

技术架构视图-构架物理设计

胡协刚

软件架构师 **UML/RUP**专家

内容提要

- 构架建模概貌
- 系统逻辑建模
- 实施模型
- 系统进程建模
- 系统部署建模

构架建模概貌

构架设计中的常见问题

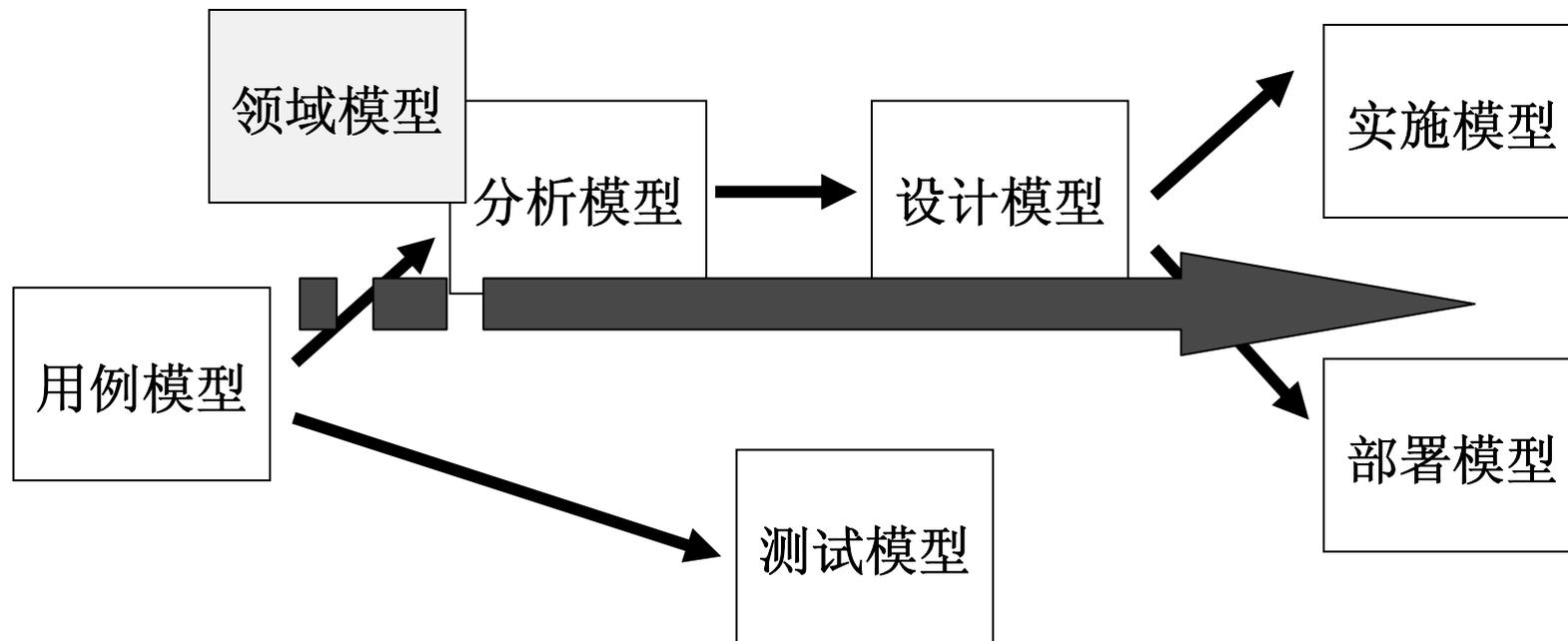
✓项目A现在遇到麻烦了；为了提高对请求的响应速度，系统必须采用多线程技术；问题是除了部分使用boost库的代码是多线程安全的外，其它代码没有作任何多线程的考虑；这是因为概要设计中一直都没有对系统进程结构的说明，大家处理进程方面的代码时都比较随意。

构架设计中的常见问题

✓项目终于要上线了，可是到现场安装系统时传来了坏消息；虽然已经知道客户的现场软硬件配置有所不同，但万万没想到其网络用防火墙隔离成内外两个网段，我们的系统必须部署到跨越两个网段的不同主机上，问题是系统各部分的通讯协议却不能穿越防火墙——“完了！软件构架文档中的部署图显然没有完全遵从客户现场的拓扑结构。”

构架建模统领整个开发过程

构架建模针对系统的轮廓和最重要方面进行表述，它定义了系统顶层的结构，为更具体的结构建模预设了范围；它通过定义构架机制，确定了系统的基本特征，从而为行为建模指明了方向。



UML中构架建模的位置

	对象系统	对象群体	对象个体
结构	组件图、部署图、包图、子系统图 (关注构架)	包图、类图、对象图 (关注对象间的关系)	类图、对象图 (关注对象内部结构)
行为	用例图、活动图 (关注需求)	序列图、协作图 (关注协作)	状态图、活动图 (关注个体生命周期)

UML构架建模的元素

- ✓UML语言支持构架建模的元素有：
 - 类
 - “具有行为”的包
 - “具有特定语义”的子系统
 - 接口
 - 包图Package
 - 构件图
 - 部署图
 - 类图

建模惯用法Modeling Convention

- 它们是什么？

- 使用什么图和模型元素
- 使用图和模型元素的规则
- 命名规范

- 示例

- 什么建模部件modeling constructs不应被使用
- 什么图必须要有
- 什么图将用来建模构架视图

示例：建模惯用法

- Use-Case用例视图

- 用例将用主动短语来命名，例如“Submit Grades”

- Logical逻辑视图

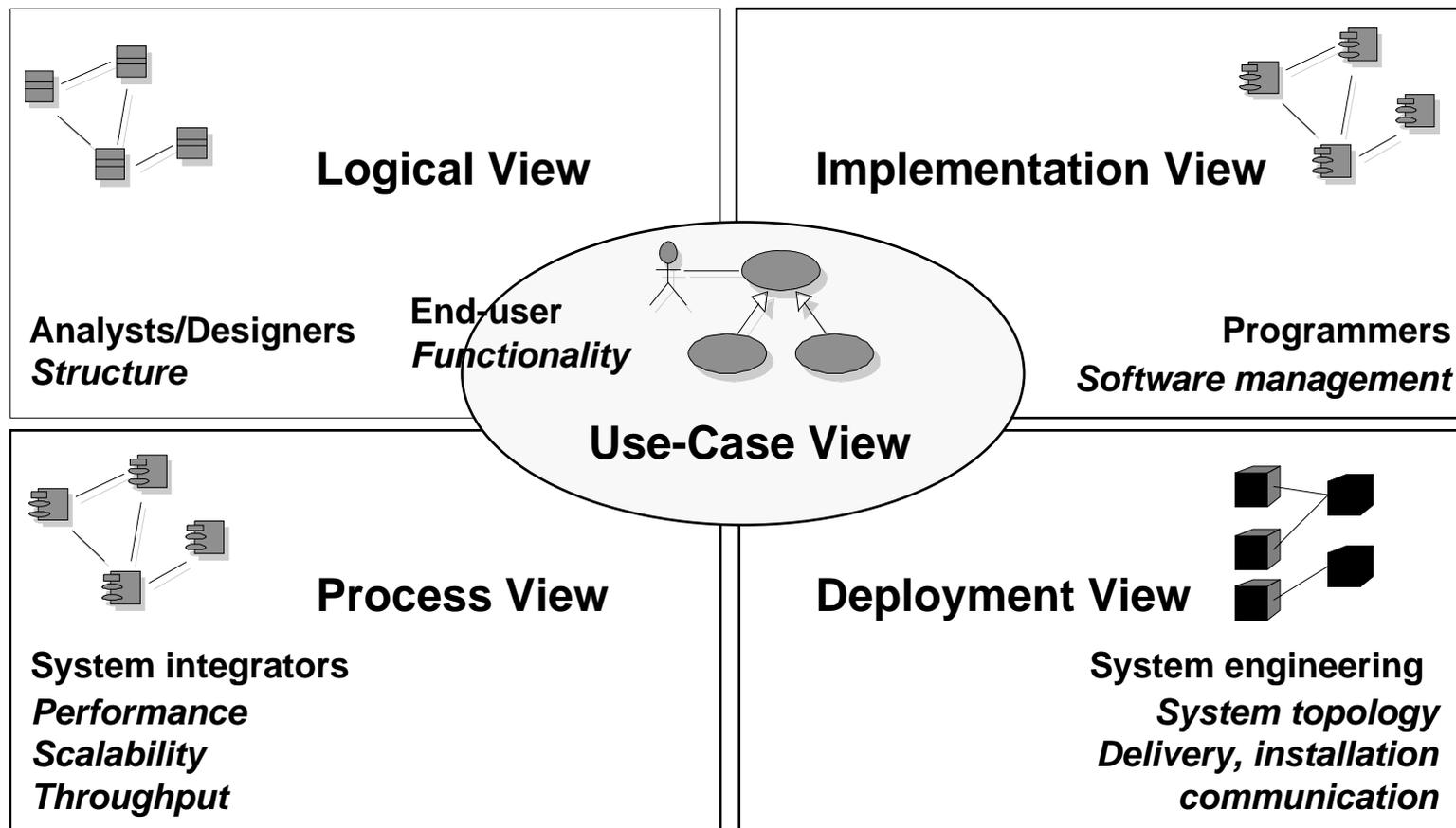
- 一个用例实现包将包括：

- 至少有一个实现追踪到每个用例

- 一个显示实现中的参与者及其关系的参与类视图 “View Of Participating Classes”

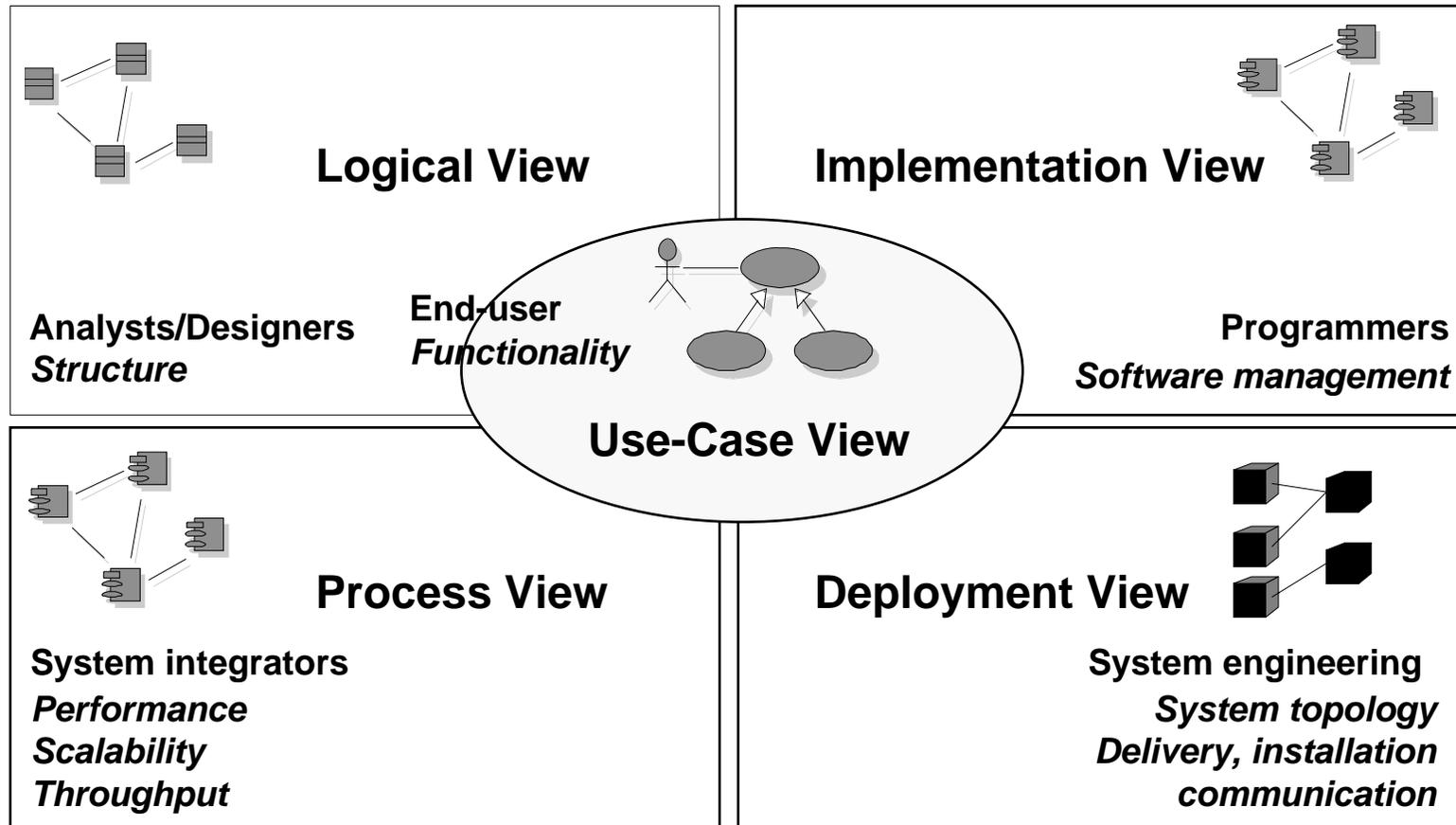
- 类应当使用与问题域尽可能匹配的名词来命名

软件构架：4+1视图模型



系统逻辑建模

软件构架：逻辑视图



The Logical View is an “architecturally significant” slice of the Design Model

表达系统结构

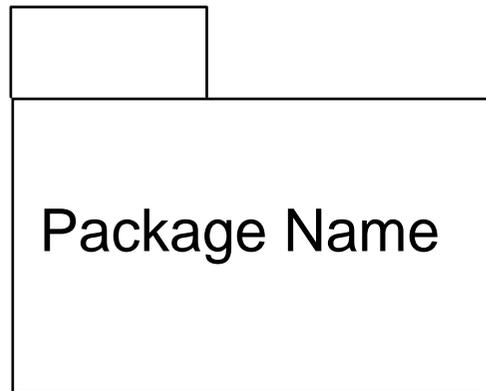
- ✓ 系统最终由若干、乃至成百上千的元素所构成，它们相互关联和依赖，这便是系统的结构；
- ✓ 然而这些元素的数量往往多到可能在开发中泛滥成灾的地步，我们不能直接以如此小的粒度来表达系统的结构，因此将模型元素组织成逻辑相关的分组变得非常关键；
- ✓ 系统结构的分解将经历系统-子系统/设计包-类的层次化精化过程。

设计模型中的包

管理设计元素的通用机制是设计包；另外一种具有更强语义的包，则成为子系统；

包Package

- 包是一种通用机制，用于将其它元素组织成分组
- 它是一种可以包含其它元素的模型元素

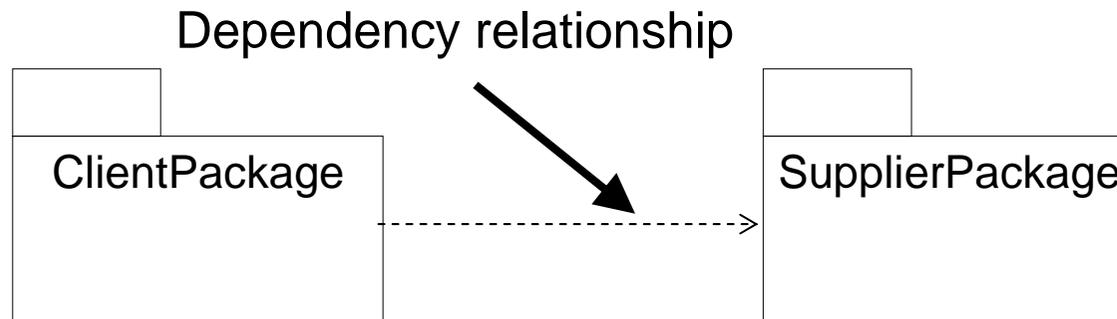


- 用途

- 组织开发中的模型
- 置于配置管理之下的一个配置单位

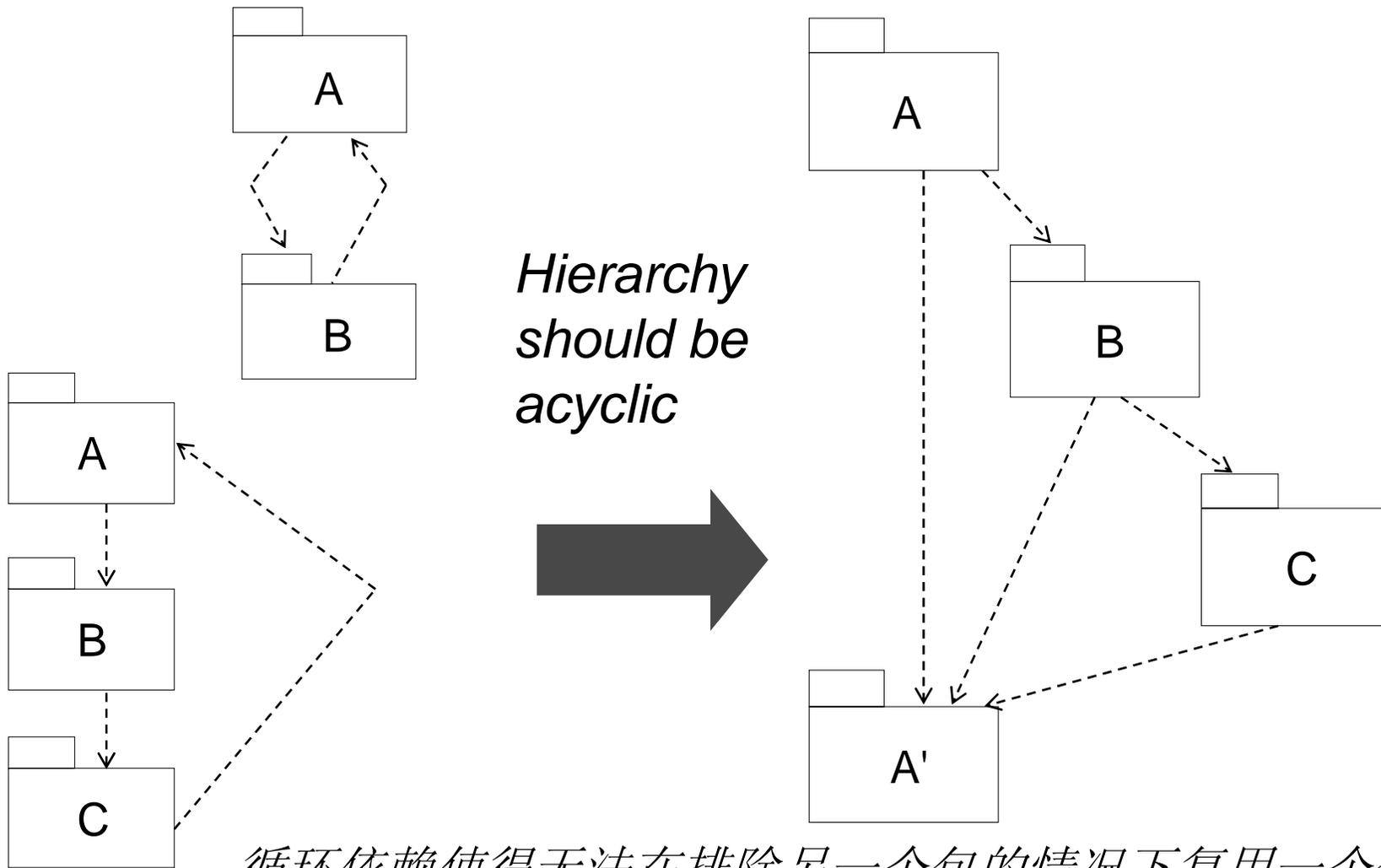
包之间的关系——依赖

- 包可以使用（引入和访问）依赖关系与其它包关联
 - 用于在一个包中引入由另一个包开放的元素
 - 允许一个包访问另一个包中的元素
 - 引入和访问关系是不可传递的



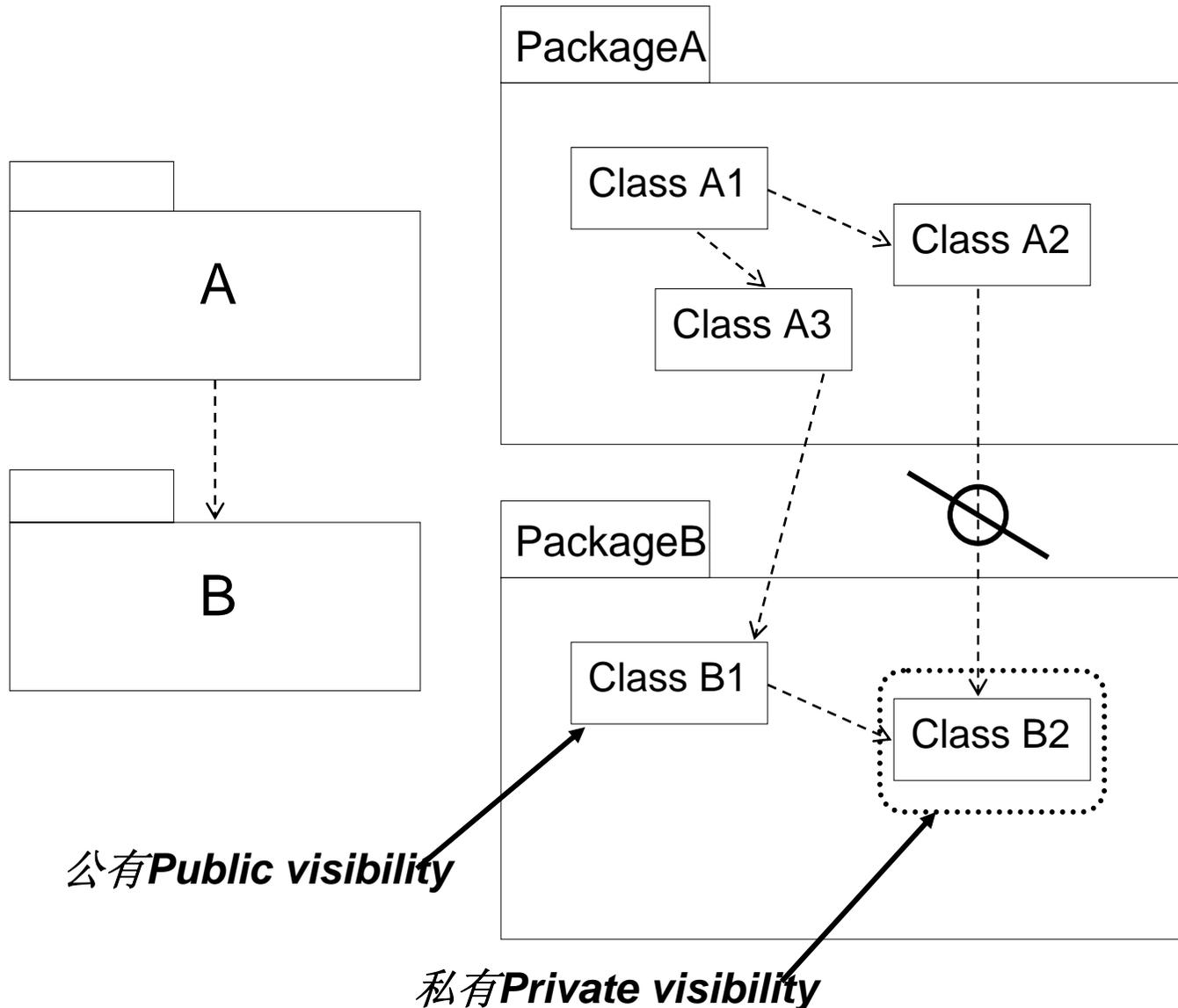
- 依赖的隐含意义
 - 对供应包的更改可能影响客户包
 - 客户包不能独立地被复用，因为它依赖于供应包

避免循环Circular依赖



循环依赖使得无法在排除另一个包的情况下复用一个包
Circular dependencies make it impossible to reuse one package without the other

包的可视性visibility

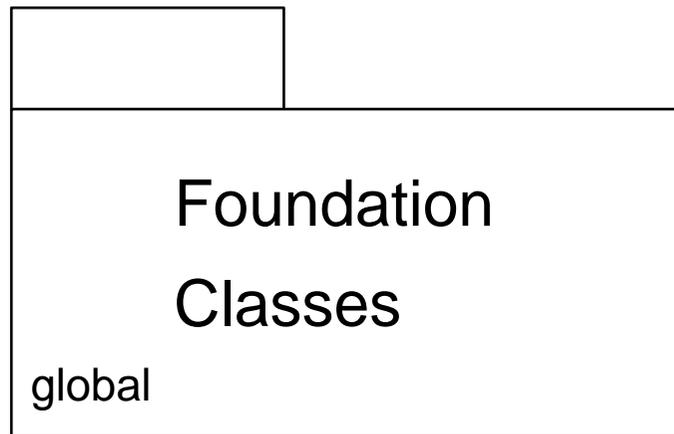


包的可见性规则

- 一个包中定义的元素在同一个包中是可见的
- 如果一个元素在一个包中是可见的，那么它对所有嵌套在这个包中的所有包都是可见的
- 如果一个包是另一个包的孩子，那么所有在父包中定义为公共的或受保护的可见性的元素对子包是可见的
- 如果一个包引入或访问另一个包，那么在被引入或访问的包中定义为公共可见的元素对引入或访问包都是可见的
- 访问或引入关系不可传递

全局Global包

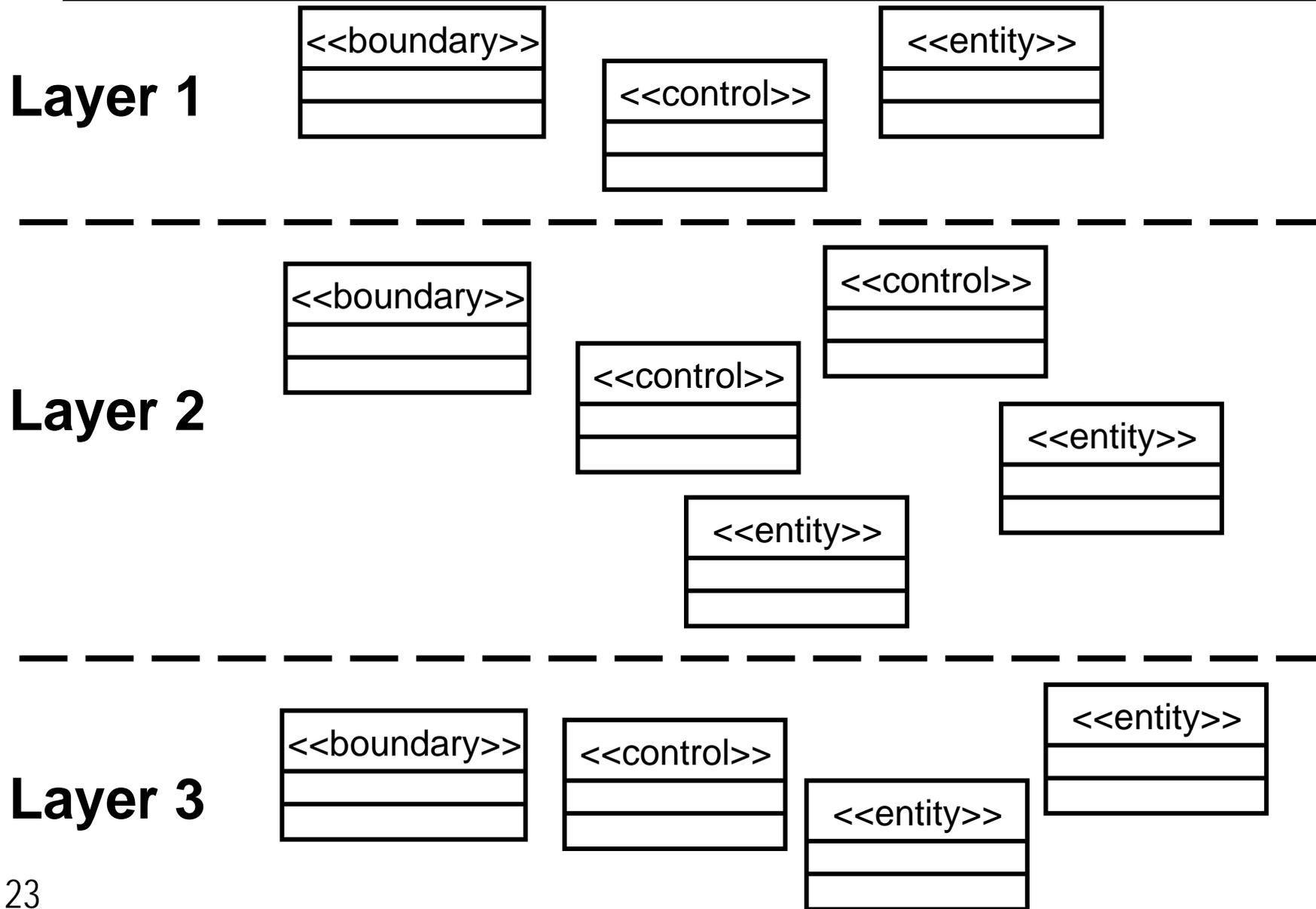
- 某种被其它所有包使用的包
- 这些包被标注为全局的



设计模型组织结构Organization

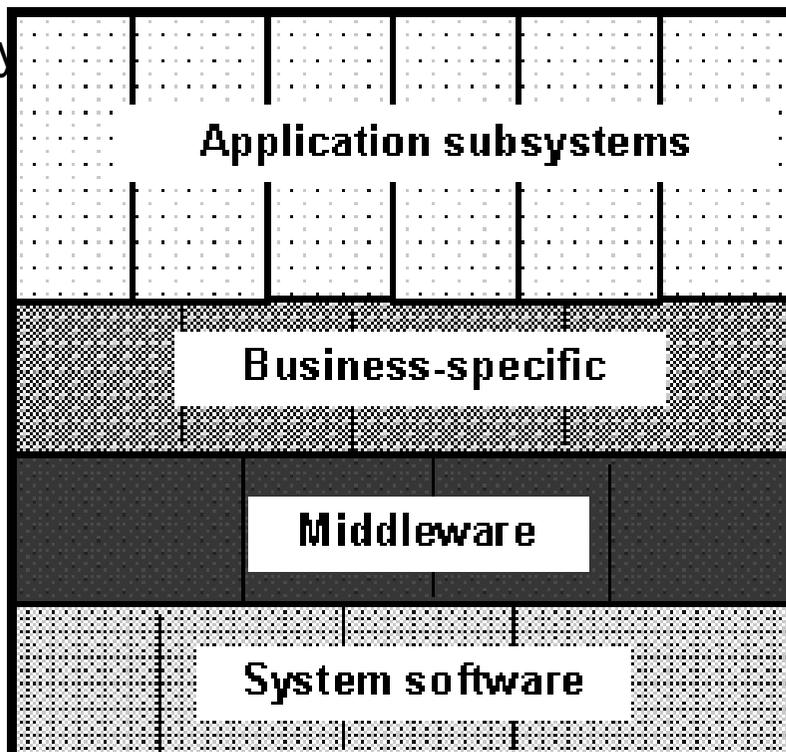
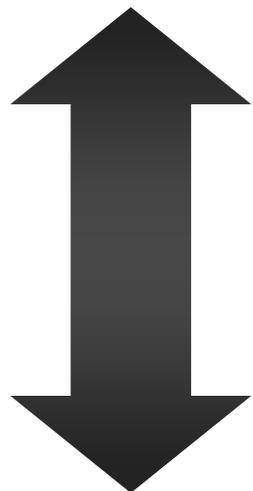
✓设计模型组织结构的最终目标是支持个人或团队进行独立的开发

示例：设计元素和构架



典型的层次化途径

特殊的功能
Specific
functionality



通用的功能
General
functionality

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

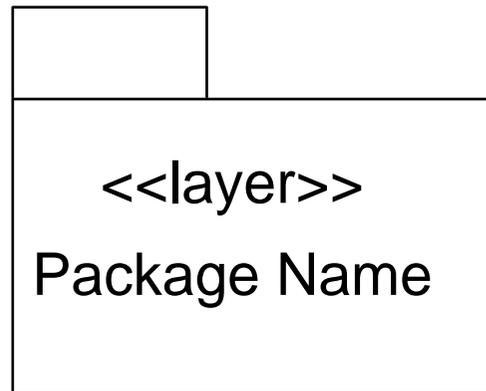
Business specific - contains a number of reusable subsystems specific to the type of business.

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.

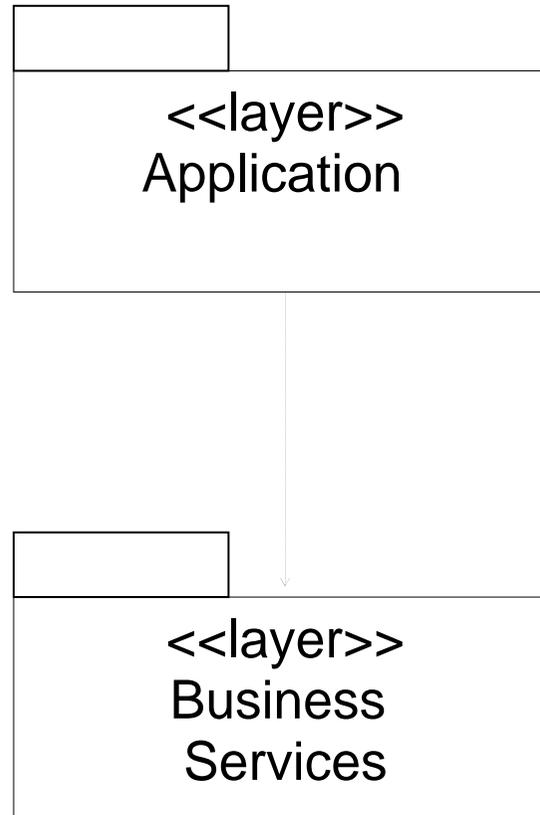
构架层建模

- 构架层可以用带版型的包来表达
- <<layer>> stereotype

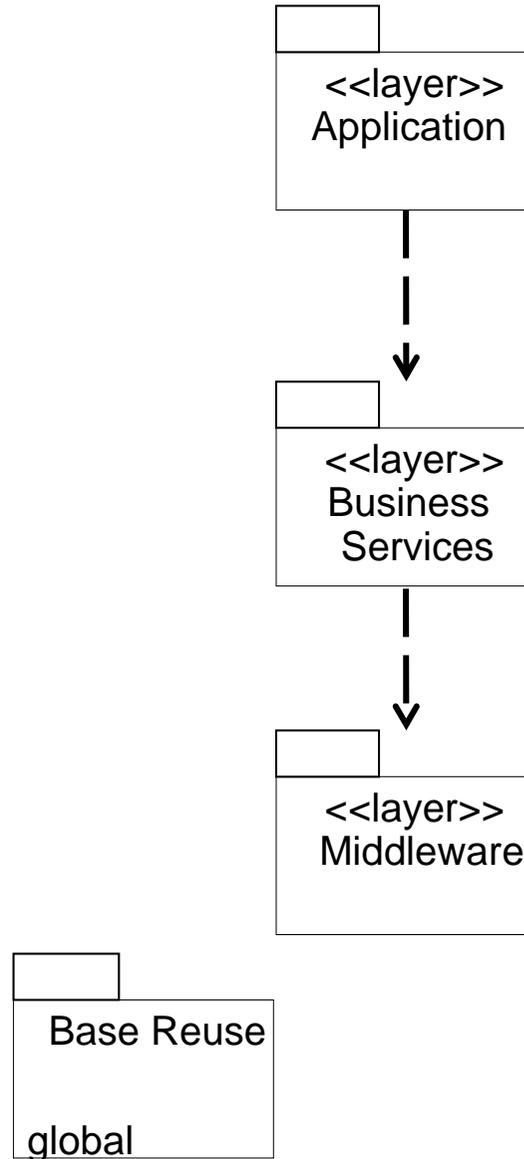


示例：模型高层组织

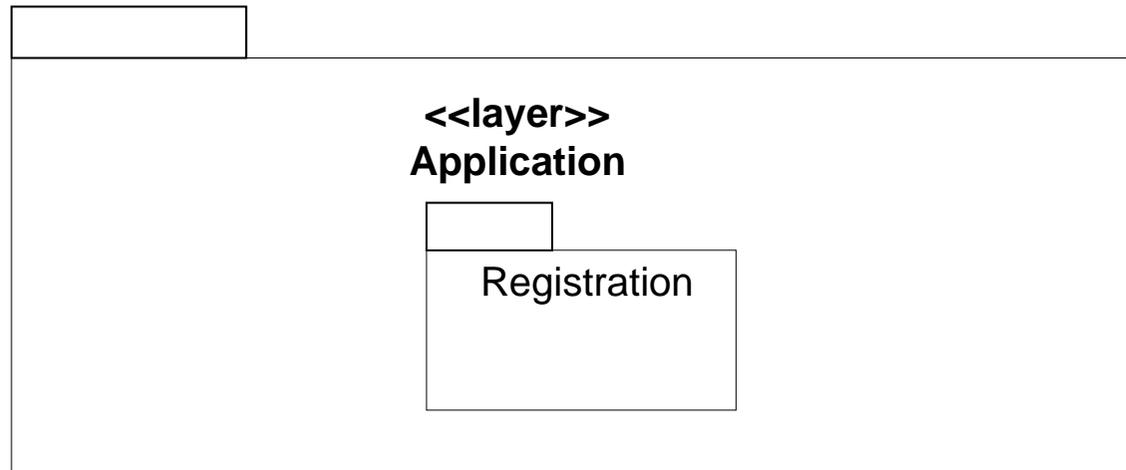
layers for the Course
Registration System



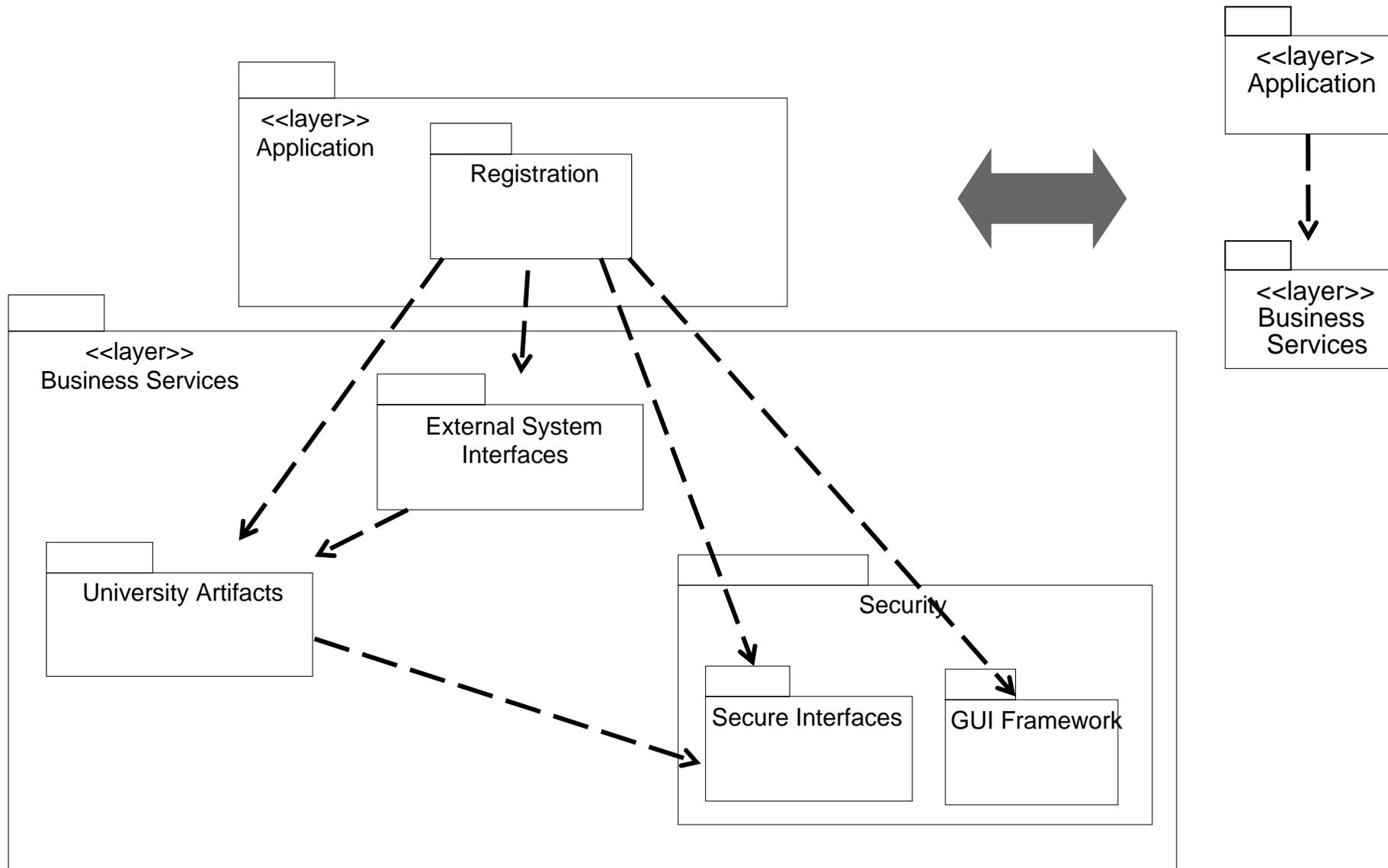
Example: Architectural Layers



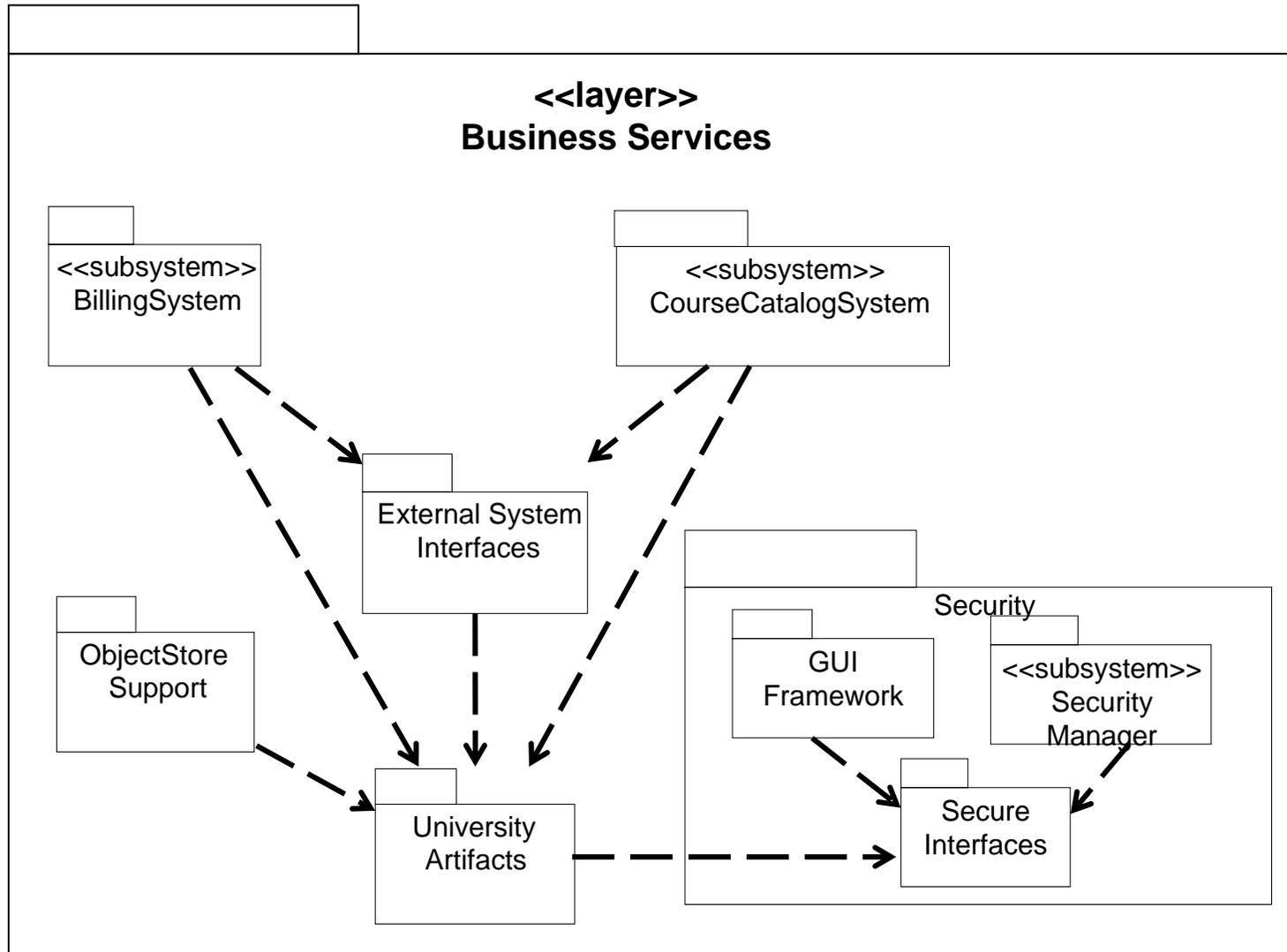
Example: Application Layer



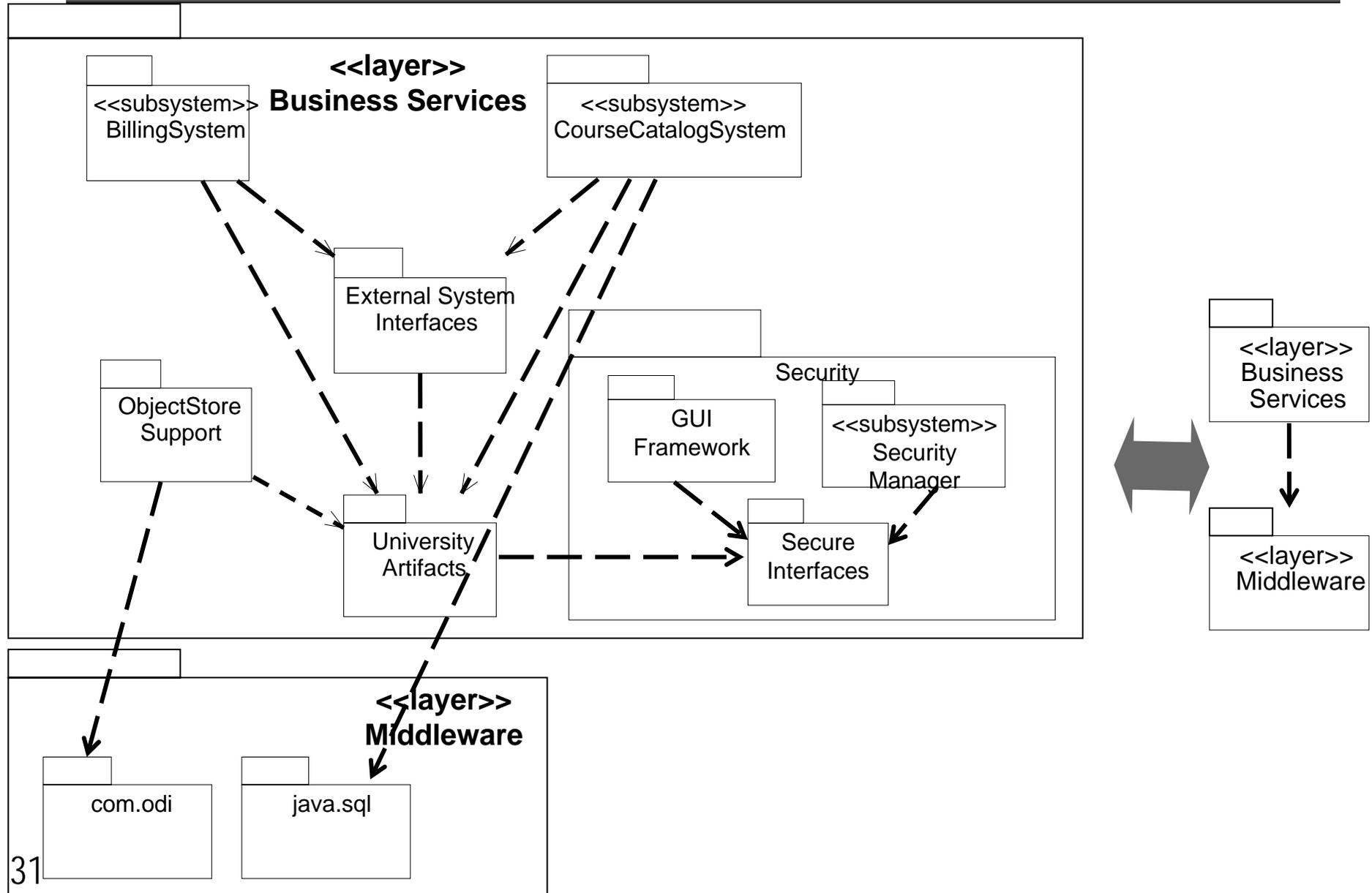
Example: Application Layer Context



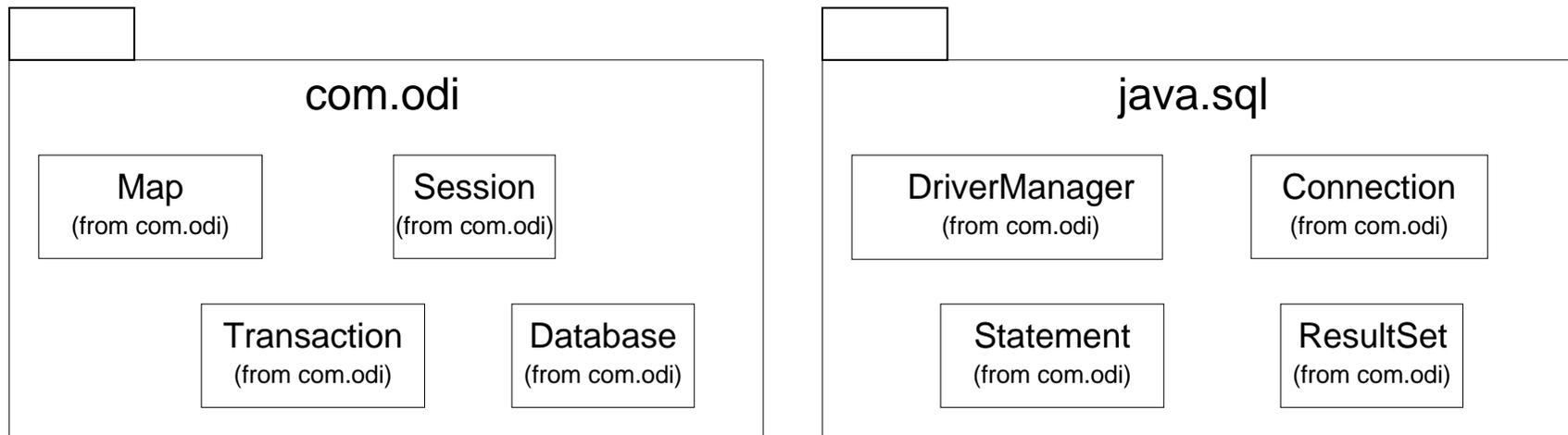
Example: Business Services Layer



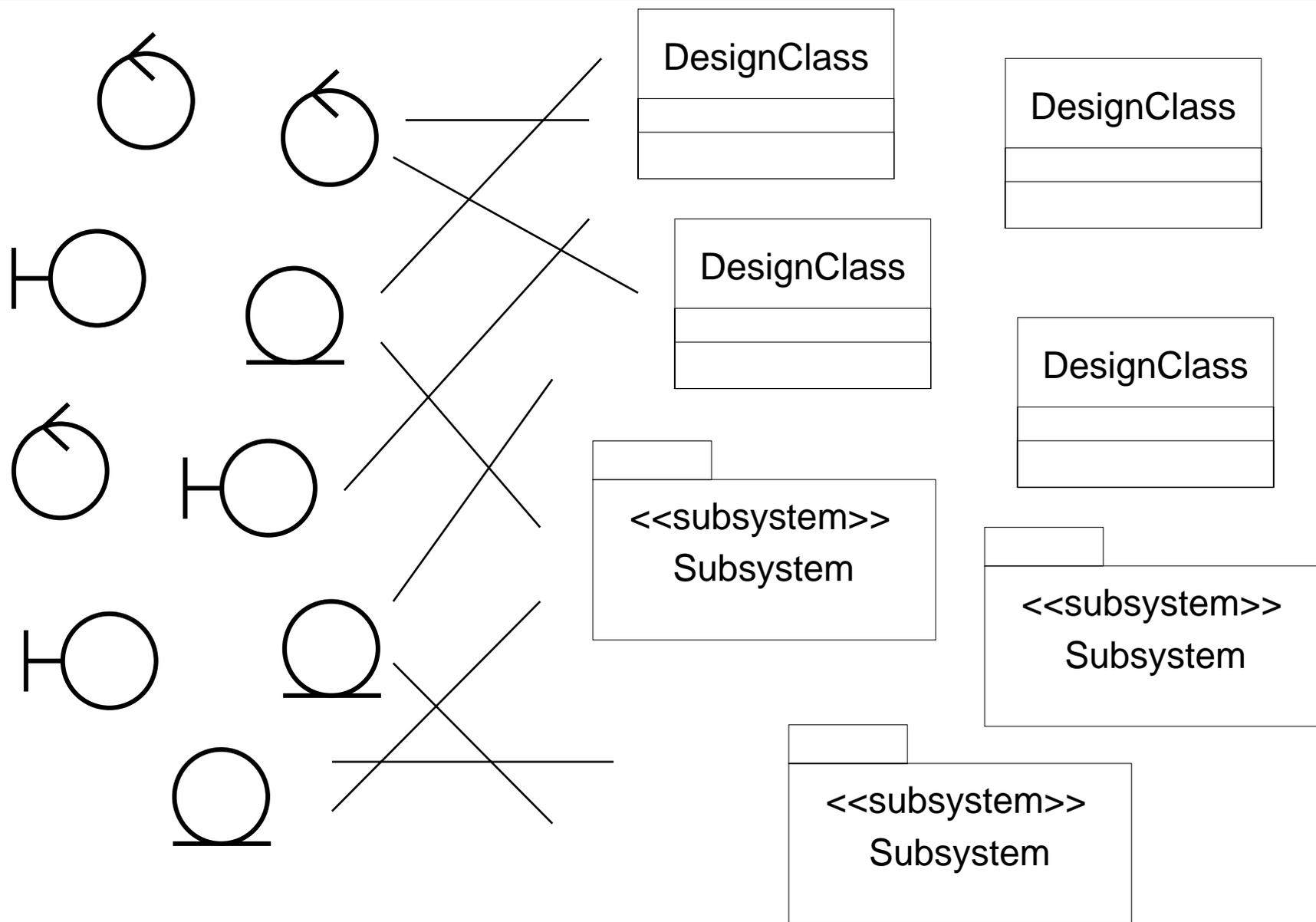
Example: Business Services Layer Context



Example: Middleware Layer



类与子系统设计



使用子系统

- 子系统可以用来将系统划分成相互独立的部件，它们将可以：

- 独立地被定制 ordered 、配置 configured、或交付 delivered

- 在接口保持不变的情况下，被独立开发

- 跨越一系列分布式计算节点来被独立部署

- 被独立地变更而不打破系统其它部分

- 子系统也可以用来：

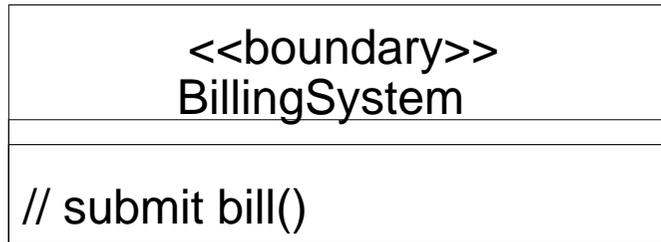
- 将系统中访问关键资源而需要有安全限制的部分分割出来成为独立控制的单元

- 在设计中代表已有产品或外部系统（例如构件）

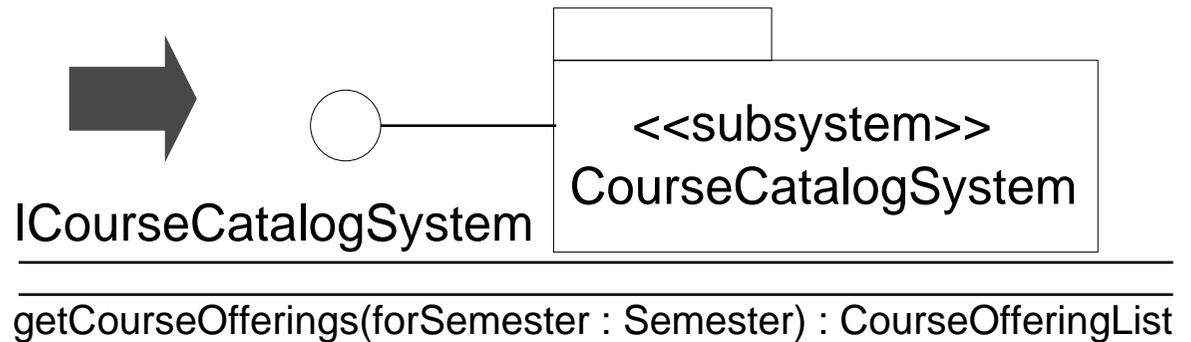
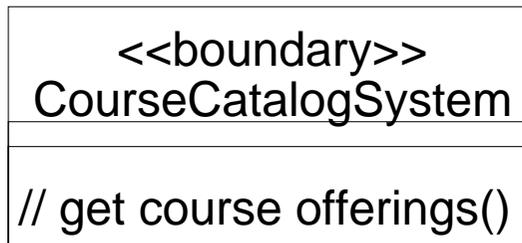
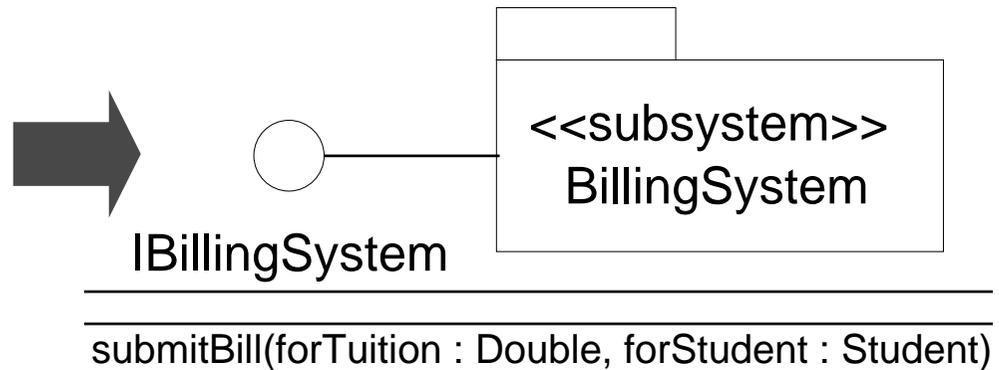
Subsystems raise the level of abstraction

实例：子系统设计

Analysis



Design



All other analysis classes map directly to design classes

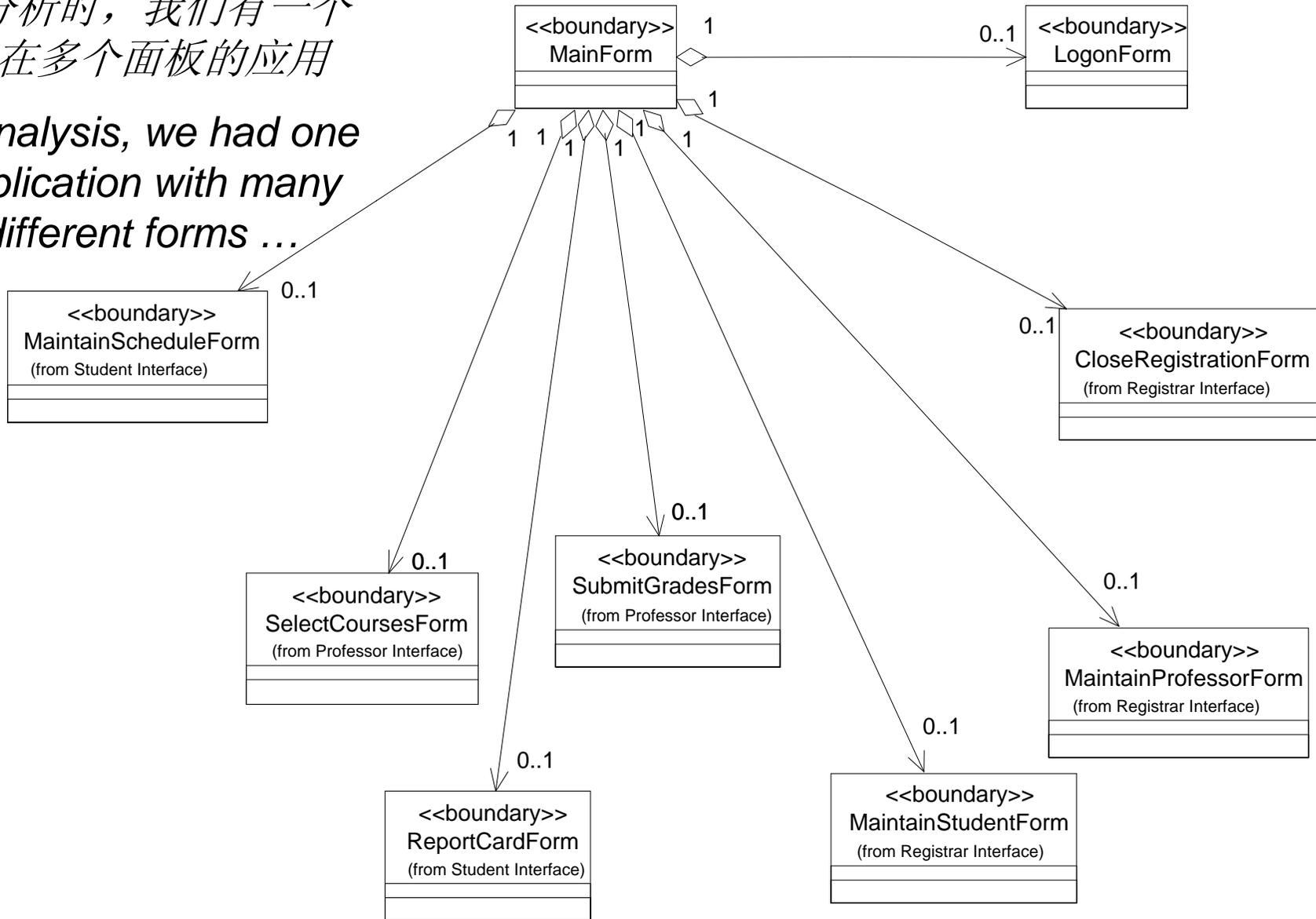
构架敏感的类型Architectural Class

- 系统中总是存在一些类，它们对构架具有决定性的影响
- 构架敏感类的职责
 - 构架机制的实现
 - 代表关键抽象——实体类实现

实例：类设计

在分析时，我们有一个
存在多个面板的应用

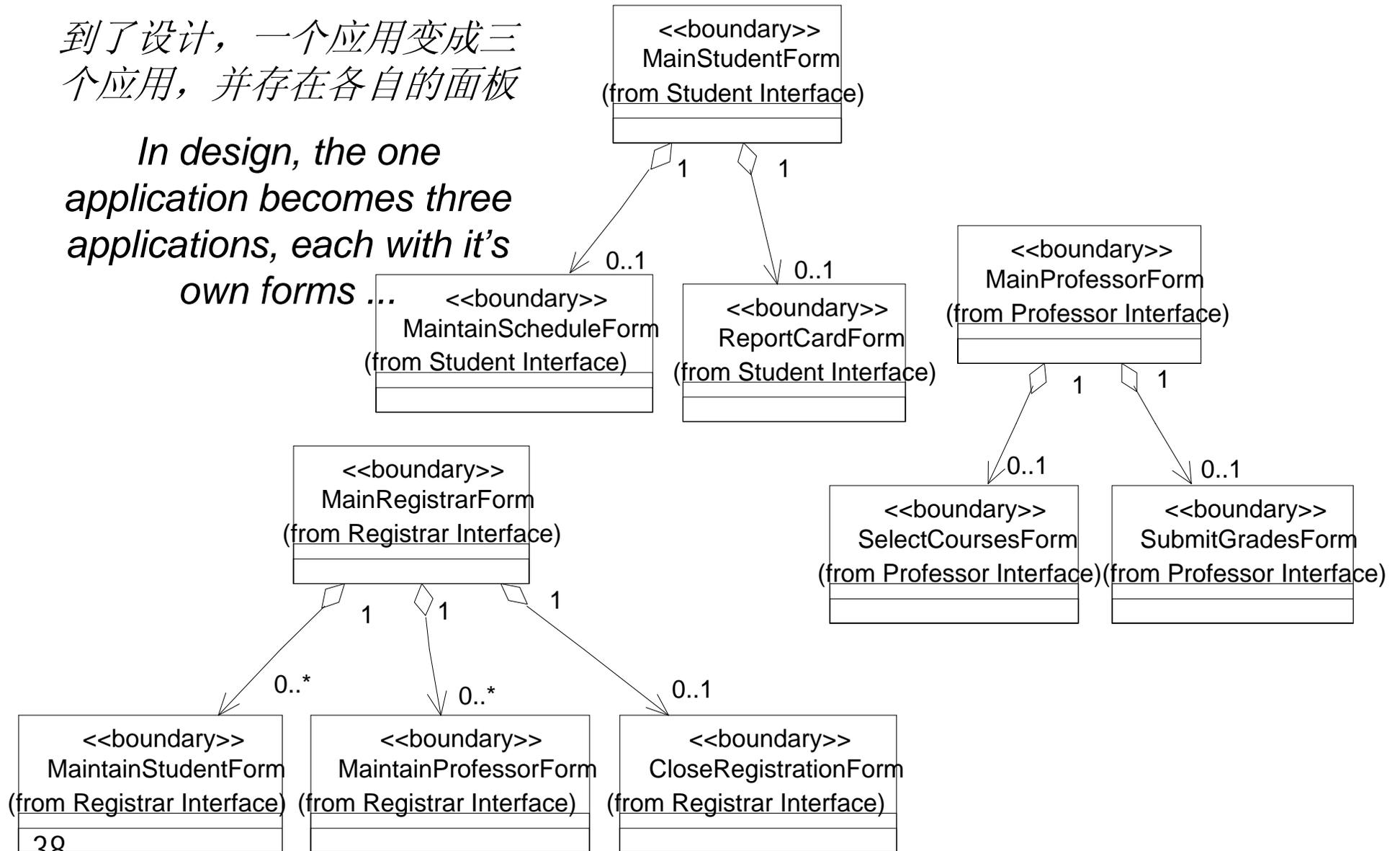
*In analysis, we had one
application with many
different forms ...*



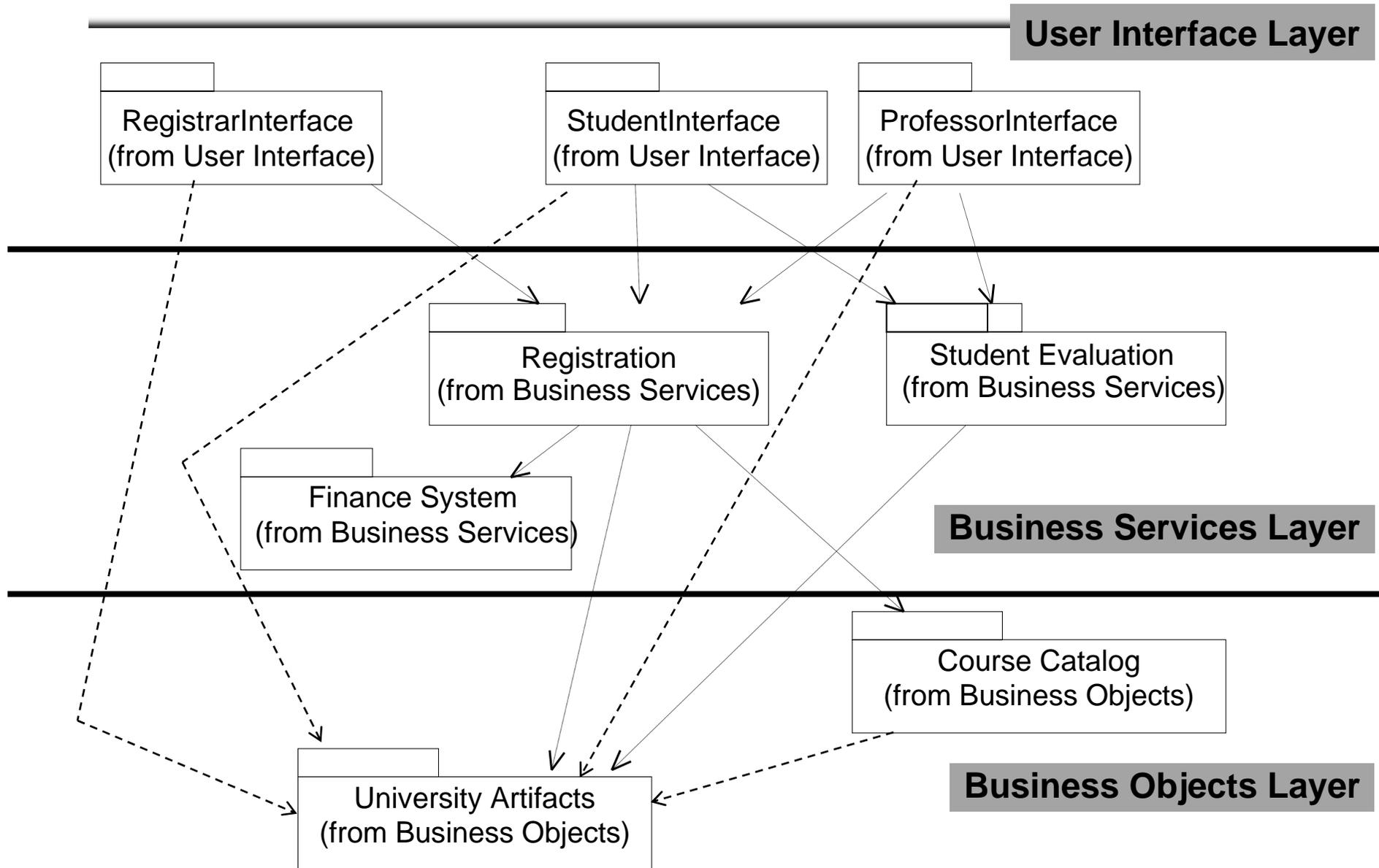
实例：类设计cont.

到了设计，一个应用变成三个应用，并存在各自的面板

In design, the one application becomes three applications, each with it's own forms ...

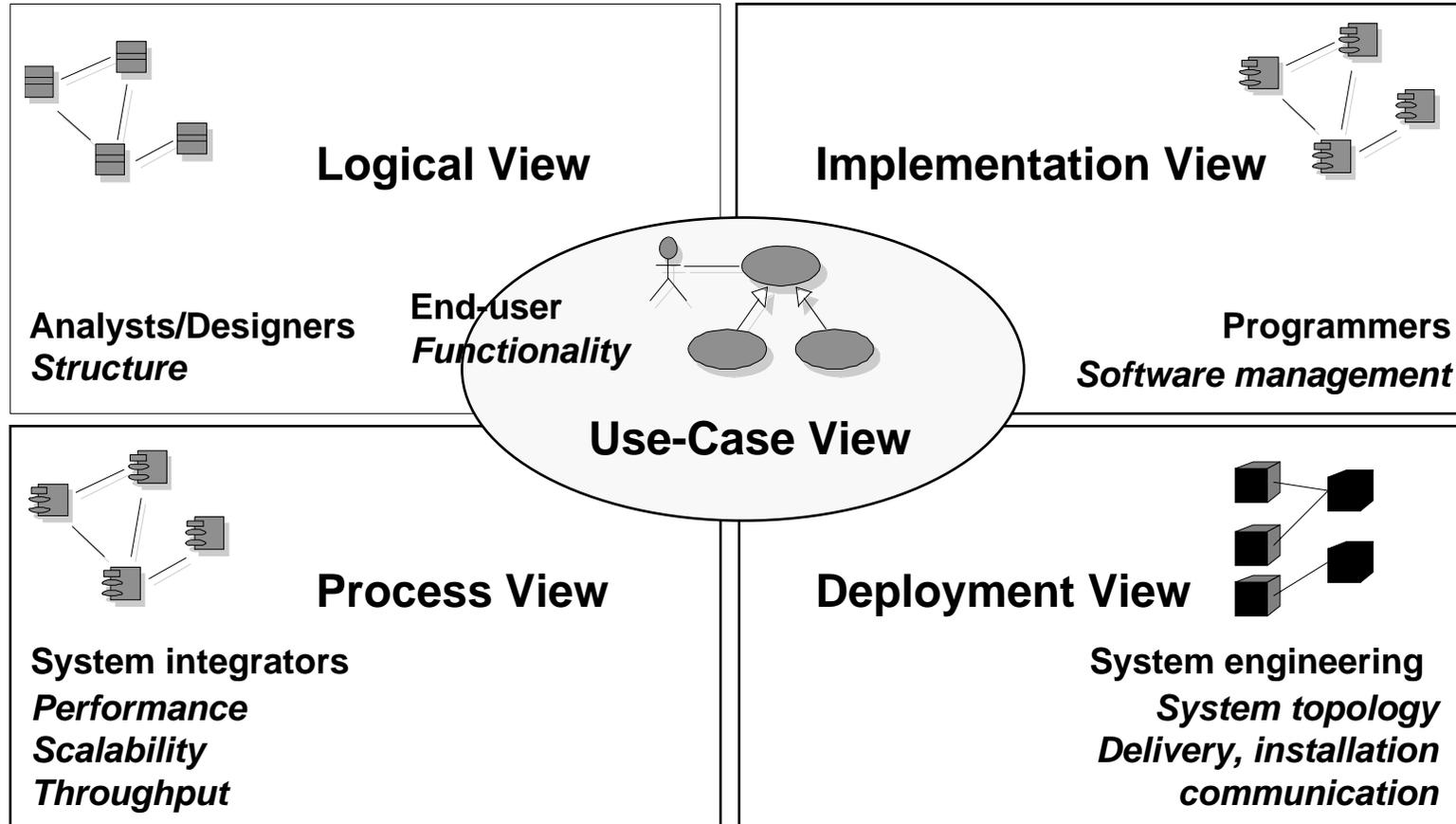


实例：包的跨层依赖



系统实施*Implementation* 视图

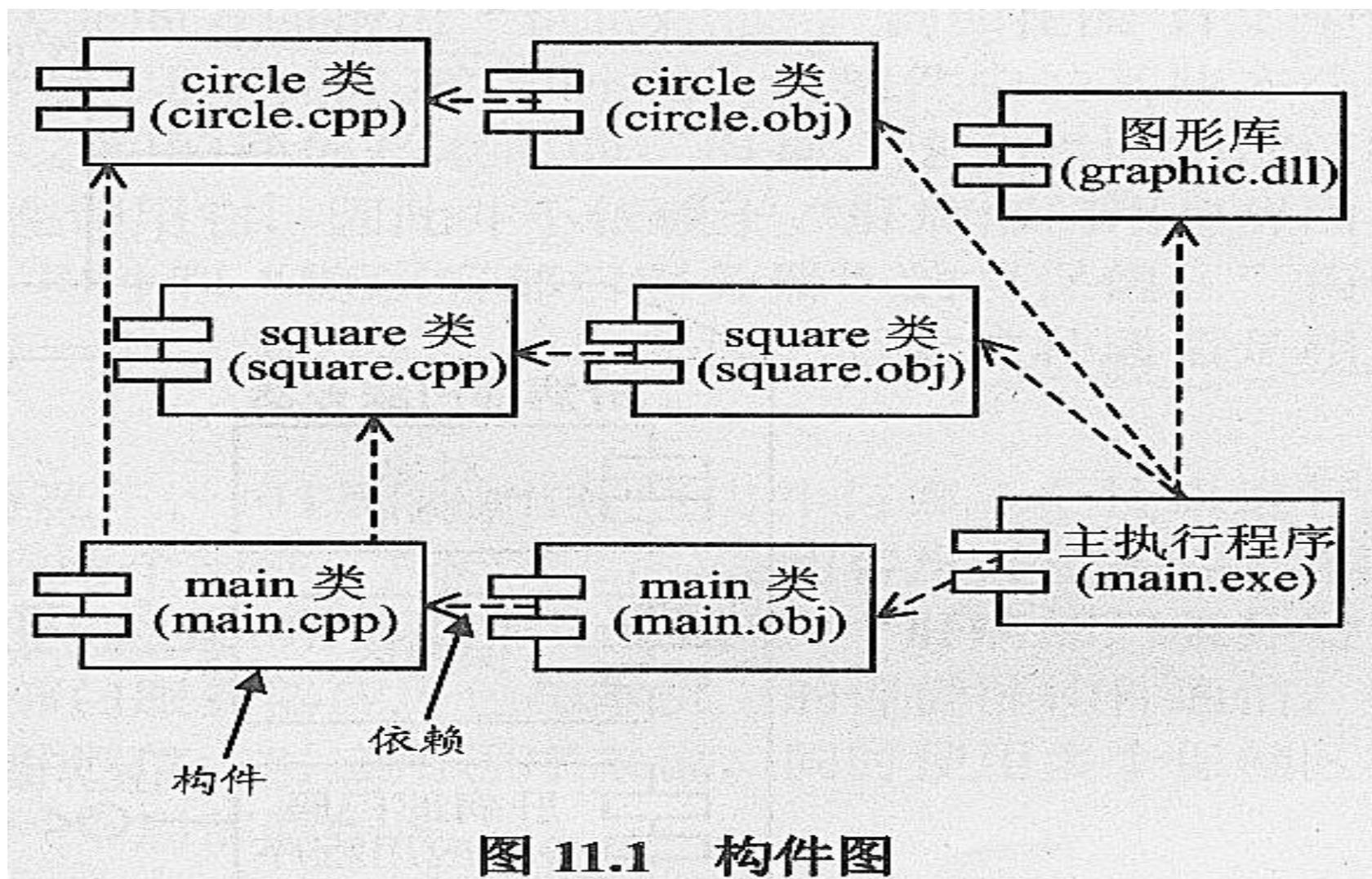
软件构架：实施视图



构件Component图

- 构件图用来表示编译、链接和执行时刻构件之间的依赖关系，以及软件构件间的接口和调用关系
- 软件构件本身是一个物理的实体（实际文件），它可以有不同的类型：
 - ① 源代码构件
 - ② 二进制构件
 - ③ 可执行构件

示例：构件Component图



系统进程*Process*建模

描述并发的步骤

- ★ ● 关键概念
- 并发需求
- 进程建模
- 将进程对应到实施环境
- 将模型元素分布到进程

关键概念：进程与线程

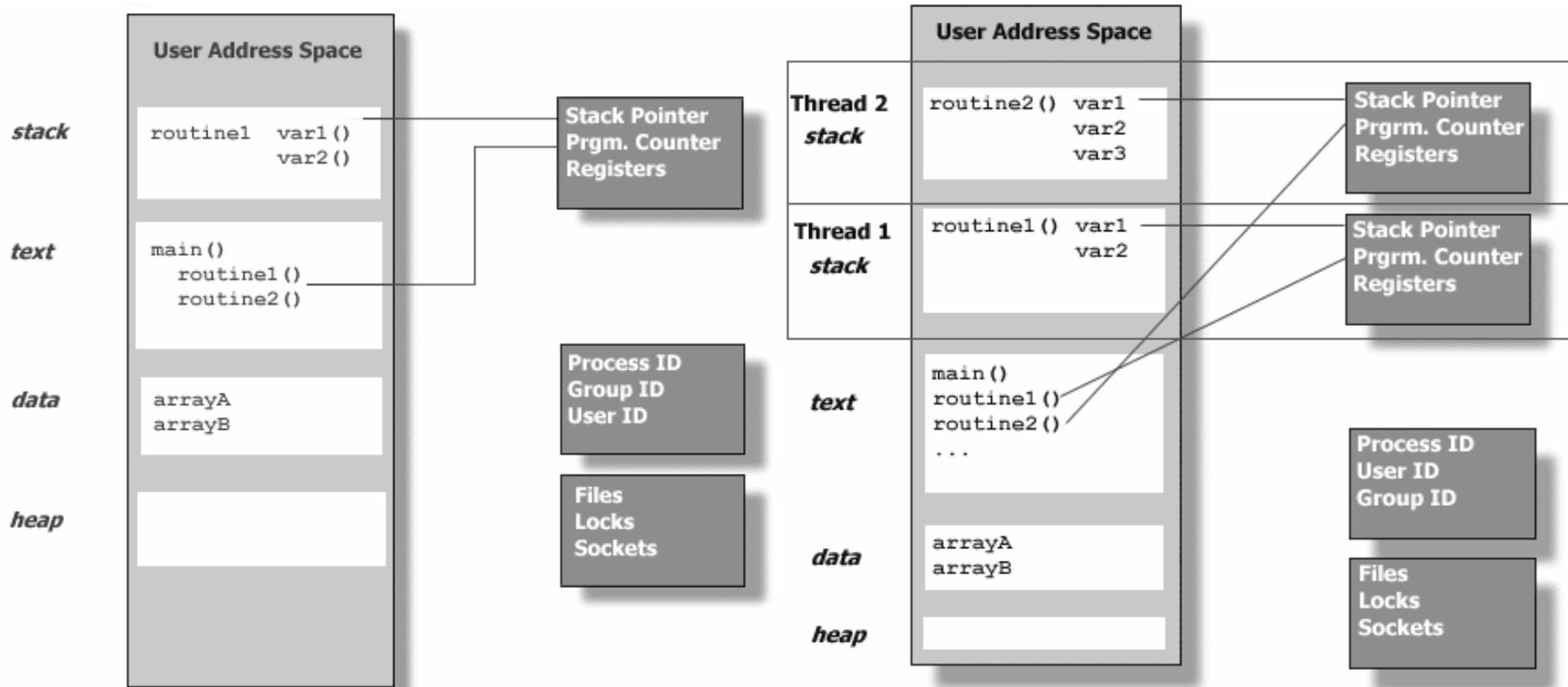
●进程Process

- 重量级的控制流程
- 进程是独立的
- 可能被划分为单独的线程

●线程Thread

- 轻量级的控制流程
- 线程在所处的进程上下文中运行

实例：Process vs. Thread



Unix Process

Threads within Unix Process

实例：Inter-process communication

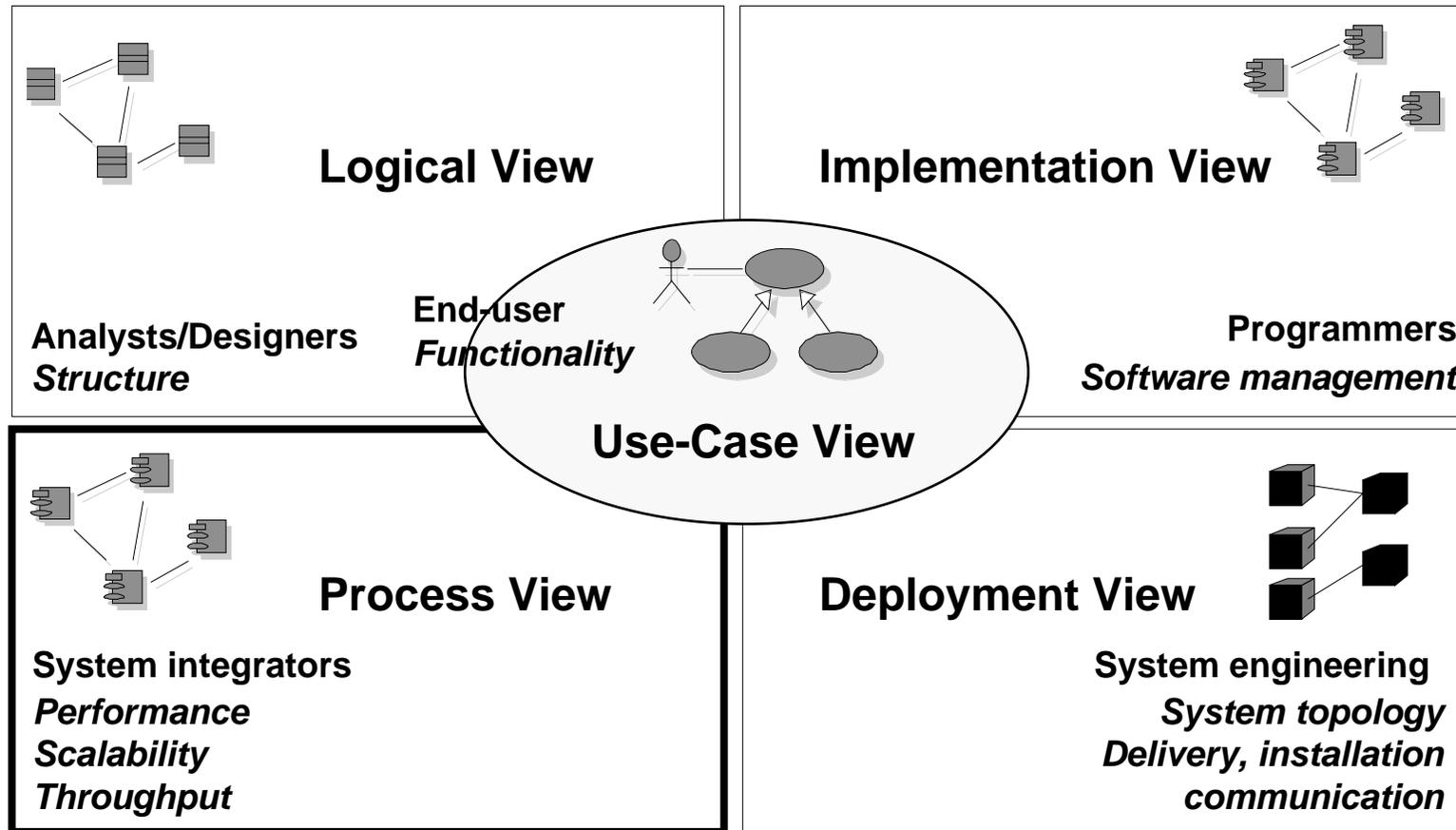
- **Unix内核支持的进程间通讯机制有：**
 - ✓ **Signals**信号
 - ✓ **Pipes**（未命名）管道
 - ✓ **FIFOS**先进先出的命名管道
 - ✓ **Message queues**消息队列
 - ✓ **Semaphores**旗语
 - ✓ **Shared memory**共享内存
- **更高级（跨节点）的进程间通讯机制有：**
 - ✓ **Socket**包交换
 - ✓ **RPC**远程过程调用
 - ✓ **RMI**

实例：Process与Thread性能对比

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
IBM 375 MHz POWER3	61.94	3.49	53.74	7.46	2.76	6.79
IBM 1.5 GHz POWER4	44.08	2.21	40.27	1.49	0.97	0.97
INTEL 1.4 GHz Xeon	23.46	3.92	11.59	2.70	0.55	1.66
INTEL 1.4 GHz Itanium 2	15.98	0.38	8.93	8.21	1.84	0.96

Timings reflect 50,000 process/thread creations, were performed with the time utility, and units are in seconds, no optimization flags.

关键概念：进程视图



The Process View is an “architecturally significant” slice of the processes/threads of the Design Model

并发需求Concurrency Requirement

- 并发需求受以下因素驱动：
 - 系统必须被分布（化）的程度
 - 系统受事件驱动event-driven的程度
 - 关键算法的计算强度
 - 环境所支持的并行运行的程度
- 并发需求依照所解决冲突的重要性来分级

实例：并发需求

- 在课程注册Course Registration系统中，并发需求来自于系统需求和构架：
 - 多个用户必须能够并行执行他们的工作
 - 如果当学生在建立日程表的期间，出现其中一门课程的报名已满，该学生必须被通知
 - 在基于风险的原型中，已经发现遗留的课程编目数据库不能满足性能要求，如果不对中间层的处理能力加以创造性的利用的话

指南：识别进程的考虑因素

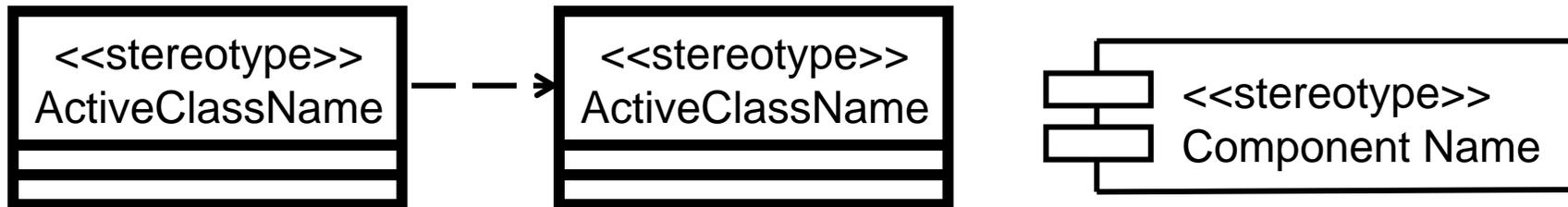
- 可能需要进行控制线划分以便：
 - 利用多个CPU和/或节点的计算能力
 - 提高CPU的利用率（避免对I/O的长时间等待）
 - 提供对外部激励（异步事件）快捷的响应速度
 - 处理时间相关事件（超时、排期的活动）
 - 为活动进行优先级排序
 - 提供伸缩性（负载均衡）
 - 将软件的不同关注区域分开（隔离保密区）
 - 提高系统可用性（冗余备份）
 - 支持主要的子系统（为DBMS安排单独进程）

回顾：进程建模

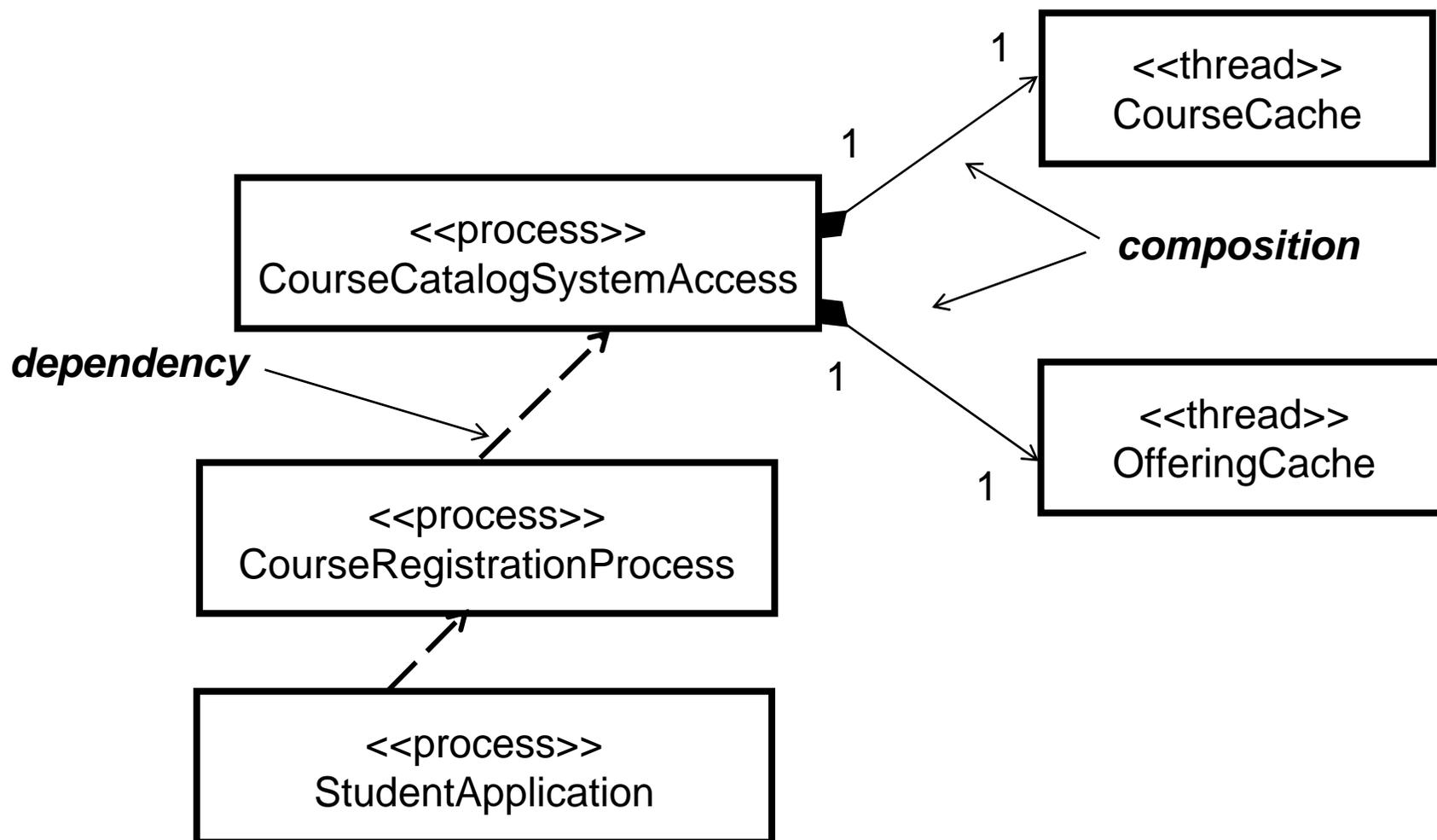
- 关键概念
- 并发需求
- ★ ● 进程建模
 - 将进程对应到实施环境
 - 将模型元素分布到进程

为进程建模

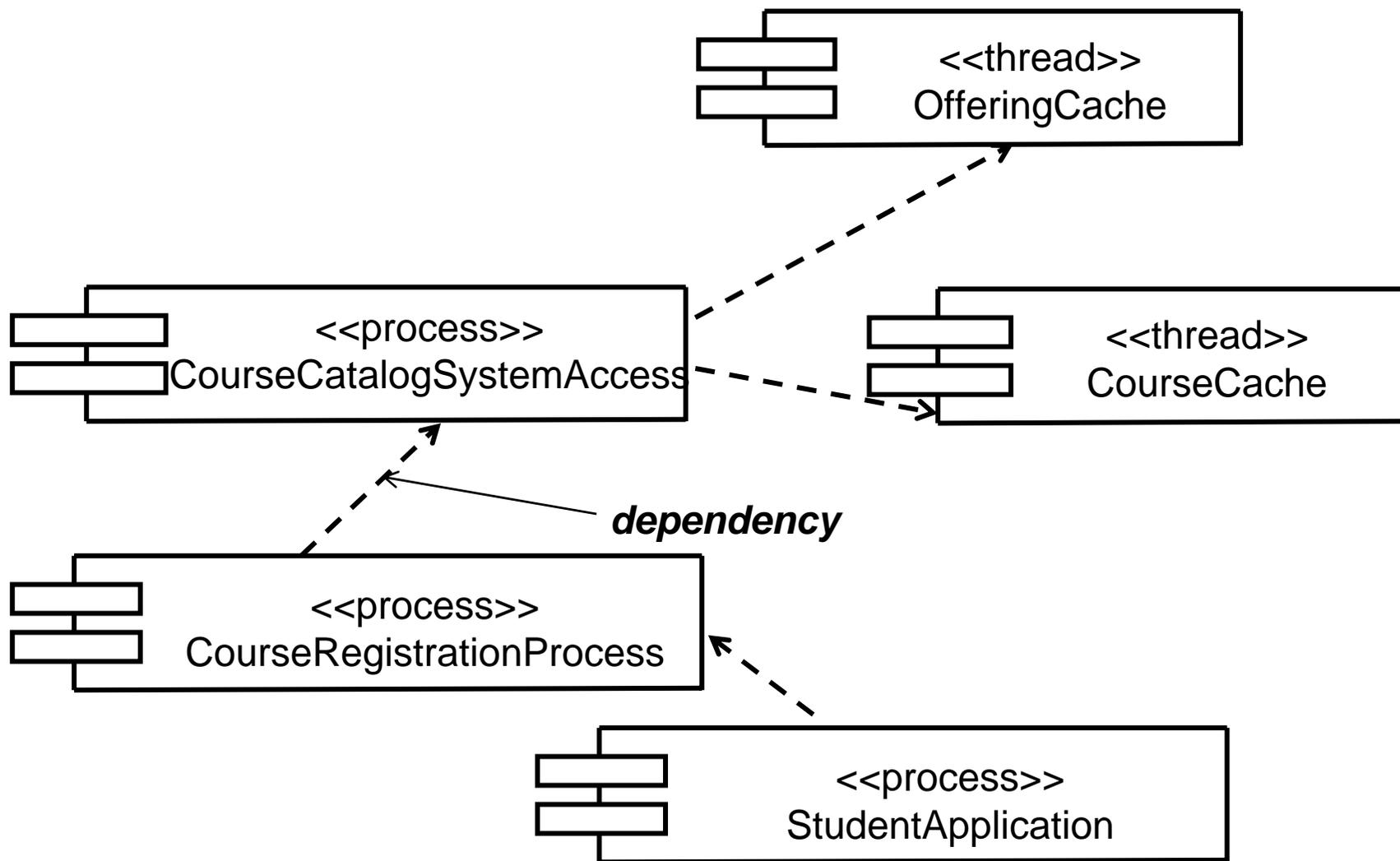
- 进程可以使用下列元素来建模
 - 主动Active类（在类图中）和主动对象（在交互图中）
 - 构件Components（在构件图中）
- 使用版型： <<process>>或<<thread>>
- 进程间的关系可以建模成依赖关系



实例：进程建模—类图



实例：进程建模—构件图



回顾：进程建模

- 关键概念
- 并发需求
- 进程建模
- ★ ● 将进程对应到实施环境
- 将模型元素分布到进程

将进程对应到实施环境

- 进程和线程必须映射到特定的实施 Implementation 部件上
- 考察的因素
 - 进程间的耦合
 - 性能需求
 - 系统的进程和线程限制
 - 已有的线程和进程
 - 可用的 IPC 进程间通讯资源

回顾：进程建模

- 关键概念
- 并发需求
- 进程建模
- 将进程对应到实施环境
- ★ ● 将模型元素分布到进程

指南：将设计元素映射到进程

- 基于下列因素：
 - 性能和并发需求
 - 分布式计算需求和并行执行的支持
 - 冗余和高可用性需求
- 被考察的类/子系统特征：
 - 自治性Autonomy
 - 附属性Subordination
 - 持久性Persistence
 - 分布性Distribution

指南：设计元素映射策略

- 两种策略（被同时使用）

- ✓由里及外Inside-Out

- 将那些密切协作，并且必须在同一控制线索中执行的元素分组在一起

- 将那些不交互的元素分开

- 重复上述步骤，直至到达最小的进程数，在此数目下仍能提供所需分布性和有效的资源利用率

- ✓由外及里Outside-In

- 为每个外部激励定义一个单独的控制线索

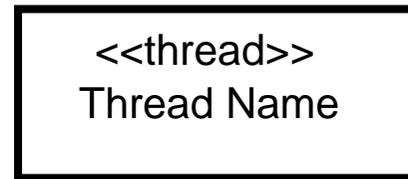
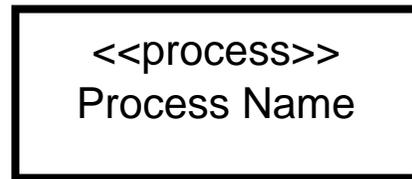
- 为每个服务定义一个单独的服务线程

- 将线程减少到系统能够支持的数目

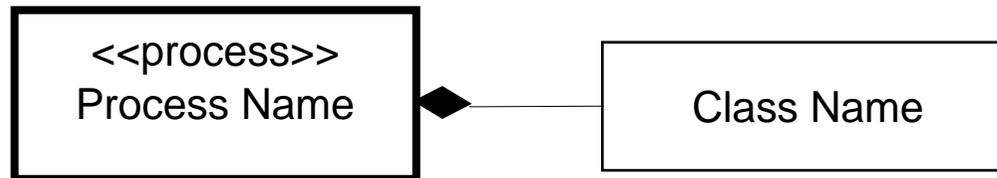
为设计元素到进程的映射建模

● 类图

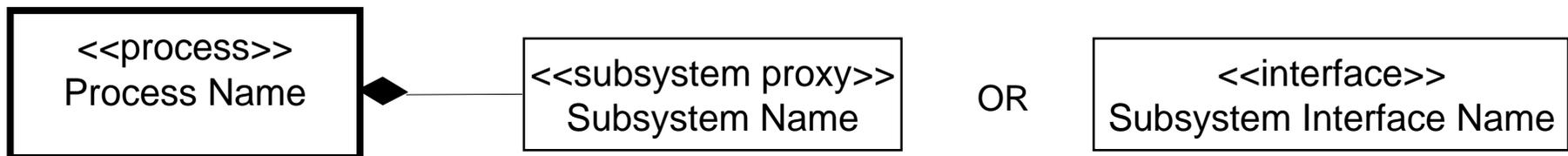
— 主动类作为进程



— 从进程到类的组合关系

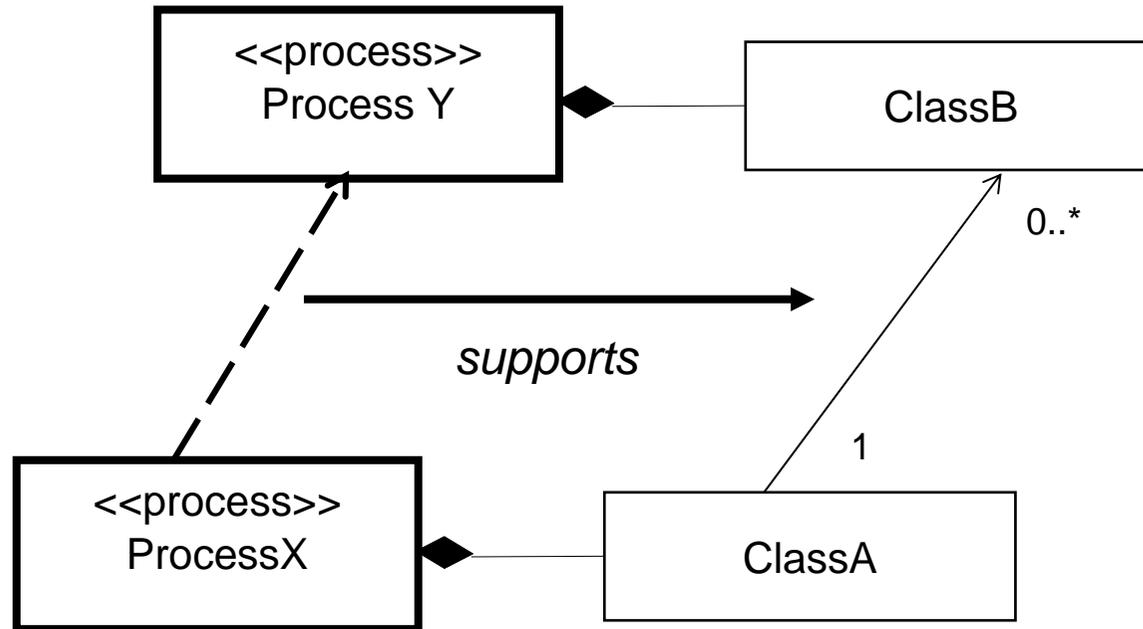


— 从进程到子系统的组合关系

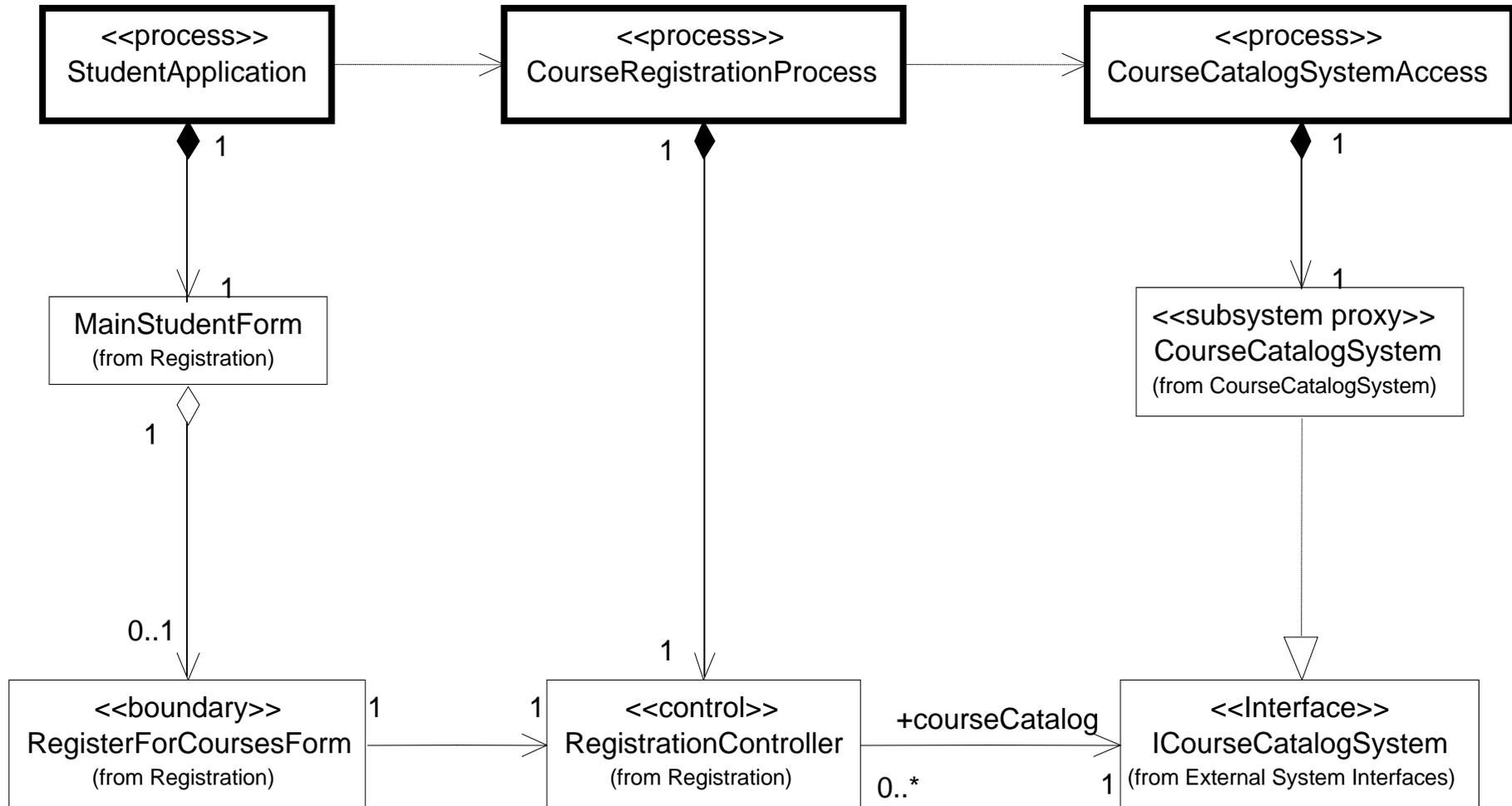


进程间关系

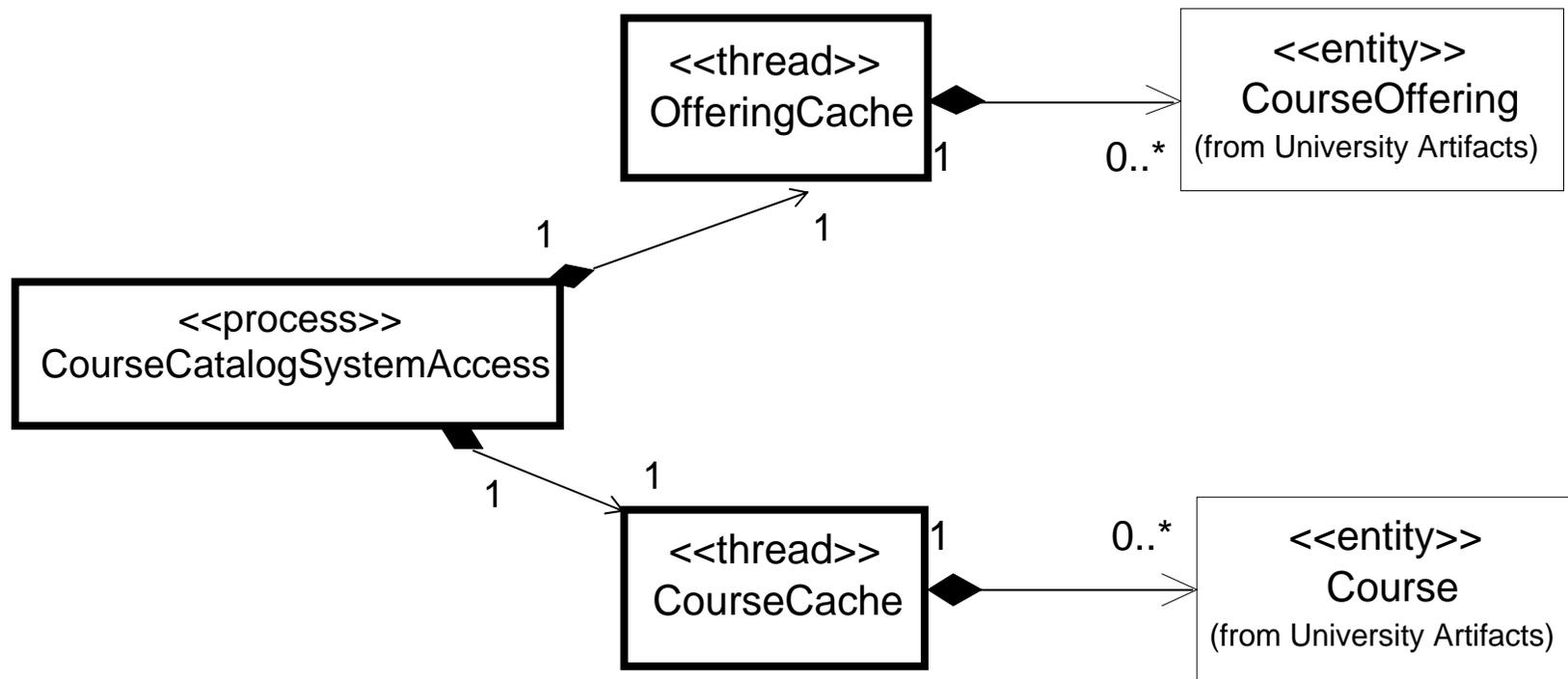
- 进程间关系必须支持设计元素间的关系



实例：课程注册进程



实例：课程注册进程 (contd.)



指南：对象生命周期及其创建顺序

- 目标系统的功能是由一系列对象实例进行协作而实现的
- 对象在进程或线程实例中被创建，其生命期不会长于进程本身
- 各个对象实例要么被其它关联对象所创建，要么被进程直接创建
- 进程实例总是要先实例化一批顶级的根对象实例，其它对象将从它们开始来创建
 - 这些根对象实例通常可以应用单子模式来创建
 - 创建其它对象实例的对象有可能是容器类实例

系统部署Deployment建模

描述分布的步骤

- ★ ● 关键概念
- 分布模式
- 网络配置
- 将进程分配到节点
- 分布机制
- 检查点

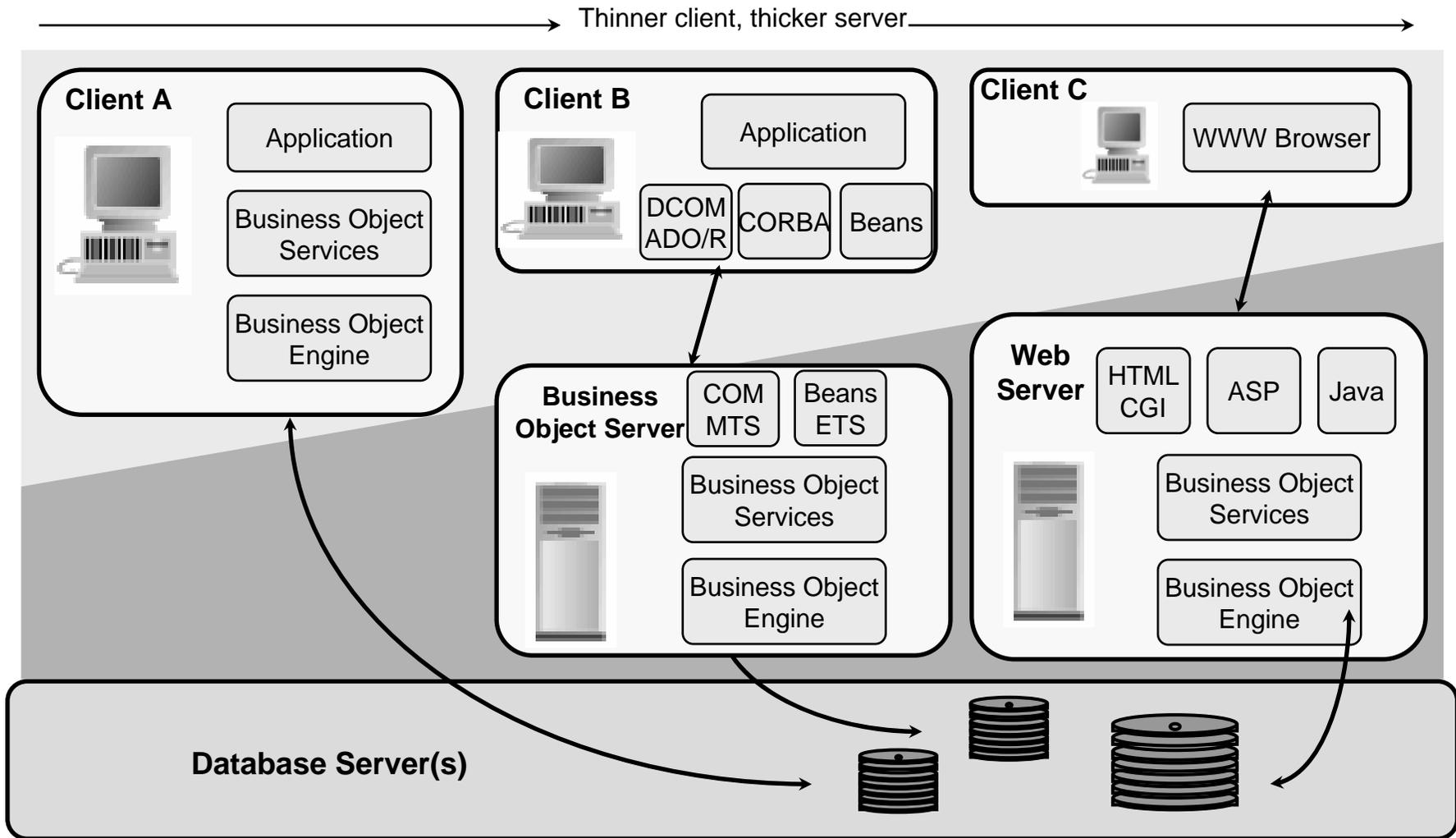
为什么要分布

- 减轻处理器的负载
- 特殊的处理需求
- 伸缩性的关系
- 经济性的关系
- 对系统的分布式访问

分布模式Distribution Pattern

- 客户 Client / 服务器 Server
 - 三层模式 3-tier
 - 胖客户 Fat Client
 - 胖服务器 Fat Server
 - 分布式客户 / 服务器 Distributed Client / Server
- 端对端 Peer-to-peer

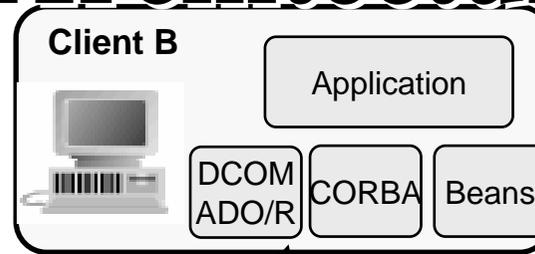
Client/Server Architectures



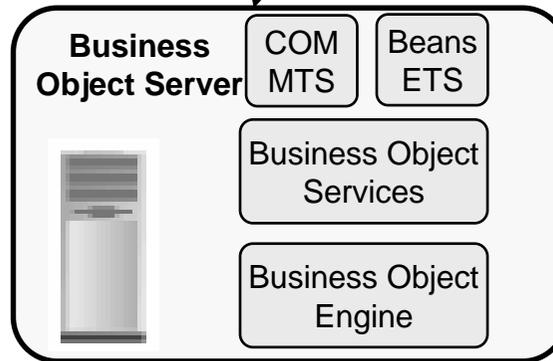
Client/Server: 3-Tier

Architecture

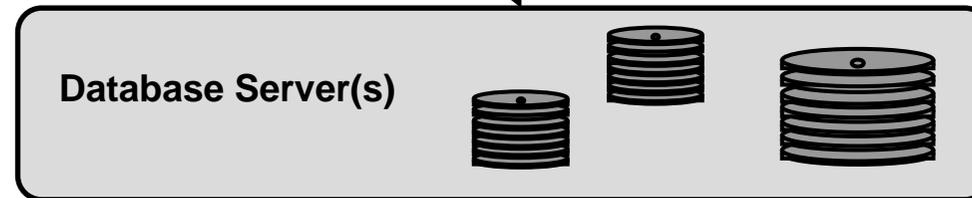
Application Services



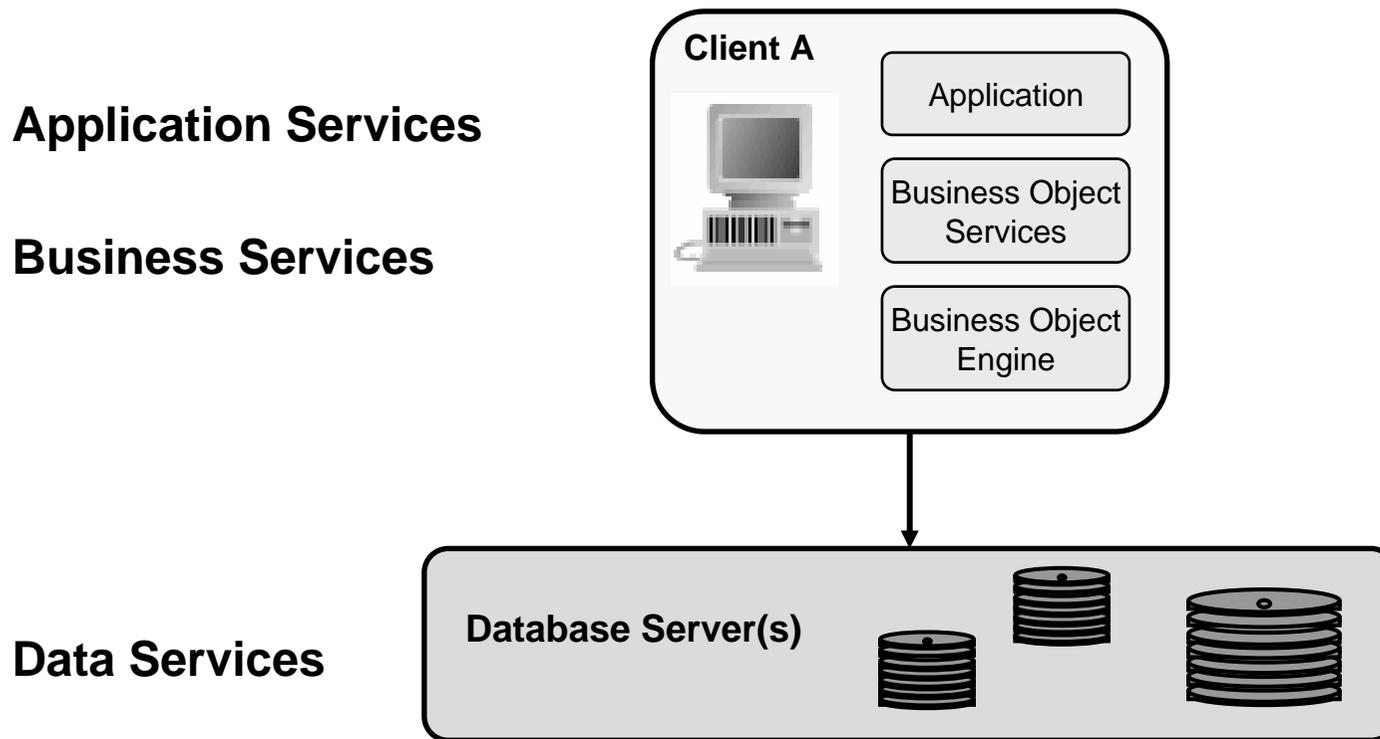
Business Services



Data Services

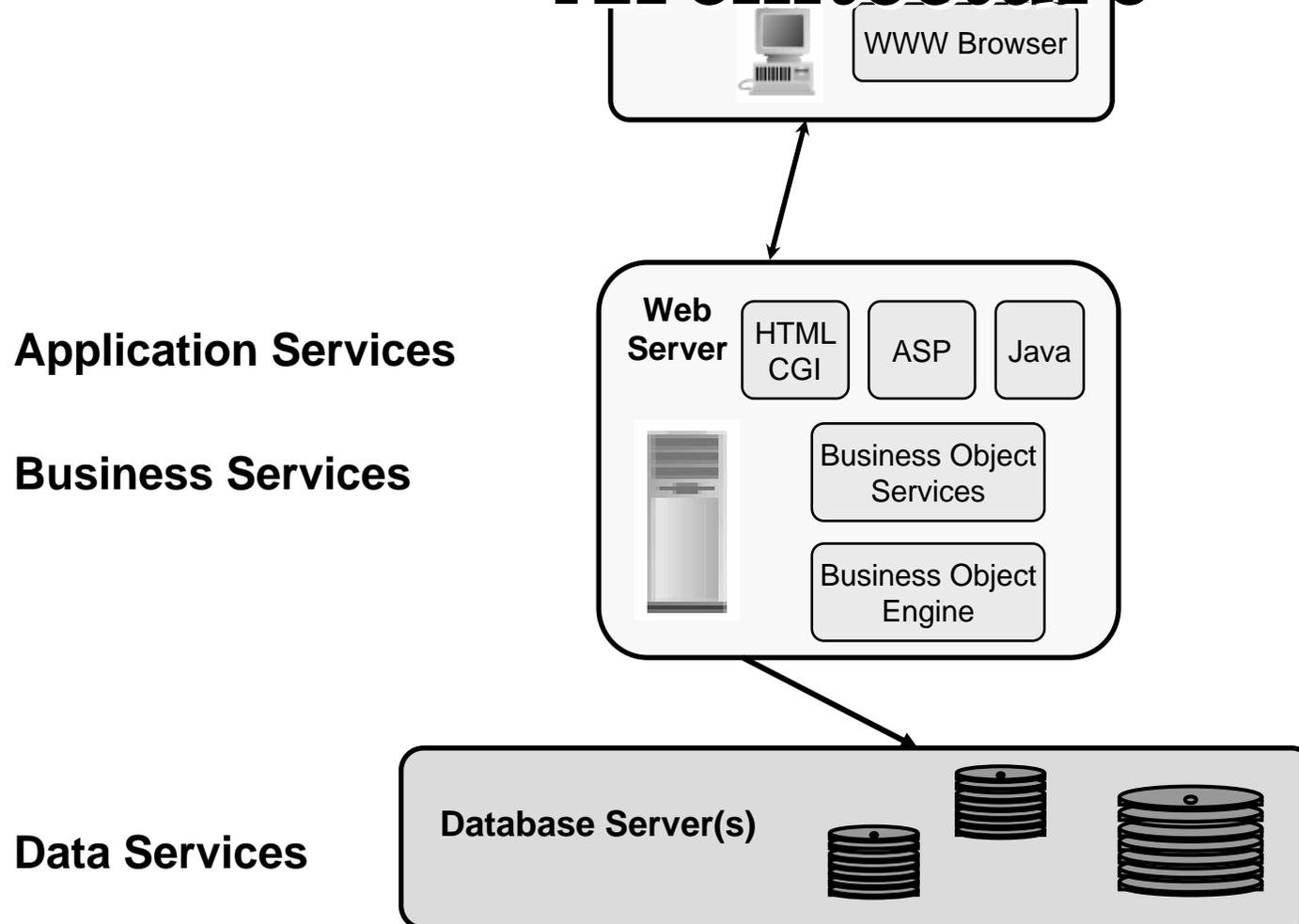


Client/Server: “Fat Client” Architecture



Client/Server: Web

Architecture

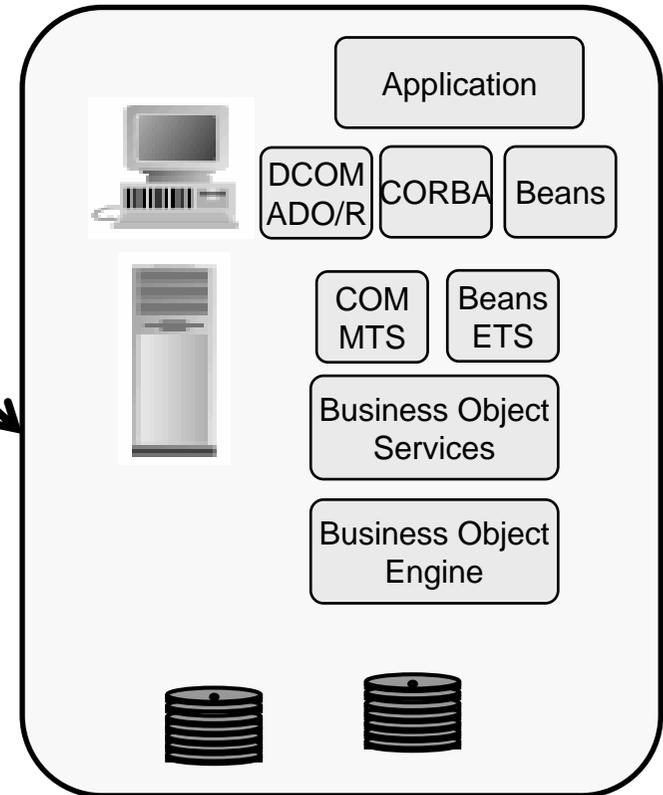
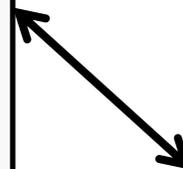
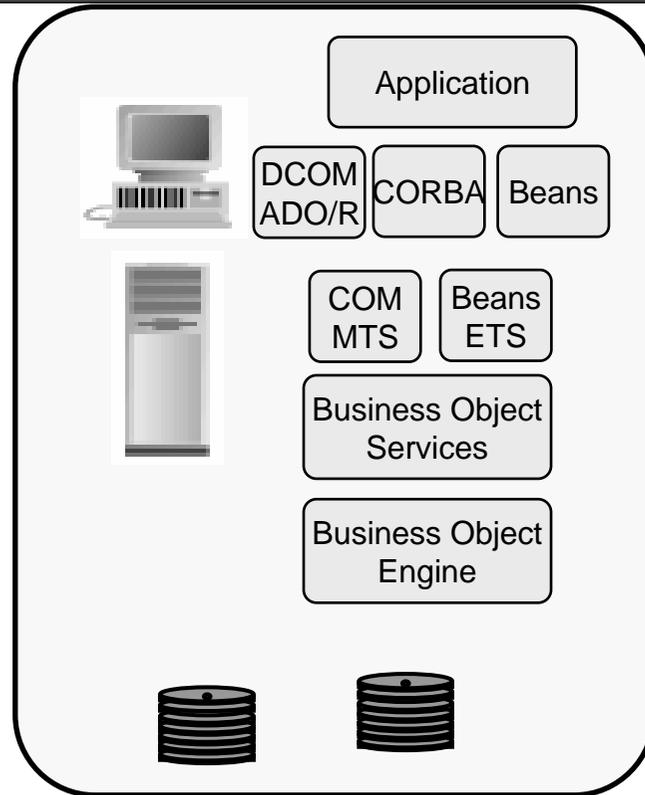


Peer-to-Peer Architecture

Application Services

Business Services

Data Services



回顾：描述分布的步骤

- 关键概念
- 分布模式
- ★ ● 网络配置
 - 将进程分配到节点
 - 分布机制
 - 检查点

网络的配置

- 最终用户工作站节点
- “无头Headless”的服务器节点
- 设备
- 特定的配置
 - 开发
 - 测试
- 特殊的处理器

用于部署模型的建模元素

● 节点 Node

– 运行时刻的物理运算资源

– 处理器节点

- 执行系统软件

– 设备节点

- 支持性设备

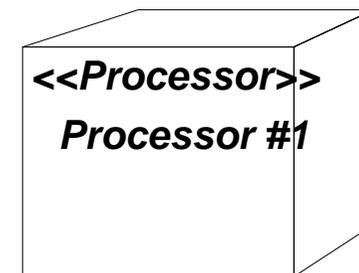
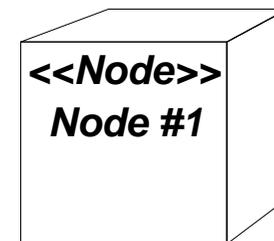
- 典型地将被一个处理器所控制

● 连接 Connection

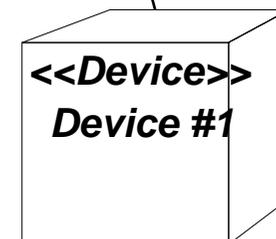
– 通讯机制

– 物理媒介

– 软件协议



Connection



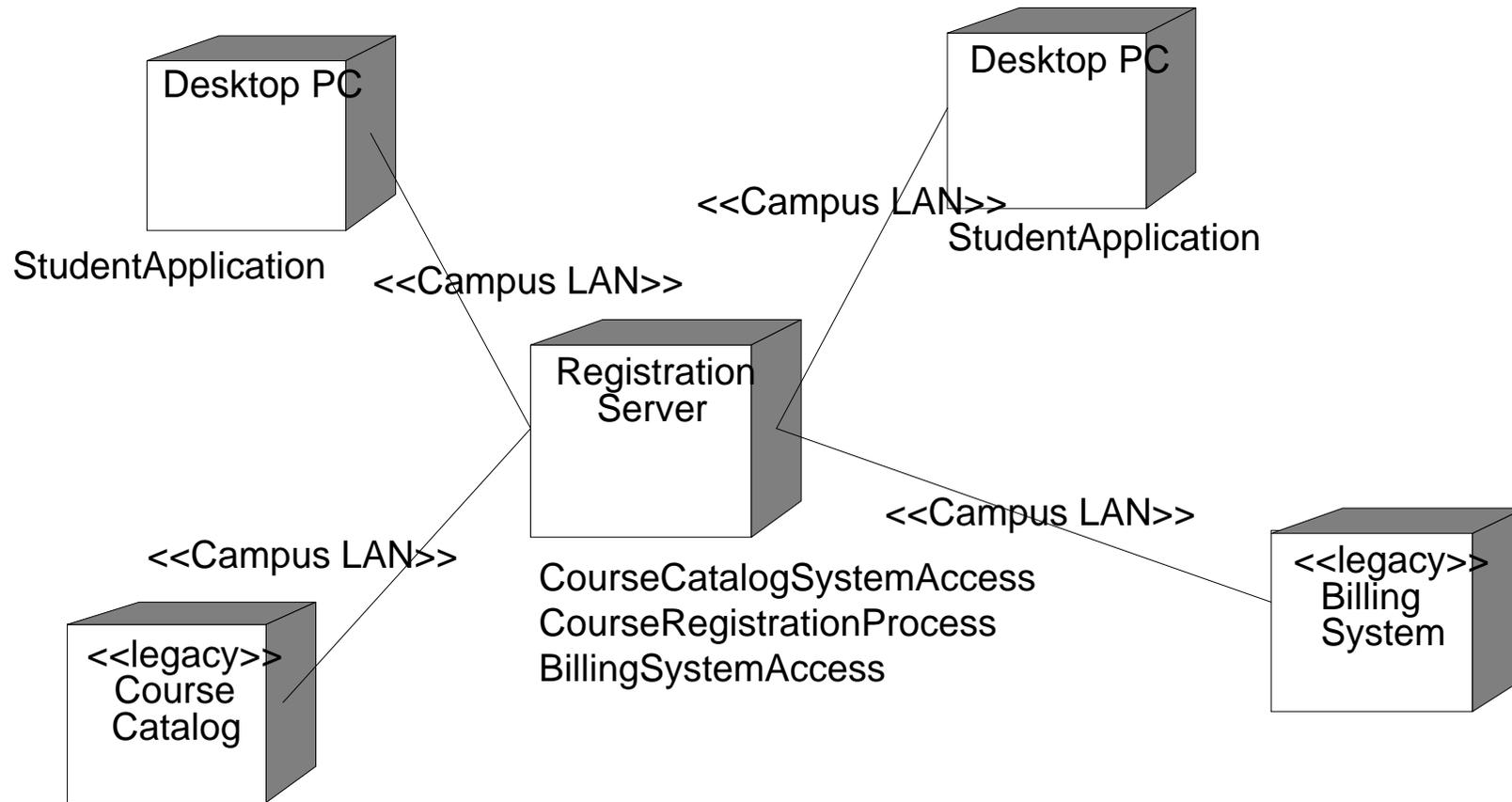
回顾：描述分布的步骤

- 关键概念
- 分布模式
- 网络配置
- ★ ● 将进程分配到节点
- 分布机制
- 检查点

进程向节点分配的考虑因素

- 分布模式
- 响应时间和系统吞吐量
- 最小化穿越网络的流量
- 节点能力
- 通讯媒介的带宽
- 可用的硬件和通讯链路
- 再次路由Rerouting的需求

实例：进程向节点分配

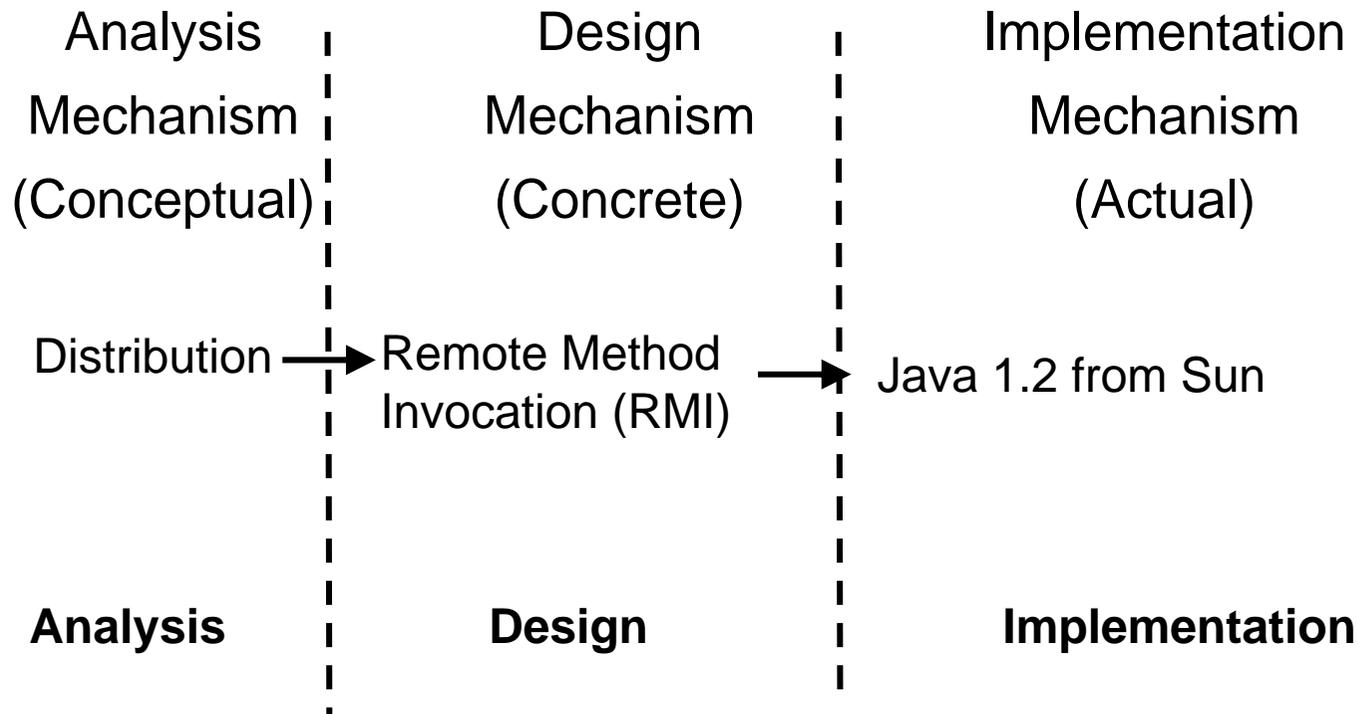


回顾：描述分布的步骤

- 关键概念
- 分布模式
- 网络配置
- 将进程分配到节点
- ★ ● 分布机制
- 检查点

示意：分布机制

●RMI 被选为实现分布的实施机制



Details in Appendix

结语

- UML构架建模的元素包括：说明构架顶层结构的包图。描述构架敏感类的类图，描述用例实现的序列图和协作图，以及描述构架重要状态的状态图等；
- 开发软件构架的主要有构架分析和构架设计；
- 除了系统的逻辑视图，表达系统的进程视图、实施视图和部署视图同样十分重要。

参考资料

<UML Distilled: A Brief Guide to the Standard
Object Modeling Language>
<Applying UML and Patterns>
<Writing Effective Use Case>
<RUP>

Q&A

谢谢!