Shanghai Jiao Tong University

软件工程
Software Engineering

上海交通大学软件工程中心

讲师一

沈备军 博士

- 上海交通大学软件工程中心副教授
- 中国首位IEEE授证的软件开发专家 IEEE CSDP
- 国际软件工程知识体系SWEBOK 第3版 联合主编
- 国际软件测试认证委员会(ISTQB) 高级专家组成员
- 国家授证的系统分析员
- 全国软件工程专委委员
- 上海软件构件化专家指导委员会委员
- 上海市软件质量资格认证专家委员会委员
- 上海市技术主管资格认证专家委员会委员
- 主页：http://cse.sjtu.edu.cn/~bjshen
- Email: bjshen@sjtu.edu.cn
- 软件大楼1303室

讲师二

陈昊鹏 博士

- 上海交通大学软件工程中心副教授
- 主页：http://reins.se.sjtu.edu.cn/~chenhp
- Email：Chen-hp@sjtu.edu.cn
- 软件大楼1111室

本课程重点

软件是如何开发出来的?

课程大纲

播下一种思想

- 播下一种思想，收获一种行为；
- 播下一种行为，收获一种习惯；
- 播下一种习惯，收获一种性格；
- 播下一种性格，收获一种命运。

——威廉.詹姆斯

教材

软件工程原理
高等教育出版社
ISBN: 978-7-04-036906-9

MOOC课程

- 本科课程
  - MIT “Software Engineering for Web Applications”，Prof. Harold Abelson，Fall 2003, http://ocw.mit.edu
  - Vanderbilt, “Pattern-Oriented Software Architectures for Concurrent and Networked Software”, Prof. Douglas C. Schmidt, March 2013, https://www.coursera.org/#course/posa
  - 孟宁，中国科大，“软件工程”，2014，http://www.icourse163.org/course/ustc-89008#/info
  - 蒋严冰，北大，“面向对象技术高级课程”，2014，http://www.topu.com/mooc/3823
- 硕士课程
  - MIT “Software Engineering Concepts”，Prof. Nancy Leveson，Fall 2005，http://ocw.mit.edu

考核方法

- 平时成绩 40%
  - 作业　80分
  - 立项答辩　20分
  - 缺勤一次扣 4 分
  - 迟到或早退一次扣 1分
- 期末考试 60%
  - 开卷、笔试 100分

助教

1班张程	13917115283	459309661@163.com
2班周珺	13817193543	328353736@qq.com
3班童燊嗣	13817128140	254091182@qq.com
4班蔡栩阳	18667047886	bakercxy@163.com

Shanghai Jiao Tong University

软件工程
Module: 软件工程引论

上海交通大学软件工程中心

本节内容

- 什么是软件
  - 软件的作用、发展、定义和特性
  - 软件危机和问题
- 什么是工程
- 什么是软件工程
  - 软件工程的概念和知识域
  - 软件工程的金三角
  - 控制软件开发的复杂性
  - 软件建模及方法

@第1章和第3章.教材

计算机软件已经成为一种驱动力

- 进行商业活动的引擎；
- 现代科学研究和工程问题解决的基础；
- 区分现代产品和服务的关键因素；
- 现代社会中不可缺少的。

应用于：
交通、医药、通讯、军事、娱乐、
办公、社交、学习……
举例：Dell, Amazon, 淘宝, Ctrip等成功案例

“智能地球”

软件的定义

软件是

- 能够完成预定功能和性能的可执行的指令（计算机程序）；
- 使得程序能够适当地操作信息的数据结构；
- 描述程序的操作和使用的文档。

现在，被普遍接受的软件的定义是：

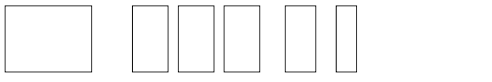
- 软件(software)是计算机系统中与硬件(hardware)相互依存的另一部分，它包括程序(program)、相关数据(data)及其说明文档(document)。

软件特征

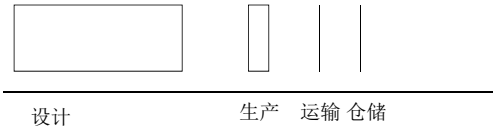
- 软件开发不同于硬件设计
  - 与硬件设计相比，软件更依赖于开发人员的业务素质、智力，以及人员的组织、合作和管理。同时对硬件而言，设计成本往往只占整个产品成本的一小部分，而软件开发占整个产品成本的大部分。
- 软件生产不同于硬件制造
  - 硬件设计完成后就投入批量制造，制造也是一个复杂的过程，其间仍可能引入质量问题；而软件成为产品之后，其制造只是简单的拷贝而已。软件的仓储和运输也非常简单。
- 软件维护不同于硬件维修
  - 硬件在运行的初期有较高的故障率，在缺陷修正后的一段时间中，故障率会降到一个较低和稳定的水平上。随着时间的改变，故障率将再次升高，这是因为硬件会受到摩损等损害，达到一定程度后就只能报废。软件是逻辑的而不是物理的，虽然不会磨损和老化，但在使用过程中的维护却比硬件复杂得多。如果软件内部的逻辑关系比较复杂，在维护过程中还可能产生新的错误。

软件是被设计的

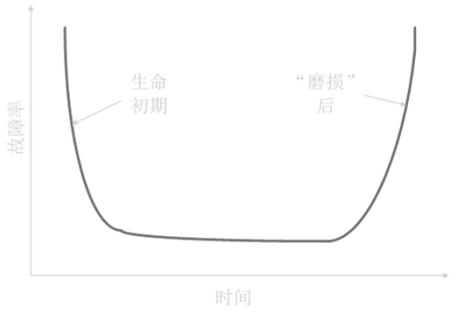
制造业：



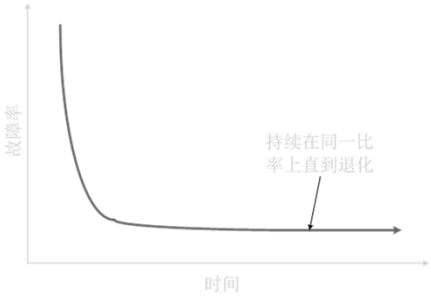
软件业：



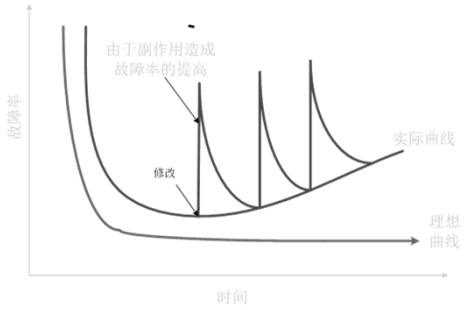
硬件的故障率曲线（浴缸曲线）



软件的故障率曲线(理想情况下)



软件的故障率曲线(实际情况下)



软件面临的新挑战

- 软件复杂性的增加
- 软件规模的不断扩大
- 软件环境的变化
- 遗留系统（Legacy System）的集成和复用
- 软件开发的高质量和敏捷性要求
- 分散的开发团队的协同

- 宇宙飞船的软件系统源程序代码多达2000万行，如果按生产效率一个人一年编写1万行代码的话，那么需要2000人年的工作量。
- Windows2000 开发团队是微软历史上最大的团队，整个团队5354人，合影时动用了直升飞机。

- 网络应用
- 多种硬件－普适计算
- 多操作系统
- 多编程语言
- 层出不穷的新技术

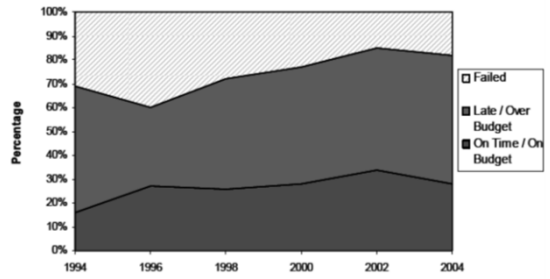
本节内容

- 什么是软件
  - 软件的作用、发展、定义和特性
  - 软件危机和问题
- 什么是工程
- 什么是软件工程
  - 软件工程的概念和知识域
  - 软件工程的金三角
  - 控制软件开发的复杂性
  - 软件建模及方法

软件危机的主要表现

- 对软件开发成本和进度的估计常常不准确。开发成本超出预算，实际进度比预定计划一再拖延的现象并不罕见。
- 用户对“已完成”系统不满意的现象经常发生。
- 软件产品的质量往往靠不住。Bug一大堆，Patch一个接一个。
- 软件的可维护程度非常之低。
- 软件通常没有适当的文档资料。
- 软件的成本不断提高。
- 软件开发生产率的提高赶不上硬件的发展和人们需求的增长。

软件项目成功率调查



Source: Standish Group Chaos Report, 1994, 1996, 1998, 2000, 2002, 2004

当前软件实践的问题

- 软件直到测试前仅仅是忽略质量的现代技术。典型地说，软件工程师
  - 没有计划他们的工作
  - 匆匆地走过需求和设计
  - 在编码时再进行设计
- 这些实践引入了大量的缺陷
  - 有经验的工程师每7-10行代码就引入一个缺陷
  - 平均中等规模的系统存在着上千个缺陷
  - 这些缺陷的大多必须靠测试发现
  - 通常要花去一倍以上的开发时间
- 目前大多数的工作方式还象30年前一样——“code-fix”

软件事故

- 1981年，由计算机程序改变而导致1/67的时间偏差，使航天飞机上的5台计算机不能同步运行。这个错误导致了航天飞机发射失败。
- 1986年，1台Therac25机器泄露致命剂量的辐射，致使两名医院病人死亡。原因是一个软件出现了问题，导致这台机器忽略了数据校验。
- 化学银行一个晚上从10多万位顾客账户上，错误地扣除了大约1500万美元的存款。原因是一个软件的问题使ATM的每一笔业务重复记录两次。
- 小事故：计算机一个月停机几次，不得不随时进行数据备份；定期重装机器；病毒四溢。

软件工程师是否需要发执照?

中国的软件产业现状

- 软件作坊 vs. 软件工厂
- 缺少系统化和规范化的管理
- 规模小，大多数在50人以下
- 生产率低
- 质量差

和美国、印度等存在较大的差距。

软件危机和问题的原因

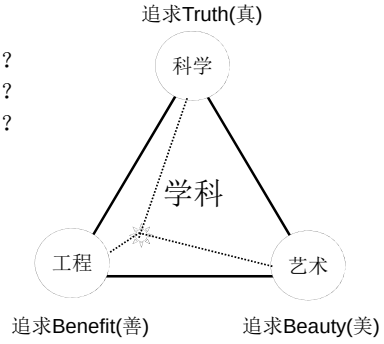
- 一方面是与软件本身的特点有关
- 另一方面是由软件开发和维护的方法、过程、管理、规范和工具不正确有关

本节内容

- 什么是软件
  - 软件的作用、发展、定义和特性
  - 软件危机和问题
- 什么是工程
- 什么是软件工程
  - 软件工程的概念和知识域
  - 软件工程的金三角
  - 控制软件开发的复杂性
  - 软件建模及方法

软件是一门什么学科？

- 软件开发
  - 是一门艺术？
  - 是一门科学？
  - 是一门工程？



什么是工程？

- 工程是对技术（或社会）实体的分析、设计、建造、验证和管理。
- 工程是一种组织良好、管理严密、各类人员协同配合、共同完成工作的学科。
- 它具有以下特性：
  - 以价值为目标
  - 高度的组织管理性
  - 多种学科的综合
  - 高度的实践性

如何做工程？

- 一个工程实践包括以下四个核心步骤：
  - 理解问题。在软件工程领域，就是需求分析；
  - 规划方案。在软件工程领域，就是设计；
  - 实施计划。在软件工程领域，就是编码；
  - 验证结果的准确性。在软件工程领域，就是测试和质量保证。

本节内容

- 什么是软件
  - 软件的作用、发展、定义和特性
  - 软件危机和问题
- 什么是工程
- 什么是软件工程
  - 软件工程的概念和知识域
  - 软件工程的金三角
  - 控制软件开发的复杂性
  - 软件建模及方法

软件工程的经典定义

- “The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works on real machines.” [Fritz Bauer]
- 软件工程就是为了经济地获得可靠的且能在实际机器上高效运行的软件而建立和使用的工程原理。

软件工程的经典定义(Cont.)

- “The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software” [IEEE 1990]
- 软件工程是将系统的、规范的、可量化的方法应用于软件的开发、运行和维护，即将工程化应用于软件。

软件工程的经典定义(Cont.)

- “Software engineering is that form of engineering that applies the principles of computer science and mathematics to achieving cost-effective solutions to software problems.” [CMU/SEI-90-TR-003]
- 软件工程就是应用计算机科学和数学的原理来经济有效地解决软件问题的一种工程。

软件工程是一个国家的战略性学科

- 软件工程的发展受到了美国DOD的极大推动
  - CMU SEI
- 如果一个国家的软件工程不强，那么这个国家就不会强大
- 软件工程要做强，关键在于创新，在于规范化的软件开发
- 本课程是软件学院的最核心课程，区别于计算机科学系。

软件工程知识体系SWEBOK

- 软件工程知识体系(Software Engineering Body of Knowledge) 简称SWEBOK
- 由IEEE国际组织推出
- 版本：当前版本 V3(2014)
- 网址：http://www.swebok.org
- 国际软件工程师证书：
  - CSDA，针对大学应届生
  - CSDP，针对有经验的工程师

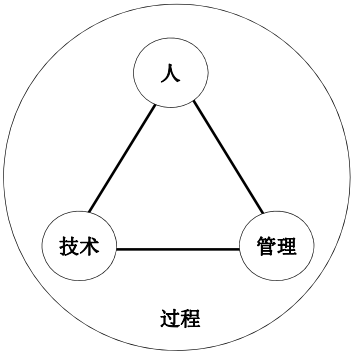
SWEBOK V3的15个知识域

- 软件工程实践类知识域
  - 软件需求、软件设计、软件构造、软件测试、软件维护、软件配置管理、软件工程管理、软件过程管理、软件工程方法、软件质量、软件工程职业实践
- 软件工程教育要求类知识域
  - 工程经济基础、计算基础、数学基础、工程基础

本节内容

- 什么是软件
  - 软件的作用、发展、定义和特性
  - 软件危机和问题
- 什么是工程
- 什么是软件工程
  - 软件工程的概念和知识域
  - 软件工程的金三角
  - 控制软件开发的复杂性
  - 软件建模及方法

软件工程的金三角



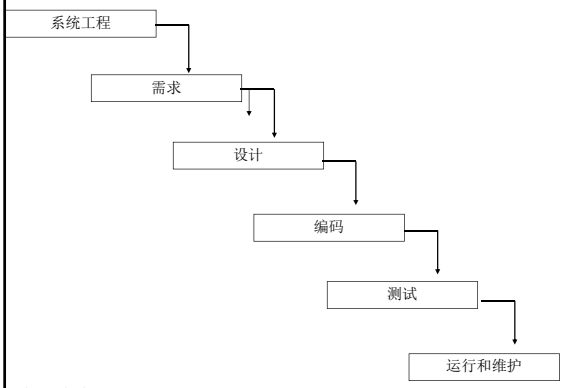
人
完成软件开发的主体

技术
提供了建造软件在技术上需要 “如何做”的方法

管理
提供了质量管理、成本管理、时间管理、范围管理等知识和技能

过程
这是将人、技术、管理结合在一起的凝聚力

软件工程技术



系统工程 (System Engineering)

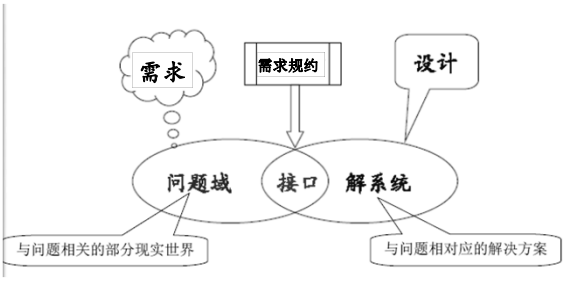
- 在软件开发之前，必须了解该软件所外的外部“系统”。
- 计算机系统包括计算机硬件、软件、人员、数据库、文档、规程等系统元素。
- 系统工程的任务：
  - 系统建模 ——系统模型
  - 系统仿真
- 系统工程的表现形式
  - 信息系统，关注企业，业务过程工程——业务模型
  - 嵌入式系统，关注产品，产品工程——产品模型
  - 多媒体系统，关注内容，内容工程——剧本

需求 (Requirement)

- 目的：澄清用户的需求。
- 描述：软件人员和用户沟通，充分理解和获取用户的需求，并进行分析，形成《软件需求规约》文档和分析模型。
- 任务：
  - 需求获取
  - 需求分析和建模
  - 需求定义
  - 需求确认
  - 需求管理

重点在What

需求与设计



设计（Design）

- 目的：建立软件的设计蓝图，是需求到代码的桥梁。
- 任务：软件人员依据软件需求，进行设计，形成《软件架构》文档和设计模型。
- 内容
  - 架构设计
  - 详细设计

重点在How

编程 (Coding)

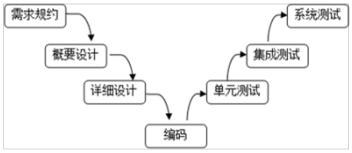
- 任务:依据设计说明书为每个模块编写程序
- 交付:程序
  - 程序内部含有必要的注释
  - 不含有语法错误
- 交付这样的程序就是编程阶段完成的里程碑

编程的误区

- 软件人员应该克制急于去编程的欲望
  - 先经过分析阶段确定用户要求，
  - 再经过设计阶段为编程制订蓝图，
  - 至此编程的条件才具备，可进入编程阶段
- 不要让compiler代你找代码的错误
  - 那样会让语义bug隐藏得更深
  - 先要自查代码的错误，再编译

测试 (Testing)

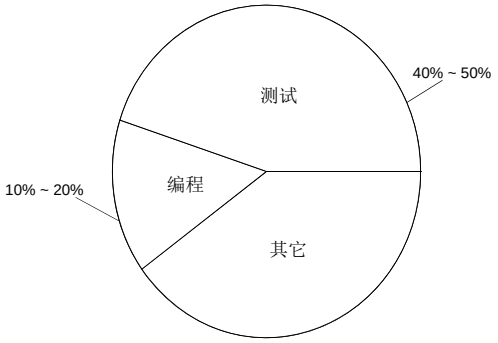
- 任务：发现并排除软件中的缺陷
- 不同层次的测试：
  - 单元测试 (Unit testing)
  - 集成测试 (Integration testing)
  - 系统测试 (System testing)



部署 (Deployment)

- 任务：交付、支持和反馈
- 部署原则：
  - Manage customer expectations for each increment
  - A complete delivery package should be assembled and tested
  - A support regime should be established
  - Instructional materials must be provided to end-users
  - Buggy software should be fixed first, delivered later

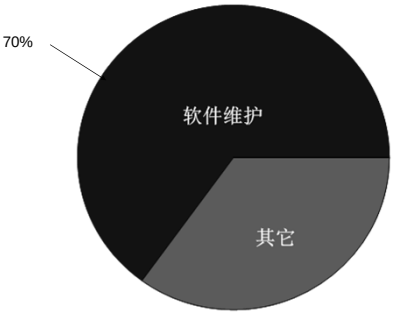
软件开发工作量分配比例



运行和维护 (Operation and Maintenance)

- 所谓维护，是指软件系统交付使用以后，为了改正错误或满足新的需要而修改软件的过程。
- 举例：一个中等规模的软件，如果开发过程要一年时间，它投入使用后，其运行时间可能持续五年，这段时间是维护阶段。
- 与开发相比，维护阶段的工作量和成本要大得多。
- 人们对软件维护的认识远不如软件开发，因为开发具有主动性和创造性，易被人们所重视。

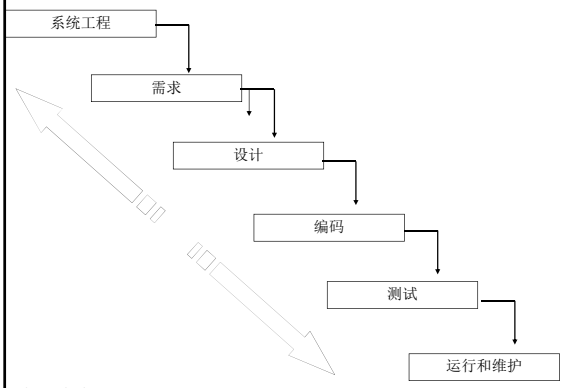
费用分配比例



维护类型

- 纠错性维护（Corrective maintenance）
  - 由于软件中的缺陷引起的修改
- 完善性维护（Perfective maintenance），
  - 根据用户在使用过程中提出的一些建设性意见而进行的维护活动
- 适应性维护（Adaptive maintenance）
  - 为适应环境的变化而修改软件的活动
- 预防性维护（Preventive maintenance）
  - 为了进一步改善软件系统的可维护性和可靠性，并为以后的改进奠定基础

软件工程的发展



本节内容

- 什么是软件
  - 软件的作用、发展、定义和特性
  - 软件危机和问题
- 什么是工程
- 什么是软件工程
  - 软件工程的概念和知识域
  - 软件工程的金三角
  - 控制软件开发的复杂性
  - 软件建模及方法

控制软件开发的复杂性

- 软件开发常常是相当复杂、不可预测、难以计划的。
- 软件开发的复杂性主要来源于：
  - 技术的复杂性
    - 不断发展
    - 使用多项技术
  - 需求的复杂性
    - 模糊
    - 不断变化
  - 人的复杂性

?

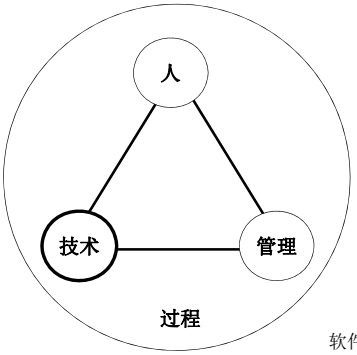
如何控制复杂性？

- 抽象
- 分解
- 迭代

本节内容

- 什么是软件
  - 软件的作用、发展、定义和特性
  - 软件危机和问题
- 什么是工程
- 什么是软件工程
  - 软件工程的概念和知识域
  - 软件工程的金三角
  - 控制软件开发的复杂性
  - 软件建模及方法

软件建模是软件工程的核心技术



人
完成软件开发的主体

技术
提供了建造软件在技术上需要 “如何做”的方法

管理
提供了质量管理、成本管理、时间管理、范围管理等知识和技能

过程
这是将人、技术、管理结合在一起的凝聚力

软件建模是软件工程的核心技术

思考

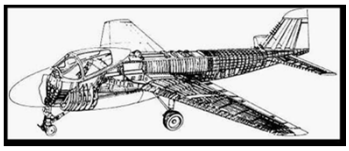
- 什么是模型？
- 为什么要建模？
- 软件模型有哪几种？

波音747

- 超过6百万个零件，仅机尾就有上百个零件
- 能承载上百个旅客
- 能不停地飞行上千公里

波音747

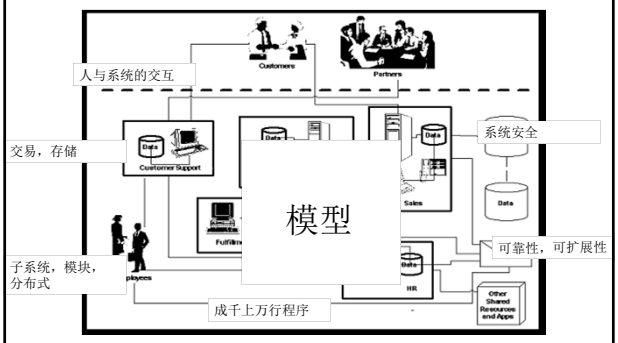
- 当波音在60年代开始生产747的时候，一共画了75，000张工程图



为什么？

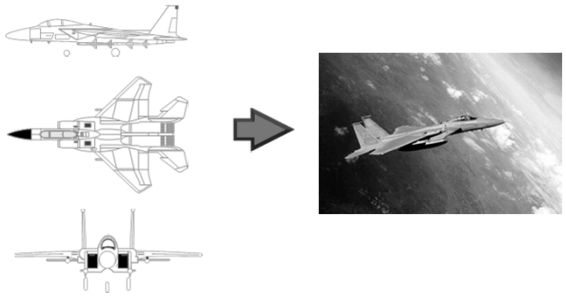
因为他们正在生产一个极其复杂的系统，
而且不能出哪怕是一点点的差错

软件开发也同样复杂



什么是模型?

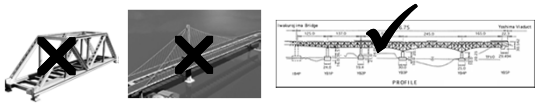
- A model is a simplification of reality.



工程师们这样做…

- 他们在建造实际的物体之前...

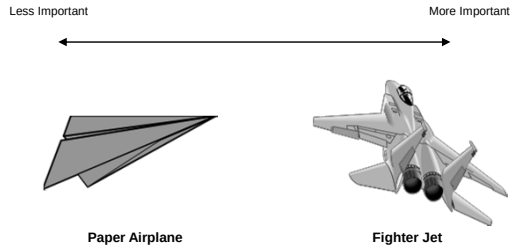
…首先建立模型　　…然后在模型的基础上
进行研究和改进



模型的功能

- 在正式启动工程项目之前发现设计中的错误和遗漏之处
  - 通过(形式化的)分析和实验，降低工程的风险
- 研究和比较不同的解决方案
- 用来和项目的所有者进行交流
  - 客户、用户、实现者、测试者、开发文档管理员，等等.
- 促进工程的实现

建模的重要性

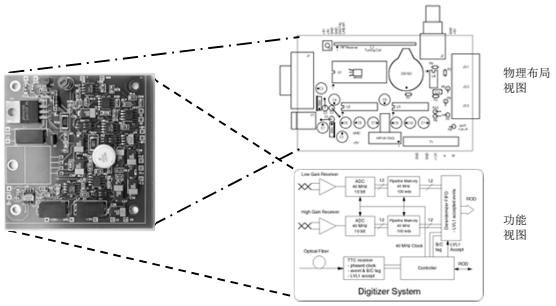


有用模型的特征

- 抽象性
  - 突出重点方面，去除无关紧要的细节
- 可理解性
  - 模型的表达方式能被模型的观察者很容易地理解
- 精确性
  - 忠实地表达被建模的系统
- 说明性
  - 能够被用来对被建模系统进行直观地分析，并得出正确的结论
- 经济性
  - 模型的建立和分析比被建模系统更廉价，更经济

作为有用的工程模型, 必需具备以上所有特征!

模型的多个视图

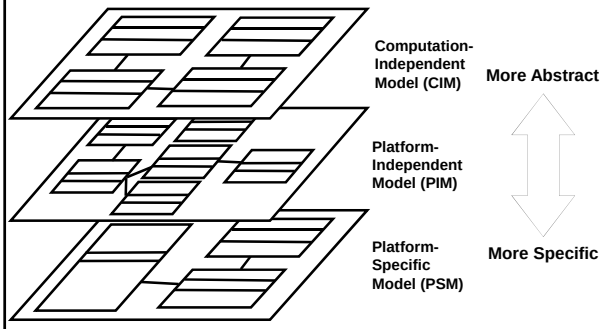


举例：UML 的软件模型视图

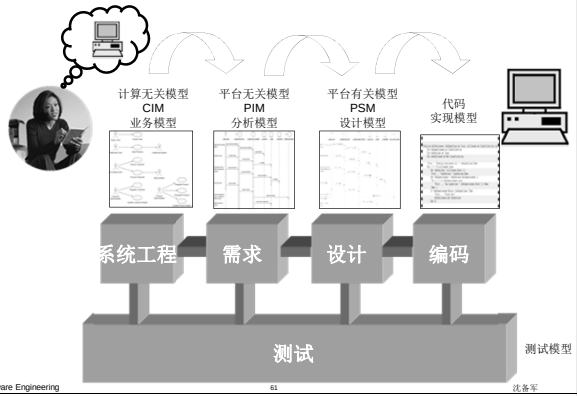
- 需求:
  - 用例图 Use Case diagram
- 结构:
  - 本体论: 类图 Class diagram
  - 实例: 对象图 Object diagram
- 行为
  - 状态图 Statechart diagram
  - 活动图 Activity diagram
  - 交互：顺序图和协作图 Sequence diagram & Collaboration diagram
- 实现:
  - 构件图 Component diagram
  - 部署图 Deployment diagram

这些视图相互集成构成一个完整的模型

软件的三种模型



模型间的转换



软件建模的方法

- 形式化方法 (Formal Method)
- 结构化方法 (Structured Method)
- 面向对象方法 (Object Oriented Method)
- 基于构件的软件开发方法 (Component Based Software Development)
- 面向服务方法 (Service Oriented Method)
- 模型驱动的开发方法 (Model-Driven Development)
- 敏捷建模方法 (Agile Modeling Method)
- ……

形式化方法

- 形式化方法是基于数学的技术开发软件，如集合论、模糊逻辑、函数。
- 形式化方法的好处：
  - 无二义性
  - 一致性
  - 正确性
  - 完整性

举例

--------AddBlock----------

△BlockHandler

Ablocks? : P BLOCKS

Ablocks? ⊆ used

BlockQueue'=BlockQueue⌢〈Ablocks?〉

used'=used∧

free'=free

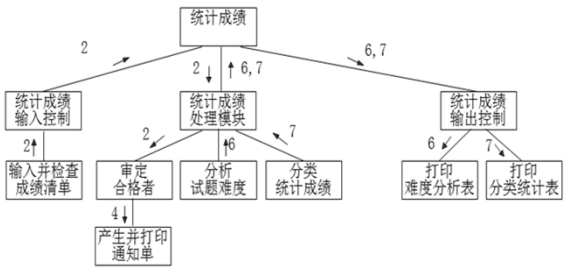
加一个块集合到队列的尾部,采用Z语言

形式化方法的不足

- 形式化规约主要关注于功能和数据，而问题的时序、控制和行为等方面却更难于表示。此外，有些问题元素(如，人机界面)最好用图形技术来刻划。
- 使用形式化方法来建立规约比其他方法更难于学习，并且对某些软件实践者来说它代表了一种重要的“文化冲击”。
- 难以支持大的复杂系统。

尚未成为主流的开发方法，实践和应用较少

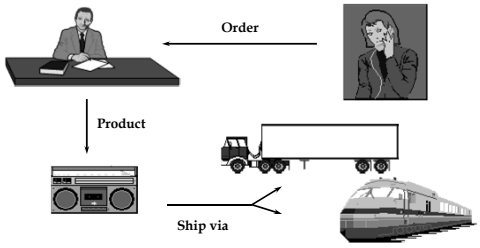
结构化设计的系统示例



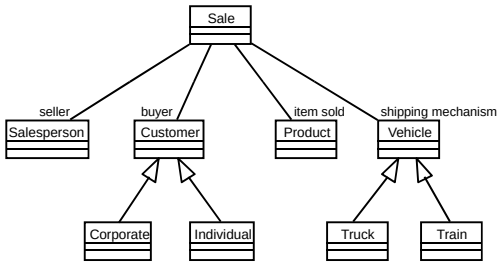
结构化方法

- 核心: 数据和处理
- 手段:自顶向下，逐步求精、模块化
- 常用建模工具:
  - 需求建模：
    - DFD(数据流图) 、 DD(数据字典)、ERD(实体关系图)、STD(状态图)
  - 设计建模：
    - 结构图（SC）
    - 流程图、 N-S图、PAD图、伪代码

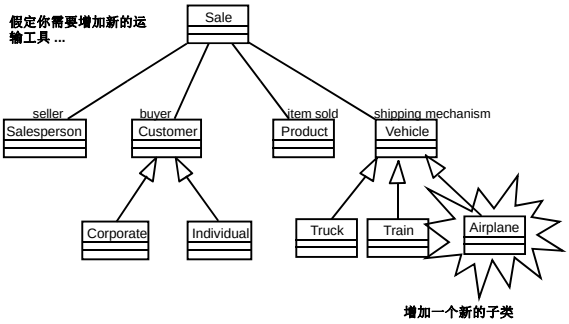
面向对象方法示例：销售订单



销售订单的类图



需求变更的影响



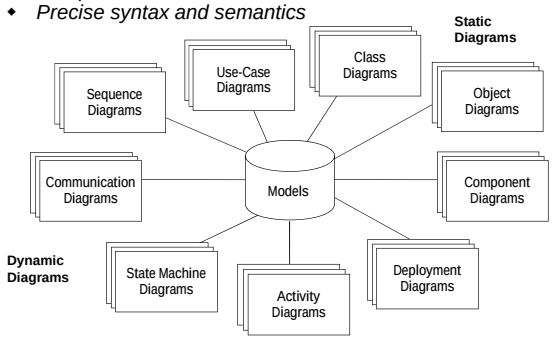
面向对象方法

- 九十年代以来的主流开发方法
  - 符合人们对客观世界的认识规律
  - 开发的系统结构易于理解、易于维护
  - 继承机制有力支持软件复用
- 常见的面向对象方法
  - Booch method 1994
  - Coad and Yourdon method 1991
  - Rambaugh method -- OMT 1991
  - Jacobson method – OOSE 1992
  - Wirfs-Brock method 1990
  - ……

⇨ 国际标准统一建模语言 UML 1997

UML模型

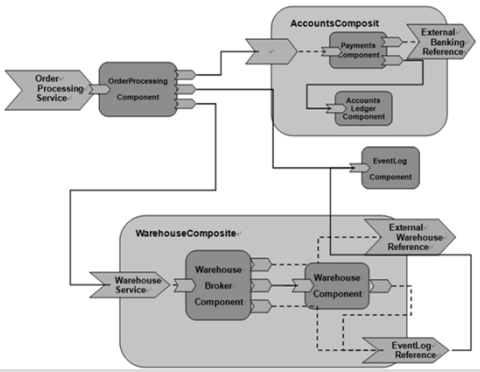
- *Multiple views*
- *Precise syntax and semantics*



UML建模工具

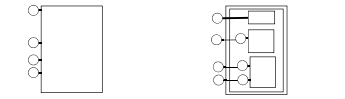
- IBM RSA 和Rational Rose
- Sybase PowerDesigner
- Sparx System EA
- Borland Together
- ArgoUML 开源
- StarUML 开源
- …

基于构件的软件系统示例

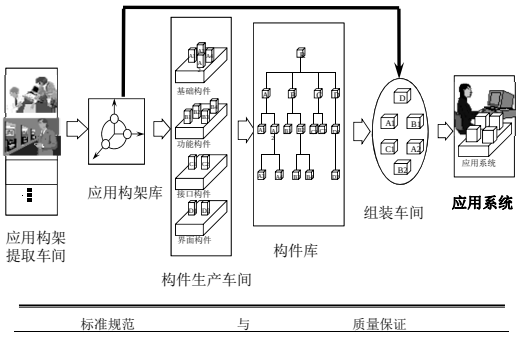


构件

- 构件是软件复用的重要手段，是核心和基础
- 构件由构件规约与构件实现两部分组成



基于构件的开发



Shanghai Jiao Tong University

软件工程
Module: 软件过程

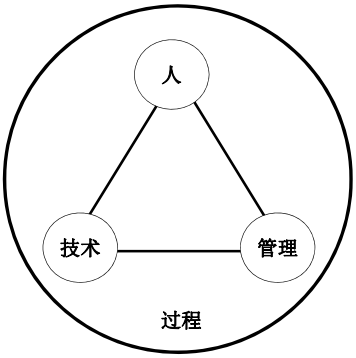
上海交通大学软件工程中心

软件过程

- 软件过程概述
- 软件生命周期模型
- 统一软件过程 RUP
- 敏捷过程

@第2章.教材

Review: 软件工程的金三角



人
完成软件开发的主体

技术
提供了建造软件在技术上需要 “如何做”的方法

管理
提供了质量管理、成本管理、时间管理、范围管理等知识和技能

过程
这是将人、技术、管理结合在一起的凝聚力

软件过程的杠杆作用点

- 每个人都体会到主动积极的优质劳动力的重要性，但是...
- …如果不理解过程，或者过程不是在“最佳实践”下运行，即使我们的技术精英也无法使工作达到最佳的状态
- 过程是产品成本、进度和质量的主要决定因素

什么是软件过程

- Defines Who is doing What, When to do it, and How to reach a certain goal.

软件过程的组成

- 软件过程，也称为软件生存周期过程，是指软件生存周期中的一系列相关过程，其中过程就是活动的集合，活动是任务的集合，任务要起到把输入加工成输出的作用。
- 活动的执行可以是顺序的、迭代的（重复的）、并行的、嵌套的，或者是有条件地引发的。

软件过程

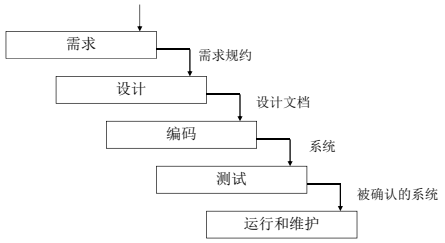
- 软件过程概述
- 软件生命周期模型
- 统一软件过程 RUP
- 敏捷过程

什么是软件生存周期模型

- 又称软件开发模型，是软件生命周期的一个框架，规定了软件开发、运作和维护等所需的过程、活动和任务。
- 软件生存周期模型分类：
  - 线性顺序模型 Waterfall Model
  - 增量式模型 Incremental Model
  - 演化模型 Evolutionary Model

瀑布型(Waterfall)

- 最早的软件开发模型
- 1970年W. Royce提出
- 又称为线性顺序模型



瀑布型特点

- 特点
  - 强调阶段的划分及其顺序性
  - 强调各阶段工作及其文档的完备性
  - 每个阶段结束之前，都从技术和管理两个角度进行严格的审查
  - 是一种严格线性的、按阶段顺序的、逐步细化的开发模式
- 适用时机
  - 所有功能、性能等要求能一次理解和描述时
  - 所有的系统功能一次交付时
  - 必须同时淘汰全部老系统时

瀑布型的价值

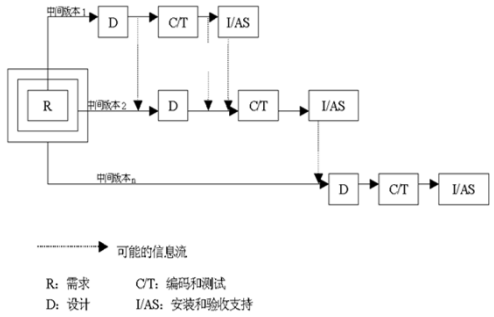
- 结构简单明了；历史较长、应用面广泛、为广大软件工作者所熟悉；已有与之配套的一组十分成熟的开发方法和丰富的支撑工具。
- 一种较为有效的管理模式：订计划、成本预算、组织开发人员,阶段评审,文档管理,从而对软件质量有一定的保证。

瀑布型的风险和缺点

- 获得完善的需求规约是非常困难的；
- 难以适应快速变化需求；
- 系统太大时,难以一次做完；
- 反馈信息慢；
- 极可能引起开发后期的大量返工，如返工到需求、设计等早期活动；
- …

增量型（Incremental)

- 构造一系列可执行的中间版本(Version by Version)



增量型需考虑的风险

- 需求未被很好地理解
- 一次要求所有功能
- 需求迅速发生变化
- 事先打算采用的技术迅速发生变化
- 长时期内仅有有限的资源（人员/资金）

增量型适用时机

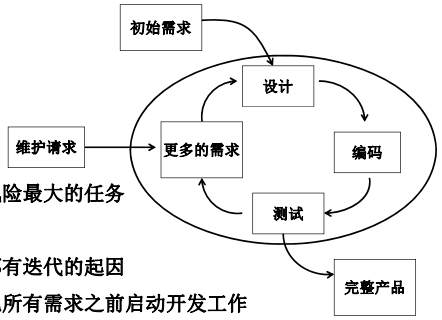
- 需要早期获得功能；
- 中间产品可以提供使用；
- 系统被自然地分割成增量；
- 工作人员/资金可以逐步增加。

演化型(Evolutionary)

- 现状：
  - 软件需求在软件开发过程中常常发生改变，想要一次迭代就开发出最终产品是不可能的
  - 紧迫的市场期限使得难以一下子完成一个完善的软件产品
- 解决方案：演化模型
  - 只要核心需求能够被很好地理解，就可以进行渐进式开发，其余需求可以在后续的迭代中进一步定义和实现。这种过程模型称为演化模型，它能很好地适应随时间演化的产品的开发。
- 特点：
  - 迭代的开发方法，渐进地开发各个可执行版本，逐步完善软件产品。每个版本在开发时，开发过程中的活动和任务顺序地或部分重叠平行地被采用。
  - 与增量模型的区别是：需求在开发早期不能被完全了解和确定，在一部分被定义后开发就开始了，然后在每个相继的版本中逐步完善。

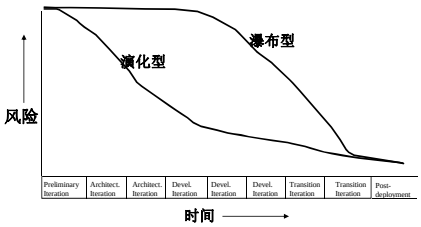
演化模型举例

基于风险的、顺序执行的演化模型



- **首先执行风险最大的任务**
- **并行开发**
- **每次迭代都有迭代的起因**
- **可以在细化所有需求之前启动开发工作**

演化型价值：降低风险

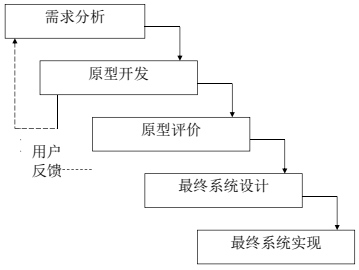


演化模型是目前采用最广泛的模型

演化模型已成为主流

- 现代软件过程都采用演化模型
  - 统一软件过程RUP
  - 敏捷过程 （SCRUM、XP等）
  - 净室（Cleanroom）软件过程
- 演化模型的“子类”
  - 原型 Prototyping
  - 螺旋模型 Spiral Model

快速原型模型 Rapid Prototyping Model

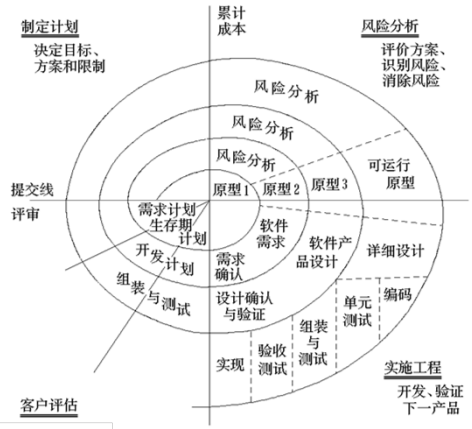


一种迭代次数为2的演化模型

快速原型模型的特点

- 特点
  - 快速开发工具
  - 循环
  - 低成本
- 种类
  - 渐进型
  - 抛弃型

螺旋模型
Spiral Model



一种基于风险的演化模型

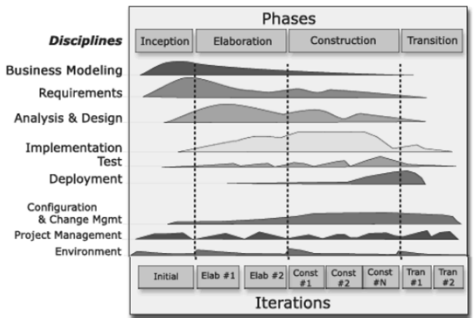
螺旋模型的特点

- B.Boehm于1988年提出
- 瀑布模型+快速原型+风险分析
- 螺旋模型沿着螺线旋转，在四个象限上分别表达四个方面的活动，即：
  - 制定计划：确定软件目标，选定实施方案，弄清项目开发的限制条件
  - 风险分析：评价所选的方案，识别风险，消除风险
  - 工程实施：实施软件开发，验证工作产品
  - 客户评估：评价开发工作，提出修正建议

软件过程

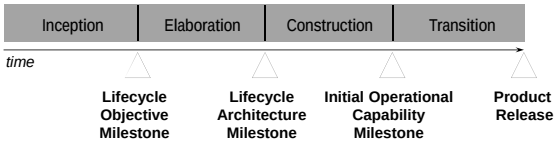
- 软件过程概述
- 软件生命周期模型
- 统一软件过程 RUP
- 敏捷过程

统一软件过程 RUP



RUP是一个风险驱动的、基于UML和构件式架构的演化开发过程 。

RUP的四个阶段

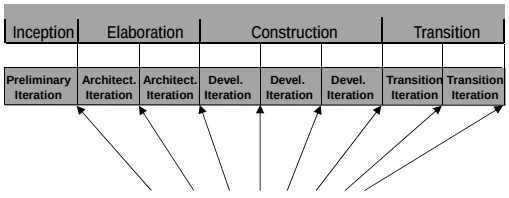


Lifecycle Objective Milestone | Lifecycle Architecture Milestone | Initial Operational Capability Milestone | Product Release

- Inception - **Define the scope of project**
- Elaboration - **Plan project, specify features, baseline architecture**
- Construction - **Build the product**
- Transition - **Transition the product into end user community**

每个阶段结束是一个大的里程碑

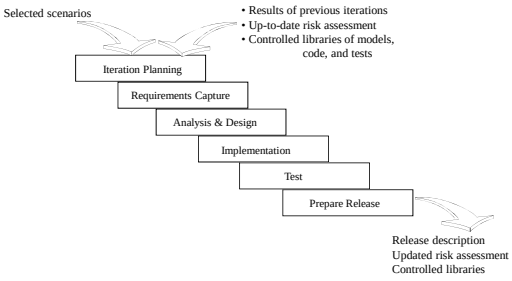
阶段和迭代



Minor Milestones: Releases

An iteration is a distinct sequence of activities with an established plan and evaluation criteria, resulting in an executable release (internal or external)

一个迭代周期：一个小的瀑布模型



软件过程

- 软件过程概述
- 软件生命周期模型
- 统一软件过程 RUP
- 敏捷过程

敏捷过程

- **敏捷过程很容易适应变化并迅速做出自我调整，在保证质量的前提下，实现企业效益的最大化。**
- **敏捷过程在保证软件开发有成功产出的前提下，尽量减少开发过程中的活动和制品，Just enough**
- **2001年2月，新方法的一些创始人在美国犹他州成立Agile 联盟（www.agilealliance.org）**

XP SCRUM Crystal ASD dx

MSF FDD DSDM Lean Development

敏捷宣言

- 较之于过程和工具，更注重人及其相互作用的价值
- 较之于无所不及的各类文档，更注重可运行的软件的价值
- 较之于合同谈判，更注重与客户合作的价值
- 较之于按计划行事，更注重响应需求变化的价值

敏捷过程的适用范围

Martin Fowler认为：新方法不是到处可适用的

适合采用敏捷过程的情况：

- 需求不确定、易挥发
- 有责任感和积极向上的开发人员
- 用户容易沟通并能参与
- 小于10个人的项目团队

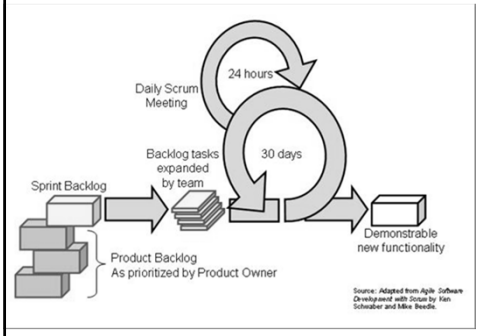
Scrum －敏捷的软件项目管理

- 1994年由Ken Schwaber和 Jeff Sutherland 提出
- Scrum一词来源于橄榄球运动，意为两队并列争球
- Scrum过程的核心：
  - 一个体育队加小队长,全体团队负责拿球向前冲
  - 团队成员能够独立地、集中地在创造性的环境下工作

Scrum的核心准则

- Iterative Development 迭代开发
  - The project deliverables are built over several iterative development cycles, each adding additional features, and each resulting in demonstratable results: working code, written documentation, viewable designs, etc.
- Team Empowerment 自我管理
  - The project team is divided into self-managing multi-function units called Sprint Teams consisting of up to six or seven people. The team is empowered to use whatever development methods or tools they think best to prepare their deliverables

Scrum过程框架



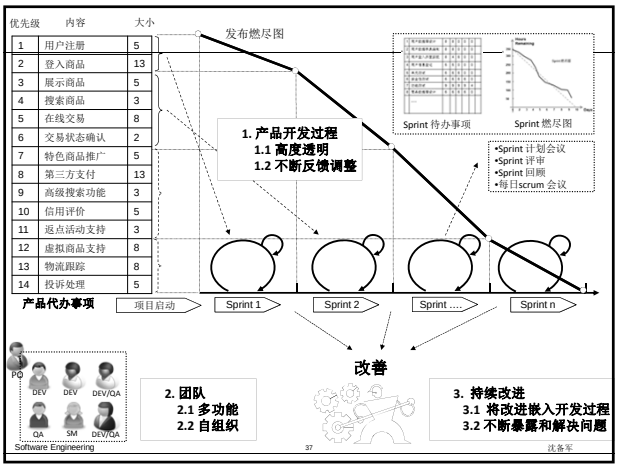
Sprint
周期为30天的迭代

Backlog
待办事项表（功能和非功能需求清单）

Daily Scrum
每日15分钟简会

Scrum Players

- Scrum 项目组
  - Product Owner
    - Adds items to product backlog list
    - Set priorities
  - Team
    - Determines sprint list (determine what can be done)
    - Develops software
  - ScrumMaster （相当于传统的项目经理角色）
    - Responsible for Scrum process
    - Removes impediments
- Other interested parties
  - Stakeholder
    - Funded project, will use it, affected by it
    - Requests enhancements
  - IT Management
    - Manpower allocation
    - Budgets & Billing
  - Senior Business Management
    - Best use of corporate resources


优先级 内容 大小
1 用户注册 5
2 登入商品 13
3 展示商品 5
4 搜索商品 3
5 在线交易 8
6 交易状态确认 2
7 特色商品推广 5
8 第三方支付 13
9 高级搜索功能 3
10 信用评价 5
11 返点活动支持 3
12 虚拟商品支持 8
13 物流跟踪 8
14 投诉处理 5
发布燃尽图
Sprint 待办事项
Sprint 燃尽图
1. 产品开发过程
1.1 高度透明
1.2 不断反馈调整
•Sprint 计划会议
•Sprint 评审
•Sprint 回顾
•每日scrum 会议
产品代办事项
项目启动
Sprint 1
Sprint 2
Sprint ....
Sprint n
改善
PO
DEV
DEV
DEV/QA
QA
SM
DEV/QA
2. 团队
2.1 多功能
2.2 自组织
3. 持续改进
3.1 将改进嵌入开发过程
3.2 不断暴露和解决问题
Software Engineering
37
注备军

Shanghai Jiao Tong University

软件工程
Module: 需求工程

上海交通大学软件工程中心

问题

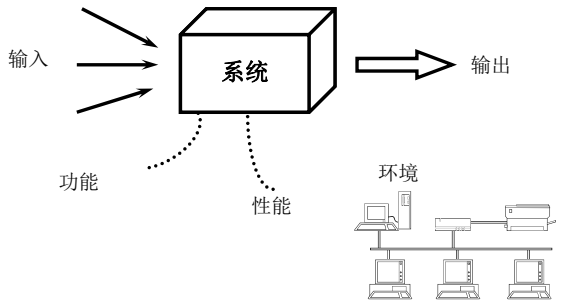
- 用户提不出需求
  - I'm not sure, but I'll know it when I see it
- 需求经常变化，项目没有时限
- 开发人员不得不大量超时工作，因为误解或二义性的需求直到开发后期才发现
- 系统测试白费了，因为测试者并未明白产品要做什么
- 功能都实现了，但由于产品的低性能、使用不方便或其它因素用户不满意
- 维护费用相当高，因为客户的许多增强要求未在需求获取阶段提出

需求工程

- 软件需求概念
- 需求获取
- 需求分析和建模
- 需求定义和验证
- 需求管理

@第5章.教材

什么是软件需求?



定义

- 需求
  - 系统必须符合的条件或能力
- 软件需求
  - 用户对目标软件系统在功能、行为、性能、设计约束等方面的期望。
  - 内容包括：FURPS +

关注What!

FURPS

Functionality	Feature Set Capabilities	Generality Security	功能需求
Usability	Human Factors Aesthetics	Consistency Documentation	非功能需求
Reliability	Frequency/Severity of Failure Recoverability	Predictability Accuracy MTBF	
Performance	Speed Efficiency Resource Usage	Throughput Response Time	
Supportability	Testability Extensibility Adaptability Maintainability Compatibility	Configurability Serviceability Installability Localizability Robustness	

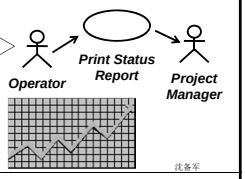
定义 Functionality 需求

详细功能需求

Trending information will be charted with a line graph showing time on the x axis, and number of defects found on the y axis.

概要功能需求

Feat 63 - the defect tracking system will provide trending information to help the project manager assess project status



定义 Reliability 需求

- “reliability”可靠性
  - The ability for the software to behave consistently in a user-acceptable manner
- 可靠性需求
  - Availability (xx.xx%)
  - Accuracy
  - Mean time between failures (xx hrs)
  - Max. bugs per/KLOC (0-x)
  - Bugs by class - critical, significant, minor

定义 Supportability 需求

- “supportability”支持性
  - The ability of the software to be easily modified to accommodate enhancements and repairs
- 支持性需求
  - Languages, DBMS, tools, etc.
  - Programming standards
  - Error handling and reporting standards
- 常常难以定义
  - If not measurable or observable, it is not a requirement
  - Is it a design constraint?
  - Is it an intent or goal?

定义 Performance 需求

- “performance”性能
  - A measure of speed or efficiency of the running system
- 性能需求
  - Capacity
  - Throughput
  - Response time
  - Memory
  - Degradation modes
  - Efficient use of scarce resources
    - Processor, memory, disk, network bandwidth

Davis Workshop, 1993

定义 Usability 需求

- “usability”可用性
  - The ease with which software can be learned and operated by the intended users
- 可用性需求
  - Training time requirements, measurable task times
  - User abilities (beginner/advanced)
  - Comparison to other systems that users know and like
  - Online help systems, tool tips, documentation needs
  - Conformity with standards
    - Examples: Windows, style guides, GUI Standards

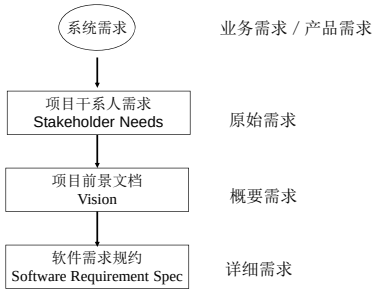
FURPS +

- 设计约束（design constraints）：规定或约束了系统的设计的需求；
- 实现需求（implementation requirements）：规定或约束了系统的编码或构建，如所需标准、编程语言、数据库完整性策略、资源限制和操作环境；
- 接口需求（interface requirements）：规定了系统必须与之交互操作的外部软件或硬件，以及对这种交互操作所使用的格式、时间或其他因素的约束；
- 物理需求（physical requirements.）：规定了系统必须具备的物理特征，可用来代表硬件要求，如物理网络配置需求。

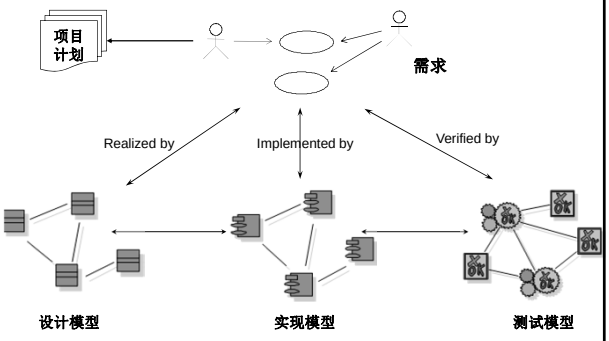
设计约束

- 一项需求允许多种设计方案
  - 设计是在这多种方案中做出选择
- 没有选择的需求就是一个设计约束
  - 它和其它需求不同
  - 将它放在软件需求的单独一节中
  - 将每个设计约束的源标识出来
  - 记录每个设计约束的原理
- 举例
  - 必须要有某一种算法
  - 必须要用数据库

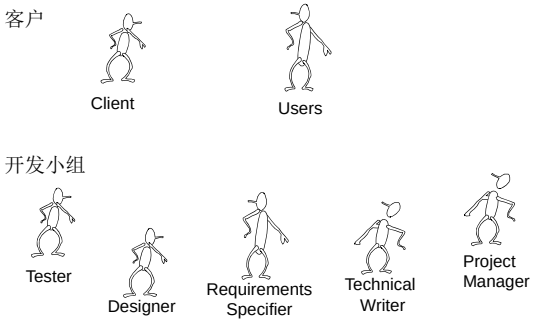
软件需求的三个层次



需求驱动开发



谁看需求?

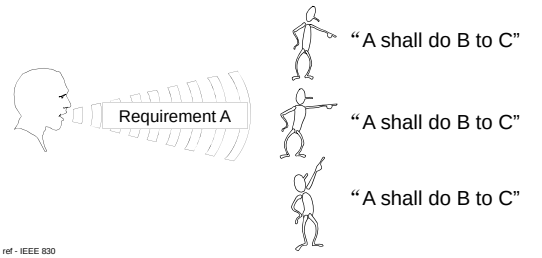


优秀需求具有的特性

- 单个优秀需求应具有的特性：
  - 完整性
  - 正确性
  - 可行性
  - 必要性
  - 划分优先级 （Why?）
  - 无二义性
  - 可验证性
- 多个优秀需求应具有的特性：
  - 完整性
  - 可理解性
  - 一致性
  - 可跟踪性

无二义性

- 需求是无二义的，如果：
  - 它只有一个解释



ref - IEEE 830

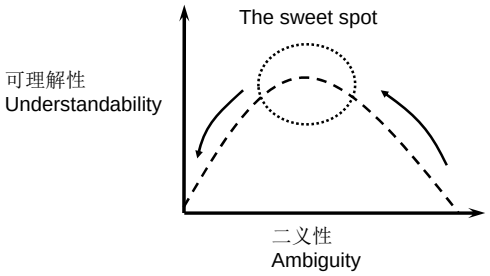
二义性练习

我没说她偷了我的钱

Mary had a little lamb

如何减少需求的二义性？

二义性与可理解性的关系



可验证性

- 需求可验证的，如果
  - 可以用人工或机器方式检查产品是否满足需求

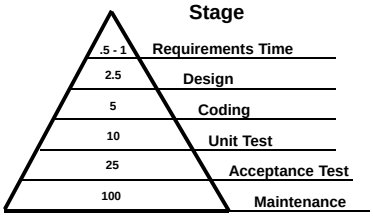
- The system supports up to 1,000 simultaneous users
- The system shall respond to an arbitrary query in 500 msec.
- The color shall be a pleasing shade of green
- The system shall be available 24 x 7
- The system shall export view data in comma-separated format

这些需求可验证吗？
如果不，如何更好地表达？

IEEE 830

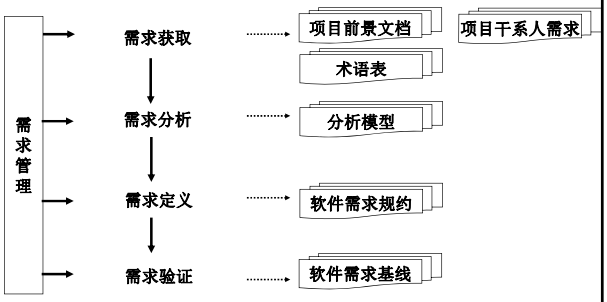
需求出错的高成本

The 1-10-100 Rule



"All together, the results show as much as a 200:1 cost ratio between finding errors in the requirements and maintenance stages of the software lifecycle."

需求工程

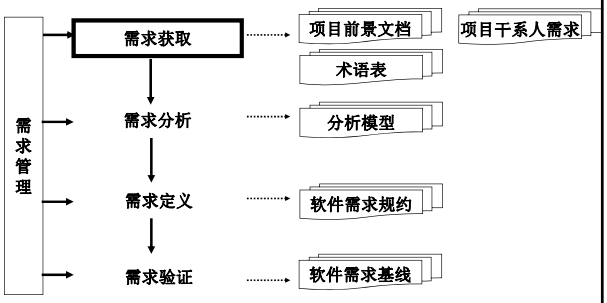


发现、获取、组织、 分析、编写和管理需求的系统方法，让客户和项目组之间达成共识。

需求工程

- 软件需求概念
- 需求获取
- 需求分析和建模
- 需求定义和验证
- 需求管理

需求工程



前景文档（Vision）

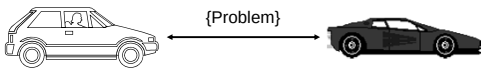
1. 简介 Introduction
2. 定位 Positioning
   2.1 商机 2.2 问题说明 2.3 产品定位
3. 项目干系人和用户描述 Stakeholder and User Descriptions
4. 产品概述 Product Overview
5. 产品特性 Product Features
6. 约束 Constraints
7. 质量范围 Quality Ranges
8. 优先级 Precedence and Priority
9. 其它产品需求 Other Product Requirements
10. 文档需求 Documentation Requirements

1）分析问题及根源

- 什么是问题?
  - 理解和记录客户的观点
  - 达成共识

从业务角度

- 什么是真正的问题?
  - 寻找问题根源，探究症结
- 避免Yes…But现象，避免IT黑洞



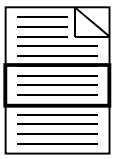
理解的 ***期望的***

识别业务解决方案

- 分析业务需求
- 提出多种解决方案
  - 技术的或(和)非技术的
- 选择最能满足业务需求的解决方案
- 启动项目,实现方案

案例研究: Course Registration System

- Review the problem statement



Course Registration
Requirements Document

举例：选课系统的问题陈述 @Vision文档

The problem of	The outdated and largely manual student registration process at Wylie College
affects	Students, professors, and College administration.
The impact of which is	A slow and costly process combined with dissatisfied students and professors.
A successful solution would	Improve the image of the College, attract more students, and streamline administrative registration functions.

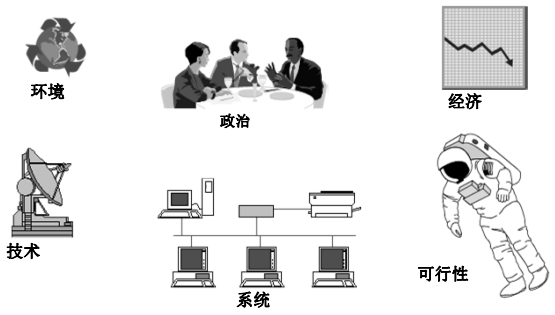
2）识别项目干系人

- 项目干系人（Stakeholder），又称为涉众、利益相关人，是积级参与项目，或其利益因项目的实施或完成而受到积极或消极影响的个人和组织，他们还会对项目的目标和结果施加影响。
  - 你的项目中有哪些项目干系人?

举例：选课系统的Stakeholder @Vision文档

Name	Represents	Role
IT Executive	IT Department and Wylie College as whole.	Responsible for project funding approval. Monitors project progress.
Registrar	The office of the registrar, administrative and data entry personnel.	Ensures that the system will meet the needs of the registrar, who has to manage the course registration data, including professor and student databases.
Student	Students	Ensures that the system will meet the needs of students.
Professor	Professors	Represents the interests of the faculty (professors).

3）识别项目的约束



举例：选课系统的约束条件 @Vision文档

- **Assumptions and Dependencies**
  - The existing Billing and Course Catalog Database Systems which reside on the College DEC VAX Mainframe will continue to be supported until at least 2005.
  - The external interfaces of the Billing and Course Catalog Database Systems are as defined in [2] and [3] and will not be altered.
  - It is assumed that the College will continue to operate and support the existing UNIX Server and the DEC VAX Mainframe until at least 2005.
  - It is assumed that additional funding will be available by 2005 to replace the legacy Billing and Course Catalog Database Systems.
  - Implementation of the new registration system in time for the January 2000 school term is dependent upon funding approval by March 1st, 1999.
- Constraints
  - The system shall not require any hardware development or procurement.
  - The course information available is limited to the type of data supported by the existing Course Catalog Database.

4）获取常用术语

- 定义项目用到的术语
- 有助于避免误解
- 记录于单独的术语表文件中

术语表
Glossary

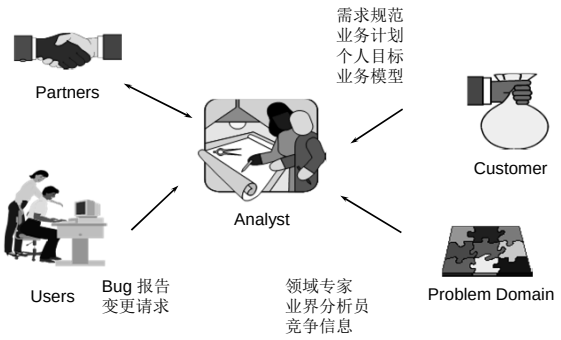
获取常用术语

- **尽早开始**
- **在项目过程中持续进行**

举例：选课系统的术语表 @Glossary文档

- **Course**
  - A class offered by the university.
- **Course Offering**
  - A specific offering for a course, including days of the week and times.
- **Course Catalog**
  - Unabridged catalog of all courses offered by the university.
- **Grade**
  - The grade for the student in a course.
- **Report Card**
  - All the grades for all courses taken by a student in a given semester.
- **Roster**
  - All the students enrolled in a particular course offering.
- **Transcript**
  - The history of the grades for all courses for a particular student.

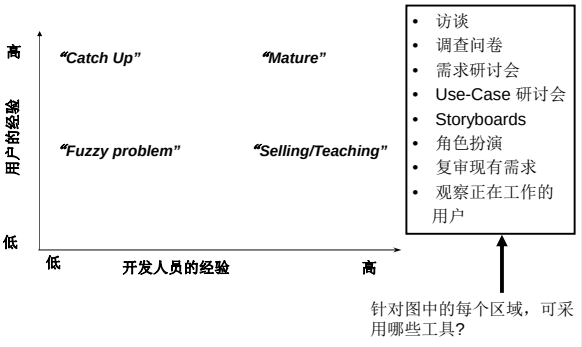
5）识别需求的来源



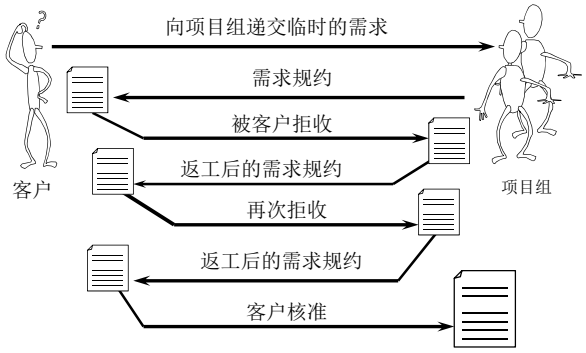
6）收集需求

- 项目干系人需求的收集技术：
  - 访谈
  - 调查问卷
  - 需求研讨会
  - Use-Case 研讨会
  - Storyboards
  - 角色扮演
  - 复审现有需求
  - 观察正在工作的用户

获取需求：采用哪种技术?



需求获取过程



7）产品定位

举例：选课系统的产品定位 @Vision文档

For	Wylie College students, professors, and the course registrar
Who	Attend, teach, or administer college courses
The Course Registration System	Is a tool
That	Enables online course registration and access to course and grade information
Unlike	The existing outdated mainframe registration system
Our product	Provides up-to-date information on all courses, registrations, teachers, and grades to all users from any PC connected via the College LAN or internet.

8）撰写产品特性

举例：选课系统的产品特性 @Vision文档

- Logon
- Register for Courses
- Course Cancellations
- Student Billings
- Enter, Update, and View Professor Information
- View Student Grades
- Select Courses to Teach
- Enter, Update, and View Student Information
- Record Student Grades
- View Course Catalog Information
- View Course Schedule
- Monitor for Course Full

9）定义质量范围

举例：选课系统的质量范围@Vision文档

- Availability:
  - The System shall be available 24 hours a day, 7 days a week.
- Usability:
  - The System shall be easy-to-use and shall be appropriate for the target market of computer-literate students and professors.
  - The System shall include online help for the user. Student and Professor users should not require the use of a hardcopy Manual to use the System.
- Maintainability:
  - The System shall be designed for ease of maintenance. All college-specific data should be table-driven and modifiable without recompilation of the System.
- **Performance Requirements**
  - The system shall support up to 2000 simultaneous users against the central database at any given time, and up to 500 simultaneous users against the local servers at any one time.
  - The system shall provide access to the legacy Course Catalog Database with no more than a 10 second latency.
  - The system shall complete 80% of all transactions within 2 minutes.

10）定义文档需求

举例：选课系统的文档需求 @Vision文档

- User Manual
- On-line Help
- Installation Guides, Configuration, Read Me File
- Labeling and Packaging

11）建立项目范围

- Feature 1: The system must...
- Feature 2: The system must...
- Feature 3: The system must...
- Feature 4: The system must...
- Feature n: The system must...

How do we know what the needs are?
How do we determine priority?
Where do we set the baseline?

△ Project Start Date

△ Target Release Date

Time

使用需求属性排列特性的优先级

	状态	风险	重要性	工作量	成本
✓ Feature Reqt. 10	Approved	Low	High		$$$
Feature Reqt. 13	Proposed	Med.	Low		$$
✓ Feature Reqt. 40	Approved	High	Mandatory		$

其他属性包括稳定性、技术难度等

12）划分特性优先级

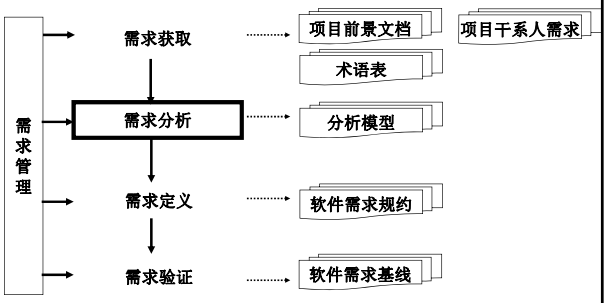
举例：选课系统的特性优先级 @Vision文档

- Release 1 :
  - Logon
  - Register for Courses
  - Interface to Course Catalog Database
  - Maintain Student Information
  - Maintain Professor Information
- Release 2 :
  - Submit Student Grades
  - View Grades
  - Select Courses to Teach

需求工程

- 软件需求概念
- 需求获取
- 需求分析和建模
- 需求定义和验证
- 需求管理

需求工程



分析建模准则

- Represent the information domain
- Represent software functions
- Represent software behavior
- Partition these representations
- Move from essence toward implementation

分析模型

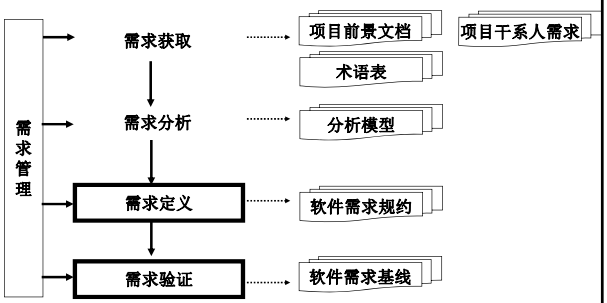
- 对需求进行分析，并进行图形建模，形成分析模型（平台无关模型PIM）

- 结构化分析模型
  - 数据流图（DFD）
  - 控制流图（CFD）
  - 数据字典（DD）
  - 实体—关系图（ERD）
  - 状态变迁图（STD）
  - 加工说明（PSPEC）
  - 控制说明（CSPEC）
- 面向对象分析模型
  - 用例图
  - 活动图
  - 类图
  - 时序图
  - 通信图
  - 状态机图

需求工程

- 软件需求概念
- 需求获取
- 需求分析和建模
- 需求定义和验证
- 需求管理

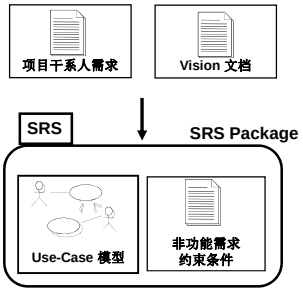
需求工程



需求定义的任务

定义详细需求- SRS
- 细化功能需求
- 细化非功能需求
- 细化约束条件
- 设计用户界面和接口

成果：软件需求规约



软件需求规约 (SRS) 定义了系统的外在行为和属性。

细化需求

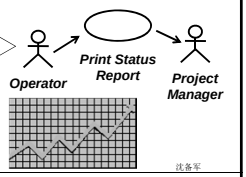
- 细化功能需求
  - 引入分析模型
- 细化非功能需求
- 细化约束条件

详细功能需求

Trending information will be charted with a line graph showing time on the x axis, and number of defects found on the y axis.

概要功能需求

Feat 63 - the defect tracking system will provide trending information to help the project manager assess project status

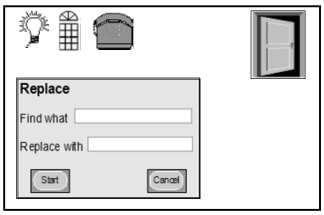


设计用户界面和接口

- 如果和外界的软件系统或硬件进行交互，则需定义通讯协议
  - 如果采用现有的协议，在功能说明中描述
  - 如果采用新的协议， 在设计时进行完整描述
- 如果和用户交互，则需简要设计用户界面
  - 图纸（在纸上）
  - 位图（采用绘图工具），或
  - 可执行代码（交互式的电子界面原型）

如何描述用户界面

- 可在功能说明中纳入屏幕框图，也可以是一个独立的用户界面原型
- 注意不要关注太多的界面设计细节



SRS模板 （采用传统技术）

1. 简介
2. 整体说明
3. 具体需求
   - 3.1 功能
   - 3.2 可用性
   - 3.3 可靠性
   - 3.4 性能
   - 3.5 可支持性
   - 3.6 设计约束
   - 3.7 联机用户文档和帮助系统需求
   - 3.8 购买的构件
   - 3.9 接口
   - 3.10 许可需求
   - 3.11 法律、版权及其他声明
   - 3.12 适用的标准
4. 支持信息
   索引、附录等

SRS模板 （采用Use-Case技术）

1. 引言
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions, Acronyms, and Abbreviations
   - 1.4 References
   - 1.5 Overview
2. 综述
   - 2.1 Use-Case Model Survey
   - 2.2 Assumptions and Dependencies
3. 具体需求
   - 3.1 Use-Case 报告
     - 3.1.1 <Use Case 1>
     - 3.1.2 ...
   - 3.2 补充说明
     - 3.2.1 可用性需求
     - 3.2.2 ...
4. 支持信息

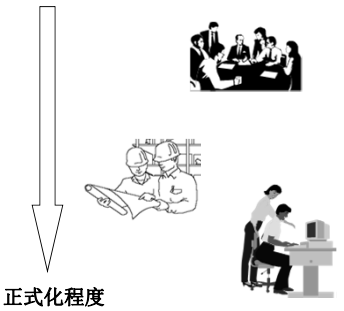
   索引、附录、用户界面原型等

需求验证

- **原型确认**
  - 抛弃型原型确认
  - 演进型原型确认
- 需求评审
  - 评审需求文档（Vision和SRS等），及时发现缺陷，寻找改进的契机，同时从评审反馈中获得知识，补充了正规的交流和培训机制，帮助团队建立对产品的共同理解

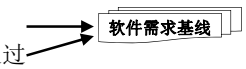
需求评审方法

- 审查
- 小组评审
- 走查
- 结对编程
- 同级桌查
- 轮查
- 临时评审



需求评审的输入和输出

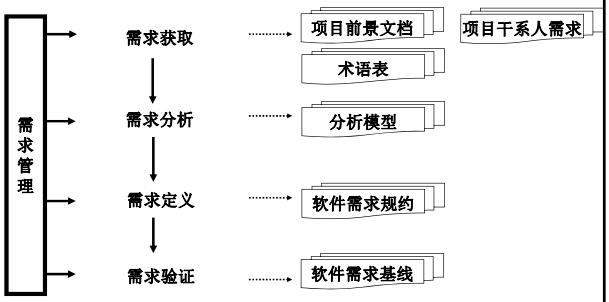
- 输入
  - 待评审的需求文档
  - Check list
- 输出
  - 评审结论
    - 通过
    - 有条件通过
    - 不通过
  - 缺陷清单



需求工程

- 软件需求概念
- 需求获取
- 需求分析和建模
- 需求定义和验证
- 需求管理

需求工程



需求管理

1）定义需求基线
2）需求变更控制和版本控制（建立新的需求基线）
3）需求跟踪

1）建立需求基线

- Feature 1: The system must...
- Feature 2: The system must...
- Feature 3: The system must...
- Feature 4: The system must...
- Feature n: The system must...

How do we know what the needs are?

How do we determine priority?

Where do we set the baseline?

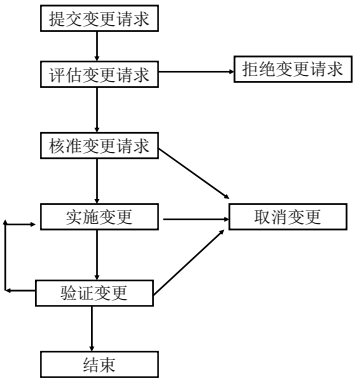
△ Project Start Date

△ Target Release Date

Time

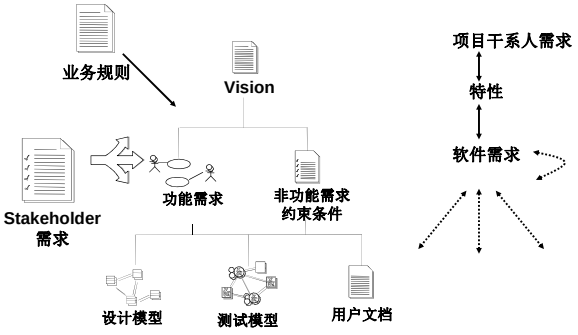
Software Engineering 67 沈备军

2）需求变更控制



Software Engineering 68 沈备军

3）需求跟踪



Software Engineering 69 沈备军

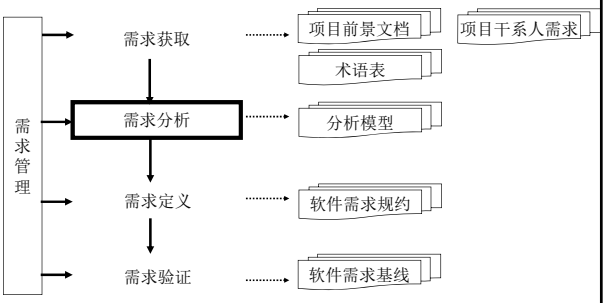
Shanghai Jiao Tong University

软件工程

Module : 构建分析模型

上海交通大学软件工程中心

需求工程



发现、获取、组织、 分析、编写和管理需求的系统方法，让客户和项目组之间达成共识。

构建分析模型

- 需求工程中的重要环节
- 分析模型是平台无关模型
- 关注 What, Not How
- 建模方法
  - 结构化分析
  - 面向对象分析
  - ……

@第3章和第5章.教材

Shanghai Jiao Tong University

软件工程
Module: 结构化分析

上海交通大学软件工程中心

结构化分析

- 结构化分析方法概述
- 数据流图
- 分层数据流图的审查
- 数据字典
- 描述基本加工的小说明
- 实体—关系图

@3.2.1节.教材

结构化方法

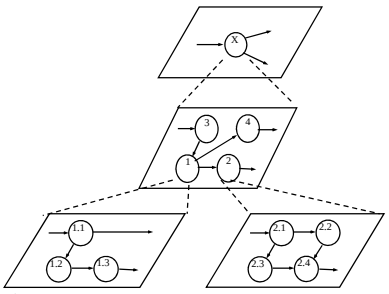
- 一种面向数据流的传统软件开发方法
- 以数据流为中心构建软件的分析模型、设计模型和实现模型
- 分为：
  - 结构化分析(Structured Analysis，简称SA)
  - 结构化设计(Structured Design，简称SD)
  - 结构化编程(Structured Programming，简称SP)

结构化方法的思想

- 发展历史
  - 提出：20世纪60年代末到70年代初
  - 成熟：20世纪70年代末到80年代中期
- 主要思想：抽象与自顶向下的逐层分解(控制复杂性的两个基本手段)
  - 抽象：从作为整体的软件系统开始(第一层),每一抽象层次上只关注于系统的输入输出
  - 分解：将系统不断分解为子系统、模块……

结构化分析方法中的抽象与分解

- 随着分解层次的增加，抽象的级别越来越低，也越接近问题的解(算法和数据结构)



结构化分析模型

- 数据字典是模型的核心，它包含了软件使用和产生所有数据的描述
- 数据流图：用于功能建模，描述系统的输入数据流如何经过一系列的加工变换逐步变换成系统的输出数据流
- 实体—关系图：用于数据建模，描述数据字典中数据之间的关系
- 状态转换图：用于行为建模，描述系统接收哪些外部事件，以及在外部事件的作用下的状态迁移情况

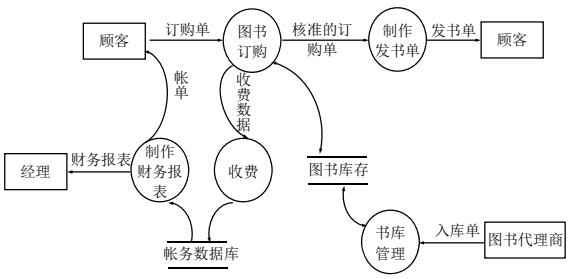
结构化分析

- 结构化分析方法概述
- 数据流图
- 分层数据流图的审查
- 数据字典
- 描述基本加工的小说明
- 实体—关系图

数据流图

- Data Flow Diagram(简称DFD)：描述输入数据流到输出数据流的变换(即加工)过程，用于对系统的功能建模，基本元素包括：
  - → 数据流(data flow)：由一组固定成分的数据组成，代表数据的流动方向
  - ○ 加工(process)：描述了输入数据流到输出数据流的变换，即将输入数据流加工成输出数据流
  - ═ 文件(file)：使用文件、数据库等保存某些数据结果供以后使用
  - □ 源或宿(source or sink)：由一组固定成分的数据组成，代表数据的流动方向

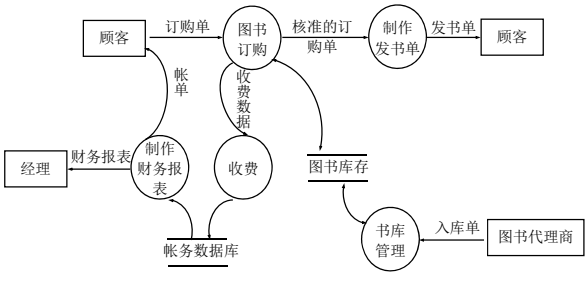
示例：图书订购系统DFD



源或宿 □

- 存在于软件系统之外的人员或组织，表示软件系统输入数据的来源和输出数据的去向，因此也称为源点和终点
  - 例如，对一个考务处理系统而言
    - 考生向系统提供报名单(输入数据流)，所以考生是考试系统(软件)的一个源
    - 考务处理系统要将考试成绩的统计分析表(输出数据流)传递给考试中心，所以考试中心是该系统的一个宿
- 源或宿用相同的图形符号表示
  - 当数据流从该符号流出时表示是源
  - 当数据流流向该符号时表示是宿
  - 当两者皆有时表示既是源又是宿

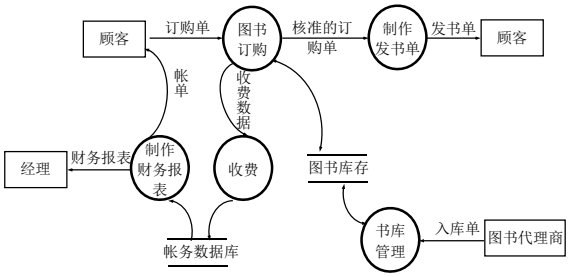
示例：图书订购系统DFD



加工和文件 ○ ═

- 加工：描述输入数据流到输出数据流的变换
  - 每个加工用一个定义明确的名字标识
  - 至少有一个输入数据流和一个输出流
  - 可以有多个输入数据流和多个输出数据流
- 文件：保存数据信息的外部单元
  - 每个文件用一个定义明确的名字标识
  - 由加工进行读写
  - DFD中称为文件，但在具体实现时可以用文件系统实现也可以用数据库系统等实现

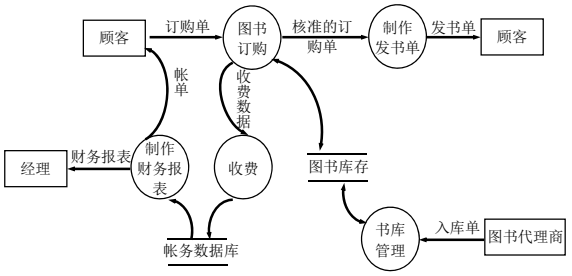
示例：图书订购系统DFD



数据流 →

- 每个数据流用由一组固定成分的数据组成并拥有一个定义明确的名字标识
  - 如：运动会管理系统中，报名单(数据流)由队名、姓名、性别、参赛项目等数据组成
- 数据流的流向
  - 从一个加工流向另一个加工
  - 从加工流向文件(写文件)
  - 从文件流向加工(读文件)
  - 从源流向加工
  - 从加工流向宿

示例：图书订购系统DFD



数据流图的扩充符号

- 描述一个加工的多个数据流之间的关系
  - 星号(*)：表示数据流之间存在“与”关系
    - 所有输入数据流同时存在时，才能进行加工处理
    - 或加工处理的结果是同时产生所有输出数据流
  - 加号(+)：表示数据流之间存在“或”关系
    - 至少存在一个输入数据流时才能进行加工处理
    - 或加工处理的结果是至少产生一个输出数据流
  - 异或(⊕)：表示数据流之间存在“异或”(互斥)关系
    - 必须存在且仅存在一个输入数据流时，才能进行加工处理
    - 或加工处理的结果是产生且仅产生一个输出数据流

对数据流图进行分层

- George Miller在著名的论文“神奇的数字7加减2：我们处理信息的能力的某种限制”中指出：人们在一段时间内的短期记忆似乎限制在5～9件事情之内
- 根据自顶向下逐层分解的思想将数据流图画成层次结构
- 每个层次画在独立的数据流图中，加工个数可大致控制在“7±2”的范围中

数据流图的各个层次

- 顶层图
  - 只有代表整个软件系统的1个加工，描述了软件系统与外界(源或宿)之间的数据流
  - 加工不必编号
- 0层图
  - 顶层图中的加工经分解后的图，只有1张
  - 加工编号分别为1，2，3，…
- X层图
  - 图的编号：若父图中的加工号x分解成某一子图，则该子图号记为“图x”
  - 加工的编号：若父图中的加工号为x的加工分解成某一子图，则该子图中的加工编号分别为x.1、x.2、x.3…

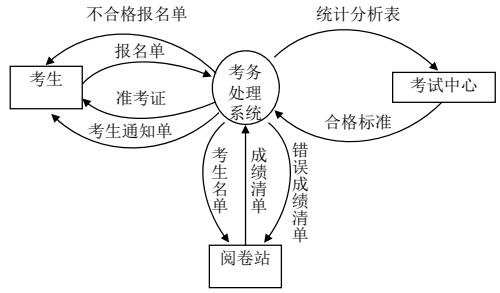
分层数据流图举例

- 简化的资格和水平考试的考务处理系统
  - 分成多个级别，如初级程序员、程序员、高级程序员、系统分析员等，凡满足一定条件的考生都可参加某一级别的考试
  - 考试的合格标准将根据每年的考试成绩由考试中心确定
  - 考试的阅卷由阅卷站进行，因此，阅卷工作不包含在软件系统中

示例：考务处理系统功能需求

- 对考生送来的报名单进行检查
- 对合格的报名单编好准考证号后将准考证送给考生，并将汇总后的考生名单送给阅卷站
- 对阅卷站送来的成绩清单进行检查，并根据考试中心制订的合格标准审定合格者
- 制作考生通知单送给考生
- 进行成绩分类统计(按地区、年龄、文化程度、职业、考试级别等分类)和试题难度分析，产生统计分析表

示例：考务处理系统顶层图



示例：部分数据流的组成

- 报名单＝地区＋序号＋姓名＋文化程度＋职业＋考试级别＋通信地址
- 正式报名单＝准考证号＋报名单
- 准考证＝地区＋序号＋姓名＋准考证号＋考试级别＋考场
- 考生名单＝｛准考证号＋考试级别｝
  其中｛w｝表示w重复多次
- 考生名册＝正式报名单
- 统计分析表＝分类统计表＋难度分析表
- 考生通知单＝准考证号＋姓名＋通信地址＋考试级别＋考试成绩＋合格标志

DFD细化－确定加工

- 将父图中某加工分解而成的子加工
  - 根据功能分解来确定加工：将一个复杂的功能分解成若干个较小的功能，较多应用于高层DFD中的分解
  - 根据业务处理流程确定加工：分析父图中待分解加工的业务处理流程，业务流程中的每一步都可能是一个子加工
  - 特别要注意在业务流程中数据流发生变化或数据流的值发生变化的地方，应该存在一个加工

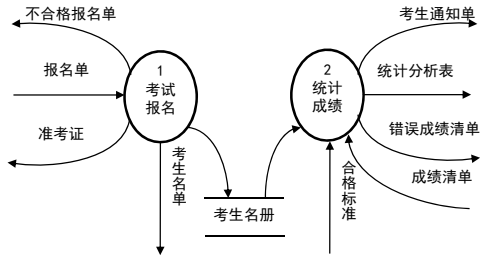
DFD细化－确定数据流

- 在父图中某加工分解而成的子图中，父图中相应加工的输入/输出数据流都是且仅是子图边界上的输入/输出数据流
- 分解后的子加工之间应增添相应的新数据流表示加工过程中的中间数据
- 如果某些中间数据需要保存以备后用，那么可以成为流向文件的数据流
- 同一个源或加工可以有多个数据流流向一个加工，如果它们不是一起到达和一起加工的，那么可以将它们分成若干个数据流

DFD细化－确定文件

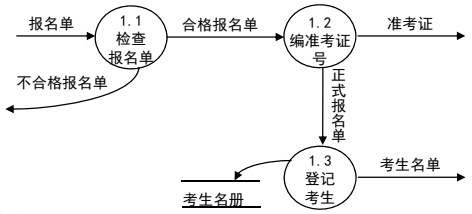
- 如果父图中该加工存在读写文件的数据流，则相应的文件和数据流都应画在子图中
- 在分解子图中，如果需要保存某些中间数据以备后用，则可以将这些数据组成一个新的文件
- 新文件(首次出现的文件)至少应有一个加工为其写入记录，同时至少存在另一个加工来读该文件的记录
- 注意：从父图中继承下来的文件在子图中可能只对其进行读，或只进行写

示例：考务处理系统0层图

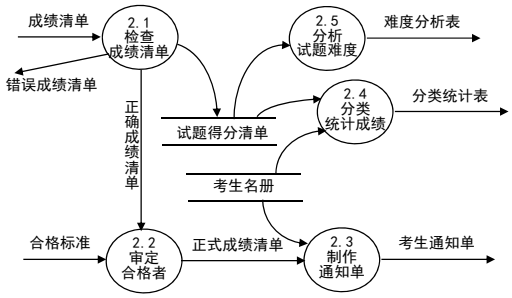


示例：考务处理系统－考试报名1子图

- 3个子加工：检查报名单、编准考证号、登记考生
- “合格报名单”和“正式报名单”是新增加的数据流，其它数据流都是加工1原有的
- 在加工1的分解中没有新的文件产生



示例：考务处理系统－统计成绩2子图



总结：画分层数据流图的步骤

1. 画系统的输入和输出
2. 画系统内部
3. 画加工内部
4. 重复第3步，直至每个尚未分解的加工都足够简单(即不必再分解)

DFD练习，画出顶层图和0层图

- 一个大城市的公共工作部门决定开发一个基于Web的坑洼跟踪和修补系统(PHTRS)：
  - **当报告路面有坑洼时，它们被赋予一个标识号，并依据街道地址、大小(1到10)、位置(路中或路边等)、地区(由街道地址确定)和修补优先级(由坑洼的大小确定)储存起来。**
  - **工作单数据和每个坑洼有关，数据包括坑洼位置和大小、修理组标识号、修理组的人数、分配的设备、修复时间长度、坑洼状态(正在处理中、已修复、临时修复、未修复)、使用填料的数量和修复的开销(由使用的时间、人数、使用的材料的设备计算得到)。**
  - **最后，产生一个损失文件以保存该坑洼造成的损失信息，并包括居民的姓名、地址、电话号码、损失类型和损失钱数。**
  - **PHTRS是基于Web的系统，可交互地进行所有的查询。**

结构化分析

- 结构化分析方法概述
- 数据流图
- 分层数据流图的审查
- 数据字典
- 描述基本加工的小说明
- 实体—关系图

分层数据流图的审查

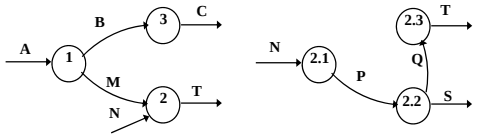
- 检查图中是否存在错误或不合理(不理想)的部分
  - 一致性：分层DFD中不存在矛盾和冲突
  - 完整性：分层DFD本身的完整性，即是否有遗漏的数据流、加工等元素

分层数据流图的一致性

- 父图与子图平衡
  - 任何一张DFD子图边界上的输入/输出数据流必须与其父图中对应的加工的输入/输出数据流保持一致
- 数据守恒
  - 一个加工所有输出数据流中的数据，必须能从该加工的输入数据流中直接获得，或者能通过该加工的处理而产生
  - 多余的数据流：加工未使用其输入数据流中的某些数据项

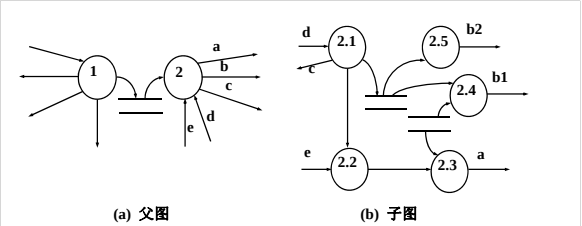
父图与子图不平衡的实例

- 加工2的输入数据流有M和N，输出数据流是T
- 而子图(右图)边界上的输入数据流是N，输出数据流是S和T



父图与子图平衡的实例

- 注意：如果父图某加工的一个数据流，对应于子图中几个数据流，而子图中组成这些数据流的数据项全体正好等于父图中的这个数据流，那么它们仍算是平衡的

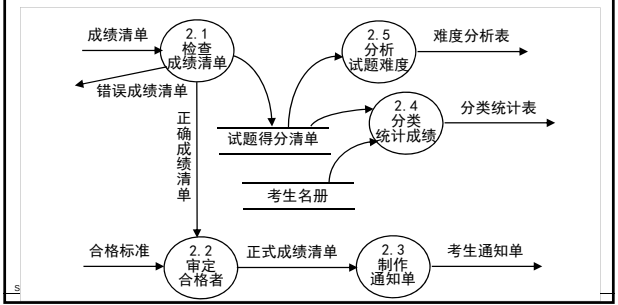


(a) 父图　　(b) 子图

a：考生通知单；b：统计分析表；b1：分类统计表；b2：难度分析表；c：错误成绩清单；d成绩清单；e合格标准。

数据不守恒的实例

由于“正式成绩清单”中缺少“考生通知单”中的姓名、通信地址等数据，这些数据也无法由加工2.3自己产生，因此，加工2.3不满足数据守恒的条件



分层数据流图的完整性

- 每个加工至少有一个输入数据流和一个输出数据流
- 在整套分层数据流中，每个文件应至少有一个加工读该文件，有另一个加工写该文件
- 分层数据流图中的每个数据流和文件都必须命名(除了流入或流出文件的数据流)，并保持与数据字典的一致
- 分层DFD中的每个基本加工(即不再分解子图的加工)都应有一个加工规约

其它需注意的问题-1

- 适当命名：每个数据流、加工、文件、源和宿都应被适应地命名，名字应符合被命名对象的实际含义
  - 名字应反映整个对象(如数据流、加工)，而不是仅反映它的某一部分
  - 避免使用空洞的、含义不清的名字，如数据、信息、处理、统计等
  - 如果发现某个数据流或加工难以命名时，往往是DFD分解不当的征兆，此时应考虑重新分解
- 画数据流而不是画控制流
  - 判断准则：这条线上是否有数据流过

其它需注意的问题-2

- 分解尽可能均匀
  - 理想目标：任何两个加工的分解层数之差不超过1
  - 应尽可能使分解均匀，对于分解不均匀的情况应重新分解
- 先考虑稳定状态，忽略琐碎的枝节
  - 先考虑稳定状态下的各种问题，暂时不考虑系统如何启动、如何结束、出错处理以及性能等问题
- 随时准备重画
  - 对于一个复杂的软件系统，往往要经过反复多次的重画和修改才能构造出完整、合理、满足用户需求的分层DFD
  - 分析阶段遗漏下来的一个错误，到开发后期要化费几百倍代价来纠正这个错误

分解的程度

- 可参照以下几条与分解有关的原则：
  - 7±2
  - 分解应自然，概念上合理、清晰
  - 只要不影响DFD的易理解性，可适当多分解几个加工，以减少层数
  - 一般说来，上层分解得快些(即多分解几个加工)，下层分解得慢些(即少分解几个加工)
  - 分解要均匀

结构化分析

- 结构化分析方法概述
- 数据流图
- 分层数据流图的审查
- 数据字典
- 描述基本加工的小说明
- 实体—关系图

数据字典

- 数据流图与数据字典是密不可分的，两者结合起来构成软件的分析模型
- 数据字典由字典条目组成，每个条目描述DFD中的一个元素
- 数据字典条目包括：数据流、文件、加工、源或宿

数据字典的描述符号

符 号	名 称	举 例
=	定义为	x=… 表示x由…组成
+	与	a+b 表示a和b
[…, …]	或	[a，b] 表示a或b
[…|…]	或	[a|b] 表示a或b
{…}	重复	{a} 表示a重复0或多次
$\{\ldots\}_m^n$	重复	$\{a\}_3^8$ 表示a重复3到8次
(…)	可选	(a) 表示a重复0或1次
"…"	基本数据元素	"a" 表a是基本数据

数据流条目的描述内容

- 名称：数据流名(可以是中文名或英文名)
- 别名：名称的另一个名字
- 简述：对数据流的简单说明
- 数据流组成：描述数据流由哪些数据项组成
- 数据流来源：描述数据流从哪个加工或源流出
- 数据流去向：描述数据流流入哪个加工或宿
- 数据量：系统中该数据流的总量
  - 如考务处理系统中“报名单”的总量是100000张
  - 或者单位时间处理的数据流数量，如80000张/天
- 峰值：某时段处理的最大数量
  - 如每天上午9：00至11：00处理60000张表单
- 注解：对该数据流的其它补充说明

数据流组成

- 数据流组成是数据流条目的核心，它列出组成该数据流的各数据项，例如：
  - 培训报名单＝姓名＋单位＋课程
  - 运动员报名单＝队名＋姓名＋性别＋｛参赛项目｝$_1^3$
- 当一个数据流的组成比较复杂时，可以将其分解成几个数据流，例如：
  - 课程＝课程名＋任课教师＋教材＋时间地点
  - 时间地点＝｛星期几＋第几节＋教室｝

文件条目的描述内容

- 名称：文件名
- 别名：同数据流条目
- 简述：对文件的简单说明
- 文件组成：描述文件的记录由哪些数据项组成(与数据流条目中的文件组成描述方法相同)
- 写文件的加工：描述哪些加工写文件
- 读文件的加工：描述哪些加工读文件
- 文件组织：描述文件的存储方式(顺序、索引)，排序的关键字
- 使用权限：描述各类用户对文件读、写、修改的使用权限
- 数据量：文件的最大记录个数
- 存取频率：描述对该文件的读写频率
- 注解：对该文件的其它补充说明

加工条目的描述内容

- 名称：加工名
- 别名：同数据流条目
- 加工号: 加工在DFD中的编号
- 简述：对加工的功能的简要说明
- 输入数据流：描述加工的输入数据流，包括读哪些文件名
- 输出数据流：描述加工的输出数据流，包括写哪些文件名
- 加工逻辑：简要描述加工逻辑，或者对加工规约的索引
  - 基本加工的加工逻辑用小说明描述，在加工条目中可填写对加工规约的索引
  - 非基本加工分解而成的DFD子图已反映了它的加工逻辑，不必书写小说明
- 异常处理：描述加工处理过程中可能出现的异常情况，及其处理方式
- 加工激发条件：描述执行加工的条件，如“身份认证正确”，“收到报名单”
- 执行频率：描述加工的执行频率，如，每月执行一次，每天0点执行
- 注解：对加工的其它补充说明

源或宿条目的描述内容

- 名称：源或宿的名(外部实体名)
- 别名：同数据流条目
- 简要描述：对源或宿的简要描述(包括指明该外部实体在DFD中是用作“源”，还是“宿”，还是“既是源又是宿”)
- 输入数据流：描述源向系统提供哪些输入数据流
- 输出数据流：描述系统向宿提供哪些输出数据流
- 注解：对源或宿的其它补充说明

DD练习

- 一个大城市的公共工作部门决定开发一个基于Web的坑洼跟踪和修补系统(PHTRS)：
  - 当报告路面有坑洼时，它们被赋予一个标识号，并依据街道地址、大小(1到10)、位置(路中或路边等)、地区(由街道地址确定)和修补优先级(由坑洼的大小确定)储存起来。
  - 工作单数据和每个坑洼有关，数据包括坑洼位置和大小、修理组标识号、修理组的人数、分配的设备、修复时间长度、坑洼状态(正在处理中、已修复、临时修复、未修复)、使用填料的数量和修复的开销(由使用的时间、人数、使用的材料的设备计算得到)。
  - 最后，产生一个损失文件以保存该坑洼造成的损失信息，并包括居民的姓名、地址、电话号码、损失类型和损失钱数。
  - PHTRS是基于Web的系统，可交互地进行所有的查询。

结构化分析

- 结构化分析方法概述
- 数据流图
- 分层数据流图的审查
- 数据字典
- 描述基本加工的小说明
- 实体—关系图

基本加工的小说明

- 小说明是基本加工的规约说明，应精确地描述用户要求一个加工“做什么”
- 包括加工的激发条件、加工逻辑、优先级、执行频率、出错处理等
- 最基本的部分是加工逻辑，即该加工的输出数据流与输入数据流之间的逻辑关系
- 加工逻辑不是对加工的设计，不涉及数据结构、算法实现、编程语言等与设计和实现有关的细节

加工逻辑的描述方法

- 结构化语言：介于自然语言和形式语言之间的一种半形式语言
- 判定表：适用于加工逻辑包含多个条件，而不同的条件组合需做不同的动作
- 判定树：判定表的变种，它本质上与判定表是相同的，只是表示形式不同

结构化语言

- 没有严格的语法
- 加工规约分为若干个段落，每个段落可分为内外两层：
  - 外层有严格的语法来描述它的控制结构
    - 如结构化英语中可使用if_then_else、while_do、repeat_until、for_do、case等结构
  - 内层可以用自然语言来描述
- 允许使用嵌套结构

“计算信用度”的结构化英语描述

```
Case 1 (No Bounced−Checks in Customer Record):
        Write Exemplary−Customer−Citation to Annual−Summary.
Case 2 (One Bounced−check):
        If Yearly−Average−Balance exceeds $ 1000.
            Remove Bounced−Check from Customer−Record.
        Otherwise.
        Recuce Credit−Limit by 10%.
Case 3 (Multiple Bounced−Checks):
        For each Bounced−Check.
            Recuce Credit−Limit by 15%.
        Set Credit−Rating to Deadbeat.
        Write Scathing−Comment to Annual−Summary.
        Write Customer−Name−and−Address to IRS−Enemies−List.
```

结构化语言书写加工规约注意事项

- 语句力求精炼
- 语句必须易读、易理解、无二义
- 主要使用祈使句，祈使句中的动词要明确表达要执行的动作
- 所有名字必须是数据字典中有定义的名字
- 不使用形容词、副词等修饰语
- 不使用含义相同的动词，如“修改”、“修正”等
- 可以使用常用的算术和关系运算符
- 总之要尽可能精确、无二义、简明扼要、易理解

判定表

- 判定表的组成元素
  - 条件桩(Condition Stub)：列出各种条件的对象，如发货单金额，赊欠天数等，每行写一个条件对象
  - 条件条目(Condition entry)：列出各条件对象的取值，条件条目的每一列表示了一个可能的条件组合
  - 动作桩(action stub)：列出所有可能采取的动作，如发出发货单等，每行写一个动作
  - 动作条目(action entry)：列出各种条件组合下应采取的动作

发货单金额	＞500	＞500	≤500	≤500
赊欠天数	＞60	≤60	＞60	≤60
发不批准通知	√			
发出批准书		√	√	√
发出发货单		√	√	√
发出赊欠报告			√	

“审批发货单”加工的判定表

判定表的其它形式

发货单金额	＞500	≤500	—
赊欠天数	＞60	＞60	≤60
发不批准通知	√		
发出批准书		√	√
发出发货单		√	√
发出赊欠报告		√	

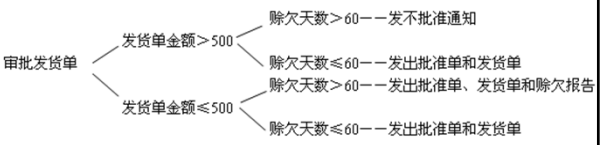
“审批发货单”加工的简化判定表

发货单金额≤500	0	0	1	1
发货单金额＞500	1	1	0	0
赊欠天数≤60	0	1	0	1
赊欠天数＞60	1	0	1	0
发不批准通知	√			
发出批准单		√	√	√
发出发货单		√	√	√
发出赊欠报告			√	

“审批发货单”加工的另一种判定表

判定树

- 本质上与判定表是相同的，只是表示形式不同
- 例如“审批发货单”加工逻辑的判定树描述入下：



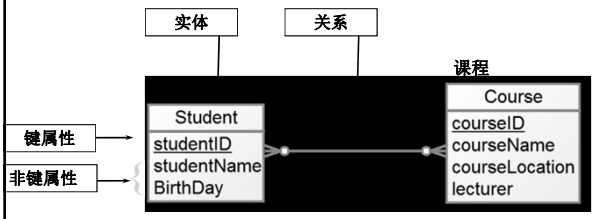
结构化分析

- 结构化分析方法概述
- 数据流图
- 分层数据流图的审查
- 数据字典
- 描述基本加工的小说明
- 实体—关系图

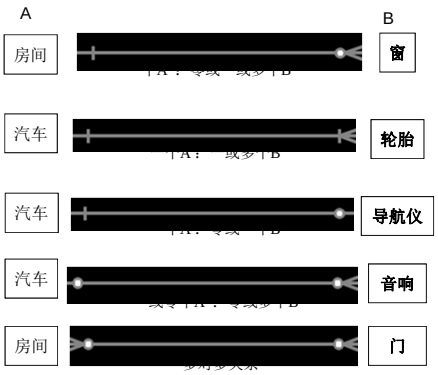
实体—关系图

- 实体—关系图（Entity-Relation Diagram, ERD）用于数据建模，描述数据字典中数据之间的关系
- 实体是客观存在的数据对象
  - 只封装数据，不封装数据的操作，和OO类不同
  - 例如: 学生, 学校, 事件, 植物
- 实体具有属性
  - 例如: 学生具有姓名和地址
- 实体间可存在多种不同关系
  - 例如：教师指导学生，学生选课

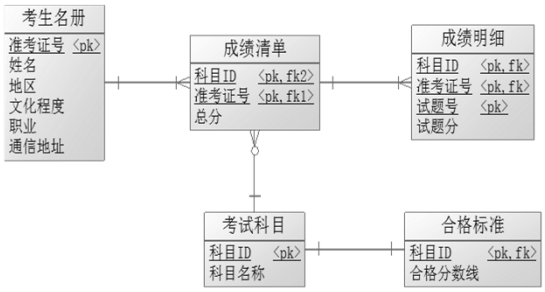
实体和关系



关系的基数



ERD 示例



ERD练习

- 一个大城市的公共工作部门决定开发一个基于Web的坑洼跟踪和修补系统(PHTRS)：
  - 当报告路面有坑洼时，它们被赋予一个标识号，并依据街道地址、大小(1到10)、位置(路中或路边等)、地区(由街道地址确定)和修补优先级(由坑洼的大小确定)储存起来。
  - 工作单数据和每个坑洼有关，数据包括坑洼位置和大小、修理组标识号、修理组的人数、分配的设备、修复时间长度、坑洼状态(正在处理中、已修复、临时修复、未修复)、使用填料的数量和修复的开销(由使用的时间、人数、使用的材料的设备计算得到)。
  - 最后，产生一个损失文件以保存该坑洼造成的损失信息，并包括居民的姓名、地址、电话号码、损失类型和损失钱数。
  - PHTRS是基于Web的系统，可交互地进行所有的查询。

Shanghai Jiao Tong University

软件工程
Module: 面向对象分析

上海交通大学软件工程中心

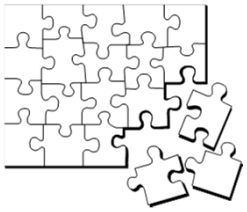
面向对象分析

- 面向对象方法概述
- 面向对象的基本概念
- 用例建模
- 建立概念模型
- 用例分析

@第3.3节,第5章.教材

What Is Object Technology?

- A set of principles (abstraction, encapsulation, polymorphism) guiding software construction, together with languages, databases, and other tools that support those principles. (*Object Technology - A Manager's Guide*, Taylor, 1997.)

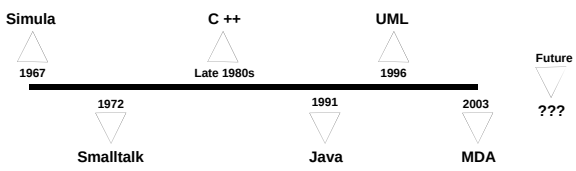


The Strengths of Object Technology

- Reflects a single paradigm
- Facilitates architectural and code reuse
- Reflects real world models more closely
- Encourages stability
- Is adaptive to change

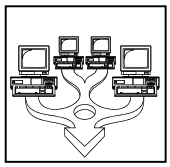
The History of Object Technology

- Major object technology milestones



Where Is Object Technology Used?

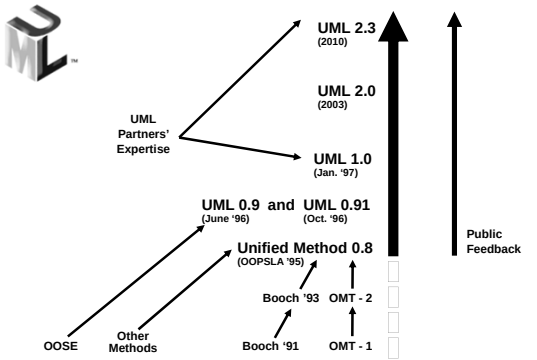
- Client/Server Systems and Web Development
- Real-time Systems
- Embedded System
- Multimedia System
- Middleware
- ……



Differences Between OO and Structured Method

- Object-orientation (OO)
  - Melds the data and data flow process together early in the lifecycle
  - Has a high level of encapsulation
  - Promotes reuse of code differently
  - Permits more software extensibility

History of the UML



面向对象方法的三个步骤

- 面向对象分析
  - Object Oriented Analysis, OOA
- 面向对象设计
  - Object Oriented Design, OOD
- 面向对象编程
  - Object Oriented Programming, OOP

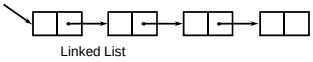
面向对象分析

- 面向对象方法概述
- 面向对象的基本概念
- 用例建模
- 建立概念模型
- 用例分析

What Is an Object?

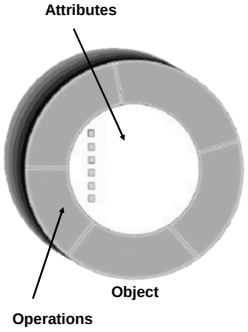
- Informally, an object represents an entity, either physical, conceptual, or software.
  - Physical entity 

  - Conceptual entity 

  - Software entity 

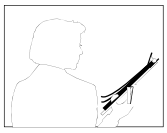
A More Formal Definition

- An object is an entity with a well-defined boundary and *identity* that encapsulates *state* and *behavior.*
  - State is represented by attributes and relationships.
  - Behavior is represented by operations, methods, and state machines.



Representing Objects in the UML

- An object is represented as a rectangle with an underlined name.



Professor J Clark

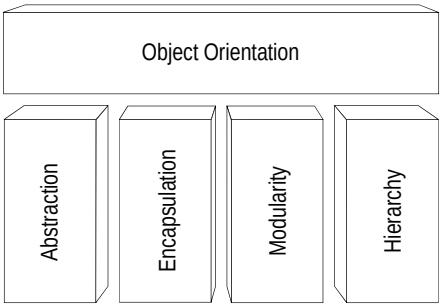
J Clark : Professor

Named Object

: Professor

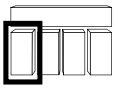
Anonymous Object

Basic Principles of Object Orientation



What Is Abstraction?

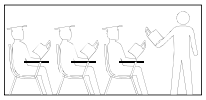
- The essential characteristics of an entity that distinguishes it from all other kinds of entities.
- Defines a boundary relative to the perspective of the viewer.
- Is not a concrete manifestation, denotes the ideal essence of something.



Example: Abstraction

Student

Professor

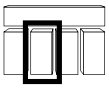


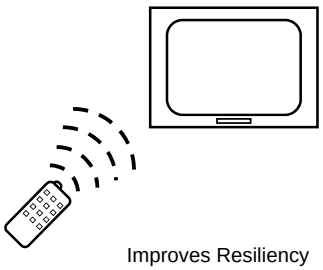
Course Offering (9:00 a.m., Monday-Wednesday-Friday)

Course (e.g. Algebra)

What Is Encapsulation?

- Hides implementation from clients.
  - Clients depend on interface.



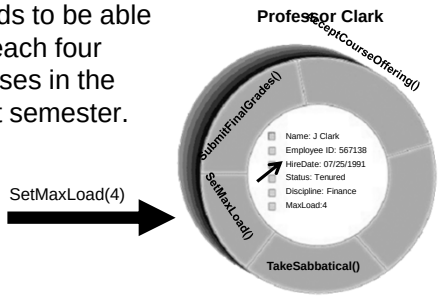


Improves Resiliency

Encapsulation Illustrated

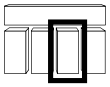
- Professor Clark needs to be able to teach four classes in the next semester.

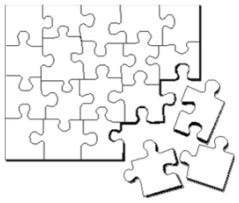
SetMaxLoad(4)



What Is Modularity?

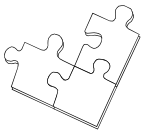
- Breaks up something complex into manageable pieces.
- Helps people understand complex systems.





Example: Modularity

- For example, break complex systems into smaller modules.



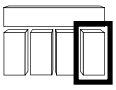
Course Registration System

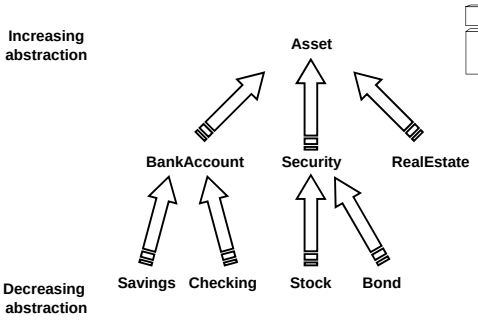
Billing System

Course Catalog System

Student Management System

What Is Hierarchy?





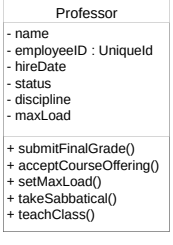
Elements at the same level of the hierarchy should be at the same level of abstraction.

What Is a Class?

- A class is a description of a set of objects that share the same *attributes*, *operations*, *relationships*, and *semantics*.
  - An object is an instance of a class.
- A class is an abstraction in that it
  - Emphasizes relevant characteristics.
  - Suppresses other characteristics.

Representing Classes in the UML

- A class is represented using a rectangle with three compartments:
  - The class name
  - The structure (attributes)
  - The behavior (operations)



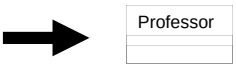
The Relationship between Classes and Objects

- A class is an abstract definition of an object.
  - It defines the structure and behavior of each object in the class.
  - It serves as a template for creating objects.
- Classes are not collections of objects.

Professor Torpie

Professor Meijer

Professor Allen



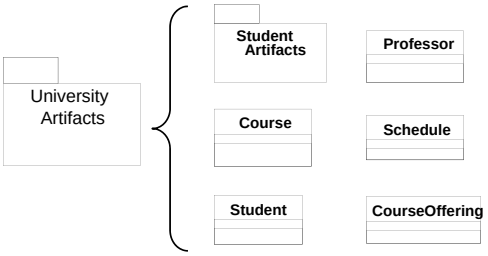
What Is a Package?

Modularity

- A general purpose mechanism for organizing elements into groups.
- A model element that can contain other model elements.
- A package can be used:
  - To organize the model under development.
  - As a unit of configuration management.

Package 示例

- The package, University Artifacts, contains one package and five classes.



面向对象分析

- 面向对象方法概述
- 面向对象的基本概念
- 用例建模
- 建立概念模型
- 用例分析

面向对象分析的步骤

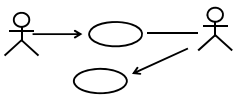
1. 用例建模
2. 建立概念模型
3. 用例分析

面向对象分析的步骤

1. 用例建模
   - 1.1识别actor和use case，画Use-Case图
   - 1.2 编写Use-Case Spec.
   - 1.3 优化Use-Case图的结构
2. 建立概念模型
3. 用例分析

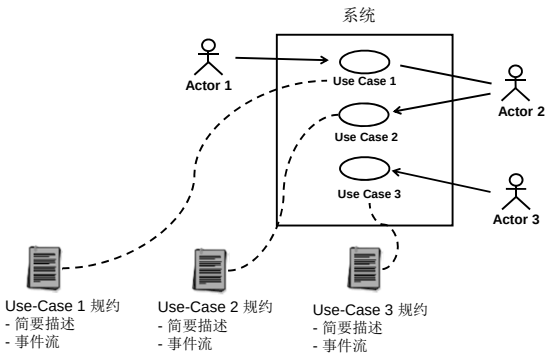
Use Case技术

- 提供涉众的观点
- 定义功能需求
- 促进理解和讨论
  - 为什么需要系统?
  - 谁和系统交互 (actors)?
  - 用户希望如何使用系统 (use cases)?
  - 系统应该有什么接口?



Use-Case 模型

Use-Case 模型的组成



面向对象分析的步骤

1. 用例建模
   - 1.1识别actor和use case，画Use-Case图
   - 1.2 编写Use-Case Spec.
   - 1.3 优化Use-Case图的结构
2. 建立概念模型
3. 用例分析

Actor 和 Use Case

Actor
和系统交互的系统外的某些人或某些东西：
- 最终用户
- 外界软件系统
- 外界硬件设备

Use case
Actor想使用系统去做的事

如何识别Actor

- 谁需要在系统的帮助下完成自己的任务？
- 需要谁去执行系统的核心功能？
- 需要谁去完成系统的管理和维护？
- 系统是否需要和外界的硬件或软件系统进行交互？

如何识别Use Case

- 每个actor的目标和需求是什么?
  - actor希望系统提供什么功能？
  - actor 将创建、存取、修改和删除数据吗？
  - actor是否要告诉系统外界的事件？
  - actor 需要被告知系统中的发生事件吗?

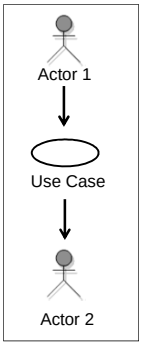
避免功能分解

现象
- use case太小
- use cases太多
- Uses case没有价值回报
- 命名以低层次的操作
  - “Operation” + “object”
  - “Function” + “data”
  - Example: “Insert Card”
- 很难理解整个模型

纠正措施
- 寻找更大的上下文
  - “为什么你要构建本系统?”
- 让自己站在用户的角度
  - “用户想得到什么”
  - “这个 use case 能满足谁的要求?”
  - “这个use case 能增值什么?”
  - “这个use case 背后有什么故事?”

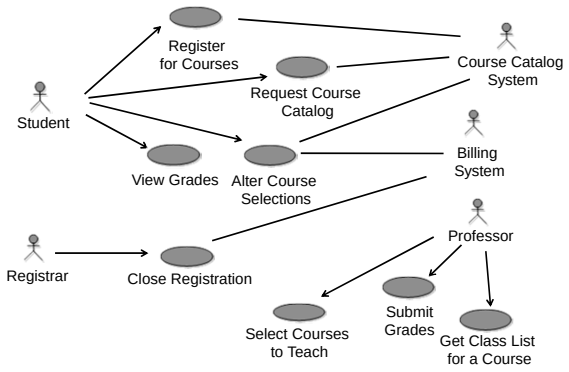
一个用例定义了和actor之间的一次完整对话。

通信-关联 Communicates-Association



- Actor和use case 间的通信渠道
- 用一条线表示
- 箭头表示谁启动通信

Course Registration System的用例图

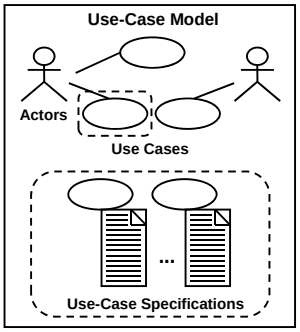


面向对象分析的步骤

1. 用例建模
   - 1.1识别actor和use case，画Use-Case图
   - 1.2 编写Use-Case Spec.
   - 1.3 优化Use-Case图的结构
2. 建立概念模型
3. 用例分析

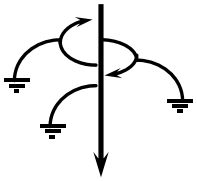
Use-Case Specifications

- Name
- Brief description
- Flow of Events
- Relationships
- Activity diagrams
- Use-Case diagrams
- Special requirements
- Pre-conditions
- Post-conditions
- Other diagrams



事件流（基本流和备选流）

- 一个基本流
  - Happy day scenario
  - Successful scenario from start to finish
- 多个备选流
  - Regular variant
  - Odd cases
  - Exceptional (error) flows



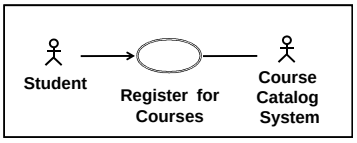
事件流举例: Get Quote

基本流 Basic Flow
1. Customer logs on
2. Customer selects 'Get Quote' function
3. Customer selects stock trading symbol
4. Get desired quote from Quote System
5. Display quote
6. Customer gets other quotes
7. Customer logs off

备选流 Alternative Flows
A1. Unidentified Trading Customer
A2. Quote System Unavailable
A3. Quit

What are other alternatives?

Scenario 是Use-Case 的实例



Scenario 1
Log on to system
Approve log on
Enter subject in search
Get course list
Display course list
Select courses
Confirm availability
Display final schedule

Scenario 2
Log on to system
Approve log on
Enter subject in search
Invalid subject
Re-enter subject
Get course list
Display course list
Select courses
Confirm availability
Display final schedule

细化基本事件流

Get Quote

Basic Flow

1. Customer Logs On
The use case starts when the Trading Customer logs on. The system validates the customer id and password. …
2. Customer Selects "Get Quote" Function
The Trading Customer chooses to "Get Quote." The system displays the list of securities on which it has quotes.
3. Customer Gets Quote
The Trading Customer selects from the list of securities or enters the trading symbol for a security. The system sends the trading symbol to the Quote System, and receives the Quote System Response.
4. Customer Gets Other Quotes
If the Trading Customer wishes to get additional quotes, the use case resumes at Step 3.
5. Customer Logs Off
The Trading Customer logs off the system. The use case ends.

将流分成几步

每步标识编号和名字

描述每步 (1-3 句)

每步是一个来回的事件

细化通用备选流

Identify Customer

Alternative Flows

A1. Unidentified Trading Customer
In Step 1 of the Basic Flow, if the system determines that the customer password is not valid, the system displays a "Sorry, not a valid customer…." message. The use case ends.

A2. Wrong Password
In Step 1 of the Basic Flow, if the system determines that the customer password id is not valid, the system displays a "Sorry, not…."

A3. Suspect Theft
In step 1 of the Basic Flow, if the customer id is on the system's list of stolen identification, the system displays a "Sorry, not …" message. The system also records the date, time. And computer address…

A4. Quit
The RU e-st System allows the Trading Customer to quit at any time during the use case. The use case ends.

A5. No Reply From User
At any time during the use case, if the system asks for input from the Trading Customer….

在任何事件流中都有发生

细化特殊备选流

Get Quote

Alternative Flows

A1 Unidentified Trading Customer
In Step 1, Customer Logs On, in the Basic flow, if the system determines that the customer id and/or password are not valid,…

A2 Quote System Unavailable
In Step 3, Customer Gets Quote, in the basic flow, if the system is unable to communicate with the Quote System, the system…

A3 Quit
The RU e-st System allows the Trading Customer to quit at any time during the use case. The use case ends.

A4 Unknown Trading Symbol
In Step 3, Customer Gets Quote, in the Basic Flow, if the system cannot recognize the trading symbol the system notifies the Trading Customer that the trading symbol is not recognizable. The use case continues at the beginning of Step 3...

A5 Quote System Cannot Locate Information
In Step 3, Customer Gets Quote, in the Basic Flow, if the Quote System responds that it does not have the requested information…

描述发生了什么

开始

条件

动作

继续

举例：备选流的另一种编号方法

基本流 Basic Flow
1. 采购员在初始申购单中添加申购子项。
2. 采购员输入供应商、交货地点、最终价格、返利、加价和费用明细。
3. 采购员增加、删除和修改申购单（包括第1、2步）直到满意为止。
4. 采购员在输入所有必要信息后，保存并完成申购单。
5. 系统检验申购单，分配申购单号，设置申购单和申购单子项状态为"未提交"。
6. 采购员提交申购单。
7. 系统设置申购单状态为"待审批"，通知采购经理审批申购单。

备选流 Alternative Flows
1-3a 退出：
1) 系统提示用户保存。
2) 用户选择不保存，系统放弃临时信息，申购单状态不变。
4a. 申购单信息不完整：
系统提示不能保存，回到步骤3。

What Is an Activity Diagram?

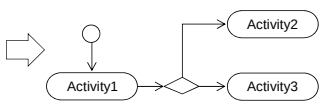
- An activity diagram in the Use-Case Model can be used to capture the activities in a use case.
- It is essentially a flow chart, showing flow of control from activity to activity.

Flow of Events

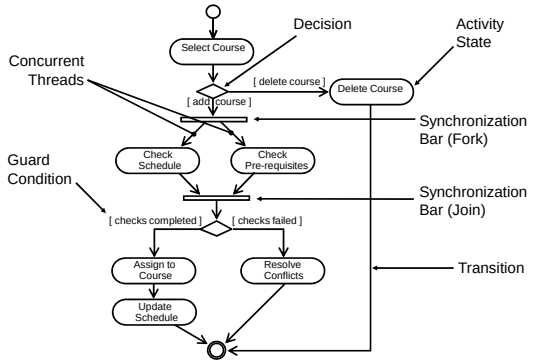
This use case starts when the Registrar requests that the system close registration.

1. The system checks to see if registration is in progress. If it is, then a message is displayed to the Registrar and the use case terminates. The Close Registration processing cannot be performed if registration is in progress.

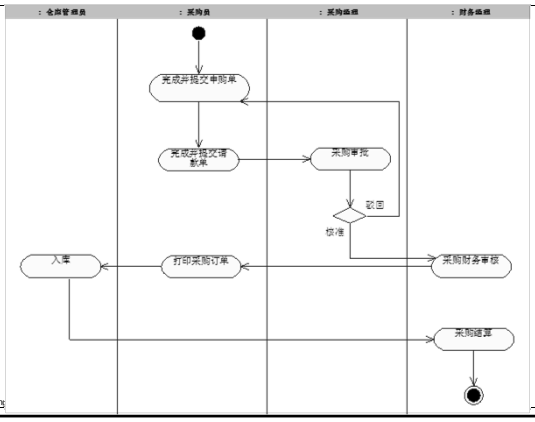
2. For each course offering, the system checks if a professor has signed up to teach the course offering and at least three students have registered. If so, the system commits the course offering for each schedule that contains it.



Example: Activity Diagram



Partitions



前置条件

- Use-case 启动的约束条件
- 不是触发Use-case 的事件
- 可选的: 仅当需要时才用

示例：

- 必须满足下面的前置条件，ATM 系统才能出钞：
  - 必须能进入 ATM 网络。
  - ATM 必须处于准备接受交易的状态。
  - ATM 中必须备有一些现金可供出钞。
  - ATM 必须备有足够的纸张打印至少一次交易的收据

后置条件

- 当use case结束时,后置条件一定为真
- 用于降低用例事件流的复杂性并提高其可读性
- 还可用来陈述在用例结束时系统执行的动作
- 可选的: 仅当需要时才用

- 示例：
  - ATM 在用例结束时总是显示“欢迎使用”消息，可将此消息记录在用例后置条件中。
  - 与此类似，如果 ATM 在如提取现金这样的用例结束时总会停止客户交易，则不管事件进程如何，将这种情况记录为用例后置条件。

其它Use Case属性

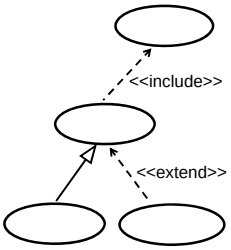
- 非功能需求
  - 有关本use-case的URPS+
- 业务规则
  - 在执行事件流时，使用到的重要业务规则或计算公式
- 扩展点
  - use-case可以通过另一use-case进行扩展
- 关系
  - 和actors及use-case的关联
- Use-case 图
  - 涉及本use-case的关系的可视化模型
- Other diagrams or enclosures
  - 交互、活动或其它图
  - 用户界面框图

面向对象分析的步骤

1. 用例建模
   - 1.1识别actor和use case，画Use-Case图
   - 1.2 编写Use-Case Spec.
   - 1.3 优化Use-Case图的结构
2. 建立概念模型
3. 用例分析

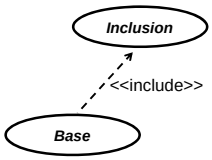
Use Case间的关系

- Include 包含
- Extend 扩展
- Generalization 泛化

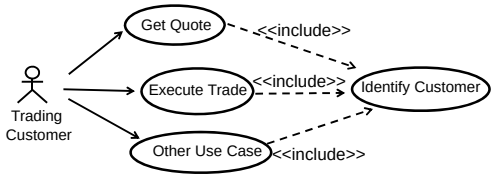


什么是Include 关系?

- 基本用例包含某个包含用例
- 包含用例定义的行为将显式插入基本用例



Include 关系示例

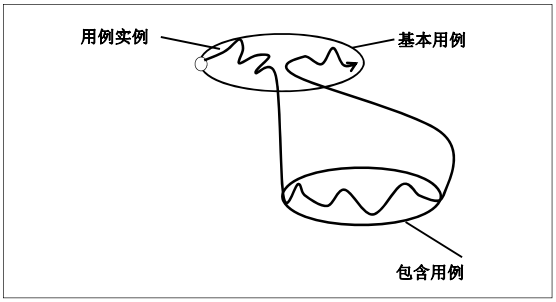


Get Quote Use Case
1. Include “Identify Customer” to verify customer’s identity
2. Display options. Customer selects “Get Quote”
3. ...

Identify Customer Use Case
1. Log on
2. Validate logon
3. Enter password
4. Verify password
A1: Not valid logon
A2: Wrong password
A3: ...

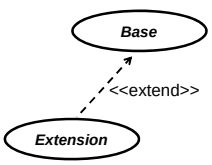
执行Include

- 到达插入点时执行包含用例

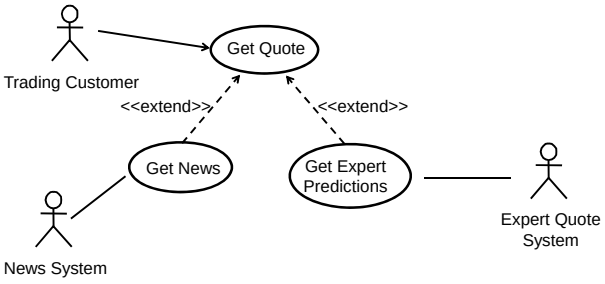


什么是Extend关系?

- 从基本用例到扩展用例的联接
  - 将扩展用例的行为插入基本用例
  - 只有当扩展条件为真是才进行插入
  - 在一个命名的扩展点插入基本用例



Extend 关系示例



Extend 关系示例(续)

Get Quote Use Case
Basic Flow:
1. **Include "Identify Customer" to verify customer's identity.**
2. **Display options.**
3. **Customer selects "Get Quote."**
4. **Customer gets quote.**
5. **Customer gets other quotes.**
6. **Customer logs off.**

A1. Quote System unavailable
…

Extension Points:
The "Optional Services" extension point occurs at Step 3 in the Basic Flow.

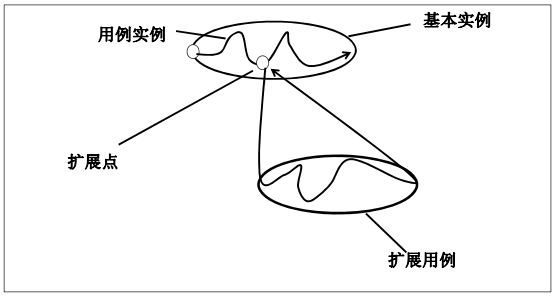
Get News Use Case
This use case extends the Get Quote Use Case, at extension point "Optional Services."

Basic Flow:
1. **If Customer selects "Get News," the system asks customer for time period and number of news items.**
2. **Customer enters time period and number of items. The system sends stock trading symbol to News System, receives reply, and displays the news to the customer.**
3. **The Get Quote Use Case continues.**

A1: News System Unavailable
A2: No News About This Stock
…

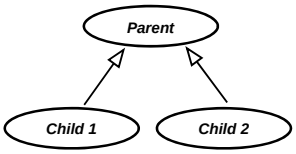
执行Extend

- 当到达扩展点并且扩展条件为真时执行

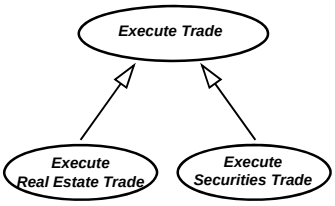


什么是 Use-Case Generalization?

- 从子用例到父用例的关系
  - 在父用例中描述通用的共享的行为
  - 在子用例中描述特殊的行为
  - 共享共同的目标



Generalization示例



Generalization 关系示例

Execute Trade Use Case
Basic Flow:
1. Customer Logs On
   Include "Identify Customer" to verify customer identity …
2. Customer Selects "Trade."
   Customer chooses trade …
3. Customer Selects Account
   Customer selects account.
   System displays account …
4. *Perform Trade*
5. Customer Begins New Trade
   If customer wants other trades, repeat from Step 3 …
6. Display Summary
   System displays summary …

A1. Account Not Operational…

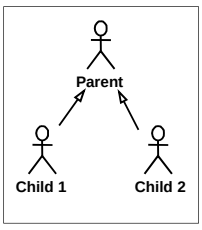
Execute Securities Trade Use Case
This use case is a child of the Execute Trade Use Case.

Basic Flow:
1. Customer Logs On
2. Customer Selects "Trade"
3. Customer Selects Account
4. Perform Trade
   If customer selects trade order type. The system performs…Limit Sell, Limit Buy, ... The system sends…
5. Customer Begins New Trade
6. Display Summary

A1: Limit Sell Order …

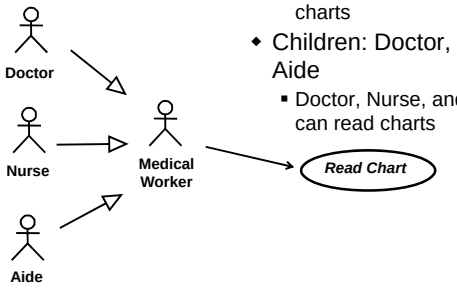
什么是Actor Generalization?

- Actors 可能有公共的属性
- 多个actors 和Use-Case交互时可能有公共的角色或目的



Actor Generalization示例

- Parent: Medical Worker
  - Medical Worker can read charts
- Children: Doctor, Nurse, Aide
  - Doctor, Nurse, and Aide can read charts



面向对象分析

- 面向对象方法概述
- 面向对象的基本概念
- 用例建模
- 建立概念模型
- 用例分析

参考附录：UML到C++的映射
UML到Java的映射
UML到VB的映射

面向对象分析的步骤

1. 用例建模
2. 建立概念模型
   - 2.1 识别Conceptual Class
   - 2.2 建立Conceptual Class之间的关系
   - 2.3 增加 Conceptual Class的属性
3. 用例分析

概念类图的作用

- When modeling the static view of a system, class diagrams are typically used in one of three ways, to model:
  - The vocabulary of a system
  - Collaborations
  - A logical database schema

面向对象分析的步骤

1. 用例建模
2. 建立概念模型
   - 2.1 识别Conceptual Class
   - 2.2 建立Conceptual Class之间的关系
   - 2.3 增加 Conceptual Class的属性
3. 用例分析

Identify Conceptual Class

- Strategies to Identify Conceptual Classes
  - Use a conceptual class category list
  - Finding conceptual classes with Noun Phrase
  - Use analysis patterns, which are existing conceptual models created by experts
    - using published resources such as Analysis Patterns [Fowler96] and Data Model Patterns [Hay96].

Use a conceptual class category list

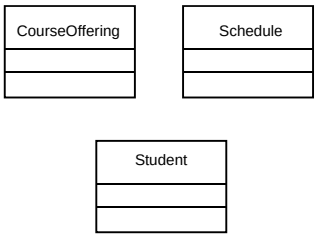
Category	Conceptual Classes
physical or tangible objects	Student Professor FulltimeStudent ParttimeStudent
form or abstract noun concepts	Course Course Offering Schedule
organizations	Dept

Finding conceptual classes with Noun Phrase

- Use use-case flow of events as input
- Underline noun clauses in the use-case flow of events
- Remove redundant candidates
- Remove vague candidates
- Remove actors (out of scope)
- Remove implementation constructs
- Remove attributes (save for later)
- Remove operations

Example: Candidate Conceptual Classes

- Register for Courses (Create Schedule)



A Common Mistake in Identifying Conceptual Classes

- Perhaps the most common mistake when creating a domain model is to represent something as an attribute when it should have been a concept.
- A rule of thumb to help prevent this mistake:
  - If we do not think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, not an attribute.
- Example
  - Should *store be an attribute of Sale, or a* separate conceptual class *Store?*
  - In the real world, a store is not considered a number or text - the term suggests a legal entity, an organization, and something occupies space.
  - Therefore, *Store should be a concept.*

面向对象分析的步骤

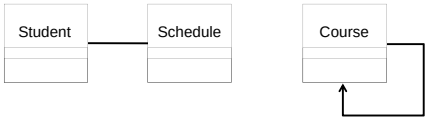
1. 用例建模
2. 建立概念模型
   - 2.1 识别Conceptual Class
   - 2.2 建立Conceptual Class之间的关系
   - 2.3 增加 Conceptual Class的属性
3. 用例分析

Class relationships

- Association
  - Aggregation
  - Composition
- Generalization
- Dependency

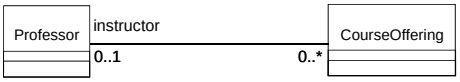
Relationships: Association

- The semantic relationship between two or more classifiers that specifies connections among their instances.
- A structural relationship specifying that objects of one thing are connected to objects of another thing.



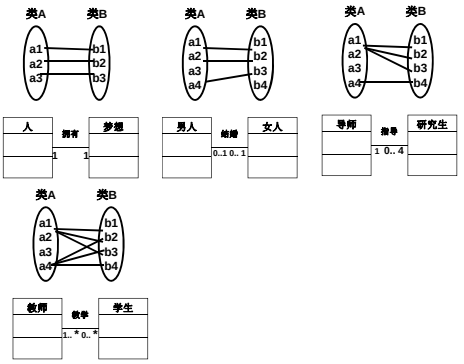
What Is Multiplicity?

- Multiplicity is the number of instances one class relates to ONE instance of another class.
- For each association, there are two multiplicity decisions to make, one for each end of the association.
  - For each instance of Professor, many Course Offerings may be taught.
  - For each instance of Course Offering, there may be either one or zero Professor as the instructor.

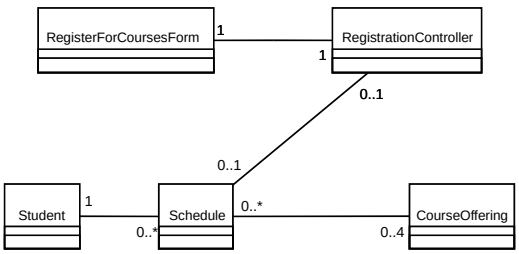


Multiplicity Indicators

Unspecified	
Exactly One	1
Zero or More	0..*
Zero or More	*
One or More	1..*
Zero or One (optional value)	0..1
Specified Range	2..4
Multiple, Disjoint Ranges	2, 4..6

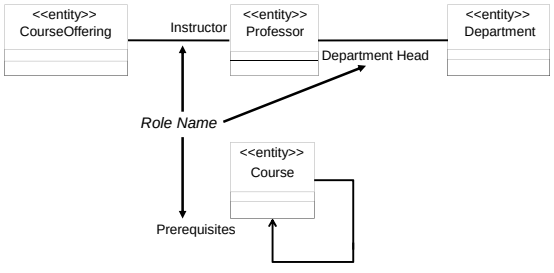


Example: Multiplicity



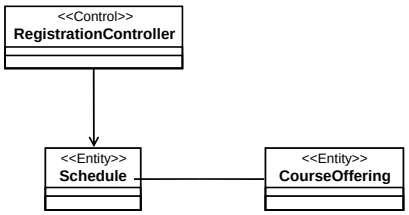
What Are Roles?

- The “face” that a class plays in the association

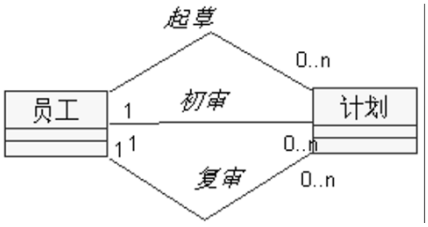


What Is Navigability?

- Indicates that it is possible to navigate from a associating class to the target class using the association

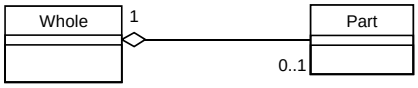


两个类之间关联同时可以有多条

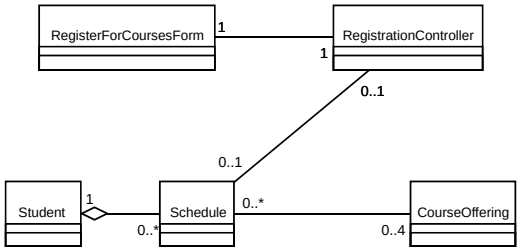


What Is an Aggregation?

- A special form of association that models a whole-part relationship between the aggregate (the whole) and its parts.
  - An aggregation is an “is a part-of” relationship.
- Multiplicity is represented like other associations.

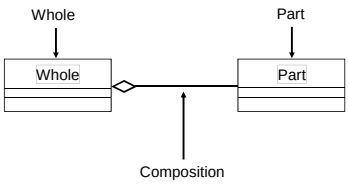


Example: Aggregation

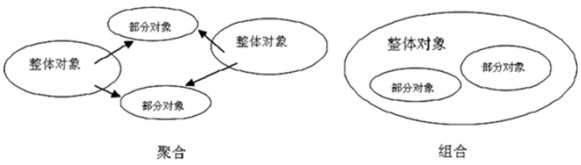


What Is Composition?

- A form of aggregation with strong ownership and coincident lifetimes
  - The parts cannot survive the whole/aggregate



Aggregation Vs. Composition

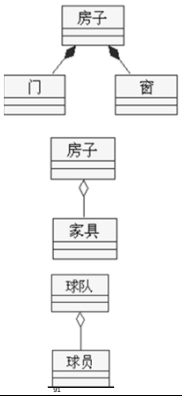


例：公司和雇员

例：订单和订单项

举例

- 组合/部分
- 容器/内容
- 集合/成员



练习

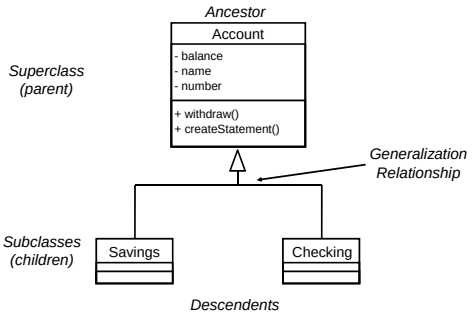
- 汽车和轮胎是关联、聚合、还是组合？
- 丈夫和妻子是关联、聚合、还是组合？

Relationships: Generalization

- A relationship among classes where one class shares the structure and/or behavior of one or more classes.
- Defines a hierarchy of abstractions where a subclass inherits from one or more superclasses.
  - Single inheritance
  - Multiple inheritance
- Is an “is a kind of” relationship.

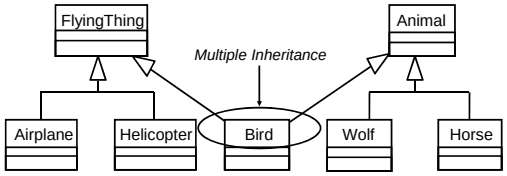
Example: Single Inheritance

- One class inherits from another.



Example: Multiple Inheritance

- A class can inherit from several other classes.



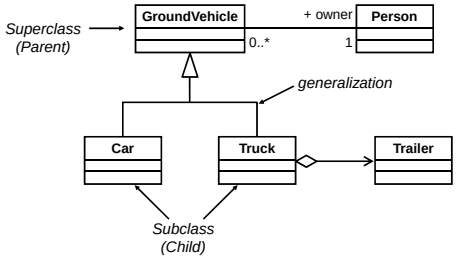
Use multiple inheritance only when needed and always with caution!

What Is Inherited?

- A subclass inherits its parent’s attributes, operations, and relationships.
- A subclass may:
  - Add additional attributes, operations, relationships.
  - Redefine inherited operations. (Use caution!)
- Common attributes, operations, and/or relationships are shown at the highest applicable level in the hierarchy.

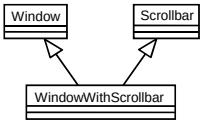
  Inheritance leverages the similarities among classes.

Example: What Gets Inherited



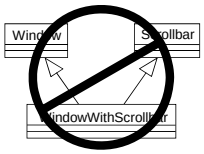
Generalization vs. Aggregation

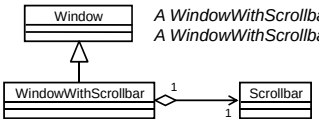
- Generalization and aggregation are often confused
  - Generalization represents an “is a” or “kind-of” relationship
  - Aggregation represents a “part-of” relationship



Is this correct?

Generalization vs. Aggregation





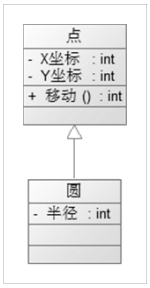
A WindowWithScrollbar “is a” Window
A WindowWithScrollbar “contains a” Scrollbar

Liskov替换原则——LSP

- Liskov Substitution Principle
  子类型应该能替换基类型

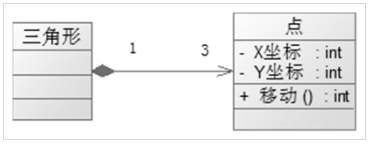
----Barbara Liskov

对吗?

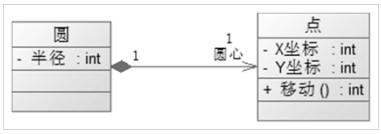


圆是一种有半径的点

子类能替换超类吗？



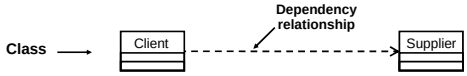
三个圆能组成三角形吗？



关联更能反映领域内涵

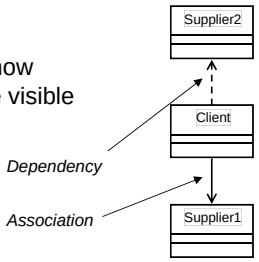
Relationships: Dependency

- A relationship between two classes where a change in one may cause a change in the other
- Non-structural, “using” relationship



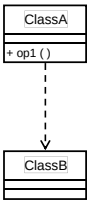
Dependencies vs. Associations

- Associations are structural relationships
- Dependencies are non-structural relationships
- In order for objects to “know each other” they must be visible
  - Local variable reference
  - Parameter reference
  - Global reference
  - Field reference



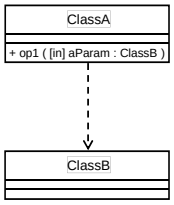
Local Variable Visibility

- The op1() operation contains a local variable of type ClassB



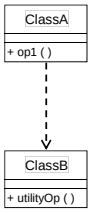
Parameter Visibility

- The ClassB instance is passed to the ClassA instance

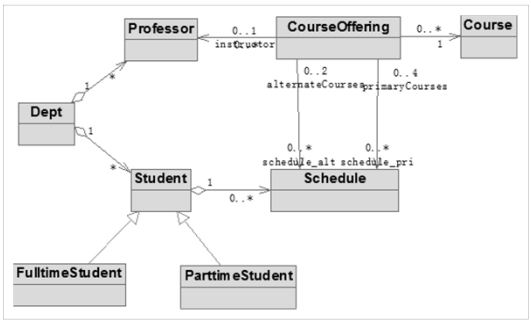


Global Visibility

- The ClassUtility instance is visible because it is global



Example: Conceptual Model



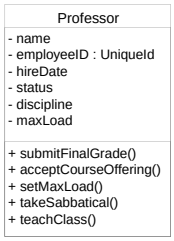
讨论

- “女儿”和“父亲”是什么关系？
- “青蛙”和“蝌蚪”是什么关系？（ Metamorphosis ）

面向对象分析的步骤

1. 用例建模
2. 建立概念模型
   - 2.1 识别Conceptual Class
   - 2.2 建立Conceptual Class之间的关系
   - 2.3 增加 Conceptual Class的属性
3. 用例分析

增加 Conceptual Class的属性



识别出概念类的主要属性，可以有遗漏，
在后续的分析与设计中，这些属性会逐渐补全。
同时可以识别出概念类的部分操作。
在概念模型中，类的属性比操作更为重要。

面向对象分析

- 面向对象方法概述
- 面向对象的基本概念
- 用例建模
- 建立概念模型
- 用例分析

哪些类相互合作实现用例模型中每个用例？

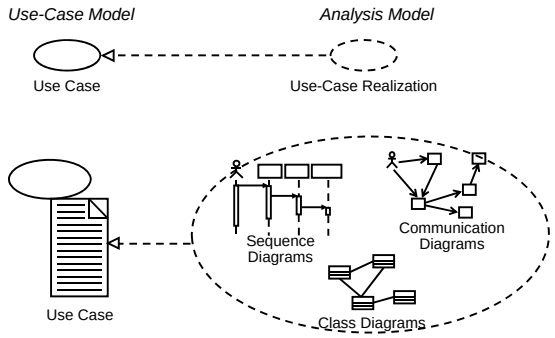
面向对象分析的步骤

1. 用例建模
2. 建立概念模型
3. 用例分析
   - 3.1 识别出用例实现
   - 3.2 针对每个用例实现：
     - －识别出分析类
     - －建立时序图，生成通信图
     - －对照通信图建立类图，完善每个分析类的属性和操作

面向对象分析的步骤

1. 用例建模
2. 建立概念模型
3. 用例分析
   - 3.1 识别出用例实现
   - 3.2 针对每个用例实现：
     - －识别出分析类
     - －建立时序图，生成通信图
     - －对照通信图建立类图，完善每个分析类的属性和操作

What is a Use-Case Realization?



面向对象分析的步骤

1. 用例建模
2. 建立概念模型
3. 用例分析

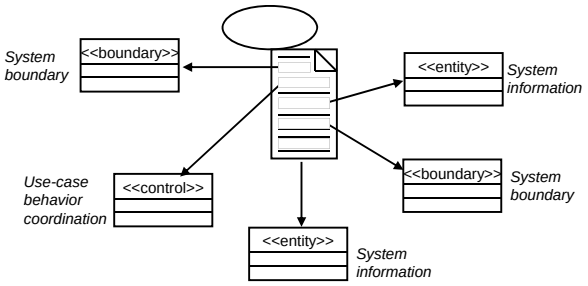
3.1 识别出用例实现
3.2 针对每个用例实现：
- 识别出分析类
- 建立时序图，生成通信图
- 对照通信图建立类图，完善每个分析类的属性和操作

三种Analysis Class

- 不同的衍型(stereotype)可采用不同的图标，也可用采用“《》”来区别。

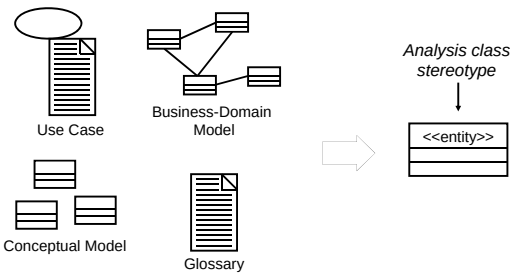
Find Analysis Classes

- The complete behavior of a use case has to be distributed to analysis classes



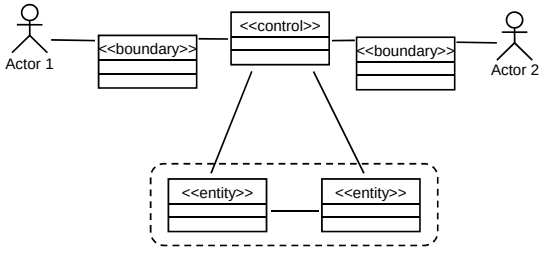
What Is an Entity Class?

- Key abstractions of the system



Environment independent.

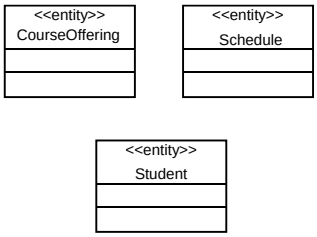
The Role of an Entity Class



Store and manage information in the system.

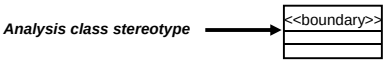
Example: Candidate Entity Classes

- Register for Courses (Create Schedule)



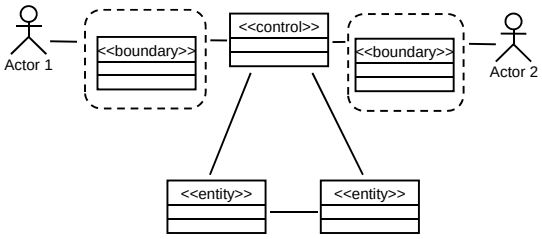
What Is a Boundary Class?

- Intermediates between the interface and something outside the system
- Several Types
  - User interface classes
  - System interface classes
  - Device interface classes
- *One boundary class per actor/use-case pair*



Environment dependent.

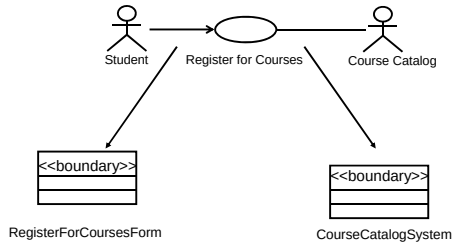
The Role of a Boundary Class



Model interaction between the system and its environment.

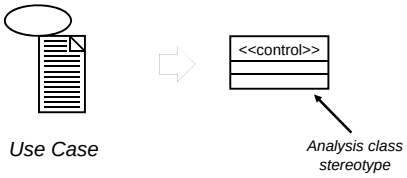
Example: Finding Boundary Classes

- One boundary class per actor/use case pair



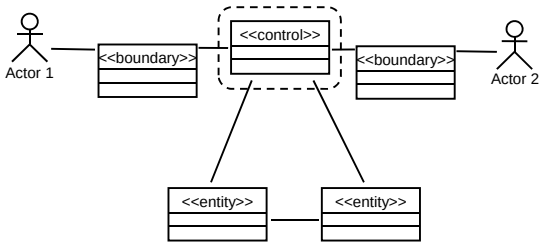
What Is a Control Class?

- Use-case behavior coordinator
  - More complex use cases generally require one or more control cases



Use-case dependent. Environment independent.

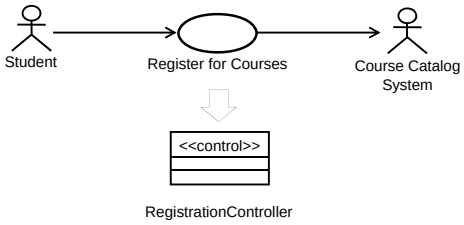
The Role of a Control Class



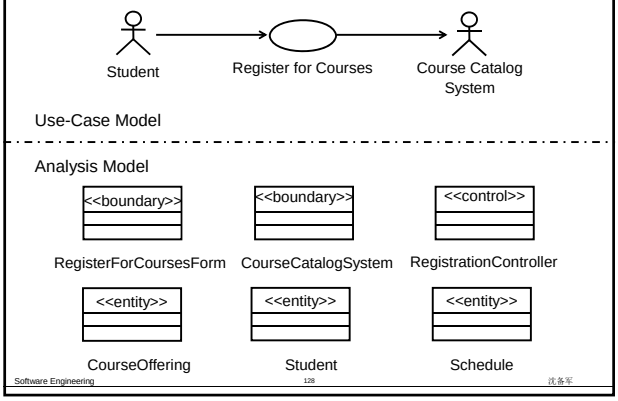
Coordinate the use-case behavior.

Example: Finding Control Classes

- In general, identify one control class per use case.
  - As analysis continues, a complex use case's control class may evolve into more than one class



Example: Summary: Analysis Classes



面向对象分析的步骤

1. 用例建模
2. 建立概念模型
3. 用例分析

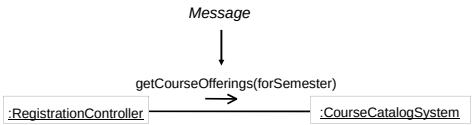
3.1 识别出用例实现
3.2 针对每个用例实现：
　　－识别出分析类
　　－建立时序图，生成通信图
　　－对照通信图建立类图，完善每个分析类的属性和操作

Objects Need to Collaborate

- Objects are useless unless they can collaborate to solve a problem.
  - Each object is responsible for its own behavior and status.
  - No one object can carry out every responsibility on its own.
- How do objects interact with each other?
  - They interact through messages.

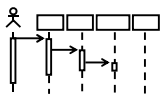
Objects Interact with Messages

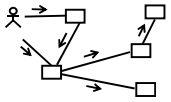
- A message shows how one object asks another object to perform some activity.



What is an Interaction Diagram?

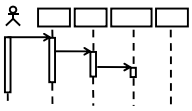
- Generic term that applies to several diagrams that emphasize object interactions
  - Sequence Diagram
    - Time oriented view of object interaction
  - Communication Diagram
    - Structural view of messaging objects


Sequence Diagrams


Communication Diagrams

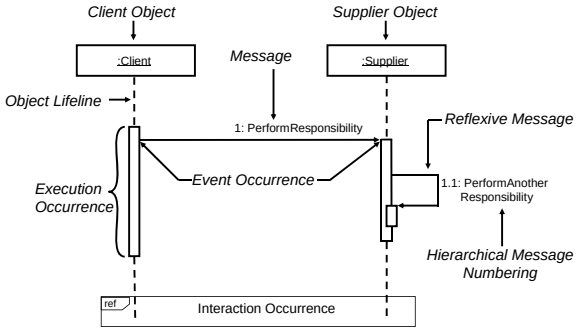
Create Sequence Diagrams

- A sequence diagram is an interaction diagram that emphasizes the time ordering of messages.
- The diagram shows:
  - The objects participating in the interaction.
  - The sequence of messages exchanged.



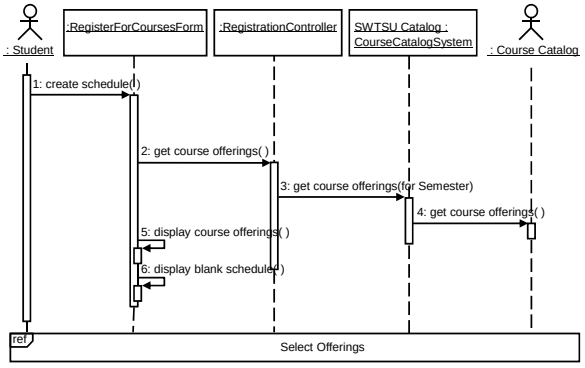
Sequence Diagram

The Anatomy of Sequence Diagrams



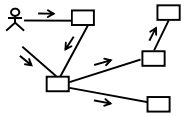
Example: Sequence Diagram

先分工，再协作！



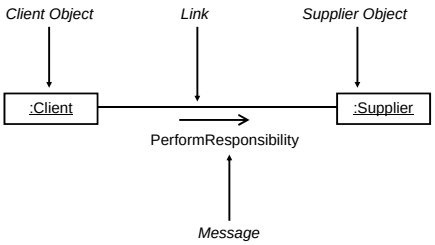
Generate Communication Diagrams

- A communication diagram emphasizes the organization of the objects that participate in an interaction.
- The communication diagram shows:
  - The objects participating in the interaction.
  - Links between the objects.
  - Messages passed between the objects.

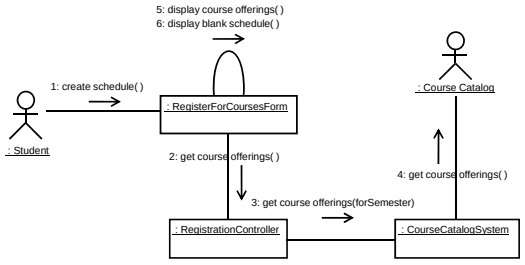


Communication Diagrams

The Anatomy of Communication Diagrams



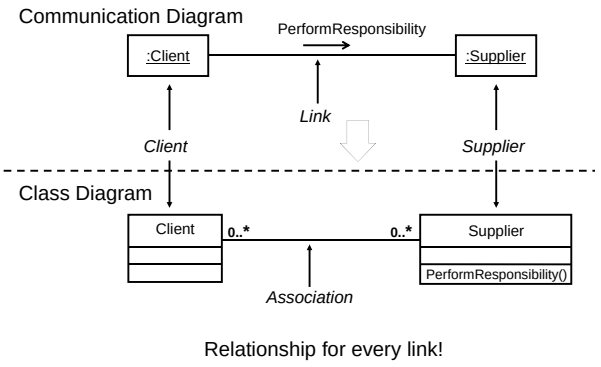
Example: Communication Diagram



面向对象分析的步骤

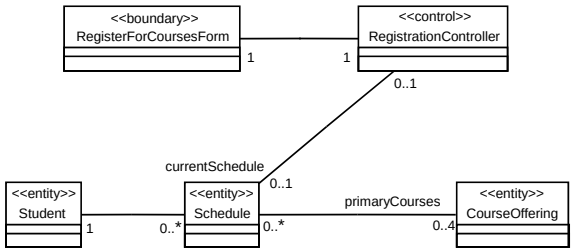
1. 用例建模
2. 建立概念模型
3. 用例分析
   - 3.1 识别出用例实现
   - 3.2 针对每个用例实现：
     - －识别出分析类
     - －建立时序图，生成通信图
     - －对照通信图建立类图，完善每个分析类的属性和操作

Finding Relationships and Operations

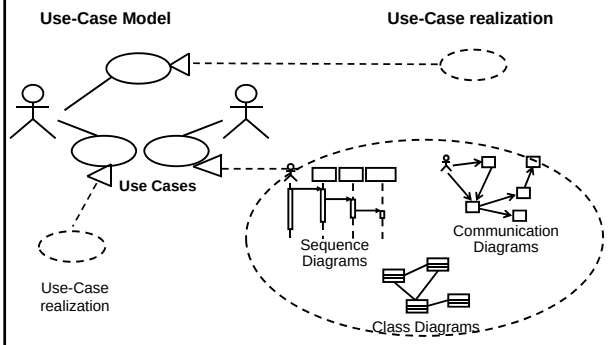


Relationship for every link!

Example: VOPC: Finding Relationships



完成用例分析



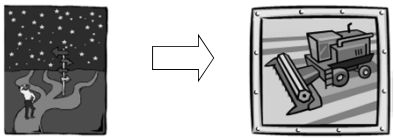
Shanghai Jiao Tong University

软件工程
Module: 设计工程

上海交通大学软件工程中心

Change our viewpoint

Do the right things ⇨ Do the things right



What How

软件需求工程解决“做什么”的问题
软件设计工程解决“怎么做”的问题

设计工程

- 软件架构设计
- 软件设计的原则
- 软件设计的复用
- 软件设计的质量

@第6章.教材

设计的步骤

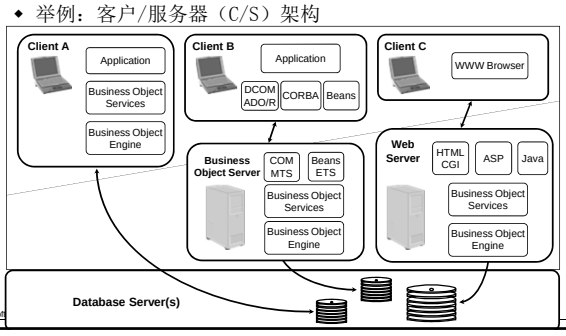
- 架构设计
  - 又称概要设计，它定义了软件的全貌(即蓝图)，记录了最重要的设计决策，并成为随后的设计与实现工作的战略指导原则。
- 详细设计
  - 又称构件级设计，它在软件架构的基础上定义各模块的内部细节，例如内部的数据结构、算法和控制流等，其所做的设计决策常常只影响单个模块的实现。

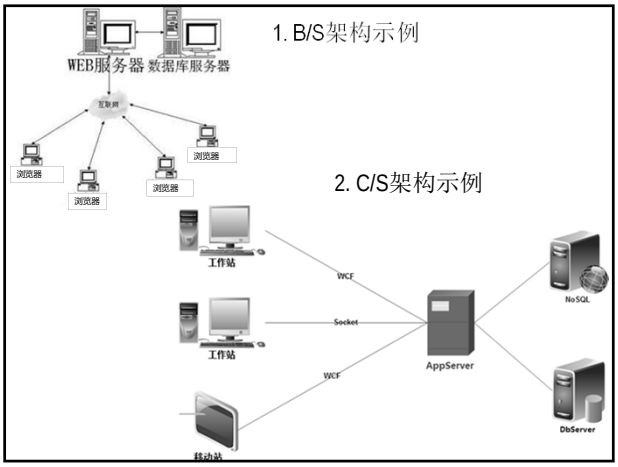
架构设计

- 架构设计(architecture design)的内容：
  - 定义软件的物理架构
  - 定义软件的逻辑架构
  - 定义软件的数据架构
  - 选择软件质量属性的设计策略
- 目标：使得软件系统在架构层面的设计上满足拟建系统功能性和非功能性需求。

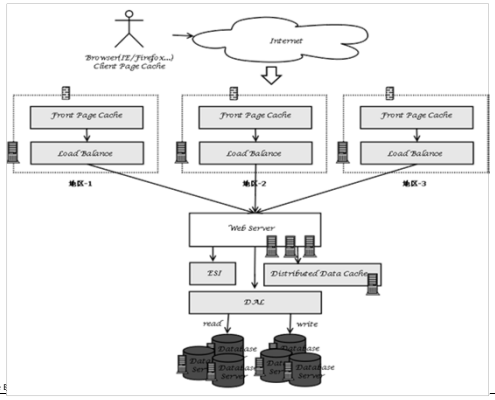
定义软件的物理架构，即架构的部署视图

- 针对分布式系统，需要定义物理架构,刻画计算机或处理节点以及网络间的关系。
- 重点考虑可靠性和性能等非功能需求的支持。
- 举例：客户/服务器（C/S）架构

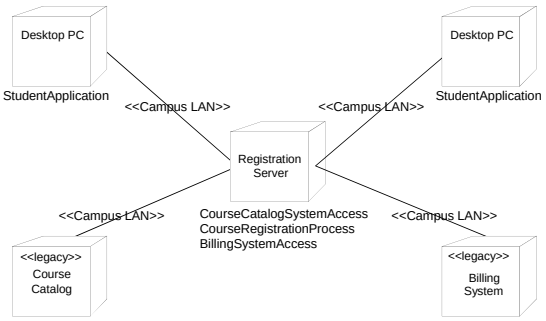




举例：大型电商网站的物理架构



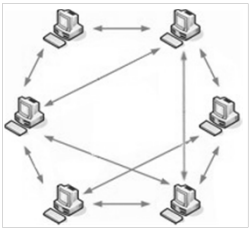
UML的物理架构的描述



选课系统的C/S物理架构

P2P物理架构

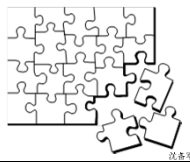
- 所有计算机节点都是对等的



- 讨论：网上四人飞行棋软件的物理架构？

定义软件的逻辑架构，即架构的逻辑视图

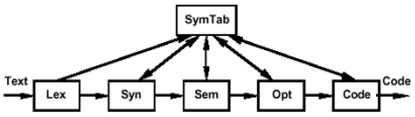
- 把软件划分为多个模块（又称构件），模块和模块相互协作，共同完成软件的需求规约。这些模块将部署在物理架构的同一节点或不同节点上。
- 重点考虑功能需求的支持。
- 所有软件都需要定义逻辑架构



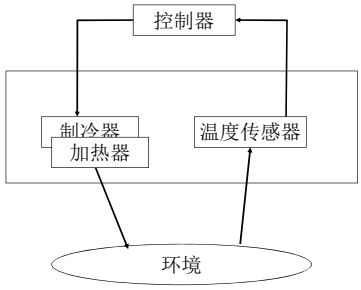
举例：编译器的逻辑架构图

(1) 传统的编译器的架构图

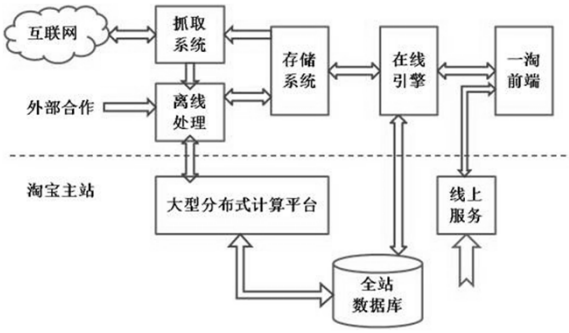
(2) 具有共享符号表的编译器的架构图



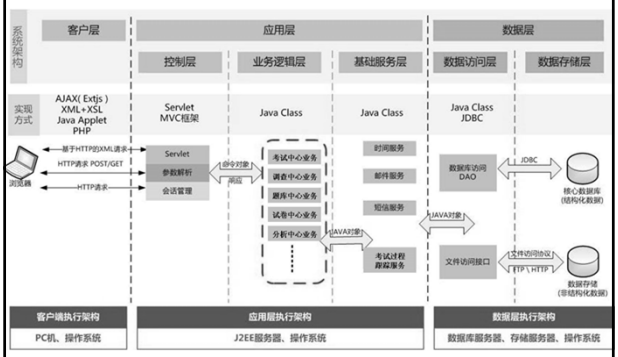
举例: 空调控制软件的逻辑架构图



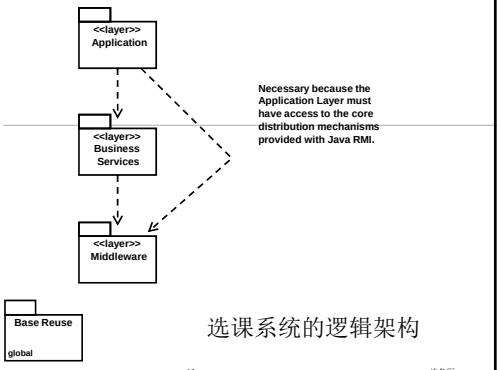
举例：一淘网（www.etao.com）的逻辑架构图



举例：基于Web的在线考试系统的架构图



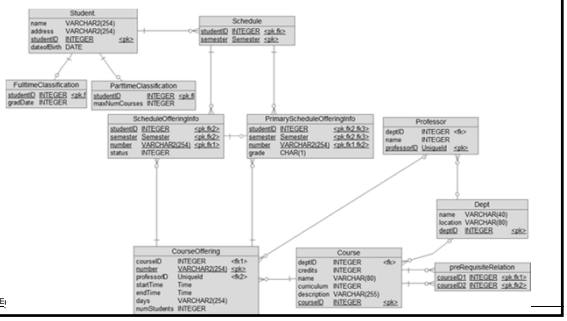
UML的逻辑架构的描述



选课系统的逻辑架构

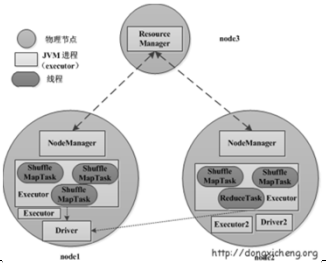
定义软件的数据架构，即架构的数据视图

- 针对以数据为中心的软件，则需定义数据架构
- 举例：选课系统的数据视图。



其他架构视图？

- 针对多进程多线程软件，需要定义进程视图
- 针对高安全性软件，需要定义安全视图
- 针对高易用性软件，需要定义界面视图
- 针对网站，需要定义页面导航视图
- ……



质量因素的架构设计战术（1）

质量因素	设计战术
易用性	1.为用户提供适当的反馈和协助 2.将用户接口与应用的其余部分分离 3.提供“取消”、“撤消”等命令 4.建立用户模型、任务模型和系统模型
可用性 即可靠性	1.错误检测，如心跳、异常、命令/响应 2.错误恢复，如表决、冗余、备件、检查点/回滚 3.错误预防，如事务、进程监视、从服务中删除
性能	1.资源需求，如提高计算效率、减少计算开销、控制采样频率、限制队列大小 2.资源管理，如引入并发、增加可用资源、维持数据或计算的多个副本 3.资源仲裁，如先进先出、优先级调度

质量因素的架构设计战术（2）

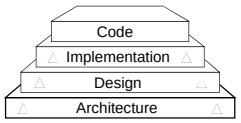
质量因素	设计战术
安全性	1.抵抗和检测攻击，如用户身份验证、用户授权、限制暴露的信息、防火墙 2.从攻击中恢复，如维持审计追踪、采用可用性中恢复战术
可测试性	1.测试的输入/输出，如记录/回放、将接口与实现分离、特化访问路线/接口 2.内部监视，如内置监视器
可维护性	1.局部化修改，如泛化模块、预期期望的变更、限制可能的选择 2.防止连锁反应，如信息隐藏、维持现有接口、限制通信路径、仲裁者的使用 3.推迟绑定时间

Characteristics of a Good Architecture

- Resilient
- Simple
- Approachable
- Clear separation of concerns
- Balanced distribution of responsibilities
- Balances economic and technology constraints

Architecture Constrains Design and Implementation

- Architecture involves a set of strategic design decisions, rules or patterns that constrain design and construction.

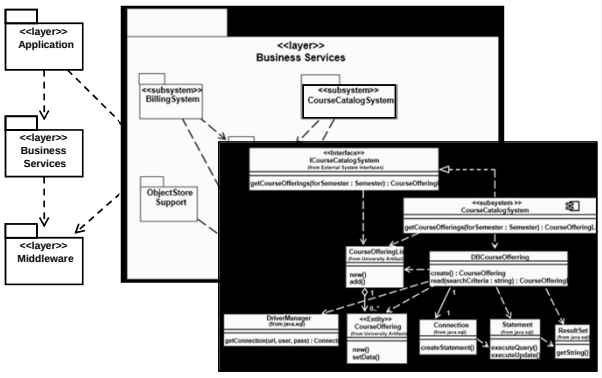


Architecture decisions are the most fundamental decisions, and changing them will have significant effects.

详细设计

- 详细设计软件架构中的每个模块：
  - 将模块进一步细分：子模块－子子模块－***
    - 原子模块为类、函数、过程等
  - 为所有数据对象定义详细的数据结构
  - 为所有在模块内发生的处理定义算法细节、控制流和数据流

选课系统的详细设计



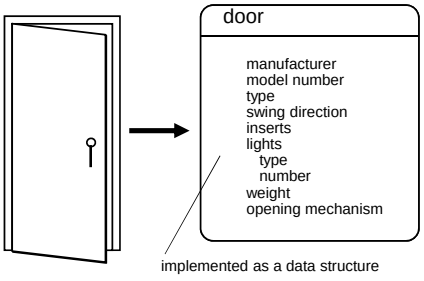
设计工程

- 软件架构设计
- 软件设计的原则 — 抽象和分解 — 模块化设计
- 软件设计的质量
- 软件设计的复用

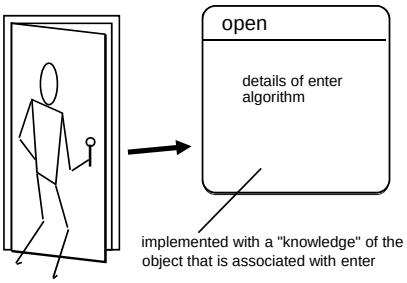
抽象 abstraction

- 抽象，是在软件规模逐渐增大、软件复杂性逐渐增大下，控制复杂性的基本策略。软件开发过程就是对软件抽象层次的一次次细化的过程：需求、架构、设计、编码。
- 在软件设计中，主要抽象手段包括：
  - 数据抽象把一个数据对象的定义抽象为一个数据类型名，用此类型名可定义多个具有相同性质的数据对象
  - 过程抽象把完成一个特定功能的动作序列抽象为一个过程名和参数表，以后通过指定过程名和实际参数调用此过程
  - 对象抽象则通过操作和属性，组合了这两种抽象，即在抽象数据类型的定义中加入一组操作的定义，以确定在此类数据对象上可以进行的操作。

数据抽象Data Abstraction



过程抽象Procedural Abstraction



分解和模块化

- 分解（decomposition）是控制复杂性的另一种有效方法，软件设计用分解来实现模块化设计。
- 模块化（modularity），即将一个复杂的系统自顶向下地分解成若干模块（module），每个模块完成一个特性，所有的模块组装起来，成为一个整体，完成整个系统所要求的特性。

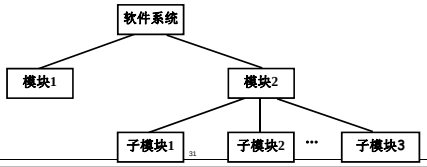
模块化是产业规模化的基础

软件模块化的好处

- 把复杂软件变简单，更易实现
- 把易变的部分和稳定的部分分开，模块可插拔可替换，应对需求变更和技术升级
  - 在计算机中，CPU是常变的模块，键盘是相对稳定的模块
- 支持多人协同开发
- 支持迭代式开发
- 模块复用和模块组装，加快开发时间，节省开发费用，提高产品质量
- 促进形成大规模开发的软件产业－软件工厂

软件模块

- 模块是能够单独命名并独立地完成一定功能的程序语句的集合
  - 例如，子系统、过程、函数、子程序、宏、类等
- 模块具有两个基本的特征：
  - 外部特征是指模块跟外部环境联系的接口和模块的功能；
  - 内部特征是指模块的内部环境具有的特点，即该模块的局部数据和处理逻辑。
- 模块可继续分解为子模块、子子模块……



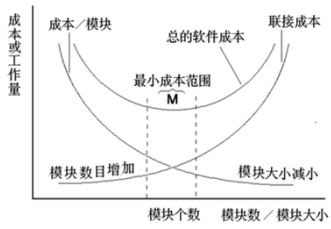
分解 decomposition

$$C(P1+P2)>C(P1)+C(P2)$$
$$E(P1+P2)>E(P1)+E(P2)$$

- C为问题的复杂度，E为解题需要的工作量
- 思考：
  - 如果我们无限制地划分软件，开发它所需的工作量会变得小到可以忽略？！
  - 事实上，影响软件开发的工作量的因素还有很多，例如模块接口费用等等
  - 上述不等式只能说明，当模块的总数增加时，单独开发各个子模块的工作量之和会有所减少

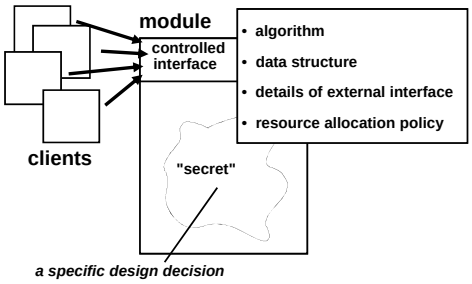
模块化的成本

- 如果模块是相互独立的，当模块变得越小，每个模块花费的工作量越低；
- 但当模块数增加时，模块间的联系也随之增加，把这些模块联接起来的工作量也随之增加。



信息隐藏 Information Hiding

- 每个模块的实现细节对于其它模块来说应该是隐蔽的



模块独立

- 模块独立的定义：
  - 模块完成独立的功能并且与其他模块的接口简单
- 模块独立的重要性：
  - 功能被划分，并且接口被简化，所以具有有效模块化的软件更易于开发。当多人分工合作开发同一个软件时，这个优点尤其重要。
  - 由于因设计和编码修改引起的副作用受到局限，错误传播被减小，并且模块复用成为可能，所以独立的模块更易于维护和测试 。

模块独立的衡量指标

- 模块独立可以由两项指标来衡量：
  - 内聚（cohesion），是一个模块内部各个元素彼此结合的紧密程度的度量
  - 耦合（coupling），是模块之间的相对独立性（互相连接的紧密程度）的度量
- 模块独立追求高内聚和低耦合。

内聚

- 一般模块的内聚性分为七种类型：
  1. 偶然内聚 coincidental cohesion
  2. 逻辑内聚 logical cohesion
  3. 时间内聚 temporal cohesion
  4. 过程内聚 procedural cohesion
  5. 通信内聚 communicational cohesion
  6. 顺序内聚 sequential cohesion
  7. 功能内聚 functional cohesion

低 ↓ 高

- 确定内聚的精确级别是不必要的，重要的该是尽量争取高内聚和识别低内聚

低、中、高内聚

- 低内聚：
  - 有时在写完一个程序之后，发现一组语句在两处或多处出现，于是把这些语句作为一个模块，这样就出现了偶然内聚的模块。
  - 如果一个模块完成的任务在逻辑上属于相同或相似的一类（例如，一个模块产生各种类型的全部输出），则称为逻辑内聚。
  - 如果一个模块包含的任务必须在同一段时间内执行（例如，模块完成各种初始化工作），就叫时间内聚。
- 中内聚：
  - 如果一个模块内的处理元素是相关的，而且必须以特定次序执行，则称为过程内聚。
  - 如果模块中所有元素都使用同一个输入数据和（或）产生同一个输出数据，则称为通信内聚。
- 高内聚：
  - 如果一个模块内的处理元素和同一个功能密切相关，而且这些处理必须顺序执行，则称为顺序内聚。
  - 如果模块内所有处理元素属于一个整体，完成一个单一的功能，则称为功能内聚。功能内聚是最高程度的内聚。

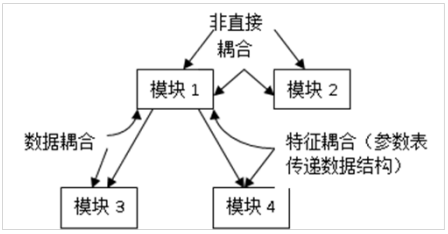
耦合

- 一般模块之间可能的耦合方式有七种类型
  1. 非直接耦合 no direct coupling
  2. 数据耦合 data coupling
  3. 特征耦合 stamp coupling
  4. 控制耦合 control coupling
  5. 外部耦合 external coupling
  6. 公共耦合 common coupling
  7. 内容耦合 content coupling

低 ↓ 高

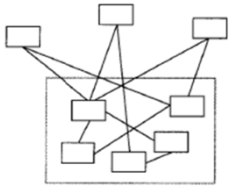
- 确定耦合的精确级别是不必要的，重要的该是尽量争取低耦合和识别高耦合

弱耦合示例

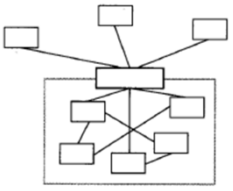


- 模块1与模块2为同级模块，相互之间没有信息传递，属于非直接耦合。
- 模块1调用模块3，交换的是简单变量，便构成数据耦合；
- 模块1调用模块4，交换的是数据结构，则构成特征耦合。

哪个模块化设计好？



设计A



设计B

设计工程

- 软件架构设计
- 软件设计的原则
- 软件设计的复用
- 软件设计的质量

设计重用 Design Reuse

- Pattern
  - Architectural style
  - Design pattern
  - Idiom
- Framework

什么是模式(Pattern)

- Christopher Alexander 说过：“每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复劳动”，1977年。
- 定义: “在一个上下文中对一种问题的解决方案”。

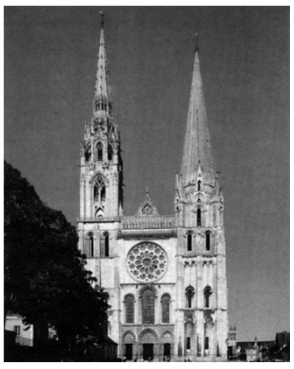
软件模式分类

- An architectural pattern (architectural style) expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.
- An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

架构风格

建筑风格——古希腊、古罗马建筑

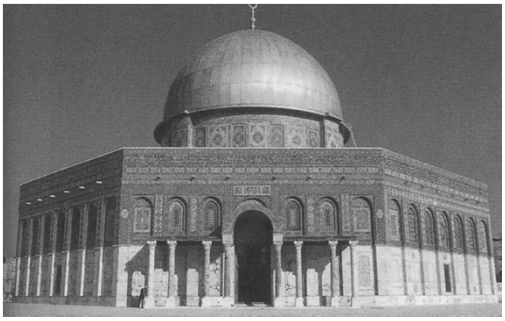
建筑风格——哥特建筑



法国夏特尔主教堂

建筑风格——中国建筑

建筑风格——伊斯兰的清真寺



耶路撒冷的圣石庙

常见的架构风格

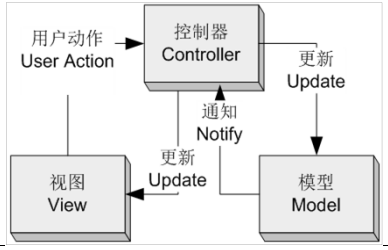
- From Mud to Structure: Organize components.
  - Layers: Organize components into layers where layer's services are only used by layer i+1.
  - Pipes and Filters: Divide the task into several sequential processing steps -- the output of task i is the input of task i+1.
  - Blackboard: Several independent programs work cooperatively on a common data structure.
- Distributed Systems :Handle distributed computation.
  - Broker: Introduce a *broker* component to achieve better decoupling of clients and servers -- brokers accept requests from clients and forward the requests to servers, then return the results back to the clients.
  - Client/Server
  - 3 Tiers

常见的架构风格（续）

- Interactive Systems: Keep a program's functional core independent of the user interface
  - Model - View - Controller: Divides the application into processing, output, and input. View and controller parts are usually observers of the model via the observer pattern
  - Presentation - Abstract - Control: Divides the application up to heirarchies or MVC-like components. Each component is dependent upon and provides functionality for the a higher-level component. There is only one top-level component
- Adaptable Systems : Design for change
  - Microkernel Encapsulate the fundamental services of the application
  - Reflection Divide the application into a meta-level and a base level to make the application "self-aware". The meta level encapsulates knowledge of the system; the base level encapsulates knowledge about the problem domain
- Others

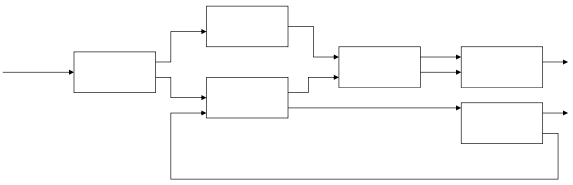
MVC

模型Model：管理系统中存储的数据和业务规则，并执行相应的计算功能。
视图View：根据模型生成提供给用户的交互界面，不同的视图可以对相同的数据产生不同的界面。
控制器Control：接收用户输入，通过调用模型获得响应，并通知视图进行用户界面的更新。



管道和过滤器（Pipes and Filters）风格

- In this style, each component has a set of inputs and a set of outputs.
- A component reads streams of data on its inputs and produces streams of data on its output.

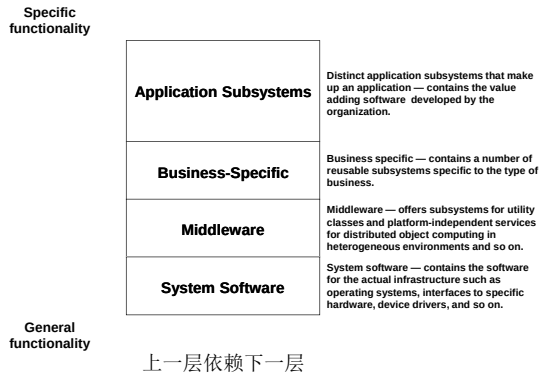


管道和过滤器架构举例

- Linux的Shell程序可以看做是典型的管道与过滤器架构的例子
- 例如下面的Shell脚本：
  - $cat *TestResults* | sort | grep *Good*

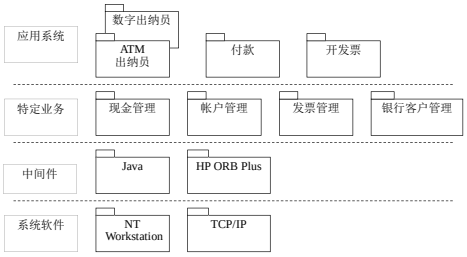
会将TestResults文件的文本进行排序，然后找出其中包含单词Good的行，并显出在屏幕上。Shell命令cat、sort和grep依次执行，就构成了一个管道-过滤器架构。

层次架构（layered architecture）风格

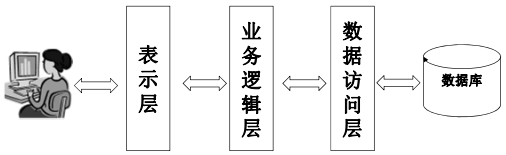


上一层依赖下一层

层次架构实例



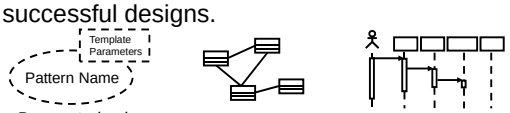
3 Tiers



- 表示层：负责向用户呈现界面，并接收用户请求发送给业务逻辑层；
- 业务逻辑层：负责执行业务逻辑以处理用户请求，并调用数据访问层提供的持久性操作；
- 数据访问层：负责执行数据库持久性操作。

设计模式 Design Patterns

- A design pattern is a solution to a common design problem.
  - Describes a common design problem
  - Describes the solution to the problem
  - Discusses the results and trade-offs of applying the pattern
- Design patterns provide the capability to reuse successful designs.



Parameterized Collaboration

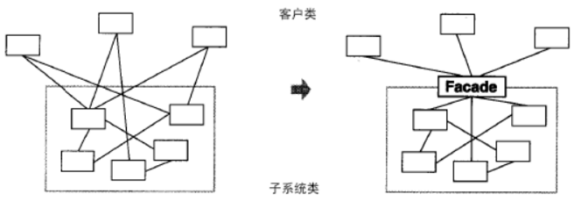
Structural Aspect

Behavioral Aspect

GOF 23个设计模式

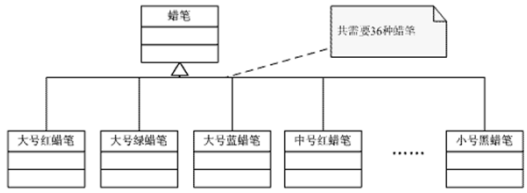
- 创建型模式
  - ABSTRACT FACTORY (抽象工厂）；BUILDER (生成器）；FACTORY METHOD (工厂方法）；PROTOTYPE (原型）；SINGLETON (单件）
- 结构型模式
  - ADAPTER（适配器）；BRIDGE（桥接）；COMPOSITE（组合）；DECORATOR（装饰）；FACADE（外观）；FLYWEIGHT（享元）；PROXY（代理）
- 行为模式
  - CHAIN OF RESPONSIBILITY (职责链)；COMMAND (命令）；INTERPRETER (解释器）；ITERATOR (迭代器）；MEDIATOR (中介者）；*MEMENTO (备忘录）；OBSERVER (观察者）；STATE (状态）；STRATERY (策略）；TEMPLATE METHOD (模板方法）；VISITOR (访问者）

举例：Facade模式



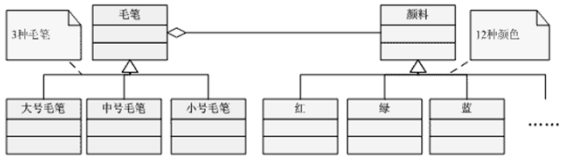
举例： Bridge模式

- 蜡笔和毛笔的故事
  - 我们需要用蜡笔或者毛笔绘制图画，画里有山水、花鸟、人物等；
  - 我们需要
    - 不同粗细的笔（填充颜色用细笔不方便，勾勒细节用粗笔无法完成）
    - 不同的色彩（彩色图画）
  - 36只蜡笔
    - 购置了粗、中、细3套蜡笔，一套12色，一共36只蜡笔



- 蜡笔和毛笔的故事（续）
  - 我们也需要36只毛笔吗？

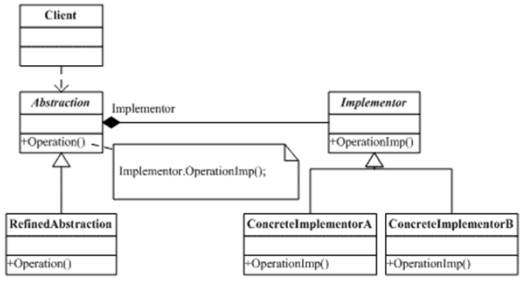
3只毛笔，12种颜料，不同的毛笔使用不同的颜料



- 毛笔和蜡笔什么区别？
  - 毛笔的方案显然更加优雅
    - 需要处理的对象数目减少（乘法变成了加法）
      - 蜡笔：3×12＝36
      - 毛笔：3＋12＝15
    - 复用
      - 12种颜料被3种毛笔复用
- 导致蜡笔模型与毛笔模型出现差异的原因是什么？
  - 笔和颜色是否能够分离
  - 更抽象些：抽象（abstraction）与实现（implementation）是否能够分离
    - 我们可以理解为：毛笔用颜料作画。
    - 颜料是直接使得画纸上某些局部出现色彩的东西，它是底层的实现。
    - 画笔是人们画画时操作的对象，它并不能直接在画纸上留下痕迹，它必须调用颜料的功能（染色功能）才能绘画；但是client（画家）确不必知道颜料是如何染色的这样的细节，他只需要关注如何操作画笔。画笔是高层的抽象。

Bridge模式

- 意图：
  - 将抽象与其实现解耦，使它们可以独立变化
- 图示：

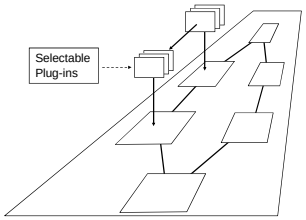


Design Pattern 的两本好书

- Erich Gamma, Richard Helm, Ralph Johns , John Vlissides, “Design patterns: elements of reusable object-oriented software”, Prentice-Hall, 1994.
- Eric Freeman, Elisabeth Freeman, et al., “Head First Design Patterns ”, O’Reilly Media Inc., 2005.

框架(Framework)

- 框架是一个代码骨架，可以使用为解决问题而设计的特定类或功能来填充这个代码骨架，使之丰满。



Frameworks with Plug-ins

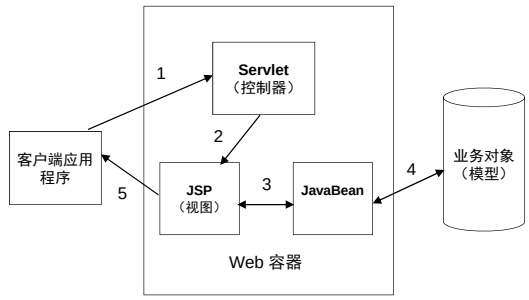
框架举例

1. MVC框架
   - J2EE
   - Struts
2. ORM 框架
   - EJB
   - Hibernate
3. 界面框架
   - Java界面框架：AWT, Swing, SWT, SwingWT, Java 2D, Java 3D，JSF
   - AJAX界面框架：DWR， DOJO
4. 复合框架
   - Spring包含Web MVC, IOC, ...
   - Java EE 包含JSF、EJB 、JAX-WS 、IOC, ...

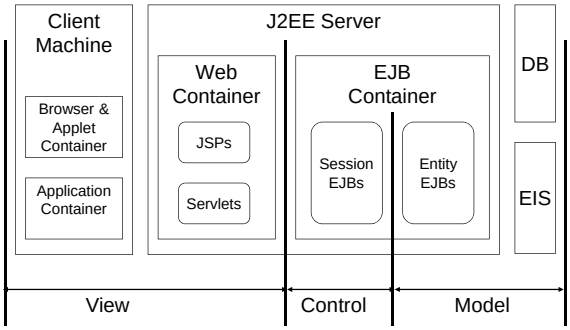
Web MVC框架 (Java)举例

- JSP Model II
- J2EE
- Struts
- JSF
- Spring Web MVC
- WebWork
- TapeStry

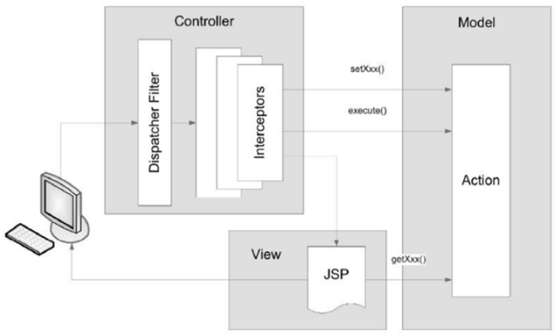
JSP Model II



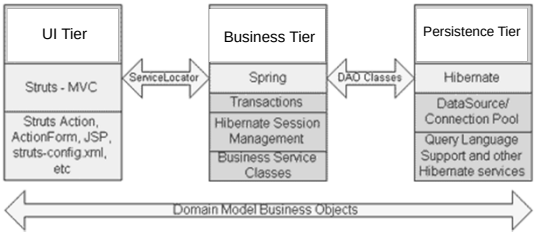
J2EE



Struts 2



SSH （Struts + Spring + Hibernate）



设计工程

- 软件架构设计
- 软件设计的原则
- 软件设计的复用
- 软件设计的质量

软件设计的质量要求

1) 设计应当模块化，高内聚、低耦合。
   - 支持多人合作开发
   - 易于测试和修改
   - 能够以演化过程实现
2) 设计应当包含数据、体系结构、接口和构件的清楚的表示。
3) 设计应根据软件需求采用可重复使用的方法进行。
4) 应使用能够有效传达其意义的表示法来表达设计模型。

7种软件设计的坏味道

1）僵化性（Rigidity）
- 很难对软件进行改动，因为每个改动都会迫使对系统其他部分的许多改动

2）脆弱性（Fragility）
- 对系统的改动会导致系统中和改动的地方在概念上无关的许多地方出现问题

3）牢固性（Immobility）
- 很难解开系统中某部分与其它部分之间的纠结，从而难以使其中的任何部分可以被分离出来被其它系统复用

4）粘滞性（Viscosity）
- 做正确的事情要比做错误的事情困难。表现为两种形式：
  - 软件粘滞性
    - 需要对软件进行修改时，可能存在多种方法。有的方法可以保持原有的设计质量，另一些方法则会破坏原有的设计质量。如果，破坏软件质量的修改比保持原有设计质量的修改更容易实施时，我们就称该软件具有“软件粘滞性”。
  - 环境粘滞性
    - 当开发环境迟钝、低效时，就会产生环境粘滞性。
    - 例如：如果编译时间很长，那么开发人员可能会放弃那些能保持设计质量，但是却需要导致大规模重新编译的改动。

5）不必要的复杂性（Needless Complexity）
- 设计中包含不具有任何好处的基础结构。

6）不必要的重复（Needless Repetition）
- 设计中包含一些重复的结构，这些结构本来可以通过单一的抽象进行统一
  - 使用Cut/Copy/Paste实施源代码级的软件复用容易导致这一问题
  - 这种代码级别的冗余，将带来修改上的问题

7）晦涩性（Opacity）
- 很难阅读和理解，不要相信你永远都会如此清楚的了解你的每一行代码，“时间会冲淡一切”。要站在阅读者的角度进行设计

Shanghai Jiao Tong University

软件工程
Module: 构建设计模型

上海交通大学软件工程中心

构建设计模型

- 设计工程中的重要环节
- 设计模型是平台有关模型
- 关注 How
- 两种建模的方法
  - 结构化设计
  - 面向对象设计
  - ……

@第3.2节和第7章.教材

分析模型 VS. 设计模型

Analysis Model	Design Model
▪ Focus on understanding the problem ▪ Idealized design ▪ Behavior ▪ System structure ▪ Functional requirements ▪ A small model	▪ Focus on understanding the solution ▪ Operations and attributes ▪ Performance ▪ Close to real code ▪ Object lifecycles ▪ Nonfunctional requirements ▪ A large model

Shanghai Jiao Tong University

软件工程
Module: 结构化设计

上海交通大学软件工程中心

结构化设计

- 结构化设计概述
- 结构设计—概要设计
- 过程设计—详细设计

@3.2节.教材

结构化设计 (Structured Design, SD)

- 结构化设计是将结构化分析得到的数据流图映射成软件结构的一种设计方法
- 强调模块化、自顶向下逐步求精、信息隐蔽、高内聚低耦合等设计准则

结构化设计的内容

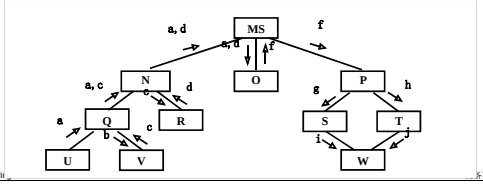
- 结构设计—概要设计
  - 结构图(Structure Chart)
  - 物理数据模型
- 过程设计—详细设计
  - 模块的处理过程
    - N-S图，PAD，PDL等

结构化设计

- 结构化设计概述
- 结构设计—概要设计
- 过程设计—详细设计

结构设计的工具——结构图

- 用结构图(Structure Chart)来描述软件系统的体系结构
- 描述一个软件系统由哪些模块组成，以及模块之间的调用关系
- 结构图的基本成分有：模块、调用和数据

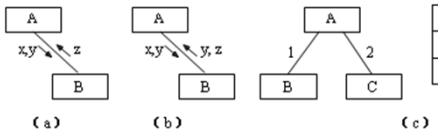


模块(module)

- 模块是指具有一定功能的可以用模块名调用的一组程序语句，包括函数和子程序。
- 一个模块具有其外部特征和内部特征
  - 外部特征包括：模块的接口(模块名、输入/输出参数、返回值等)和模块的功能
  - 内部特征包括：模块的内部数据和完成其功能的程序代码

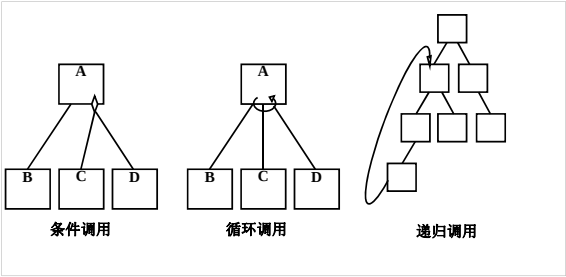
调用和数据

- 调用(call)：用从一个模块指向另一个模块的箭头来表示，其含义是前者调用了后者
  - 为了方便，有时常用直线替代箭头，此时，表示位于上方的模块调用位于下方的模块
- 数据(data)：模块调用时需传递的参数可通过在调用箭头旁附加一个小箭头和数据名来表示



	输入	输出
1	x,y	z
2	z,w	w

结构图中的辅助符号

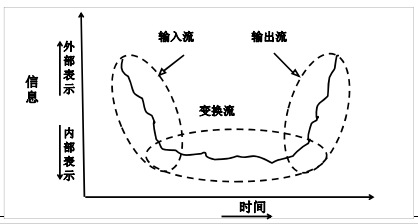


数据流图到结构图的映射

- 结构化设计是将结构化分析的结果(数据流图)映射成软件的体系结构(结构图)
- 将数据流图分为变换型数据流图和事务型数据流图，对应的映射分别称为变换分析和事务分析

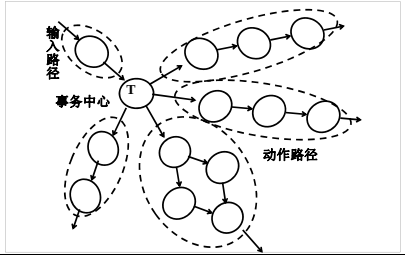
变换流

- 特征：数据流图可明显地分成三部分
  - 输入：信息沿着输入路径进入系统，并将输入信息的外部形式经过编辑、格式转换、合法性检查、预处理等辅助性加工后变成内部形式
  - 变换：内部形式的信息由变换中心进行处理
  - 输出：然后沿着输出路径经过格式转换、组成物理块、缓冲处理等辅助性加工后变成输出信息送到系统外



事务流

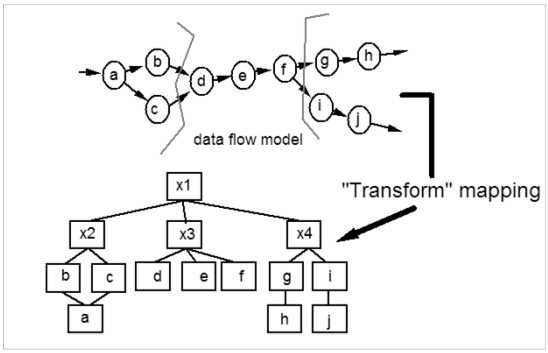
- 特征：数据流沿着输入路径到达一个事务中心，事务中心根据输入数据的类型在若干条动作路径中选择一条来执行
- 事务中心的任务是：接收输入数据(即事务)；分析每个事务的类型；根据事务类型选择执行一条动作路径



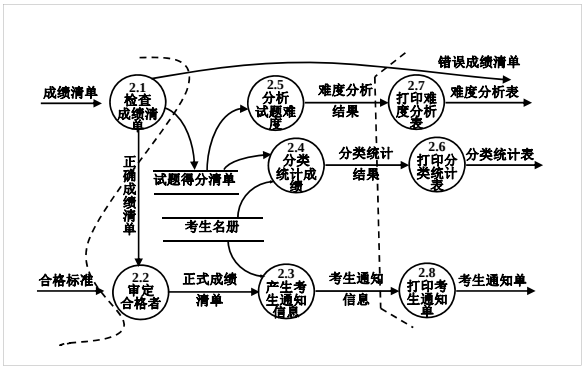
变换分析

- 变换分析的任务是将变换型的DFD映射成初始的结构图，步骤如下：
  - 划定输入流和输出流的边界，确定变换中心
  - 进行第一级分解：将DFD映射成变换型的程序结构
  - 进行第二级分解：将DFD中的加工映射成结构图中的一个适当的模块
  - 标注输入输出信息：根据DFD，在初始结构图上标注模块之间传递的输入信息和输出信息

变换映射

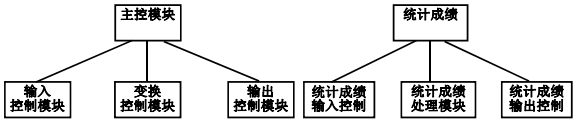


示例：统计成绩子图的输入、输出流边界



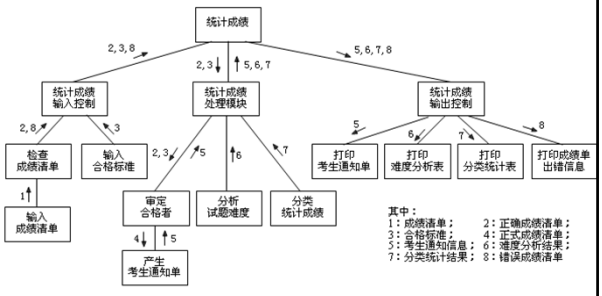
进行第一级分解

- 将DFD映射成变换型的程序结构
- 大型的软件系统第一级分解时可多分解几个模块，以减少最终结构图的层次数
  - 例如，每条输入或输出路径画一个模块，每个主要变换功能各画一个模块



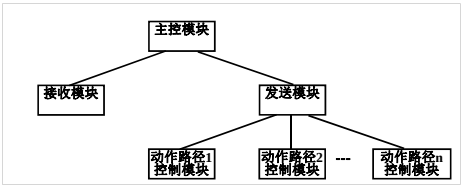
变换型的结构图 “统计成绩”第一级分解的结构图

“统计成绩”第二级分解的结构图



事务分析

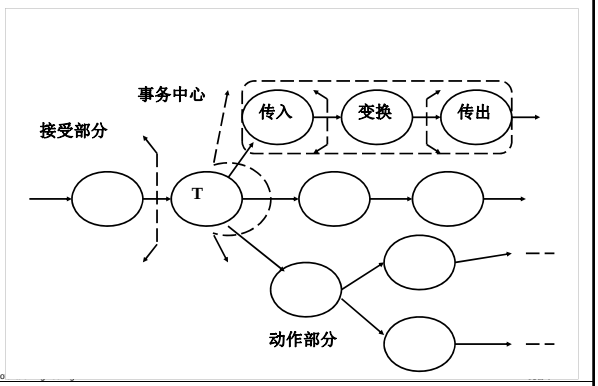
- 将事务型DFD映射成初始的结构图
  - 实例：银行业务中有存款、取款、查询余额、开户、转帐等多种事务，这种软件通常是接收一个事务，然后根据事务的类型执行一个事务处理的功能
- 事务型的结构图包括：
  - 主控模块：完成整个系统的功能
  - 接收模块：接收输入数据(事务)
  - 发送模块：根据输入事务的类型，选择一个动作路径控制模块
  - 动作路径控制模块：完成相应的动作路径所执行的子功能



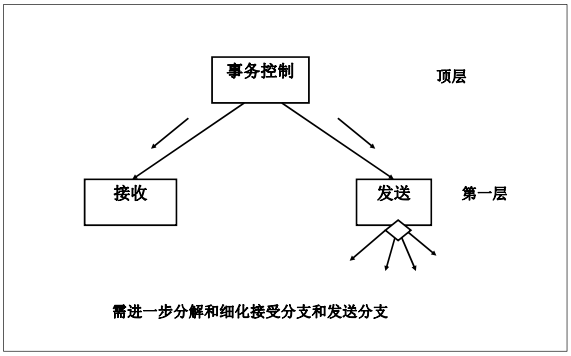
事务分析的步骤

- 在DFD图上确定边界
  - 事务中心
  - 接受部分（包括接受路径）
  - 发送部分（包括全部动作路径）
- 画出SC图框架
  - DFD图的三个部分分别映射为事务控制模块，接受模块和动作发送模块
- 分解和细化接受分支和发送分支

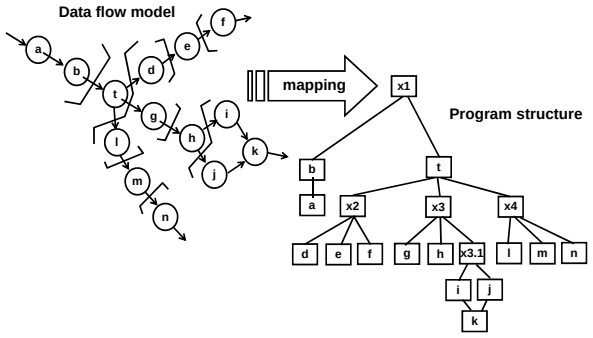
1）划分DFD



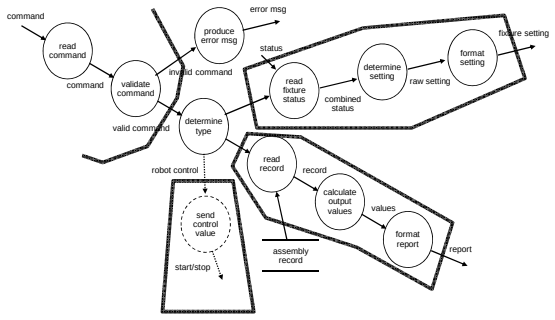
2）画出SC图框架



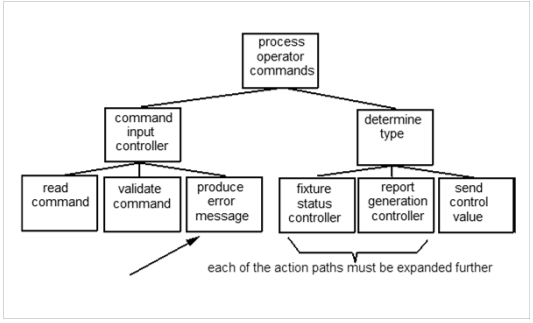
事务映射



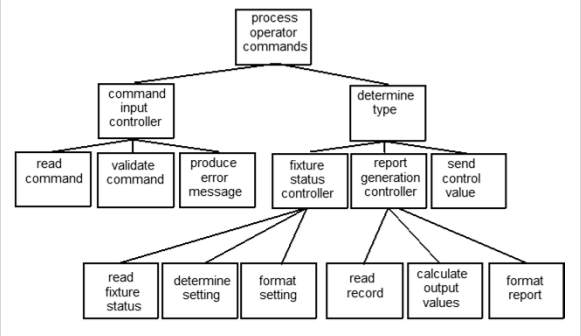
示例：1）划分DFD



示例：2）画出SC图框架



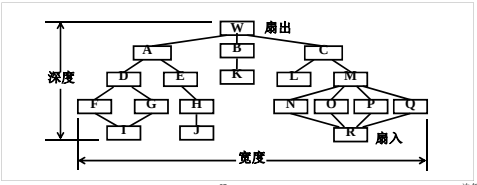
示例：3）细化SC图



优化结构图

结构图的相关度量

- 深度：程序结构图中控制的层数，例如图中所示的结构图的深度是5
- 宽度：程序结构图中同一层次上模块总数的最大值，例如图中所示的结构图的宽度为7
- 扇出(fan out)：该模块直接调用的模块数目。例如，例如图中模块M的扇出是4，模块A的是2，模块B的扇出是1
- 扇入(fan in)：能直接调用该模块的模块数目。例如图中模块G的扇入是1，模块I的扇入是2，模块R的扇入是4



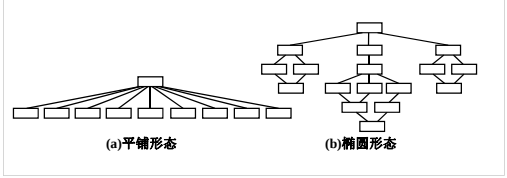
相关指标的含义

- 深度和宽度在一定程序上反映了程序的规模和复杂程度
  - 相对而言，如果程序结构图的深度和宽度较大，则说明程序的规模和复杂程度都较大。
  - 模块的扇入扇出会影响结构图的深度和宽度，例如减少模块的扇出，可能导致宽度变小而深度增加
- 一个模块的扇出过大通常意味着该模块比较复杂，然而扇出太少，可能导致深度的增加
  - 一般情况，一个模块的扇出以3～9为宜
- 一个模块的扇入表示有多少模块可直接调用它，它反映了该模块的复用(reuse)程度，因此模块的扇入越大越好

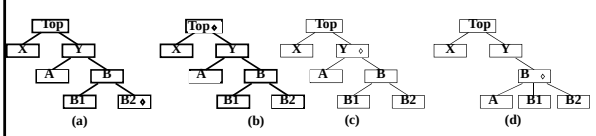
启发式设计策略

按照模块化设计原则，相应的启发式设计策略如下：

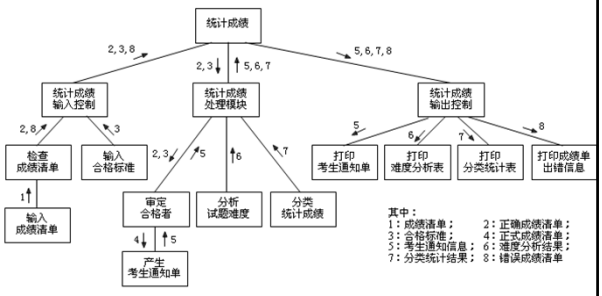
1) 降低耦合度，提高内聚度
2) 避免高扇出，并随着深度的增加，力求高扇入
   - 避免如图a那样的“平铺”形态，较好的结构图形态是如图b那样的“椭圆”型



3) 模块的影响范围应限制在该模块的控制范围内
   - 图a中，模块B2的影响范围(模块A)不在其控制范围(模块B2)内
   - 图b中，决策控制是在顶层模块，其影响范围(A、B2)在控制范围内，但是从决策控制模块到被控模块之间相差多个层次
   - c和d较合适，d为最好

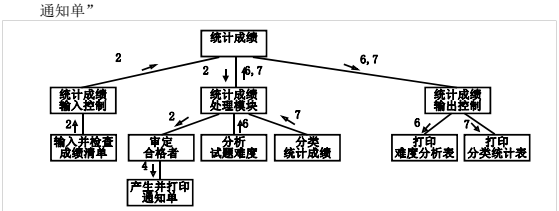


“统计成绩”结构图的改进-0



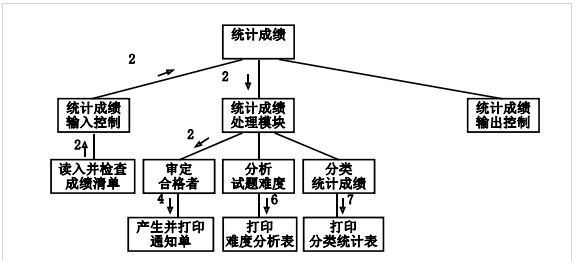
“统计成绩”结构图的改进-1

- 先将一些比较简单的模块合并到与其功能相一致的模块中，以减少耦合度
  - 将“输入成绩清单”、“检查成绩清单”、“打印成绩单出错信息”合并成“输入并检查成绩清单”
  - 将“输入合格标准”与“审定合格者”合并，仍取名“审定合格者”，但它包含读入合格标准功能
  - 将“产生考生通知单”与“打印考生通知单”合并成“产生并打印考生通知单”



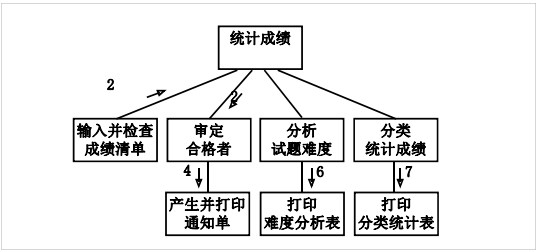
“统计成绩”结构图的改进-2

- 降低模块间的耦合程度
  - 将“打印难度分析表”模块和“打印分类统计表”模块分别作为“分析试题难度”模块和“分类统计成绩”模块的下属模块



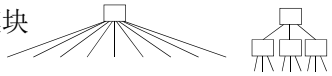
“统计成绩”结构图的改进-3

- 删去“统计成绩输出控制”
- “统计成绩输入控制”模块和“统计成绩处理模块”均为“管道”模块，也可删去



结构图改进技巧

- 减少模块间的耦合度
- 消除重复功能
- 消除“管道”模块
- 模块的大小适中
- 避免高扇出
- 应尽可能研究整张结构图，而不是只考虑其中的一部分



高扇出时重新分解

结构化设计

- 结构化设计概述
- 结构设计—概要设计
- 过程设计—详细设计

过程设计

- 目的
  - 确定模块采用的算法和块内数据结构
- 任务：编写软件的“过程设计说明书”
  - 为每个模块确定采用的算法
  - 确定每一模块使用的数据结构
  - 确定模块接口的细节

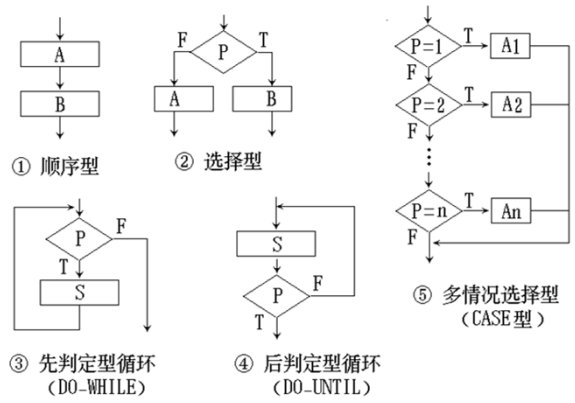
过程设计的原则

- 清晰第一的设计风格
- 结构化的控制结构
- 逐步细化的实现方法

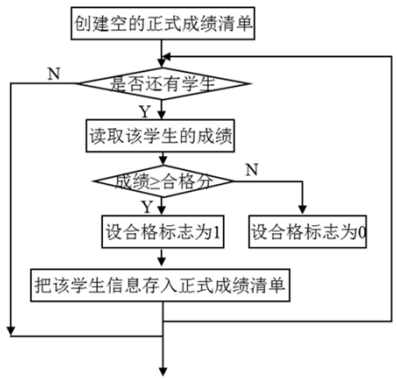
过程设计工具

- 流程图（Flow Diagram）
- N-S图
- PAD图（Problem Analysis Diagram）
- PDL语言

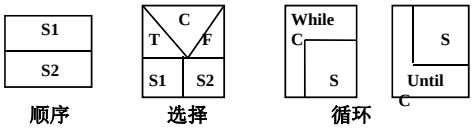
流程图



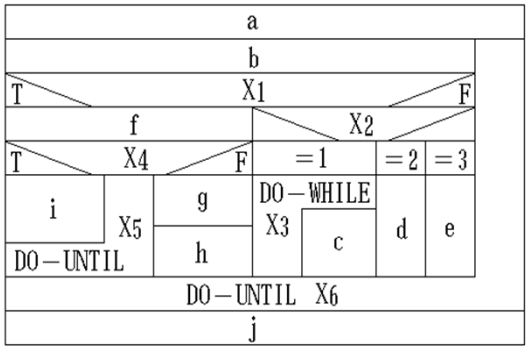
“审定合格者”模块的程序流程图



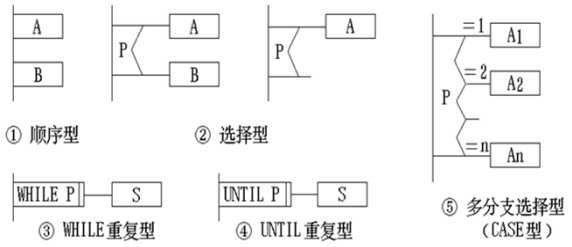
N-S图（盒图）



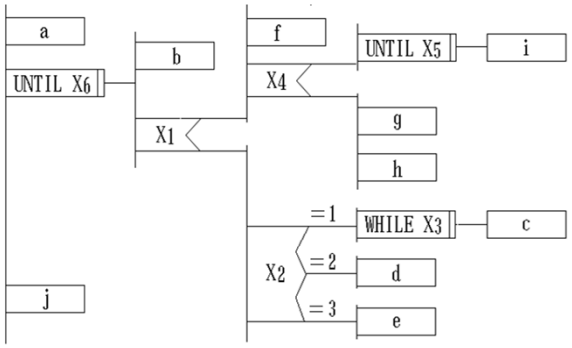
N-S图示例



问题分析图(PAD)



PAD示例



PDL (Program Design Language)

- PDL是一种用于描述功能模块的算法设计和加工细节的语言。称为设计程序用语言。它是一种伪码。
- 伪码的语法规则分为“外语法”和“内语法”。
- PDL具有严格的关键字外语法，用于定义控制结构和数据结构，同时它的表示实际操作和条件的内语法可使用自然语言的词汇。

示例: 拼词检查程序

```
PROCEDURE  spellcheck  IS
BEGIN
   split document into single  words
   look up words in dictionary
   display words which are not in dictionary
   create a new dictionary
END spellcheck
```

Shanghai Jiao Tong University

软件工程

Module: 面向对象设计

上海交通大学软件工程中心

Agenda

- OO Design Overview
- The Principles of Object-Oriented Design
- Architectural Design
- Component Design
- Class Design
- From Design to Implementation

@第6章.教材

软件工程概论

Introduction to Software Engineering

OO DESIGN OVERVIEW

Understanding Design

- A process that uses the products of analysis to produce a specification for implementing a system.
- A logical description of how a system will work.
- Emphasizes a conceptual solution (in software and hardware) that fulfills the requirements, rather than its implementation.

- *“do the right thing (analysis), and do the thing right (design)”.*

Analysis Versus Design

Analysis

- Focus on understanding the problem
- Idealized design
- Behavior
- System structure
- Functional requirements
- A small model

Design

- Focus on understanding the solution
- Operations and attributes
- Performance
- Close to real code
- Object lifecycles
- Nonfunctional requirements
- A large model

Object Oriented Design

- The specification of a logical software solution in terms of software objects,
  - **such as their classes, attributes, methods, and collaborations.**

- During object-oriented design (or simply, object design) there is an emphasis on defining software objects and how they collaborate to fulfill the requirements.

软件工程概论

Introduction to Software Engineering

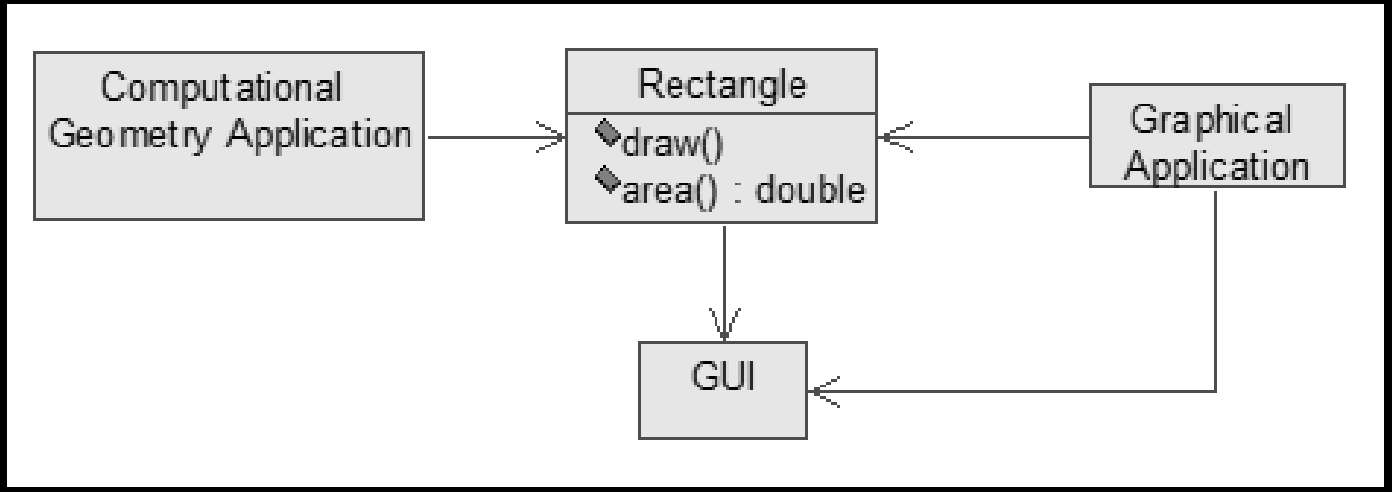
THE PRINCIPLES OF OBJECT-ORIENTED DESIGN

The Principles of Object-Oriented Design

- 单一职责原则
  - SRP，Single Responsibility Principle
- 里氏替换原则
  - LSP， Liskov Substitution Principle
- 依赖倒置原则
  - DIP，Dependence Inversion Principle
- 接口隔离原则
  - ISP，The Interface Segregation Principle
- 开-闭原则
  - OCP，Open-Closed Principle

SRP: The Single – Responsibility Principle

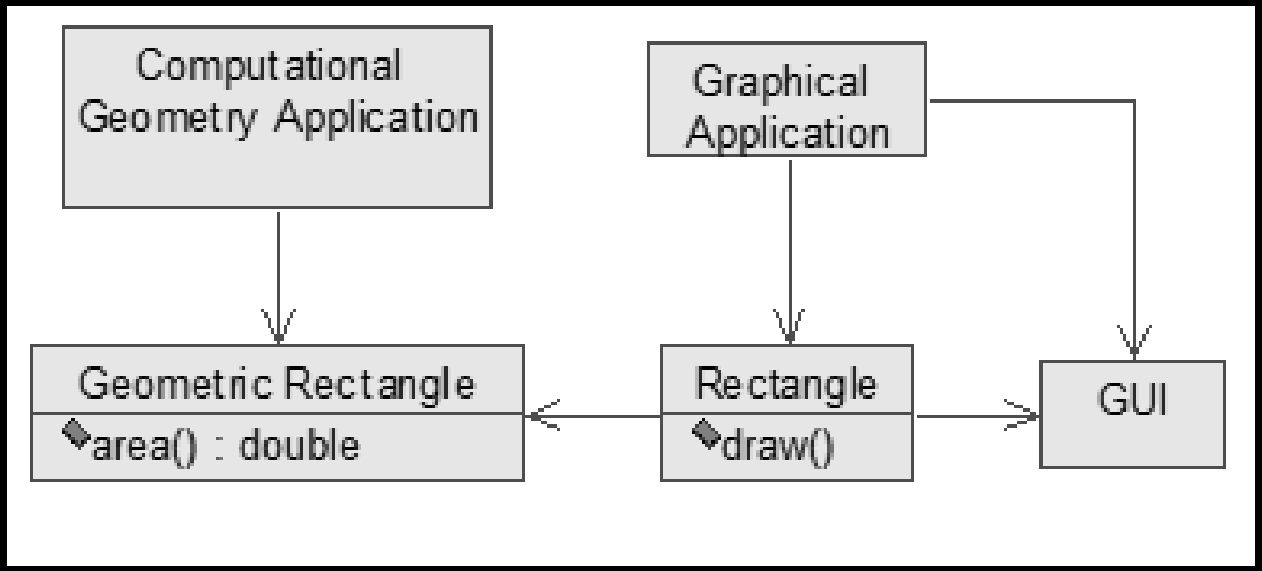
- A Class should have only one reason to change
  - **In the context of the SRP, we define a responsibility to be "a reason for change"**
- Example:



- The rectangle class has two responsibilities
  - **This design violates the SRP.**

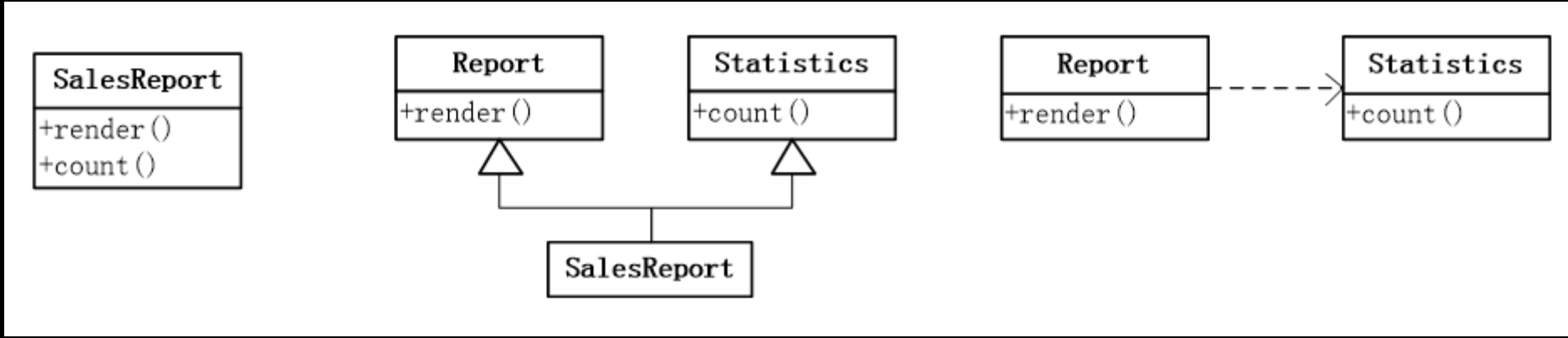
SRP: The Single – Responsibility Principle

- Separating Coupled Responsibilities



SRP: The Single – Responsibility Principle

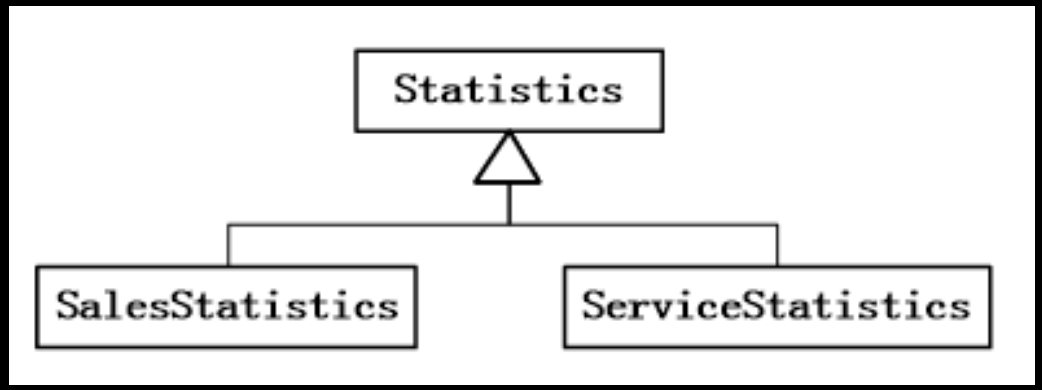
- Separating Coupled Responsibilities

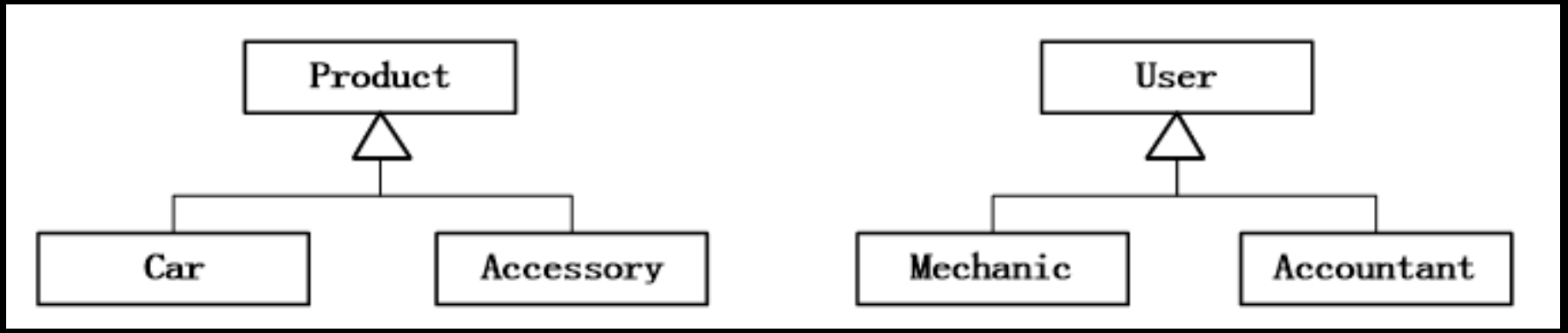


LSP: The Liskov Substitution Principle

- The Liskov Substitution Principle
  - subtypes must be substitutable for their base types
    - SubClass can instead of parent Class
  - If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T [Liskov88].
    - If using implementation of T or subclass of T to instead of T, Software P which is associated with type T(interface or abstract class) can work well.
  - Essential： the objects which is in the same inherit architecture should have behavior features.
- Example:
  - In real world , penguin and bird is a “is-a” relationship. If we design penguin is a subclass of bird class ,we violate LSP principle .because bird can fly( ) while penguin can not fly. So how to change this design?

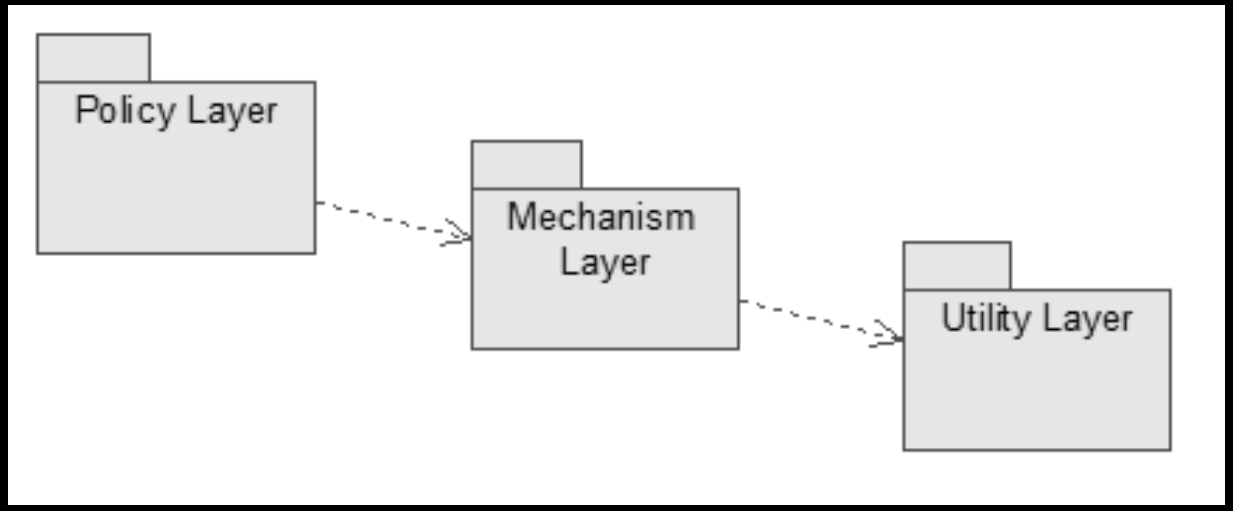
LSP: The Liskov Substitution Principle





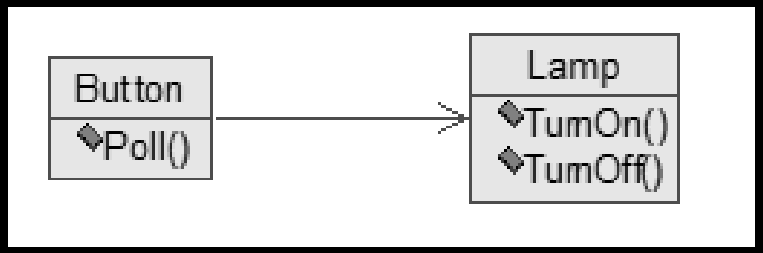
DIP: The Dependency Inversion Principle

- The Dependency Inversion Principle
  - **High-level modules should not depend on low-level modules.**
    - **Both should depend on abstractions**
  - **Abstraction should not depend on details**
    - **Details should depend on abstractions**
- Example

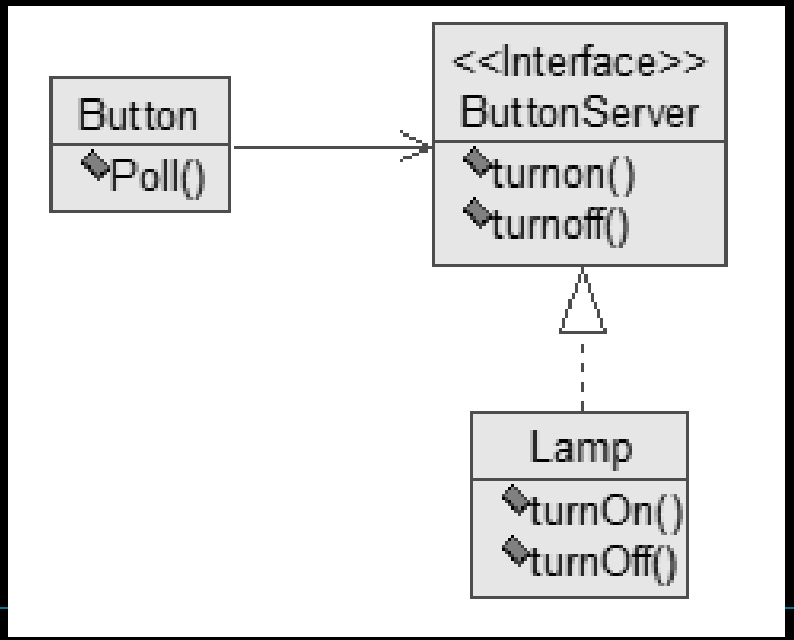


DIP: The Dependency Inversion Principle

- Depend on Abstractions
  - **Naive model of a Button and a Lamp**

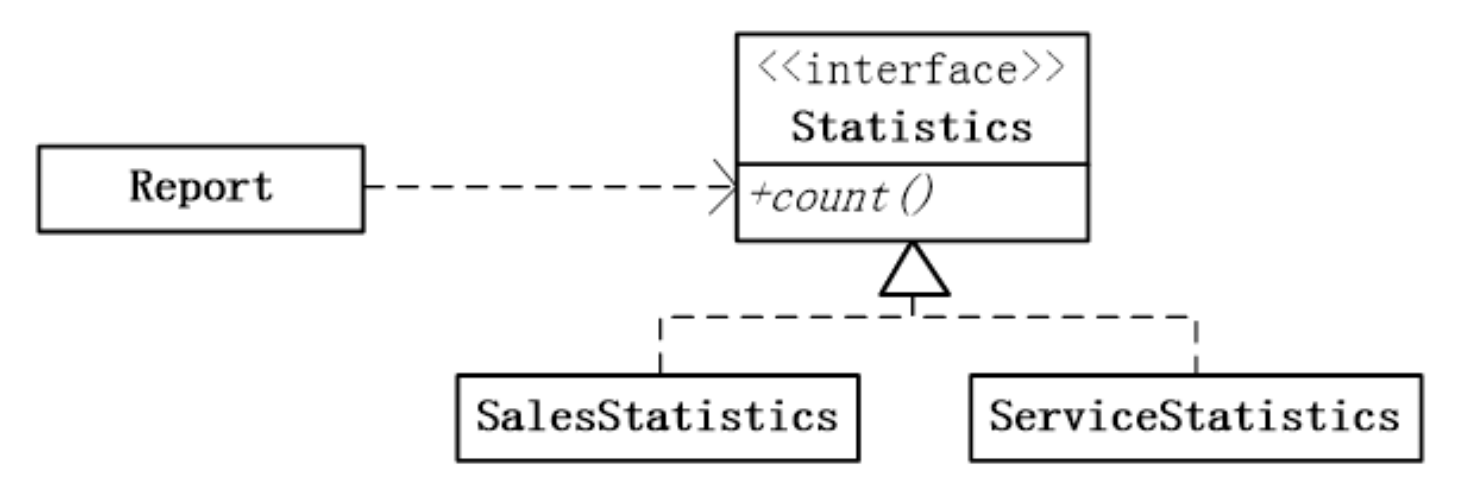


  - **Dependency inversion applied to the Lamp**



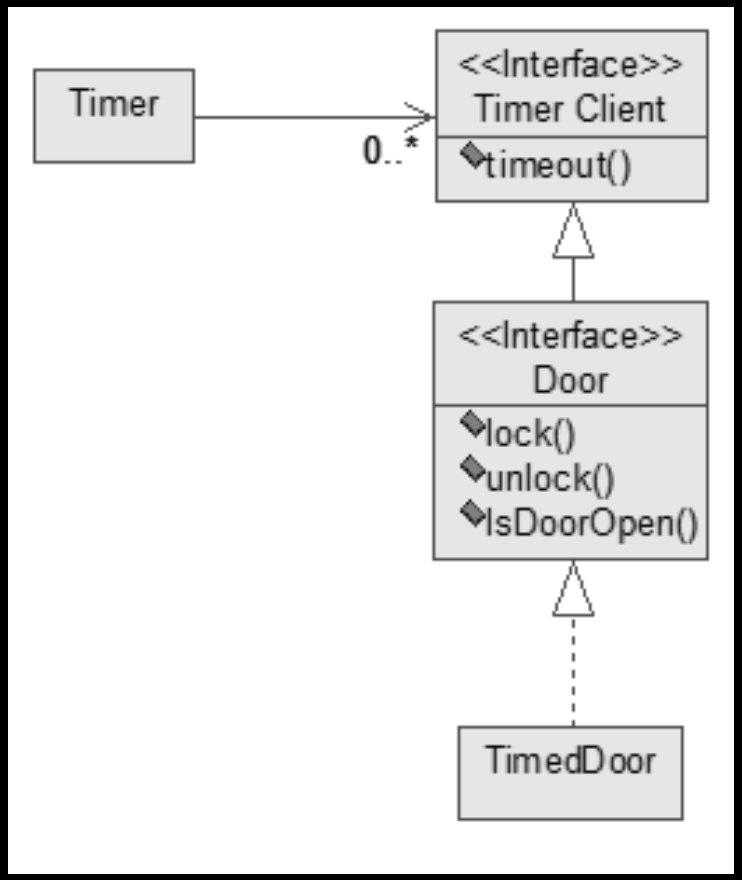
DIP: The Dependency Inversion Principle

- Example



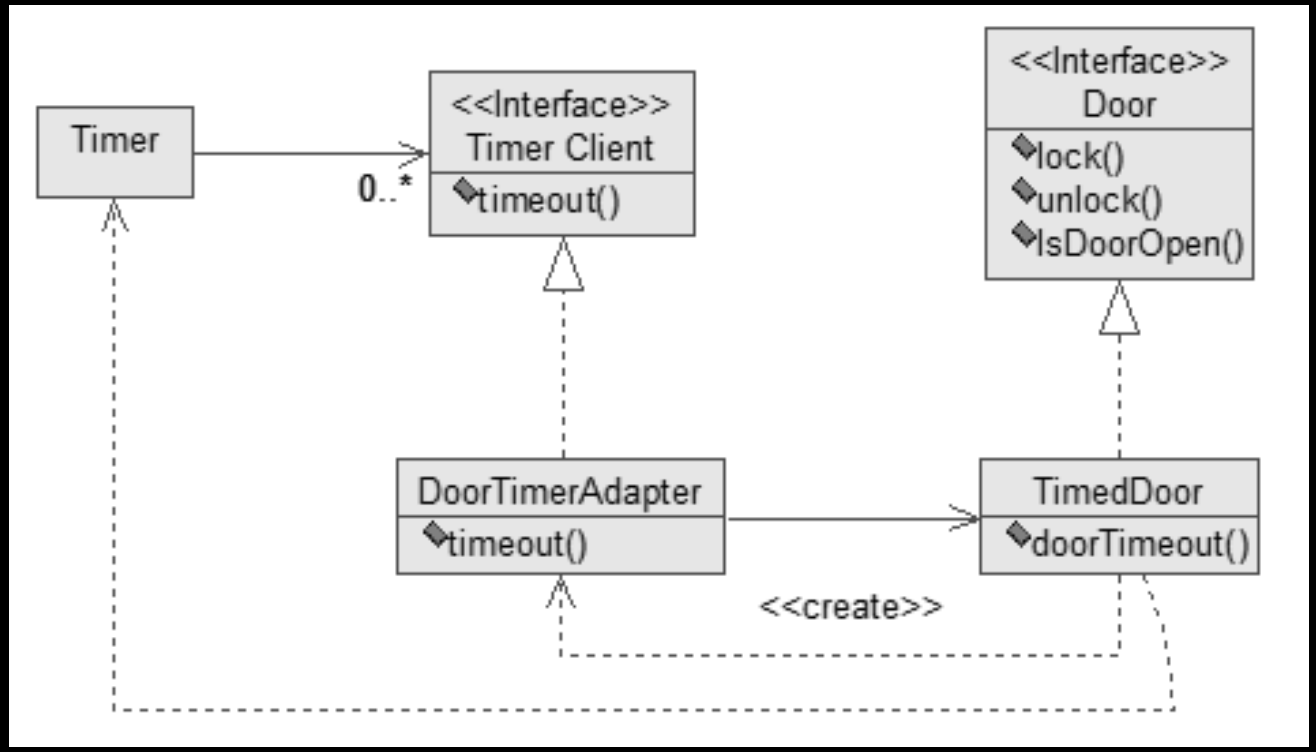
ISP: The Interface Segregation Principle

- The Interface Segregation Principle
  - **Clients should not forced to depend on methods that they do not use**
- Example



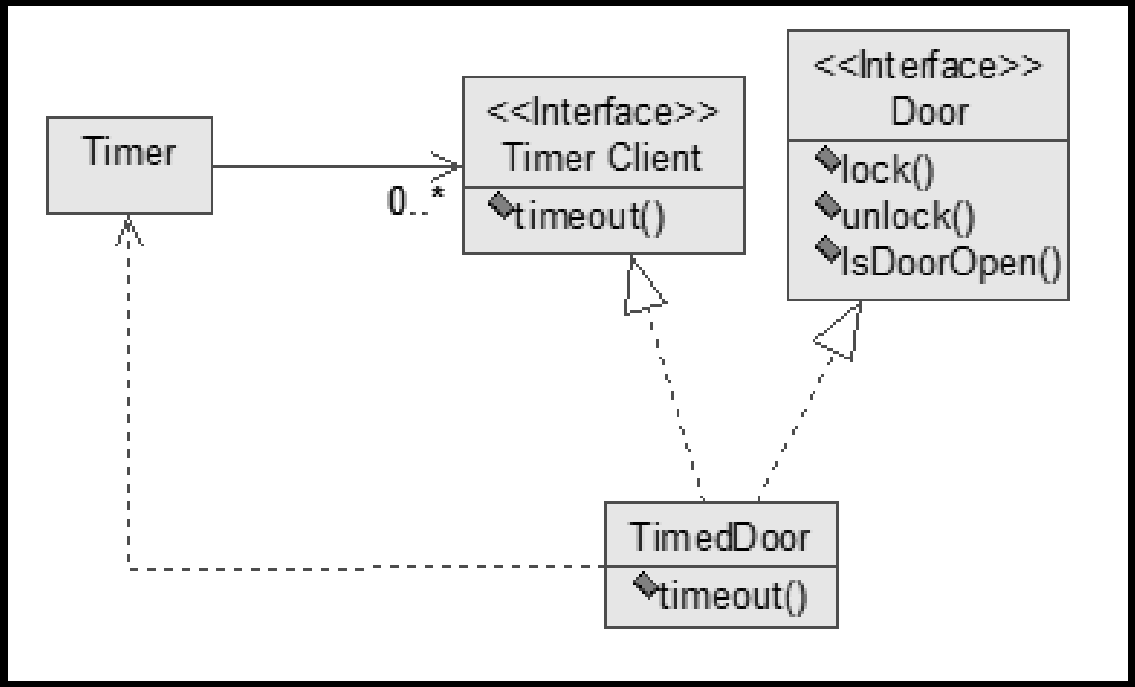
ISP: The Interface Segregation Principle

- Example



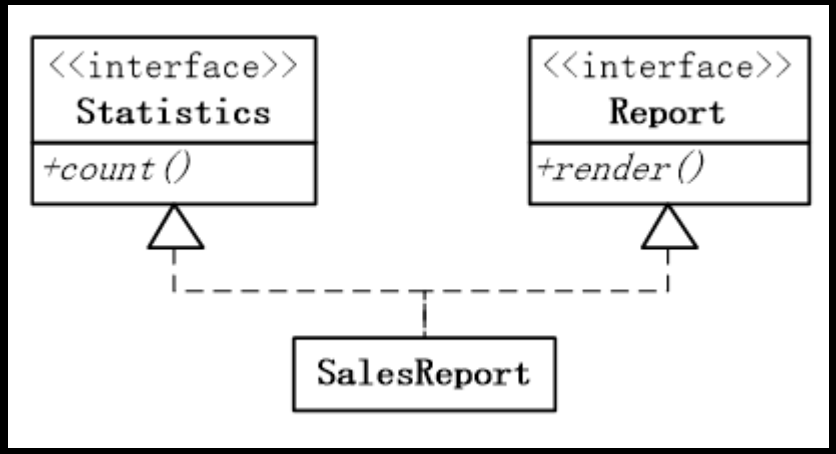
ISP: The Interface Segregation Principle

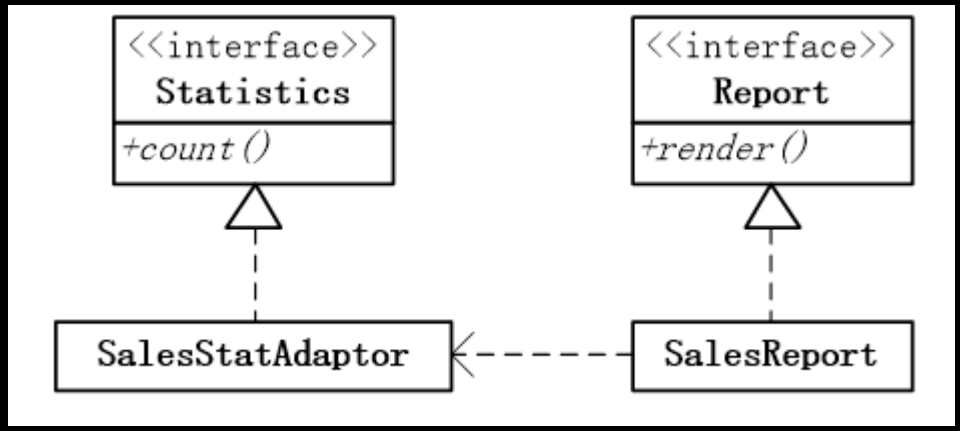
- Example



ISP: The Interface Segregation Principle

- Example

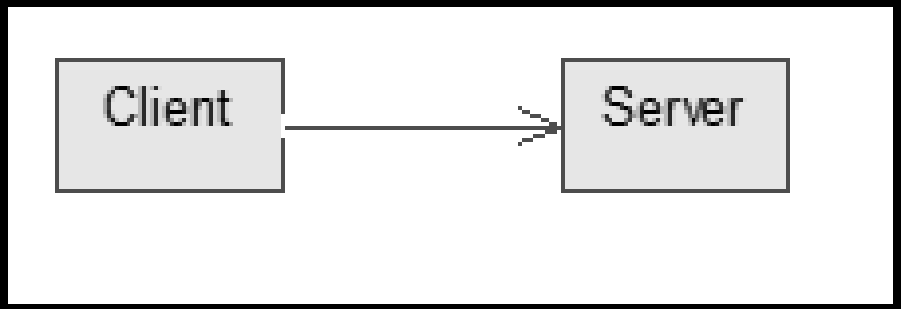




OCP: The Open-Closed Principle

- The Open-Closed Principle (OCP), described by Bertrand Meyer in [Meyer88]

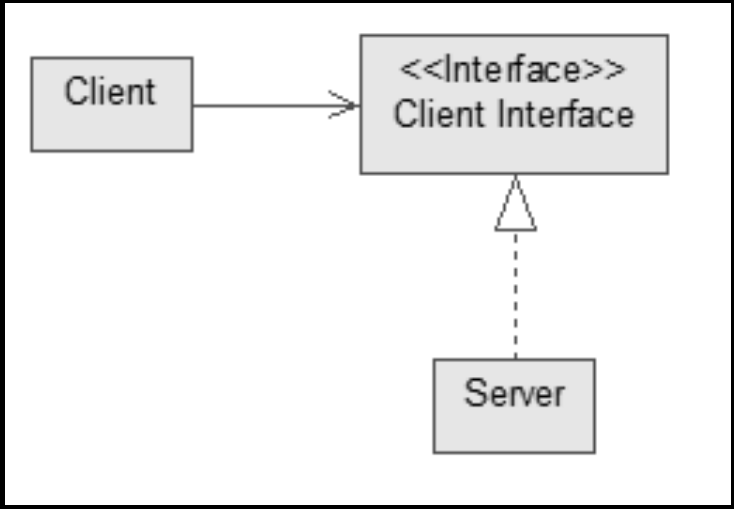
- A definition of OCP is:
  - **Modules should be both open (for extension; adaptable) and closed (the module is closed to modification in ways that affect clients).**
    - **In OCP, "module" includes all discrete software elements, including methods, classes, subsystems, applications, and so forth**
- Example:



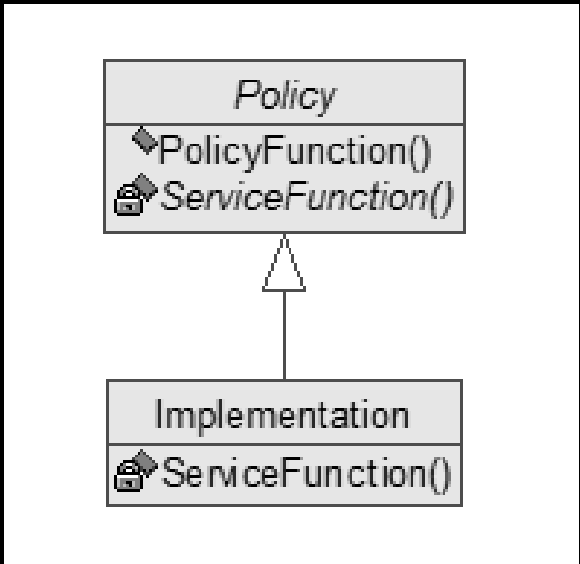
  - **client is not open and closed**

OCP: The Open-Closed Principle

- The Design conforms to the OCP
  - **Strategy pattern: Client is both open and closed**

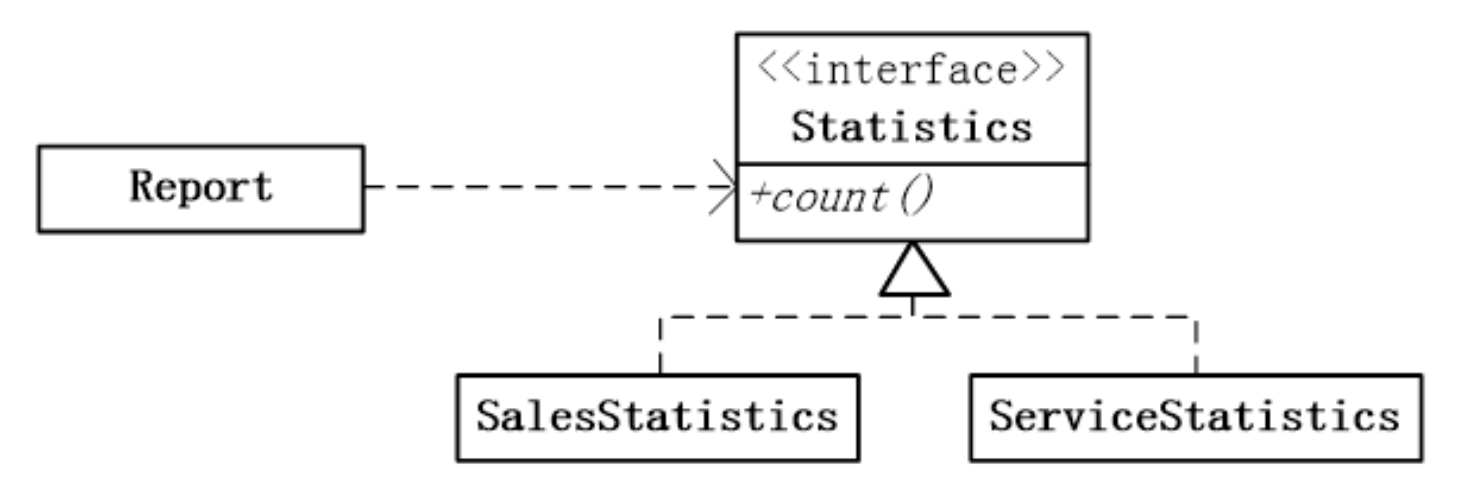


- Another example
  - **Template Method pattern:**
    - **Base class is open and closed**



OCP: The Open-Closed Principle

- Example

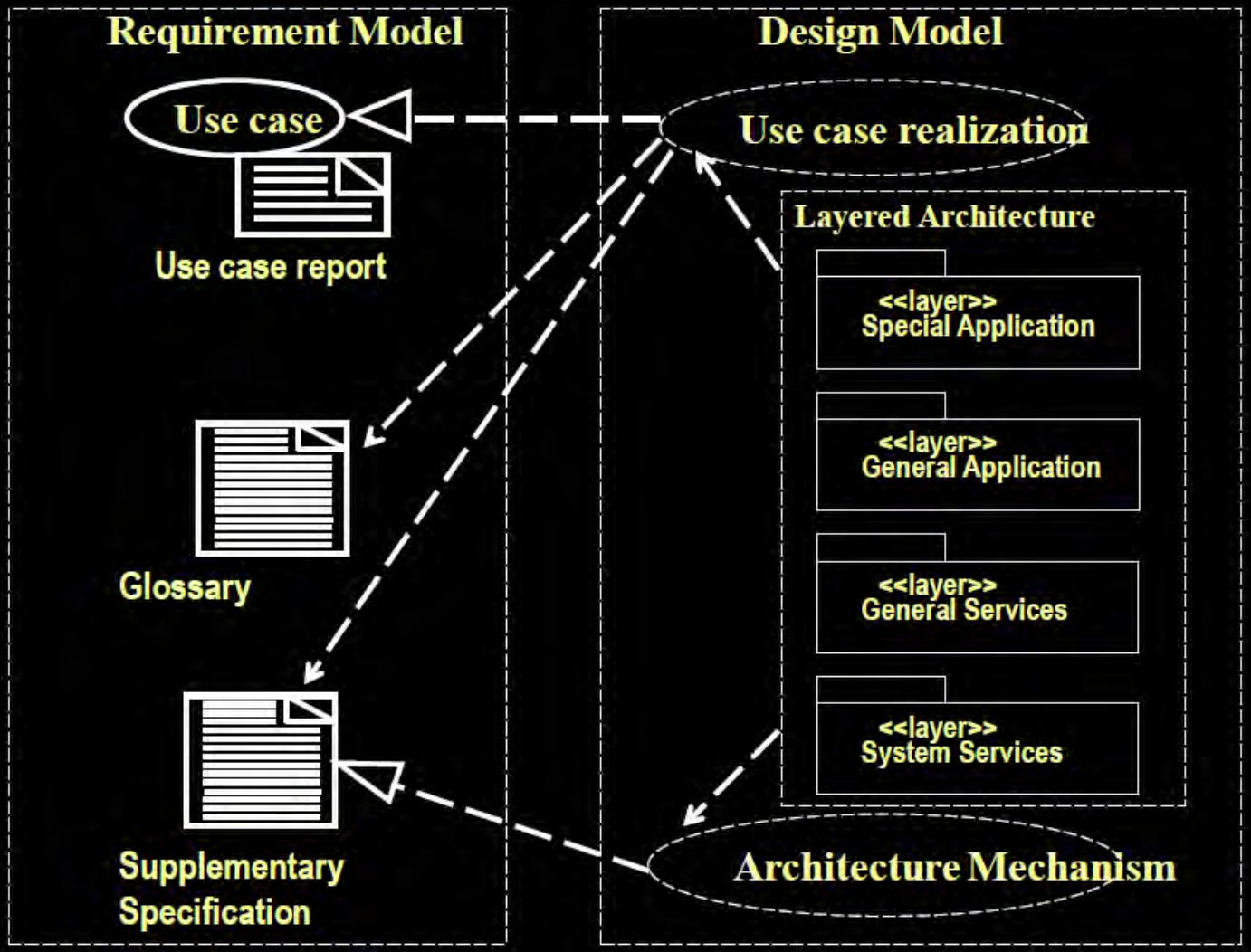


软件工程概论

Introduction to Software Engineering

ARCHITECTURAL DESIGN AND UML PACKAGE DIAGRAM

Design Model Overview

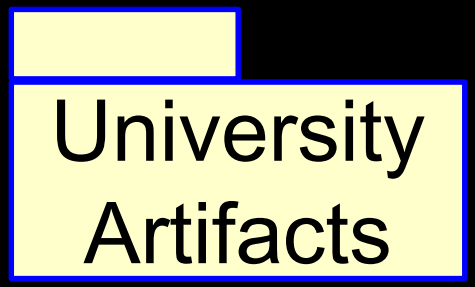


Architectural Design and UML Package Diagram

- UML Package Diagram
- Principles of Package Design
- Architectural Patterns
- Resolution of Architectural Factors
- Identify Design Elements
- Organizing the design model packages

Review: What Is a Package?

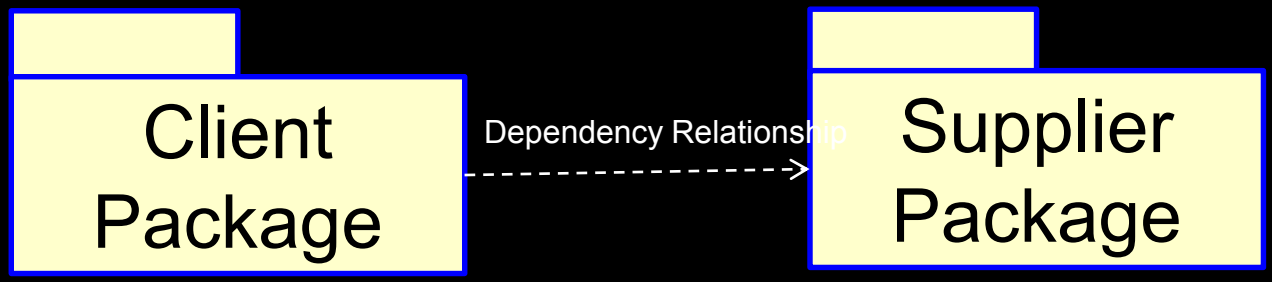
- A package is a general purpose mechanism for organizing elements into groups.
- It is a model element that can contain other model elements.



- A package can be used
  - **To organize the model under development.**
  - **As a unit of configuration management.**

Package Relationships: Dependency

- Packages can be related to one another using a dependency relationship



- Dependency Implications
  - **Changes to the Supplier package may affect the Client package**
  - **The Client package cannot be reused independently because it depends on the Supplier package**

Package Organization Guidelines

- Package Functionally Cohesive Vertical and Horizontal Slices
- Package a Family of Interfaces
- Package by Work and by Clusters of Unstable Classes
- Most Responsible Are Most Stable
- Factor out Independent Types
- Use Factories to Reduce Dependency on Concrete Packages
- No Cycles in Packages

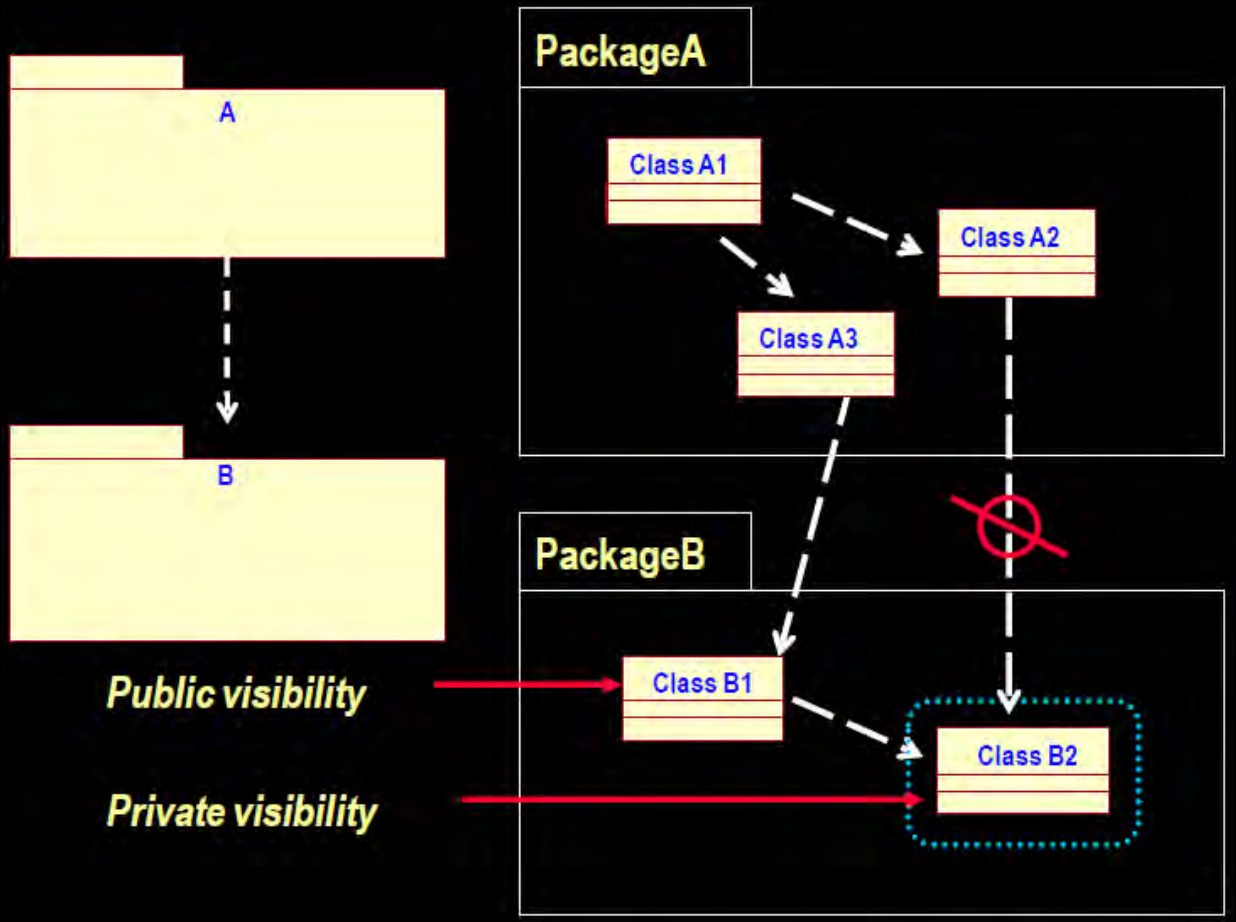
Packaging Tips: Functionally Related Classes

- Criteria for determining if classes are functionally related
  - **Changes in one class' behavior and/or structure necessitate changes in another class**
  - **Removal of one class impacts the other class**
  - **Two objects interact with a large number of messages or have a complex intercommunication**
  - **A boundary class can be functionally related to a particular entity class if the function of the boundary class is to present the entity class**
  - **Two classes interact with, or are affected by changes in the same actor**
  - **Two classes have relationships between each other**
  - **One class creates instances of another class**

Packaging Tips: Functionally Related Classes

- Criteria for determining when two classes should NOT be placed in the same package
  - **Two classes that are related to different actors should not be placed in the same package**
  - **An optional and a mandatory class should not be placed in the same package**

Package Dependencies: Package Element Visibility



Only public classes can be referenced outside of the owning package

OO Principle: Encapsulation

Principles of Package Design

- Granularity: The Principles of Package Cohesion
  - **REP: The Release-Reuse Equivalency Principle**
  - **CRP: The Common Reuse Principle**
  - **CCP: The Common Closure Principle**
- Stability: The Principles of Package Coupling
  - **ADP: The Acyclic Dependencies Principle**
  - **SDP: The Stable Dependencies Principle**
  - **SAP: The Stable Abstractions Principle**

REP: The Release-Reuse Equivalency Principle

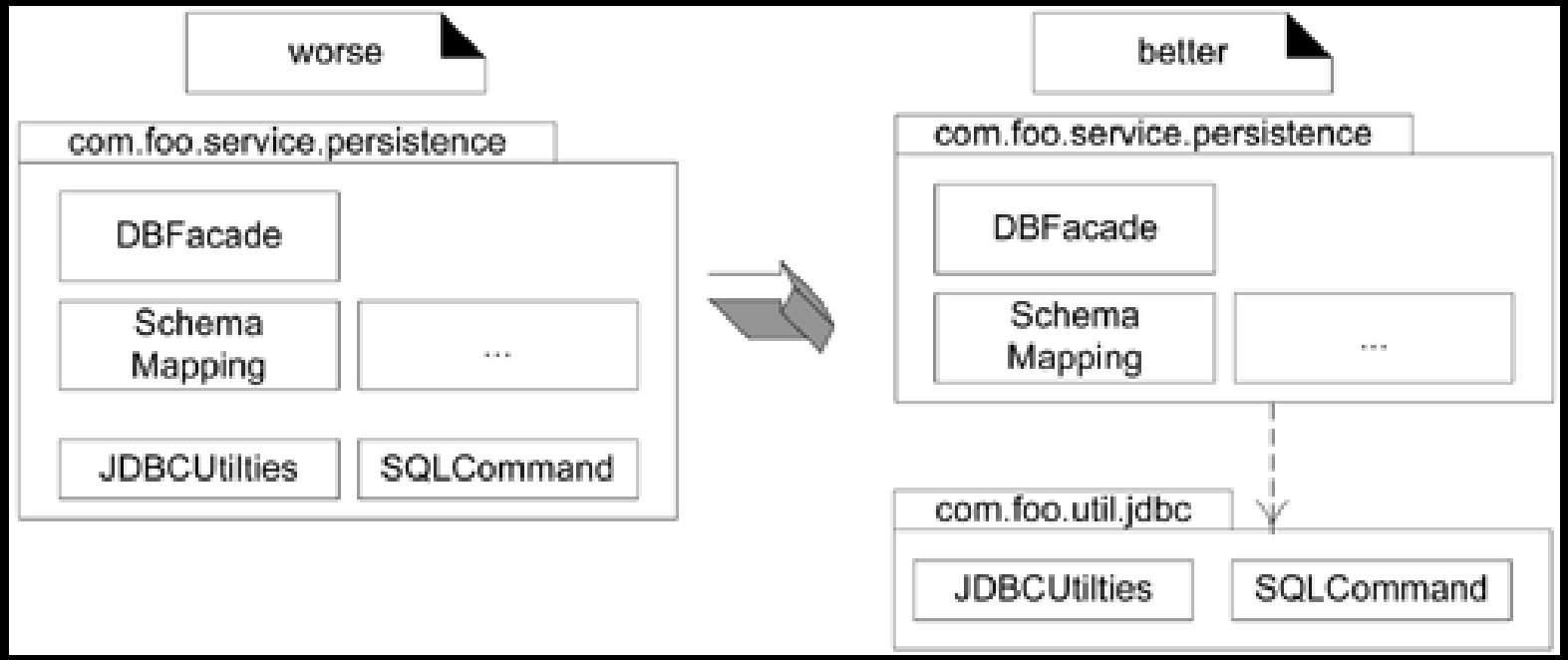
- The Release-Reuse Equivalency Principle
- The granule of reuse is the granule of release
  - The granularity of reuse is the granularity of release
  - packages are usually the basic unit of development work and of release
  - In order to provide the guarantees that reusers need, authors must organize their software into reusable packages and then track those packages with release numbers
  - the granule of reuse (i.e.,a package) can be no smaller than granule of release.
    - Anything that we reuse must also be released and tracked.
  - Either all of the classes in a package are reusable or none of them are.
    - If a package contains software that should be reused, then it should not also contain software that is not designed for reuse.

CRP: The Common Reuse Principle

- The Common Reuse Principle
  - **The classes in a package are reused together.**
    - **If you reuse one of the classes in a package, you reuse them all.**
- CRP helps us to decide which classes should be placed into a package
  - **classes that tend to be reused together belong in the same package.**
- CRP tell us more about what classes shouldn't be together.
  - **classes which are not tightly bound to each other with class relationships should not be in the same package.**

CRP: The Common Reuse Principle

- Guideline: Factor out Independent Types
  - **Organize types that can be used independently or in different contexts into separate packages.**
- Example

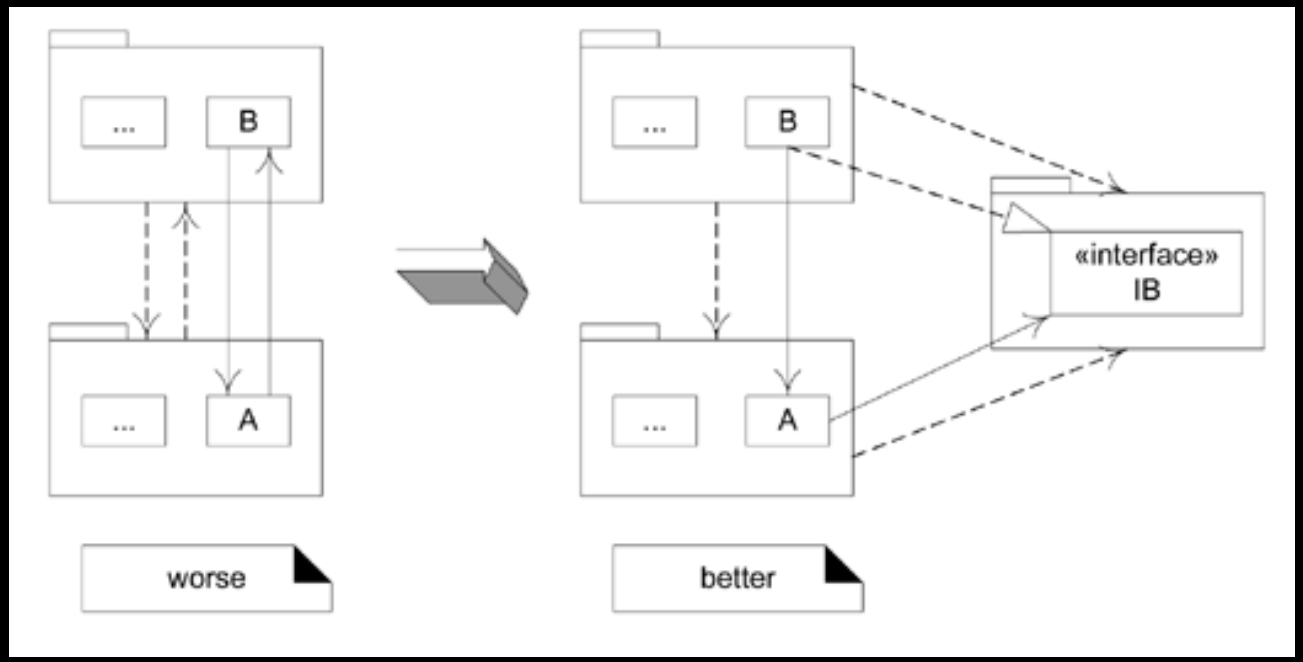


CCP: The Common Closure Principle

- The Common Closure Principle
  - **The classes in a package should be closed together against the same kinds of changes.**
    - **A change that affects a package affects all the classes in that package and no other package**
  - **maintainability is more important**
- Guideline: Package by Work and by Clusters of Unstable Classes
  - **Reduce widespread dependency on unstable packages**
  - **Example:**
    - **1) there is an existing large package P1 with thirty classes, and**
    - **2) there is a work trend that a particular subset of ten classes is regularly modified and re-released.**
      - **refactor P1 into P1-a and P1-b, where P1-b contains the ten frequently worked on classes.**

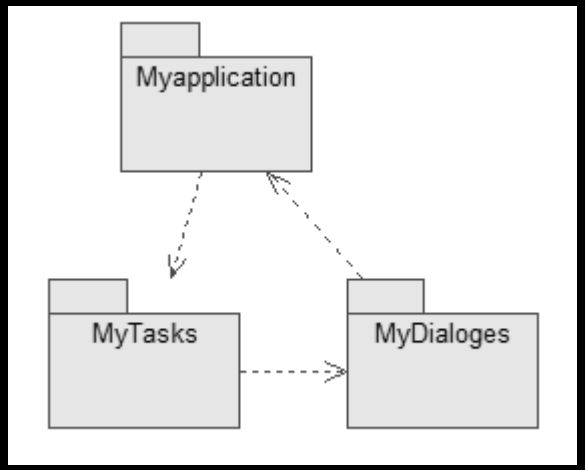
ADP: The Acyclic Dependencies Principle

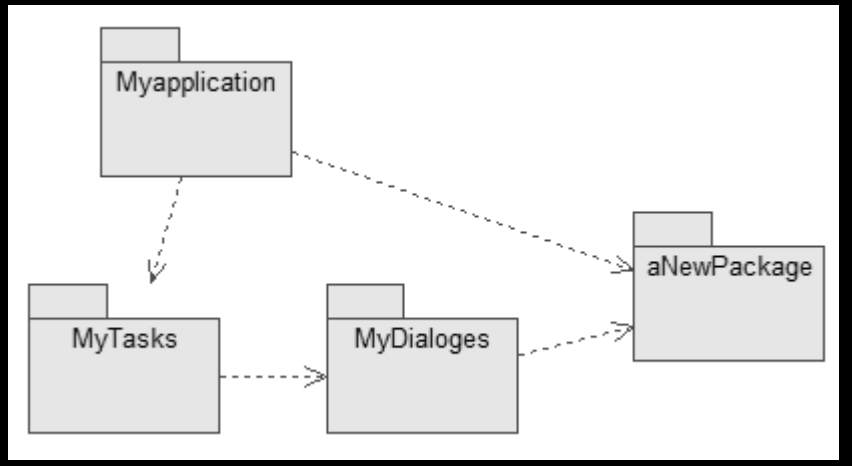
- The Acyclic Dependencies Principle
  - **Allow no cycles in the package-dependency graph.**
- There are two solutions:
  - **Factor out the types participating in the cycle into a new smaller package.**
  - **Break the cycle with an interface.**

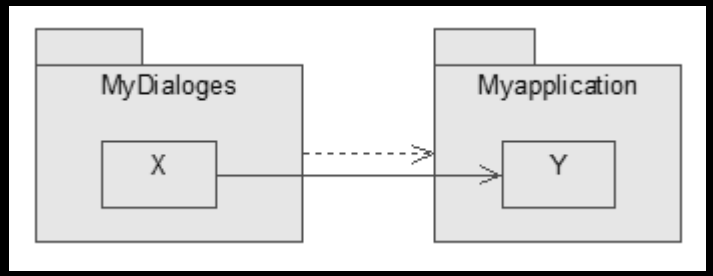


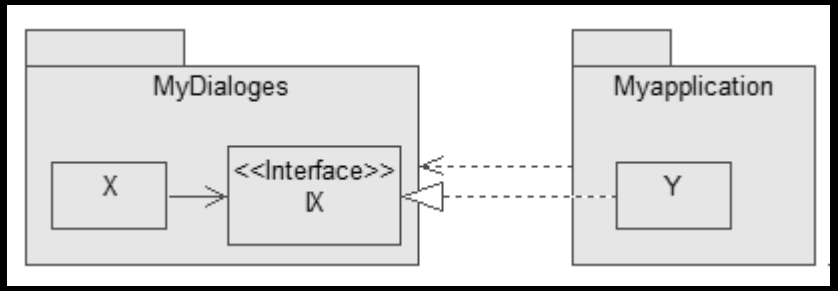
ADP: The Acyclic Dependencies Principle

- Example



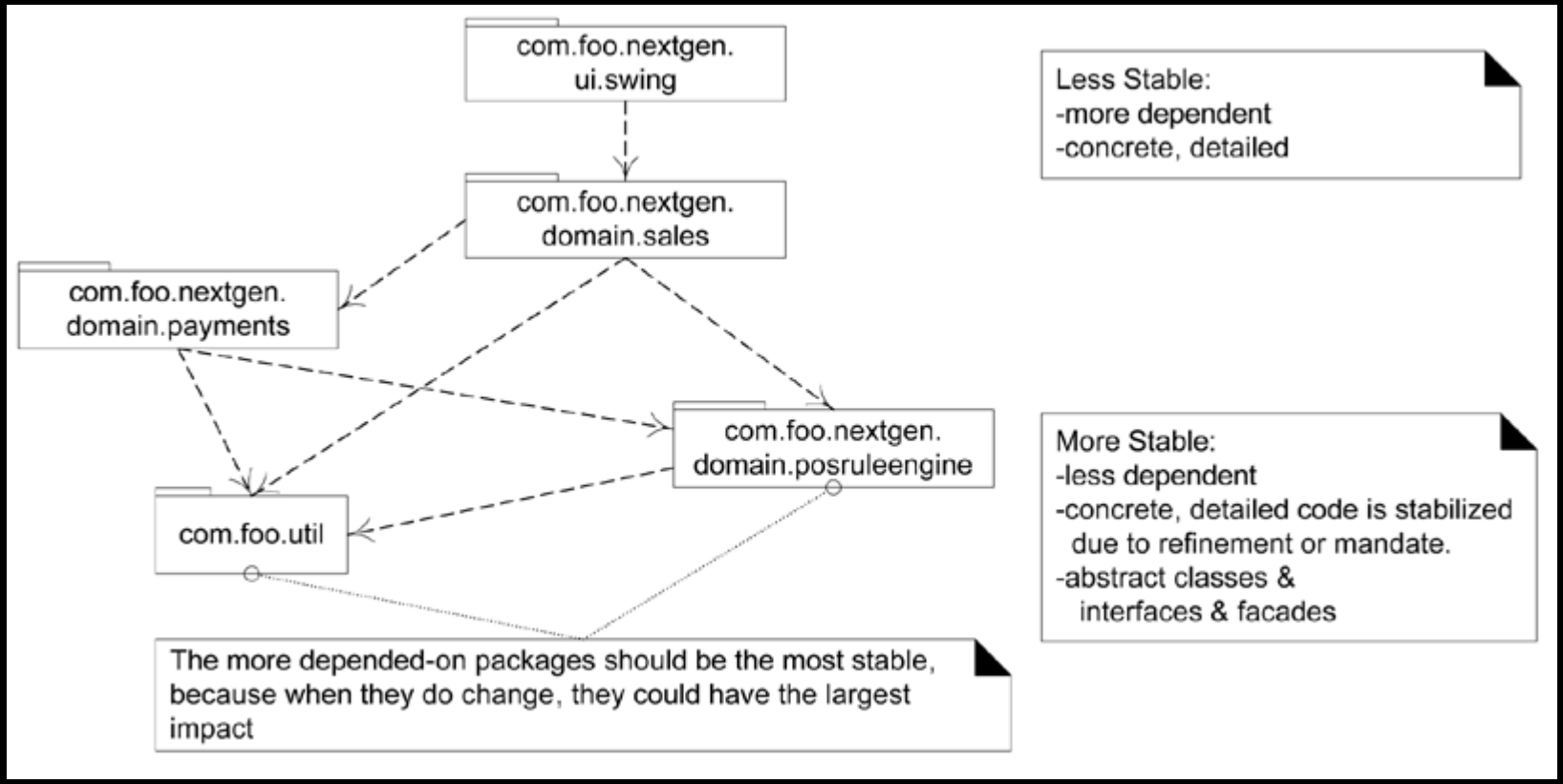






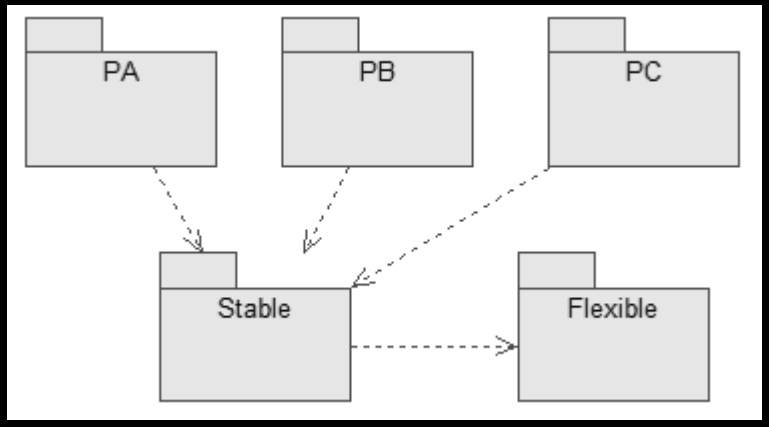
SDP: The Stable Dependencies Principle

- The Stable Dependencies Principle
  - **Depend in the direction of stability**
- Guideline: Most Responsible Are Most Stable

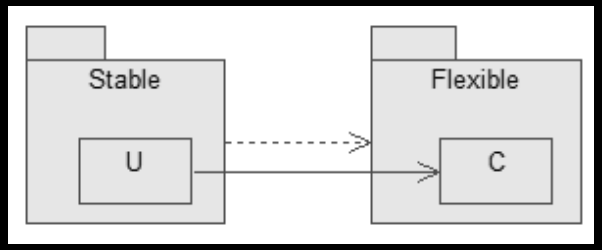


SDP: The Stable Dependencies Principle

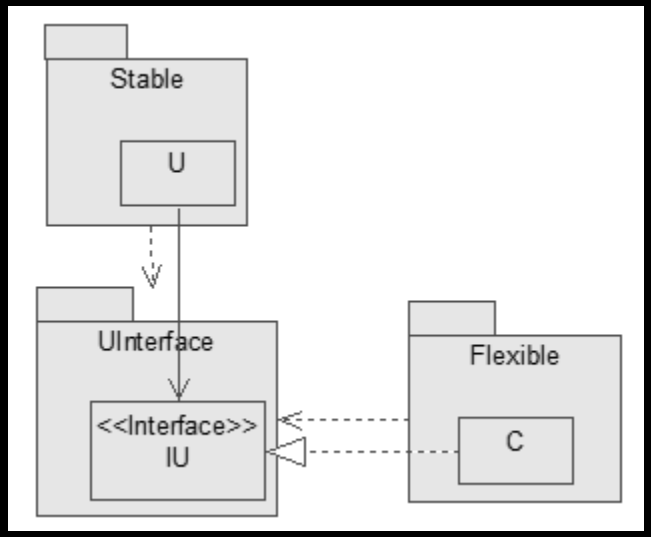
- Example
  - Violation of SDP



  - The cause of the bad dependency



  - Fixing the stability violation using DIP

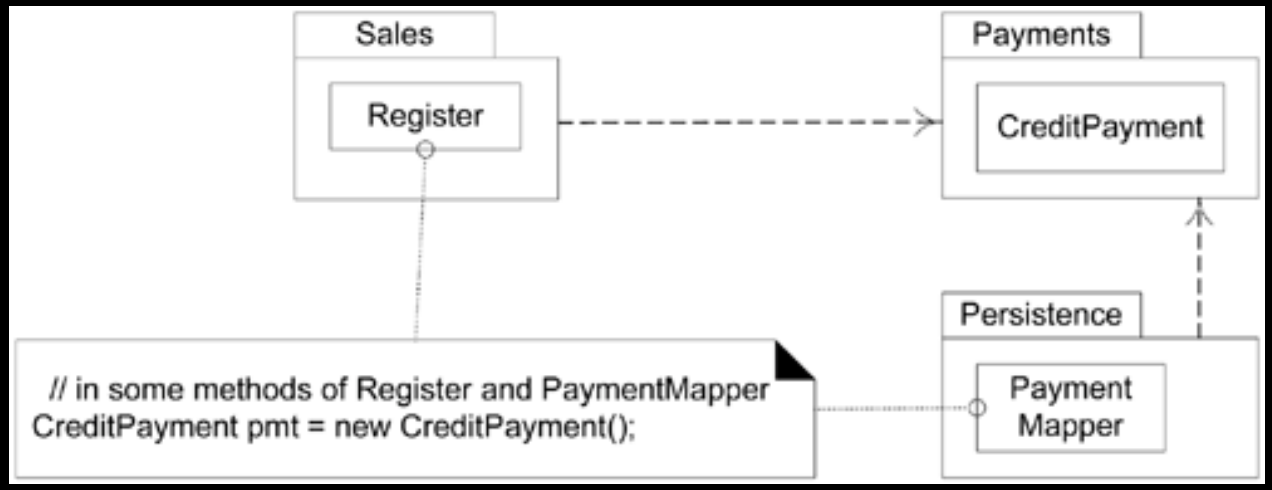


SAP: The Stable Abstractions Principle

- The Stable Abstractions Principle
  - A package should be as abstract as it is stable
    - a stable package should also be abstract
      - Its stability does not prevent it from being extended
    - a instable package should be concrete
      - Its instability allows the concrete code within it to be easily changed
  - The SAP and SDP combined amount to the DIP for packages.
- Guideline: Package a Family of Interfaces
  - Place a family of functionally related interfaces in a separate package separate from implementation classes.
    - The Java technologies EJB package javax.ejb is an example

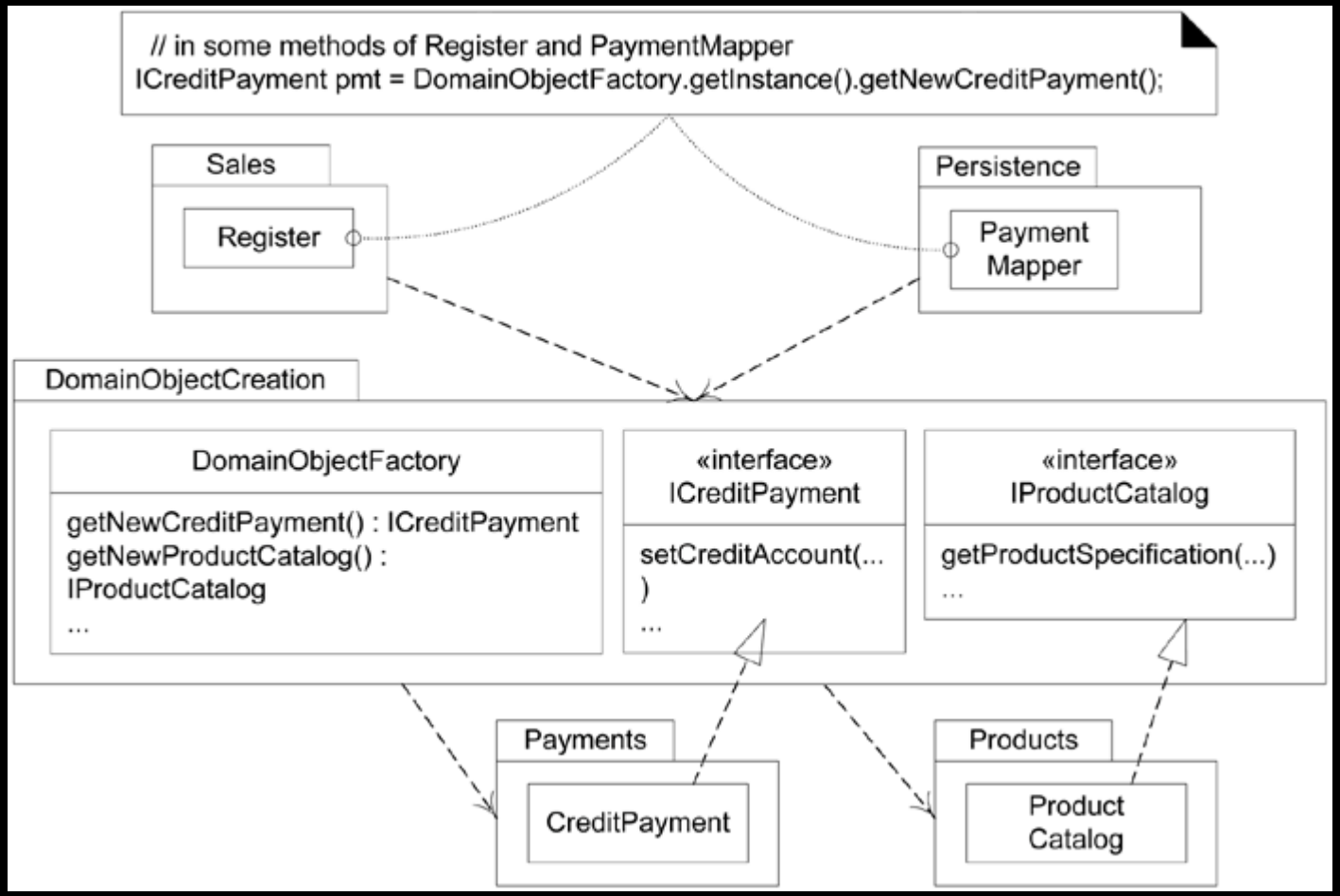
SAP: The Stable Abstractions Principle

- Guideline: Use Factories to Reduce Dependency on Concrete Packages
  - **One way to increase package stability is to reduce its dependency on concrete classes in other packages**



SAP: The Stable Abstractions Principle

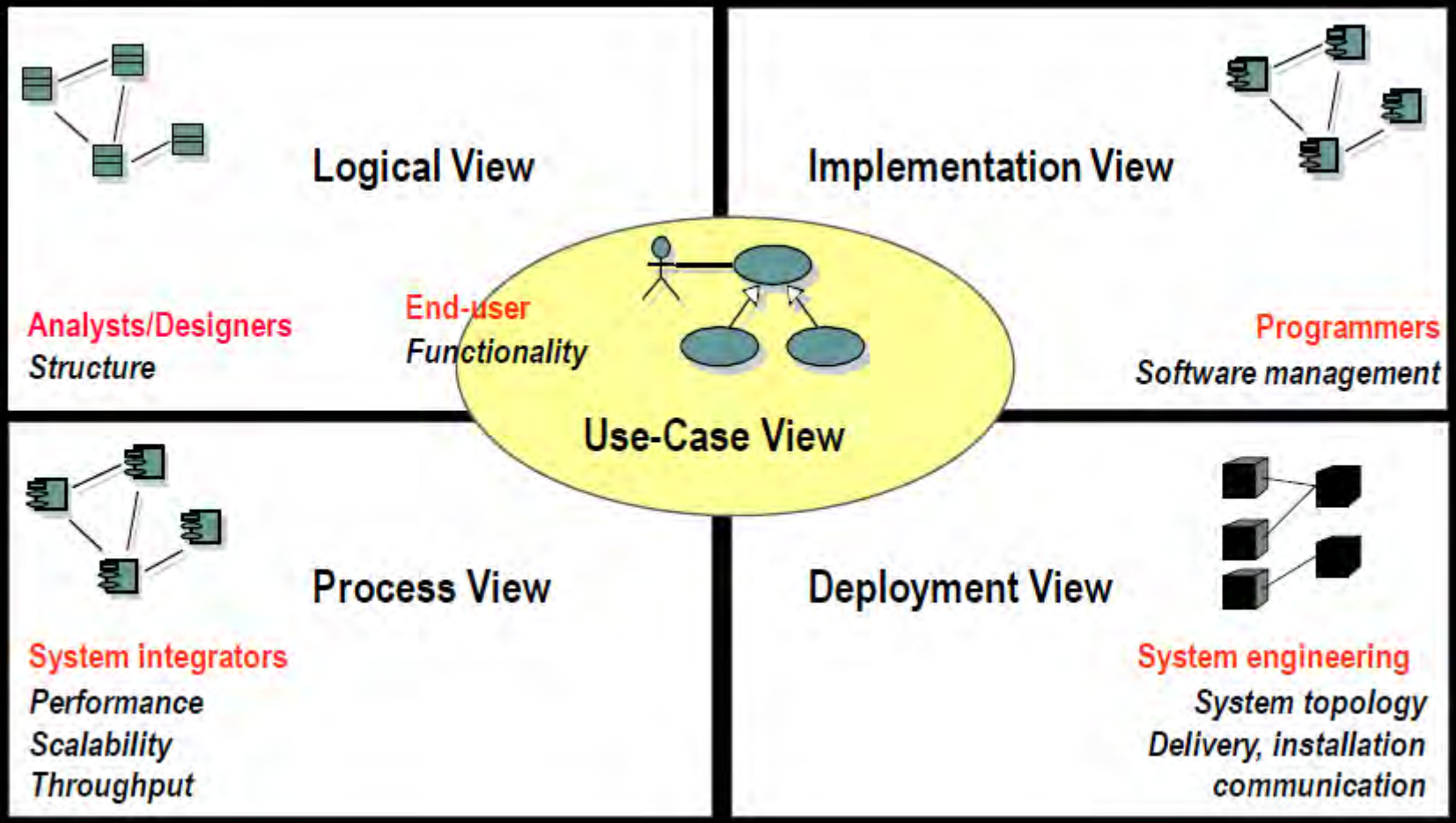
- Reduce coupling to a concrete package by using a factory object



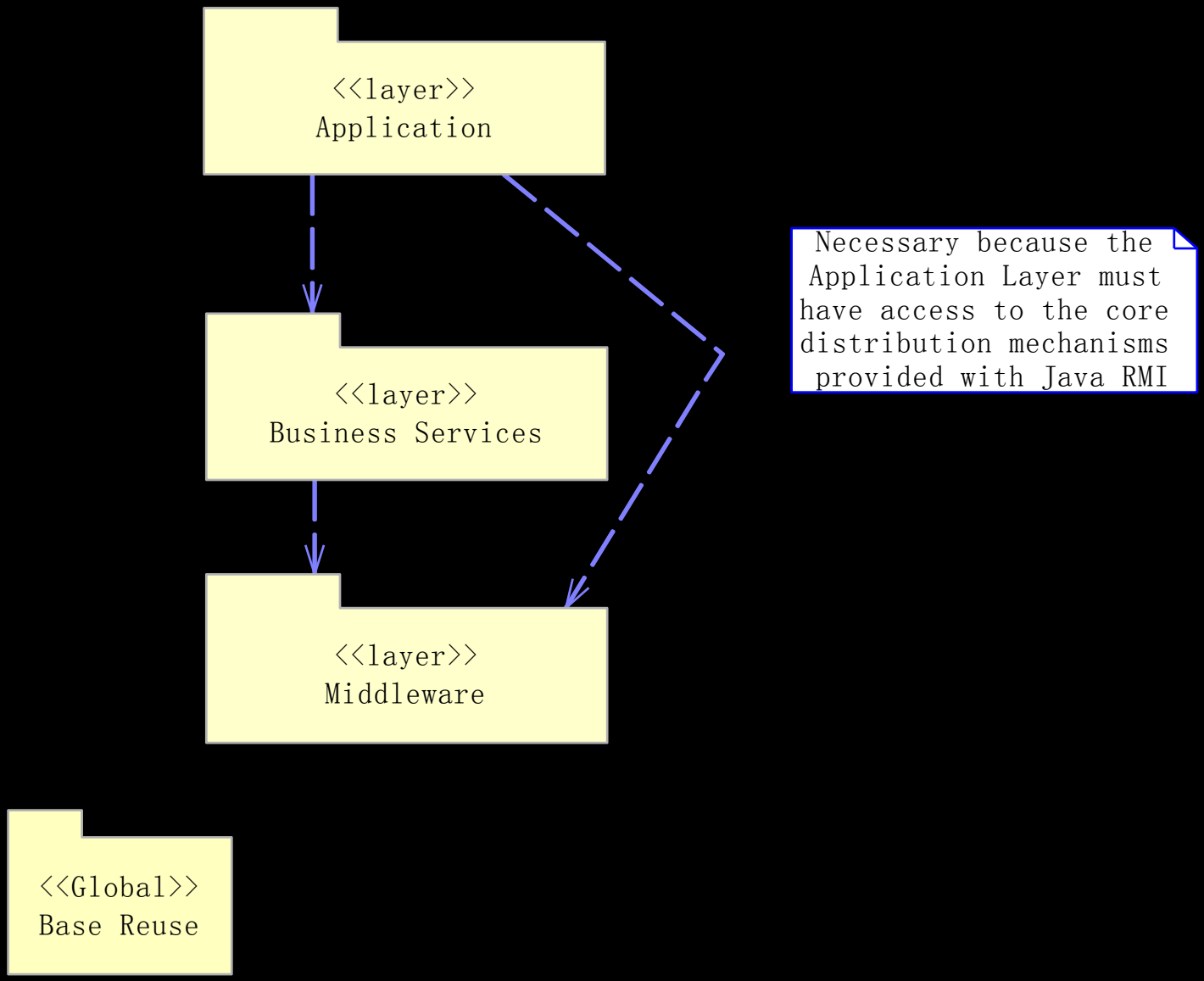
Review: What Is Architecture?

- Software architecture encompasses a set of significant decisions about the organization of a software system.
  - **Selection of the structural elements and their interfaces by which a system is composed**
  - **Behavior as specified in collaborations among those elements**
  - **Composition of these structural and behavioral elements into larger subsystems**
  - **Architectural style that guides this organization**

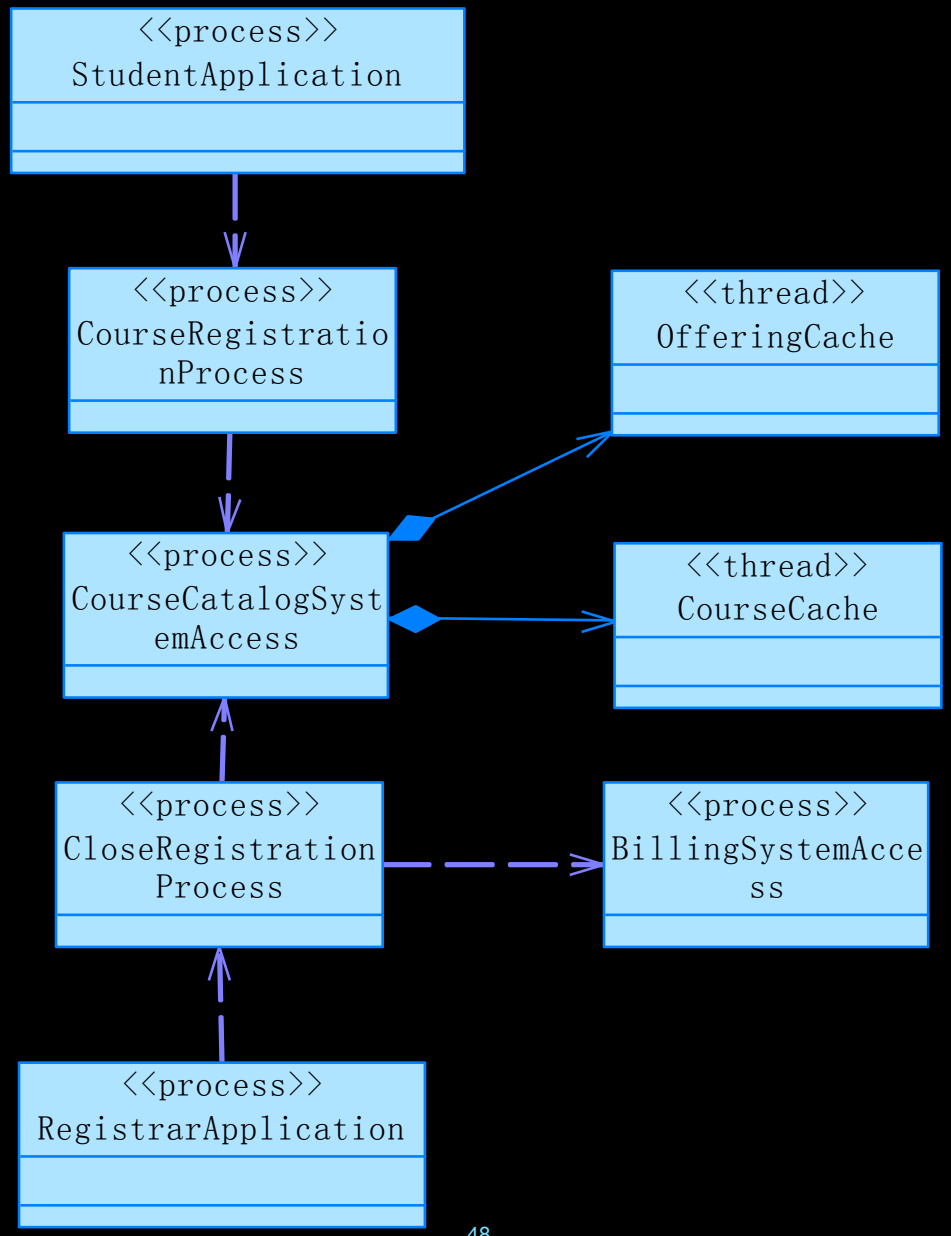
Software Architecture: The “4+1 View” Model



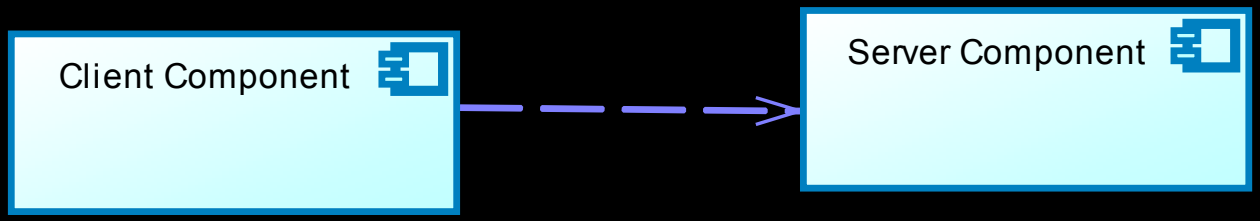
Logic View


<<layer>>
Application
<<layer>>
Business Services
<<layer>>
Middleware
<<Global>>
Base Reuse
Necessary because the Application Layer must have access to the core distribution mechanisms provided with Java RMI

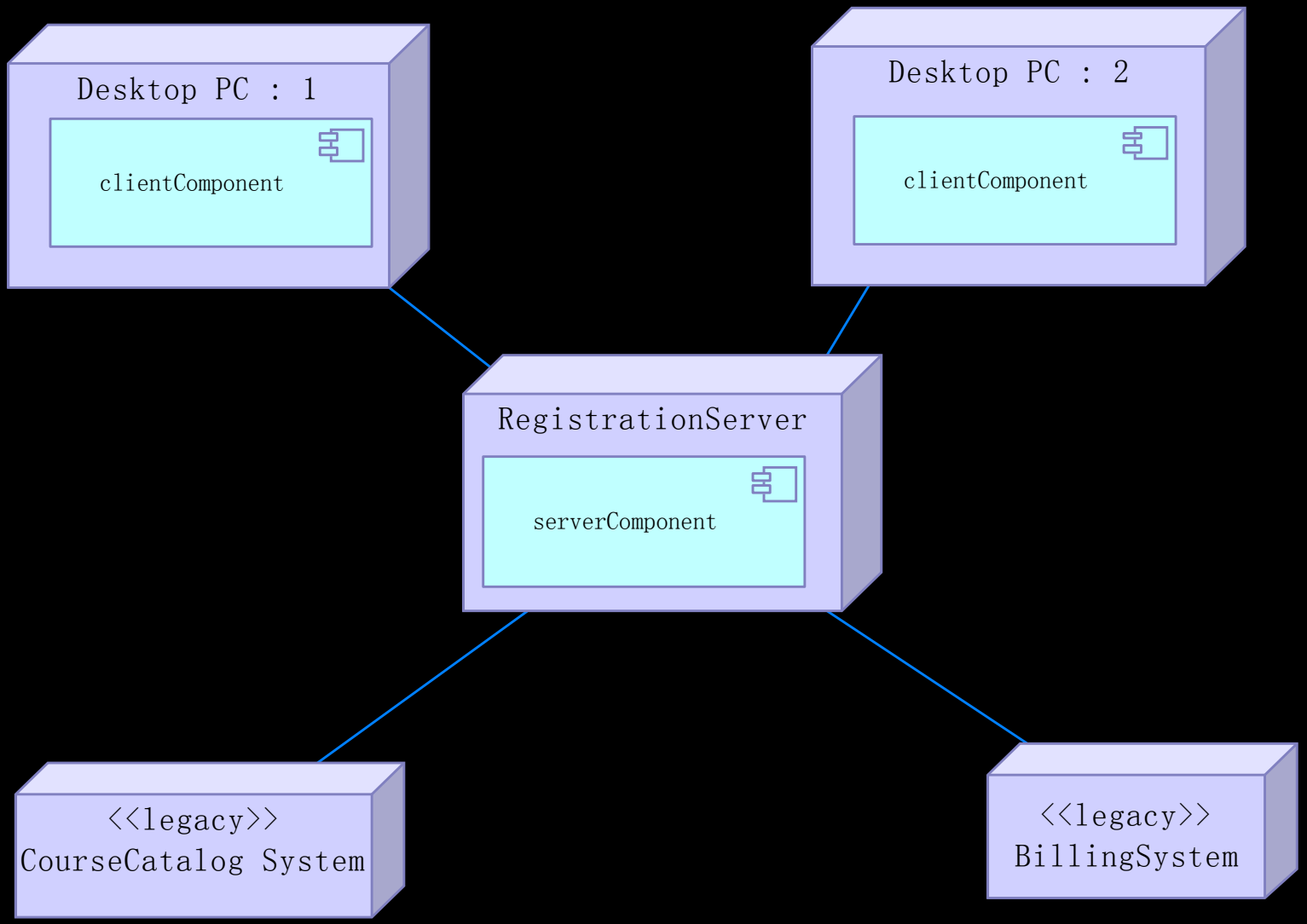
Process View



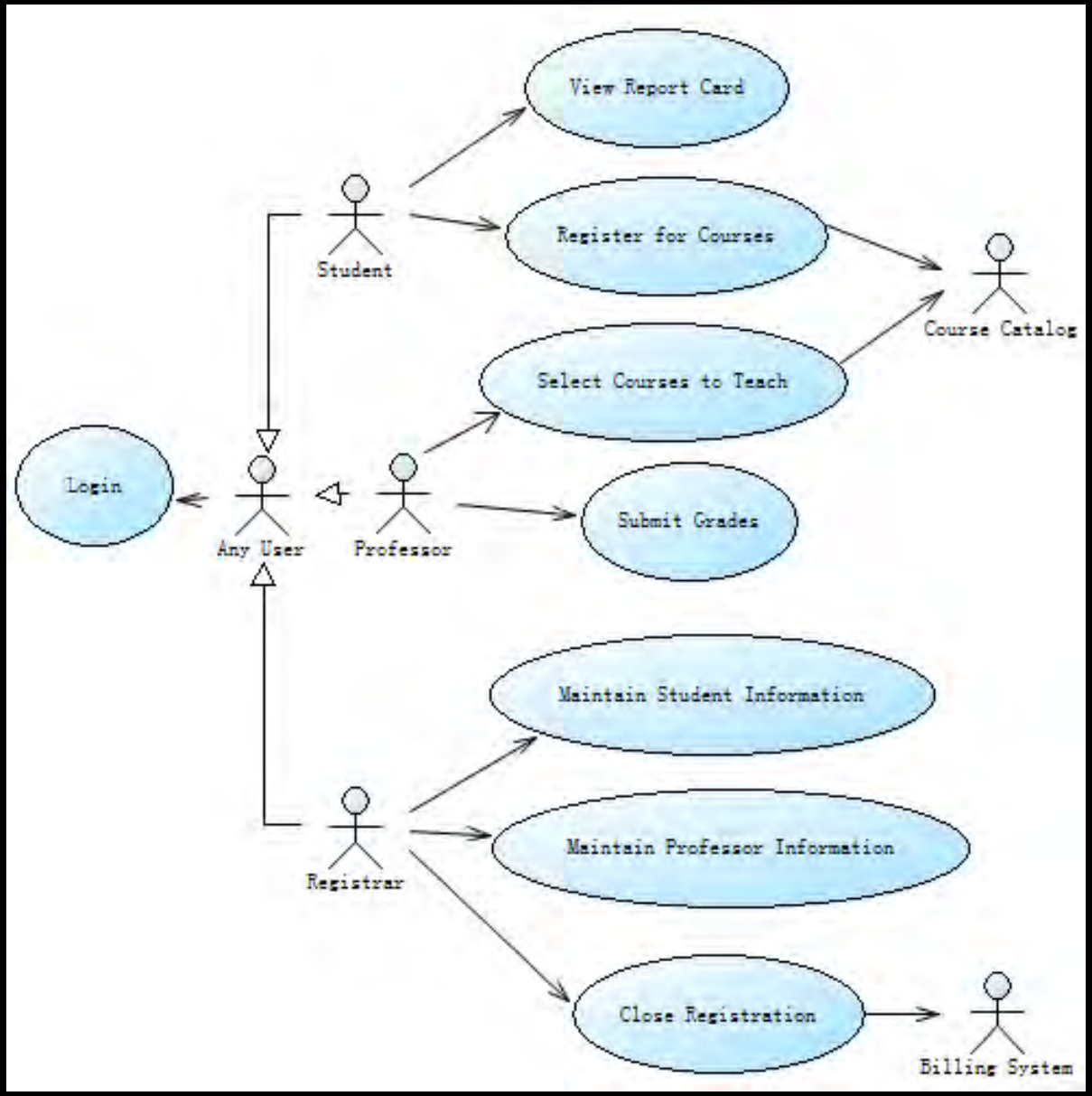
Implementation View (Development View)



Deployment View



Usecase View



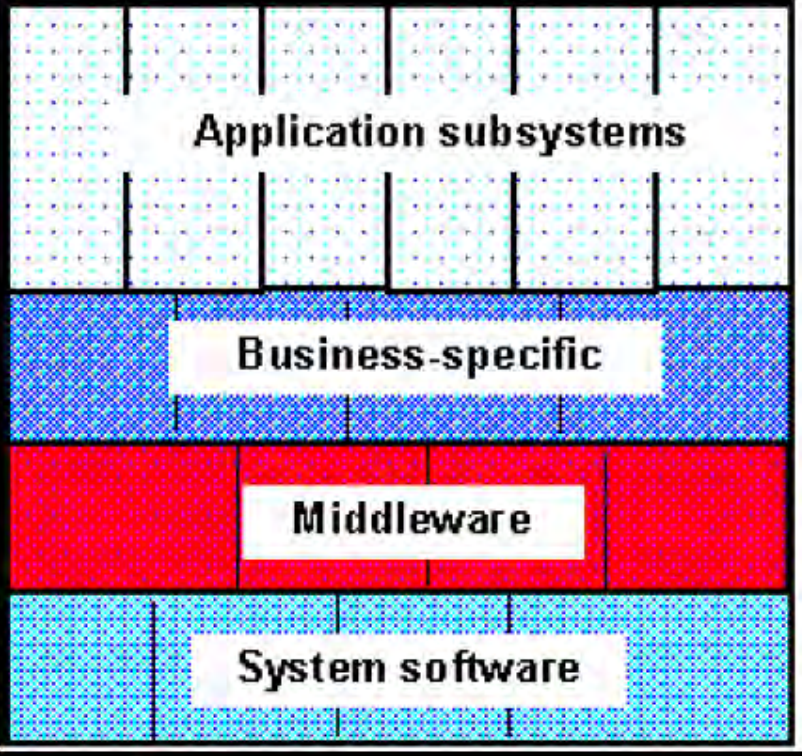
What is An Architectural Pattern?

- An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them
- – Buschman et al, “Pattern-Oriented Software architecture — A System of Patterns”

- Architectural patterns - Common Architectural Styles
  - **Layers**
  - **Model-view-controller (M-V-C)**
  - **Pipes and filters**
  - **Blackboard**

Typical Layering Approach

Specific functionality



General functionality

Distinct application subsystem that make up an application - contains the value adding software developed by the organization.

Business specific - contains a number of reusable sybsystems specific to the type of business.

Middleware - offers subsystems for utility classes and platform-independent services for distributed object computing in heterogeneous environments and so on.

System software - contains the software for the actual infrastructure such as operating systems, interfaces to specific hardware, device drivers and so on.

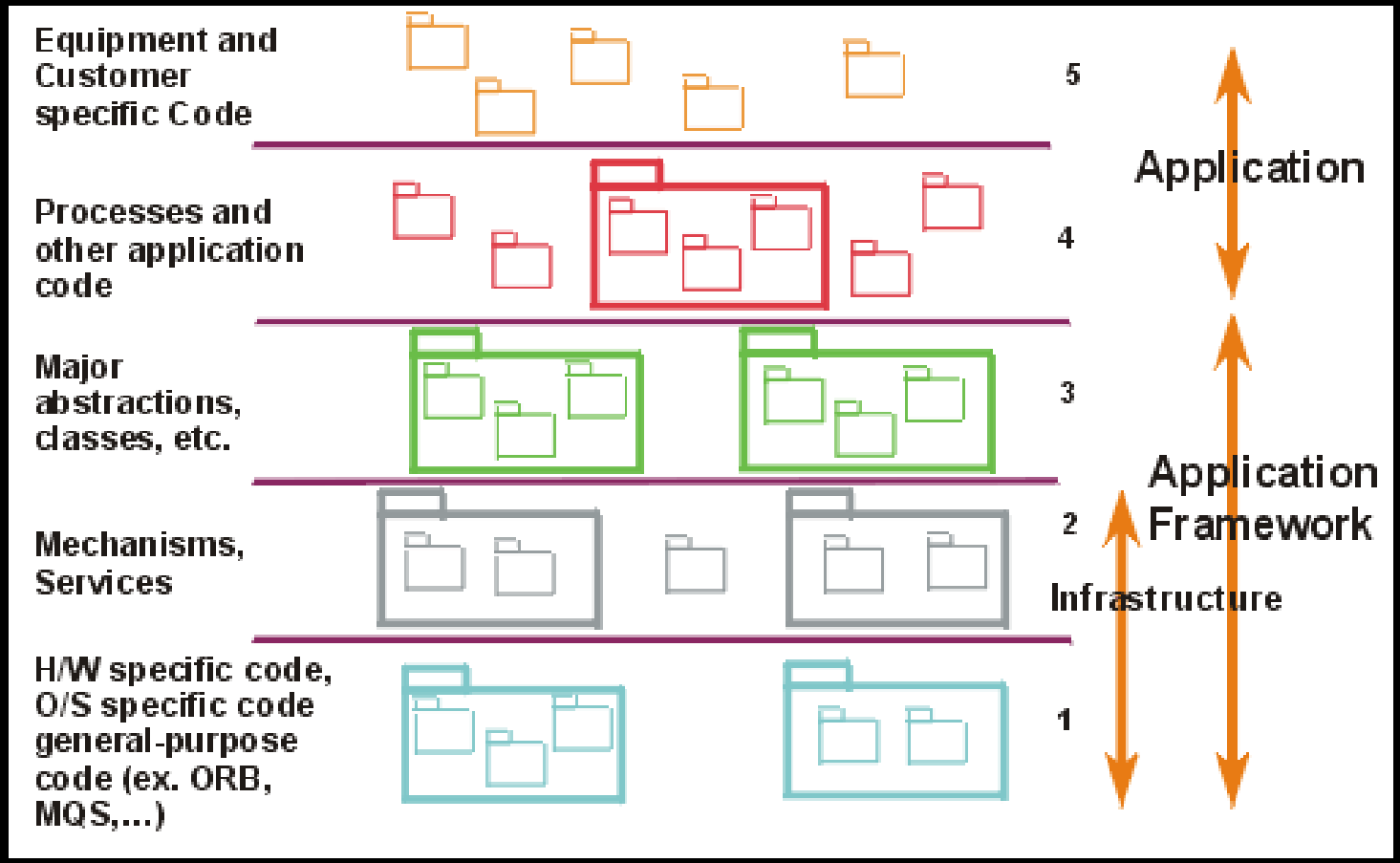
Examples of Architectural Patterns - Layers

- Pattern Name
  - **Layers**
- Context
  - **A large system that requires decomposition.**
- Problem
  - **A system which must handle issues at different levels of abstraction.**
  - **For example: hardware control issues, common services issues and domain-specific issues. It would be extremely undesirable to write vertical components that handle issues at all levels. The same issue would have to be handled (possibly inconsistently) multiple times in different components.**

Examples of Architectural Patterns - Layers

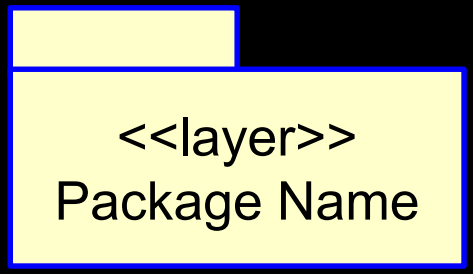
- Forces
  - Parts of the system should be replaceable
  - Changes in components should not ripple
  - Similar responsibilities should be grouped together
  - Size of components – complex components may have to be decomposed
- Solution
  - Structure the systems into groups of components that form layers on top of each other.
  - Make upper layers use services of the layers below only (never above).
  - Try not to use services other than those of the layer directly below (don't skip layers unless intermediate layers would only add pass through components).

Examples of Architectural Patterns - Layers

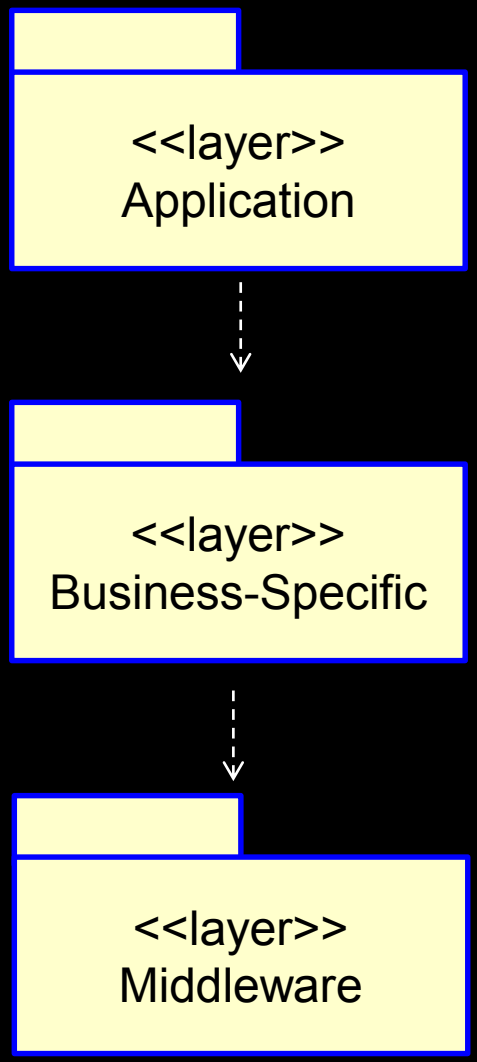


Modeling Architectural Layers

- Architectural layers can be modeled using stereotyped packages
- <<layer>> stereotype

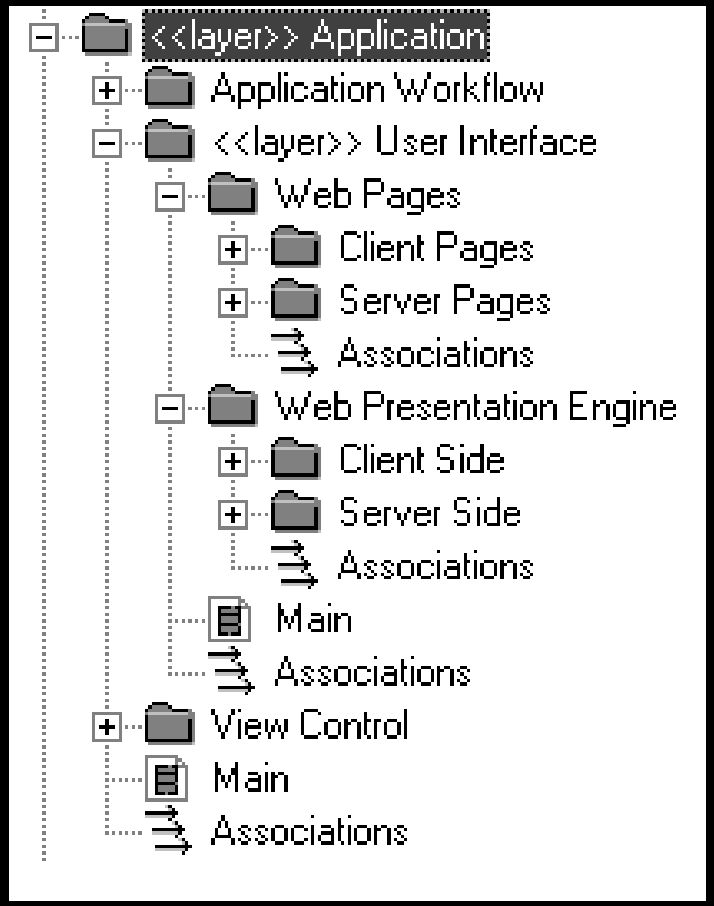


Layer – Reuse driving

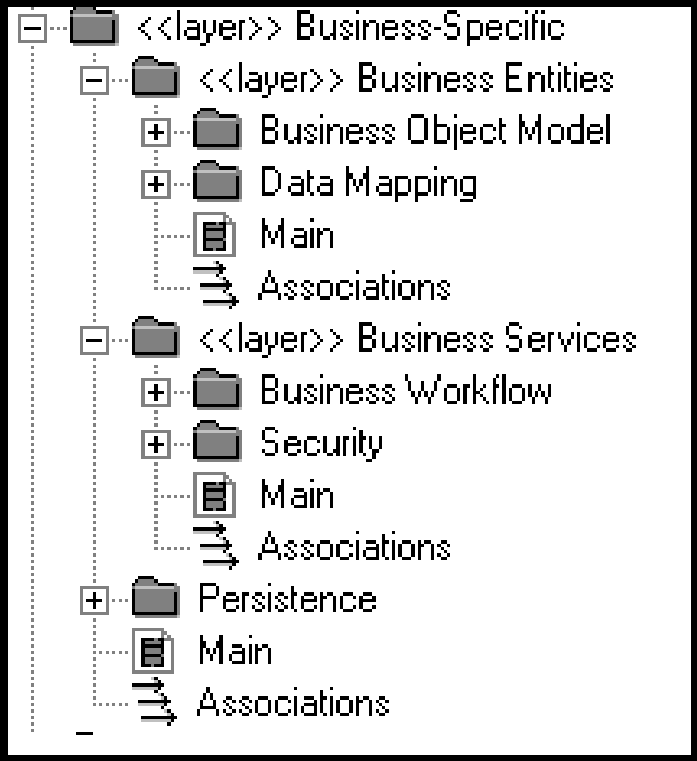


Example

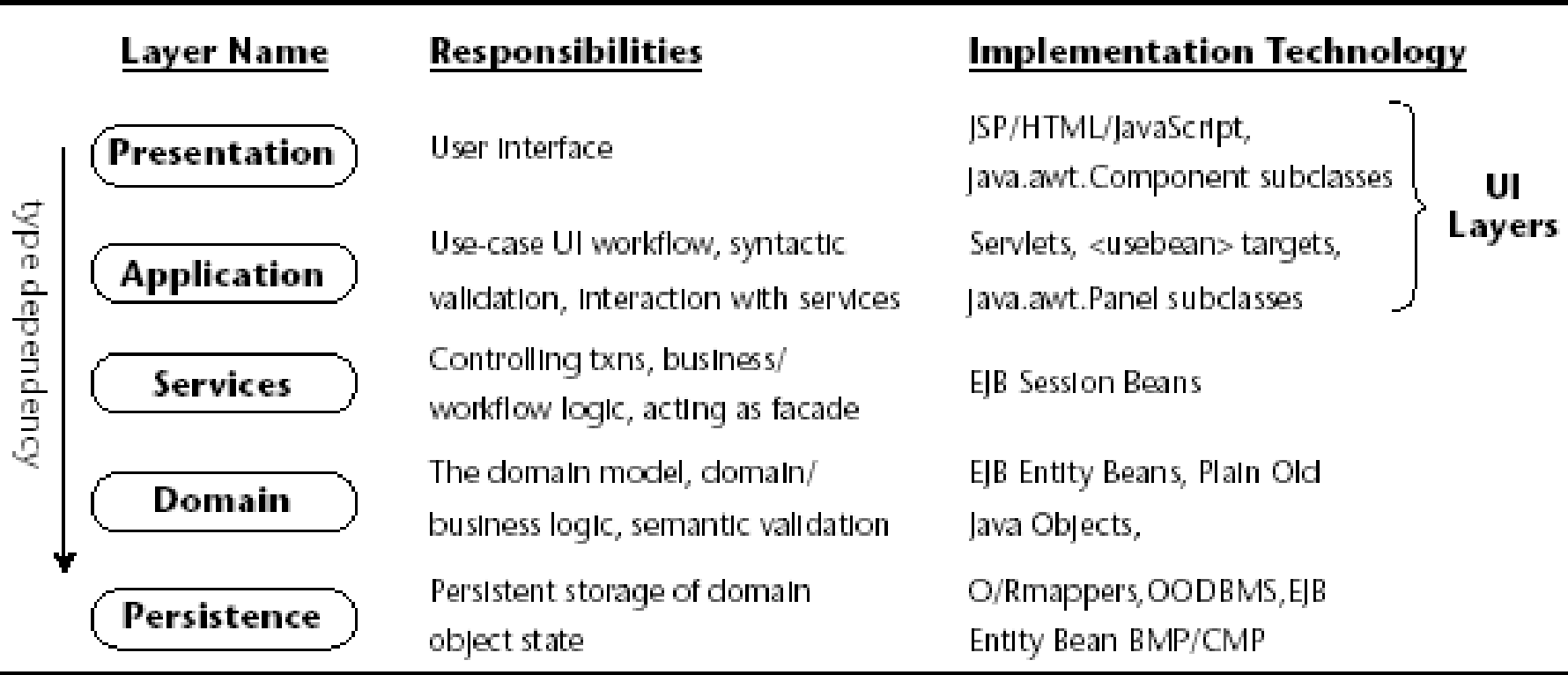
Application Layer



Business Specific Layer



Key Layers


Layer Name
Responsibilities
Implementation Technology
Presentation
User Interface
JSP/HTML/JavaScript, Java.awt.Component subclasses
Application
Use-case UI workflow, syntactic validation, interaction with services
Servlets, <usebean> targets, Java.awt.Panel subclasses
UI Layers
Services
Controlling txns, business/ workflow logic, acting as facade
EJB Session Beans
Domain
The domain model, domain/ business logic, semantic validation
EJB Entity Beans, Plain Old Java Objects,
Persistence
Persistent storage of domain object state
O/Rmappers, OODBMS, EJB Entity Bean BMP/CMP
type dependency

Upward Collaboration with Observer

- When the lower Application or Domain layer needs to communicate upward with the Presentation layer, it is usually via the Observer pattern
  - **UI objects in the higher Presentation layer implement an interface such as PropertyListener or AlarmListener, and are subscribers or listeners to events (such as property or alarm events) coming from objects in the lower layers.**
  - **The lower layer objects are directly sending messages to the upper layer UI objects, but the coupling is only to the objects viewed as things that implement an interface such as PropertyListener, not viewed as specific GUI windows.**

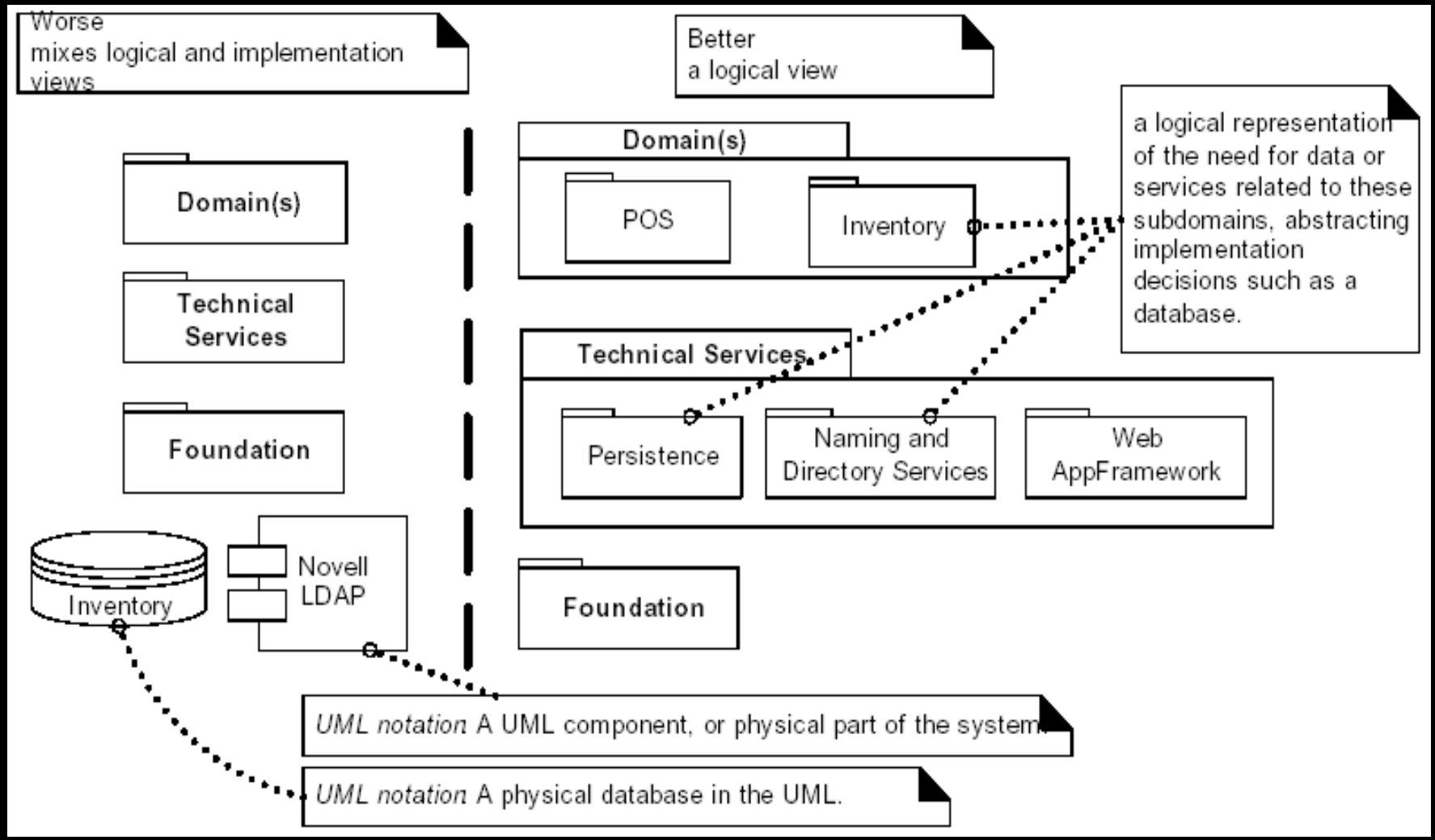
Comments on typical coupling between layers

- All higher layers have dependencies on the Technical Services and Foundations layer
  - **For example, in Java all layers depend on java.util package elements**
- It is primarily the Domain layer that has dependency on the Business Infrastructure layer
- Presentation layer makes calls on the Application layer, which makes service calls on the Domain layer;
  - **the Presentation layer does not call on the Domain, unless there is no Application layer.**

Comments on typical coupling between layers

- If it is a single-process "desktop" application
  - **software objects in the Domain layer are directly visible to, or passed between, Presentation, Application, and to a lesser extent, Technical Services.**
- If it is a distributed system
  - **then serializable replicates (also known as data holder or value objects) of objects in the Domain layer are usually passed to a Presentation layer.**
  - **In this case, the Domain layer is deployed on a server computer, and client nodes get copies of server data.**

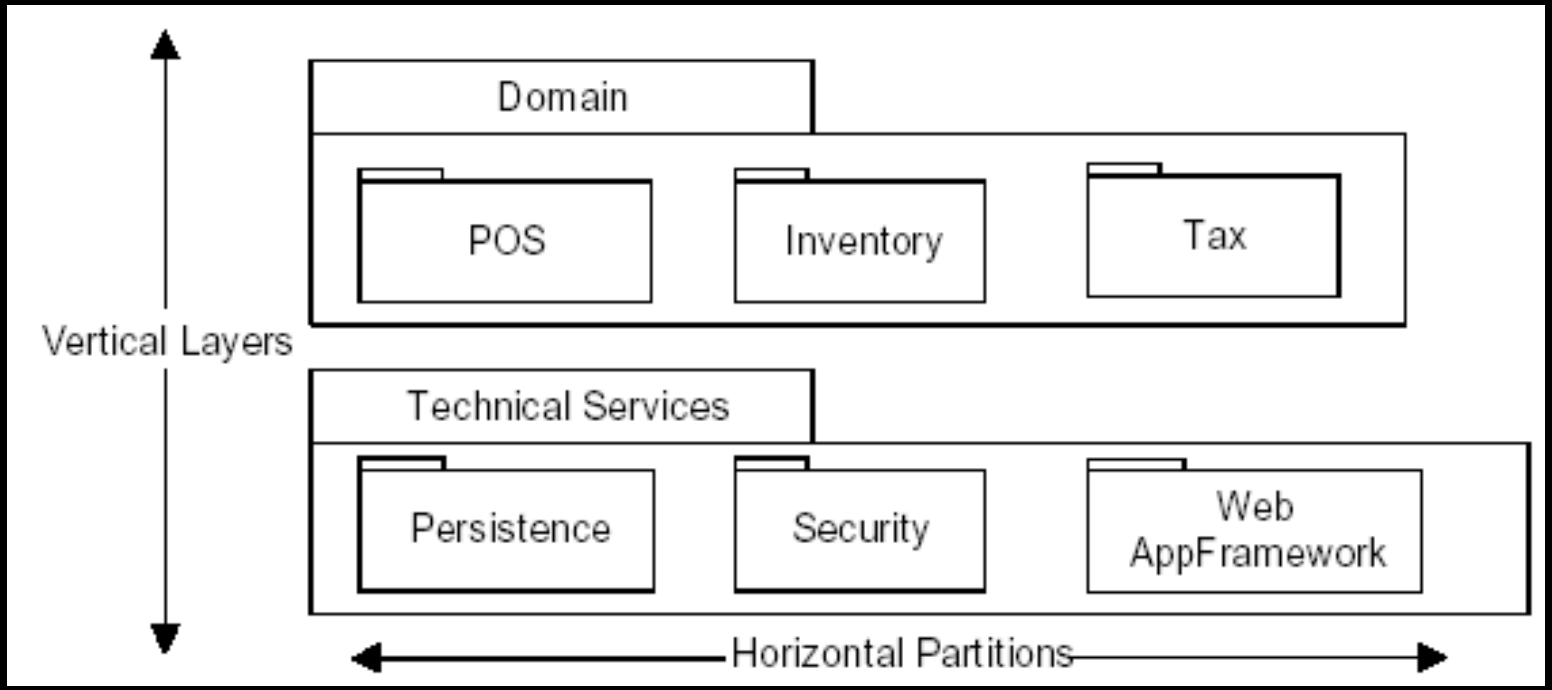
Mixing views of the architecture



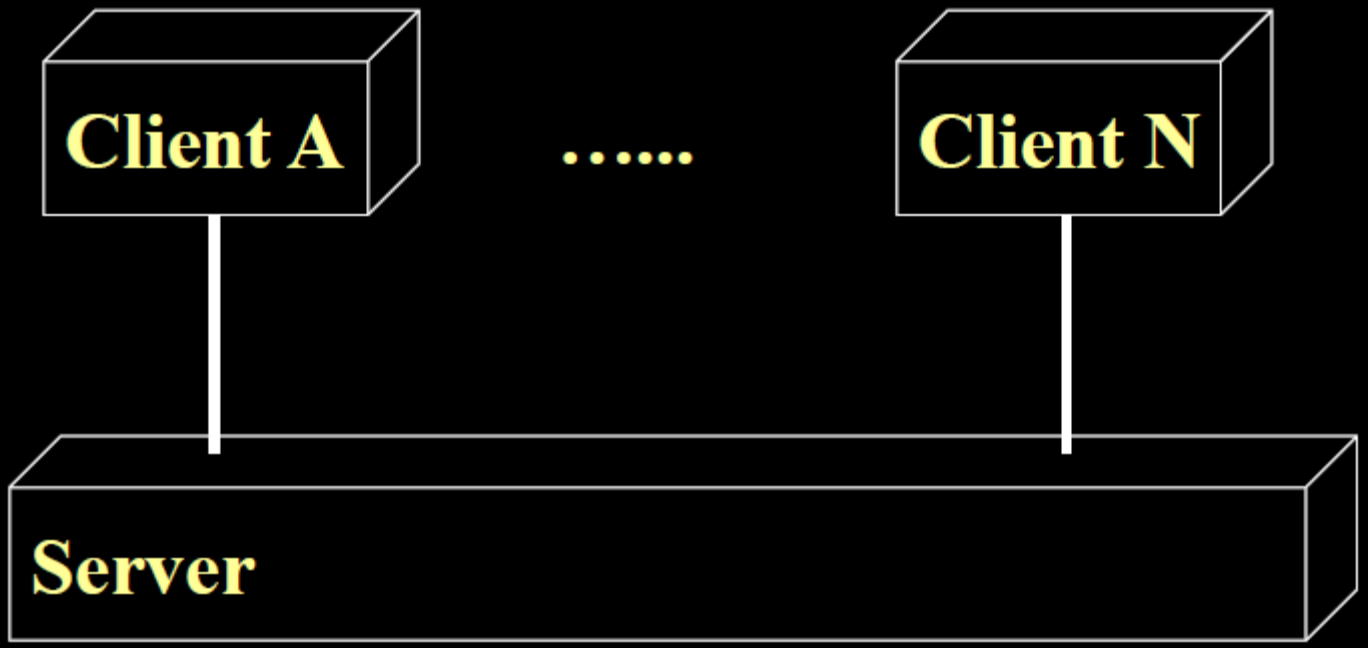
Terminology: Tiers, Layers, and Partitions

- The original notion of a *tier* in architecture was a logical layer, not a physical node, but the word has become widely used to mean a physical processing node (or cluster of nodes)
  - **such as the "client tier" (the client computer).**
- The *layers* of an architecture are said to represent the vertical slices
- The *partitions* represent a horizontal division of relatively parallel subsystems of a layer.

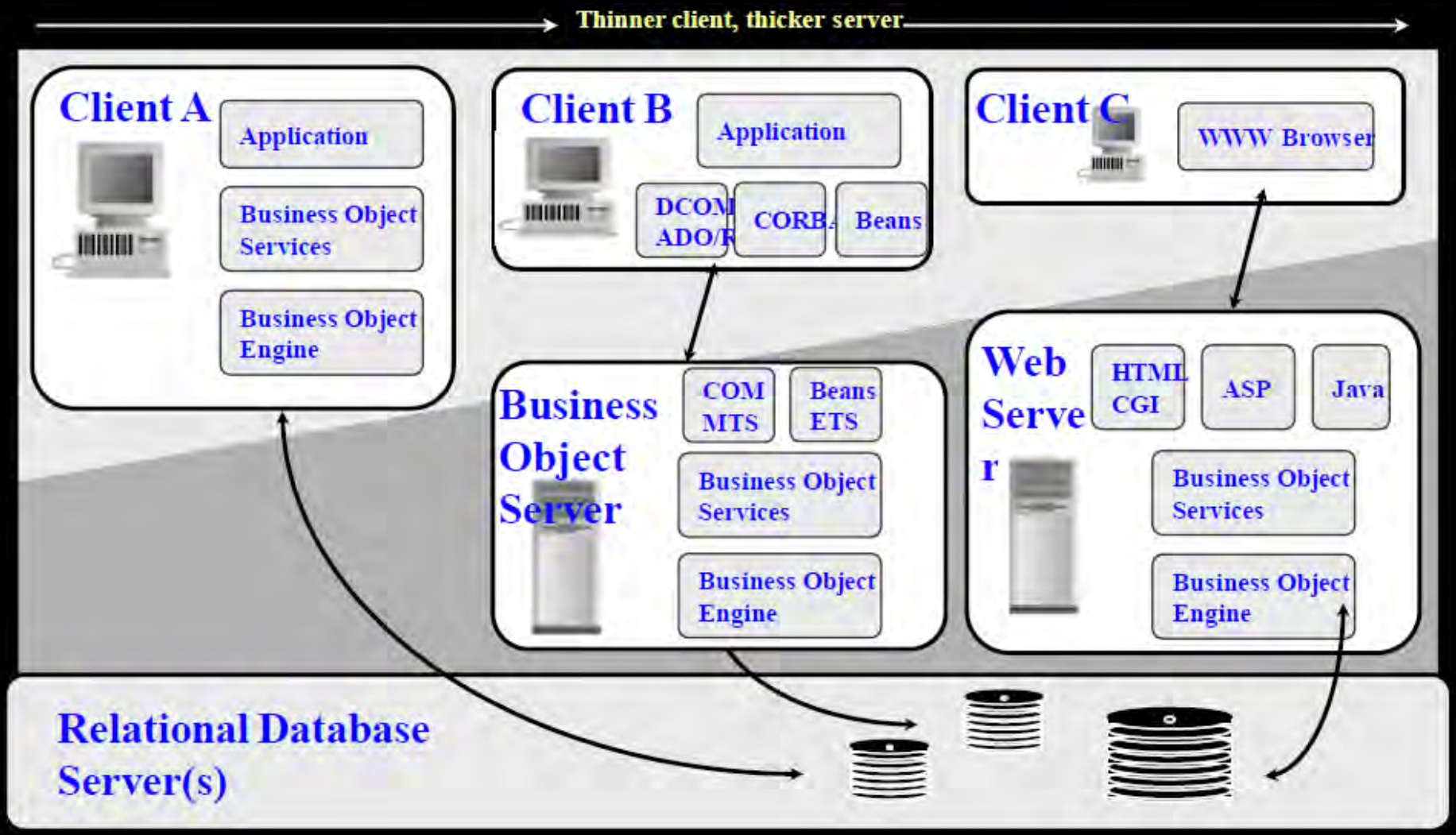
Layers and partitions



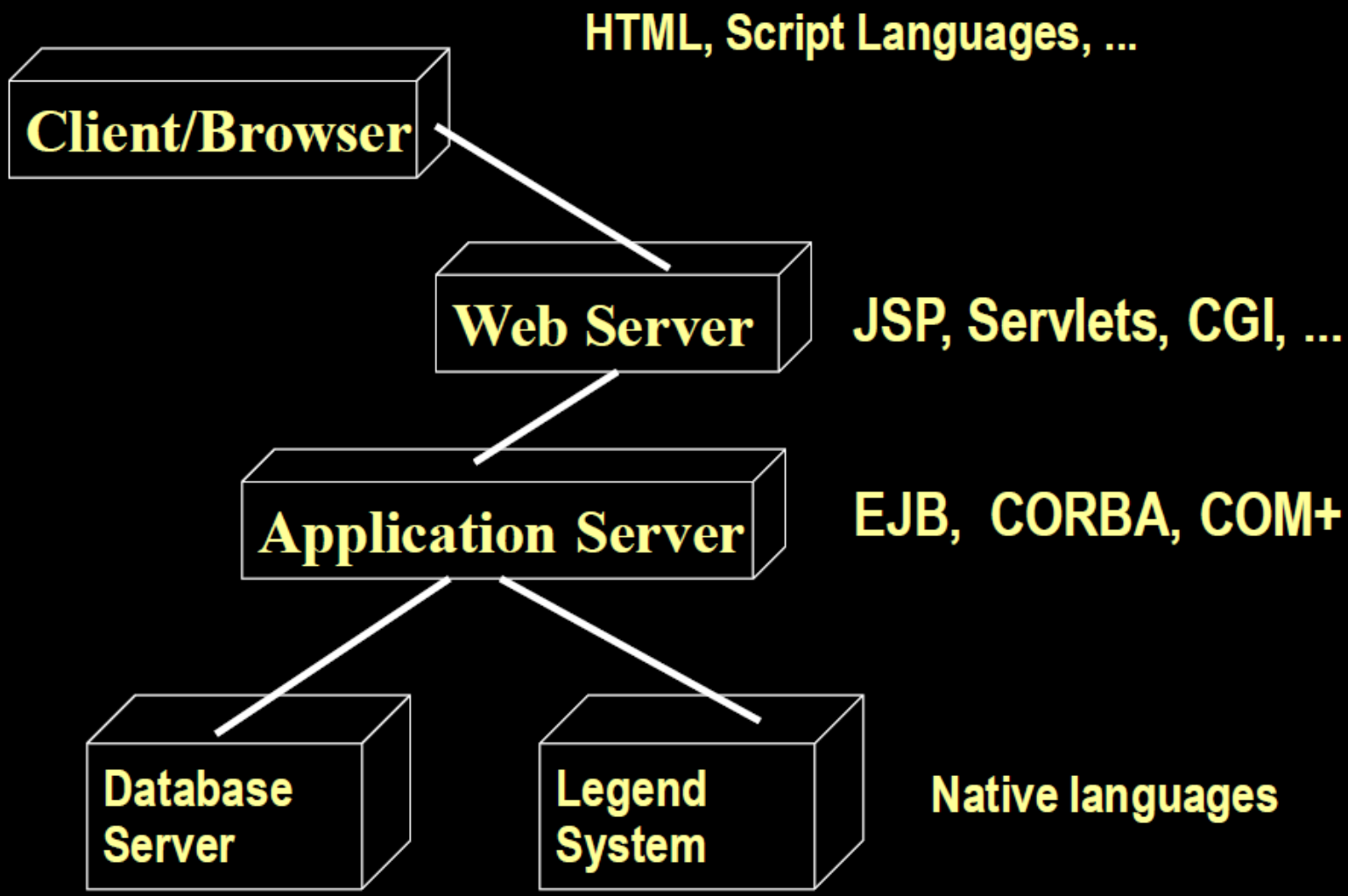
N-Tier (C/S)



N-Tier (3 tier)


Thinner client, thicker server
Client A
Application
Business Object Services
Business Object Engine
Client B
Application
DCOM ADO/R
CORBA
Beans
Client C
WWW Browser
Business Object Server
COM MTS
Beans ETS
Business Object Services
Business Object Engine
Web Server
HTML CGI
ASP
Java
Business Object Services
Business Object Engine
Relational Database Server(s)

N-Tier (n-tier)

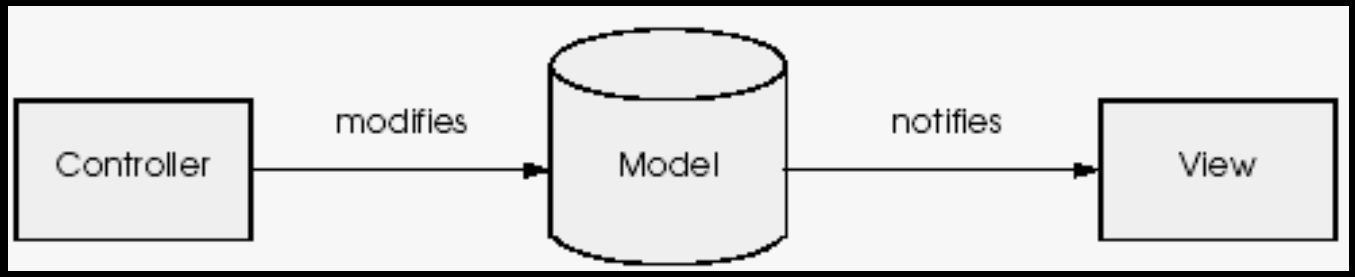

HTML, Script Languages, ...
Client/Browser
Web Server
JSP, Servlets, CGI, ...
Application Server
EJB, CORBA, COM+
Database Server
Legend System
Native languages

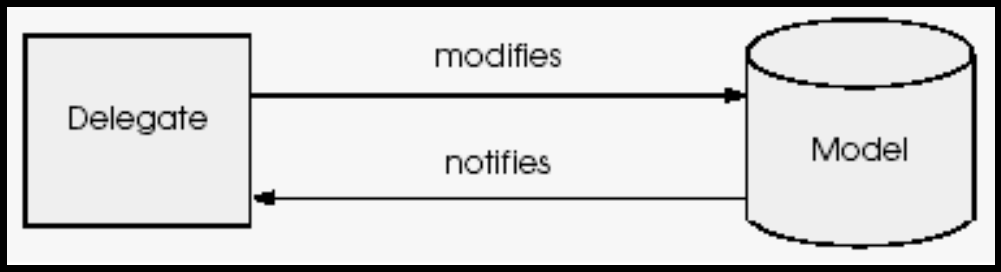
Architectural Pattern - MVC

- Name
  - **MVC (Model-View-Controller)**
- Context and forces
  - **we have a data model and several representations of the data**
    - **We want to modularize the system**
    - **Data representation must be kept up to date**
- Problem
  - **how to modularize the system**
- Solution
  - **the model holds the data (and does data modification), the view represents the data, the controller handles user input**

Model-View-Controller Architecture

- The *model-view-controller* architecture (MVC) separates application data (contained in the *model*) from graphical presentation components (the *view*) and input-processing logic (the *controller*).





Examples of Architectural Patterns - Blackboard

- Pattern Name
  - **Blackboard**
- Context
  - **A domain in which no closed (algorithmic) approach to solving a problem is known or feasible. Examples are AI systems, voice recognition, and surveillance systems.**
- Problem
  - **Multiple problem-solving agents (knowledge agents) must cooperate to solve a problem that cannot be solved by any of the individual agents. The results of the work of the individual agents must be accessible to all the other agents so they can evaluate whether they can contribute to finding a solution and post results of their work.**

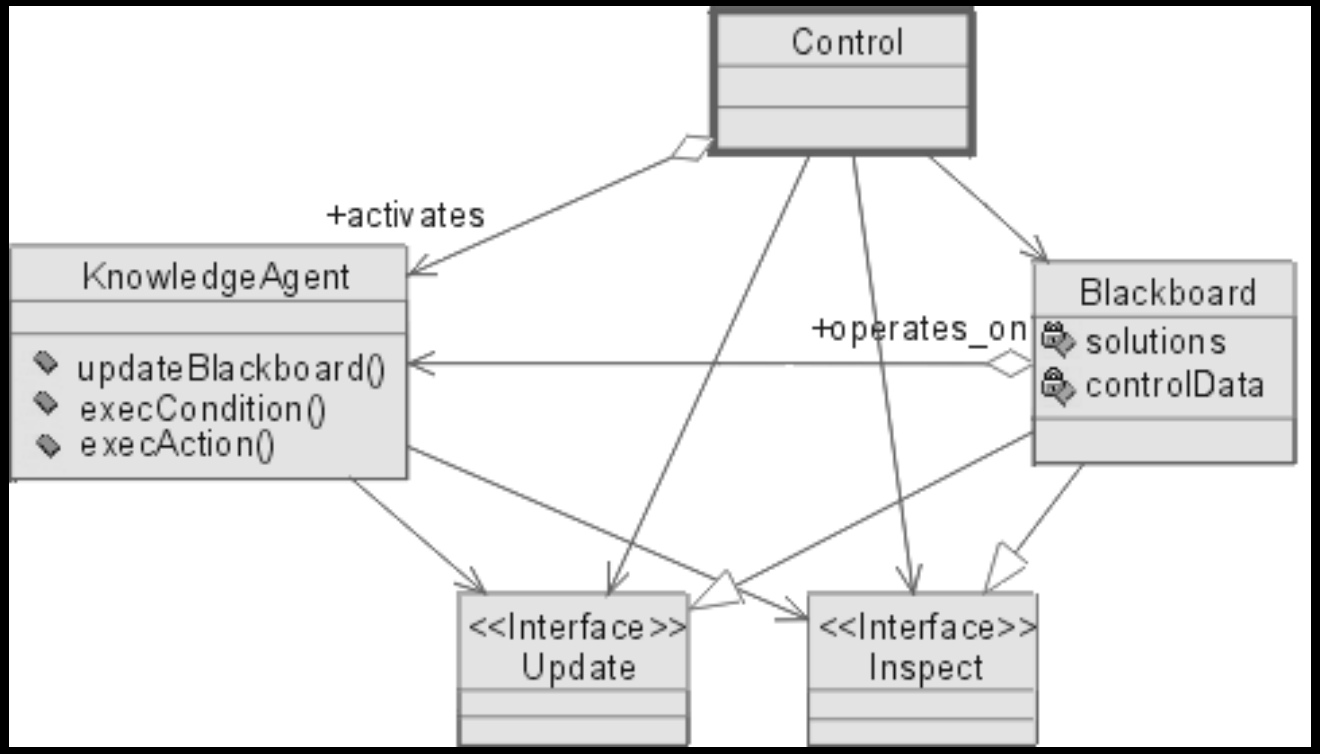
Examples of Architectural Patterns - Blackboard

- Forces
  - **Sequence** in which knowledge agents can contribute to solving the problem is **not deterministic** and may depend on problem solving strategies.
  - Input from **different** agents (results or partial solutions) may have **different** representations.
  - Agents do **not** know of each other's existence directly but can evaluate each other's posted contributions

Examples of Architectural Patterns - Blackboard

- Solution
  - A number of Knowledge Agents have access to a shared data store called the Blackboard.
  - The blackboard provides an interface to inspect and update its content.
  - The Control module/object activates the agents following some strategy.
  - Upon activation an agent inspects that blackboard to see if it can contribute to solving the problem.
  - If the agent determines that it can contribute, the control object can allow the agents to put its partial (or final) solution on the board.

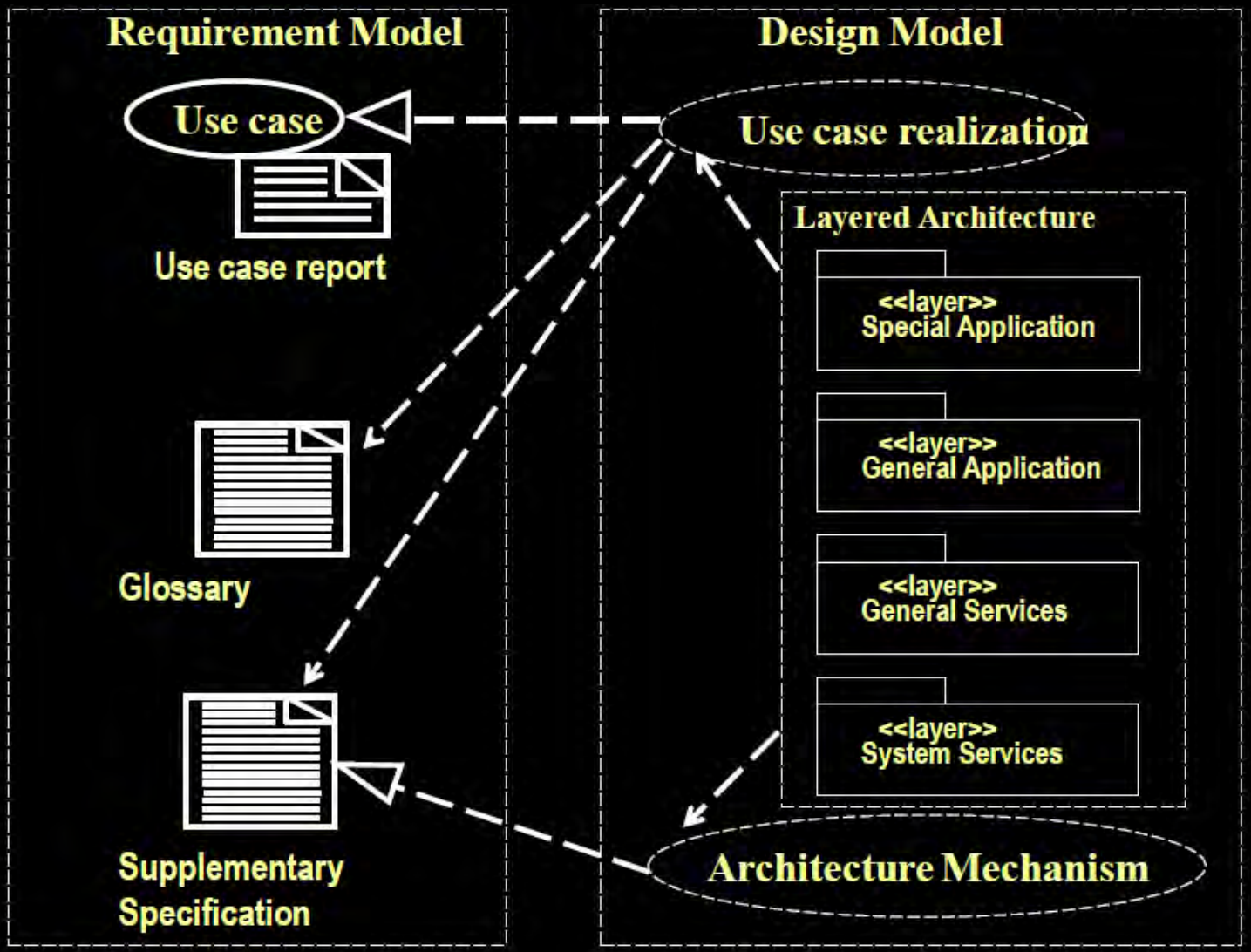
Examples of Architectural Patterns - Blackboard



Resolution of Architectural Factors

- Recording Architectural Alternatives, Decisions, and Motivation - technical memos
- Example of Technical Memo

Design Model Overview



Technical Memo

- All architectural methods recommend keeping a record of alternative solutions, decisions, influential factors, and motivations for the noteworthy issues and decisions.
- Such records have been called technical memos, issue cards, and architectural approach documents, with varying degrees of formality and sophistication.
  - **In some methods, these memos are the basis for yet another step of review and refinement.**
- In the UP, the memos should be recorded in the SAD.
- An important aspect of the technical memo is the motivation or rationale.
- Explaining the rationale of rejecting the alternatives is important.

Example of Technical Memo

Technical Memo Issue:
Reliability—Recovery from Remote Service Failure

Solution Summary: Location transparency using service lookup, failover from remote to local, and local service partial replication.

Factors

- Robust recovery from remote service failure (e.g., tax calculator, inventory)
- Robust recovery from remote product (e.g., descriptions and prices) database failure

Solution

Achieve protected variation with respect to location of services using an Adapter created in a Services-Factory. Where possible, offer local implementations of remote services, usually with simplified or constrained behavior. For example, the local tax calculator will use constant tax rates. The local product information database will be a small cache of the most common products. Inventory updates will be stored and forwarded at reconnection.

See also the *Adaptability—Third-Party Services* technical memo for the adaptability aspects of this solutions, because remote service implementations will vary at each installation.

To satisfy the quality scenarios of reconnection with the remote services ASAP, use smart Proxy objects for the services, that on each service call test for remote service reactivation, and redirect to them when possible.

Motivation

Retailers really don't want to stop making sales! Therefore, if the NextGen POS offers this level of reliability and recovery, it will be a very attractive product, as none of our competitors provide this capability. The small product cache is motivated by very limited client-side resources. The real third-party tax calculator is not replicated on the client primarily because of the higher licensing costs, and configuration efforts (as each calculator installation requires almost weekly adjustments). This design also supports the evolution point of future customers willing and able to permanently replicate services such as the tax calculator to each client terminal.

Unresolved Issues

none

Alternatives Considered

A "gold level" quality of service agreement with remote credit authorization services to improve reliability. It was available, but much too expensive.

Architectural Information in the UP Artifacts

- The architectural decisions are recorded in the SAD(Software Architecture Document).
  - **This includes the technical memos and descriptions of the architectural views.**

Sample Structure of a SAD

- Architectural Representation
  - **Summary of how the architecture will be described in this document, such as using by technical memos and the architectural views. This is useful for someone unfamiliar with the idea of technical memos or views. Note that not all views are necessary.**
- Architectural Factors and Decisions
  - **Reference to the Supplementary Specification to view the Factor Table. Also, the set of technical memos the summarize the decisions.**
- Logical View
  - **UML package diagrams, and class diagrams of major elements. Commentary on the large scale structure and functionality of major components.**
- Process View
  - **UML class and interaction diagrams illustrating the processes and threads of the system. Group this by threads and processes that interact. Comment on how the interprocess communication works**

Sample Structure of a SAD

- Use-Case View
  - **Brief summary of the most architecturally significant use cases. UML interaction diagrams for some architectural significant use-case realizations, or scenarios, with commentary on the diagrams explaining how they illustrate the major architectural elements.**
- Deployment View
  - **UML deployment diagrams showing the nodes and allocation of processes and components. Commentary on the networking.**
- Data View
  - **Overview of the persistent data schema, the schema mapping from objects to persistent data (usually in a relational database), the mechanism of mapping from objects to a database, database stored procedures and triggers.**
  - **A view onto the UP Data Model, visualized with UML class diagrams used to describe a data model.**

Identify Design Elements

- Purpose：
  - To analyze interactions of analysis classes to identify design model elements

1. Identify Classes, Active Classes and Subsystems
2. Identify Subsystem Interfaces
3. Identify and Specify Events
4. Identify and Specify Signals

Identify Design Elements

- In design, analysis classes evolve into a number of different kinds of design elements
  - **analysis classes represent conceptual things which can perform behavior**
  - ***classes***
    - **to represent a set of rather fine-grained responsibilities;**
  - ***subsystems***
    - **to represent a set of coarse-grained responsibilities, perhaps composed of a further set of subsystems, but ultimately a set of classes;**
  - ***active classes***
    - **to represent threads of control in the system;**
  - ***interfaces***
    - **to represent abstract declarations of responsibilities provided by a class or subsystem.**

Identify Design Elements

- In addition, in design we shall also identify
  - ***events***
    - **which are specifications of interesting occurrences in time and space that usually (if they are noteworthy) require some response from the system; and**
  - ***signals***
    - **to represent asynchronous mechanisms used to communicate certain types of events within the system.**

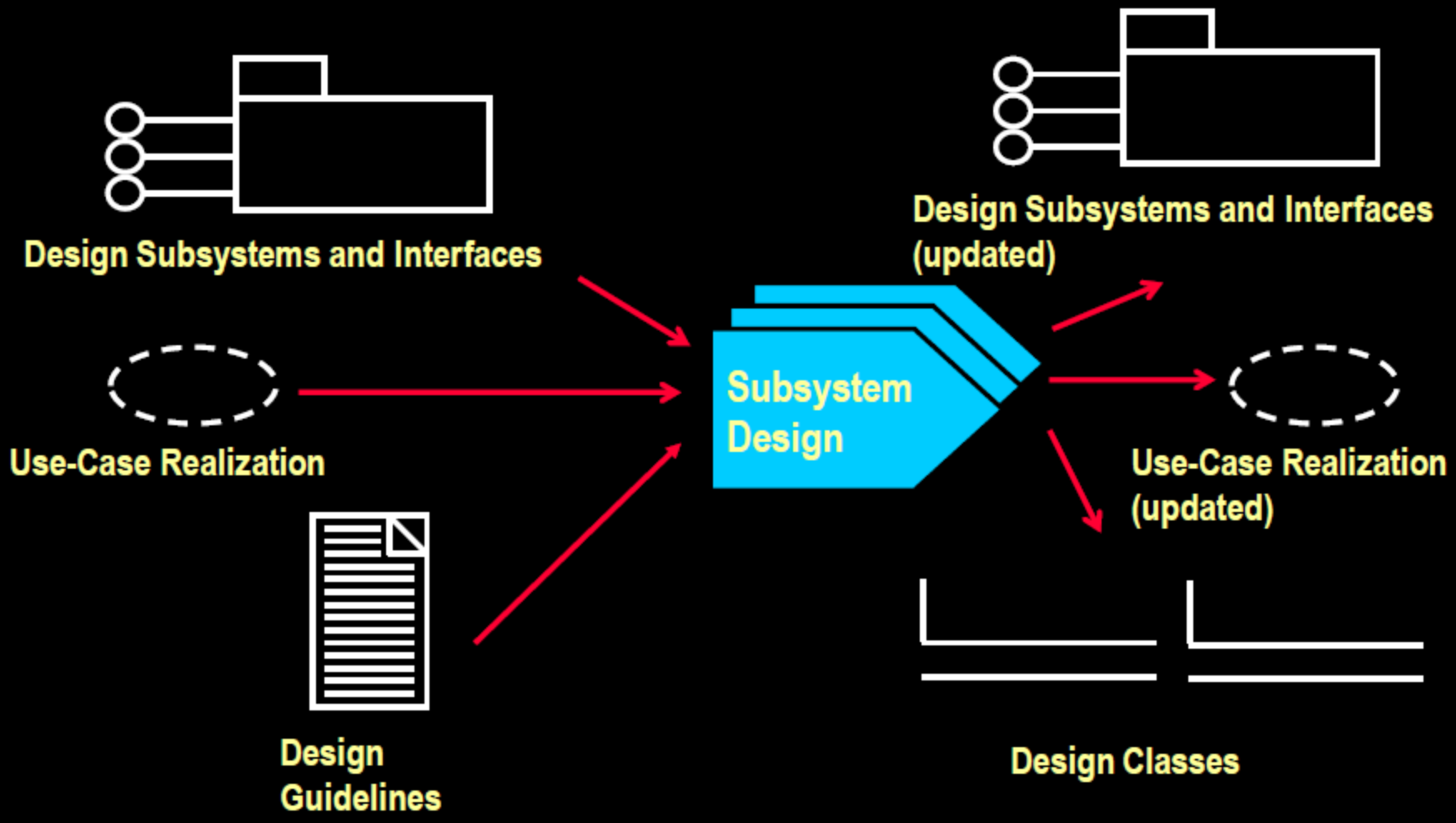
- Events and the Signals that are used to communicate them, allow us to describe the asynchronous triggers of behavior to which the system must respond.

软件工程概论

Introduction to Software Engineering

SUBSYSTEM DESIGN

Subsystem Design Overview



Subsystem Design - Purpose

- To define the behaviors specified in the subsystem's interfaces in terms of collaborations of contained classes
- To document the internal structure of the subsystem
- To define realizations between the subsystem's interfaces and contained classes
- To determine the dependencies upon other subsystems

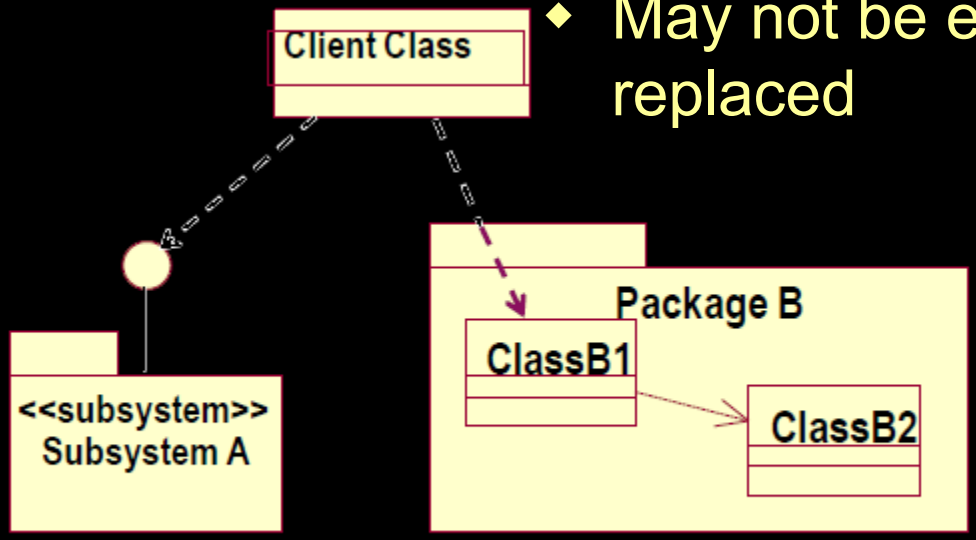
Packages versus Subsystems

Subsystems

- Provide behavior
- Completely encapsulate their contents
- Are easily replaced

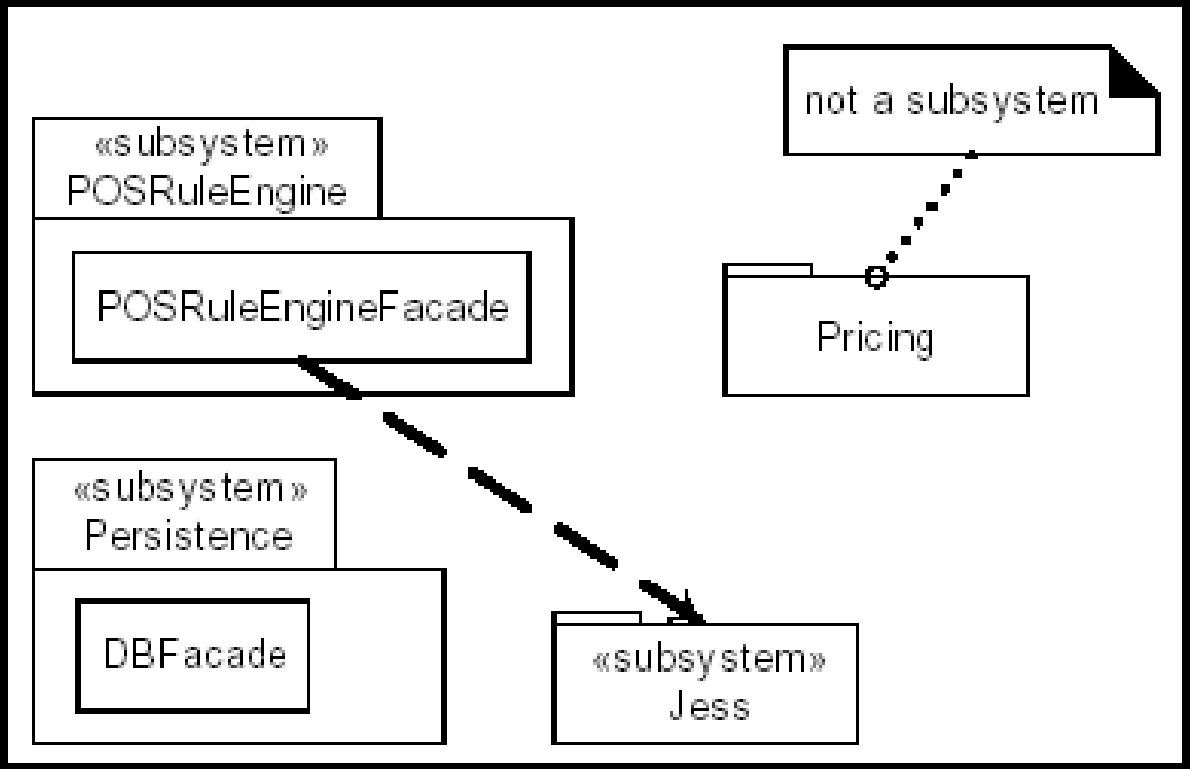
Packages

- Don't provide behavior
- Don't completely encapsulate their contents
- May not be easily replaced

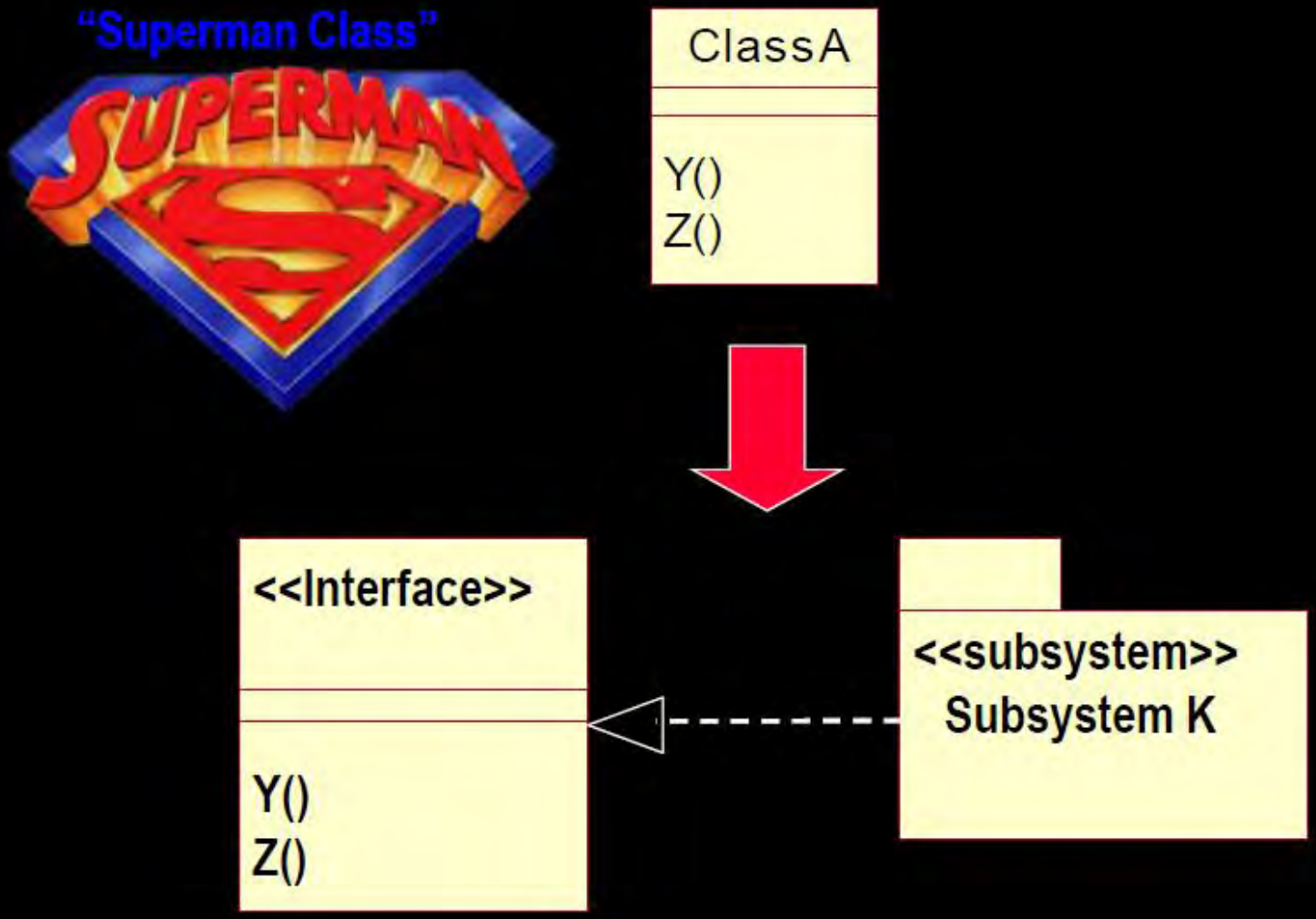


Encapsulation is the key!

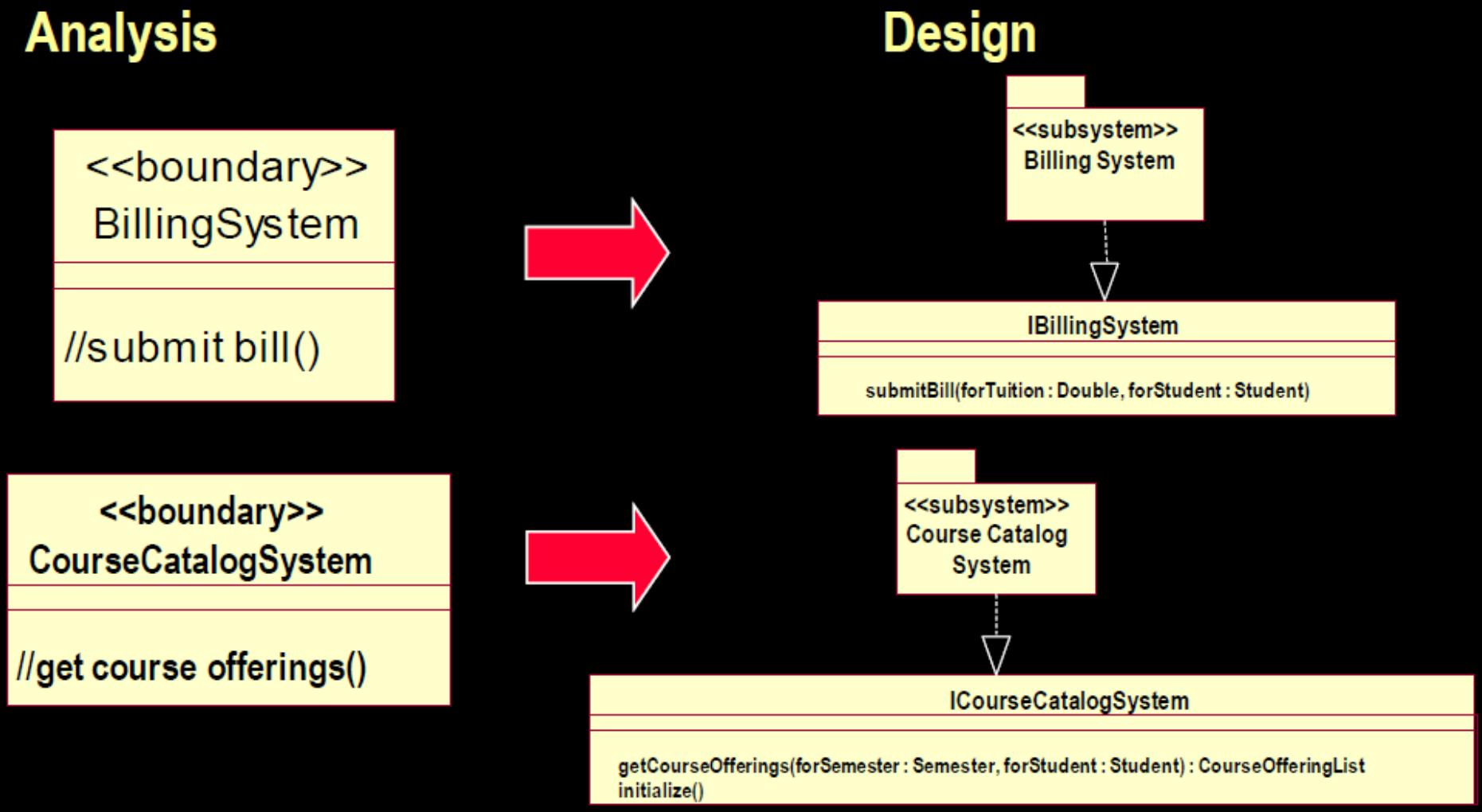
Subsystem example



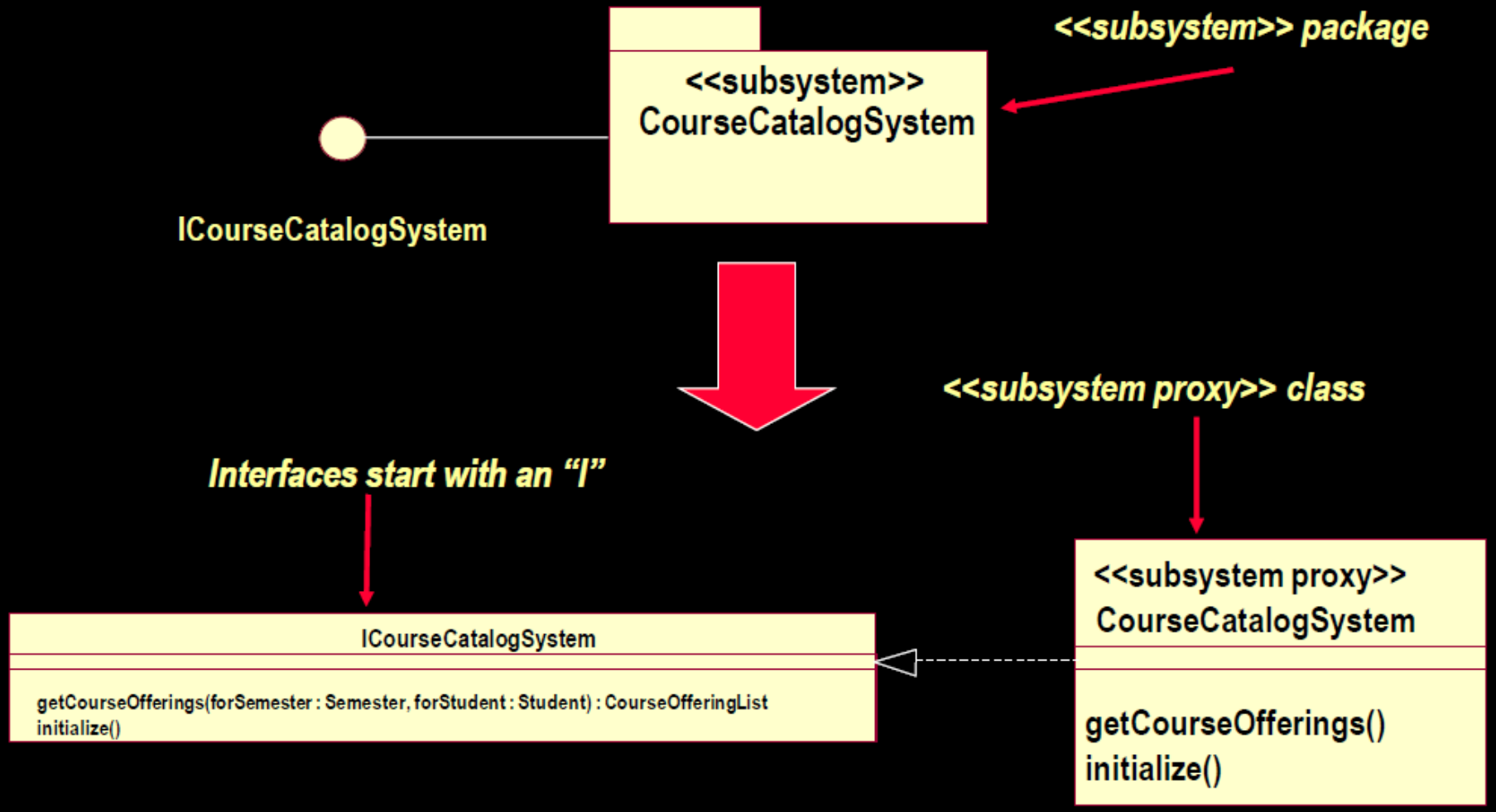
Identifying Subsystems



Example: Design Subsystems and Interfaces

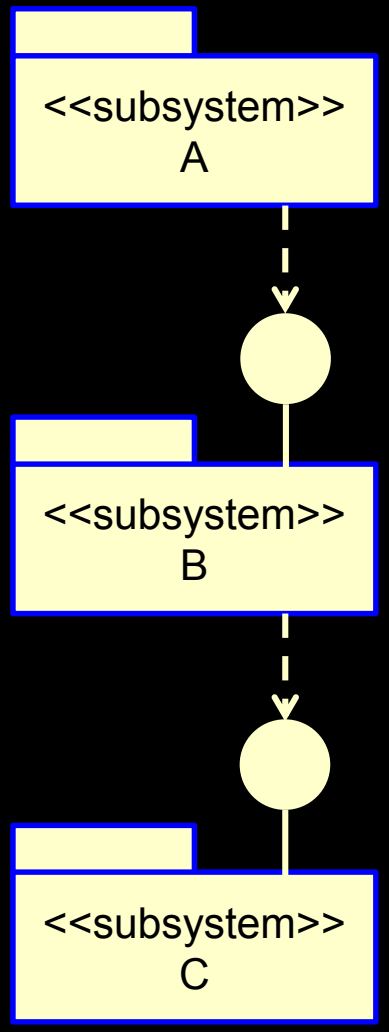


Modeling Convention: Subsystems and Interfaces



Subsystem Guidelines

- Goals
  - Loose coupling
  - Portability, plug-and-play compatibility
  - Insulation from change
  - Independent evolution
- Strong Suggestions
  - Don’t expose details, only interfaces
  - Only depend on other interfaces



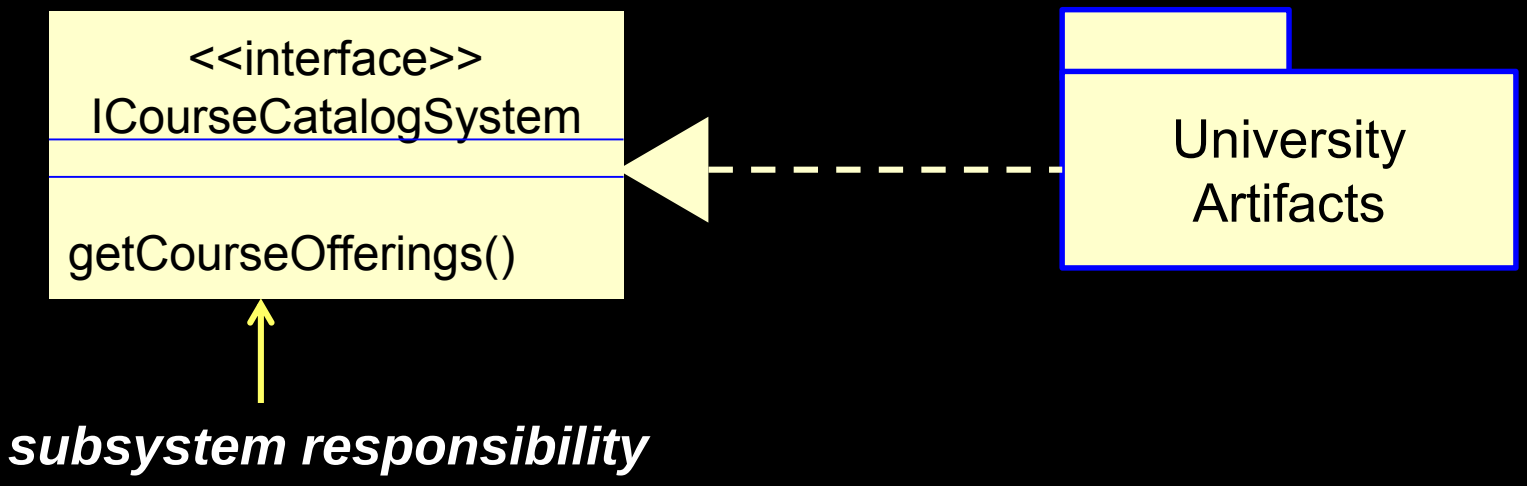
Key is abstraction and encapsulation

Subsystem Design Steps

- Distribute subsystem behavior to subsystem elements
- Document subsystem elements
- Describe subsystem dependencies

Subsystem Responsibilities

- Subsystem responsibilities defined by interface operations
  - **Model interface realizations**
- Interface operations may be realized by
  - **Internal class operations**
  - **Internal subsystem operations**

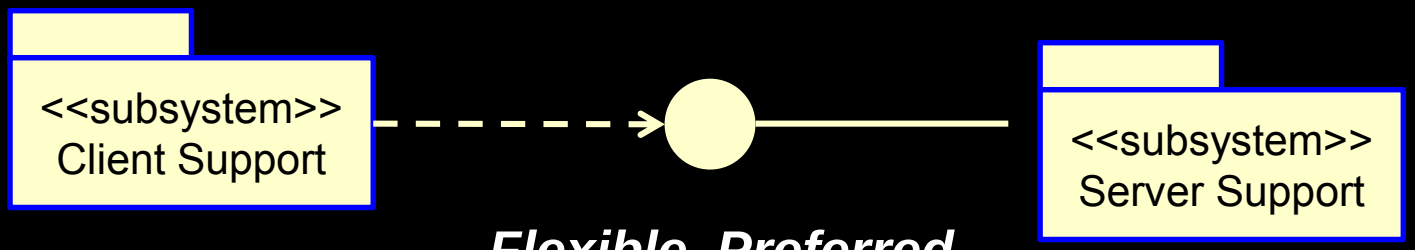


Distributing Subsystem Responsibilities

- Identify new, or reuse existing, design elements (e.g., classes and/or subsystems)
- Allocate subsystem responsibilities to design elements
- Incorporate applicable mechanisms (e.g., persistence, distribution, etc.)
- Document design element collaborations in "interface realizations"
  - **One or more interaction diagrams per interface operation**
  - **Class diagram(s) containing the required design element relationships**
- Revisit "Identify Design Elements"
  - **Adjust subsystem boundaries and/or dependencies, as needed**

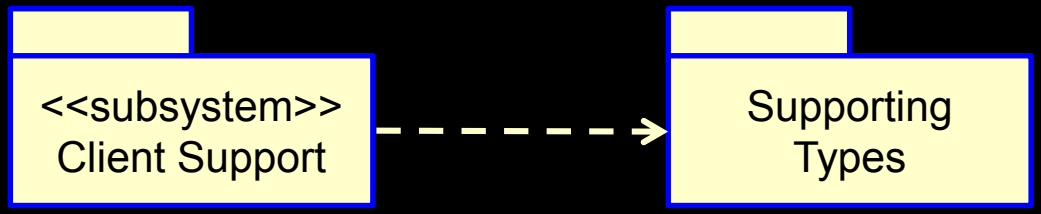
Subsystem Dependencies: Guidelines

- Subsystem dependency on a subsystem



Flexible, Preferred

- Subsystem dependency on a package



Use with care

软件工程概论

Introduction to Software Engineering

COMPONENT DESIGN

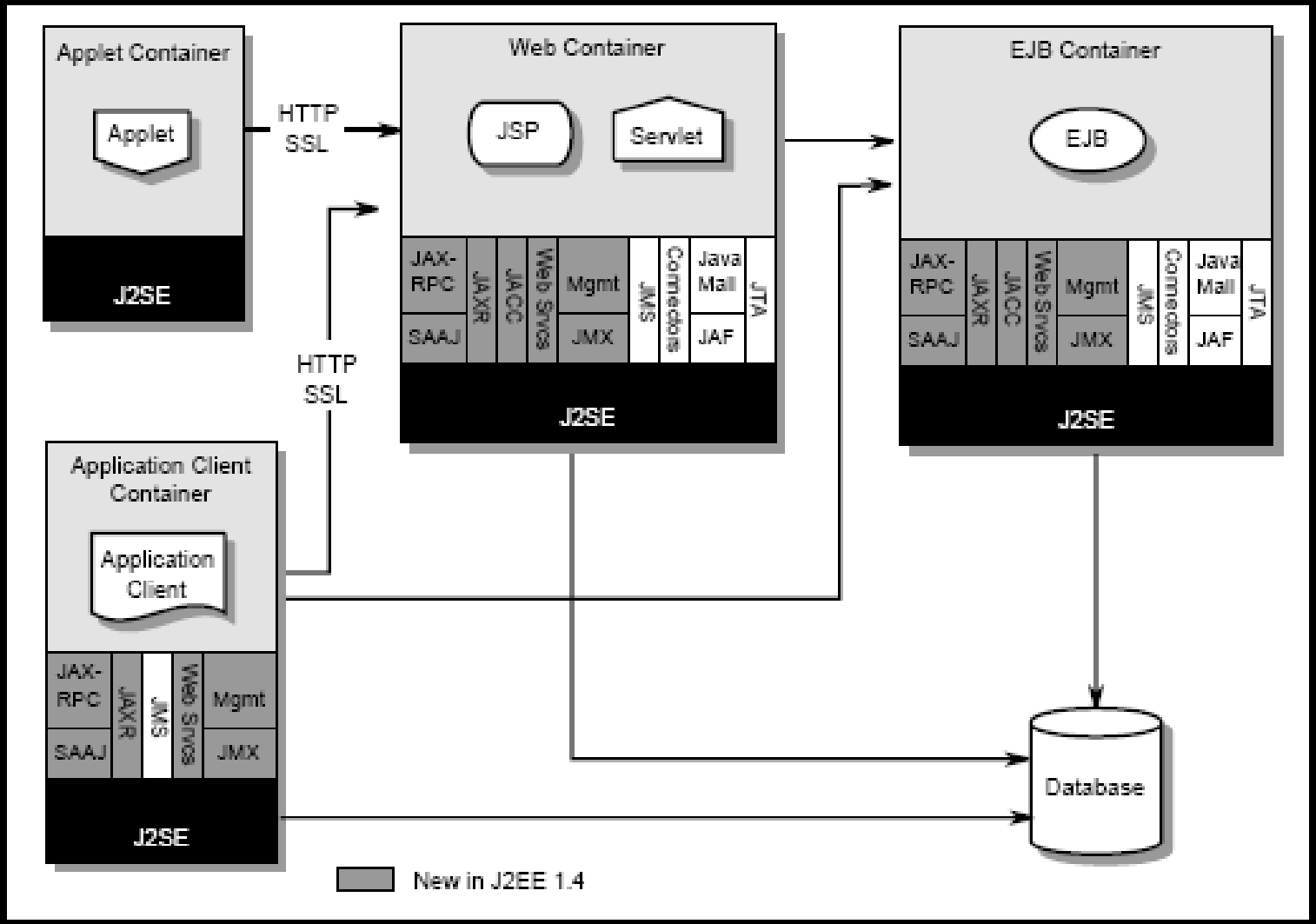
Component Design and UML CD & CSD

- Components
- UML Component Diagram (CD)
- UML Composite Structure Diagram (CSD)
- Subsystem Design

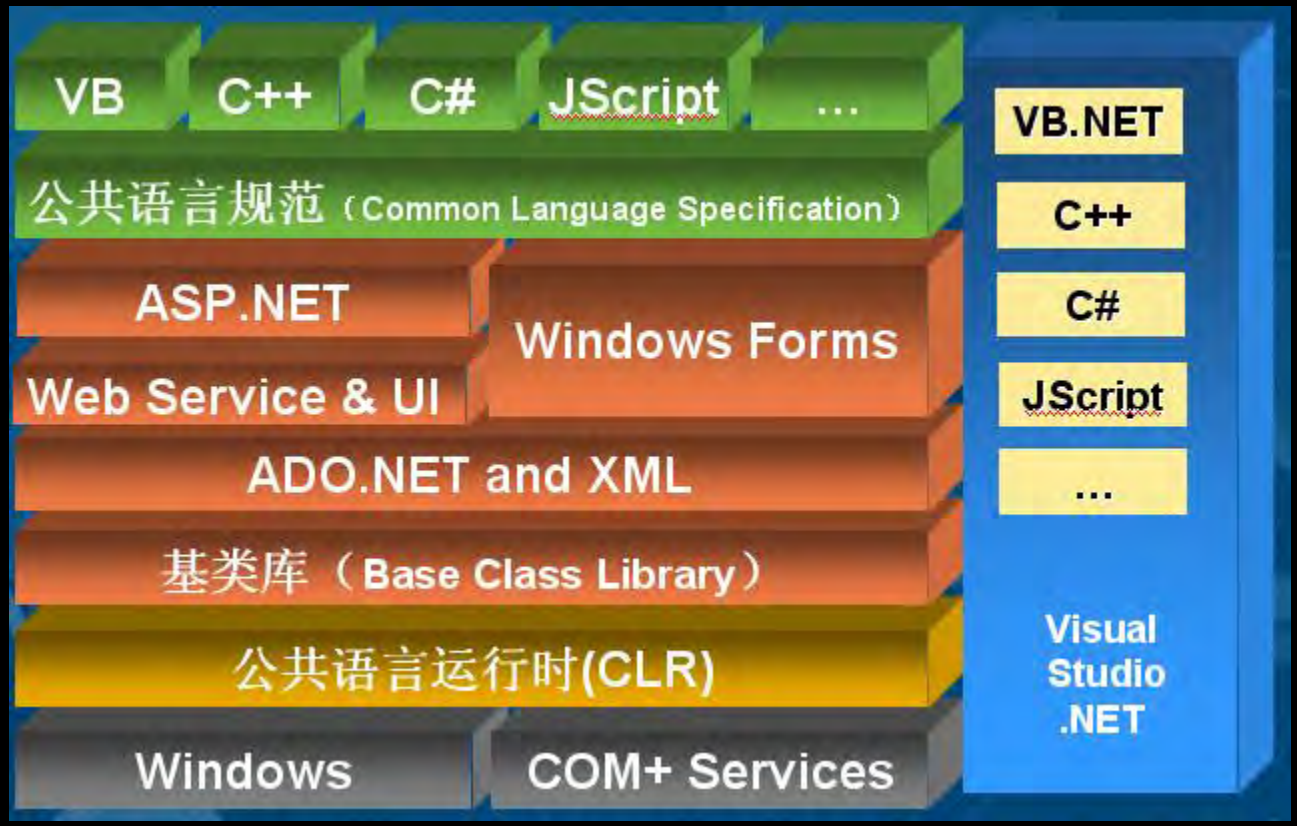
Component

- Definition of a (Software) Component
  - A non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture.
  - A component conforms to and provides the physical realization of a set of interfaces.
  - A physical, replaceable part of a system that packages implementation and conforms to and provides the realization of a set of interfaces.
  - A component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files.

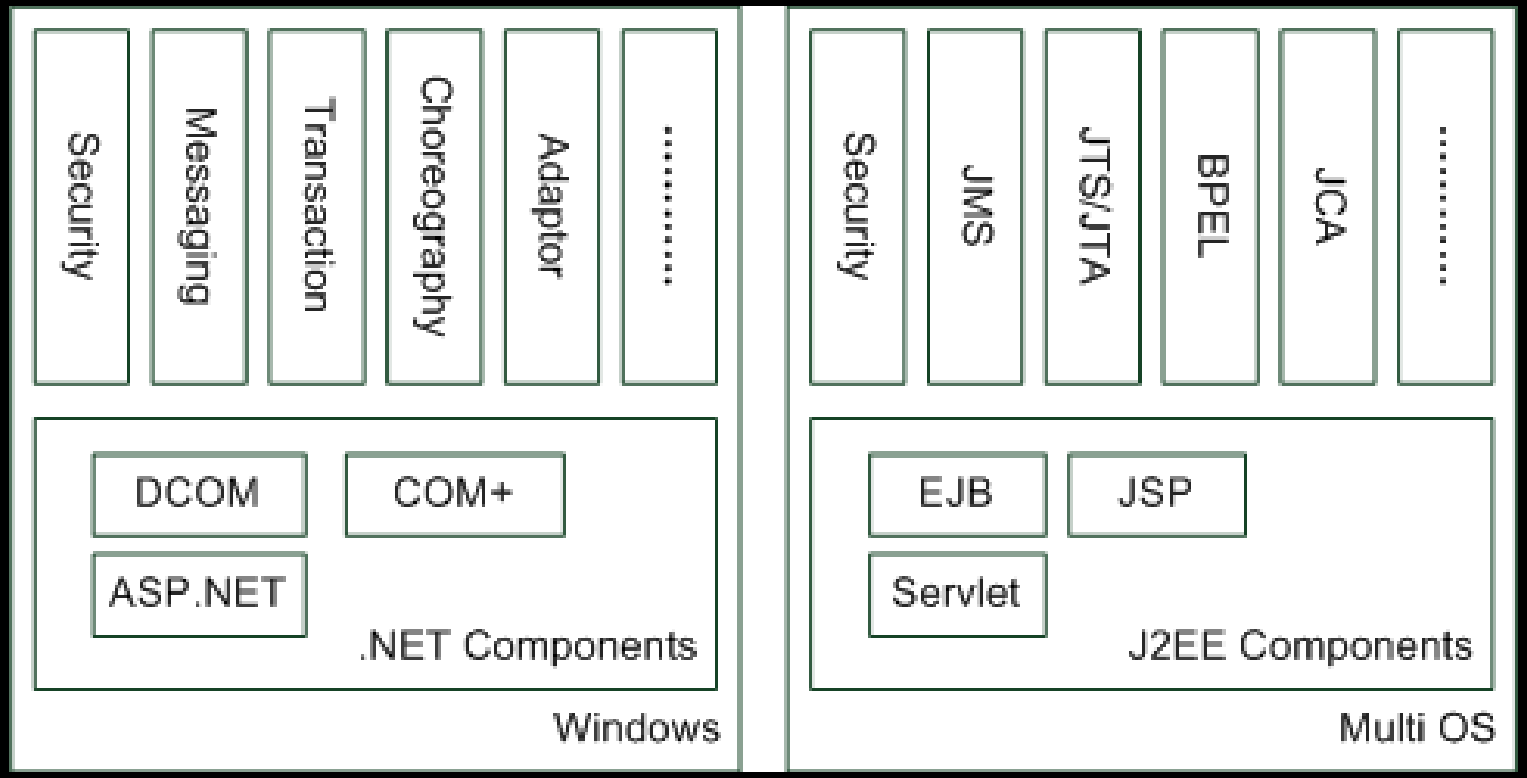
Sun Java EE


Applet Container
Applet
J2SE
HTTP
SSL
Web Container
JSP
Servlet
JAX-RPC
SAAJ
JAXR
JACC
Web Srvcs
Mgmt
JMX
JMS
Connectors
Java Mail
JAF
JTA
J2SE
EJB Container
EJB
JAX-RPC
SAAJ
JAXR
JACC
Web Srvcs
Mgmt
JMX
JMS
Connectors
Java Mail
JAF
JTA
J2SE
HTTP
SSL
Application Client Container
Application Client
JAX-RPC
SAAJ
JAXR
JMS
Web Srvcs
Mgmt
JMX
J2SE
Database
New in J2EE 1.4

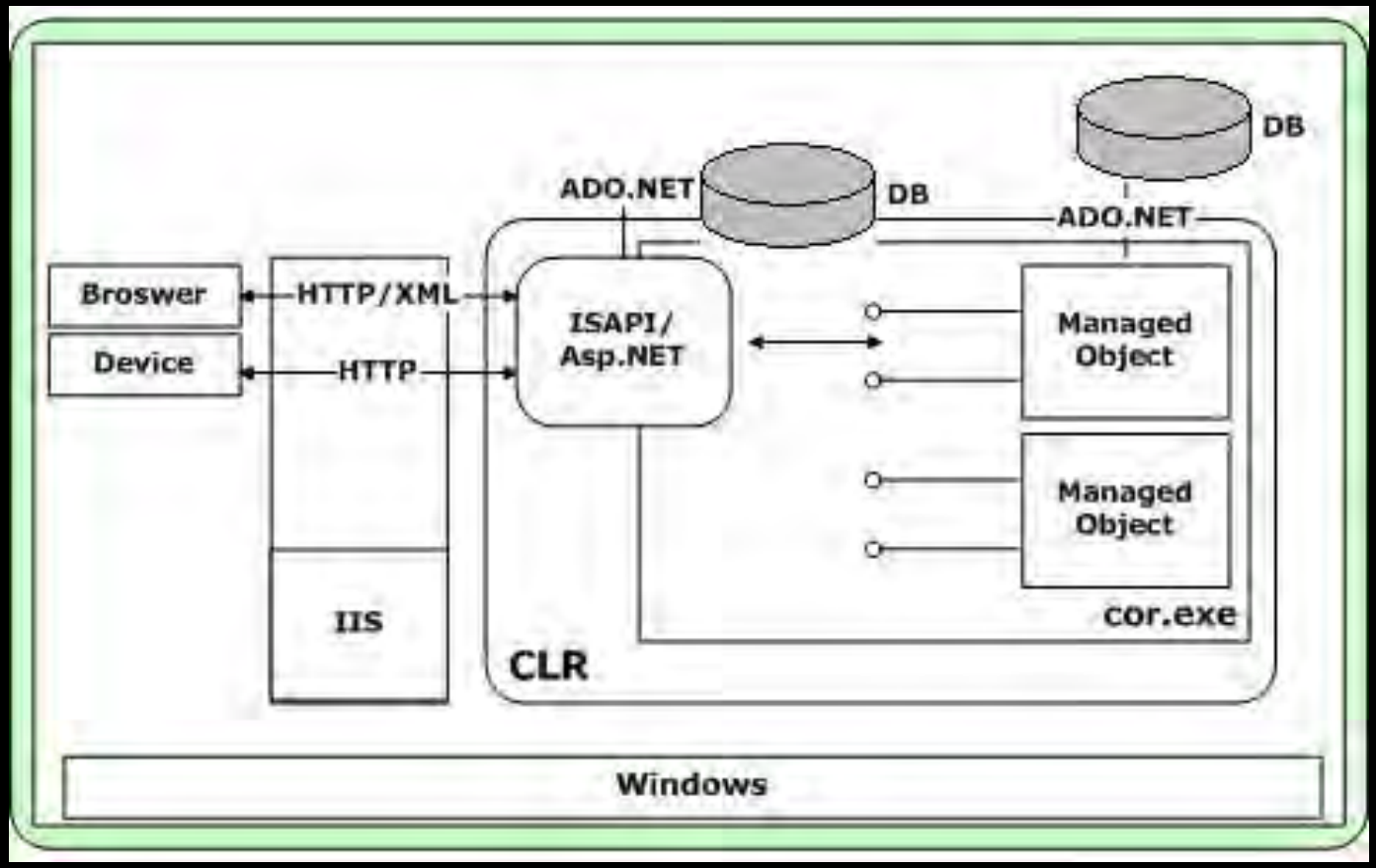
Microsoft .NET



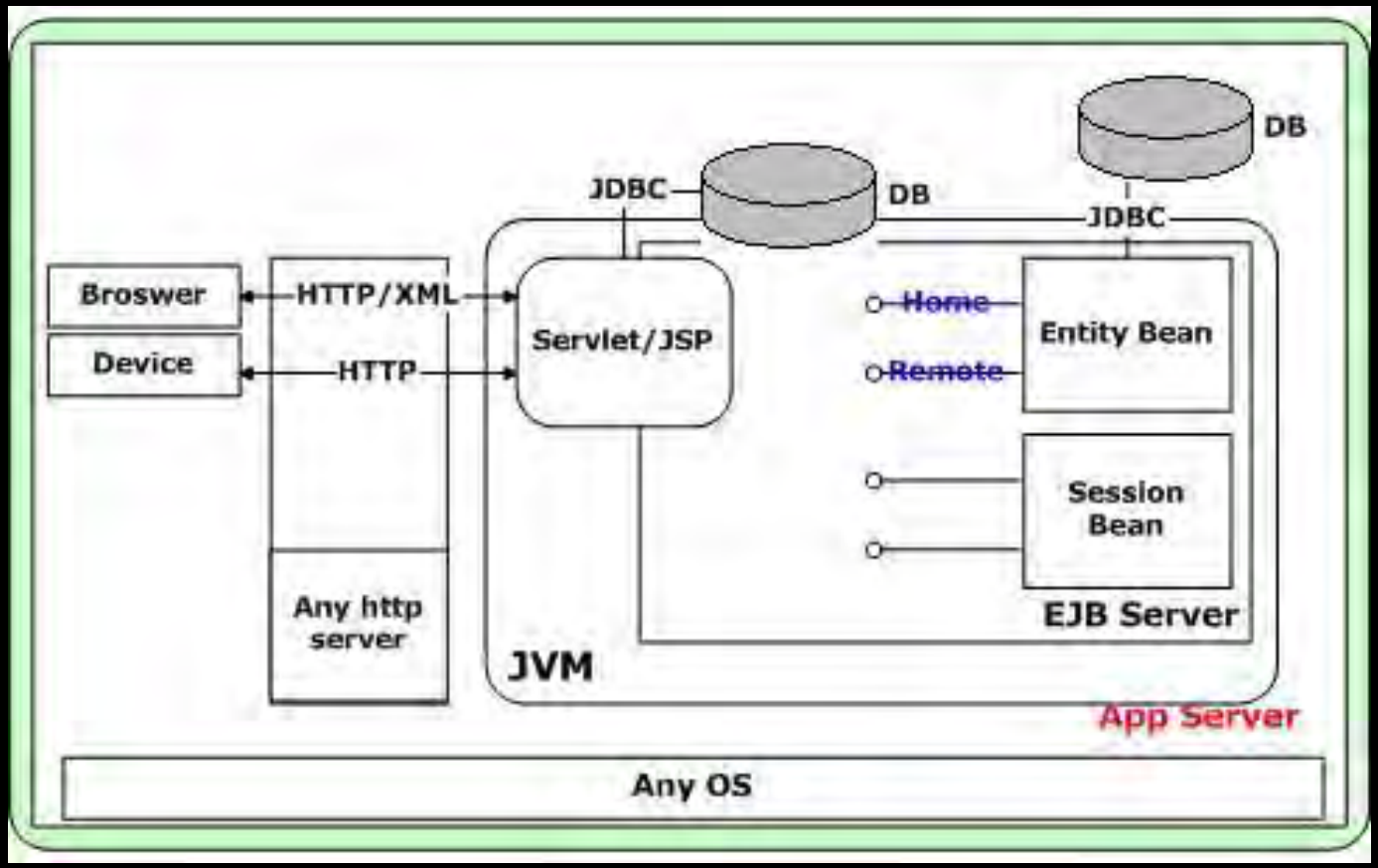
Java EE vs .NET



Java EE vs .NET



Java EE vs .NET



Components in UML

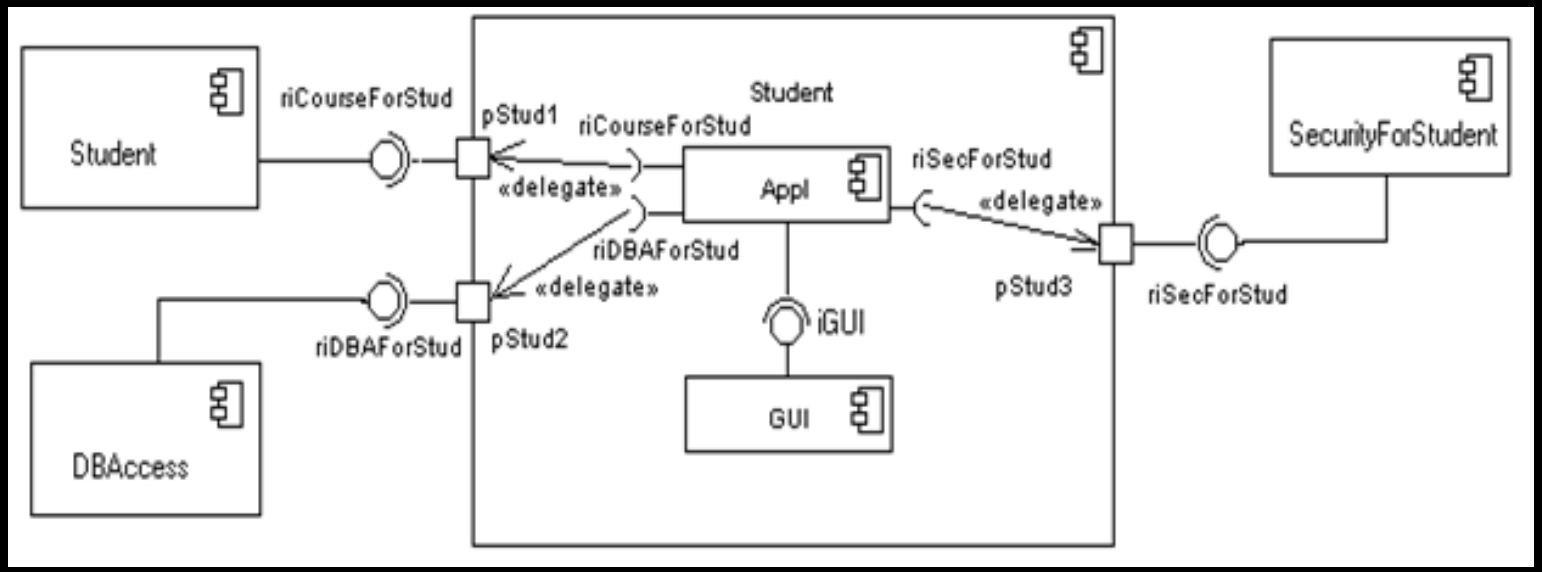
- Modular unit with well-defined interfaces that is replaceable within its environment
- Autonomous unit within a system
  - **Has one or more provided and required interfaces**
  - **Its internals are hidden and inaccessible**
  - **A component is encapsulated**
  - **Its dependencies are designed such that it can be treated as independently as possible**

Components in UML

- One of the most important amendments in UML is to enhance the support of CBSD
  - **In UML**
    - **components can used for the entire life-cycle modeling, and finally optimized in the deployment or run-time environment.**
  - **The internal structures of component can be described using composite structure diagram.**
  - **component conforms to and provides the realization of a set of interfaces,**
    - **as a replaceable part of a system, component encapsulates the behavior and states of its internal elements**
  - **Components can be assembled**
    - **and connected the interfaces between collaborating components to provide system functionality**

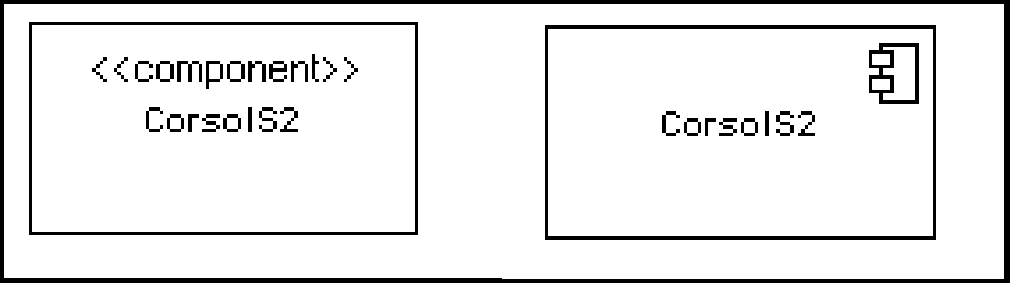
UML Component Diagram

- shows an encapsulated class and its interfaces, ports, and internal structure consisting of nested components and connectors.
- address the static design implementation view of a system.



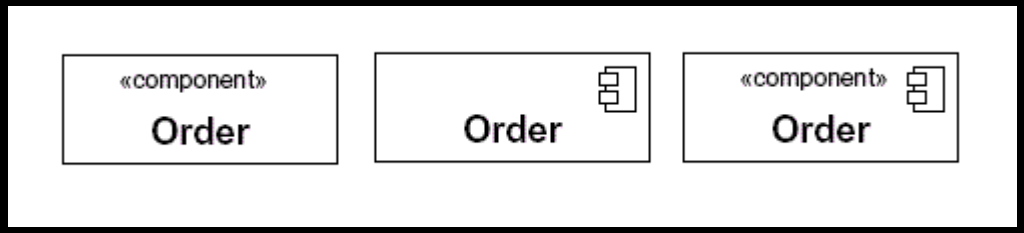
UML Component Notation

- A component is shown as a rectangle with
  - A keyword **<<component>>**
  - Optionally, in the right hand corner a component icon can be displayed
    - A component icon is a rectangle with two smaller rectangles jutting out from the left-hand side
    - This symbol is a visual stereotype
  - The component name
- Components can be labeled with a stereotype there are a number of standard stereotypes
  - ex: **<<entity>>**, **<<subsystem>>**



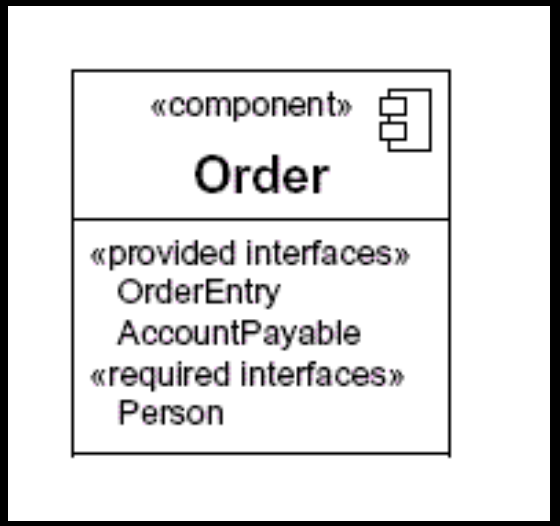
UML Component Diagram - Component

- Component
  - a replaceable part of a system that **conforms to** and **provides** the realization of a set of interfaces



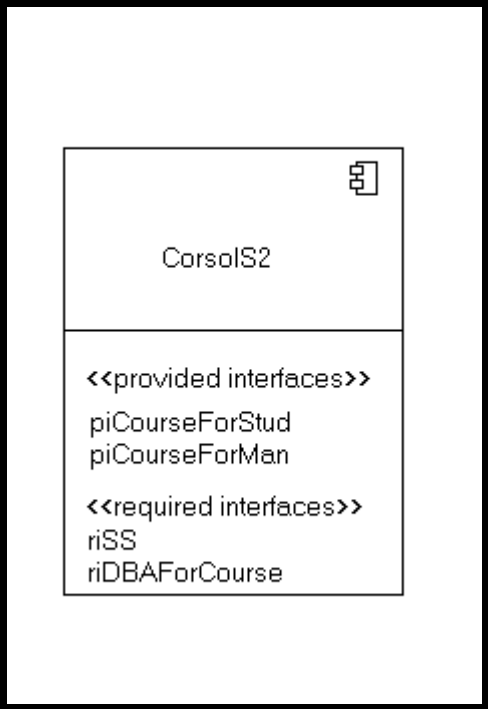
- Component and Interface
  - Component **uses** Interface

  - Component **implements** Interface



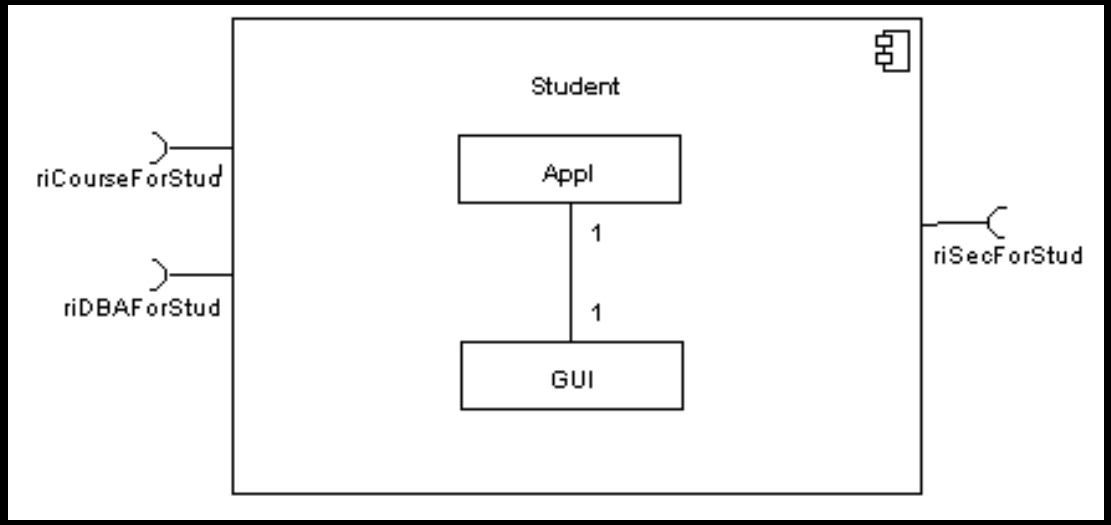
External view of a component

- A component have an external view and an internal view
  - An **external view** (or black box view) shows publicly visible properties and operations
  - An external view of a component is by means of interface symbols sticking out of the component box
  - The interface can be listed in the **compartment** of a component box



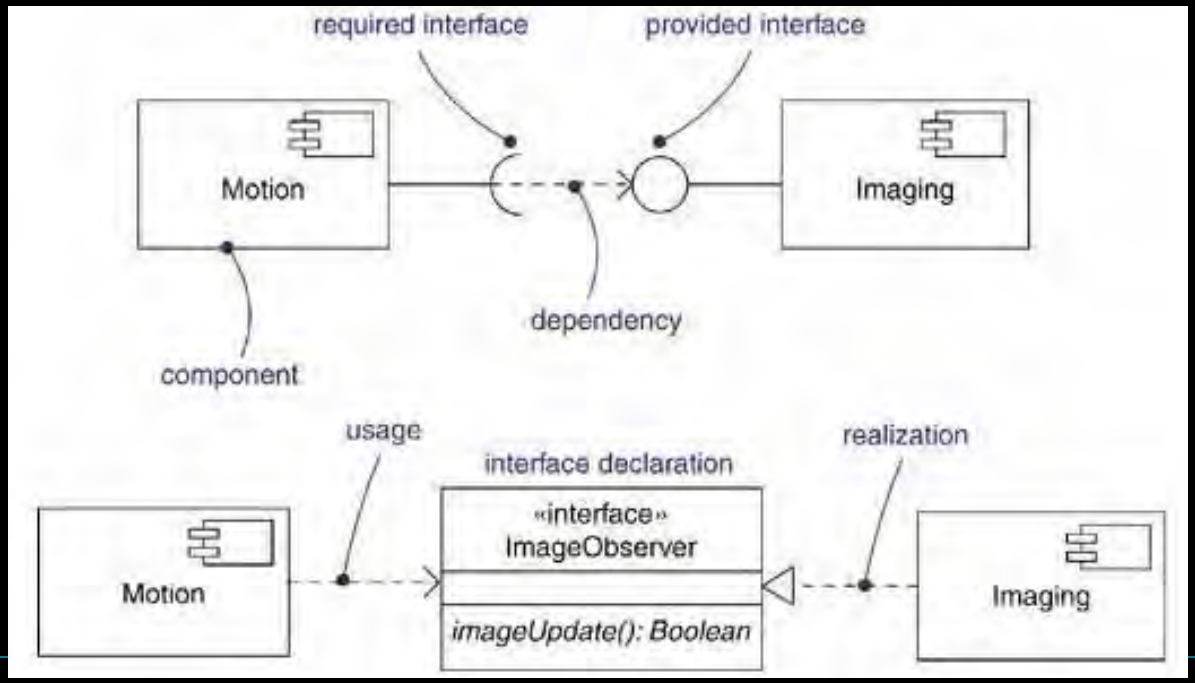
Internal view of a component

- An internal, or white box view of a component is where the realizing classes/components are nested within the component shape
- Realization is a relationship between two set of model elements
  - **One represents a specification**
  - **The other represent an implementation of the latter**



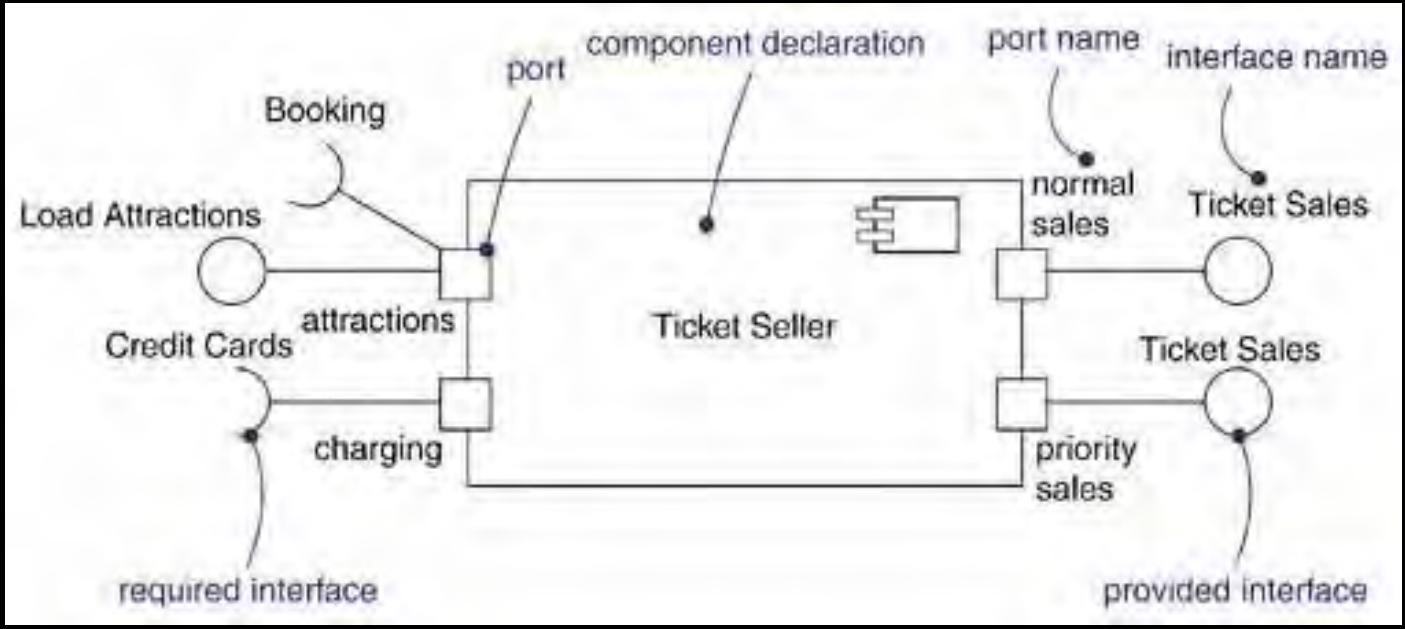
UML Component Diagram - Interface

- a collection of operations that specify a service that is provided by or requested from a class or component
  - **provided interface**
    - **Characterize services that the component offers to its environment**
  - **required interface**
    - **Characterize services that the component expects from its environment**



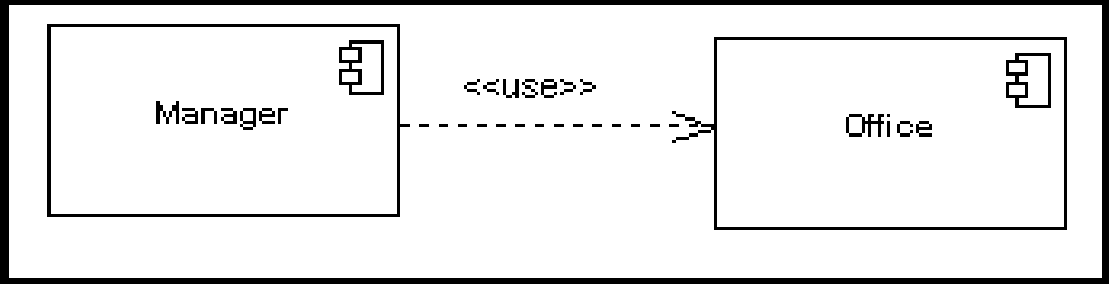
UML Component Diagram - Port

- Port
  - a specific **window** into an encapsulated component **accepting messages to and from the component conforming to specified interfaces**



Dependencies

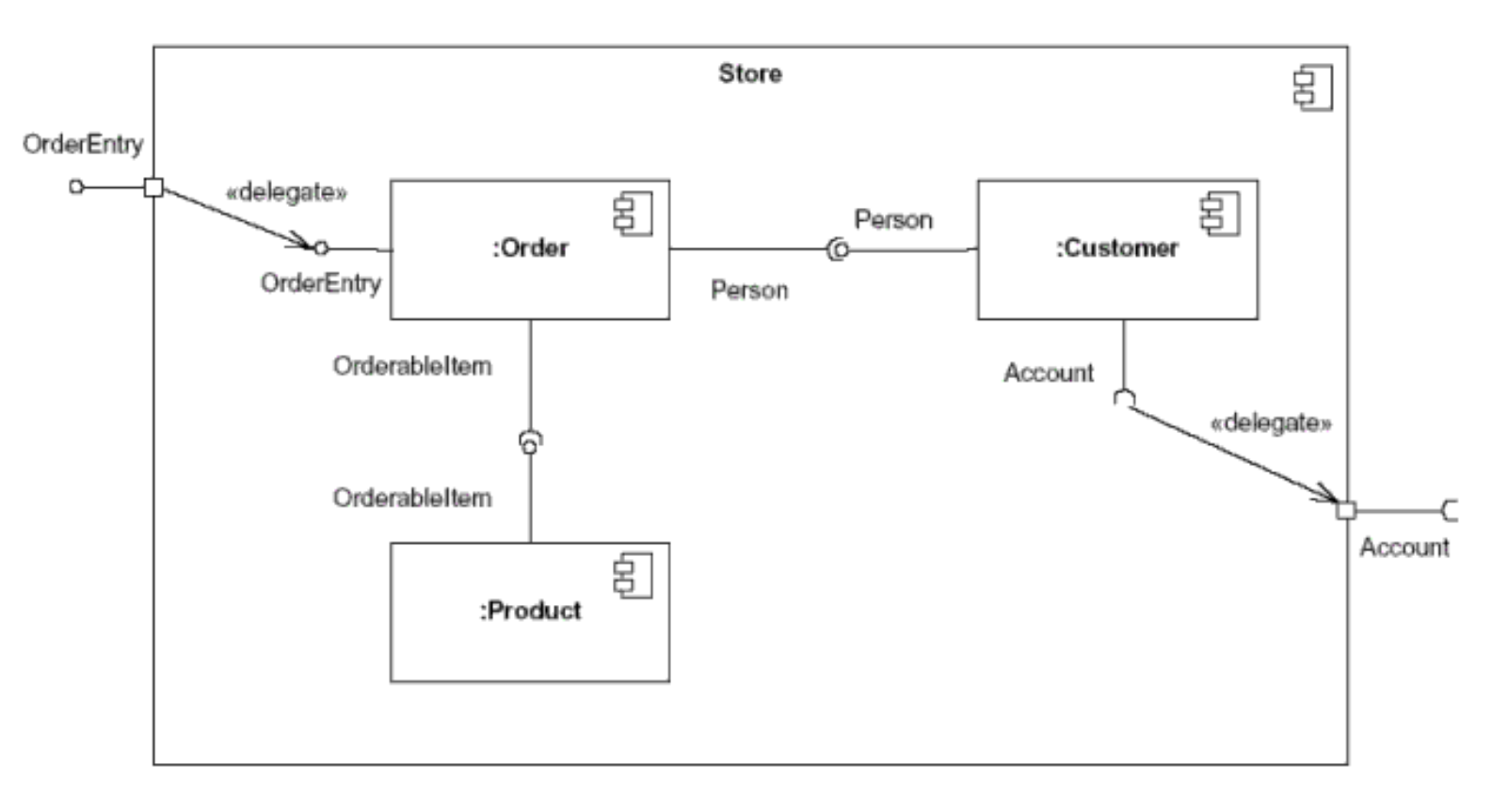
- Components can be connected by usage dependencies



- Usage Dependency
  - A usage dependency is relationship which one element requires another element for its full implementation
  - Is a dependency in which the client requires the presence of the supplier
  - Is shown as dashed arrow with a <<use>> keyword
  - The arrowhead point from the dependent component to the one of which it is dependent

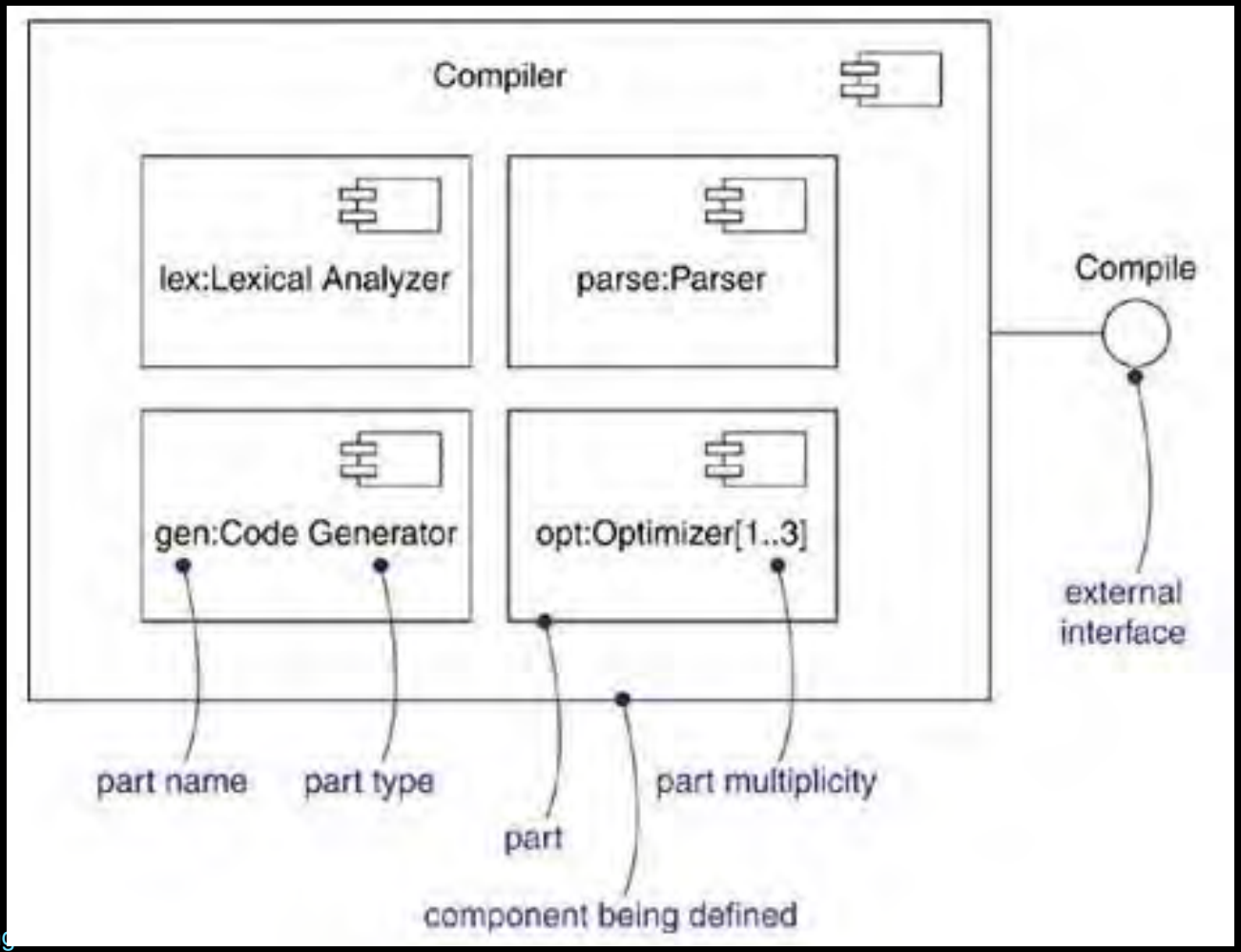
UML Component Diagram - Internal Structure

- Internal structure
  - the implementation of a component by means of a set of parts that are connected together in a specific way



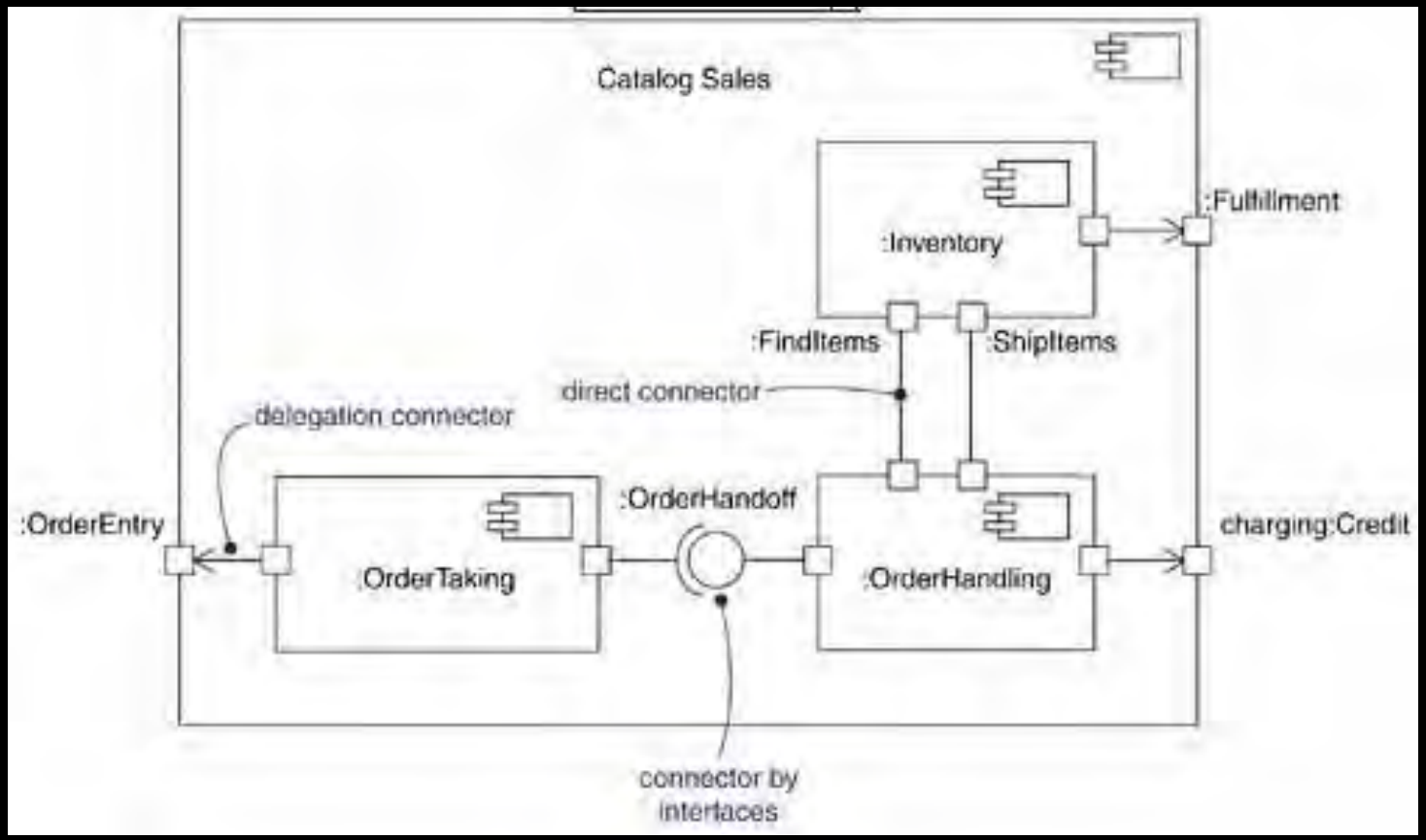
UML Component Diagram - Part

- Part
  - a unit of the implementation of a component



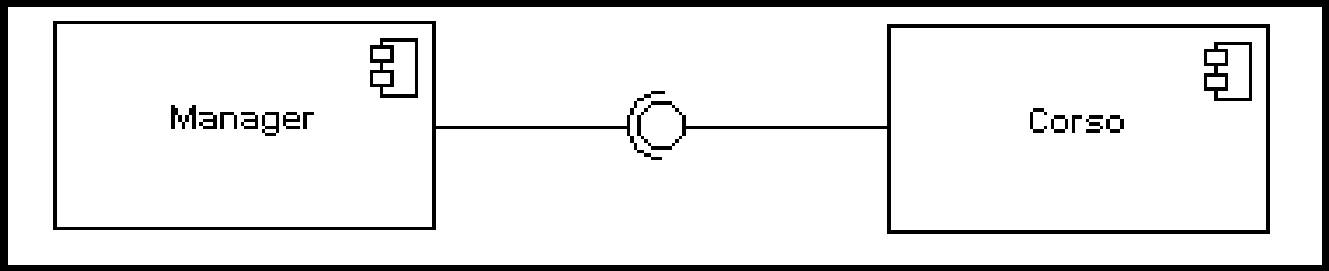
UML Component Diagram - Connector

- a communication relationship between two parts or ports within the context of a component
  - **Two kinds of Connectors: Assembly and Delegation**



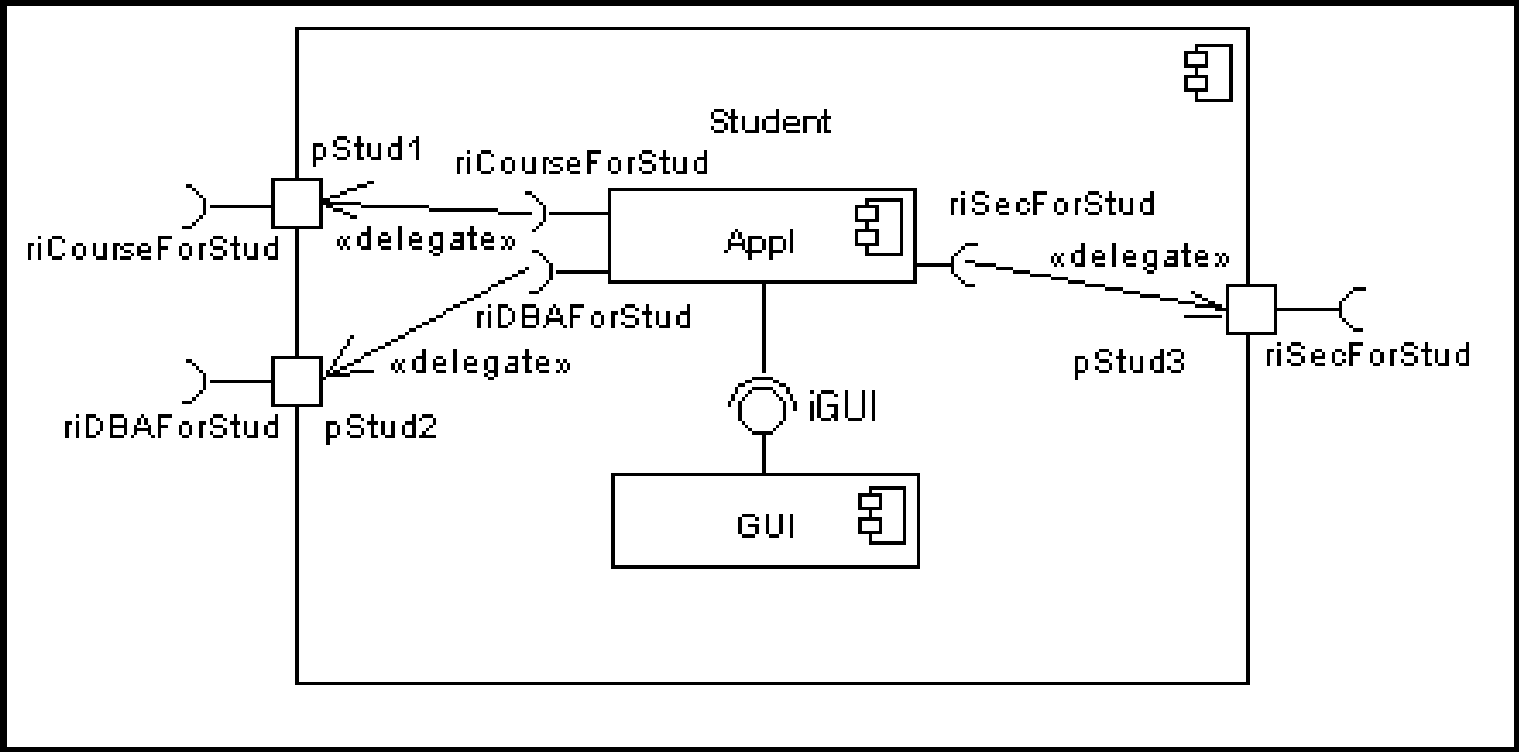
Assembly Connector

- A connector between two components defines that one component provides the services that another component requires
  - It must only be defined from a required interface to a provided interface
  - An assembly connector is notated by a —ball-and-socket connection
  - This notation allows for succinct graphical wiring of components

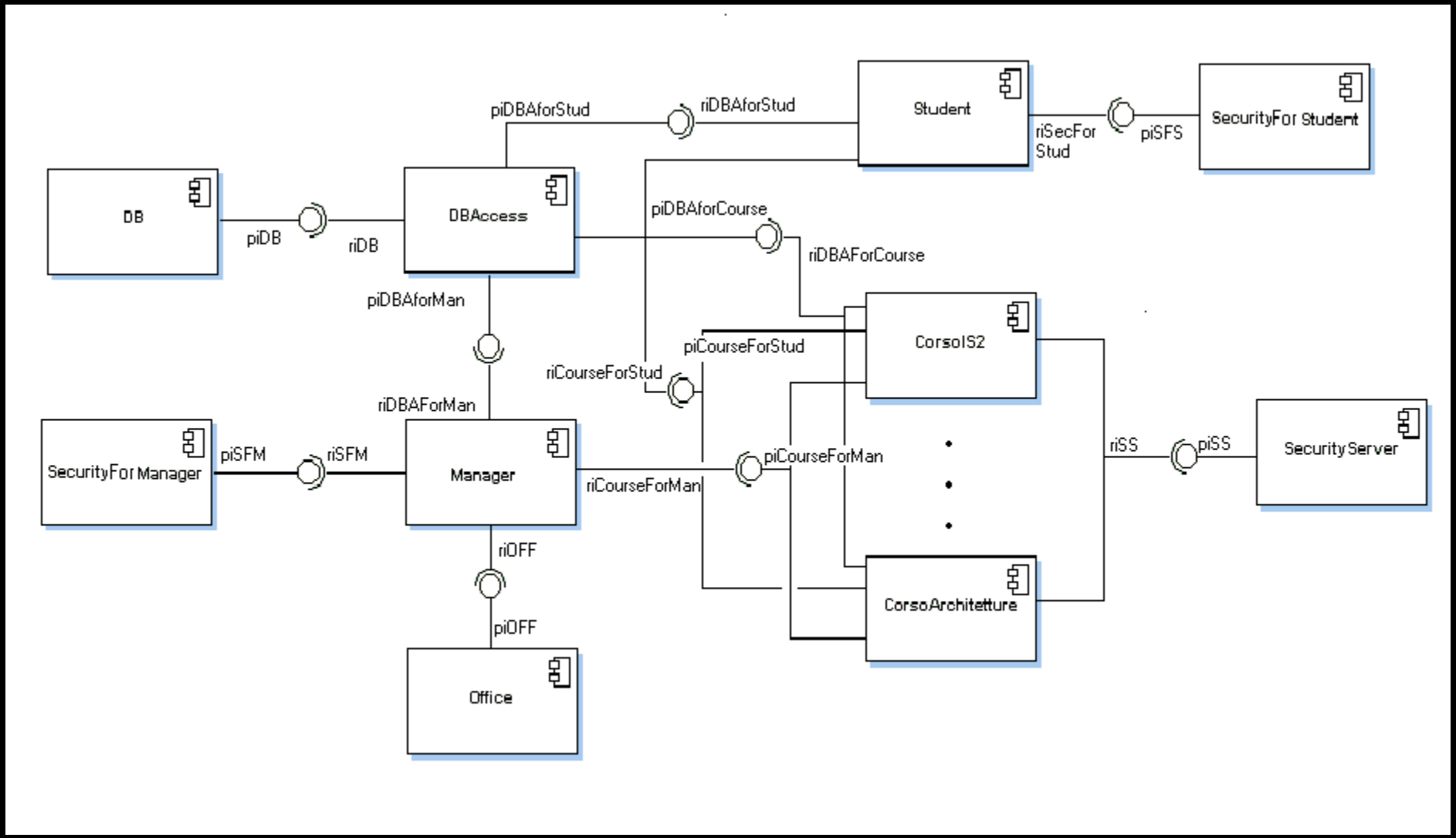


Delegation Connector

- Links the external contract of a component to the internal realization
  - Represents the forwarding of signals
  - It must only be defined between used interfaces or ports of the same kind

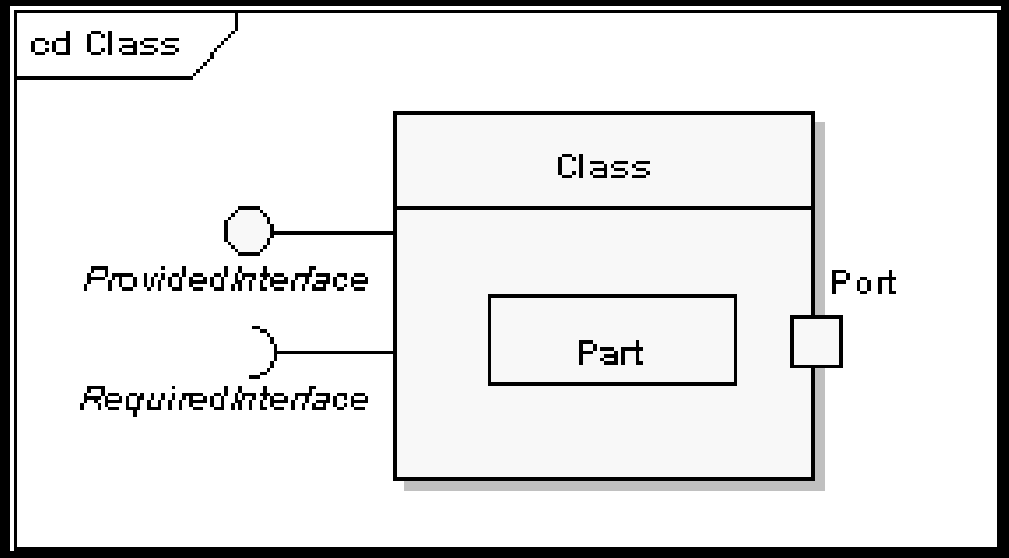


UML Component Diagram -Example



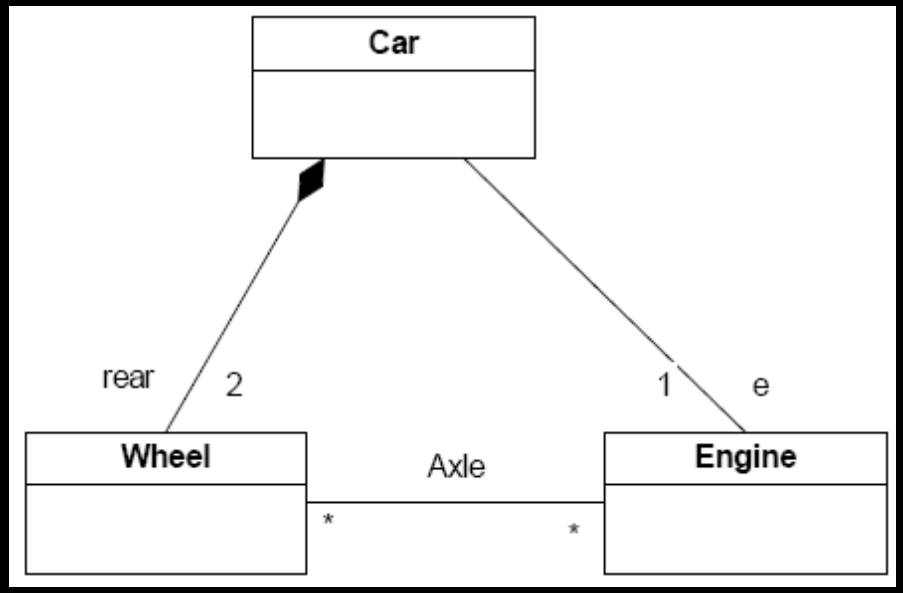
UML Composite Structure Diagram

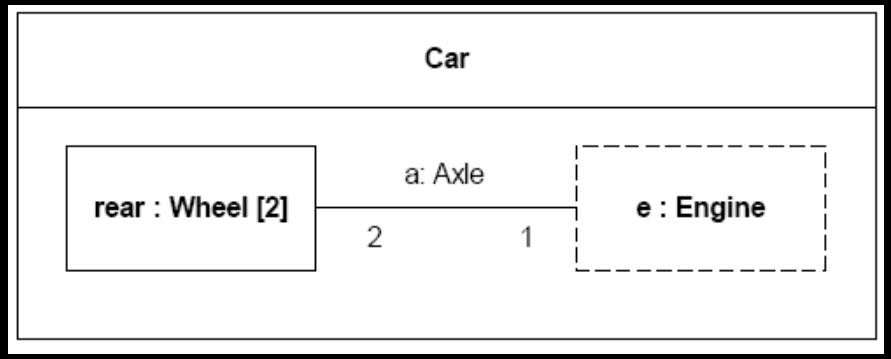
- a diagram that shows the internal structure of a classifier, including its interaction points to other parts of the system.
- It shows the configuration and relationship of parts, that together, perform the behavior of the containing classifier.



Part and Property

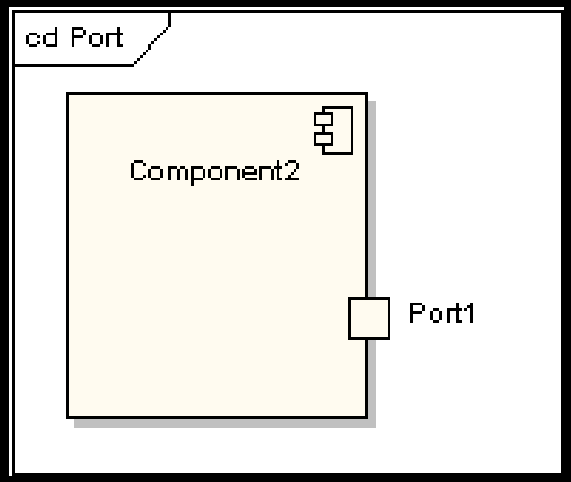
- two possible views of the Car class
  - **Car is shown as having a composition association with role name rear to a class Wheel and an association with role name e to a class Engine**
  - **it is specified that rear and e belong to the internal structure of the class Car. This allows specification of detail that holds only for instances of the Wheel and Engine classes within the context of the class Car, but which will not hold for wheels and engines in general.**

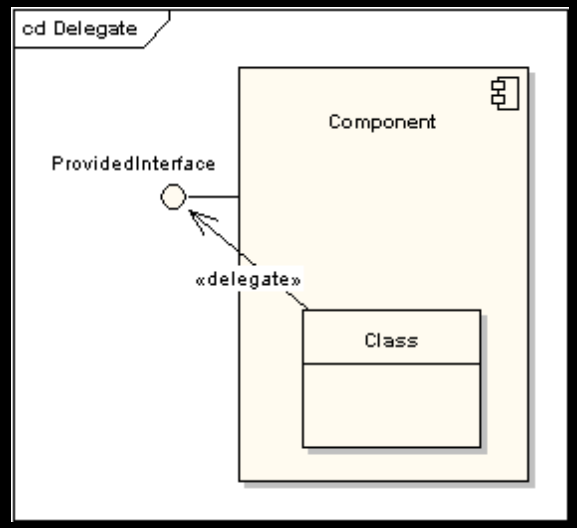




UML Composite Structure Diagram

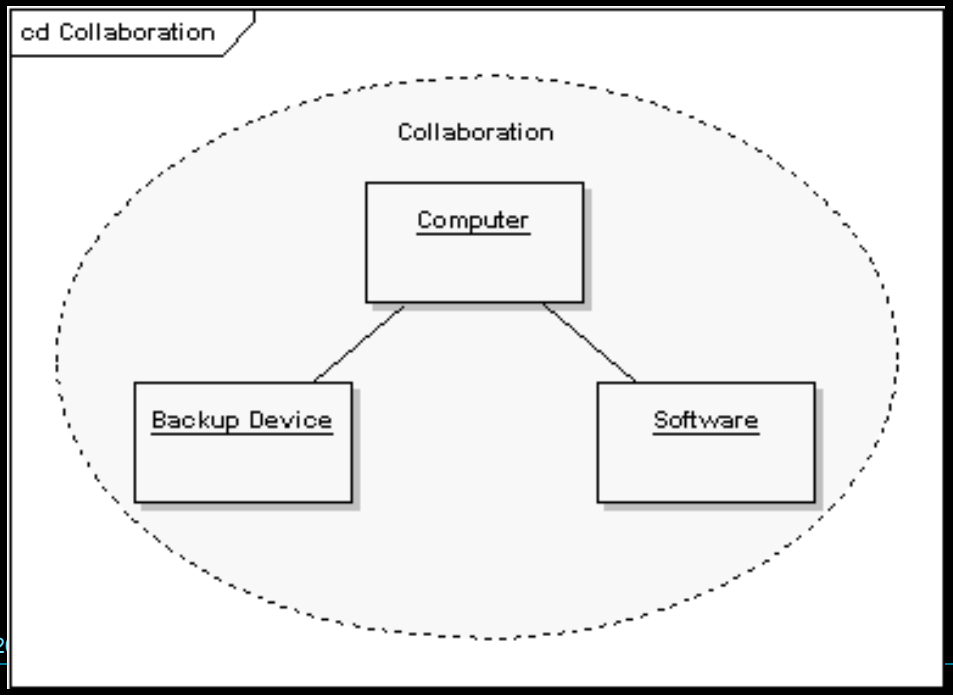
- Port
  - a point for conducting interactions between the internals of the classifier and its environment.
- Delegate
  - A delegate connector is used for defining the internal workings of a component's external ports and interfaces.
  - It connects an external contract of a component as shown by its ports to the internal realization of the behavior of the component's part.





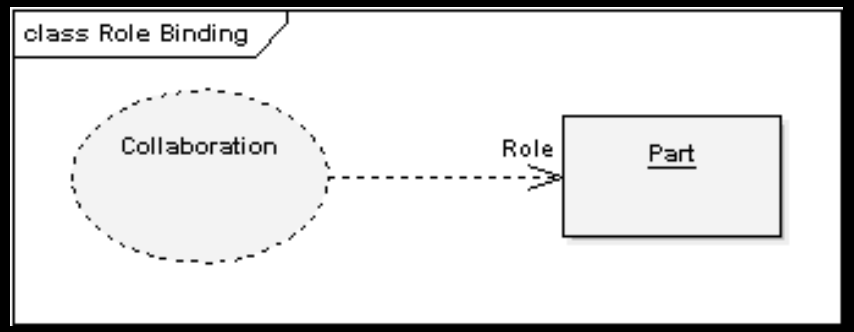
Composite Structure Diagram - Collaboration

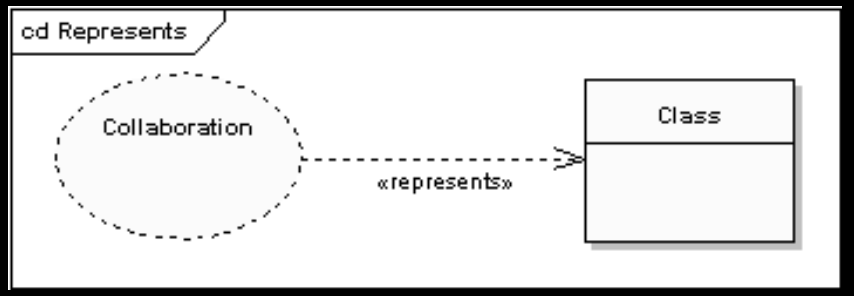
- defines a set of co-operating roles used collectively to illustrate a specific functionality.
- Collaborations have two aspects:
  - **a structural aspect**
    - **specifies the classes, interfaces, and other elements that work together to carry out the named collaboration;**
  - **a behavioral aspect**
    - **specifies the dynamics of how those elements interact.**

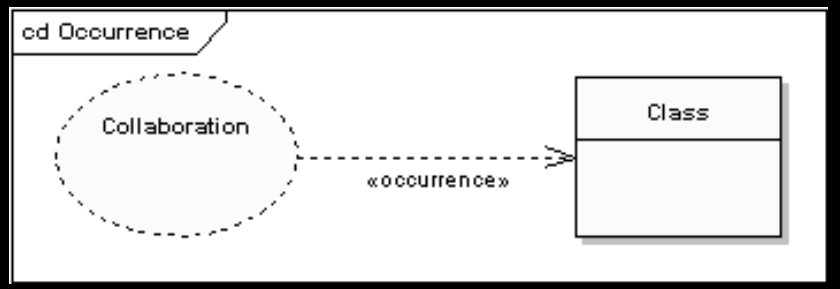


Composite Structure Diagram - Collaboration

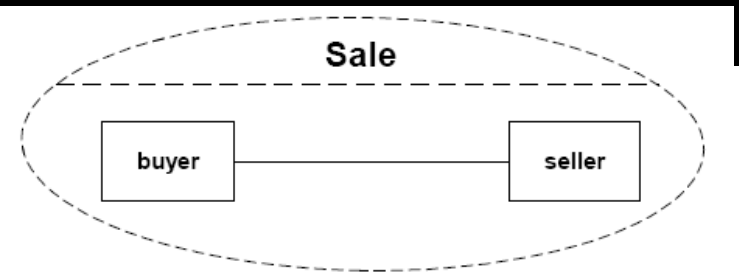
- Role Binding
  - A role binding connector is drawn from a collaboration to the classifier that fulfills the role.
- Represents
  - A represents connector may be drawn from a collaboration to a classifier to show that a collaboration is used in the classifier
- Occurrence
  - An occurrence connector may be drawn from a collaboration to a classifier to show that a collaboration represents the classifier

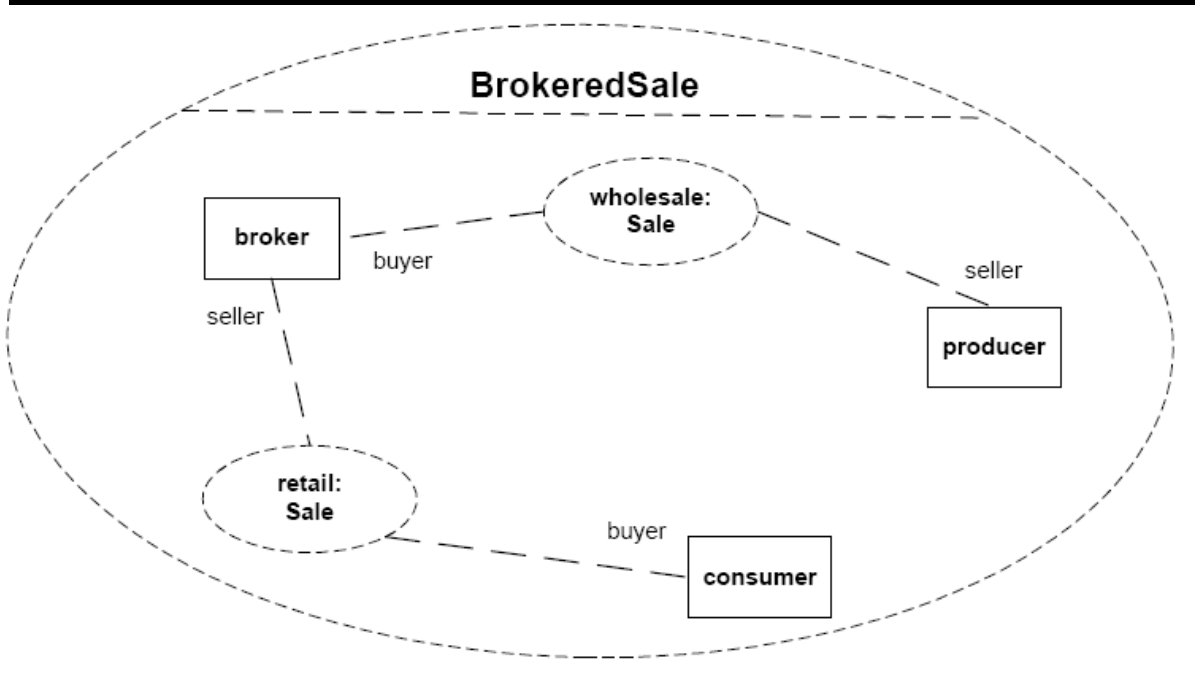


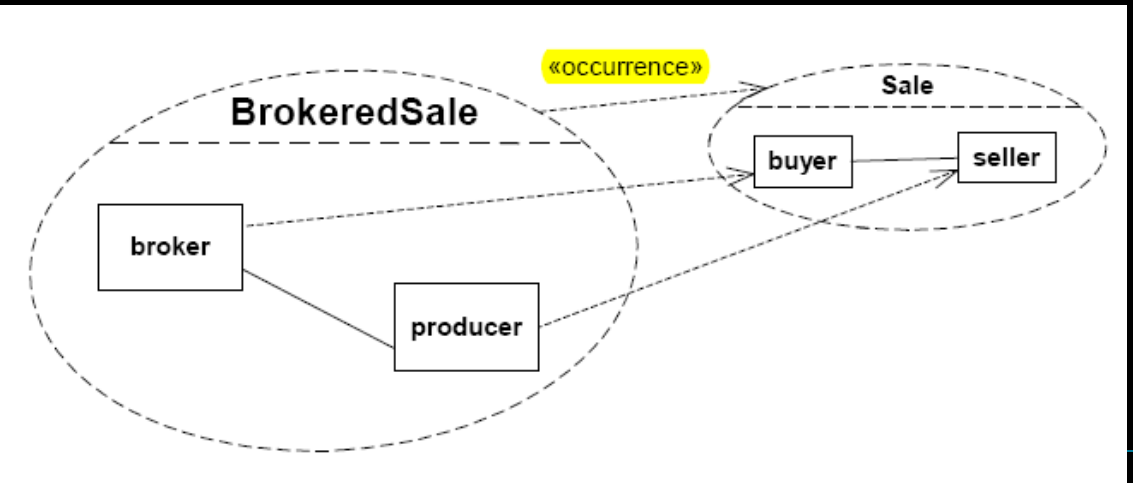




Composite Structure Diagram - Collaboration







软件工程概论

Introduction to Software Engineering

CLASS DESIGN AND UML STATE MACHINE DIAGRAM

Class Design and UML State Machine Diagram

- Class Design
  - **Create Initial Design Classes**
  - **Define Operations**
  - **Define Methods**
  - **Define States**
  - **Define Attributes**
  - **Define Dependencies**
  - **Define Associations**
- UML State Machine Diagram

Class Design Considerations

- Class stereotype
  - **Boundary**
  - **Entity**
  - **Control**
- Applicable design patterns
- Architectural mechanisms
  - **Persistence**
  - **Distribution**
  - **etc.**

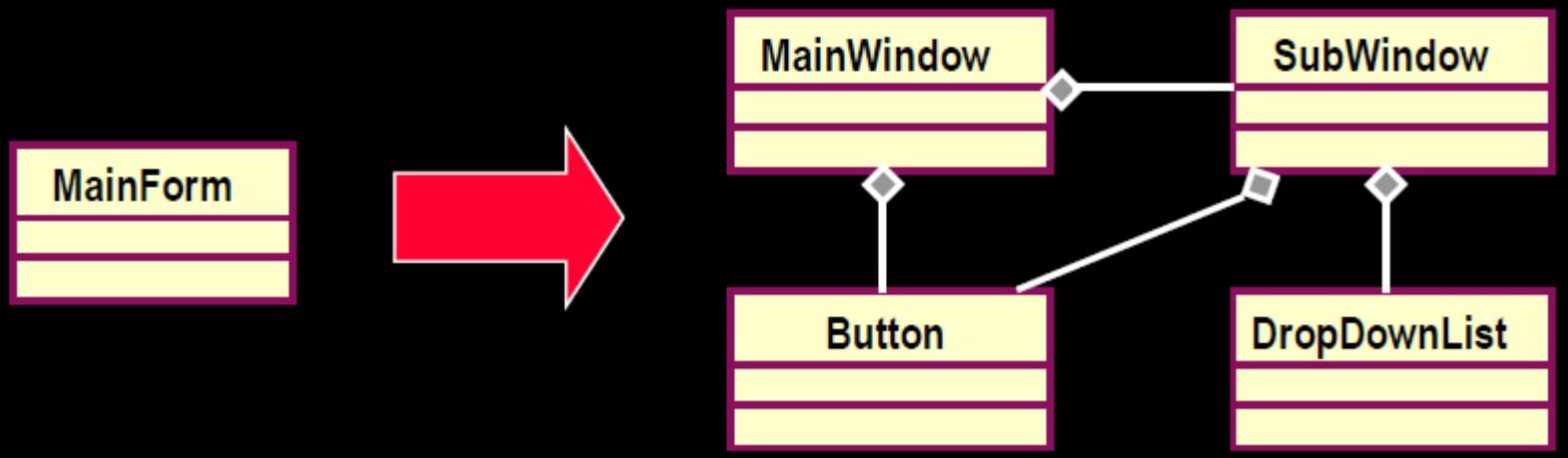
How Many Classes Are Needed?

- Many, simple classes means that each class
  - Encapsulates **less** of the overall system intelligence
  - Is **more** reusable
  - Is **easier** to implement
- A few, complex classes means that each class
  - Encapsulates a **large** portion of the overall system intelligence
  - Is **less** likely to be reusable
  - Is **more difficult** to implement

A class should have a single well focused purpose. A class should do one thing and do it well!

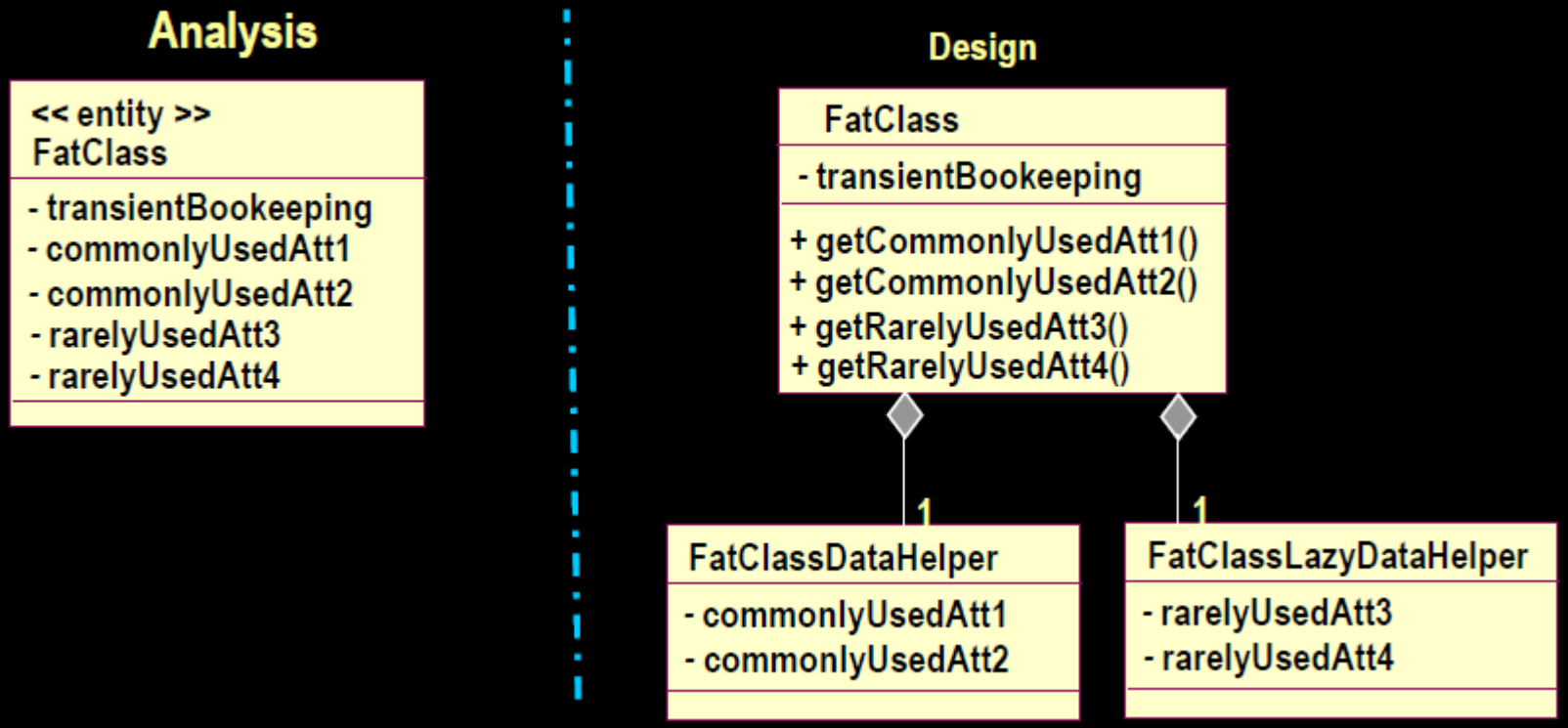
Strategies for Designing Boundary Classes

- User interface (UI) boundary classes
  - **What user interface development tools will be used?**
  - **How much of the interface can be created by the development tool?**
- External system interface boundary classes
  - **Usually model as subsystem**



Strategies for Designing Entity Classes

- Entity objects are often passive and persistent
- Performance requirements may force some refactoring
- See Identify Persistent Classes step

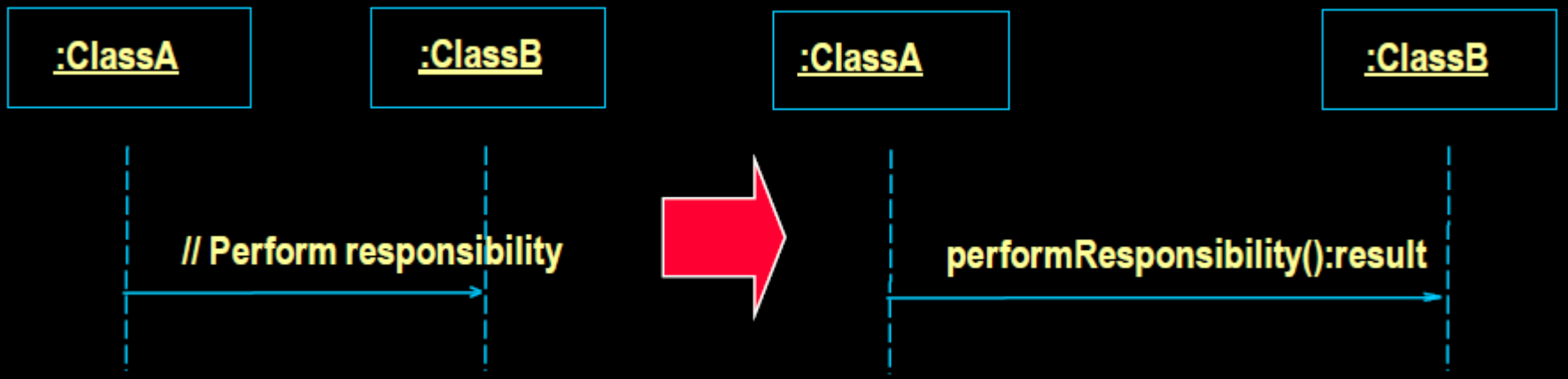


Strategies for Designing Control Classes

- What happens to Control Classes?
  - **Are they really needed?**
  - **Should they be split?**
- How do you decide?
  - **Complexity**
  - **Change probability**
  - **Distribution and performance**
  - **Transaction management**

Operations: Where Do You Find Them?

- Messages displayed in interaction diagrams



- Other implementation dependent functionality
  - **Manager functions**
  - **Need for class copies**
  - **Need to test for equality**

Name and Describe the Operations

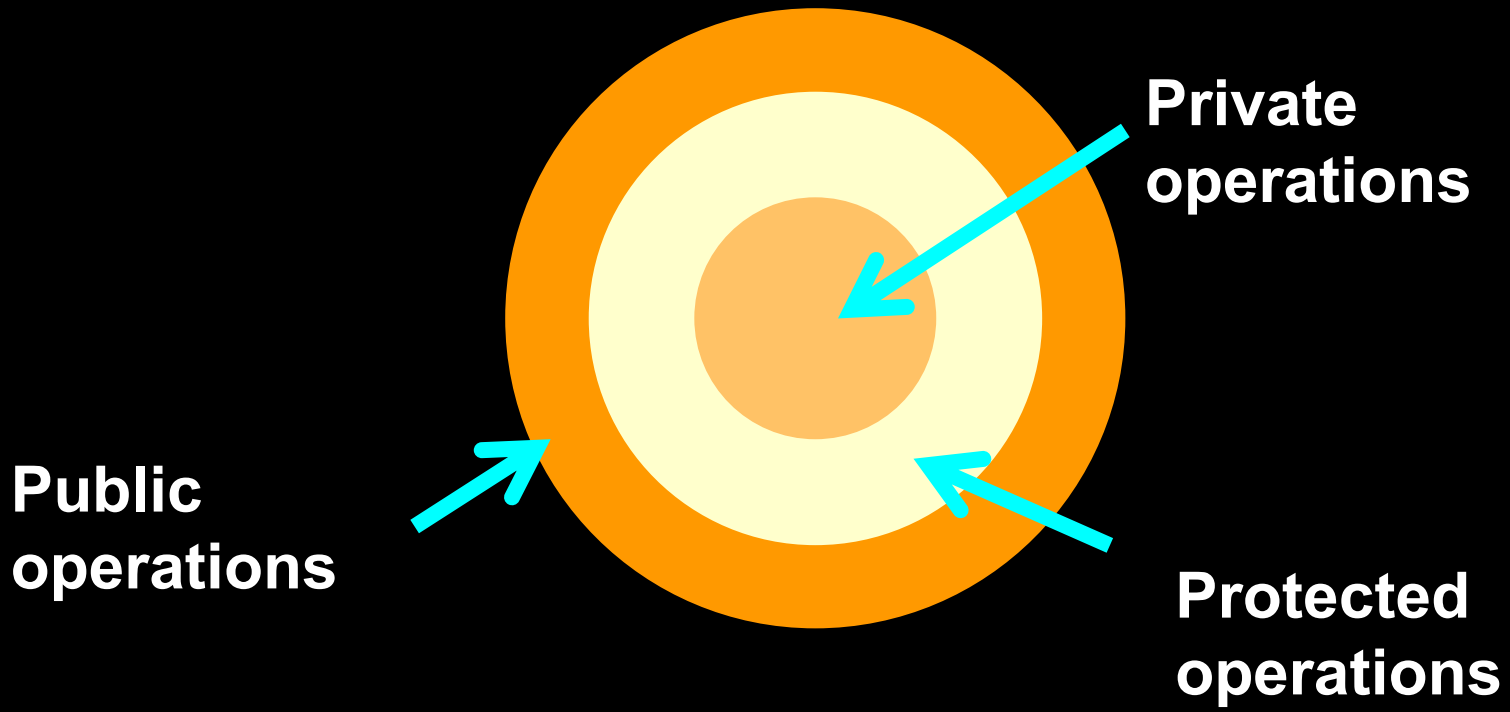
- Appropriate operation names
  - **Indicate the outcome**
  - **Use client perspective**
  - **Consistent across classes**
- Define operation signatures
  - **operationName(parameter : class,..) : returnType**
- Provide short description, including meaning of all parameters

Guidelines: Designing Operation Signatures

- When designing operation signatures, consider if parameters are:
  - **Passed by-value or by-reference?**
  - **Changed by the operation?**
  - **Optional?**
  - **Set to default values?**
  - **In valid parameter ranges?**
- The fewer the parameters, the better
- Pass objects instead of "data bits"

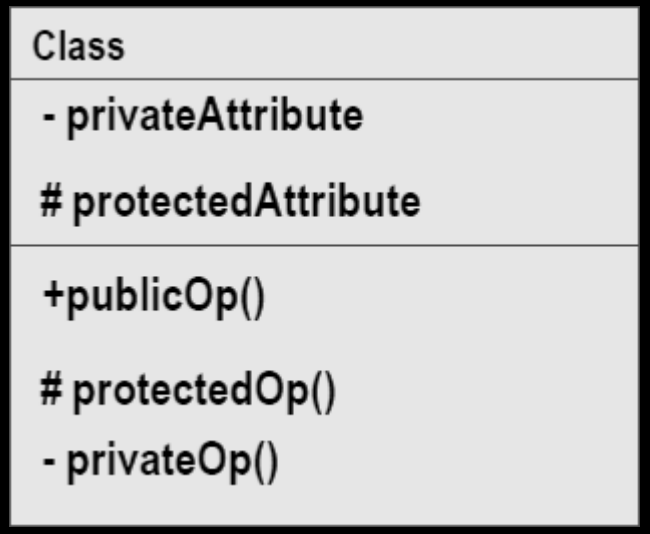
Operation Visibility

- Visibility is used to enforce encapsulation
- May be public, protected, or private



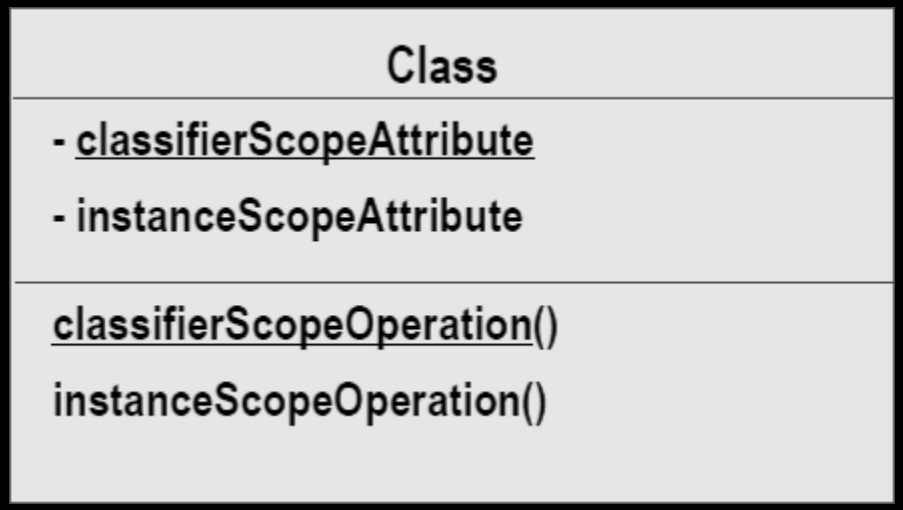
How Is Visibility Noted?

- The following symbols are used to specify export control:
  - **+ Public access**
  - **# Protected access**
  - **- Private access**

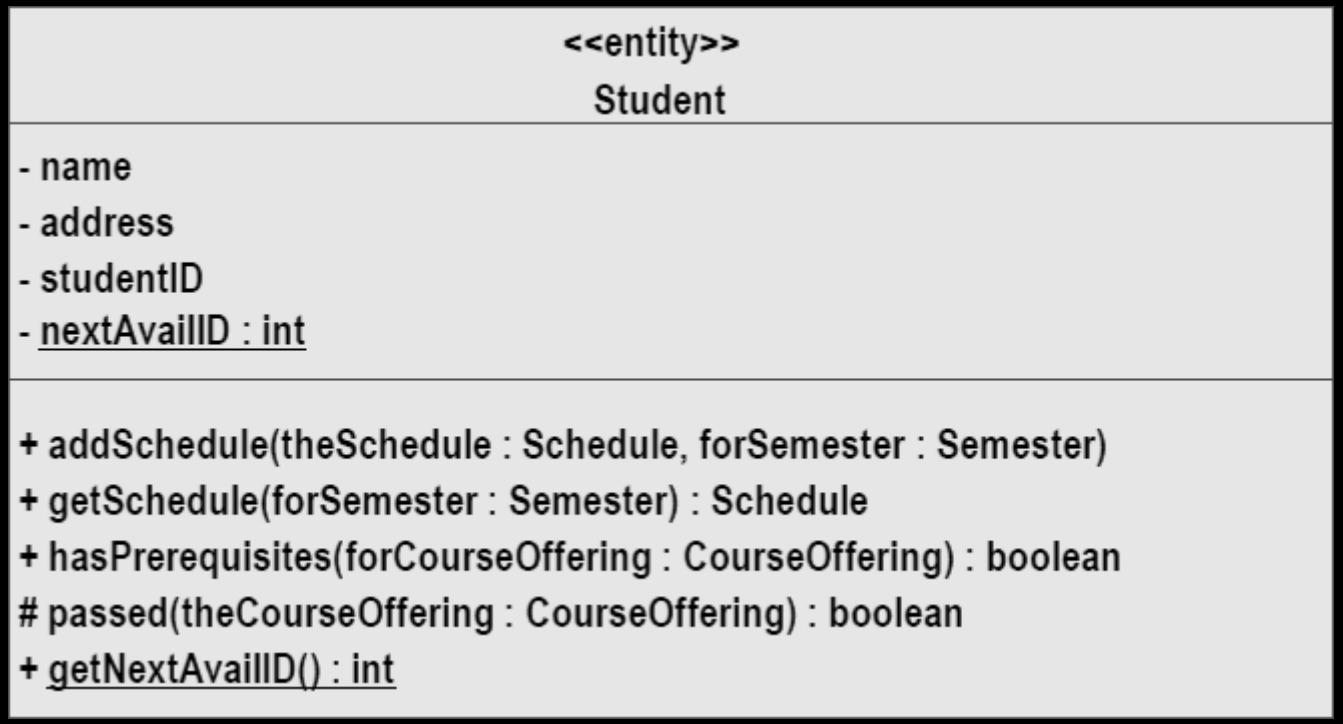


Scope

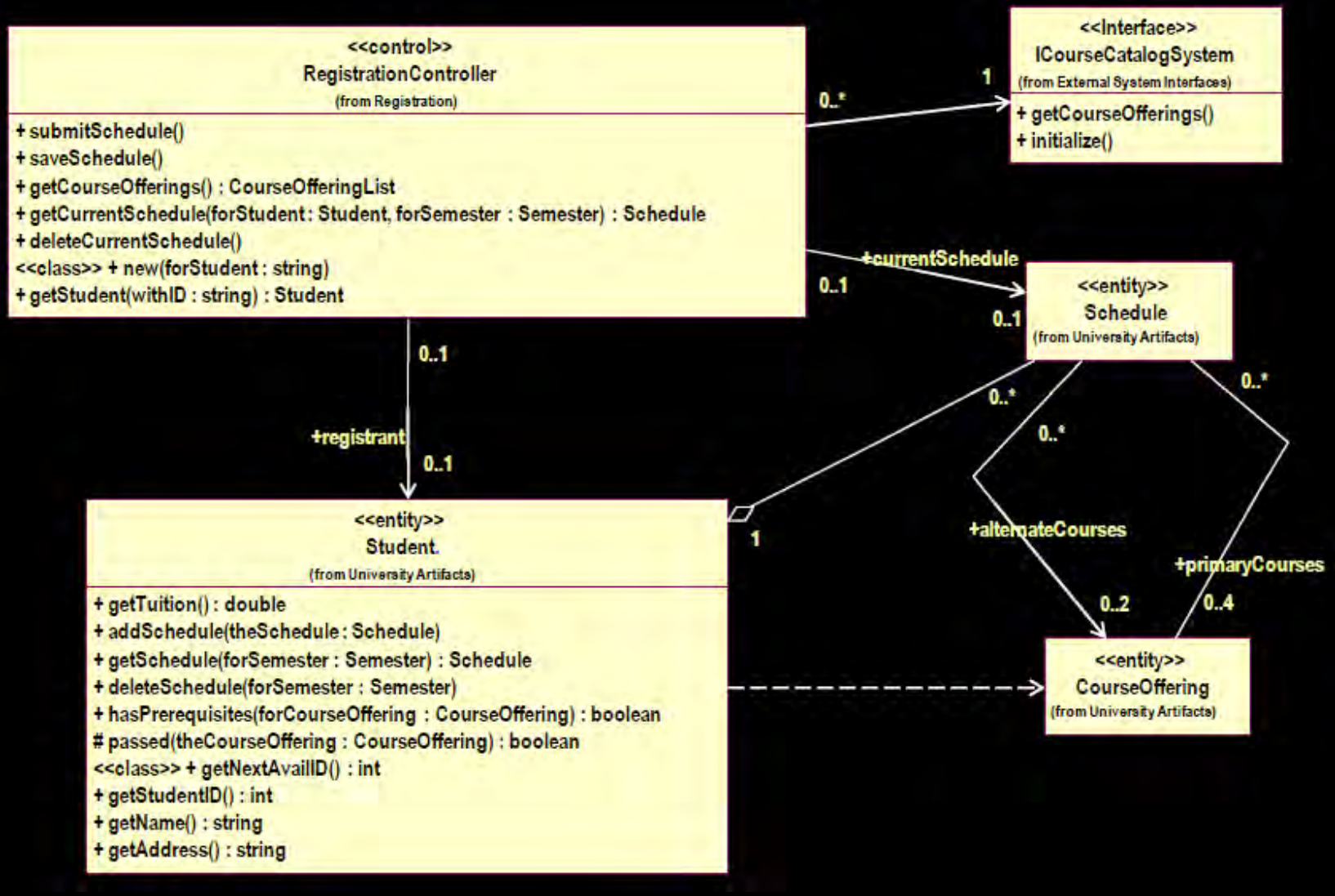
- Determines number of instances of the attribute/operation
  - **Instance: one instance for each class instance**
  - **Classifier: one instance for all class instances**
- Classifier scope is denoted by underlining the attribute/operation name



Example: Scope



Example: Define Operations



Define Methods

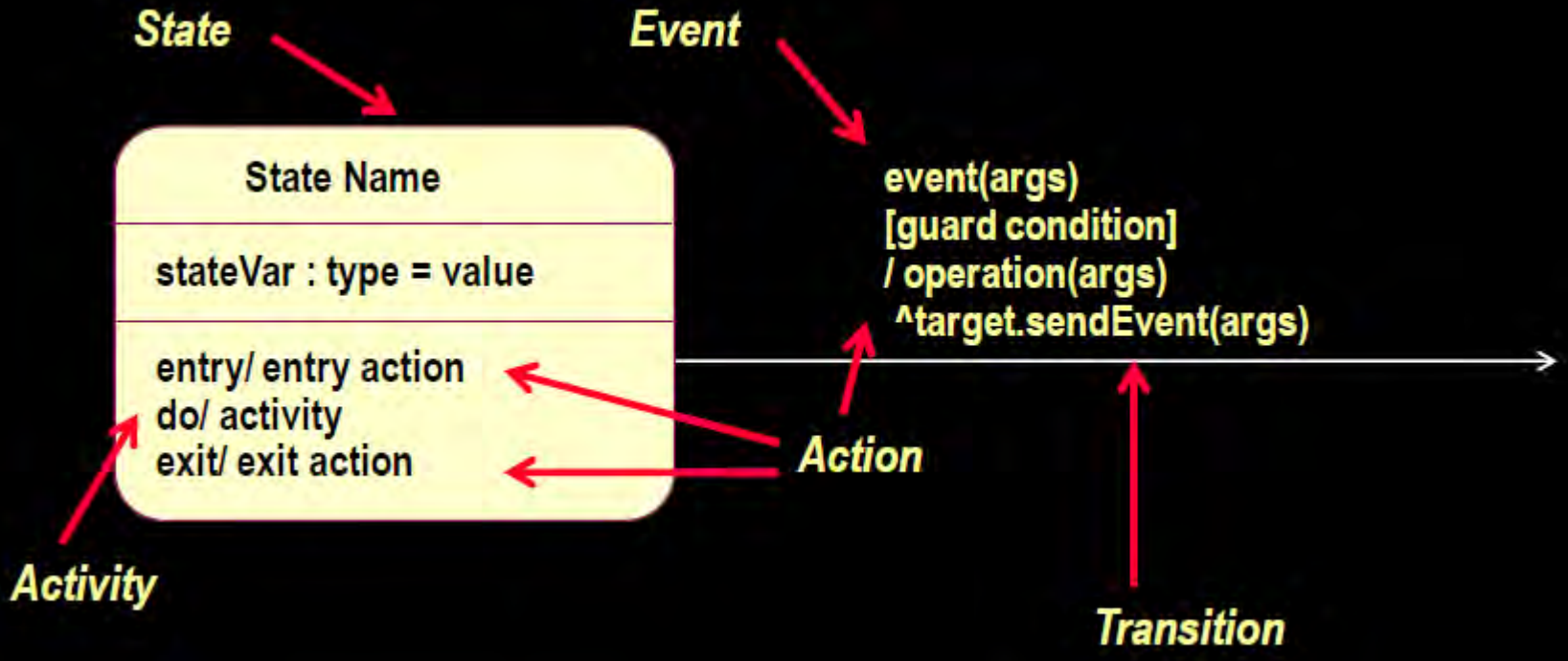
- What is a method?
  - **Describes operation implementation**
- Purpose
  - **Define special aspects of operation implementation**
- Things to consider:
  - **Special algorithms**
  - **Other objects and operations to be used**
  - **How attributes and parameters are to be implemented and used**
  - **How relationships are to be implemented and used**

Define States

- Purpose
  - Design how an object's state affects its behavior
  - Develop state charts to model this behavior
- Things to consider :
  - Which objects have significant state?
  - How to determine an object's possible states?
  - How do statecharts map to the rest of the model?

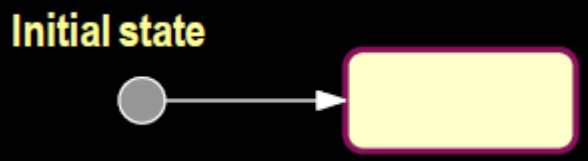
What is a Statechart?

- A directed graph of states (nodes) connected by transitions(directed arcs)
- Describes the life history of a reactive

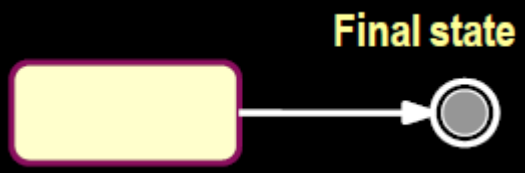


Special States

- Initial state
  - The state entered when an object is created
  - Mandatory
  - Can only have one initial state



- Final state
  - Indicates the object‘s end of lif
  - Optional
  - May have more than one Final



Identify and Define the States

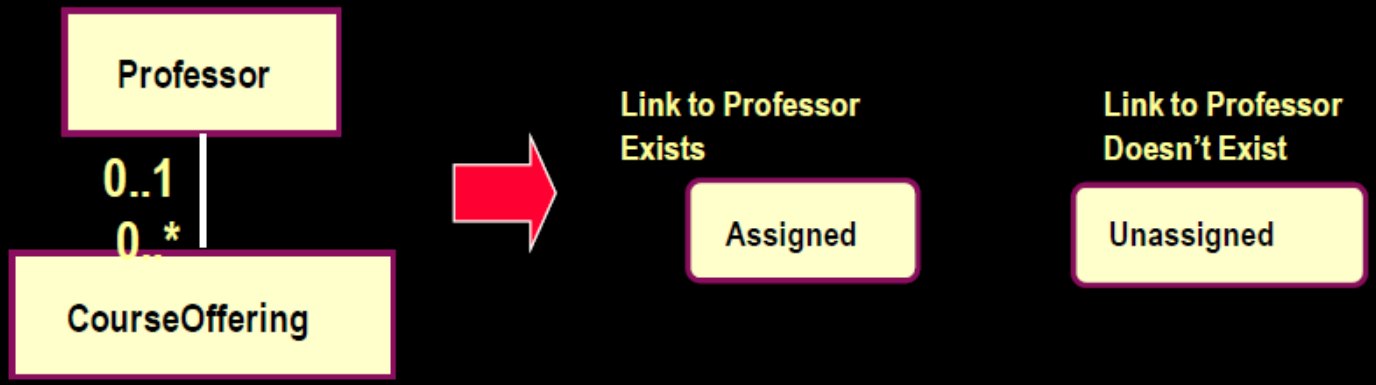
- Significant, dynamic attributes

The maximum number of students per course offering is 10

numStudents < 10 → Open

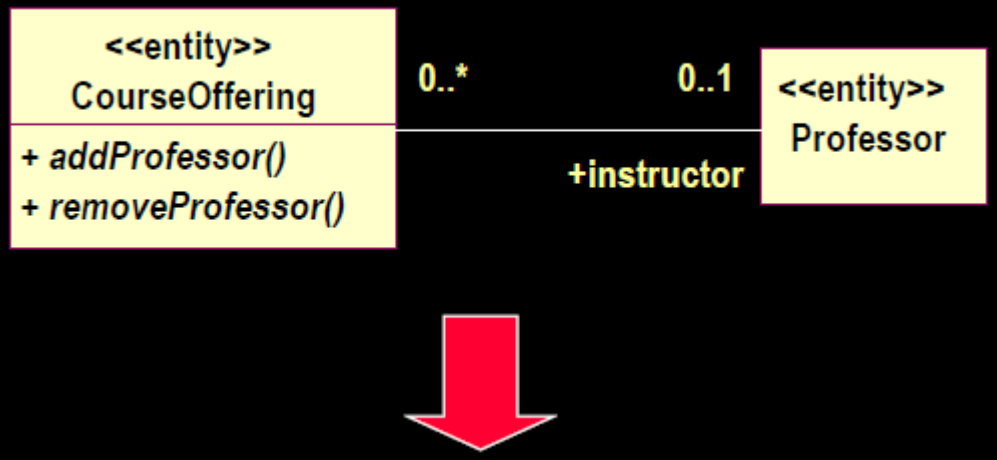
numStudents >= 10 → Closed

- Existence and non-existence of certain links



Identify the Events

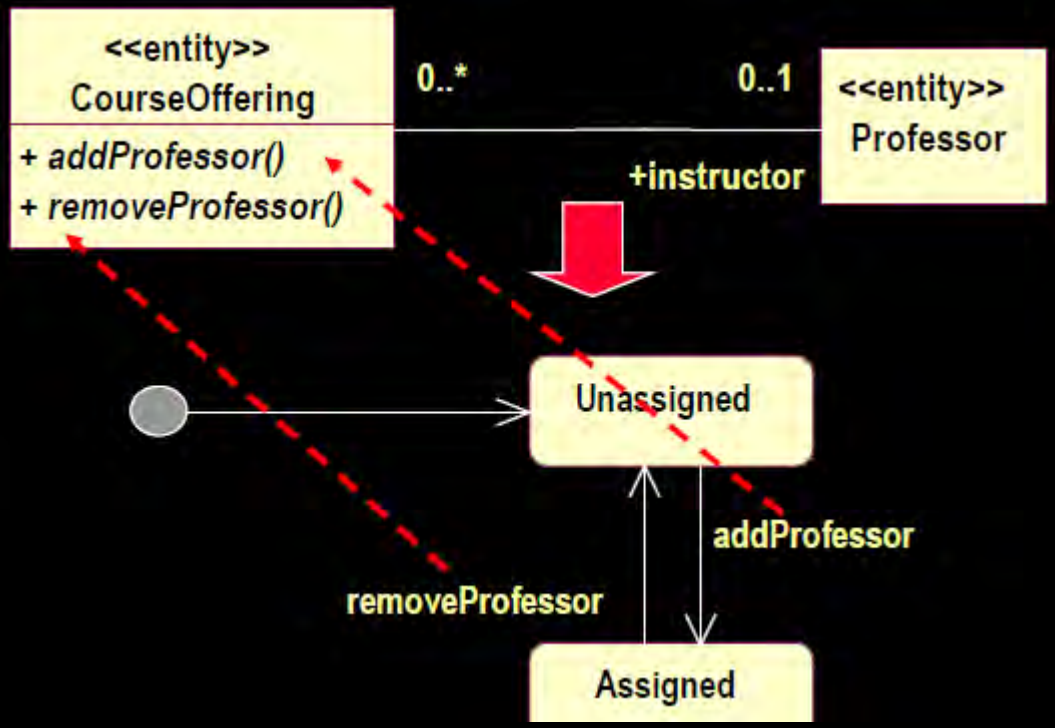
- Look at the class interface operations



Events: addProfessor, removeProfessor

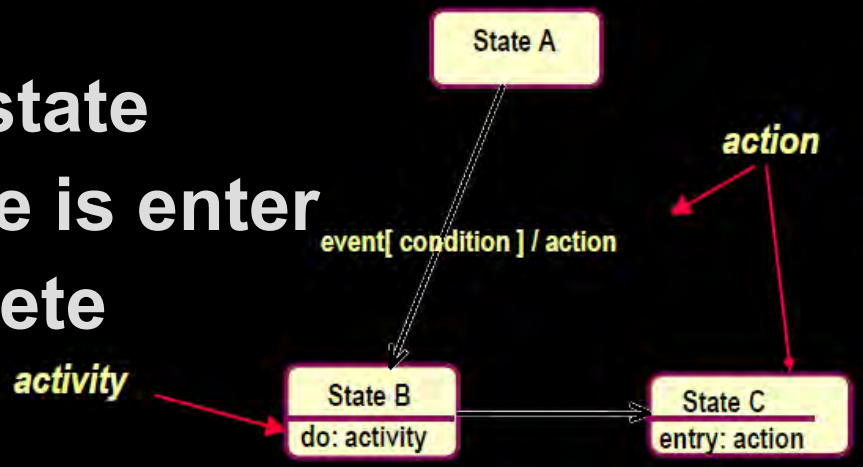
Identify the Transitions

- For each state, determine what events cause transitions to what states, including guard conditions, when needed
- Transitions describe what happens in response to the receipt of an event

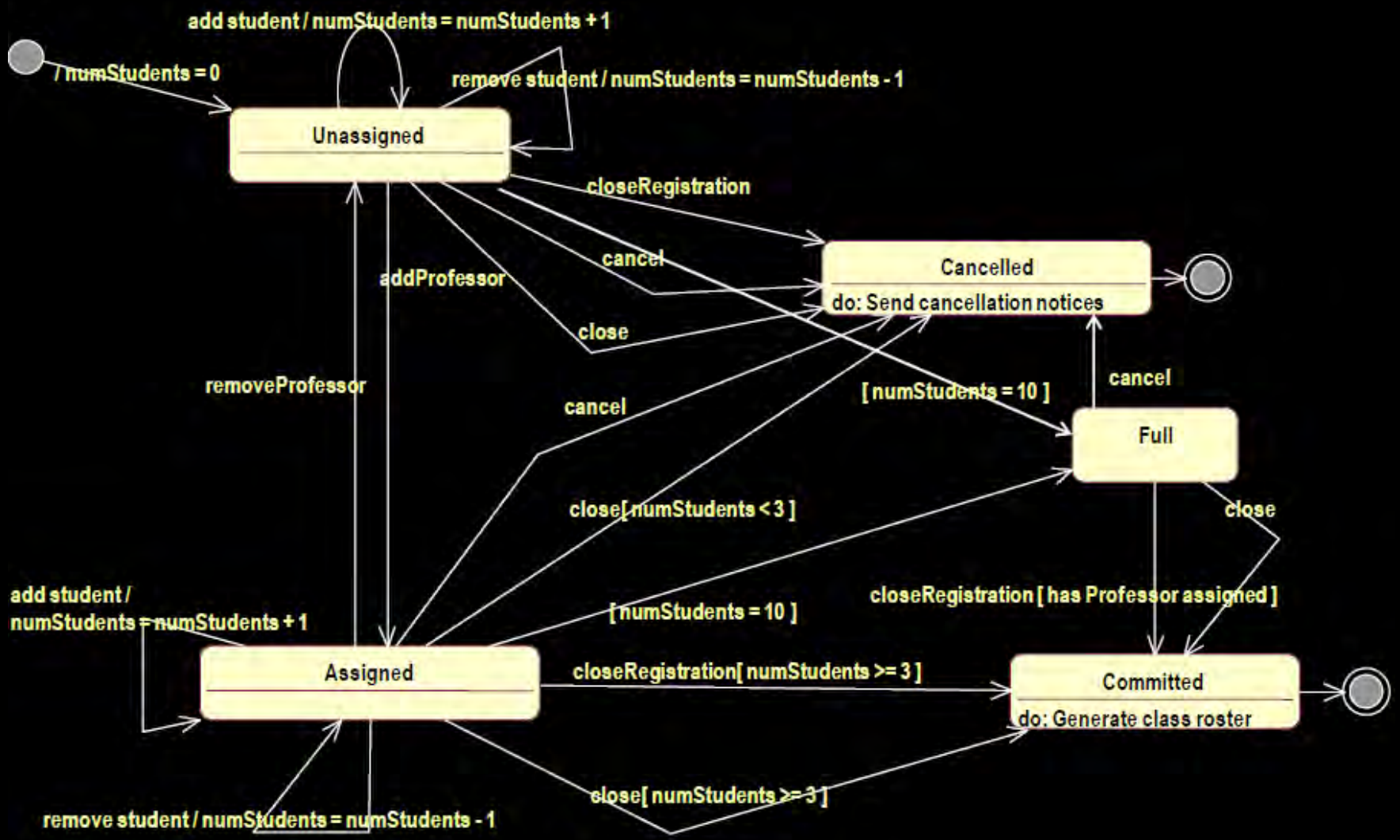


Add Activities and Actions

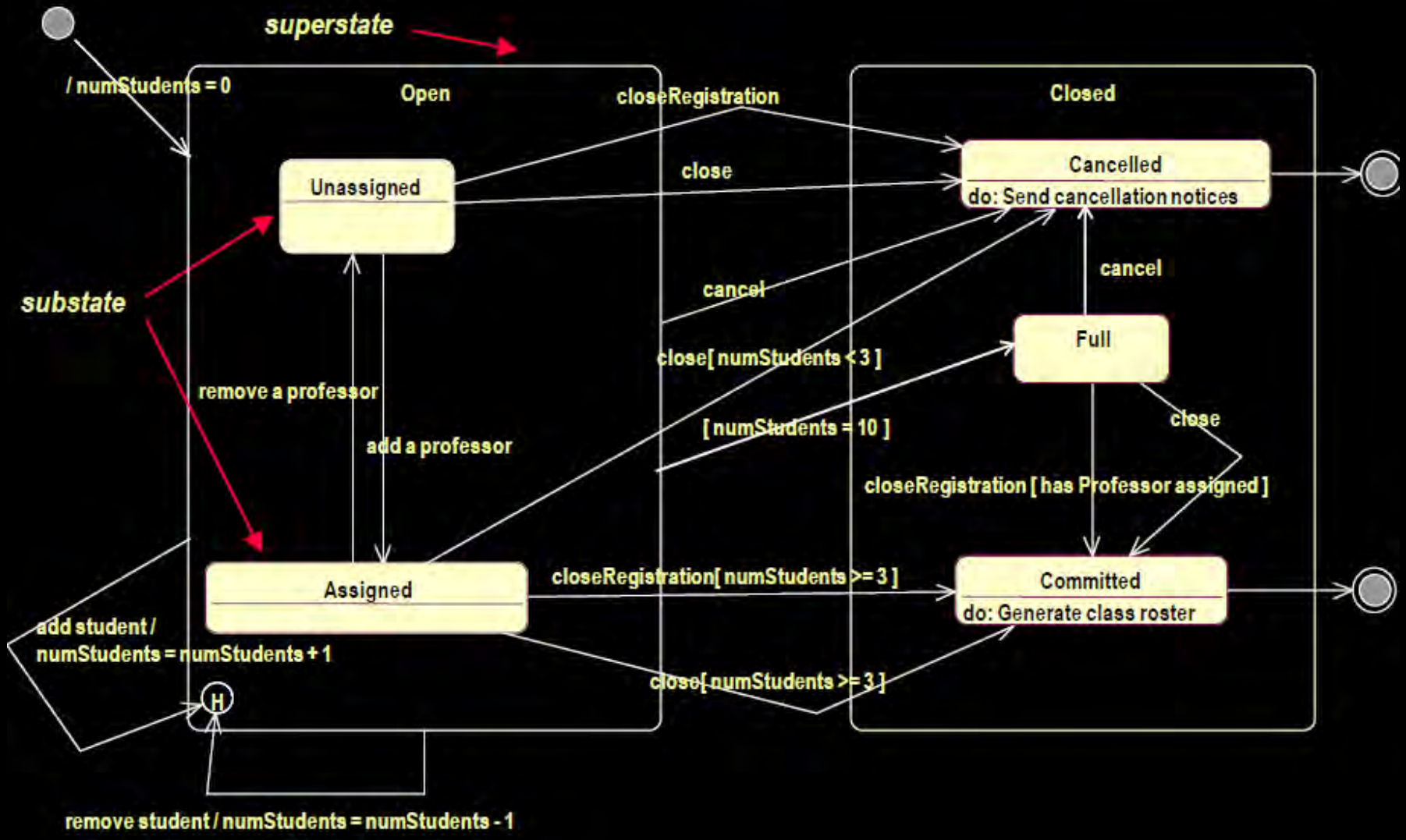
- Activities
  - Associated with a state
  - Start when the state is enter
  - Take time to complete
  - Interruptible
- Actions
  - Associated with a transition
  - Take an insignificant amount of time to complete
  - Non-interruptible



Example: Statechart



Example: Statechart With Nested States and History

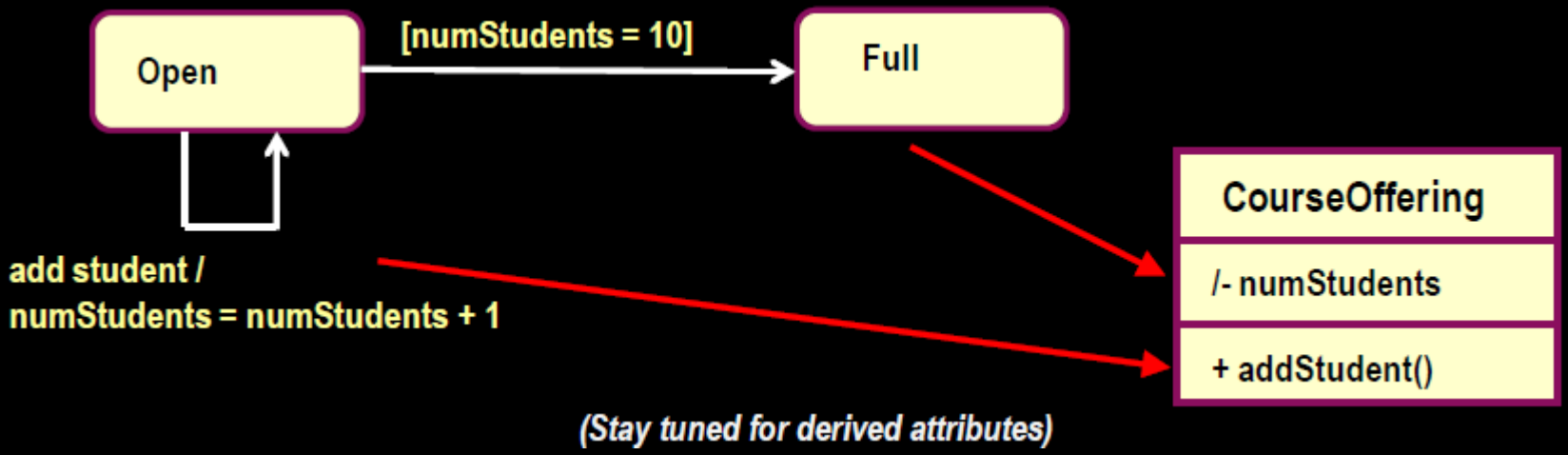


Which Objects Have Significant State?

- Objects whose role is clarified by state transitions
- Complex use cases that are state-controlled
- It is not necessary to model all objects
  - **Objects with straightforward mapping to implementation**
  - **Objects that are not state-controlled**
  - **Objects with only one computational state**

How Do Statecharts Map to the Rest of the Model?

- Events may map to operations
- Methods should be updated with state specific information
- States are often represented using attributes
  - **This serves as input into the "Define Attributes" step**



Attributes: How Do You Find Them?

- Examine method descriptions
- Examine states
- Any information the class itself needs to maintain

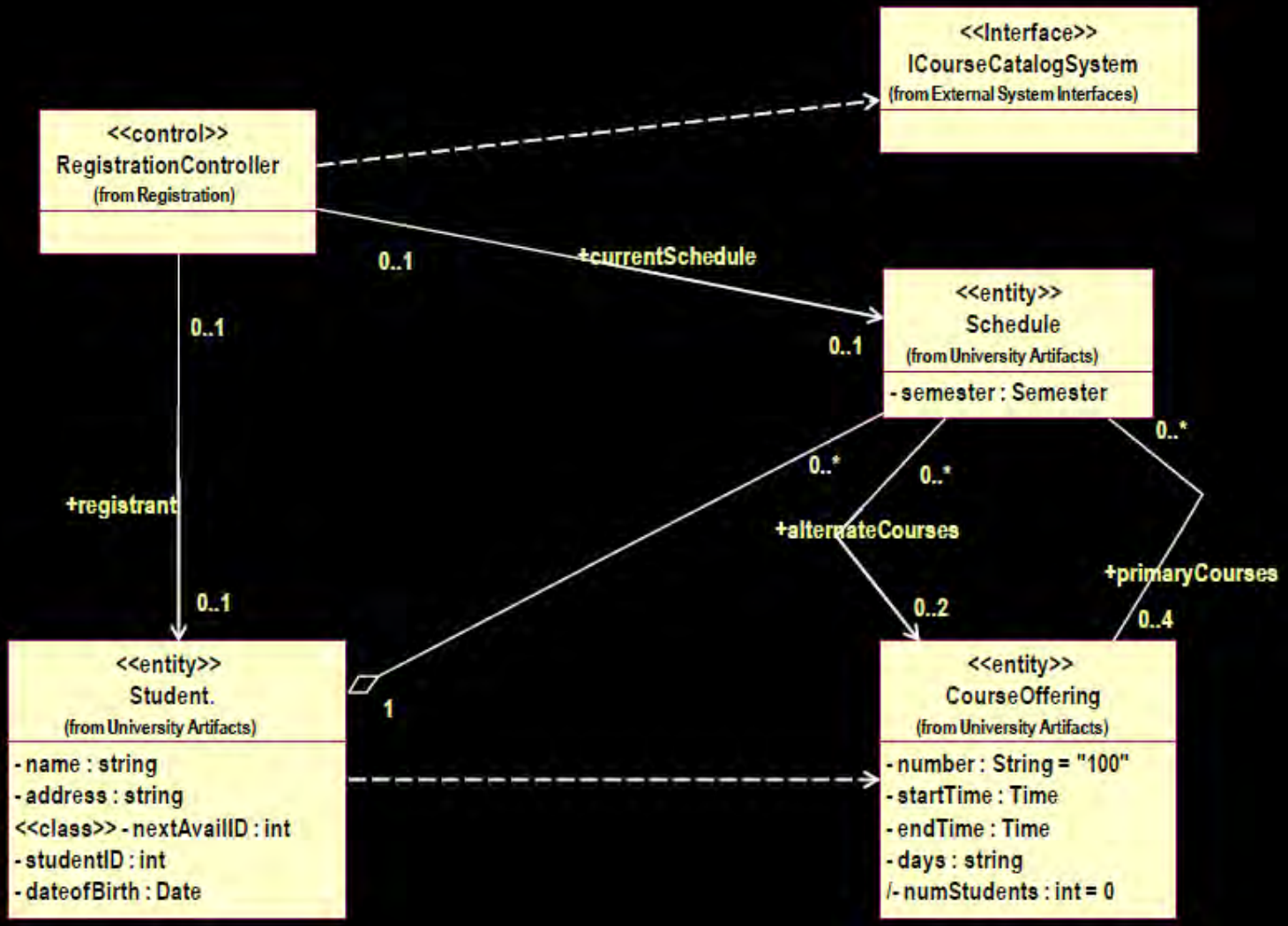
Attribute Representations

- Specify name, type, and optional default value
  - **attributeName : Type = Default**
- Follow naming conventions of implementation language and project
- Type should be an elementary data type in implementation language
  - **Built-in data type, user-defined data type, or user-defined class**
- Specify visibility
  - **Public: '+'**
  - **Private: '-'**
  - **Protected: '#'**

Derived Attributes

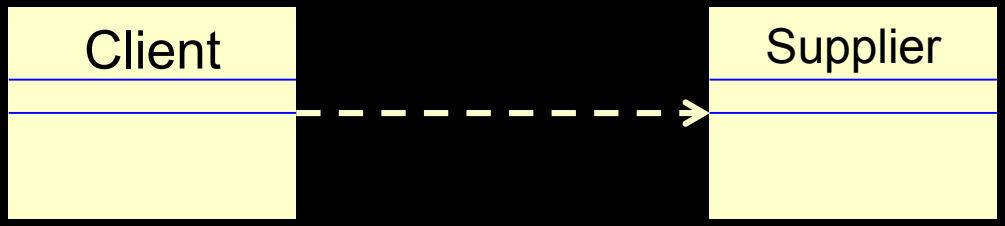
- What is a derived attribute?
  - **An attribute whose value may be calculated based on the value of other attribute(s)**
- When do you use them?
  - **When there is not enough time to re-calculate the value every time it is needed**
  - **Trade-off runtime performance vs. memory required**

Example: Define Attributes



Define Dependency

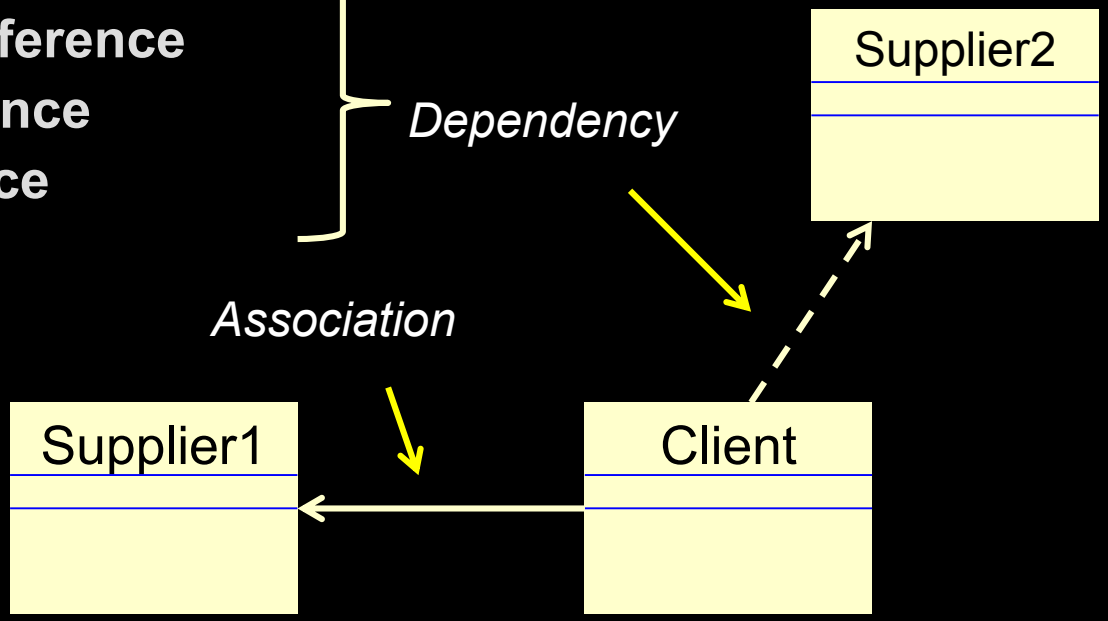
- What Is a Dependency?
  - **A relationship between two objects**



- Purpose
  - **Determine where structural relationships are NOT required**
- Things to look for :
  - **What causes the supplier to be visible to the client**

Dependencies vs. Associations

- Associations are structural relationships
- Dependencies are non-structural relationships
- In order for objects to "know each other" they must be visible
  - **Local variable reference**
  - **Parameter reference**
  - **Global reference**
  - **Field reference**



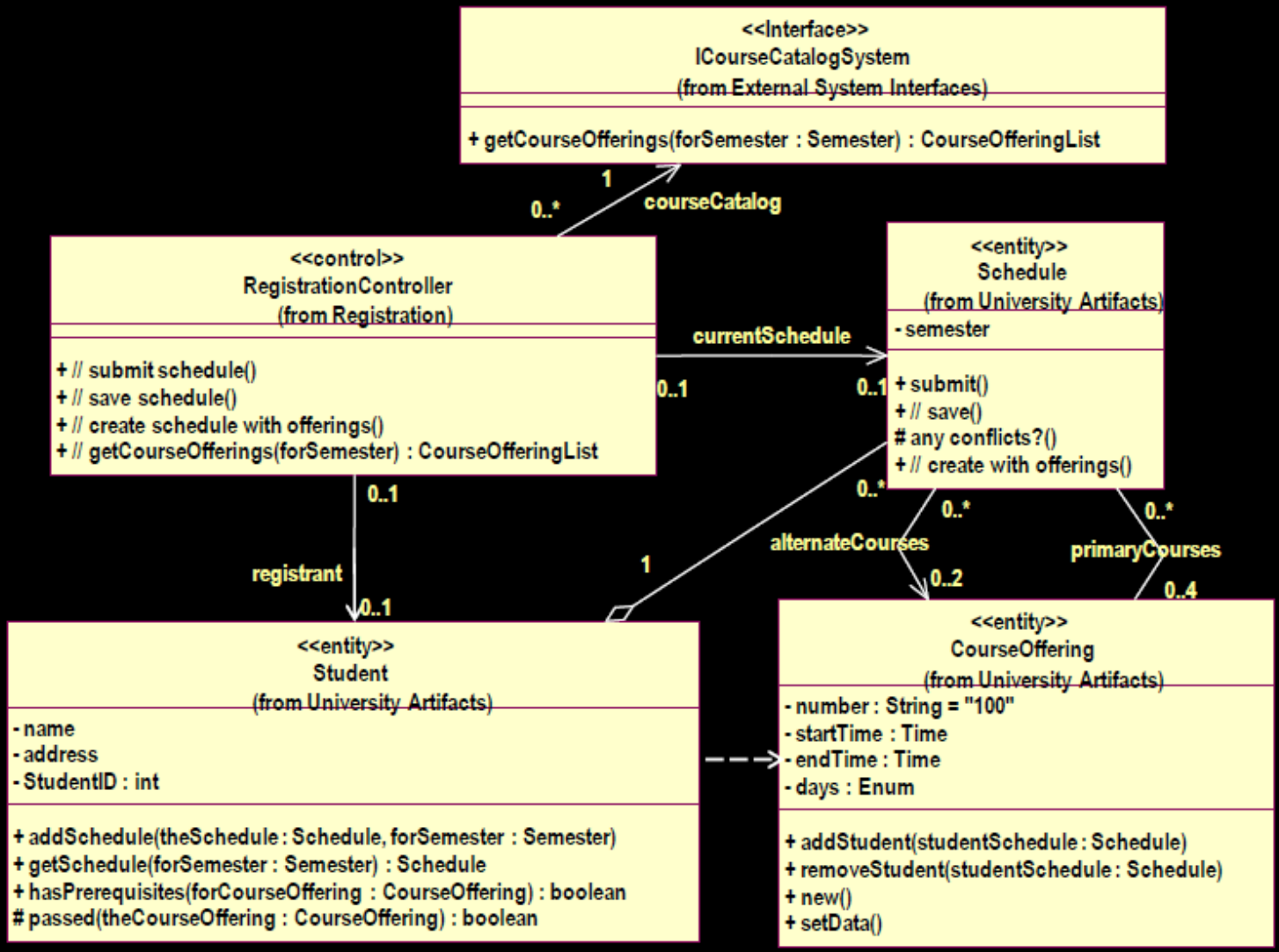
Associations vs. Dependencies in Collaborations

- An instance of an association is a link
  - **All links become associations unless they have global, local or parameter visibility**
  - **Context dependent relationship**
- Dependencies are transient links
  - **Have a limited duration**
  - **Context independent relationship**
  - **Summary relationship**

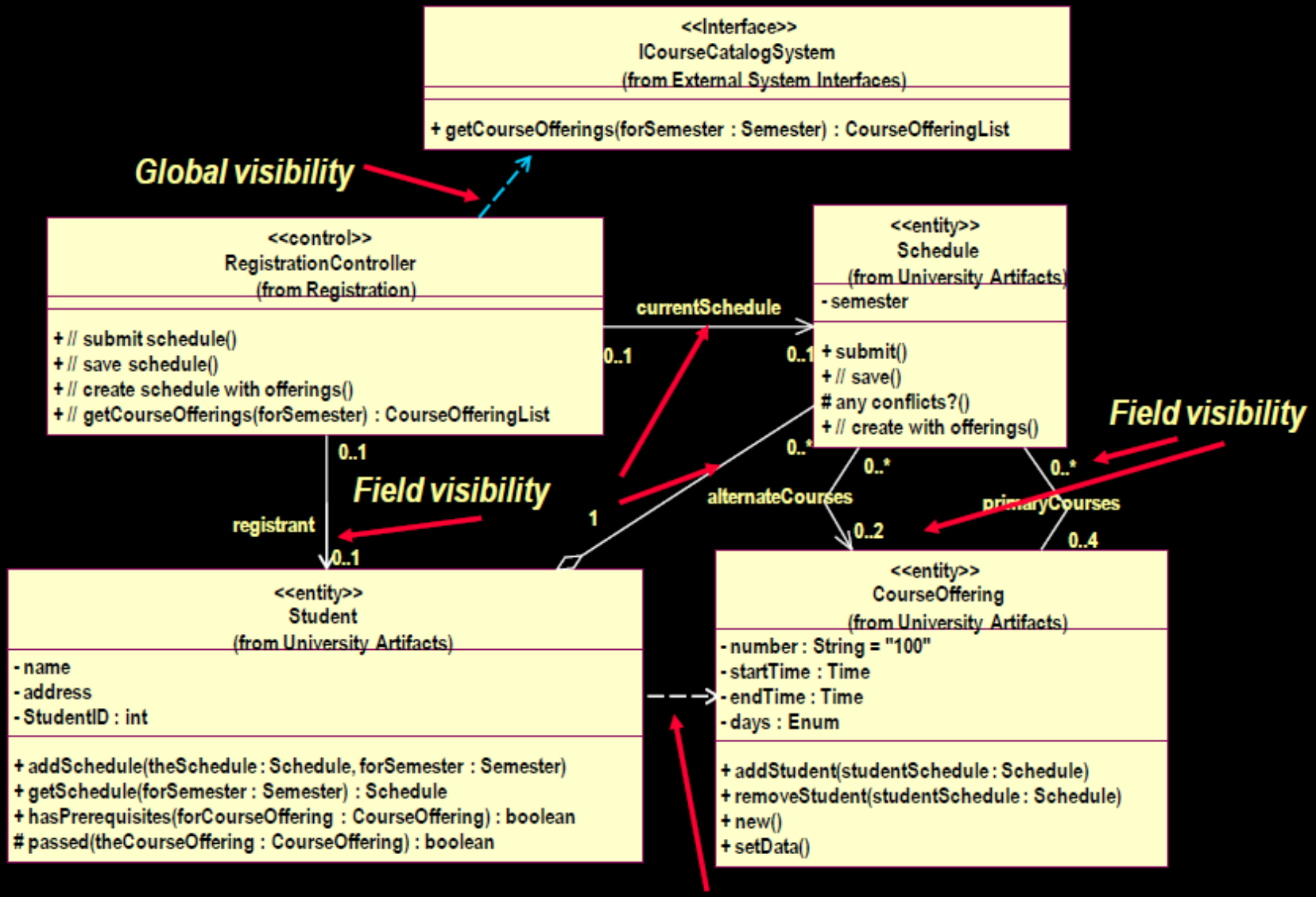
Identifying Dependencies: Considerations

- Permanent relationships - Association (field visibility)
- Transient relationships - Dependency
  - **Multiple objects share the same instance**
    - **Pass instance as a parameter (parameter visibility)**
    - **Make instance a managed global (global visibility)**
  - **Multiple objects don't share the same instance (local visibility)**
- How long does it take to create/destroy?
  - **Expensive? Use field, parameter, or global visibility**
  - **Strive for the lightest relationships possible**

Example: Define Dependencies (before)



Example: Define Dependencies (after)

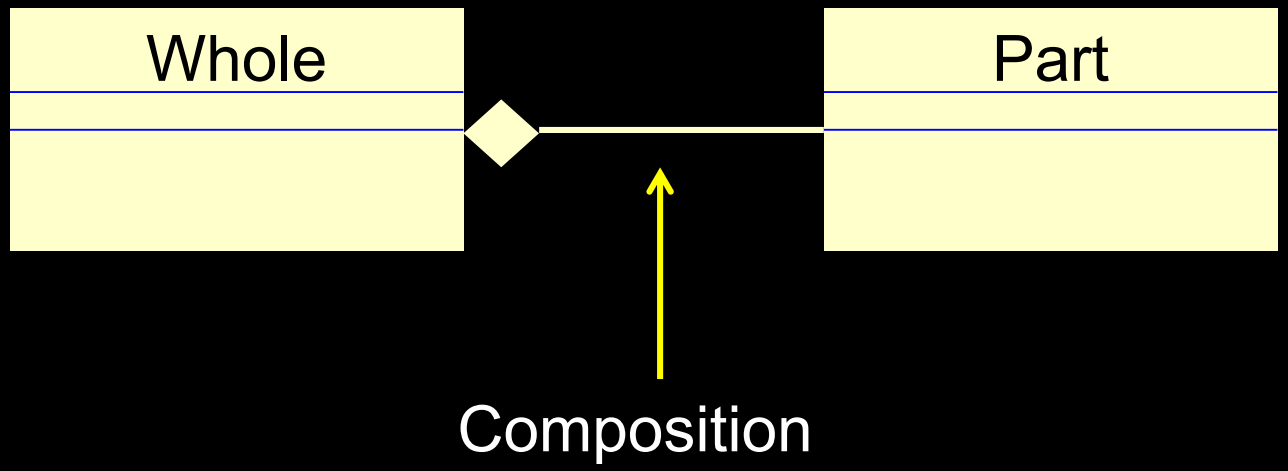


Define Associations

- Purpose
  - **Refine remaining associations**
- Things to look for :
  - **Association vs. Aggregation**
  - **Aggregation vs. Composition**
  - **Attribute vs. Association**
  - **Navigability**
  - **Association class design**
  - **Multiplicity design**

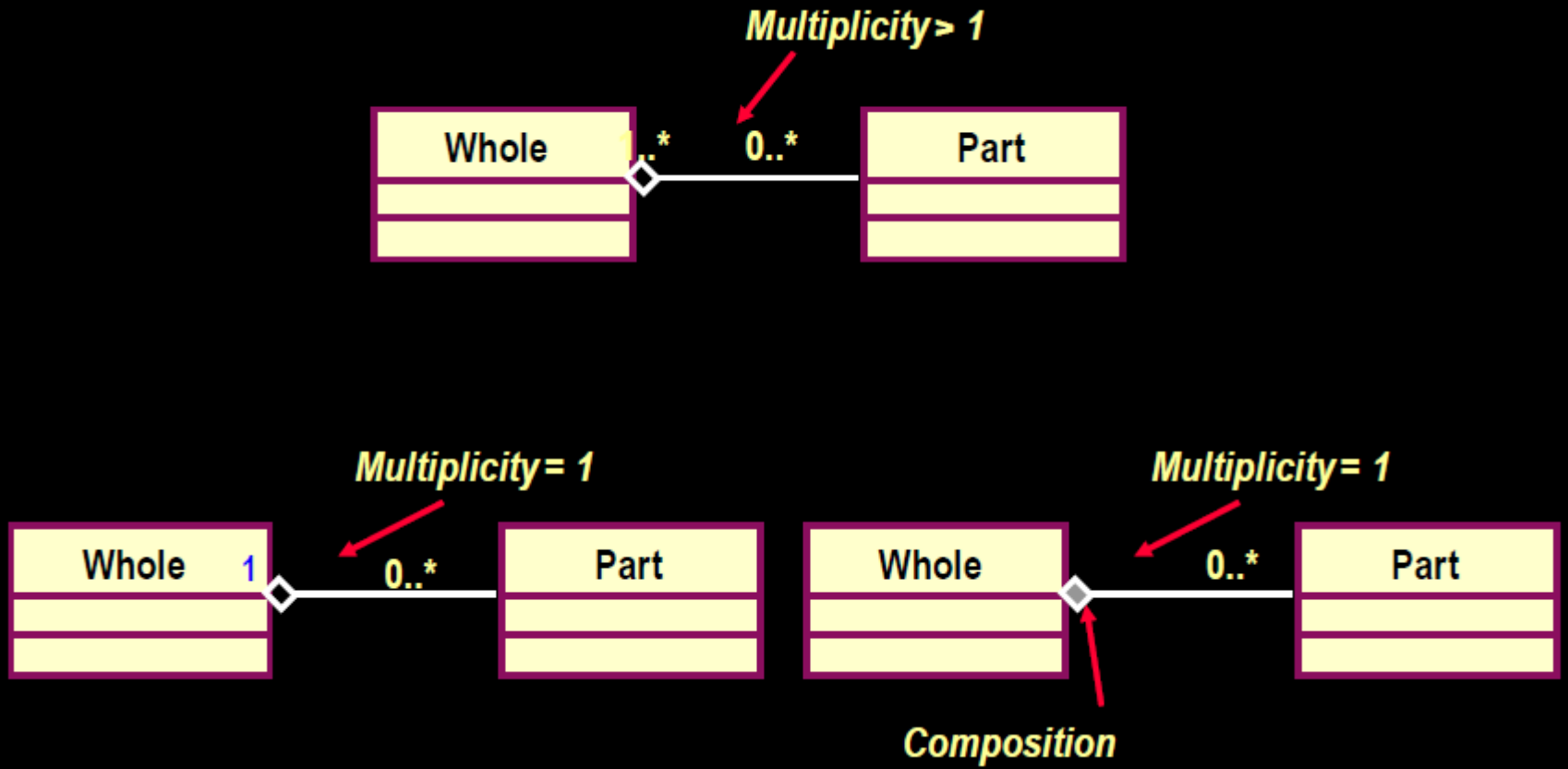
What is Composition?

- A form of aggregation with strong ownership and coincident lifetimes
  - **The parts cannot survive the whole/aggregate**



Aggregation: Shared Vs. Non-shared

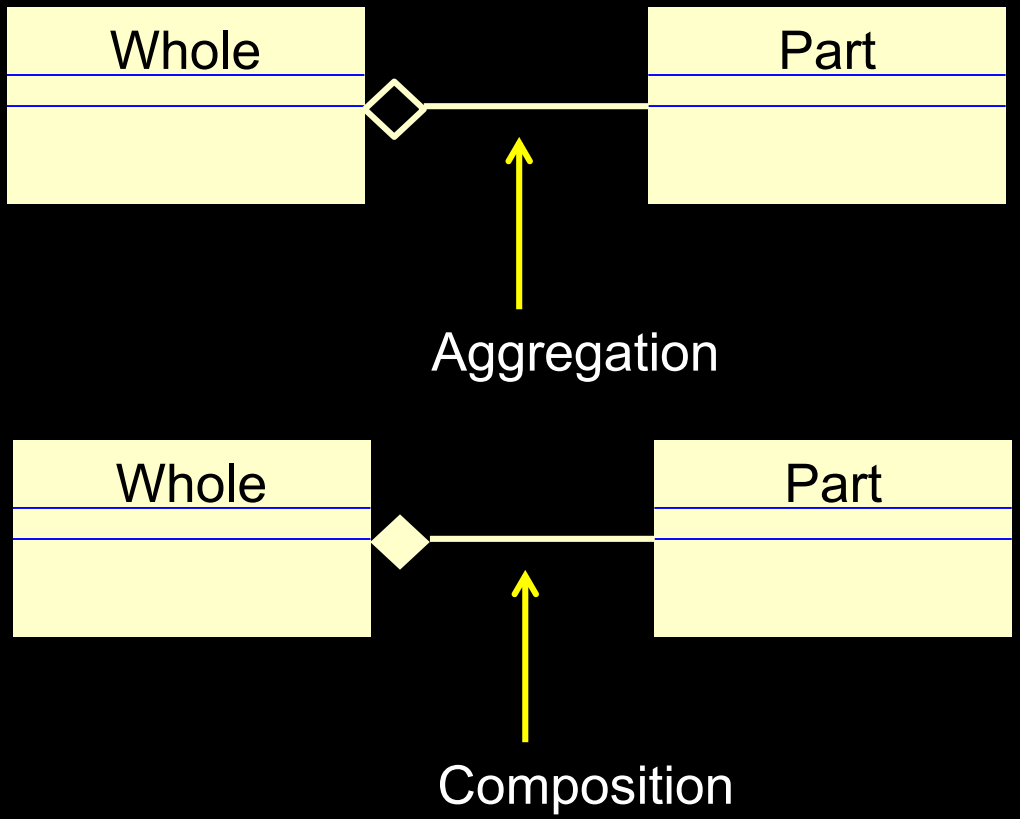
- Shared Aggregation



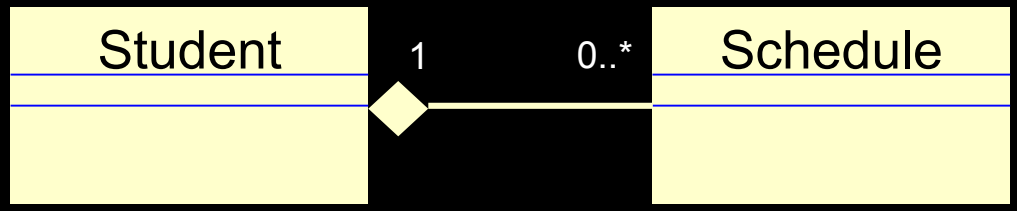
By definition, composition is non-shared aggregation

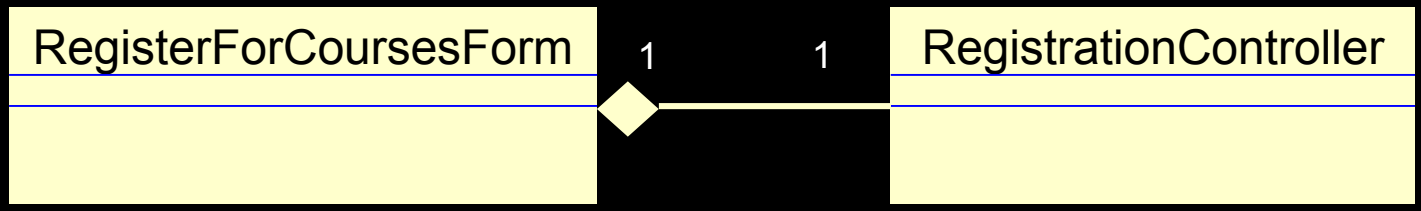
Aggregation or Composition?

- Consideration
  - **Lifetimes of Class1 and Class2**



Example: Composition

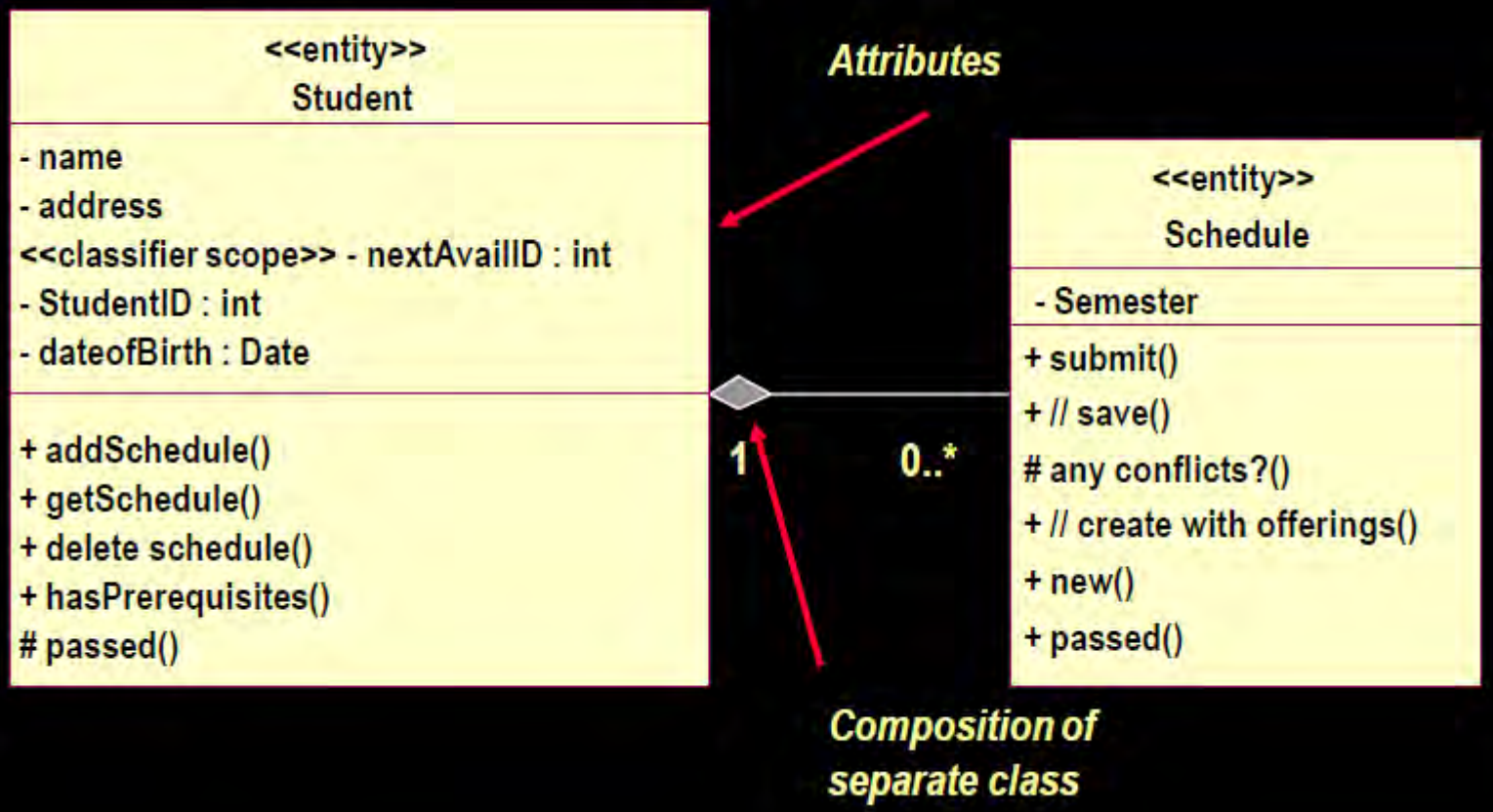




Attributes Vs Composition

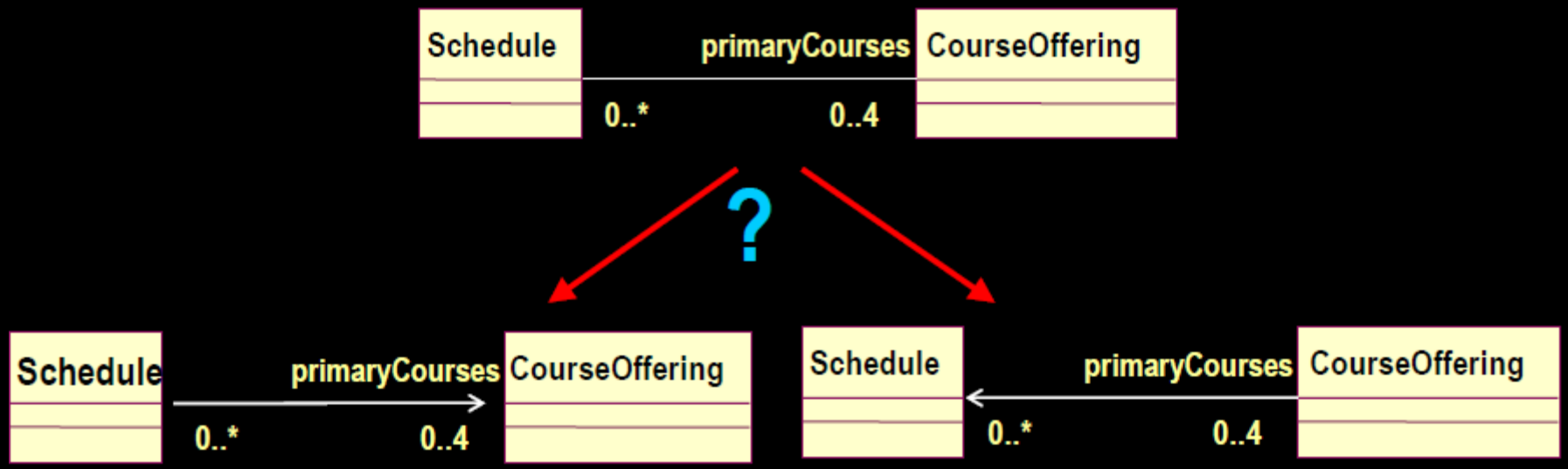
- Use composition when
  - **Properties need independent identities**
  - **Multiple classes have the same properties**
  - **Properties have a complex structure and properties of their own**
  - **Properties have complex behavior of their own**
  - **Properties have relationships of their own**
- Otherwise use attributes

Example: Attributes Vs Composition



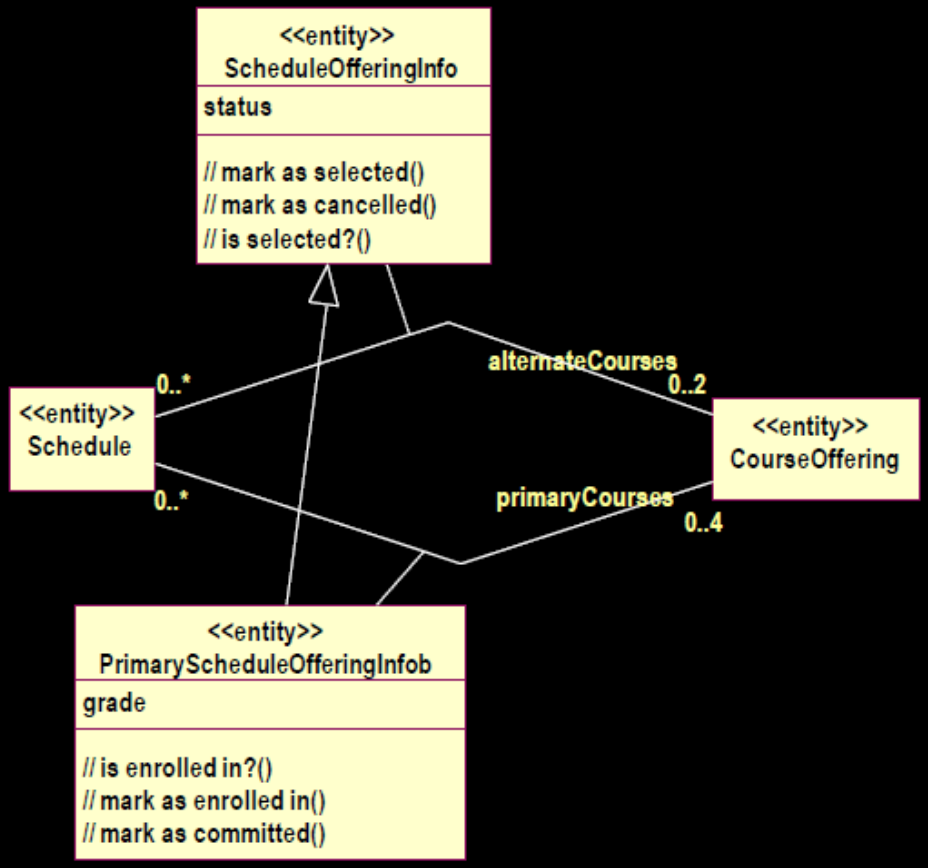
Navigability: Which Directions Are Really Needed?

- Explore interaction diagrams
- Even when both directions seem required, one may work
  - **Navigability in one direction is infrequent**
  - **Number of instances of one class is small**

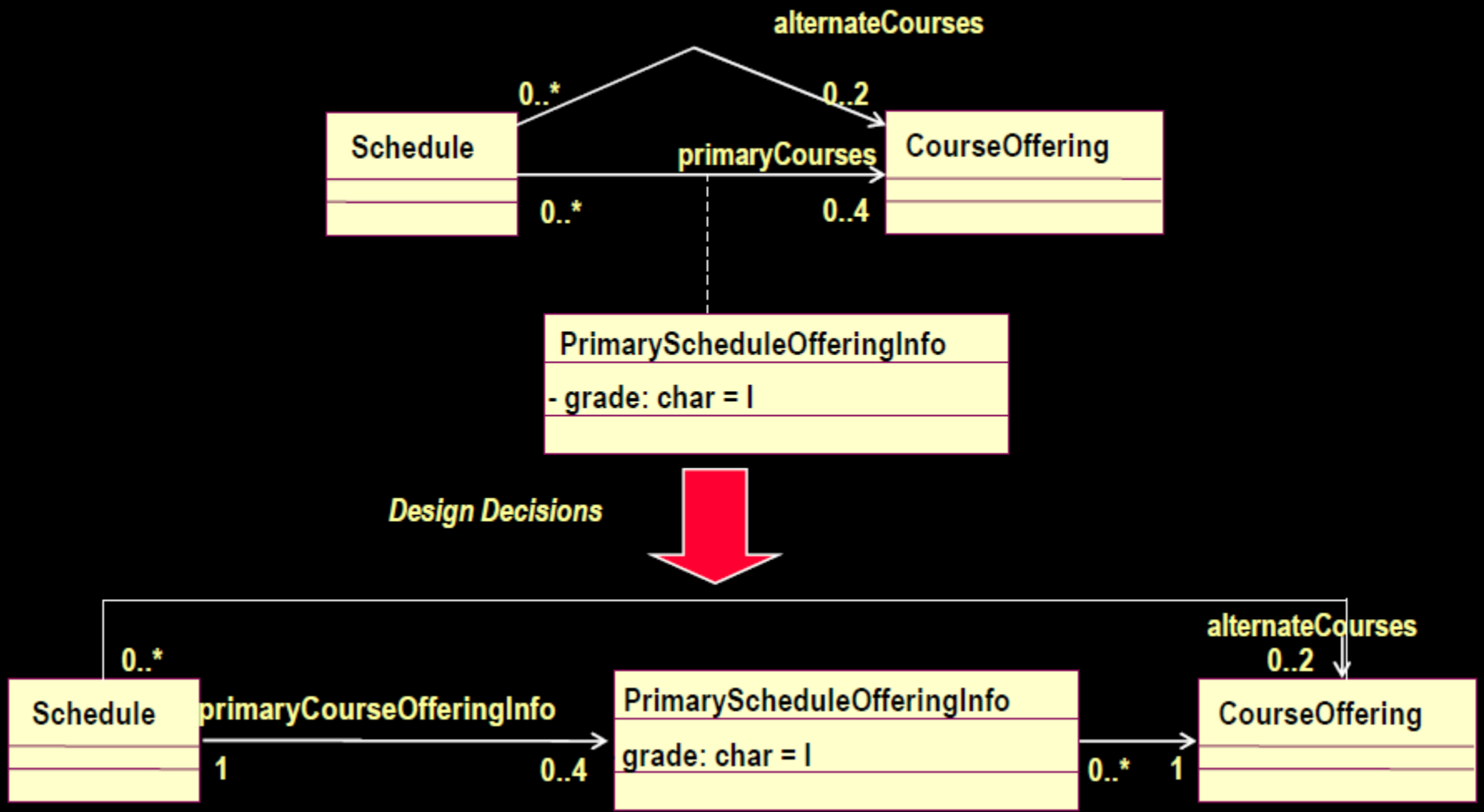


Association Class

- A class “attached” to an association
- Contains properties of a relationship
- One instance per link

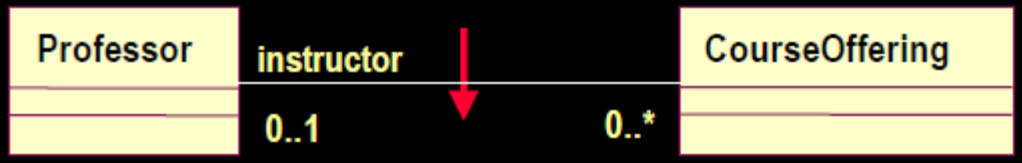


Example: Association Class Design



Multiplicity Design

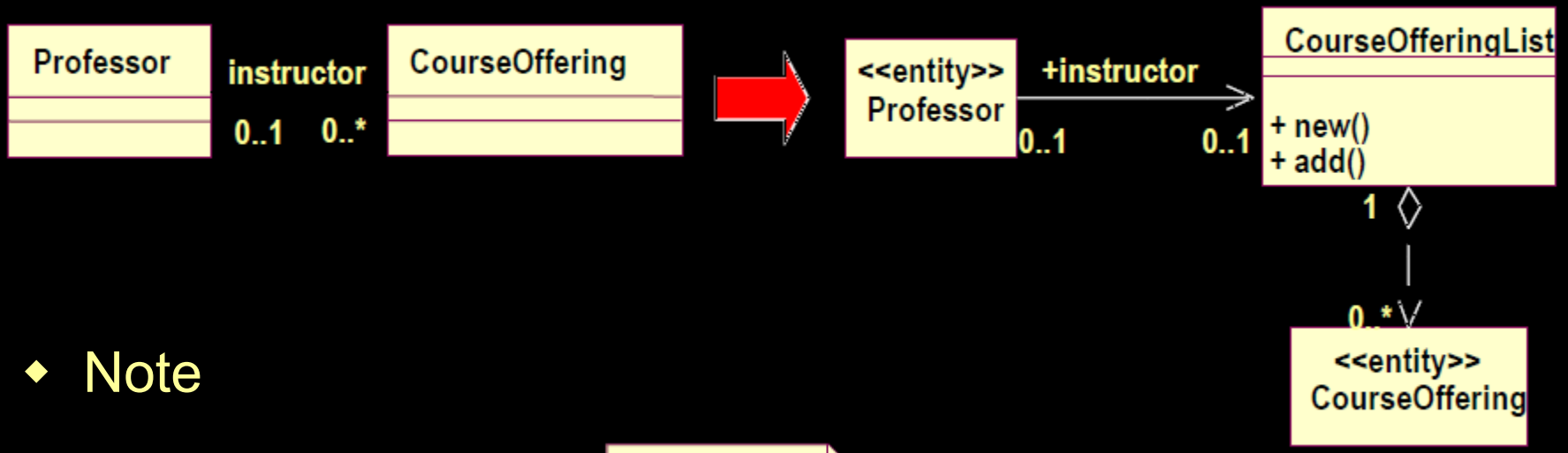
- Multiplicity = 1, or Multiplicity = 0..1
  - **May be implemented directly as a simple value or pointer**
  - **No further "design" is required**



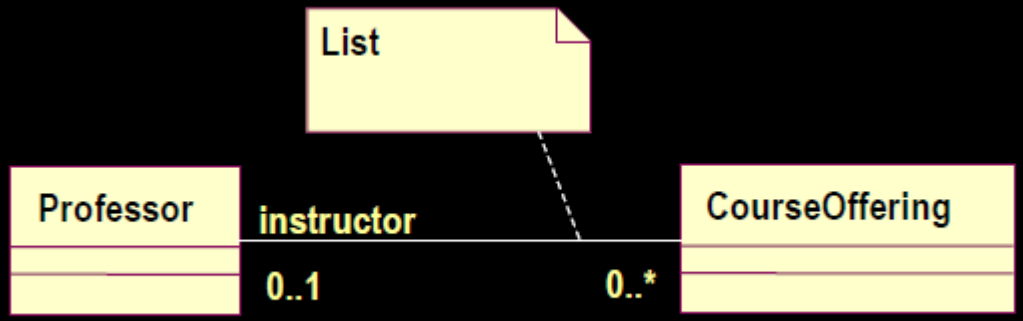
- Multiplicity > 1
  - **Cannot use a simple value or pointer**
  - **Further "design" may be required**

Multiplicity Design Options

- Explicit modeling of a container class

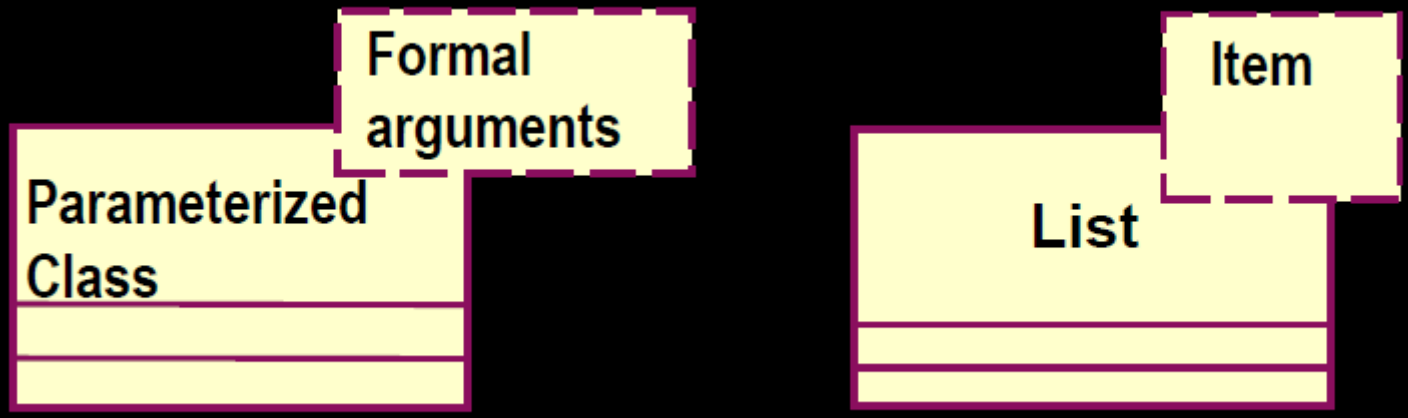


- Note

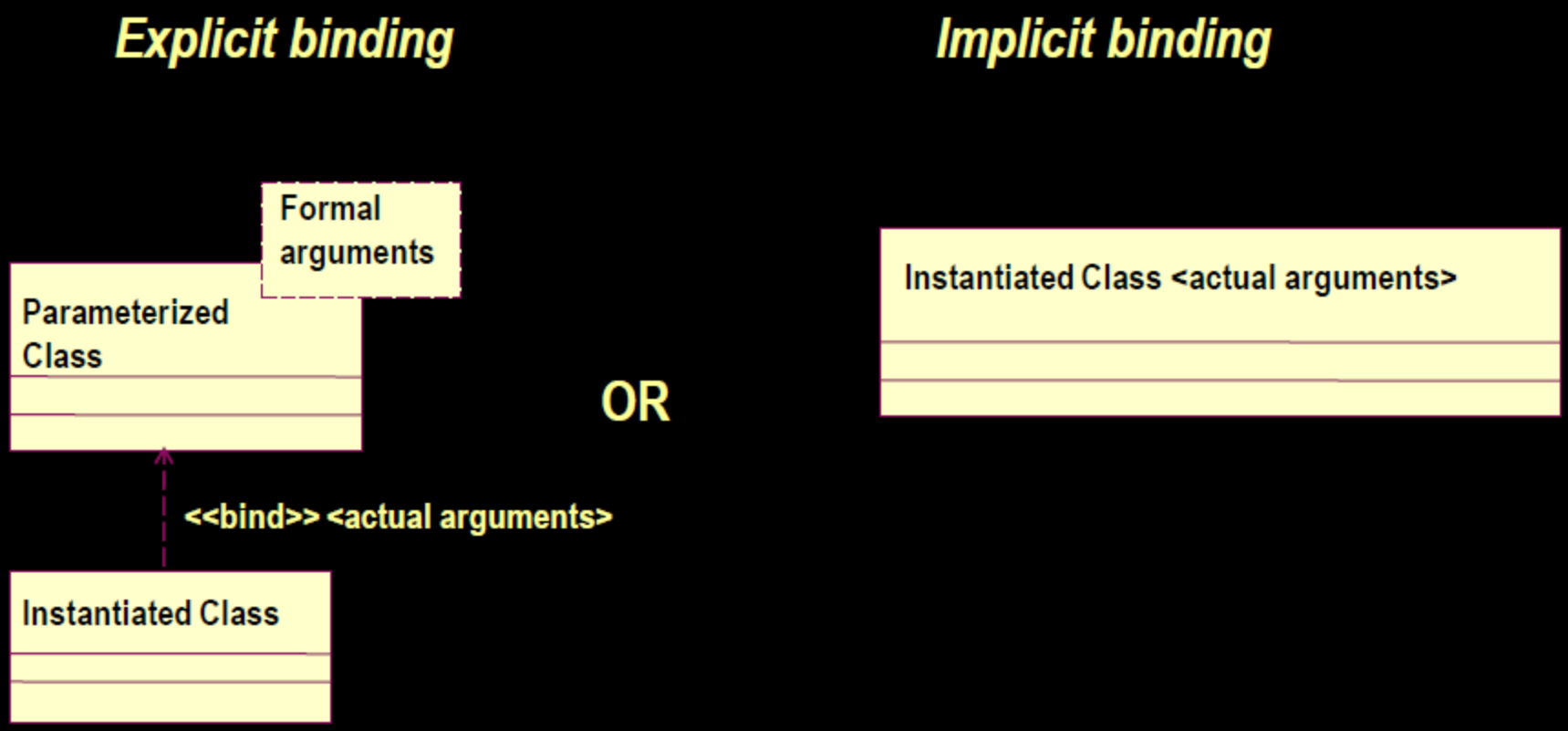


What is a Parameterized Class (template)?

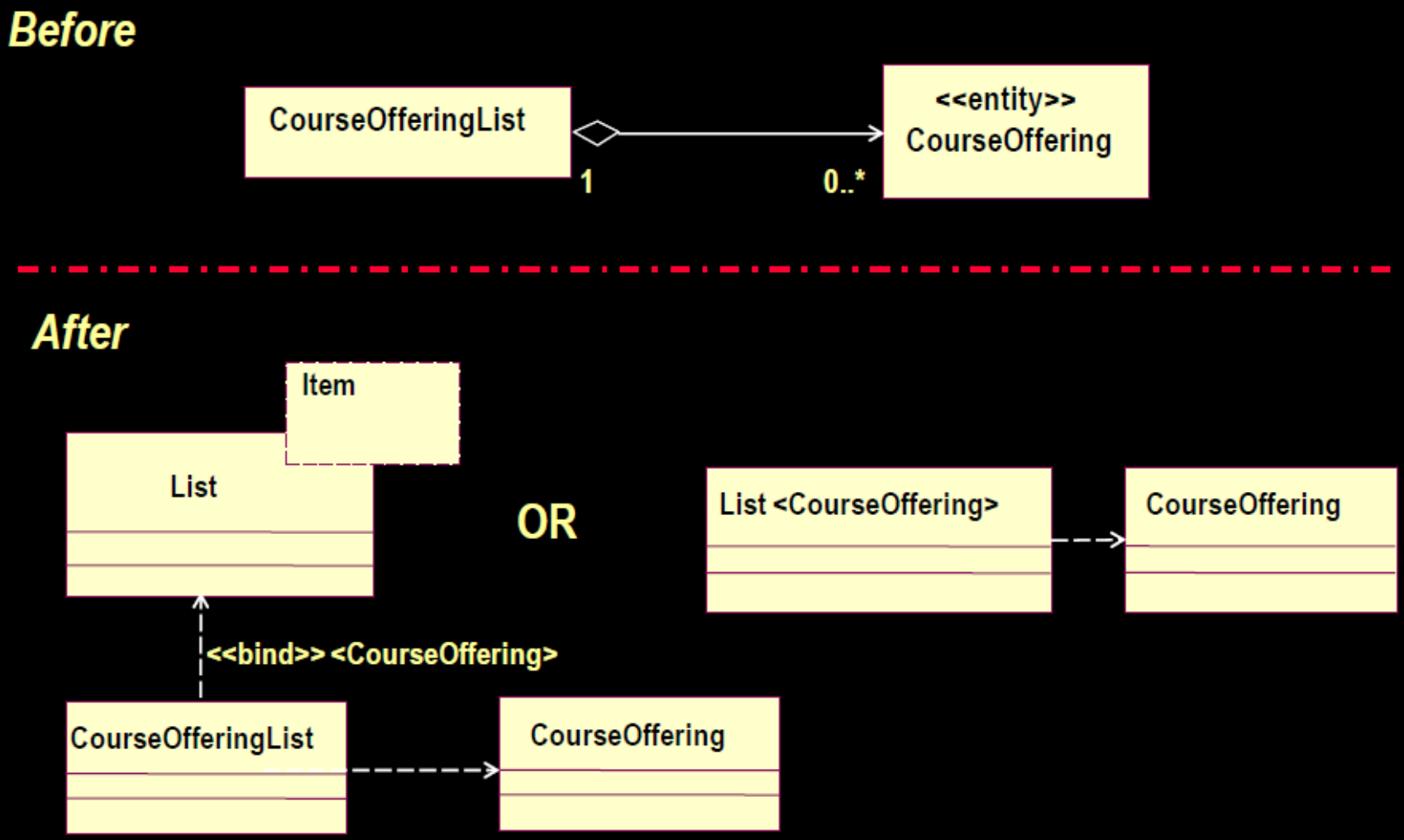
- A class definition which defines other classes
- Often used for container classes
  - **Some common container classes:**
    - **Sets, lists, dictionaries, stacks, queues, …**



Instantiating a Parameterized Class



Example: Instantiating a Parameterized Class

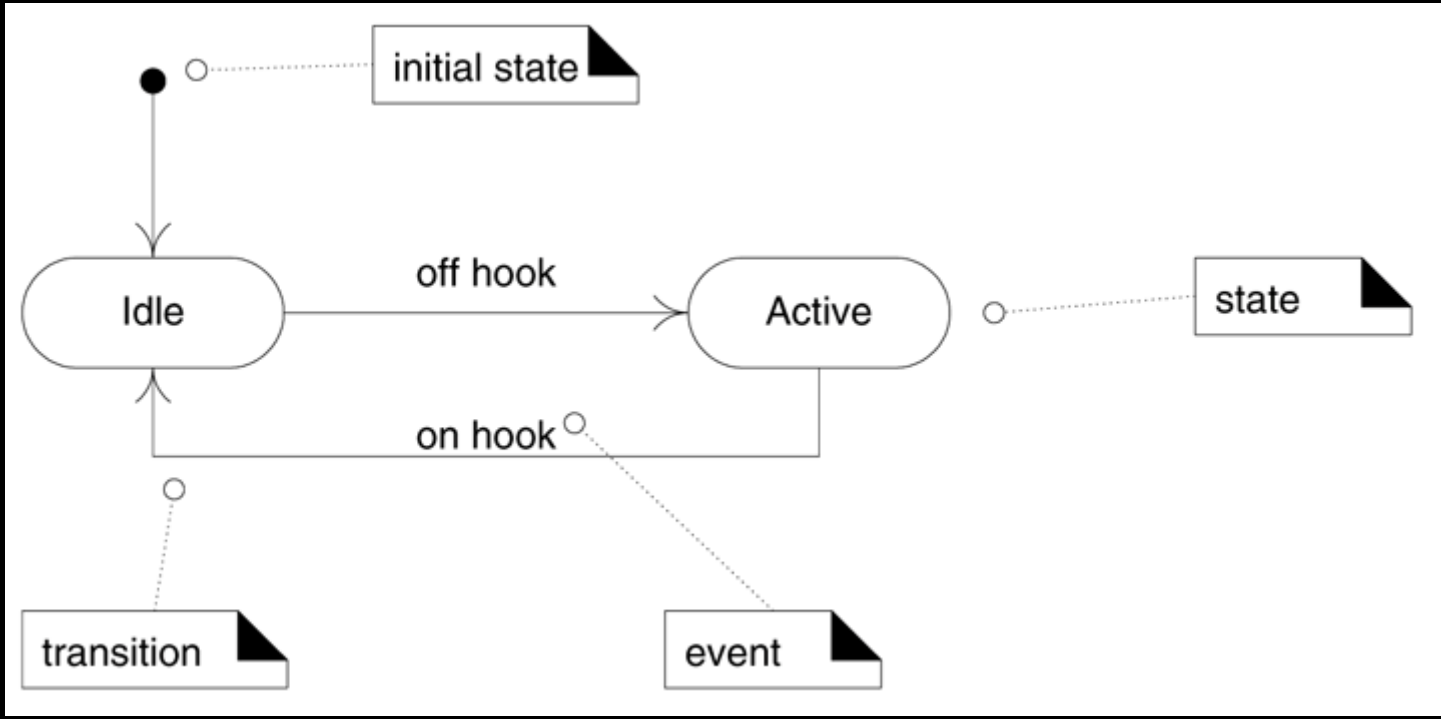


Multiplicity Design: Optionality

- If a link is optional, make sure to include an operation to test for the existence of the link

UML State machine diagram

- A state machine diagram illustrates the interesting events and states of an object, and the behavior of an object in reaction to an event



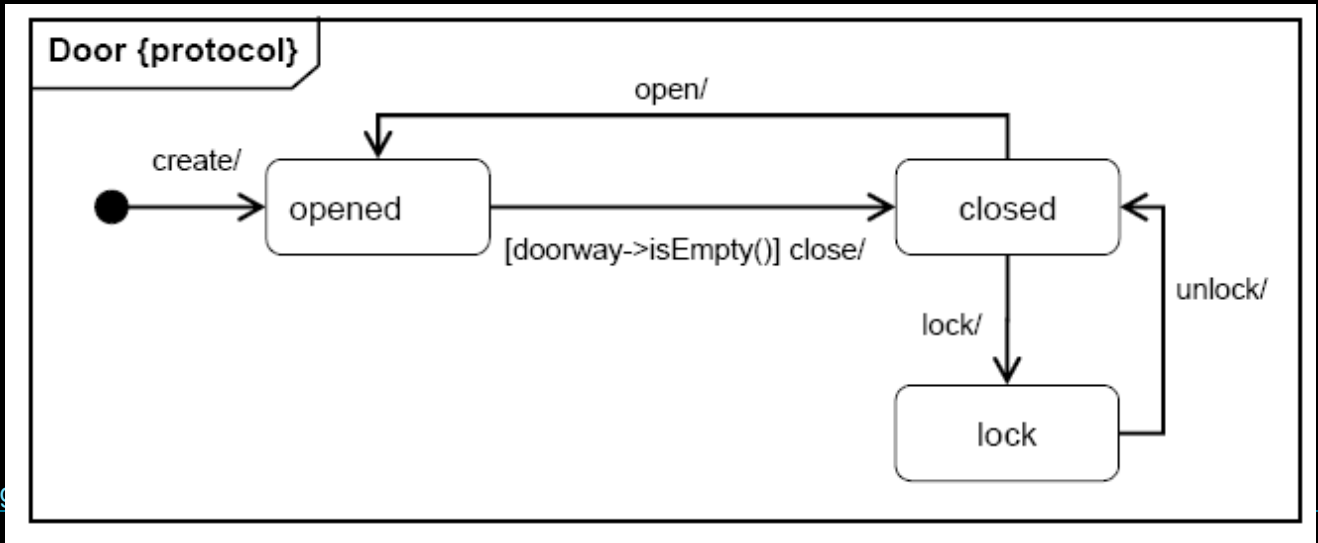
State machine diagram for a telephone

State machine

- A state machine is a graph of states and transitions
  - **State machines can be used to model the behavior of a class, subsystem, or entire application**
- UML has two types of state machines:
  - **Behavioral state machines**
    - **Show the behavior of model elements such as objects.**
    - **A behavioral state machine represents a specific implementation of an element.**
  - **Protocol state machines**
    - **Show the behavior of a protocol.**
    - **Protocol state machines show how participants may trigger changes in a protocol's state and the corresponding changes in the system.**
    - **Protocol state machines aren't typically tied to a particular implementation, and they show the required protocol behavior.**

Protocol state machine

- protocol state machines are a specialization of behavioral state machines
  - **protocol state machines are not tied to an implementation and have restrictions on their transitions**
  - **Protocol state machines capture the behavior of a protocol, such as HTTP or a challenge-response speakeasy door.**
    - **They aren't tied to a particular implementation of a protocol; rather, they specify the state changes and events associated with a protocol-based communication.**



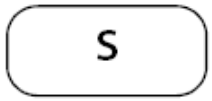
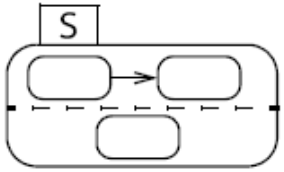
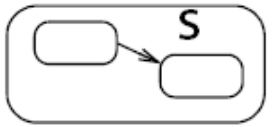
Definitions: Events

- An event is a significant or noteworthy occurrence
- Kinds of Events

Event Type	*Description*	*Syntax*
call event	Receipt of an explicit synchronous call request by an object	op (a:T)
change event	A change in value of a Boolean expression	**when** (exp)
signal event	Receipt of an explicit, named, asynchronous communication among objects	sname (a:T)
time event	The arrival of an absolute time or the passage of a relative amount of time	**after** (time)

Definitions: States

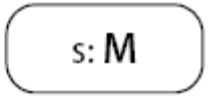
- A state is the condition of an object at a moment between events
- Kinds of States

State Kind	*Description*	*Notation*
simple state	A state with no substructure	
orthogonal state	A state that is divided into two or more regions. One direct substate from each region is concurrently active when the composite state is active.	
nonorthogonal state	A composite state that contains one or more direct substates, exactly one of which is active at one time when the composite state is active	

Kinds of States

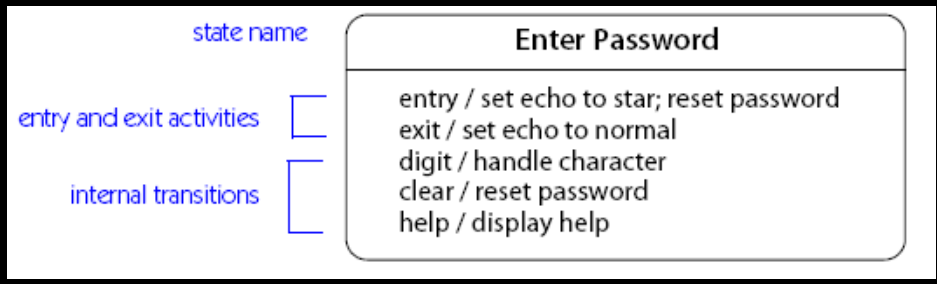
initial state	A pseudostate that indicates the starting state when the enclosing state is invoked	
final state	A special state whose activation indicates the enclosing state has completed activity	
terminate	A special state whose activation terminates execution of the object owning the state machine	
junction	A pseudostate that chains transition segments into a single run-to-completion transition	
choice	A pseudostate that performs a dynamic branch within a single run-to-completion transition	
history state	A pseudostate whose activation restores the previously active state within a composite state	

Kinds of States

State Kind	*Description*	*Notation*
submachine state	A state that references a state machine definition, which conceptually replaces the submachine state	
entry point	A externally visible pseudostate within a state machine that identifies an internal state as a target	
exit point	A externally visible pseudostate within a state machine that identifies an internal state as a source	

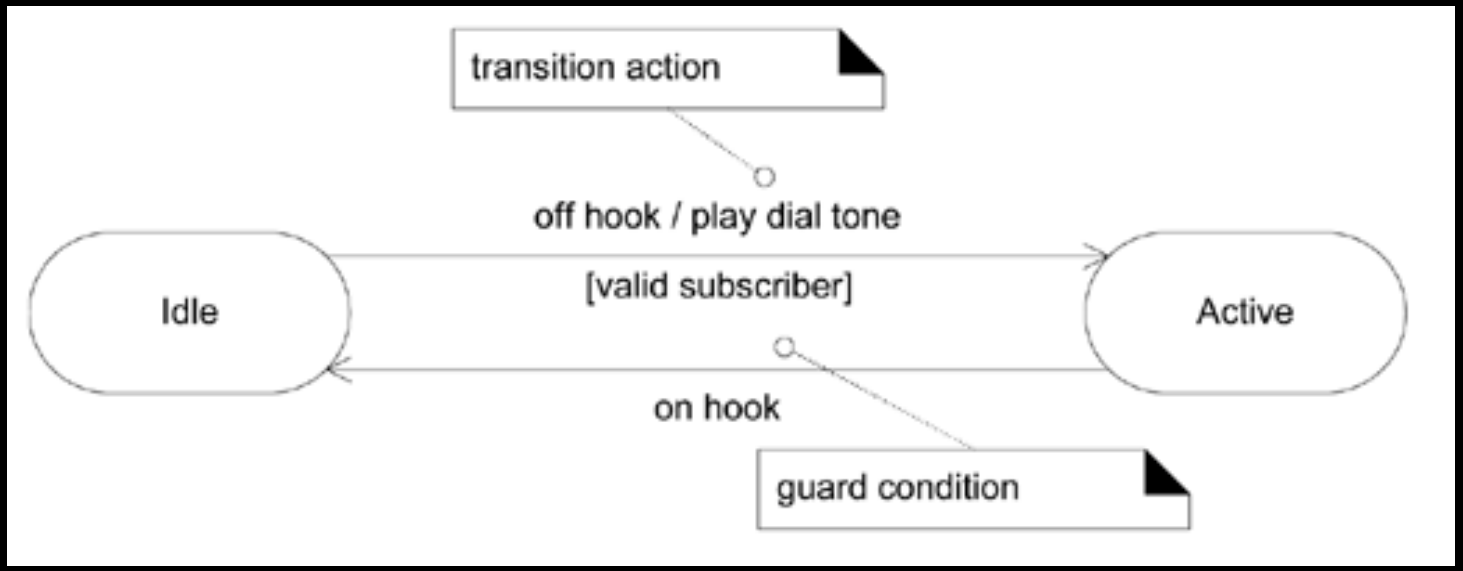
Definitions: Transitions

- A transition is a relationship between two states that indicates that when an event occurs, the object moves from the prior state to the subsequent state
- Kinds of Transitions

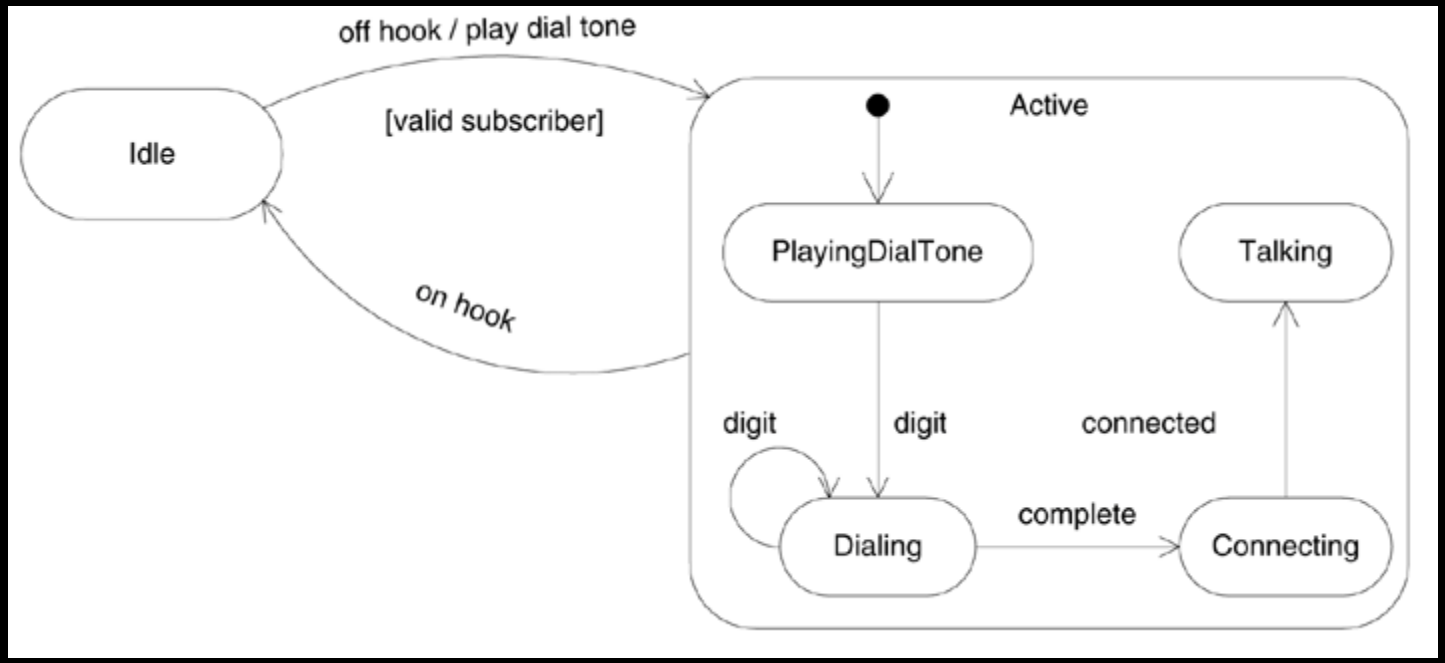


Transition Kind	*Description*	*Syntax*
entry transition	The specification of an entry activity that is executed when a state is entered	entry/ activity
exit transition	The specification of an exit activity that is executed when a state is exited	exit/ activity
external transition	A response to an event that causes a change of state or a self-transition, together with a specified effect. It may also cause the execution of exit and/ or entry activities for states that are exited or entered.	e(a:T)[guard]/activity
internal transition	A response to an event that causes the execution of an effect but does not cause a change of state or execution of exit or entry activities	e(a:T)[guard]/activity

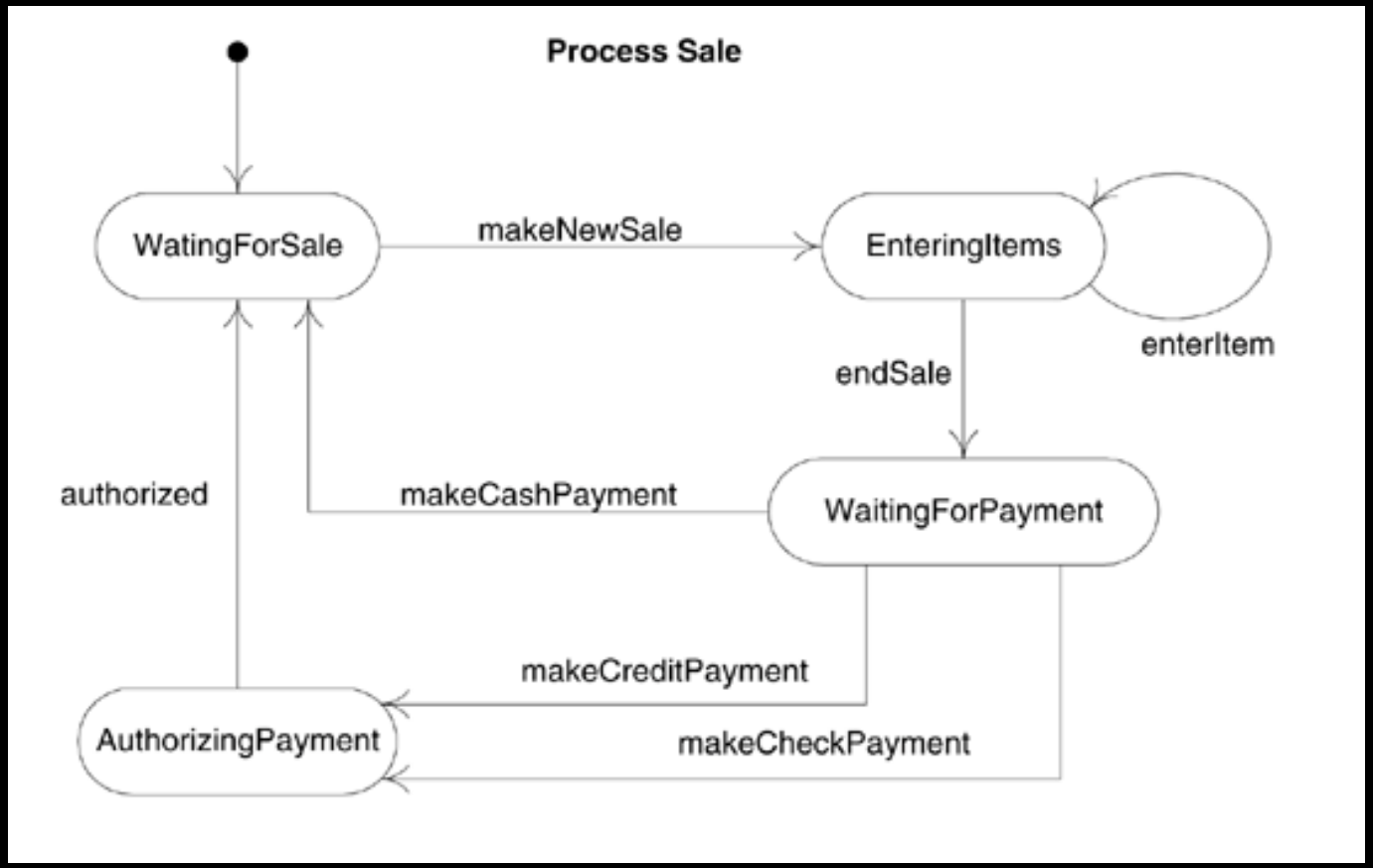
Transition Actions and Guards



Nested States



A sample state machine



软件工程概论

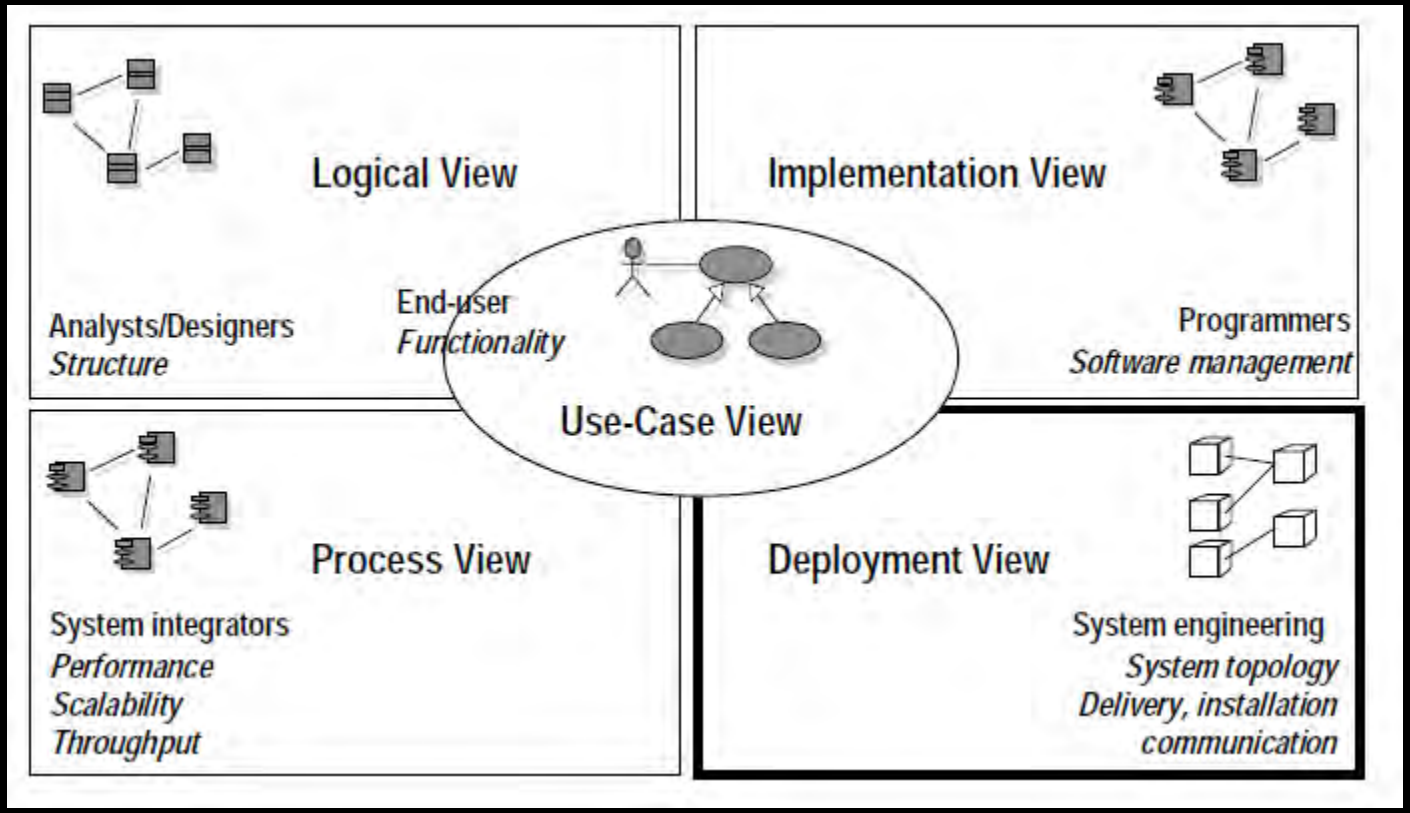
Introduction to Software Engineering

IMPLEMENTATION MODEL AND UML DEPLOYMENT DIAGRAM

Implementation Model and UML Deployment Diagram

- The Deployment View
- UML Deployment Diagram
  - **What is Node?**
  - **What Is Connection?**
  - **What Is Artifact ?**

Key Concepts: The Deployment View



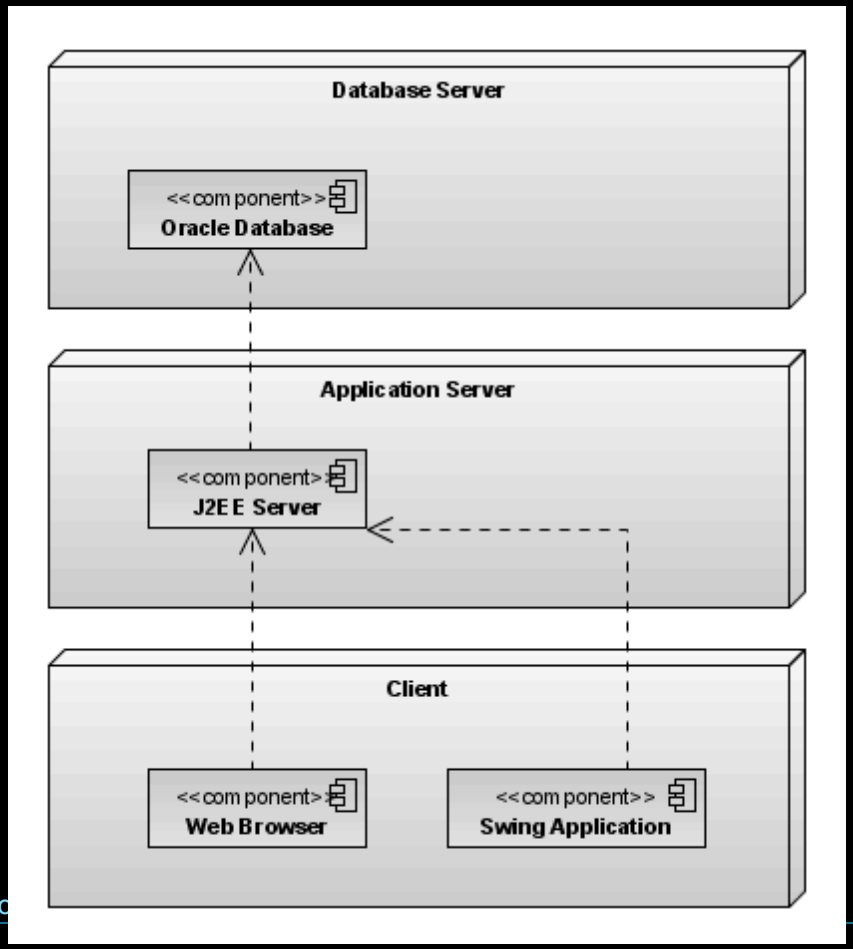
The Deployment View is an "architecturally significant" slice of the Deployment Model.

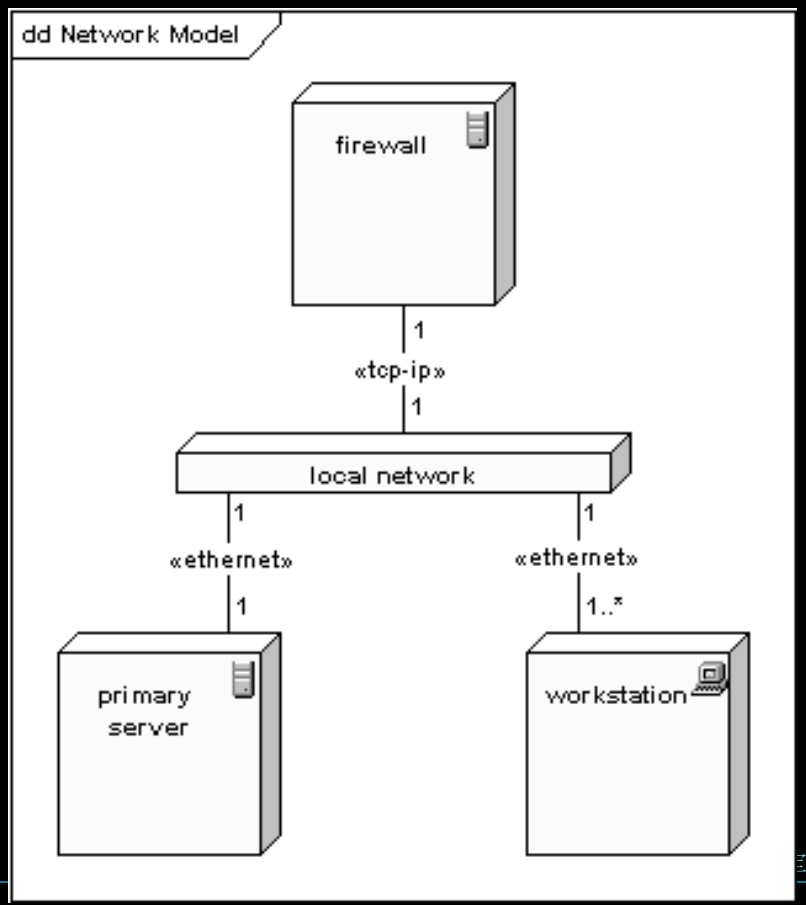
UML Deployment Diagram

- A deployment diagram is a diagram that shows the configuration of run time processing nodes and the components that live on them.
- Captures the topology of a system's hardware
- Built as part of architectural specification
  - **Purpose**
  - **Specify the distribution of components**
  - **Identify performance bottlenecks**
- Developed by architects, networking engineers, and system engineers

UML Deployment Diagram

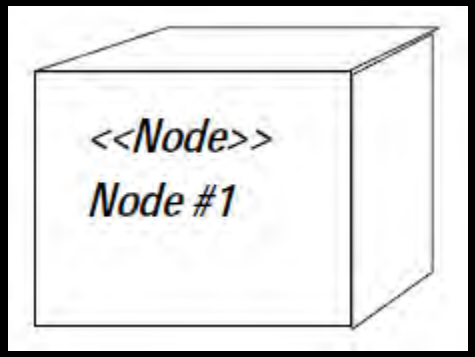
- Models the run-time architecture of a system
- A diagram that shows the configuration of run time processing nodes and the artifacts that live on them.





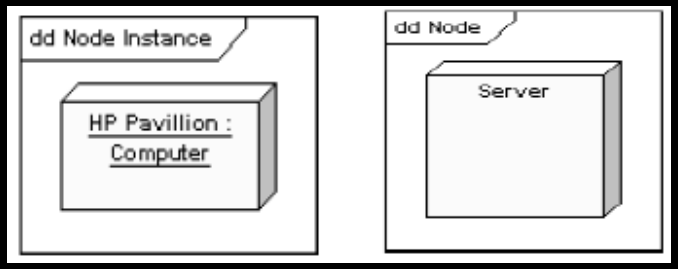
What is Node?

- A node is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability.
- A set of components may reside on a node and may also migrate from node to node.
- Graphically, a node is rendered as a cube, usually including only its name.

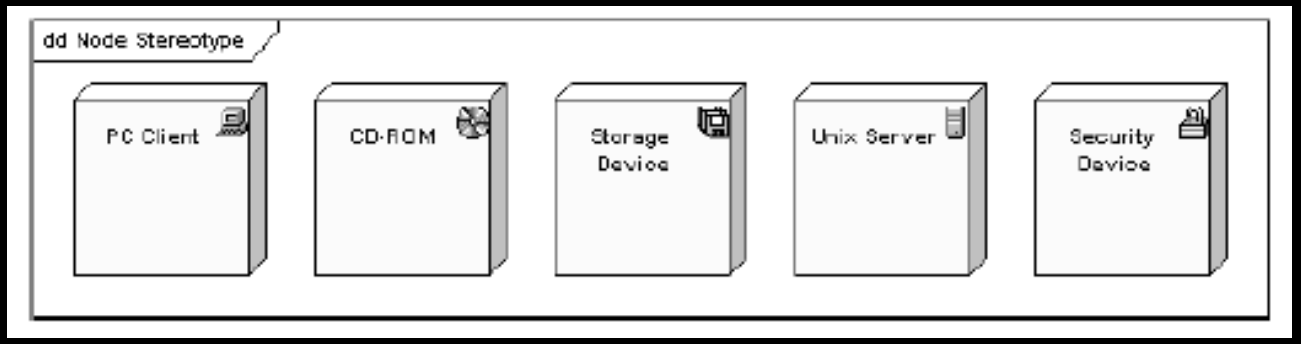


Deployment Diagram - Node

- Node Instance



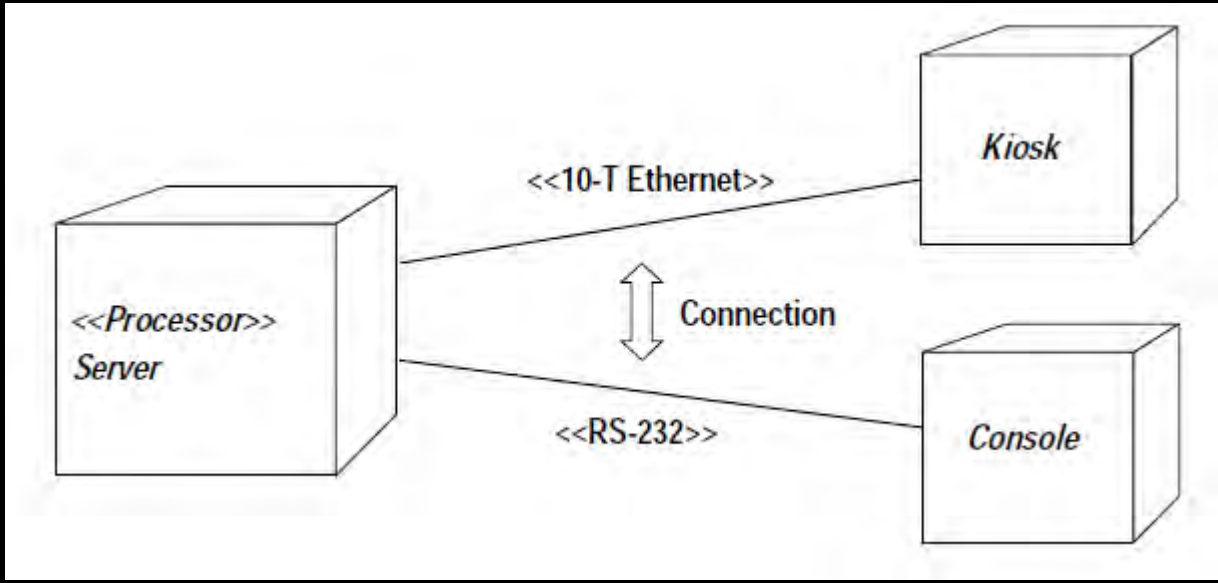
- Node Stereotypes
  - A number of standard stereotypes are provided for nodes,
  - namely «cd-rom», «computer», «disk array», «pc», «pc client», «pc server», «secure», «server», «storage», «unix server», «user pc»



- Association
  - In the context of a deployment diagram, an association represents a communication path between nodes

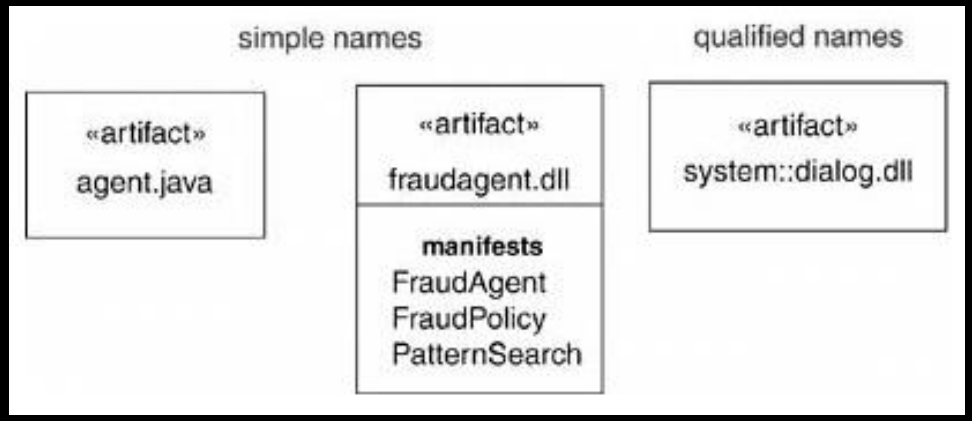
What Is a Connection?

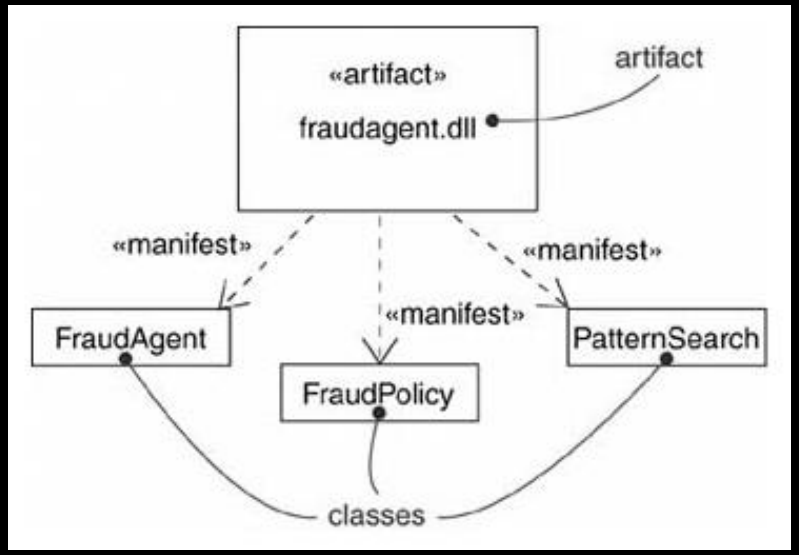
- A connection represents a:
  - **Communication mechanism**
    - **Physical medium**
    - **Software protocol**



Deployment Diagram - Artifact

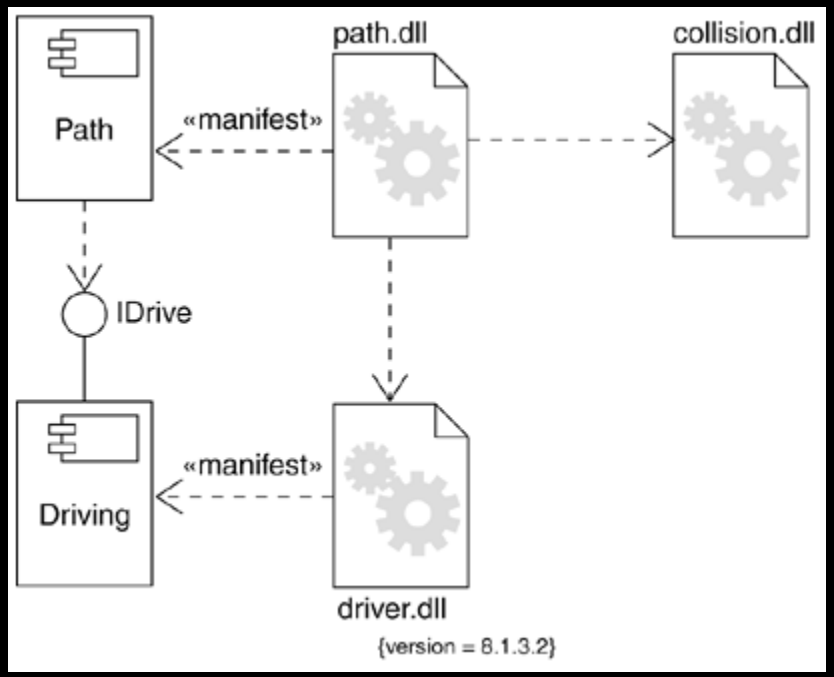
- Artifact
  - A physical part of a system that exists at the level of the implementation platform.
  - Graphically, an artifact is rendered as a rectangle with the keyword «artifact».





Deployment Diagram - Artifact Diagram

- Artifact Diagram
  - A variety of deployment diagram
  - shows a set of artifacts and their relationships.
  - commonly contain
    - artifacts
    - dependency, generalization, association, and realization relationships



软件工程概论

Introduction to Software Engineering

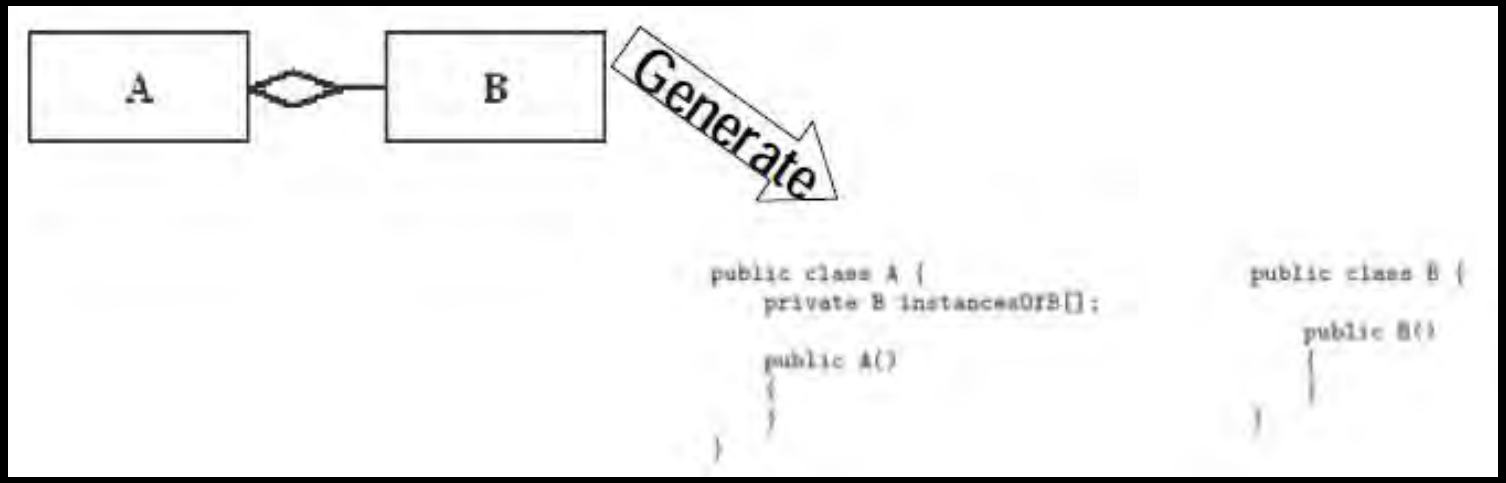
FORWARD, REVERSE, AND ROUND-TRIP ENGINEERING

Forward, Reverse, and Round-Trip Engineering

- Forward Engineering
- Reverse Engineering
- Round-Trip Engineering

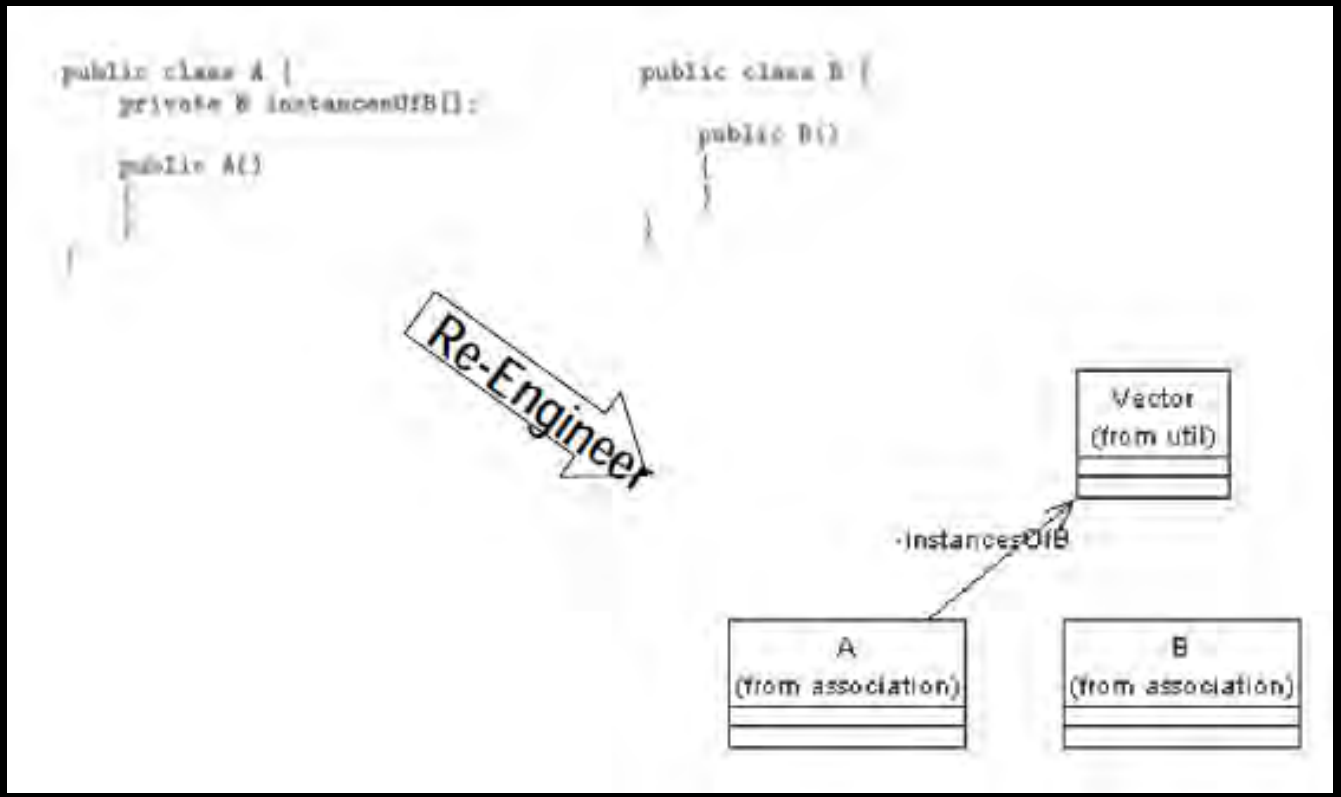
Forward Engineering

- Forward engineering means the generation of code from UML diagrams
- Many of the tools can only do the static models:
  - **They can generate class diagrams from code, but can't generate interaction diagrams.**
  - **For forward engineering, they can generate the basic (e.g., Java) class definition from a class diagram, but not the method bodies from interaction diagrams.**
- Demo



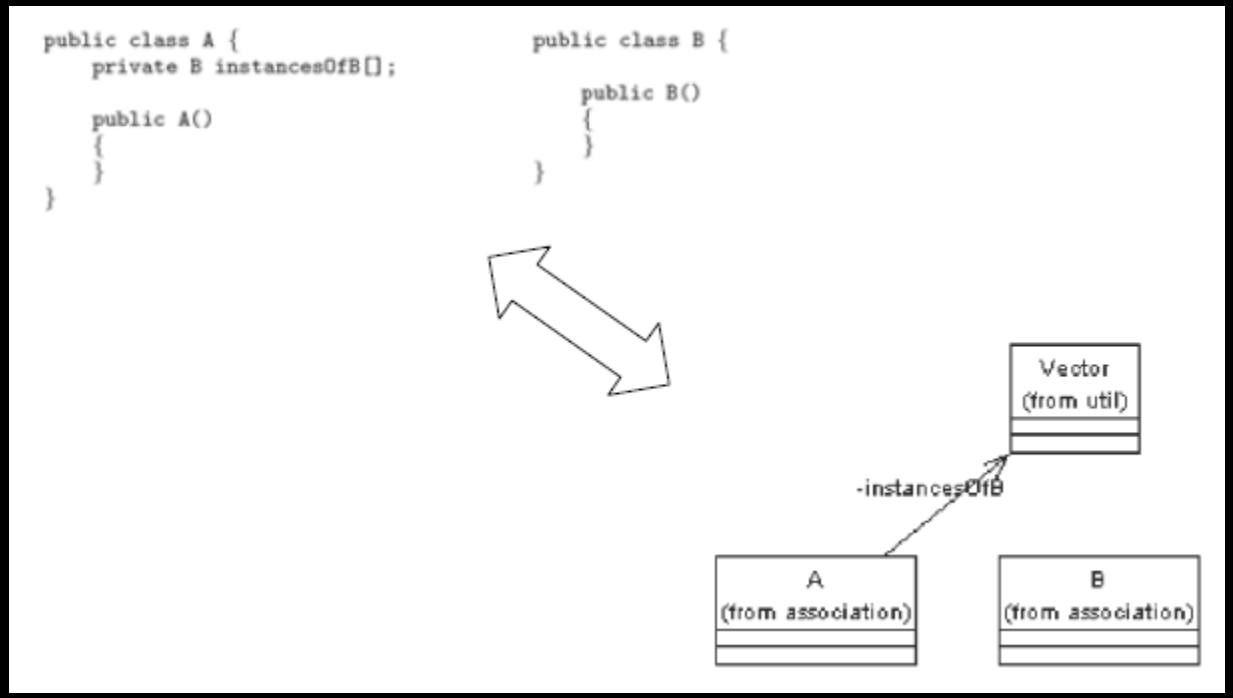
Reverse Engineering

- Reverse engineering means generation of UML diagrams from code
- Demo



Round-Trip Engineering

- Round-trip engineering closes the loop
  - **the tool supports generation in either direction and can synchronize between UML diagrams and code, ideally automatically and immediately as either is changed.**
- Demo



软件工程概论

Introduction to Software Engineering

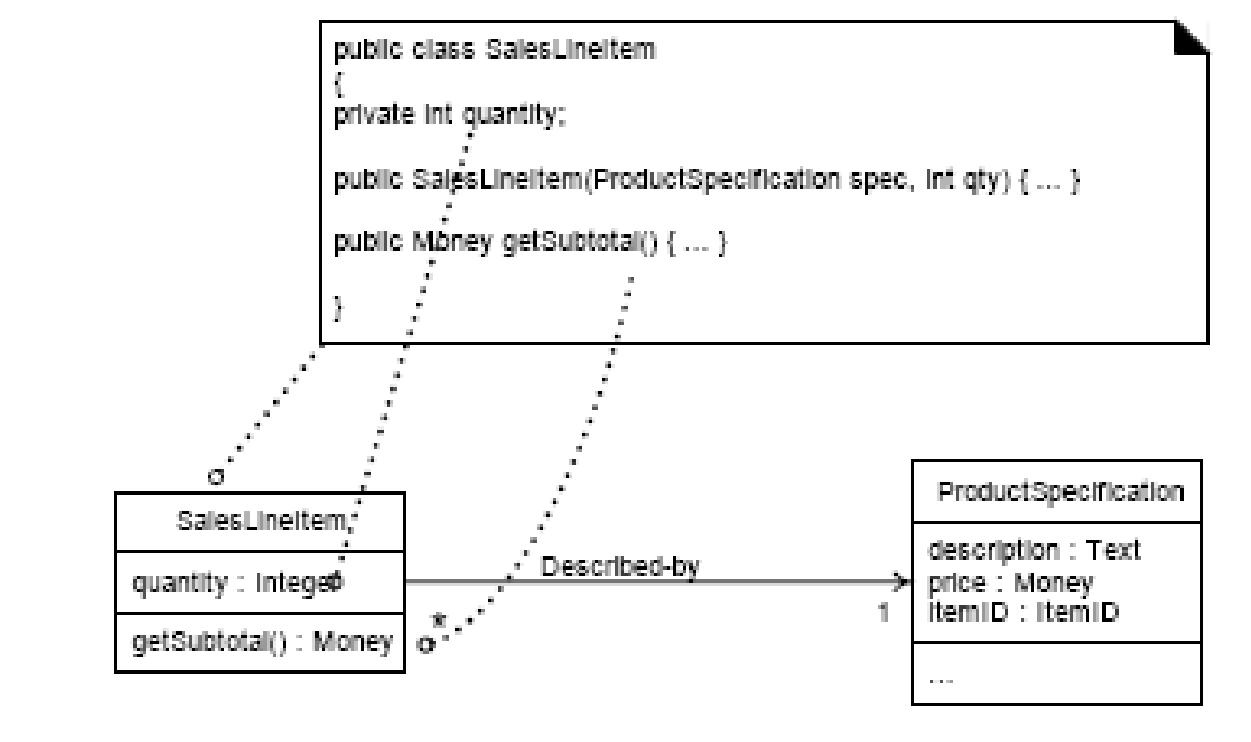
CODE AND TEST

Code and Test

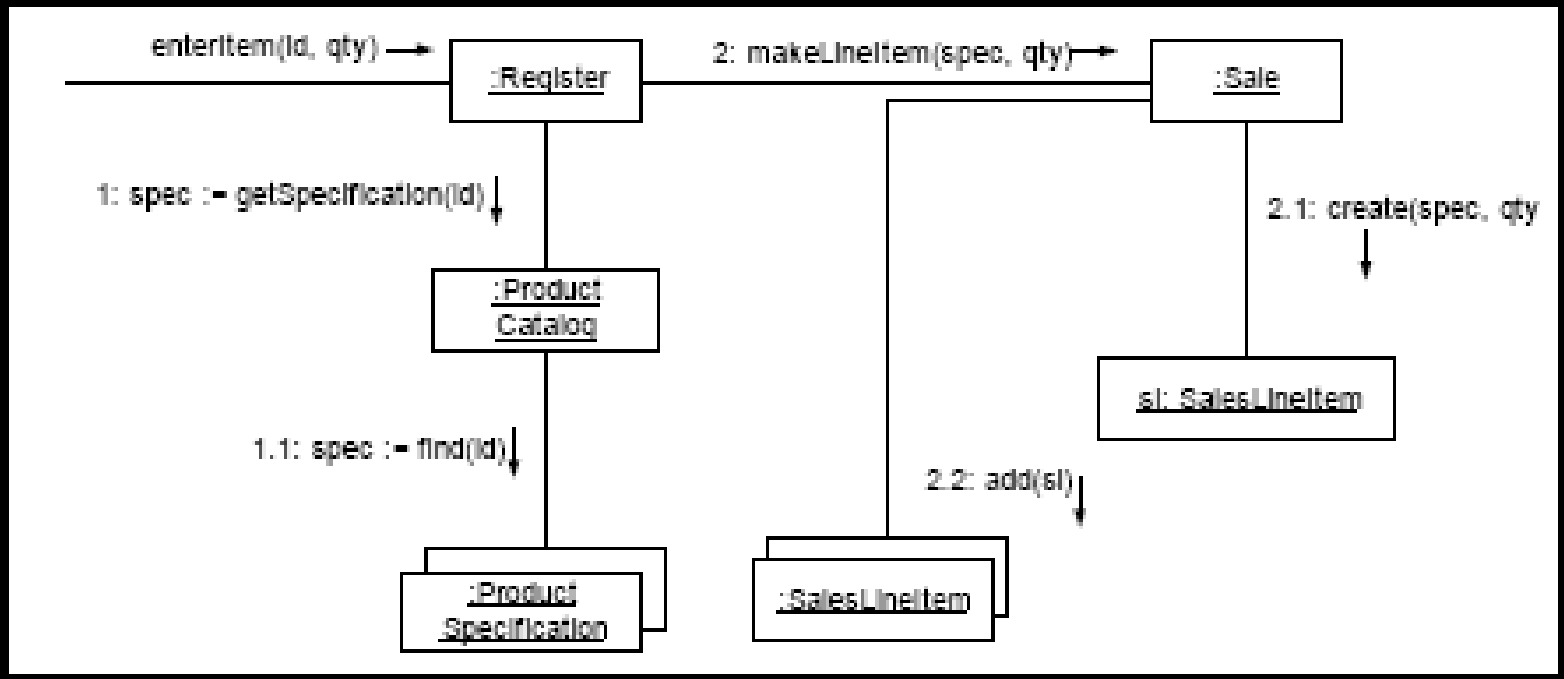
- Creating Class Definitions from Class Diagram
- Creating Methods from Interaction Diagrams
- Collection Classes in Code
- Test-Driven Development
- Refactoring

Creating Class Definitions from Class Diagram

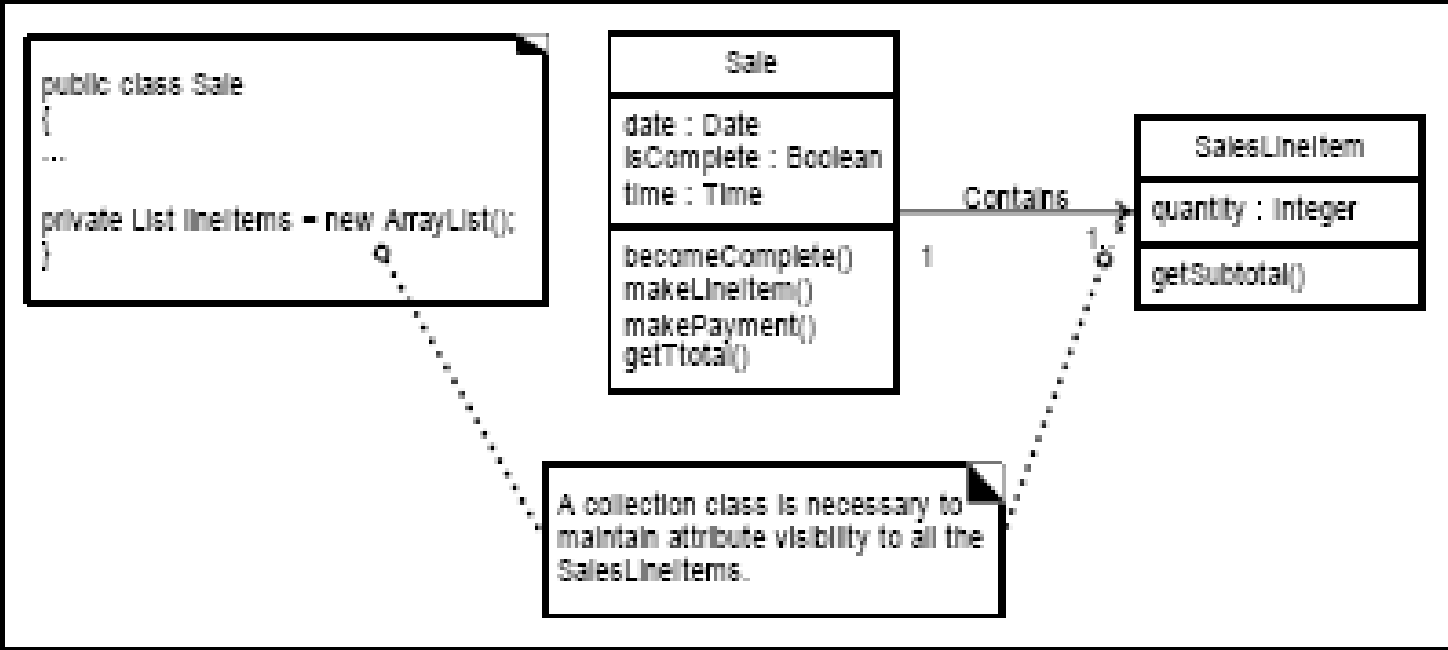
- Defining a Class with Method Signatures and Attributes



Creating Methods from Interaction Diagrams



Collection Classes in Code



Test-Driven Development

- An excellent practice promoted by the iterative and agile XP method, and applicable to the UP, is test-driven development (TDD).
  - **It is also known as test-first development**
- In OO unit testing TDD-style, test code is written before the class to be tested and the developer writes unit testing code for nearly all production code.
- Unit testing framework
  - **The most popular unit testing framework is the xUnit family (for many languages)**
    - **For Java, the popular version is JUnit.**
    - **There's also an NUnit for .NET**

Refactoring

- Refactoring is a structured, disciplined method to rewrite or restructure existing code without changing its external behavior,
  - **applying small transformation steps combined with re-executing tests each step.**
- Continuously refactoring code is another XP practice and applicable to all iterative methods
- Code that's been well-refactored is short, tight, clear, and without duplication it looks like the work of a master programmer.
  - **Code that doesn't have these qualities smells bad or has code smells.**

软件工程概论

Introduction to Software Engineering

END OF THIS TOPIC !

Shanghai Jiao Tong University

软件工程
Module: 用户界面设计

上海交通大学软件工程中心

软件界面设计

Easy to learn?
Easy to use?
Easy to understand?

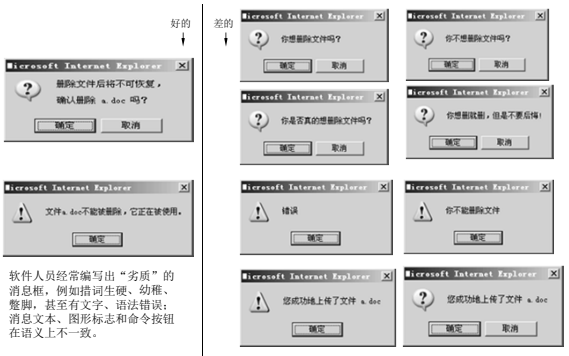
软件用户界面设计要综合考虑“易用性(Usability)设计”、“艺术设计”和“技术实现”

界面设计中典型错误

Typical Design Errors

lack of consistency
too much memorization
no guidance / help
no context sensitivity
poor response
Arcane/unfriendly

有时连简单的消息框都设计不好



软件人员经常编写出“劣质”的消息框，例如措词生硬、幼稚、蹩脚，甚至有文字、语法错误；消息文本、图形标志和命令按钮在语义上不一致。

问题和原因

- 教育缺陷
  - 国内绝大多数大学的计算机学科没有开设软件用户界面设计的课程，也没有开设相关的如人机工程学、美学、心理学等课程。理工科人士天性喜欢钻研技术，不太关心用户需要什么。
- 服务对象错位
  - 开发人员在设计用户界面方面不仅存在先天的教育缺陷，更加糟糕的是还常常犯“错位”的毛病。以为只要自己感觉用户界面漂亮、使用起来方便，那么用户也一定会满意。

用户界面设计

- 界面设计的原则
- 人机交互方式
- 界面设计的过程
- 界面设计的问题

@第6.6节.教材

界面设计的原则

- 易学性（Learnability）
  - 系统应容易学习和掌握，不应对用户有额外的知识和技能要求。
  - 用户可以通过两种途径来学习系统，即：系统的联机手册；系统功能的操作演示及例子。
- 用户熟悉性（User familiarity）
  - 界面应以用户导向的名称和观念为主，而不是以计算机的概念为主。这能让用户更快地熟悉系统，使用系统。
- 一致性（Consistency）
  - 系统的各个界面之间，甚至不同系统之间，应具有相似的界面外观、布局，相似的人机交互方式以及相似的信息显示格式等。

- 减少意外（Minimal surprise）
  - 系统功能和行为对用户应是明确、清楚的。
  - 例如：系统有标准的界面；系统不会产生异常的结果，在相同情况下总会有相同的行为；系统有预定的响应时间等。
- 易恢复性（Recoverability）
  - 系统设计应该能够对可能出现的错误进行检测和处理，提供机制允许用户从错误中恢复过来。
- 提供用户指南（User guidance）
  - 系统应提供及时的用户反馈和帮助功能。
- 用户多样性（User diversity）
  - 系统应适应各类用户（从偶然型用户、生疏型用户到熟练型用户，直至专家型用户）的使用需要，提供满足其要求的界面形式。

黄金规则

- **让用户驾驭软件，不是软件驾驭用户**
- **减少用户的记忆**
- **保持界面的一致性**

用户界面设计

- 界面设计的原则
- 人机交互方式
- 界面设计的过程
- 界面设计的问题

人机交互方式

- 人机交互方式的选择是界面设计的重要决策之一，设计者应根据软件的需求，选择一种以上的交互方式，进行人机界面设计。
- 所谓人机交互方式，是指人机之间交换信息的组织形式或语言方式，又称对话方式、交互技术等。

常见的人机交互方式

- 问答式对话
  - 最简单的人机交互方式。它是由系统启动的对话，系统使用自然语言的指导性提问，提示用户进行回答，用户回答可以通过键盘输入字符串或书写笔进行手写输入。
  - 这种方式的对话，简单时是采用非选择形式，即要求输入Yes / No；复杂时是把回答限制在很小范围的答案集内，系统再根据用户的回答去执行相应的功能，或将使用者输入信息记录保存，放入数据库或资源库中。
  - 优点：容易使用、学习，软件编程实现容易，用户回答范围小，因此不易出错。
  - 缺点：效率不高，速度慢，灵活性差，修改扩充不方便

- 直接操纵
  - 使用鼠标、手势等代替命令或菜单选择中的键盘输入，并通过指向可视对象与动作，用户可以迅速执行任务和立即观察结果。
  - 示例：可视化编辑器、飞行控制系统和电视游戏等。
  - 优点：直接操纵对新手很有吸引力，对知识断层的用户来说是容易记住的，并且经过仔细设计，对经常用户来说也可以快速地执行任务。
- 菜单选择
  - 用户读一个菜单，选择一个对他们的任务最合适的菜单项，开始该动作并观察其效果。
  - 优点：如果术语和菜单项的意义是可理解且明确的，则用户可以用少量的学习或记忆和很少的击键次数来完成任务。菜单选择方式对保证不变的屏幕设计，证实完整性和支持维护等有很大的益处。

- 填表
  - 在需要输入数据时，菜单选择通常显得很不方便，填表则较适宜。
  - 用户看见一个相关字段的显示，在该字段中移动光标，在需要的地方输入数据。在填表时，用户必须理解字段的标题，知道值的允许范围和数据输入方法，能够对出错信息做出反应。
  - 对有知识断层的用户或经常性用户来说是最合适的。
- 命令语言
  - 对熟练型用户来说，命令语言提供了一个控制和创造性的氛围。用户学习句法并能够迅速地表达复杂的任务，而不必阅读容易分散注意力的提示信息。
  - 但这类界面出错率通常比较高，培训是必须的，保持性也比较差，很难提供出错信息和联机求助。

- 自然语言
  - 各种自然语言在人际之间进行交流时是寻常之举。如果在人与计算机之间也能使用自然语言进行通信、交互，应该说是最理想和最方便的。
  - 在这种交互中，计算机能理解用户用自然语言（包括键盘输入文本、手写文字、语音输入等）表达的请求，并把系统的理解转换成系统语言，然后执行相应的应用功能。
  - 优点：具有用户无需学习训练就能以自然交流方式使用计算机的优点
  - 缺点：具有输入冗长文字，自然语言语义有二义性，需要具有应用领域的知识基础以及编程实现困难等缺点。到目前为止，成功案例还比较少。
  - 自然语言界面是最理想、最友好的人机界面类型，但是要变为现实，仍有很多工作要做。

新的人机交互方式－－可穿戴技术

- 视线、肢体动作、脸部表情……

美国的汽车技术供应商哈曼公司2012年推出了一种全新的车载体感识别系统，驾驶员只需眨眨眼、点点头、挥挥手，就可以在驾驶的同时自如地使用汽车的内置功能。

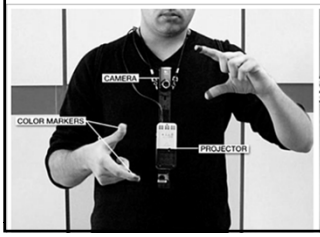
人机交互眼镜
通过转动眼睛和眨眼就能控制电脑

- 3D体感摄影机 Kinect
  - 骨架追踪
  - 语音辨识
  - 人脸辨识

- **耳感应开关**
  - “嘴角笑一下就能让房间的灯打开,或是让洗衣机开始工作”
  - “如果有一台iPod的话,那么伸一下舌头就能让它停止或启动音乐播放,就像是爱因斯坦那副著名照片中的表情.
  - 如果眨一下左眼,它就跳动到下一首歌,右眼是退回”

- **传感手套：将手语转换成文本**

- 第六感系统
  - 通过摄像头等随身传感器识别周围环境中的物体，并可将操作界面投影到各种物体上，或直接通过手势进行操作。



- 动作捕捉（Motion Capture）
  - 广泛应用于3D动画电影的创作，由专业动作师进行动作演示，然后抓取到计算机中形成动作数据，并附加在想要实现该动作的角色上。

- Google Glass
  - 功能类似一台整合了谷歌云服务的智能手机。其独特的形态便于用户随时获取各类信息，并通过投影显示器以增强现实的方式提供给用户。

其他可穿戴技术

- **BrainLink意念头箍**
  - 由深圳市宏智力科技有限公司专为iOS系统研发，了解佩戴者的大脑状态，例如是否专注、紧张、放松或疲劳等。佩戴者通过主动调节自己的专注度和放松度来给予指令通过蓝牙无线连接智能手机、平板电脑、手提电脑、台式电脑或智能电视等终端设备，实现意念力互动操控。

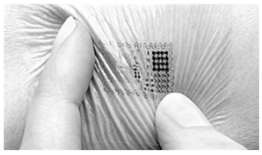
可植入设备

- 皮下射频识别（RFID）芯片的小型胶囊
  - 让你与密码说“再见”


- “左撇子”的指南针
  - 将一个微型指南针封入一块硅套内，植入皮肤下，露出一个超薄的细须，当用户面朝北方时，这一细须会被激活，轻微地刺激皮肤。

可植入设备（续）

- 电子纹身
  - 由超薄电极、电子元件、传感器、无线电源和通信系统组成，测量佩戴者的心率、血压、皮肤的温度、紧张度、水化作用的水平

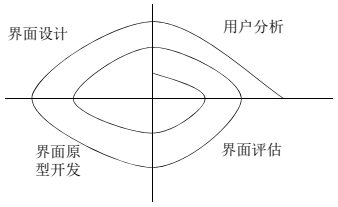
- 能恢复失明病患视力的视网膜植入设备

用户界面设计

- 界面设计的原则
- 人机交互方式
- 界面设计的过程
- 界面设计的问题

用户界面设计过程

- 用户界面的设计过程是迭代的，包括四个活动：
  - 用户分析
  - 界面设计
  - 界面原型开发
  - 界面评估



根据用户的特点设计人机界面

- 用户分类：
  - 外行型：不熟悉计算机操作，对系统很少或毫无认识
  - 初学型：对计算机有一些经验，对新系统不熟悉，需要相当多的支持
  - 熟练型：对系统有丰富的使用经验，能熟练操作，但不了解系统的内部结构，不能纠正意外错误，不能扩充系统的能力
  - 专家型：了解系统内部的结构，有系统工作机制的专门知识，具有维护和修改系统的能力，希望为他们提供具备修改和扩充系统能力的复杂界面

举例

WinXp控制面板的向导功能适合不太熟练的用户

讨论

- 以下系统的用户具有什么特点？应有采用什么界面风格？
  - POS系统
  - ATM系统
  - 银行前台系统
  - 银行后台管理系统

用户界面设计

- 界面设计的原则
- 人机交互方式
- 界面设计的过程
- 界面设计的问题

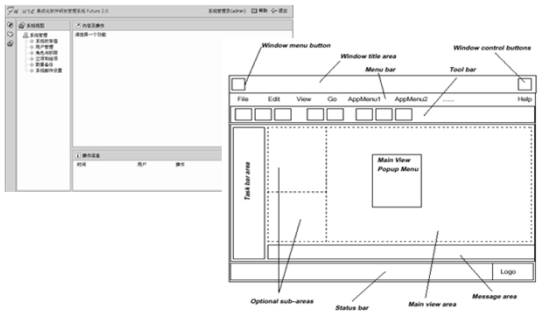
1）系统响应时间 （ Response time ）

- 系统响应时间指从用户执行某个控制动作（如按回车键或点鼠标）到软件作出响应（期望的输出或动作）的时间。
- 时间长度：系统响应时间长会使用户感到不安和沮丧。人的一般容忍度为15秒。
- 可变性：稳定的响应时间（如1秒）比不定的响应时间（如0.1秒到2.5秒）要好。用户往往比较敏感，总是关心界面背后是否发生了异常。

及时反馈操作信息

- 当用户进行某项操作后，如果过了一会儿（几秒钟）用户界面一点反应都没有，这将使用户感到迷茫和不安，因为他不知道是自己操作错了还是软件死机了。
- 及时反馈信息很重要，至少要让用户心里有数，知道该任务处理得怎么样了，有什么样的结果。
- 例如下载一个文件，界面上应当显示“百分比”或相关数字来表示下载的进度，否则人们不知道要等待多少时间。如果某些事务处理不能提供进度等数据，那么至少要给出提示信息如“正在处理，请等待...”。最好是提供合适的动画，让用户明白软件正在干活、没有死机。

示例



专业软件一般都要提供反馈操作信息的窗口区域。

2）帮助设施（ Help facilities ）

- 关于帮助设施，在设计时须考虑如下问题：
  - 在系统交互时，是否总能得到各种系统功能的帮助？是提供部分功能的帮助还是提供全部功能的帮助。
  - 用户怎样请求帮助？使用帮助菜单、特殊功能键还是HELP命令。
  - 怎样表示帮助？在另一个窗口中、指出参考某个文档（不是理想的方法）还是在屏幕特定位置的简单提示。
  - 用户怎样回到正常的交互方式？可做的选择有：屏幕上显示返回键、功能键或控制序列。
  - 怎样构造帮助信息？是平面式（所有信息均通过关键字来访问）、分层式（用户可以进一步查询得到更详细的信息）还是超文本式。

3）出错处理（ Error handling ）

- 交互系统给出的出错消息和警告应具备以下特征：
  - 消息以用户可以理解的术语描述问题。
  - 消息应提供如何从错误中恢复的建议性意见。
  - 消息应指出错误可能导致哪些不良后果（比如破坏数），以便用户检查是否出现了这些情况或帮助用户进行改正。
  - 消息应伴随着视觉或听觉上的提示，也就是说，显示消息时应该伴随警告声或者消息用闪耀方式，或明显表示错误的颜色显示。
  - 消息应是“非批评性的”（nonjudgmental），即不能指责用户。

防错处理

- 在设计界面时必须考虑防错处理，目的是让用户不必为避免犯错误而提心吊胆、小心翼翼地操作。
- 常见的防错处理措施有：
  - 对输入数据进行校验。如果用户输入错误的数据，软件应当识别错误并且提示用户改正数据。
  - 对于在某些情况下不应该使用的菜单项和命令按钮，应当将其“失效”（变成灰色，可见但不可操作）或者“隐藏”。
  - 执行破坏性的操作之前，应当获得用户的确认。
  - 尽量提供Undo功能，用户可以撤销刚才的操作。

4）菜单和命令交互（ Menu and command Interaction）

- 在提供命令或菜单交互方式时，必须考虑以下设计问题：
  - 是否每个菜单选项都有对应的命令？
  - 采用何种命令形式？有3种选择：控制序列（例如，Ctrl+P），功能键和输入命令。
  - 学习和记忆命令的难度有多大？忘记了命令怎么办？
  - 用户是否可以定制或缩写命令？
  - 在界面环境中菜单标签是否是自解释的？
  - 子菜单是否与主菜单项所指功能相一致？

5) 可访问性 6）国际化

- 应用系统的可访问性
  - 确保系统界面能让那些身体上面临挑战的用户也易于访问，即，为视觉、听觉、活动性、语音和学习等方面有障碍的用户提供系统的访问机制。
- 国际化
  - 软件工程师和他们的经理往往会低估建立一个适应不同国家和不同语言需要的用户界面所应付出的努力和技能。
  - 用户界面应该被设计成能够容纳需要交付给所有软件用户的核心功能，同时能够针对特定的市场进行界面定制以反映其本地化特征。
    - 文字－－Unicode
    - 颜色
    - 阅读次序
    - 货币单位
    - 风俗习惯

7）合理的布局和合理的色彩

- 合理的布局
  - 界面的总体布局应当有一定的逻辑性，最好能够与工作流程吻合。界面设计人员只有仔细地分析软件的需求，才能提取对界面布局有价值的信息。
  - 窗口（或页面）上的界面元素的布局应当整齐清爽。界面元素应当在水平或者垂直方向对齐，行、列的间距保持一致。
    - 窗体的尺寸要合适，界面元素不应放得太满，边界处需要留有一定的空间，也不可过于宽松，显得零乱。
    - 界面元素需要一致的对齐方式，以避免参差不齐的视觉效果。同类的界面元素尽量保持大小一致，起码要保证高度或宽度的一致（例如命令按钮）。逻辑相关的元素要就近放置，便于用户操作。
    - 要善于利用窗体和界面元素的空白，以及分割用的线条。

合理的色彩

- 相比于布局，设计合理的色彩就困难多了，因为色彩的组合千变万化，并且人们对颜色的喜好也极不相同。
  - 例如，人们对黑色的理解差异很大。
- 一般规律：
  - 如果不是为了显示真实感的图形和图像，那么应当限制一帧屏幕的色彩数目，因为人们在观察屏幕的时候很难同时记住多种色彩。
  - 应当根据对象的重要性来选择颜色，重要的对象应当用醒目的色彩表示。
  - 使用颜色的时候应当保持一致性，例如错误提示信息用红色表示，正常信息用绿色表示，那么切勿乱用红色和绿色。
  - 在表达信息时，不要过分依赖颜色，因为有些用户可能色盲或色弱。

示例

1）”危险，有风险，未知，基本安全” 的颜色，有什么讲究吗？
2）“黄色”看不清楚
3）文字不完整，在数字后面加“个”
4）前后项留白合适的间隔。

国际化（Internationalization）

- 软件的国际化是大势所趋。在设计用户界面的时候应当充分考虑语言和文化的差异。尽可能使用标准的图解方式和国际通行的语言，要求简单易懂，易于翻译，方便不同母语的用户。
- 翻译文字要地道，要符合本地习惯，不能硬翻译，否则太不专业。
  - MSN Messenger在发送文件的时候，出现：

  - 明显是汉语中的病句，正确的翻译应该是“文件正在传输，剩余2407 KB”。
- 特别要留意下列元素的国际化问题：
  - 字体、提示信息、在线帮助
  - 货币、度量单位
  - 日期格式（如MM/DD/YY、Year-MM-DD等格式）
  - 人的名字、电话号码、通信地址
  - 图标、标签
  - 阅读顺序或习惯

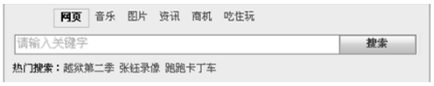
案例分析－上网助手

问： 实名搜索是3721公司的盛名之作，这个搜索的界面有什么不妥吗？

答：

➢“地址”两字让人费解，让人搞不清楚是住址，还是http网址？搜索一个名字行不行？

➢“试一试，浏览器地址栏中也可以直接搜索”这句话有点多余（建议换一种提示方式），用户不知道究竟从哪里搜索。 如果从地址栏搜索，那么会跳离上网助手的页面，可能放跑了用户。



1）由于“性感鸡”和其它广告太醒目了，喧宾夺主，使得下面的“功能排列”不被注意，或者潜意识地以为“功能排列”的所有条目都是宣传广告。

2）我到处找“上网助手”的完整的功能树，希望能够直接操作下级功能菜单，竟然没有看到“功能排列”这个区域。

3）“下载上网助手”和“助手工具下载”在语义上重叠，初级用户搞不明白要下载哪个东西

上网助手的“首页”以及每个Tab页面的“功能介绍”页面，其内容和布局都比较乱，用户摸不清楚界面的规律，增加了记忆难度，降低使用效率。本图中，

1）右侧“热门功能推荐”，在逻辑上不是“IE修复的范畴”，应当放在其它合适的地方，避免扰乱用户的思路。

2）左侧的介绍不完整，少了4项功能的介绍，不一致。

IE 的Toolbar按钮一般不会有下拉菜单（与流行软件的界面元素不一致）。如果要下拉菜单的话，应该加“下拉”标记。“清理”和“修复”含有立即执行、并且会改动用户计算机设置的含义，用户担心点击“清理”和“修复”按钮将出现不期望的操作。要设法消除用户的畏惧和疑虑，让用户明白有下拉菜单。

8）移动界面设计的问题

工具的进化

Released:**1998!**
CPU:**233 MHz**
RAM:**32 MB(512 MB max)!**
Storage:**4GB(+ optional drive)**
Display:**38.1 cm Millions of colours!**
Dimensions:**40.1 x 38.6 x 44.7 cm**
Weight:**17.3 kgs**

<10 年 ...

Released:**2007!**
CPU:**412 MHz**
RAM:**128 MB!**
Storage:**4GB (8 GB max)**
Display:**8.9 cm Millions of colours!**
Dimensions:**11.4 x 6.1 x 1.2 cm**
Weight:**135 g**

场景的改变

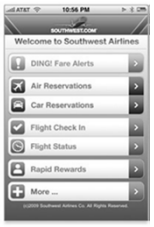
移动中！

挑战

- 尺寸小
  - 同时无法展示丰富的内容，界面应简单，只显示重要的内容，操作步骤简单
  - 根据用户的行为特点和位置等信息，智能地显示或推荐用户喜欢的信息
  - 尽可能减少不必要的文本输入
- 移动设备种类多，界面尺寸不同
  - 要求界面具有自适应性
- 用户的层次、爱好和习惯众多
  - 易用性要求更高
  - 支持个性化定制，提供个性化服务

举例：简洁的界面

- 美国西南航空公司的web站和对应的iPhone app比较
  - iPhone app简洁专注于客户需求：机票预订、登机手续、查询航班状态、查询里程等，再没有其他的多余内容。



- 比如Path，它把五个常用的按钮，集成到“+”里。点击加号以后，有拍照，音乐等功能。而界面上，打开这个应用，最直观的就是最主要的信息，没有其他的干扰。比如之前有多少人看过我的图片，它把这个信息直接集成在图片右上角，没有占据太多地方，点击之后，可以发表情、评论、直接删除等，做到了隐藏，是个非常干净、漂亮的页面。

举例：帮用户做决策

输入查询条件，出现酒店列表

输入查询条件，出现酒店列表

理解特定用户
直接出现周边酒店地图

举例：善用隐喻

创新－－利用移动设备的感知设计

- 多点触控
- 地理定位
- 运动方向
- 手持定向
- 语音输入
- 照相摄像
- 实时通知
- 设备连接
- 刷卡扫描
- 电子标签
- 陀螺仪
- ……

举例：操作方式的创新

- 比方说，我现在在这个位置，想知道某一些位置有什么好吃的，一种方式是定位了以后，直接就把附近所有东西显示出来。
- 还有一种方式，我的手在上面滑动，它会记录下轨迹，我只滑动这么一个区域，那就只会现显示这个区域内的商户。这种方式特别直观，而且用户想怎么样就怎么样，想画一个五角星就画一个五角星，想画一条线也可以，它只给你想要的地方的那些内容，这就是一种创新。

Shanghai Jiao Tong University

软件工程

Module: 编码和版本管理

上海交通大学计算机系

编码的目的和质量要求

编码

模块的构件级设计（不可执行的） ⟶ 源程序（可执行的）

- 程序设计语言的特性和程序设计风格会深刻地影响软件的质量和可维护性。
- 为了保证程序编码的质量，程序员必须深刻理解、熟练掌握并正确地运用程序设计语言的特性。此外，还要求源程序具有良好的结构性和良好的程序设计风格。

编码和版本管理

- 程序设计语言
- 编码准则和规范
- 软件版本管理
- 软件持续集成

@第12.3.4节.教材

计算机上语言的层次

纯二进制	Byte或word，指令、数据不分
机器码或符号机器码	用一些符号来代表指令，如sub代表减，Add代表加等，机器地址用十进制。有时汇编语言等同于符号机器码。
符号汇编语言	变量名用符号，地址也可用符号而非数字。编制的程序称为汇编语言程序。
宏汇编语言	用户可以定义新指令及子程序
高级语言	源程序编译为目标程序，或解释执行
甚高级语言	高级程序语言的高层规约语言，提供比高级程序语言更高级的语言设施。又称为“可执行的规约语言”。有时也不区别于高级语言。

高级程序设计语言的分类

- 说明式（*declarative*）语言
  - 函数式 Lisp/Scheme, ML, Haskell …
  - 数据流 Id, Val …
  - 逻辑式 或基于约束的 Prolog, spreedsheets …
  - 基于模板的 XSLT …
- 命令式（*imperative*） 语言
  - 冯. 诺伊曼 C, Ada, Fortran …
    - 脚本式 Perl, Python, PHP…
  - 面向对象 Smalltalk, Eiffel, C++, Java …

语言的选择

- 选择编码语言的标准
  - 应用领域
  - 算法与计算复杂性
  - 数据结构的复杂性
  - 效率的考虑

语言选择举例

- 如果编写对性能要求苛刻，或和操作系统结合紧密的程序，必然选择c。
- 如果需要跨平台，又要广泛的支持的话，选择java。
- 如果编写大程序，可能的化尽量用python，不行了再用java和c。因为python带来了生产力。
- 编写文本的处理程序用perl。在linux下，最方便的工具语言是perl，它有强大的社区和代码库的支持。
- 编写知识的处理程序用prolog。
- 编写最灵活，最模糊的程序用lisp。
- 编写office程序用vba。
- 编写服务器端程序，用php、perl、python、asp、jsp。
- 编写数据库程序最简单的语言是vb或delphi。
- 如果只作为简单应用的工具语言，python和ruby是更好的选择，他们的跨平台移植性好，应用也比较广泛。其中python更适合入门和交流，长期使用也不错。ruby是对python不满意的另一个选择，它提供了很多额外的功能。

编码和版本管理

- 程序设计语言
- 编码准则和规范
- 软件版本管理
- 软件持续集成

编码准则－1

- Preparation. Before you write one line of code, be sure you:
  - Understand of the problem you're trying to solve
  - Understand basic design principles and concepts.
  - Pick an appropriate programming language.
  - Select an appropriate programming environment and tools.
  - Create unit tests for each component you plan to create.

编码准则－2

- Coding. As you begin writing code, be sure you:
  - Follow structured programming practice.
  - Select data structures that will meet the needs of the design.
  - Create interfaces consistent with the software architecture.
  - Keep conditional logic as simple as possible.
  - Create nested loops in a way that makes them easily testable.
  - Select meaningful variable names and follow other local coding standards.
  - Write code that is self-documenting.
  - Create a visual layout that aids understanding.

编码准则－3

- Validation. After you've completed the first pass, be sure you:
  - Conduct a code walkthrough when appropriate.
  - Perform unit tests and correct errors you've uncovered.
  - Refactor the code.

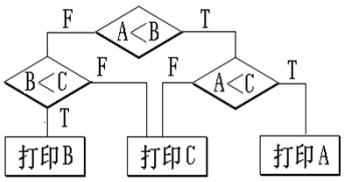
编码的风格

- 追求“聪明”和“技巧”⟶ 提倡“简明”和“直接”
- 使用标准的控制结构
- 清晰的前提下求取效率
  - Make it right before you make it faster.
  - Make it clear before you make it faster.
  - Keep it right when you make it faster.
    （求快不忘保持程序正确）
  - Keep it simple to make it faster.
    （保持程序简单以求快）
  - Don't sacrifice clarity for “efficiency”.
    （书写清楚，不要为“效率”牺牲清楚）

GOTO语句是有害的

- 尽可能避免使用GOTO语句。
- 使用语言中的顺序、选择、重复等有限的基本控制结构表示程序逻辑。

例: 打印A, B, C三数中最小者程序



程序1

```
      if ( A < B ) goto 120;
      if ( B < C ) goto 110;
100   write ( C );
      goto 140;
110   write ( B );
      goto 140;
120   if ( A < C ) goto 130;
      goto 100;
130   write ( A );
140   end
```


程序2

```
if ( A < B ) and ( A < C ) then
  write ( A )
else
  if ( A ≥ B ) and ( B < C ) then
    write ( B )
  else
    write ( C )
  endif
endif
```


源程序的文档化 （code documentation）

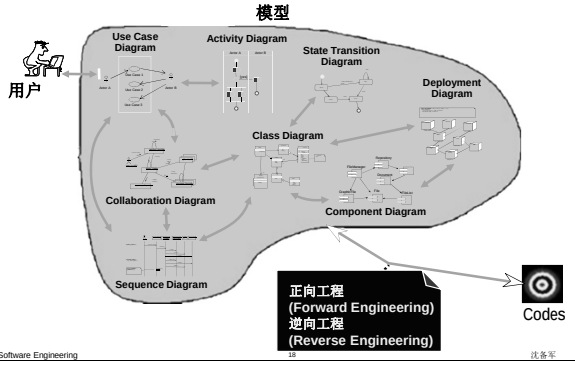
- 有意义的变量名称
- 适当的注释
- 标准的书写格式
  - 用分层缩进的写法显示嵌套结构的层次；
  - 在注释段与程序段、以及不同程序段之间插入空行；
  - 每行只写一条语句；
  - 书写表达式时，适当使用空格或圆括号等作隔离符；

好的源程序本身就是详细设计文档！

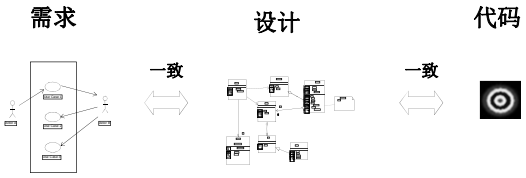
编码规范

浏览C++/Java 的编码规范

双向工程



确保模型和代码的一致



著名的竞赛

- ACM ICPC
  - http://icpc.baylor.edu/icpc/
- topcoder
  - http://www.topcoder.com/
- ICSE软件工程竞赛 **SCORE**
  - http://score.elet.polimi.it
- ACM研究竞赛 SRC: Student Research Competition
  - http://www.acm.org/src/
  - 以学术论文来参加竞赛
  - 论文全部发表在各国际会议或杂志上

编码和版本管理

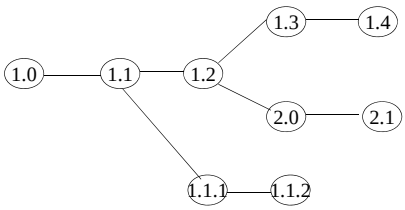
- 程序设计语言
- 编码准则和规范
- 软件版本管理
- 软件持续集成

软件配置项

- 软件版本管理的基本实体是软件配置项(Software Configuration Item, SCI)。
- 一个软件配置项是在软件生存周期所产生或使用的一个工作产品或一组相关的工作产品，包括代码、文档、模型、数据等。

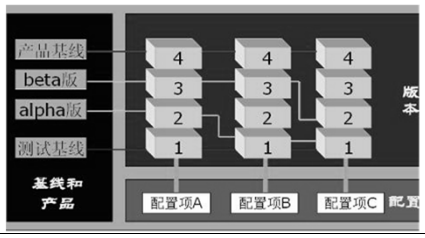
版本 version

- 配置项在软件开发过程中会不断变更，形成多个版本(Version)。
- 每个配置项的版本演化历史可以形象地表示为图形化的版本树(Version Tree)。



基线 baseline

- 基线是项目储存库中每个配置项版本在特定时期的一个“快照”，它提供一个正式标准，随后的工作基于此标准，并且只有经过授权后才能变更这个标准。
- 通常将交付给客户的基线称为一个“发布”（Release），为内部开发用的基线则称为一个“构建”（Build）。

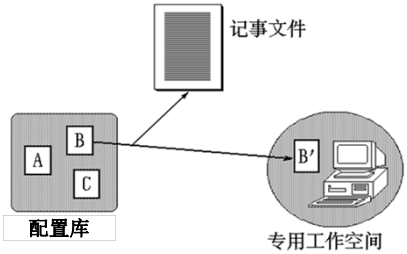


版本控制

- 版本管理记录每个配置项的变更履历，并控制基线的生成。
- 目的：
  - 对软件开发过程中配置项的各个版本提供有效的追踪手段，保证在需要时可回到旧版本，避免文件丢失和相互覆盖；
  - 通过对版本库的访问控制避免未经授权的访问和修改，达到有效保护软件资产和知识产权的目的；
  - 实现团队并行开发、提高开发效率。

配置库CM Repository

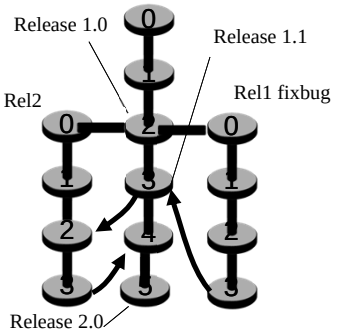
- 存储配置管理信息及软件软件配置项的版本信息。



Check-in和Check-out

- 软件配置项通过检入(Check-in)，进入配置库，开始“冻结”
- 由于各种原因需要变更，从配置库中检出（Check-out）配置项
- check in和check out通过加锁协调多用户操作
- 每次check in时，在配置库上都会生成新的版本

并行开发

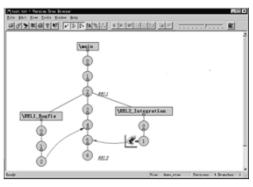


版本控制的最佳实践

- 合理组织项目及子项目结构
- 避免多人Check-Out
- 合理管理权限
- 避免长时间不Check-in以及不Get Last Version
- 避免强行修改未Check-Out的本地文件的Read-Only属性
- 建议代码Check-in之前需通过单元测试
- 每天或定期备份所有数据

版本控制工具

- Merant PVCS
- IBM Rational Clearcase
- Microsoft Visual Source Safe
- CVS (Freeware)
- SVN (Freeware)
- Gitlab(Freeware)
- Google Code (云平台)
- GitHub (云平台)
- …



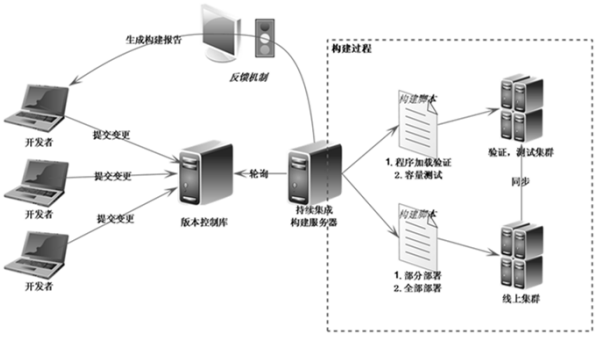
编码和版本管理

- 程序设计语言
- 编码准则和规范
- 软件版本管理
- 软件持续集成

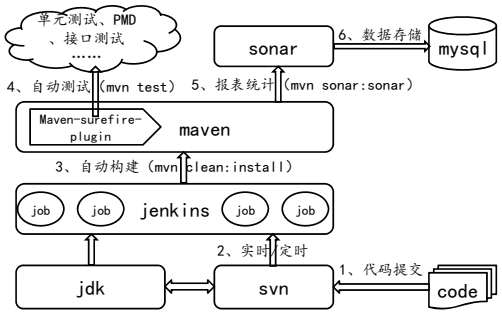
持续集成概念

- **持续集成是一种软件开发实践，即团队开发成员经常集成它们的工作，通常每个成员每天至少集成一次，也就意味着每天可能会发生多次集成。**
- **每次集成都通过自动化的构建（包括编译，发布，自动化测试）来验证，从而尽快地发现集成错误。**

持续集成流程



持续集成示例



Shanghai Jiao Tong University

软件工程

Module: 软件测试

上海交通大学软件工程中心

本节内容

- **测试概述**
- 测试技术
- 测试策略
- 测试过程
- 自动化测试

@第8章.教材

测试（testing）的目的

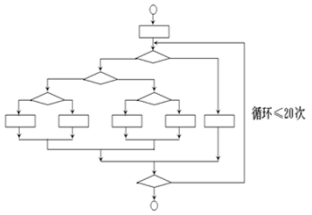
- 测试（testing）的目的
  - 发现软件的错误，从而保证软件质量
- 成功的测试
  - 发现了未曾发现的错误
- 与调试（debugging）的不同在哪？
  - 定位和纠正错误
  - 保证程序的可靠运行

有关软件测试的错误观点

“软件测试是为了证明程序是正确的，即测试能发现程序中所有的错误”。事实上这是不可能的。要通过测试发现程序中的所有错误，就要穷举所有可能的输入数据。

对于一个输入三个16位字长的整型数据的程序，输入数据的所有组合情况有$2^{48} \approx 3*10^{14}$，如果测试一个数据需1ms，则即使一年365天一天24小时不停地测试，也需要约1万年。

对一个具有多重选择和循环嵌套的程序，不同的路径数目可能是天文数字。例如一个小程序的流程图，它包括了一个执行20次的循环，其循环体有五个分支。这个循环的不同执行路径数达520条，如果对每一条路径进行测试需要1毫秒，那么即使一年工作365×24小时，要想把所有路径测试完，大约需3170年。

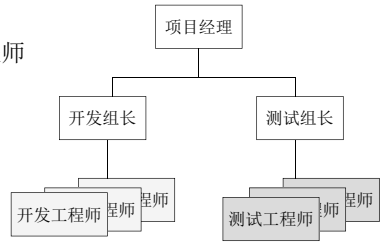


测试准则

- 所有的软件测试应追溯到用户的需求
- 穷举测试是不可能的
- 根据软件错误的聚集性规律，对存在错误的程序段应进行重点测试
- 尽早地和不断地进行软件测试
- 避免测试自己的程序
- 制定测试计划，避免测试的随意性
- 测试应该从小到大

测试组的组成

- 测试经理/测试组长
- 测试工程师
  - 测试开发工程师
  - 测试工程师



测试工程师和开发工程师的能力要求和工作习惯是不同的

本节内容

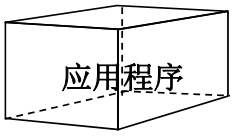
- 测试概述
- **测试技术**
- 测试策略
- 测试过程
- 自动化测试

软件测试技术

	白盒测试	黑盒测试
基于直觉和经验	即兴测试*	
	探索式测试*	
基于代码	控制流测试*	
	基本路径测试*	
	数据流测试	
基于规约		等价类划分*
		边界值分析*
		随机测试*
基于错误	错误猜测*	
	变异测试	
基于模型		因果图/判定表
		基于有限状态机的测试
		基于形式化规约的测试
专用测试技术（即基于应用类型）	面向对象的测试	
	基于构件的测试	
	并发程序的测试	
		基于Web的测试
		图形用户界面的测试
		协议一致性的测试
		实时系统的测试

白盒测试

- 白盒测试把被测软件看作一个透明的白盒子，测试人员可以完全了解软件的设计或代码，按照软件内部逻辑进行测试。
- 白盒测试又称玻璃盒测试。



控制流测试（属白盒测试）

1）语句覆盖法
2）判定覆盖（分支）
3）条件覆盖
4）判定/条件覆盖
5）条件组合覆盖
6）路径覆盖

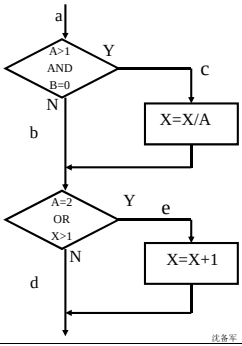
1）语句覆盖法

- 使得程序中的每一个语句至少被遍历一次
- 举例：对下列子程序进行测试

```
procedure example(y,z:real;var x:real);
begin
    if (y>1) and (z=0) then x:=x/y;
    if (y=2) or (x>1) then x:=x+1;
end;
```

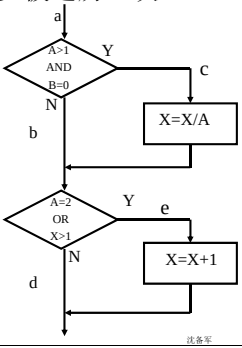
程序流程图

- 测试用例：
  A=2，B=0，X=3



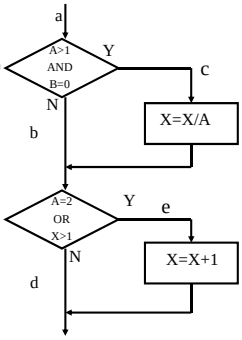
2）判定覆盖（分支）

- 使得程序中每一个分支至少被遍历一次
- 测试用例
  1. A=2，B=0，X=1
     （沿路径 ace）
  2. A=1，B=0，X=0
     （沿路径abd）



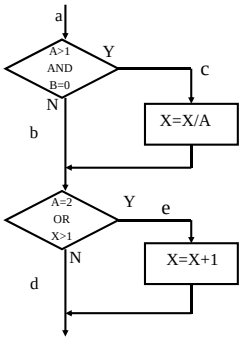
3）条件覆盖

- 使得每个判定的条件获取各种可能的结果
- 在a点 A>1，A≤1，B=0，B≠0
- 在b点 A=2，A≠2，X>1，X≤1
- 测试用例
  1. A=2，B=0，X=4
     （沿路径ace）
  2. A=1，B=1，X=1
     （沿路径abd）



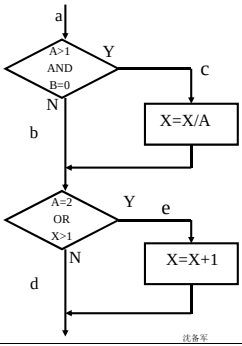
4）判定/条件覆盖

- 使得判定中的条件取得各种可能的值，并使得每个判定取得各种可能的结果
- 测试用例
  1. A=2，B=0，X=4
     （沿路径ace）
  2. A=1，B=1，X=1
     （沿路径abd）



5）条件组合覆盖

- 使得每个判定条件的各种可能组合都至少出现一次
- 要求
  1. A>1，B= 0
  2. A>1，B≠0
  3. A≤1，B=0
  4. A≤1，B≠0
  5. A=2，X>1
  6. A=2，X≤1
  7. A≠2，X>1
  8. A≠2，X≤1
- 测试用例
  1. A=2，B=0，X=4
  2. A=2，B=1，X=1
  3. A=1，B=0，X=2
  4. A=1，B=1，X=1

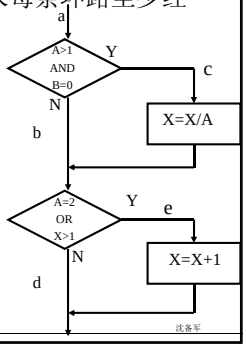


6）路径覆盖

- 覆盖程序中所有可能的路径
- 如果程序中包含环路，则要求每条环路至少经过一次

测试用例：

A	B	X	覆盖路径	
2	0	3	a c e	L_1
1	0	1	a b d	L_2
2	1	1	a b e	L_3
3	0	1	a c d	L_4

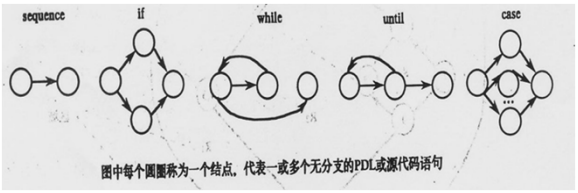


基本路径测试（属白盒测试）

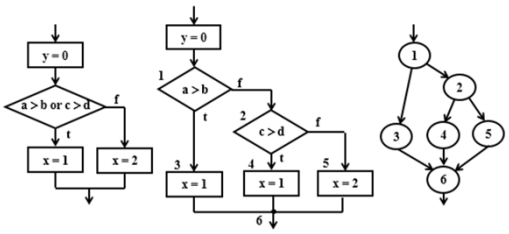
- 在实际问题中，一个不太复杂的程序，特别是包含循环的程序，其路径数可能非常大。因此测试常常难以做到覆盖程序中的所有路径，为此，我们希望把测试的程序路径数压缩到一定的范围内。
- 基本路径测试是Tom McCabe提出的一种白盒测试技术，这种方法首先根据程序或程序流程图画出控制流图（flow graph），并计算其区域数，然后确定一组独立的程序执行路径（称为基本路径），最后为每一条基本路径设计一个测试用例。

程序的控制流图

- 控制流图由结点和边组成，分别用圆和箭头表示。程序流程图中一个连续的处理框（对应于程序中的顺序语句）序列和一个判定框（对应于程序中的条件控制语句）映射成控制流图中的一个结点，程序流程图中的箭头（对应于程序中的控制转向）映射成控制流图中的一条边。对于程序流程图中多个箭头的交汇点可以映射成控制流图中的一个结点（空结点）。

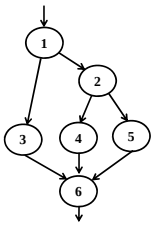


上述映射的前提是程序流程图的判定中不包含复合条件。如果判定中包含了复合条件，那么必须先将其转换成等价的简单条件的程序流程图。

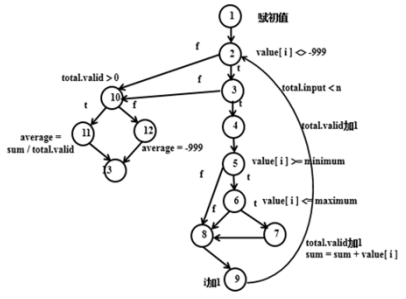


a）含复合条件的程序流程图　b）只含简单条件的程序流程图　c）对应的控制流图

- 我们把流图中由结点和边组成的闭合部分称为一个区域（region），在计算区域数时，图的外部部分也作为一个区域。例如，右图所示的流图的区域数为3。
- 独立路径是指程序中至少引进一个新的处理语句序列或一个新条件的任一路径，在流图中，独立路径至少包含一条在定义该路径之前未曾用到过的边。在基本路径测试时，独立路径的数目就是流图的区域数。



例如，对一个PDL程序进行基本路径测试，该程序的功能是：最多输入N个值（以-999为输入结束标志），计算位于给定范围内的那些值（称为有效输入值）的平均值，以及输入值的个数和有效值的个数。



其区域数为6，我们选取独立路径如下：

- 路径1：1-2-10-11-13
- 路径2：1-2-10-12-13
- 路径3：1-2-3-10-11-13
- 路径4：1-2-3-4-5-8-9-2-10-12-13
- 路径5：1-2-3-4-5-6-8-9-2-10-12-13
- 路径6：1-2-3-4-5-6-7-8-9-2-10-11-13

为每一条独立路径设计测试用例。

假设：n = 5；minimum = 0；maximum = 100。

路径1： 1-2-10-11-13
- 测试数据：value = [90，-999，0，0，0]
- 预期结果：Average = 90，total.input = 1，total.valid = 1

路径2： 1-2-10-12-13
- 测试数据：value = [-999 ，0，0，0，0]
- 预期结果：Average = -999，total.input = 0，total.valid = 0

路径3： 1-2-3-10-11-13
- 测试数据：value = [-1，90，70，-1，80]
- 预期结果：Average = 80，total.input = 5，total.valid = 3

路径4： 1-2-3-4-5-8-9-2-10-12-13
- 测试数据：value = [-1，-2，-3，-4，-999]
- 预期结果：Average = -999，total.input = 4，total.valid = 0

路径5： 1-2-3-4-5-6-8-9-2-10-12-13
- 测试数据：value = [120，110，101，-999，0]
- 预期结果：Average = -999，total.input = 3，total.valid = 0

路径6： 1-2-3-4-5-6-7-8-9-2-10-11-13
- 测试数据：value = [95，90，70，65，-999]
- 预期结果：Average = 80，total.input = 4，total.valid = 4

黑盒测试

- 黑盒测试把程序看成一个黑盒子，完全不考虑程序内部结构和处理过程。
- 黑盒测试是在程序接口进行测试，它只是检查程序功能是否按照需求规约正常使用。
- 黑盒测试又称功能测试、行为测试，在软件开发后期执行。

几种黑盒测试技术

1）等价类划分法
2）边界值划分法
3）错误推测法
4）因果图法

1）等价类划分(equivalence partitioning)

- 基本概念
  - 试遍所有输入数据是不可能的
  - 等价类划分的办法是把程序的输入域划分成若干部分，然后从每个部分中选取少数代表性数据当作测试用例。
  - 输入的数据划分为合理等价类和不合理等价类
- 例子：
  - 输入数据：每个学生可以选取修1至3门课程则
  - 合理等价类：选修1至3门课程
  - 不合理等价类：没选修课程以及超过3门课程

确定等价类原则

（1）如果输入条件规定了取值范围或值的个数，则可确定一个有效等价类和两个无效等价类。
- 例1 “…项数可以从1到999…”
  - 有效等价类为 “1≤ 项数≤ 999”
  - 无效等价类为 “项数<1” 及 “项数>999”
- 例2 “学生选课允许2门至4门”
  - 有效等价类： 选课2至4门
  - 无效等价类： 只选一门课或未选课
    选课超过4门

确定等价类原则（续）

（2）输入条件规定了输入值的集合，或是规定了“必须如何”的条件，则可确定一个有效等价类和一个无效等价类。
- 例：“必须以标识符以字母开头的字符串”
  - 有效等价类： 以字母开头的字符串
  - 无效等价类： 以非字母开头的字符串

（3）如果确知，已划分的等价类中各元素在程序中的处理方式是不同的，则应将此等价类进一步划小。

设计测试用例

（1）设计一个测试用例，使其尽可能多地覆盖有效等价类，重复这一步，最终使得所有有效等价类均被覆盖。

（2）设计一个测试用例，使其只覆盖一个无效等价类，重复这一步，最终使得所有无效等价类均被覆盖。

- 例：项数可以从1到999
  - 有效等价类“1≤ 项数≤ 999” -- 777
  - 无效等价类
    - “项数<1” --- 0
    - “项数>999” --- 1200

2）边界值划分(boundary value analysis)

- 基本概念
  - 边界值划分法使被测程序在边界值及其附近运行，从而更有效地暴露程序中潜藏的错误
  - 不仅根据输入条件，它还根据输出情况设计测试用例
- 例子：
  - 输入条件-1.0到1.0
  - 则选择-1.0，1.0，-1.001和1.001等

边值分析遵循的原则

①如果输入条件规定了取值范围，或是规定了值的个数，应以该范围的边界内及刚刚超出范围的边界外的值，或是分别对最大、最小个数及稍小于最小、稍大于最大个数作为测试用例。

- 例1：“重量在10公斤至50公斤范围内的邮件，其邮费计算公式为……”
  - 应取10及50，还应取10.01，49.99，9.99，50.01等
- 例2： “某输入文件可包含1至255个记录，……”
  - 可取1和255，还应取0及256等

边值分析遵循的原则（续）

②针对需求的每个输出条件使用前面的第①条原则。

- 例1：计算出“每月保险金扣除额为0至1165.25元”
  - 可取0．00及1165.2、还可取 -0.01及1165.26等
- 例2： 情报检索系统要求每次“最多显示1～4条情报摘要”
  - 可取1和4，还应取0和5等。

边值分析遵循的原则（续）

③如果程序规格说明中提到的输入或输出域是个有序的集合（如顺序文件、表格等），就应注意选取有序集的第一个和最后一个元素作为测试用例。

3）错误推测法(error guessing)

- 基本概念
  - 猜测被测程序在哪些地方容易出错
  - 针对可能的薄弱环节来设计测试用例
- 例子1：对一个排序程序，可测试：
  - 输入表为空
  - 输入表中只有一行
  - 输入表中所有的值具有相同的值
  - 输入表已经是排序的
- 例子2：测试二分法检索子程序，可考虑：
  - 表中只有一个元素
  - 表长为2n
  - 表长为2n-1
  - 表长为2n+1

4）因果图法（Cause - Effect Graphics）

- When
  - 检查输入条件的各种组合情况
  - 在等价类划分方法和边界值方法中未考虑输入条件的各种组合，当输入条件比较多时，输入条件组合的数目会相当大
- How
  - 分析需求规约，找出因（输入条件）和果（输出或程序状态的修改）
  - 画出因果图
  - 通过因果图功能说明转换成一张判定表，然后为判定表的每一例设计测试用例

例如，有一个处理单价为5角钱的饮料自动售货机软件，其需求规约如下：

有一个处理单价为1元5角的盒装饮料的自动售货机软件。若投入1元5角硬币，按下“可乐”，“雪碧”或“红茶”按钮，相应的饮料就送出来。若投入的是两元硬币，在送出饮料的同时退还5角硬币。

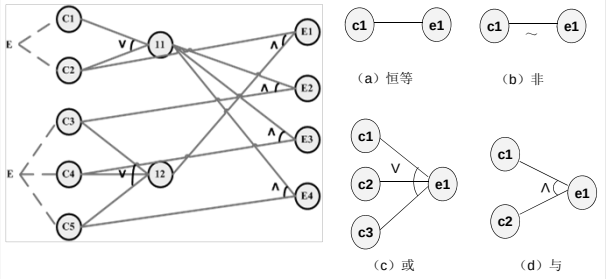
步骤一 分析需求规约，列出原因和结果

原因	c1:投入1元5角硬币；
	c2:投入2元硬币；
	c3:按“可乐”按钮；
	c4:按“雪碧”按钮；
	c5:按“红茶”按钮；
中间	11：已投币
状态	12：已按钮
结果	a1:退还5角硬币；
	a2:送出“可乐”饮料；
	a3:送出“雪碧”饮料；
	a4:送出“红茶”饮料；

步骤二 画出因果图

所有原因结点列在左边，所有结果结点列在右边。

由于原因C1与C2，C3，C4与C5不能同时发生，分别加上互斥约束E。



步骤三 根据因果图画出判定表

	1	2	3	4	5	6	7	8	9	10	11
c1:投入1元5角硬币	1	1	1	1	0	0	0	0	0	0	0
c2:投入2元硬币	0	0	0	0	1	1	1	1	0	0	0
c3:按“可乐”按钮	1	0	0	0	1	0	0	0	1	0	0
c4:按“雪碧”按钮	0	1	0	0	0	1	0	0	0	1	0
c5:按“红茶”按钮	0	0	1	0	0	0	1	0	0	0	1
11：已投币	1	1	1	1	1	1	1	1	0	0	0
12：已按钮	1	1	1	0	1	1	1	0	1	1	1
a1:退还5角硬币					√	√	√				
a2:送出“可乐”饮料	√				√						
a3:送出“雪碧”饮料		√				√					
a4:送出“红茶”饮料			√				√				

步骤四 根据判定表设计测试用例

为判定表的每个有意义的列设计一个测试用例

用例编号	测试用例	预期输出
1	投入1元5角，按“可乐”	送出“可乐”饮料
2	投入1元5角，按“雪碧”	送出“雪碧”饮料
3	投入1元5角，按“红茶”	送出“红茶”饮料
4	投入2元，按“可乐”	找5角，送出“可乐”
5	投入2元，按“雪碧”	找5角，送出“雪碧”
6	投入2元，按“红茶”	找5角，送出“红茶”

综合运用多种黑盒测试用例设计策略

- 不管情况怎样，都使用边值分析方法；
- 对输入和输出划分合格的和不合格的两个等价类，作为补充；
- 如果规范中含有输入条件的组合，便从因果图开始：
  原因：输入条件或输入条件的等价类；
  结果：输出条件或系统变换；
  因果图：连接原因与结果的布尔图；
- 使用“猜测”技巧，增加一些测试用例

讨论：三角形测试

- 从键盘上输入三个整数，这三个数值表示三角形三条边的长度。然后，输出信息，以表明这个三角形是等腰、等边或是一般三角形，或不能构成三角形。
- 请采用黑盒测试方法设计测试用例。

本节内容

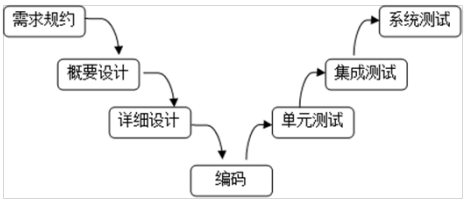
- 测试概述
- 测试技术
- **测试策略**
- 测试过程
- 自动化测试

测试策略

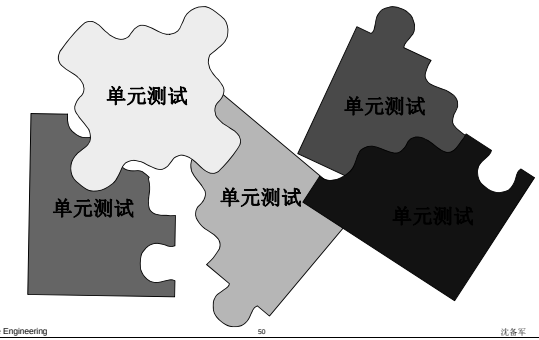
- 按测试层次分类
  - 单元测试、集成测试、系统测试
- 按软件质量属性分类
  - 功能性测试、可靠性测试、性能测试、易用性测试、可移植性测试、可维护性测试
- 其他测试策略
  - 验收测试、α 测试、β 测试、安装测试、回归测试

测试层次

- 不同层次的测试：
  - 单元测试 (Unit testing)
  - 集成测试 (Integration testing)
  - 系统测试 (System testing)

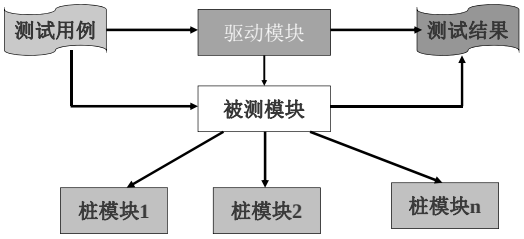


单元测试



单元测试

- 单元测试（unit testing），又称为模块测试，是针对软件结构中独立的基本单元（如函数、子过程、类）进行的测试。



单元测试——测什么？

- 单元测试是针对每个基本单元，重点关注5个方面：
  - 模块接口
    - 保证被测基本单元的信息能够正常地流入和流出
  - 局部数据结构
    - 确保临时存储的数据在算法的整个执行过程中能维持其完整性
  - 边界条件
    - 在到达边界值的极限或受限处理的情形下仍能正确执行
  - 独立的路径
    - 执行控制结构中的所有独立路径以确保基本单元中的所有语句至少执行一次
  - 错误处理路径
    - 对所有的错误处理路径进行测试

常采用白盒测试技术

单元测试——何时测试和由谁测?

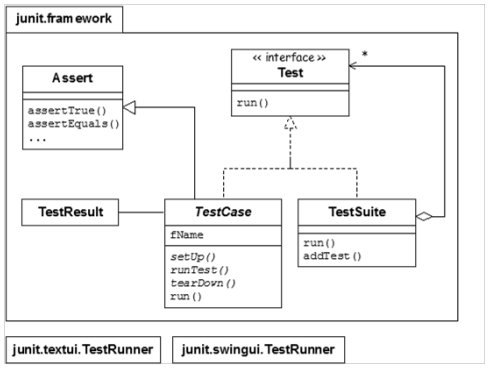
- When
  - 一般地，该基本单元的编码完成后就可以对其进行单元测试。
  - 也可以提前，即测试驱动开发 (test driven development)，在详细设计的时候就编写测试用例，然后再编写程序代码来满足这些测试用例。
- Who
  - 单元测试可看作是编码工作的一部分，应该由程序员完成，也就是说，经过了单元测试的代码才是已完成的代码，提交产品代码时也要同时提交测试代码。
  - 测试小组可以对此作一定程度的审核。

自动化的单元测试

测试驱动的开发（TDD）

- 只有用代码编写的UT，才能够重现，才能真正节约未来手工测试的时间。
- 只有用代码编写的UT ,才能做到自动化，才能在软件开发的任何时候都能快速，简单的大批量执行，保证能准确地定位错误，保证不会因为修改而引入新的错误。在系统开发的后期尤为明显。
- 目前最流行的单元测试工具是xUnit系列框架，常用的根据语言不同分为JUnit（java），CppUnit（C++），DUnit （Delphi ），NUnit（.net），PhpUnit（Php ）等等。

JUnit框架



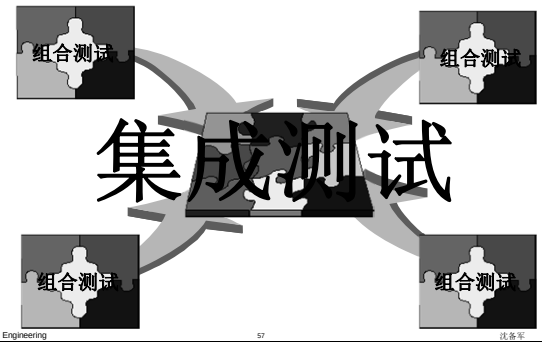
JUnit 测试用例（Test Case）

```
public class TestPerson extends TestCase {
   /** A unit test to verify the name is formatted correctly.     */
   public void testGetFullName( ) {
      Person p = new Person("Aidan", "Burke");
      assertEquals("Aidan Burke", p.getFullName( ));
   }

   /**   A unit test to verify that nulls are handled properly.     */
   public void testNullsInName( ) {
      Person p = new Person(null, "Burke");
      assertEquals("? Burke", p.getFullName( ));
      // this code is only executed if the previous assertEquals
      p = new Person("Tanner", null);
      assertEquals("Tanner ?", p.getFullName( ));
   }
}
```

Test case 扩展自TestCase,每个方法负责对某种情况测试，测试结果为true/false

集成测试



集成测试

- 集成测试（integration testing），又称组装测试，它根据设计将软件模块组装起来，进行有序的、递增的测试，并通过测试评价它们之间的交互。
- 集成测试重点关注：
  - 在把各个软件单元连接起来的时候，穿越单元接口的数据是否会丢失；
  - 一个软件单元的功能是否会对另一个软件单元的功能产生不利的影响；
  - 各个子功能组合起来，能否达到预期要求的父功能；
  - 全局数据结构是否有问题；
  - 单个软件单元的误差累积起来，是否会放大，从而达到不能接受的程度。

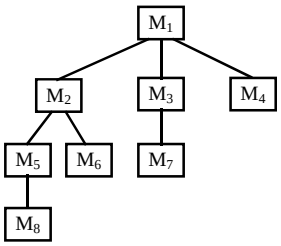
常采用白盒测试+黑盒测试技术

软件集成策略

- 增量式集成
  - 自顶向下集成
  - 由底向上集成
  - 混合方式集成
    - 对软件中上层使用自顶向下集成，对软件的中下层采用自底向上集成。
- 一次性集成
  - 缺点：接口错误发现晚，错误定位困难
  - 优点：可以并行测试和调试所有软件单元

集成测试示例

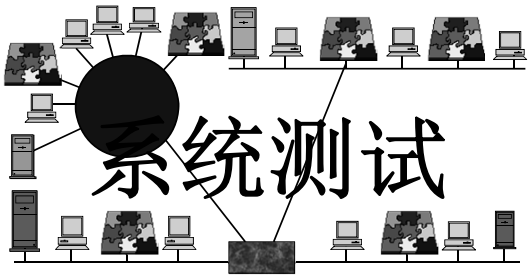
- 自顶向下深度优先测试次序：
  M_1、M_2、M_5、M_8、M_6、M_3、M_7、M_4
- 自顶向下广度优先测试次序：
  M_1、M_2、M_3、M_4、M_5、M_6、M_7、M_8
- 自底向上集成测试次序：
  M_8、M_5、M_6、M_7、M_2、M_3、M_4、M_1



集成测试——何时测试和由谁测?

- When
  - 根据集成测试策略，和单元测试并行或之后进行。
- Who
  - 一般由项目经理组织软件测试工程师或由独立的测试部门进行。
  - 程序员也可测试。

系统测试



系统测试

常采用黑盒测试技术

- When
  - 软件集成及集成测试完成后，对整个软件系统进行的一系列测试，称为系统测试（system testing）。
- Why
  - 系统测试的目的是为了验证系统是否满足需求规约。
- What
  - 测试内容包括功能测试和非功能测试，其中非功能测试常常是系统测试的重点，例如：可靠性测试、性能测试、易用性测试、可维护性测试、可移植性测试等。
  - 如果该软件只是一个大的计算机系统的一个组成部分，此时应将软件与计算机系统的其他元素集成起来，检验它能否与计算机系统的其他元素协调地工作。
- Who
  - 系统测试一般由与开发无直接责任关系的独立方负责，例如项目组的软件测试工程师、测试部门、第三方评测机构、客户等

软件质量属性的测试

- 功能性测试
- 可靠性测试
- 性能测试
- 易用性测试
- 可移植性测试
- 可维护性测试

功能性测试

- 功能性测试（functionality testing），又称为正确性测试或一致性测试，其目的是用以确认软件在指定条件下使用时，软件产品提供满足明确和隐含要求的功能的能力。
  - **适用性测试**，测试软件为指定的任务和用户目标提供一组合适的功能的能力。
  - **准确性测试**，测试软件提供具有所需精度的正确或相符的结果或效果的能力。
  - **互操作测试**，测试软件与一个或更多的规定系统进行交互的能力。
  - **安全性测试**，测试软件保护信息和数据的能力，以使未授权的人员或系统不能阅读或修改这些信息和数据，而不拒绝授权人员或系统对它们的访问。
  - **功能依从性测试**，测试规约中所有的功能都应实现，而且应是正确的。

- 例如，4S系统的系统测试时
  - 按其用例模型进行功能依从性测试，覆盖每个用例的各个基本流和备选流；
  - 在系统安全性测试时，测试者扮演一个试图攻击系统的角色，采用各种方式攻击系统：
    - 截获或破译4S系统的用户名和密码；借助于某种软件攻击4S系统；
    - “制服”4S系统，使得别人无法访问；故意引发系统错误，期望在系统恢复过程中侵入系统；
    - 通过浏览非保密的数据，从中找到进入4S系统的钥匙等等。

可靠性测试

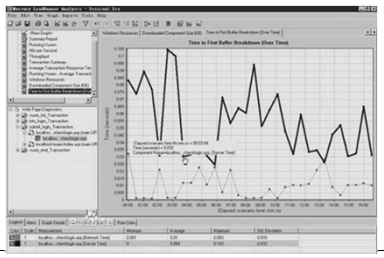
- 软件可靠性测试（reliability testing）用以测试在故障发生时，软件产品维持规定的绩效级别的能力。
  - **成熟性测试**，测试软件为避免由软件中故障而导致失效的能力。
  - **容错性测试**，测试在软件出现故障或者违反其指定接口的情况下，软件维持规定的性能级别的能力。
  - **易恢复性测试**，测试在失效发生的情况下，软件重建规定的性能级别并恢复受直接影响的数据的能力。
  - **可靠性的依从性测试**，测试软件遵循与可靠性相关的规约、标准或法规的能力。

- 例如，4S系统的系统测试时，
  - 连续运行4S系统100小时，计算系统正常运行时间占总时间的比率，以及故障恢复的平均用时；
  - 通过强制系统重启、拔网线等各种手段，让4S系统发生故障，然后观察系统是否还能降级运行，是否能及时地恢复运行，数据库中的数据是否因此留下了“脏”数据等。

性能测试

- 性能测试（performance testing）用来测试软件在规定条件下，相对于所用资源的数量，可提供适当性能的能力。
  - **时间特性测试**：测试在规定条件下，软件执行其功能时，提供适当的响应和处理时间以及吞吐率的能力。
  - **资源利用性测试**：测试在规定条件下，软件执行其功能时，使用合适数量和类别的资源的能力。这些资源包括CPU、内存、网络等。
  - **性能依从性测试**：测试软件遵循性能相关的规约、标准或法规的能力。
- 压力测试（stress testing），又称强度测试，是一种超常情况下的性能测试。它需要在超常数量、频率或资源的方式下执行系统，以获得系统对非正常情况下（如大数据量的输入、处理和输出，大并发数等）的承受程度。
- 自动化的性能/压力测试：HP的Loadrunner，开源的Jmeter等。

- 例如，4S系统的系统测试时，
  - 采用Loadrunner进行自动化性能测试，模拟不同数量的并发用户，在不同的数据量情况下，观察系统的响应时间和资源使用情况。



易用性测试

- 易用性测试（usability testing）用以评价用户学习和使用软件（包括用户文档）的难易程度、支持用户任务的有效程度、从用户的错误中恢复的能力。
  - **易理解性测试**，测试软件使用户能理解软件是否合适以及如何能将软件用于特定的任务和使用条件的能力。
  - **易学性测试**，测试软件使用户能学习其应用的能力。
  - **易操作性测试**，测试软件使用户能操作和控制它的能力。
  - **吸引性测试**，测试软件吸引用户的能力。
  - **易用依从性测试**，测试软件遵循易用相关的规约、标准、风格指南或法规的能力。
- 易用性测试可以采用模拟用户的方式进行，也可以通过观察用户的操作行为来执行。

- 例如，4S系统的易用性测试时，
  - 测量用户所需的培训时间
  - 观察销售人员完成下订单的任务平均需要多少步骤
  - 测试各界面的风格是否一致
  - 信息显示和反馈是否准确
  - 是否提供在线的支持帮助等等。

可移植性测试

- 可移植性测试（portability testing）用以测试软件从一种环境迁移到另外一种环境的能力。
  - **适应性测试**，测试软件毋需采用额外的活动或手段就可适应不同指定环境的能力。
  - **易安装性测试**，测试软件在指定环境中被安装的能力。
  - **共存性测试**，测试软件在公共环境中同与其分享公共资源的其他独立软件共存的能力。
  - **易替换性测试**，测试软件在同样环境下，替代另一个相同用途的指定软件产品的能力。
  - 可移植性的依从性测试，测试软件遵循可移植性相关的规约、标准或法规的能力。

- 例如，4S系统的可移植性测试时，
  - 在支持的MS Window和Linux两种操作系统的规定版本上，可以方便地进行4S系统的安装。
  - 用IE、Firefox等多个版本的浏览器都能正常访问。
  - 能和已安装的MS Office等软件共存。
  - 同时检查4S系统提供的安装文档和初始化数据。

可维护性测试

- 可维护性测试（maintainability testing）用以测试软件可被修改的能力，包括纠正、改进或软件对环境、需求变化的适应。
  - **易分析性测试**，测试软件诊断缺陷或失效原因或识别待修改部分的能力。
  - **易改变性测试**，测试软件使指定的修改可以被实现的能力。
  - **稳定性测试**，测试软件避免由于软件修改而造成意外结果的能力。
  - **易测试性测试**，测试软件使已修改软件能被确认的能力。
  - **维护依从性测试**，测试软件遵循可维护性相关的规约、标准或法规的能力。

- 一般，软件可维护性更多的是通过度量来进行评估。
  - 常用可维护性度量：软件的复杂性、规模、可重用性、耦合度和内聚度、注解代码率、缺陷的平均修复成本、变更的平均实现成本等。
  - 软件越复杂、规模越大、可重用性越低、耦合度越在、内聚度越小、注解代码行数所占比率越小、缺陷的平均修复成本或变更的平均实现成本越高，则软件可维护性越差。

其他测试策略

- 验收测试
- α 测试和 β 测试
- 安装测试
- 回归测试

验收测试

- 针对应用型软件
- 所谓验收测试（acceptance testing），是确定一个开发完成的软件系统是否符合其验收准则，使用户/客户或其他授权实体能确定是否接受此软件系统的正式测试。
- 验收测试是以用户为主的测试。验收测试原则上在用户所在地进行，但如经用户同意也可以在开发机构内模拟用户环境下进行。
- 项目经理负责组织验收组进行最终验收测试。验收组应由用户代表、相关专家、项目组成员等组成。验收组根据合同、《需求规约》或《验收测试计划》对软件成品进行验收测试。

α测试和β测试

- 针对通用产品
- α 测试是邀请小规模的、有代表性的潜在用户，在开发环境中，由开发者“指导”下进行的测试（试用），开发者负责记录使用中出现的问题和软件的缺陷，因此α测试是在一个受控的环境中进行的。
- β 测试是由用户在一个或多个用户环境下进行的测试，是产品正式发布前的系统测试形式。一组有代表性的用户和消费者在典型操作条件下尝试做常规使用，由用户记录下测试中发现的问题或任何希望改进的建议，报告给开发者。
- β 测试与α 测试不同的是，β 测试时开发者通常不在测试现场，由用户去使用，软件在一个开发者不能控制的环境中的“活的”试用。

安装测试

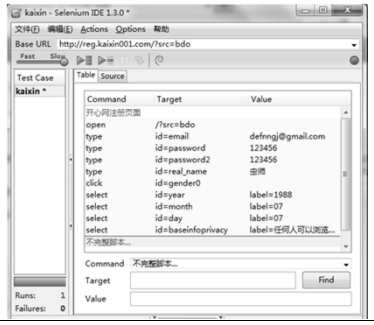
- 安装测试（installation testing）用以确保该软件在各种支持的平台下，各种允许的安装情况下，都能成功安装。
  - *首次安装、升级、完整的或自定义的安装等*
- 除了安装软件，常常还需要安装用户文档以及初始化数据，这些同样需要进行检查。

回归测试（regression testing）

- 为了保证软件返工时没有引进新的错误，要全部或部分地重复以前做过的测试。
- 在集成测试、缺陷纠正后的重新测试、迭代开发的后续迭代测试中常常用到回归测试。
- 回归测试可以手工执行，也可以使用自动化的回归测试工具（又称功能测试工具）。回归测试工具使得软件工程师能够捕获到执行过的测试，然后进行回放和比较。
- 回归测试应重新执行所有执行过的测试，或者对受影响的软件部分进行局部回归测试。仅仅对修改的软件部分进行重新测试常常不够的。

自动化的回归测试

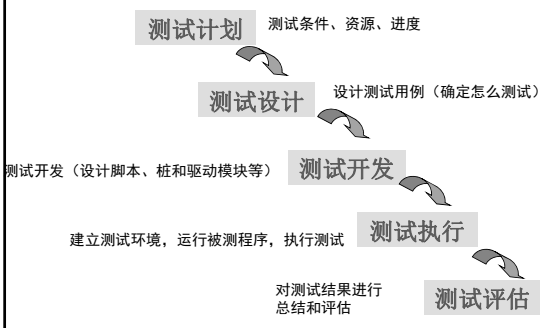
- HP的QTP
- 开源的selenium



本节内容

- 测试概述
- 测试技术
- 测试策略
- **测试过程**
- 自动化测试

测试过程



- 测试计划、测试设计和测试开发在软件开发完成前进行
- 测试执行只能在软件开发完成后进行

1）测试计划

- 测试目的
- 测试对象
- 测试范围
- 文档的检验
- 测试策略和测试技术
- 测试过程
- 进度安排
- 资源
- 测试开始、结束标准
- 测试文档和测试记录

开始测试的标准

- 在测试计划中规定开始测试的标准
- 开始测试的常用标准
  - 通过冒烟测试

2）测试设计

- 任务：设计测试用例
- 测试用例（test case）是按一定顺序执行的与测试目标相关的一系列测试。其主要内容包括：
  - 前置条件 (Pre-conditions) Optional
  - 测试输入 (Test input)
  - 观察点 (Observation Points) Optional
  - 控制点 (Control Points) Optional
  - 期望结果 (Expected Results)
  - 后置条件 (Post-conditions) Optional

示例：测试用例/测试记录表

××××公司×××项目测试用例及测试实施记录 记录编号：YDDZ2000-SYDAS2000TP02
项目编号：SYDAS2000 项目名称：XXXX配网自动化系统 测试规范版本号：2.0
开发负责人：xx 开发部门：配网 测试规范最新更新时间：2001-03-01

编号	标题	步骤	期望结果	现象及记录	结果	开发人员反馈
1	系统设置模块					
1-1	用户管理		添加,修改，删除用户，设置权限。			
1-1-1	添加用户	1.点击菜单中的用户管理菜单项 进入用户管理窗口。 2.点击（新建）命令按钮。 3.然后，分别输入用户信息。 4.最后，点击（保存）命令按钮。	添加用户			

3）测试开发

- 任务：开发测试脚本、桩和驱动模块
- 测试脚本（test script）是具有正规语法的数据和指令的集合，在测试执行自动工具使用中，通常以文件形式保存

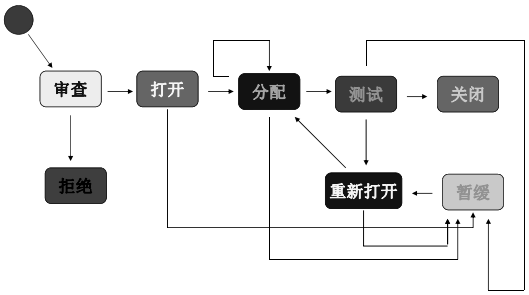
4）测试执行

- 任务：执行测试用例
- 对于手动测试：
  - 测试者按事先准备好的手工过程进行测试，测试者输入数据、观察输出、记录发现的问题。
- 对于自动测试：
  - 可能只需要启动测试工具，并告诉工具执行哪些测试用例。
- 文档：测试记录、缺陷报告

缺陷报告

- 内容包括：缺陷名称、分类、等级、发现时间，发现人，所执行的测试用例、现象等
- 缺陷等级
  - 5级：灾难性的--系统崩溃、数据被破坏
  - 4级：很严重的--数据被破坏
  - 3级：严重的--特性不能运行，无法替代
  - 2级：中等的--特性不能运行，可替代
  - 1级：烦恼的--提示不正确，报警不确切
  - 0级：轻微的--表面化的错误，拼写错等
- 缺陷报告通常保存在缺陷跟踪系统

缺陷跟踪系统



例如：开源的bugzilla和Mantis软件

5）测试评估

- 通过评估测试的步骤是否按计划进行，以发现是否存在测试中的随意性，以及分析没有按照测试计划执行的原因。
- 通过评估测试的覆盖率情况、测试用例的通过率、测试的结果与测试的目标一致性，来评估测试的有效性，确定是否需要补充测试和复测或回归测试。
- 通过分析软件缺陷的严重性和缺陷的分布情况，向委托客户提供咨询意见或建议。

终止测试的标准

- 在测试计划中规定终止测试的标准
- 终止测试的常用标准
  - 所有严重的缺陷都已纠正，剩余的缺陷密度少于0.01%
  - 100%测试覆盖度
  - 缺陷数收敛了

测试报告

- 被测软件的名称和标识
- 测试环境
- 测试对象
- 测试起止日期
- 测试人员
- 测试过程
- 测试结果
- 缺陷清单
- 等

测试度量

- 如果不做正式的、定量的度量，就只能对测试过程有效性作出定性的描述
- 原始测试数据
  - 缺陷的数量
  - 软件的规模（KLOC、功能点）
  - 测试成本、测试工作量
  - 测试时间

主要测试度量指标

- 测试中发现的全部缺陷数
- 客户发现的全部缺陷数
- 缺陷检测有效性
  - 测试中发现的全部缺陷数/（测试中发现的全部缺陷数+客户发现的全部缺陷数）
- 缺陷排除有效性
  - 测试中改正的全部缺陷数/测试中发现的全部缺陷数
- 测试用例设计效率
  - 测试中发现的全部缺陷数/运行的测试用例数

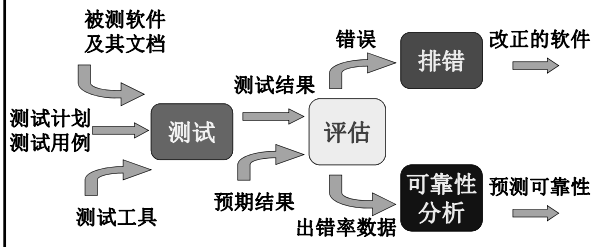
主要测试度量指标（续）

- 缺陷密度 = 缺陷数/软件规模
- 覆盖率
  - 测试需求覆盖率
  - 测试执行覆盖率
  - 用例通过覆盖率
  - 代码覆盖率
  - 功能覆盖率
- 缺陷分布
  - 按测试时间（趋势）
  - 按测试一定周期
  - 按软件开发组
  - 按缺陷严重性
  - 按缺陷状态

测试的文档

- 测试计划 – 在测试计划阶段
- 测试用例文档 – 在测试设计阶段
- 缺陷报告、测试记录 – 在测试执行阶段
- 测试报告 – 在测试完成后

测试的数据流



本节内容

- 测试概述
- 测试技术
- 测试策略
- 测试过程
- **自动化测试**

自动化测试

- 工具类型
  - 单元测试工具，即白盒测试工具
  - 性能测试工具
  - 功能测试工具，即回归测试工具
  - 缺陷跟踪工具
  - 测试数据生成工具
  - 测试管理工具
  - 等
- 工具产品
  - HP Mercury： WinRunner, LoadRunner …
  - IBM Rational
  - Compuware： QA Run, QA Load, QA Director …
  - Freeware：JUnit, selenium，Jmeter, Bugzilla …
  - ……

什么情况下适合用自动测试？

- 产品型项目
- 增量式开发、持续集成项目
- 能够自动编译、自动发布的系统
- 回归测试
- 多次重复的机械性动作，如性能测试
- 需要频繁运行的测试
- 将烦琐的任务转化为自动化测试

什么情况下不适合用自动测试

- 定制型项目（一次性的）
- 项目周期很短的项目
- 业务规则复杂的对象
- 美观、声音、易用性测试
- 测试很少运行
- 软件不稳定
- 涉及物理交互

自动化测试的误区

- 期望自动化测试能取代手工测试
  - 不能期望自动化测试来取代手工测试， 测试主要还是要靠人工的；
- 期望自动测试发现大量新的缺陷
  - 事实证明新缺陷越多，自动化测试失败的几率就越大。发现更多的新缺陷应该是手工测试的主要目的。测试专家James Bach总结得 85%的缺陷靠手工发现，而自动化测试只能发现15%的缺陷。自动化测试能够很好的发现老缺陷。

自动化测试的误区（续）

- 工具本身不具有想象力
  - 工具毕竟是工具，出现一些需要思考、体验、界面美观方面的测试，自动化测试工具无能为力
- 技术问题、组织问题、脚本维护
  - 自动化测试的推行，有很多阻力，比如组织是否重视， 是否成立这样的测试团队，是否有这样的技术水平，对于测试脚本的维护工作量也挺大的，是否值得维护等等问题都必须考虑。

Shanghai Jiao Tong University

软件工程

Module: 软件演化和维护

上海交通大学软件工程中心

思考

- 软件演化的规律是如何的？
- 软件维护就是改错吗？
- 软件维护和软件新产品开发哪个更具挑战？
- 软件维护的关键技术是什么？如何保证它的成功？
-

软件演化和维护

- 软件演化
- 软件维护

@第9章.教材

演化性是软件的基本属性

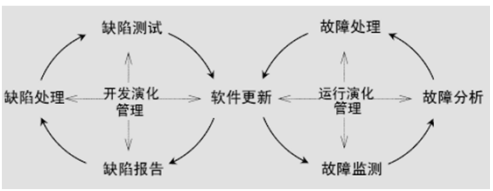
- “变化”是现实世界永恒的主题，只有“变化”才能发展。
- Lehman认为，现实世界的系统要么变得越来越没有价值，要么进行持续不断的演化变化以适应环境的变化。
- 软件是对现实世界中问题空间与解空间的具体描述，是客观事物的一种反映。现实世界是不断演化的，因此，演化性是软件的基本属性。

软件演化

- 什么是软件演化 (Software Evolution)
  - 是指对软件进行维护和更新的一种行为，它是软件生命周期中始终存在的变化活动。
- 按生命周期的不同阶段，软件演化可分为：
  - 开发演化
    - 创造一个新软件的过程，它强调要在一定的约束条件下从头开始实施，占软件演化的30%
  - 运行演化
    - 又称软件维护（Software Maintenance），是软件系统交付使用以后，为了改正错误或满足新的需要而修改软件的过程，它强调必须在现有系统的限定和约束条件下实施，占软件演化的70%。

软件开发演化与运行演化

- 在传统环境下，运行演化在开发演化后发生
- 在网络环境下，开发演化与运行演化已呈现出交织协同、并生共长的态势，软件运行状态改变的同时，软件版本也不断升级



Lehman的8条软件演化法则

- (1974)持续变更法则。软件必须持续改进，否则就会变得越来越不令人满意。
- (1974)复杂度递增法则。软件的复杂性随着演化不断增加，除非采取措施使系统保持或降低复杂性。
- (1974)自调节法则。软件的演化过程可以自动调节产品分布和过程测量，以接近正常状态。
- (1978)组织稳定性法则。在软件的生命周期中，组织的平均开发效率是稳定的。

- (1978)通晓法则。随着软件的演化，所有相关人员（如开发人员、销售人员和用户）都必须始终掌握软件的内容和行为，以便达到满意的演化效果。软件的过度增长会削弱对其内容和行为的通晓程度，因此软件应保持均衡的增长率。
- (1991)功能持续增加法则。在软件的生命周期中，软件功能必须持续增加，否则用户的满意度会降低。
- (1996)质量衰减法则。如果没有严格的维护和适应性调整使之适应运行环境的变化，软件的质量会逐渐衰减。
- (1996)反馈系统法则。软件演化过程是由多层次、多循环、多主体的反馈系统组成，而且要想在任何合理的基础上达到有意义的改进就必须这样进行处理。

软件演化和维护

- 软件演化
- 软件维护

软件维护 “冰山”

- 有关调查结果表明，软件维护是软件工程中最消耗资源的活动，很多软件公司中软件维护的成本已经达到了整个软件生存周期资源的40%到70%，甚至达到了90%。
- 软件系统趋于大型化和复杂化，大多数软件在设计时没有考虑到将来的修改问题，常常还伴有开发人员变动频繁、文档不够详细、维护周期过长等等问题，这些问题导致维护活动的时间与花费不断增加。

维护类型

	纠错	增强
积极的	预防性维护	完善性维护
被动的	纠错性维护	适应性维护

- 纠错性维护（Corrective maintenance）
  - 由于软件中的缺陷引起的修改
- 完善性维护（Perfective maintenance），
  - 根据用户在使用过程中提出的一些建设性意见而进行的维护活动
- 适应性维护（Adaptive maintenance）
  - 为适应环境的变化而修改软件的活动
- 预防性维护（Preventive maintenance）
  - 为了进一步改善软件系统的可维护性和可靠性，并为以后的改进奠定基础

维护的技术问题

- 有限理解（Limited understanding），对他人开发软件进行维护时，如何快速理解程序并找到需要修改或纠错的地方？
- 在软件维护过程中需要大量的回归测试，耗时耗力。
- 当软件非常关键以致不能停机时，如何进行在线维护而不影响软件的运行？
- 影响分析，如何对现有软件的变更所进行的全面分析？
- 如何在开发中促进和遵循软件的可维护性？
  - 易分析性（Analyzability）、易改变性（Changeability）、稳定性（Stability）和易测试性（Testability）

维护成本

- 维护的工作可划分成：
  - 生产性活动 如，分析评价、修改设计、编写程序代码等
  - 非生产性活动 如，程序代码功能理解、数据结构解释、接口特点和性能界限分析等
- 维护工作量的模型

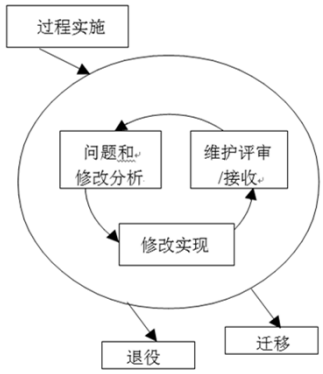
$$M = p + Ke^{c-d}$$

  - M：维护的总工作量；
  - P：生产性工作量；
  - K：经验常数；
  - e：软件的规模；
  - c：复杂程度；
  - d：维护人员对软件的熟悉程度

影响维护成本的因素

- 操作环境：硬件和软件
  - 软件的规模越大、复杂性越高、年龄越大，硬件的能力越低，软件维护的成本和工作量就越大。
- 组织环境：策略、竞争力、过程、产品和个人
  - 软件开发过程和维护过程越规范，采用的设计方法和编程语言模块化程度越高，工程师对软件的熟悉程度越高、能力越强，产品的可靠性和安全性要求越低，软件维护的成本和工作量就越小。

维护过程



维护活动

- 过程实施（Process Implementation）。建立维护过程期间应执行的计划和规程，包括维护计划、维护规程、问题解决规程、用户反馈计划、移交计划、配置管理计划。
- 问题和修改分析（Problem and Modification Analysis）。在修改软件前，要分析修改请求/问题报告，以确定其对组织、现行系统和接口系统的影响，提出可能的方案建议并形成文档，通过核准形成期望的解决方案。
- 修改实现（Modification Implementation）。根据计划和方案更新相应的需求、设计和代码，并进行测试等软件验证工作。

- 维护评审/接收（Maintenance Review/Acceptance）。对上述的维护进行评审，以确保对软件的修改是正确的，并且这些修改是使用正确的方法按批准的要求完成的。
- 迁移（Migration）。在软件的生存周期期间，如果需要将它迁移到一个新环境，则应制订迁移计划、通告用户迁移、提供迁移培训、把软件迁移至新环境、通告迁移完成情况、评估新环境的影响、并进行旧软件和数据的归档。在迁移实施时，旧环境和新环境可以并行进行工作，以便平稳迁移到新环境。
- 退役（Retirement）。软件一旦结束使用生存周期，必须退役。退役时，要制定退役计划、通知用户退役、提供退役培训、实施退役、通告退役完成情况、并进行旧软件和数据的归档。在制定退役计划时，要分析退役的成本和影响、决定是局部还是全部退役、是否用新软件来代替退役软件等。

软件维护技术

- 程序理解
- 逆向工程（Reverse Engineering）
- 再工程（Reengineering）

程序理解

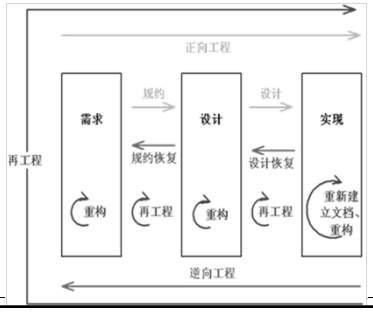
- 软件维护的总工作量大约一半被用在理解程序上。
- 程序理解通过提取并分析程序中各种实体之间的关系，形成系统的不同形式和层次的抽象表示，完成程序设计领域到应用领域的映射。
- 程序员在理解程序的过程中，经常通过反复三个活动——阅读关于程序的文档，阅读源代码，运行程序来捕捉信息。
- 程序理解工具
  - 基于程序结构的可视化工具，通过分析程序的结构，抽取其中各种实体，使用图形表示这些实体和它们之间的关系，可以直观地为维护者提供不同抽象层次上的信息。
  - 帮助维护者导航浏览源代码，为浏览工作提供着眼点，缩小需要浏览的代码范围。

逆向工程

- 逆向工程是分析软件，识别出软件的组成成份及其相互的关系，以更高的抽象层次来刻画软件的过程，它并不改变软件本身，也不产生新的软件。
- 逆向工程主要分为以下几类:
  - 重新文档化（redocumentation）：分析软件，改进或提供软件新的文档。
  - 设计恢复（design recovery）：从代码中抽象出设计信息；
  - 规约恢复（specification recovery）：分析软件，导出需求规约信息；
  - 重构（refactoring, restructuring）：在同一抽象级别上转换软件描述形式，而不改变原有软件的功能；
  - 数据逆向工程（data reverse engineering）：从数据库物理模式中获取逻辑模式，如实体关系图。

再工程

- 再工程是在逆向工程所获信息的基础上修改或重构已有的软件，产生一个新版本的过程，它将逆向工程、重构和正向工程组合起来，将现存系统重新构造为新的形式。



讨论

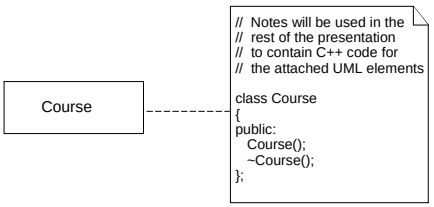
- 一个大型大学有一个大型计算机系统，用于存储和管理所有学生和教职工的信息。该系统：已经使用了25年，它采用Cobol结构化程序设计技术开发，并与关系数据库通信；它运行在一台IBM主机上；有50多万行代码。该系统已经进行过多次修改，既有经过策划的修改，也有快速修改，现在维护的成本过高。认识到有这些问题，大学希望利用面向对象的开发优势，但是不幸的是，维护这个系统的90%以上的员工都是新人，并不熟悉系统的实现。
- 如何办？

Center for Software Engineering
Shanghai Jiao Tong University

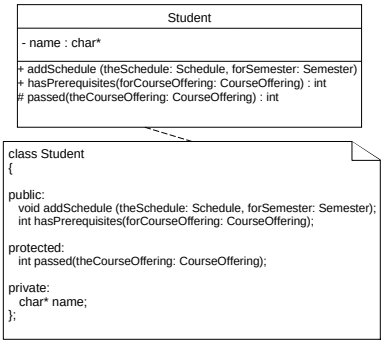
软件工程
Appendix: UML to C++ Mapping

上海交通大学软件工程中心 http://cse.sjtu.edu.cn

Mapping Representation: Notes



Visibility for Attributes and Operations



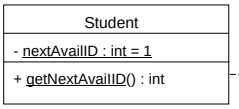

```
class Student
{

public:
   void addSchedule (theSchedule: Schedule, forSemester: Semester);
   int hasPrerequisites(forCourseOffering: CourseOffering);

protected:
   int passed(theCourseOffering: CourseOffering);

private:
   char* name;
};
```


Class Scope Attributes and Operations



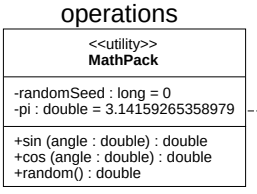

```
class Student
{

public:
   static int getNextAvailID();

private:
   static int nextAvailID = 1;
};
```


Utility Class

- A grouping of “free” attributes and operations

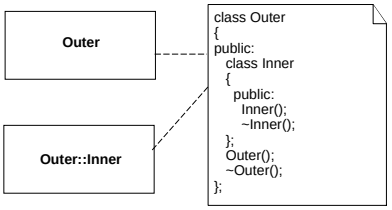



```
class MathPack
{
public:
    static double sin(double angle);
    static double cos(double angle);
    static double random();
private:
    static long randomSeed = 0;
    static double pi = 3.14159265358979;
};
```

```
void main(void)
. . .
     myCos = MathPack::cos(90.0);
. . .
```

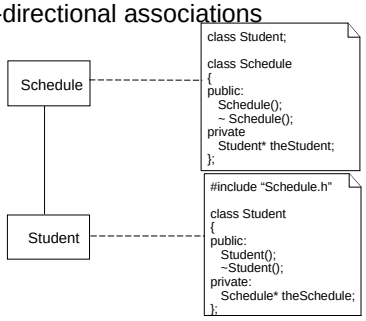

Nested Class

- Hide a class that is relevant only for implementation



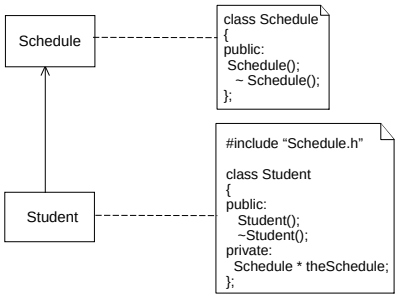
Associations

- Bi-directional associations

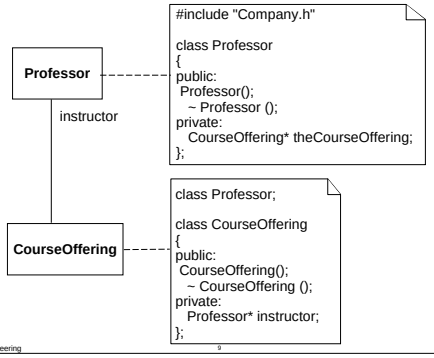


Association Navigability

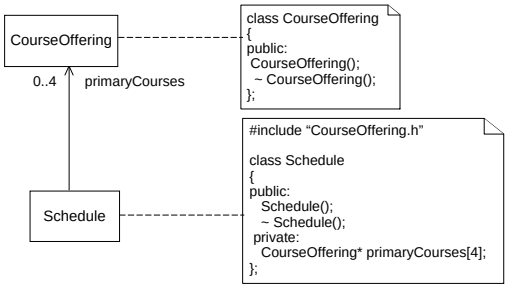
- Uni-directional associations



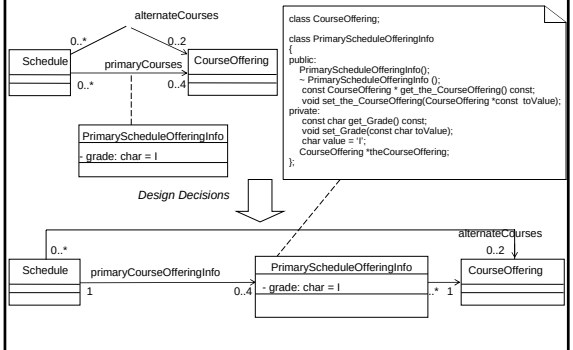
Association Roles



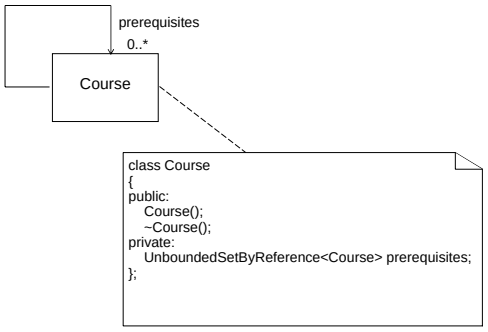
Association Multiplicity



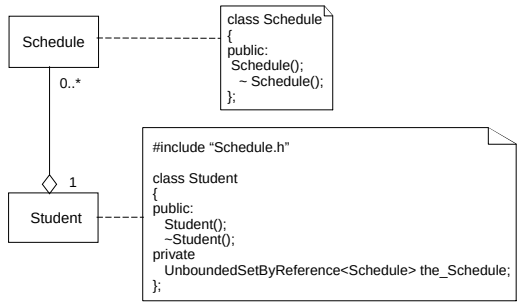
Association Class



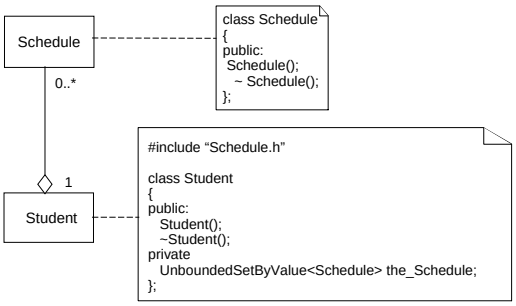
Reflexive Associations



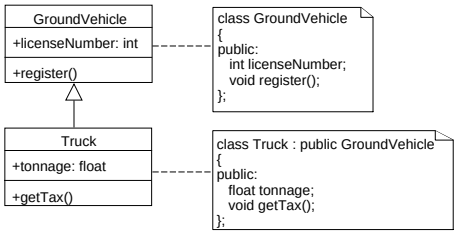
Aggregation



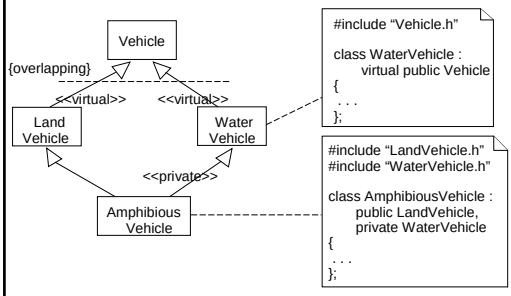
Composition



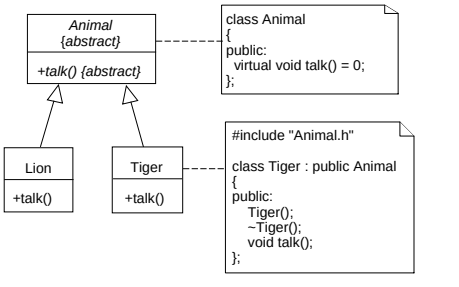
Generalization



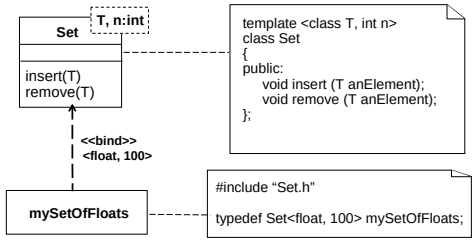
Multiple Inheritance



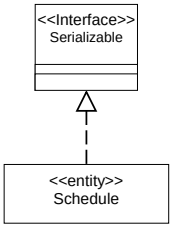
Abstract Class



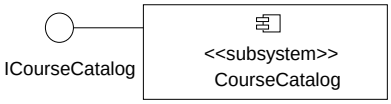
Parameterized Class



Interfaces and Realizes Relationships



Subsystems

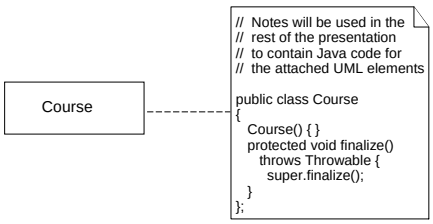


Center for Software Engineering
Shanghai Jiao Tong University

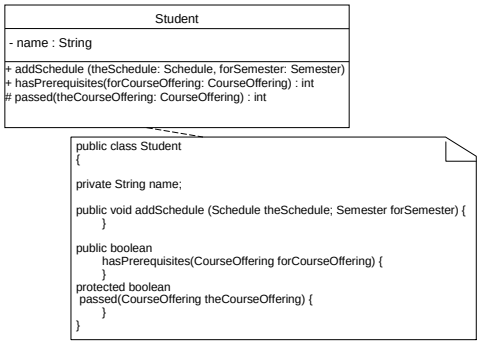
软件工程
Appendix: UML to Java Mapping

上海交通大学软件工程中心 http://cse.sjtu.edu.cn

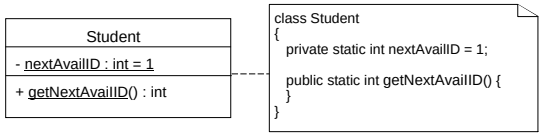
Mapping Representation: Notes



Visibility for Attributes and Operations

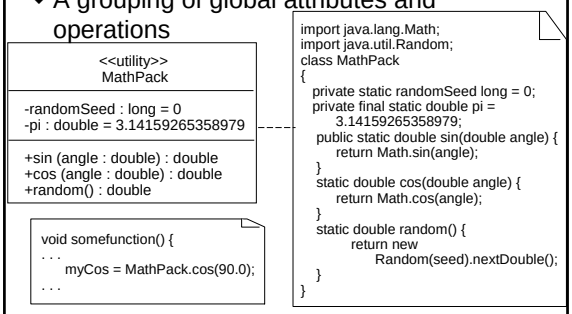


Class Scope Attributes and Operations



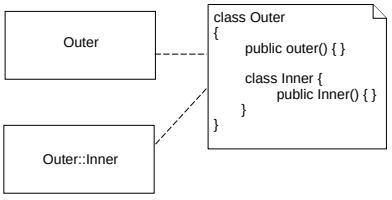
Utility Class

- A grouping of global attributes and operations



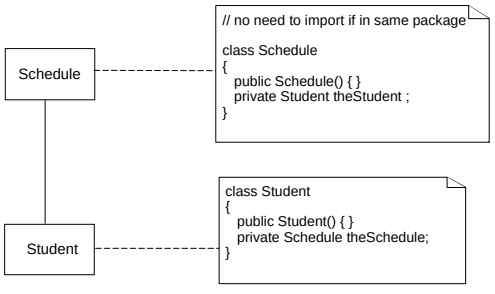
Nested Class

- Hide a class that is relevant only for implementation



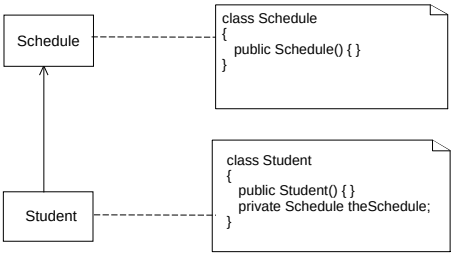
Associations

- Bi-directional associations

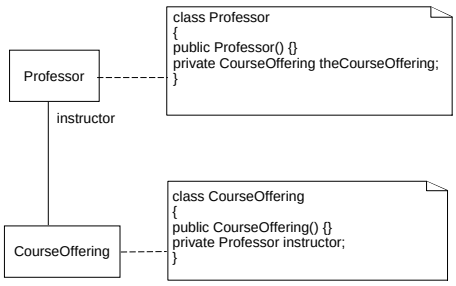


Association Navigability

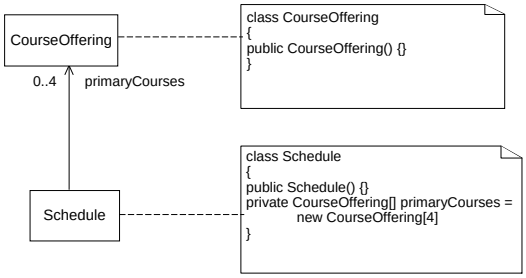
- Uni-directional associations



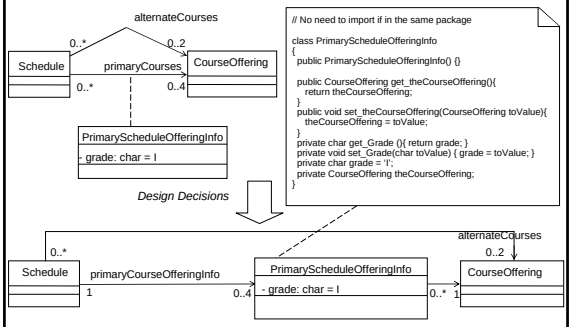
Association Roles



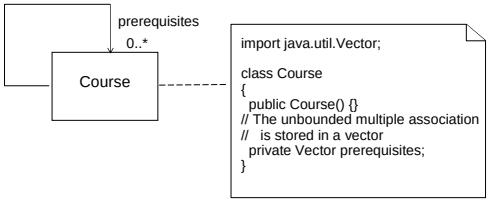
Association Multiplicity



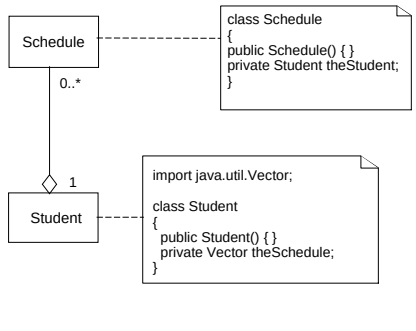
Association Class



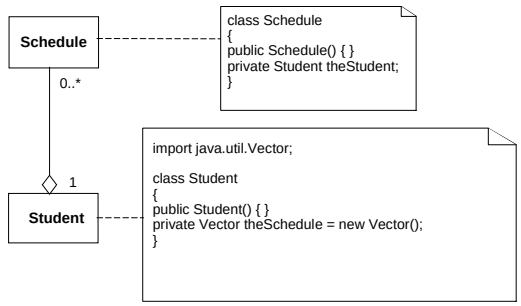
Reflexive Associations



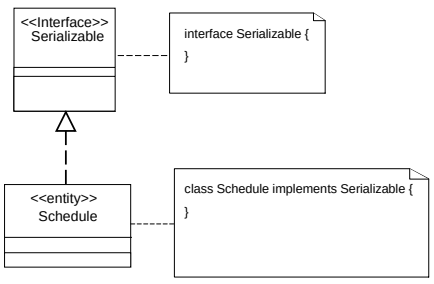
Aggregation



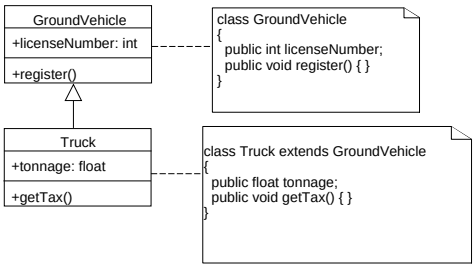
Composition



Interfaces and Realizes Relationships

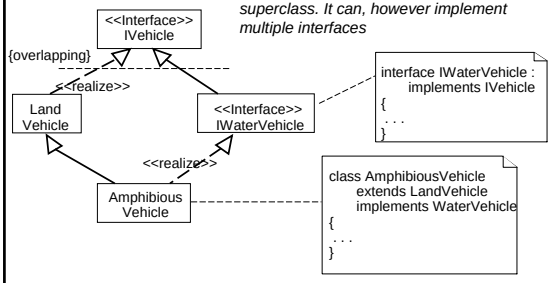


Generalization

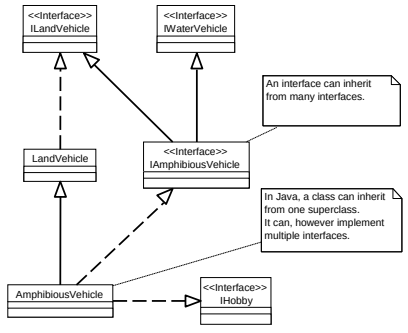


Multiple Inheritance

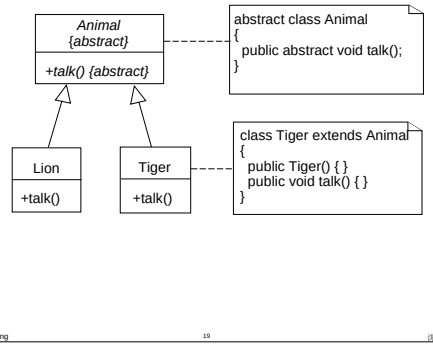
In Java, a class can only inherit from one superclass. It can, however implement multiple interfaces



Multiple Inheritance (continued)

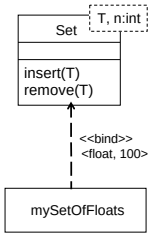


Abstract Class

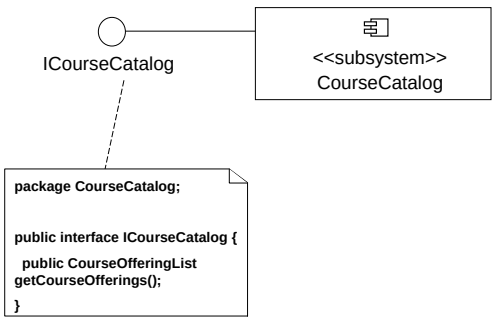


Parameterized Class

Java does not support parameterized classes



Subsystems



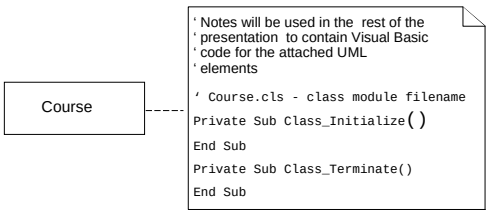
gb1 Should this note be deleted along with the related code (getCourseOfferings() : CourseOfferingList)?
gbastoky, 2004-6-18

Center for Software Engineering
Shanghai Jiao Tong University

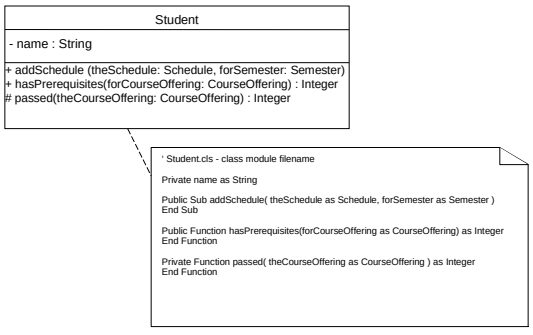
软件工程
Appendix: UML to VB Mapping

上海交通大学软件工程中心 http://cse.sjtu.edu.cn

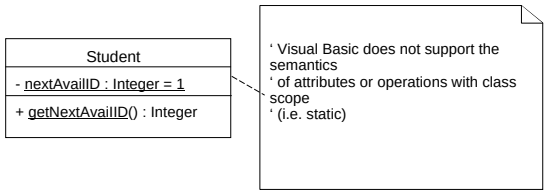
Mapping Representation: Notes



Visibility for Attributes and Operations

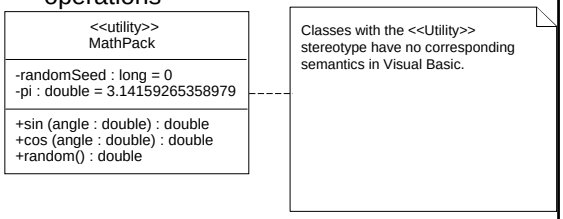


Class Scope Attributes and Operations



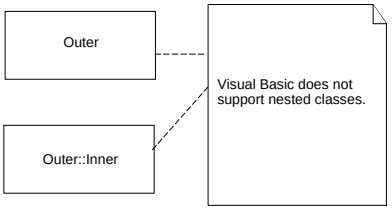
Utility Class

- A grouping of global attributes and operations



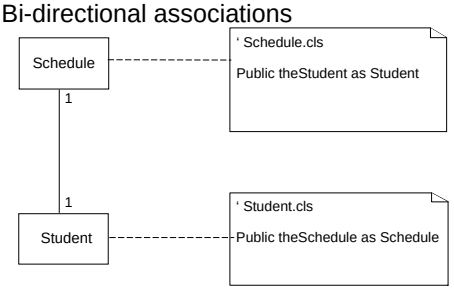
Nested Class

- Hide a class that is relevant only for implementation



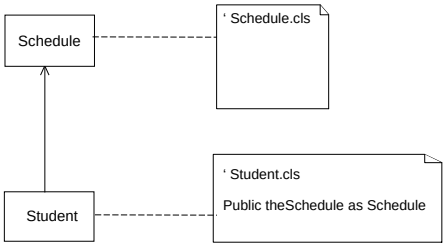
Associations

- Bi-directional associations

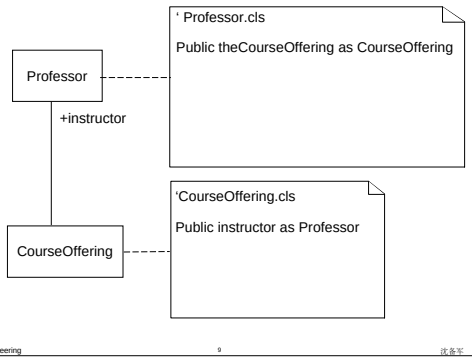


Association Navigability

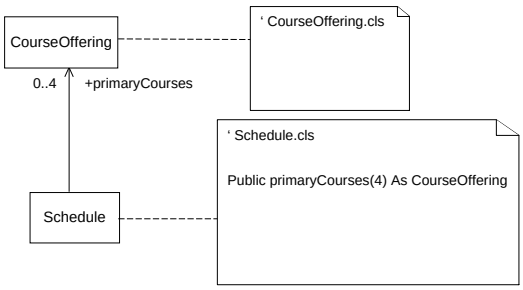
- Uni-directional associations



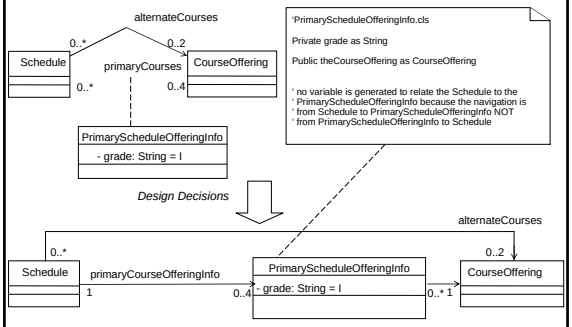
Association Roles



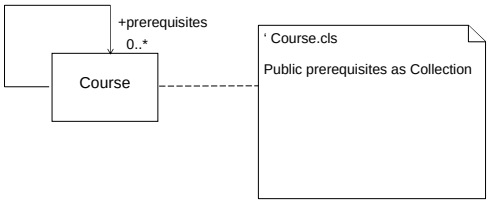
Association Multiplicity



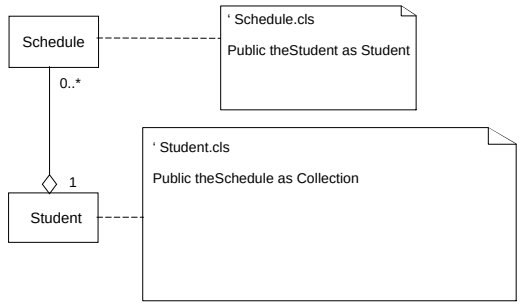
Association Class



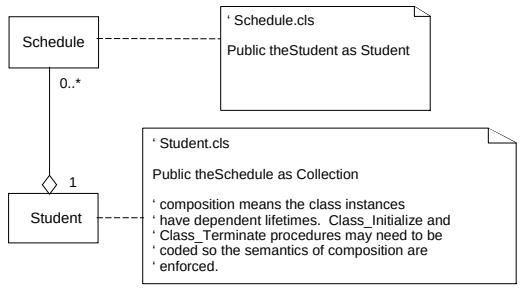
Reflexive Associations



Aggregation

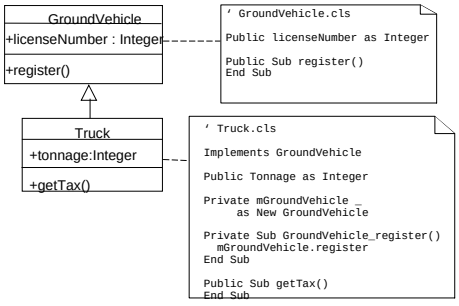


Composition

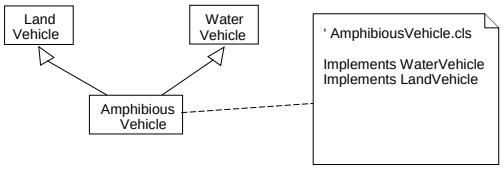


Generalization

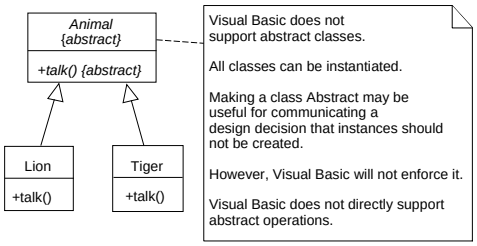
Visual Basic does NOT support generalization (like COM).
It simulates generalization using Delegation.



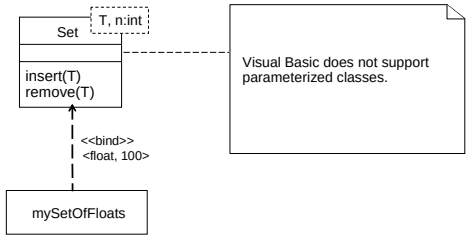
Multiple Inheritance



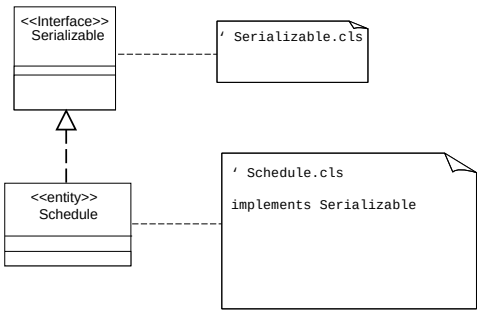
Abstract Class



Parameterized Class



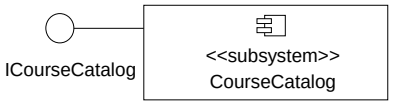
Interfaces and Realizes Relationships



Software Engineering 19 沈备军

Subsystems

Visual Basic does NOT directly support subsystems



Software Engineering 20 沈备军