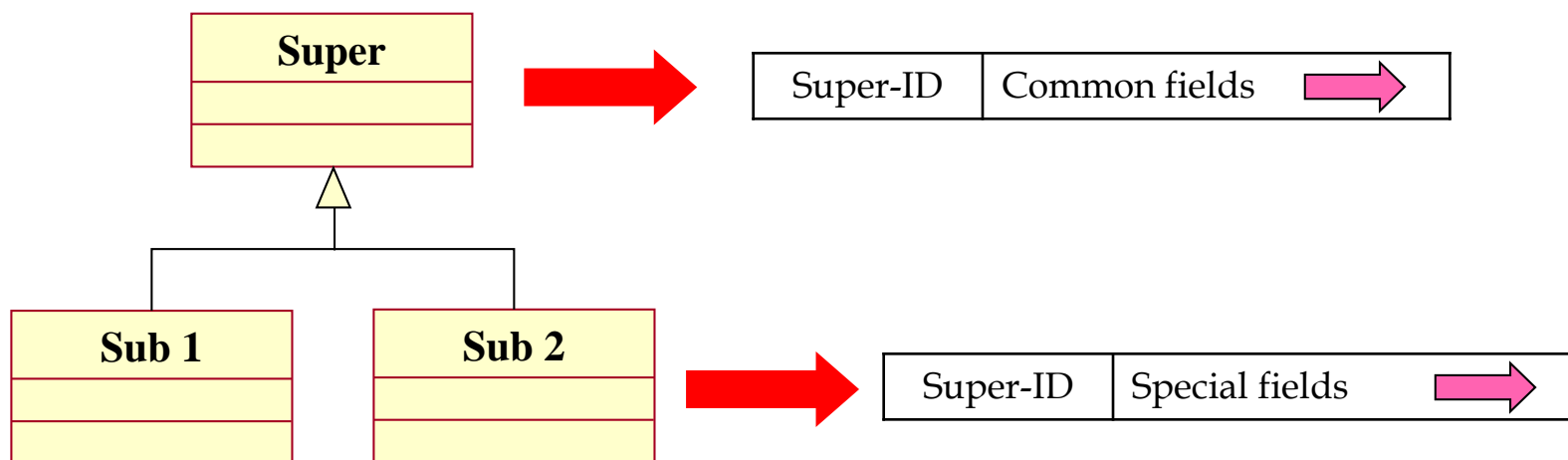
OO到DB的映射

- 为以下类图设计关系数据表



将继承关系映射到关系数据库

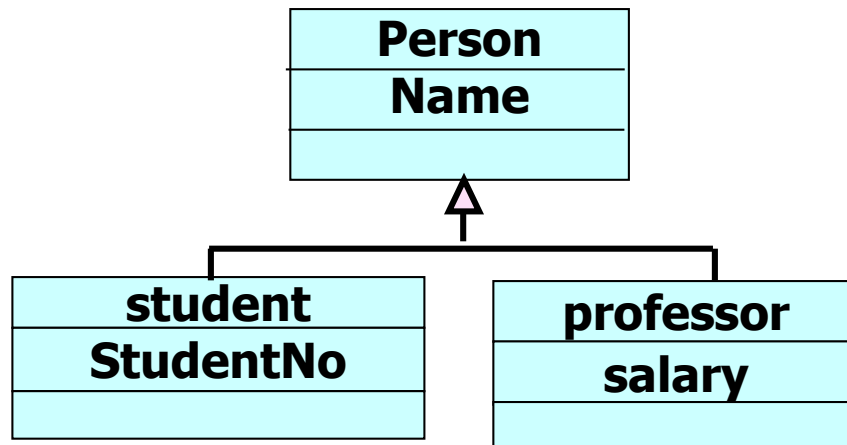
- 策略1：分别建立父类和子类的三张数据表



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

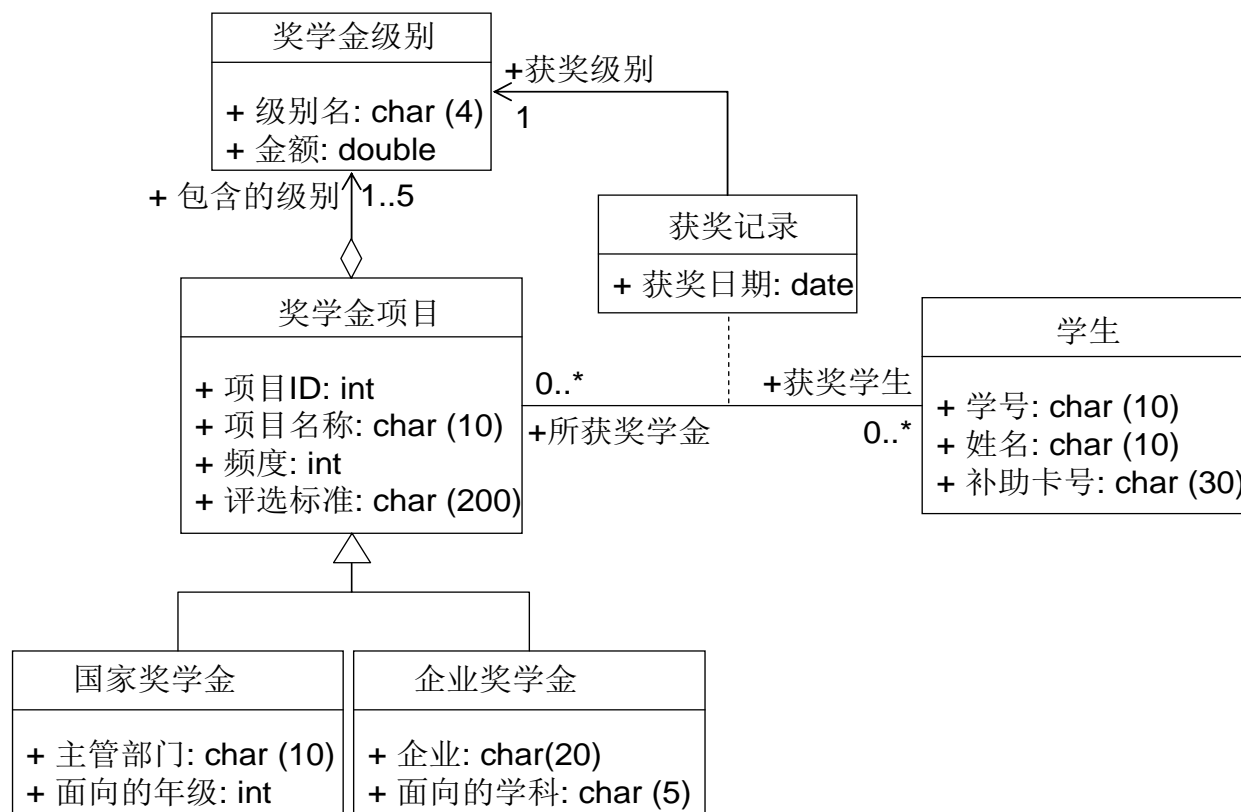
?

分别建立
父类和子类的
三张数据表

?

课堂讨论: OO→DB

- 下图是哈工大学生奖学金发放系统的局部领域类图，其中所有实体类均需要持久化存储，且数据被存储于关系型数据库中。
- 为该系统设计关系数据模式，针对每一张表，给出表名、属性名、数据类型、键标识。



课堂讨论：数据模式和OO谁先行

- 在设计软件系统时，很多学生习惯了先把所需要存储的关系数据模式设计出来，再反过来考虑逻辑层的OO设计。
- 还有人主张，应该先从顶层的OO设计开始，然后指导底层的数据库或文件设计。
- 关系数据模式和OO，在进行软件设计过程中应该先做哪个？
- 你认为这么做的优点是什么？



4 面向对象的测试



传统软件的测试 vs. OO测试

- 传统的测试：

- 单元测试→集成测试→系统测试；
- 从小到大逐步进行；

- 面向对象软件的结构不再是传统的功能模块结构，作为一个整体，原有集成测试所要求的逐步将开发的模块搭建在一起进行测试的方法已成为不可能。
- 传统的测试模型与方法对OO软件已经不再适用。

单元测试和集成测试

■ 传统的单元测试：

- 测试对象：功能模块；
- 测试依据：详细设计阶段的程序流程图；
- 测试方法：白盒测试；

■ OO中的单元测试：

- 测试对象：类(属性+方法)，只有在类非常复杂时，才以单个方法作为测试对象；
- 测试依据：类的规格说明；
- 测试方法：类测试

■ 传统的集成测试：

- 自顶向下的集成测试；
- 自底向上的集成测试；

■ OO的集成测试：

- 因为不存在层次结构，上述两种集成策略均失去意义；
- 采用“类集成测试方法”。

测试策略与方法

- **测试类的操作**：使用前面介绍的黑盒测试和白盒测试方法，对类的每一个操作进行单独测试；
- **测试类**：使用等价类划分、基于状态的测试等方法，对每一个类进行单独测试；
- **测试类集成**：传统的自顶向下和自底向上集成方法不适合一组关联类的集成测试，可以使用基于用例场景的测试方法对一组关联类进行集成测试；
- **测试面向对象的系统**：与任何类型的系统一样进行测试。

类测试

- 类测试的目的：验证类的实现是否和它的说明完全一致；
- 类测试的对象：
 - 由于类的内部封装了各种属性和消息传递方式，类实例化后产生的对象都有相对的生命周期和活动范围，因此，除了要测试类中包含的属性、方法，还要测试类的状态；
- 类测试的过程：
 - 构造测试用例、进行实际测试；

类测试的方法

- 测试单个类的方法有三种：
 - 随机测试
 - 划分测试
 - 基于故障的测试

类测试的方法(1): 随机测试

- 银行应用系统中有一个**Account**类，它包含以下操作：
 - open()、setup()、deposit()、withdraw()、balance()、summarize()、creditLimit()、close()
- 一个**Account**类实例的最小行为的生命历史包含以下操作：
 - open • setup • deposit • withdraw • close
 - 这是Account类的最小测试序列。
- 但是，大量其他的行为可以在这个序列中发生：
 - open • setup • deposit • [deposit | withdraw | balance | summarize | creditLimit]ⁿ • withdraw • close
- 从该序列可随机产生一系列不同的操作序列，作为测试用例：
 - 测试用例 #1: open • setup • deposit • deposit • balance • summarize • withdraw • close
 - 测试用例#2: open • setup • deposit • withdraw • deposit • balance • creditLimit • withdraw • close

类测试的方法(2): 划分测试

- 与传统黑盒测试中的等价类划分方法类似，划分测试可减少测试特定类所需的测试用例的数量。
- 对输入和输出进行分类，设计测试用例以检查每个类。
 - 基于状态的划分：根据操作改变类状态的能力对类操作进行分类，分别进行测试；
 - 基于属性的划分：根据类操作使用的属性来划分类操作；
 - 基于类别的划分：根据类操作所完成的功能类别来划分类操作；

类测试的方法(2): 划分测试

■ 基于状态的划分

- Account类中，状态操作包括deposit()和withdraw()，其他均为非状态操作。
- 设计两个测试用例，分别进行测试。
 - # p1: open • setup • deposit • withdraw • withdraw • close
 - # p2: open • setup • deposit • summarize • creditLimit • withdraw • close
- # p1测试改变状态的操作，# p2测试不改变状态的操作；

■ 基于属性的划分

■ 基于类别的划分

类测试的方法(2): 划分测试

- 基于状态的划分

- 基于属性的划分

- Account类中, 使用属性balance来定义划分, 从而将操作划分为3个类别:
 - 读取balance的操作: `balance()`, `summarize()`, `creditLimit()`;
 - 修改balance的操作: `deposit()`, `withdraw()`;
 - 不读取也不修改balance的操作: 其他;
- 然后为每一类设计测试用例;

- 基于类别的划分

类测试的方法(2): 划分测试

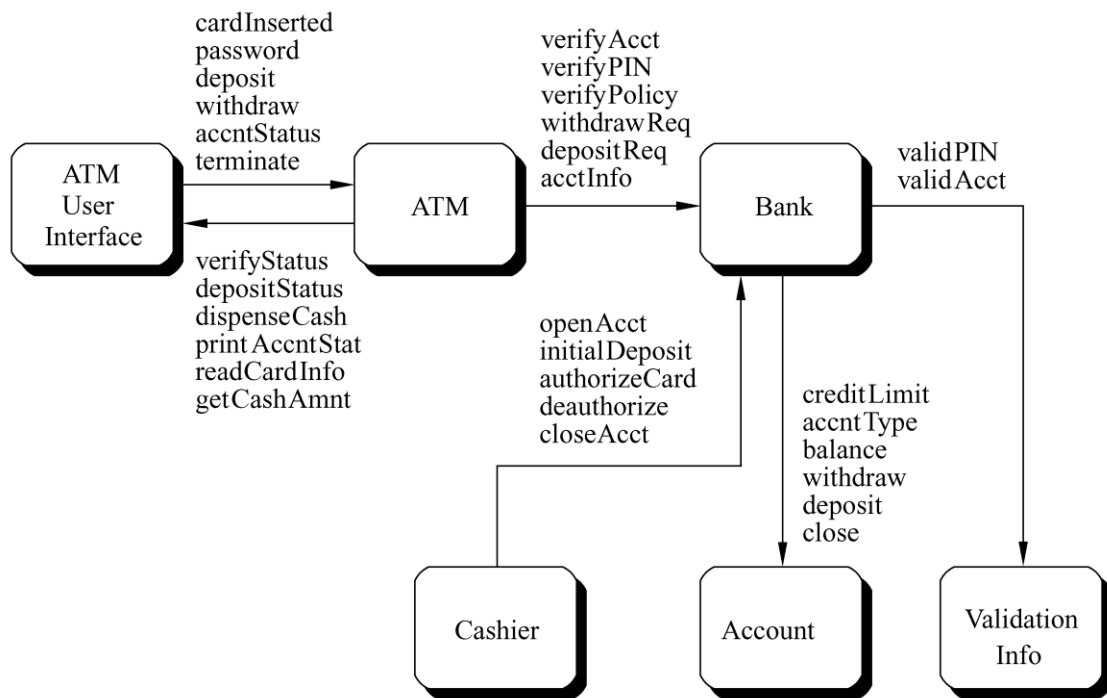
- 基于状态的划分
- 基于属性的划分
- 基于类别的划分
 - 把Account类中的操作分类为以下四类:
 - 初始化操作: open(), setup();
 - 计算操作: deposit(), withdraw();
 - 查询操作: balance(), summarize(), creditLimit();
 - 终止操作: close()
 - 然后为每个类别设计测试用例。

类测试的方法(3): 基于故障的测试

- 推测软件中可能有的错误，然后设计出最可能发现这些错误的测试用例；
- 应仔细研究分析模型和设计模型，而且在很大程度上要依靠测试人员的经验和直觉。

类集成测试

- 在集成测试阶段，必须对类间协作进行测试，设计测试用例。
- 测试方法：
 - 随机测试方法
 - 划分测试方法
 - 基于情景的测试
 - 行为测试



(1) 生成多类的随机测试用例

- 对每个客户类，使用类操作符列表来生成一系列随机测试序列。这些操作符向服务器类实例发送消息。
- 对所生成的每个消息，确定协作类和在服务器对象中的对应操作符。
- 对服务器对象中的每个操作符(已经被来自客户对象的消息调用)，确定传递的消息。
- 对每个消息，确定下一层被调用的操作符，并把这些操作符结合进测试序列中。

(1) 生成多类的随机测试用例

■ 考虑Bank类相对于ATM类的操作序列：

— verifyAcct • verifyPIN • [(verifyPolicy • withdrawReq) | depositReq | acctInfoREQ]ⁿ

■ 对Bank类的随机测试用例可能是：

— #r3: verifyAcct • verifyPIN • depositReq

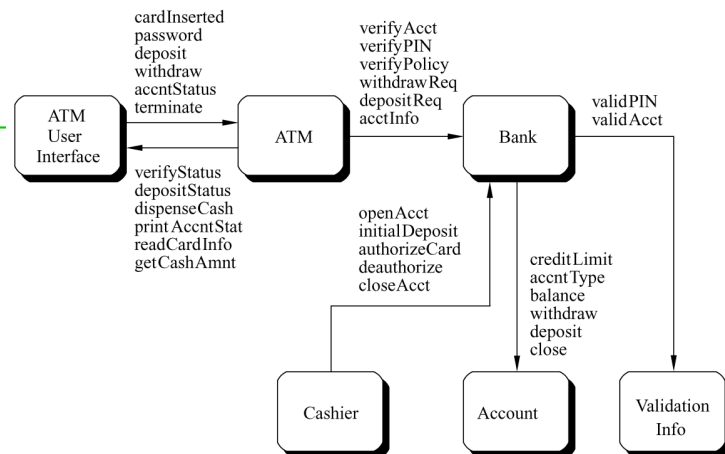
■ 为了考虑在上述这个测试中涉及的协作者，需要考虑与#r3中的每个操作相关联的消息。

— Bank必须和ValidationInfo协作以执行verifyAcct和verifyPIN；

— Bank必须和Account协作以执行depositReq；

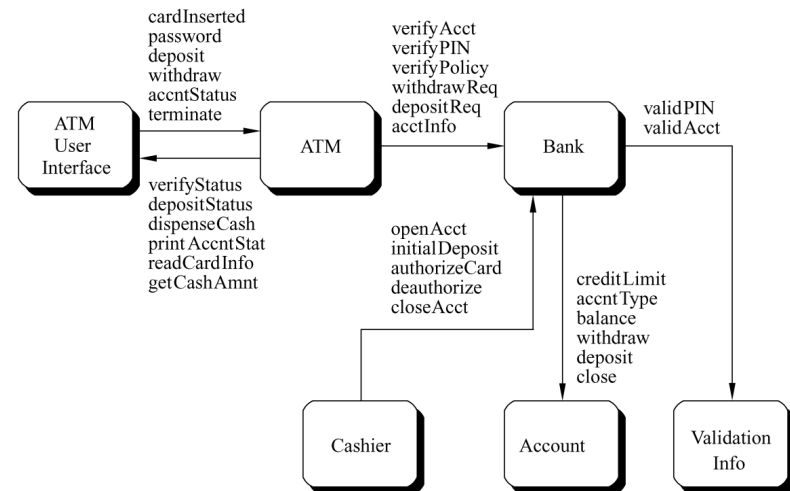
■ 因此，测试上面提到的协作的新测试用例是：

— #r4: verifyAcct_{Bank} • [validAcct_{ValidationInfo}] • verifyPIN_{Bank} • [validPIN_{validationInfo}] • depositReq • [deposit_{account}]



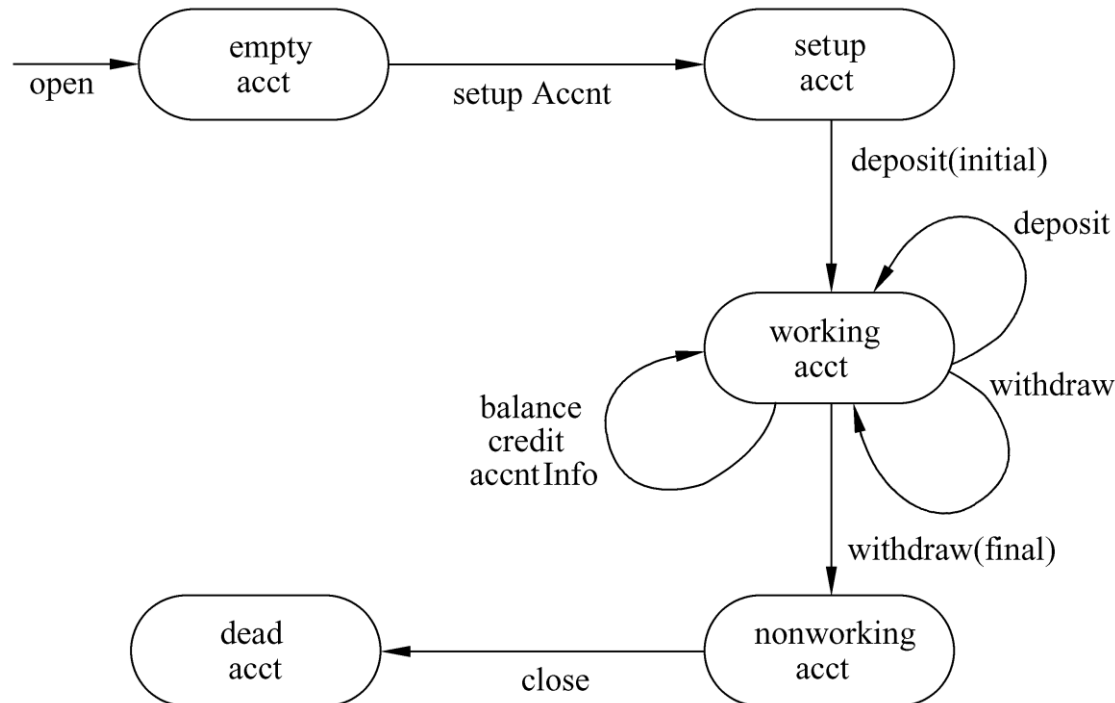
(2) 多类的划分测试方法

- 多个类的划分测试方法类似于单个类的划分测试方法。
- 但是，对于多类测试来说，应该扩充测试序列以包括那些通过发送给协作类的消息而被调用的操作。
- 另一种划分测试方法，根据与特定类的接口来划分类操作。
- 例如：**Bank**类接收来自**ATM**类和**Cashier**类的消息，因此可以通过把**Bank**类中的方法划分成服务于**ATM**的和服务于**Cashier**的两类来测试。



(3) 从动态模型导出测试用例

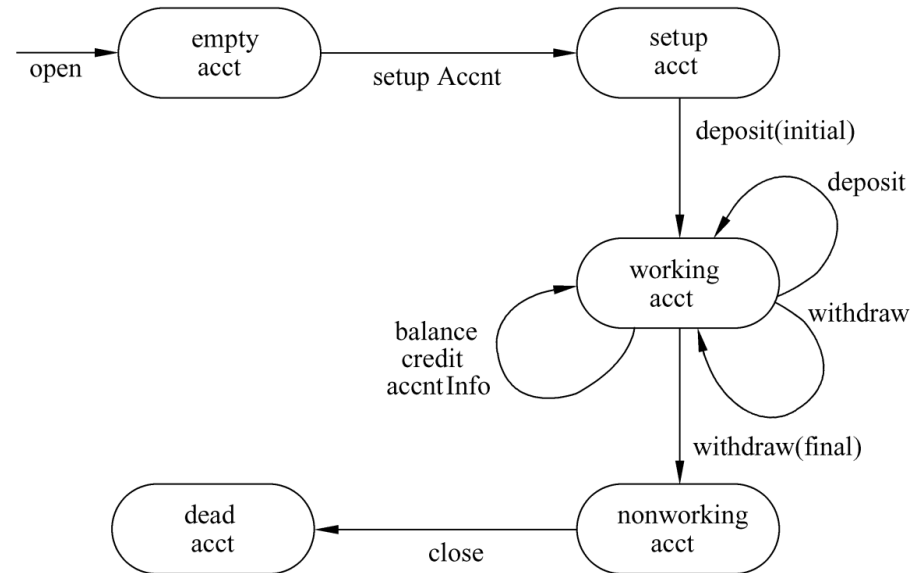
- 从类的状态转换图(statechart diagram)可以导出测试该类(及与其协作的那些类)的动态行为的测试用例。
- 以Account类的状态图为例：



(3) 从动态模型导出测试用例

- 设计出的测试用例应该覆盖所有状态：

- #s1: open • setupAcct • deposit(initial) • withdraw(final) • close



- 加入附加的测试序列，得出其他测试用例：

- #s2: open • setupAcct • deposit(initial) • deposit • balance • credit • withdraw(final) • close
- #s3: open • setupAcct • deposit(initial) • deposit • withdraw • acctInfo • withdraw(final) • close



5 小结



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

课堂讨论：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，分析它们之间是否存在哪些异同，最终的UML中分别包含了三者的哪些思想。



课外阅读：Pressman教材第6-7、
18-20章



课堂讨论

■ OO的五大设计原则SOLID

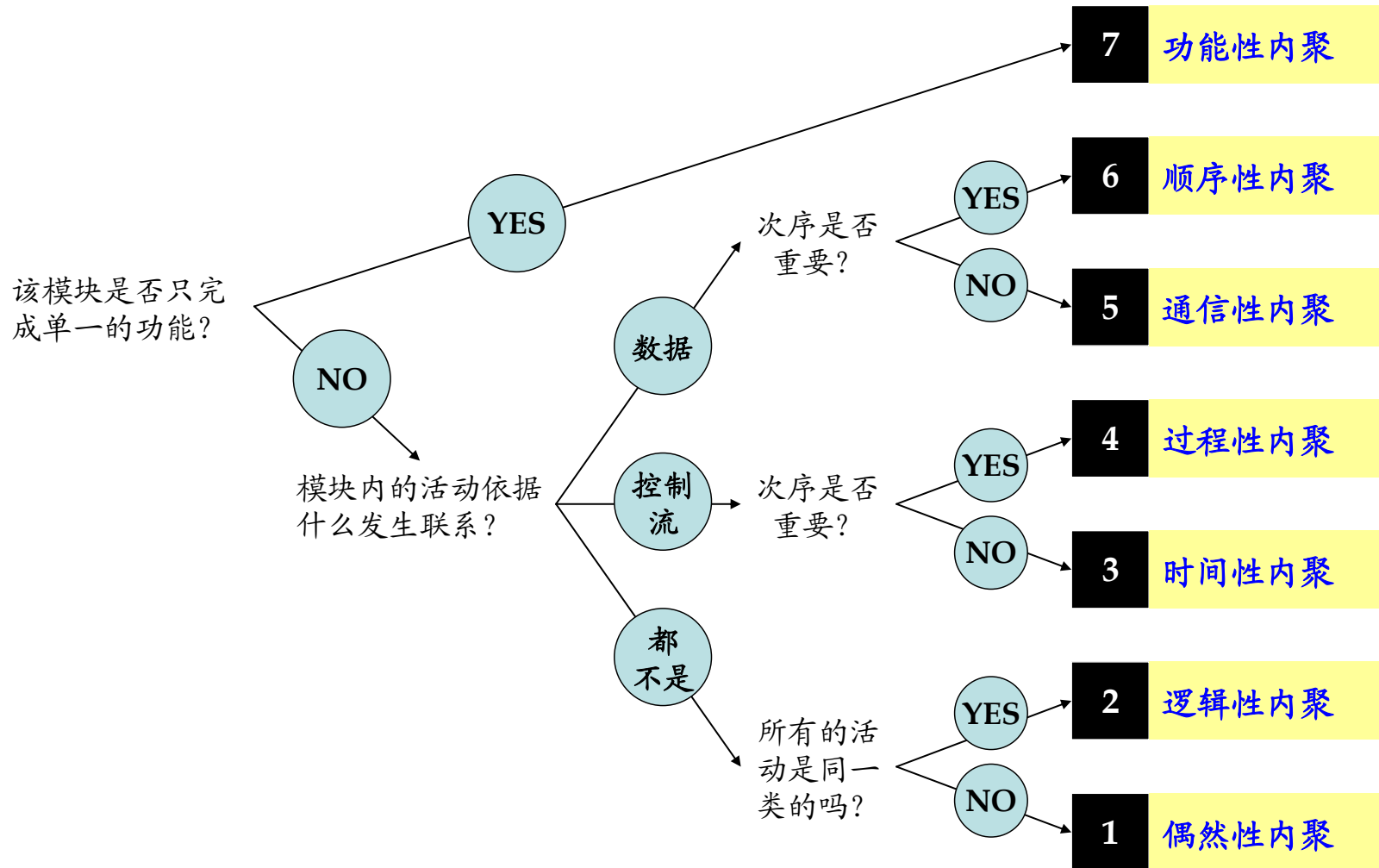
- (SRP) The Single Responsibility Principle 单一责任原则
- (OCP) The Open-Closed Principle 开放封闭原则
- (LSP) The Liskov Substitution Principle Liskov替换原则
- (ISP) The Interface Segregation Principle 接口隔离原则
- (DIP) The Dependency Inversion Principle 依赖转置原则

- 通过查阅资料了解它们的基本思想，它们共同应对软件设计中面临的什么NFR，并通过例子解释说明。

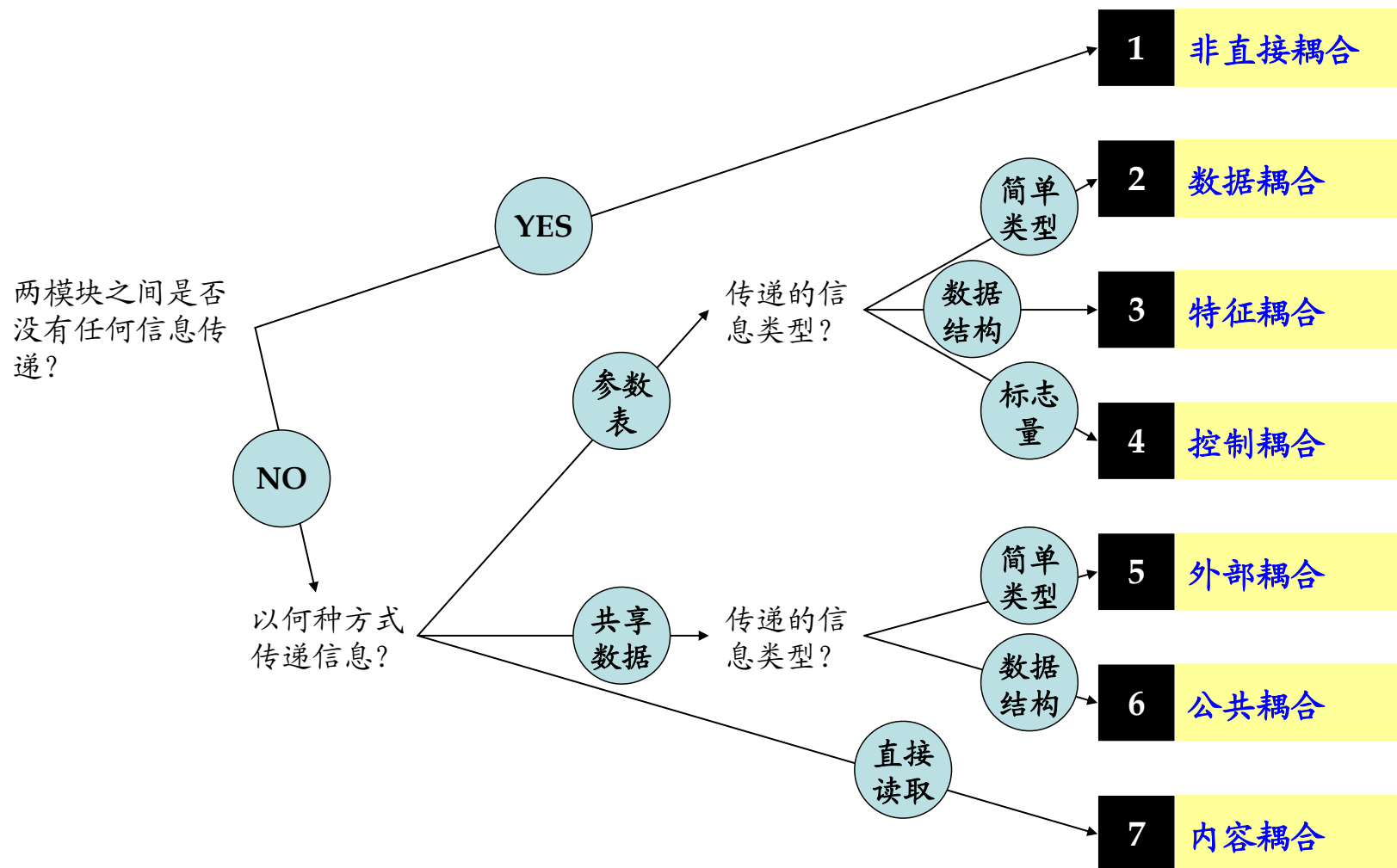
课堂讨论

- 模块化设计的目标：高内聚、低耦合(**high cohesion and low coupling**)
 - 模块内部的聚合度(Cohesion): 某一模块内部包含的各功能之间相互关联的紧密程度，有七种类型；
 - 模块之间的耦合度(Coupling): 多个模块之间相互关联的紧密程度，也有七种类型；
- “高内聚、低耦合”在OO中的体现：
 - (REP) The Reuse/Release Equivalency Principle 复用/发布等价原则
 - (CCP) The Common Closure Principle 共同封闭原则
 - (CRP) The Common Reuse Principle 共同复用原则
 - (ADP) The Acyclic Dependencies Principle 无圈依赖原则
 - (SDP) The Stable Dependencies Principle 稳定依赖原则
 - (SAP) The Stable Abstraction Principle 稳定抽象原则

内聚强度的划分



耦合强度的等级



课堂讨论

- 七种聚合度和七种耦合度；
- “高内聚、低耦合”在OO中体现的六项原则；
- 通过查阅资料，了解它们各自的特征是什么、会带来什么样的优点和危害？



結束

2017年11月6日