系统分析与设计归纳（IBM DEV475 部分）

> 考试说明：
>
> 考试时间为：2020.11.4,14:30-16:30.
>
> 地点：A2-307/308
>
> 考试范围：课程参考文献、课堂教学内容及相关的基础知识。
>
> 英文试卷，可以中文作答。
>
> 左保河

左保河老师应该只讲了 IBM DEV475 的前 7 章。不保证正确及完整性。

Zhc

2020-11-3

目录

1. 软件开发问题的症状
   A. 与用户或业务需要不匹配
   B. 需求未被解决
   C. 模块不能整合
   D. 维护困难
   E. 瑕疵发现得晚
   F. 最终用户体验的质量差
   G. 负载下的性能差
   H. 不协调的团队效果
   I. 构建和发布问题

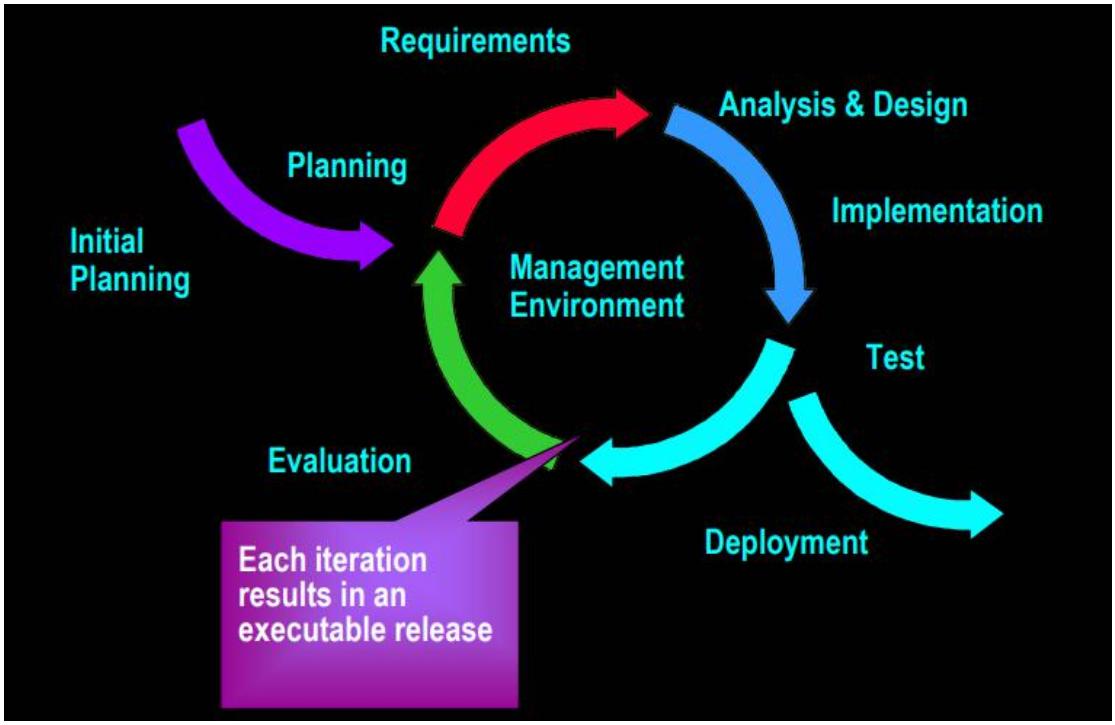
2. 追踪问题的根本原因

症状	**根本原因**	**最佳实践**
与用户或业务需要不匹配	不充分的需求	迭代式开发（develop iteratively）
需求未被解决	不明确的沟通	管理需求（manage requirements）
模块不能整合	脆弱的架构	使用组件架构（use component architectures）
维护困难	巨大的复杂度	可视化建模（UML）model visually
瑕疵发现得晚	未被发现的不一致	持续核查质量（continuously verify quality）
最终用户体验的质量差	糟糕的测试	管理变更（manage change）
负载下的性能差	主观的看法	
不协调的团队效果	瀑布模型	
构建和发布问题	不充分的自动化	

3. 6 个最佳实践
   A. 迭代式开发
      （1）瀑布模型的特点
      瀑布过程：

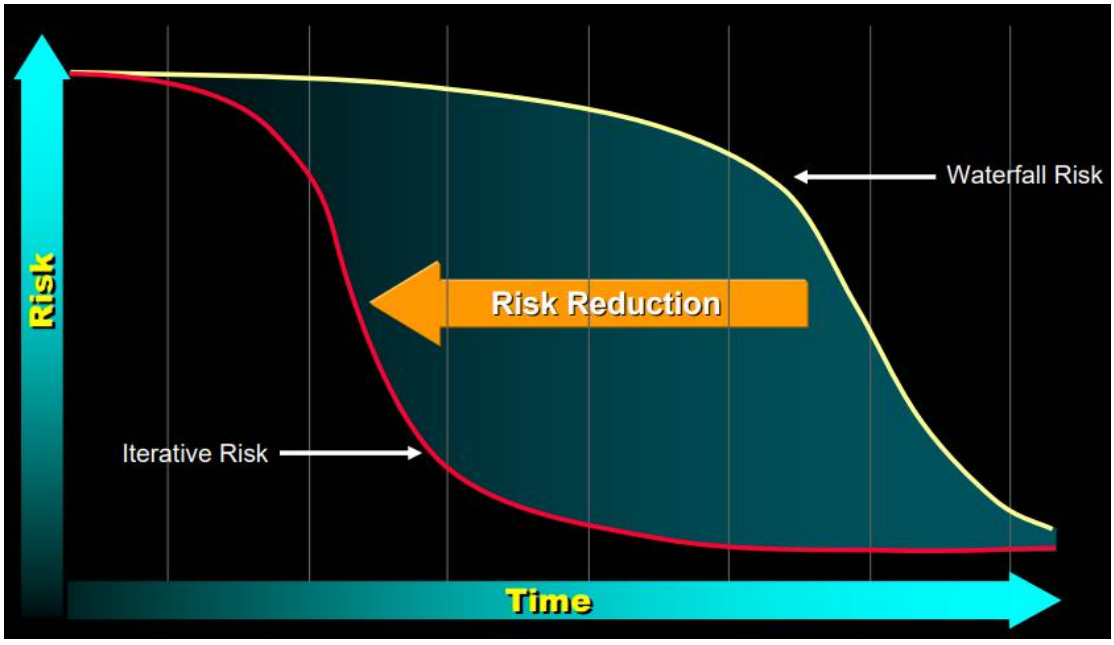
延迟了关键风险决议的确认。
通过评估工作制品来度量进展，而工作制品是糟糕的指标来估计完成时间。
延迟且集中了整合和测试阶段。
阻止了早期部署。
频繁导致重大的计划外迭代。

（2）迭代式开发产生一个可执行的（发布版本）



每次迭代产生一个可执行的发布版本。

（3）风险概述



B. 管理需求

（1）需求管理

通过采取一个系统的方法来发现、组织、归档和管理一个软件应用不断变化的需求，确保你解决了正确的问题并构建了正确的系统。

（2）需求管理的不同方面

分析问题；

明白用户需要；

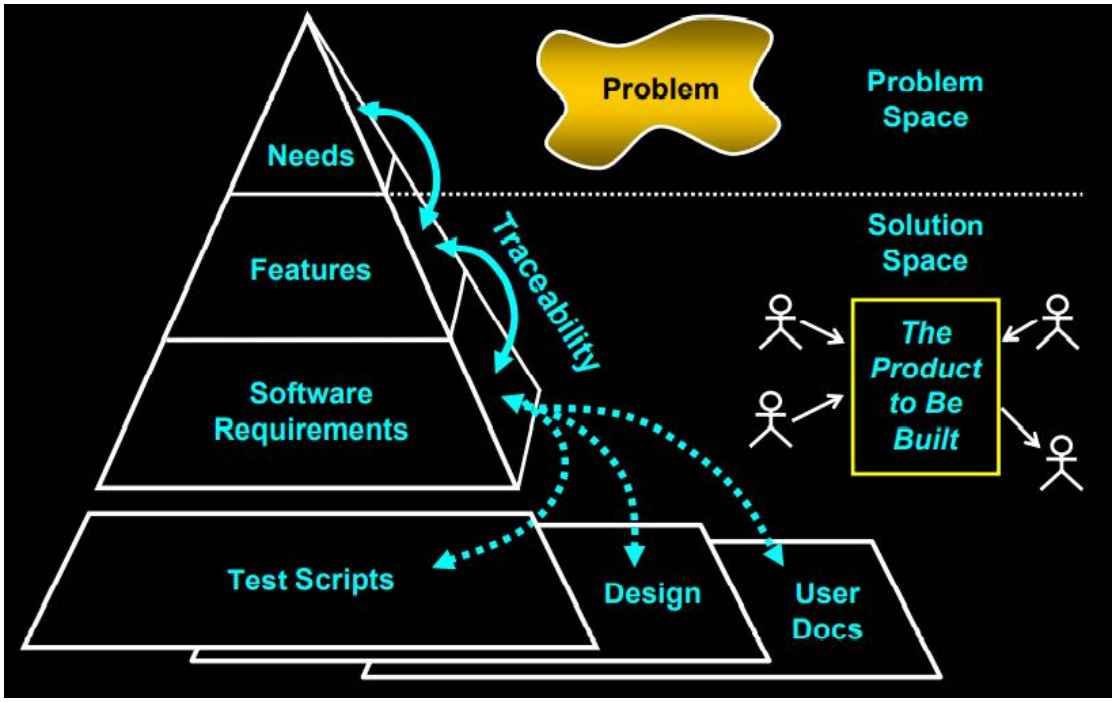
定义系统；

管理范围（scope）；

改进系统定义；

管理不断变化的需求。

（3）领域图（Map of the Territory）



C. 使用组件架构

（1）有弹性的（resilient）基于组件的架构

有弹性的：符合当前和未来的需求；提高可扩展性；允许复用；封装（encapsulate）系统依赖。

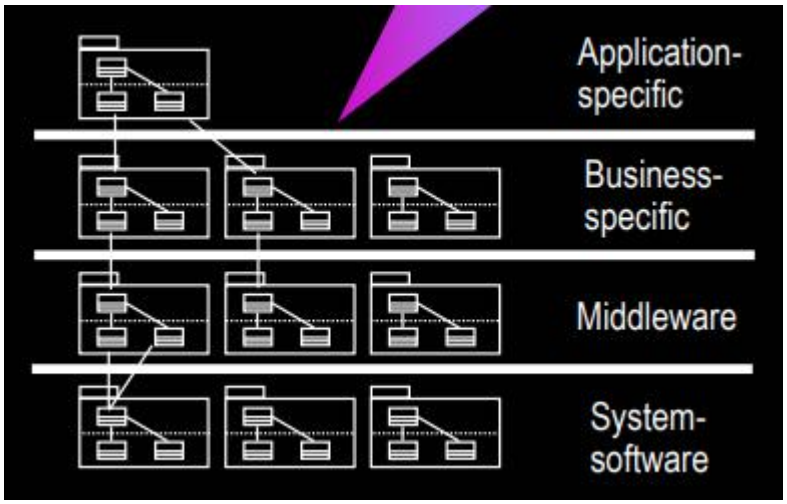
基于组件的：复用或自定义（customize）组件；从商业上可用的组件中选择；增量式（incrementally）演化（evolve）现存的软件。

（2）基于组件的架构的目的

复用的基础：组件复用和架构复用。

项目管理的基础：计划、人员配置和交付（delivery）。

理智的（intellectual）控制：管理复杂度和保持完整性（integrity）。



分层的基于组件的架构

D. 可视化建模（UML）

（1）为何要可视化建模？

捕捉结构和行为；

展示系统元素如何相互配合；

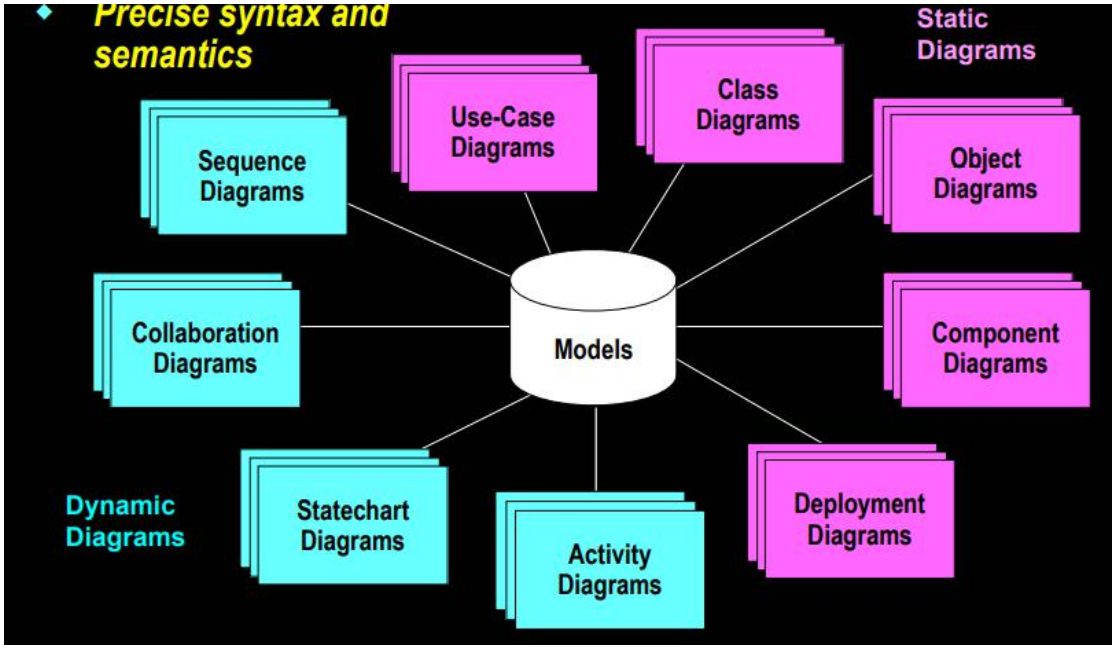
保持设计和实现的一致性；

适当地隐藏或揭露细节；

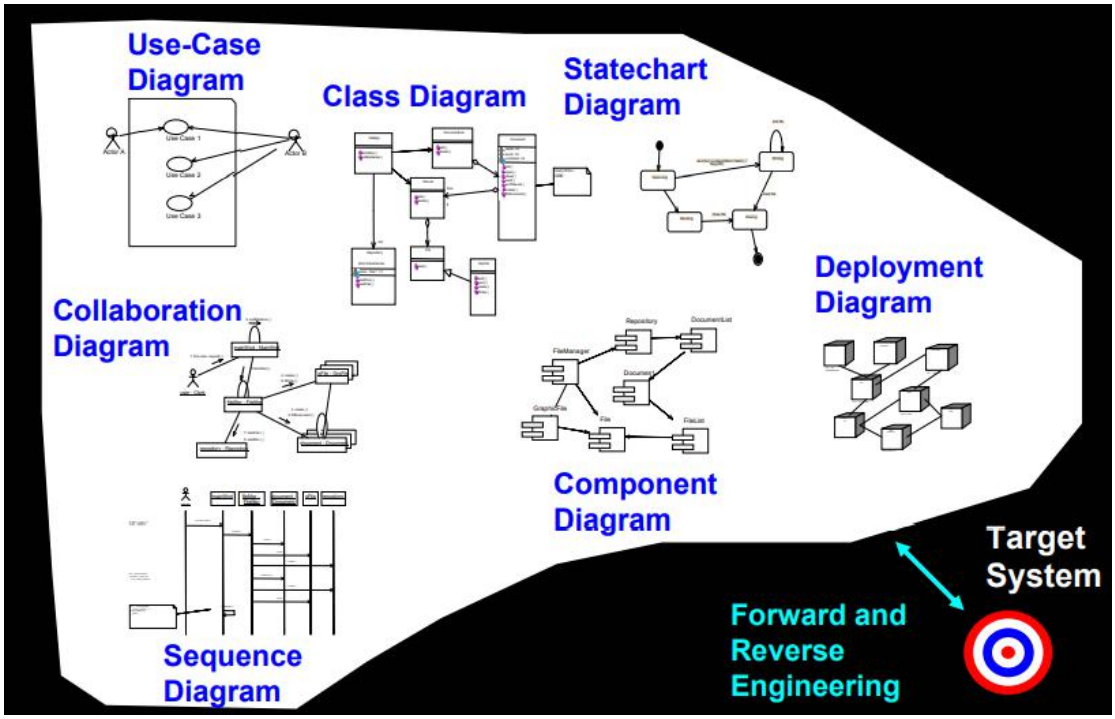
促进无歧义的沟通（UML 为所有从业人员提供了一门语言）。

（2）用统一建模语言进行可视化建模

多种多样的视图、准确的句法（syntax）和含义（semantic）。

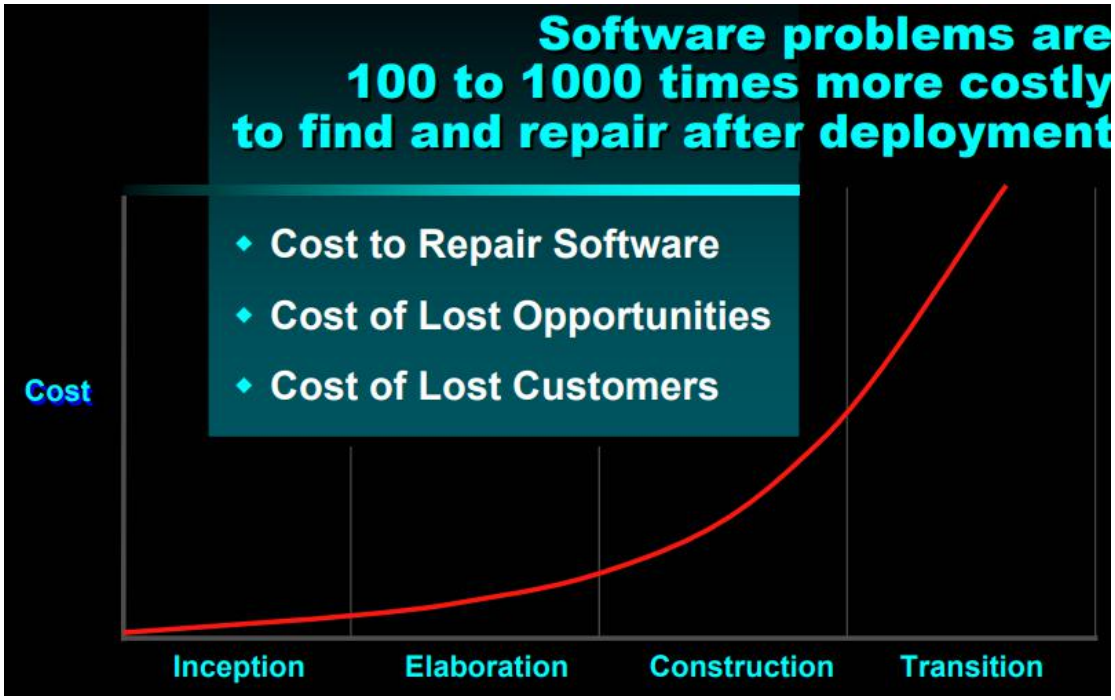


（3）用 UML 图进行可视化建模



E. 持续核查质量

（1）持续核查你的软件的质量



部署后，发现和修复软件问题的成本会变为原来的 100 到 1000 倍：修复软件的成本、机会损失的成本和失去客户的成本。

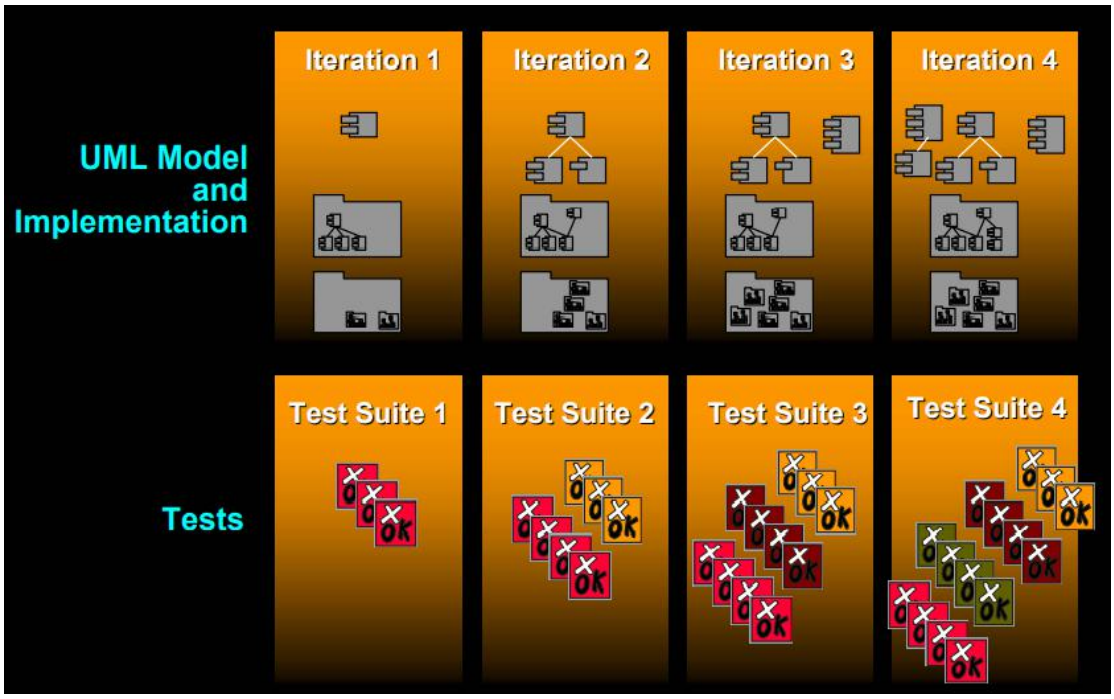
（2）质量的测试维度

（a）可用性（usability）：考虑最终用户使用的方便性来测试应用。

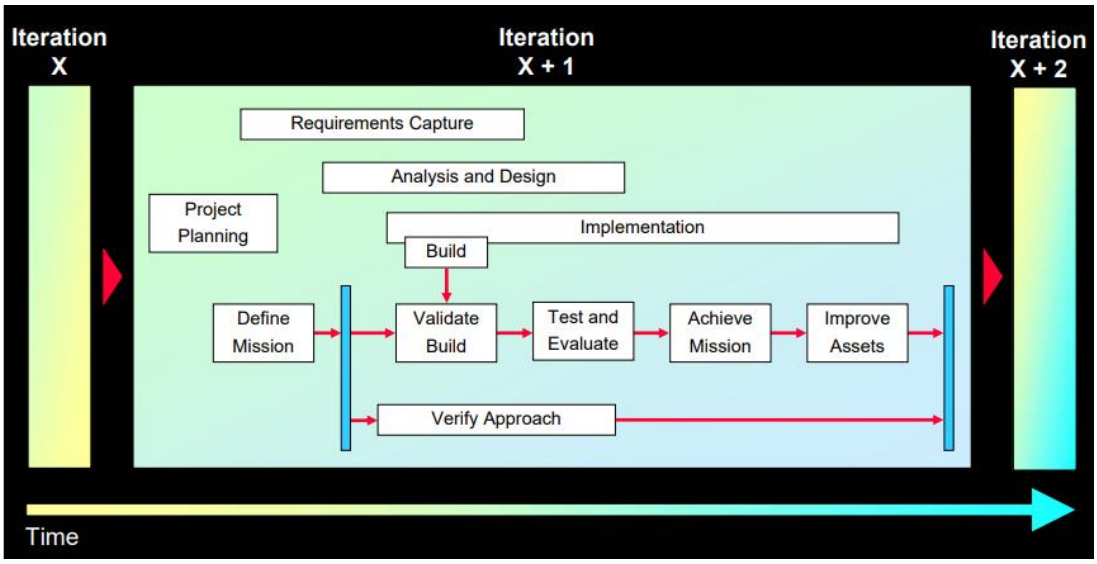
（b）功能性（functionality）：测试每个使用场景的准确工作。

（c）可靠性（reliability）：测试应用是否表现得一致和可预测。
（d）可支持性（supportability）：测试在生产使用中维护和支持应用的能力。
（e）性能（performance）：测试在平均和峰值负载下的在线响应。

（3）测试每次迭代



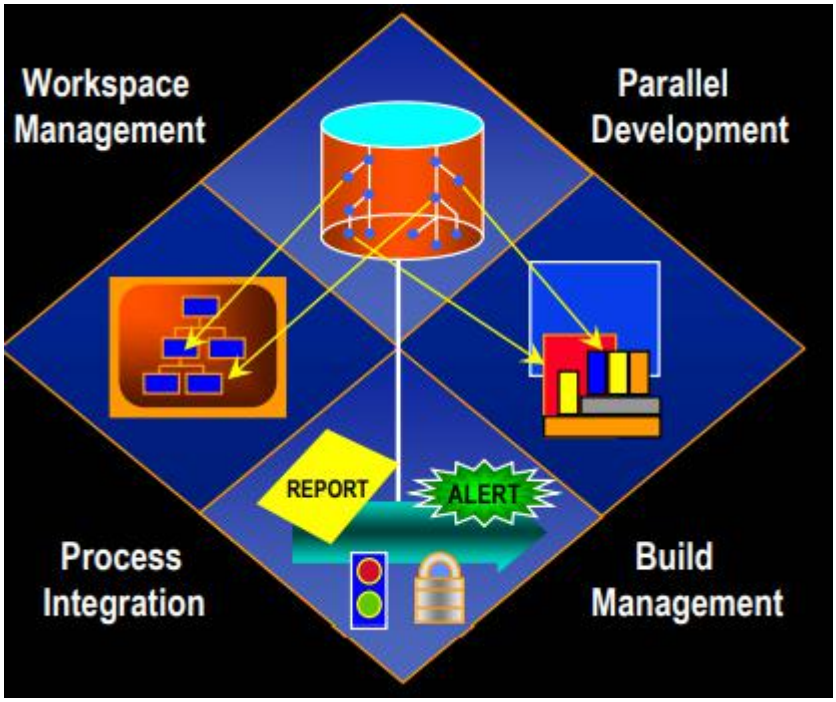
（4）在产品开发的生命周期中测试



F. 管理变更

（1）你想控制什么？

（a）保护每个开发者的工作空间。
（b）自动化整合/构建管理。
（c）并行开发。



配置管理不只是 check-in（相反）和 check-out（标记代码段，防止别人修改）。参考：软件开发中说的 check out, check in 是什么意思？
https://zhidao.baidu.com/question/209940618.html

（2）变更管理系统的方面

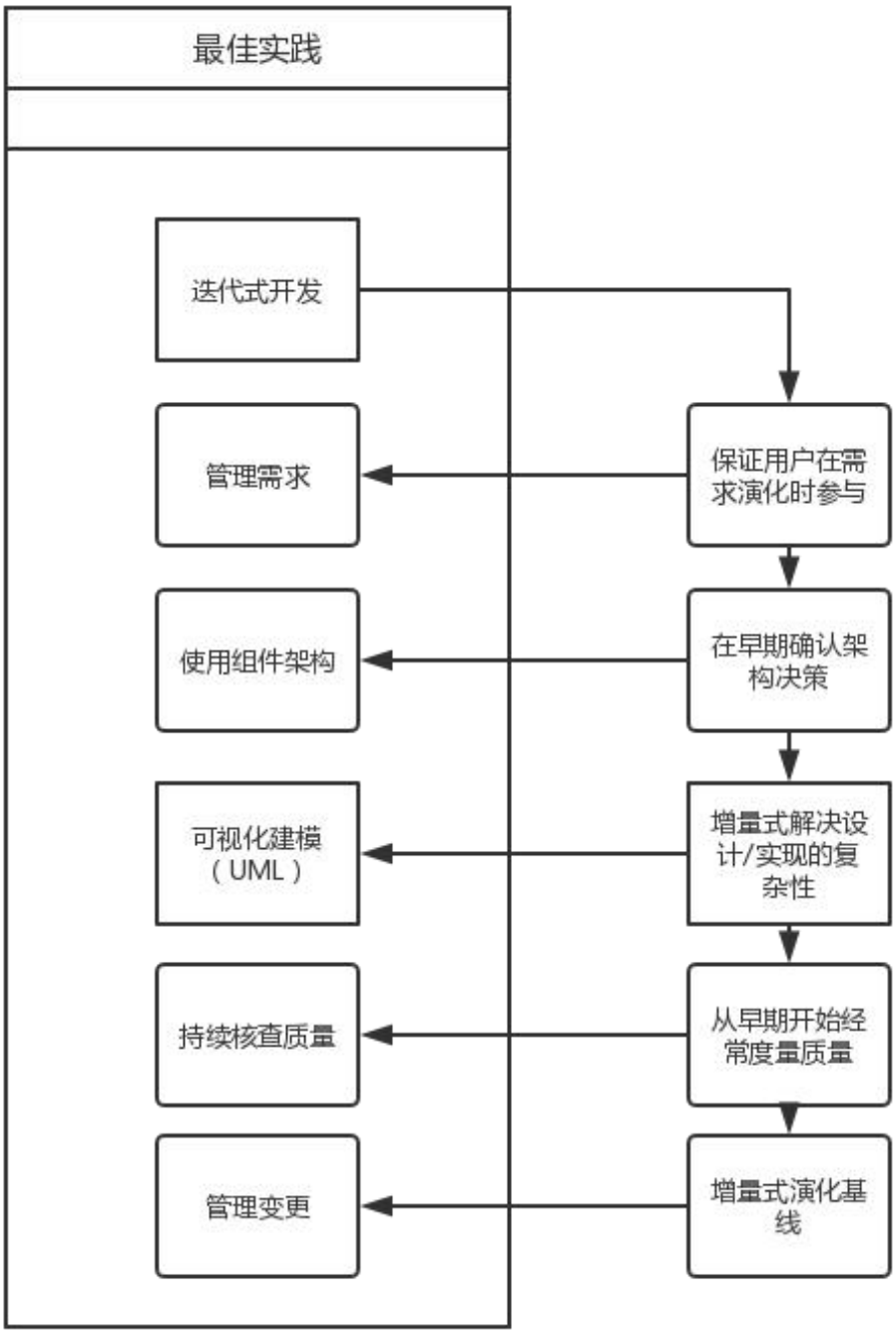
变更请求管理（Change Request Management，CRM）、
配置状态报告、
配置管理、
变更追踪、
版本选择、
软件制造。

（3）统一变更管理（Unified Change Management, UCM）

包括：

（a）生命周期的管理：系统和项目管理。
（b）基于活动的管理：任务、缺陷和增强。
（c）进展追踪：表格和报告。

（4）最佳实践帮助每个人



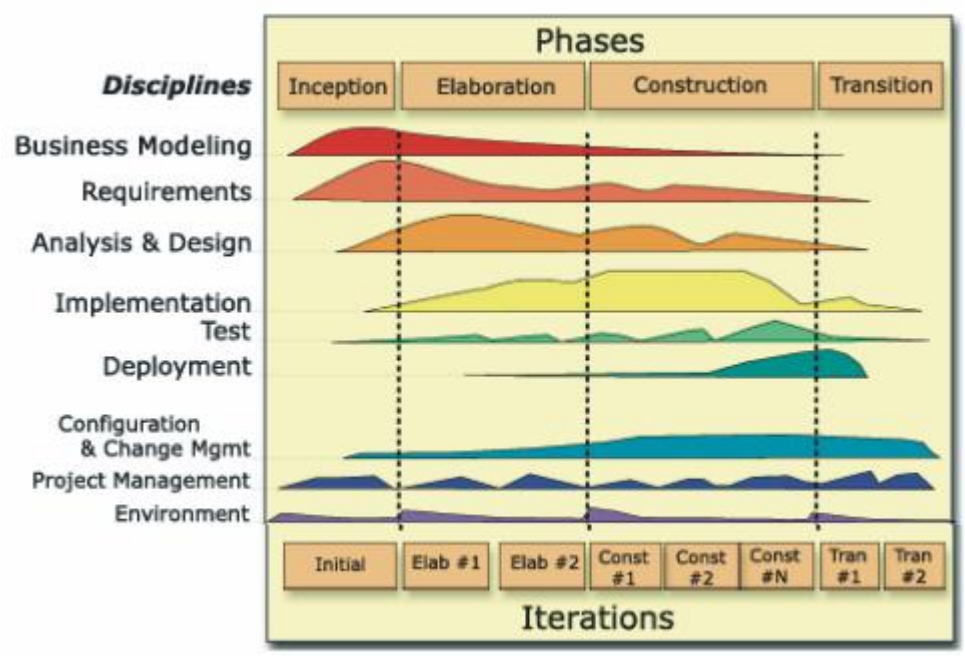
4. 在 6 个最佳实践背景下的 RUP

A. Rational 统一过程（Rational Unified Process）实现最佳实践

B. 达到最佳实践

迭代式方法；

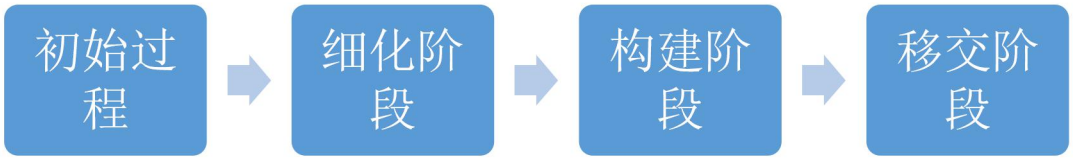
为活动和制品提供的指导；
过程的焦点放在架构上；
用例驱动并设计实现；
模型抽象系统。



C. 基于团队的过程定义

一个过程定义了谁干什么，何时干，如何干，从而达到特定的目标。

D. 过程结构：生命周期



（1）初始阶段（inception）：定义项目的范围。

（2）细化阶段（elaboration）：计划项目、详述特性和基础架构（baseline architecture）。

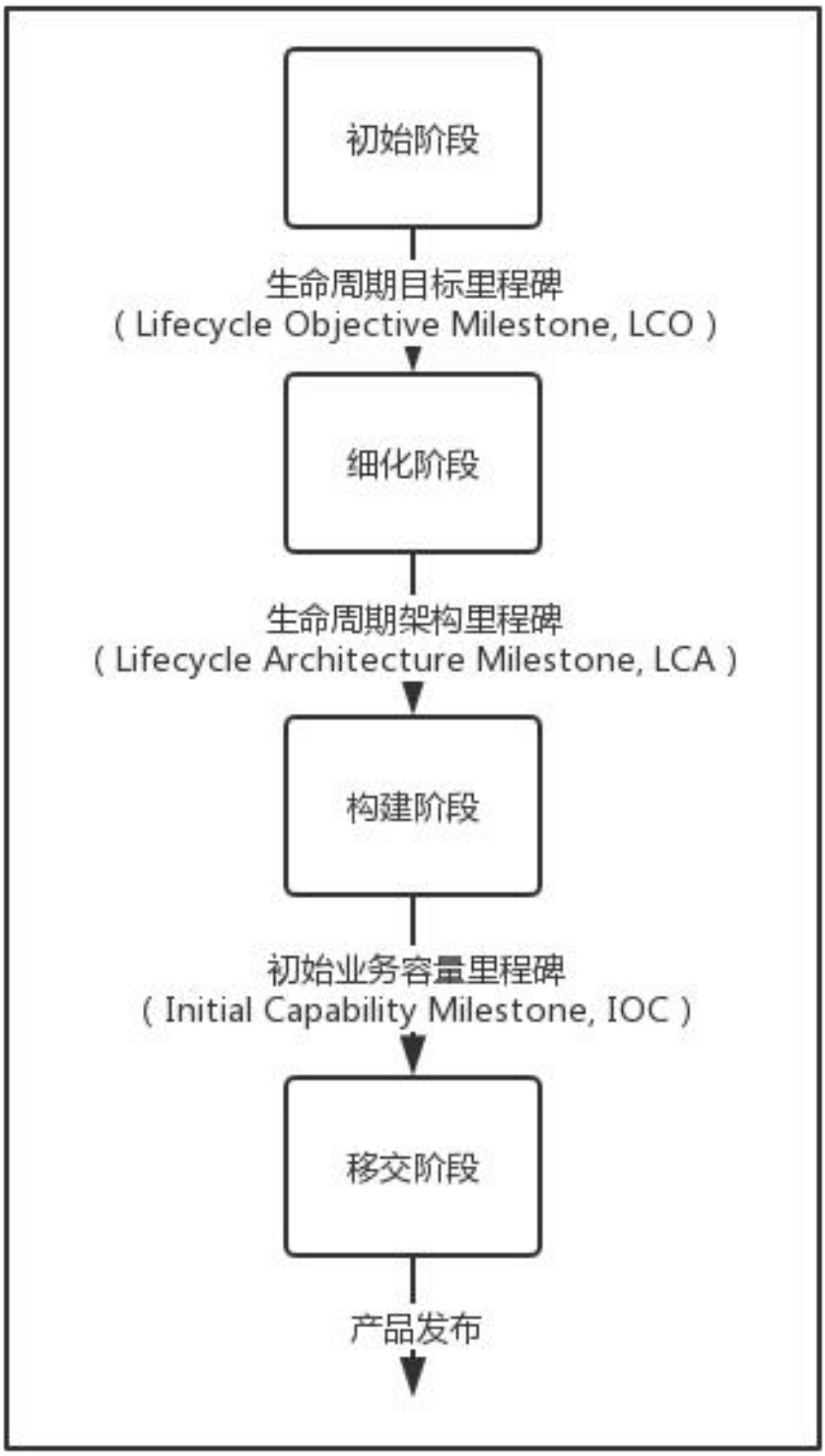
（3）构建阶段（construction）：构建产品。

（4）移交阶段（transition）：移交产品给最终用户社区。

参考：Rational 统一过程（Rational Unified Process）

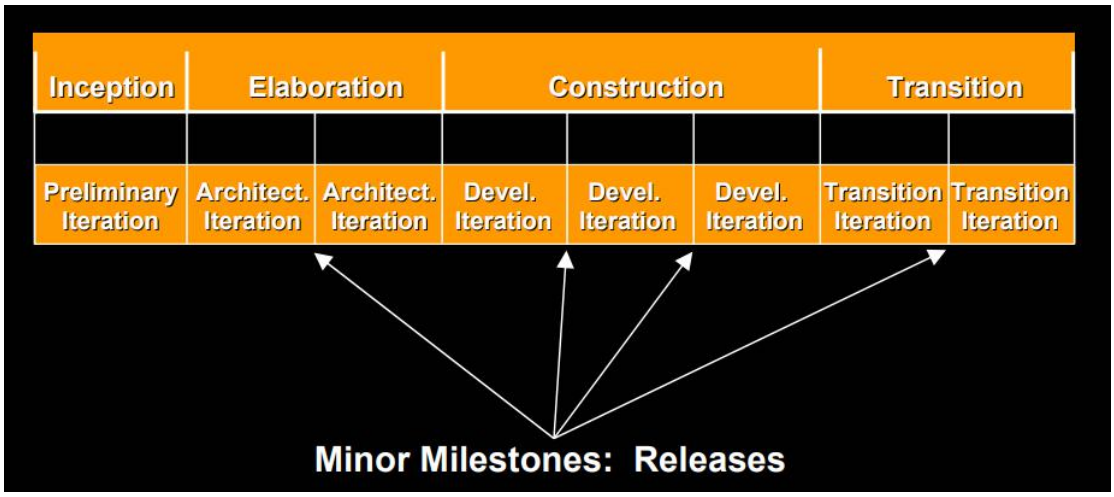
https://blog.csdn.net/u010560443/article/details/50716815

E. 阶段边界标记主要里程碑


初始阶段
生命周期目标里程碑
(Lifecycle Objective Milestone, LCO)
细化阶段
生命周期架构里程碑
(Lifecycle Architecture Milestone, LCA)
构建阶段
初始业务容量里程碑
(Initial Capability Milestone, IOC)
移交阶段
产品发布

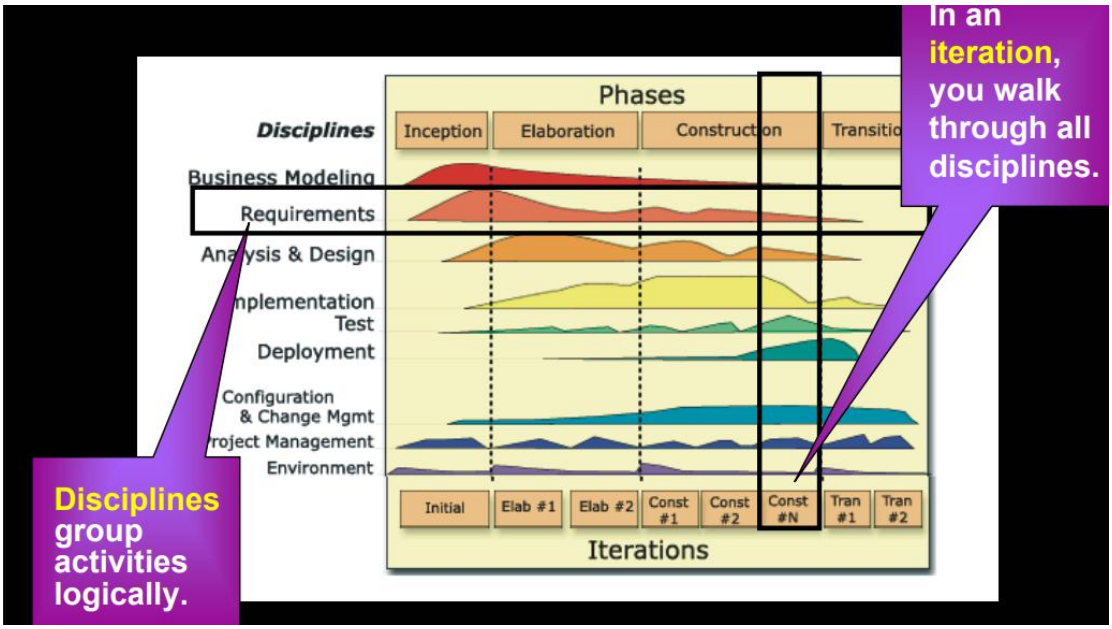
F. 迭代和阶段

一次迭代是一个清晰的活动系列，基于一个确定的计划和评估标准，最终导致一个可执行的发布（内部或外部的）。



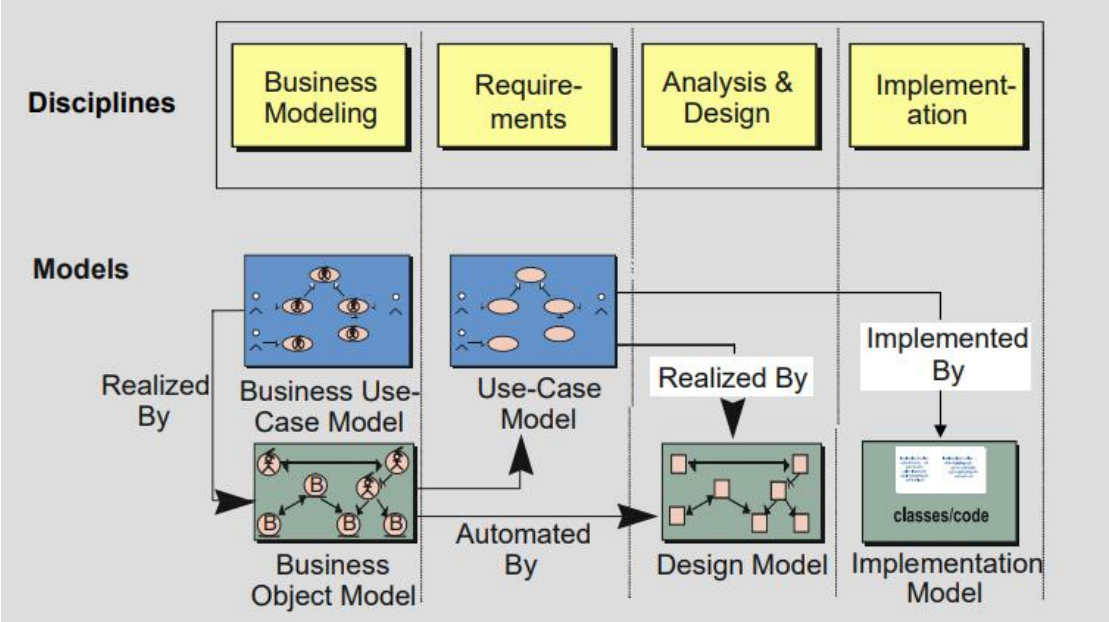
次要里程碑：发布

G. 集中在一起：迭代方法

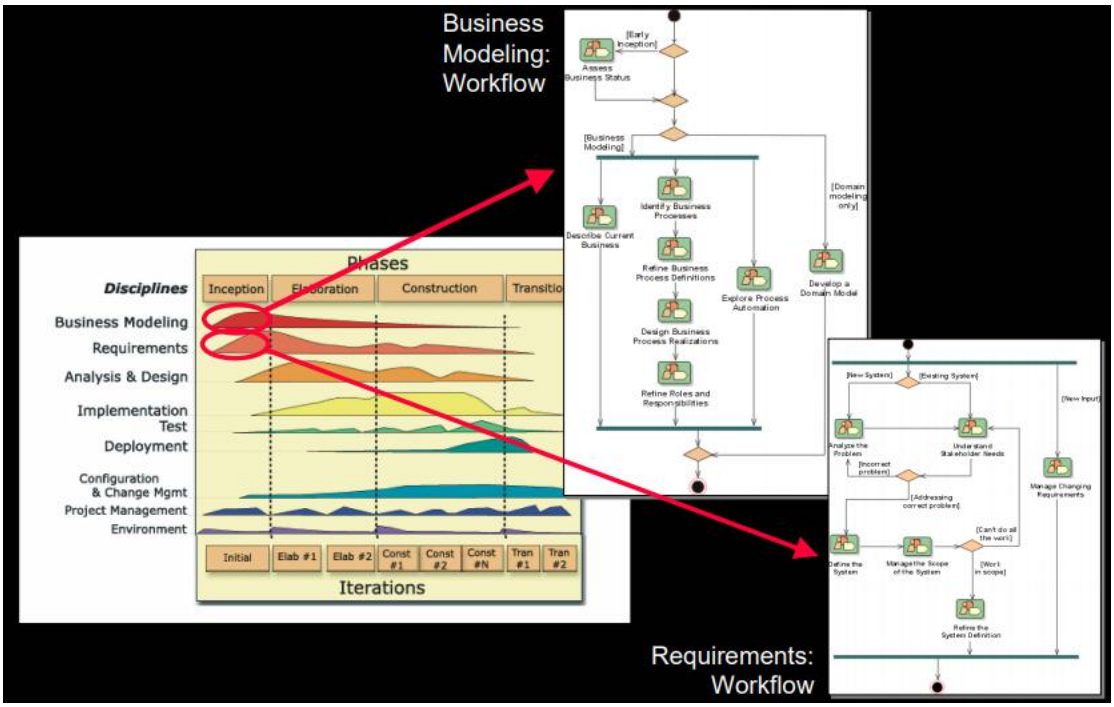


行为准则（discipline）有条理地聚集了各种活动。在一个迭代阶段，你遵守所有的行为准则。

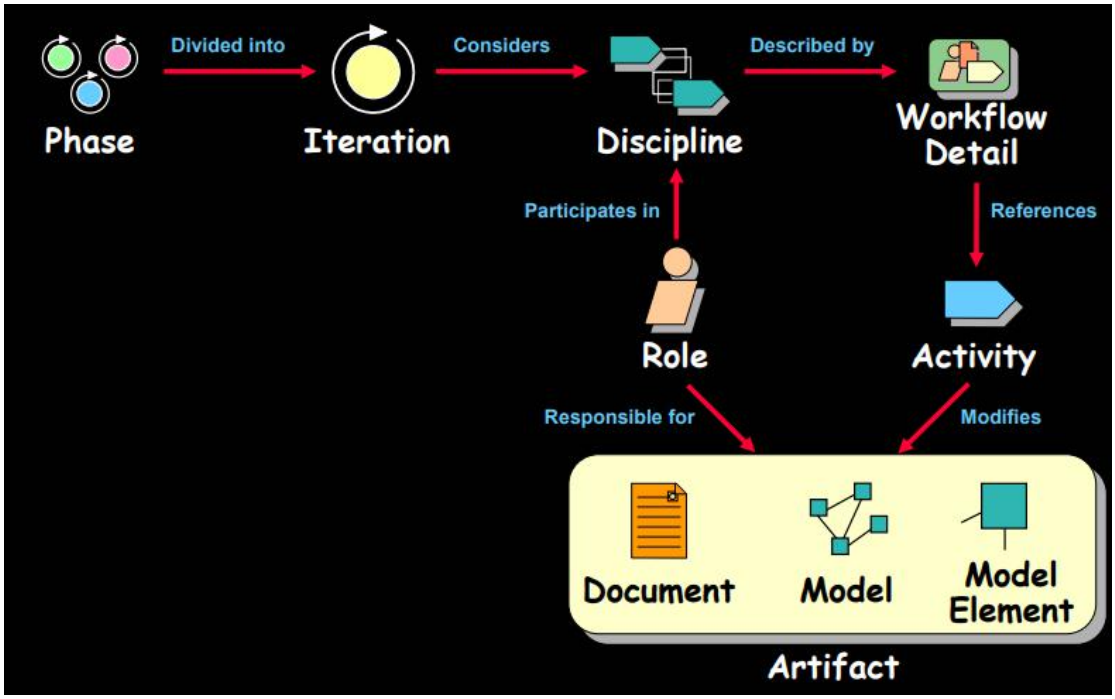
H. 行为准则生成模型



I. 行为准则指导迭代式开发



J. RUP 概念的概述



K. 小结

（1）最佳实践通过解决根本原因来指导软件工程。

（2）最佳实践帮助（reinforce）每个人。

（3）过程指导一个团队，告诉他们谁做什么，何时，如何。

（4）Rational 统一过程是一种达到最佳实践的手段。

5. 需求概述

A. 目的

（1）建立并维护客户和其他干系人（stakeholder）在系统应该干什么（功能）上的一致。

（2）让系统的开发人员更好地理解系统需求。

（3）界定（delimit）系统。

（4）为计划迭代的技术内容提供基础。

（5）为估计开发系统的成本和时间提供基础。

（6）定义系统的用户界面（interface）。

B. 相关的需求制品

（1）用例模型：用例（use case）、参与者、用例规约（use-case specification）

（2）术语表（glossary）

（3）补充规约（supplementary specification）

C. 系统行为（system behavior）

（1）系统行为是一个系统如何做事（act）和反应（react）：这是一个系统外部可见和可测试的活动。

（2）系统行为被捕获在用例中：用例描述了系统、系统的环境和系统与其环境的关系。

D. 用例建模的主要概念

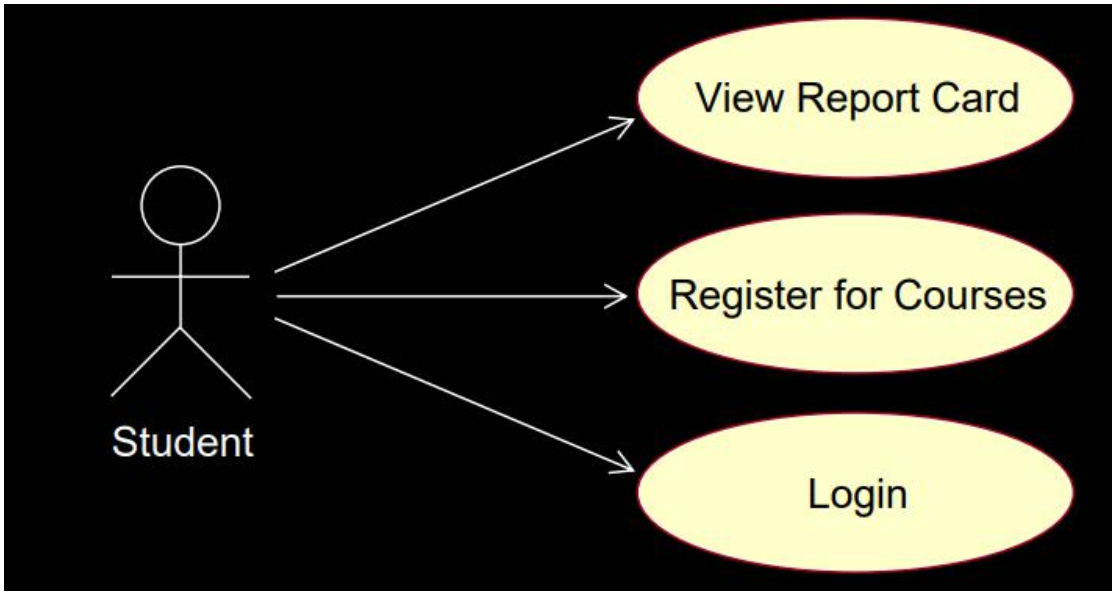
参与者表示与系统互动的所有人或物。

用例是系统做出的、会产生对特定参与者有价值的可观测结果的一系列行为。

E. 用例模型

从用例的角度看，一个描述系统功能性需求的模型。

一个关于系统的计划的功能（用例）及其环境（参与者）的模型。



用例模型的好处：沟通（对最终用户和领域专家）、标识（对用户）和认证。

F. 用例规约（use-case specifications）

名称（name）、简述（brief description）、事件流（flow of events）、关系（relationships）、活动图（activity diagrams）、用例图（use-case diagrams）、特殊需求（special requirements）、前提条件（pre-conditions）、后续条件（post-conditions）和其他视图（other diagrams）

G. 用例的事件流

有一个正常的流，被称为基本流（basic flow）。

几个备选流（alternative flows）：规则的变体（regular variants）、反常例子（odd cases）、异常流（exceptional flows，以处理错误情况）

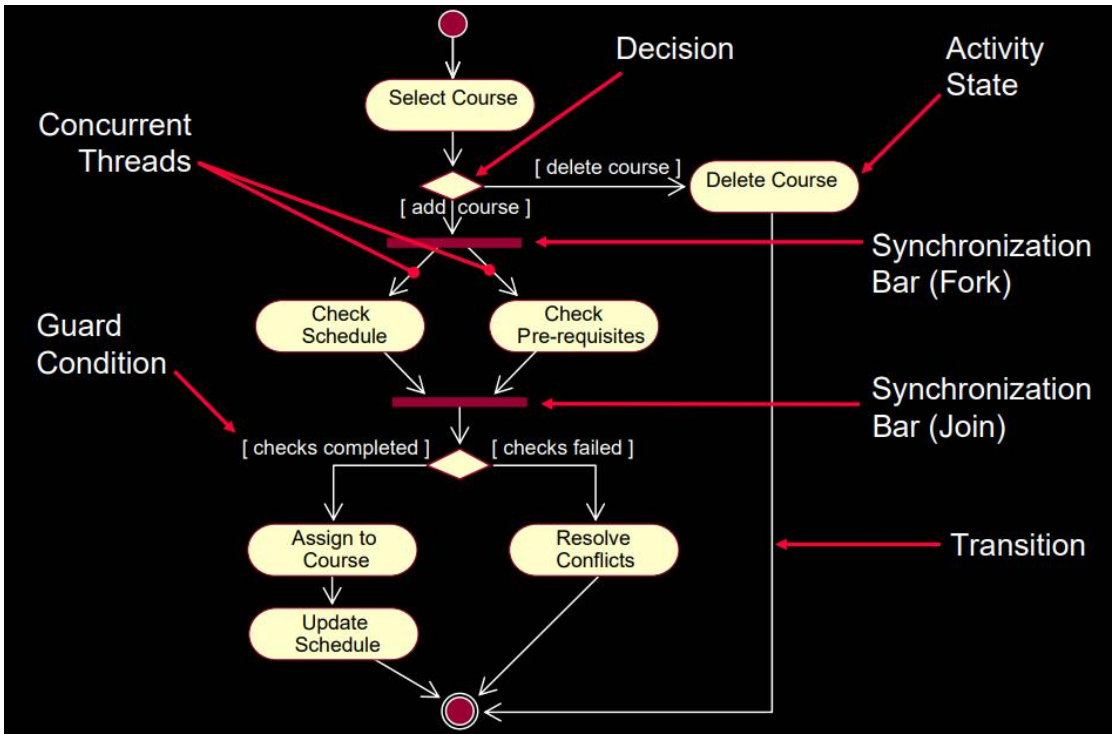
H. 场景

一个场景（scenario）是一个用例中的一个实例。

I. 活动图

在用例模型中，一个活动图（activity diagram）能被用于捕获用例中的活动。

它本质上是一个流图（flow chart），展示从活动到活动的控制流。



J. 补充规约

功能（functionality）、可用性（usability）、可靠性（reliability）、性能（performance）、支持性（supportability）和设计约束（design constraints）

6. 分析和设计

A. 概述：输入用例模型、术语表和补充规约，输出设计模型、架构文档和数据模型。

B. 分析 vs 设计

分析：重点是理解问题、理想化设计、行为、系统结构、功能性需求、一个小模型。

设计：重点是理解解决方案、操作和属性、性能、接近真实代码、对象的生命周期、非功能性需求、一个大模型。

C. 分析和设计既不是自顶向下的也不是自底向上的。

D. 架构

软件架构包括一组关于软件系统组织的关键决策（significant decisions）：

选择结构性元素及其接口来组成系统。

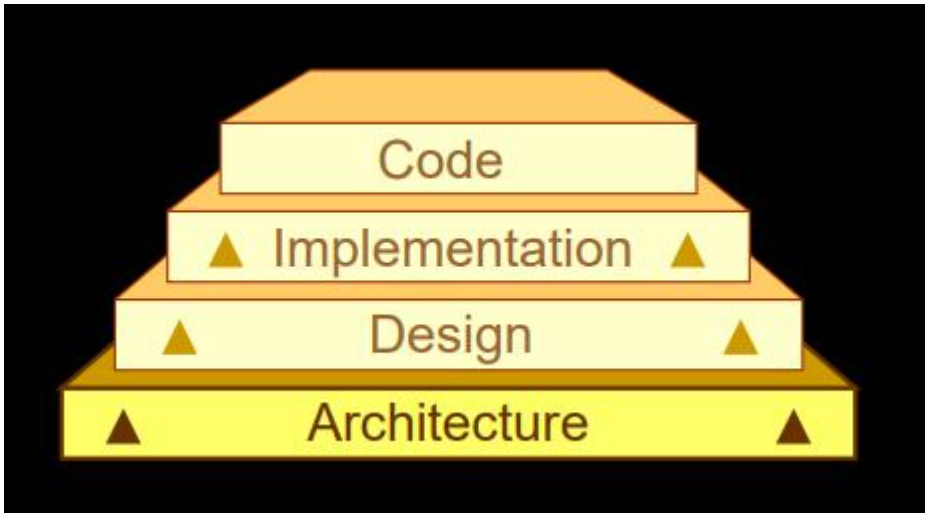
在元素间的协作中指明其行为。

这些结构和行为的元素如何组成更大的子系统。

指导软件的组织的架构风格。

E. 架构约束了设计和实现。

架构包括一组约束设计和构造的全局设计决策、规则或模式。

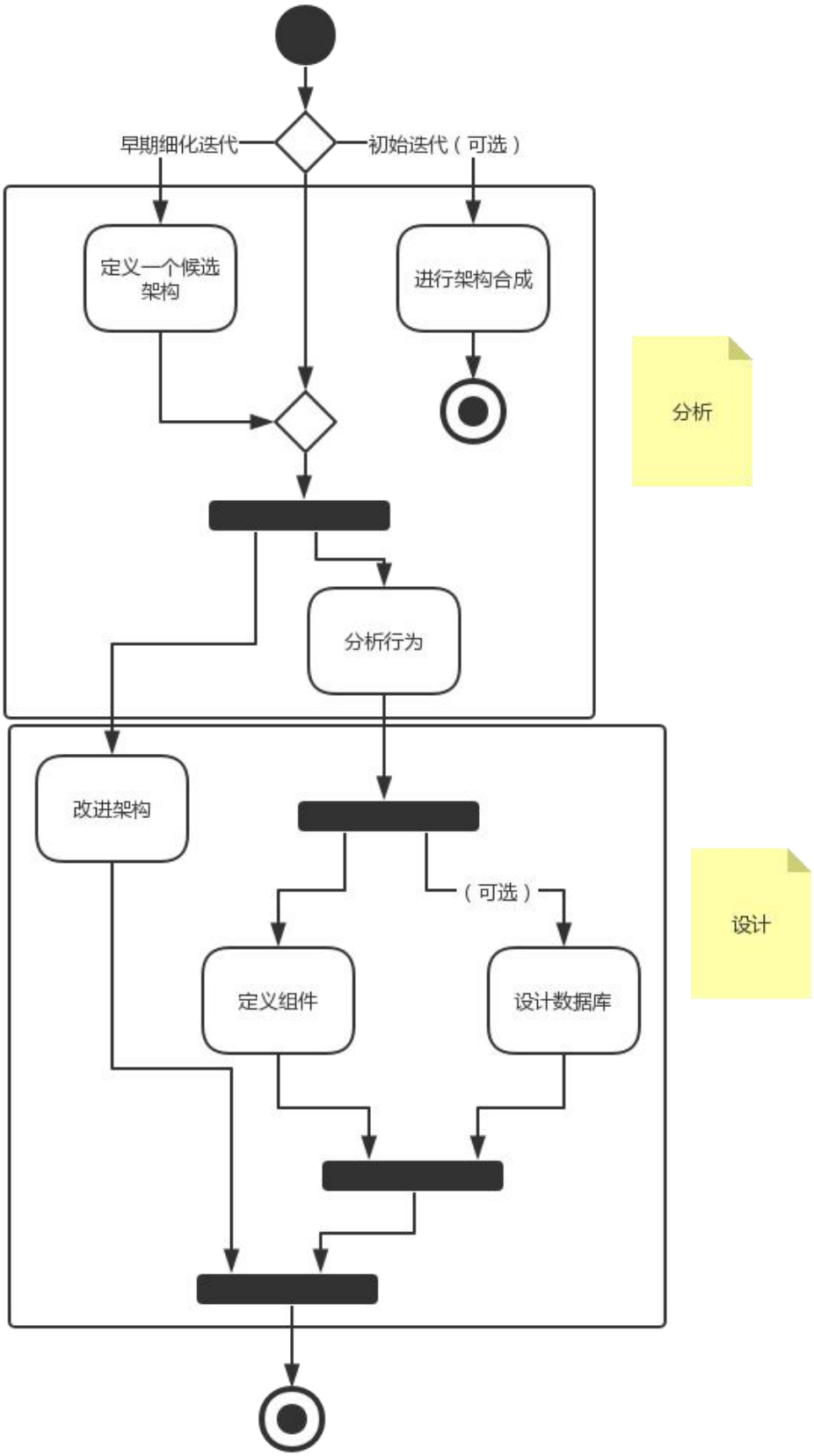


架构决策是最基础的决策，改变它们会有重大的效果。

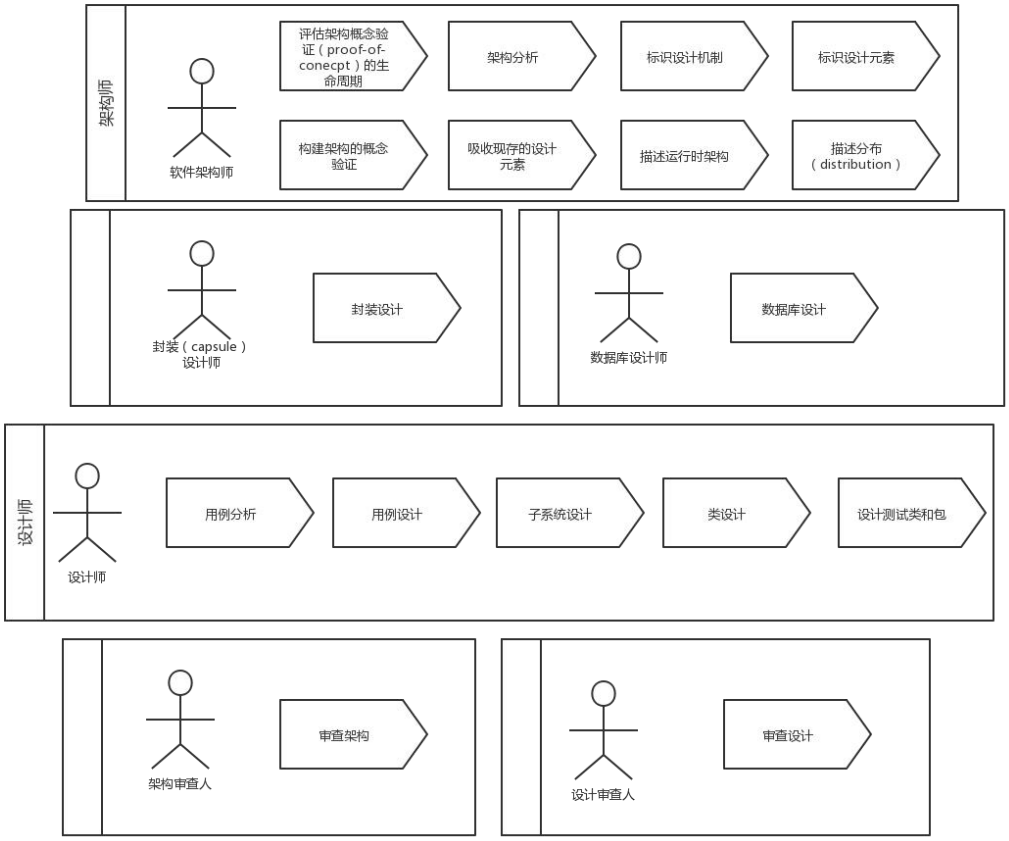
F. 软件架构：4+1 视图模型

逻辑视图、实现视图、进程视图、部署视图和用例视图

G. 分析和设计工作流


早期细化迭代
初始迭代（可选）
定义一个候选架构
进行架构合成
分析
分析行为
改进架构
（可选）
设计
定义组件
设计数据库

H. 分析和设计活动概述



参考：概念证明
https://baike.baidu.com/item/%E6%A6%82%E5%BF%B5%E8%AF%81%E6%98%8E

I. 软件架构师的职责

软件架构师领导并协调（coordinate）技术活动和制品。

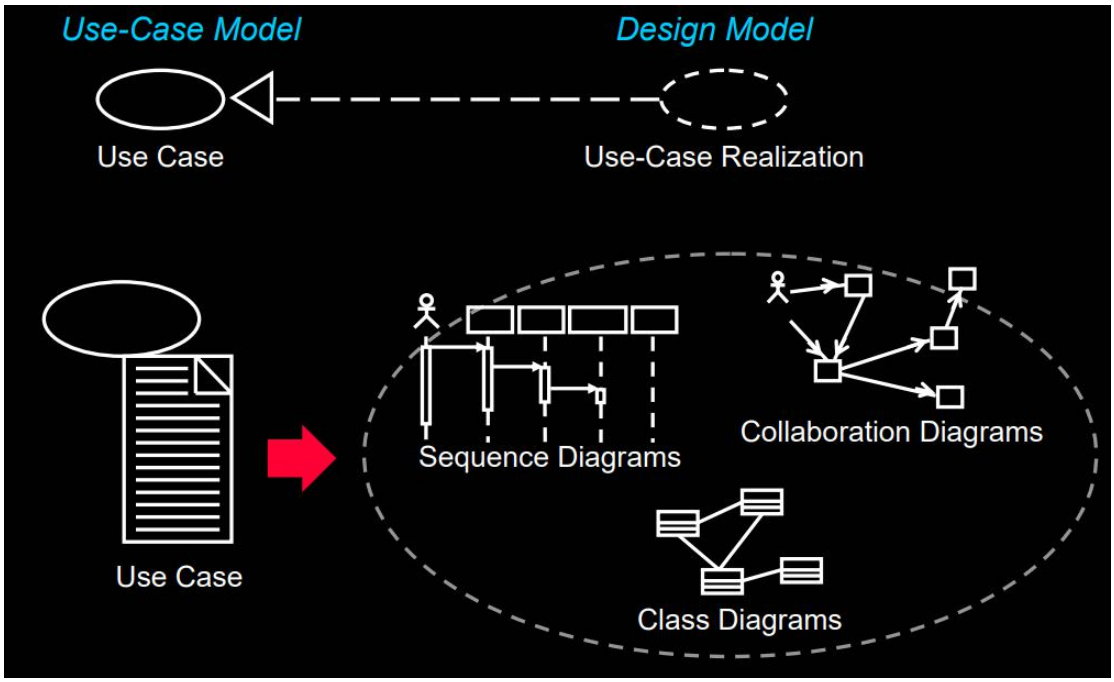
分析模型、设计模型、软件架构文档、参考架构、部署模型

J. 设计师的职责

设计师必须知道用例建模技巧、系统需求和软件设计技巧。

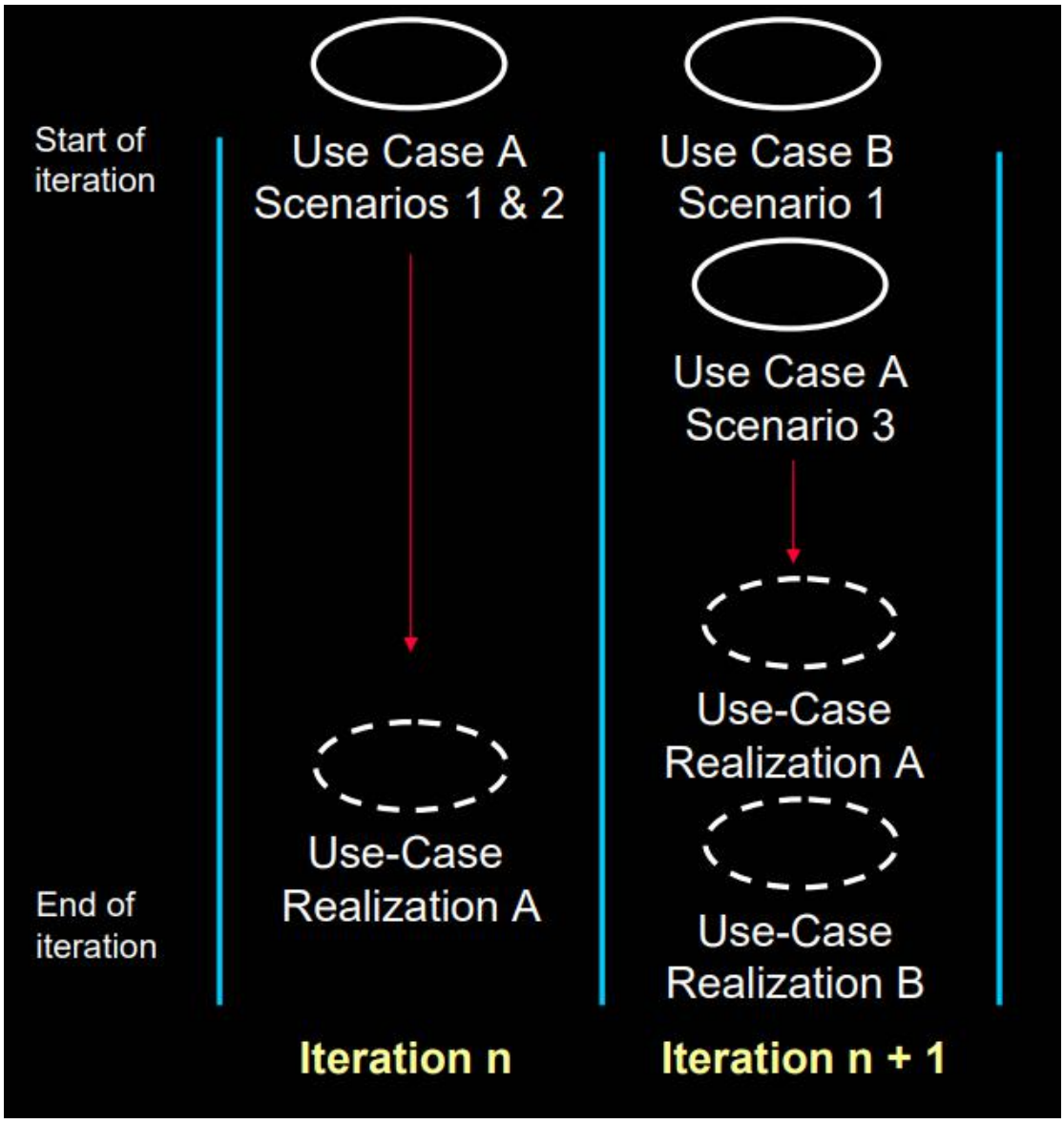
用例实现（realization）、包/子系统、类

K. 用例实现（use-case realization）



从用例到时序图、协作图和类图

L. 在迭代过程中的分析和设计



每次迭代实现更多的用例

7. 架构分析

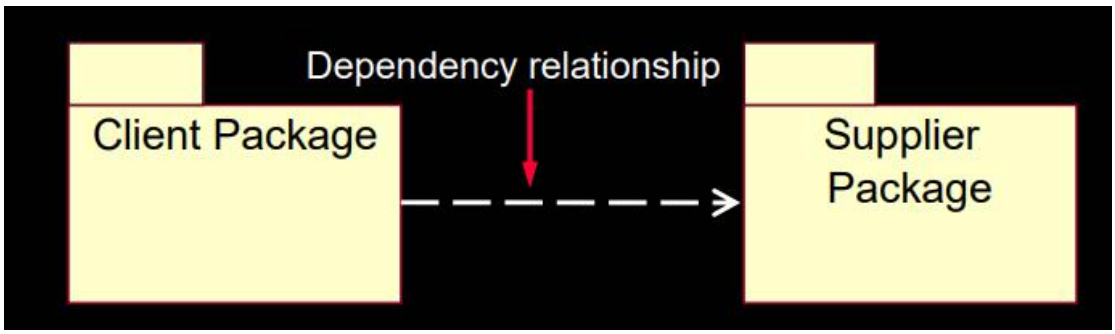
A. 架构分析概述

补充规约、术语表、软件架构文档、参考架构、愿景文档（vision document）、设计模型、部署模型、用例模型、项目细节规范（project-specific guidelines）

属于“定义候选架构”的内容。

B. 包的关系：依赖

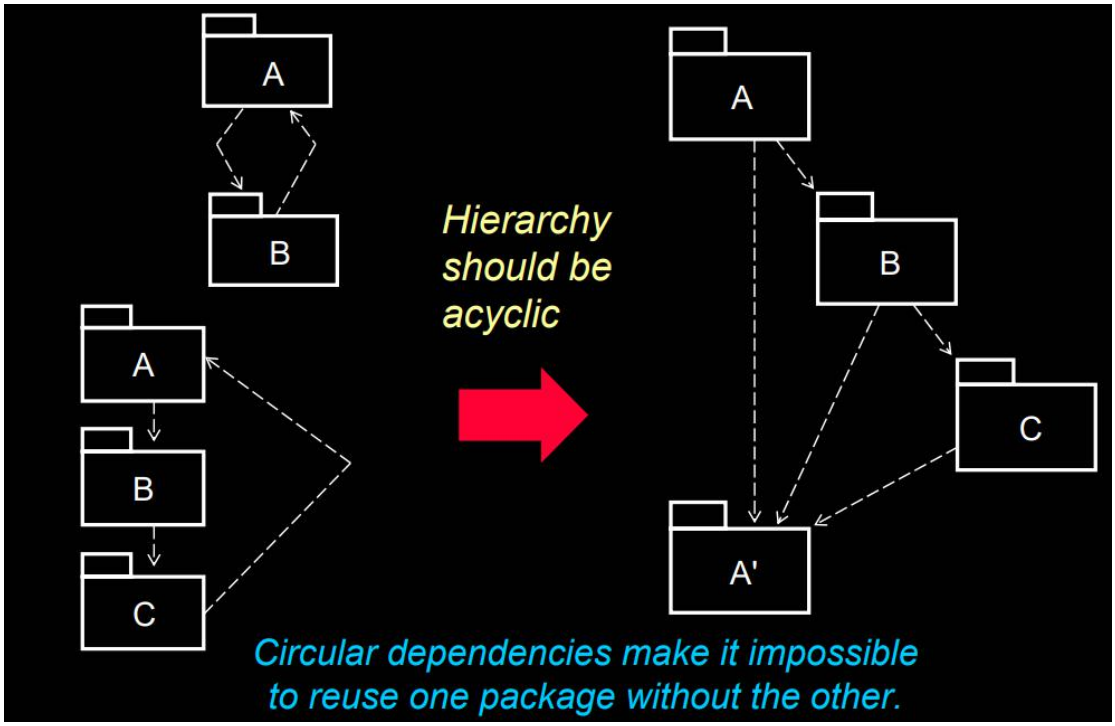
包可以与另一个包有依赖关系。



客户包、依赖关系、供应包

依赖带来的影响：改变供应包可能影响客户包；客户包不能被独立复用由于它依赖供应包。

C. 避免循环依赖



分层结构应该是无环的；循环依赖导致无法单独复用其中一个包。

D. 模式和框架

模式：为上下文中的常见问题提供一种常用的解决方案。

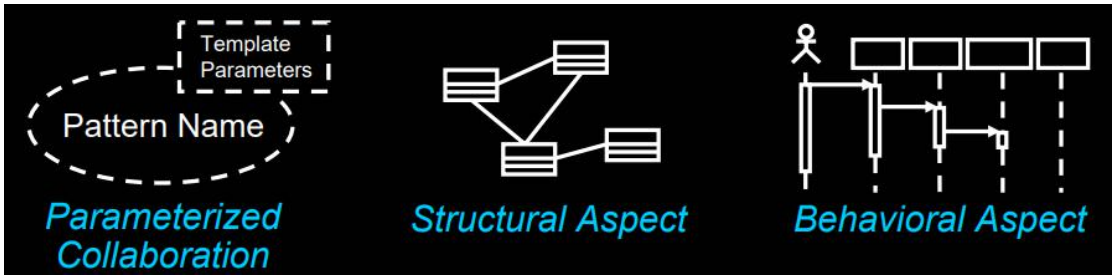
分析/设计模式：为狭窄领域内的技术问题提供解决方案；提供解决方案的一个片段或一块拼图（a piece of the puzzle）。

框架（framework）：定义通用的方法来解决问题；提供一个纲要性的解决方案，其细节可能是分析/设计模式。

E. 设计模式

一个设计模式是一个对常见设计问题的解决方案：描述一个常见的设计问题；描述问题的解决方案；讨论应用模式的结果和权衡（trade-offs）。

设计模式提供复用成功设计的能力。



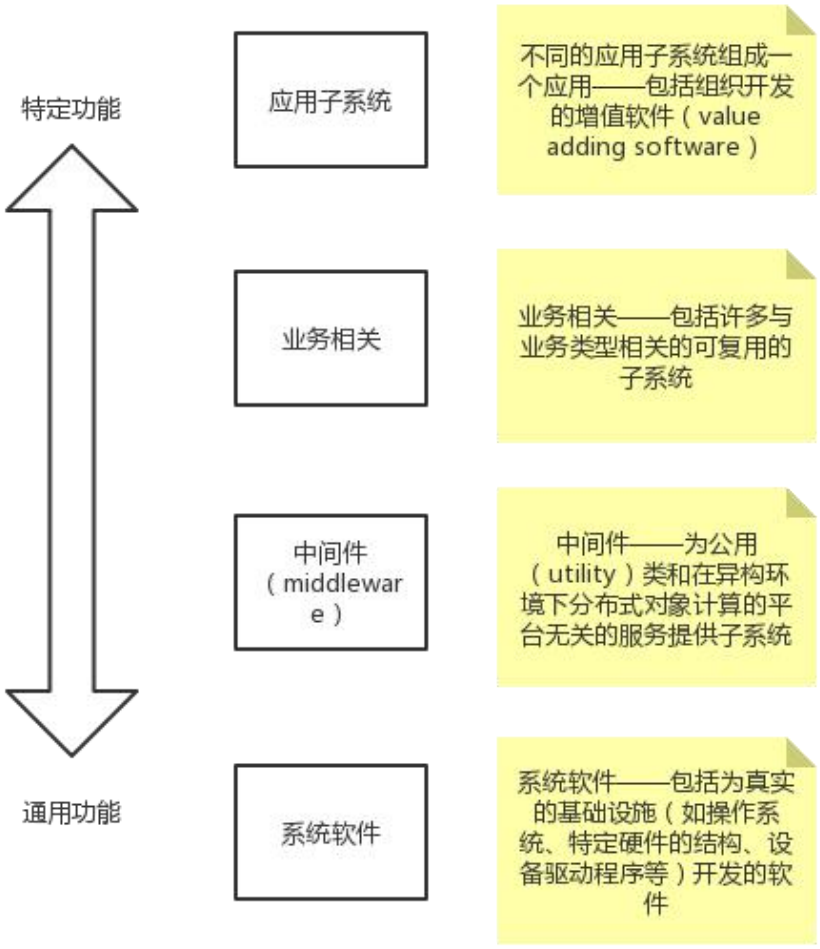
参数化协作、结构方面、行为方面

F. 架构模式

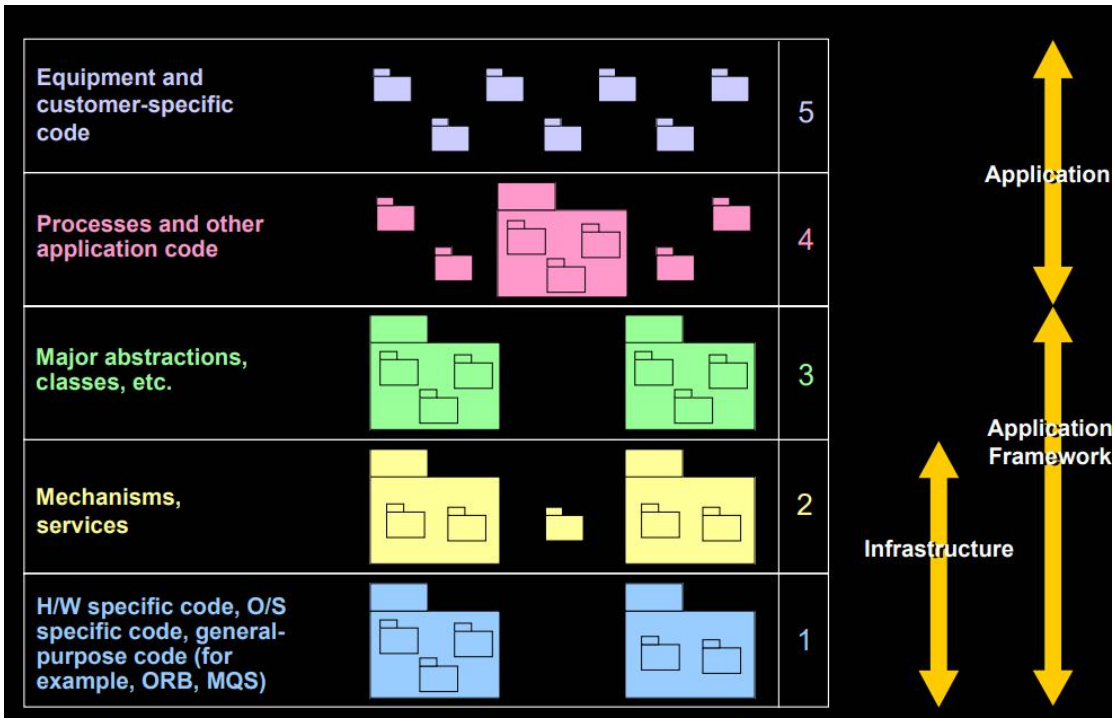
一个架构模式为软件系统表达了一个基础的结构组织纲要。它提供了一组预定义的子系统，特化它们的职责，并包含了组织它们的关系的规定和指导。

例子：分层、模型-视图-控制器（MVC）、管道和过滤器、黑板

G. 典型的分层方法



H. 架构模式：分层



分层的考虑：

（1）抽象的层次：同一组的元素在相同的抽象层次。

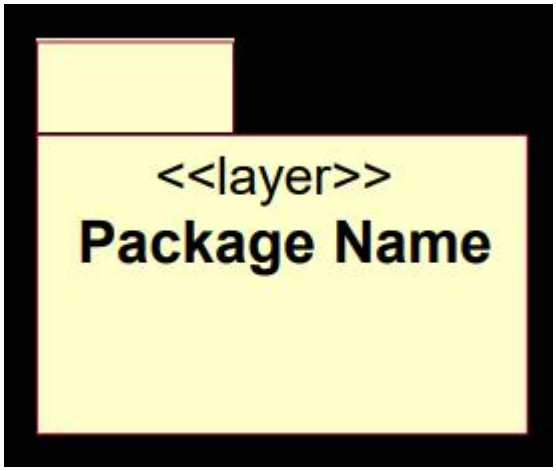
（2）影响的分离：一个组像一起的东西（group like things together）；分离不同事物；应用 vs 域模型元素。

（3）弹性（resiliency）：松散耦合（loose coupling）；聚焦于封装变化（concentrate on encapsulating change）；用户界面、业务规则和已有数据常常有很大可能会改变。

I. 架构分层建模

架构层次可以用模式化的包（stereotyped packages）来建模。

<<layer>>标记。



J. 关键抽象

一个关键抽象（key abstraction）是一个概念，正常情况下在需求中被提出，必须

能被系统处理。

关键抽象的来源：领域知识、需求、术语表、域模型或业务模型（如有）。

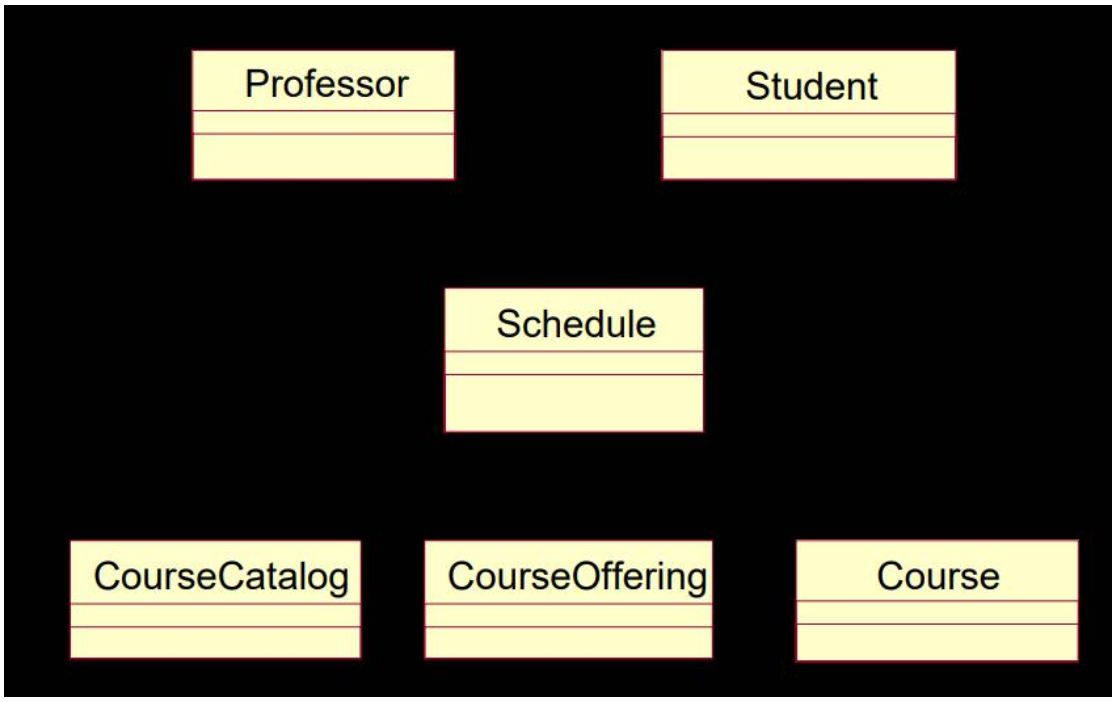
K. 定义关键抽象

定义分析类的关系。

在类图中为分析类和关系建模：包括分析类的简述。

映射分析类到必须的分析机制上。

例子：



参考：【建模】分析类 https://blog.csdn.net/weixin_43800786/article/details/86484179

L. 用例实现的价值

提供从分析和设计到需求的可追踪性。

架构师创建用例实现。

8. 用例分析

A. 用例分析概述

术语表、项目细节规范、软件架构文档、用例实现、分析模型、分析类、用例模型、补充规约

属于“行为分析”的内容。

B. 用例分析的步骤

（1）补充用例描述。

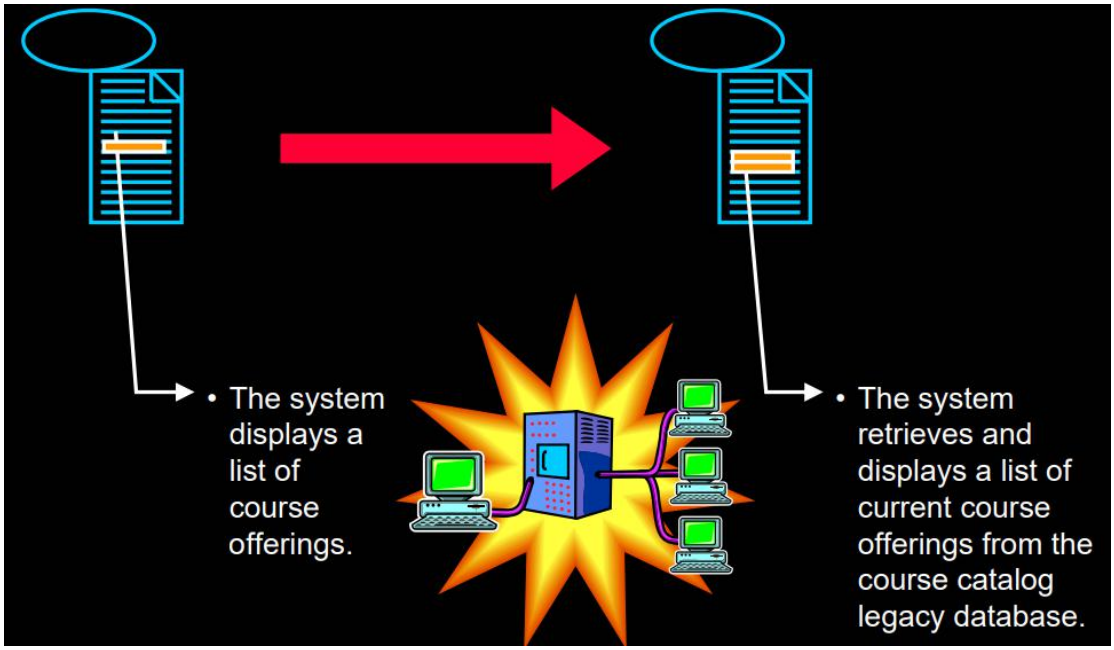
（2）对每个用例实现：从用例行为中发现类；把用例行为分配给类。

（3）对于每个产生的分析类：描述职责；描述属性和关联（associations）；使分析机制合格（qualify analysis mechanism）。

（4）统一分析类。

C. 补充用例描述

例子：

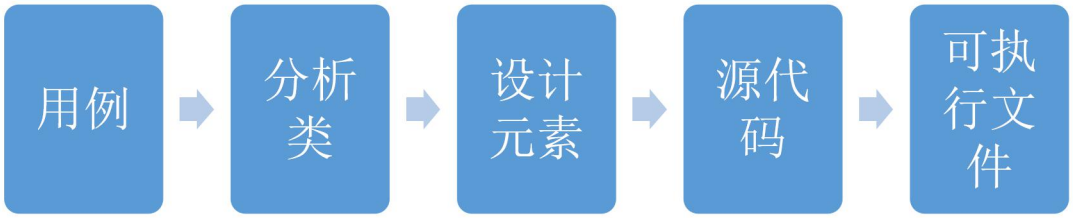


系统显示待选课程列表。

系统从剩余课程目录数据库中检索并显示当前待选课程的列表。

D. 从用例行为中发现类

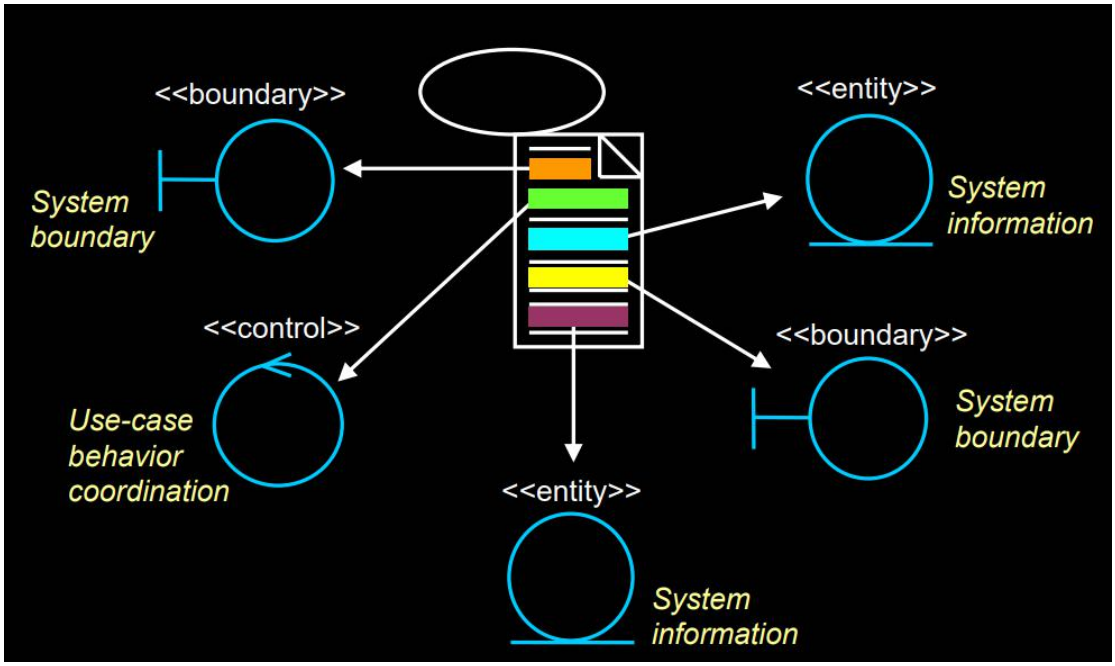
（1）用例分析：生成可执行文件的第一步



用例分析：从用例到分析类

（2）一个用例的完整行为必须被分配到分析类。

（3）分析类



实体类（entity class）、边界类（boundary class）和控制类（control class）

（4）边界类

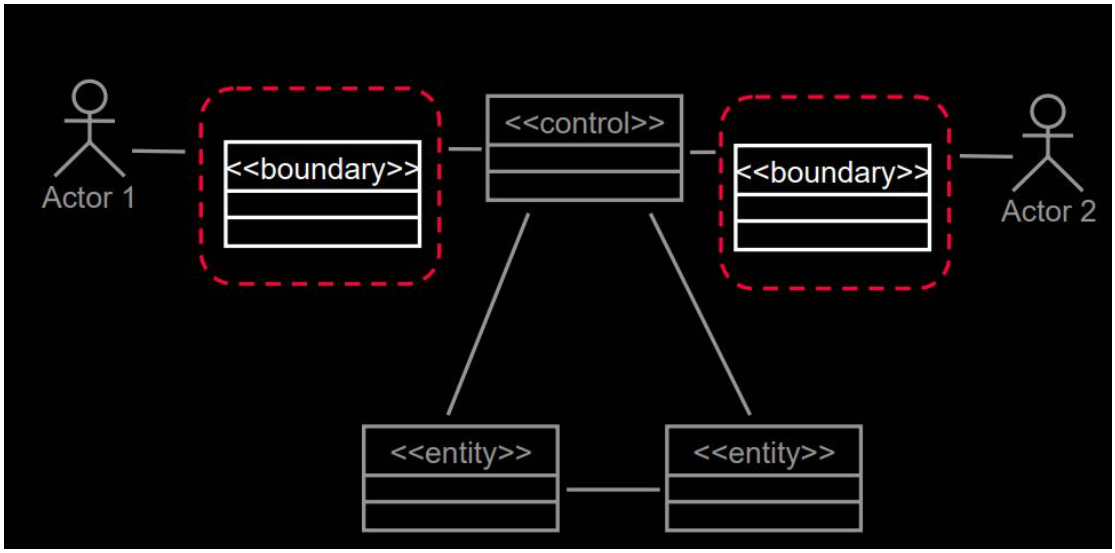
接口和系统外的事物之间的中介。

边界类的类型：用户接口类、系统接口类、设备接口类。

每个参与者/用例对有一个边界类。

边界类是环境相关的。

边界类的角色：将系统及其环境的互动建模。



边界类的准则：

用户接口类：聚焦于展示给用户的信息；请勿聚焦于用户界面的细节。

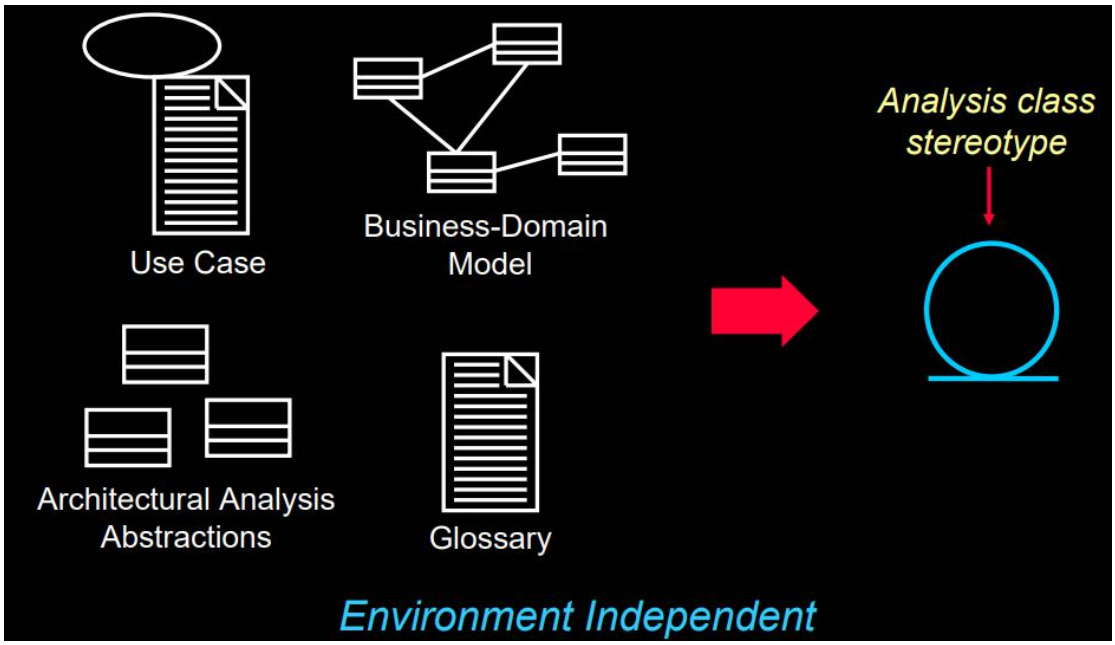
系统和设备的接口类：聚焦于必须定义什么协议；请勿聚焦于如何实现协议。

总之，聚焦于职责而非细节。

参考：边界类 https://baike.baidu.com/item/%E8%BE%B9%E7%95%8C%E7%B1%BB

（5）实体类

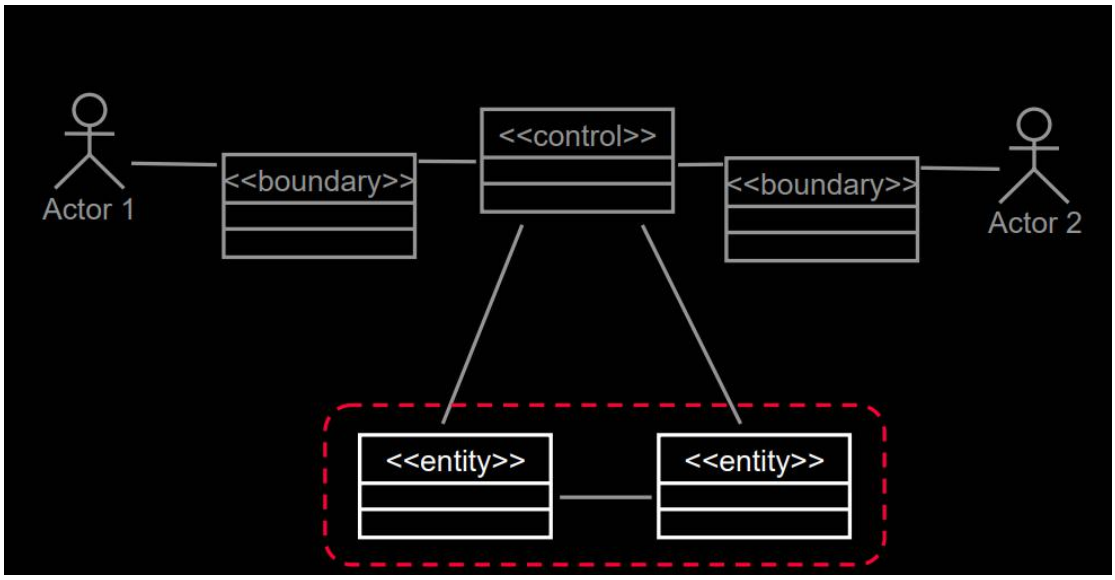
实体类是系统的关键抽象。



从用例、业务-领域模型、架构分析抽象、术语表到分析类

实体类是环境无关的。

实体类的角色：



储存并管理系统内的信息。

找出实体类：

使用用例的事件流作为输入。

用例的关键抽象。

传统方法：过滤名词

A 在用例的事件流中标出名词。

B 移除多余的候选。

Γ 移除含糊的候选。
Δ 移除参与者（范围外的）。
E 移除实现的概念。
Z 移除属性（但为以后保存）。
H 移除操作。

参考：实体类 https://baike.baidu.com/item/%E5%AE%9E%E4%BD%93%E7%B1%BB

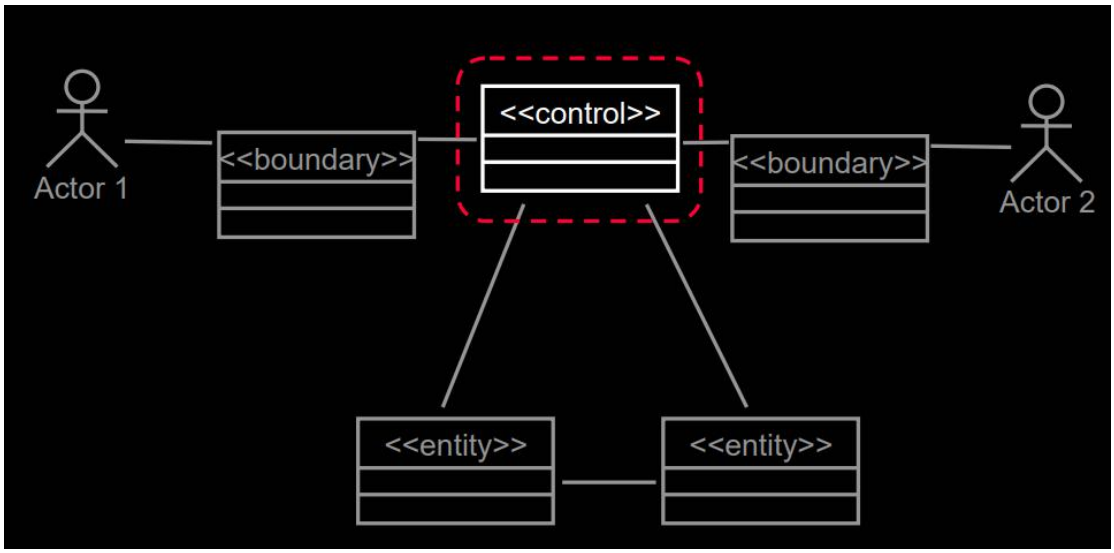
（6）控制类

控制类是用例行为的协调者：更复杂的用例一般需要一个或多个控制类。

从用例到分析类

控制类与用例有关，与环境无关。

控制类的角色：



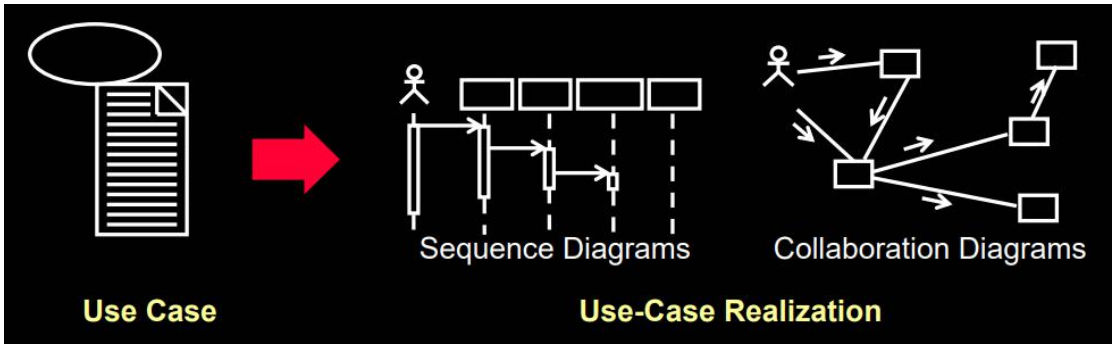
协调用例的行为

找出控制类：一般地，给那个用例标识一个控制类。（随着分析的继续，一个复杂用例可能演化出多个控制类）

参考：控制类 https://baike.baidu.com/item/%E6%8E%A7%E5%88%B6%E7%B1%BB

E. 把用例行为分配给类

对每个用例的事件流：标识分析类；分配用例职责给分析类；在互动图中为分析类的互动建模。



准则：给类分配职责

（1）使用分析类标记作为规则：

边界类：与跟参与者沟通有关的行为。

实体类：与抽象中数据封装有关的行为。

控制类：与一个用例或一个非常重要的事件流的部分相关的行为。

（2）是否是拥有数据的类负责相关的职责：

如果一个类有数据，就负责相关的职责。

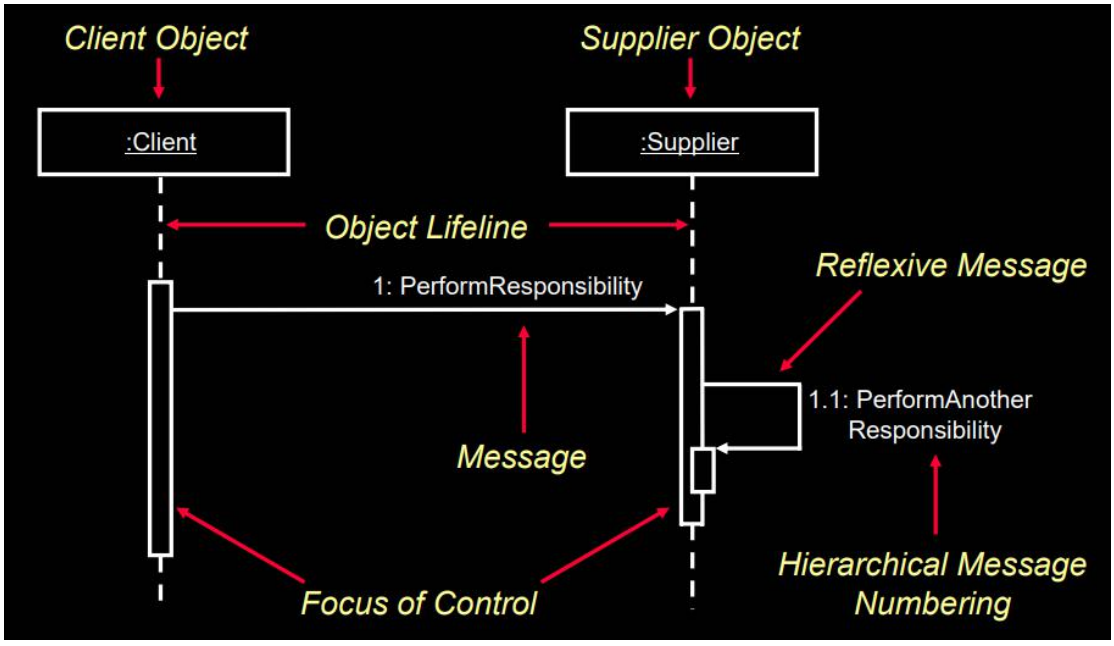
如果多个类有数据：

其中一个类负责，并与其他类产生关系。

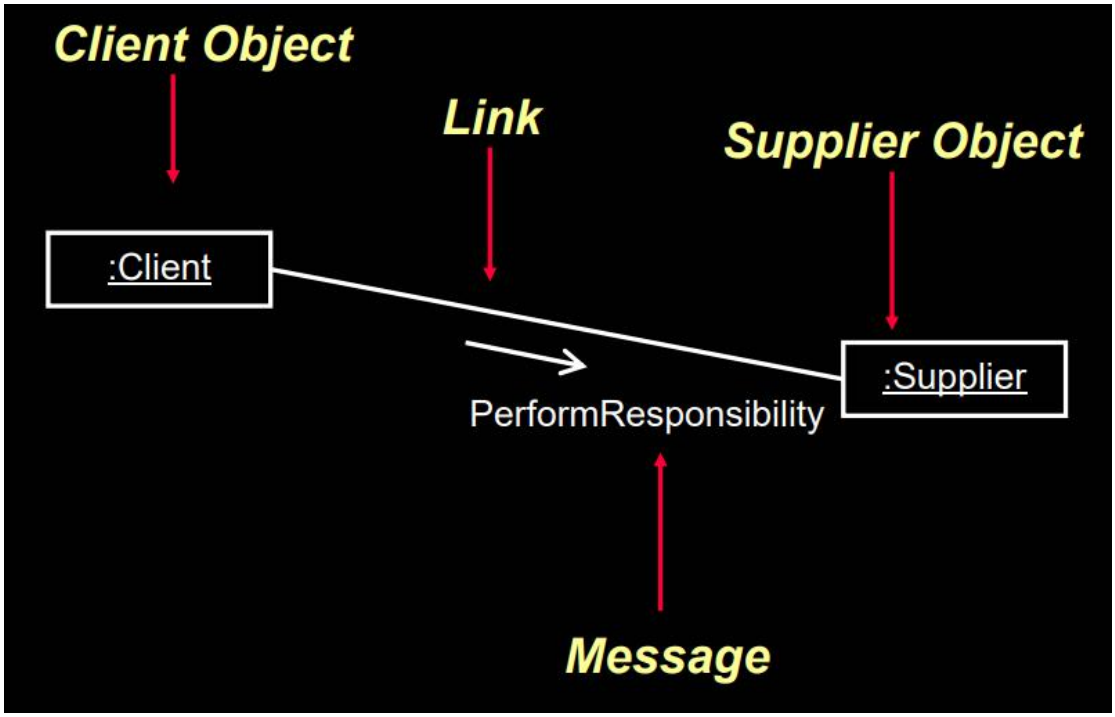
创建一个新类来负责，并与需要运行职责的类产生关系。

把职责放在控制类上，并与需要运行职责的类产生关系。

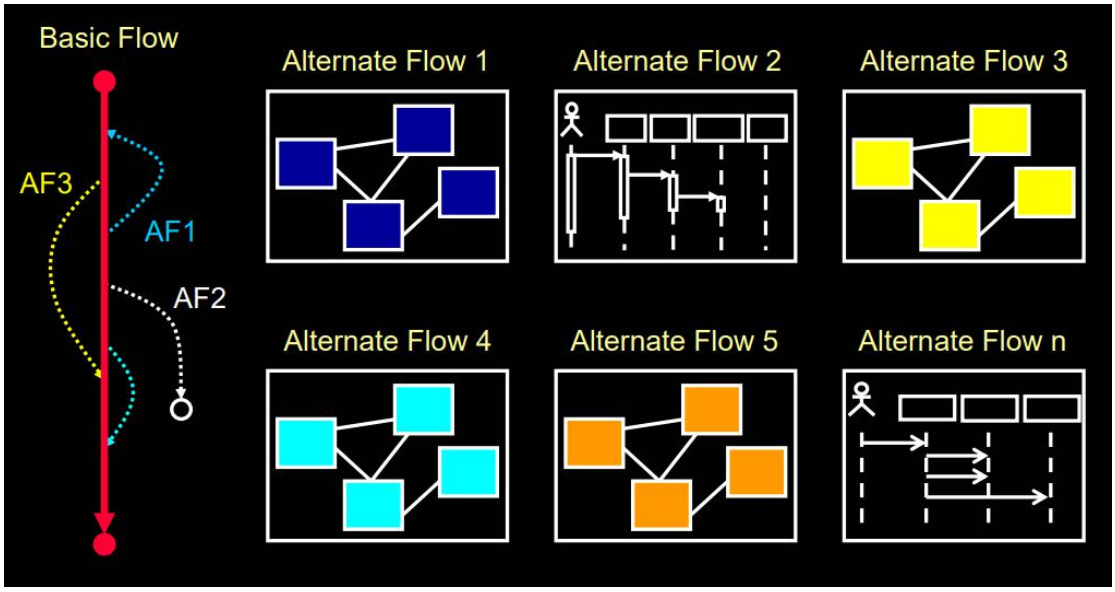
对时序图的解析：



对协作图的解析：



一张互动图不够好。



协作图 vs 时序图：

协作图：

（1）展示了除互动外的关系。

（2）更有利于协作的可视化模式。

（3）更有利于将对一个给定对象的效果可视化。

（4）更易于用作头脑风暴的会话。

时序图：

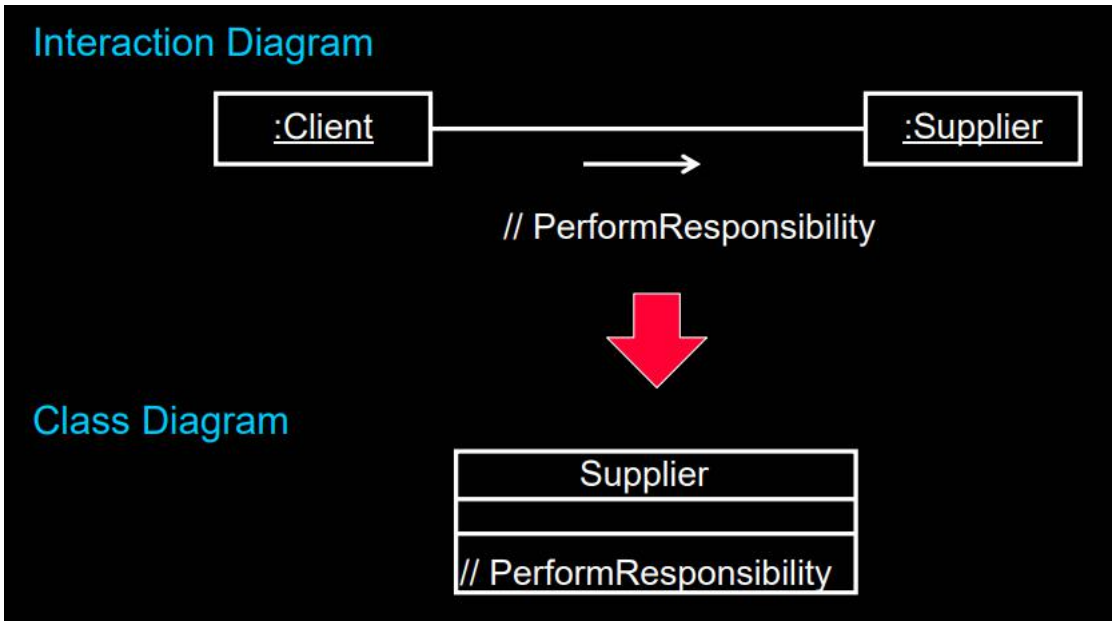
（1）展示了清楚的消息时序。

（2）有利于可视化整个流。

（3）有利于实时的规约和复杂场景。

F. 描述职责

（1）职责



（2）维持一致性

按临界性（criticality）的顺序：

类中多余的职责

类中脱节的职责

只有一个职责的类

没有职责的类

更好的行为分配

与许多其他类互动

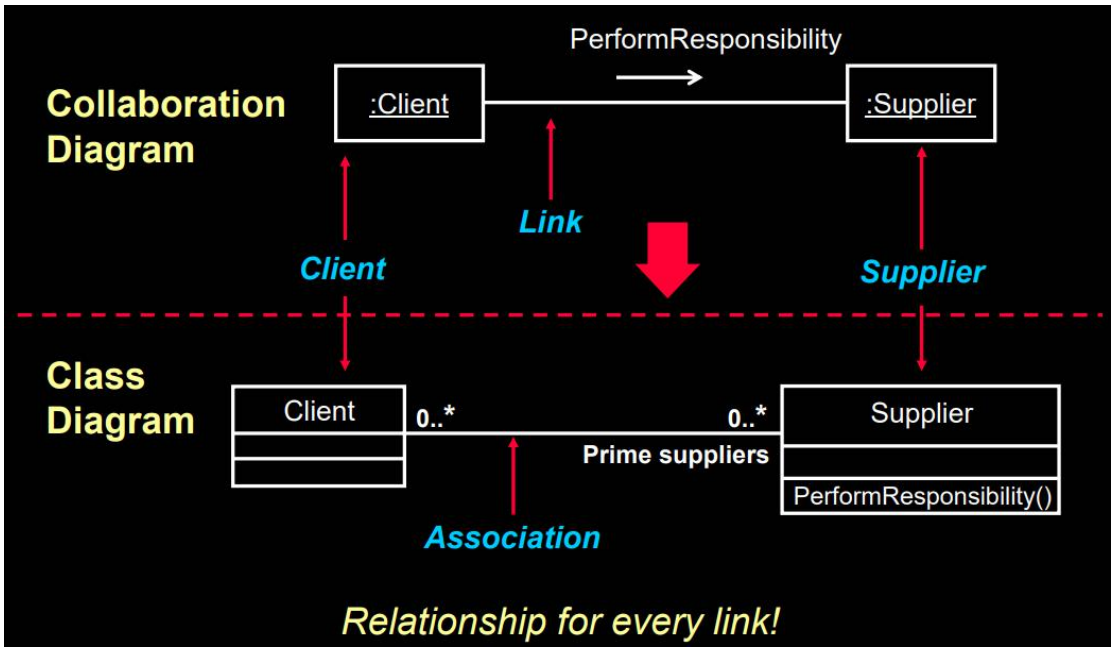
G. 描述属性和关联

（1）找出属性（attribute）

标识类的属性（property）/特性（characteristic）

被标识类保存的信息

没有变成类的名词：重要信息、一个对象独有的信息、没有行为的信息



（2）关联或聚集？

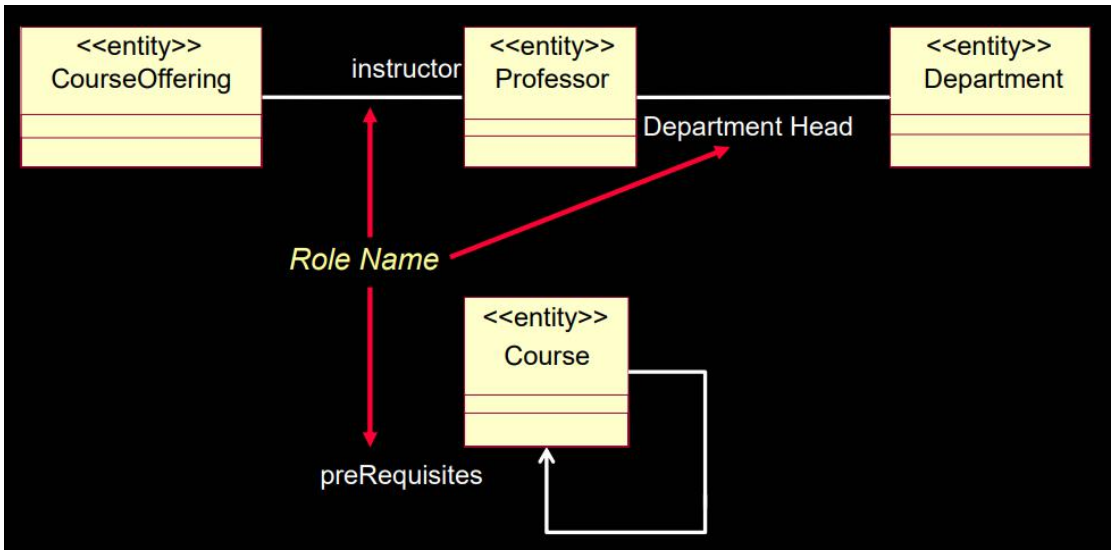
聚集：两个对象被整体-局部的关系紧密结合。

关联：两个对象通常被认为是独立的，即使它们通常是相关的。

当不确定时，使用关联。

（3）角色（role）

一个类在关联中扮演的“面孔”（face）。

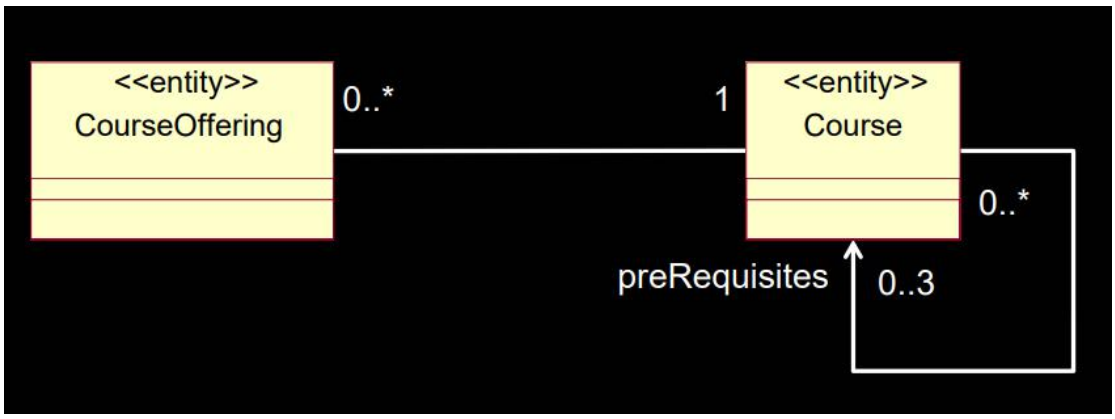


（4）多重性（multiplicity）

多重性回答以下两个问题：

关联时强制的还是可选的？

被连接的实例数的最小值和最大值？



参考：一起来学 UML（4）——类图中的多重性（Multiplicity）
https://blog.csdn.net/viggirl/article/details/8587721

H. 使分析机制合格

（1）分析机制（analysis mechanism）是分析时通过给设计师提供一个复杂行为的速记表示法来减少分析复杂度并改善其一致性的方法。

（2）描述分析机制

将所有分析机制收集在一个清单里。

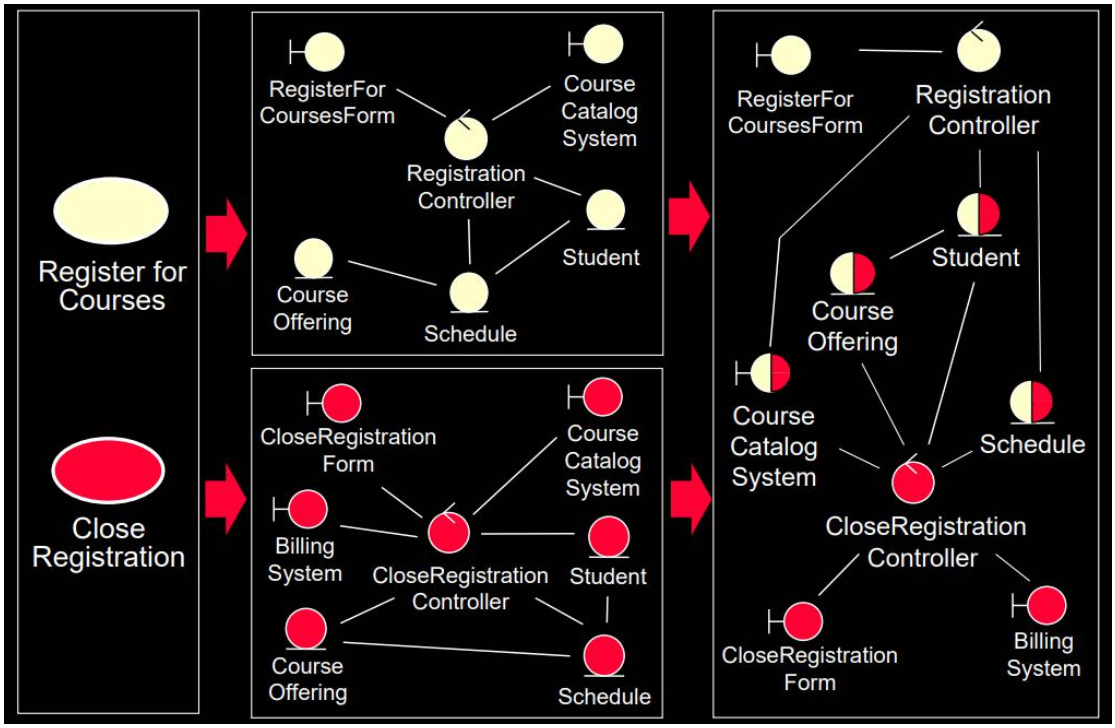
把一个客户类的图画到分析机制里面。

标识分析机制的特性。

例子：

Analysis Class	Analysis Mechanism(s)
Student	Persistency, Security
Schedule	Persistency, Security
CourseOffering	Persistency, Legacy Interface
Course	Persistency, Legacy Interface
RegistrationController	Distribution

I. 统一分析类

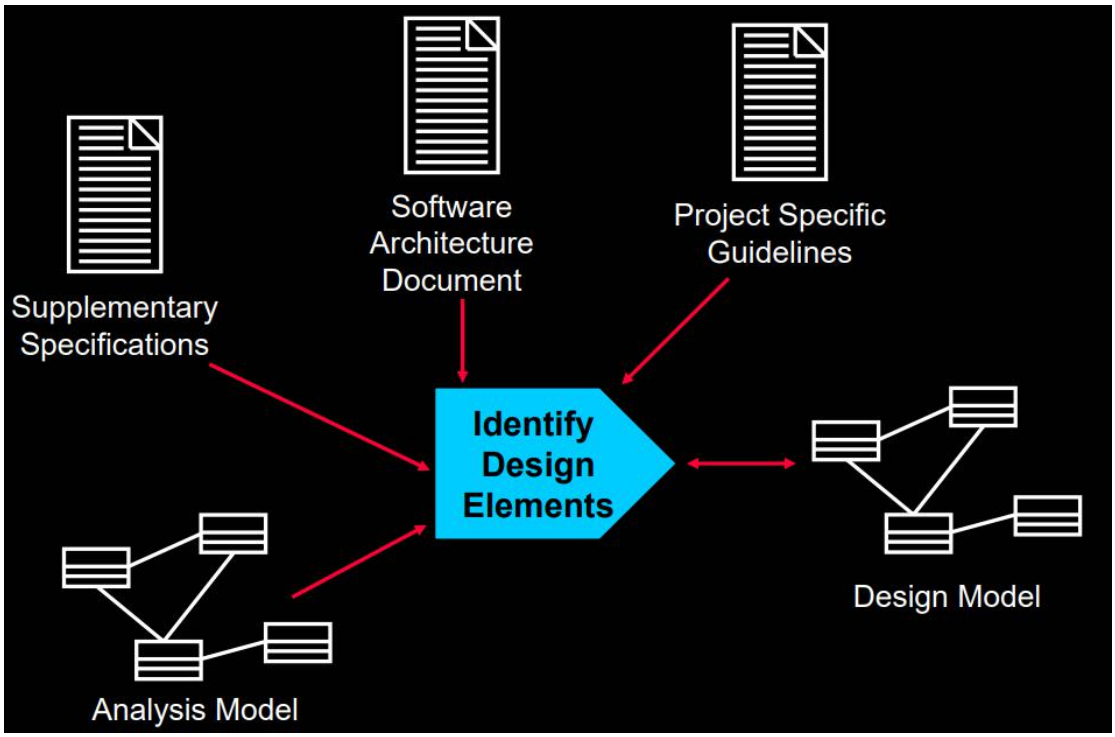


评估结果：从设计模型和分析类到术语表、补充规约和用例模型

9. 标识设计元素

A. 标识设计元素概述

补充规约、软件架构文档、项目相关准则、设计模型、分析模型



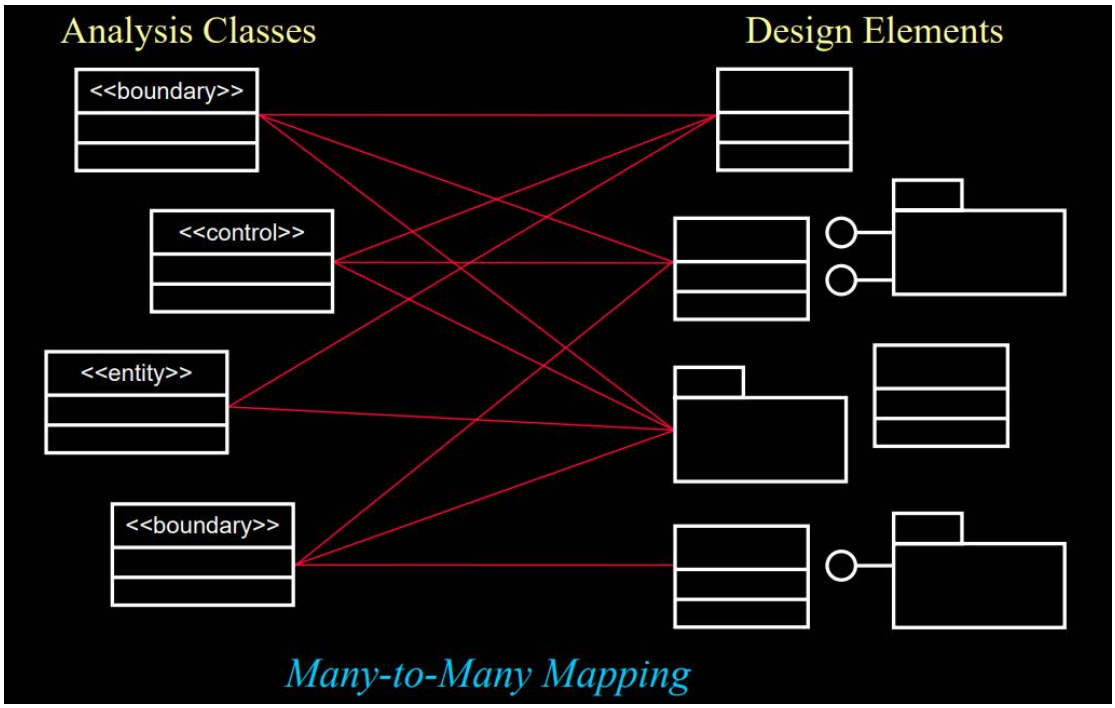
属于改善架构。

B. 标识设计元素的步骤

（1）标识类和子系统。
（2）标识子系统接口。
（3）更新设计模型的组织。

标识类和子系统：

C. 从分析类到设计元素



多对多映射

D. 标识设计类

（1）若有以下情况，一个分析类直接映射到设计类：

它是简单类。

它代表一个逻辑抽象。

（2）更复杂的分析类可能：

分成多个类。

变成一个包。

变成一个子系统（稍后讨论）。

以上的任意组合。

E. 在包中将设计类分组

你可以定义自己的包装标准，基于许多不同的因素，包括：

配置单元

开发团队的资源分配

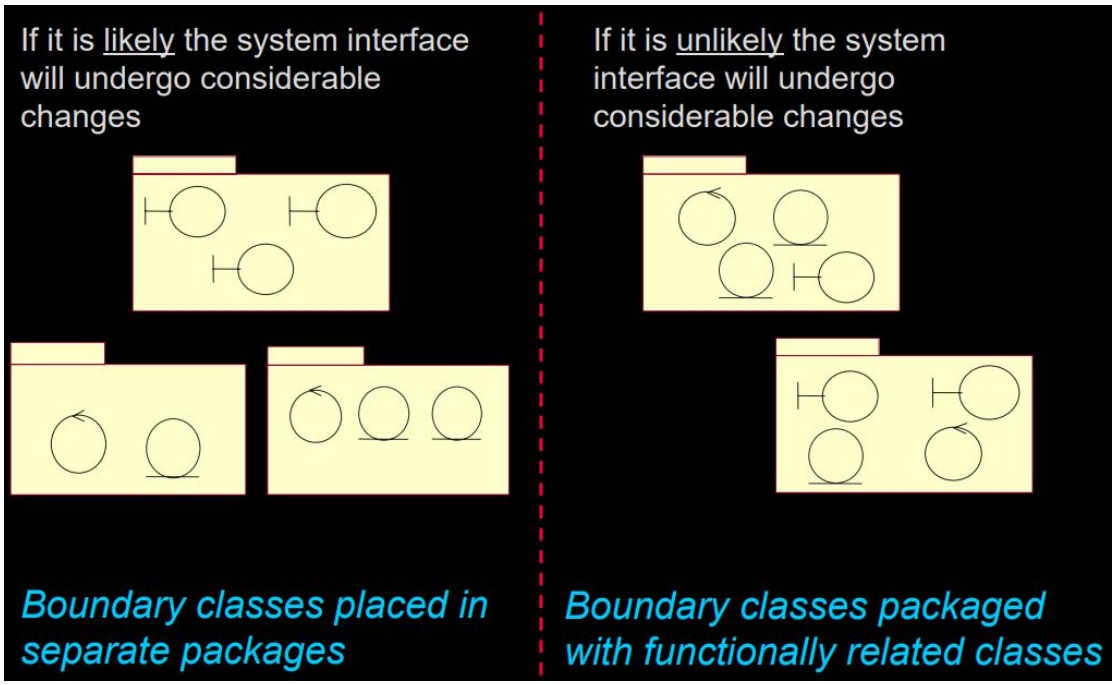
反映用户的类型

代表系统使用的现存的产品和服务

F. 包装提示：边界类

（1）若系统接口很可能经历意料之内的改变，则边界类应放在分开的包。

（2）若系统接口不太可能经历意料之内的改变，则边界类应该和功能相关的类放在同一个包中。



G. 包装提示：功能相关的类

决定类是否功能相关的标准：

（1）改变一个类的行为和/或结构必然会改变另一个类。

（2）移除一个类会影响到其他类。

（3）两个对象交换许多消息或有复杂的相互沟通。

（4）一个边界类可能与一个特定的实体类功能相关，如果边界类的功能会作用在这个实体类上。

（5）两个类互动，或被相同参与者的改变影响。

（6）两个类相互具有关系。

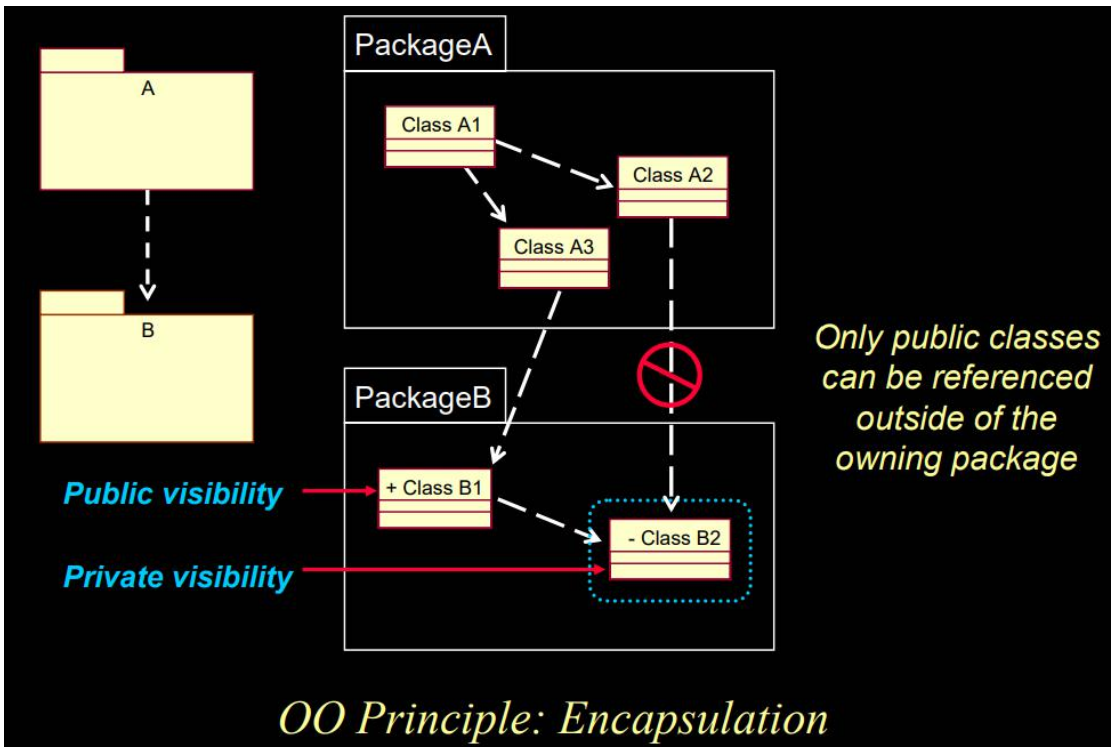
（7）一个类创建了另一个类的实例。

决定两个类不应被放在相同的包的标准：

（1）两个类与不同参与者相关。

（2）一个可选的和一个强制的类不应被放在相同的包中。

H. 包的依赖：包的元素可见性



只有公共的类能被所在包外的引用。

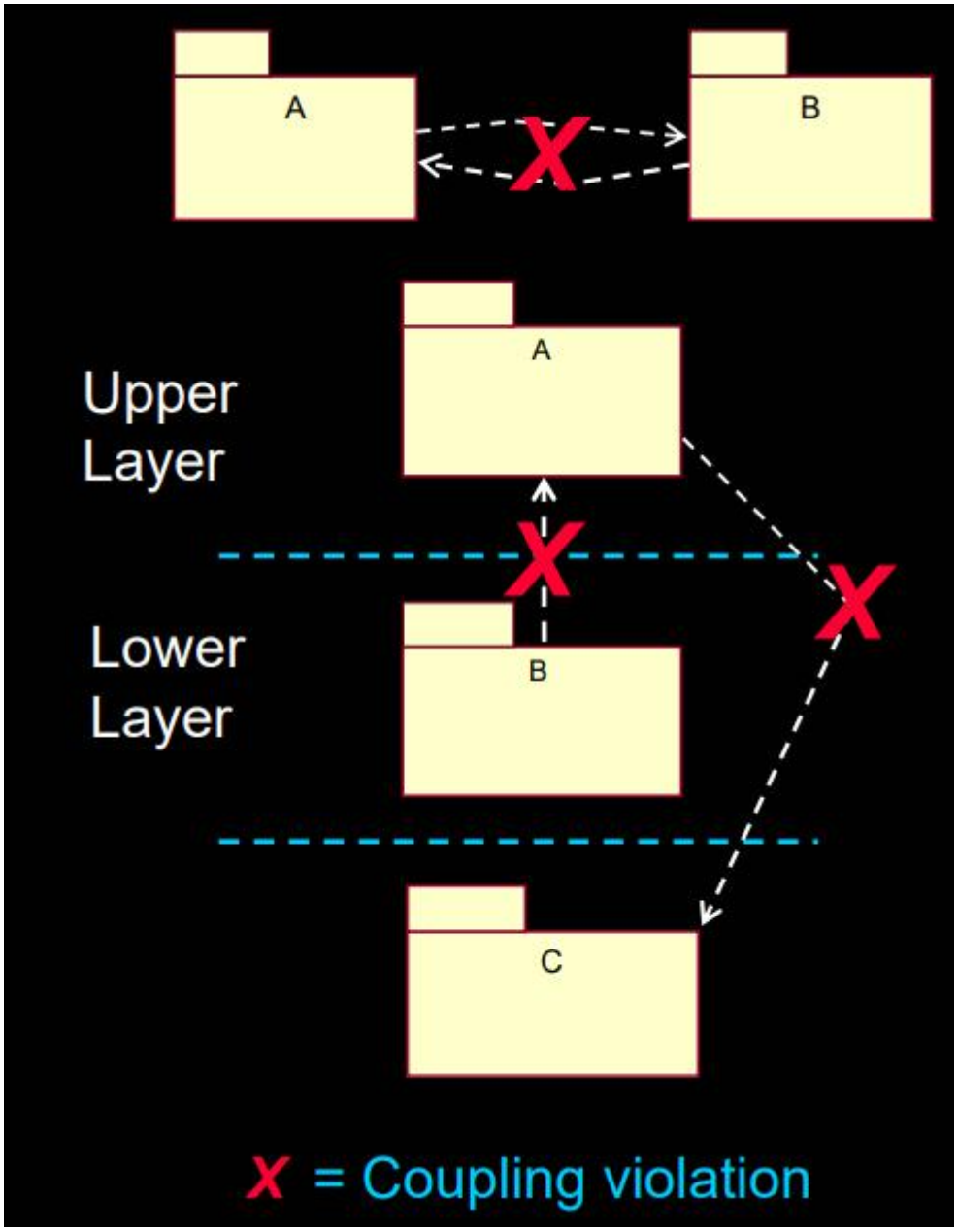
面向对象原则：封装

I. 包的耦合：提示

（1）包不应被交叉耦合。

（2）在较低层次的包不应依赖更高层次的包。

（3）一般地，依赖不应跳过层。



J. 包 vs 子系统

子系统：

（1）提供行为。

（2）完全封装了它们的内容。

（3）容易被代替。

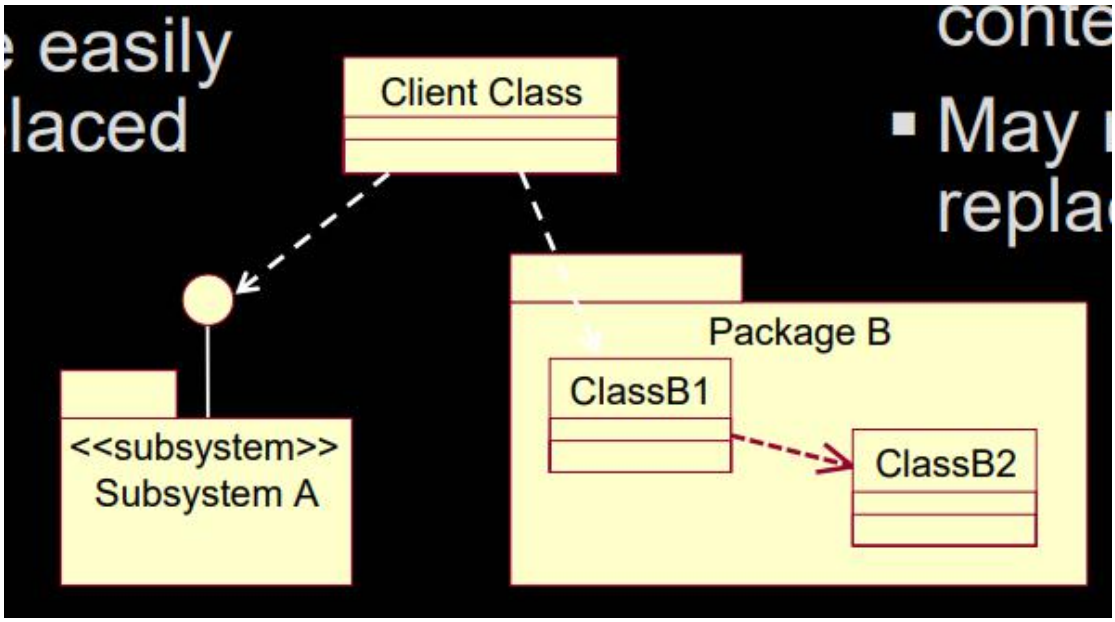
包：

（1）不提供行为。

（2）不完全封装其内容。

（3）不能被轻易代替。

关键是封装



K. 子系统的使用

子系统能被用于把系统划分成多个能相互独立地进行以下操作部分：

（1）被要求（ordered）、配置或交付

（2）被开发，如果接口保持不变

（3）被部署在一组分布式计算节点上

（4）不涉及系统的其他部分的改变

子系统能被用于：

（1）把系统划分成能在关键资源上提供有限安全性的单元

（2）在设计中表示现存产品或外部系统（例如组件）

子系统提升了抽象层级。

L. 标识子系统的提示

（1）看对象的协作。

（2）寻找可选性。

（3）考虑系统的用户接口。

（4）考虑参与者。

（5）寻找类之间的耦合（coupling）和内聚（cohesion）。

（6）看置换（substitution）。

（7）看分布。

（8）看不稳定性（volatility）。

M. 候选子系统

可能演化成子系统的分析类：

（1）提供复杂服务和/或基础功能的类

（2）边界类（用户界面和外部系统接口）

设计中现存的产品或外部系统（如组件）：

（1）通信软件

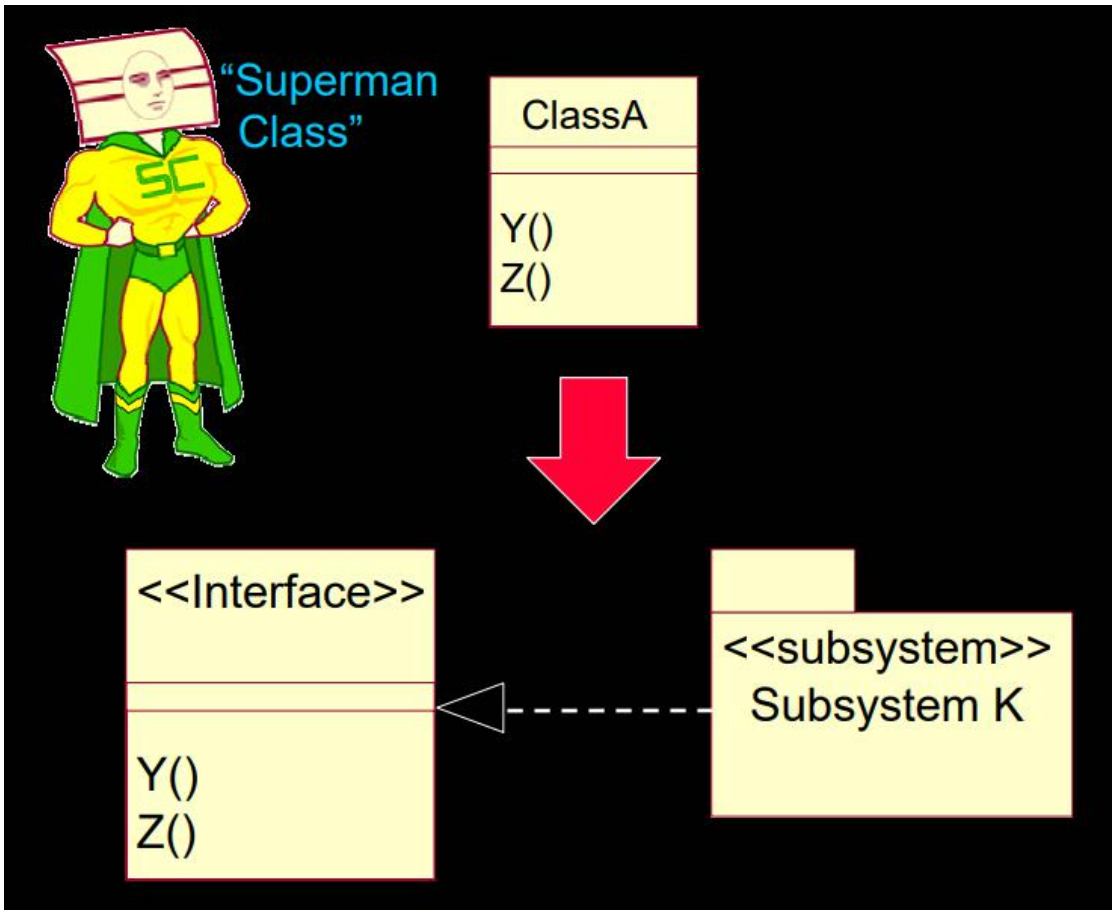
（2）数据库访问支持

（3）类型和数据结构

（4）常见的通用程序

（5）应用相关的产品

N. 标识子系统



标识子系统接口：

O. 标识接口

目的：基于子系统的职责来标识其接口。

步骤：

（1）为所有子系统标识一组候选接口。

（2）寻找接口的相似性。

（3）定义接口的依赖。

（4）映射接口到子系统。

（5）定义接口相关的行为。

（6）包装接口。

稳定、良好定义的接口是建立稳定、有弹性的架构的关键。

P. 接口准则

（1）接口名称：

反映在系统中的参与者。

（2）接口的描述：

表达职责。

（3）操作的定义：

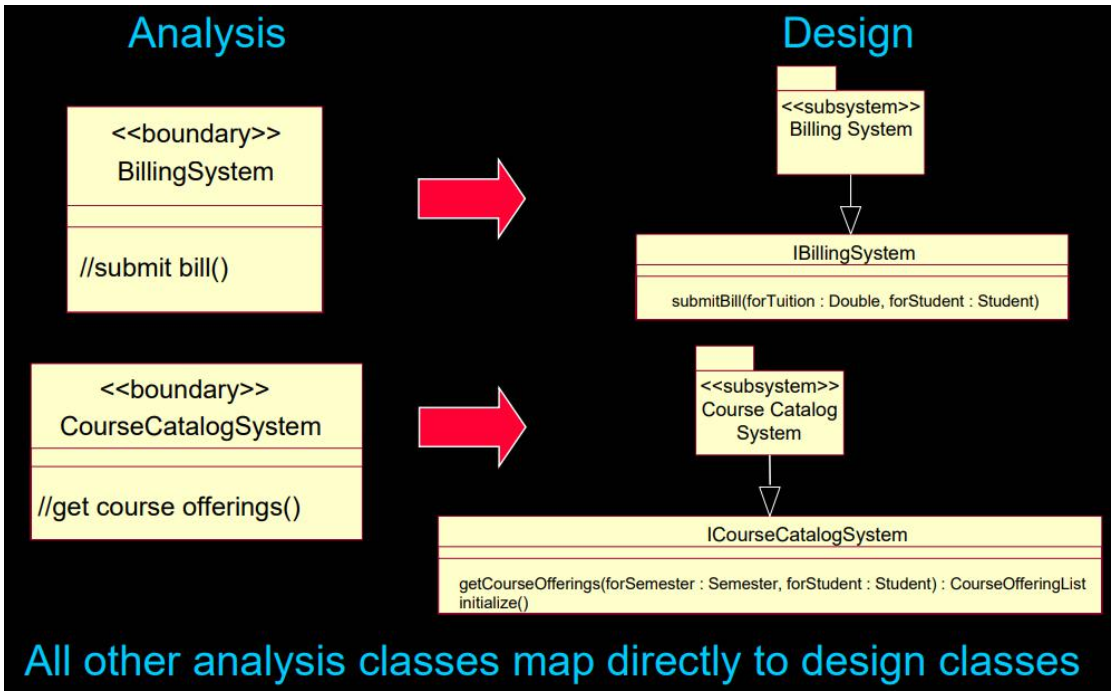
名称应该反映操作结果。

描述操作做了什么、所有参数和结果。

（4）接口文档：

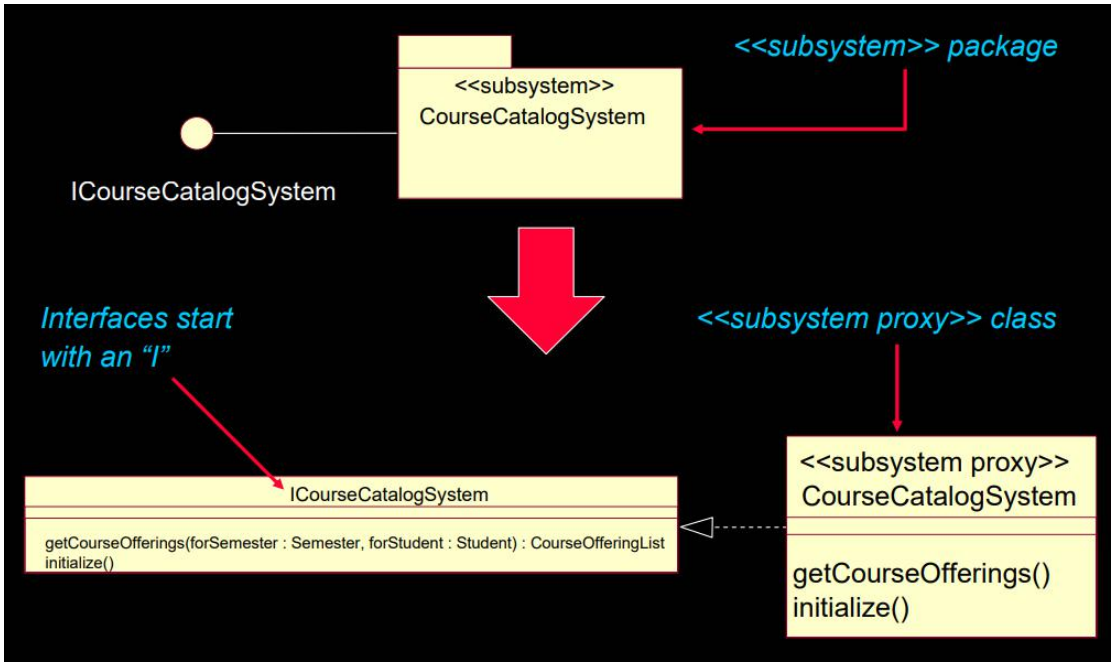
包的支持信息：时序和状态视图、测试计划等。

例子：设计子系统和接口



所有分析类直接映射到设计类。

Q. 建模习惯：子系统和接口



标识复用机会：

R. 复用机会的标识

目的：基于其接口，定义现存的子系统和/或组件在哪里能被复用。

步骤：

（1）寻找相似的接口。

（2）修改新的接口来提高契合度（fit）。

（3）用现存的接口来代替候选接口。

（4）映射候选的子系统到现存的组件上。

S. 可能的复用机会

（1）被开发系统的内部：认出包和子系统的共同特征（commonality）。

（2）被开发系统的外部：

商业上可用的组件。

来源于之前开发的应用的组件。

相反的工程组件。

更新设计模型的组织：

T. 分层的考虑

（1）可见性：只能依赖当前或更低级的层。

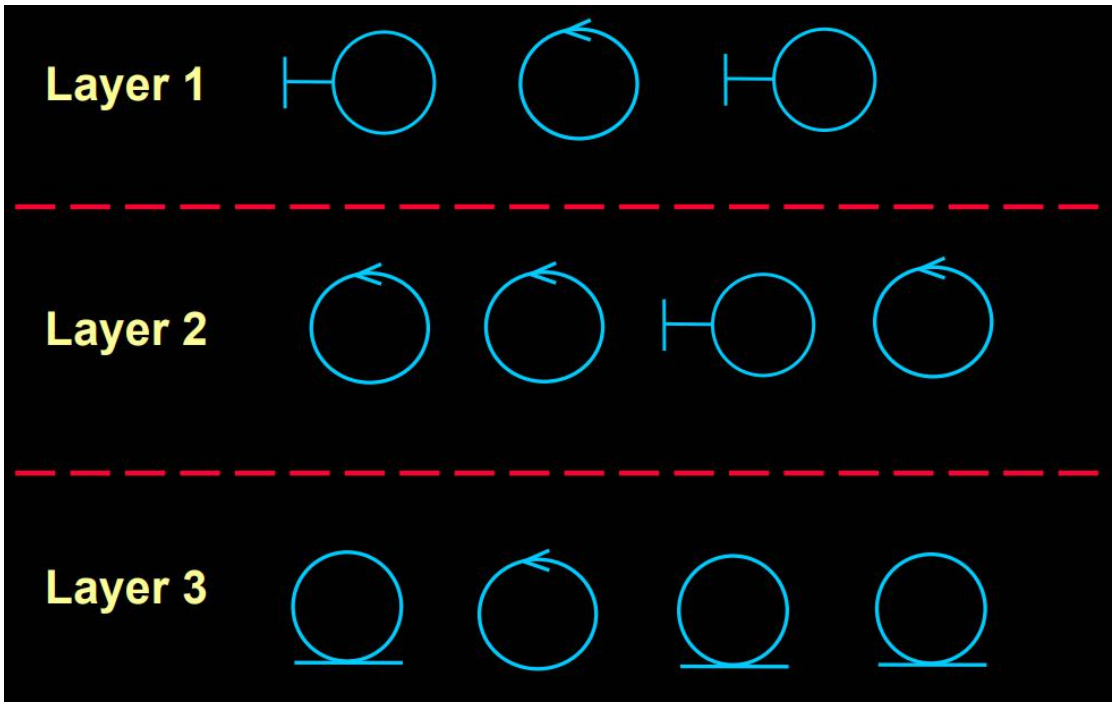
（2）不稳定性：高层被需求影响；底层被环境影响。

（3）通用性：更抽象的模型元素在更低的层。

（4）层的数量：小系统 3-4 层；复杂系统 5-7 层。

目标是减少耦合并使维护更容易。

U. 设计元素和结构



V. 划分考虑

（1）耦合和内聚

（2）用户组织

（3）能力和技术域（skill area）

（4）系统分布

（5）秘密（secrecy）

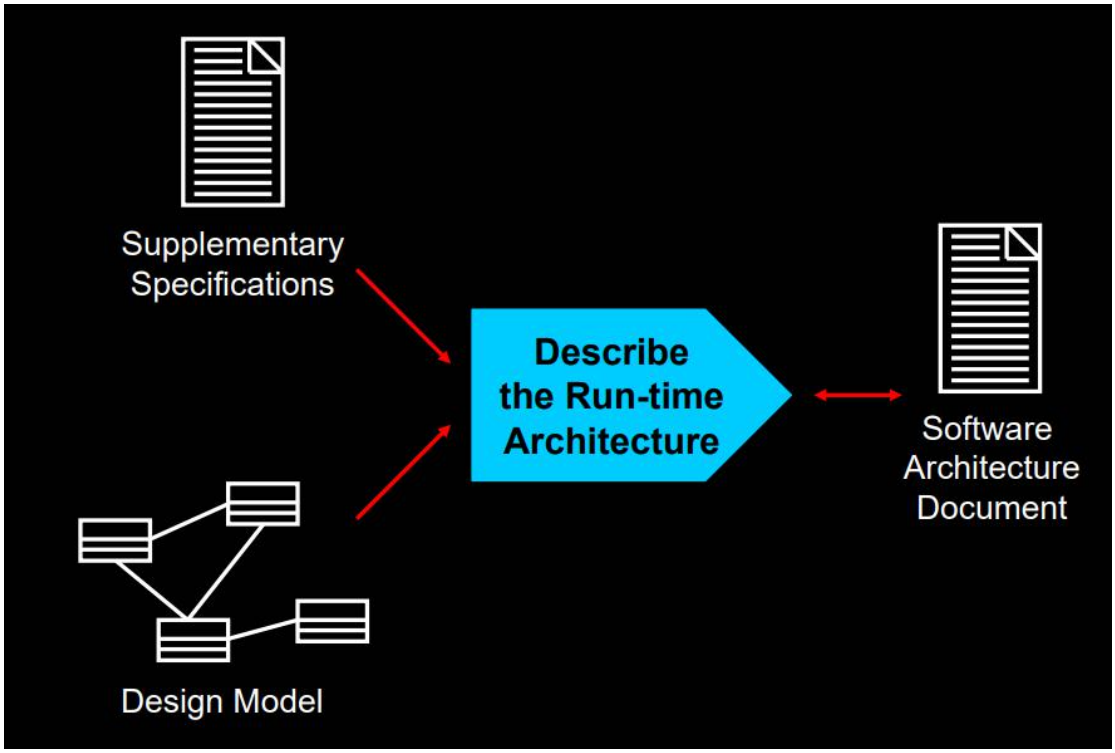
（6）易变性（variability）

尽量避免循环依赖。

10. 运行时架构

A. 运行时架构概述

补充规约、设计模型、软件架构文档



属于改进架构。

B. 进程图

进程图是设计模型中关于进程和线程的架构上具有重大意义的图。

C. 并发（concurrency）

并行路段需要很少的协调（coordination）。

为了安全互动，双向路段需要一些协调。

十字路口需要小心的协调。

D. 并发的意义

（1）软件可能需要响应看起来随机生成的事件。

（2）若有多个 CPU 可用，任务并行可以提高性能。

（3）通过并发能改善系统的控制。

E. 实现并发：并发机制（concurrency mechanism）

（1）为了支持并发性，系统必须提供多线程的控制（multiple threads of control）。

（2）常见的并发机制：

多进程（multiprocessing）：多 CPU 同时执行。

多任务（multitasking）：操作系统通过插入不同任务的执行来在单个 CPU 上模拟并发性。

基于应用的解决方案：应用软件负责适时在不同代码分支间切换。

F. 描述运行时架构的步骤

（1）分析并发性需求。

（2）标识进程和线程。
（3）标识进程的生命周期。
（4）将进程映射到实现上。
（5）在进程间分配模型元素。

分析并发性需求：

G. 并发性需求

并发性需求是被以下驱动的：

（1）系统分布度
（2）系统事件驱动度
（3）关键算法的计算强度
（4）环境支持并行执行的程度

并发性需求根据解决冲突的重要性进行排序。

例子：并发性需求

在课程注册系统中，并发性需求来自于需求和架构：

（1）多个用户必须能同时进行他们的工作。
（2）如果有学生选课时待选课程满了，该学生必须被提示。
（3）基于风险的原型发现，在没有创造性使用中层处理能力的情况下，遗留的课程目录数据库不能满足我们的性能需求。

标识进程和线程：

H. 进程和线程

进程：提供重量级的控制流；是独立的；能被分成单独的线程。
线程：提供轻量级的控制流；在一个进程的上下文中运行。

I. 标识进程和线程

对每个系统需要的分开的控制流，创建一个进程或线程。
分开的控制的线程可能被用于：

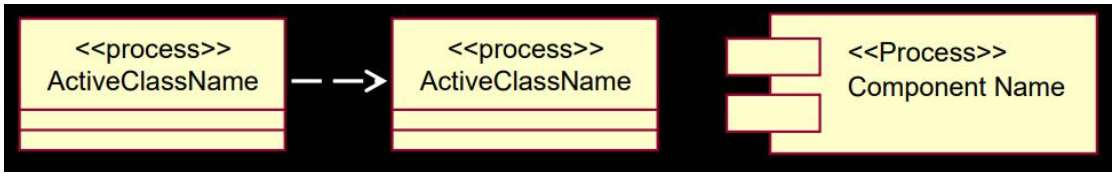
（1）利用多个 CPU 和/或节点
（2）提高 CPU 利用率
（3）服务的时间相关的事件
（4）对任务划分优先级
（5）达到可扩展性（共享加载）
（6）分离软件区域的影响
（7）提升系统可用性
（8）支持主要的子系统

J. 对进程建模

（1）进程可以用以下方法建模：
活动类（类图）和对象（互动图）
组件（组件图）

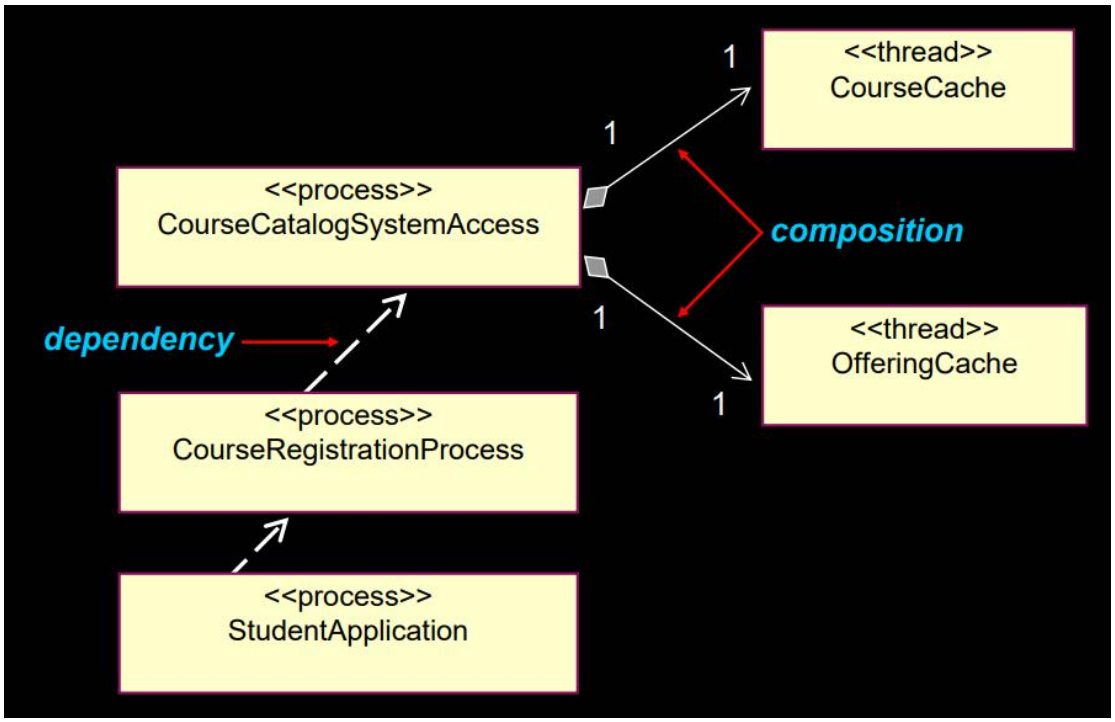
（2）模式标记（stereotype）：<<process>>或<<thread>>

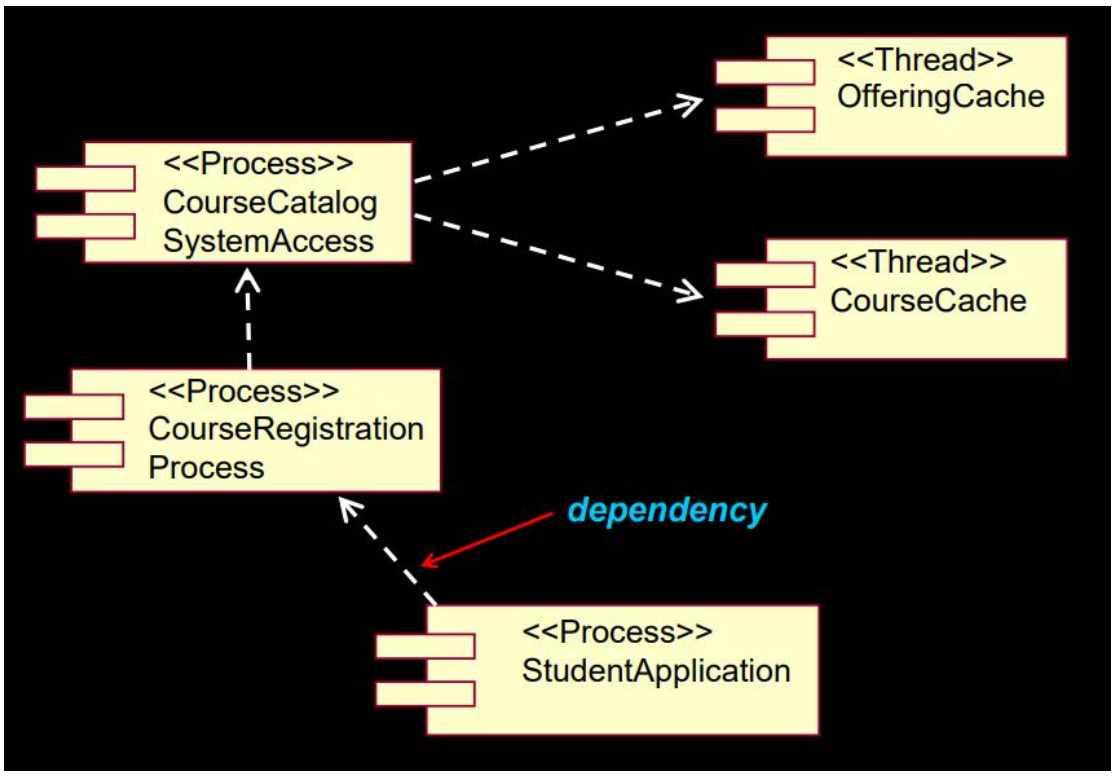
（3）进程关系能用依赖建模。



课程能用类图建模进程和线程。

例子：





标识进程的生命周期：

K. 创建和销毁进程和线程

（1）单进程架构

应用开始时创建进程。

应用结束时销毁进程。

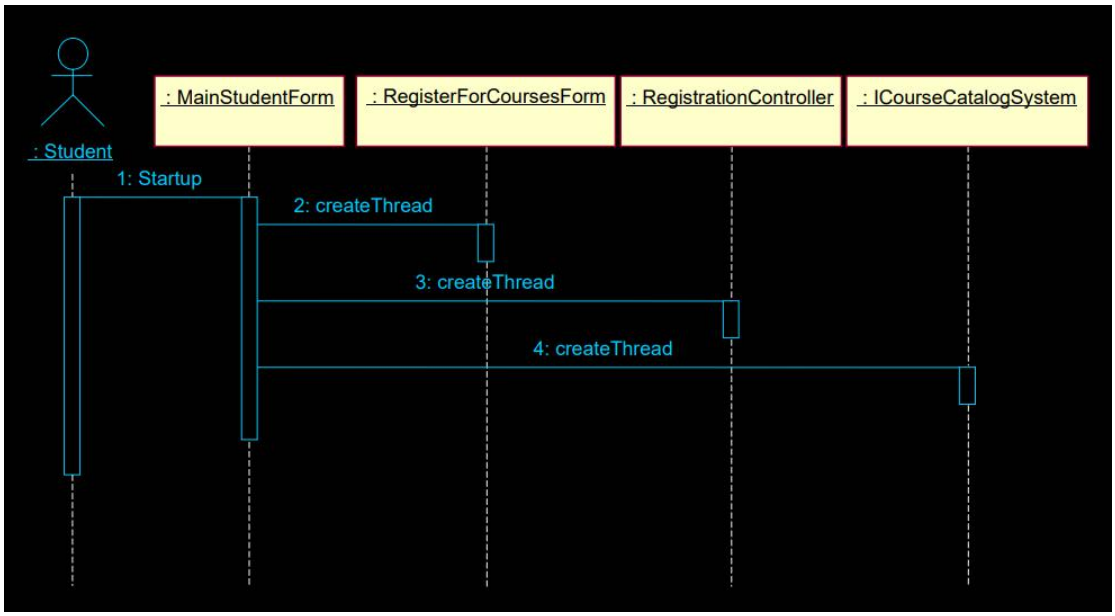
（2）多进程架构

新进程通常被初始进程创建，初始进程在应用开始时创建。

每个进程必须被单独销毁。

注意：课程注册系统利用多进程架构。

例子：创建进程和线程



应用开始时创建线程。

映射进程到实现上：

L. 把进程映射到实现上

（1）进程和线程必须被映射到特定的实现结构上。

（2）考虑：

进程耦合性

性能需求

系统进程和线程限制

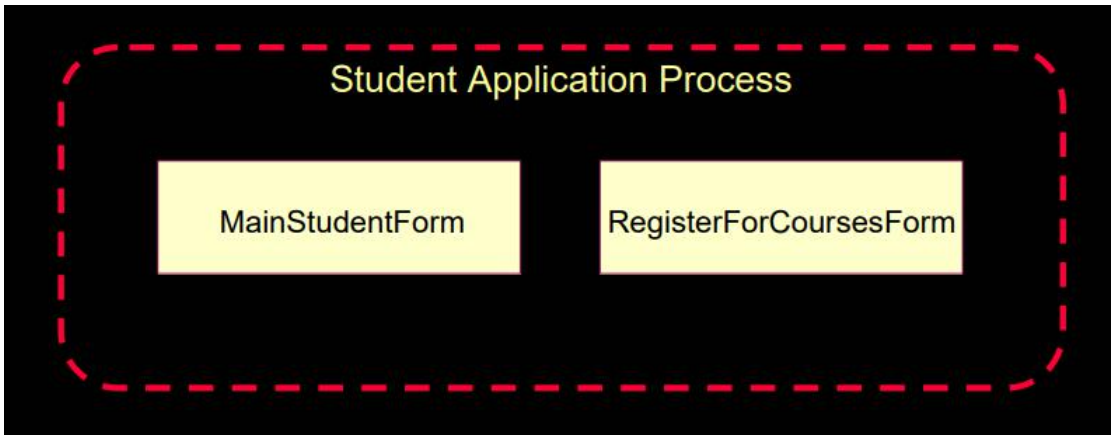
现存的线程和进程

IPC 资源的可用性

在进程间分配模型元素：

M. 设计元素的分配

一个给定类或子系统的实例必须在一个进程内：他们可能被多个进程执行。



N. 元素到进程的设计考虑

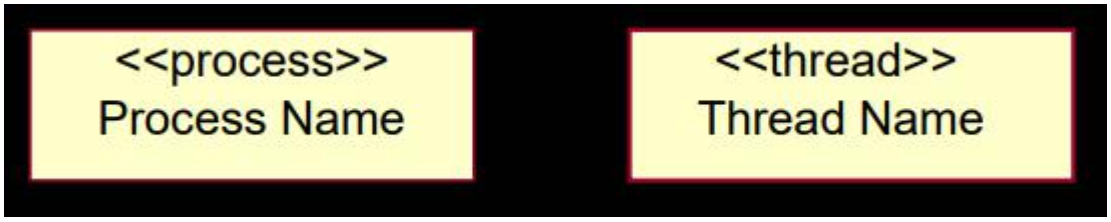
基于：

（1）性能和并发性需求
（2）分布需求和并行执行的支持
（3）冗余和可用性需求

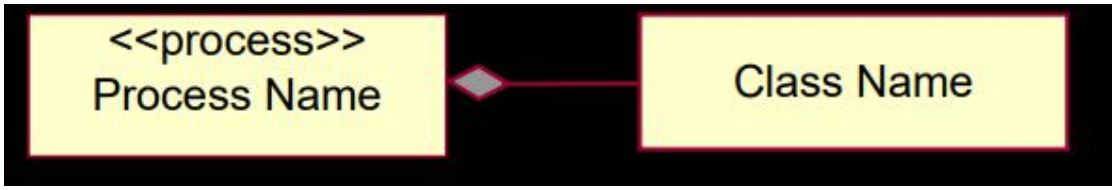
类/子系统的特性也要考虑：
（1）自主
（2）从属
（3）持久
（4）分布

O. 元素到进程的设计策略
两种策略（同时使用）
从内而外：
（1）将紧密合作和必须在同一控制线程中执行的元素分到同一组
（2）分离不互动的元素
（3）重复直到你达到了最小数量的进程，仍然提供所需分布和高效的资源利用。
从外而内：
（1）为每个外部刺激（stimuli）定义一个分开的控制线程
（2）为每个服务定义一个分开的服务器控制线程
（3）减少线程数量指导能被支持

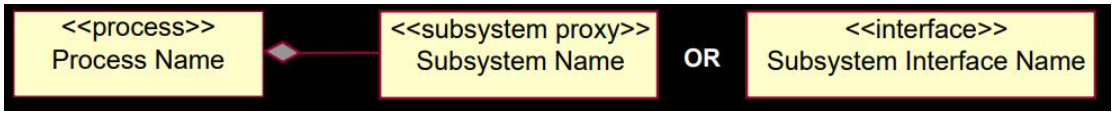
P. 对从元素到进程的映射建模
类图：
活动类作为进程；



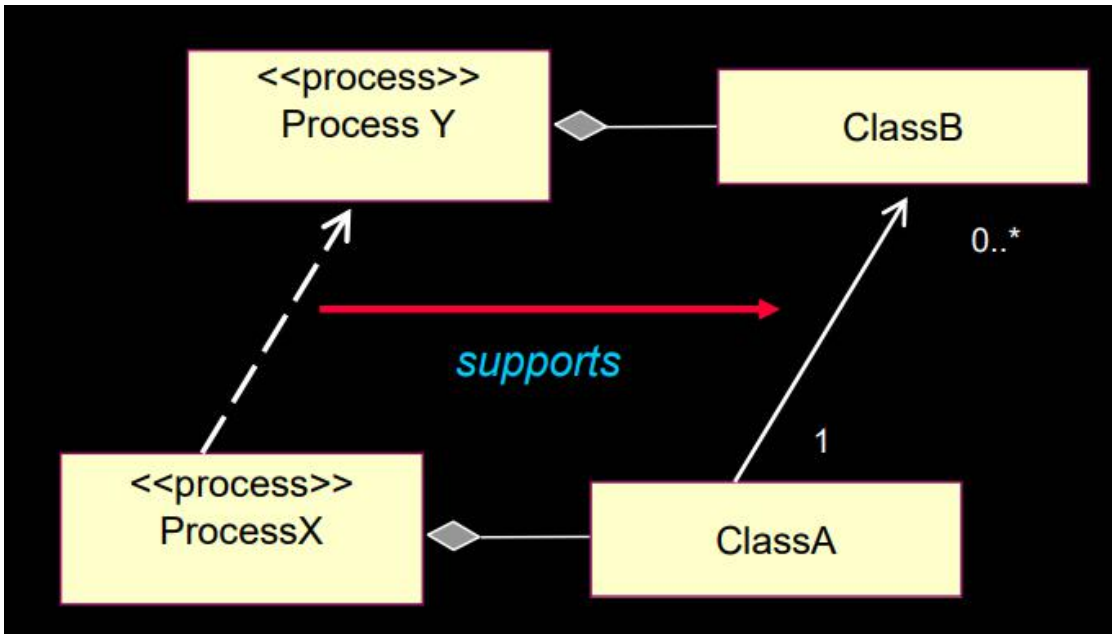
从进程到类的组成关系；



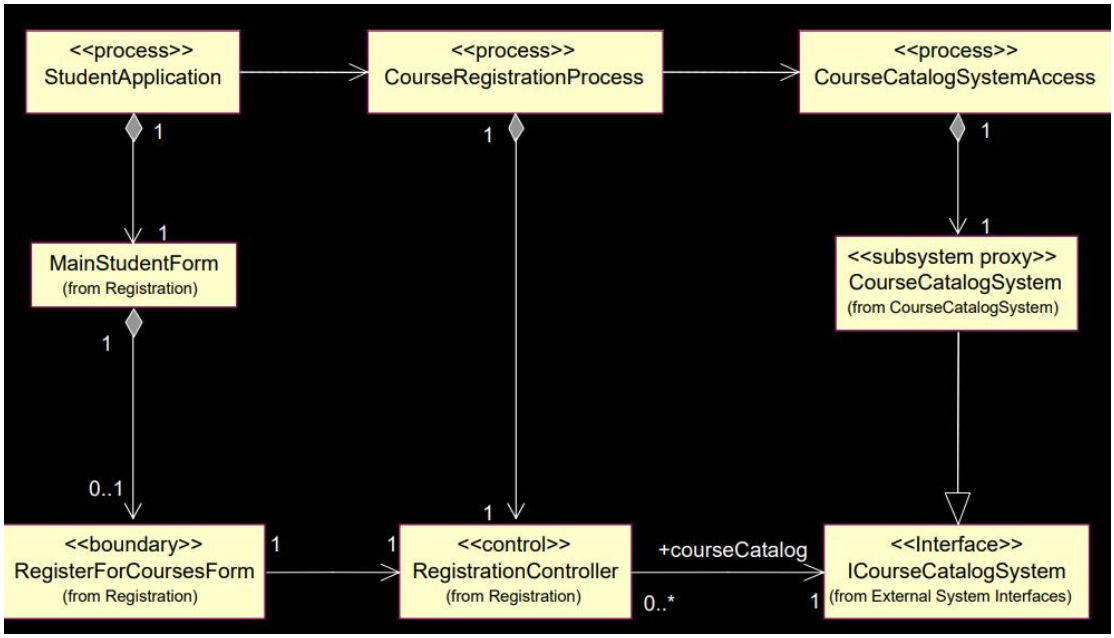
从进程到子系统的组成关系。

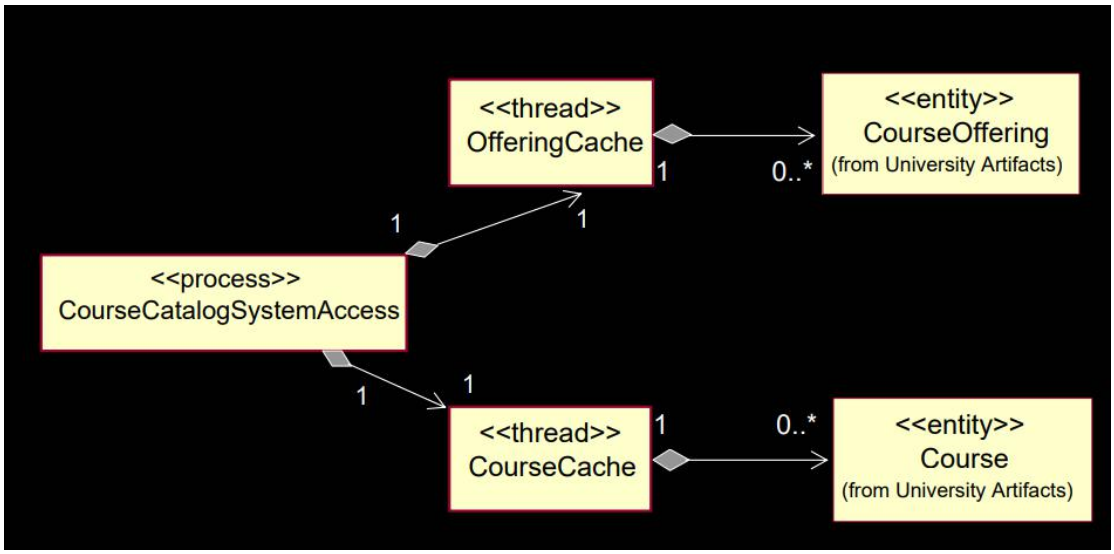


Q. 进程关系
进程关系必须支持设计元素关系。


<<process>>
Process Y
ClassB
0..*
supports
1
<<process>>
ProcessX
ClassA

例子：


<<process>>
StudentApplication
<<process>>
CourseRegistrationProcess
<<process>>
CourseCatalogSystemAccess
1
1
1
1
1
1
MainStudentForm
(from Registration)
<<subsystem proxy>>
CourseCatalogSystem
(from CourseCatalogSystem)
1
0..1
1
<<boundary>>
RegisterForCoursesForm
(from Registration)
1
1
<<control>>
RegistrationController
(from Registration)
+courseCatalog
0..*
1
<<Interface>>
ICourseCatalogSystem
(from External System Interfaces)

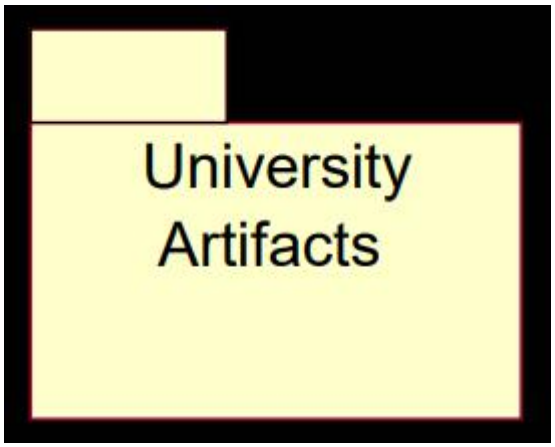


11. 重要定义

A. 包

包（package）是为了将组织元素分到不同组中的目的广泛的机制。

它是一种能包含其他模型元素的模型元素。



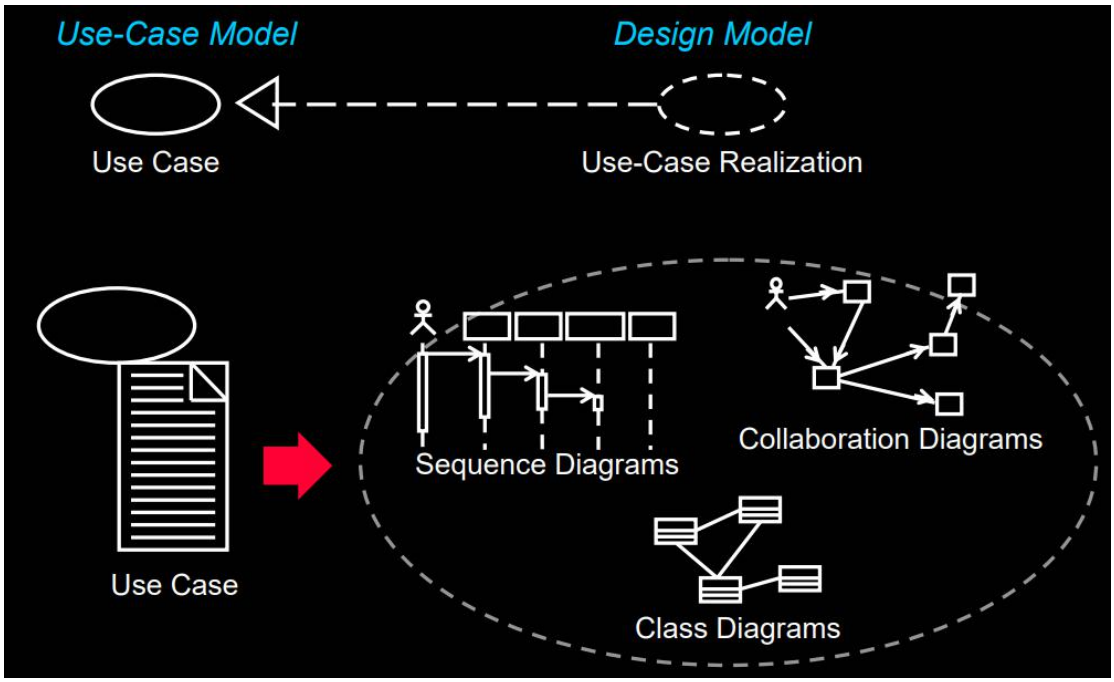
一个包能被用于：

组织开发下的模型。

作为一个配置管理的单元。

B. 用例实现

用例实现（use-case realization）



C. 类（class）

一种抽象

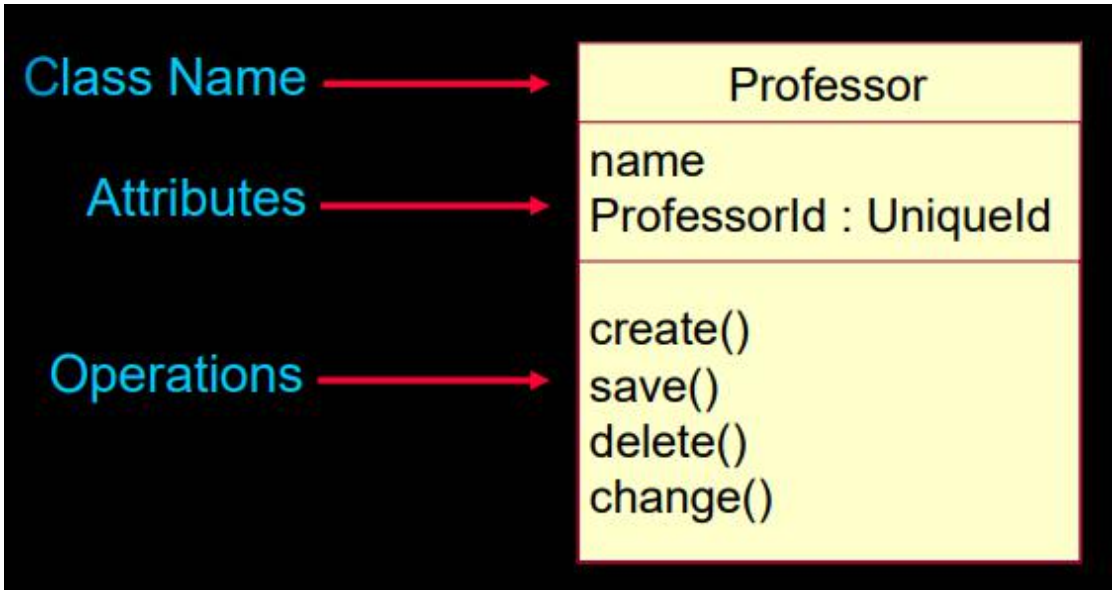
描述一组有共同点的对象：

属性（property, attribute）

行为（behavior, operation）

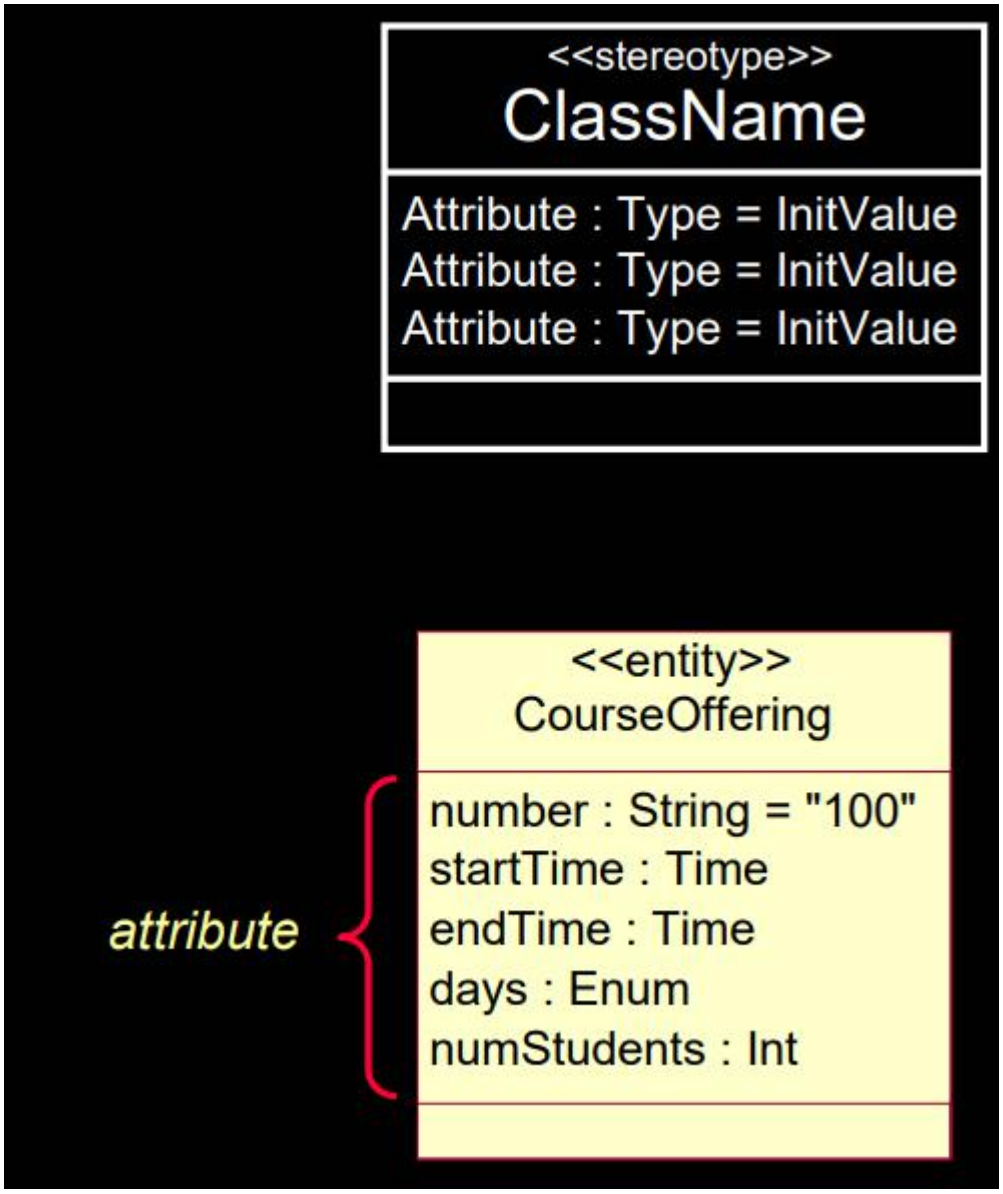
关系（relationship）

语义（semantic）



D. 属性（attribute）

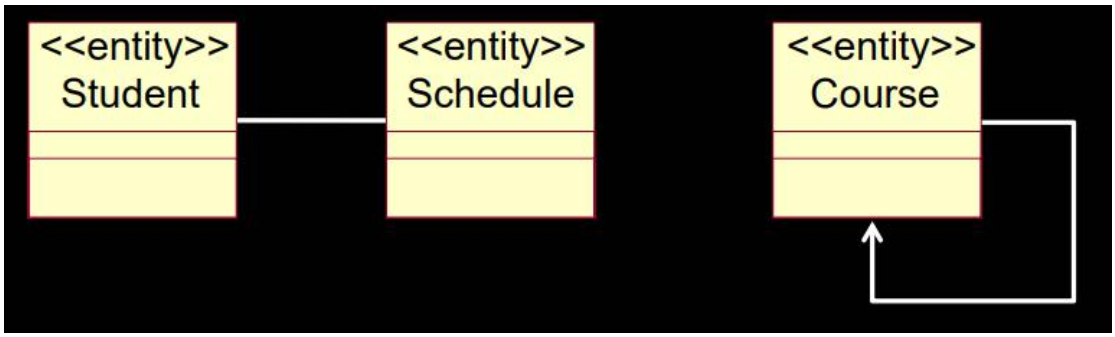
在分析时，不要花时间在属性的署名上。



E. 关联（association）

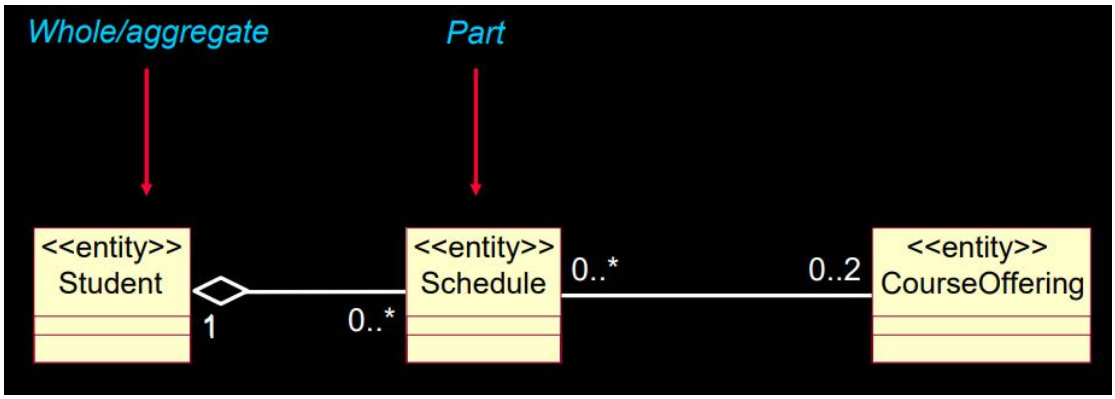
两个或更多的规定他们的实例的连接分类器（classifier）之间的语义关系。（The semantic relationship between two or more classifiers that specifies connections among their instances）。

一个结构的关系，指明一个事物的对象是与另一个的对象相连。



F. 聚集（aggregation）

一种特殊类型的关联，其在一个整体及其部分之间建模一个整体-局部的关系。



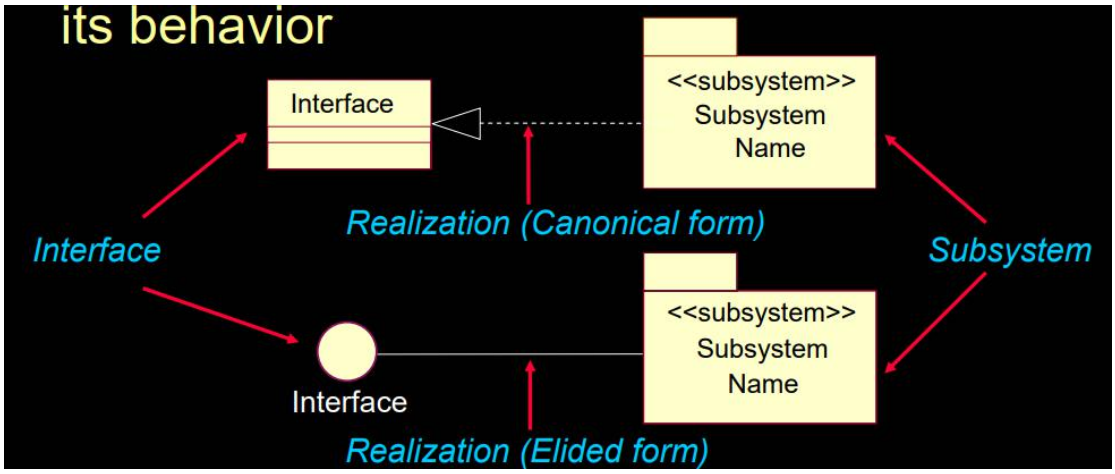
G. 多重性（multiplicity）

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

H. 子系统（subsystem）和接口（interface）

是一个包（能包括其他模型元素）和一个类（有行为）的“杂交”。

实现一个或多个接口来定义其行为。



子系统：

完全封装了行为。

代表一个独立拥有清晰接口的能力（可能被复用） represent an independent capability with clear interfaces (potential for reuse)

为多个实现的变种建模