南京大学
NANJING UNIVERSITY

面向对象设计方法

张天
软件工程组
ztluck@nju.edu.cn
2025年秋季

提纲

- 面向对象设计概述
- 基于UML的面向对象软件设计
- 案例
- 总结

面向对象设计概述

- 软件设计的复杂性
  - “软件的复杂性是固有的，软件可能是人类所能制造出来的最复杂的实体”。
    - --图灵奖获得者、著名的计算机专家，被称为IBM360系列计算机之父的F.Brooks
  - 这种固有的复杂性使得开发成员之间的通讯变得困难，开发费用超支，开发时间延期，导致产品有缺陷，不易理解，不可靠，难以使用，功能难以扩充等。
    - ——软件危机
  - 无法采用某种方法彻底消除软件的复杂性，因此软件危机只能是通过控制复杂性的方法解决。

面向对象设计概述

- 控制软件复杂性的手段
  - 结构化设计方法
    - 功能分解、自顶向下，逐步求精
      - 对复杂系统采用“各个击破”（分治，divide and conquer）
    - 抽象：抽取系统中的基本特征而忽略非基本特征的部分
    - 模块化：高内聚，低耦合
    - 信息隐藏

面向对象设计概述

- 结构化设计方法存在的问题
  - 功能与数据分离设计
    - 对现实世界的认识与程序实现之间存在理解上的鸿沟
    - 设计难以维护：
      - 多处、多层的功能与数据维护的一致性
  - 自顶向下功能分解方法
    - 极大地限制了软件的可重用性，导致大量重复性工作

基于UML的面向对象设计

- 运用面向对象概念来构造系统设计模型
- 在建造一个复杂系统时，开发者必须从多种不同的角度来抽象系统，使用准确的符号来构造模型，然后检查这些模型是否符合系统的需求，并逐步添加细节，从而将这些模型转化成实现方案
- 从面向对象分析模型构造设计模型，建立与分析模型、可执行体之间的对应关系
- 为软件项目涉众提供交流的文档
- 建模语言是面向对象建模中的一个非常关键的因素
  - UML
    - 标准的统一建模语言
    - 支持面向对象

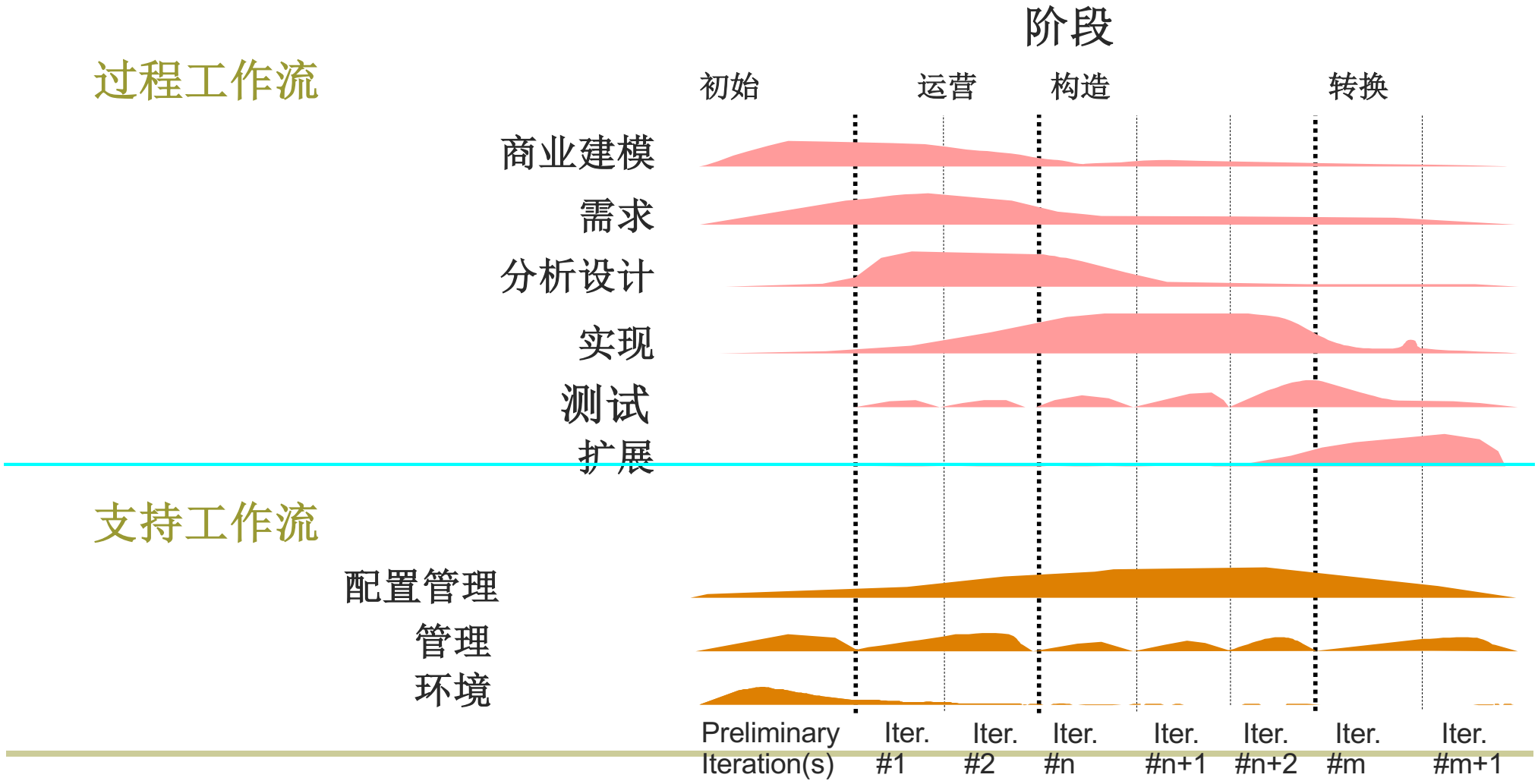
面向对象设计概述

面向对象设计遵循分解，抽象，模块化，信息隐蔽等设计原则

- 类
- 对象:类是对象的类型，对象是类的实例
- 封装和信息隐藏：将属性和操作包装成一个单元，使得对状态的访问和修改只能通过封装提供的接口进行。
- 消息:对象间发送请求的载体
- 继承
- 多态性和动态连编
- 对象之间的联系
  - 分类结构：一般与特殊的关系
  - 组成结构：部分与整体的关系
  - 实例连接：对象之间的静态联系
  - 消息连接：对象之间的通信联系

面向对象设计概述

面向对象设计——迭代式增量



面向对象设计概述

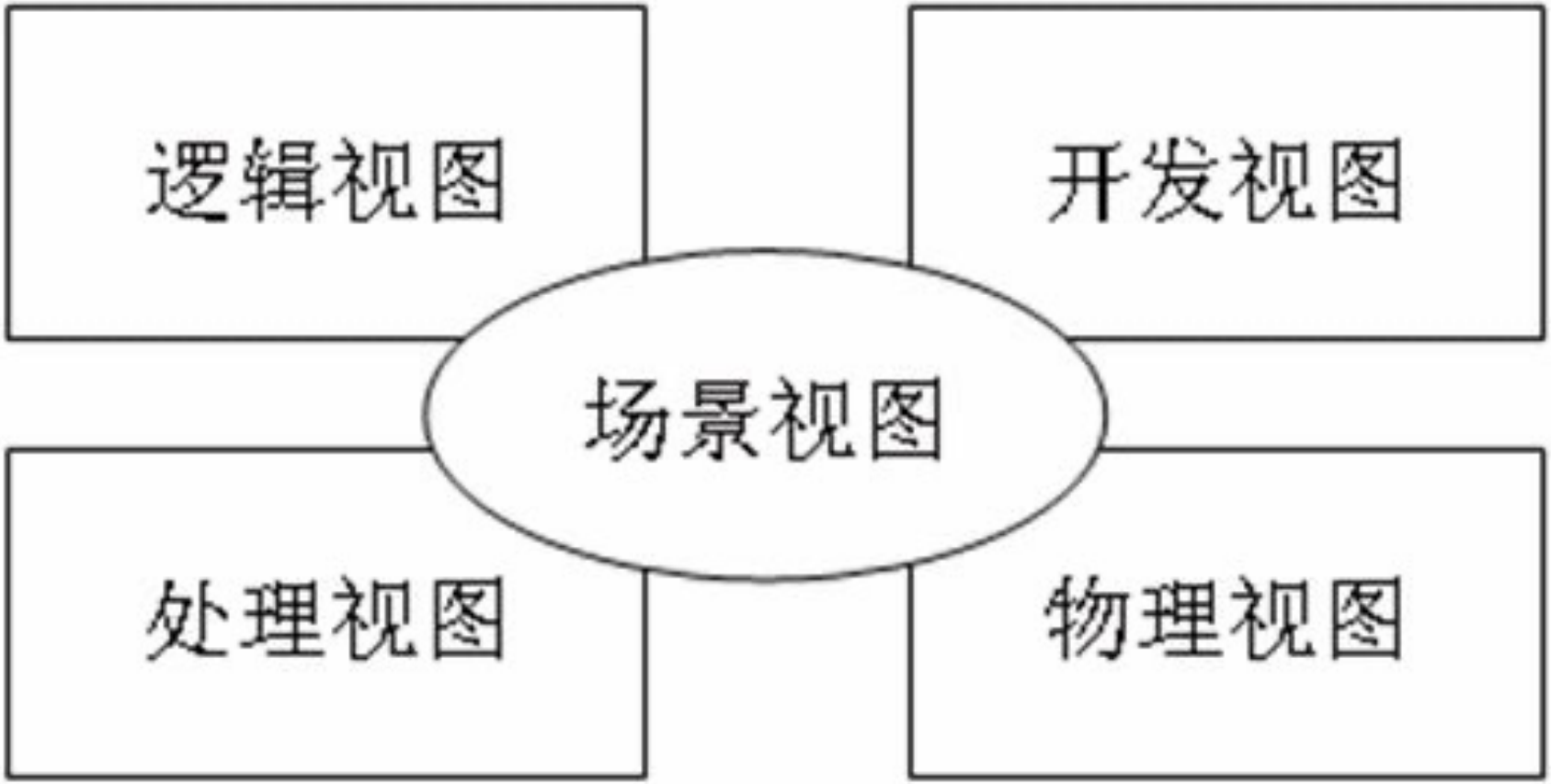
- 面向对象设计的前提：面向对象分析阶段的制品
  - 分析模型：通过分析和理解问题域，获取描述问题域和系统责任所需的类及对象，并表达它们的内部构成和外部关系，以及系统的任务等。
  - 设计任务：
    - 通过设计将**OOA** 模型精化成**OOD** 模型，并且补充与一些实现有关的部分，如人机界面、数据存储、任务管理等。
  - 设计方法
    - 传统面向对象设计：
      - BOOCH方法、OMT方法、OOSE方法
    - 基于UML的面向对象设计

基于UML的面向对象设计

描述系统的视角：

- 系统的使用实例：从系统外部的操作者的角度描述系统的功能。
- 系统的逻辑结构：描述系统内部的静态结构和动态行为，即从内部描述如何设计实现系统功能。
- 系统的构成：描述系统由哪些程序构件所组成。
- 系统的并发性：描述系统的并发性，强调并发系统中存在的各种通信和同步问题。
- 系统的配置：描述系统的软件和各种硬件设备之间的配置关系。

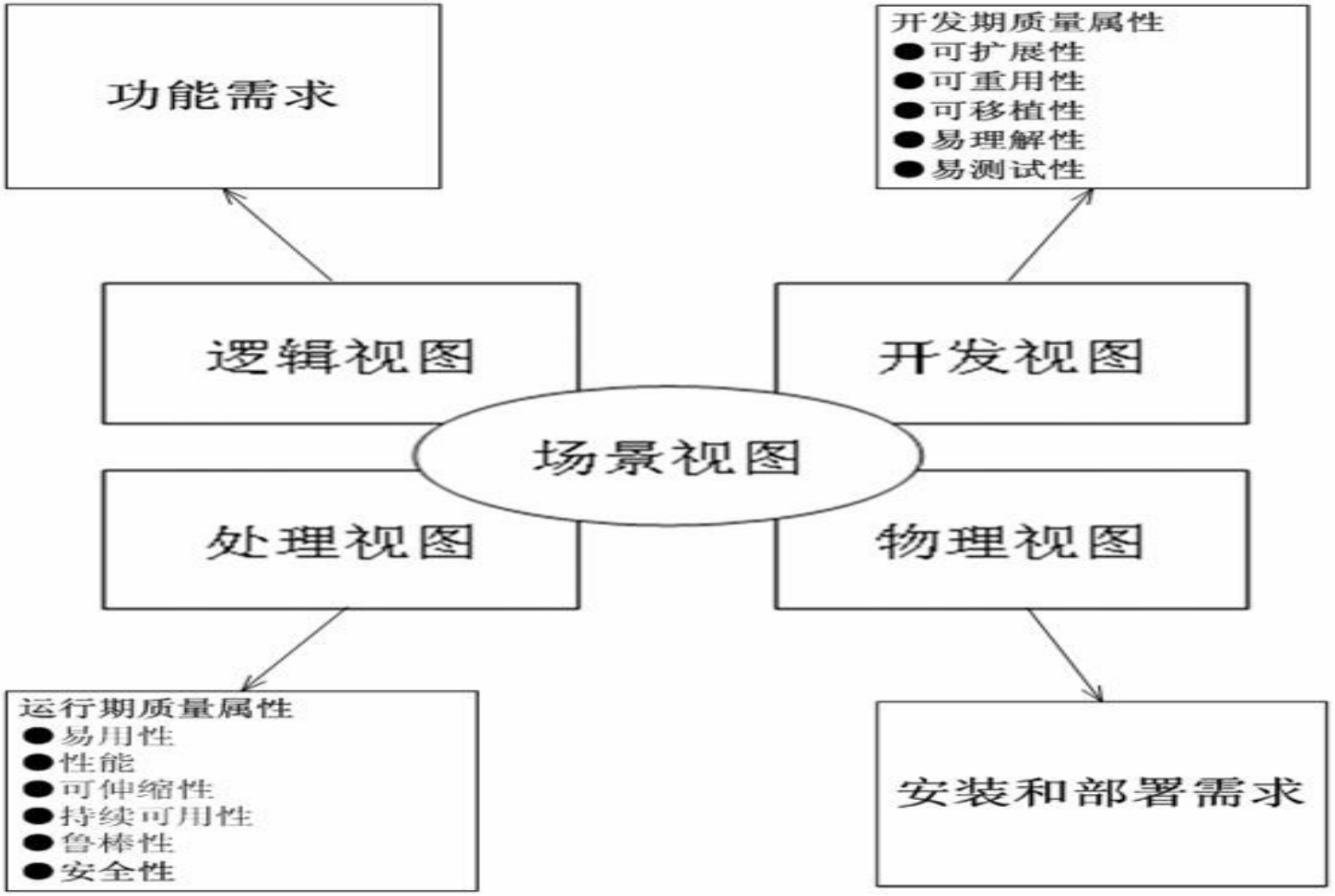
4+1视图方法



UML建模

- 不同架构视图承载不同的系统分析与设计决策，支持不同的目标和用途：
  - 逻辑视图：当采用面向对象的方法时，逻辑视图即分析与设计模型。
  - 开发视图：描述软件在开发环境下的静态组织。
  - 处理视图：描述系统的并发和同步方面的设计。
  - 物理视图：描述软件如何映射到硬件，反映系统在分布方面的设计。

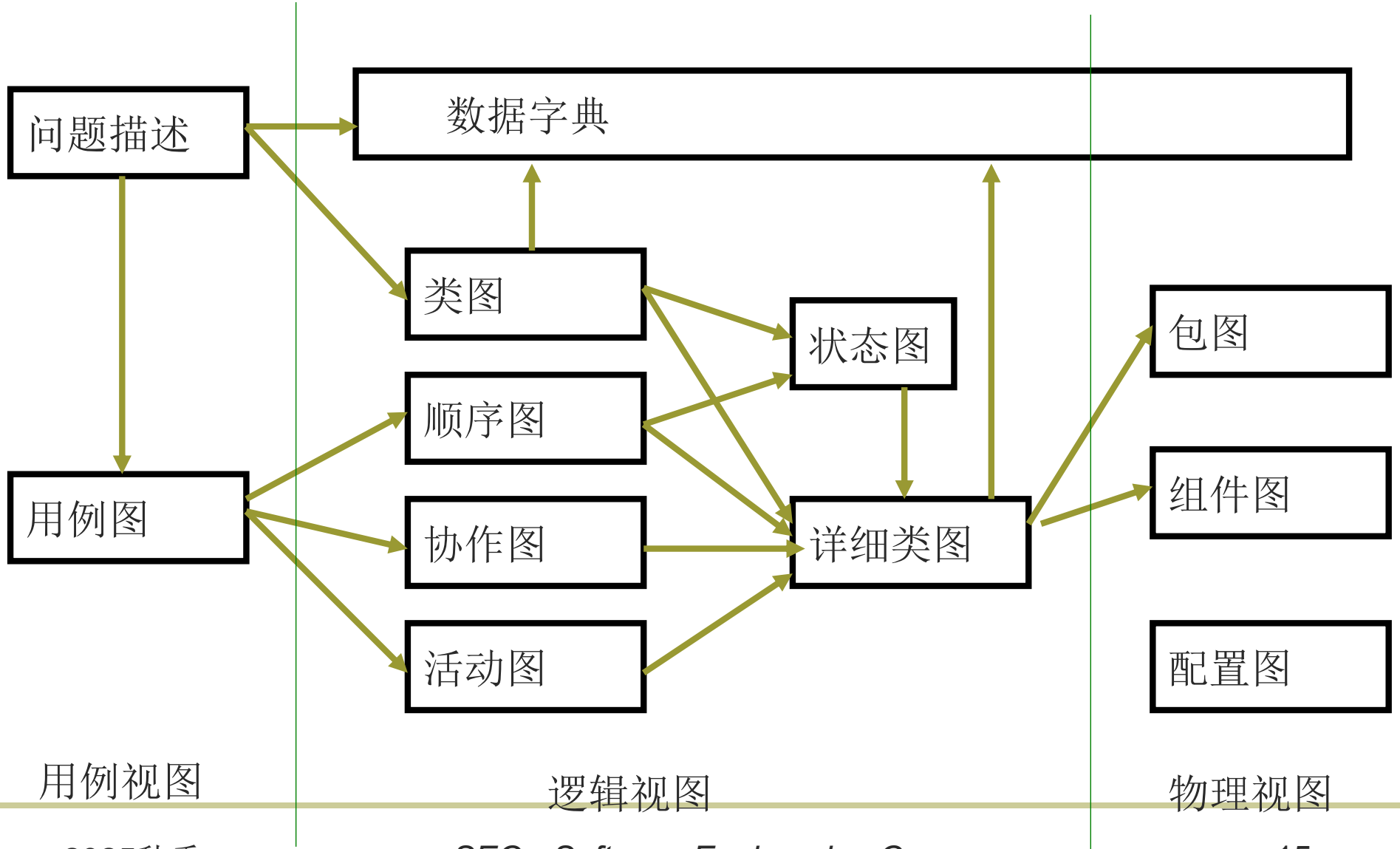
基于UML的面向对象设计



基于UML的面向对象设计

- 逻辑视图：设计满足功能需求的架构
  - 关注功能：用户可见的功能，为实现用户功能而必须提供的“辅助功能模块”；可能是逻辑层、功能模块等。
- 开发视图：设计满足开发期质量属性的架构
  - 关注程序包，不仅包括要编写的源程序，还包括可以直接使用的第三方SDK和现成框架、类库，以及开发的系统将运行于其上的系统软件或中间件。
  - 开发视图和逻辑视图之间可能存在一定的映射关系：比如逻辑层一般会映射到多个程序包等。
- 处理视图：设计满足运行期质量属性的架构
  - 关注进程、线程、对象等运行时概念，以及相关的并发、同步、通信等问题。
  - 处理视图和开发视图的关系：开发视图一般偏重程序包在编译时期的静态依赖关系，而这些程序运行起来之后会表现为对象、线程、进程，处理视图比较关注的正是这些运行时单元的交互问题。
- 物理视图：和部署相关的架构决策
  - 关注"目标程序及其依赖的运行库和系统软件"最终如何安装或部署到物理机器，以及如何部署机器和网络来配合软件系统的可靠性、可伸缩性等要求。
  - 物理视图和处理视图的关系：处理视图特别关注目标程序的动态执行情况，而物理视图重视目标程序的静态位置问题；物理视图是综合考虑软件系统和整个IT系统相互影响的架构视图。

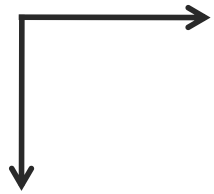
基于UML的面向对象设计



面向对象分析阶段的建模

- 功能
  - 用例图
- 结构
  - 类图
  - 对象图
  - 包图
  - 复合结构图
- 行为
  - 状态机图
  - 顺序图
  - 活动图
  - 通讯图
  - 交互概观图
- 实现
  - 构件图
  - 部署图
- 时间
  - 定时图

介绍顺序：

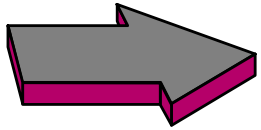


基于UML的面向对象设计

OOA

Develop model of requirements

User's Perspective



OOD

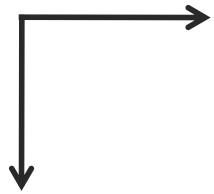
Add detail and design decisions

Developer's Perspective

面向对象设计阶段的建模

- 功能
  - 用例图
- 结构
  - 类图
  - 对象图
  - 包图
  - 复合结构图
- 行为
  - 状态机图
  - 顺序图
  - 活动图
  - 通讯图
  - 交互概观图
- 实现
  - 构件图
  - 部署图
- 时间
  - 定时图

介绍顺序：



基于UML建模的面向对象设计

- 设计原则
  - 模块化
    - 层次分解
  - 耦合
    - 两个类（子系统）之间的关联程度
  - 内聚
    - 类（子系统）内部的相关程度
  - 复用
    - 体系结构
    - 模式

基于UML建模的面向对象设计

- 设计过程
  - 确定分析—》设计映射
  - 构造结构模型：对象驱动
    - 精化类设计，设计类图：类＋关系
    - 进行包组织，设计包图：类图＋关系
  - 构造行为模型：场景驱动
    - 对复杂类设计状态机图
    - 对重要事件流，设计细化顺序图
    - 对任务流程或类方法实现，设计活动图
  - 构造体系结构模型：
    - 组件图：组件选择、组件间通信
    - 配置图
      - 子系统部署

- 设计类
  - 考虑与实现有关的因素，具体描述操作的参数、属性和类型等。
  - 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，则应该将其映射成“子系统接口”。通常，被动参与者对应的边界类被映射成子系统接口。
  - 类的设计应当充分利用预定义的系统类库或其他来源的现有类，并采用继承、复用、演化等方法设计所需要的新类。

基于UML建模的面向对象设计

- 类的精化设计（Refinement）
  - 边界类的设计策略
    - 用户界面设计因素
      - 用户界面的开发工具
      - 所创建的界面数量
    - 外部系统接口类
  - 实体类的设计策略
    - 考虑性能需求对实体对象的影响
  - 控制类的设计策略
    - 是否真正需要？是否应该继续细分？
    - 考虑复杂性、变化适应性、分布性和性能、事务处理等要求

基于UML建模的面向对象设计

- 类的精化设计
  - 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

基于UML建模的面向对象设计

- 类的精化设计
  - 定义操作
    - 找出满足基本逻辑要求的操作
    - 补充必要的辅助操作
      - 初始化
      - 验证两个实例是否等同
      - ……
    - 完整地描述操作
      - 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
  - 简要说明操作的内部实现逻辑
  - 为复杂的操作实现逻辑构造活动图
    - 每个活动必须扩展成一个和多个操作，每个操作被指派给特定的对象来实现。

基于UML建模的面向对象设计

- 类的精化设计
  - 定义属性
    - 具体说明属性的名称、类型、缺省值、可见性等
  - 基本原则
    - 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。

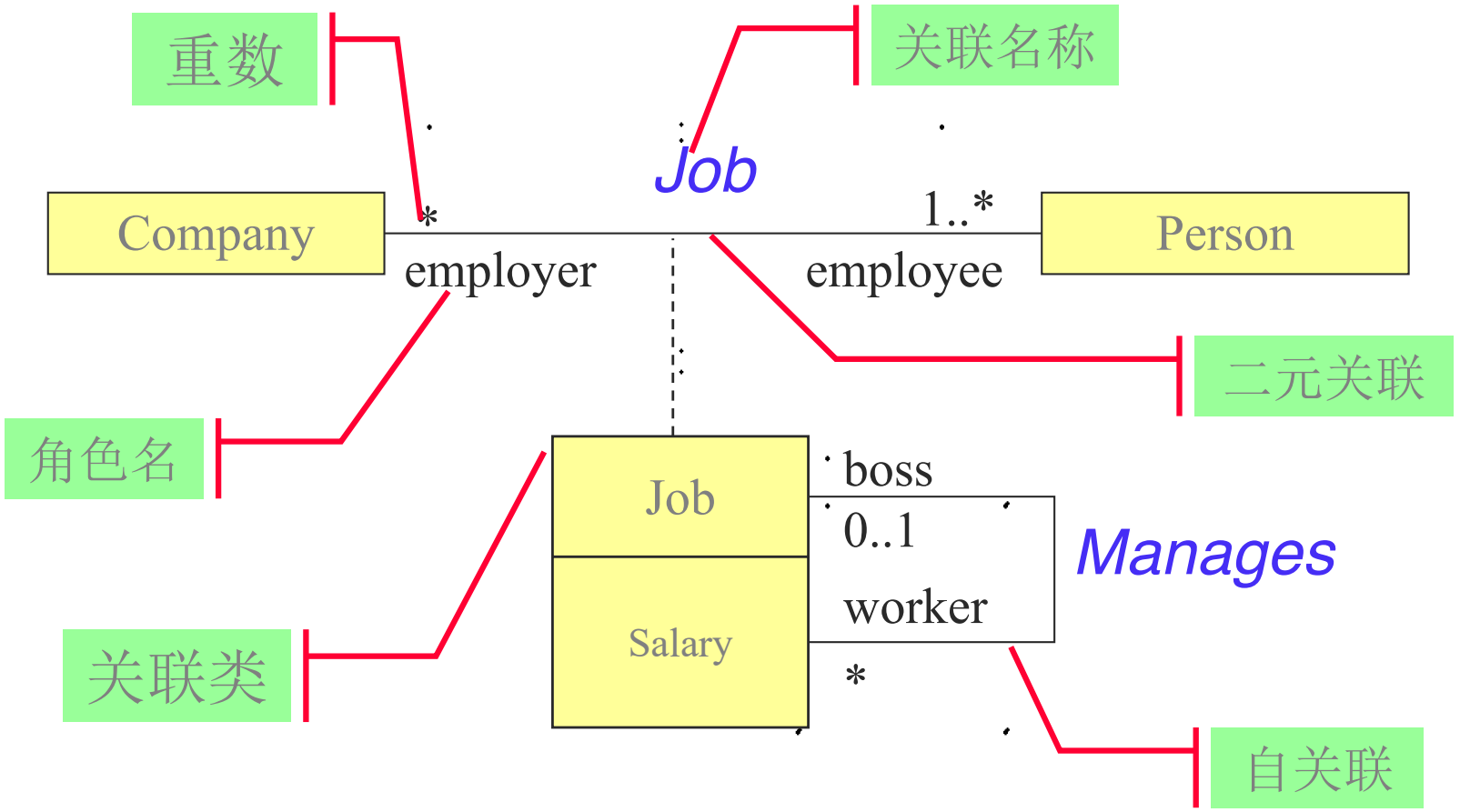
基于UML建模的面向对象设计

- 类的精化设计
  - 明确类的实例的行为
  - 定义状态及其转换事件、转换条件
    - 在详细设计阶段，状态建模一般只发生在依赖状态展示不同行为的类上
    - 为类的复杂状态变化构造状态图

基于UML建模的面向对象设计

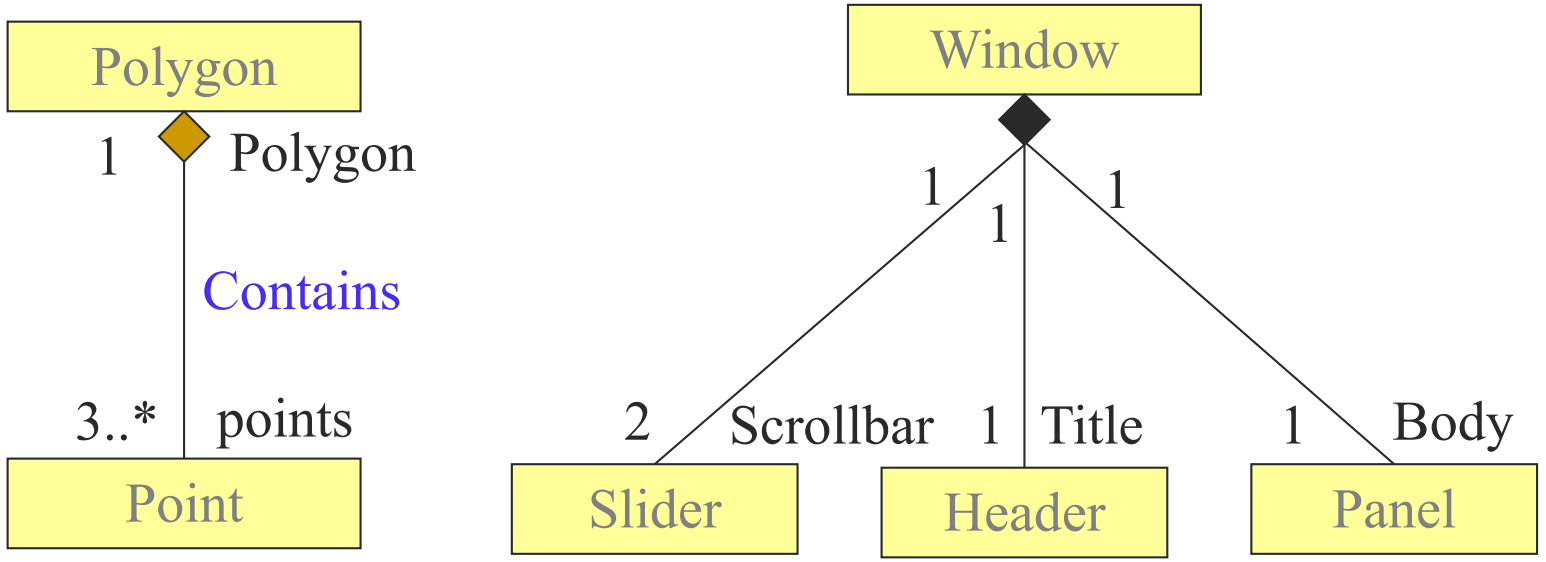
- 定义类间关系并构造类图
  - 设计阶段，需要进一步确定详细的关联关系、依赖关系和聚合关系等。
  - 不同对象之间的可能连接
  - 关联（Association）
    - 聚集，组成
  - 泛化（Generalization）
  - 依赖（Dependency）
  - 实现（Realization）
  - 约束（Constraint)

关联


重数
关联名称
Job
Company
*
employer
1..*
employee
Person
二元关联
角色名
Job
boss
0..1
Manages
worker
Salary
*
关联类
自关联

关联

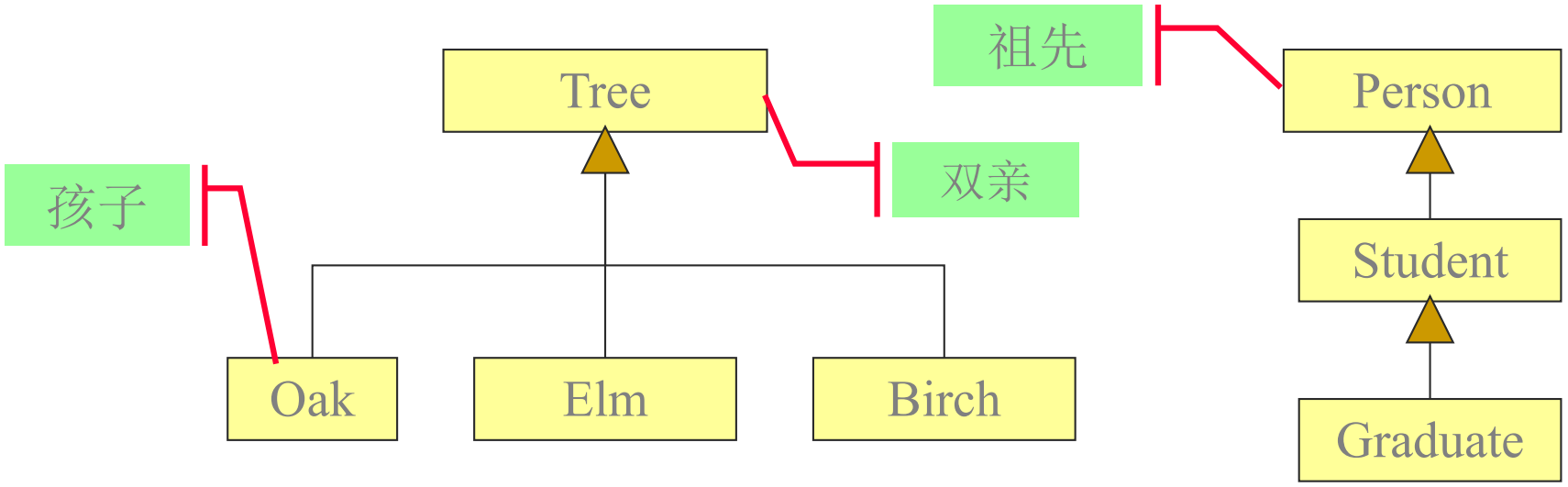
- 聚集（Aggregation）用来表达整体—部分关系的关联。组合（Composition）是一种聚集，是关联更强的形式。



聚集　　组合

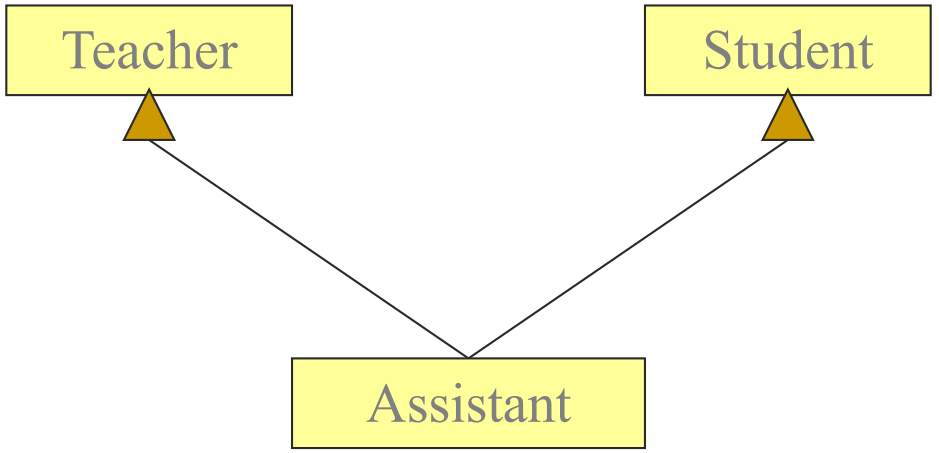
泛化

- 泛化是一般化和具体化之间的一种关系。
- 继承就是一种泛化关系，更一般化的描述称为双亲，双亲的双亲称为祖先，更具体化的描述称为孩子，在类的范畴，双亲对应超类，孩子对应子类。



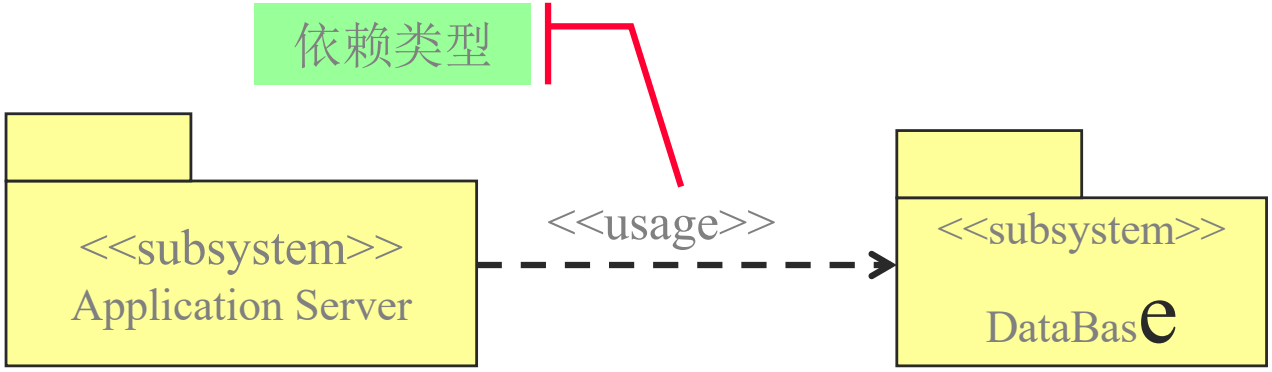
泛化

- 多重继承：一个孩子可以从多个双亲继承属性和方法。多重继承可能存在冲突，因为被继承的双亲可能存在相同的类声明，这时，最好显式解决冲突问题。



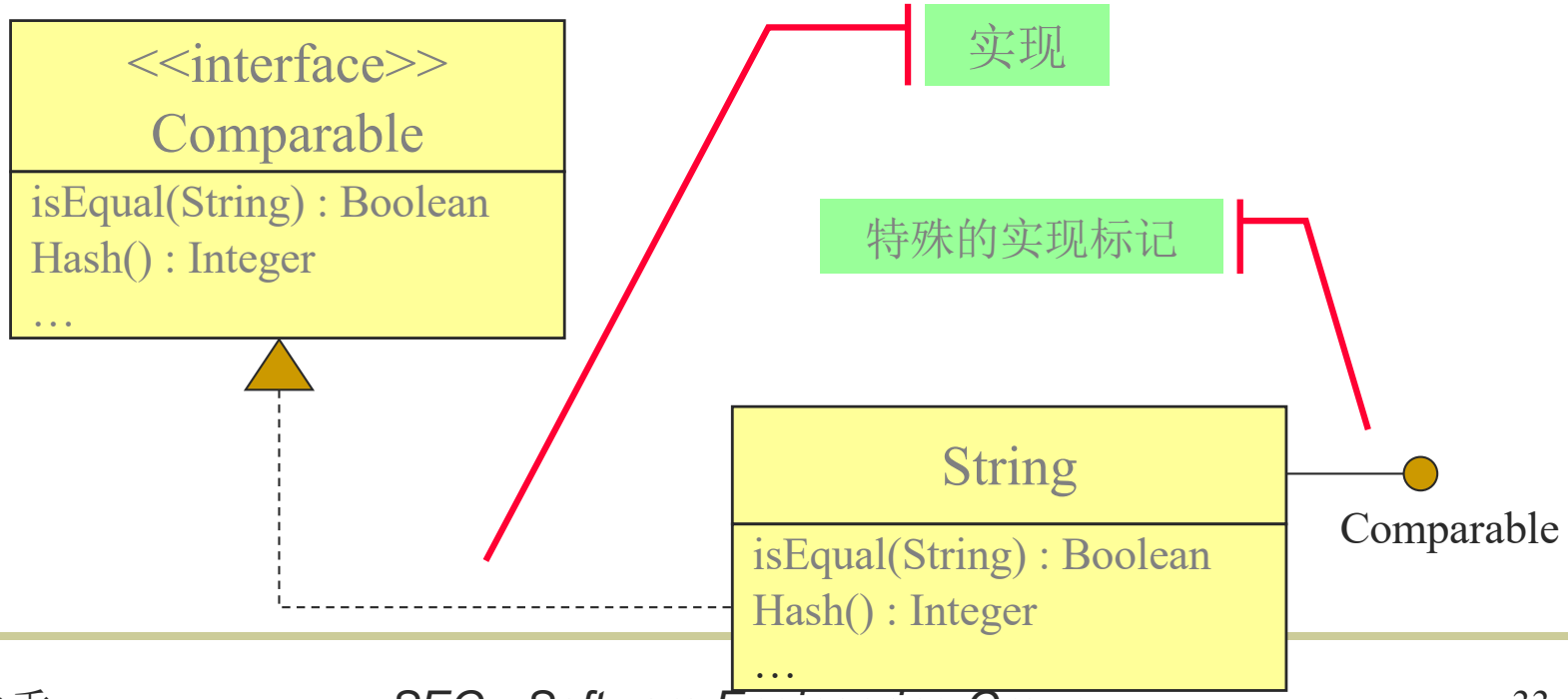
依赖

- 依赖指明两个或两个以上模型元素之间的关系。
- 依赖有很多种类，比如：实现（realize）、使用、（usage）、实例化（instantiate）、调用（call），派生（derive）、访问（access）、引入（import）、友元（friend）等等。



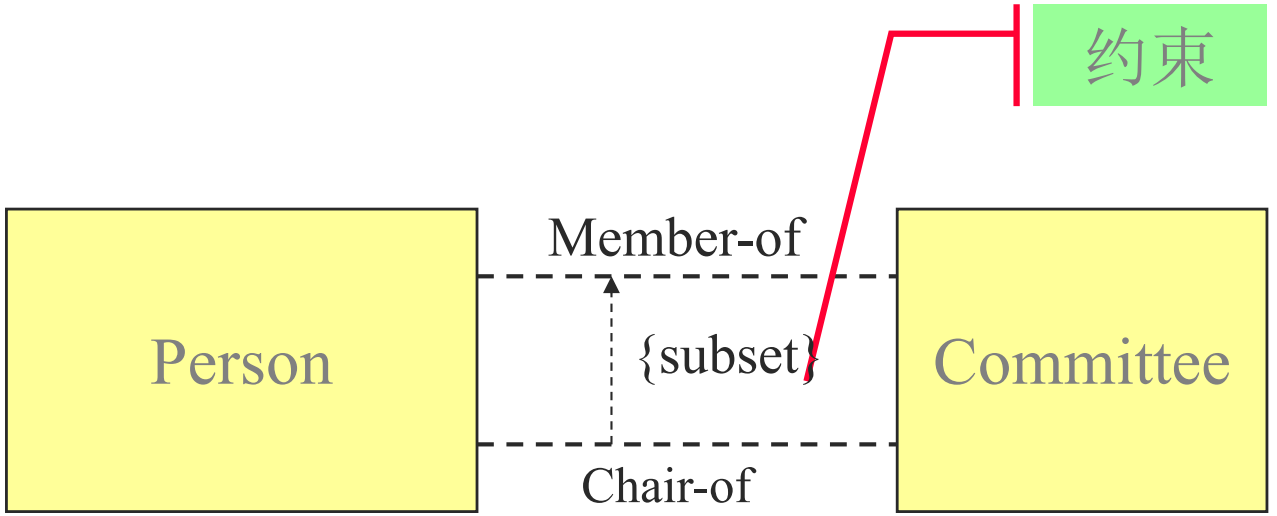
实现

- 实现是依赖的一种，但由于它具有特殊意义，所以将它独立讲述。实现是连接说明和实现之间的关系。



约束

- 约束用来表示各种限制，如关联路径上的限制，和属性特征检测（存在、所有）。

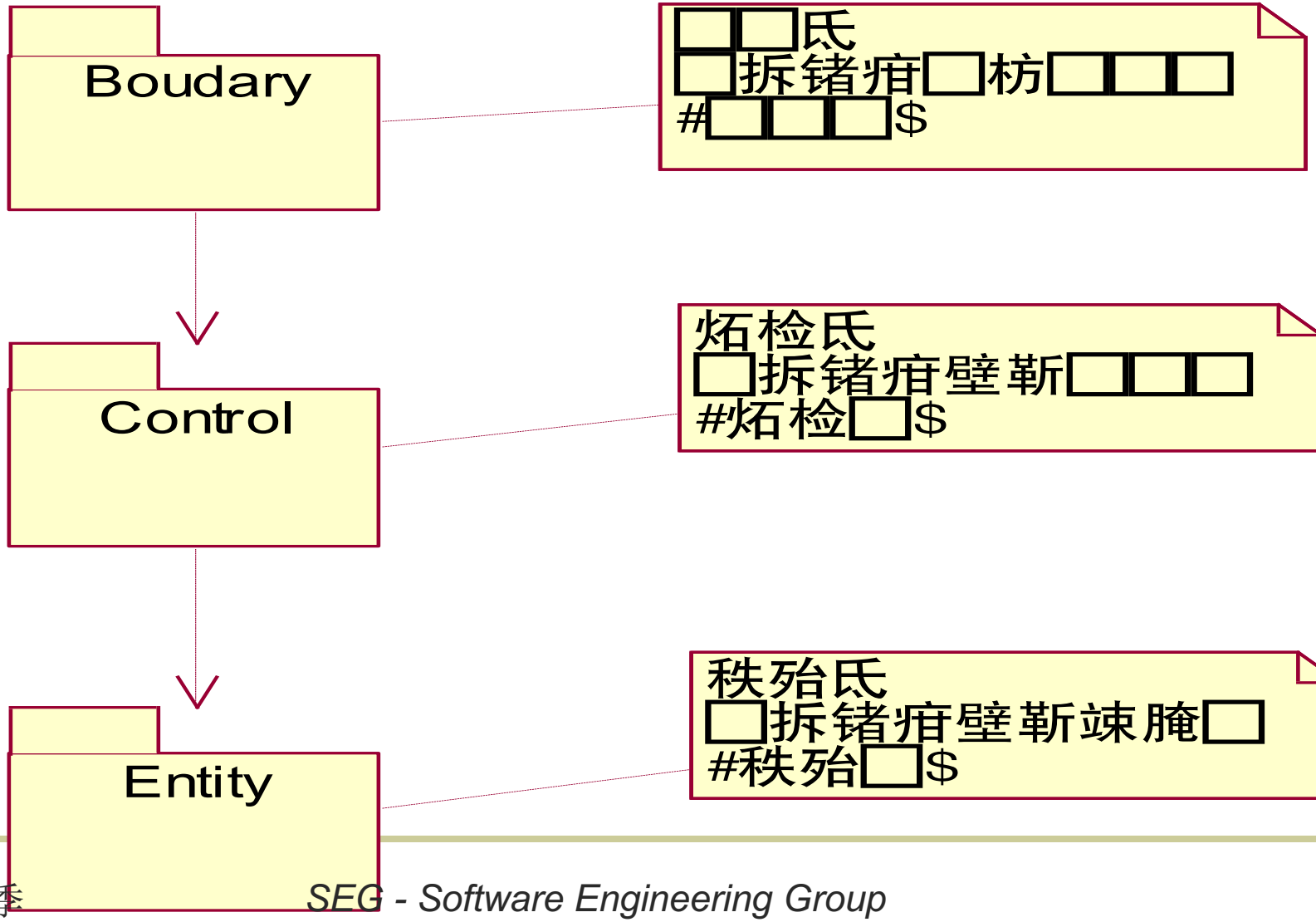


- 包与包图
  - 任何大系统都必须划分为较小的单元，以便人们在某一时刻可以和有限的信息工作，使团队的工作不相互影响。
  - 包可以包含各种模型元素和其它的包，包之间还可能存在一定的依赖
  - 包是一个逻辑类或其它包的集合
  - 包是一个高内聚、低耦合的类集合。
  - 包图描述了包和包之间的静态关系

基于UML建模的面向对象设计

- 基于类图构造包图
  - 包
  - 包间关系：依赖关系
  - 包图通常包括由设计模型分解的子系统、接口、依赖、关键设计类和用例实现的设计。
    - 体现子系统
      - 功能或实现范围上的划分
    - 体现类的组织结构
      - 用户界面层—控制层—业务层—持久层
  - 用包图描述体系结构（从设计角度）
    - 从设计模型的角度，描述系统的体系结构；

基于UML建模的面向对象设计



基于UML建模的面向对象设计

活动图设计

- 活动图描述系统中各种活动的执行顺序，通常用于描述一个操作中所要进行的各项活动的执行流程。同时，它也常被用来描述一个用例的处理流程，或者某种交互流程。
- 活动图适于表示用例中的事件流和过程，也可以用来表示复杂的算法以及并发处理进程。
- 活动图由一些活动组成，图中同时包括了对这些活动的说明。当一个活动执行完毕之后，控制将沿着控制转移箭头转向下一个活动。活动图中还可以方便地描述控制转移的条件以及并行执行等要求。

- 活动图设计
  - 活动图最适合支持描述并行行为，这使之成为支持工作流建模的最好工具。
  - 活动图最大的缺点是很难清楚地描述动作与对象之间的关系。
  - 对于以下情况可以使用活动图:

（1）分析用例;

（2）理解牵涉多个用例的工作流;

（3）处理多线程应用。

  - 在下列情况下，一般不要使用活动图:

（1）显示对象间合作;

（2）显示对象在其生命周期内的运转情况。

基于UML建模的面向对象设计

- 状态图设计
  - 状态图是对类的一种补充描述，它展示了此类对象所具有的可能的状态以及某些事件发生时其状态的转移情况。
  - 是描述一个实体基于事件反应的动态行为，显示了该实体如何根据当前所处的状态对不同的时间做出反应的。
  - 状态图用于显示状态机（它指定对象所在的状态序列）、使对象达到这些状态的事件和条件、以及达到这些状态时所发生的操作。
  - 状态由圆角矩形表示。状态的改变称作转移，状态转移由箭头表示，箭头旁可以标出转移发生的条件。状态转移可以伴随有某个动作，它表明当转移发生时系统要做什么。
  - 状态图适于表示跨越多个用例的单个对象的行为。

基于UML建模的面向对象设计

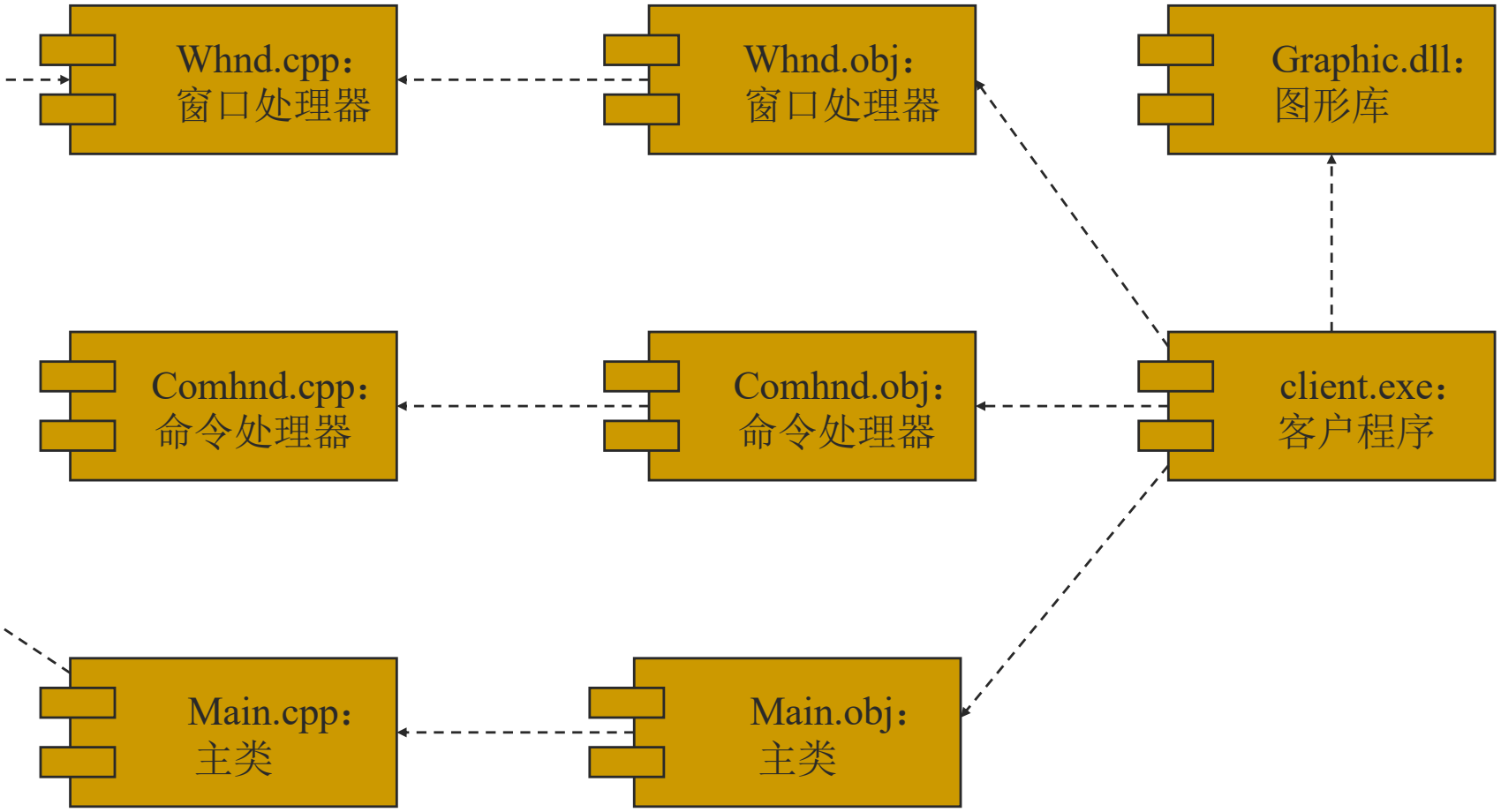
- 交互图设计（仅考虑顺序图）
  - 顺序图用来描述对象之间的动态交互关系，着重体现对象间消息传递的时间顺序。
  - 顺序图存在两个轴：水平轴表示不同的对象，垂直轴表示时间。
  - 顺序图中的对象用带垂直虚线的矩形框表示，在矩形框内标有对象名和类名。垂直虚线称为对象的生命线
  - 顺序图中的消息用箭头表示。箭头的形状表示消息的类型，有同步消息、异步消息、返回消息等等
    - 简单消息：不考虑通信过程的内部细节，用普通的有向箭头表示。
    - 同步消息：发出消息后必须等待消息处理过程完毕并返回处理结果。表示图元与简单消息相同。
    - 异步消息。发出消息后不必等待消息处理过程的返回。用特别的单向箭头表示。
    - 返回消息。

- 定义组件
  - 组件是可重用的系统片段，具有良好定义接口的物理实现单元。每个组件包含了系统设计中某些类的实现。
  - 组件设计的原则：良好的组件不直接依赖于其它组件，而是依赖于其它组件所支持的接口。这样的好处是系统中的组件可以被支持相同接口的组件所取代。
  - 一个组件可能是源代码、可执行程序或动态库。
  - 构造组件图：组件以及组件之间的依赖

基于UML建模的面向对象设计

- 设计组件（子系统）与接口
  - 子系统是具有独立的说明和实现部分的包，它代表了与系统其它部分具有清晰接口的清晰单元，它通常代表了系统在功能或实现范围上的划分。
  - 子系统是组织设计模型的一种手段，用以描述大粒度的构件，通常由设计类、用例实现、接口和其它子系统等组成。
  - 子系统的划分应该符合高内聚低耦合的原则。
  - 接口表示由设计类和子系统提供的操作。
  - 在确定了设计元素之后，需要描述子系统的行为，也就是准确定义接口操作的集合。同时，还要确定“子系统接口”与其他设计元素之间的依赖关系。

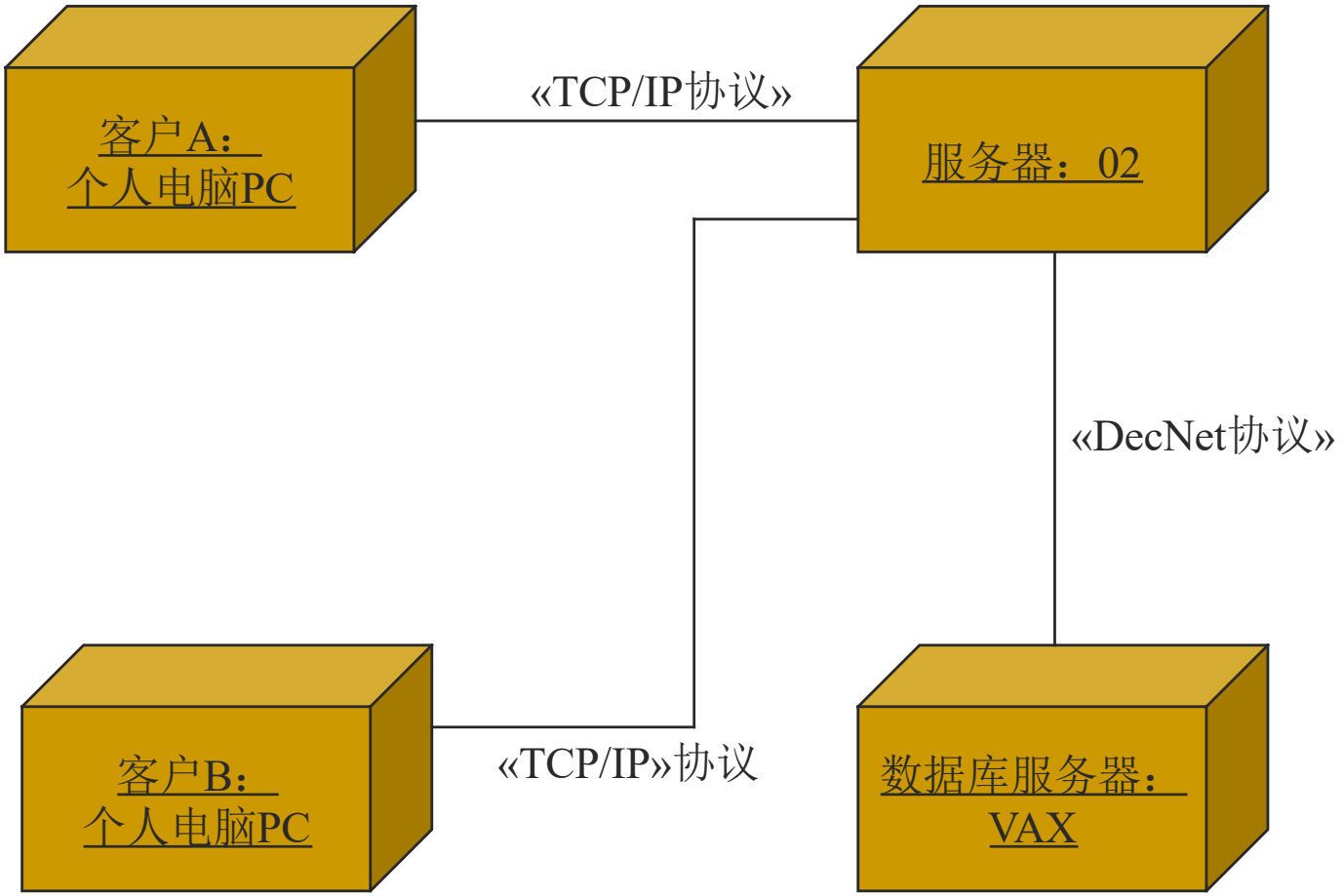
组件图


Whnd.cpp：
窗口处理器
Whnd.obj：
窗口处理器
Graphic.dll：
图形库
Comhnd.cpp：
命令处理器
Comhnd.obj：
命令处理器
client.exe：
客户程序
Main.cpp：
主类
Main.obj：
主类

基于UML建模的面向对象设计

- 构造部署图
  - 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组件的配置。
  - 根据系统在计算节点上的功能分布，描述整体的物理分布。
  - 节点代表系统运行时的物理对象，节点通常拥有运算能力，它可以容纳对象和组件实例
- 体系结构描述（从部署角度）
  - 具体包括部署模型的体系结构方面视图。
  - 设计层的体系结构在物理架构上的映射
    - 构件－节点映射

部署图


客户A：
个人电脑PC
«TCP/IP协议»
服务器：02
«DecNet协议»
客户B：
个人电脑PC
«TCP/IP»协议
数据库服务器：
VAX

基于UML建模的面向对象设计

- 小结
  - 输入：分析模型
  - 设计过程：设计用例实施方案、设计技术支撑方案、设计UI、精化设计模型
  - 输出：设计模型
    - 架构模型
      - 组件图
      - 配置图
    - 结构模型：
      - 类图
      - 包图
    - 行为模型
      - 状态机图
      - 顺序图
      - 活动图

基于UML的面向对象软件设计——案例分析

- 系统需求
- 用例图
- 在用例场景中选择对象
- 确定对象交互
- 设计类以及类之间的关系，设计类图
- 确定对象行为的状态图，
- 体系结构设计：包图，构件图，部署图
- 类和代码的确定

课程注册管理系统需求描述

- 研究生院/教务处建立一个本学期的课程目录
  - 一种课程可能有不同的时间、地点和听课对象的安排
  - 有不同的数据库来管理有关课程，学生和教师的不同信息
- 每个学生可选4门必修课和2门选修课，以便出现课程报满或取消的情况可以从备选中选择。
- 每门课程不超过10个学生。不少于3个学生。少于3个学生的课程将被取消。
- 一旦学生进行了选课注册，学校计费系统根据学生的注册和奖学金状态向学生发出交款通知。
- 学生可以利用这个系统在注册后的一段时间内修改选课计划，如增加或删除课程。
- 教授必须能够在线访问系统，以了解他教授的课程，他们也必须能够看见登记上课的学生情况。
- 系统的每一个用户通过他自己的口令验证来访问系统

执行者（Actor）的确定

执行者不是系统的一部分—它们代表任何必与系统交互的人或事物。

一个执行者有可能：

- 只向系统输入信息
- 只从系统获得信息
- 即向系统输入信息，又从系统获得信息

执行者（Actor）的确定

通常，执行者需要在问题陈述中挖掘，或者通过与户或领域专家进行对话来获取。为了获取角色，可以在人，其他的软件，硬件设备，数据存储，或者网络目录中进行找寻。

下面一些问题对获取执行者是很有帮助的。

- 谁使用系统
- 谁安装系统
- 谁启动系统
- 谁维护系统

执行者（Actor）的确定

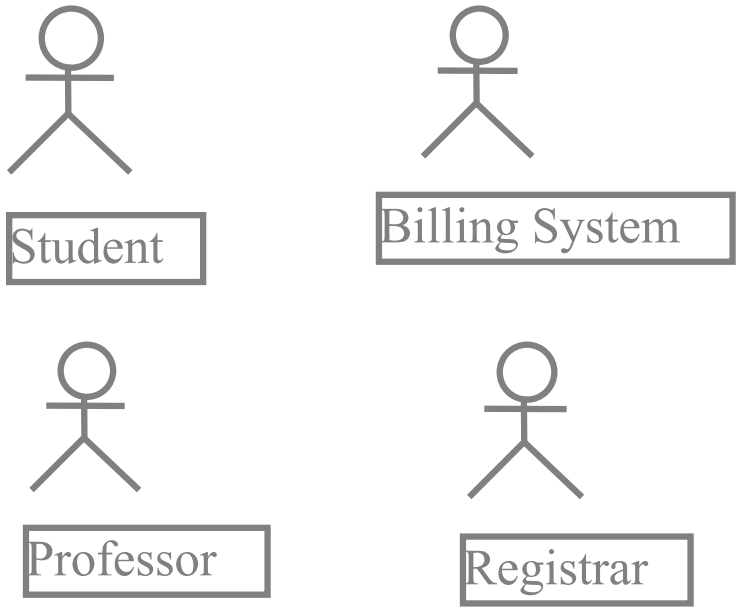
- 谁关闭系统
- 哪些其他的系统使用此系统
- 谁从此系统获取信息
- 谁为此系统提供信息
- 系统是否用到外部资源吗？
- 是否有人扮演多个角色？
- 是否有多人扮演同一个角色？
- 系统是否同遗留系统有交互？

执行者（Actor）的确定

- 学生要使用本系统注册所选课程
- 教授要使用本系统选择所教授的课程
- **教务员**要使用本系统创建课程目录和课表
- **教务员**要使用本系统维护所有相关课程，教授和学生的信息
- 计费系统能从本系统获取有关选课学生的信息

执行者（Actor）的确定

- 执行者是人或其它外部系统他/它将在系统开发和运行过程中和系统进行交互、对话。
  - 学生
  - 教授
  - 帐单系统
  - 注册系统



执行者（Actor）描述

每一个执行者都应该有一个简要的描述，用以指出该执行者在与系统交互式时扮演的角色

- **Student**—a person who is registered to take classes at the university
- **Professor**—a person who is certified to teach classes at the university
- **Registrar**—the person who is responsible for the maintenance of the Course Registration System
- **Billing System**—the external system responsible for student billing

用例（Use Case）的确定

- 用例描述了系统对外表现的特征和性能
  - 用例是系统的一个功能模块，是系统执行者和系统之间进行对话的一系列相关活动
- 如何寻找用例
  - 查看用户提交的文档
  - 询问系统的使用者
  - 对每个执行者进行分析，抽象他和系统之间可能的交互方法

用例（Use Case）的确定

一旦获取了角色，就可以对每个角色提出问题以获取用例。以下问题可供参考：

- 执行者要求系统提供哪些功能？
- 执行者需要读、产生、删除、修改或存储的信息有哪些类型？
- 必须提醒执行者的系统事件有哪些？或者执行者必须提醒系统的事件有哪些？怎样把这些事件表示成用例中的功能？
- 为了完整的描述用例，还需要知道执行者的某些典型功能能否被系统自动实现
- 系统需要何种输入输出？输入从何来？输出到何处?
- 当前系统（可能是手工系统而不是计算机系统）的主要问题有哪些？

用例（Use Case）的确定

- 一个好的用例是一个完整的(complete from beginning to end)、关于系统某一项主要功能的描述
- 设计用例时的注意事项
  - 用例独立于实现
  - 用例是系统的高级视图，数目不易过多
  - 用例的命名使用业务术语，而不是技术术语

用例（Use Case）的确定

- 学生选课、把学生加入对应课程的选课名单、通知学生交费，3个还是1个用例？
- 教务员向系统添加、删除、修改课程信息， 3个还是1个用例？

用例（Use Case）的确定

- 学生
  - 注册，浏览，增加，删除具体课程
- 教授
  - 索取课程花名册，选择执教的课程
- 教务员
  - 维护课程信息，维护学生信息，维护教授信息，产生课程目录
- 计费系统
  - 接受注册情况、计算注册费用，发出交款通知

用例（Use Case）的确定

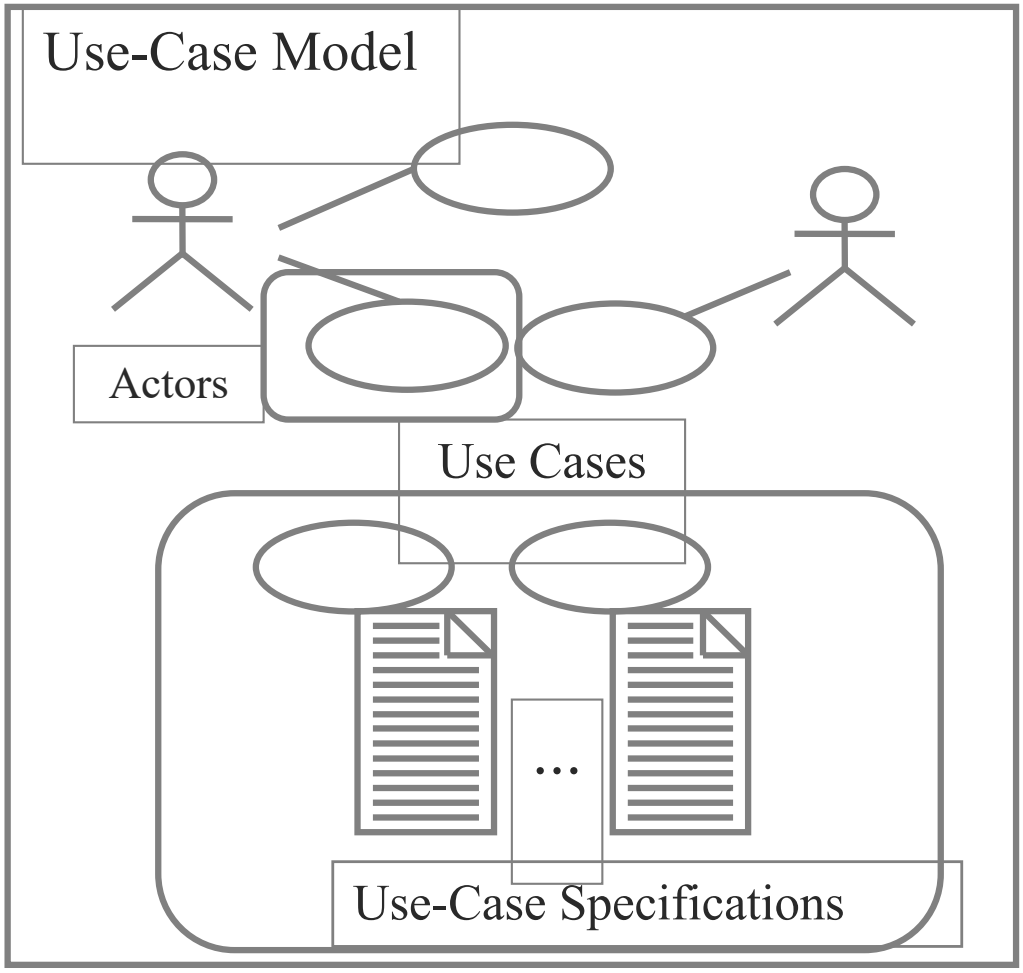
- Register for courses
- Select courses to teach
- Request course roster
- Maintain course information
- Maintain professor information
- Maintain student information
- Create course catalog

用例（Use Case）的描述

- Use Cases被描述在
  - 简短的描述
    - Use Case 的高级描述，用以指出它的执行者以及它提供的功能。
  - 事件流程
    - 运行过程中的执行序列

用例（Use Case）的规约

- 名称
- 简要描述
- 事件流
- 关系
- 活动图
- 用例图
- 特殊要求
- 前提条件
- 后置条件
- 其它要素



用例（Use Case）的事件流

- 具体说明为了执行这样一个使用范例，系统需要经历哪些过程内容
- 应该描述系统做什么(what)，而不是描述怎么做(how)
- 应该用问题域(business domain)的术语描述，而不是用解域(implementation domain)的术语描述

用例（Use Case）的事件流

- 用例什么时候、如何启动/终止
- 用例与执行者有什么交互
- 用例需要什么数据
- 常规事件流
- 可选事件流
- 异常事件流

用例（Use Case）的事件流

- **X Flow of Events for the <name> Use Case**
- **X.1 Brief Description**
- **X.2 Flow of Events**
- *X.2.1 Basic Flow*
- *X.2.2 Alternative Flows*
- **X.3 Special Requirements**
- **X.4 Pre-Conditions**
- **X.5 Post-Conditions**
- **X.6 Extension Points**

注册课程的事件流

- 注册课程：

  1.学生输入ID号

  2.系统验证ID号有效，并且提示学生选择当前的学期或未来的学期。

  3.学生输入学期，系统提示学生选择活动：

  - 创建一个课表，创建的子事件流将被激发
  - 浏览一个课表，浏览的子事件流将被激发
  - 改变一个课表，改变的子事件流将被激发
    - 删除一门课程，删除的子事件流将被激发
    - 添加一门课程，添加的子事件流将被激发

  4.学生指示活动结束，系统将打印学生课表，通知学生注册结束。

  5.注册系统送一个帐单给计费系统处理。

注册课程的异常事件流

- 另一种情况
  - 在前页中2，如果是无效ID号输入，系统禁止访问。
  - 在前页中3，如果在创建一个课表时，系统中课表已经存在，系统将提示进行其他选择（浏览或改变）。

注册课程的子事件流

- 创建课表
  - 学生输入四个主课号，和两门备选课程号．学生提交选课请求。系统将：

    1: 检查选课条件满足

    2: 如果课程仍然没有选满，将学生加入到课程中。
  - 另一种情况(异常事件流)

    如果主课程不能提供，系统将选择备选课程

注册课程的子事件流

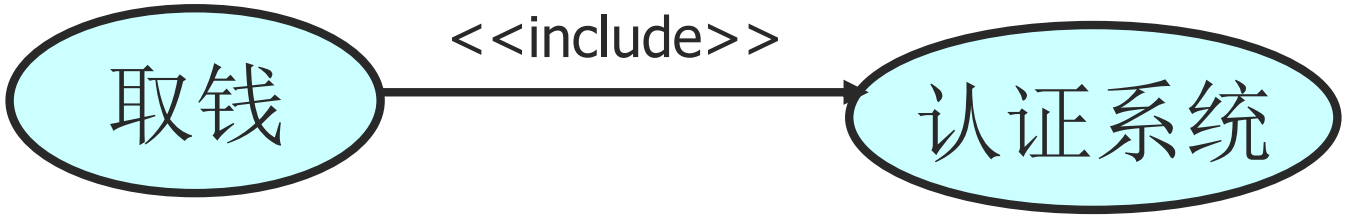
- 浏览课表
  - 学生请求一个给定学期的所有注册课程的信息，系统显示学生注册的所有课程信息，包括课程名，课程号，选课号，一周的天数，时间，地点，学分。

注册课程的子事件流

- 改变课表－删除一个课程
  - 学生指示哪个课程被删除，系统检查没有超过变更的最后期限，系统从课程中删除学生，系统通知学生请求被处理。
- 改变课表－增加一个课程
  - 学生指示哪个课程被增加，系统检查没有超过变更的最后期限，系统将：
    1. 验证增加的课程没有超过最大人数
    2. 检查选课条件满足
    3. 如果课程开放，将学生加入到课程中。

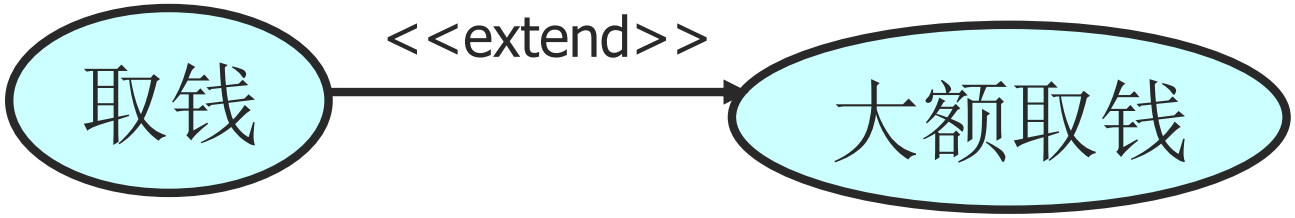
用例（Use Case）间的关系

- Include
  - «include»关系可用来描述某一个特性可能为一个或多个其它使用范例所共有。



用例（Use Case）间的关系

- Extend
  - « extend»关系描述的是可能扩展（ optional ）的特性



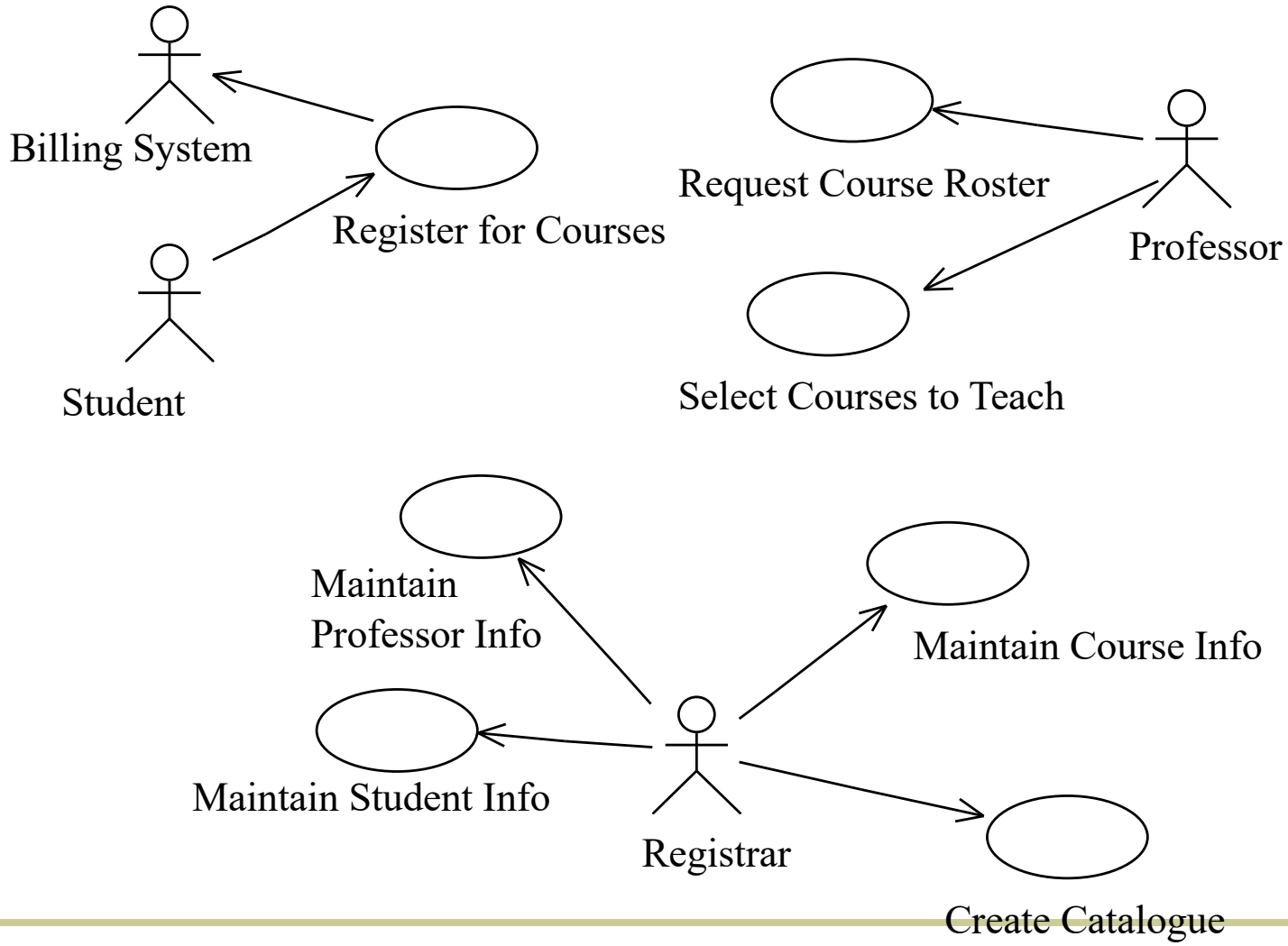
用例图（Use Case Diagram）

- 用例图描述各个执行者在各个用例中的参与情况，描述系统为用户所感知的外部视图。
- 用例图的功能：
  - 捕获系统用户需求
  - 描述系统边界
  - 指明系统外部行为
  - 指导系统开发者的功能开发
  - 系统建模的起点，指导所有的类图和交互图的设计
  - 产生测试用例，用户文档
  - 估计项目大小和进度。

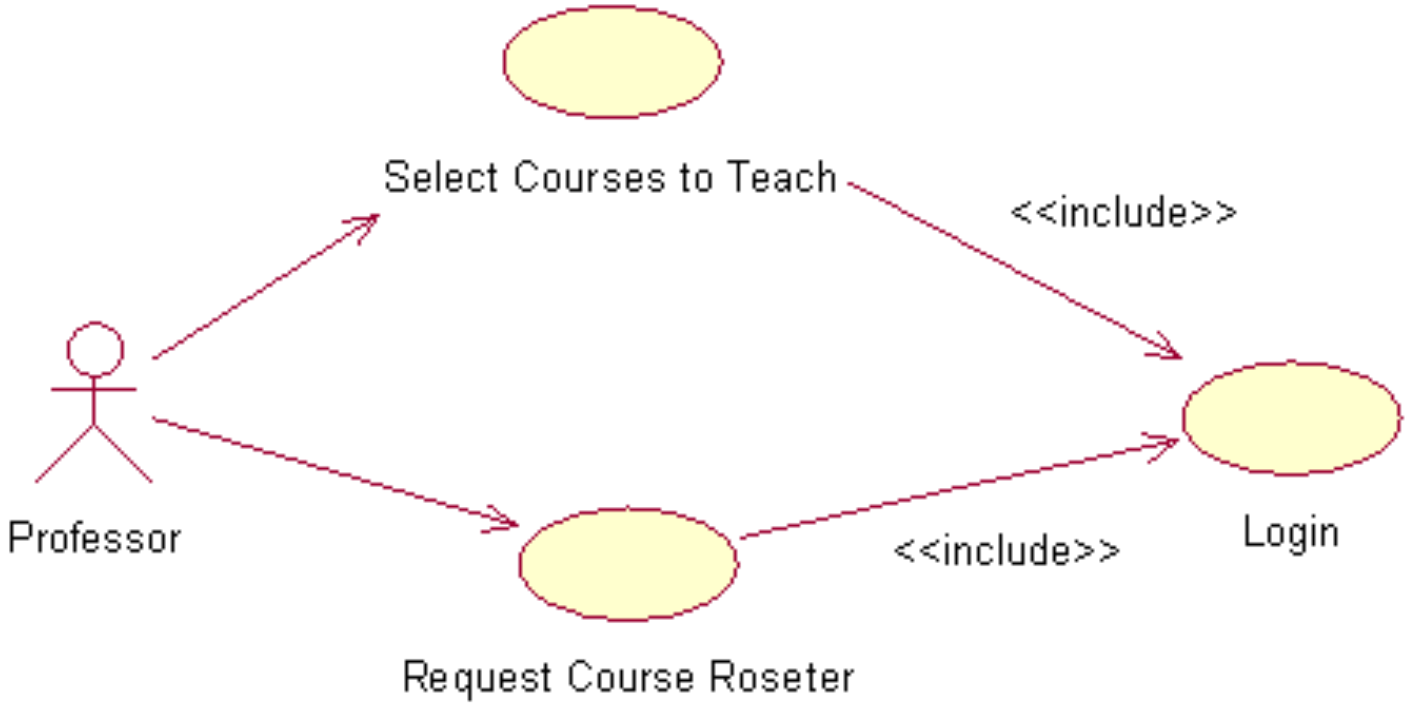
用例图（Use Case Diagram）

- 主用例图(Main Use Case diagram)描述了系统的边界(actors)和系统的主要功能(use cases)。
- 通常每个系统都有一个主用例图。
- 其它用例图则根据需要添加。

用例图（Use Case Diagram）



用例图（Use Case Diagram）



什么是对象

- 对象是一种概念、抽象或具有明确的边界的事情和应用目标
- 对象是具有：
  - 状态
  - 行为
  - 特性

注册课程的场景（Scenario）

- John 输入一个学生ID号369 52 3449，系统验证该号，系统询问学期信息，John指示现在的学期，并且选择创建一个新的课表。
- 从一个可选课程列表中，John选择了主课English 101, Geology 110, World History 200, and College Algebra 110. 然后他选择了备选课程Music Theory 110和Introduction to Java Programming 180.
- 系统判定John有选课条件，并把他加入了课程的花名册中。
- 系统指示选课活动结束，系统打印课表，将四门课程的帐单送往帐单系统处理。

注册课程的辅助场景（Secondary Scenario）

- 一些值得考虑的辅助案况:
  - 学生没有选择4门主课。
  - 主课没有提供。
  - 主课和备选课程没有提供
  - 不能把学生加入花名册。
  - 不能创建学生的课表

注册课程的场景名词

- John
- 系统
- 学期
- 新的课表
- 主课
- Geology 110
- College Algebra 110
- Music Theory 110
- 选课条件
- 选课活动
- 帐单
- 帐单系统

学生ID号 369523449

该号

现在的学期

可选课程列表

English 101

World History 200

备选课程

Introduction to Java Programming 180

课程的花名册

课表

四门课

名词过滤

- John -- filtered (角色)
- 学生ID号 369523449 -- filtered (一个学生的属性)
- 系统 -- filtered (就是要构建的)
- 该号-- filtered (与学生ID号相同)
- 学期 -- filtered (状态– 当选择发生时)
- 现在的学期 -- filtered (同学期)
- 新的课表 -- candidate object
- 可选课程列表 -- candidate object
- 主课 -- filtered (选择课程的状态)
- English 101 -- candidate object
- Geology 110-- candidate object

名词过滤

- World History 200 -- candidate object
- College Algebra 110 -- candidate object
- 备选课程 -- filtered (选择课程状态)
- Music Theory 110 -- candidate object
- Introduction to Java Programming 180 -- candidate object
- 选课条件 -- candidate object
- 课程的花名册 -- candidate object
- 选课活动 -- filtered (英语表达)
- 课表 -- filtered (同新的课表)
- 帐单 -- candidate object
- 四门课 -- filtered (帐单系统需要的信息)
- 帐单系统 -- filtered (角色)

场景中的对象

- 新的课表 – 一个学生一学期的课程列表
- 可选课程列表 – 在一个学期教授的课程列表
- English 101 – 一个学期的一门课程
- Geology 110 – 一个学期的一门课程
- World History 200 --一个学期的一门课程
- College Algebra 110 --一个学期的一门课程
- Music Theory 110 --一个学期的一门课程
- Introduction to Java Programming 180 --一个学期的一门课程
- 选课条件– 必须在选修一个课程前完成的课程列表
- 课程的花名册 – 一个特定课程的学生名单
- 帐单 – 帐单系统需要的信息

交互图

- 制作交互图（顺序图）的两步法：
  - 第一步关注客户关心的高级信息。消息不映射为操作，对象不映射为类。这些图只是为了让客户和系统设计人员对系统的逻辑流程进行交流
  - 客户同意第一步框图之后，系统设计人员再加入一些细节。如：在图中添加一些对象用来处理控制、安全、错误处理、数据库连接、进程间通信等问题

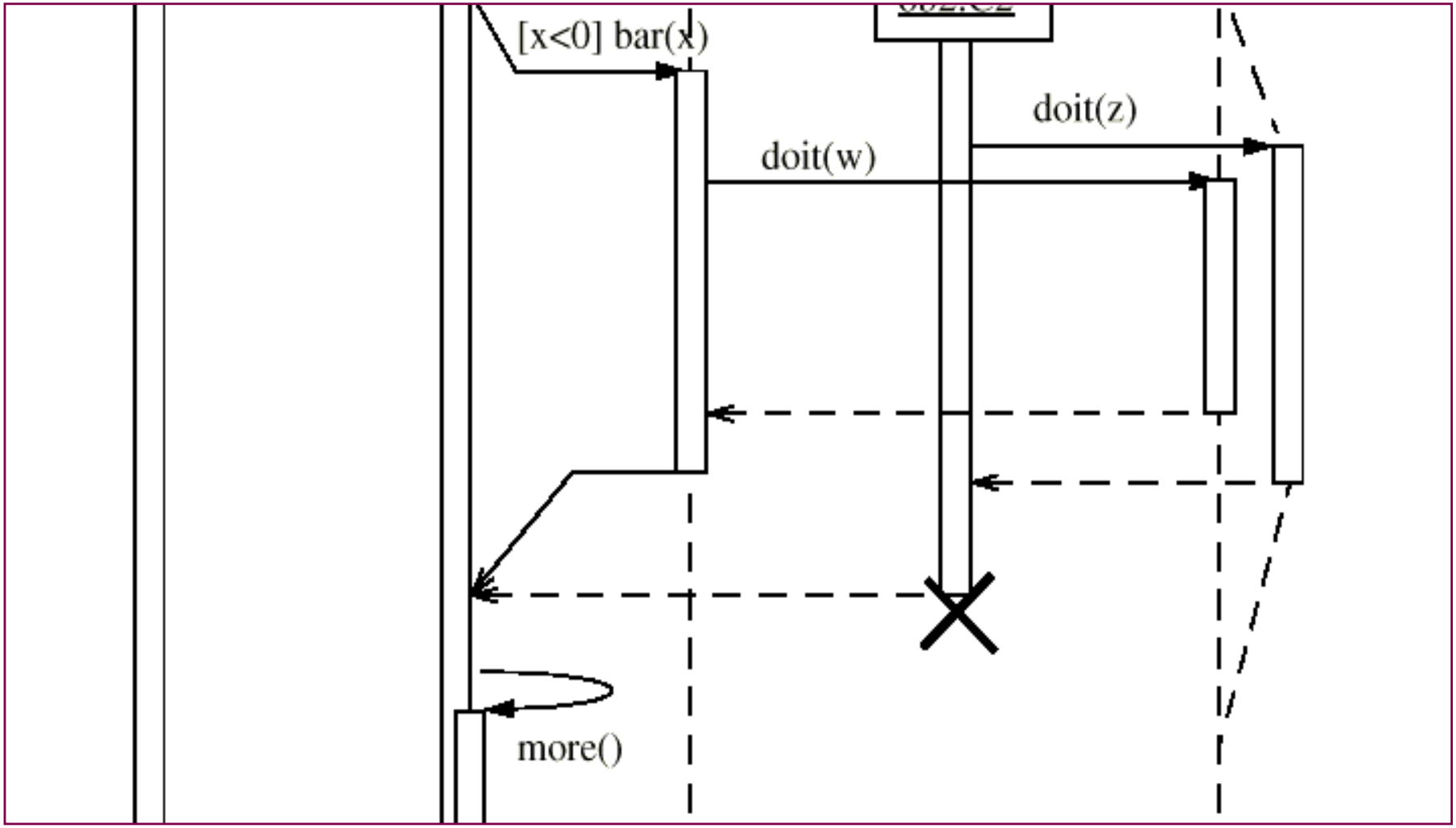
顺序图

- 顺序图用来描述对象之间的动态交互关系，着重体现对象间消息传递的时间顺序。
- 顺序图存在两个轴：水平轴表示不同的对象，垂直轴表示时间。
- 顺序图中的对象用带垂直虚线的矩形框表示，在矩形框内标有对象名和类名。垂直虚线称为对象的生命线
- 顺序图中的消息用箭头表示。箭头的形状表示消息的类型，有同步消息、异步消息、返回消息等等

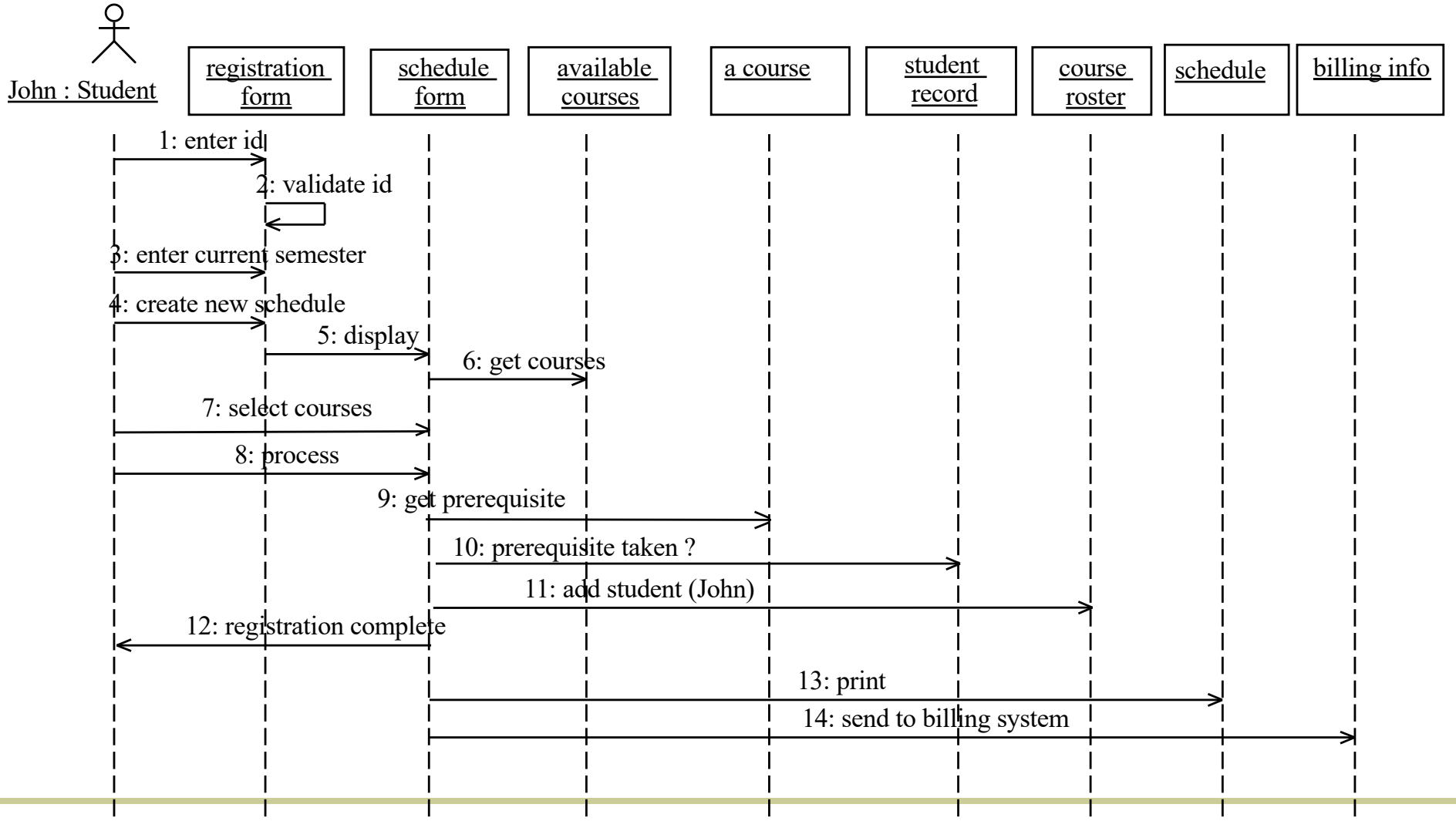
顺序图(sequence diagram)

- 顺序图描述了系统运行的一个情景，通常对于用例图中的每一个用例，都对应若干个顺序图

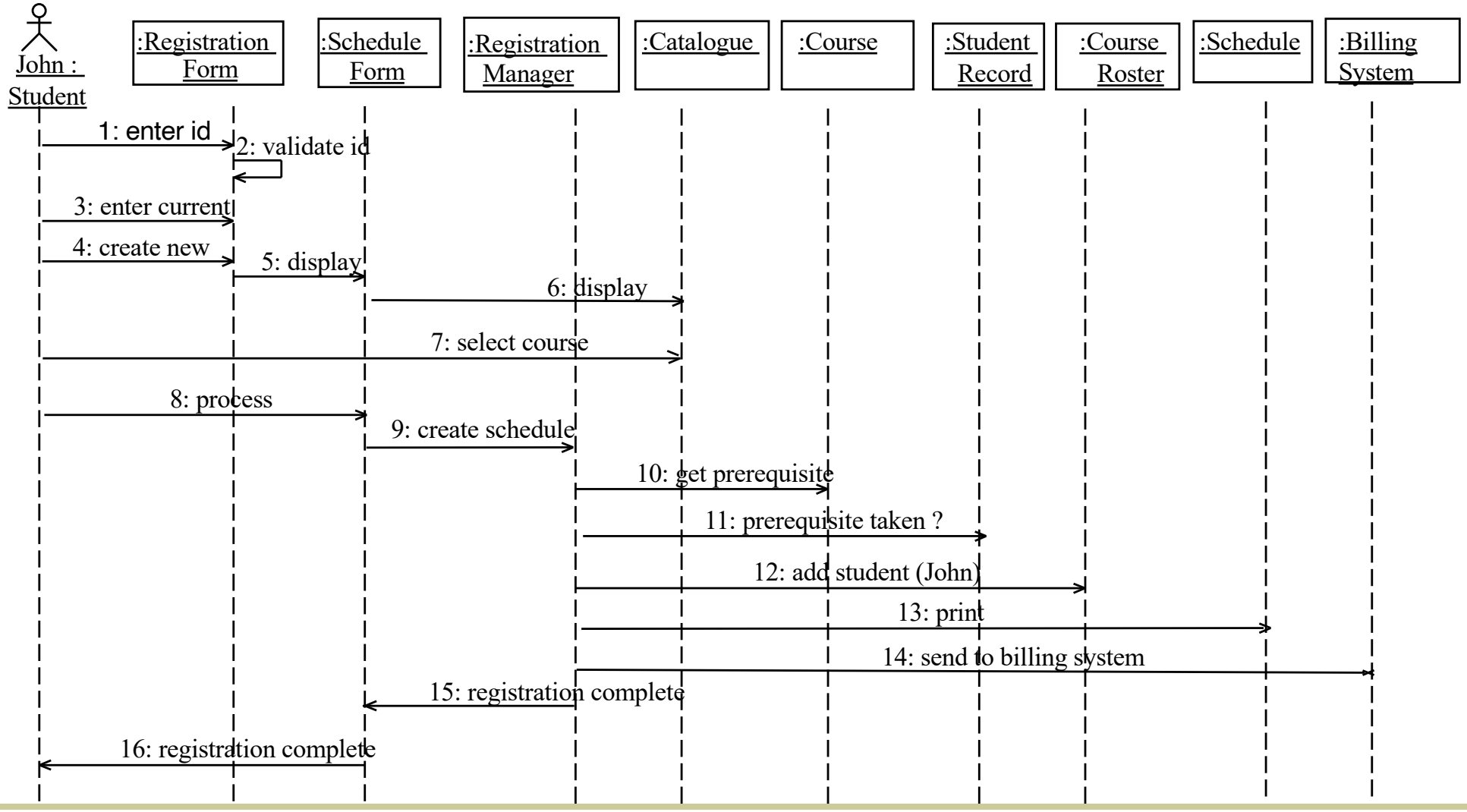
顺序图(sequence diagram)



注册课程的顺序图


John : Student
registration form
schedule form
available courses
a course
student record
course roster
schedule
billing info
1: enter id
2: validate id
3: enter current semester
4: create new schedule
5: display
6: get courses
7: select courses
8: process
9: get prerequisite
10: prerequisite taken ?
11: add student (John)
12: registration complete
13: print
14: send to billing system

修改后的顺序图


John : Student
:Registration Form
:Schedule Form
:Registration Manager
:Catalogue
:Course
:Student Record
:Course Roster
:Schedule
:Billing System
1: enter id
2: validate id
3: enter current
4: create new
5: display
6: display
7: select course
8: process
9: create schedule
10: get prerequisite
11: prerequisite taken ?
12: add student (John)
13: print
14: send to billing system
15: registration complete
16: registration complete

包图和类图

- 类图描述类和类之间的静态关系。
- 包是一个高内聚、低耦合的类集合。
- 包图描述了包和包之间的静态关系。

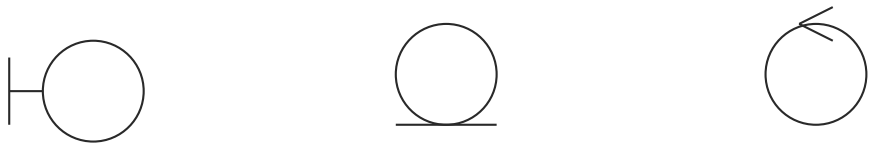
类图

- 类图是OO方法的核心。它与数据模型不同，不仅显示了信息的结构，还显示了系统的行为。
- 使用类图时有三种不同的透视角度：
  - 概念层：类图描述的应用领域中的概念，这些概念与实现时的类并不是一一映射的。概念模型独立于程序设计语言
  - 说明层：该层次考察的是软件的接口部分，而不是实现部分，也就是说考察的是类型而不是类
  - 实现层：只有在这一层中的类才是严格意义上的类，它揭示了软件实现体的构成情况

类的构造型 (Stereotype)

UML中有三种主要类构造型：

- 边界类 (Boundary)：位于系统与外界的交界处，包括所有窗体、报表、硬件接口以及与其他系统的接口
- 实体类 (entity)：保存要放进持续存储体的信息
- 控制类 (Control)：负责协调其他类的工作



查找类

- 类是具有相同结构和行为的对象的集合
- Stereotype是建模元素的新类型，这种建模元素扩展了metamodel的语义
  - 每个类最少有一种stereotypes
- 在分析中有三种普通的stereotypes
  - 实体类
    - 模型信息和相关行为广泛的永久的独立于它的环境
  - 边界类
    - 系统环境和内部工作见的模型关联
  - 控制类
    - 一个或多个模型控制行为规格

查找类

- Use cases与Sequence diagram可以对查找实体和边界类型进行检查
- 最初，给每一个use case建立一个控制类
  - 控制类可以作为分析过程被归并
- 例子：注册课程的Use Case
  - 边界类
    - 注册表单、选课课表、计费界面、课程的花名册
  - 实体类
    - 课程、选课目录、学生课表、学生选课纪录
  - 控制类
    - 注册管理器

类的描述

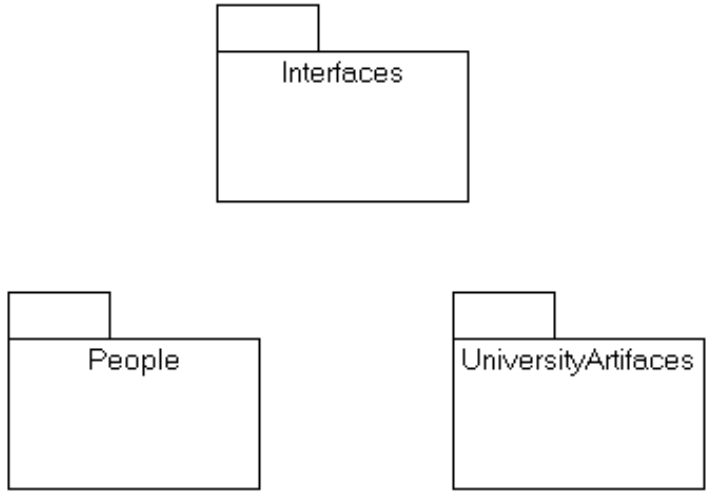
- 一旦类被建立，它应该被定义
  - 它包含类的责任和目的描述
  - 类的规格说明包含类的额外信息
  - 类的类型

使用类图的几点建议

- 不要试图在项目的初始阶段使用所有的符号。应当从简单概念开始，然后逐渐丰富类图
- 在项目的不同开发阶段，应当使用不同的观点来画类图。在分析阶段使用概念层的类图，在设计阶段使用说明层的类图，在实现阶段使用实现层的类图
- 不要为每一个事物都画一个模型，应该把精力放在关键的领域。使用类图的最大危险就是过早的陷入实现细节。
- 模型是否真实的反映了研究领域的实际
- 模型和模型中的元素是否有清楚的目的和职责
- 模型和模型中的元素大小是否适中。

包

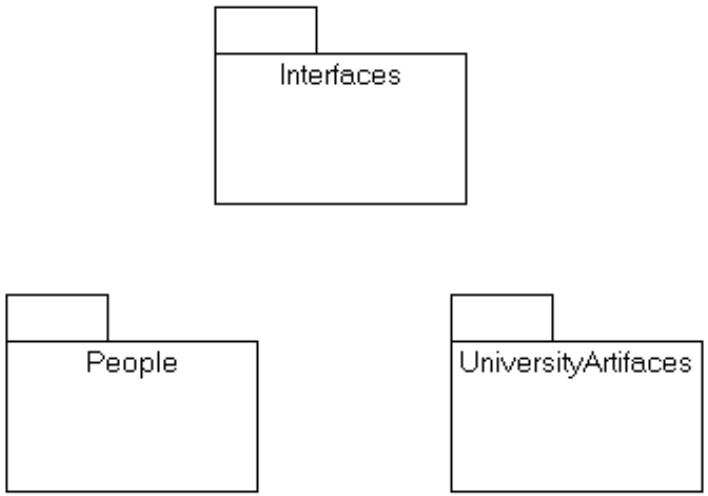
- 把登记系统中的类放在三个包中
  - 界面、人和学校制品
- 一旦包被建立，它应被定义
  - 定义描述了包的目的
  - 合适的类被重新分配在



主类图

- 逻辑视图最初包含一个视图
  - 该图形被称为Main
- 主类图是逻辑视图中典型的高级包视图

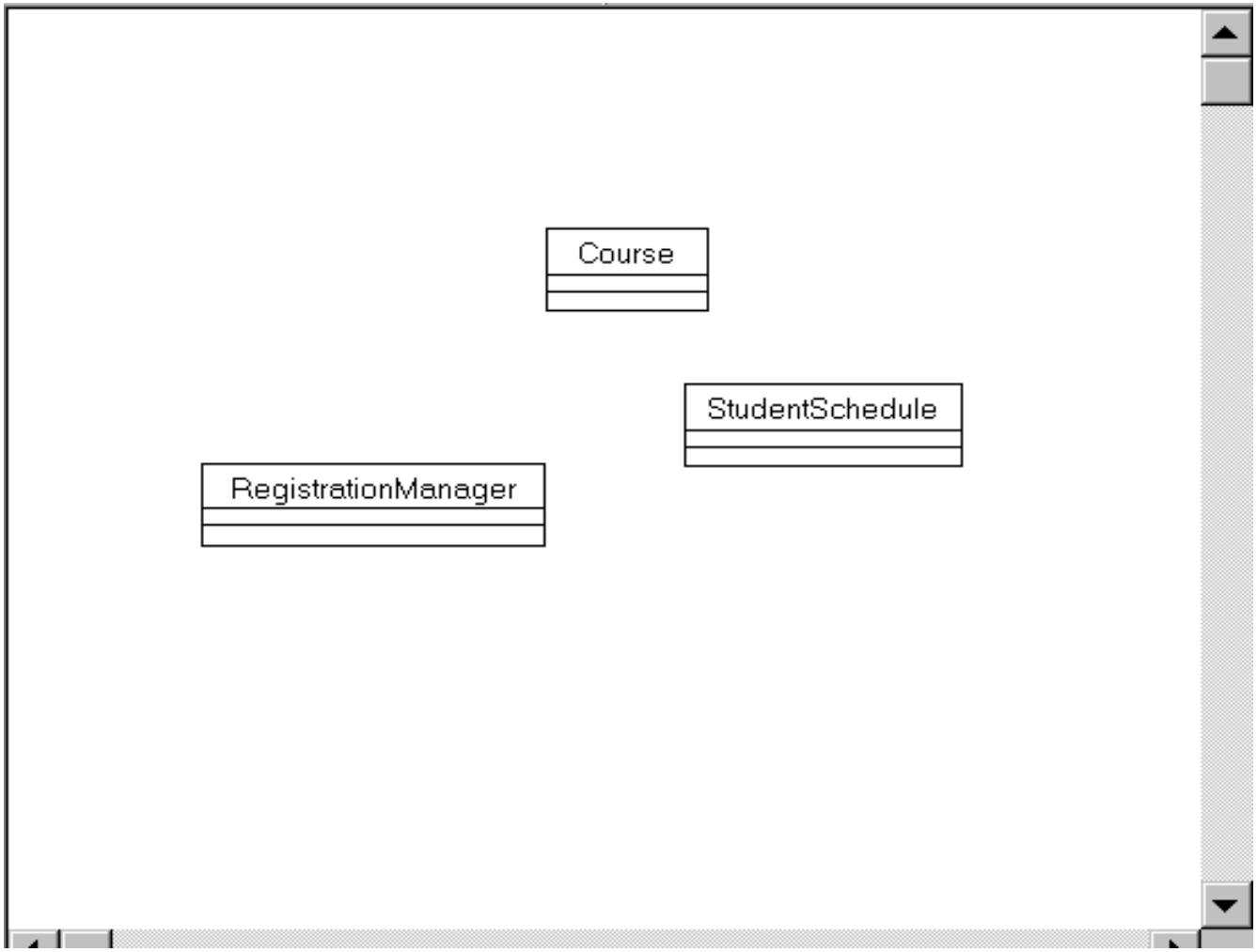
课程注册系统的主类图



在包中进行浏览

- 每个包一般都有自己的主类图
- 该图形一般展现
  - 包中的“公众”类
    - 其它包中的类可以和它关联
  - 公众类连接
    - 在分析后加入类图

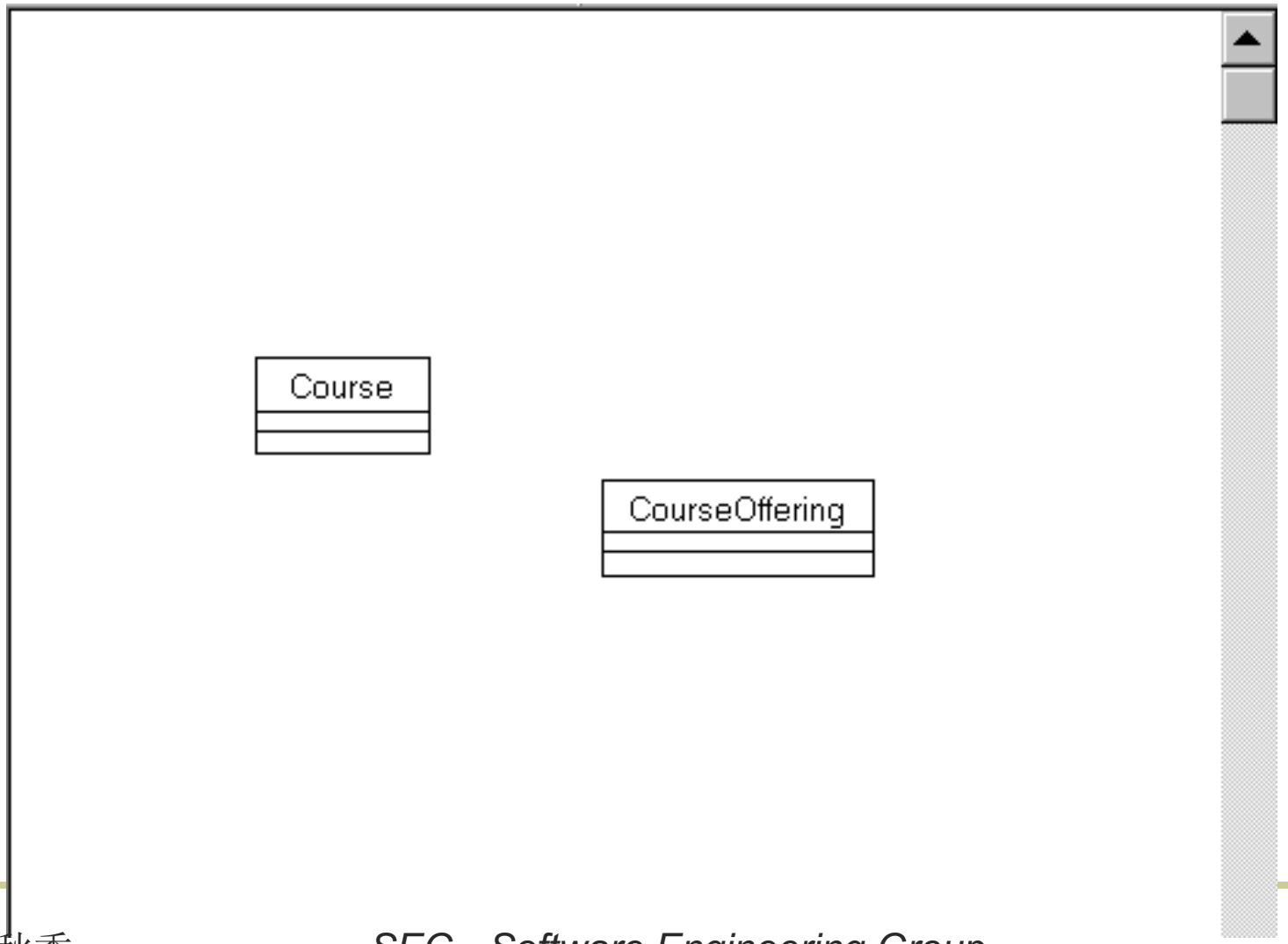
学校制品包中的主类图



额外的类图

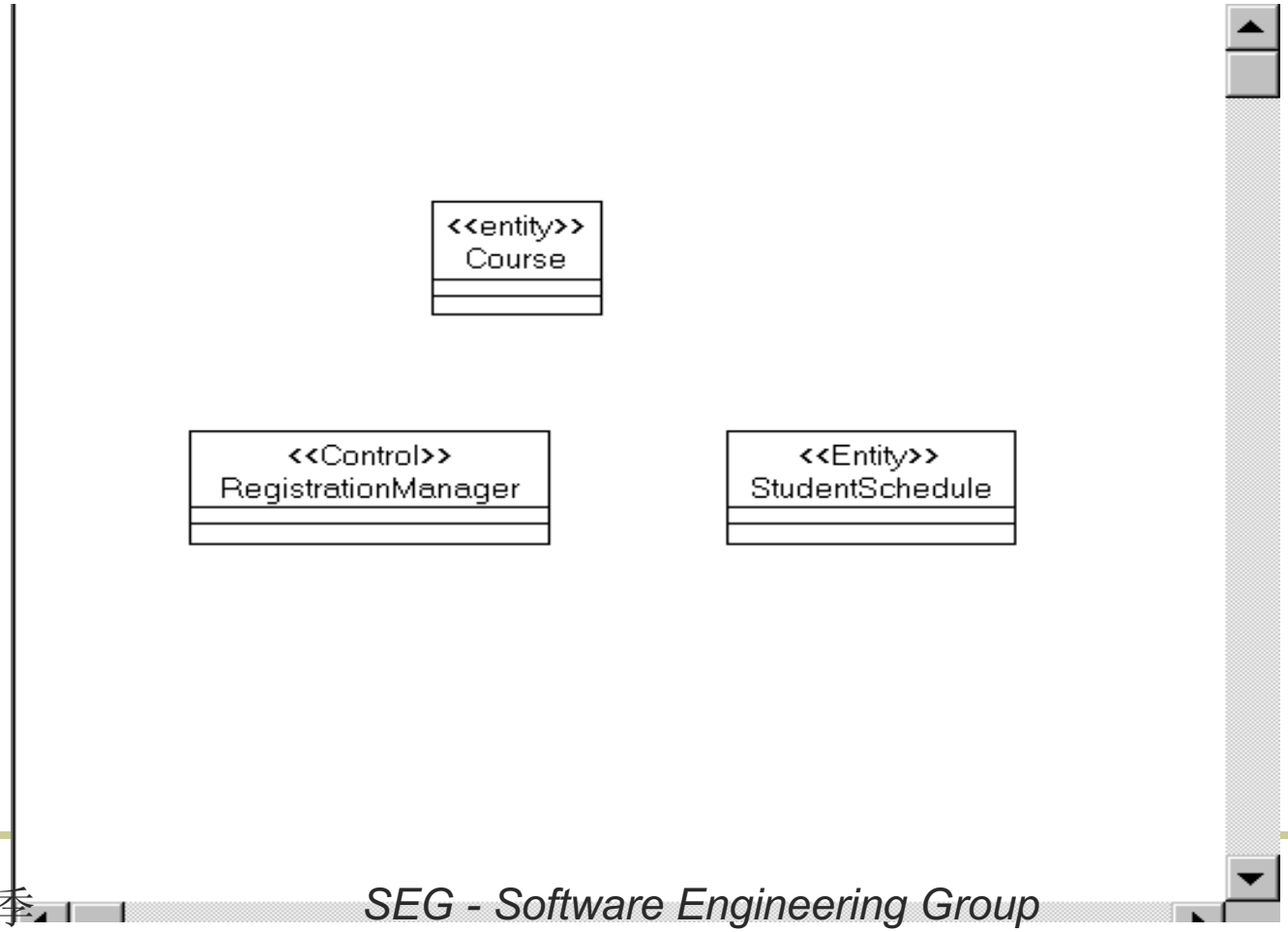
- 需要时可以加入额外的类图
- 它们展现了模型中包和类的另一种“视图”
- 例子：
  - 用例中多个类的视图
  - 包中“私有”类的视图
  - 一个或多个类的视图及它们的属性和操作
  - inheritance hierarchy视图

学校事物包中的额外类图



展现Stereotypes

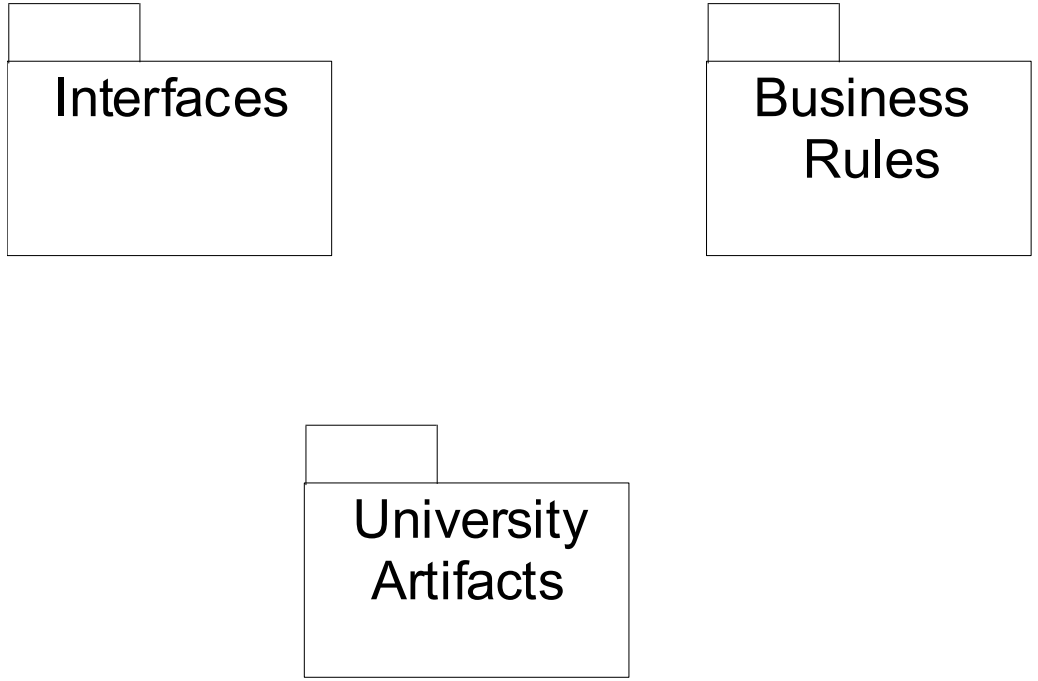
- 类的stereotype可以展现在类图中



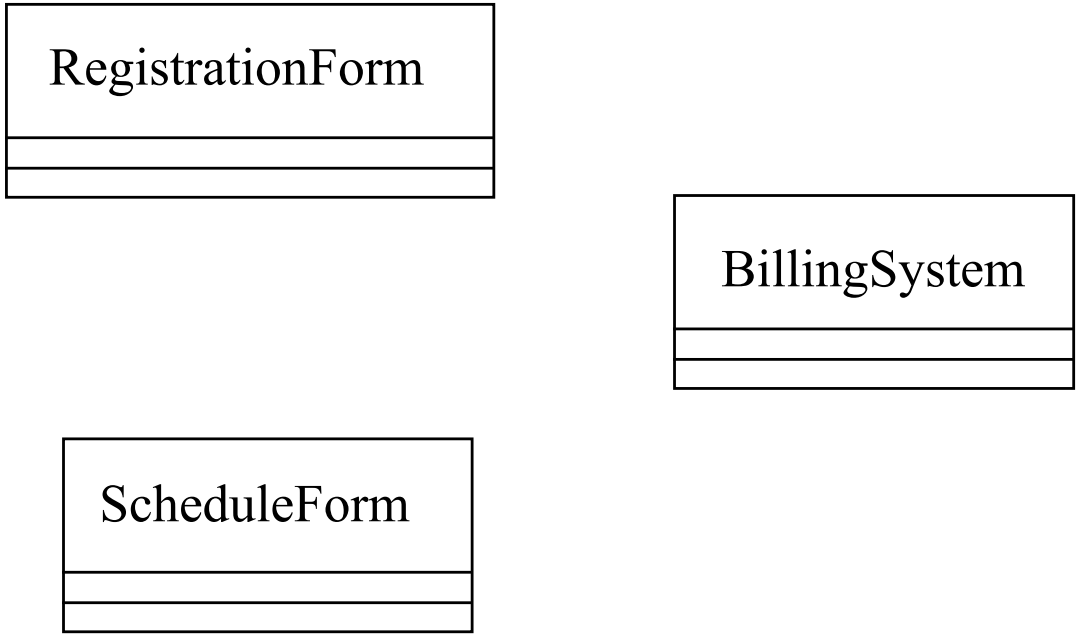
注册系统中的包

- 组成三个包
  - UniversityArtifacts , BusinessRules, and Interfaces
- UniversityArtifacts
  - Catalogue, Course, StudentRecord, CourseRoster, Schedule
- BusinessRules
  - RegistrationManager
- Interfaces
  - RegistrationForm, ScheduleForm, BillingSystem

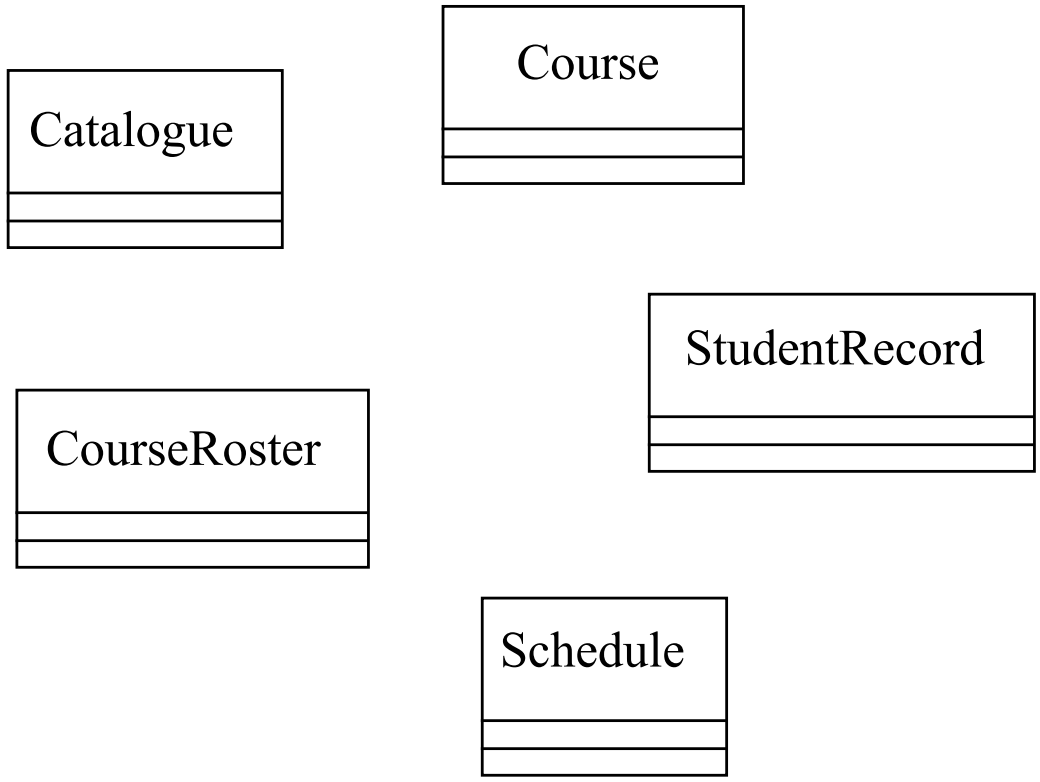
注册系统中的主要类图



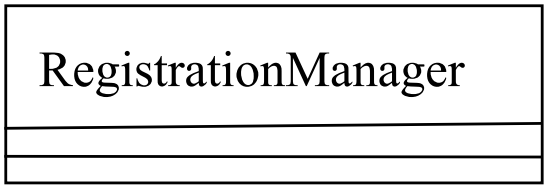
Interfaces包中主要类图



UniversityArtifacts包中类图



BusinessRules包中的类图



关系

- 建立关联和聚合关系
- 用名称、角色和多种指示增加关系
- 建立反身关系
- 加入强制关系

关联和聚合

- Use Case可以检测并决定两个类之间是否应该存在关系
  - 只要两个对象可以互相识别，它们就可以通信
  - 关联和聚合为通信提供了一条途径
- 关联是两个类间的非直接连接
- 聚合是关联的一种强制模式
  - 它描述整体与部分之间的关系

关联还是聚合？

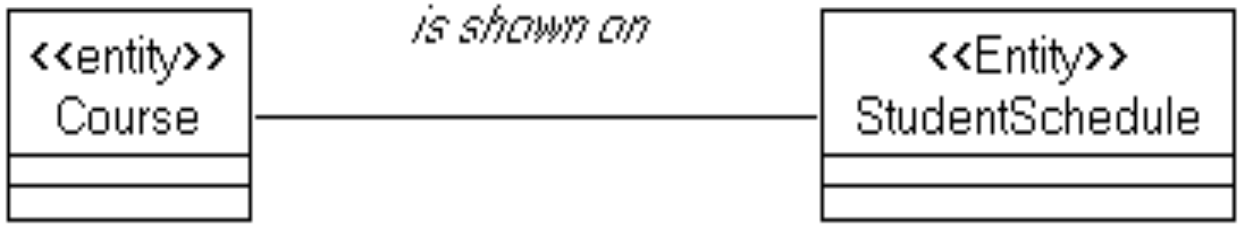
- 如果两个对象通过整体和部分的关系具有紧密的边界
  - 这种关系称为聚合
- 如果两个对象通常被人为是独立的
  - 这种关系称为关联

关系和类图

- 包中的Main类图一般包含：
  - 包中的公众类
    - 其它包中的类可以跟它进行通话的类
  - 其它包中的类和公众类进行通信
- 如果需要，关系则被加入另外一个图形

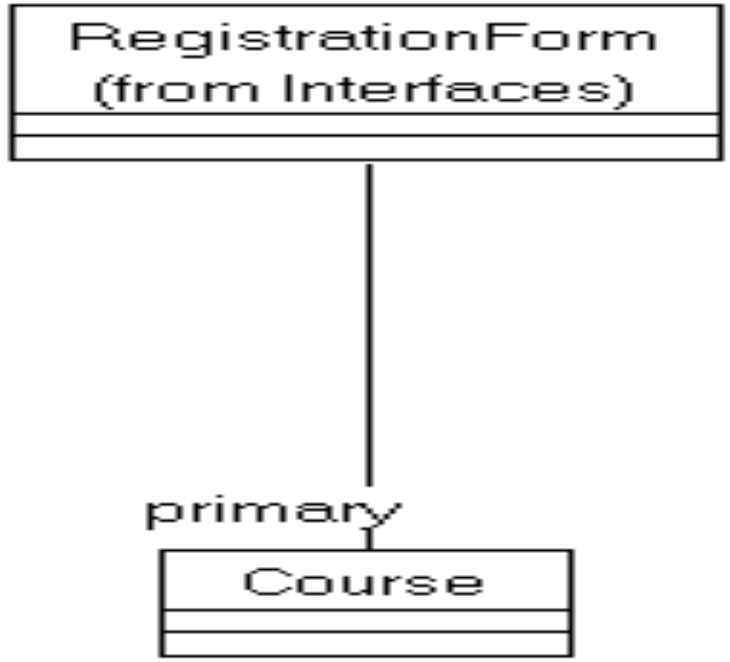
关联名称

- 关联或聚合可以被命名
  - 通常是动词或动词短语



角色名称

- 在类间的关联中角色表示目的或能力
  - 通常是名词或名词短语

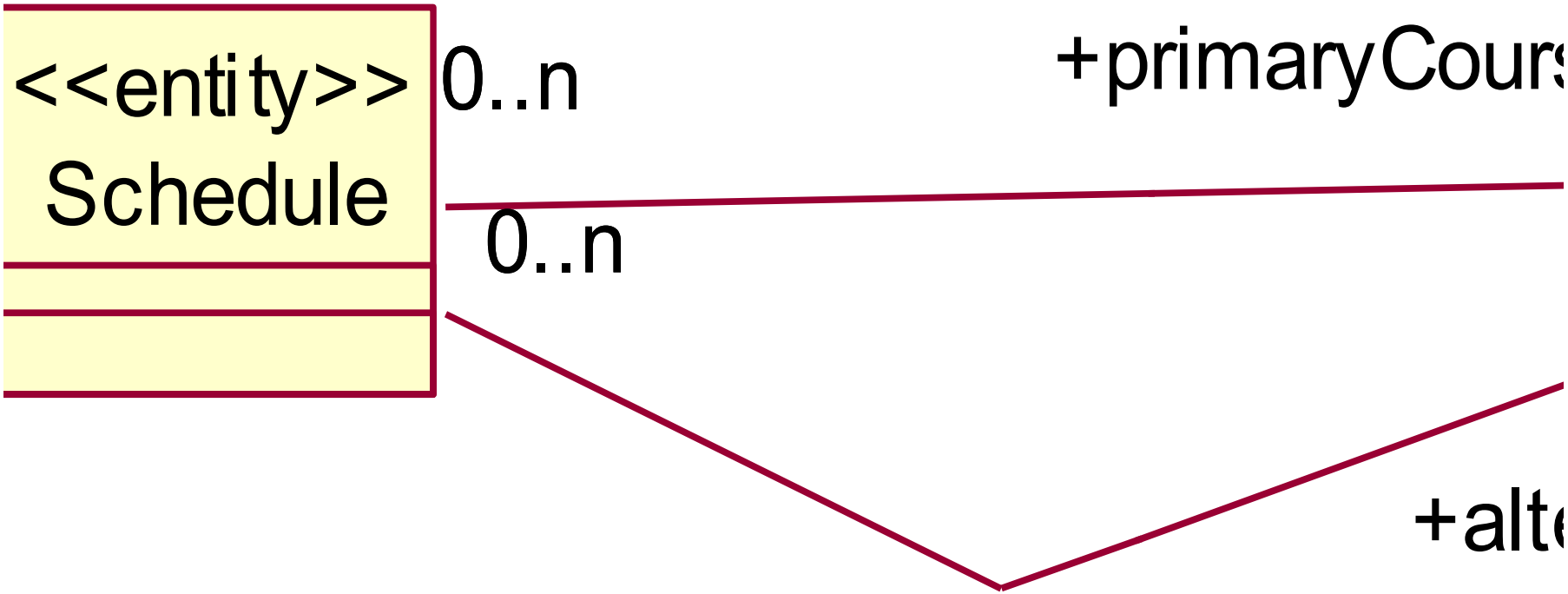


关联度

- 每一个关联和聚合的尾布都包含多种关联度
  - 在关系中指示多个对象的编号

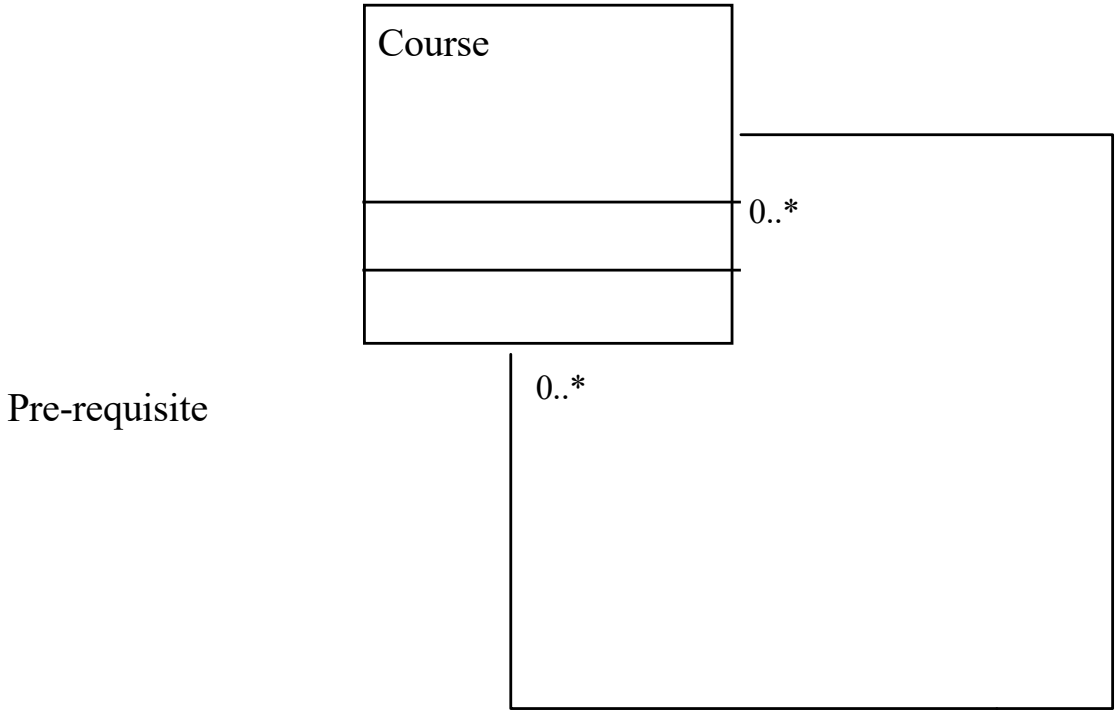
1	只有一
0..*	零或多
1..*	一或多
0..1	零或一
2..7	指定范围

关联度



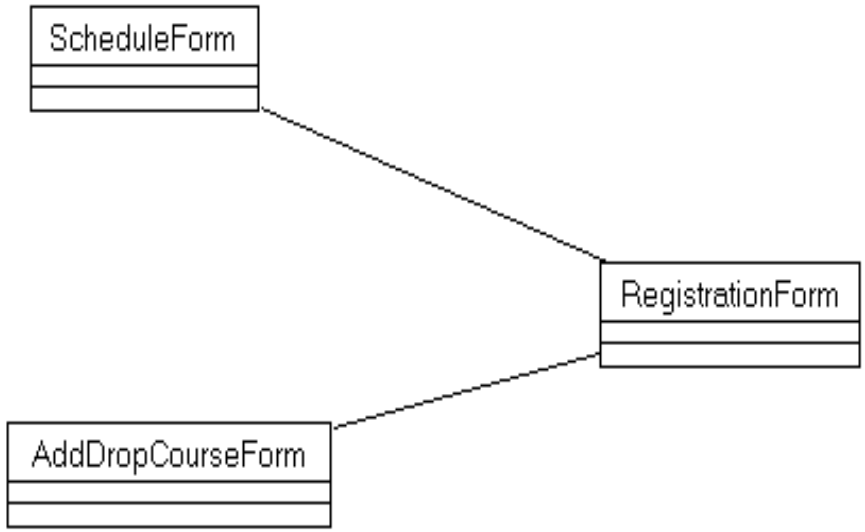
反身关系

- 在反身关系中，同一个类中的多个对象可以有许多合作方式

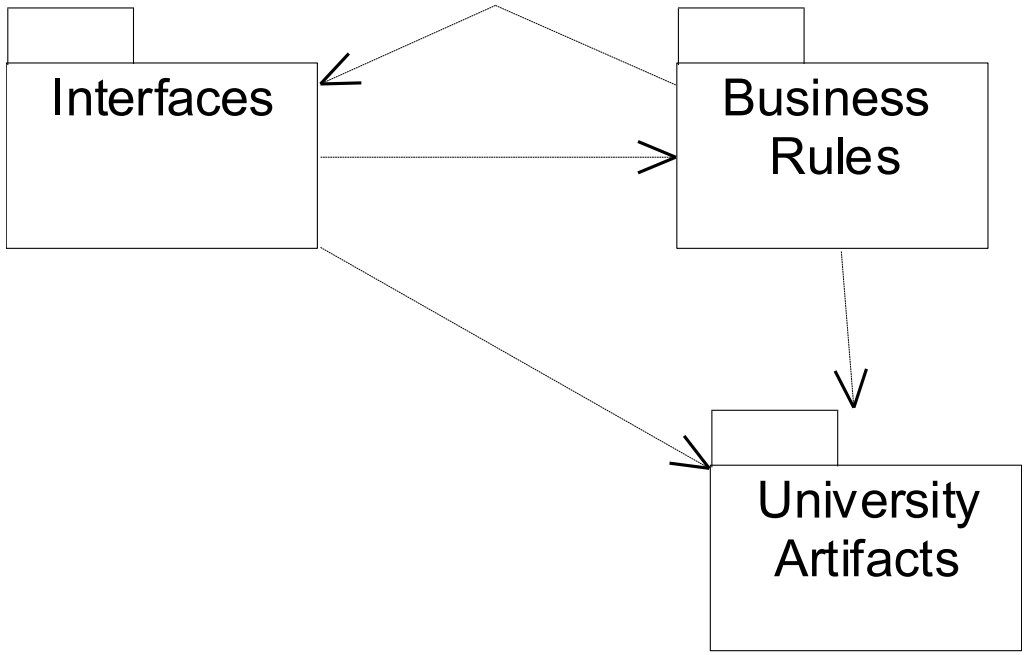


更新类图

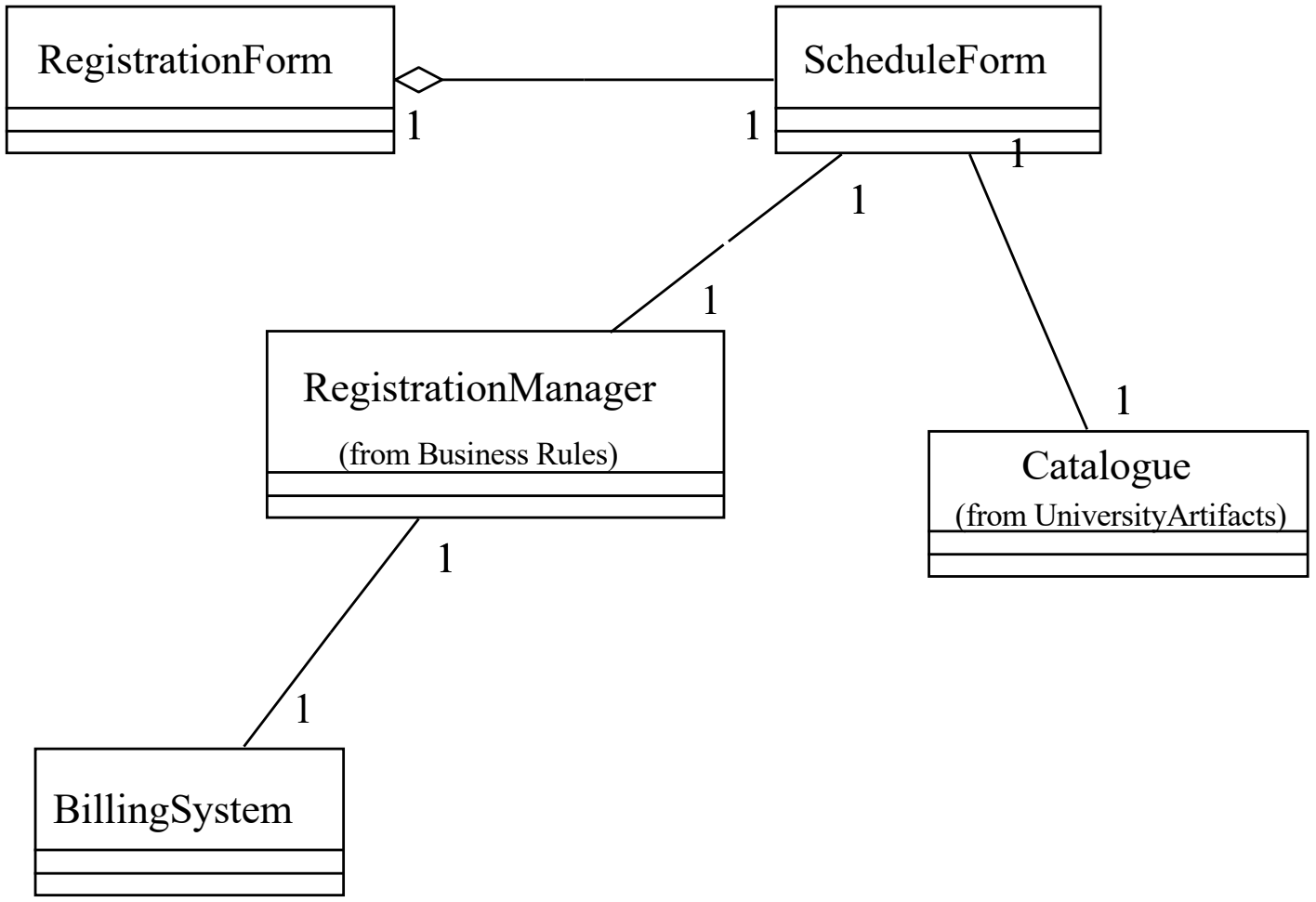
- 一旦关联或聚合被建立，其它类图也可以被更新，以便展现关系



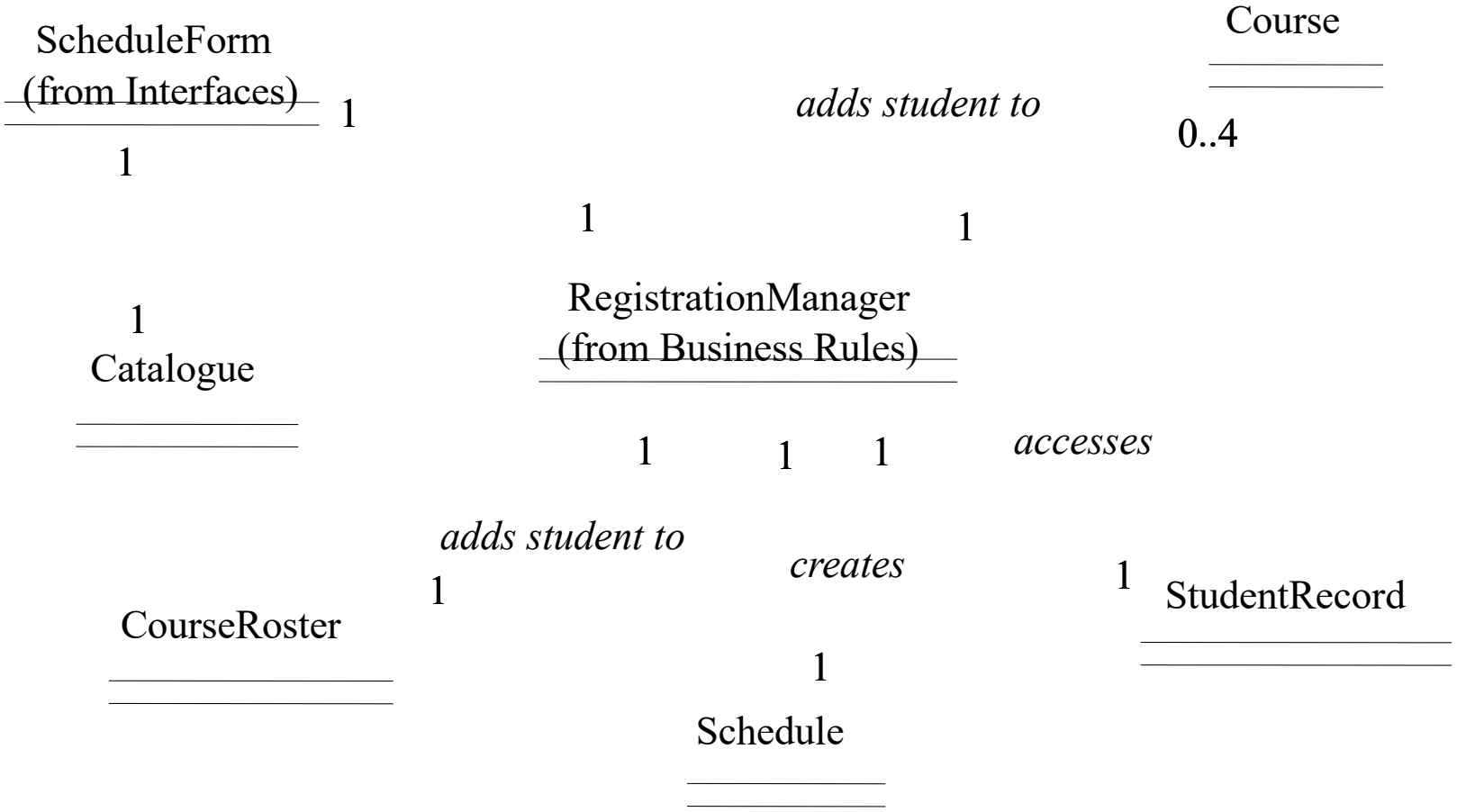
注册系统中的升级类图



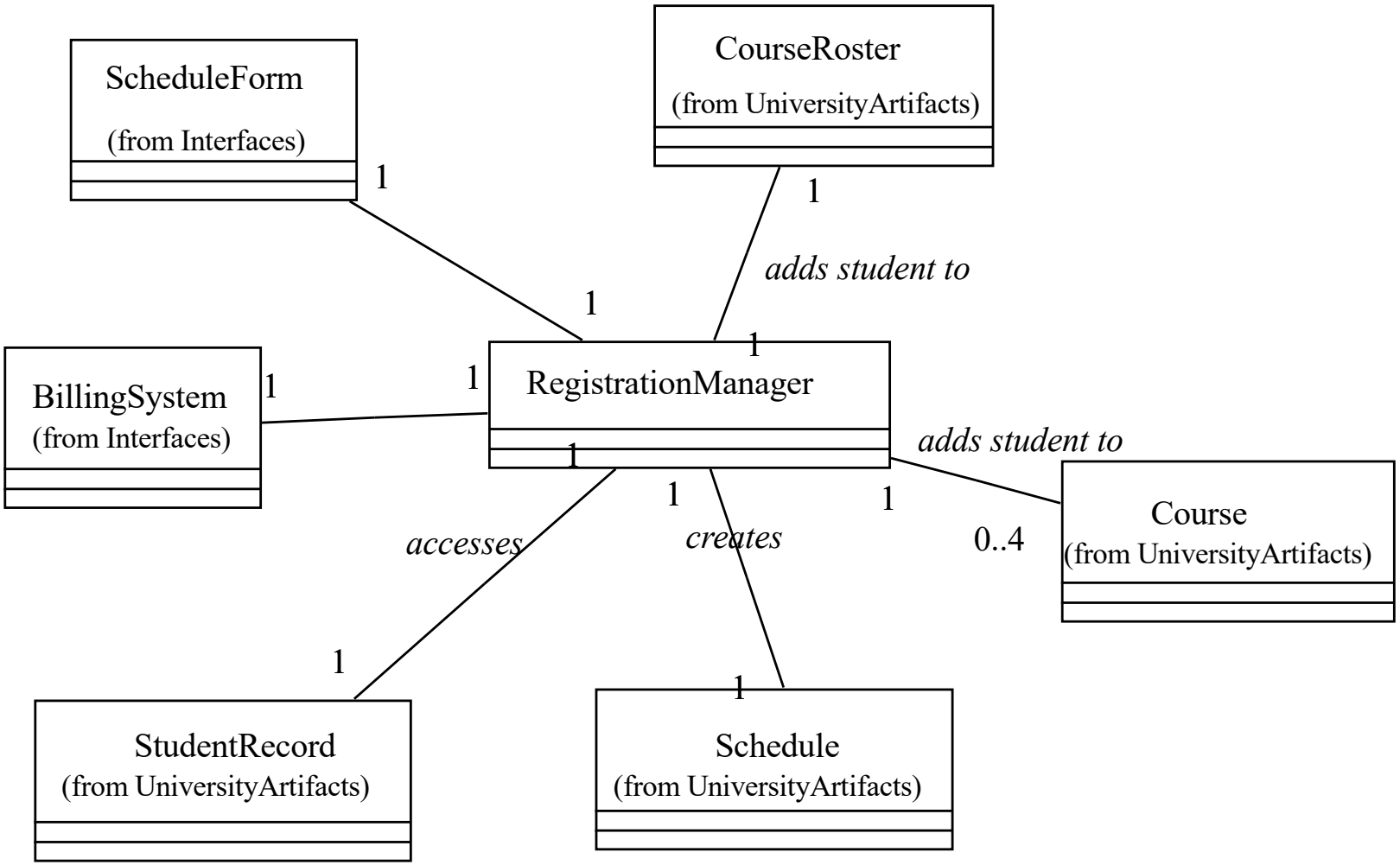
Interfaces包中升级的类图



UniversityArtifacts包中升级的类图



BusinessRules包中升级的类图



操作和属性

- 为类建立操作和属性
- 验证操作和属性
- 在类图上显示操作和属性

什么是操作

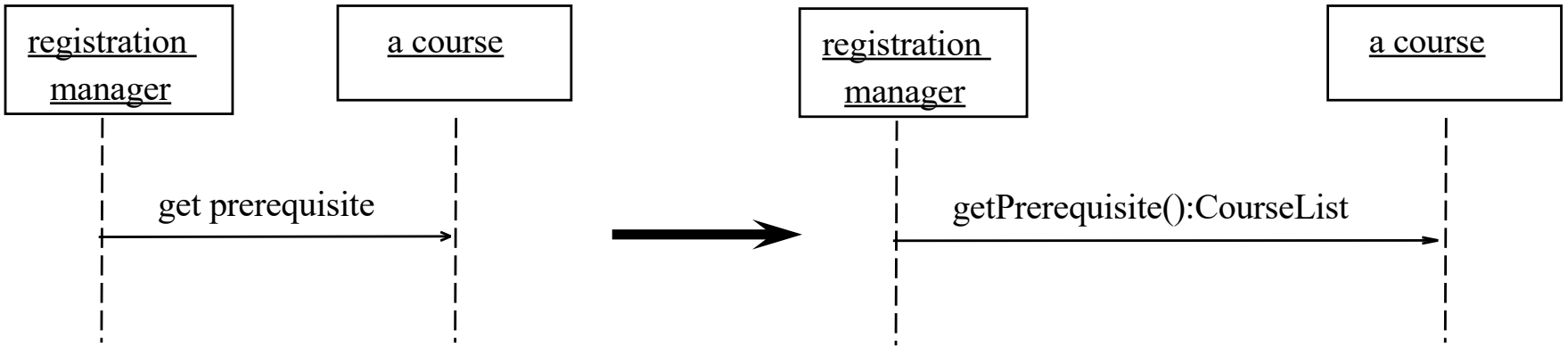
- 类具体表达一套责任，这种责任定义了类中对象的行为
  - 类的责任通过操作被执行
- 操作应该执行一种简单的功能

操作和交互图

- 在序列图或协同图中显示的消息通常是类的操作（消息接收者）
- 从一个边界类发消息到另一个边界类可以通过一个图形用户界面（GUI）来实现，它通常是不成熟的操作
  - 它可以通过GUI建立者的性能被实现

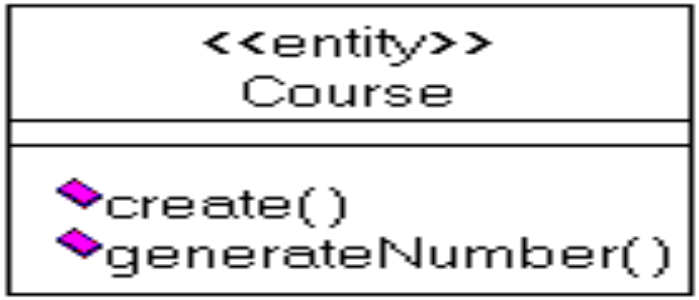
在序列图中将消息映射为操作

- 显示在顺序图或者协作图上的消息通常是接收类的操作
  - 消息转换成操作，加入类图



在类图中建立操作

- 操作可以通过类图建立，也可以通过类的规格说明建立操作

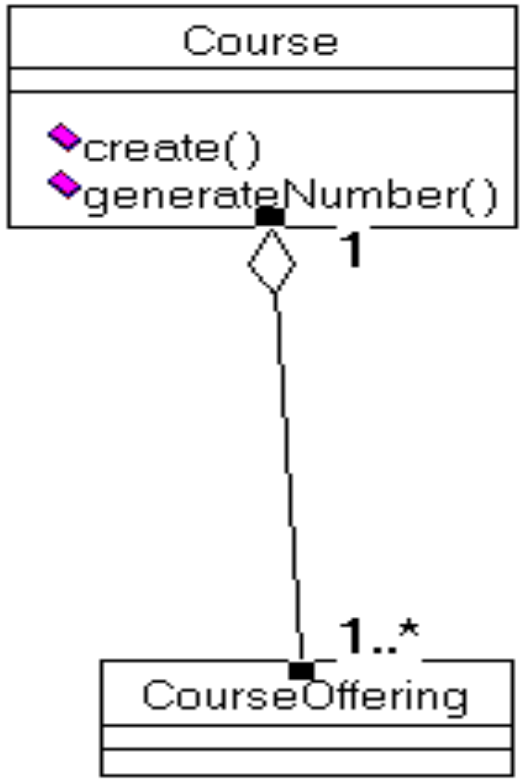


验证操作

- 操作名称应该有一定风格规范
  - 提供跨项目的一致性
  - 引导多个可维护的模块和代码
- 操作的命名应该可以显示它的结果，而不是执行操作后的步骤
  - 例子：getGrade()、instead of calculateGrade()
- 操作应从接受者的愿望命名，而不是发送者
- 每一个操作应该有一个清晰简明的定义

在类图中显示操作

- 操作可以在类图中显示



继承

- 建立一个称为登记用户的超类
- 为登记用户类建立学生信息和教授信息子类
  - 将一个普通的属性或操作移动到新的超类中
- 必要时重新分配关系
- 必要时加入强制信息

对象行为

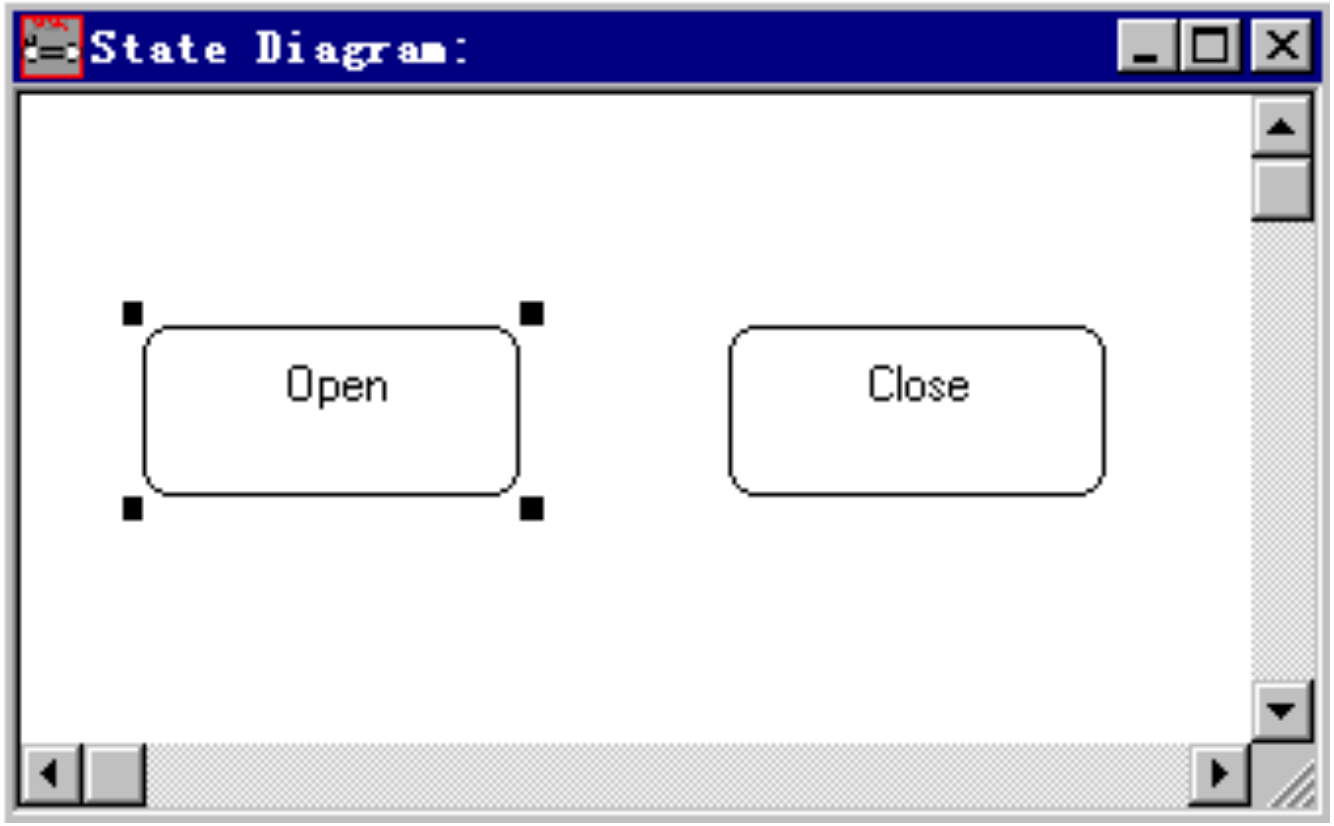
- 建立状态转换图
  - 状态
  - 转换
  - 动作和活动
  - 嵌套状态

什么是状态机图

- 状态机图用于描述给定类的发展历史，导致状态转换的事件和导致状态改变的活动
- 对象状态是对象可以存在的可能条件
- 为类的重要动态行为建立状态机图

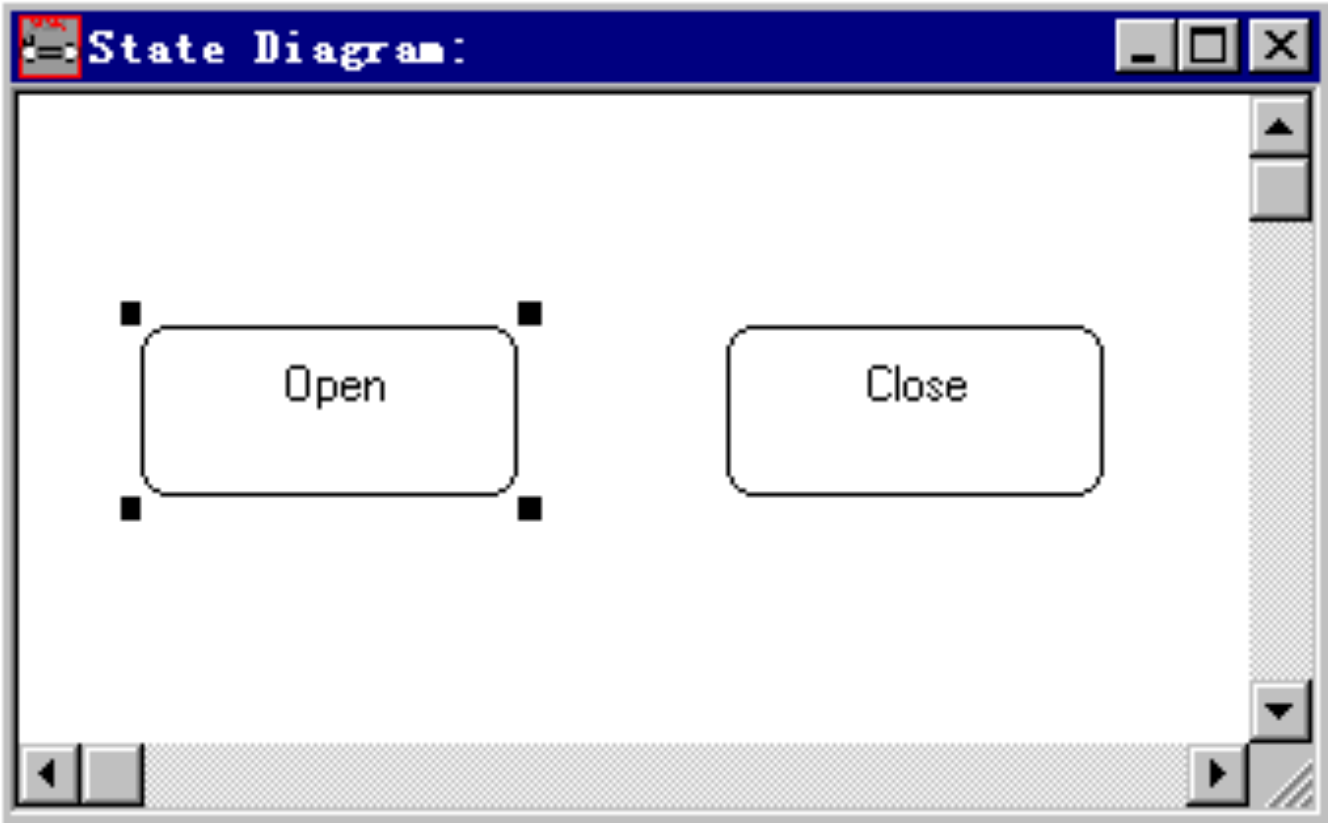
什么是状态

- 状态是对象可以存在的可能条件



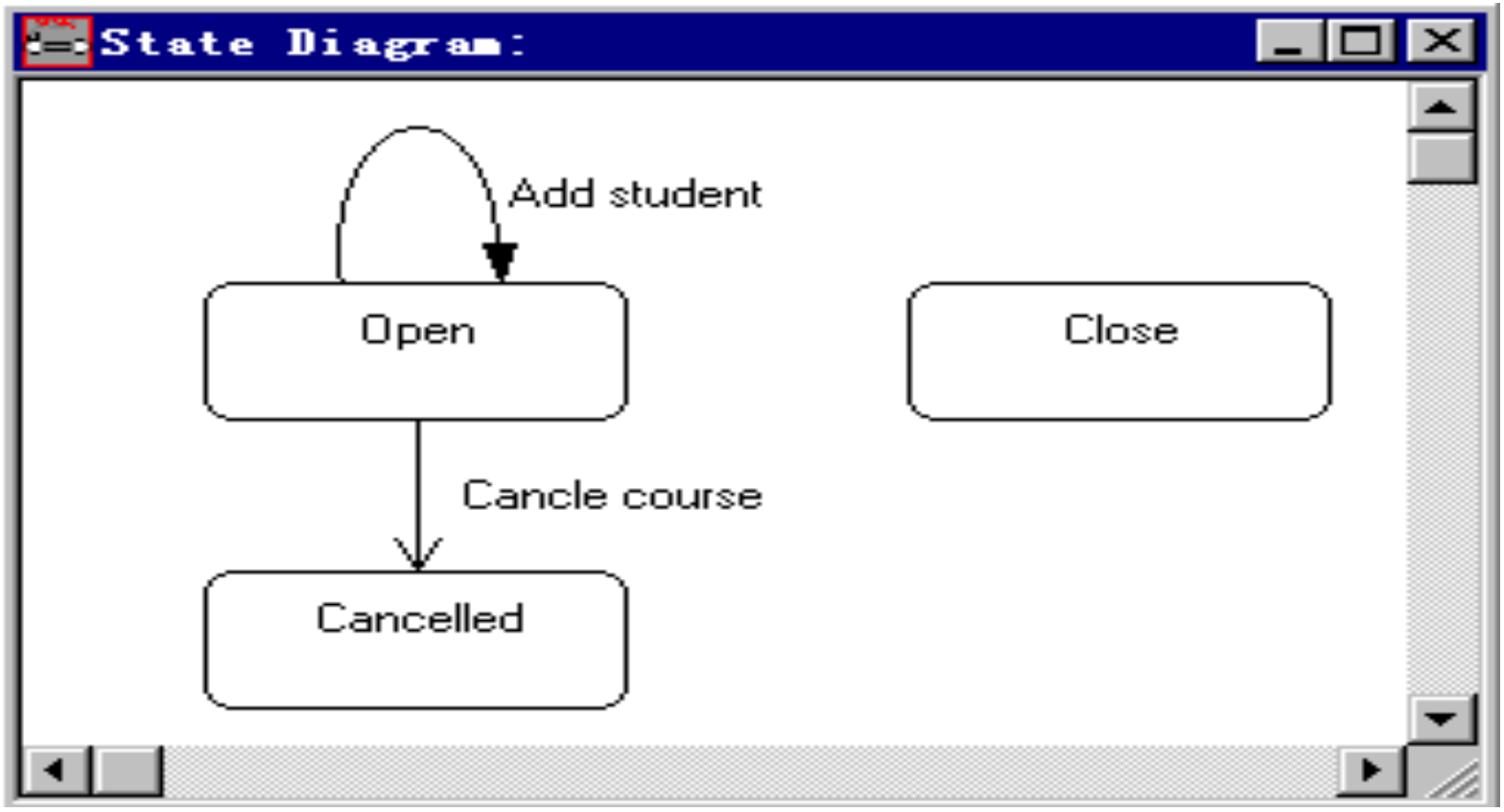
状态转换

- 状态转换是从最初状态到成功状态的改变



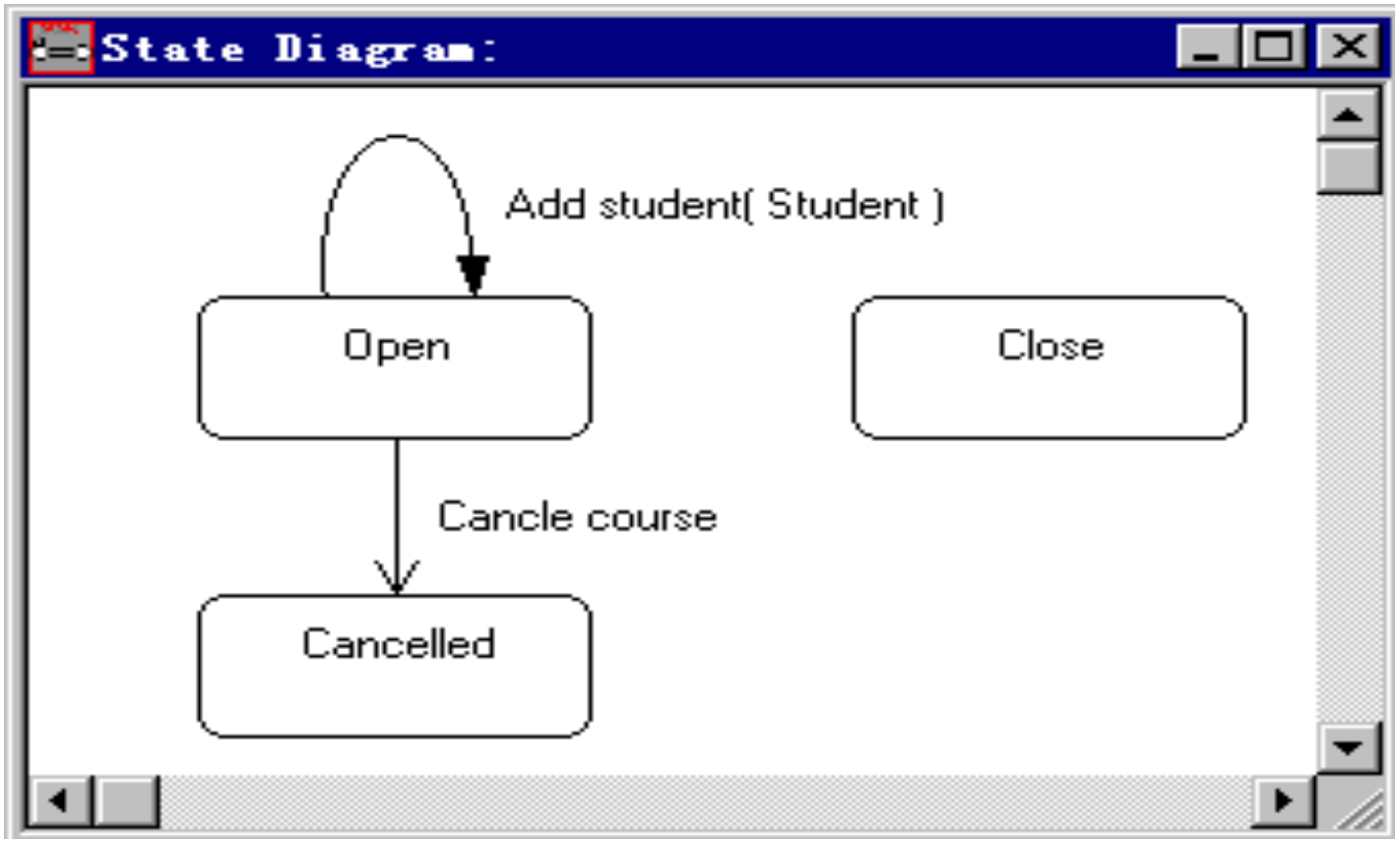
反身状态转换

- 反身状态转换是一种初始状态等于成功状态的转换



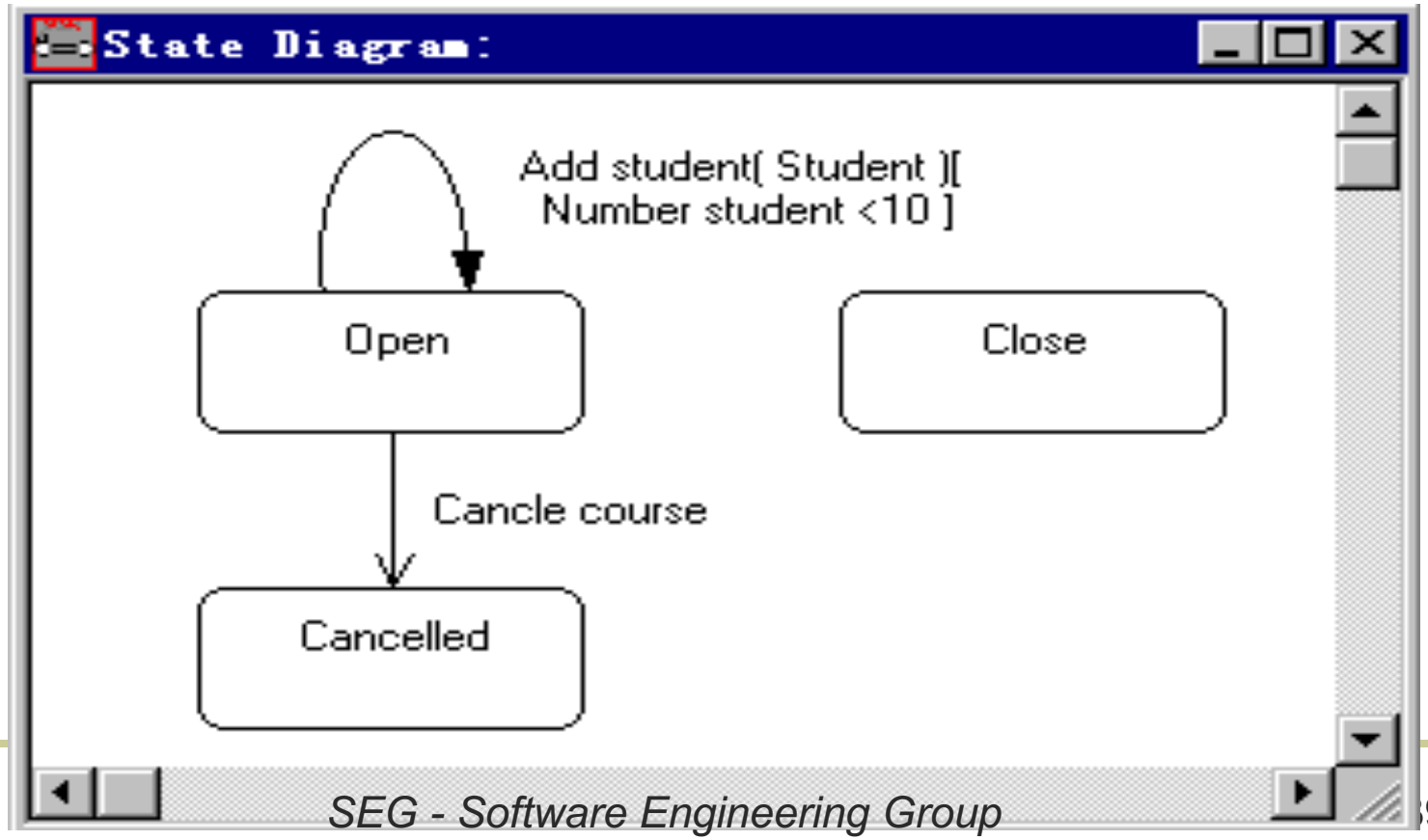
状态转换Arguments

- 伴随一个事件的数据就是一个argument



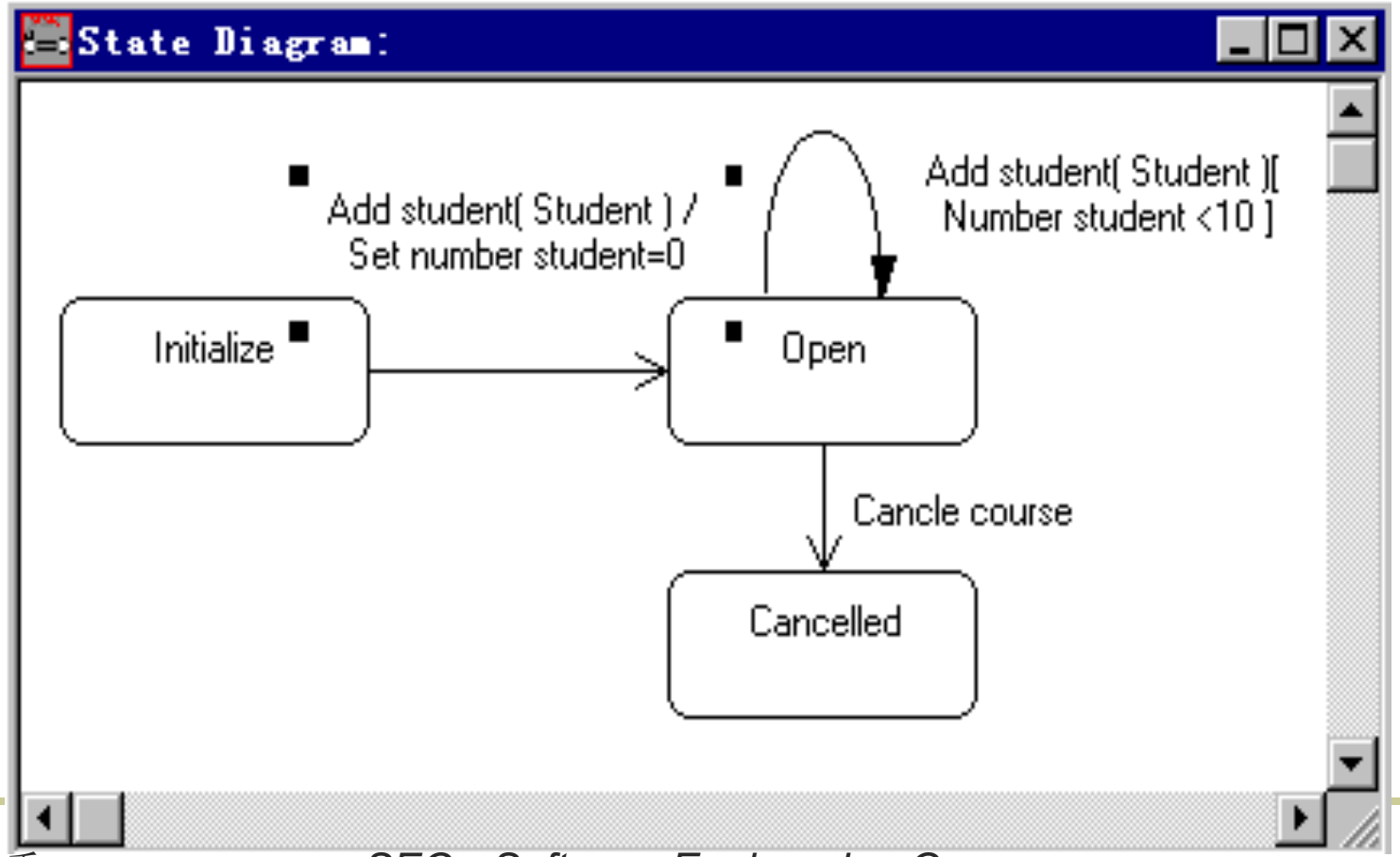
卫式条件（Guarded)状态转换

- 通过卫式条件（guard）的使用，转换可以形成条件



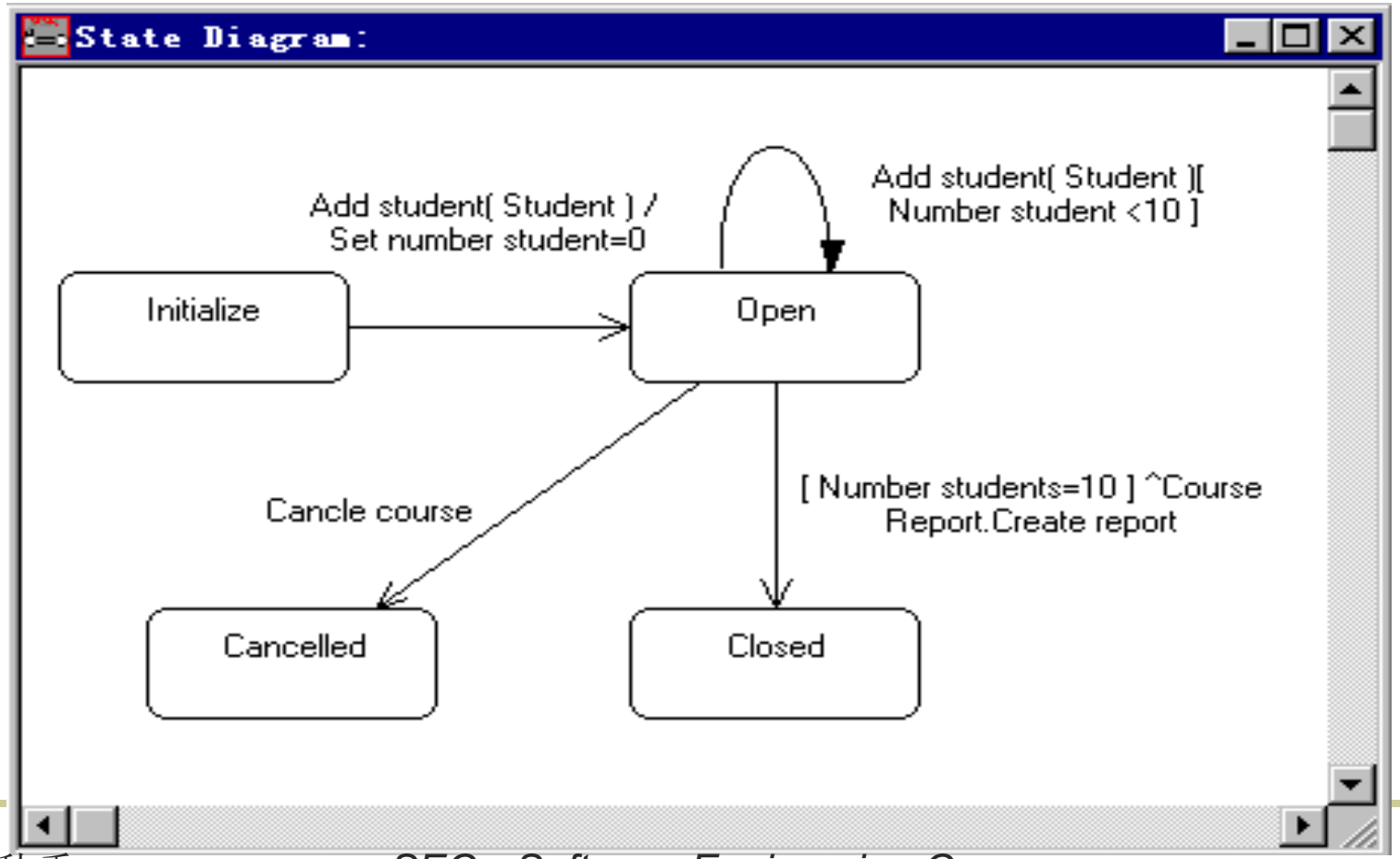
活动

- 活动是伴随事件转换的操作



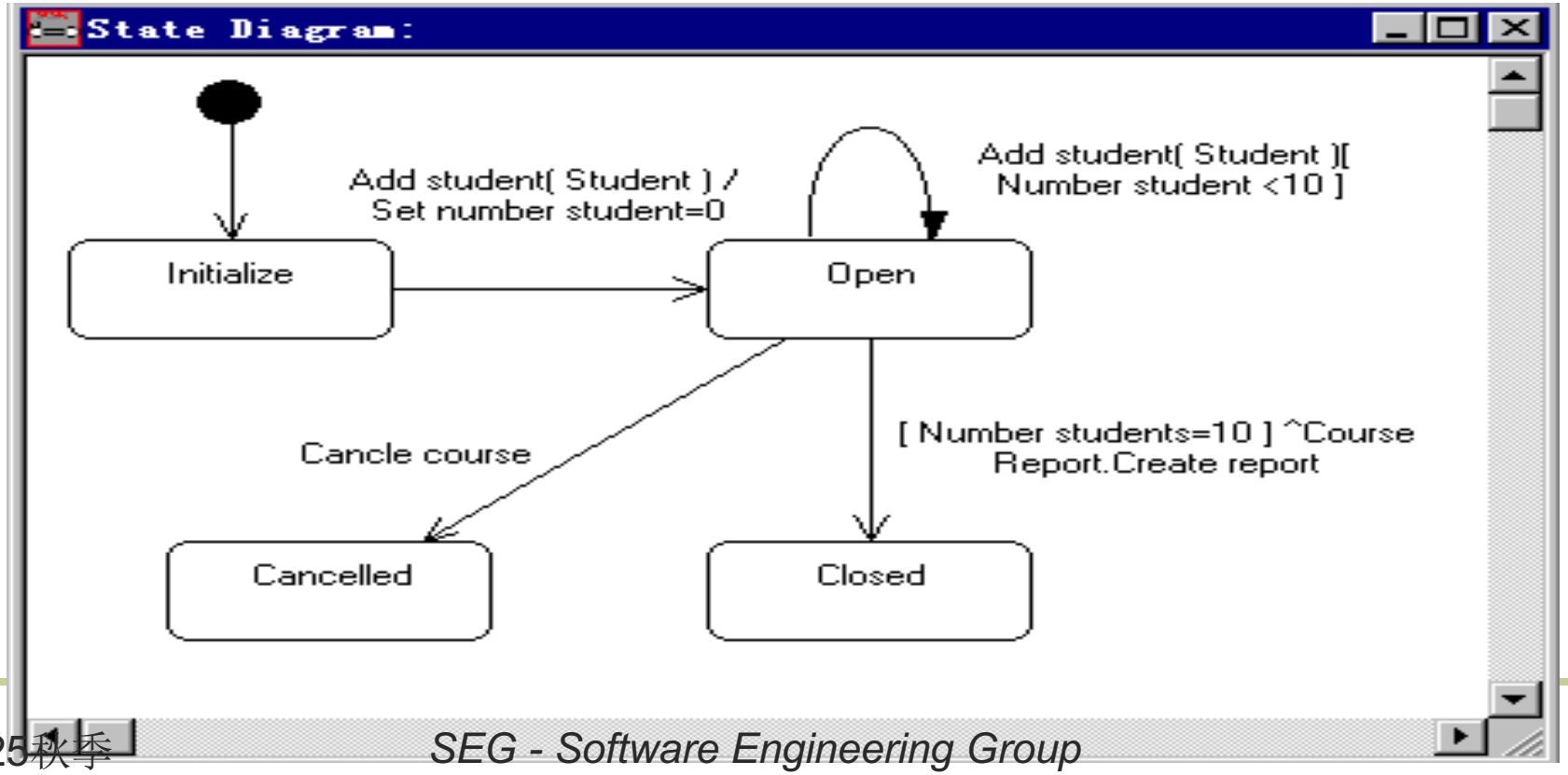
发送事件

- 事件可以触发传送另一个事件



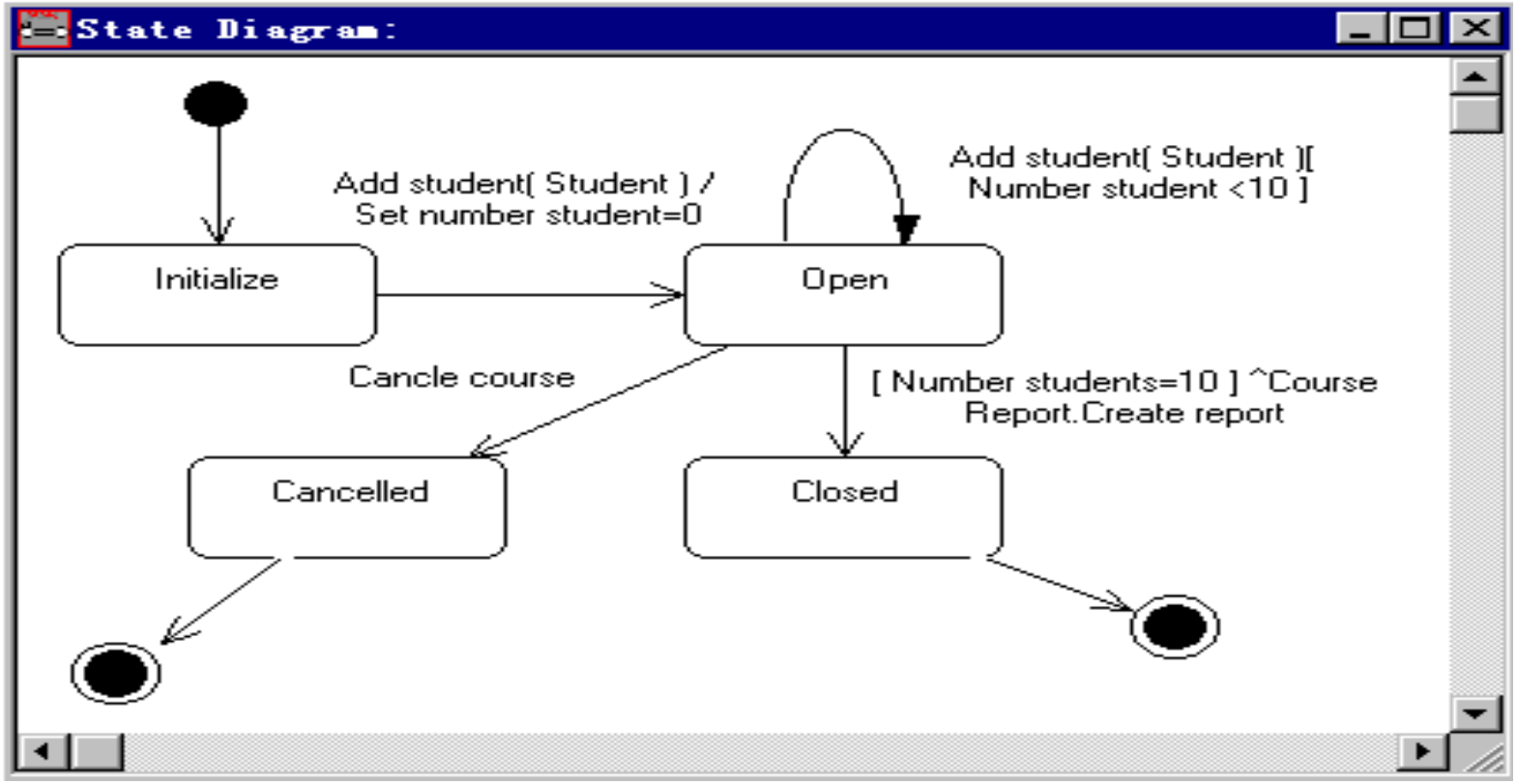
起始状态

- 起始状态是对象的最初状态
  - 只能有一个起始状态



终止状态

- 终止状态是对象最后的状态
  - 可以没有终止状态，也可以存在多个终止状态

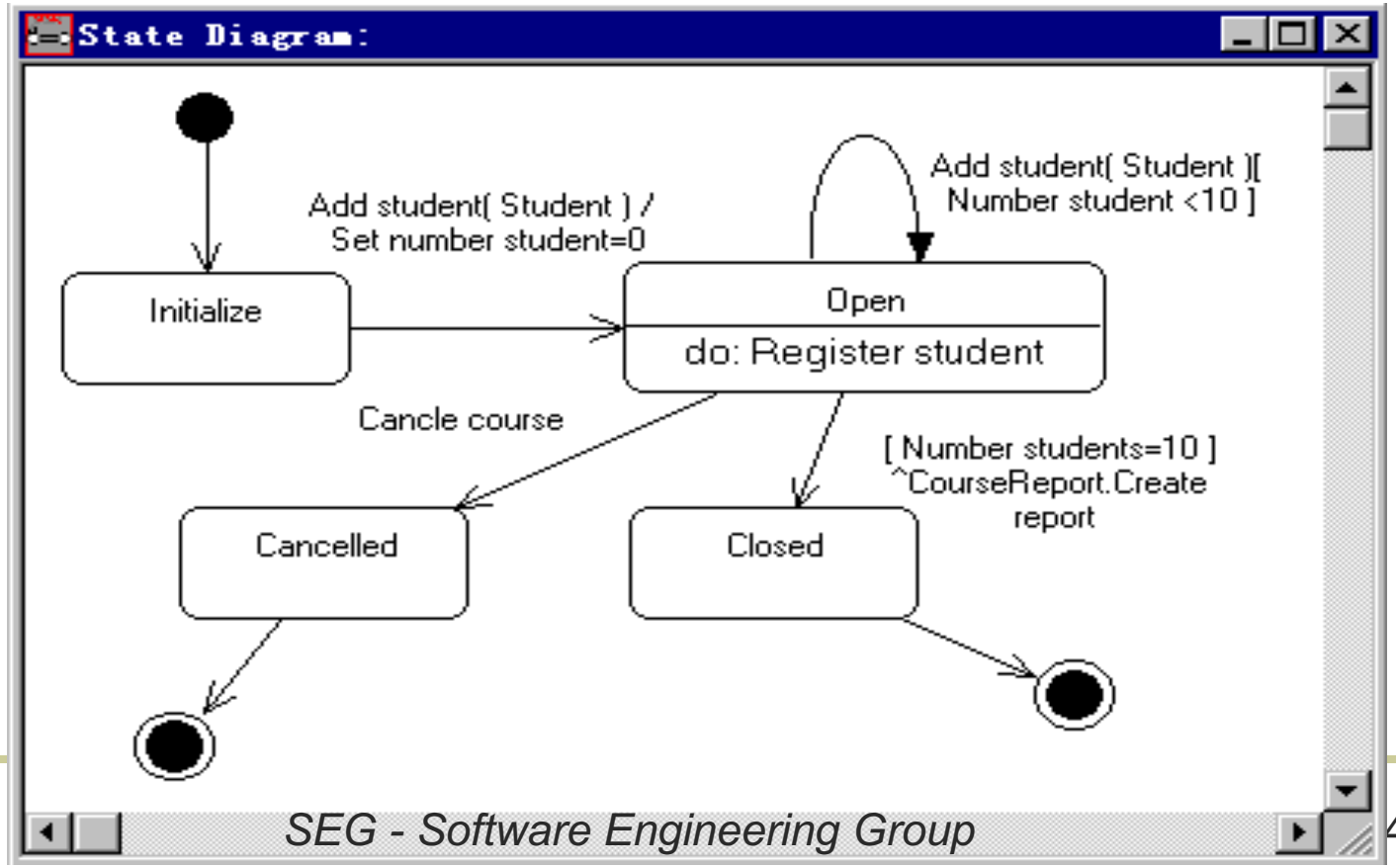


状态活动类型

- 简单状态
  - 用自由格式文本代表发生的事件
- 发送事件
  - 一个活动出发下一个事件

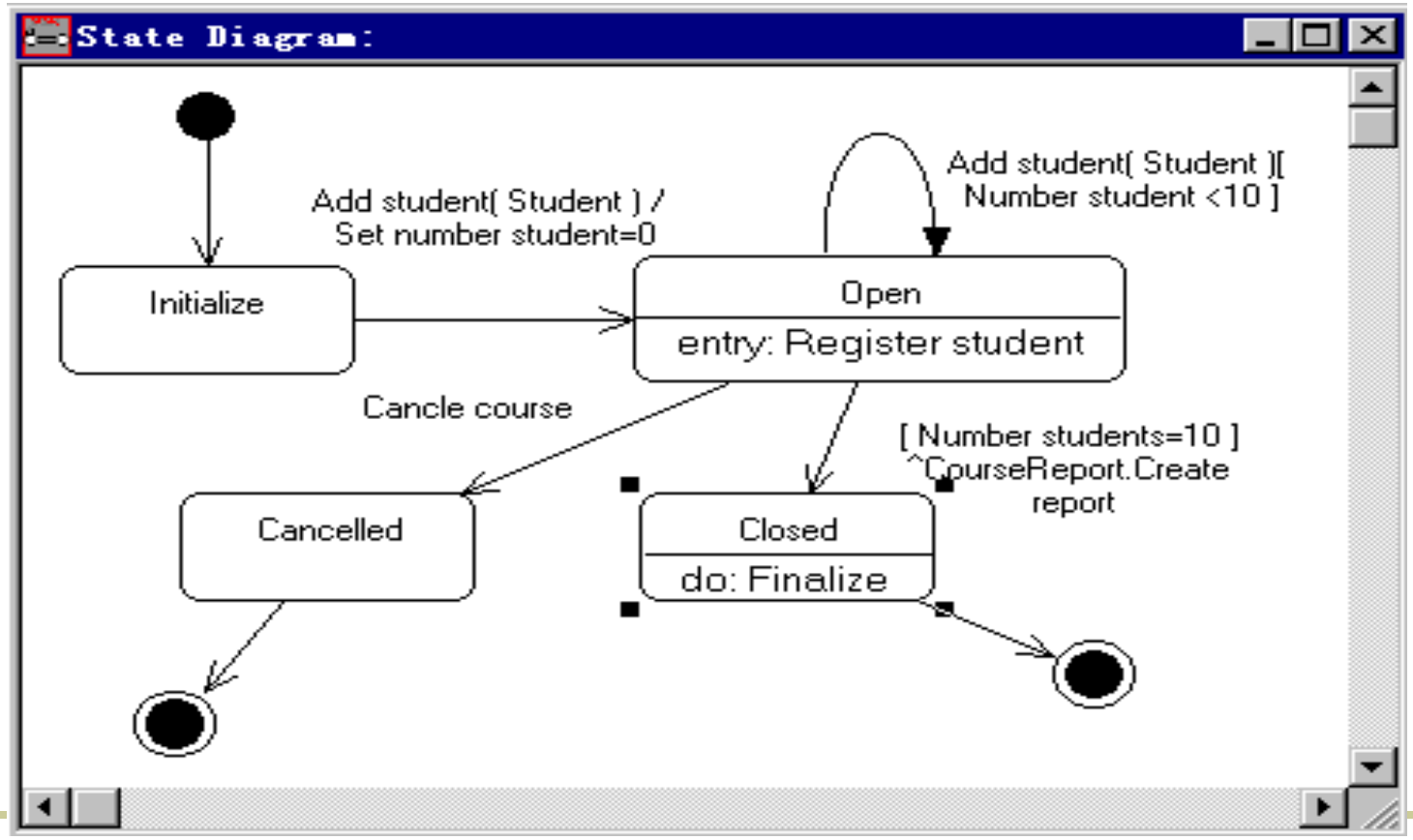
状态中的活动

- 通过关键词的输入，活动被放置在先前状态中



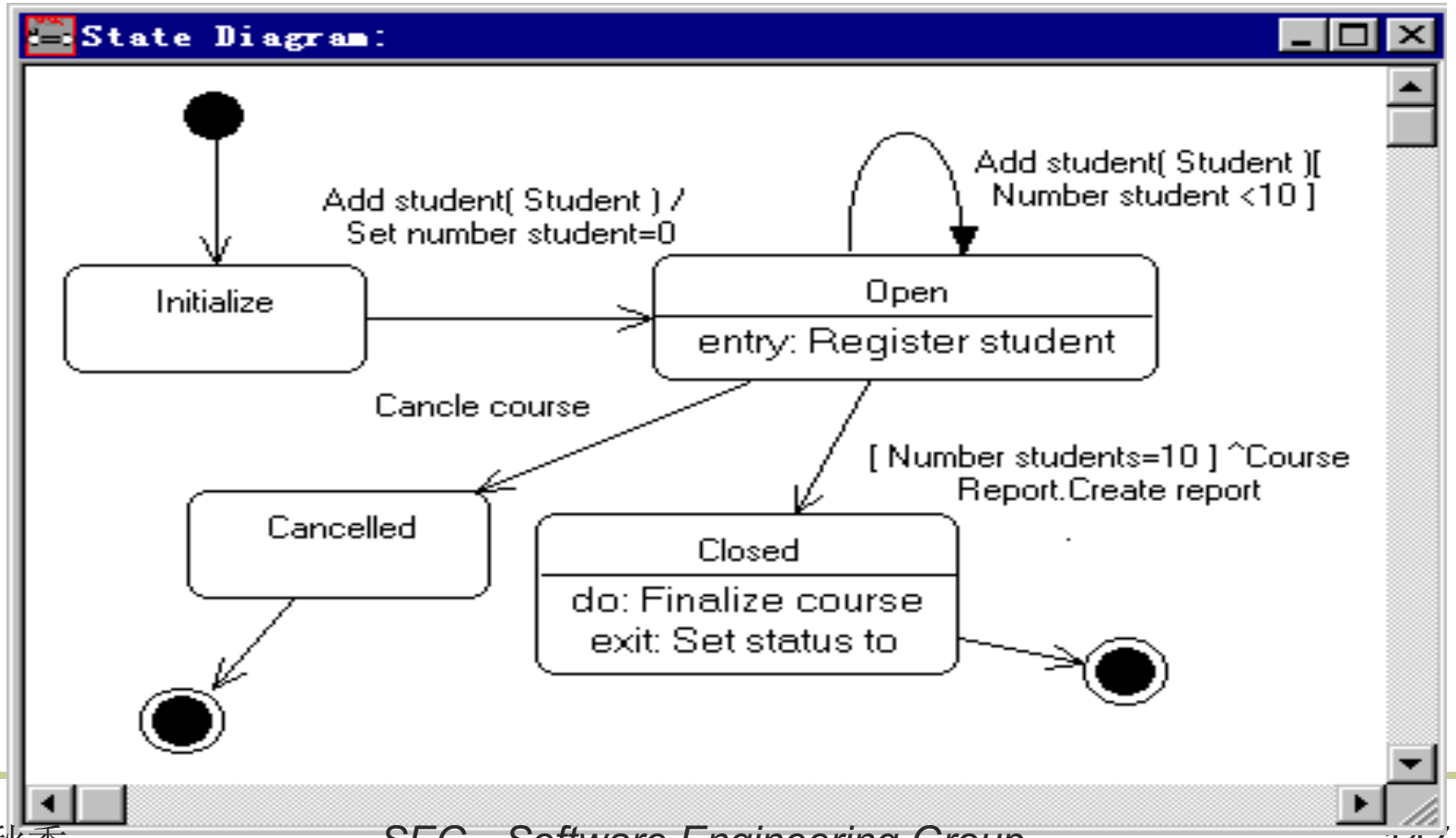
活动被输入直到从状态中退出

- 通过关键词do，活动被放置在先前的状态中



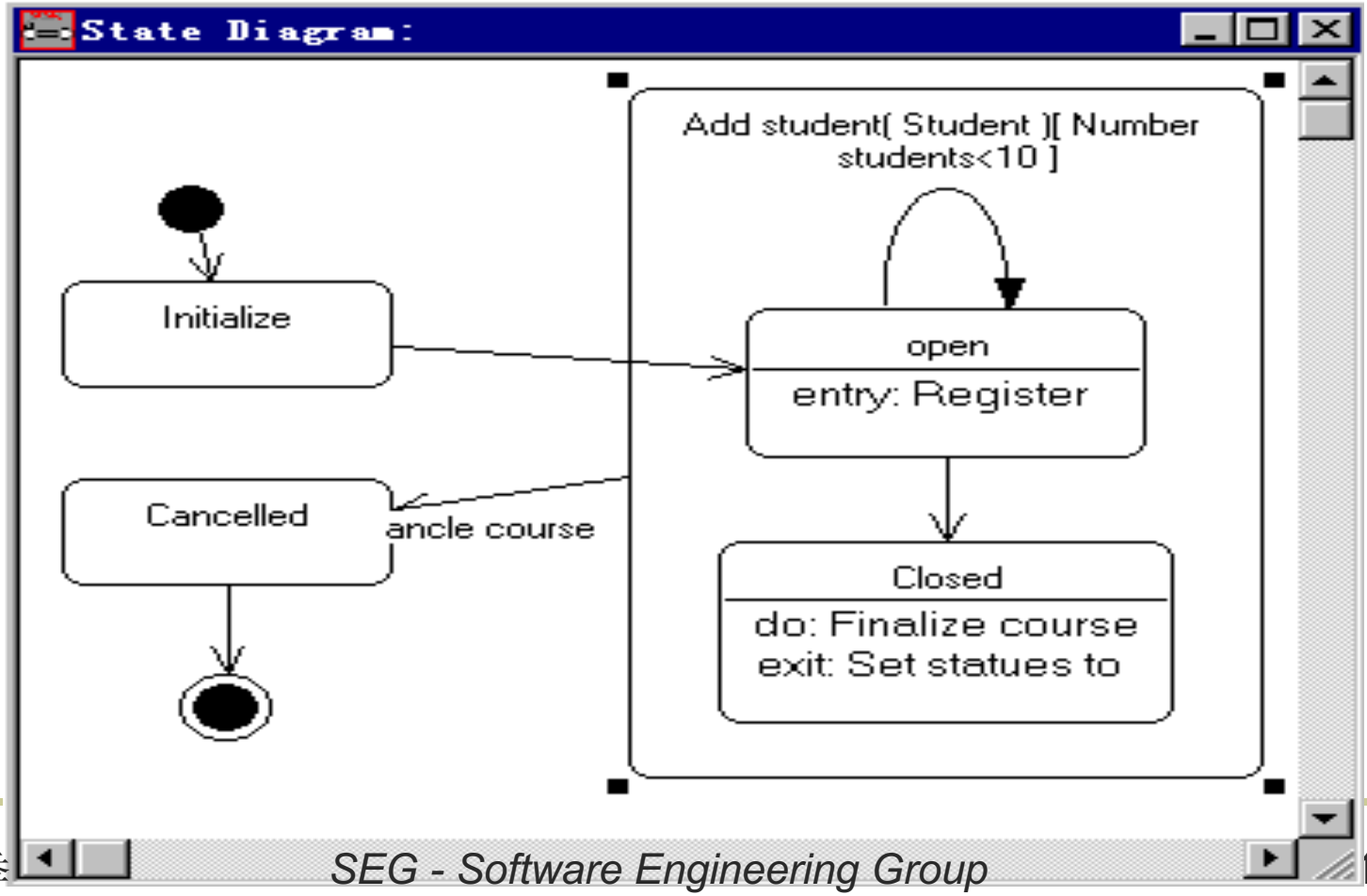
活动从状态中退出

- 通过输入关键词exit，活动被放置在先前状态中

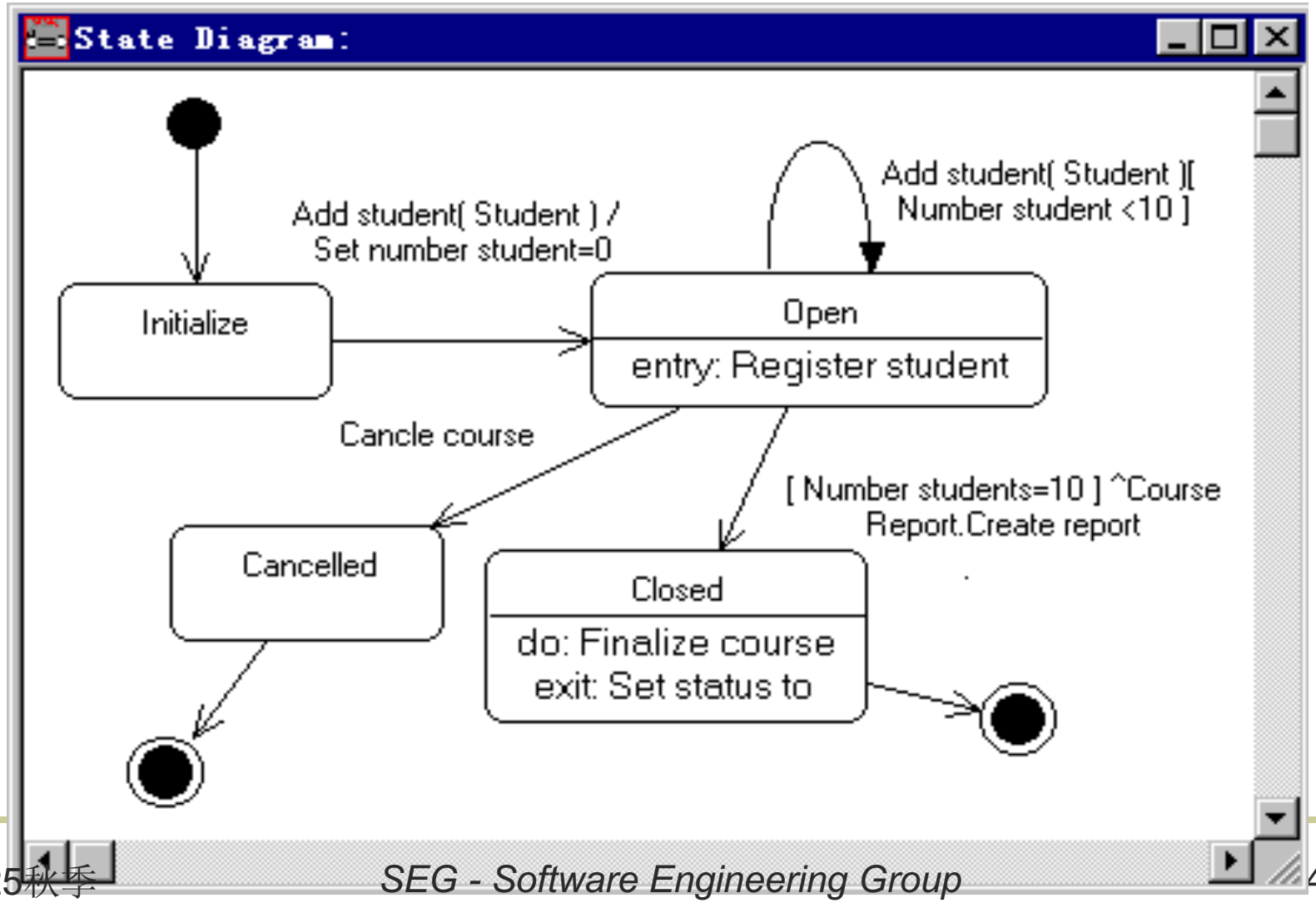


嵌套状态

- 嵌套状态可以用于将复杂的图形简单化



待选课程类的状态图


State Diagram:
Initialize
Add student(Student) /
Set number student=0
Open
entry: Register student
Add student(Student)[
Number student <10]
Cancle course
[Number students=10] ^Course
Report.Create report
Cancelled
Closed
do: Finalize course
exit: Set status to

活动图(Activity diagram)

- 活动图是用状态机对工作流进行建模的特殊形式，它和流程图很类似，不过它支持并发控制。
- 活动图一般不描述所有的运算细节，它显示活动的流，但不显示执行活动的对象。
- 活动图处于系统的外部和内部视图之间，所以它可以作为设计的起点，为了完成设计，每个活动必须扩展成一个和多个操作，每个操作被指派给特定的对象来实现。
- 将不同对象控制的活动划分在一起，这类划分可以通过分隔的区域来表达，由于它们的外观，每个区域称为泳道（swimlane）。

活动图(Activity diagram)

- 活动图由活动和转移组成。
- 活动图中的菱形框是判断标志，表示条件转移。
- 活动图对表示并发很有用。在活动图中使用一个称为同步条的水平粗线可以将一条转移分为多个并发执行的分支，或将多个分支合为一条转移。此时，只有输入的转移全部有效才能执行后面的活动。

启发性原则

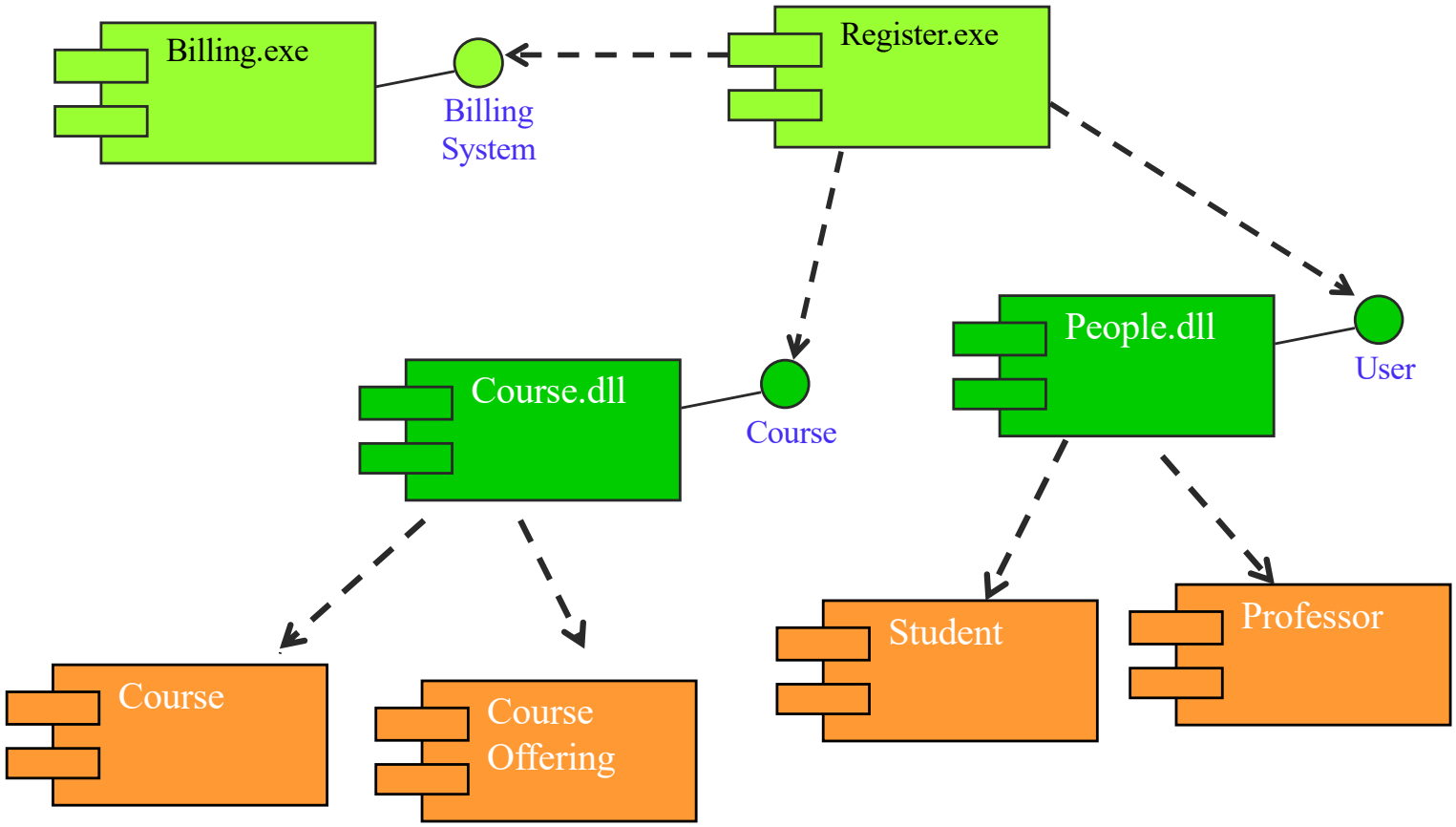
顺序图，活动图，状态机图共同描述了系统的动态特征，使用时有以下注意点：

- 不要为系统中每个类都画状态图，正确做法是：为了帮助理解而画状态图
- 状态图不适合表现多个对象的合作
- 使用顺序图和合作图时，参与交互的对象不应该太多，否则会失去清晰性
- 如果要描述多线程等复杂行为，使用活动图
- 顺序图和合作图不适合对行为进行精确定义，此时应该使用状态图

组件图(Component diagram)

- 组件图描述可重用的系统组件以及组件之间的依赖。组件可以是源代码组件、二进制代码组件和可执行代码组件。组件可以存在于编译、链接、执行等多个场合。

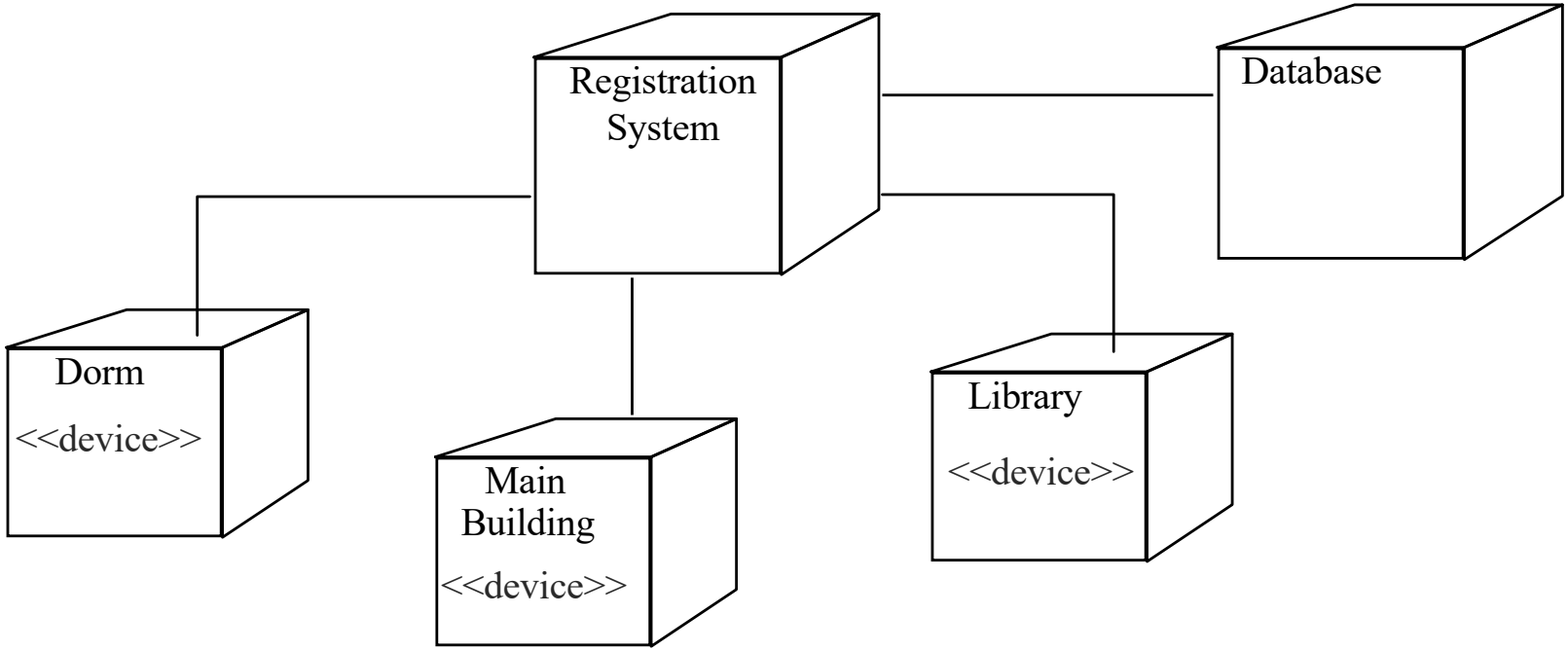
组件图(Component diagram)



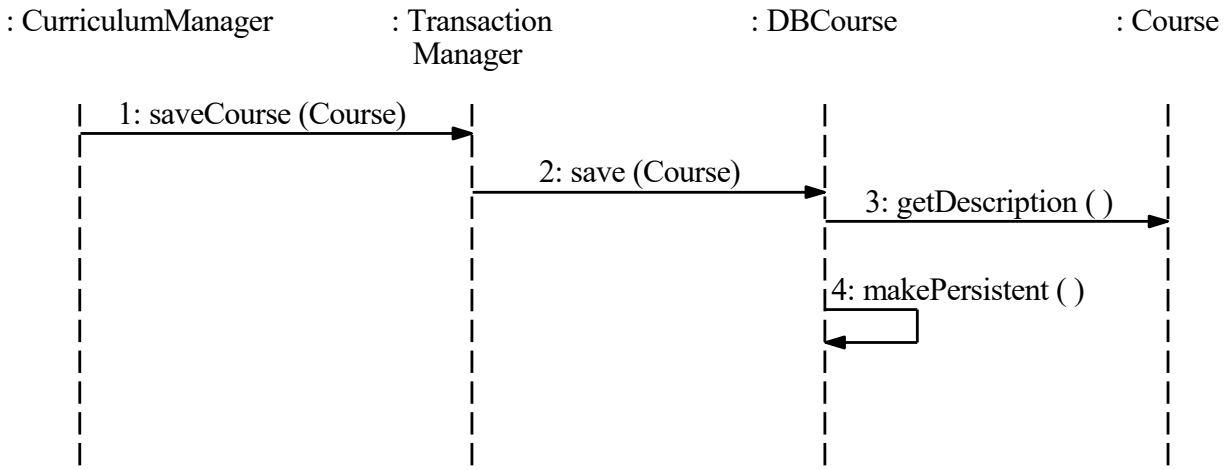
部署图(Deployment diagram)

- 部署图描述系统资源在运行时的物理分布。系统资源称为节点。
- 部署图显示网络上的所有节点、节点间的连接和每个节点上运行的进程。
- 处理器：任何具有处理功能的机器，如服务器，工作站。处理器用边框为黑色的立方体表示。
- 设备：没有处理功能的机器，如打印机，扫描仪。设备用边框为白色的立方体表示。

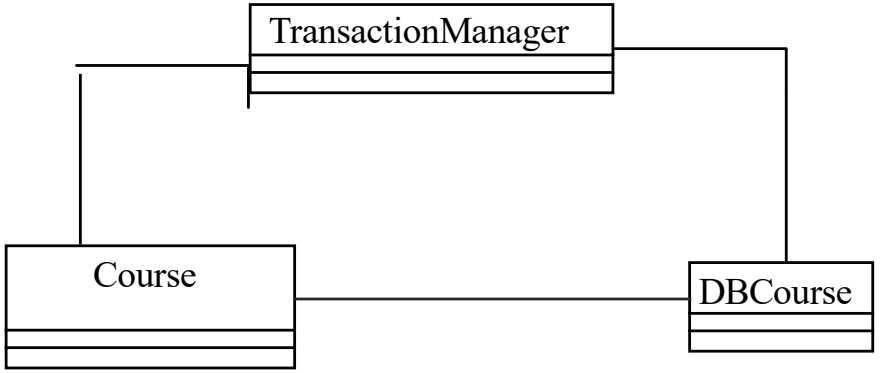
课程注册系统的部署图


Registration System
Database
Dorm
<<device>>
Main Building
<<device>>
Library
<<device>>

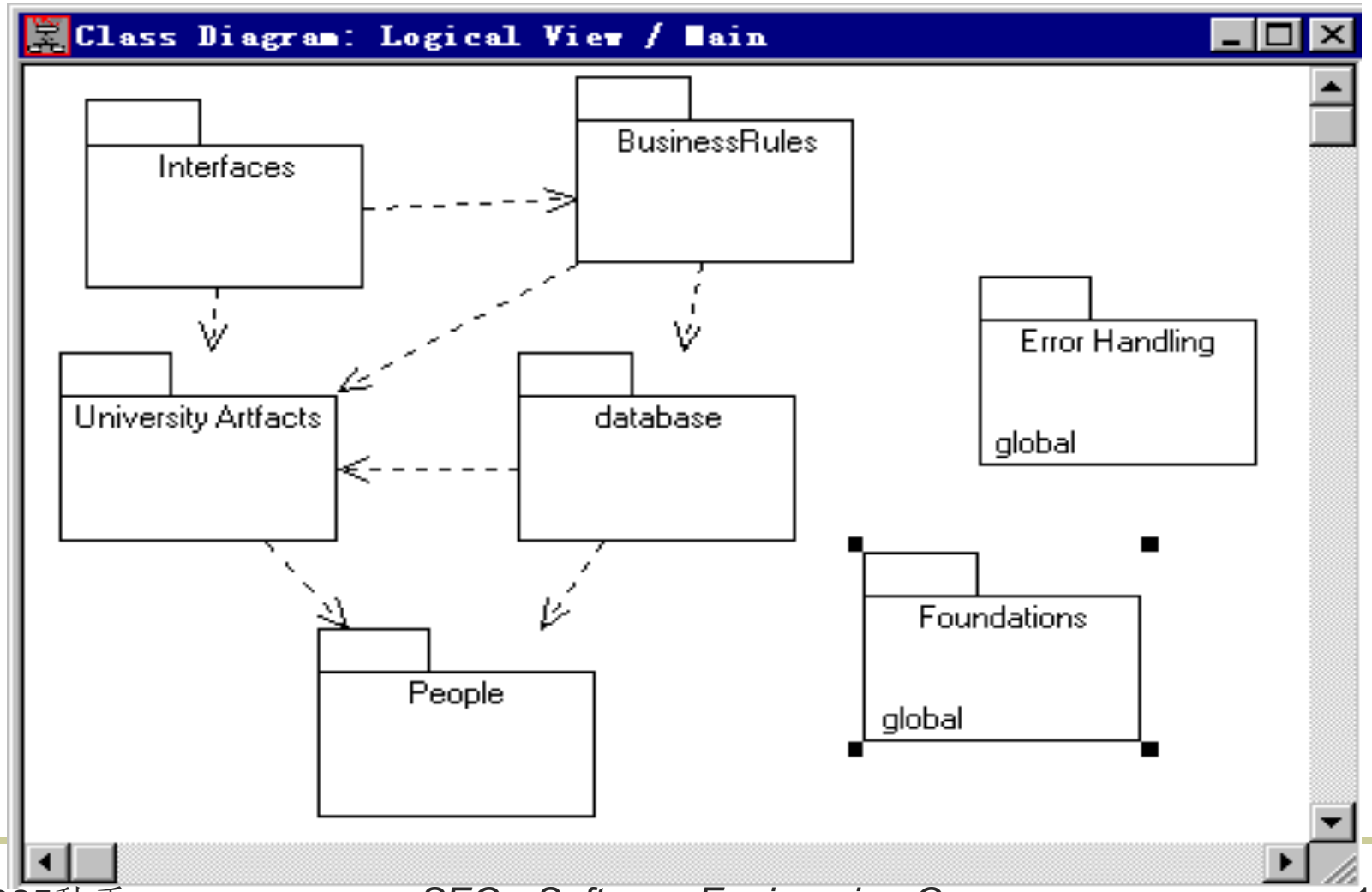
分割数据库的行为



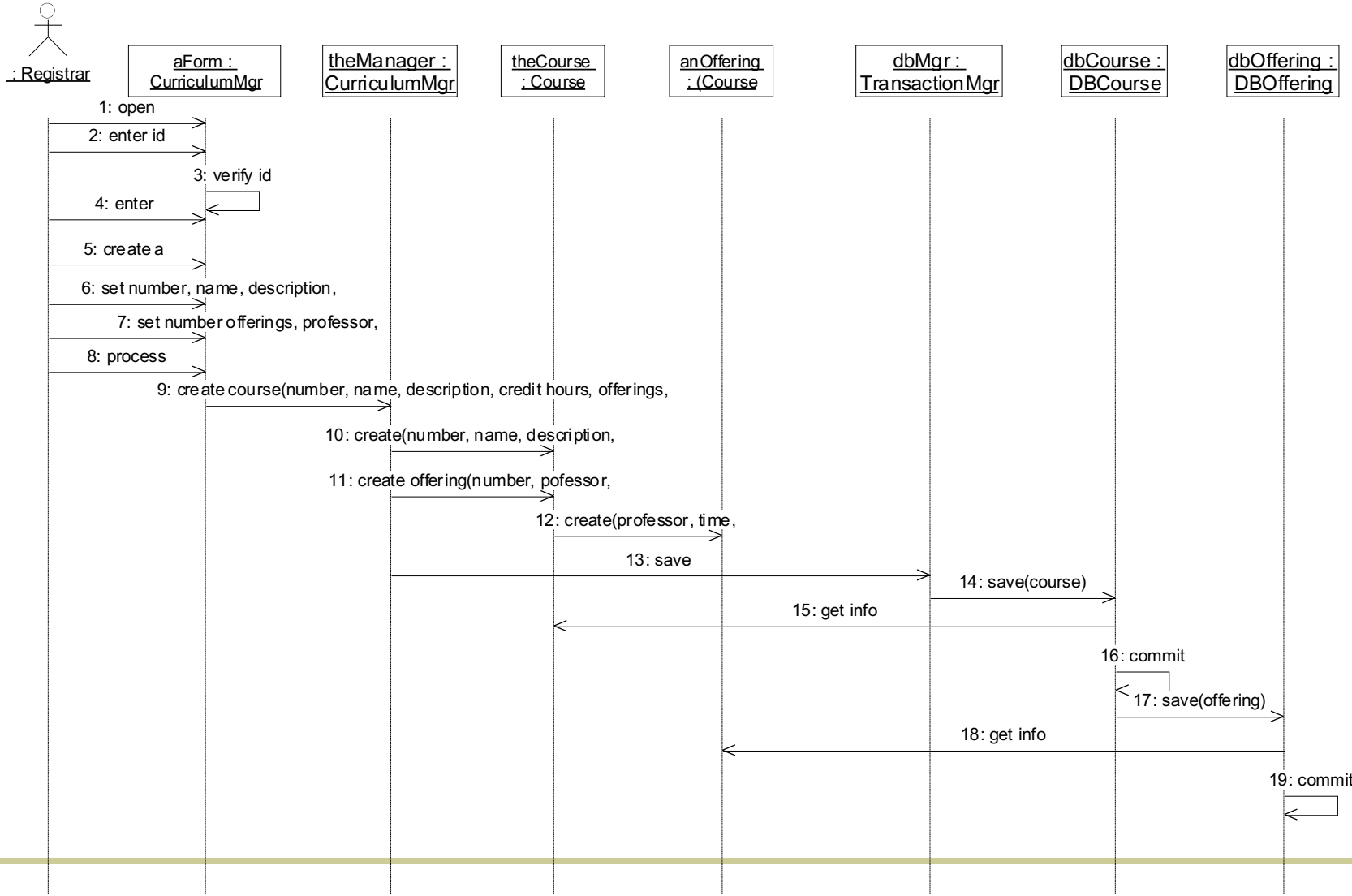
The TransactionManager separates the logical (Course) from the physical (DBCourse)



有数据库参与的类图


Class Diagram: Logical View / Main
Interfaces
BusinessRules
Error Handling
global
University Artfacts
database
Foundations
global
People

有数据库的顺序图


: Registrar
aForm : CurriculumMgr
theManager : CurriculumMgr
theCourse : Course
anOffering : (Course
dbMgr : TransactionMgr
dbCourse : DBCourse
dbOffering : DBOffering
1: open
2: enter id
3: verify id
4: enter
5: create a
6: set number, name, description,
7: set number offerings, professor,
8: process
9: create course(number, name, description, credit hours, offerings,
10: create(number, name, description,
11: create offering(number, pofessor,
12: create(professor, time,
13: save
14: save(course)
15: get info
16: commit
17: save(offering)
18: get info
19: commit

内置的数据类型

Course
-description : char* -name : char* -creditHours : short -maxStudents : short

```
class Course {
  public:
  …
  private:
  char *description;
  char *name;
  short creditHours;
  short maxStudents;
};
```

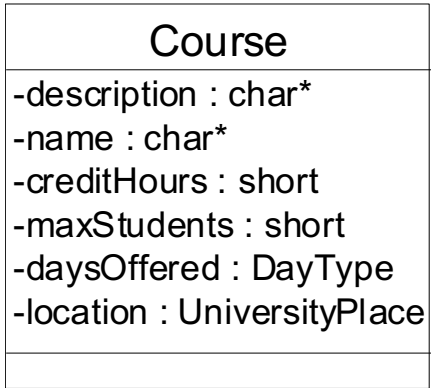
用户定义的数据类型

Course
-description : char* -name : char* -creditHours : short -maxStudents : short -daysOffered : dayType

```
enum dayType { MW, MWF, TT };
class Course {
   public:
   …
   private:
   char *description;
   char *name;
   short creditHours;
   short maxStudents;
   dayType daysOffered;
};
```

用户定义的类

```
class UniversityPlace {
  public:
  …
  private:
  char *building;
  short room;
};
```




```
#include "UniversityPlace.h";
enum dayType { MW, MWF, TT };
class Course {
   public:
   …
   private:
   char *description;
   char *name;
   short creditHours;
   short maxStudents;
   dayType daysOffered;
   UniversityPlace location;
};
```

Course类的属性和操作

Course
-description : char* -name : char* -daysOffered : DayType -creditHours : short -location : UniversityPlace -maxStudents : short
+isFull ():bool +addStudent (newStudent : Student*):void +getDescription ():const char* +save ():void +getName ():const char* +getDaysOffered ():dayType +getCreditHours ():short +getLocation ():const UniversityPlace +Course (name : char&,location : UniversityPlace&,desc : char&,days : DayType,hours : short,maxStudents : short) +~Course () +Course (: const Course&)

总结

- 学习内容
  - 面向对象设计
  - UML设计建模
- 实践要求
  - 会设计各种UML设计模型
    - 类图、包图、顺序图、状态机图、活动图、组件图、配置图
  - 能使用基于UML的面向对象设计完成所选课程项目的设计，并提交软件设计说明书