软件工程
第十章　构件级设计

乔立民

qlm@hit.edu.cn

2011年5月18日

主要内容

- **10.1 什么是构件**
- **10.2 构件设计原则**
- **10.3 构件设计步骤**
- **10.4 设计规格说明**

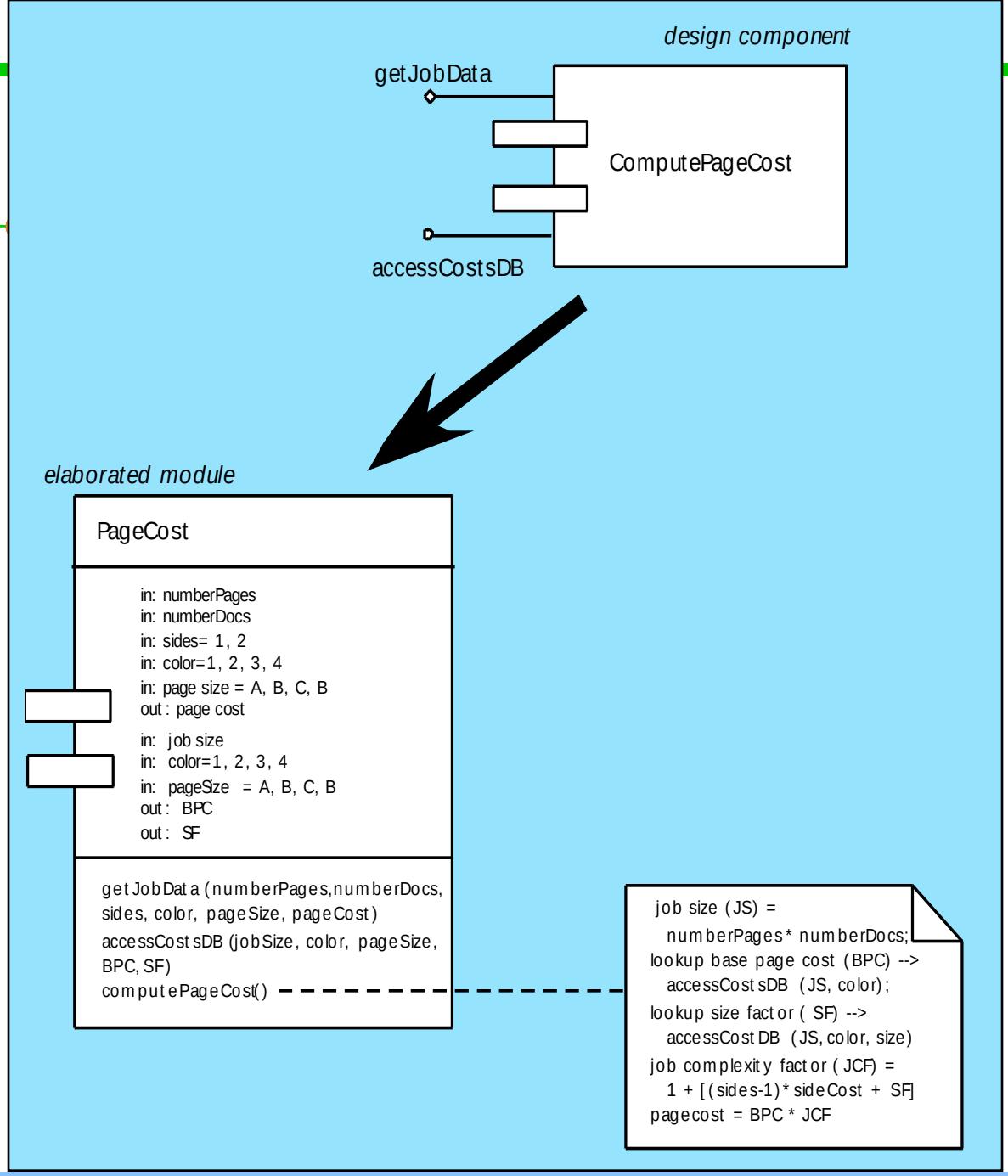
构件级设计

- **什么是构件？**
  - 系统中某一定型化的、可配置的和可替换的部件，该部件封装了实现并暴露一系列接口
- **面向对象的观点**
  - 构件包括一个协作类集合
- **传统观点**
  - 一个构件就是程序的一个功能要素，程序有处理逻辑及实现处理逻辑所需的内部数据结构以及能够保证构件被调用和实现数据传递的接口构成

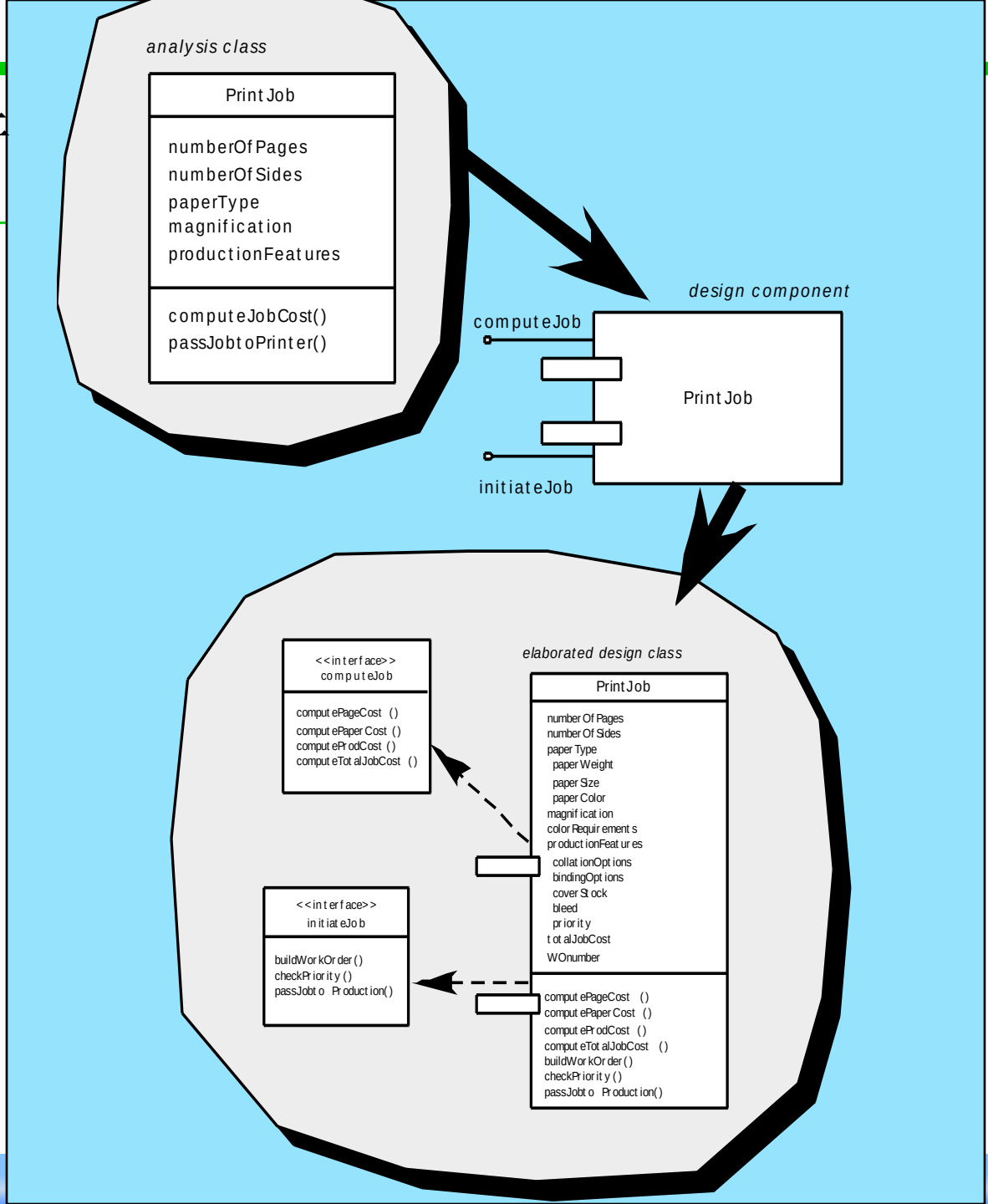
构件化带来的好处

- 管理上：将大的任务分割成小的子任务，并行开发提高效率
- 维护上：通过系统间解耦，使得一部分发生变化不会波及到其他方面
- 理解上：将复杂问题分解成小问题，每个构件关注一个问题，降低开发难度。
- 设计的目标就是根据设计原则和设计模式将职责分配给恰当的类
- 核心：构件设计的原则？

传统构件


design component
getJobData
ComputePageCost
accessCostsDB
elaborated module
PageCost
in: numberPages
in: numberDocs
in: sides= 1, 2
in: color=1, 2, 3, 4
in: page size = A, B, C, B
out: page cost
in: job size
in: color=1, 2, 3, 4
in: pageSize = A, B, C, B
out: BPC
out: SF
getJobData (numberPages,numberDocs, sides, color, pageSize, pageCost)
accessCostsDB (jobSize, color, pageSize, BPC, SF)
computePageCost()
job size (JS) =
numberPages* numberDocs;
lookup base page cost (BPC) -->
accessCostsDB (JS, color);
lookup size factor (SF) -->
accessCostDB (JS, color, size)
job complexity factor (JCF) =
1 + [(sides-1)* sideCost + SF]
pagecost = BPC * JCF

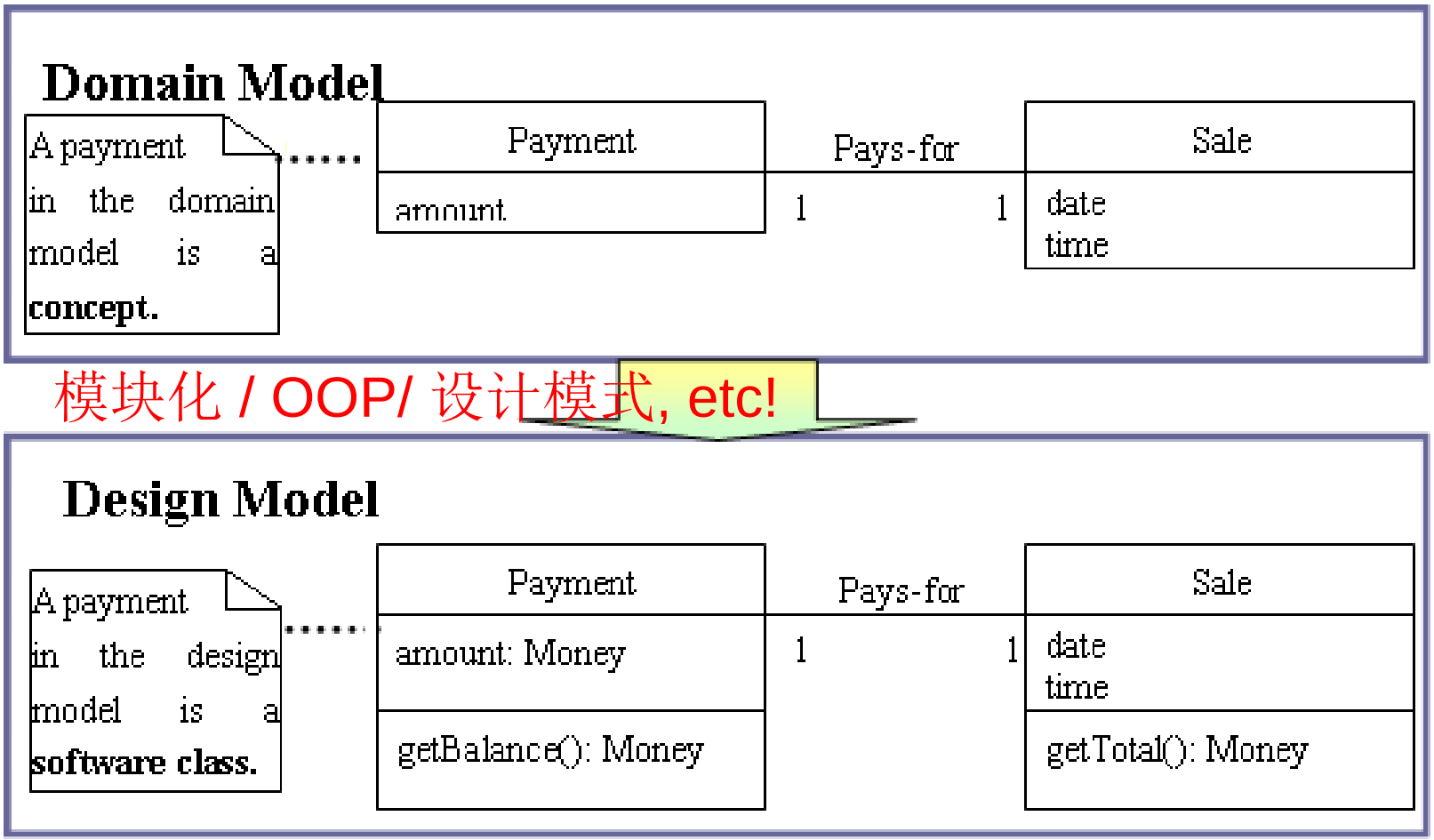
面向对象构件


analysis class
PrintJob
numberOfPages
numberOfSides
paperType
magnification
productionFeatures
computeJobCost()
passJobtoPrinter()
design component
computeJob
PrintJob
initiateJob
elaborated design class
<<interface>>
computeJob
computePageCost ()
computePaperCost ()
computeProdCost ()
computeTotalJobCost ()
PrintJob
numberOfPages
numberOfSides
paperType
paperWeight
paperSize
paperColor
magnification
colorRequirements
productionFeatures
collationOptions
bindingOptions
coverStock
bleed
priority
totalJobCost
WOnumber
computePageCost ()
computePaperCost ()
computeProdCost ()
computeTotalJobCost ()
buildWorkOrder ()
checkPriority ()
passJobto Production()
<<interface>>
initiateJob
buildWorkOrder ()
checkPriority ()
passJobto Production()

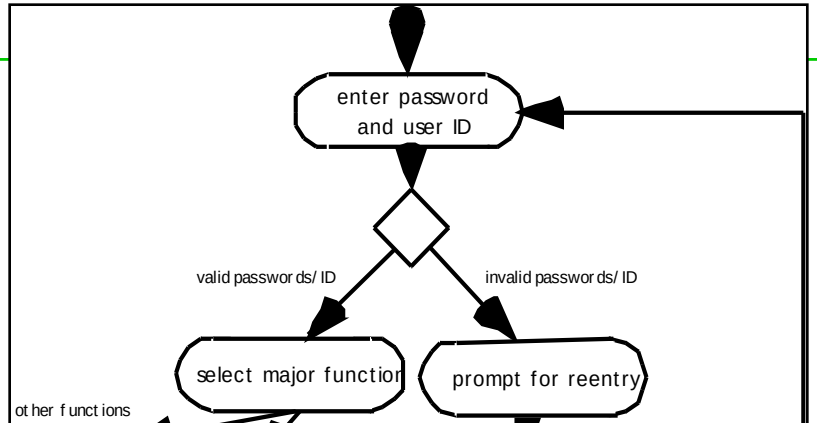
构件级设计

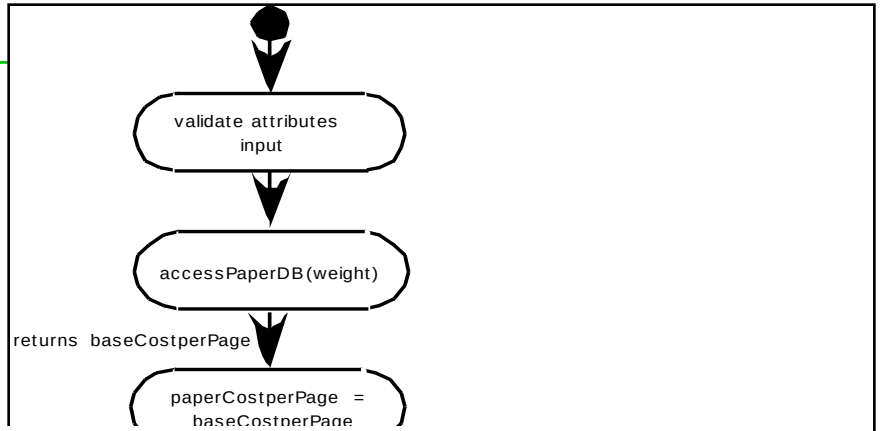
- **相当于详细设计(即设计对象内部的具体实现)；**
- **细化需求分析模型和系统体系结构设计模型；**
- **识别新的对象；**
- **在系统所需的应用对象与可复用的商业构件之间建立关联；**
  - 识别系统中的应用对象；
  - 调整已有的构件；
  - 给出每个子系统/类的精确规格说明。

OOD——设计类图


Domain Model
A payment in the domain model is a concept.
Payment
amount
Pays-for
1
1
Sale
date
time
模块化 / OOP/ 设计模式, etc!
Design Model
A payment in the design model is a software class.
Payment
amount: Money
getBalance(): Money
Pays-for
1
1
Sale
date
time
getTotal(): Money

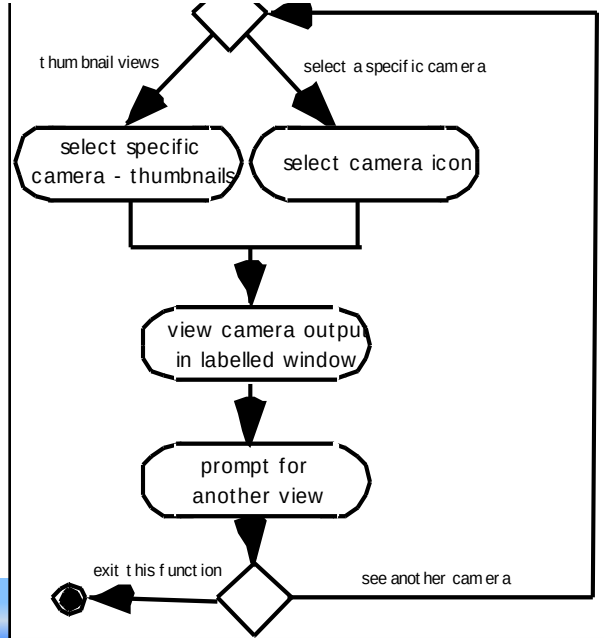
OOD——设计活动图



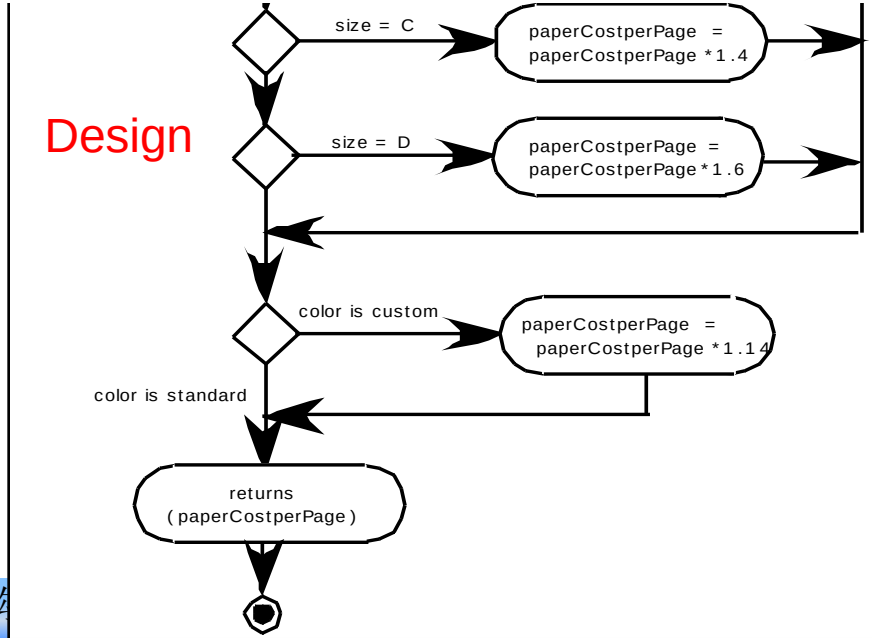


1.根据类图进行细节设计

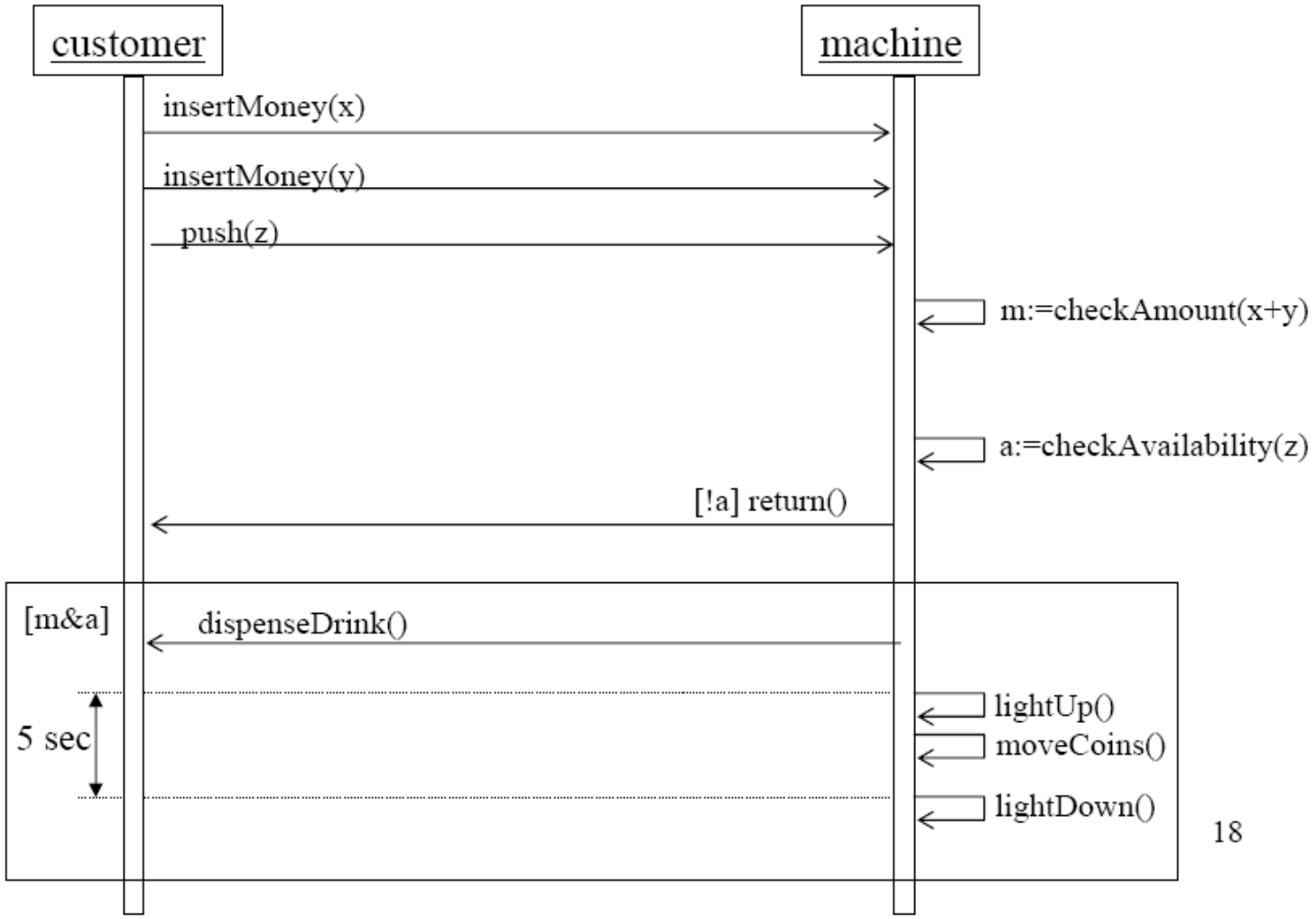
2.根据模块化/OOP/职责等将事件/动作 分配给类方法



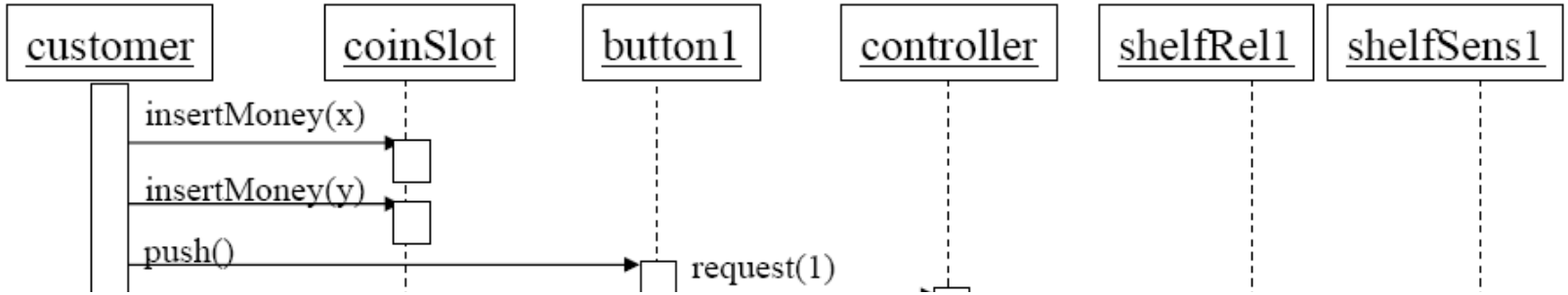
Analysis Design



OOD ——分析顺序图

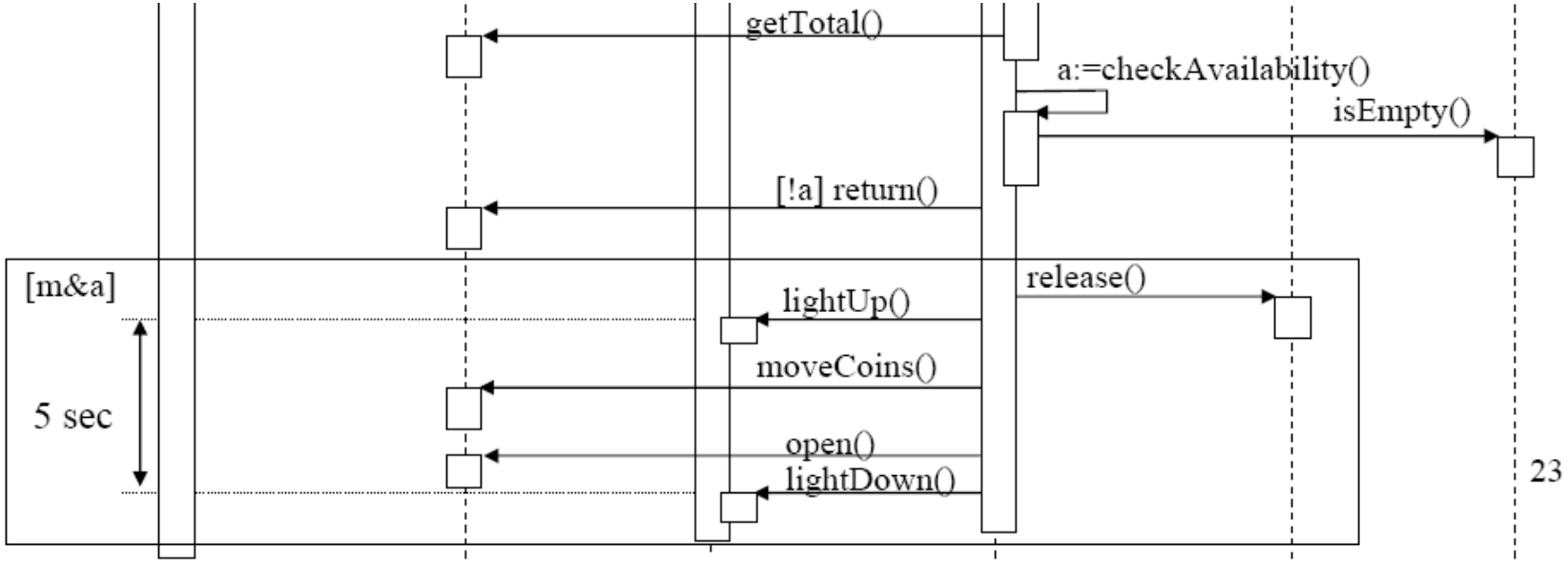

customer
machine
insertMoney(x)
insertMoney(y)
push(z)
m:=checkAmount(x+y)
a:=checkAvailability(z)
[!a] return()
[m&a]
dispenseDrink()
5 sec
lightUp()
moveCoins()
lightDown()
18

OO Design—设计顺序图

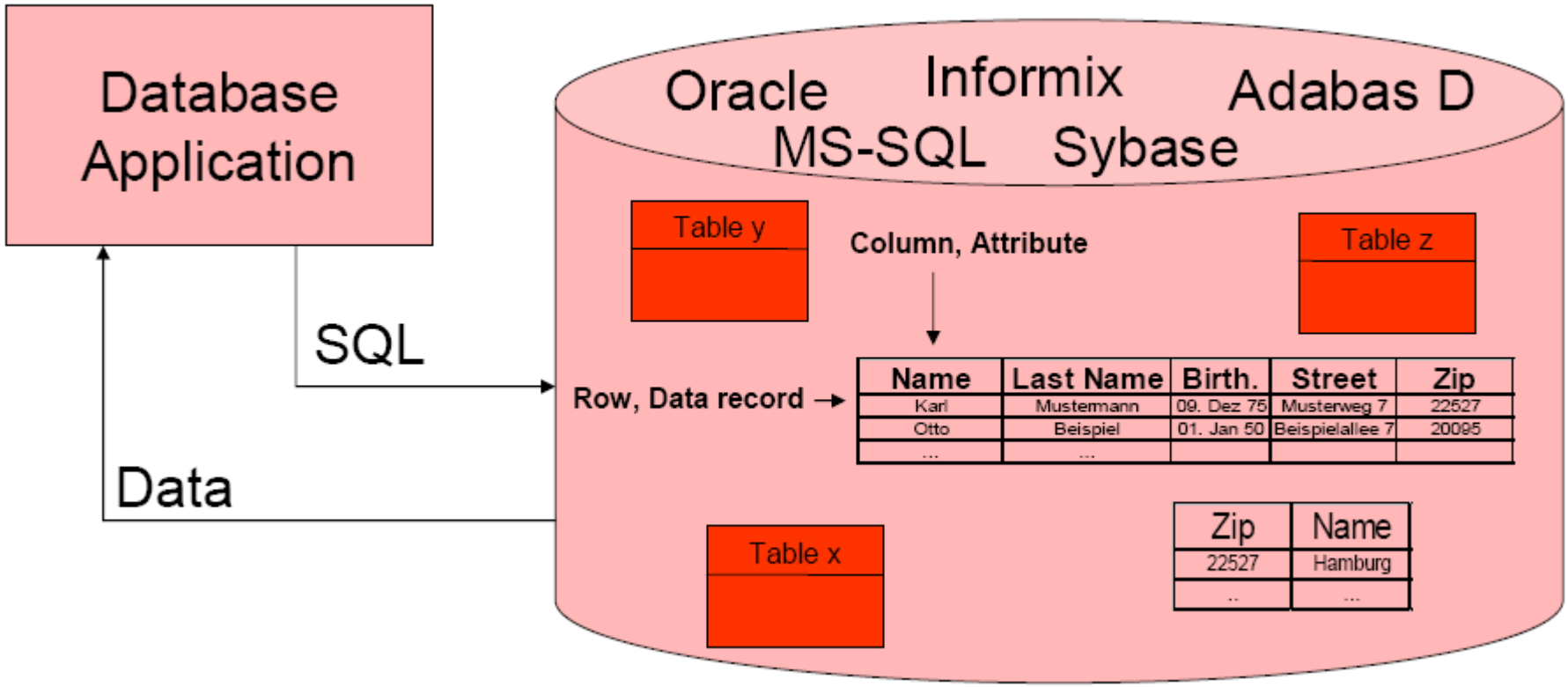


1.根据类图细节设计

2.根据职责/模块化/OOP等将操作分配给类方法



关系数据库设计


Database Application
SQL
Data
Oracle
Informix
Adabas D
MS-SQL
Sybase
Table y
Column, Attribute
Table z
Row, Data record
Name
Last Name
Birth.
Street
Zip
Karl
Mustermann
09. Dez 75
Musterweg 7
22527
Otto
Beispiel
01. Jan 50
Beispielallee 7
20095
Table x
Zip
Name
22527
Hamburg

关系数据库设计

- **为每个实体建立一张表(table)；**
- **为每个表确定主键(primary key)、外键(foreign key)；**
- **确定表间关系：**
  - 增加外部码表示一对多(1:n)关系
  - 建立新表，表示多对多(m:n)关系
- **检查数据字典；**
- **对数据表进行规范化(3NF即可)**
- **数据库完整性设计**
- **数据库安全性设计**

应用《数据库系统》课程中学习的相关知识

主要内容

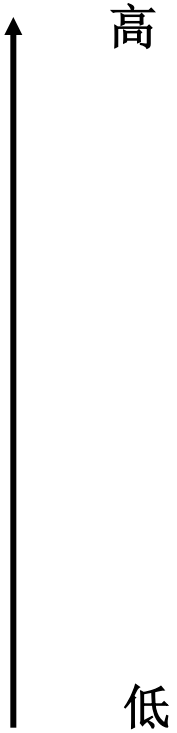
- **10.1 什么是构件**
- **10.2 构件设计原则**
  - 10.2.1 内聚性与耦合性
  - 10.2.2 构件设计原则
- **10.3 构件设计步骤**
- **10.4 设计规格说明**

内聚性

- **某一构件内部包含的各功能之间相互关联的紧密程度；**
- **传统观点**
  - 构件的专诚性
- **面向对象观点**
  - 构件或者类只封装哪些相互关系密切，以及与构件或类自身有密切关系的属性或操作

内聚性

1. 功能内聚
2. 分层内聚
3. 通信内聚
4. 顺序内聚
5. 过程内聚
6. 暂时内聚
7. 实用内聚



(1)功能内聚

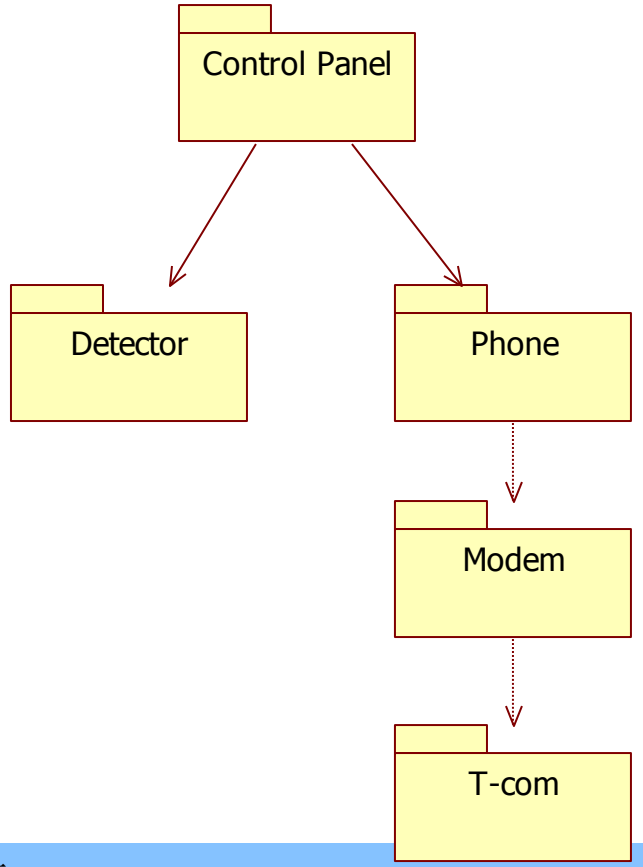
- **构件中各个部分都是为完成一项单一的功能而协同工作**
  - 类只执行单一的计算并返回结果
  - 无副作用(执行前后系统状态相同)，对其他模块无影响；
  - 这类模块通常粒度最小，且不可分解；
- **例如：**
  - 根据输入的角度，计算其正弦值；
  - 求一组数中的最大值；
- **“集中精力做一件事情”**

优点：

- 模块的功能只是生成特定的输出而没有副作用，容易理解该模块。
- 因功能内聚的模块没有副作用，所以模块的可复用性高。
- 功能单一、易修改、易替换、易维护

(2)分层内聚

- 由包、构件和类来表现
- 高层能够访问低层的服务，但低层不能访问高层

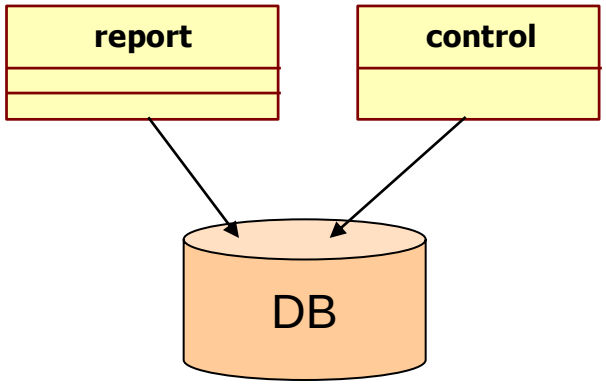


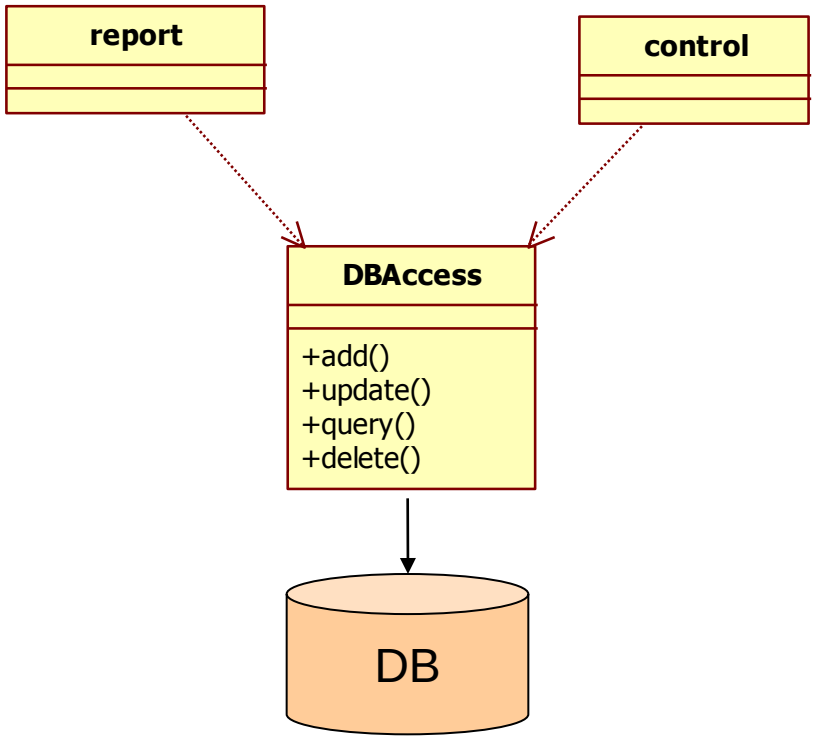
优点：横向功能内聚性好

缺点：效率较低

(3)通讯内聚

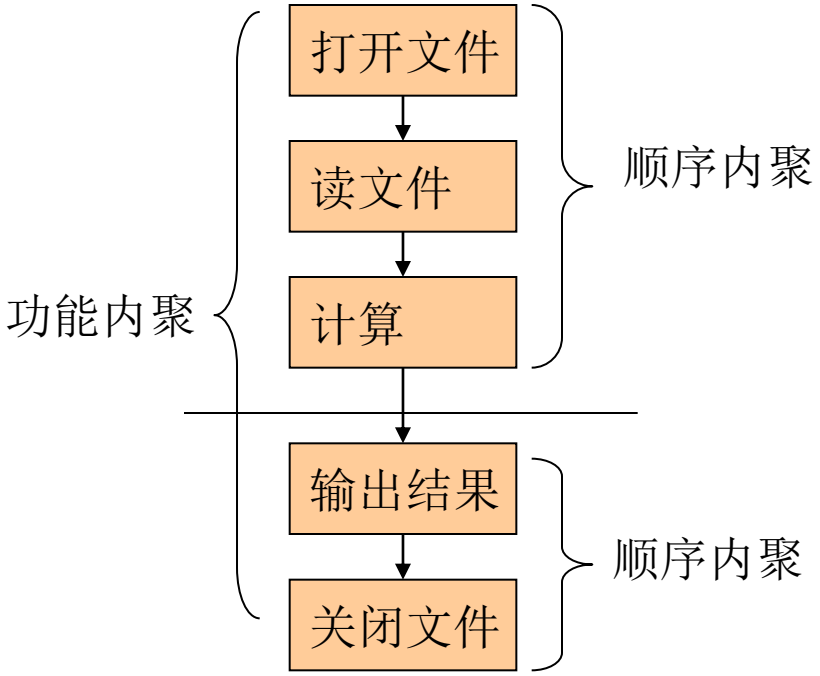
- 访问相同数据的所有操作被定义在一个类中
- 这个类只负责数据的查询、访问和存储





(4)顺序内聚

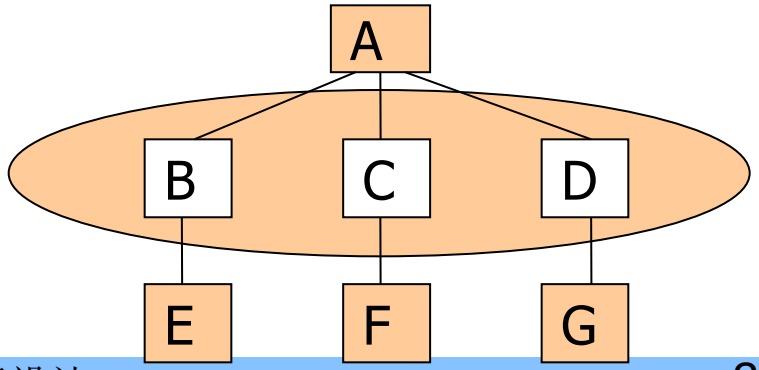
- 模块中的操作遵循某一特定的顺序
- 构件或者操作按照前者为后者提供输入的方式组合



(5)过程内聚

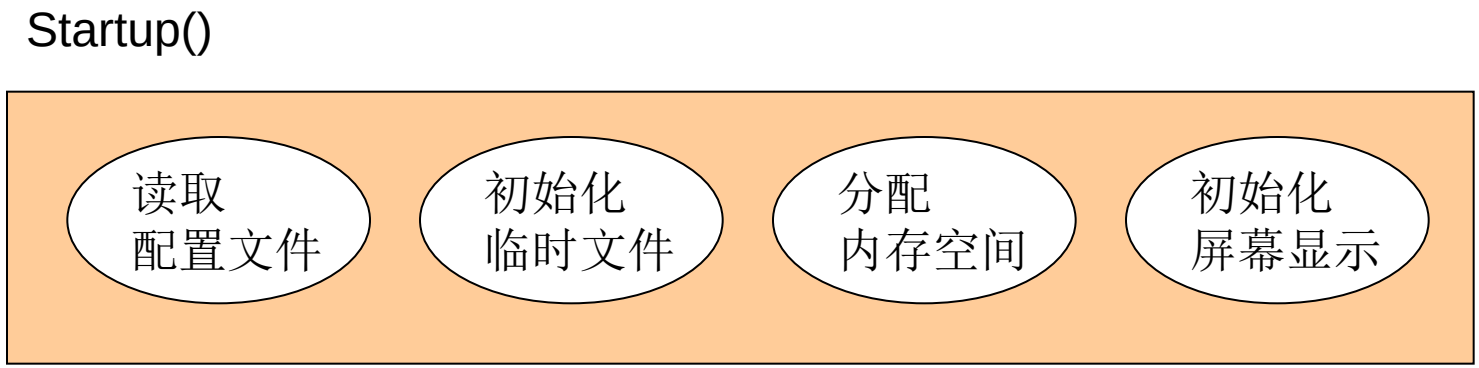
- 一个模块中同时含有几个操作，这些操作之间既无顺序关系，也无数据共享关系；
- 它们的执行与否由外面传进来的控制标志所决定
- 之所以称之为过程内聚，是因为这些操作仅仅是因为控制流，或者说在同一过程内的原因才联系到一起的，它们都被包括在一个很大的**if**或者**case**语句中，彼此之间并没有任何其它逻辑上的联系。

问题：接口难于理解；完成多个操作的代码互相纠缠在一起，导致严重的维护问题。



(6)暂时内聚

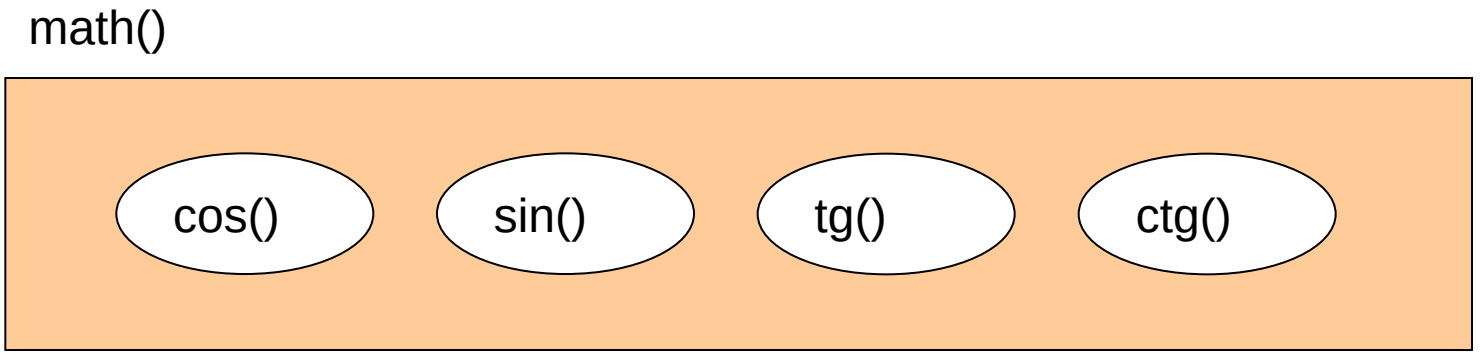
- **模块的各个成分必须在同一时间段执行，但各个成分之间无必然的联系**



问题：不同的功能混在一个模块中，有时共用部分编码，使局部功能的修改牵动全局。

(7)实用内聚

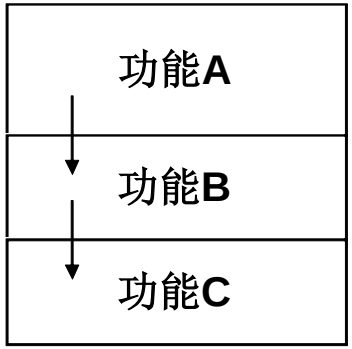
- 构成模块的各组成部分无任何关联。
- 通常用于库函数管理，将多个相互无关但比较功能类似的模块放置在同一个模块内。



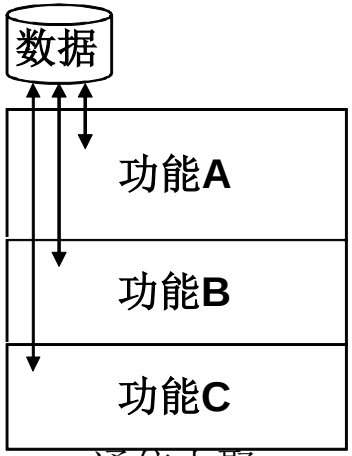
缺点：产品的可维护性退化；模块不可复用，增加软件成本。
解决：将模块分成更小的模块，每个小模块执行一个操作。

内聚强度的划分

功能内聚
完备的、相关的功能

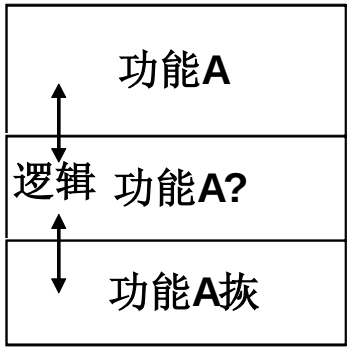


分层内聚
高层访问低层的服务



通信内聚
访问同样的数据

顺序内聚
按功能的顺序相关



过程内聚
类似的功能

暂时内聚
按时间相关

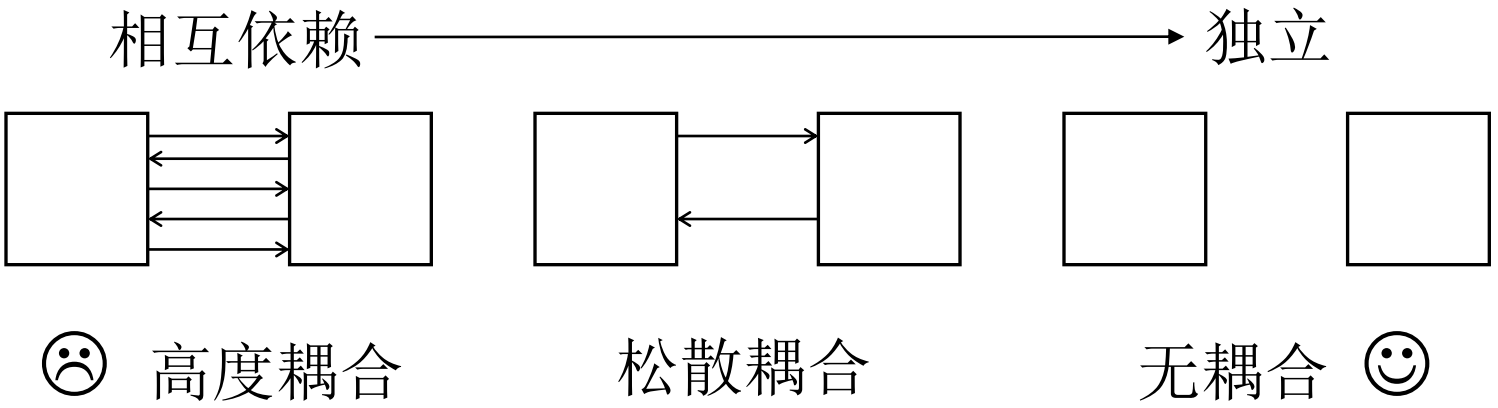
实用内聚
各部分不相关

尽量采用前三种内聚，避免后四种内聚

耦合性

- **传统观点**
  - 构件和其他构件或外部世界的连接程度
- **面向对象观点**
  - 耦合性是类之间彼此联系程度的一种定性度量

模块独立性之耦合度



影响模块之间发生耦合的因素

- 一个构件引用另一个构件；
- 一个构件传递给另一个构件数据；
- 某个构件控制或调用其他构件；
- 构件之间接口的复杂程度；

耦合性

- **多个模块之间相互关联的紧密程度**

1. 例程调用耦合
2. 数据耦合
3. 印记耦合
4. 类型使用耦合
5. 控制耦合
6. 外部耦合
7. 共用耦合
8. 内容耦合

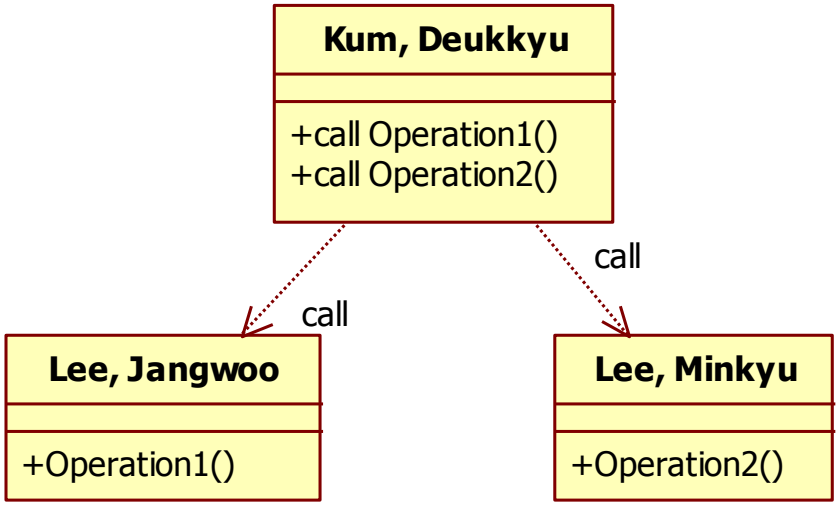
弱耦合

中耦合

强耦合

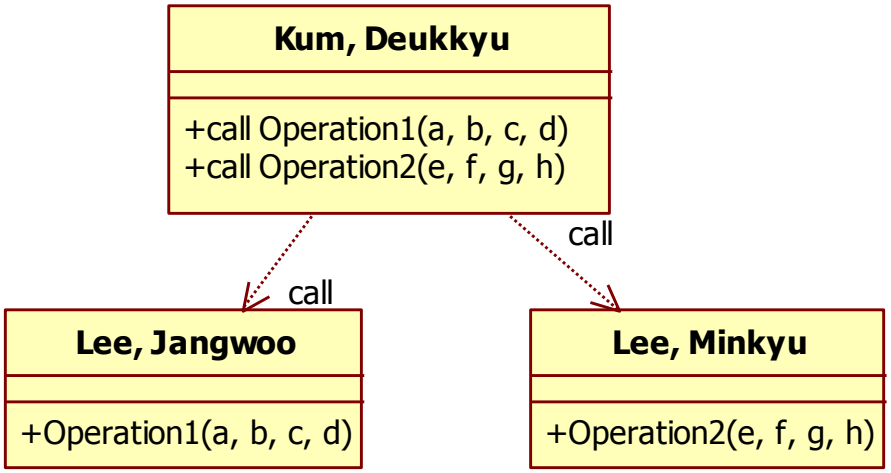
(1) 例程调用耦合

- 当一个操作调用另外一个操作时就会发生此种耦合
- 简单的、常见的、必要的



(2) 数据耦合

- 当操作需要传递较长的数据参数时就会发生此种耦合
- 过多的参数为测试、维护带来困难

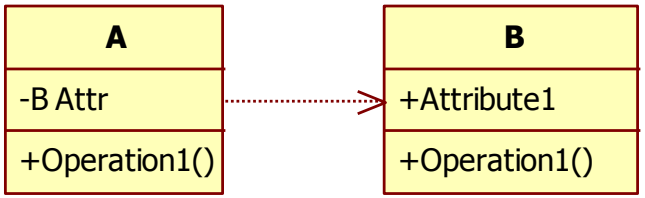


(3) 印记耦合

- 当类B被声明为类A的某一操作中的参数类型时发生的耦合
- 类B作为类A定义的一部分，类B的修改影响到类A

(4) 类型使用耦合

- 当构件A使用了在构件B中定义的一个数据类型时会发生此种耦合
- 一个类将某个变量声明为另一个类的类型

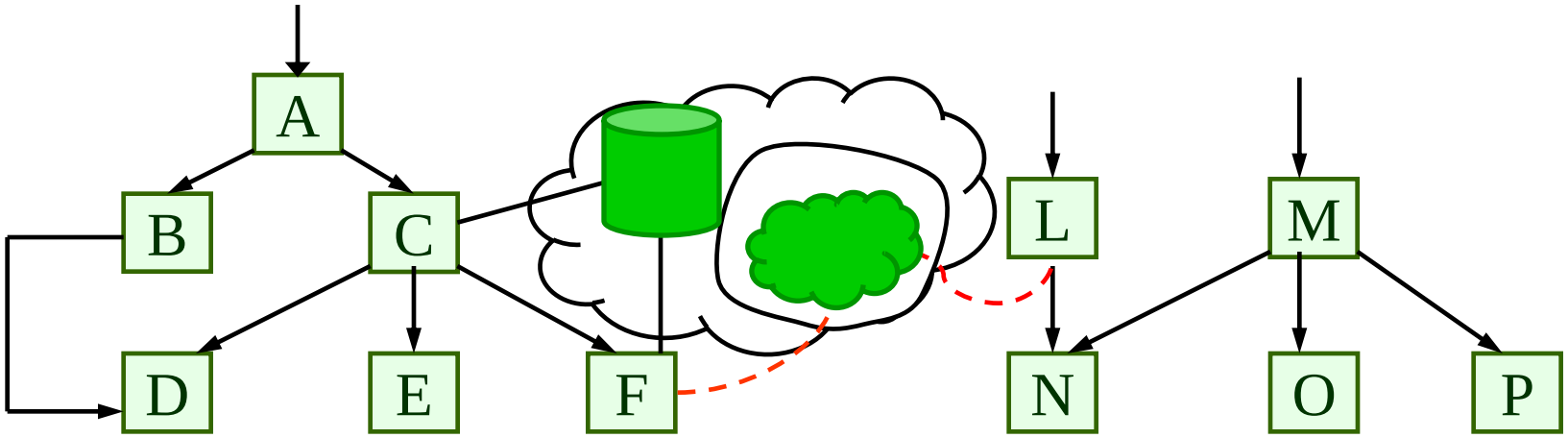


(5) 控制耦合（过程内聚）

- 当操作A调用操作B，并向B传递了一个控制标记时，就会发生此种耦合。
- 控制标记会指引B中的逻辑流程
- B中的一个不相关变更可能导致A传递的控制标记失去意义，A就必须变更

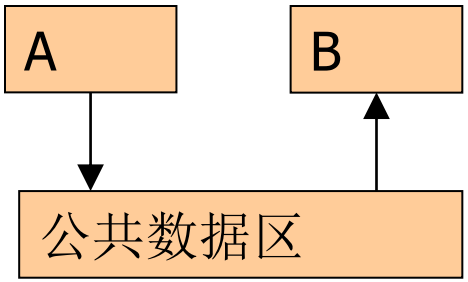
(6) 外部耦合

- 当一个构件和基础设施构件（如操作系统功能，数据库功能、网络通信功能等）进行通信和协作时会发生此种耦合

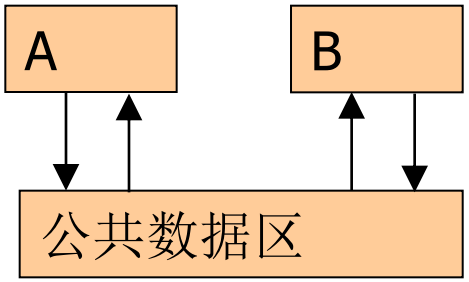


(7) 共用耦合

- 若一组模块都访问同一个公共数据环境，则它们之间的耦合就称为公共耦合。
- 公共的数据环境可以是全局变量、共享的通信区、内存的公共覆盖区等。



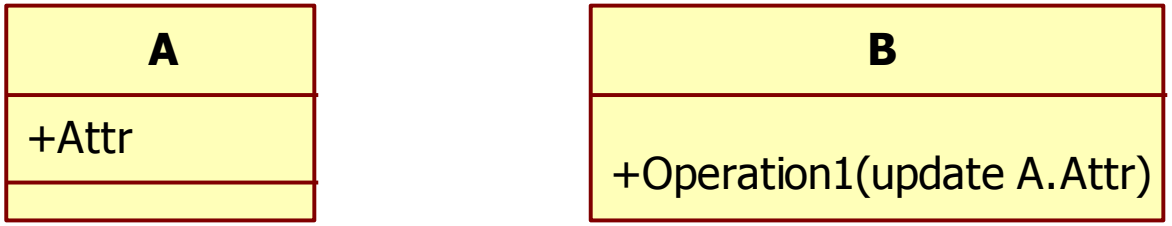
松散的公共耦合



紧密的公共耦合

(8) 内容耦合

- 一个类暗中修改另一个类的内部数据;
- 内容耦合违反了“信息隐藏”的原则。



关于耦合度的小结

- **耦合是影响软件复杂程度和设计质量的重要因素，应建立模块间耦合度尽可能松散的系统；**

- **降低模块间耦合度：**
  - 尽量使用数据、印记、类型使用耦合
  - 限制共用、外部耦合的范围
  - 坚决避免使用内容耦合

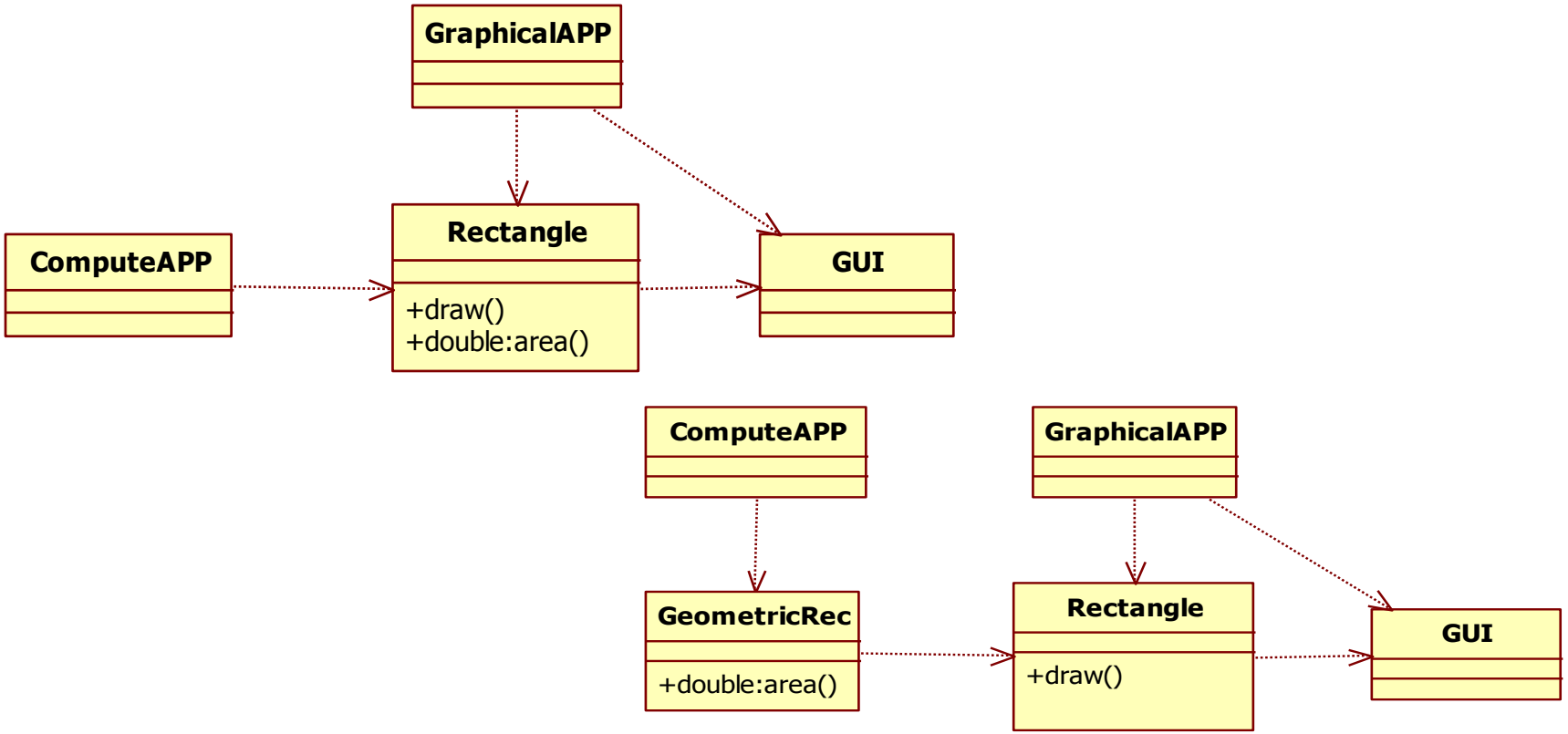
  - 尽量少的交换数据
  - 尽量小的接口
  - 尽量简单的数据交换

主要内容

- **10.1 什么是构件**
- **10.2 构件设计原则**
  - 10.2.1 内聚性与耦合性
  - 10.2.2 构件设计原则
- **10.3 构件设计步骤**
- **10.4 设计规格说明**

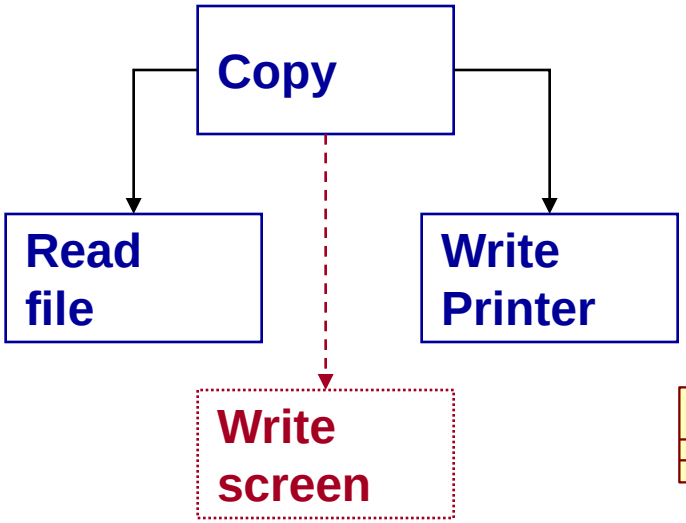
1.单一职责原则 (Single Responsibility Principle,SRP)

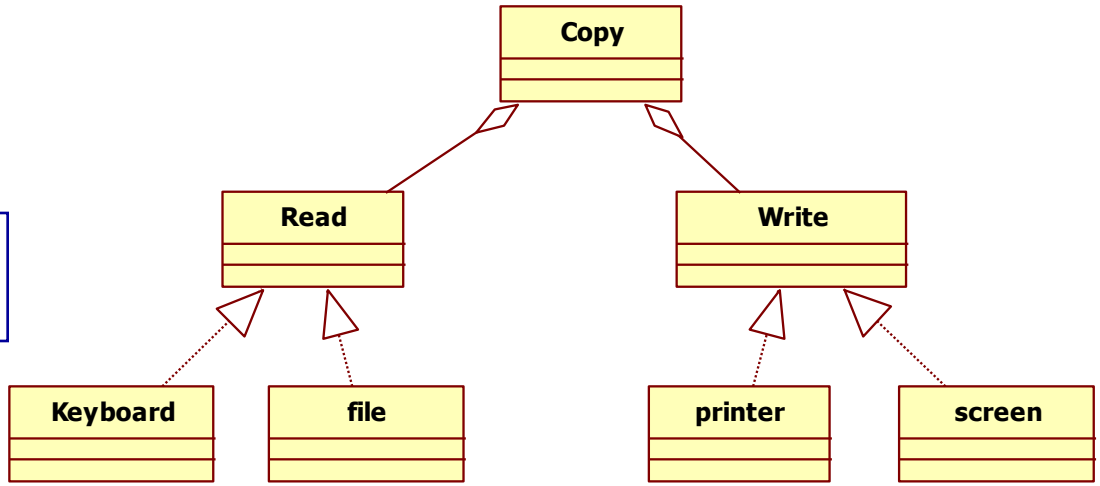
- 一个类应该有且仅有一个职责
- 职责：变化的原因



2.开闭原则(The Open-Close Principle,OCP)

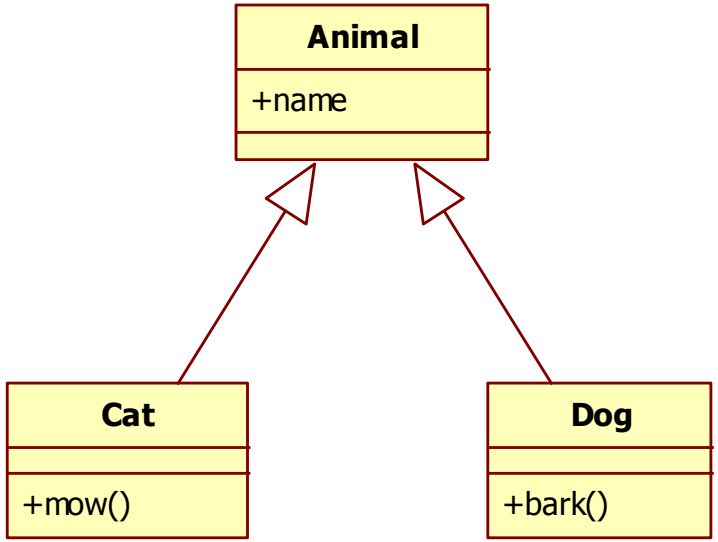
- 对于扩展是开放的：适应需求的变化
- 对于更改是封闭的：不影响已有的程序
- 关键是抽象





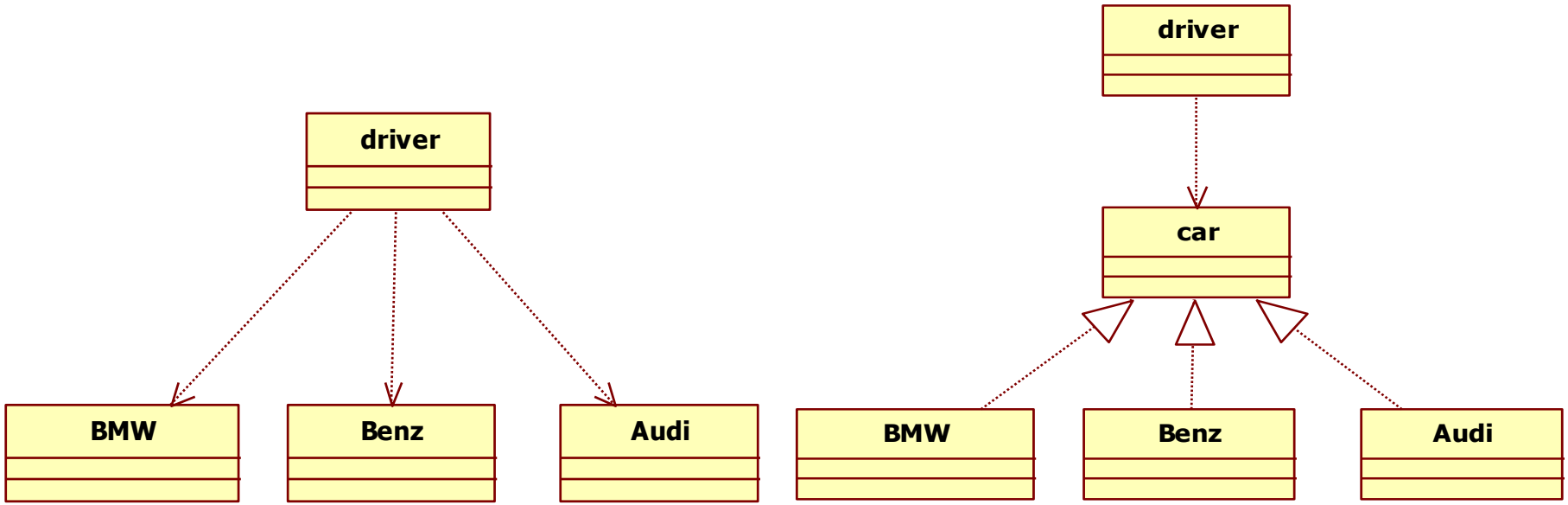
3.Liskov替换原则 (Liskov Substitution Principle,LSP)

- 子类必须能够替换它们的基类
- 按照接口和抽象类的规范继承父类
- 遵循LSP可实现动态加载



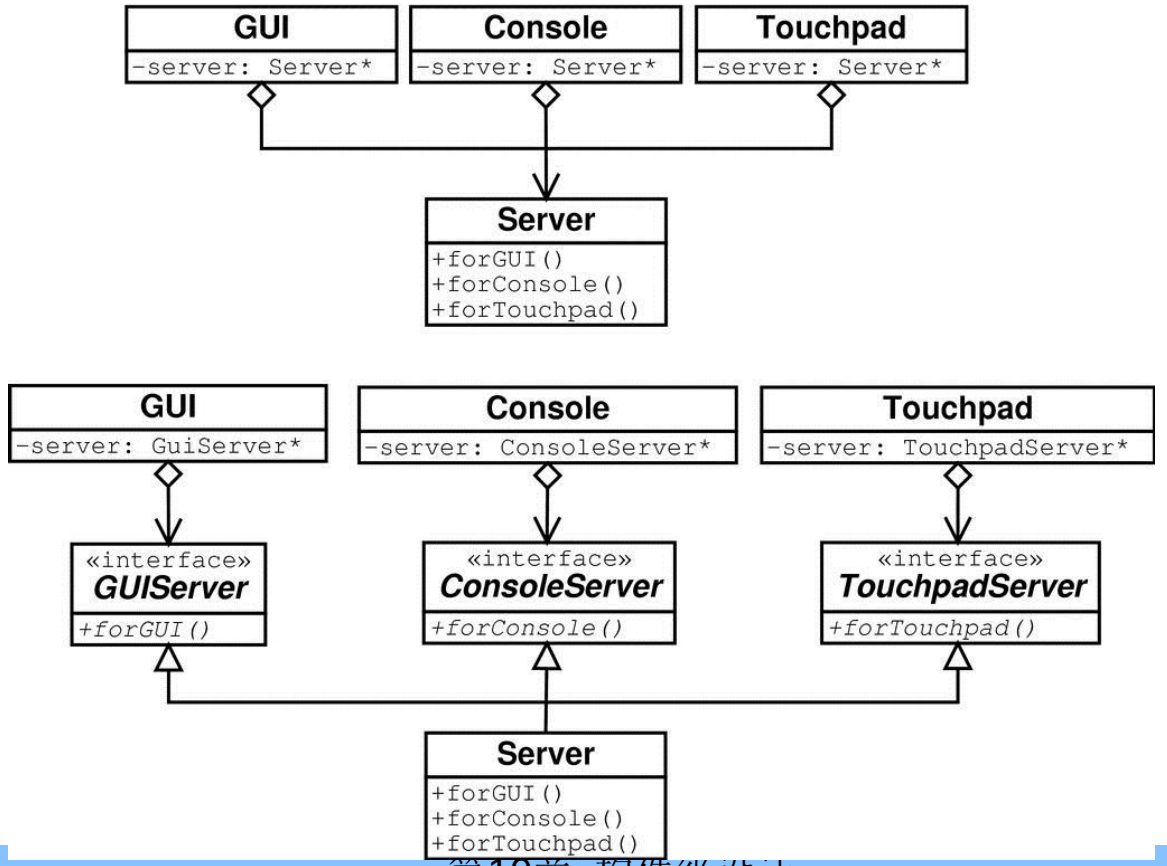
4.依赖倒置原则 (Dependency Inversion Principle,DIP)

- 依赖于抽象，而非具体实现



5.接口分离原 (Interface Segregation Principle,ISP)

- 多个用户专用接口比一个通用接口要好



构件打包的基本原则

- **发布复用等价性原则(Release Reuse Equivalency Principle REP)**
  - 复用的粒度就是发布的粒度
- **共同封装原则(Common Closure Principle,CCP)**
  - 一同变更的类应该合在一起
- **共同复用原则(Common Reuse Principle,CRP)**
  - 不能一起复用的类不能被分到一组

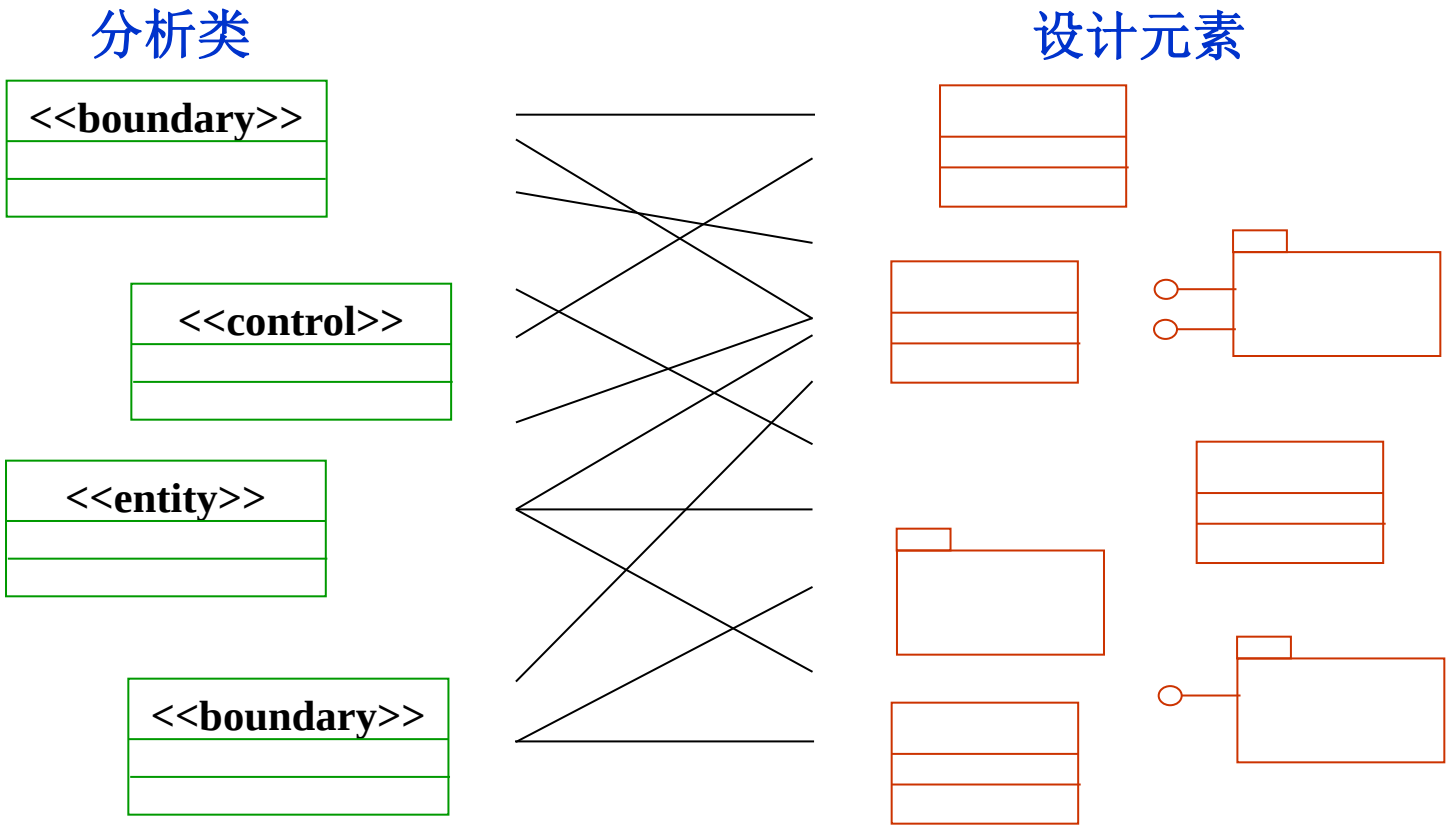
主要内容

- **10.1 设计建模**
- **10.2 构件设计原则**
- **10.3 构件设计步骤**
- **10.4 设计规格说明**

实施构件级设计

- **步骤1：标识所有与概念类相对应的设计类**
- **步骤2：确定所有与基础设施域相对应的设计类**
- **步骤3：细化所有不能作为复用构件的设计类**
  - 步骤3a：在类或构件的协作时说明消息的细节
  - 步骤3b：为每一个构件确定适当的接口
  - 步骤3c：细化属性并且定义相应的数据类型和数据结构
  - 步骤3d：详细描述每个操作的处理流
- **步骤4：说明持久数据源（数据库和文件）并确定管理数据源所需要的类**
- **步骤5：开发并且细化类或构件的行为表示**
- **步骤6：细化部署图以提供额外的实现细节**
- **步骤7：考虑每一个构件级设计表示，并且时刻考虑其他选择**

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。

例：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***
	……	……

实施构件级设计

- **步骤1：标识所有与问题类相对应的设计类**
- **步骤2：确定所有与基础设施域相对应的设计类**
- **步骤3：细化所有不能作为复用构件的设计类**
  - 步骤3a：在类或构件的协作时说明消息的细节
  - 步骤3b：为每一个构件确定适当的接口
  - 步骤3c：细化属性并且定义相应的数据类型和数据结构
  - 步骤3d：详细描述每个操作的处理流
- **步骤4：说明持久数据源（数据库和文件）并确定管理数据源所需要的类**
- **步骤5：开发并且细化类或构件的行为表示**
- **步骤6：细化部署图以提供额外的实现细节**
- **步骤7：考虑每一个构件级设计表示，并且时刻考虑其他选择**

标识基础设施设计类

- **GUI（图形用户接口）构件**
- **操作系统构件**
- **与硬件通信构件（网络、读写设备等）**
- **对象和数据管理构件**

实施构件级设计

- **步骤1：标识所有与问题类相对应的设计类**
- **步骤2：确定所有与基础设施域相对应的设计类**
- **步骤3：细化所有不能作为复用构件的设计类**
  - 步骤3a：为每一个构件确定适当的接口
  - 步骤3b：细化属性并且定义相应的数据类型和数据结构
  - 步骤3c：在类或构件的协作时说明消息的细节，详细描述每个操作的处理流
  - 步骤3d：细化类之间的关系
- **步骤4：说明持久数据源（数据库和文件）并确定管理数据源所需要的类**
- **步骤5：开发并且细化类或构件的行为表示**
- **步骤6：细化部署图以提供额外的实现细节**
- **步骤7：考虑每一个构件级设计表示，并且时刻考虑其他选择**

细化设计类的基本步骤

1. **确定接口**
2. **细化属性**
3. **细化操作**
4. **细化关系**
   - 细化依赖关系
   - 细化关联关系
   - 细化泛化关系

1.为构件确定适当的接口

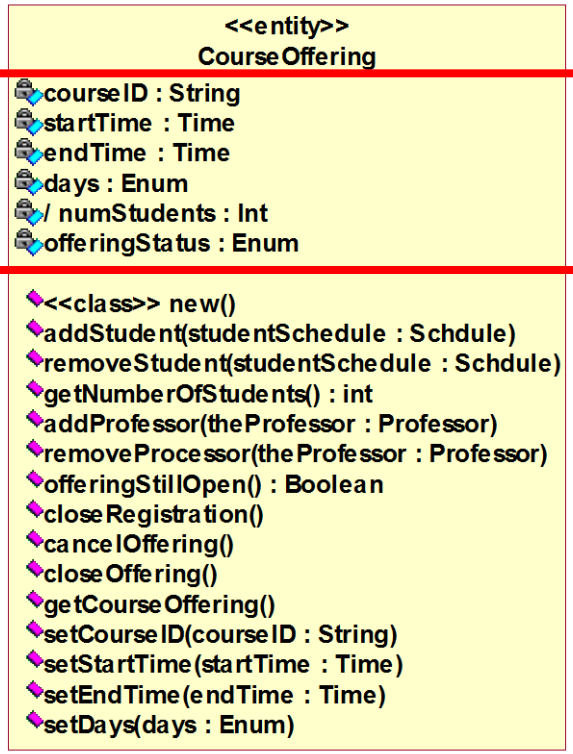
- 简单构件的Public 方法
- 复杂构件单独定义接口类，构件负责实现接口
- 过程化构件的函数
- 确定接口的参数

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。
    - 例如：类CourseOffering中的“学生数目”

      / numStudents : int



2.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。

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”是否被充分利用起来：**
  - 对属性进行CRUD；
  - 对状态进行各种变更；

3.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。**

- **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；

3.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**

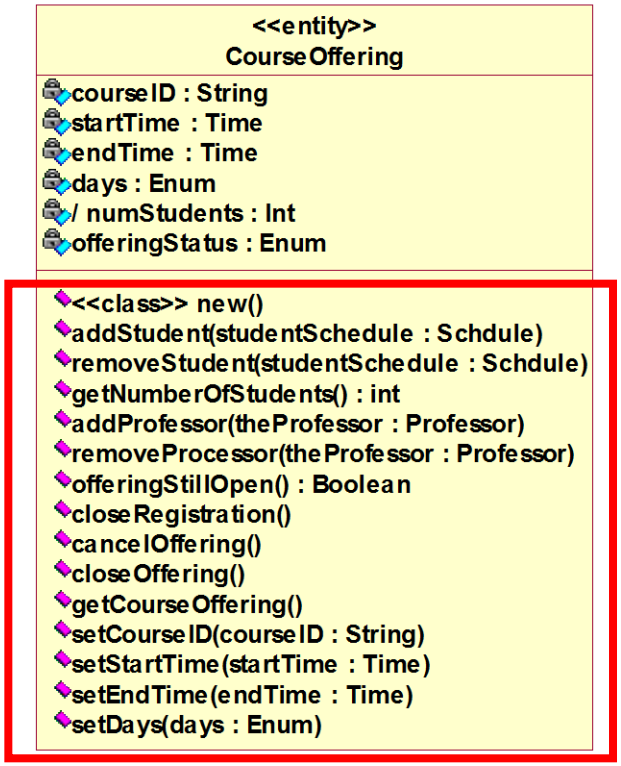
    addProfessor (theProfessor:Professor)

    removeProfessor (theProfessor:Professor)

    offeringStillOpen ( ) : Boolean

    getNumberOfStudents ( ) : int

    ……



3.细化操作

- 一个例子：*BorrowerInfo*类
  - 构造函数

    <<class>> + new ( )
  - 属性赋值

    + setName(name:String)

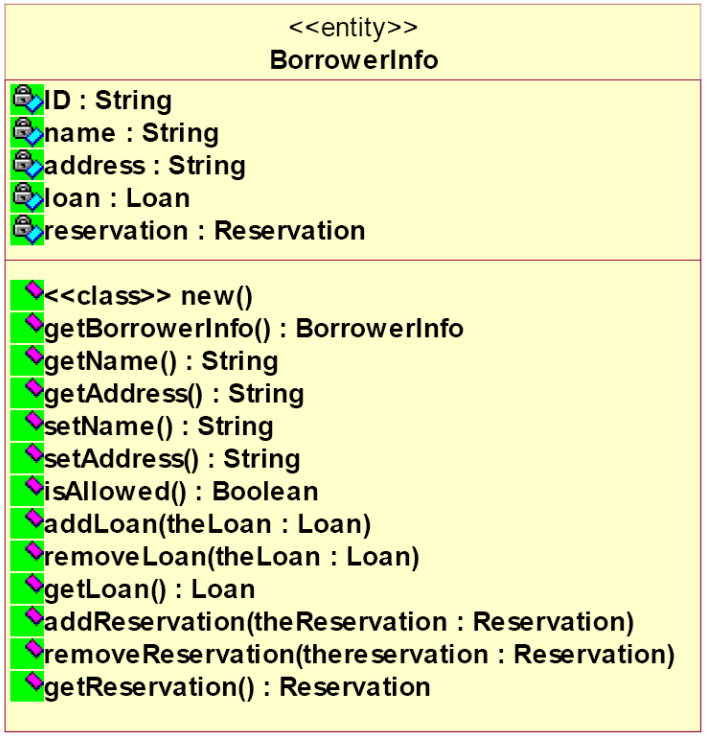
    + setAddress(address:String)
  - 其他

    + addLoan(theLoan:Loan)

    + removeLoan(theLoan:Loan)

    + isAllowed( ) : Boolean

    ……



3.细化操作

new ( )

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

addProfessor (theProfessor : Professor)

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

removeProfessor (theProfessor : Professor)

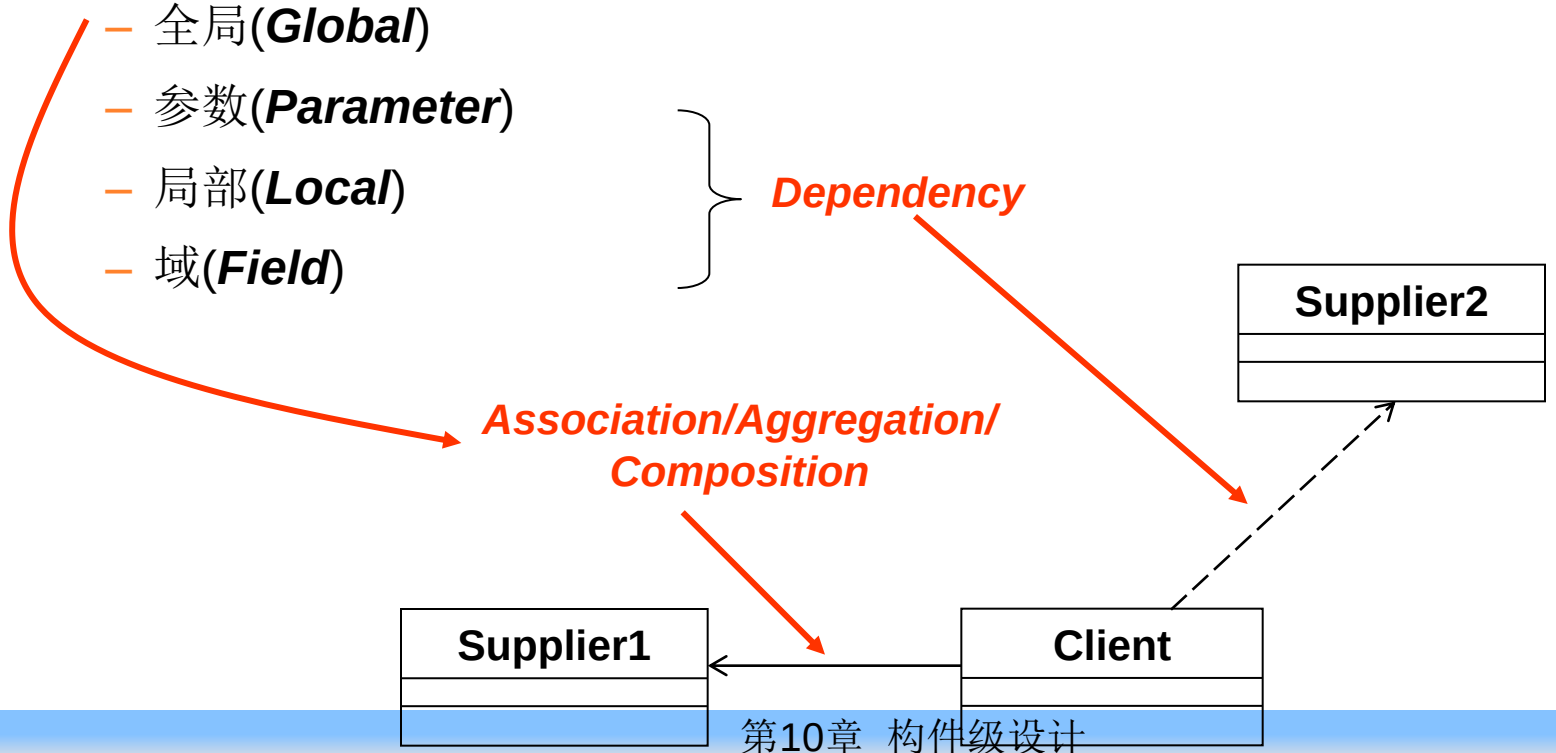
```
if offeringStatus = assigned {
    offeringStatus := unassigned;
    courseInstructor := NULL;
}
else errorState ( );
```

closeOffering ( )

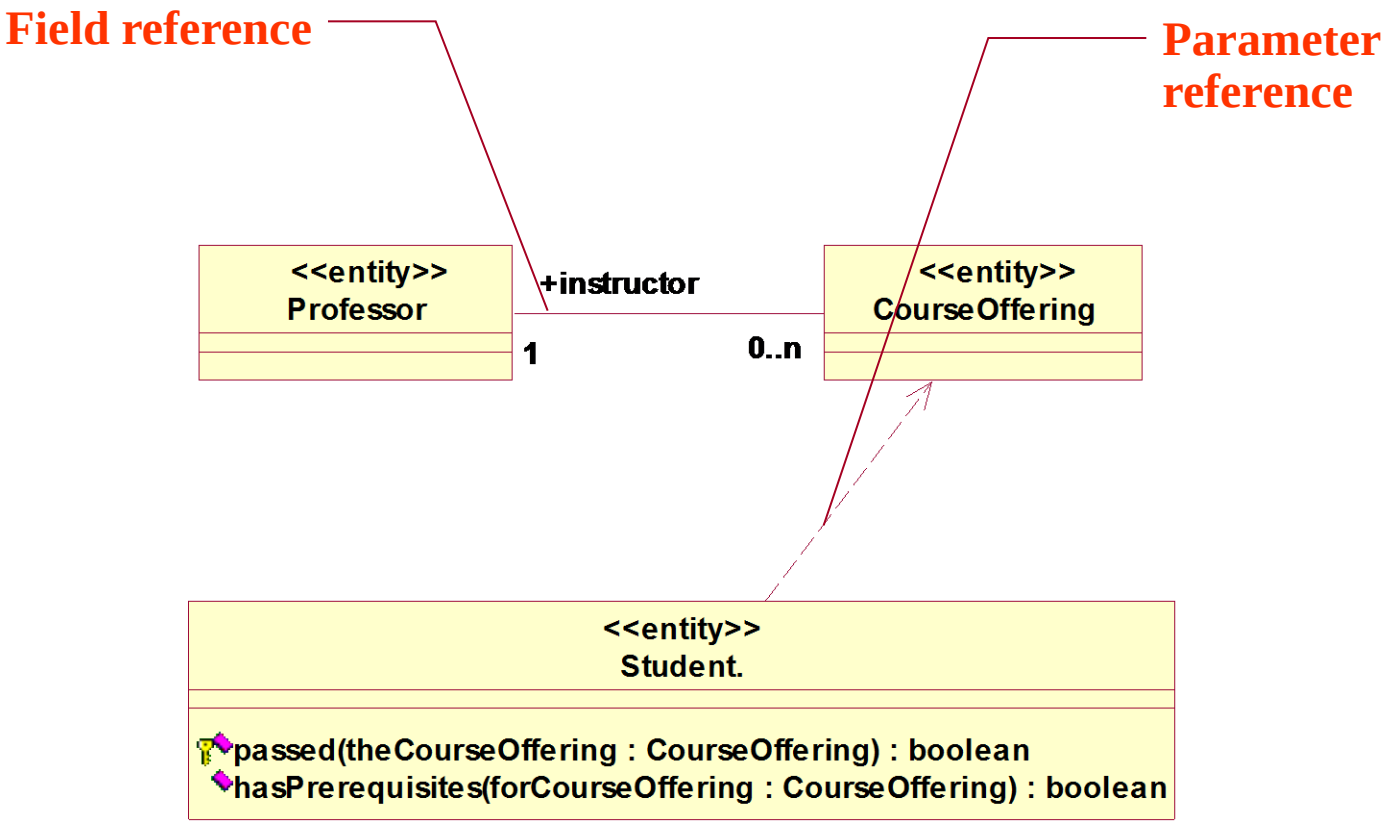
```
switch ( offeringStatus ) {
  case unassigned:
        cancelOffering ( );
        offeringStatus := cancelled;
          break;
  case assigned:
        if ( numStudents <
minStudents )     cancelOffering ( );
           offeringStatus := cancelled;
         else
           offeringStatus :=
committed;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

4.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***)

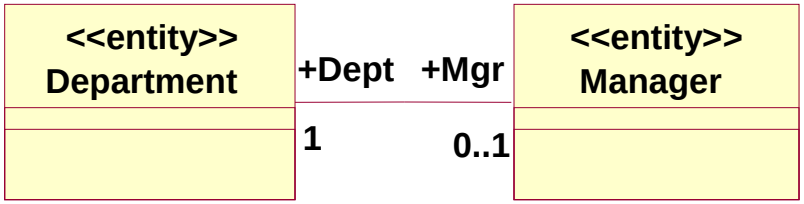


定义类之间的关系：示例


Field reference
Parameter reference
<<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)

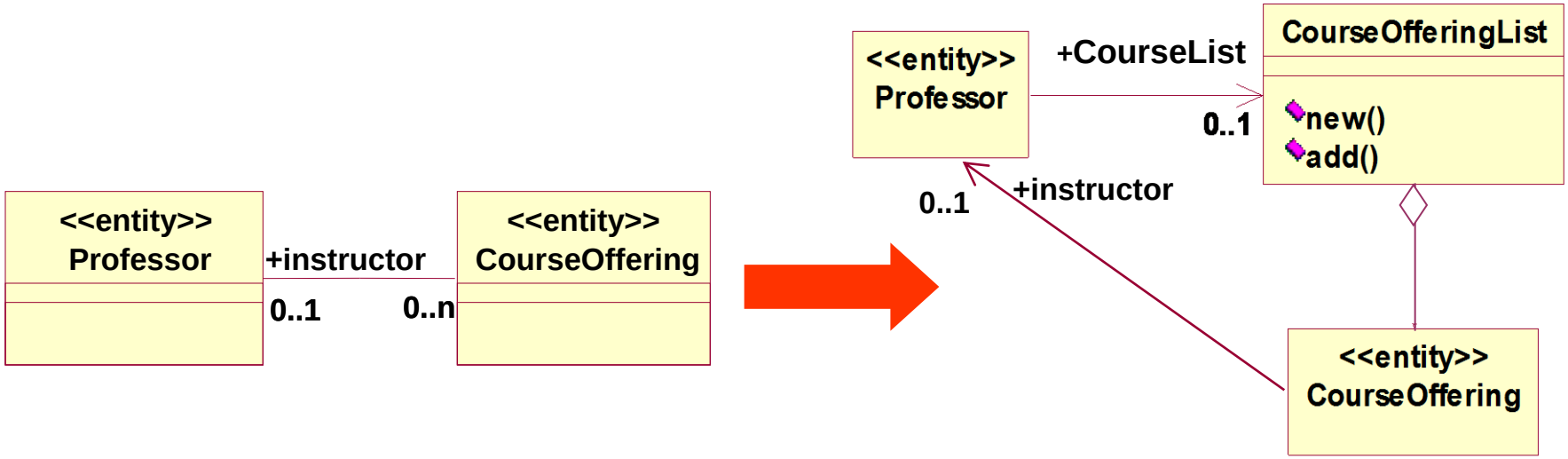
- **根据“多重性”进行设计(multiplicity-oriented design)**
- **情况1：Multiplicity = 1或Multiplicity = 0..1**
  - 可以直接用一个单一属性/指针加以实现，无需再作设计；
  - 若是双向关联：
    - Department类中有一个属性：+Mgr: Manager
    - Manager类中有一个属性：+Dept: Department
  - 若是单向关联：
    - 只在关联关系发出的类中增加关联属性。



需要将这种“关联属性”增加到属性列表中，并更新操作列表

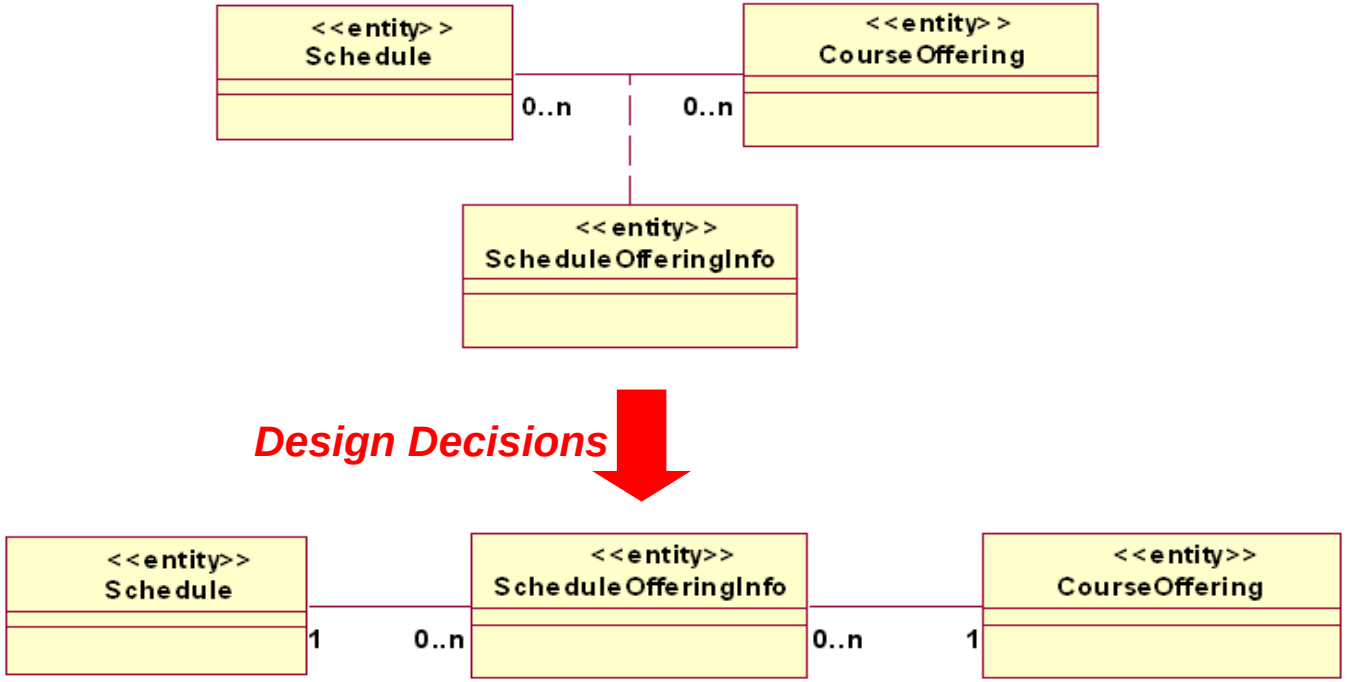
定义关联关系(Association/Composition/Aggregation)

- **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。

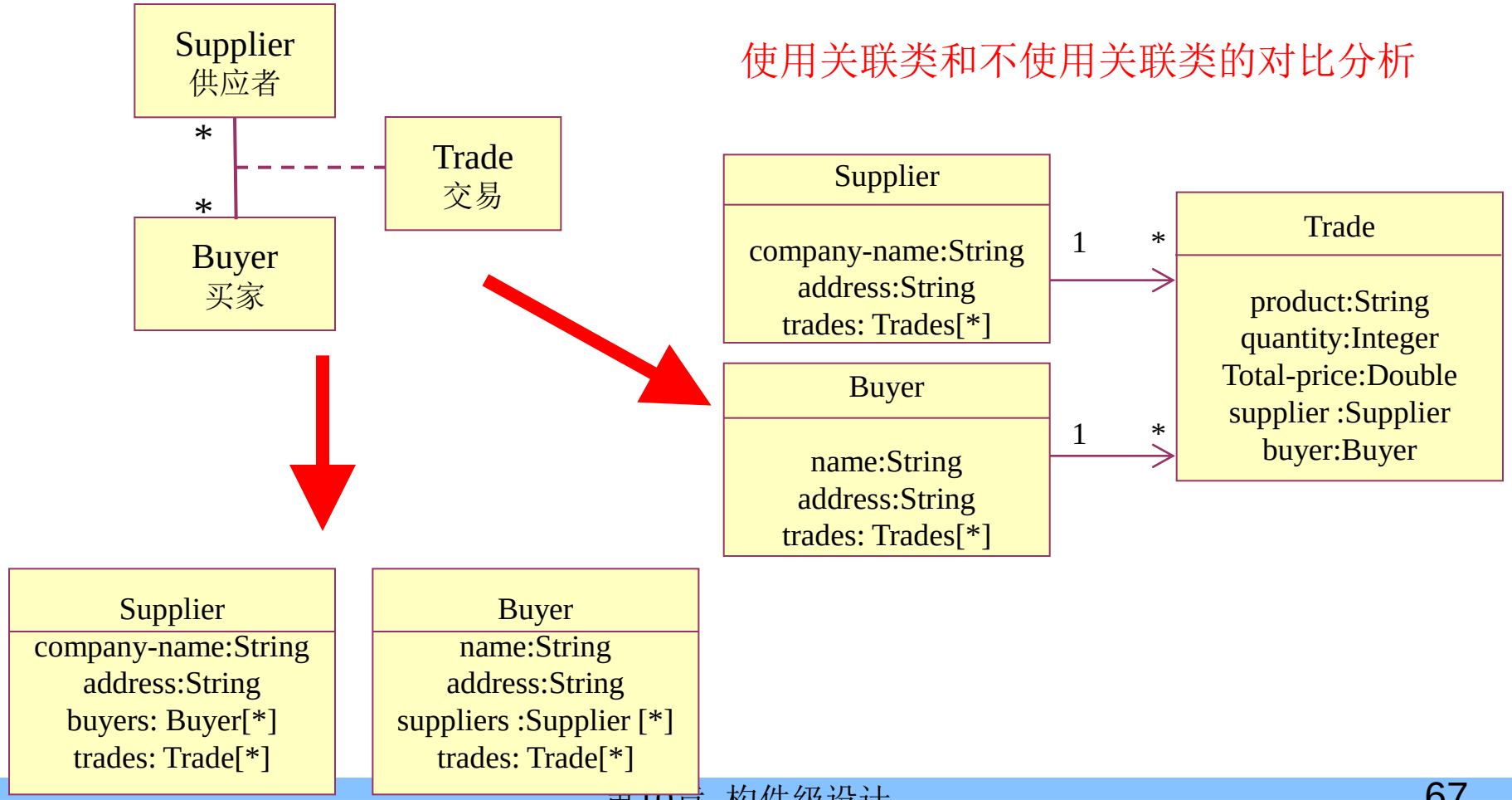


定义关联关系 (Association/Composition/Aggregation)

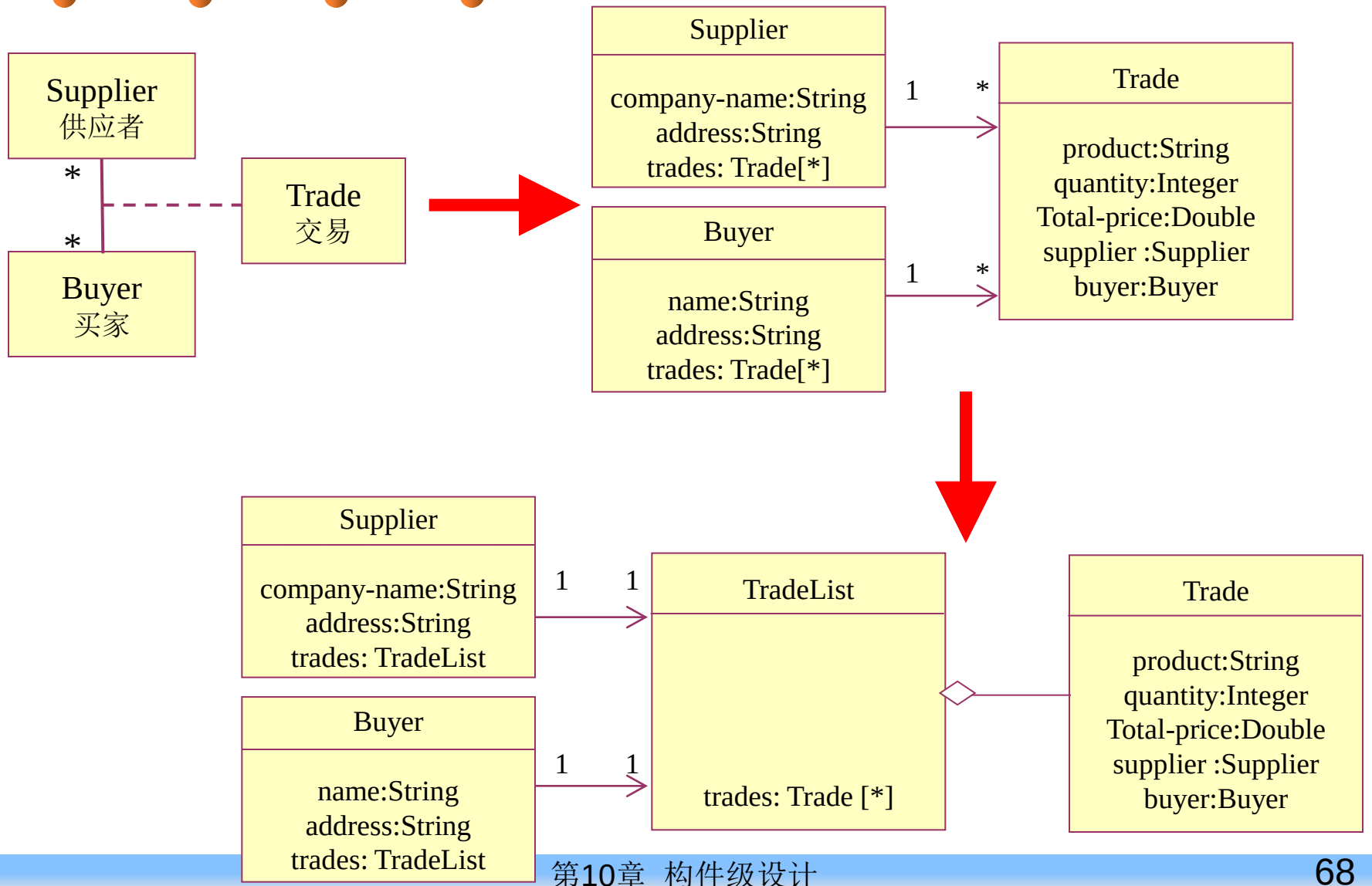
- 情况3：有些情况下，关联关系本身也可能具有属性，可以使用“关联类”将这种关系建模。
- 举例：选课表Schedule与开设课程CourseOffering



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



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



实施构件级设计

- **步骤1：标识所有与问题类相对应的设计类**
- **步骤2：确定所有与基础设施域相对应的设计类**
- **步骤3：细化所有不能作为复用构件的设计类**
  - 步骤3a：在类或构件的协作时说明消息的细节
  - 步骤3b：为每一个构件确定适当的接口
  - 步骤3c：细化属性并且定义相应的数据类型和数据结构
  - 步骤3d：详细描述每个操作的处理流
- **步骤4：说明持久数据源（数据库和文件）并确定管理数据源所需要的类**
- **步骤5：开发并且细化类或构件的行为表示**
- **步骤6：细化部署图以提供额外的实现细节**
- **步骤7：考虑每一个构件级设计表示，并且时刻考虑其他选择**

1 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)

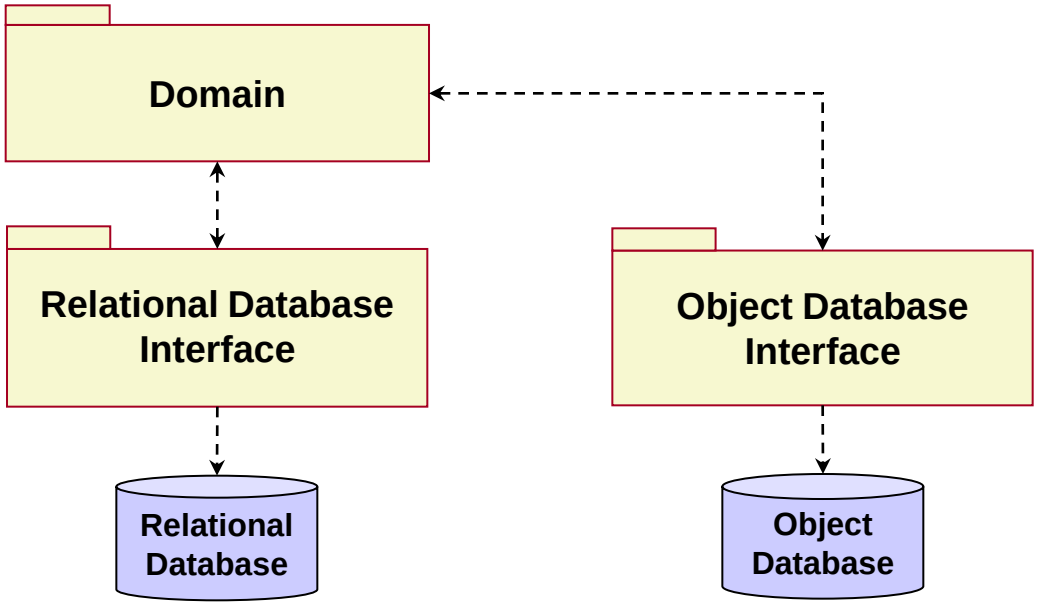
- **数据存储策略**
  - 数据文件：由操作系统提供的存储形式，应用系统将数据按字节顺序存储，并定义如何以及何时检索数据。
  - 关系数据库：数据是以表的形式存储在预先定义好的成为Schema的类型中。
  - 面向对象数据库：将对象和关系作为数据一起存储。
  - 存储策略的选择：取决于非功能性的需求；

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

数据存储策略

- 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。

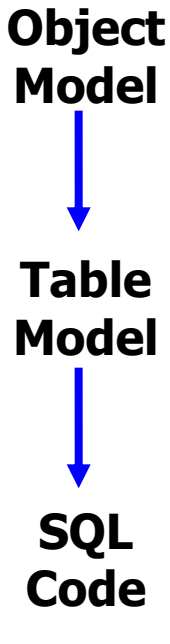


2 OO设计中的数据库设计

- 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)**

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)。



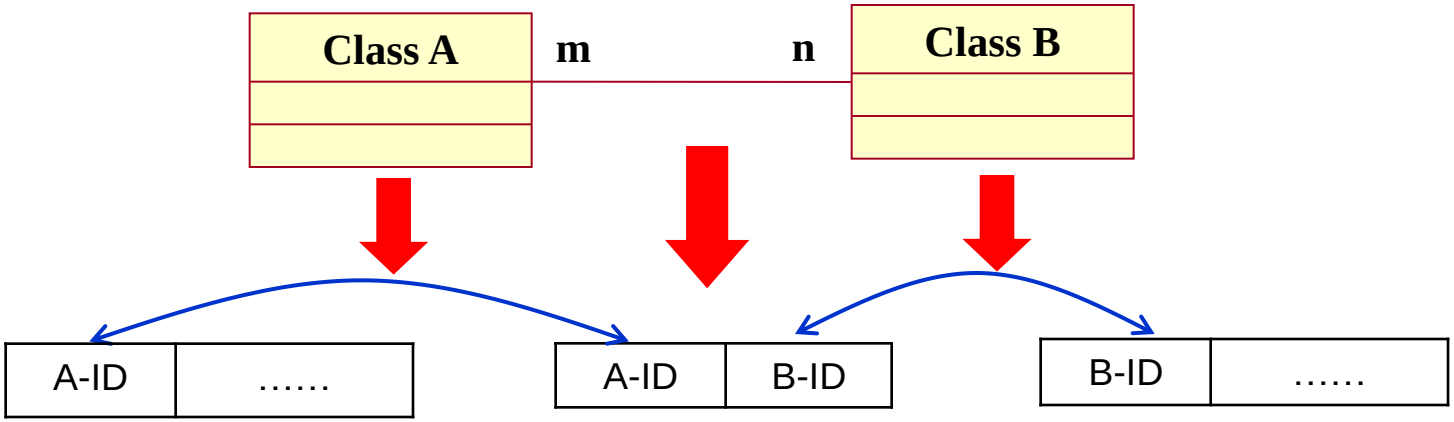
person
Name address

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);
```

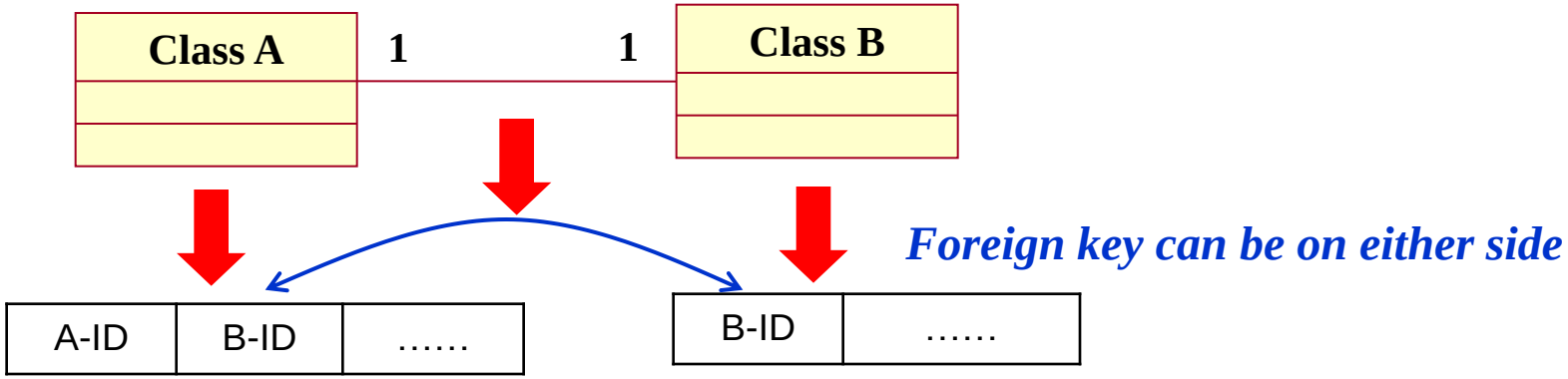
将关联映射到关系数据库(统一的、简单的途径)

- **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；

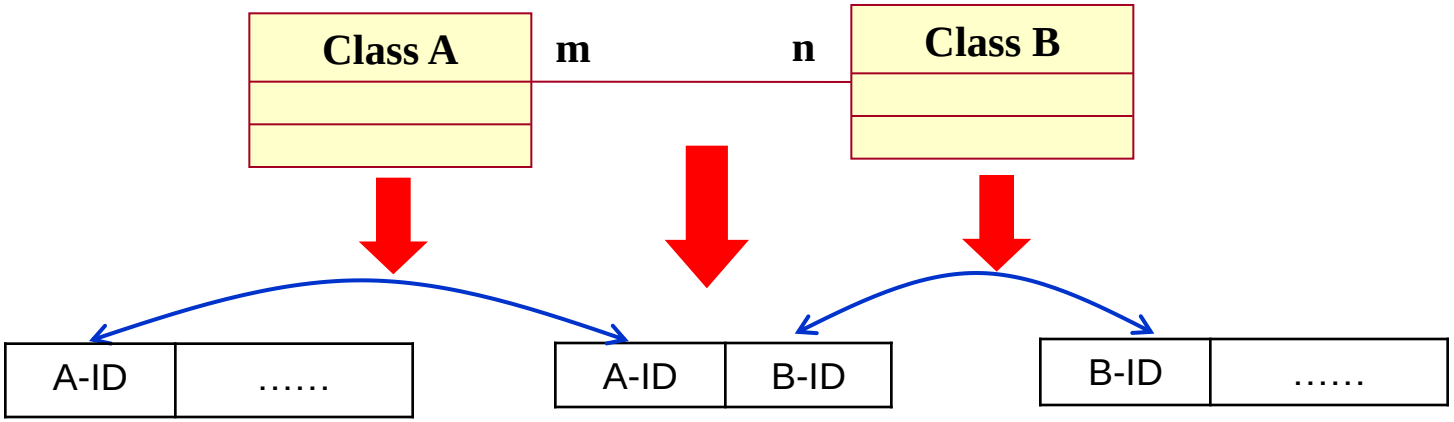


将关联映射到关系数据库(1:1和m:n的关联关系)

- **Implementing 1 to 1**

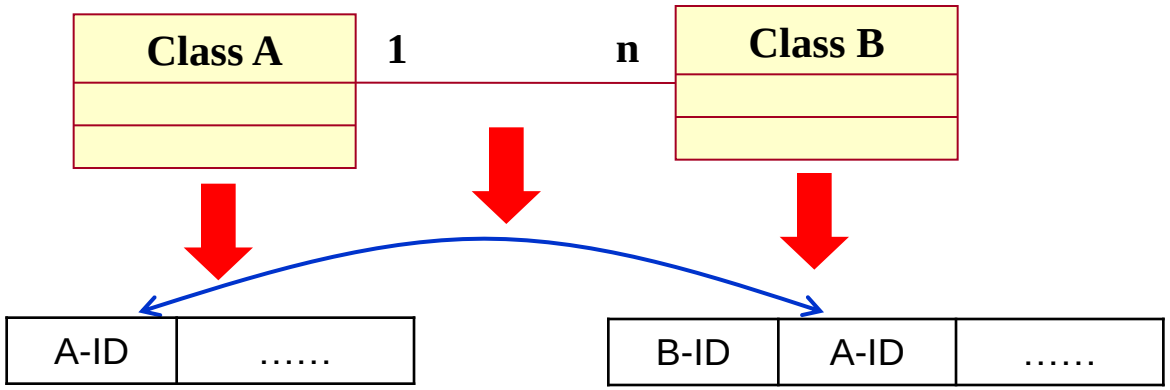


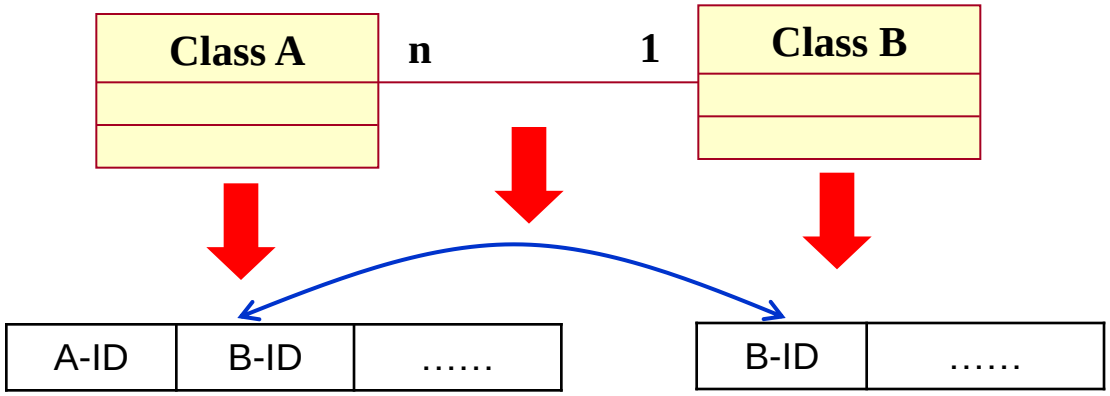
- **Implementing m:n**



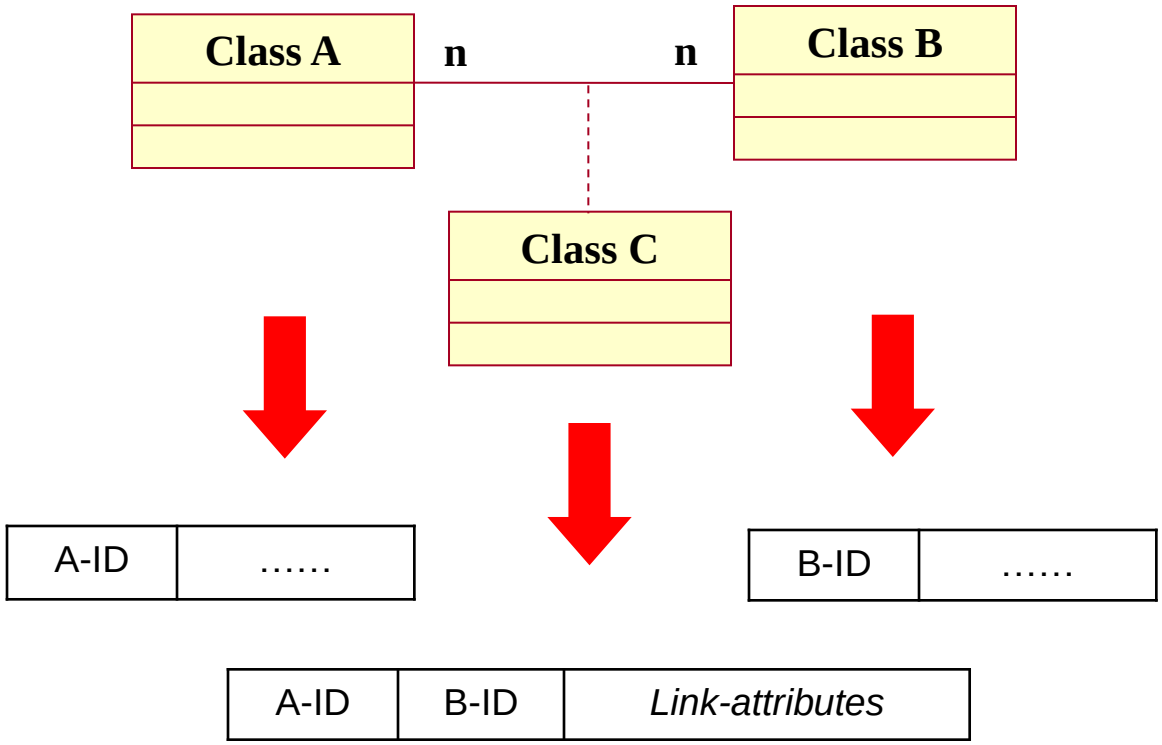
将关联映射到关系数据库(1:n的关联关系)

- **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

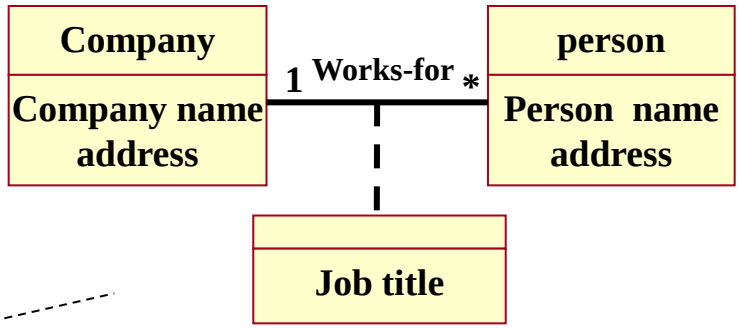
Attribute Name	nulls	Domain
Company_name address	N N	Char Char

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

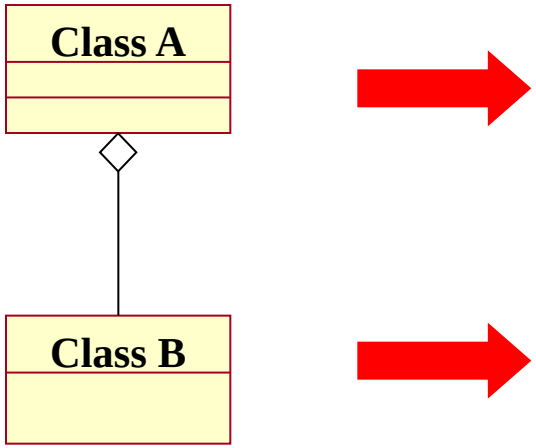
Attribute Name	nulls	Domain
Company_name address	N N	Char Char

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

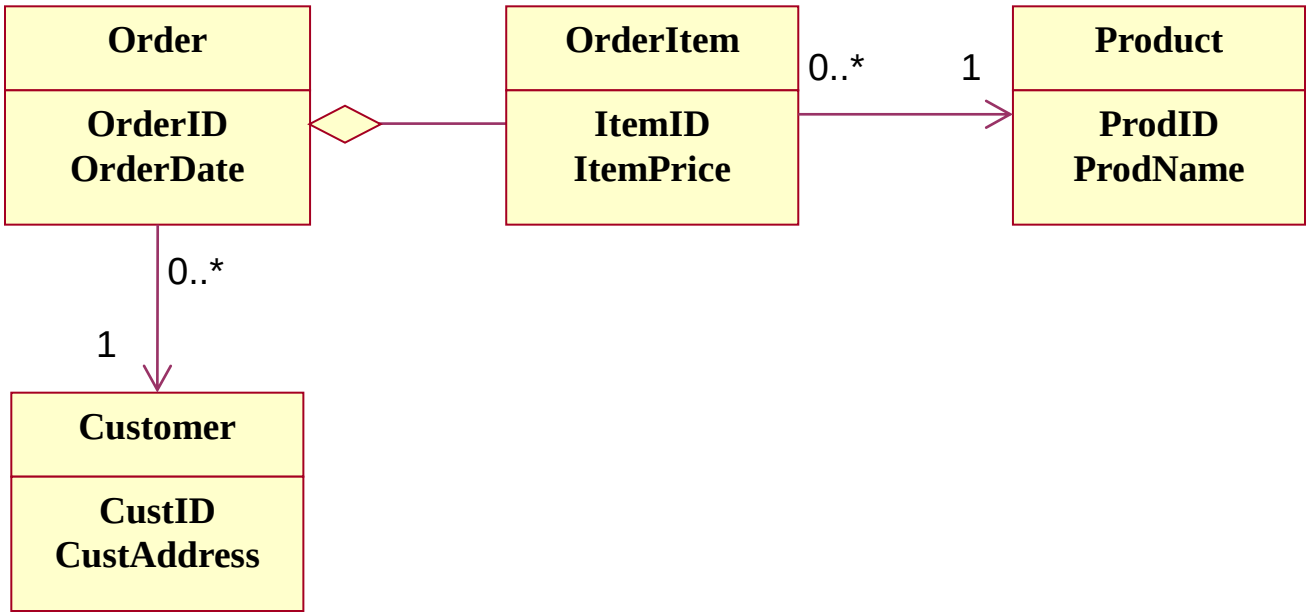
A-ID	X	Y
:	:	:

“部分”表

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

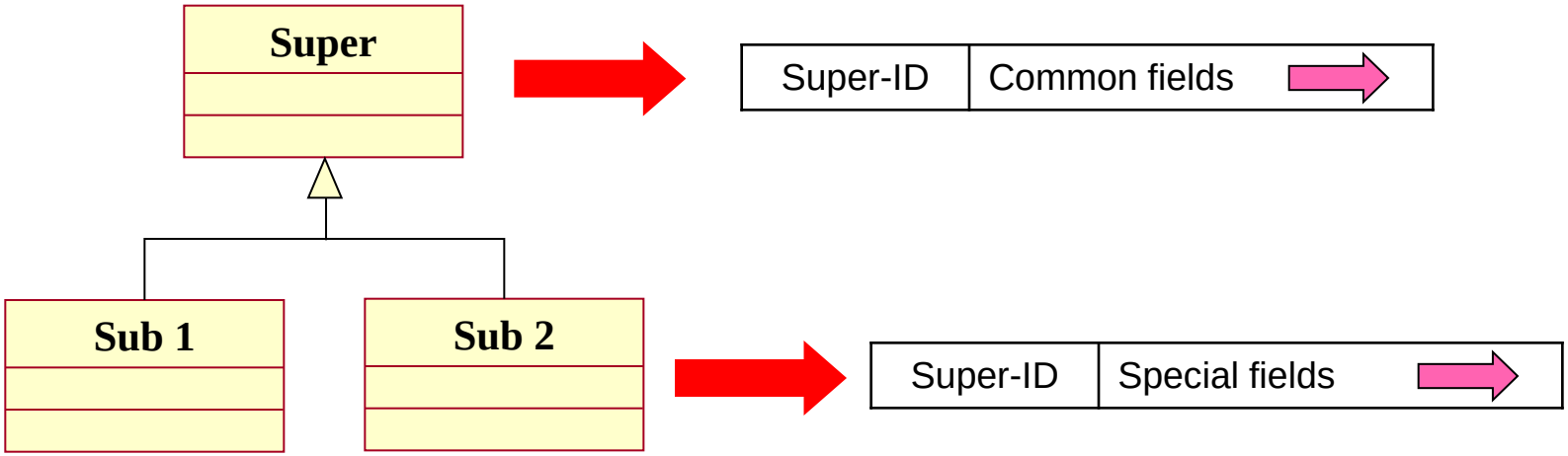
一个简单的课堂练习

- 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

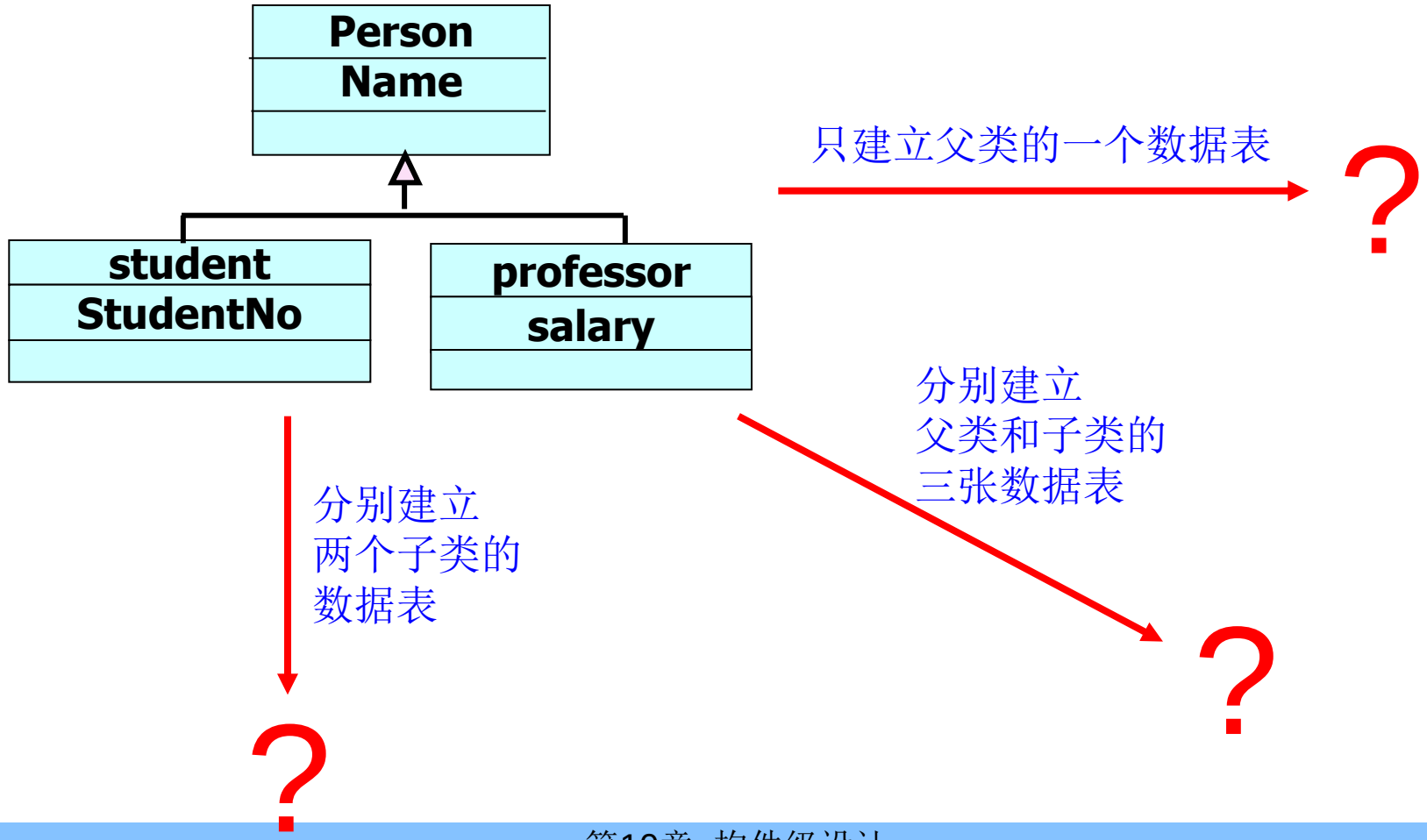
- 策略1：分别建立父类和子类的三张数据表



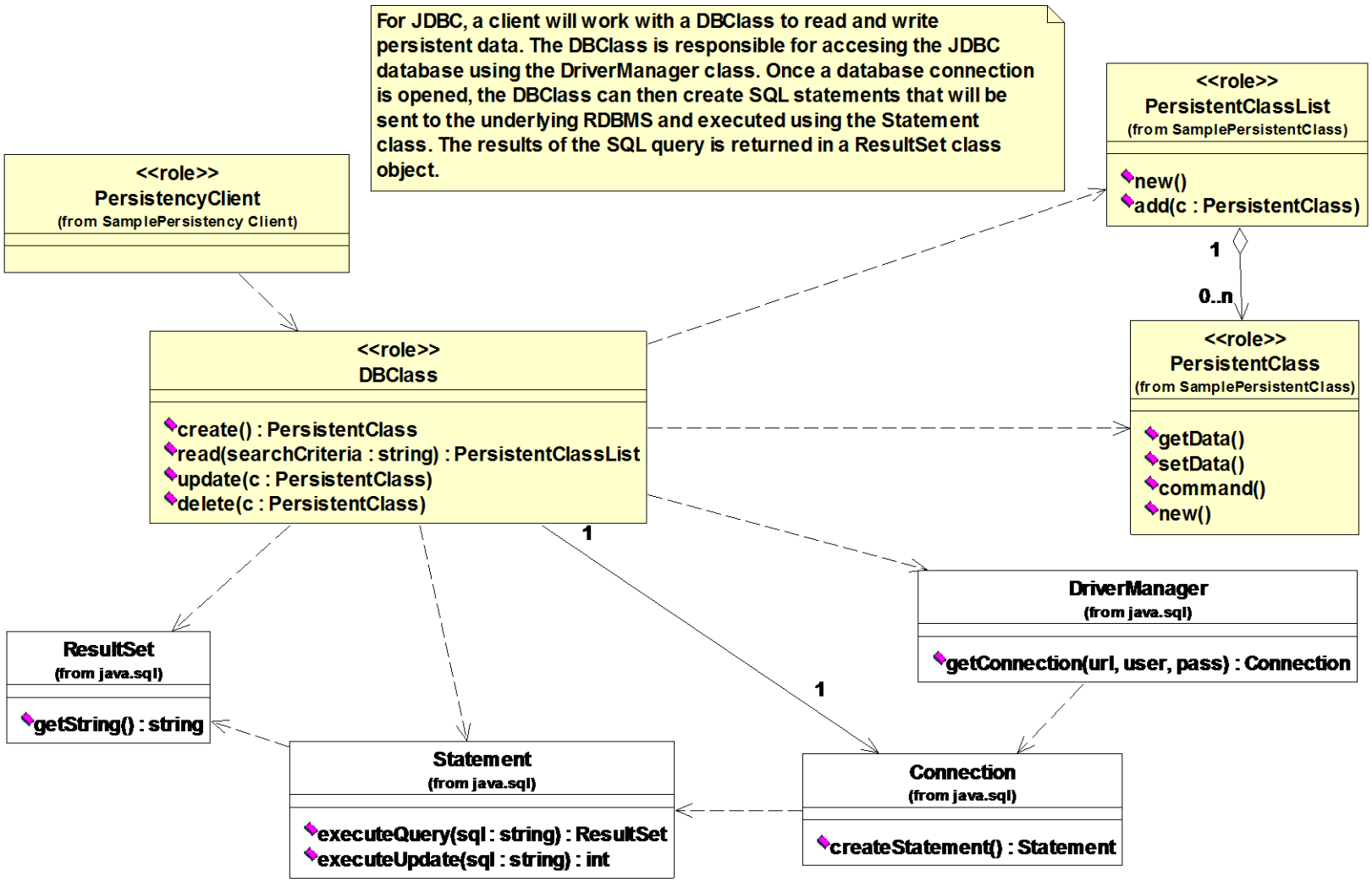
Requires a join to get the object

- 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

将对象映射到关系数据库(继承关系)



Example: Persistency:RDBMS:JDBC



插入语：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)
  - …

实施构件级设计

- **步骤1：标识所有与问题类相对应的设计类**
- **步骤2：确定所有与基础设施域相对应的设计类**
- **步骤3：细化所有不能作为复用构件的设计类**
  - 步骤3a：在类或构件的协作时说明消息的细节
  - 步骤3b：为每一个构件确定适当的接口
  - 步骤3c：细化属性并且定义相应的数据类型和数据结构
  - 步骤3d：详细描述每个操作的处理流
- **步骤4：说明持久数据源（数据库和文件）并确定管理数据源所需要的类**
- **步骤5：开发并且细化类或构件的行为表示**
- **步骤6：细化部署图以提供额外的实现细节**
- **步骤7：考虑每一个构件级设计表示，并且时刻考虑其他选择**

- 算法设计（程序流程图）

实施构件级设计

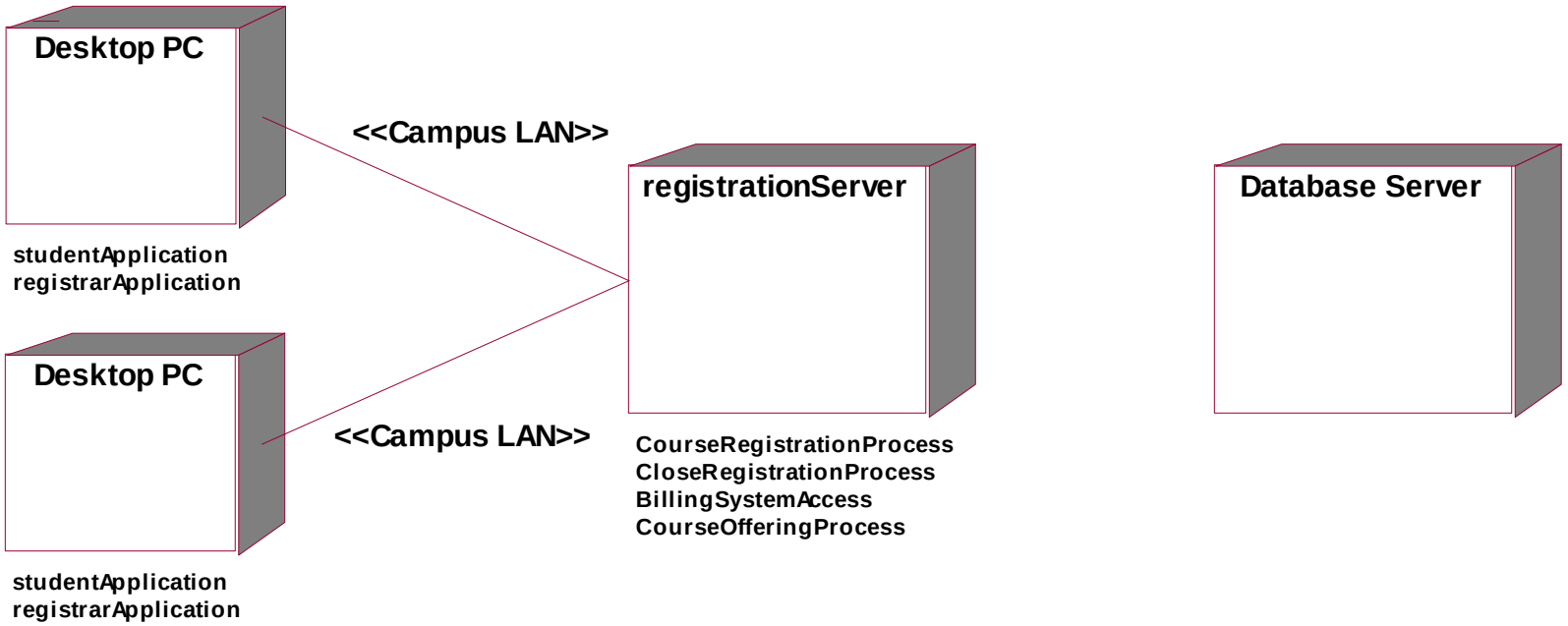
- **步骤1：标识所有与问题类相对应的设计类**
- **步骤2：确定所有与基础设施域相对应的设计类**
- **步骤3：细化所有不能作为复用构件的设计类**
  - 步骤3a：在类或构件的协作时说明消息的细节
  - 步骤3b：为每一个构件确定适当的接口
  - 步骤3c：细化属性并且定义相应的数据类型和数据结构
  - 步骤3d：详细描述每个操作的处理流
- **步骤4：说明持久数据源（数据库和文件）并确定管理数据源所需要的类**
- **步骤5：开发并且细化类或构件的行为表示**
- **步骤6：细化部署图以提供额外的实现细节**
- **步骤7：考虑每一个构件级设计表示，并且时刻考虑其他选择**

1 部署子系统

- **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。**

- **系统通常使用分布式的多台硬件设备，通过UML的部署图(deployment diagram)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。
  - 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。

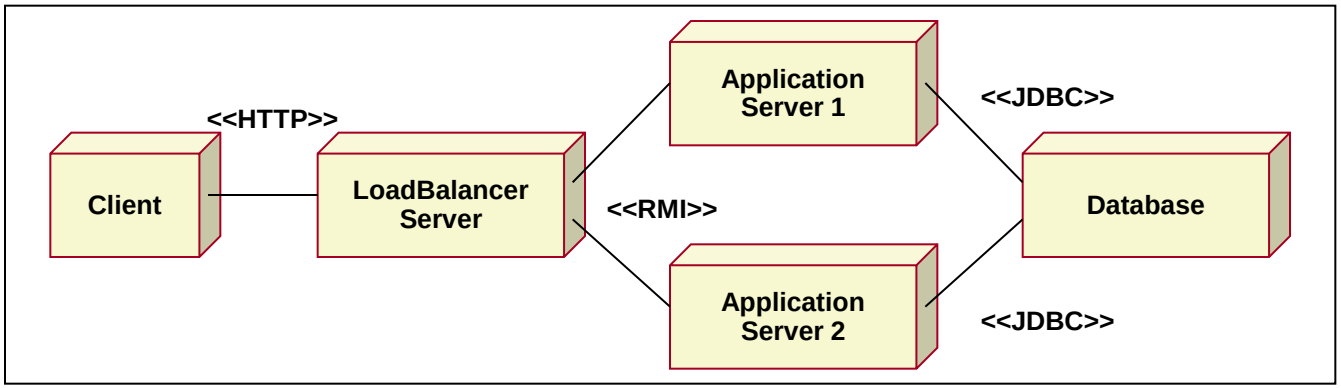
部署子系统

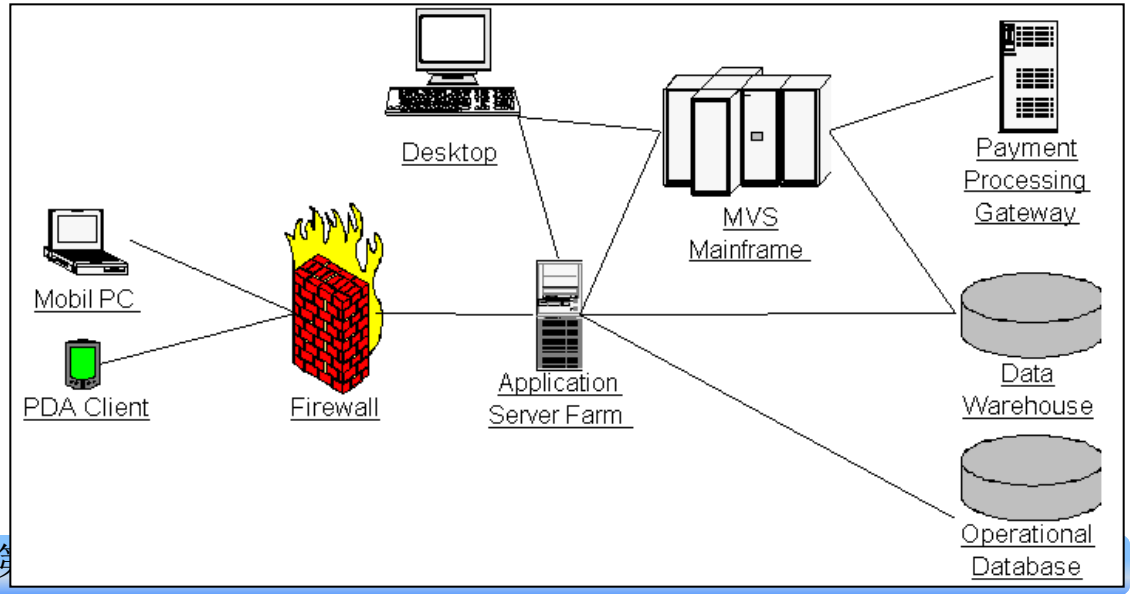

Desktop PC
studentApplication
registrarApplication
<<Campus LAN>>
registrationServer
CourseRegistrationProcess
CloseRegistrationProcess
BillingSystemAccess
CourseOfferingProcess
Database Server
Desktop PC
<<Campus LAN>>
studentApplication
registrarApplication

关于部署图

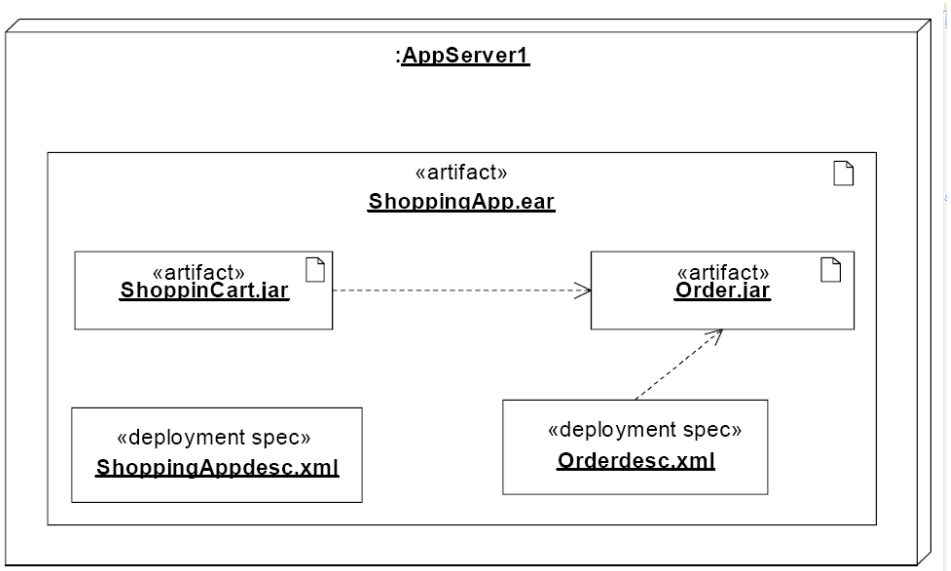
- **部署图(deployment diagram)：**
  - 节点(node)：一组运行资源，如计算机、设备或存储器等。每个节点用一个立方体来表示，
  - 节点的命名：client、Application Server、Database Server、Mainframe等较通用的名字；
  - 节点立方体之间的连接表示这些节点之间的通信关系，通常有以下类型：异步、同步；HTTP、SOAP；JDBC、ODBC；RMI、RPC；等等；
- **部署图在两个层面的作用：**
  - High-level：描述系统中各硬件之间的物理通讯关系；
  - Low-level：描述各软件实体被配置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

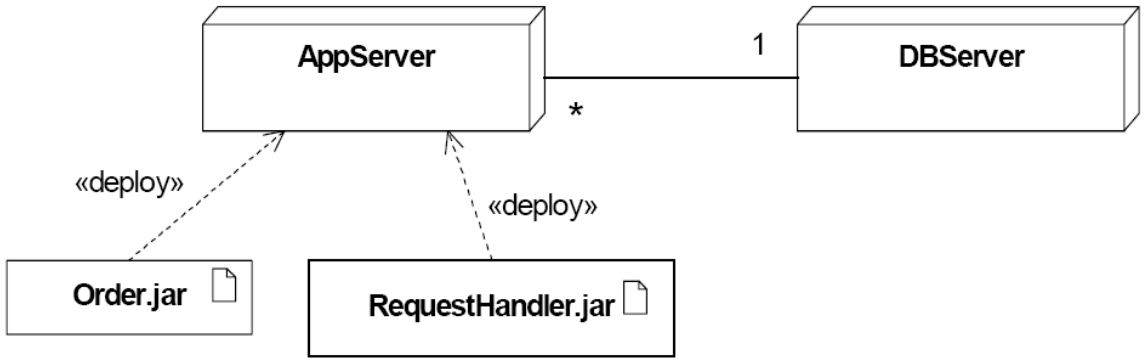
High-level Deployment Diagram





Low-level Deployment Diagram

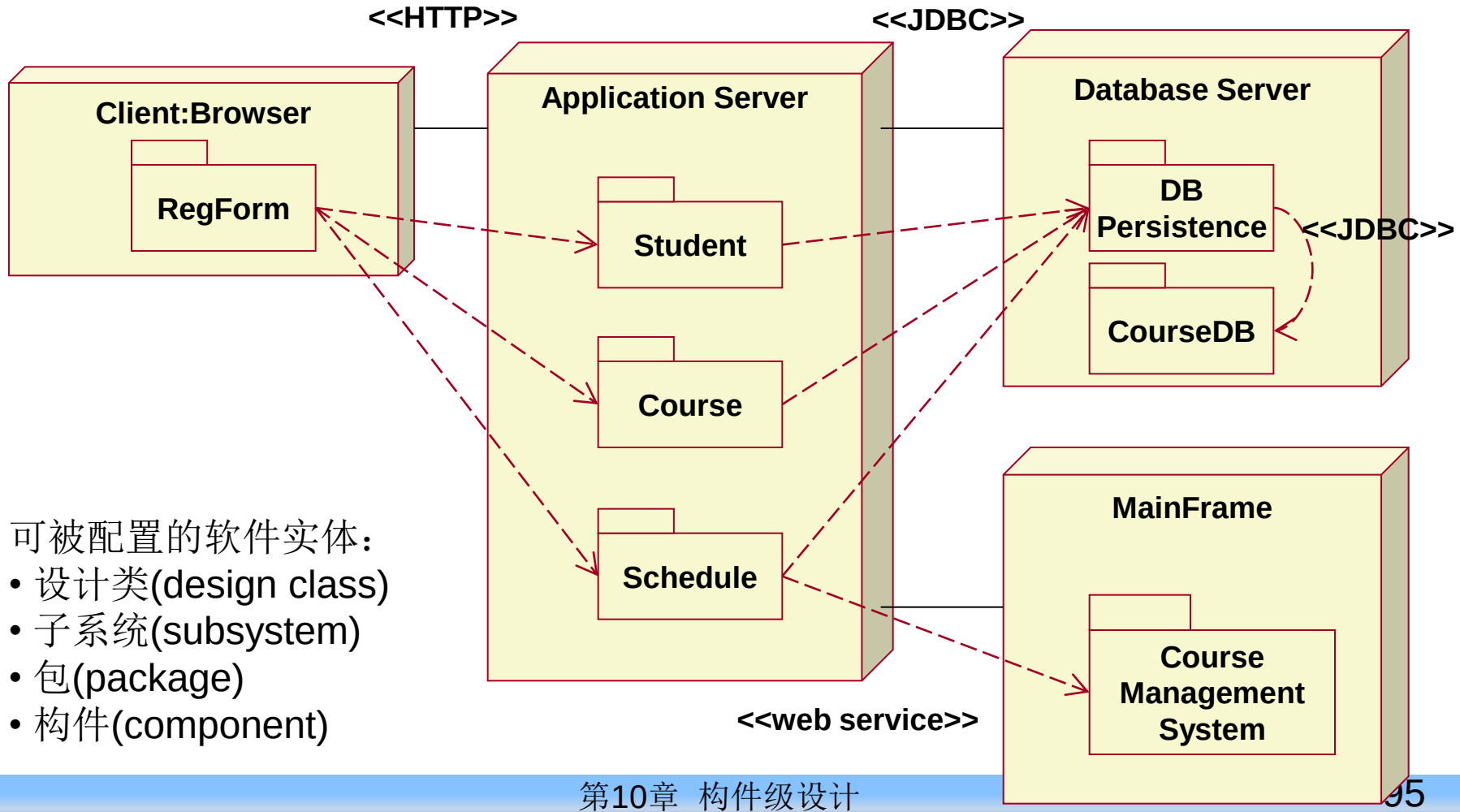




绘制部署图(deployment diagram)

- **确定“节点(node)”：**
  - 标示系统中的硬件设备，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)**

Low-level Deployment Diagram：课程注册系统



可被配置的软件实体：

- 设计类(design class)
- 子系统(subsystem)
- 包(package)
- 构件(component)

实施构件级设计

- **步骤1：标识所有与问题类相对应的设计类**
- **步骤2：确定所有与基础设施域相对应的设计类**
- **步骤3：细化所有不能作为复用构件的设计类**
  - 步骤3a：在类或构件的协作时说明消息的细节
  - 步骤3b：为每一个构件确定适当的接口
  - 步骤3c：细化属性并且定义相应的数据类型和数据结构
  - 步骤3d：详细描述每个操作的处理流
- **步骤4：说明持久数据源（数据库和文件）并确定管理数据源所需要的类**
- **步骤5：开发并且细化类或构件的行为表示**
- **步骤6：细化部署图以提供额外的实现细节**
- **步骤7：考虑每一个构件级设计表示，并且时刻考虑其他选择**

- 迭代的设计过程
- 考虑设计原则
- 应用设计模式

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？

主要内容

- **10.1 设计建模**
- **10.2 构件设计原则**
- **10.3 构件设计步骤**
- **10.4 设计规格说明**

设计规格说明书

- **设计规格说明书(Software Design Description, SDD)**
  - 设计活动的结果，精确的阐述一个软件系统如何提供需求的功能和性能；
  - 为了使设计人员和软件编码人员对该软件的具体设计方案有一个共同的理解，使之成为下阶段开发工作的基础。

SDD应包含的内容

需求分析的产出物

- 范围：描述设计工作的整体范围，其大部分内容来自需求规格说明书(SRS)；

概要设计

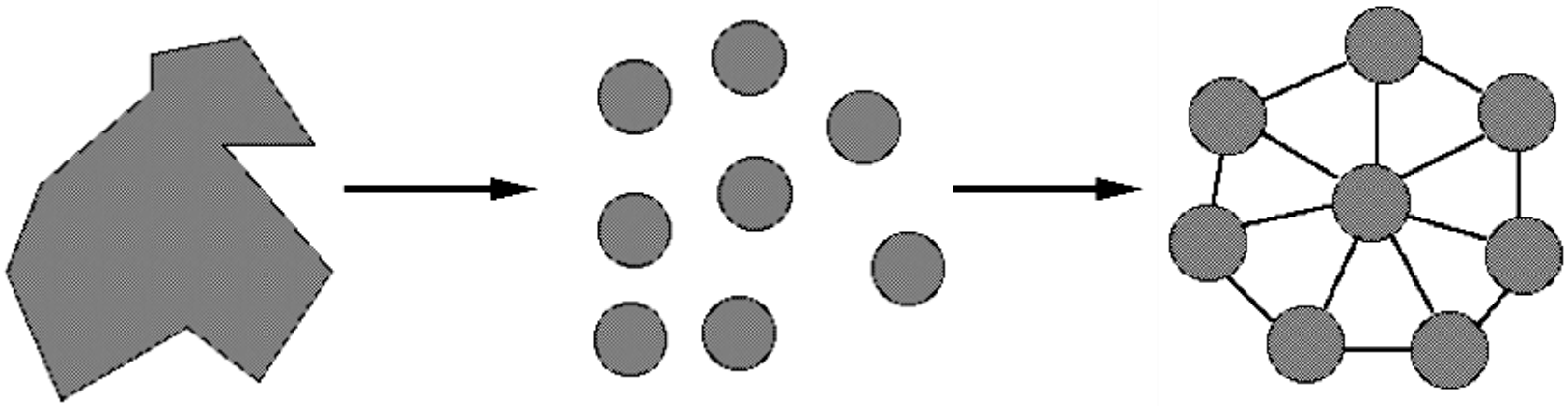
- 体系结构设计：体系结构风格、模块划分、模块之间的依赖关系；
- 接口设计：用户界面设计、外部数据/系统/设备的接口；
- 数据设计：数据结构、外部文件的结构、外部数据库的结构；

详细设计

- 过程设计：每个模块的具体加工流程；
- 其他方面

SDD的基本组织单元：设计元素

- **设计元素(Design entity)：**
  - 软件系统可表达为一系列设计元素的集合；
  - 每个设计元素具有特定的属性；
  - 设计元素之间具有特定的关系；



软件需求　　设计元素　　设计模型

设计元素(design entity)

- **设计元素是软件设计过程中被识别出来的一个基本功能单元，在结构上和功能上能够唯一的区别于其他元素。**
  - 我们学过的结构化设计和OO设计中有哪些典型的设计元素类型？
- **设计元素的种类：**
  - 系统、子系统；
  - 数据存储、模块；
  - 对象/类；
  - 过程、算法；

设计元素的属性(1)

- **针对每一个设计元素，在SDD中应描述其具体的属性：**
  - 唯一标识(identification)
  - 类型(type)
  - 目的(purpose)：该设计元素为何存在？满足SRS中的哪些需求？
  - 功能(function)：该设计元素能完成何种业务？
    - 如果是功能实体，需描述如何实现“输入→输出”的转换；
    - 如果是数据实体，需描述其存储的数据实体的具体格式
  - 子实体(subordinates)：该实体由哪些其他实体构成？
    - SC中的父子关系；
    - 类图中的组合/聚合关系；

设计元素的属性(2)

- **依赖关系(dependencies)：描述该设计实体与其他实体之间的关联关系；**
- **接口(interface)：描述该设计实体如何与其他实体之间进行交互，包括交互的方法、规则与UI**
  - 方法：基于参数传递的调用；消息传递；共享数据；…；
  - 规则：通讯协议、数据格式、数据范围、数据值的含义、出错信息含义、…；
  - 界面：界面的样式与使用方式；

设计元素的属性(3)

- **资源(resources)：描述该设计实体所使用的外部资源**
  - 物理设备：打印机、磁盘阵列、读卡器、磁带机、…
  - 软件设施：操作系统服务、基础功能库、…
  - 处理器资源：CPU时钟、内存分配、缓存、…
  - 应描述：
    - 资源的基本信息；
    - 设计实体与资源进行交互的方法与规则
    - 资源所需的时间长度与周期；
    - 资源所需的数量和大小；
    - 资源短缺或争夺/死锁时的处理措施；

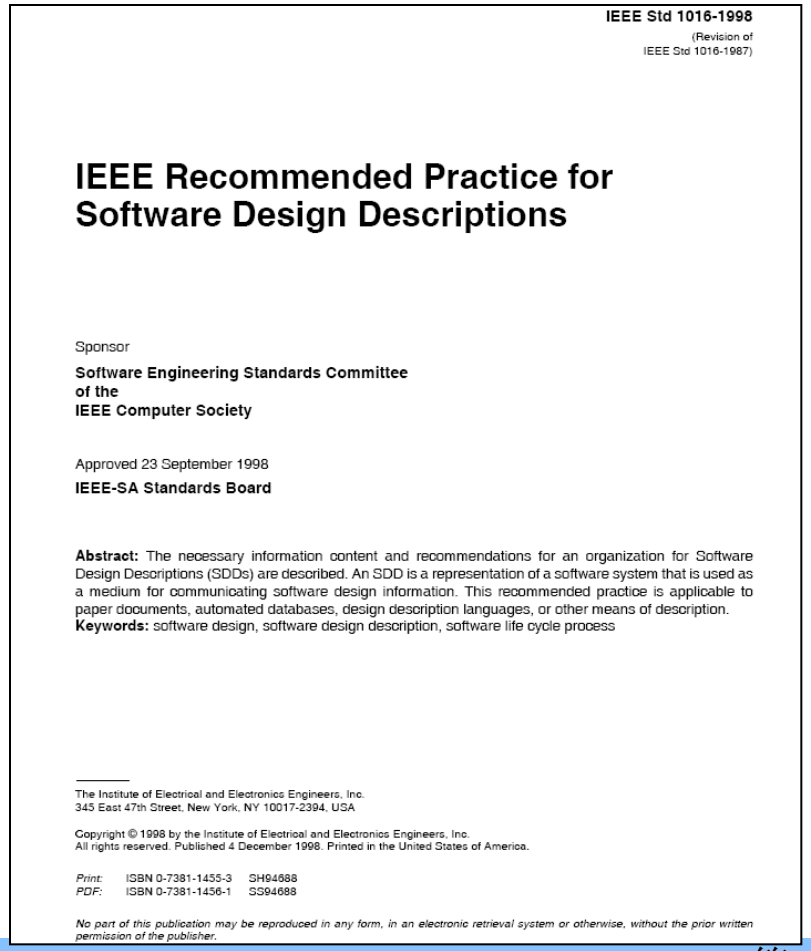
设计元素的属性(4)

- **加工过程(processing)：描述该设计实体为实现其功能而设计的内部加工过程**
  - 算法；程序流程；例外处理流程；
  - 加工过程启动的前置条件；
  - 加工过程终止的后置条件；

设计元素的属性(5)

- **数据(data)：描述该设计实体内部使用的数据元素**
  - 语义；
  - 所采用的数据结构：文件、数组、堆栈、队列、内存、…；
  - 用途：值、控制参数、循环变量、指针、…；
  - 类型：静态数据；动态数据；
  - 初始值；
  - 合法性验证条件；

SDD模板



IEEE Std 1016-1998
(Revision of
IEEE Std 1016-1987)

IEEE Recommended Practice for Software Design Descriptions

Sponsor

Software Engineering Standards Committee of the IEEE Computer Society

Approved 23 September 1998

IEEE-SA Standards Board

Abstract: The necessary information content and recommendations for an organization for Software Design Descriptions (SDDs) are described. An SDD is a representation of a software system that is used as a medium for communicating software design information. This recommended practice is applicable to paper documents, automated databases, design description languages, or other means of description.
Keywords: software design, software design description, software life cycle process

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

Print: ISBN 0-7381-1455-3 SH94688
PDF: ISBN 0-7381-1456-1 SS94688

- **采用IEEE标准1016-1998所给出的模板进行SDD书写：**
- **IEEE Std 1016-1998: IEEE Recommended Practice for Software Design Descriptions**

IEEE COMPUTER SOCIETY

IEEE

SDD的六大组成部分

1. 引言(Introduction)
2. 参考文献(Reference)
3. 分解说明(Decomposition description)
4. 依赖关系说明(Dependency description)
5. 接口说明(Interface description)
6. 详细设计(Detailed design description)

(1-2) 引言与参考文献

1 引言

概要叙述软件设计说明，便于读者理解文档如何编写以及如何阅读和解释；

1.1 目的

1.2 范围

1.3 定义和缩写

2 参考文献

列举编写SDD时所参考的资料、技术标准或其他资源，以方便读者查阅这些文献。

(3-4-5) 概要设计　(6) 详细设计

3 分解说明

软件系统的体系结构风格、模块/子系统划分、软件流程设计、数据设计等；

4 依赖关系说明

定义上述分解之间的依赖关系；

5 接口说明

描述各类接口，如用户界面、模块之间的接口、模块与系统外部环境的接口等。

6 详细设计

用于描述各模块内部的流程与数据的详细设计结果。

(3) 分解说明

3.1 体系结构说明

3.2 模块分解

3.2.1 模块1说明

3.2.2 模块2说明

……

3.3 并发进程

3.3.1 进程1说明

3.3.2 进程2说明

……

3.4 数据分解

3.4.1 数据项1说明

3.4.2 模块项2说明

……

- 系统的整体体系结构风格选择(模块化、分层、事件、B/S、C/S、P2P、等等);
- 说明风格选择的依据;
- 给出图形化的体系结构设计方案(例如SC图);
- 列出模块清单：设计元素。

针对每一个模块，说明其具体的功能，建立与SRS需求项之间追溯关系;

针对每一个数据结构，描述其具体的构成。

(4) 依赖关系说明

4.1 模块间的依赖关系

4.2 进程间的依赖关系

4.3 数据依赖关系

(5) 接口说明

5.1 模块接口

5.1.1 模块1说明

5.1.2 模块2说明

……

5.2 进程接口

5.2.1 进程1说明

5.2.2 进程2说明

……

5.3 外部设备接口

5.4 外部软件系统接口

5.5 用户界面

5.5.1 用户界面1说明

5.5.2 用户界面2说明

……

(6) 详细设计

6.1 模块详细设计

6.1.1 模块1详细设计

6.1.2 模块2详细设计

……

6.2 数据详细设计

6.2.1 数据实体1详细设计

6.2.2 数据实体2详细设计

……

结束

2011年5月18日