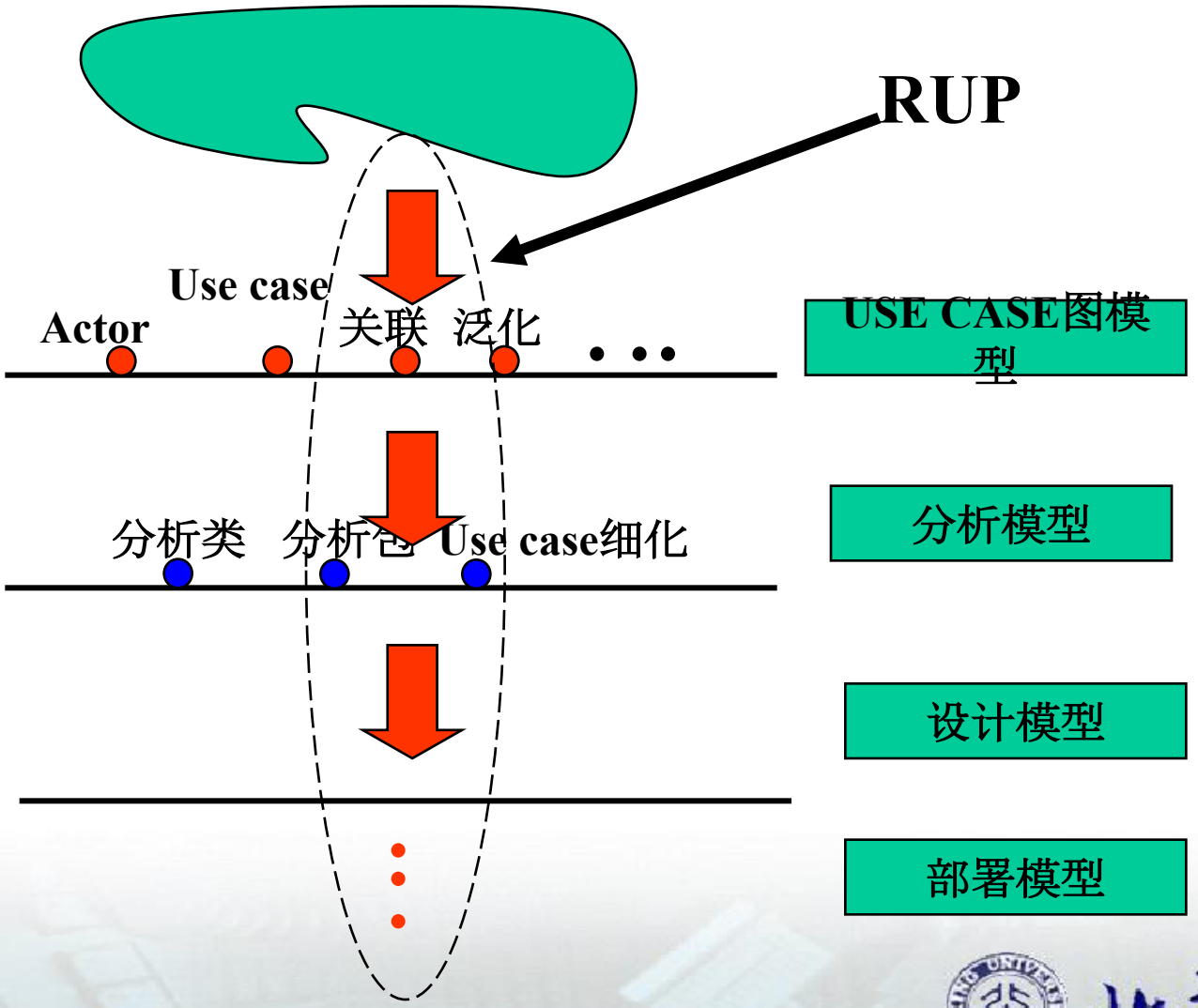
第六章

面向对象方法-RUP

北京大学

UML与RUP



统一软件过程(RUP)

1) 概述

(1) UML是一种可视化的建模语言，而不是一种特定的软件开发方法学。作为一种软件开发方法学，为了支持软件开发活动，例如软件设计，至少涉及三方面的内容：一是应定义设计抽象层，即给出该层的一些术语，二是应给出该层的模型表达工具，三是应给出如何把需求层的模型映射为设计层的模型，即过程。UML仅包括前两方面的内容，即给出了一些可用于定义软件开发各抽象层的术语（符号），给出了各层表达模型的工具。

(2)RUP的本质及特点

本质：是“一般的过程框架”．即：

--为软件开发，进行不同抽象层之间“映射”，安排其开发活动的次序，指定任务和需要开发的制品，提供了指导；

--为对项目中的制品和活动进行监控与度量，提供了相应的准则。

换言之，RUP比较完整地定义了将用户需求转换成产品所需要的活动集，并提供了活动指南以及对产生相关文档的要求。

适用于：大多数软件系统的开发，涉及

-不同应用领域　-不同类型的组织

-不同的技能水平　-不同的项目规模

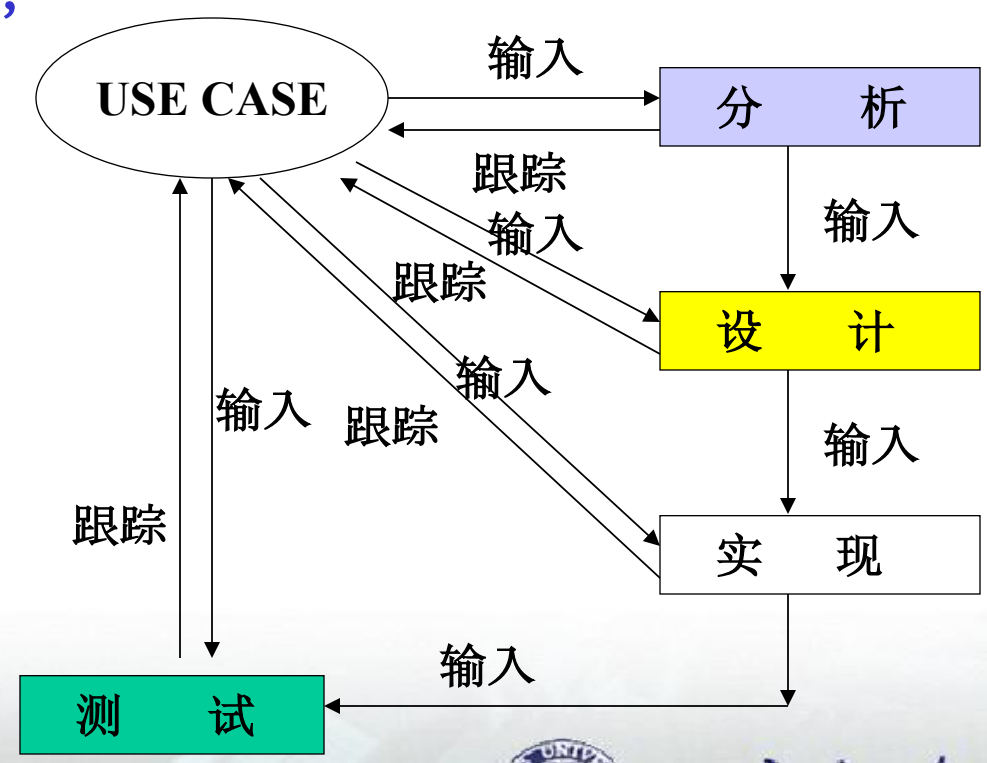
可见, RUP 和UML 是“统一”的方法学。

RUP的突出特点

是一种以用况（Use Case）为驱动的、以体系结构为中心的、迭代、增量式开发。

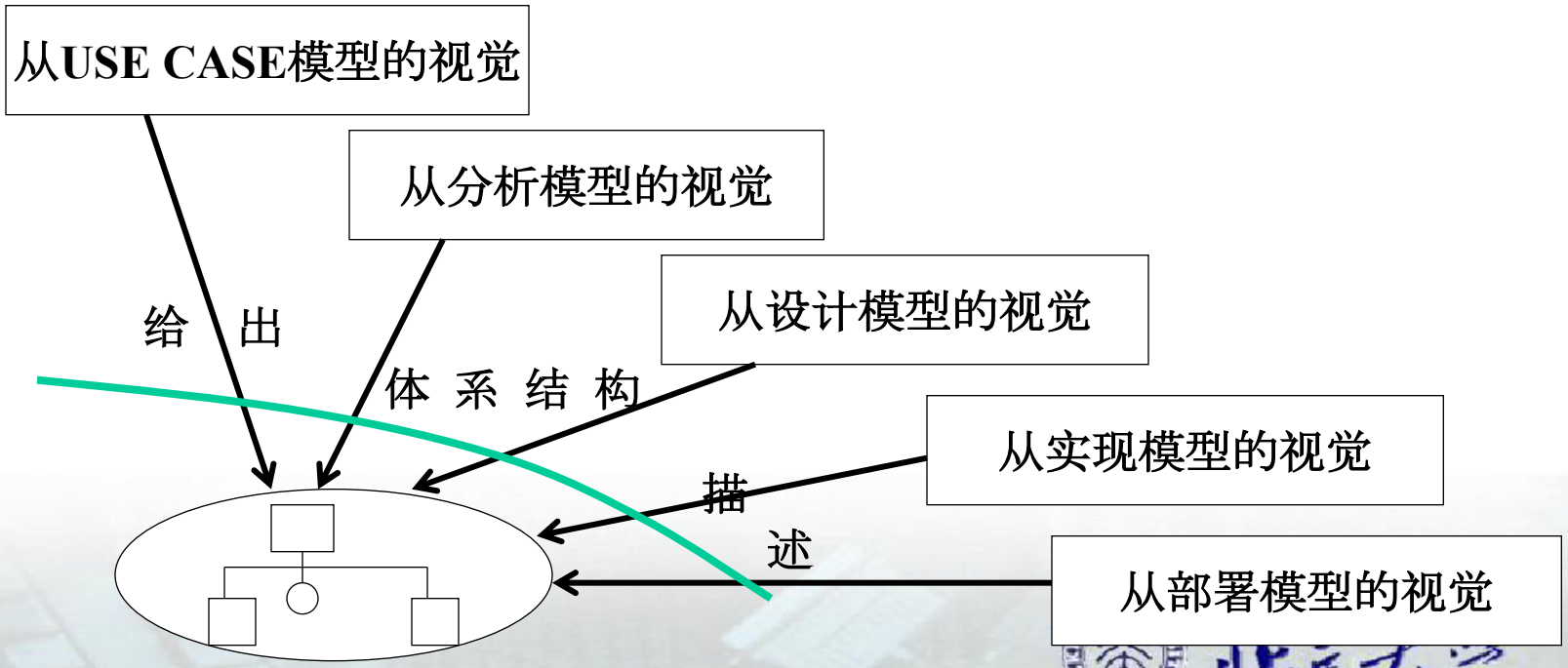
♦以用况为驱动

意指在系统的生存周期中，以用况作为基础，驱动有关人员对所要建立系统之功能需求进行交流，驱动系统分析、设计、实现和测试等活动，包括制定计划、分配任务、监控执行和进行测试等，并将它们有机地组合为一体，使各个阶段中都可以回溯到用户的实际需求。



♦以体系结构为中心

意指在系统的生存周期中，开发的任何阶段（RUP规定了四个阶段，即初始阶段、细化阶段、构造阶段和移交阶段）都要给出相关模型视角下的有关体系结构的描述，作为构思、构造、管理和改善系统的主要制品。



系统体系结构是对系统语义的概括表述，内含一些决策，主要涉及软件系统的组织（包括构成系统的结构元素、各元素的接口、由元素间的各种协作所描述的各元素行为、由结构元素和行为元素构成的子系统、相关的系统功能和性能、其他约束等）以及支持这种组织的体系结构风格。

系统体系结构对所有与项目有关人员来说都是能够理解的，因此便于用户和其他关注者对系统达到共识，以便建立和控制系统的开发、复用和演化。

因此，在系统体系结构描述中，应关注子系统、构件、接口、协作、关系和节点等重要模型元素，而忽略其他细节。

具体地说，体系结构描述应根据相关模型的视角：

① 展示对体系结构有意义上的用况、子系统(不涉及子系统的隐含成分和私有成分，及其变种)、接口、类(主要为主动类)、构件、节点和协作；

② 展示对系统体系结构有意义的非功能需求，例如性能、安全、分布和并发等；

③ 简述相关的平台、遗产、所用的商业软件、框架和模板机制等；以及

④ 各种体系结构模式。

例如，为获得系统用况模型视角下的系统体系结构描述，应：

① 在一般性地了解系统用况之后，勾画与特定用况和平台无关的系统体系结构轮廓。

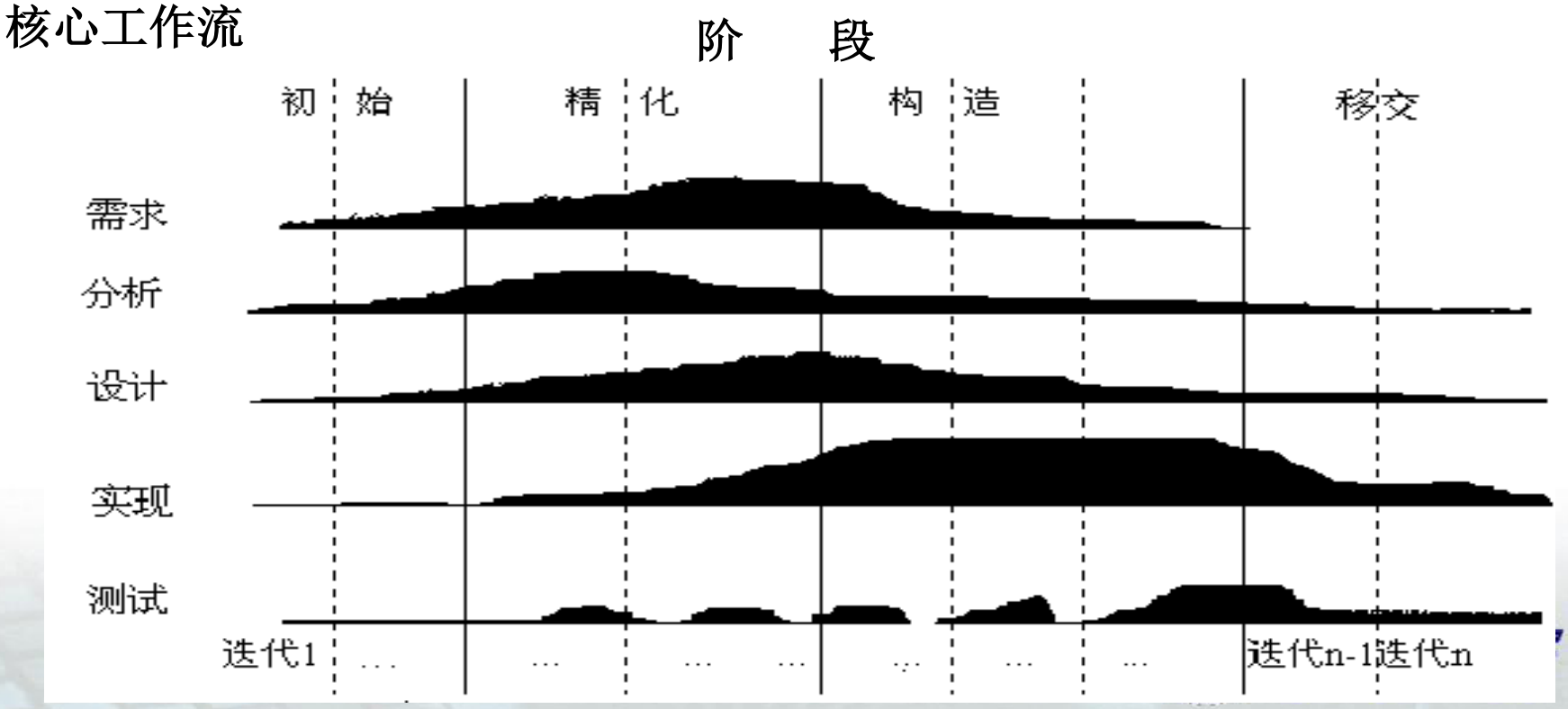
② 关注一些关键用况。所谓关键用况，是指那些有助于降低最大风险的用况，对系统用户来说是最重要的用况，以及有助于实现所有重要的功能而不遗留任何重大问题的用况。

③ 给出每一关键用况的描述。其中应考虑软件需求、中间件、遗产系统和非功能性需求等，以便产生更加成熟的用况和更多的系统体系结构成分。

④ 对以上三步进行迭代，得到一个文档化的体系结构基线。并在此基础上，形成一个稳定的系统体系结构描述。

♦迭代、增量式开发

意指通过开发活动的迭代，不断地产生相应的增量。在RUP中，规定了四个开发阶段：初始阶段（the inception phase）、精化阶段（the elaboration phase）、构造阶段（the construction phase）和移交阶段（the transition phase）



每次迭代都要按照专门的计划和评估标准，通过一组明确的活动，产生一个内部的或外部的发布版本。两次相邻迭代所得到的发布版本之差，称为一个增量，因此增量是系统中一个较小的、可管理的部分(一个或几个构造块)。

贯穿整个生存周期的迭代，形成了项目开发的一些里程碑。

--每一阶段的结束，是项目的一个主里程碑（共四个），产生系统的一个体系结构基线，即模型集合所处的当时状态。

--主里程碑是管理者与开发者的同步点，以决定是否继续进行项目，确定项目的进度、预算和需求等。

--在四个阶段中的每一次迭代的结束，是一个次里程碑，产生一个增量。次里程碑是如何进行后续迭代的决策点。

--系统体系结构基线的建立，是精化阶段的一个目标，期间通过不断演化，到该阶段末得到这一基线，是系统的“骨架”.

--该基线包括早期版本的用况模型、分析模型、设计模型、部署模型、实现模型和测试模型，但此时用况模型和分析模型较为成熟。

注:该基线应该是坚实的，因为它是开发人员当时和将来进行开发时所要遵循的标准;并应与最终系统(对客户发布的产品) 几乎具有同样的骨架。但最后形成的体系结构基线是系统各种模型和各模型视角下体系结构描述的一个集合。

--在实践中，体系结构描述和体系结构基线往往同时开发，以便指导整个软件开发的生命周期。期间，体系结构描述不断更新，以便反映体系结构基线的变化。

综上可知，**RUP**的迭代增量式开发，是演化模型的一个变体，即规定了“大”的迭代数目-四阶段，并规定了每次迭代的目标。

初始阶段的基本目标是：

❶获得与特定用况和平台无关的系统体系结构轮廓，以此建立产品功能范围；

❷编制初始的业务实例，从业务角度指出该项目的价值，减少项目主要的错误风险。

简言之，其目标是：建立该项目的生存周期目标

（objectives）

精化阶段的基本目标是：

通过捕获并描述系统的大部分需求（一些关键用况），建立系统体系结构基线的第一个版本，主要包括用况模型和分析模型，减少次要的错误风险.从而到该阶段末，就能够估算成本、进度，并能详细地规划构造阶段。

构造阶段的基本目标是：

❶通过演化，形成最终的系统体系结构基线（包括系统的各种模型和各模型视角下体系结构描述）；

❷开发了完整的系统，确保产品可以开始向客户交付,即具有初始操作能力。

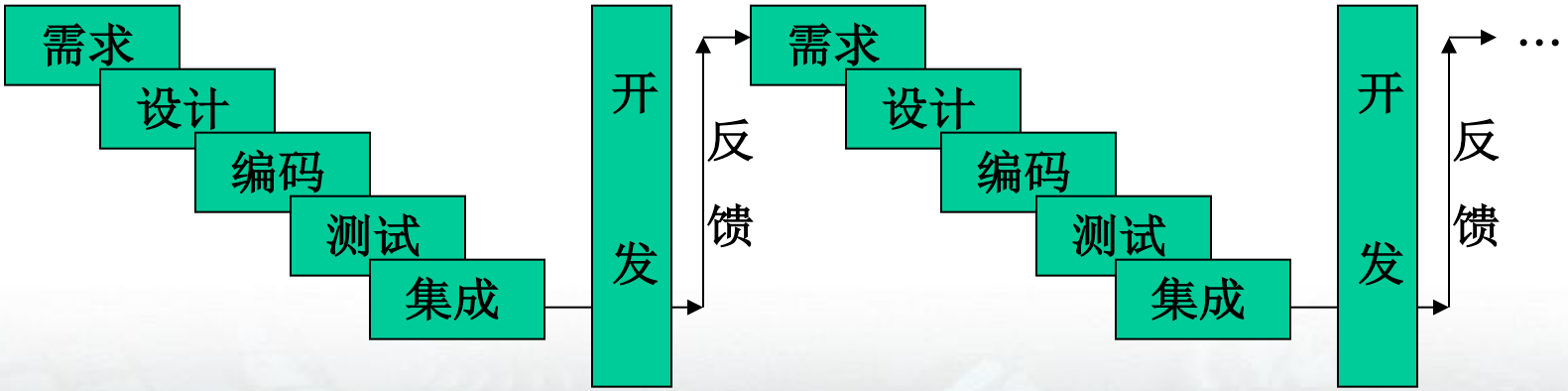
交付阶段的基本目标是：确保有一个实在的产品，发布给用户群。期间，培训用户如何使用该软件。

注：这4个阶段是演化模型的一个变体。

演化模型（Evolutionary model）　　- 一种迭代风范

是一种有弹性的过程模式，由一些小的开发步组成，每一步历经需求分析、设计、实现和验证，产生软件产品的一个增量。通过这些迭代，完成最终软件产品的开发。

- 针对事先不能完整地定义需求
- 针对用户的核心需求,开发核心系统
- 根据用户的反馈,实施活动的迭代



核心系统开发　　第二次迭代

演化模型还具有以下优点：

❶在需求不能予以规约时，可以使用这一演化模型。

❷用户可以通过运行系统的实践，对需求进行改进。

❸与瀑布模型相比，需要更多用户/获取方的参与。

缺点有：

❶演化模型的使用,需要有力的管理。

❷演化模型的使用很容易成为不编写需求或设计文档的借口，即使很好地理解了需求或设计。

❸用户/获取方不易理解演化模型的自然属性，因此当结果不够理想时，可能产生抱怨。

2) 需求获取的目标及其基本途径

需求获取面临的挑战

- 问题空间理解
- 人与人之间的交流
- 需求的不断变化

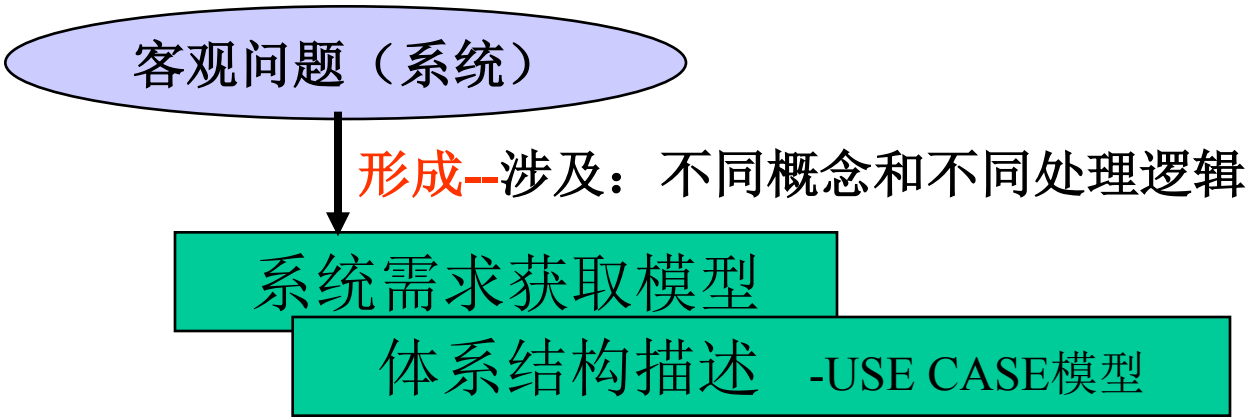
需求获取技术特征

· 方便通讯（使用易于理解的语言）
· 提供定义系统边界的方法
· 提供划分、抽象、投影等方法
· 允许采用多种可供选择的设计方法
· 适应需求的变化
· 支持使用问题空间的术语，思考问题和编制文档
…

(1) 需求获取的目标

对大系统的开发来说，需求一般包括需求获取和需求分析

需求获取的目标是：



(2) 实现需求获取目标的基本途径

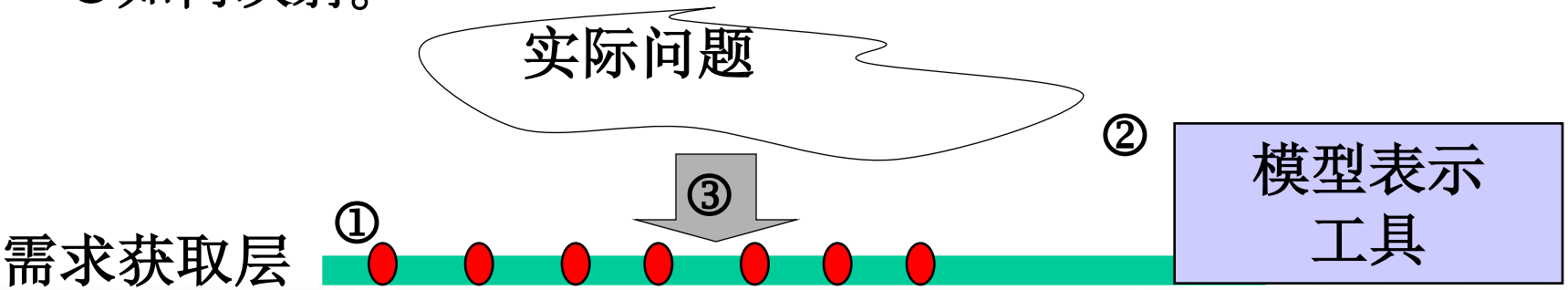
实现实际问题到软件开发需求获取层的映射，即从软件开发的角度-实现第一次抽象。

其中至少涉及以下3个问题：

①如何定义需求获取层，即给出该层的术语；

②如何确定模型表示工具；

③如何映射。

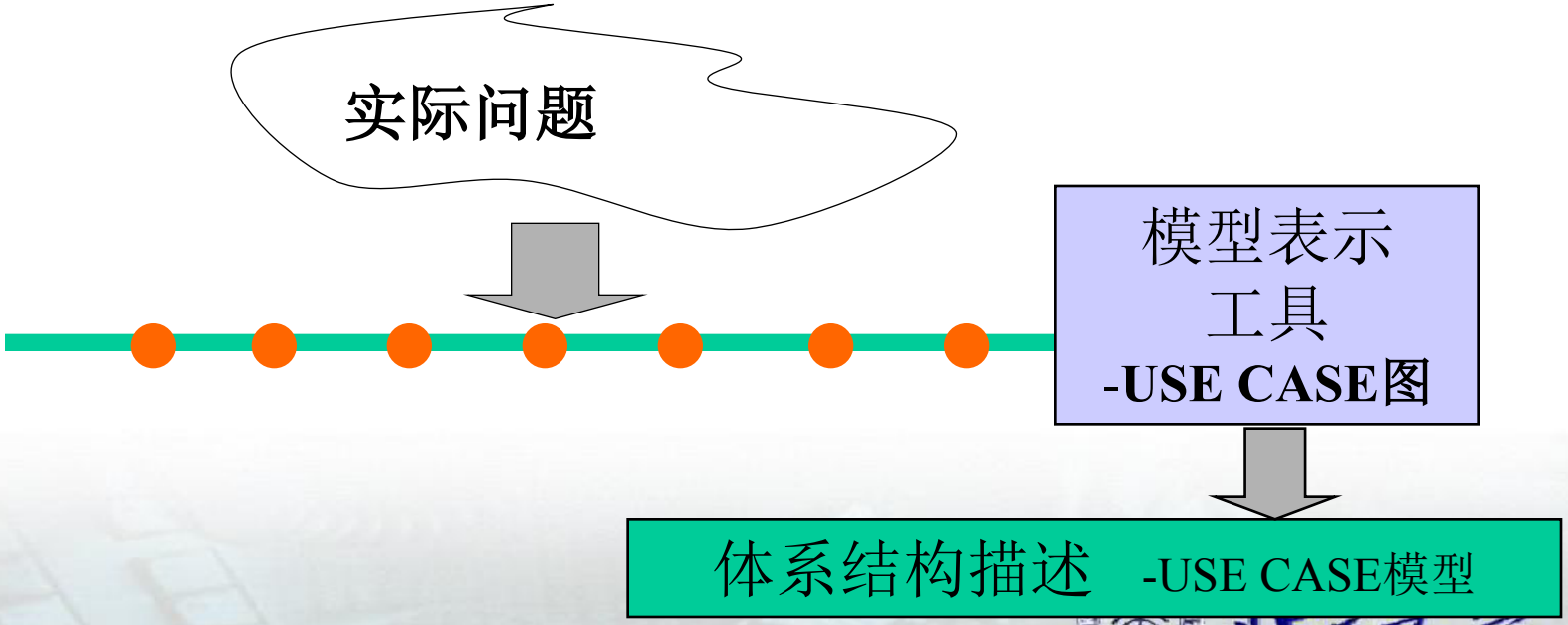


①需求获取层的术语（概念）及表达模型的工具

术语:• USE CASE

- actor 以及
- 4个表达关系的概念：关联、包含、扩展、泛化

工具:USE CASE图



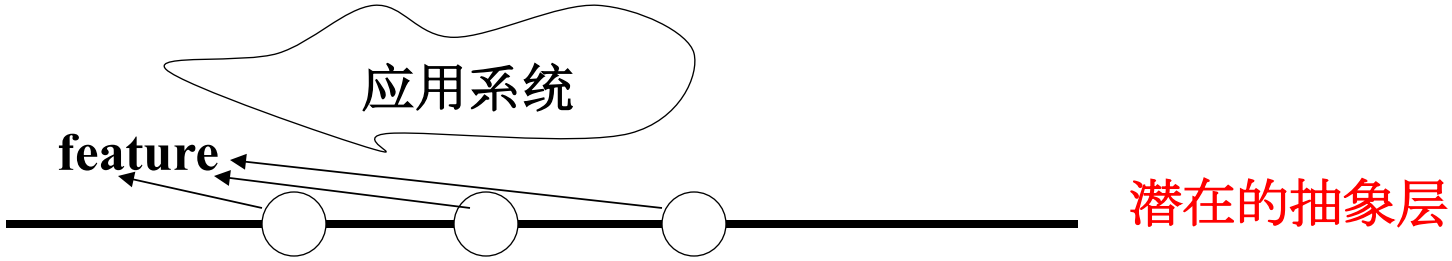
②如何映射--需求工作流

总体来说，需求工作流包括以下四步，但它们并非是严格分离的。

要做的工作	*产生的制品*
-列出候选的需求	特征（Feature）列表
-理解系统语境	领域模型或业务模型
-捕获功能需求	Use case 模型
-捕获非功能需求	补充需求或针对一些特定需求的use cases

其中:❶特征（Feature）列表

特征:一个新的项（item）及相关的简要描述（shrinks），称为特征（feature）。

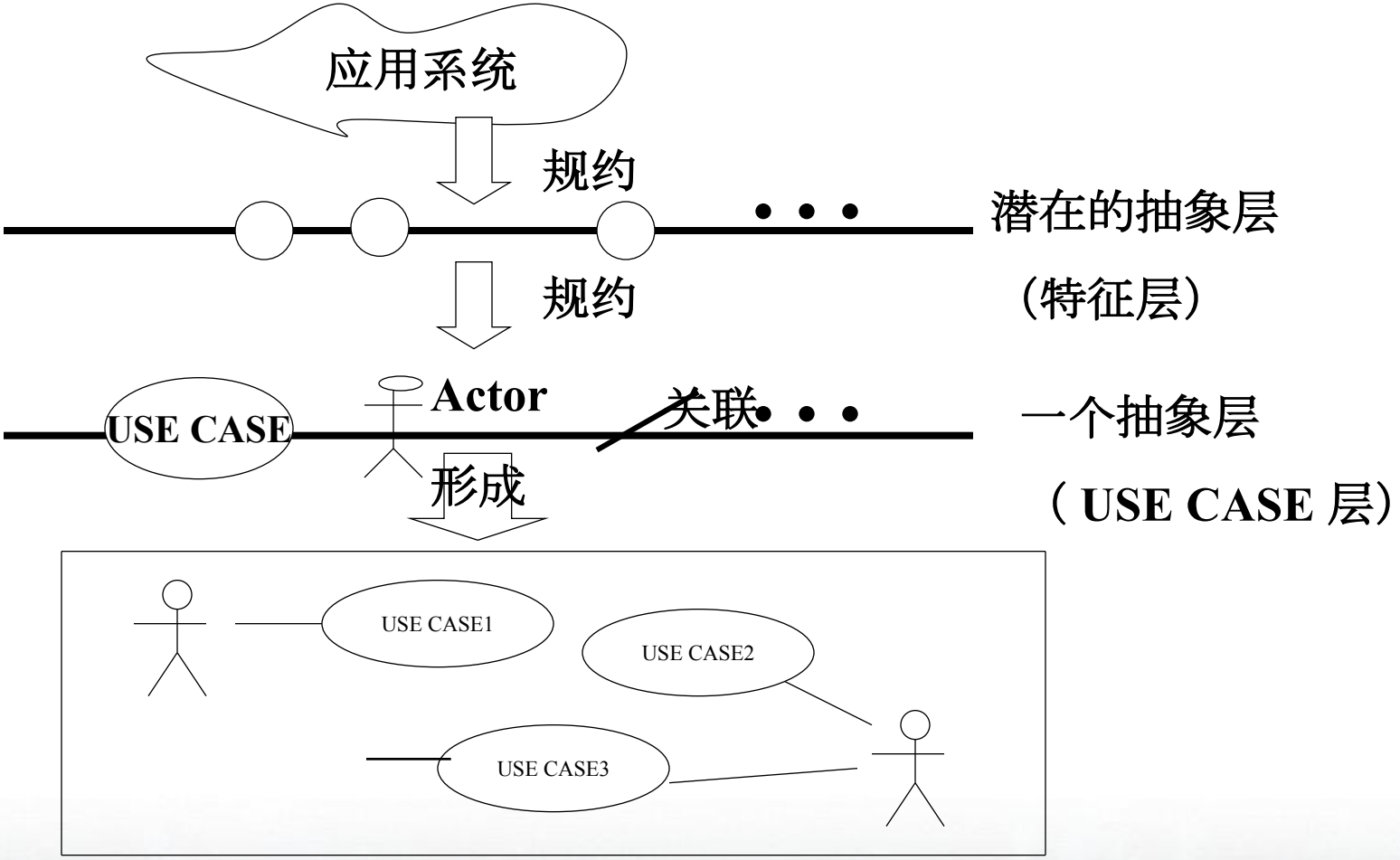


例如：按学科计算每一学生的期末考试平均成绩。

统计2科以上不及格的人数。

给出各分段（0-60，60-85，85-100）的人数分布情况。

特征作为需求, 被转换为其它制品:



制品：USE CASE模型

关于特征的几点说明：

-每一特征有一个简短的名字和简要的说明或定义。

-每一特征还有一组对规划有意义的信息，可以包括：

- ◆状态（Status），例如，提交，批准，确认是否是组成的等。
- ◆ 估算的实现成本。(所需的资源类型和人/时）。
- ◆ 优先级（Priority）(e.g.,critical, important, or ancillary)。
- ◆ 实现中相联的风险等级。

❷ 业务模型或领域模型

领域模型

作用:领域模型捕获了系统语境中的一些重要对象类型。其中领域对象表示系统工作环境中存在的事物或发生的事件

一般来说，领域类以三种形态出现：

◆业务对象：表示那些被业务所操纵（manipulate）的事物（thing），例如定单，帐目和合同等

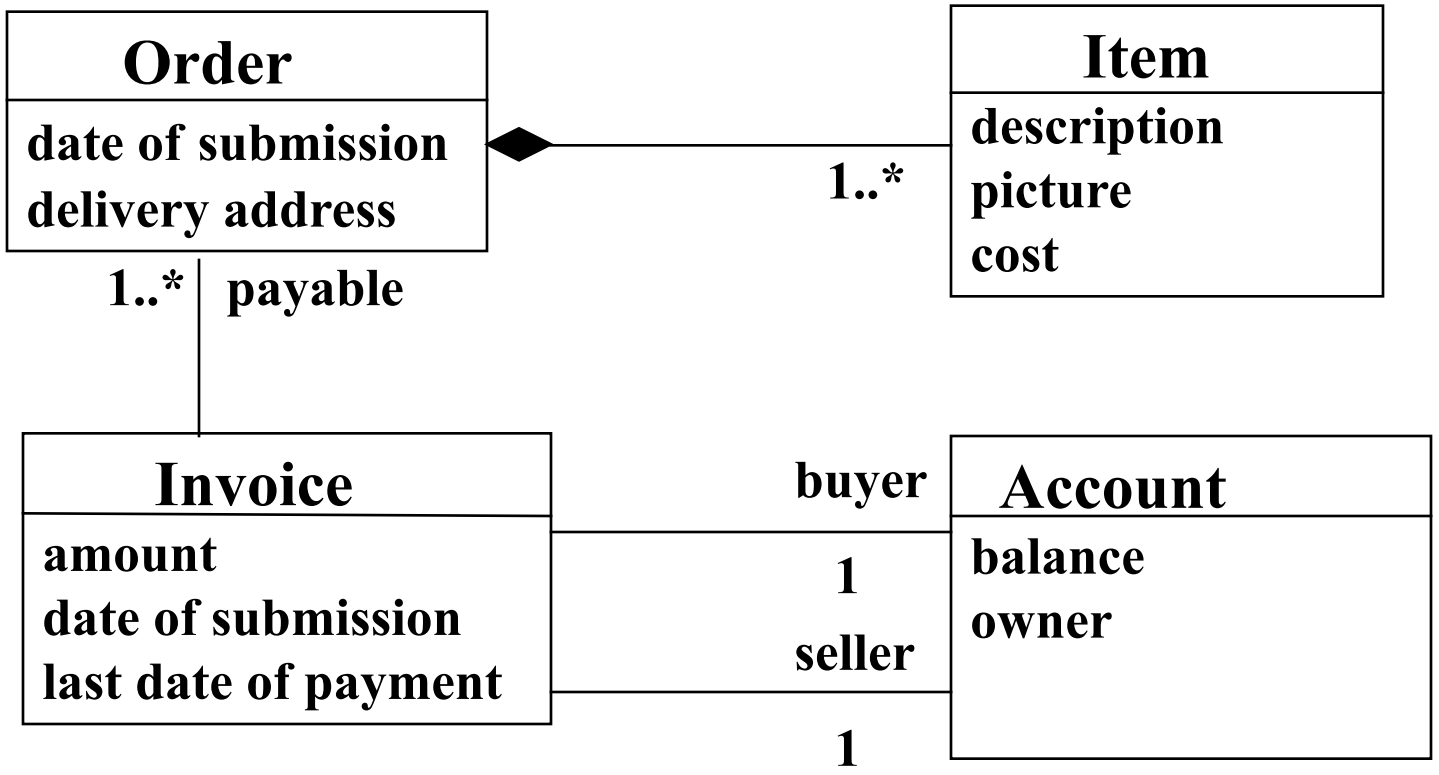
◆实在对象（Real-world objects）和概念：例如飞机，火箭等

◆事件（Events）：例如飞机到达，飞机起飞等

一般来说，领域模型是以类图予以描述的。

例如: 领域类

Order, Invoice, Item, and Account



注:领域模型中的一个类图,捕获了该系统语境中的一些重要的概念.

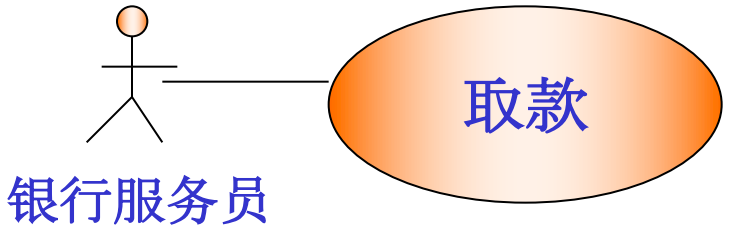
业务模型

业务模型可以分为以下2个层次：

- 业务 use case 模型

  抽象一个特定业务.一般可用USE CASE图来表达,其中以业务use case来描述该业务的处理（business processes）, 以业务 actors来描述与该业务交互的客户（customers ）.

  例如:



- 业务对象模型

业务对象模型是一个业务的内部（interior）模型。通过一组workers 、 business entities 、 work units来细化业务use case模型中的每一个业务use case。

其中， *Business entity* :表示某些事物（ something）, 例如 一张发票
它们在一个业务use case中被使用之。

work unit :是这样实体的一个集合，对最终用户而言，形成了可认知的整体。

workers: 使用该业务use case的工作人员.

注:Business entities 和 work unit可作为领域类，用于表达领域中的概念，例如定单，栏目，发票等。

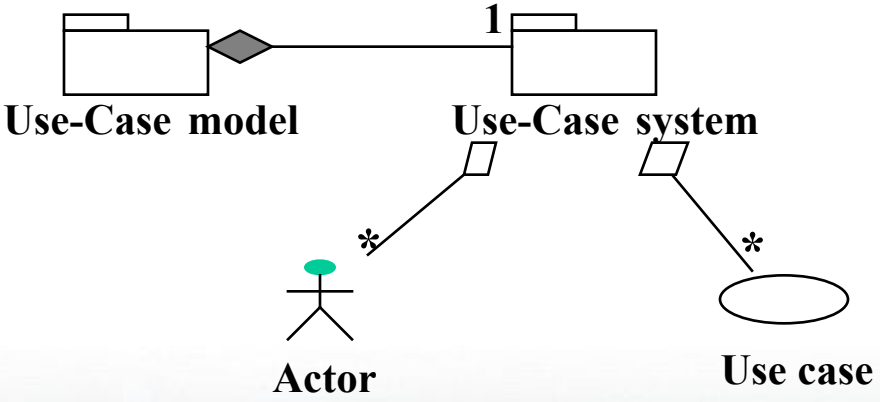
一般地,每一个业务use case的细化可以通过交互图和活动图来表达。

❸Use-Case 模型

◆ Use-Case 模型 用以表达客户认可的需求-系统必须满足的条件和能力。

◆ Use-Case模型 作为客户和开发人员之间的一种共识。

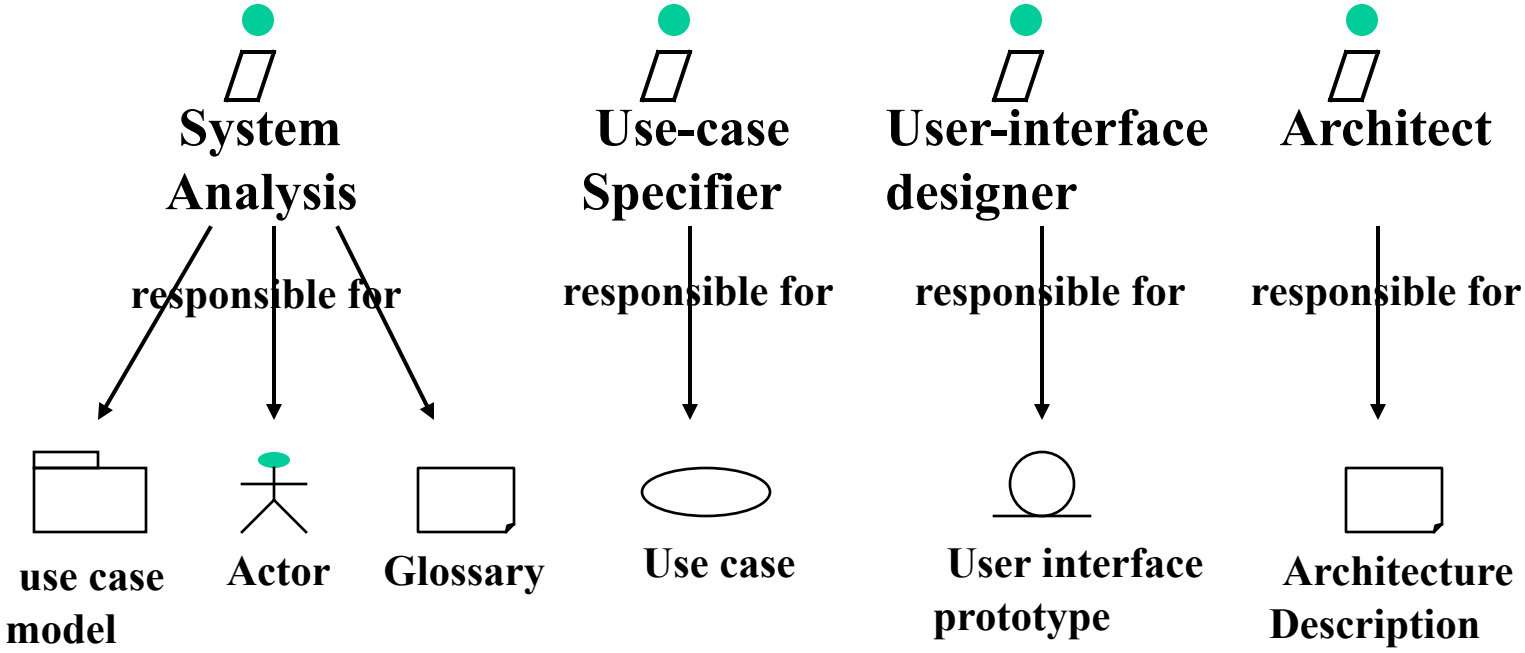
◆ Use-Case 模型是一个系统的一种模型，包括 actors、use cases 以及它们之间的关系。



The Use-Case system denotes the top-level package of the model

Use-Case 模型以及其内容

参与需求工作流的有关人员



③构造系统USE CASE模型中的活动

活动1:发现并描述Actor和USE CASE

任务1:发现 Actor

- 当存在业务模型时,可直接发现一些候选的 actors，即：
  - 对于业务中的每一个worker，可以建议一个候选的 actor
  - 对于每一个将要使用该信息系统的业务actor,即每一个业务客户，可建议一个候选的actor。
- 当有或没有领域模型时

分析人员就要与客户一起标识 actor，并将所标识 actor进行分类，形成 一些候选的 actors。

Note:还要标识表示外部系统的actor和系统维护和运行所需要的actor 。

针对候选的actors ,可用以下2条准则来确定最终系统actors：

第一条准则：至少要识别出一个用户，可以扮演候选的actor。

该准则将帮助我们仅发现那些相关的actors，避免actor 仅是一些想象的“事物”。

第二条准则：系统中不同actors 实例之间，其角色的重叠应是最少的。

其中,如果2个或多个actors有着几乎相同的角色，那么就应该考虑：

- 是否将这些角色组合到一个actor的角色中，或
- 是否需要发现另外一个“一般化”的actor，使之具有那些重叠的、公共的角色，并可以通过“泛化”，形成那些特定actor。

任务2:Actors的命名与描述

Actors的命名：

对actors的命名,可“传达”（convey）所期望的语义,因此给出恰当的名字是非常重要的。

Actors的描述：

对actor的描述，应包含其角色（责任）以及为了完成其责任所需要的条件。

例如：the *Buyer*, *Seller*

Buyer

A *Buyer* represents a person who is responsible for buying goods or services as described in the business use case Sales: from Order to Delivery. This person may be an individual or someone within a business organization. The Buyer of goods and services need the *Billing and Payment System* to send order and to pay invoices.

Seller

A *Seller* represents a person who sells and delivers goods or services. The Seller uses the system to look for new orders and to send order confirmations, invoices, and payment reminders.

任务3:发现Use Case

对 use case的理解

在UML中，用况是系统向它的参与者提供结果（值）的功能块，表达参与者使用系统的方式，因此一个用况可用于规约系统可执行的、与参与者进行交互的一个动作序列，包括其中的可选动作序列，用况并有其自己的属性。

一个用况的实例，其行为表现为：

--首先，系统外部的一个参与者实例，直接启动该用况实例中的一条路径，并使之处于一个开始状态；

--继之,执行这条路径中的动作序列，使该用况实例从一个状态转化为另一状态；其中，在执行该序列的每一动作中，包含内部计算、路径选择和向某一参与者发送消息；

--在一个新的状态中，等待actor发送另一外部消息，以此引发该状态下的动作之执行。

--如此继续，经历了许多状态，直到该用况实例被终止。

其中,由于我们把系统用况模型中的用况实例看作是原子的，因此只存在一类发生在参与者实例和用况实例之间的交互。这意味着用况实例不能与其它用况实例发生交互，但每个用况的行为可以被其它用况所中断。并且在大部分情况里，是一个参与者实例引发一个用况实例，但也有可能由一个事件所引发，例如由系统之外的定时时钟所引发。

--当已有一个业务模型时，可直接标识一些临时的用况,即为其中的一个“*工作人员*”或“*业务参与者*”的角色，对应地创建一个用况;

继之,对这些暂时的用况进行细化和调整，并确定*工作人员*和*业务参与者*的哪些任务可由系统自动地予以实现，而后对确定的用况进行重新组织，以便更好地适应系统参与者的的要求。

--当没有业务模型时，就要与客户和/或用户一起来标识用况。其中，需要一个一个地审阅参与者,为每一个参与者建议一些侯选的用况。例如，研究一个参与者需要哪些用况，为什么需要这些用况，并研究参与者的创建、改变、跟踪、迁移等工作，通常需要哪些用况来支持之，研究*业务用况*中使用的业务对象，例如定单和帐目等。

不论采用什么方法，在发现候选用况中还需要进一步考虑一些问题，例如：

●参与者是否还可能要通知系统一些外部事件，包括已经发生的一些事件，例如：发票已经过期。

●是否还可能存在一些其他的参与者，他们执行系统的启动、终止和维护。

●是否存在一些侯选的用况，不宜作为系统最终的用况，而应把它们作为其他用况的组成部分。

●创建的用况是否可以作为一个功能单元，容易修改、测试和管理之等。

●对用户和系统之间的一个交互序列，是否在一个用况中予以规约，还是在多个用况中予以规约等。

在确定系统最终的用况当中，会涉及一个很难处理的问题，即用况范围大小的确定。其中必须考虑：它是否是完整的（complete），是否是另一用况的继续.

下面给出两条确定用况大小的基本准则：

第一条准则：用况应为它的参与者产生有值的结果 (*result of value)*，特别地，用况应向一个特定的参与者交付了可见的结果（值）.

这一准则的目的是为了避免发现的用况太小。

第二条准则：用况最好只有一个特定的参与者(*particular actor)*。

这条准则的目的是为了使所标识的用况都有一个真实用户，从而可以避免发现的 用况太大。

注:基于一些参与者而第一次发现的用况，常常需要予以重新组织，重新评估，使之更加“稳定”。例如，

假定已经有了一个体系结构描述，那么就要对新捕获的用况进行必要的调整，以便适应已有的体系结构。

任务4:用况的描述

对用况的描述程度，取决于是在什么时候。一般或在发现时，或在需要精化时（参见4）任务3）。

在用况发现时，其描述一般应该：

首先给出该UAE CASE的名字；继之，给出该用况的概要描述。根据需要，概要描述一般可以采用两种形式，一种是简单地给出用况的功能，例如：

“Pay Invoice Use Cases

“The use case Pay Invoice is used by a Buyer to schedule invoice payments. The Pay Invoice use case then effects the payment on the due date.”

第二种形式是首先概括其动作，而后一步一步地列出系统与其参与者交互时所要做的事情。例如：

“Before this use case can be initiated, the Buyer has already received an invoice (delivered by another use case called Invoice Buyer) and has also received the goods or services ordered.:

1. The buyer studies the invoice to pay and checks that it is consistent with the original order.

2. The Buyer schedules the invoice for payment by the bank.

3. On the day payment is due, the system checks to see if there is enough money in the Buyer’s account. If enough money is available, the transaction is made.”

通过以上活动1：

--形成了系统的粗略用况模型，记为用况模型[概述]。

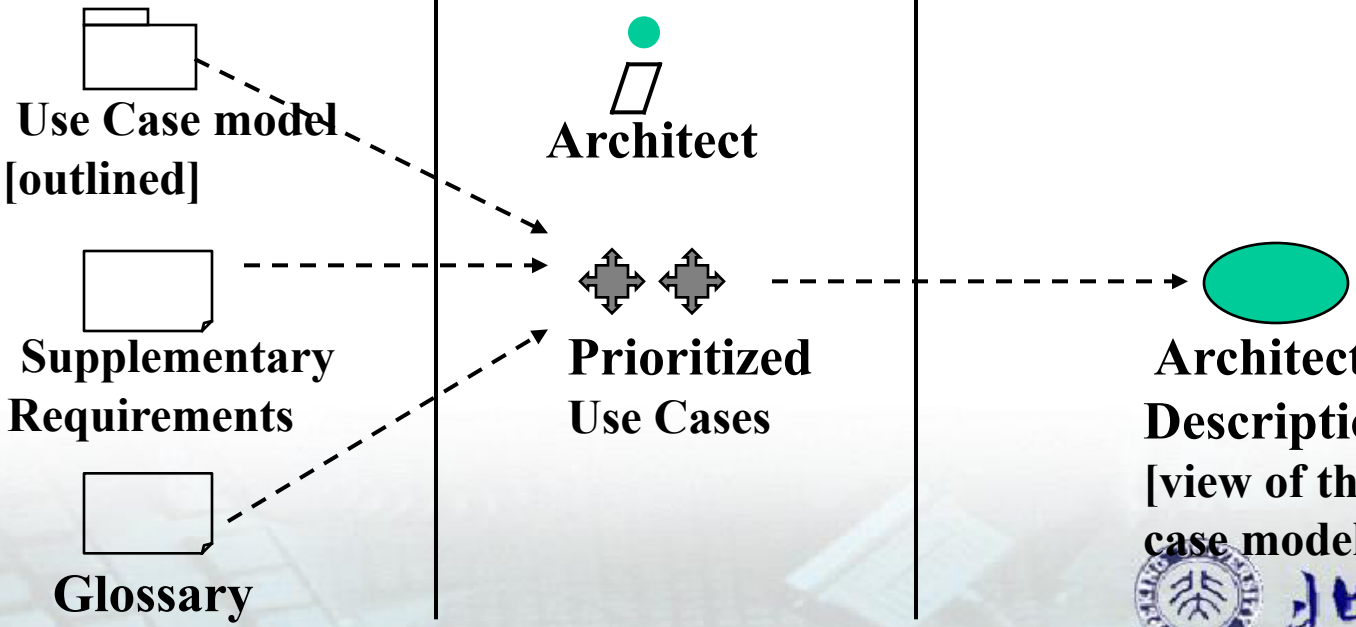
--以这一用况模型[概述]为此基础上，进而可形成系统的一些非功能需求和相应的应用术语集。

活动2:确定use case的优先级（Priority）

❶目标

这一活动目标是，在活动1）和2）的基础上，确定哪些用况适宜在早期迭代中予以开发，哪些用况适宜在后期迭代中予以开发，并形成系统用况模型[概述]视觉下的体系结构描述。

❷输入与输出



❸实施

--刻画在体系结构方面具有意义的用况，包括:

♦对某一重要、关键功能的用况的描述;

♦包括对那些必须在软件生存周期早期予以开发的、某一重要的用况的描述。

❹作用

--确定在规划的下一个迭代中，需要哪些开发活动和任务;

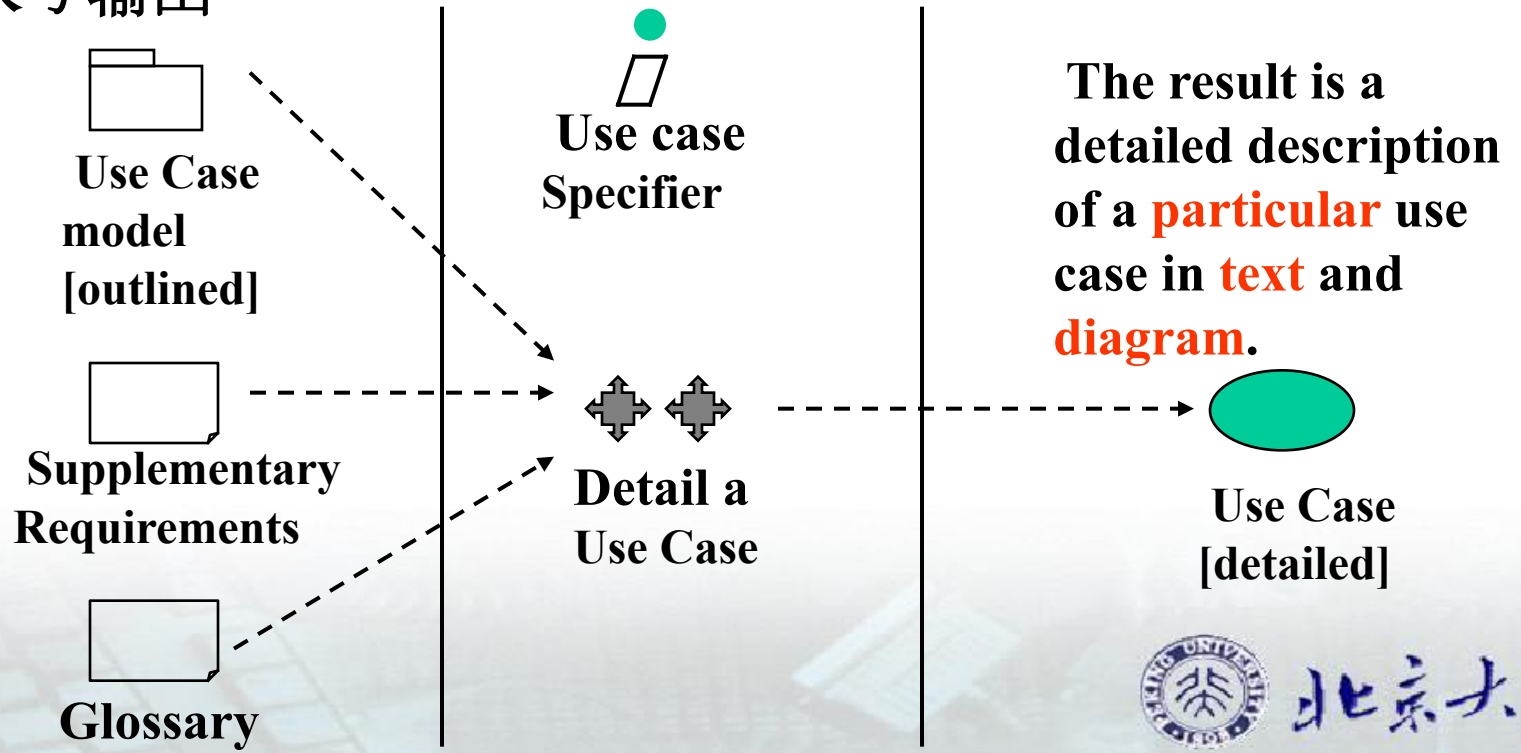
--在这一规划中,当然还需要考虑其它非技术因素，例如系统开发的业务和经济方面的因素。

活动3: Use Case精化

❶目的

详细地描述事件流，包括use case是怎样开始的，是怎样结束的，是怎样与actors 进行交互的。

❷输入与输出



❸精化途径

涉及:◆如何描述一个 use case中所有可选的路径；

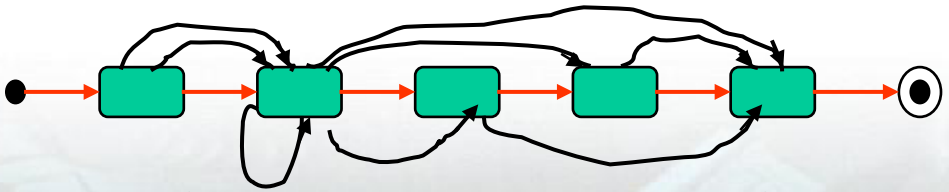
◆在一个 use case的描述中包括的内容；

◆ 如何在必要时形式化地给出use case的描述。

有效技术：事件流技术

对用况的精化描述，可以用事件流（Flow of Events）描述技术，规约当一个用况执行时，系统做了什么，以及系统如何与其参与者参与者进行交互。

一个用况可以被认为有一个开始状态，一些中间状态，并从一个状态转换为另一状态，如下所示：



其中，红箭头线表示基本路径，曲线是其它可选路径。

首先，从开始状态到终止状态选择一条完整的基本路径，并对其进行描述。其中，这一基本路径的选择应该是用户认为它是一条最通常的、并对相关参与者产生最有意义值的路径。一般来说，这一基本条路径还应包含系统的一些例外和异常处理。

接之，描述其余可选路径。其中，是否把其中那些很小的可选路径，作为基本路径的组成部分，还是作为一条独立的路径，这是一个设计决策问题，取决于该描述是否精确，是否容易阅读。

例如：“Paths of the Pay Invoice Use Case

Precondition: The buyer has received the goods or services ordered and at least one invoice from the system. The buyer now plans to schedule the invoice(s) for payment.

Flow of Events

Basic Path

1 The buyer invokes the use case by beginning to browse the invoices received by the system. The system checks that the content of each invoice is consistent with order confirmations received early(as part of the Confirm Order use case) and somehow indicates this to the buyer. The order confirmation describes which items will be delivered, when , where, and at what price.

2 The buyer decides to schedule an invoice for payment by the bank, and the system generates a payment request to transfer money to the seller's account. Note that a buyer may not schedule the same invoice for payment twice.

3 later, if there is enough money in the buyer's account, a payment transaction is made on the scheduled date. During the transaction, money is transferred from the buyer's account to the seller's account, as described by the abstract use case Perform Transaction(which is used by Pay Invoice). The buyer and the seller are notified of the result of the transaction. The bank collect a fee for the transaction, which is withdrawn from the buyer's account by the system.

4 The use case instance terminates.
Alternative Path
In Step 2, the buyer may instead ask the system to send an invoice rejection back to the seller.In Step 3, if there is not enough money in the account, the use case will cancel the payment and notify the buyer.

Post-condition: The use case instance ends when the invoice has been paid or when the invoice payment was canceled and no money was transferred. ”

一般来说，在用况的一个精化描述中，其基本内容包括：

① 定义一个前置条件，用于表达该用况的开始状态；

② 定义第一个要执行的动作，例如上例中的 Step 1，描述该用况是如何开始的，什么时候开始。

③ 定义该用况中基本路径所要求的动作及其次序。上例中以次序号（Step 1-4)）定义了动作及其执行次序；

④ 描述该用况是如何结束的，什么时候结束（例如 Step 4）；

⑤ 定义一个后置条件，用于表达可能的结束状态；

⑥ 描述基本路径之外的可选路径；

⑦ 定义系统与参与者之间的交互以及它们之间的交换 (例如Step 2 和Step 3)，即描述该用况的动作是如何被相关参与者激发的，以及它们是如何响应参与者的要求。其中，如果该系统与其它系统交互，则必须规约这一交互,例如引用一个标准的通讯协议。

⑧ 描述系统中使用的有关对象、值和资源 (例如Step 3)，即描述在一个该用况的动作序列中，如何使用为该用况属性所赋予的值。

总之,在用况描述中，必须明确描述系统做什么（执行的动作），描述参与者做什么，并从参与者做什么中分离出系统的责任。否则用况描述就不够精确。

注意：

❶一个事件流的描述应包括一组管理上适于修改、复审、设计、实现和测试的动作序列，并作为用户手册的一节内容。

❷对每一个用况的事件流，一般采用正文来描述其中的动作序列，只有当该用况的动作序列和/或该用况与其参与者的交互较为复杂时，才可使用活动图、状态图或交互图来描述之，而且在实际工作中的很多情况下，正文描述和这些图之间往往是相互补充的。

❸不管采用什么描述技术，都必须描述所有的可选路径，否则就不能说给出了该用况的规约。

❹规约人员应当：

紧密地与该use case的实际用户一起工作；在与其交谈中，通常需要记录他们对该use case 的理解,并与其讨论建议方案。

只有当满足以下条件时，才能说结束了用况描述：

- 用况是很容易可理解的；
- 用况是正确的（即捕获了正确的需求）；
- 用况是完备的（例如，描述了所有可能的路径）；
- 用况是一致的。

用况的描述可以在需求捕获结束的复审会中，由分析员予以评估，也可以由用户和客户予以评估。但仅客户和用户才能确认用况是否是准确的。

关于半形式化的Use-Case描述

- *前置条件*

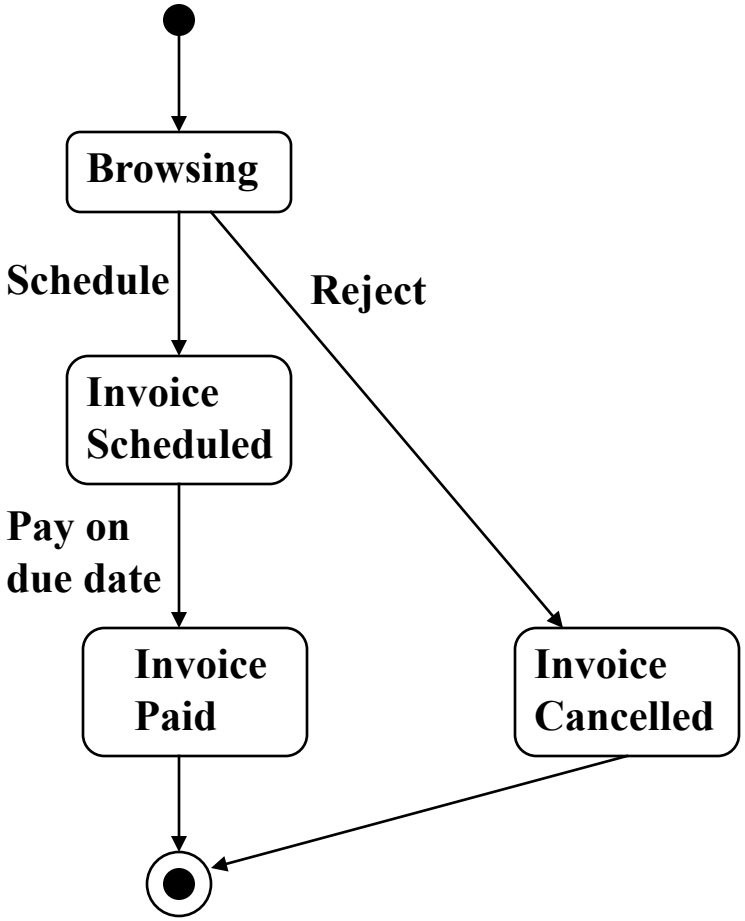
对于一个复杂的实时系统， use case可能是相当负责的，例如actors和 use case 之间的交互可能经过相当多的状态和状态转换，从而几乎不可能保持正文描述的use case 的一致性。

- *相关的技术*

为此，需要使用更结构化的描述技术，这样的技术一般使用可视化的建模技术，来描述use cases 。以下技术可以帮助系统分析人员更好地理解use cases ：

❶UML 状态图

用于描述use case的状态和状态之间的转换。



The statechart diagram for the Pay Invoice use case showing how an instance of the Invoice use case moves over several states in series of state transitions.

❷活动图

用于描述状态之间更详细的动作序列。

注：活动图源于SDL的状态转换图（SDL state transition diagrams），它是已予证明的、用于电信的一种语言。

❸交互图

用于描述一个use case的实例如何与 actor 的一个实例进行交互。为此，交互图给出了use case 以及参与的actor（s）。

注意:

•在使用这些图当中，由于问题的复杂性，有时可能会出现一些大的、复杂的图，以至于很难阅读和理解，特别对于那些不是软件开发人员来说更难阅读。

•这些图是一些更接近开发细节的图，应与系统其它模型保持一致。

建议：

应谨慎地使用这些图，在一般情况下，应采用 **use case**的正文描述（即事件流描述）。

在很多情况中，正文描述和这些图是互补的。

活动4:用户界面的原型构造

❶ 目的

建造一个用户界面的原型，使用户有效地执行use cases。

❷步骤

第一步，用户界面的逻辑设计

第二步，物理用户界面的设计

第三步，开发用户界面原型，演示为了执行该use case，用户怎样使用该系统。

注：如何进行以上三步，可参见有关文献。

活动5 :Use Case 模型的结构化

前置条件:

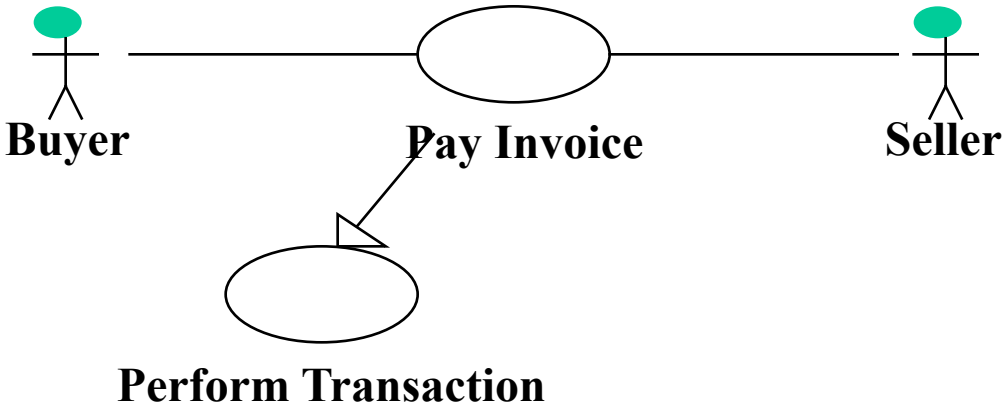
- 系统分析员已经标识了actors 和 use cases, 已经以图予以了描述，并给出了整个use case 模型的说明。
- use case 规约人员已经对每一use case开发了详细的描述。

做什么：

- 抽取use case描述中一般的、共享的功能，用于特定use case描述。
- 抽取use case描述中附加的或可选的（additional or optional）功能，它们可能被扩展为特定use case描述。

❶使用泛化关系，标识并描述那些共享功能

例如：

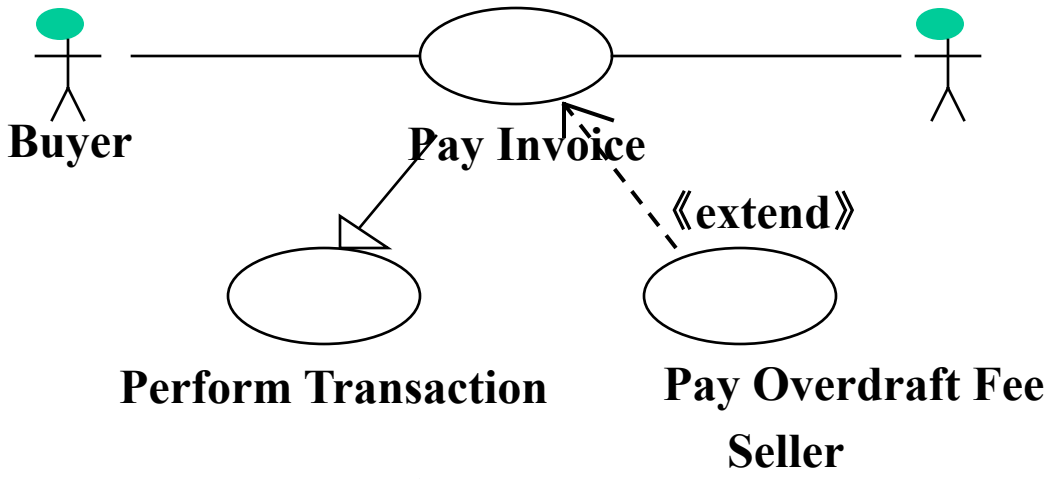


Pay Invoice 和 Perform Transaction

这2个 use case之间的泛化关系

❷使用扩展关系，标识并描述附加的或可选的功能

例如：



Pay Invoice 和 Pay Overdraft Fee

这2个use cases之间的扩展关系

❸标识use case之间的其它关系

use cases之间还包括其它关系，例如包含关系（*include*）。

在用况的结构化中，应注意以下问题：

- 在建立用况的结构中，应尽可能地反映用况的实际情况。否则，不论对用户或客户，还是对开发人员本身，要理解这些用况以及它们的意图就变得相当困难。
- 在用况的结构化中，不论是施加什么结构，对新引入的用况都不应太小或太大。因为，在以后的开发中，对每一个用况都需要对其做进一步处理，成为一个特定的制品。例如在需求获取阶段，用况规约人员需要给出它的描述，而在后续的分析和设计中，设计人员需要对不同的用况予以细化（realization），这样，如果用况太小或太大，势必产生一系列管理上的问题。

●在建立用况的结构中，应尽量避免对用况模型中的用况功能进行分解。最好在分析和设计中对每一用况进行精化（**refining**），其中如果需要的话，以面向对象风格把用况中所定义的功能作为概念分析层上对象之间的协作，这样可以对需求产生更深入的理解。

通过对用况模型的结构化，最终形成系统的一个精化用况模型，记为用况模型[精化]。

RUP需求获取小结(四点)

❶ 需求获取以及相关制品

work to be done	*result artifacts*
-List candidate requirements	Feature list
-Understand system context	Business or domain model
-Capture functional requirements	Use case model
-Capture nonfunctional requirements	Supplementary requirements or individual use cases (for use case specific req.)

❷业务模型或领域模型

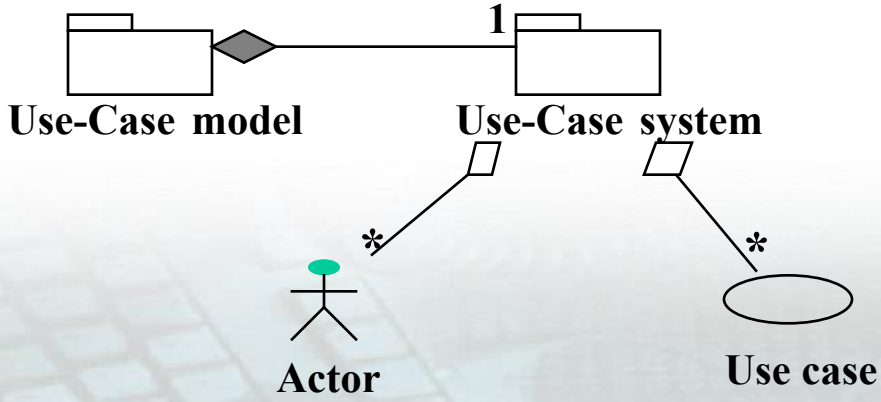
建立了系统的语境，是创建系 统use case 模型的基础。

❸use case 模型

◆ Use-Case Model 是软件和客户就需求的一个共识，即系统必须具有的条件（conditions）和能力（capabilities） 。

◆ The Use-Case 模型为系统分析、设计、实现以及测试提供了基本的输入。

◆ A Use-Case 模型是系统的一个模型，包含系统中 actors 、use cases 以及它们之间的关系。即：



The Use-Case model and its contents. The Use-Case system denotes the top-level package of the model

◆ use-case 模型捕获了功能需求。非功能需求特定于单个的use case，其规约具有一般性，并不针对一个特定的use case 。

◆ use-case 模型的描述，可以通过：

- a survey description

-a detailed description of each use case.

◆对于use case 模型中的每一 use case ， 应驱动开发的后续工作，并实现它们的无缝连接。即在分析阶段和设计阶段，应标识相匹配的 use-case 细化（realization）， 并标识测试阶段中的一组测试用例（ test cases ）。

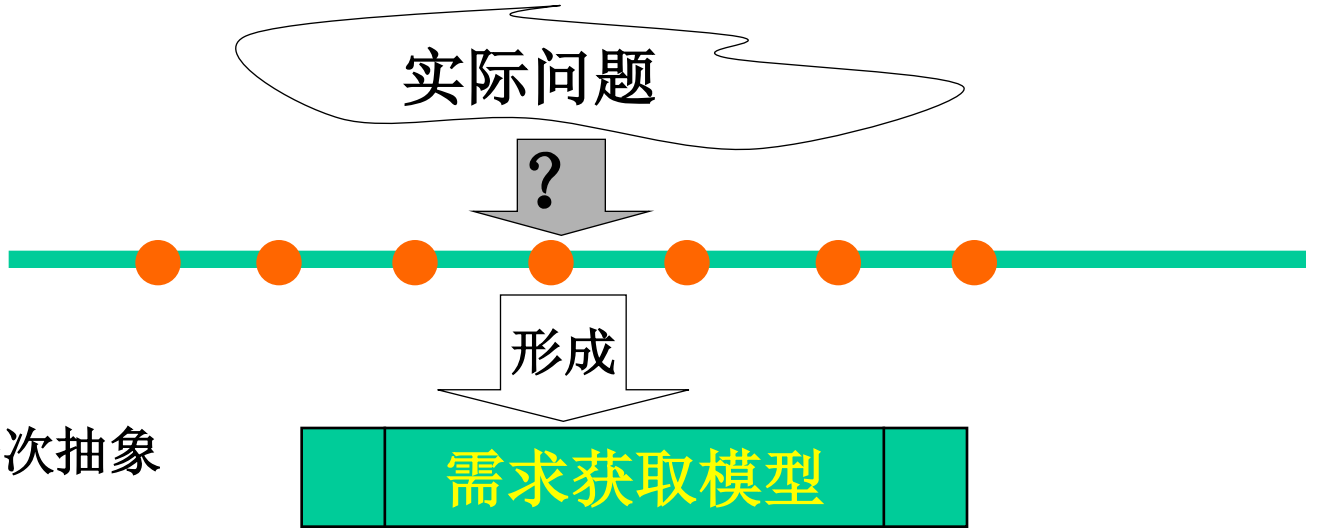
❹捕获需求阶段的活动

序号	输入	活动	执行者	输出
1	业务模型或领域模型,补充需求, 特征表	发现参与者和用况	系统分析员、客户、用户、其他分析员	用况模型$_{概述}$，术语表
2	用况模型$_{概述}$，补充需求，术语表	赋予用况优先级	体系结构设计者	体系结构描述 用况模型角度
3	用况模型$_{概述}$，补充需求，术语表	细化用况	用况描述者	用况$_{详述}$
4	用况$_{详述}$，用况模型$_{概述}$，补充需求，术语表	人机接口原型化	人机接口设计者	人机接口原型
5	用况$_{详述}$，用况模型$_{概述}$，补充需求，术语表	构造用况模型	系统分析员	用况模型$_{详述}$

3) 需求分析的目标及其途径

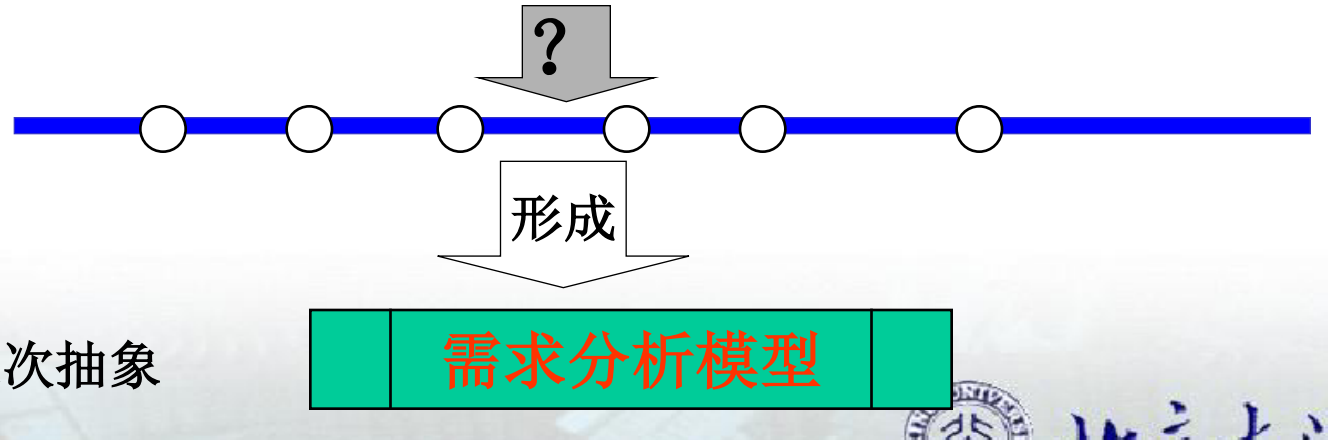
(1)目标:

需求获取



注：实现第一次抽象

需求分析



注：实现第二次抽象

(2) 实现需求分析目标的基本途径

实现需求获取层到需求分析层的映射，即从软件开发的角度-实现第二次抽象。如同实际问题层到需求获取层一样,其中至少涉及以下3个问题：

①如何定义需求分析层，即给出该层的术语；

②如何确定模型表示工具；

③如何映射。

①需求分析层的术语（概念）

术语:•分析类（Analysis class）

•Use Case细化（Use Case Realization-Analysis）

•分析包（ Analysis Package）

其中：

❶分析类（Analysis class）

一个分析类抽象地表达了

系统设计中 的一个或多个类和/或子系统。

分析类的基本性质：

• 分析类关注处理功能需求，而将非功能需求的处理延迟到以后的设计和实现活动中，并作为类的特殊需求。

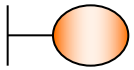
- 分析类的行为一般是通过高层的责任予以定义的(一个责任是高内聚的一组由类所定义的行为的正文描述),很少通过操作和其声明（signatures）予以定义或提供接口。
- 分析类的属性也是在很高层次上定义的。这类属性经常是问题域上的概念，并可通过问题域就可以了解其含义。
- 分析类所涉及的关联，多数是概念性的，例如关联的导航性，在分析中并非十分重要，而在设计中就是基本的。

目的：使分析类在问题域中更加突出，与设计和实现中的类相比，粒度大，是概念性的。

分析类的种类：

三种：边界类（Boundary classes）　　实体类（ Entity classes）

控制类（ Control classes）

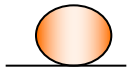
--边界类(Boundary classes): 

内涵：用于模型化 系统和其actors 之间的交互。该交互一般涉及向用户和外部系统发出请求和从他们那里接受信息。

与设计平台的关系：边界类常常是在更高的概念层上，对windows, forms, communication interfaces, printer interfaces, sensors, terminals, and APIs 等的抽象，忽略其中的一些细节-例如用户窗口的宽度，并且不需要描述该交互的物理实现（realize）。

基于的设计原理：分离不同的用户界面或通讯界面，形成一个或多个边界类。

--实体类（Entity classes）：



内涵：用于模型化 那些需要长期足留系统的对象-例如人的信息,以及与行为相关的某些现象—例如实际的一个事件。

与业务类的关系：

在大多数情况下，实体类对应业务模型中的业务类。其中一个主要区别是：现在所考虑的实体类，一般是要由系统处理的那些对象。

与设计平台的关系：

实体类一般表示一个逻辑数据结构和属性，以理解系统依赖什么信息。

基于的设计原理：

分离待处理的不同信息,形成一些不同的对象。

--控制类（Control Classes）：

内涵 :用于模型化 系统的动态性（dynamics）, 包括:

- 用于表达协同、定序、事务以及对其它对象的控制；
- 用于封装那些与特定 use case 有关的控制；
- 用于表达复杂的推导和计算，例如不与 任何存贮在系统中的特定信息有关的业务逻辑.

简言之,控制类处理并协调边界类对象和实体类对象的基本动作（actions）和控制流，并向它们委派工作。

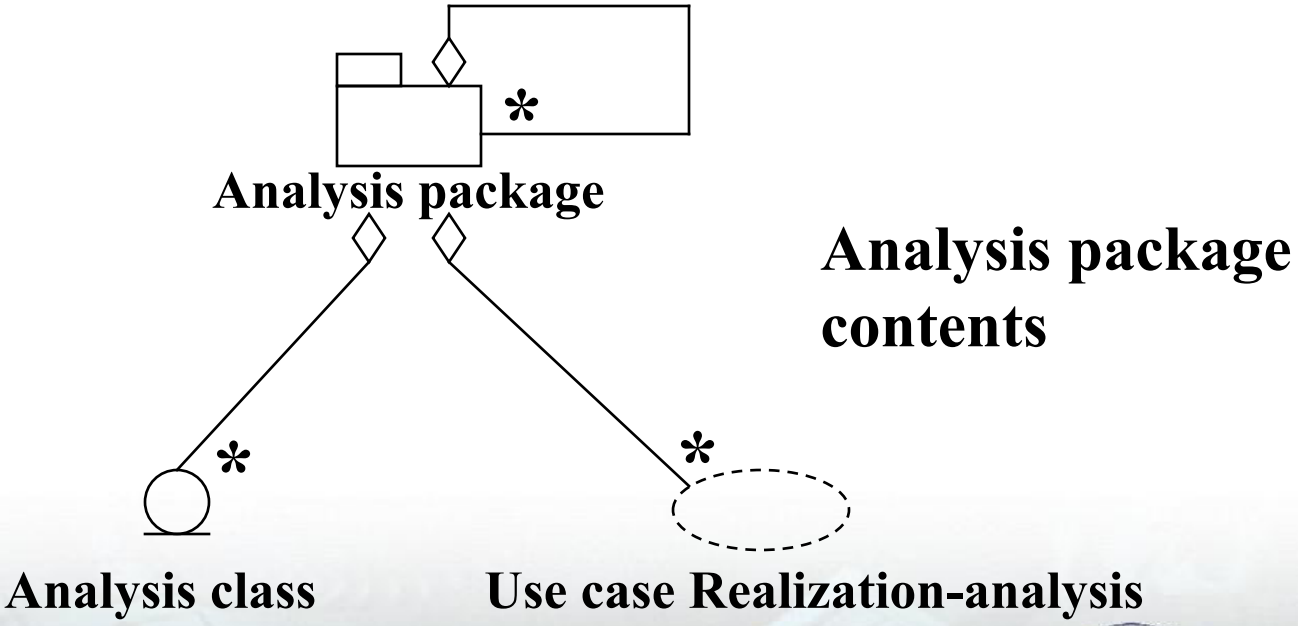
基于的设计原理： 问题分离

控制类分离一些不同的控制、协调、定序以及不同的复杂业务逻辑，并予以封装，形成一些所谓的控制类，其中一般要涉及其他一些对象。其中而不能封装那些与actors交互有关的问题（由边界类予以封装）也不能封装那些与系统处理的信息有关的问题（由实体类予以封装）

❷分析包（Analysis Package）

分析包提供了一种组织分析制品的手段，形成一些可管理的部分。

分析包可包含分析类、 use case 细化以及其它分析包：



分析包的主要特征（ **characteristic**）

- 高内聚、低耦合；
- 表达了对所要分析问题的一种分离；例如，将不同领域知识分为不同的包予以分析;
- 对具有领域知识的人来说，是可以阅读、理解的。这是由于包是针对功能需求和问题域予以创建的;
- 很有可能成为一些子系统或成为一些子系统的组成部分。在某些情况中，甚至可以反映一个完整的顶层设计。

❸ Use-Case 细化（Use-Case Realization-Analysis）

内涵（何谓 Use-Case 细化？）

一个Use-Case 细化是分析模型中的一个协作（a collaboration），描述了一个特定的 Use-Case如何运用分析类以及分析类的交互对象进行细化和执行 。

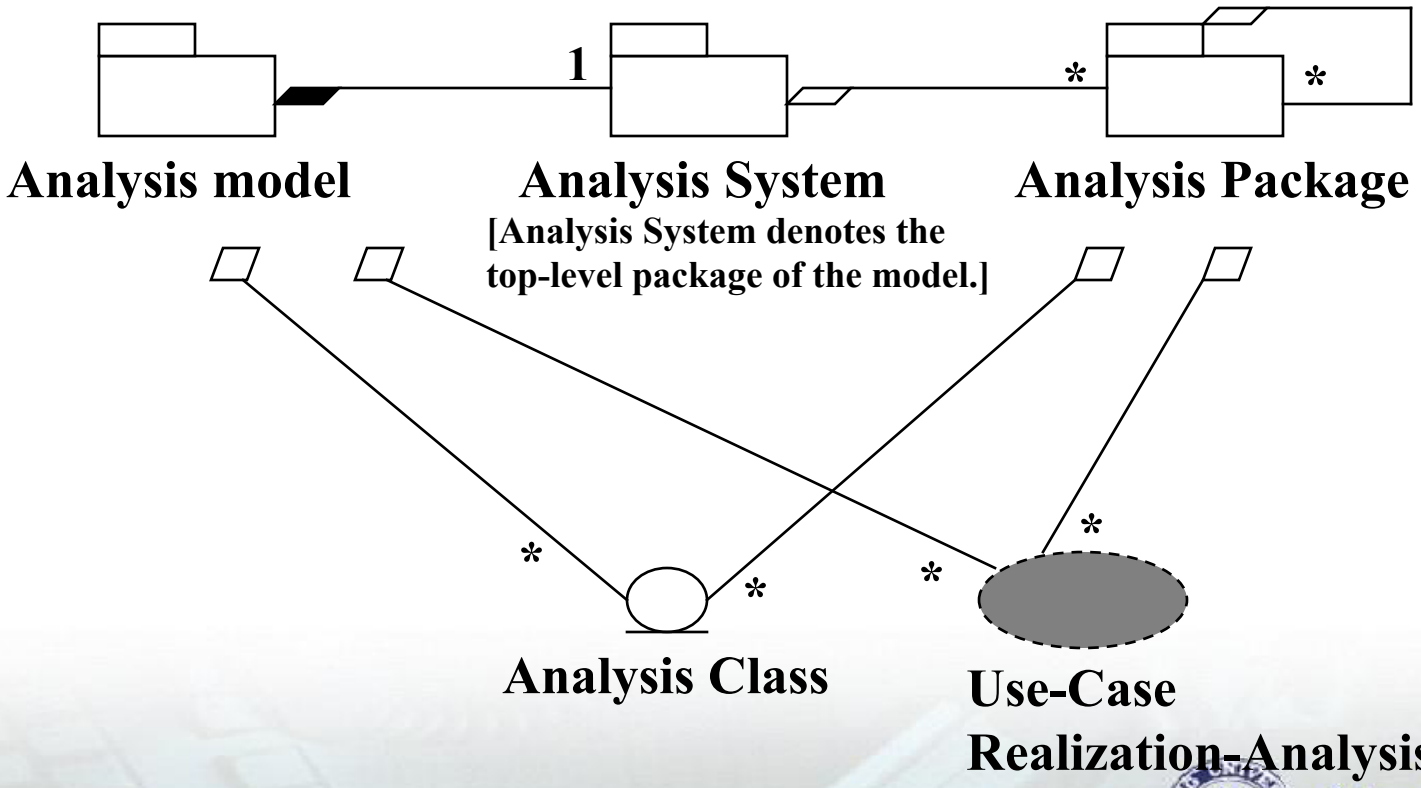
作用： Use-Case 细化对use-case模型中的一个特定的 use case 提供了一中直接方式的跟踪。

如何表达Use-Case 细化？

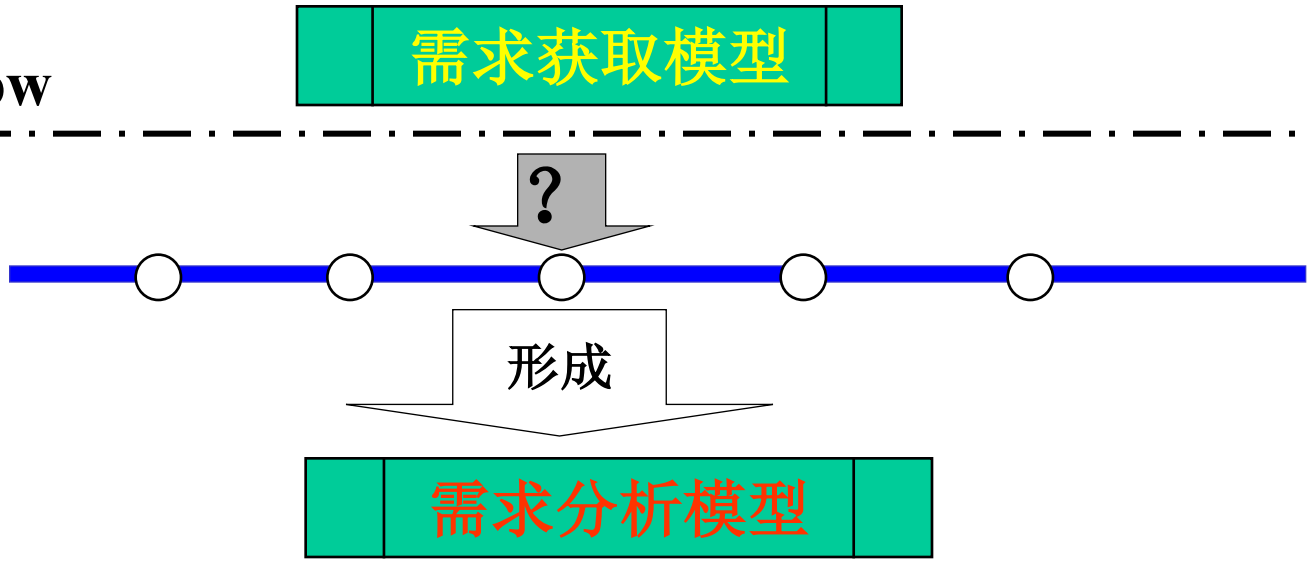
- ♦正文的事件流
- ♦类图
- ♦交互图

②模型表达

分析模型是分析包的一个层次结构，包含分析类和use-case细化。



③ Workflow



活动1:体系结构分析（Architectural Analysis）

目标：通过标识 ❶分析包，

❷ 分析类，

❸ 公共的特定需求，

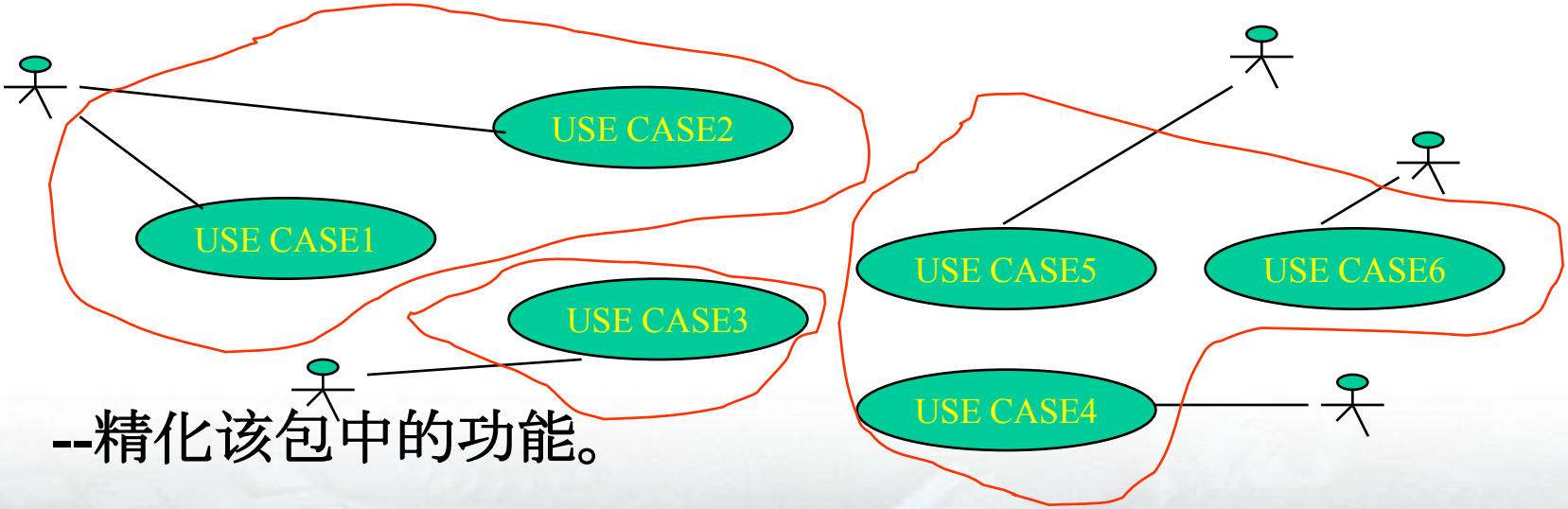
建立分析模型和体系结构的“骨架”

任务1：标识分析包

基本输入： use case 模型

--基于功能需求和问题域，即考虑需要的应用和业务，对分析工作进行划分，形成一些分析包。

--依据use cases的主要功能（方面），把一些use cases 分派到特定的包中。



--精化该包中的功能。

其中, use cases的分派， 需要考虑以下问题,以支持包的高内聚：

❶一个包中的 use cases 是否支持一个特定的业务过程；

❷一个包中的 use cases 是否支持一个特定的系统actor。

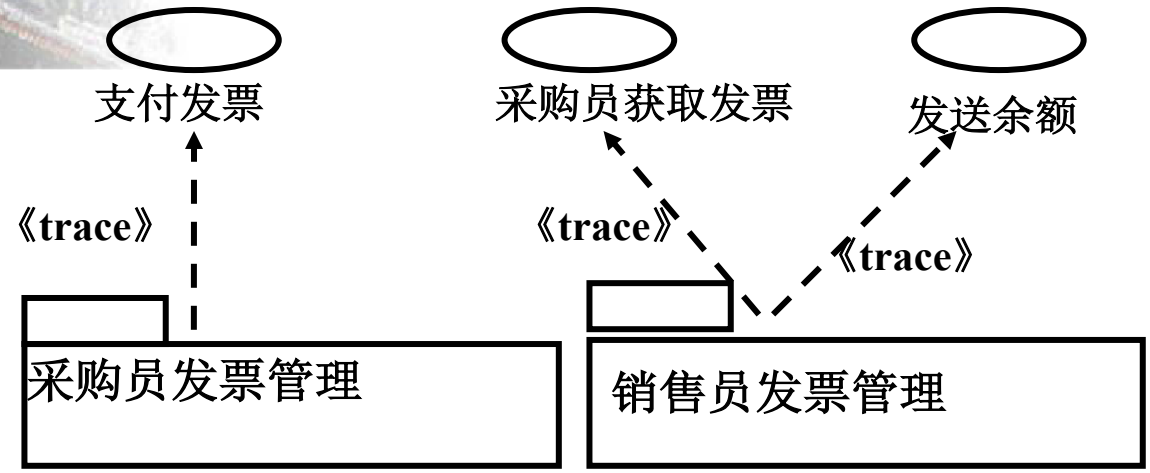
❸一个包中的 use cases 是否具有“泛化”和“扩展”关系。

注意：

•一个分析包将一些变化分别局部到（ localize to ）一个业务过程， 一个actor的行为， 一组高耦合的use cases

•通过 use cases细化， 可以演化包的结构。

例如：

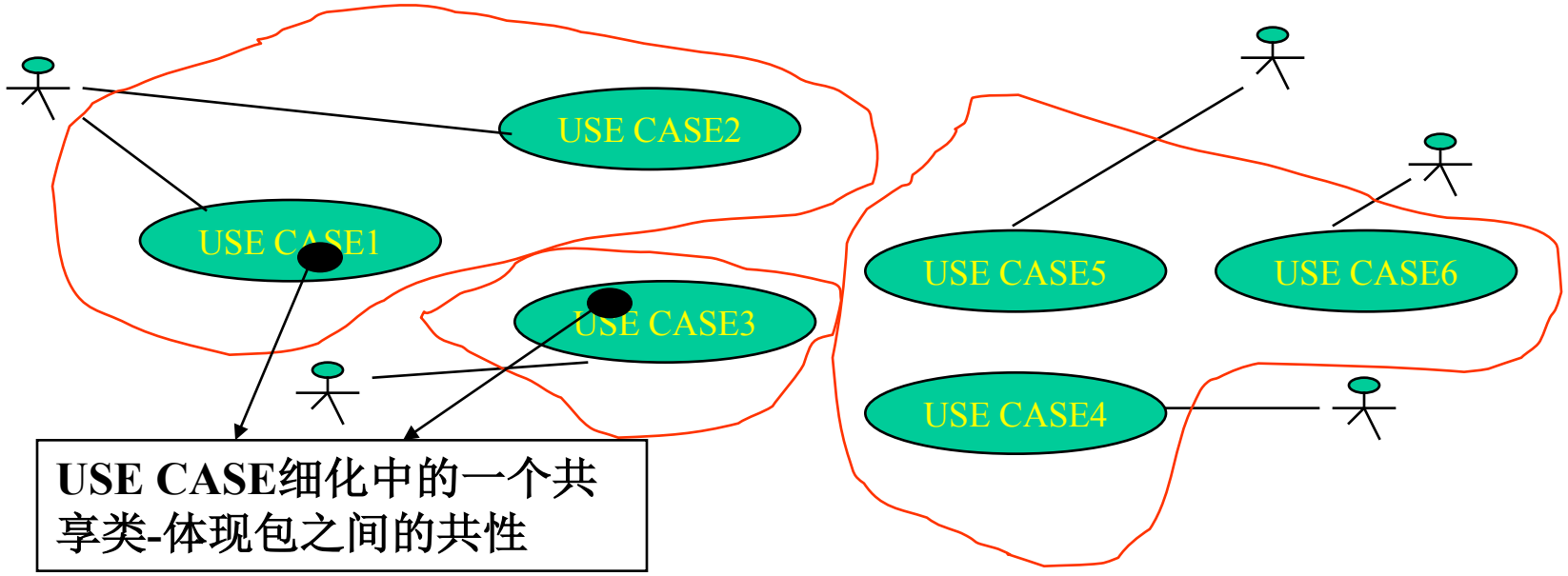


该例表明，如何基于网络银行的用况模型来标识它的分析包,其中三个用况：“获取发票的采购员”、“支付发票”、“发送余额”，紧紧围绕“买卖”业务中的有关发票处理问题，因此，可以把这些用况作为一个分析包。

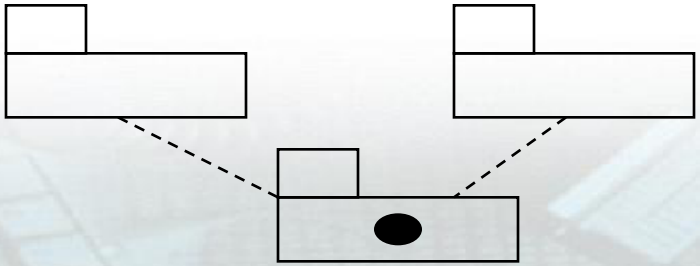
但是，网络银行软件面对的是具有不同要求的客户，有的客户仅是采购员，而另一客户仅是销售员，而有的客户既是采购员，又是销售员。因此，为了满足客户的这一需求，就需要把这三个用况分为如图的2个分析包：“采购员发票管理”和“销售员发票管理”。

注意：支持“买卖”业务过程还有其它用况，但为了使例子简单而予忽略之。

任务2：处理分析包之间的共性



方法：抽取共享类，继之把共享类放到一个包中，并让其它包依赖该类或该类所在的更一般化的包。



任务3：标识服务包（Service Packages）

在包的层次结构中，除了考虑以上那种对“共性”的依赖外，时常还要考虑服务依赖，即应用层的包依赖下层提供的服务包。

何谓服务，在RUP中服务是指功能上紧密相关的、可被多个用况使用的一组动作。

一个服务的主要特征是：

♦服务是不能分割的，或系统完整地提供之，或不提供之；

♦服务是针对客户而言的，而用况是针对用户而言的，即当细化一个用况时，可能需要多个服务，即一个用况需要多个不同服务中的动作。

何谓服务包:由服务所组成的包称为服务包。

显然服务包是一个功能包，通常包含一组功能相关的类。

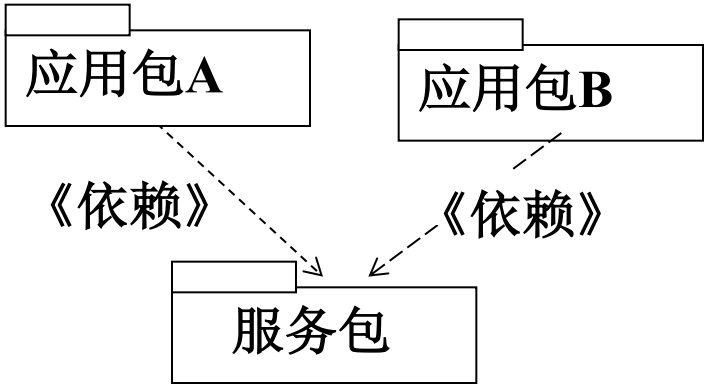
服务包的主要特征是：

- 是不可分离的，即如果客户需要这一包，就要其中的所有类；
- 一般只涉及一个参与者或很少几个参与者；
- 可构成以后设计和实现的一个基本输入，可形成一个服务子系统，以支持设计模型和实现模型的结构化；
- 所定义的功能，往往可以作为设计和实现中的一个发布单位，因此它所表达的功能可能是系统的内嵌（“add-in”）功能；
- 服务包可独立执行，对于同一服务的不同方面，可以由系统的2个不同服务包提供，例如：

  “英文的拼写检查”；　“美语的拼写检查”
- 服务包之间的依赖，通常是非常受限制的。

可见，服务包的引入，可用于系统的结构化处理，即：

依据服务包所提供的服务，把它放到分析包层次结构中的低层，以便应用包对它的引用。



当系统功能需求得到了很好理解，并且已存在很多分析类的情况下，才能有效地标识服务包。

标识服务包的基本方法:

- ♦或依据客户对有关服务的要求，
- ♦或直接将多个功能相关的分析类所提供的服务，标识为一个服务包。

例如，假定：

- 类A的一个变化，很可能要求类B的一个变化；
- 把类A拿掉，使类B成为多余的；
- 类A的一个对象与类B的一个对象，可能需要以多个不同的消息发生交互。

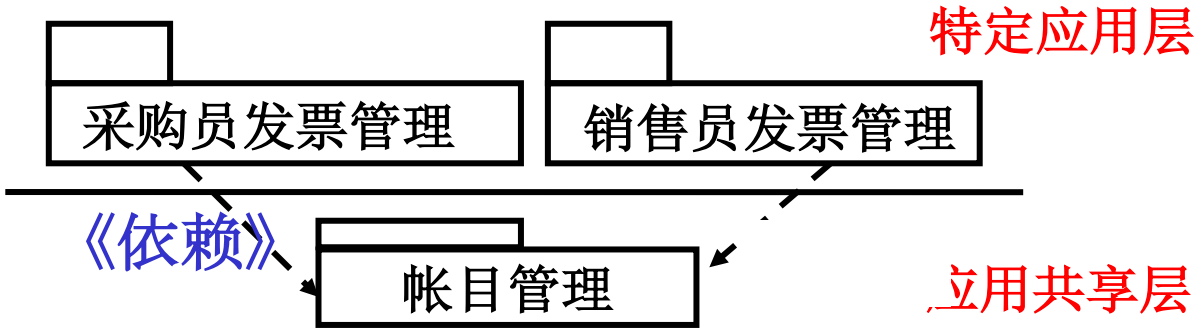
这时就可以将类A和类B所提供的服务标识为一个服务包。

任务4 ：定义分析包的依赖

目标：发现相对独立的包，实现包的高内聚和低耦合。

途径:通常使用特定应用包和公用包,把分析模型分为两个层面,清晰地区分特殊性功能和一般性功能。

例如:



分析包的层次和依赖

通过以上任务1、2、3和4，就可以形成对体系结构是具有重要影响的分析包的层次结构。

任务5：标识重要的实体类

目标:标识在体系结构方面具有重要意义的实体类。为此：

途径:

首先，基于需求捕获中标识的领域类和业务实体，发现其中对体系结构具有意义的类，作为分析模型中的实体类，以此形成体系结构的初始骨架。一般情况下,即使对于一个比较大的系统，这样的类也只在10个-20个左右。

此时，不必标识过多的类，也不必了解更多的细节，因为在以后对use-case进行细化，标识其中实际需要的实体类时，可能还需要重复以上的工作。

其次，使用领域模型中领域类之间的聚合和关联，或使用业务模型中业务实体之间的聚合和关联，来标识这些实体类之间一组暂定的关联。

任务6：标识公共的特定需求

在分析期间，不论是对已标识的包还是对已标识的分析类，为了支持以后的设计和实现，都可能会出现一些特殊需求。例如对以下问题的约束和限制：

- 永久性；
- 分布与并发；
- 安全性;
- 容错能力
- 事务管理等。

因此，该任务的目标是标识每一特殊需求的关键特征。

通过以上六项任务的实施，其中系统化地使用了有关分析包、服务包、依赖、关键实体类等术语，给出了系统的体系结构描述，称为分析模型视角下的体系结构描述，记为体系结构描述[分析]。

可见,分析模型视觉下的体系结构描述,描述一些在体系结构方面具有重要意义的制品。一般包括：

- 分析包以及它们的依赖
- 一些关键的分析类

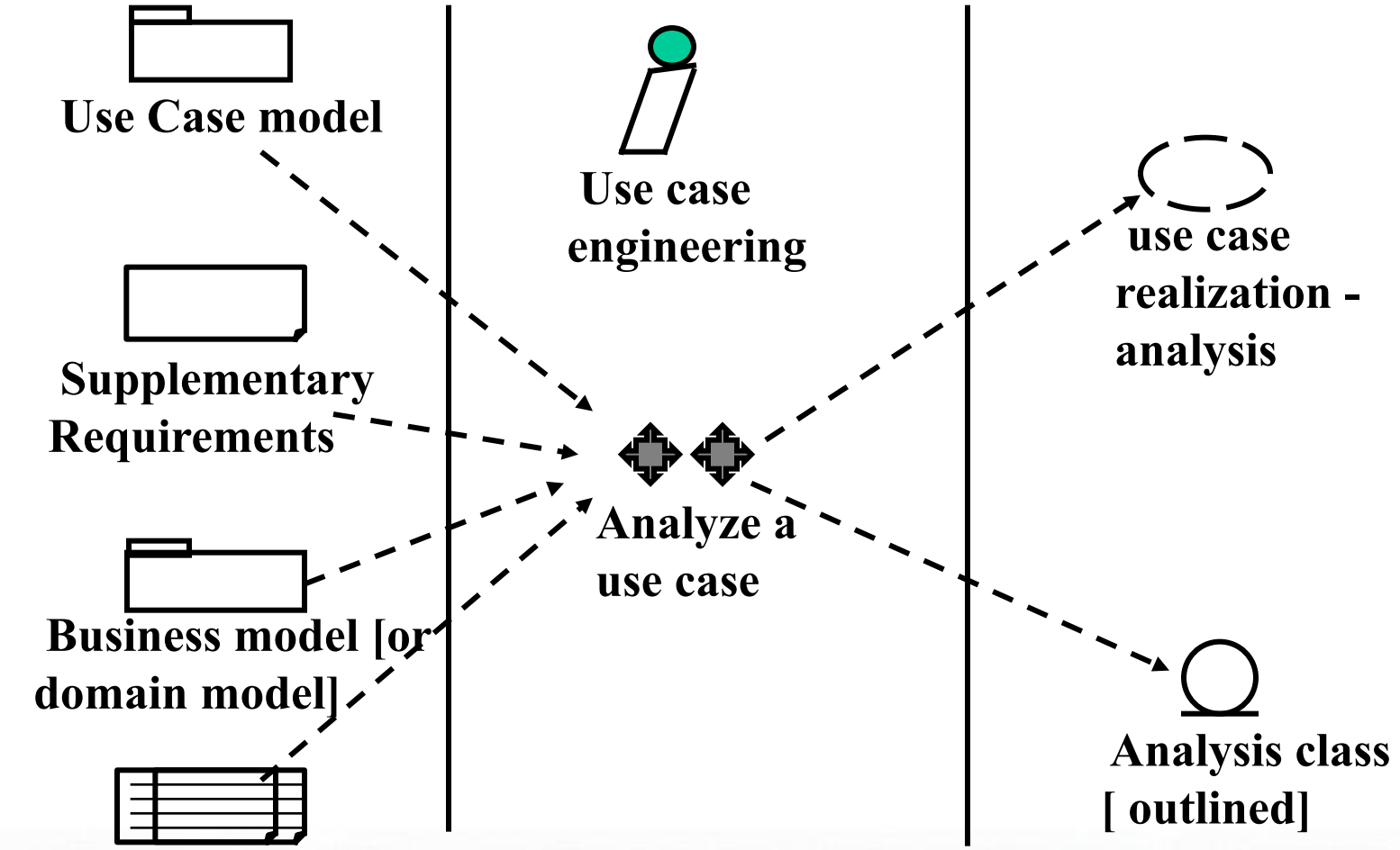
活动2: Use Case分析

目标：

- 标识那些在use case事件流的执行中所需要的对象和类。
- 将 use case的行为，分布 （Distribute）到交互的分析对象。
- 捕获use case细化上的特定需求。

（Another term for use case analysis would be use-case refinement. We refine each use case as a collaboration of analysis classes.）

分析一个 use case的输入和输出



任务1：标识分析类

标识那些细化use case所需要的实体类、控制类和边界类。并给出它们的名字、责任、属性和关系。

--准备工作

首先应从系统之外的角度，来精化用况的事件流正文描述.

--一般性指导：

❶ 标识实体类

在活动1的任务5中，已经标识了一些对体系结构具有重要影响的实体类，在此基础上，基于该用况的事件流正文描述，发现其中其余逻辑对象（即“大”对象），并依据类的定义，来标识其它实体类。

其中,应:

- 学习/研究use case的描述；
- 更深入地研究已有的领域模型；
- 考虑在 use-case细化中还需要什么信息，以实现其功能

还要思考:,

是否应把这些信息标识为边界类、控制类的属性；

是否在use case细化中需要这些信息；

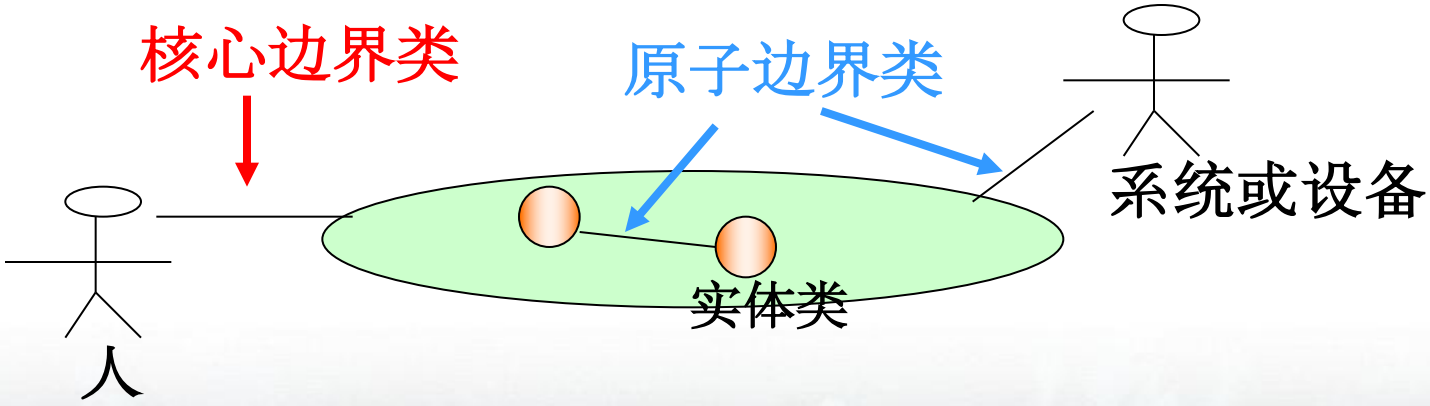
如果是的话，这样的“信息”就不应作为实体类。

❷ 标识边界类

在标识该用况的边界类中，可把边界类分为核心边界类和原子边界类:

核心边界类:是指系统与人（参与者）进行交互的接口；

原子边界类:是指系统与外部系统、设备等进行交互的接口，或实体类之间进行交互的接口。



- 核心边界类的标识

为每一个与该用况交互的人（参与者），标识一个核心边界类，并用这个类来表示用户接口的基本窗口（**primary window**）。其中，应考虑以下**2**个问题**:**

第一个问题是：如果该参与者已经与已标识的某一边界类进行交互，那么就应该考虑是否复用那个边界类，减少基本窗口的数目；

第二个问题是：考虑是否在这一核心边界类与其他边界类之间存在泛化关系，尤其重要的是，考虑这一核心边界类是否是某些原子边界类的父类。

●原子边界类的标识

--对以前发现的每一个实体类，如果它们所表达的一些逻辑对象在相应的用况执行期间， 参与者（人）需要通过一个核心边界类与这些逻辑对象进行交互，那么就为这样的实体类标识一个原子边界类。继之， 可依据不同的可用性准则，对这些原子边界类进行精化，以利于创建“好”的用户接口。

--对该用况的每个外部系统的参与者,标识一个原子边界类,用于表达通信界面。其中，如果该通讯涉及多层协议，那么就有必要区别对待这些不同的层次，为所关注的不同层标识不同的边界类。

❸标识控制类

为负责处理该用况细化的控制和协调，标识一个控制类，并依据该用况的需求，精化这一控制类。其中应当注意：

- 有时，由控制类负责的一些控制最好封装在一个边界类中，特别地，如果当相关的参与者在这一控制中其到很大作用时，可能就不需要控制类。
- 有时，由控制类负责的控制是相当复杂的，这时最好把这样的控制封装在2个或多个控制类中。

在完成实体类、边界类和控制类的标识之后，一般可采用一个类图，汇聚参与该用况细化的所有分析类，并在其细化中给出中所使用的各种关系。

任务2：分析（类）对象交互的描述

工具：

使用交互图来描述（类）对象之间的交互。

途径：在创建一个交互图当中，

首先，确定细化该用况所必要的交互，这一般应从用况的流开始，一次经过一个流。其中涉及参与交互的分析对象和参与者实例，通常在交互的定序中，自然就能在任务中所标识的分析类中发现参与交互的客体。例如，

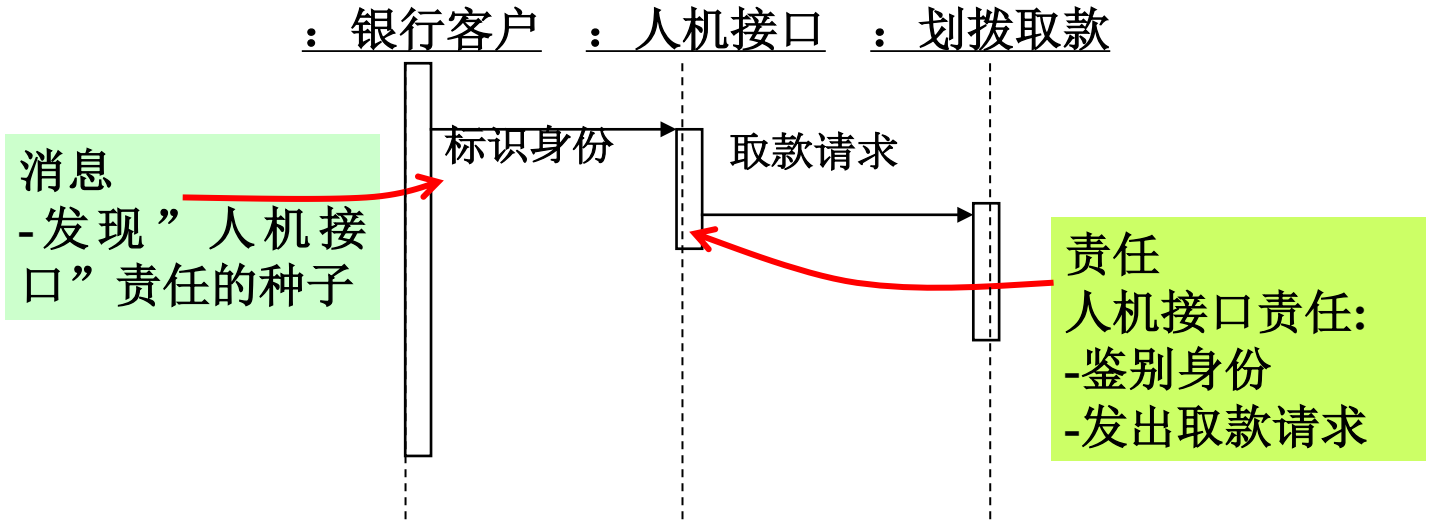
“银行客户”和用况“取款”的交互，所涉及的对象有：

：银行客户，　　：人机接口，

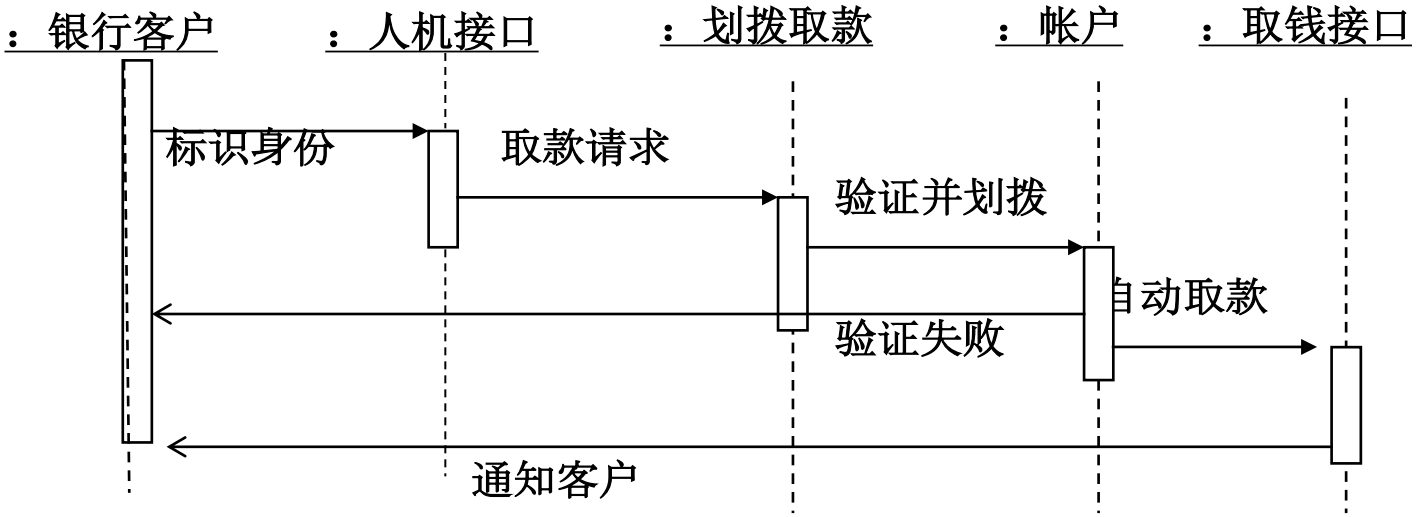
：取钱接口，　　：划拨，

：帐户

其次，分派该用况的功能，这一般应基于激活该用况的一个消息--作为“种子”，来发现相关对象的责任。例如



最后，再根据其责任，发现该交互图中各个链。其中，应在与这一用况细化相关的交互图中或在类图中，明确给出一个关联中所有对象之间的链，例如：



注意：

协作图中的链，通常是分析类之间的关联实例。这些关联或已经存在，或以这些链定义了有关关联的需求。在这一步中，应该在与这一 **use-case**细化相关的类图中，给出所有明显的关联

对象之间的链，以及有关每一特定对象的需求（由消息捕获而来的），应该予以特别的关注。

如果 use case 具有不同的、有区别的流或子流，一般需要为每一个 流创建一个交互图。这样：

-可以使 use-case 细化更加清晰；

-有助于使抽取的协作图可用来表达一般性、复用的交互

对系统用况模型中的每一用况实施上述两项任务，就可形成整个系统的用况细化[分析]。

活动3：类的分析

通过活动1和活动2，已经标识了系统中所有分析类，但没有给出它们的详细描述.

目标：一是标识并维护分析类的责任；

二是基于它们在用况细化中的角色，标识并维护分析类的属性和关系；

三是捕获分析类细化中的特殊需求。

一般性指导:

任务1：标识责任

组合一个类在不同用况细化中所扮演的角色.

首先从类的角色中抽取该类的责任，而后基于用况的一次又一次的细化（包括用况细化[分析]，用况细化[设计]等），不断增加其责任或变更已有的责任。

任务2：标识属性

一个属性规约了类的一个特性，一般隐含在类的责任要求之中，因此，标识类的属性就要关注类的责任的要求。

•实体类属性的标识

实体类属性的标识一般特别明显。特别地，如果一个实体类能跟踪到一个领域模型中的领域类，或能跟踪到业务模型中的业务实体类，那么领域类和业务实体类的属性是标识实体类属性的有价值的输入。

- 边界类属性的标识

在标识边界类属性中，对那些与人（参与者）交互的边界类，其属性常常表达由该参与者操纵的信息项，例如，标记的正文栏。而对那些与外系统（参与者）交互的边界类，其属性常常表现为一个通讯接口的性质。

- 控制类属性的标识

由于控制类属性通常具有较短的生命期，因此一般很少标识控制类的属性。但是，在特定情况下，为了表达用况细化期间那些需要计算的值或需要推导的值，那么就要为这些值标识一些相应的属性。

在标识分析类的属性中，应当注意以下问题：

- 所标识的属性，其名字一般为名词；
- 属性类型一定是问题域中的概念，一般不要限定其实现环境；例如，在分析中，“帐”可以是一个属性，而在设计中，与之对应的可以是“整型”。并且在选择一个属性类型时，应考虑复用已有的类型。
- 如果因为属性的原因使类变的非常复杂且不易理解，那么这时就应考虑把其中某些属性分离出来，形成一个“整体/部分”结构。
- 属性的表达一般只简单给出由类处理的性质（property），并且可以在该类的责任描述中给出。其中如果一个类的属性很多或很复杂，可以在类图中仅给出属性框。

任务3：标识关联和聚合

❶标识关联

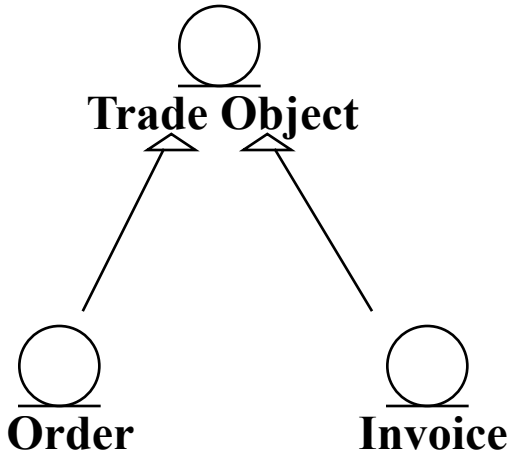
交互图中的链，表达了分析对象与其它对象的交互，因此这些链通常是类之间关联的实例。因此应研究协作图中的链，确定需要哪些关联，并定义关联的多重性、角色名、自关联、关联类、限定角色以及N元关联等。

❷标识聚合

当一些对象表达了一些相互包含的概念，例如一辆轿车，包含一名驾驶员和一些乘客；或当一些对象表达了一些相互组合的概念，例如一辆轿车，由一个发动机和多个车轮组成；或当一些对象表达了客体的一个概念集，例如一个家庭，有父亲、母亲和儿子，这时就把这一事实标识为聚合。

❸标识泛化

由于泛化的基本目的是使分析模型更容易予以理解，因此，具有一般意义的类应在更高的概念层上。例如：



The Trade object generalizes Invoice and Order.

其中，由于定单和发票具有类似的责任，因此可以作为交易对象的一个特殊化对象。

活动4：包的分析

目标:一是确保分析包尽可能与其它包相对独立；

二确保分析包实现了它的目标，即细化了某些领域类或用况；

三是描述依赖，以益于可以估计未来的变化。

一般性指导：

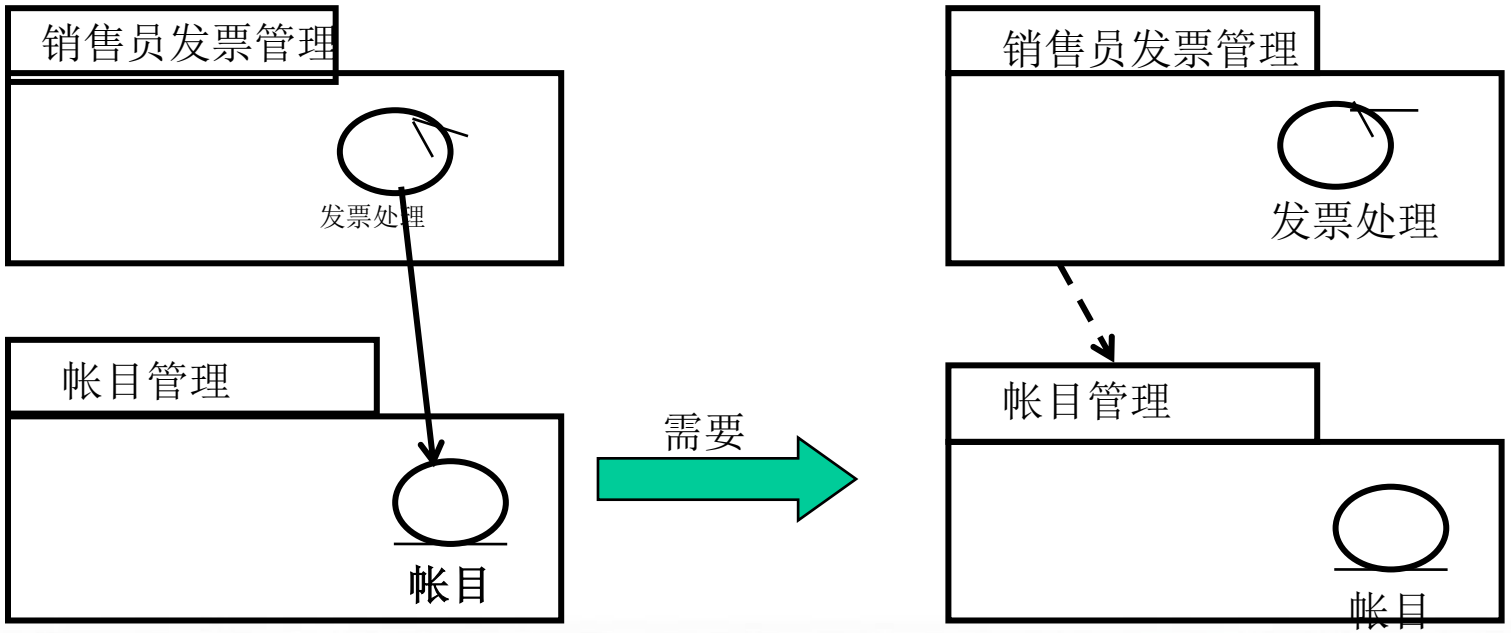
❶ 如果一个包中的类与其他一些包中的类具有关联，那么就应在该包与那些其它包之间定义一个依赖并维护之. 参见下页例.

❷ 确保该包所包含类是正确的，并力求使这些包仅包含相关对象的功能，实现包的高内聚。

❸ 限制与其他包的依赖。特别地，如果有些包所包含的类过多地依赖其他包，那么对这些包就要考虑重新布局问题。

例如：

在包“销售员发票管理”中包含一个类“发票处理”，而该类与包“帐目管理”中的类“帐目”有关联，这样就需要在这两个包之间建立一个相应的依赖。



RUP需求分析小结(四点)

第一点:关于需求分析

RUP的分析如同结构化分析一样，其目标之一是在一个特定的抽象层上建立系统分析模型。为此，作为一种特定的系统分析方法学：

首先，给出了三个术语：分析包、分析类和用况细化.

这三个术语可以表达“大粒度”的概念，开发人员使用这些术语可以规约系统分析中所要使用的信息。

① 分析包

分析包体现了“局部化”、“问题分离”等软件设计原理。通过分析包这一术语可以把系统一些变化局部到一个业务过程、一个参与者的行为，或一组紧密相关的用况，形成一些不同的系统分析包。

一个分析包中可以包含一些分析类、用况细化和一些子包。

服务包和共享包是一些特殊的分析包，服务包将一些变化局部到系统提供的一些单个的服务中，并展示了一个重要的指南，即在分析期间可以通过服务包来构造复用。

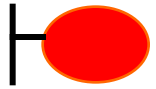
通过依赖，可以形成包的一个层次结构，其中应用包一般位于该结构的上层，而服务包和共享包位于该结构的下层。包的层次结构可以作为系统的顶层设计。

② 分析类

分析类是一种粒度比较大的类，有其责任、概念性的属性和关系。

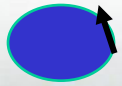
分析类分为边界类、实体类和控制类。

在应用中，一般将用户接口、一个通讯接口方面的一个变化，局部化到一个或多个边界类中；



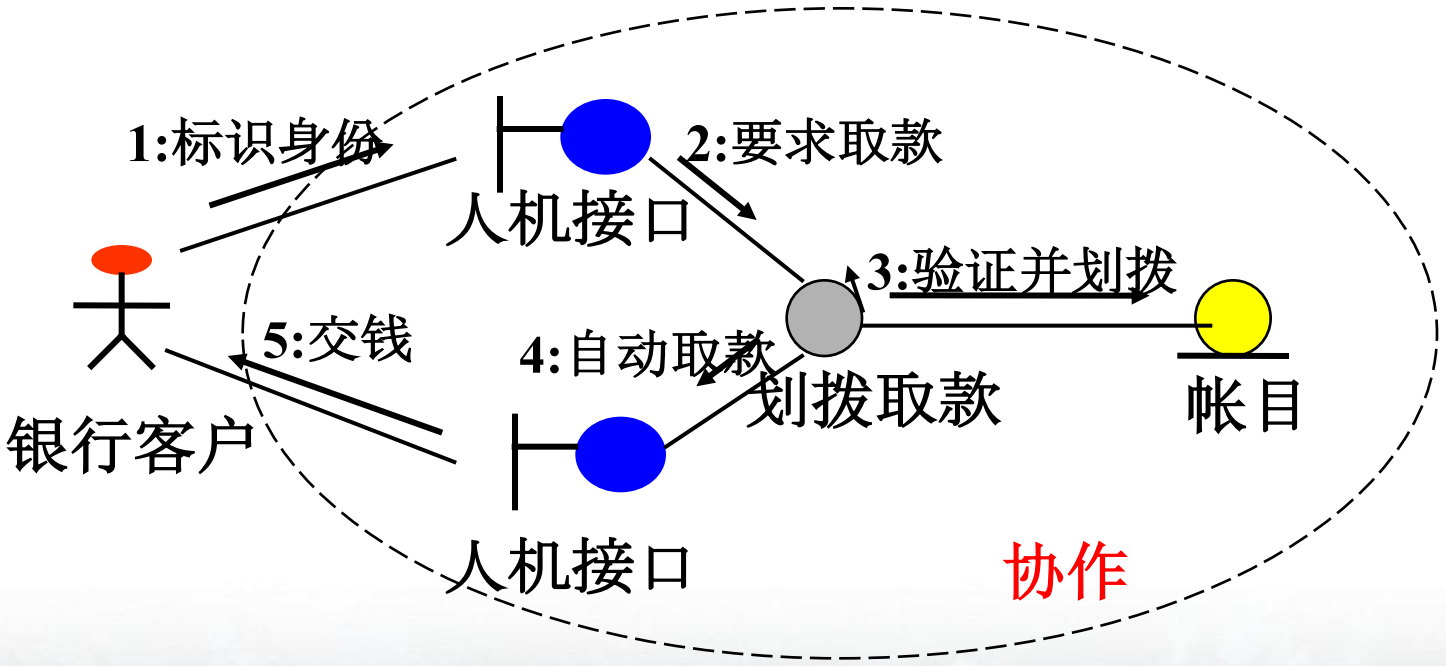
一般将系统所处理信息方面的一个变化，局部化到一个或多个实体类中；

一般将控制、协调、定序、事务和复杂业务逻辑（它们要涉及多个边界和/或实体对象）方面的一个变化，局部化到一个或多个控制类中。



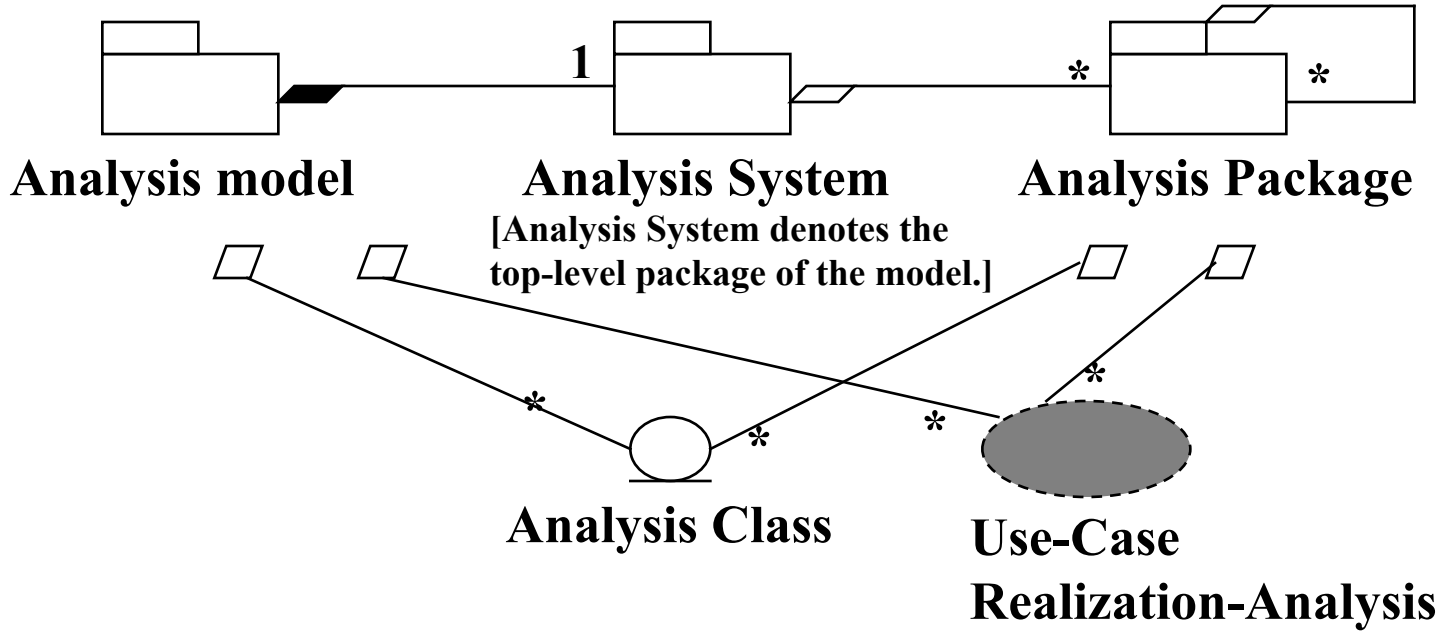
③ 用况细化

用况细化是一个协作，可用于对系统用况模型中的用况进行精化。换言之，用况细化将一些变化局部到对应的用况中，通过分析类之间的协作，规约用况的行为是如何实现的。



注意:精化了use case和相关的特殊需求。

其次,给出了分析模型的语法,用于表达该抽象层上的系统模型。



分析模型是对需求的一种精化，给出了在这一抽象层上的系统结构。

最后，给出了实施系统分析的活动，支持开发人员系统化地使用用以上三个术语所表达的信息，建立系统分析模型。

序号	输入	活动	执行者	输出
1	用况模型、补充需求、业务模型或领域模型、体系结构描述$_{用况模型角度}$	体系结构分析	体系结构设计者	分析包$_{概述}$、分析类$_{概述}$、体系结构描述$_{分析模型角度}$
2	用况模型、补充需求、业务模型或领域模型、体系结构描述$_{分析模型角度}$	分析用况	用况工程师	用况$_{实现-分析}$、分析类$_{概述}$
3	用况$_{实现-分析}$、分析类$_{概述}$	对类分析	构件工程师	分析类$_{完成}$
4	系统体系结构描述$_{分析模型角度}$、分析包$_{概述}$	对包进行分析	构件工程师	分析包$_{完成}$

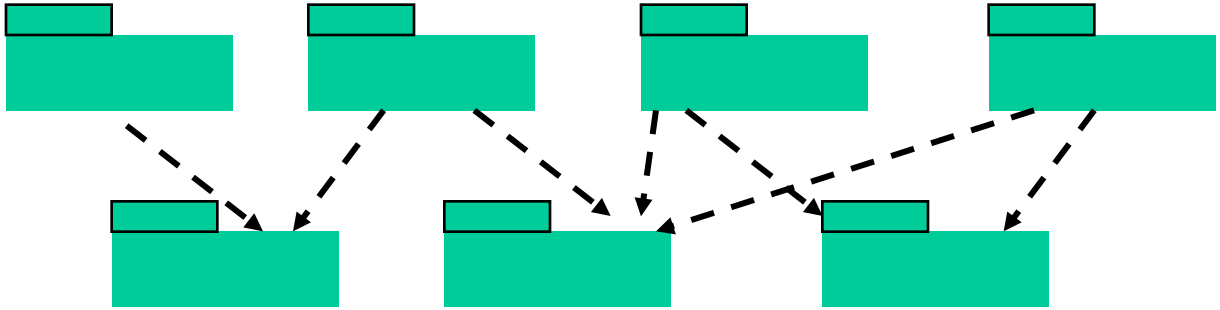
第二点:关于分析模型视角下的体系结构描述

基于RUP的特征-以体系结构为中心，分析的目标之二是建立系统分析模型视角下的体系结构描述。

该描述表达了在体系结构方面具有重要意义的元素，包括包之间的依赖

基本原则：

向“稳定”的包进行依赖！



和那些具有较多关联的分析类。

可见，分析模型视角下的体系结构描述将体系结构方面的一些变化局部到一些依赖和分析类的关联上。

第三点: Use Case模型与分析模型的比较

Use-Case Model	Analysis Model
❶ 使用客户语言来描述的；	❶ 使用开发者语言来描述的；
❷ 给出的是系统对外的视图；	❷ 给出的是系统对内的视图；
❸ 使用use cases 予以结构化的,但给出的是外部视角下的系统结构；	❸ 使用衍型类予以结构化的,但给出的是内部视角下的系统结构；
❹ 可以作为客户和开发者之间关于“系统应做什么、不应做什么”的契约；	❹ 可以作为开发者理解系统如何勾画、如何设计和如何实现的基础；
❺ 需求之间可能存在一些冗余、不一致和冲突等问题；	❺ 在需求之间不应存在冗余、不一致和冲突等问题；

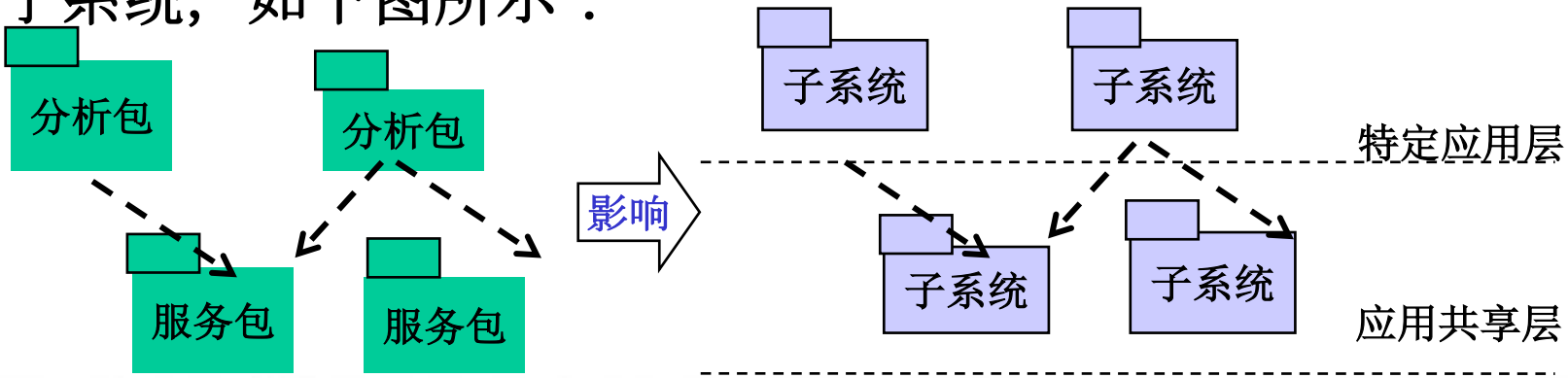
Use-Case Model	Analysis Model
❻ 捕获的是系统功能，包括在体系结构方面具有意义的功能；	❻ 给出的是细化的系统功能，包括在体系结构方面具有意义的功能；
❼ 定义了一些进一步需要在分析模型中予以分析的use case。	❼ 定义了use case模型中每一个use case的细化。

第四点:分析模型对以后工作的影响

如果把分析模型作为以后设计活动的基本输入，那么对设计模型以及相应的体系结构描述将在产生以下及方面影响：

- 对设计中子系统的影响

分析包一般将影响设计子系统的结构。一般情况下，分析包和服务包应分别对应设计中特定应用层的子系统和应用共享层的子系统，如下图所示：



特别地，在许多情况下，服务包和对应的服务子系统之间是一对一（同构）的。

- 对设计类的影响

分析包可以作为类设计时的规格说明。即:

一方面，分析类以及它们的责任、属性和关系应是进行类的操作、属性和关系设计的逻辑输入。

另一方面，在对具有不同衍型的分析类进行设计时，需要不同的技术和技能，例如:

--实体类的设计，常常需要使用数据库技术;

--边界类的设计常常需要使用用户界面技术，当在考虑数据库技术和用户界面技术时，有关分析类的大多数特殊需求，例如永久性、并发等，应由对应的设计类予以处理。

- 对Use case细化[设计]的影响

Use case细化[分析]对Use case细化[设计]将有两个方面作用:

一是它们有助于为use case创建更精确的规格说明。其中可采用状态图或活动图等，把每一个use case描述为分析类之间的一个协作，替代use-case 模型中每一个use case的事件流描述,这样就对系统需求产生了一个可理解的形式规约。

一是当对 use cases进行设计时，Use-case细化[分析]可作为其输入。 这有助于:

- 标识参与Use-case细化[设计]中的设计类。
- 确定Use-case细化[设计]中,依据所考虑的技术（数据库技术，用户界面技术），需要处理的需求,即在Use-case细化[分析]中所捕获的大多数特殊需求。

总之，设计中应尽量地保持分析模型的结构。

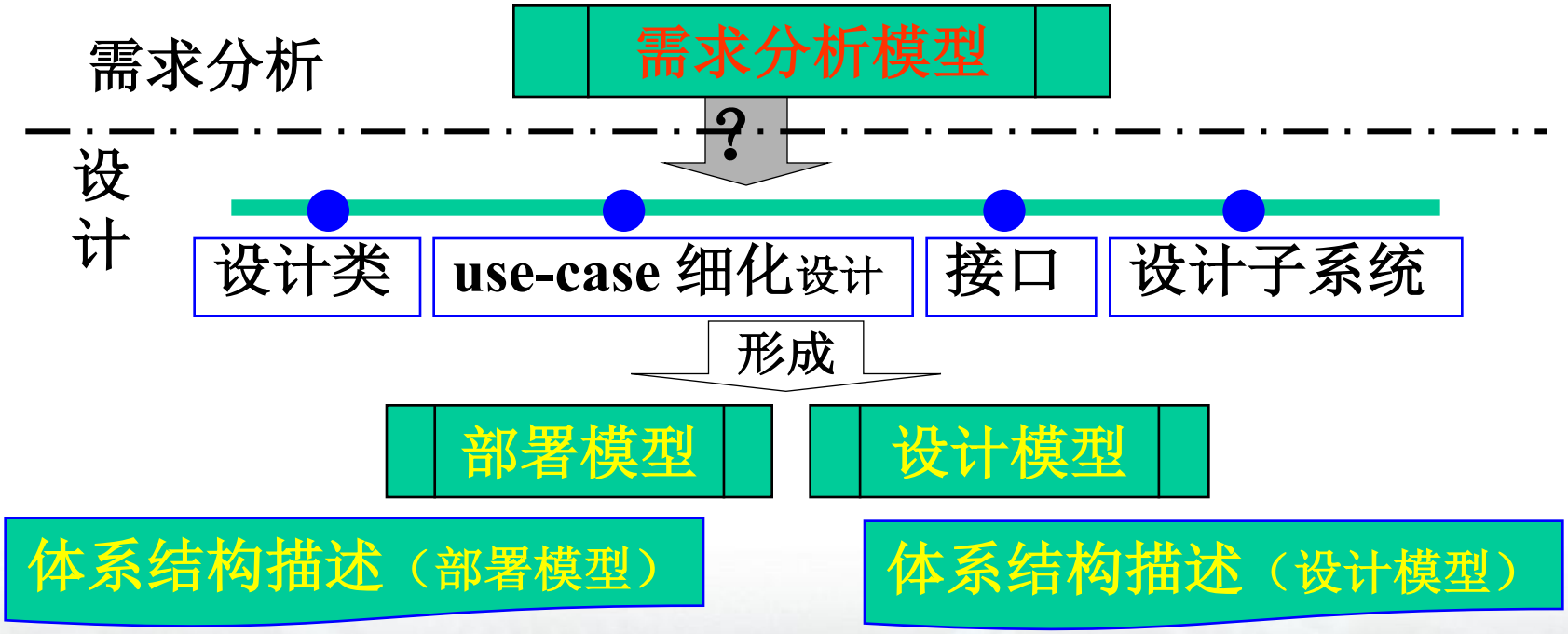
- 对创建设计模型视角下体系结构描述的影响

分析模型视角下的体系结构描述，可以作为创建设计模型视角下体系结构描述的输入。其中，通过关注跟踪依赖，不但可以使不同视图中的元素相互跟踪，而且还可以使在体系结构方面有意义的想法几乎“平稳”地“流经”不同的模型。

4)、设计的目标及其途径

(1) 目标：

设计的基本输入是分析的结果，定义满足分析所需要的结构.



(2) 实现设计目标的基本途径

实现需求分析层到设计层的映射，即从软件开发的角度-实现第三次抽象。如同实际问题层到需求获取层一样，其中至少涉及以下3个问题：

①如何定义设计层，即给出该层的术语；

②如何确定模型表示工具；

③如何映射。

①设计层的术语-回答第一个问题

❶设计类（Design class）

一个设计类是对系统实现中一个类或类似构造的一个无缝抽象。设计类的主要特征为：

Operations

Attributes

Relationships

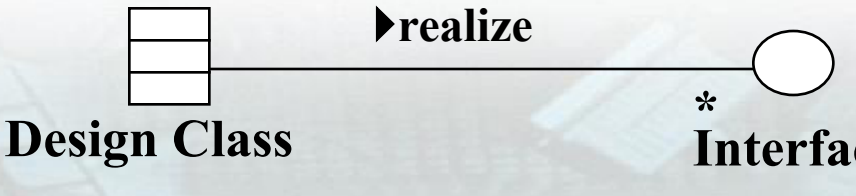
Methods

Implementation requirements

is active:{true, false}

Note:active class: its objects maintain own thread of control and run concurrently with other active objects.

设计类可细化为多个接口，如下图所示：



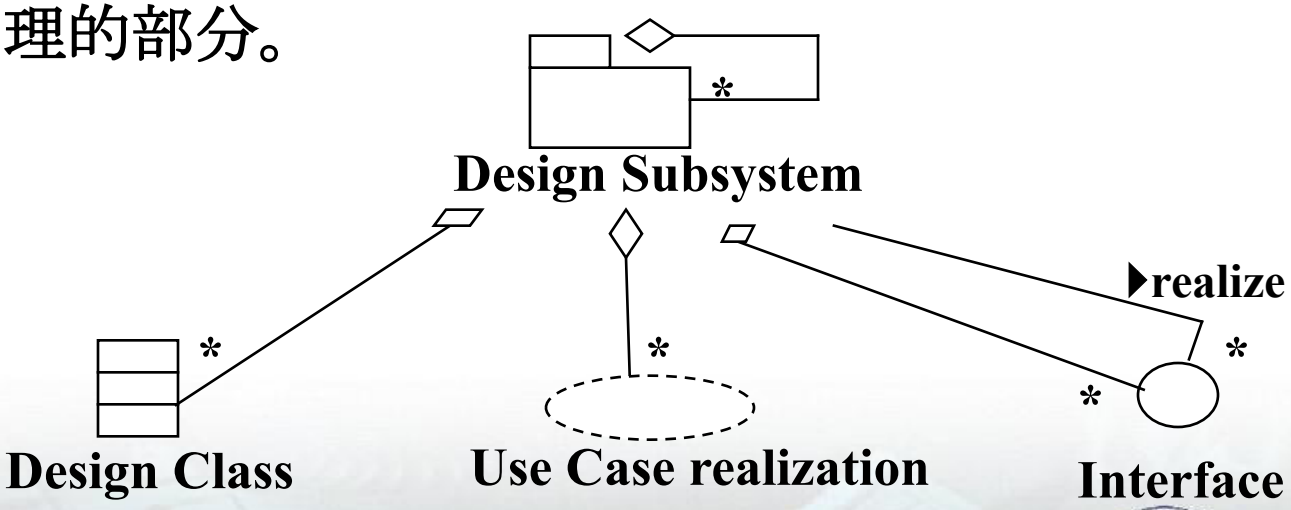
❷用况细化[设计]

用况细化[设计]是设计模型中的一个协作，其中，使用设计类及其对象，描述一个特定用况是如何予以细化的，如何执行的。

表达协作的工具可以是类图、交互图和正文事件流等。

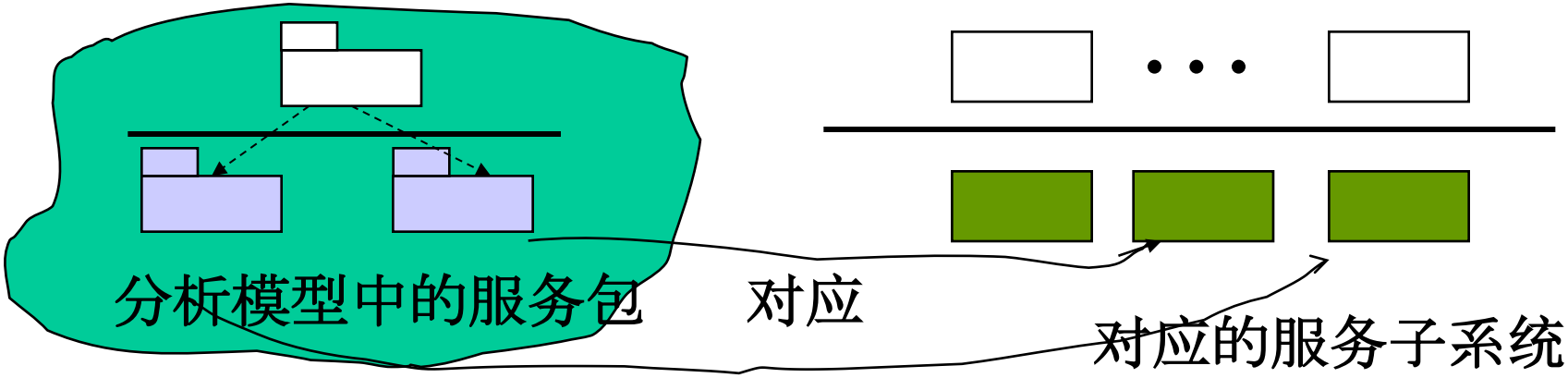
❸设计子系统

设计子系统提供了一种组织设计制品的手段，成为一些可以更易管理的部分。



设计子系统和分析模型之间的关系：

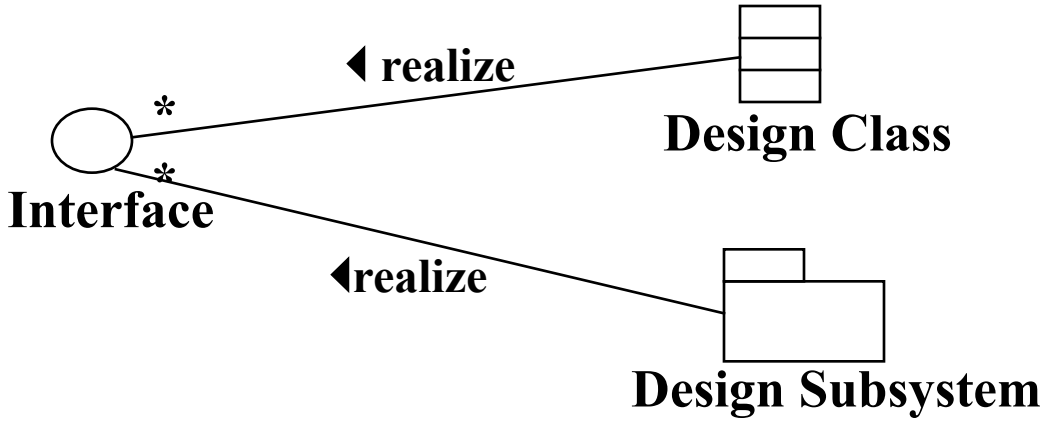
分析模型中的包结构,一般对应设计子系统的层次结构。特别是,分析模型中的服务包一般对应设计子系统层次结构低层上的服务子系统.如下图所示。



在应用中，一般通过使用服务子系统，把一些变化局部化到不同的服务子系统中，形成一些封装相应变化的单个服务。

❹接口（Interface）

接口用于规约由设计类和设计子系统提供的操作。因此，一个接口的重要关联是：



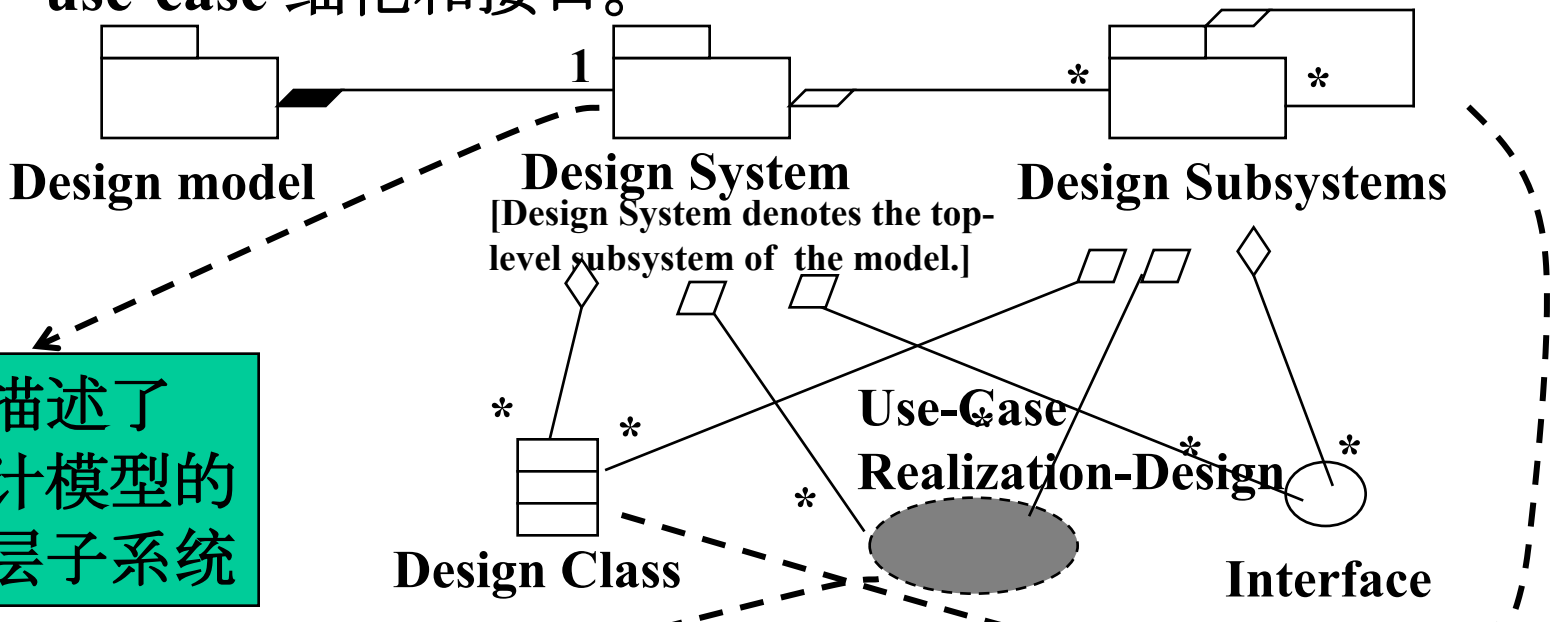
可见，♦接口提供了一种分离功能的手段，其中使用了与实现中方法对应的操作，因此提供接口的设计类和设计子系统，必须提供细化该接口操作的方法。

♦子系统之间的大多数接口，对体系结构是有意义的。

②设计模型、部署模型 -回答第二个问题

❶设计模型

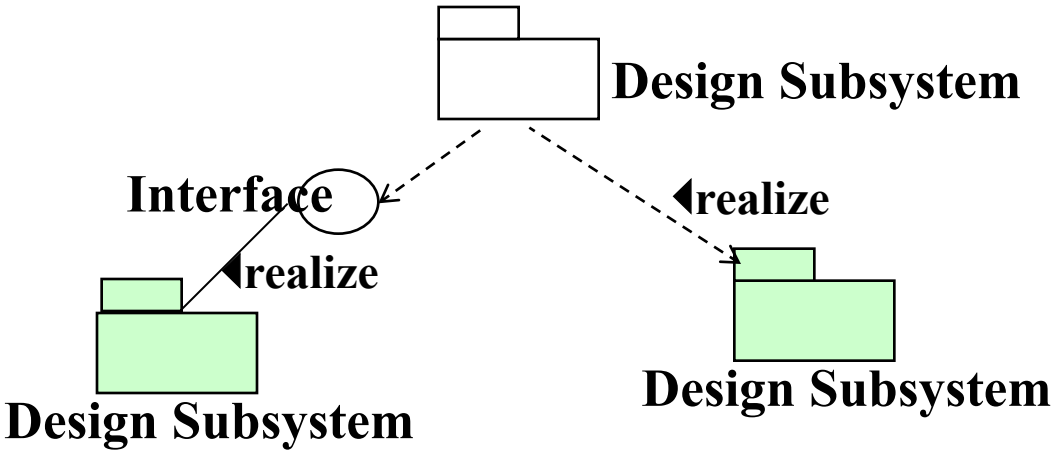
是设计子系统的层次结构（ hierarchy），包含设计类、 use-case 细化和接口。



设计模型视角下的体系结构描述，是从设计模型的角度，描述那些在体系结构方面具有意义的制品。

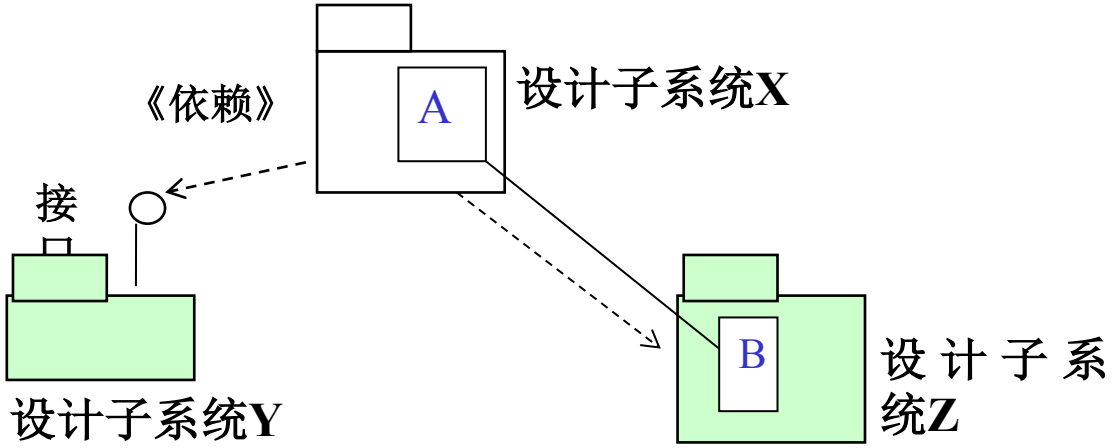
通常，这些制品包括：

- 子系统的结构，包括它们的接口以及它们之间的依赖。



注释:由于子系统和它们的接口构成了系统的基本结构，因此一般来说子系统的结构对体系结构而言是非常有意义的。

- 对体系结构有意义设计类，如下所示：



注释:其中类A和类B存在一个关联，使设计子系统X和设计子系统B之间具有依赖关系，因此类A和类B是对体系结构具有意义的类。

对体系结构有意义的设计类，一般包括：

--对体系结构有意义的分析类所对应那些设计类；

--具有一般性、核心的主动类；

--表达通用设计机制的设计类，以及

--与以上这些设计类相关的其他设计类。

对于这样一些对体系结构有意义的设计类而言，一般只考虑抽象类，而不考虑它的子类，除非该子类展现了某些与其抽象类不同的行为，并对体系结构有意义。

- **Use-case 细化**［设计］

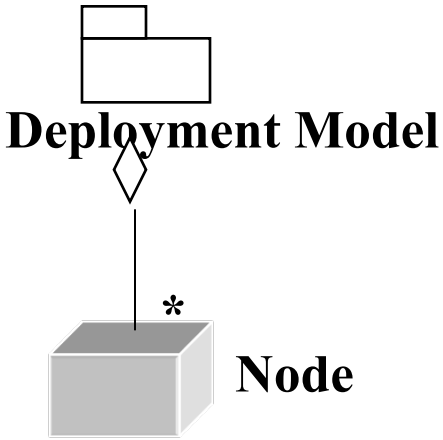
某些重要的、关键功能的、必须在软件生存周期早期予以开发的**Use-case 细化**［设计］。

—这样的**Use-case 细化**［设计］，涉及许多设计类，也可能涉及许多子系统，或涉及以前提及到的关键设计类。

—通常，**Use-case 细化**［设计］对应**use-case**模型视觉的体系结构中的 **use cases**，对应分析模型视觉的体系结构中的 **use cases**细化［分析］。

❷部署模型（Deployment Model）

部署模型是一个对象模型，描述了系统的物理分布，即怎样把功能分布于各个节点（nodes）。



可见,部署模型本身展现了软件体系结构和系统体系结构（硬件）之间的一个映射（mapping）。

关于部署模型的几点说明：

- ♦部署模型包含一些节点及节点之间的关系，其中每一个节点表达一个计算资源，常常是处理器或类似的硬件设备。
- ♦节点功能是由部署在该结点上构件所定义的；节点之间的关系是由节点之间的通讯手段表达的，例如*internet*、*intranet*和 总线等。
- ♦在实际应用中部署模型可以描述多个不同的网络配置，包括测试配置和仿真配置。

部署模型视角下的体系结构描述

从部署模型的视角，描述该模型中那些对体系结构有意义的制品。但由于部署模型是十分重要的模型，因此应描述其体系结构视觉下的所有方面，包括节点与构件(实现期间所发现的那些构件)之间的映射。

③工作流-回答第三个问题

实施准备：开始设计时，应更深入地理解以下问题：

-非功能需求；

-有关对程序设计语言的限制（constrains）；

-数据库技术；

-用况技术；

-事务（transaction）技术等.

活动1:体系结构设计（ Architectural Design）

目标:

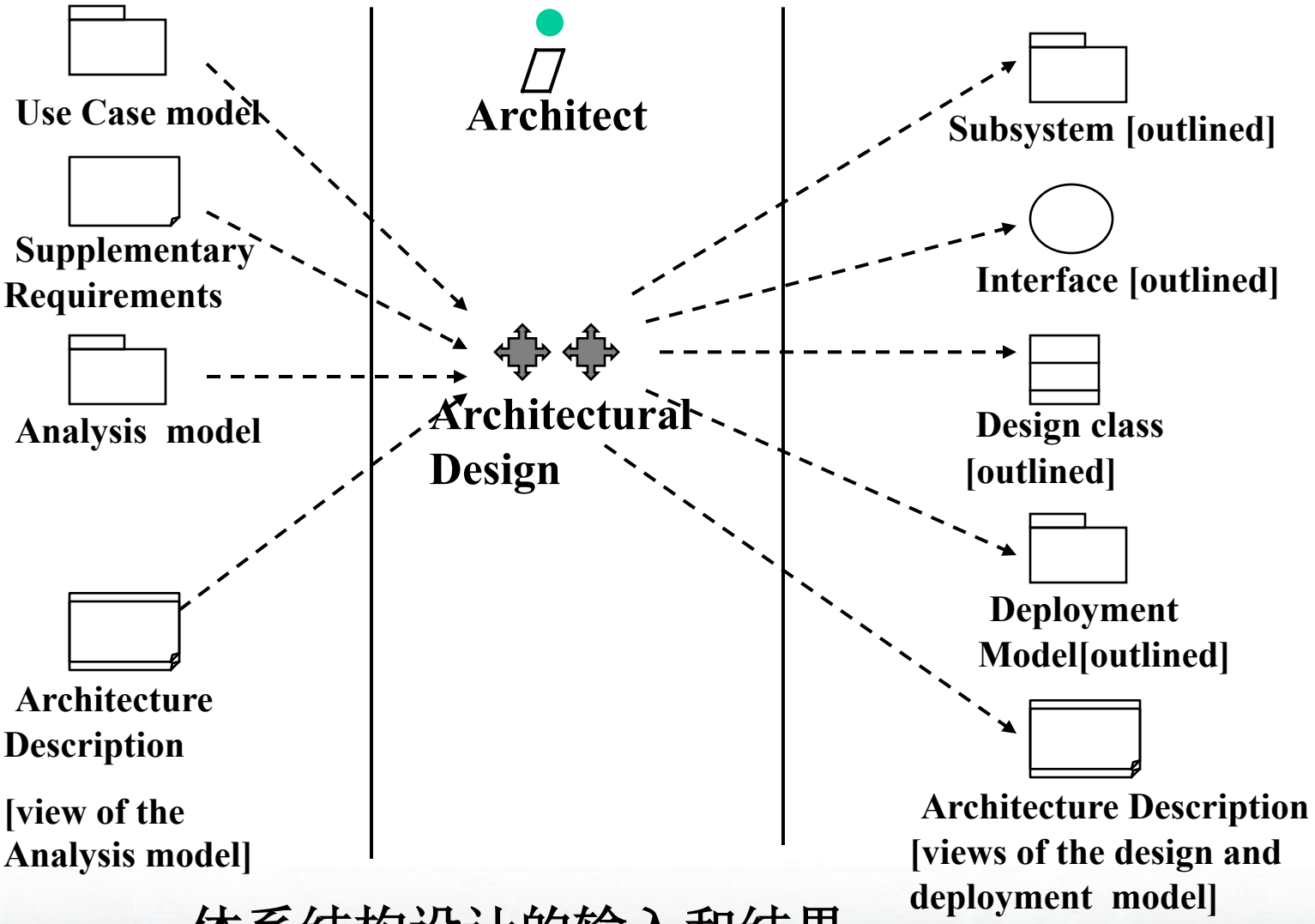
给出设计模型和部署模型，以及这两个模型视觉下的体系结构描述。为此，需要标识：

- 节点，以及它们的网络配置；

-子系统，以及它们的接口；

-在体系结构方面具有意义的设计类，例如主动类。

-设计机制，它们处理共性需求，例如，在分析期间所捕获的有关分析类、 use-case细化［分析］的永久性、分布性以及性能等方面的特殊需求。



体系结构设计的输入和结果

（The input and result of architectural design）

任务1:标识节点和它们的网络配置

- 标识需要涉及的节点以及它们的能力（处理能力和内存规模）；
- 标识节点之间的连接（connections）类型,以及使用的通讯协议；
- 标识这些连接和通讯协议的特征，如宽带，可用性和质量等；
- 标识需要的冗余处理能力、失败模式、移植处理（process migration）以及数据备份等。

以上称为网络配置.

--网络配置经常对软件的体系结构具有很大影响.

网络配置的基本途径：

- 了解网络配置模式

  通常使用三元模式：客户端（用户交互）、数据库功能以及业务/应用逻辑。而C/S模式是一种特殊的情况。

- 了解节点以及它们连接的限制和可能性,而后就可以考虑使用有关的技术，例如ORB（对象请求代理）和数据复制服务器等，来实现系统的分布。其中,在进行网络节点之间的功能分布时,一般会需要一些主动类。

期间，还可能需要为测试、仿真等来配置网络，这些配置很可能使我们开始考虑为什么如此在网络节点上来分布系统功能。

任务2:标识子系统和它们的接口

设计子系统提供了一种将设计模型组织成一些可管理部分（pieces）的手段。.

一般情况下:● 或通过对设计工作的划分,来发现子系统;

● 在或设计模型的不断进化中，通过对其较大的结构进行分解时来发现子系统。

标识子系统的一些指导:

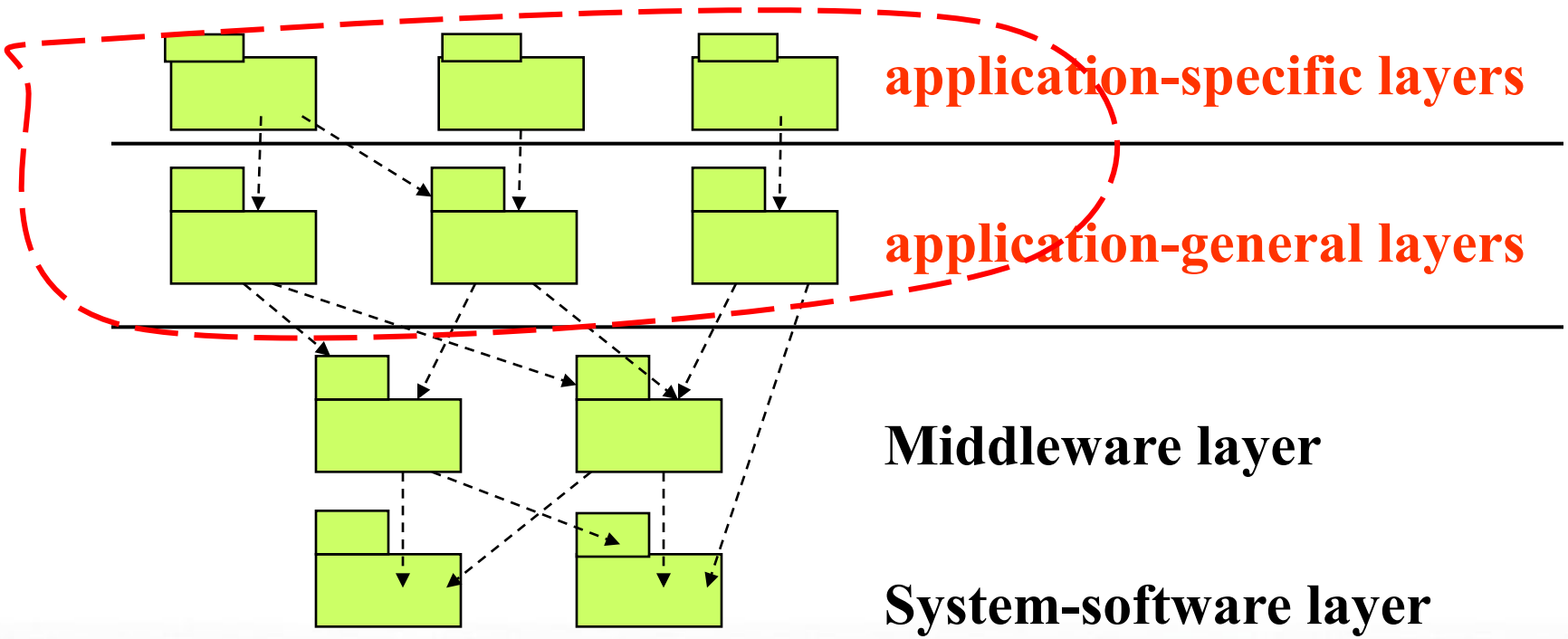
第一　标识可复用资产

发现组织内可作为子系统使用的的产品.

注:这一工作有助于思考设计模型中的复用问题。

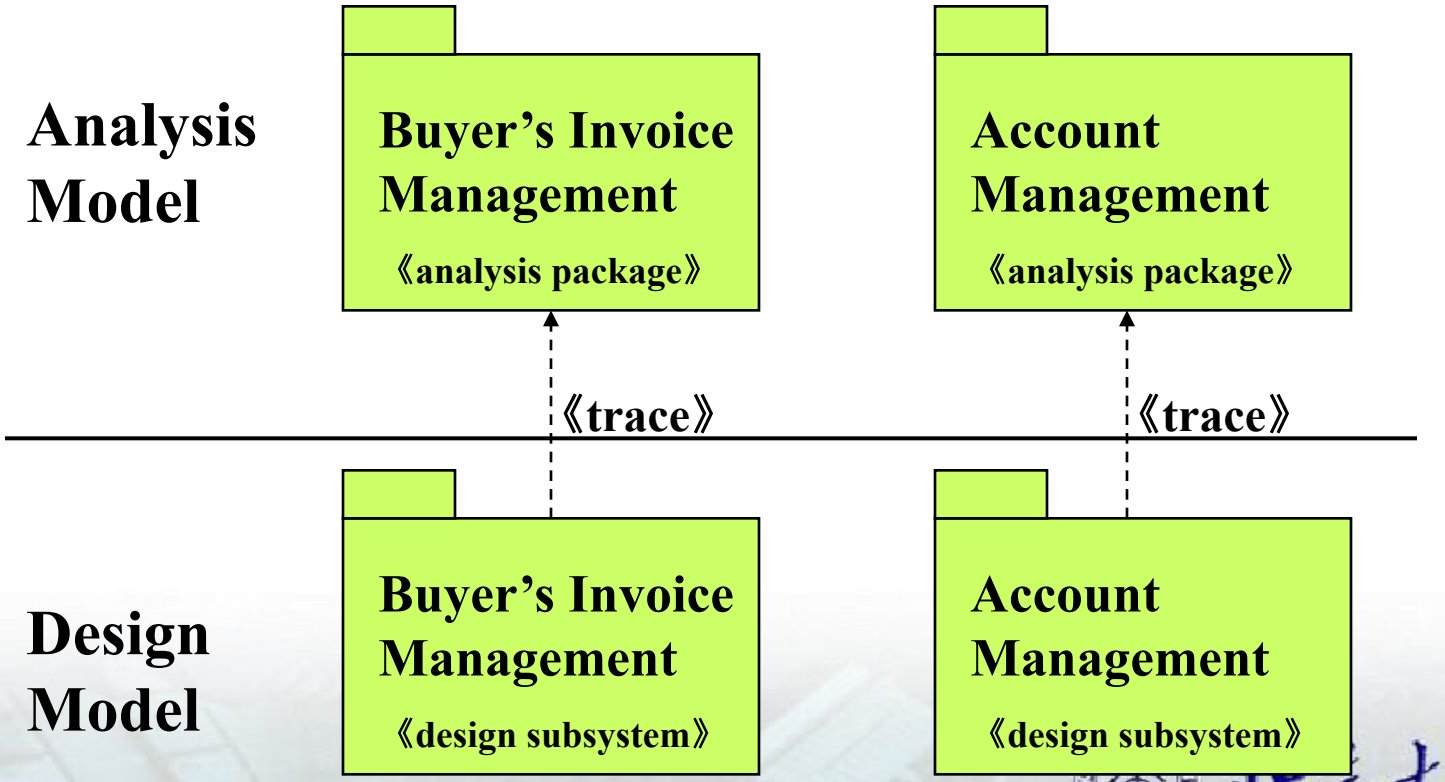
第二, 标识应用子系统

应用子系统可分为二个层次:

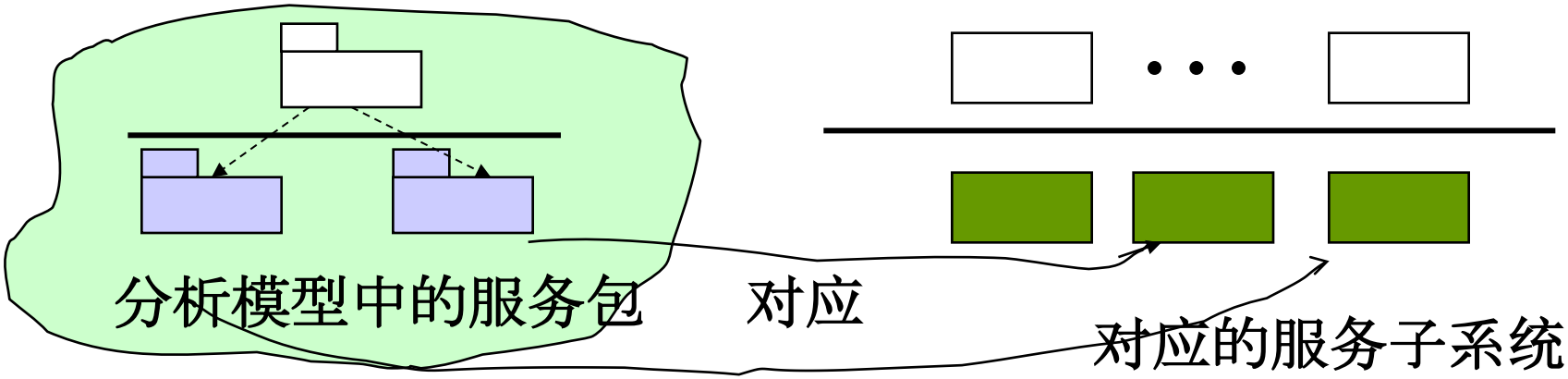


第一步:基于分析期间存在的分析包结构，发现其中有哪些分析包可以作为设计模型中的子系统。

例如：基于已有的分析包来发现设计子系统

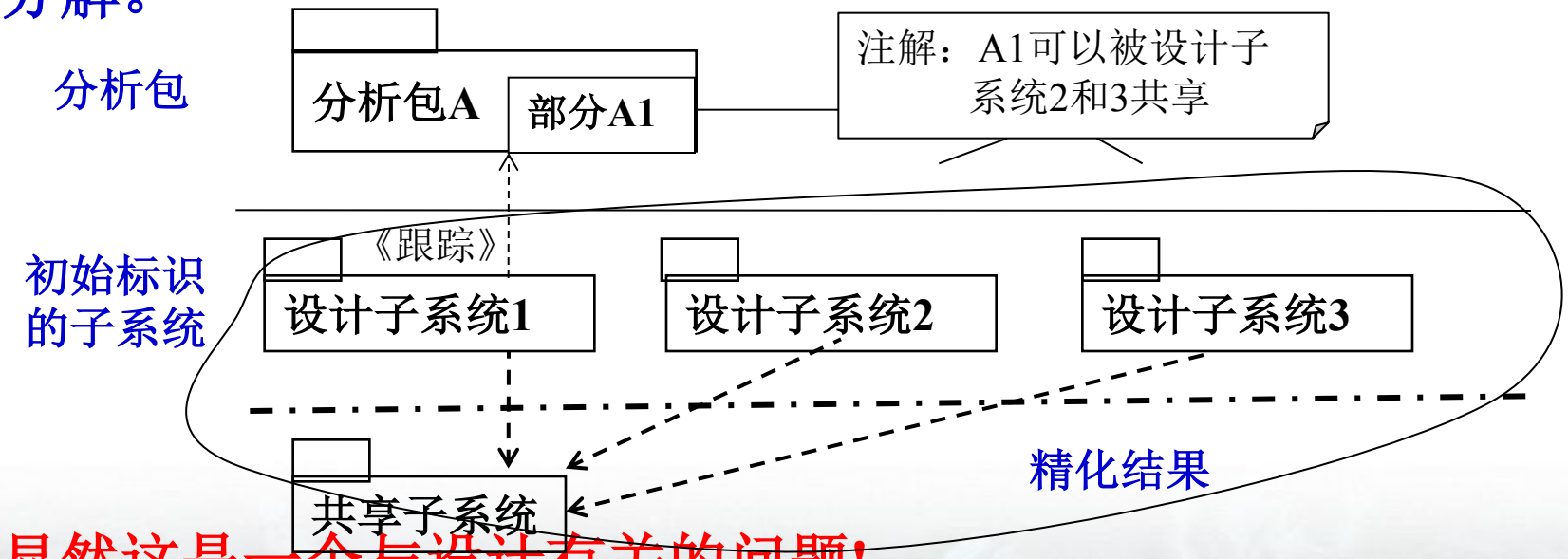


特别地,应将分析期间所形成的那些服务包，标识为服务子系统,以避免破坏提供这些服务的那个/些系统的结构。



第二步:针对以上初始标识的子系统，考虑一些需要处理的与系统设计、实现以及部署中有关问题，以期对之进行精化。例如:

- 一个分析包的一部分（Part）是否可以被其它多个子系统共享和复用。如果可以的话，就应对该分析包所对应的子系统进行分解。



显然这是一个与设计有关的问题!

•分析包中是否存在某些部分,它们可复用已有的软件产品,例如中间件或其他已有的系统/软件子系统.如果存在的话,就应把这些部分的功能分派给相应的软件产品.

•如果分析包的某些部分可由组织的遗产系统所替代，那么就应把该遗产系统“封装”为一个子系统，并建立该包所对应的子系统与封装后的那个子系统之间的依赖。

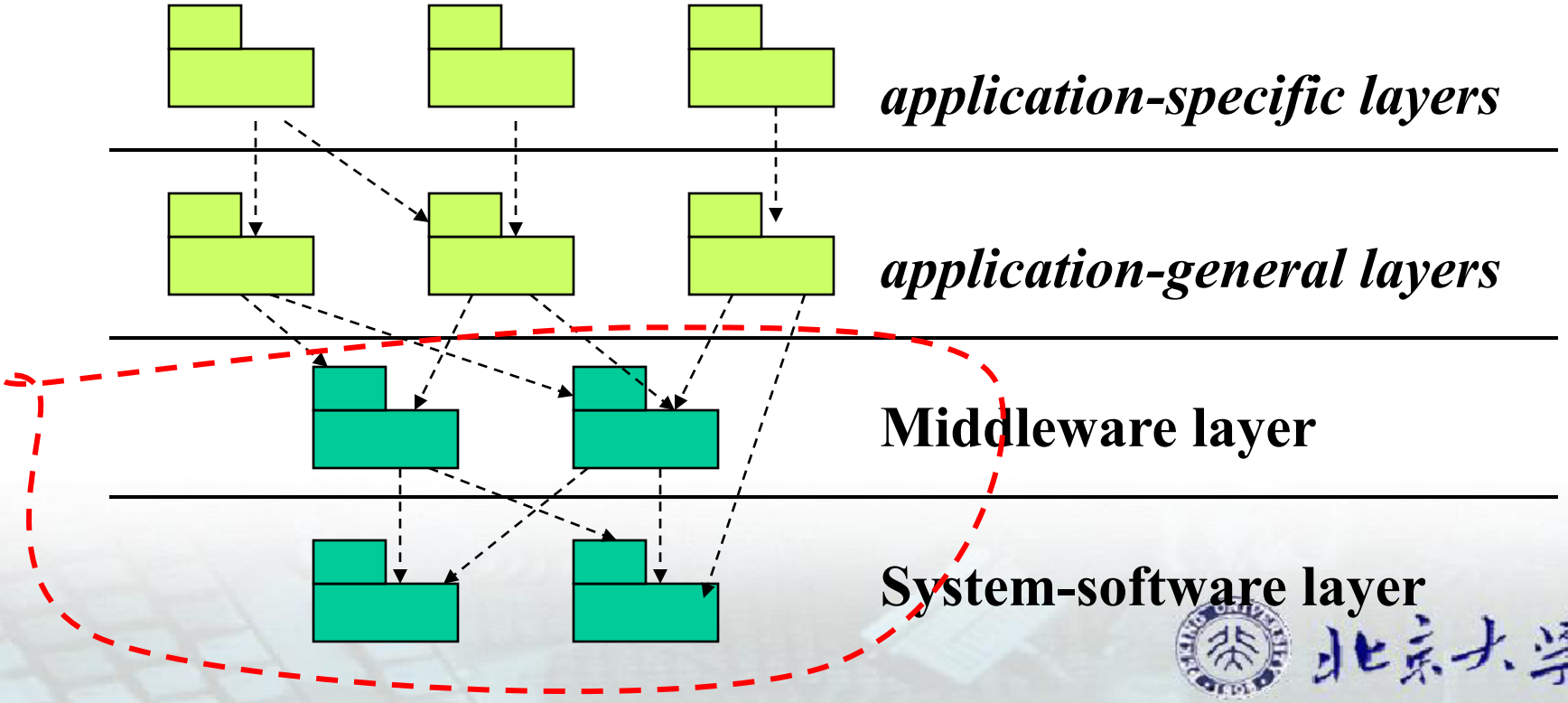
• 如果分析包并不反映一个合适的工作划分，那么就要对该包所对应的子系统进行调整，形成一个新的子系统结构。

• 如果以前的分析包没有给出到节点的部署方案，那么，为了处理部署问题，就有可能将对应的设计子系统进一步分解成为一些更小的子系统，以此最小化网络流量等。

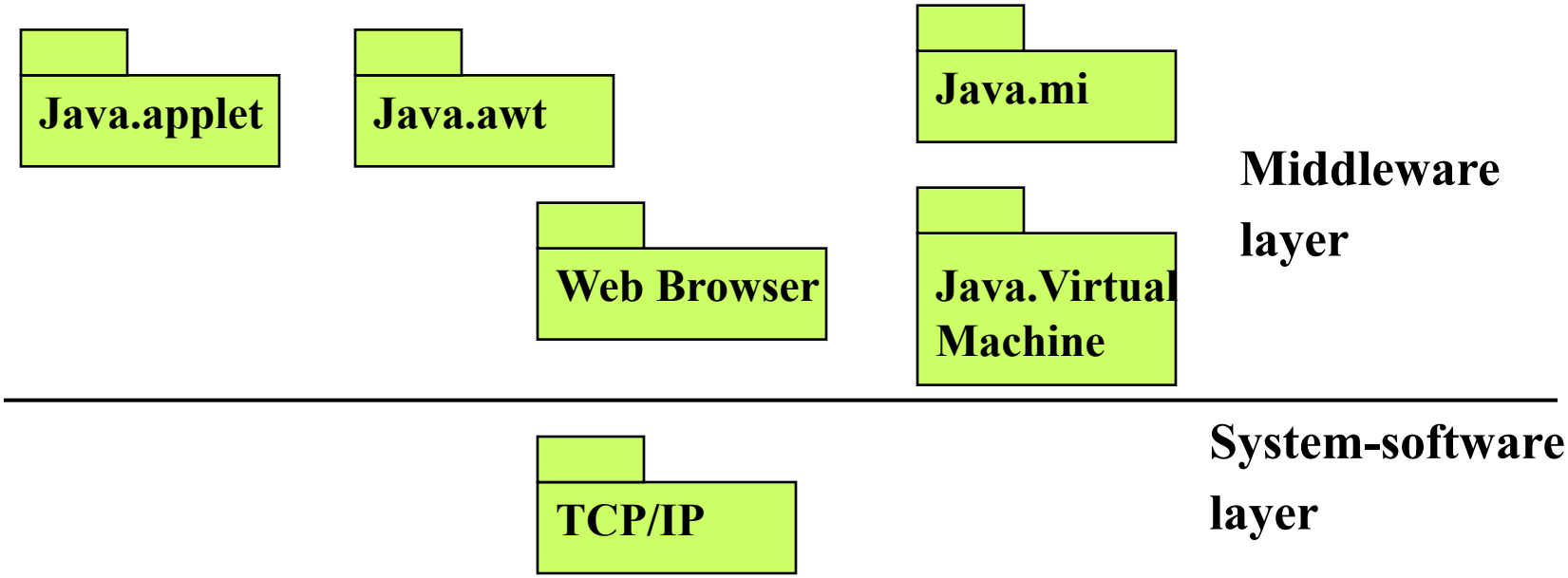
第三,标识中间件和系统/软件子系统

中间件和产品/子系统是一个系统的基础。选择并集成软件产品，是初始阶段和精化阶段的2个基本问题。其中：

- 应确认所选择的软件产品是否适合系统体系结构；
- 应确认它们是否提供了一种成本有效的系统实现。



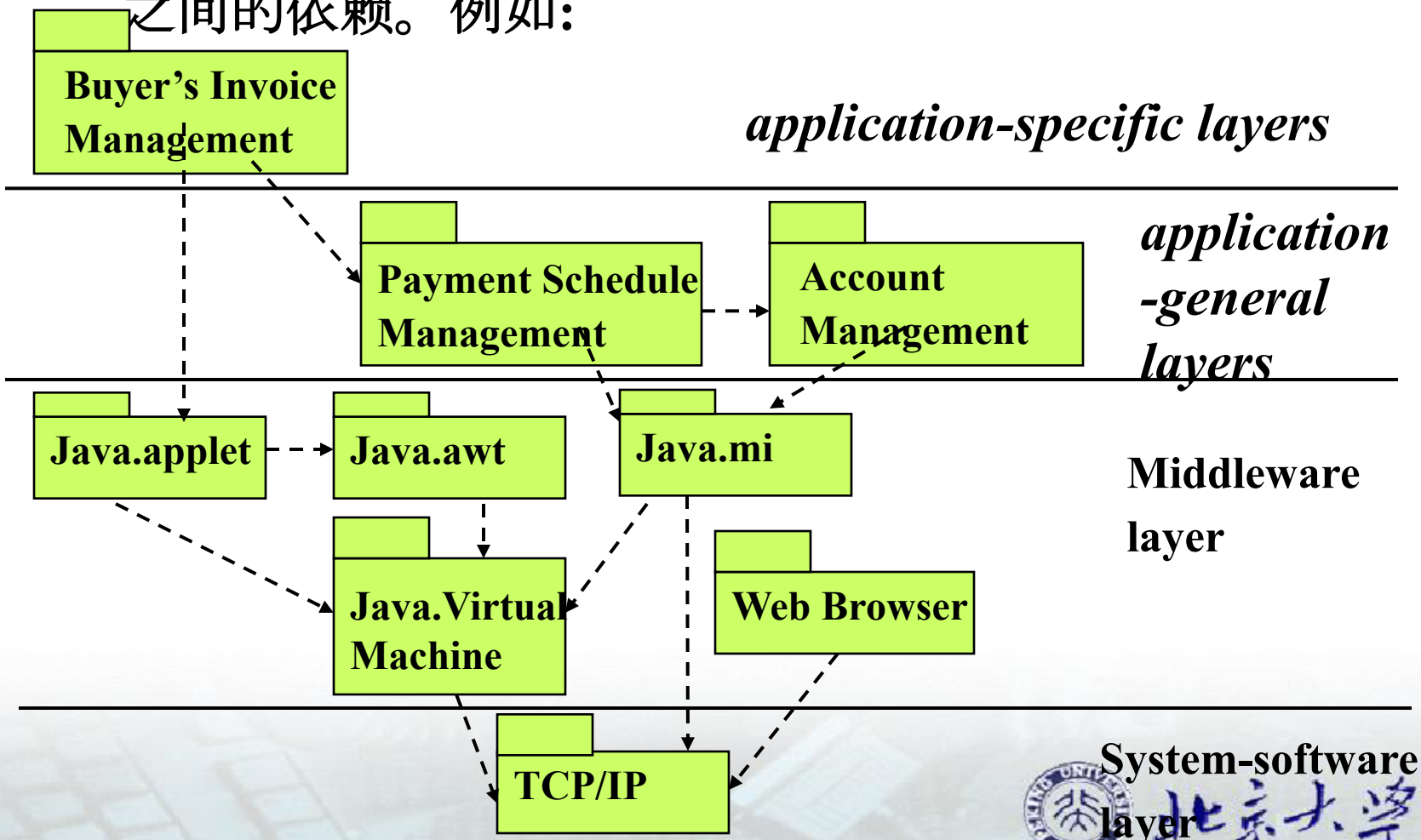
例如:



其中：Java 中间件层提供了平台无关的、图形化的用户接口(java.awt)，并分布了对象计算 (java.rmi).该图没有显示子系统的依赖，但说明了如何将java服务组织为中间件层中的子系统。

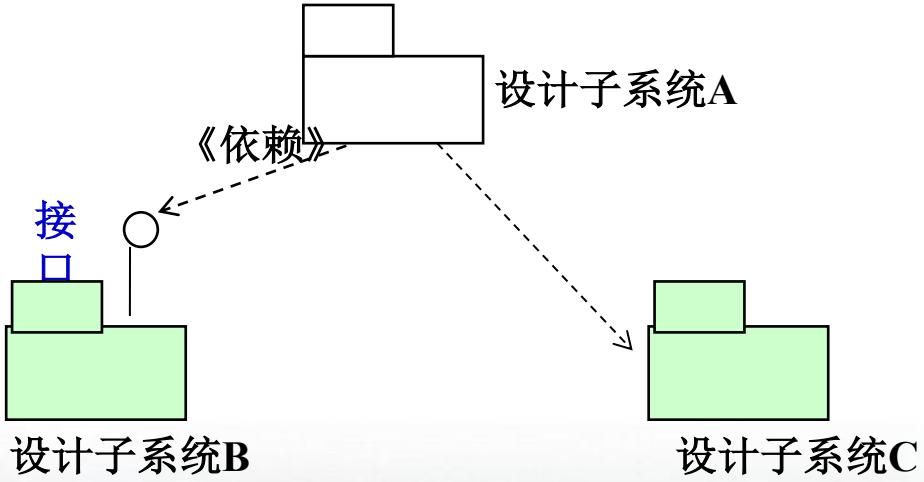
第四，定义子系统依赖（Defining Subsystem Dependencies）

--如果了解子系统的内容,那么就应根据内容来定义子系统之间的依赖。例如:

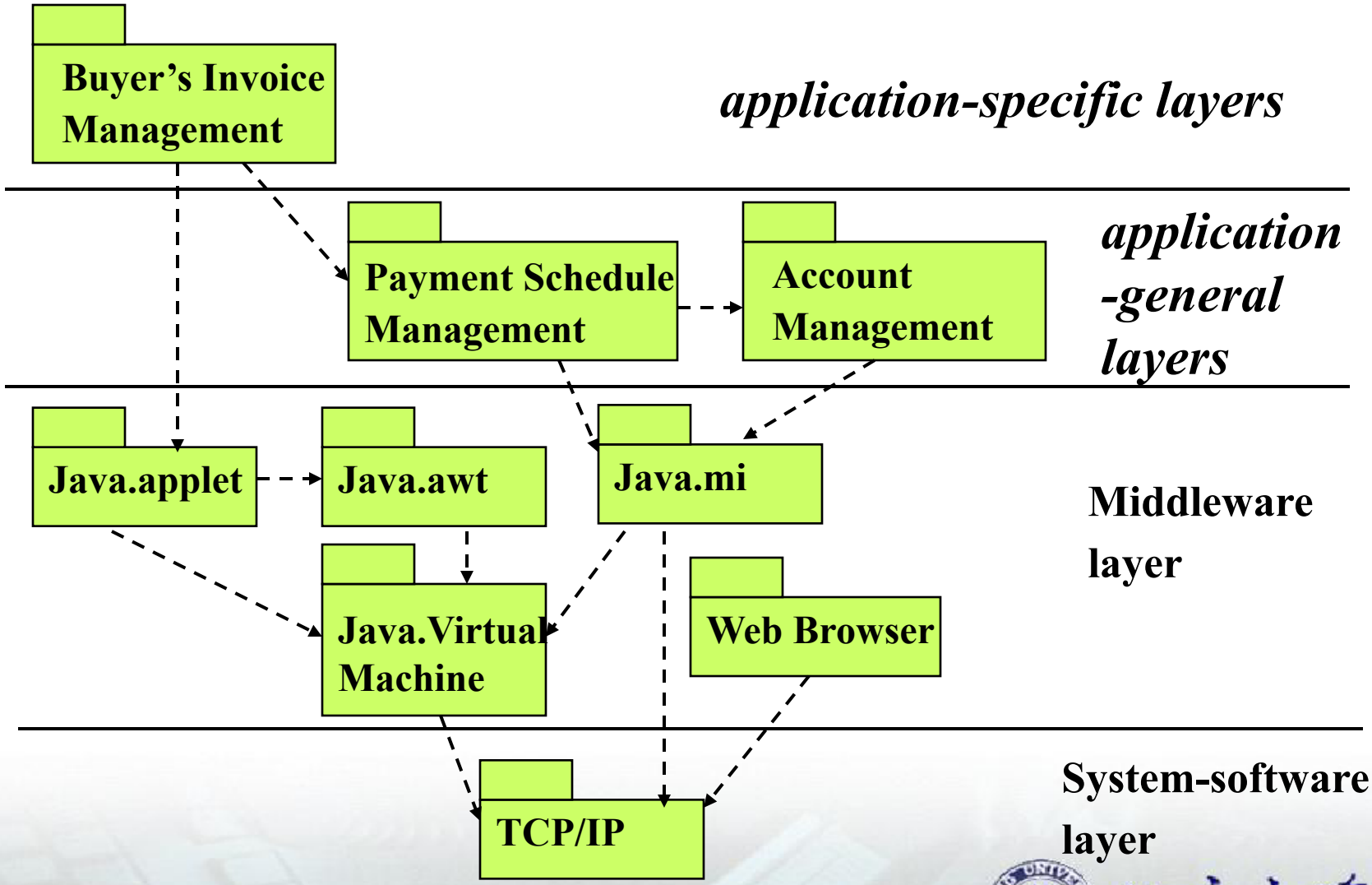


--如果开始不了解子系统的内容,那么就应分析设计子系统所对应的那些分析包之间的依赖。这些依赖很可能成为设计模型中子系统之间的依赖。

--如果在定义子系统的依赖中,使用了子系统之间的接口,那么这些依赖就应针对接口而不针对子系统本身。

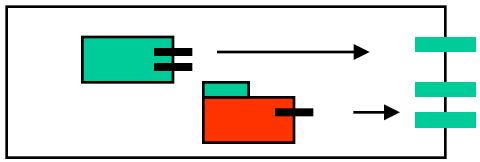


例如：


Buyer's Invoice Management
application-specific layers
Payment Schedule Management
Account Management
application -general layers
Java.applet
Java.awt
Java.mi
Middleware layer
Java.Virtual Machine
Web Browser
TCP/IP
System-software layer

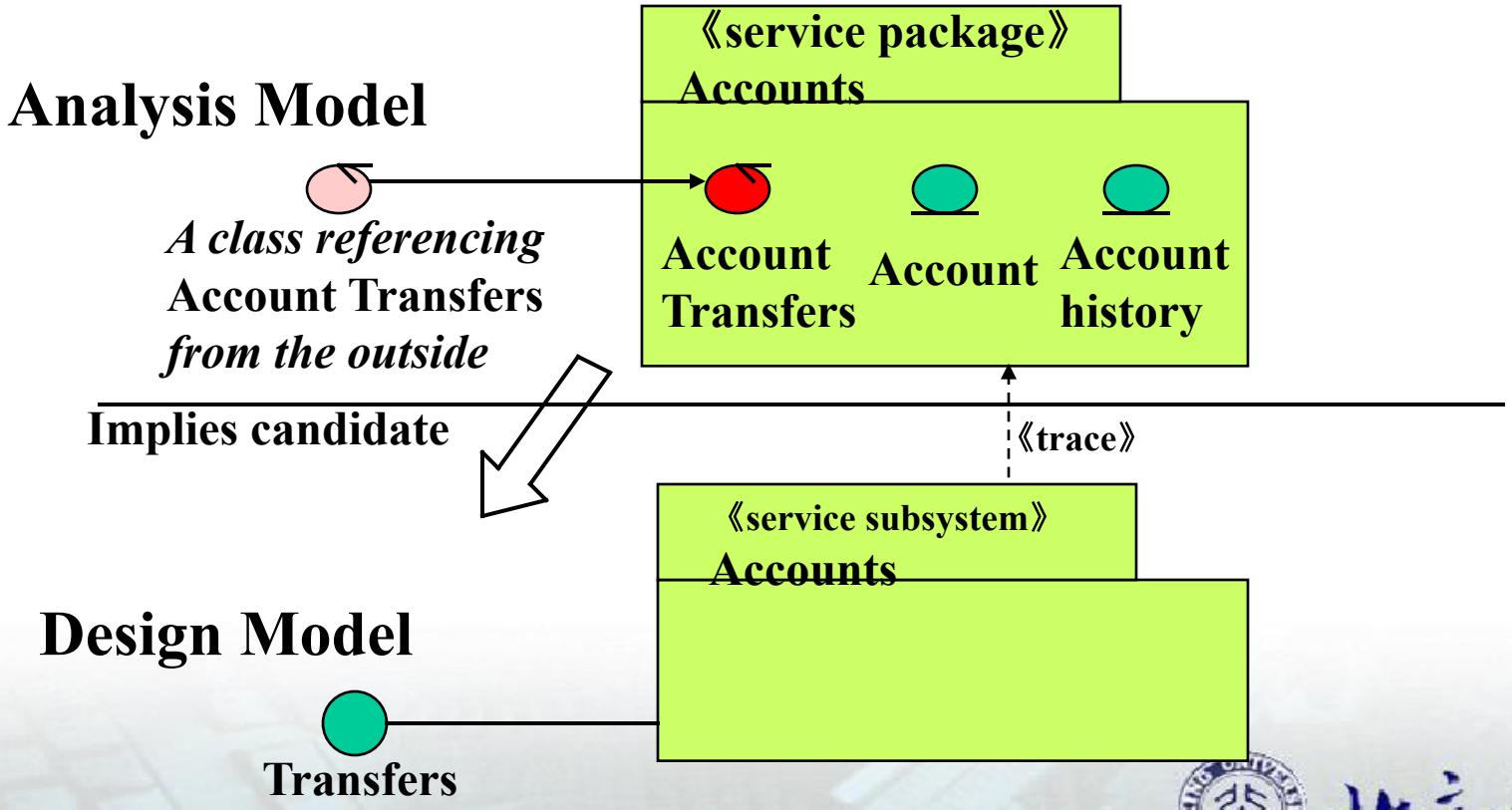
第五，标识子系统接口（Identifying Subsystem Interfaces）

由子系统所提供的接口，定义了对外可访问的一些操作。这些接口或是由该子系统中的类提供的，或是由该子系统中其它子系统提供的。

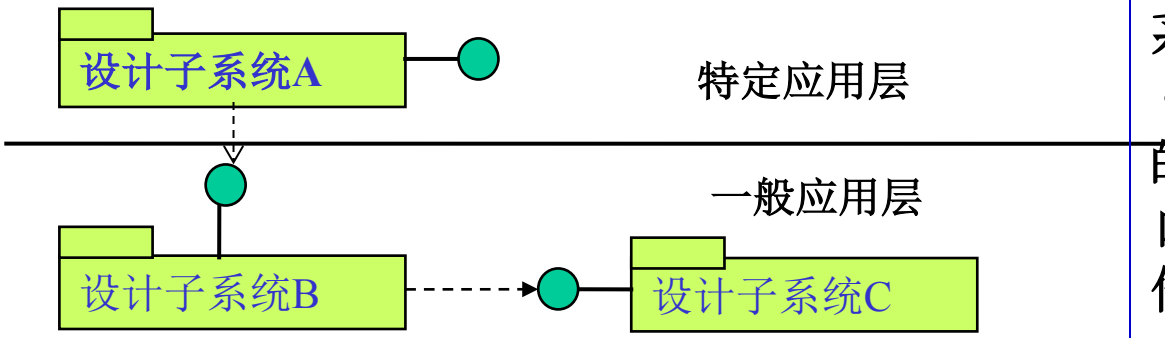


- 首先,要考虑以上发现的那些子系统之间的依赖：一般地,当一个子系统具有一个指向它的依赖，可能就需要提供一个接口；
- 其次,如果一个子系统存在一个所跟踪的分析包，那么就要基于该包对外所引用的任意分析类，来发现该子系统的一些后选的接口。

例如：Accounts 服务包包括一个被该包之外引用的分析类 Account Transfers，由此就可以标识一个初始接口 Transfers，该接口由设计模型中的那个对应的Accounts 服务子系统所提供。



使用这一途径，可初始地标识设计模型中上2层的一些接口，即：

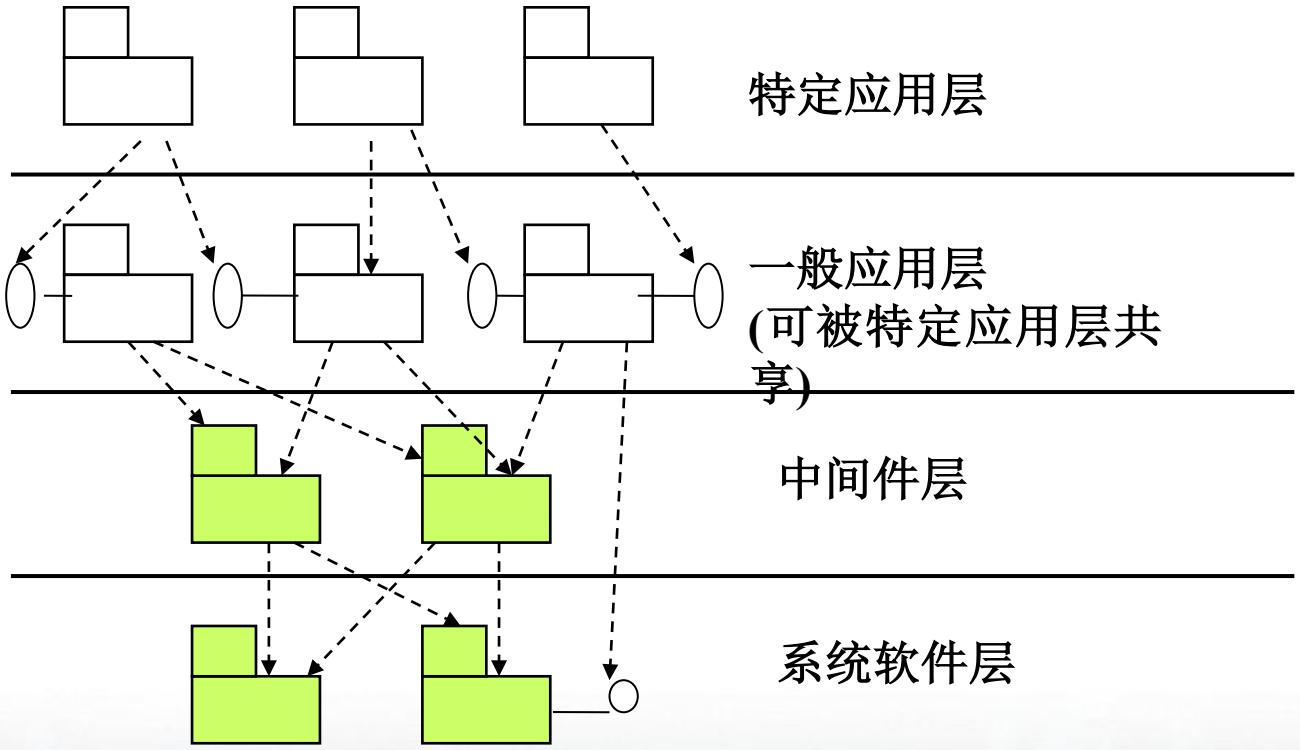


- 通过接口,可以精化子系统间的依赖；
- 只标识接口是不充分的，还必须标识每一接口中需要定义的那些操作。

（如何做？参见–design a use case一节）

注:对于下二层(中间件层和系统软件层)的接口，标识其接口相对要简单一些。因为在这二个层次上的子系统，封装了软件产品，而且这样的产品常常具有某种形式的、事先已定义的一些接口。

当完成任务2时，就可形成系统的初始顶层设计-系统的子系统结构，如下所示：



任务3:标识在体系结构方面具有意义的设计类和它们的接口

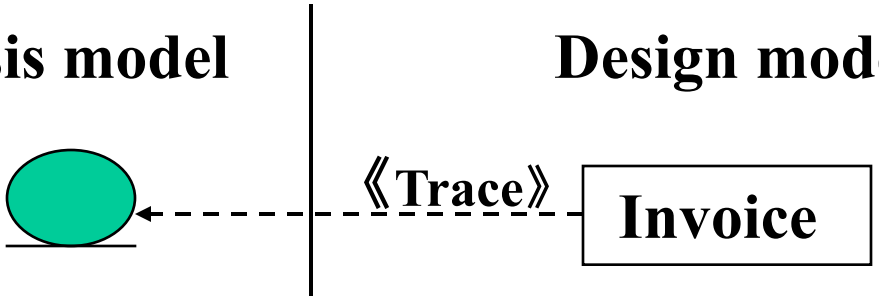
实际工作中，往往是在软件生存周期的早期来标识在体系结构方面有意义的设计类，但是此时没有标识更多的类，也没有探索其更多的细节。

注:在以后的类设计中,应：

- 标识大多数设计类，并基于use-case设计活动的结果，对其进行精化。
- 为了证明设计类，还要做一些工作，例如要验证一个设计类是否参与了一个use-cases细化，否则这样的设计类就是不必要的。

第一:标识在体系结构方面有意义的设计类.

根据在体系结构方面具有意义的分析类，可以初始地标识出这样的一些设计类。例如：



The Invoice design class is initially outlined from the Invoice entity class.

并可以使用这些分析类之间的关系，直接标识出相对应设计类之间的一组关系。

第二:标识主动类（Identifying Active Classes）

主动类的标识

一般应该考虑系统的一些并发需求，例如：

❶关于系统在节点上的分布（distribution）需求.

为了支持这一分布，处理节点之间的通讯，可能每个节点至少需要一个主动对象.

首先，要考虑主动对象的生存周期，考虑主动对象应如何通讯、同步和共享信息。

继之，再将它们分配到部署模型的节点。其中应考虑节点的容量，例如处理器的数目、内存的规模；考虑连接的特征，例如它们的带宽和可用性。对此，一个基本规则是:

认真控制网络通信量.因为它对系统所要求的计算资源（包括软件和硬件）具有重要影响.

❷关于系统启动、终止、激活以及死锁避免、节点再配置等需求.

针对以上需求,也需要一些主动对象来处理之。

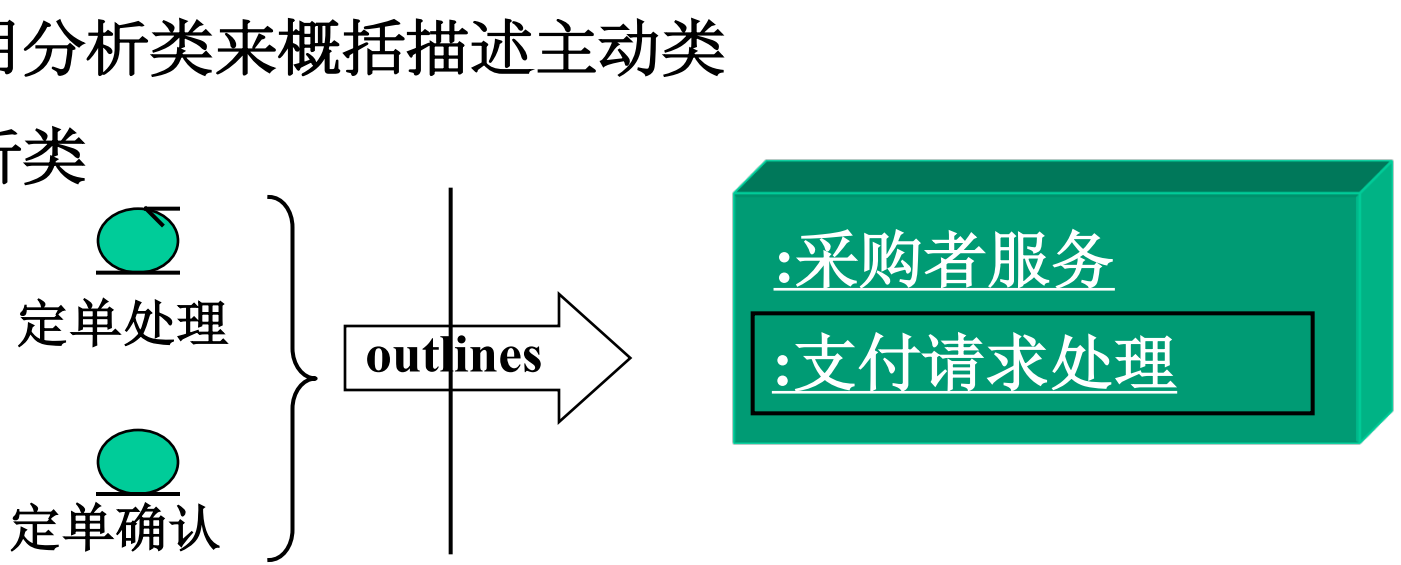
主动类的描述

--开始时的概括描述:

以分析结果和部署模型作为输入，然后通过主动类,把分析类（或其部分）所对应的设计映射到节点上,而后才能给出主动类的概要描述，其中:♦可以使用以前标识的子系统，或

♦需要对系统分解进行精化。

例如:使用分析类来概括描述主动类



该例使用分析类来概要描述主动类“支付请求处理”。在对分析类“定单处理”和“定单确认”进行设计时，首先必须考虑是否将它们主要部分分配到节点“采购员服务”上（这是一个设计决策问题），而后再考虑是否标识一个主动类“支付请求处理”，用于封装对这一主要部分的控制线程。

注意:对于表达一个关键过程的主动类，是否将其作为一个候选的可执行构件，这是实现中的任务。因此，如果在实现中把这样的主动类作为一个可执行构件，那么这一步把主动对象分配到一个节点上，就是实现期间把可执行构件分配到该节点的一个重要输入。

注意：

其中如果把一个子系统整个分配到一个节点时，那么该子系统的所有约束就应一起分布给该节点上的一个可执行构件。

任务4:标识处理一般性的设计机制

（Identifying General design mechanism）

首先,应研究共性需求，例如在分析期间所标识的有关用况细化[分析]中对设计类的特殊需求；

继之，应确定如何处理它们，给出可用的设计和实现技术，从而就得到一组可处理共性的设计机制。

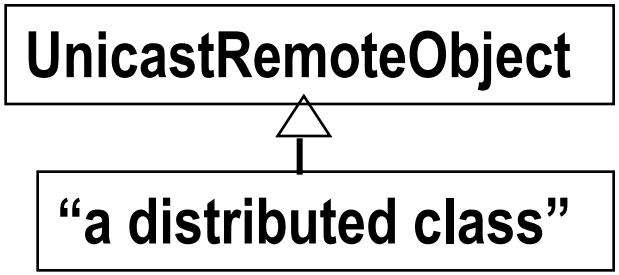
这样的控制机制本身可以表现为设计类，表现为协作，甚至可以表现为子系统。

设计机制所处理的共性需求，常常包括：

- 永久性；（Persistency）
- 透明对象的分布；（Transparent object distribution）
- 安全特征；（Security features）
- 错误检测与揭示；（Error detection and recovery）
- 事务管理等。（Transaction management）

例1：标识处理透明对象分布的设计机制

在网络银行软件中，“发票”显然是一个分布对象．为了实现该对象的分布，可以把每个需要分布到个节点的类，作为Java抽象类Java.rmi.UnicastRemoteObject的一个子类，支持远程消息调用（RMI），从而形成了一个处理分布对象的设计机制，如下：



设计机制：透明对象的分布

显然这一设计机制体现为设计类的一种泛化结构。

例2：标识事务管理的设计机制

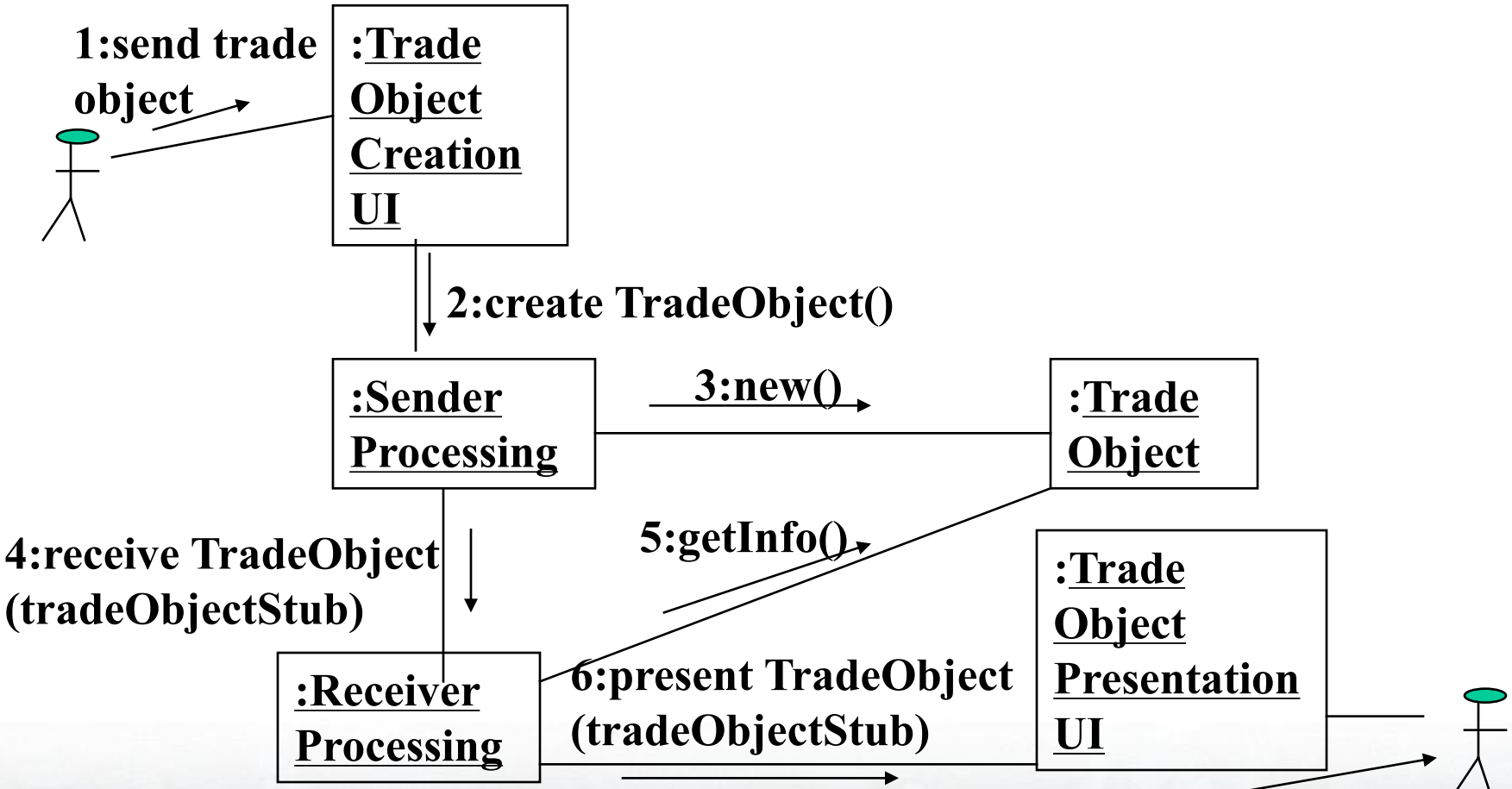
为事务管理设计一个相应的机制，一般使用用况细化中一个具有共性的协作。例如：

假定在体系结构描述中，使用用况及其它们的关系标识了参与者创建一个交易对象（例如定单或发票），并将其发送给另一参与者。

显然,可以把这标识为一个具有共性的事务管理，并且该事务管理的模式为：

- 当采购员决定定货或采购服务时，调用用况“订购货物或服务”。该用况允许采购员向销售员来指定并自动发送一个定单；
- 当采购员决定将发票送给一个采购员时，调用用况“采购员获取发票”, 该用况自动地将发票发送给采购员。

对这样一种具有一般性的行为，可以采用如下图所示的协作，将其设计为一种事务管理设计机制。



设计机制：事务管理

—通过一个通讯图，表明交易对象的创建\发送\接受

其中：

首先，对象“Trade Object Creation UI”接受来自参与者“ Sender actor”的信息,作为创建交易对象 “Trade Object”的输入；

接之，对象“Trade Object Creation UI”请求对象Sender Processing来创建该交易对象Trade Object .

而后,对象Sender Processing请求创建对应的类“Trade Object”来创建一个实例.

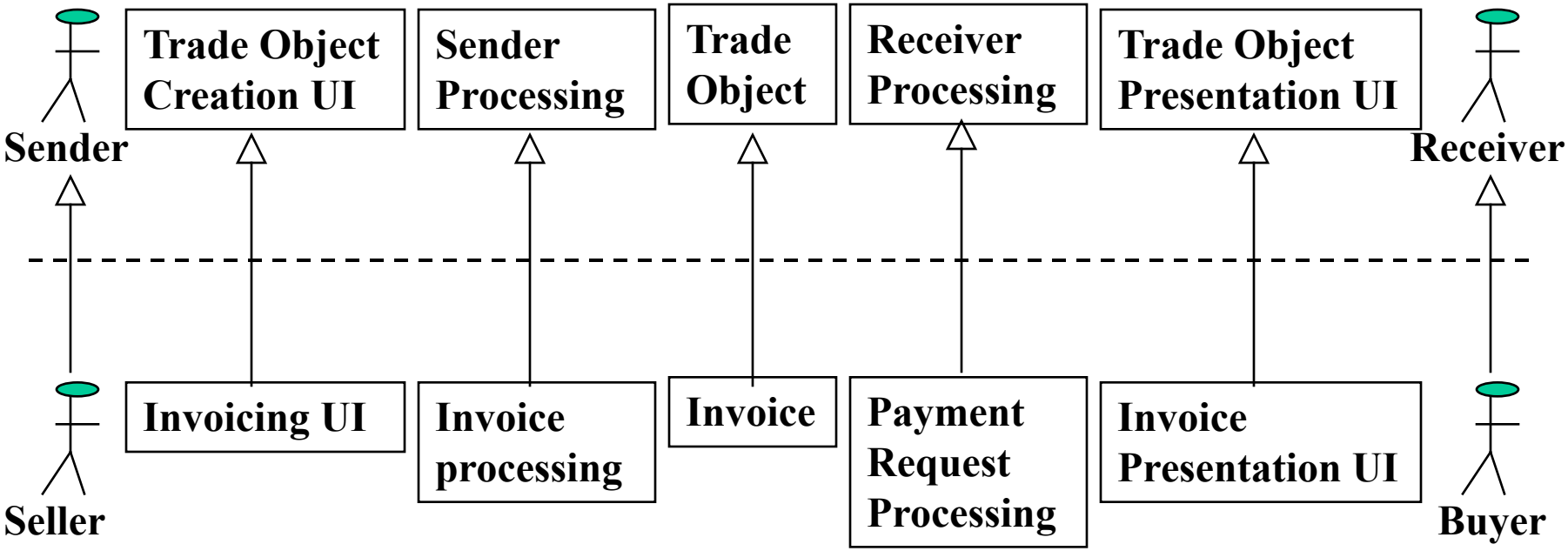
接之，对象“Invoice Processing”向对象“Receiver Processing”提交该交易对象的引用.

当需要时，对象“Receiver Processing”可以查询对象“Trade Object”，获取更多的信息.

对象“Receiver Processing”向对象“Trade Object Creation UI”发送具有更多内容的对象“Trade Object” .

最后,将对象Trade Object提交给接受的参与者.

当然在需要时（例如，在细化用况“Invoice”时），可以通过对参与该一般性协作的每一个抽象类目的子类型化，形成如下的具体类目：



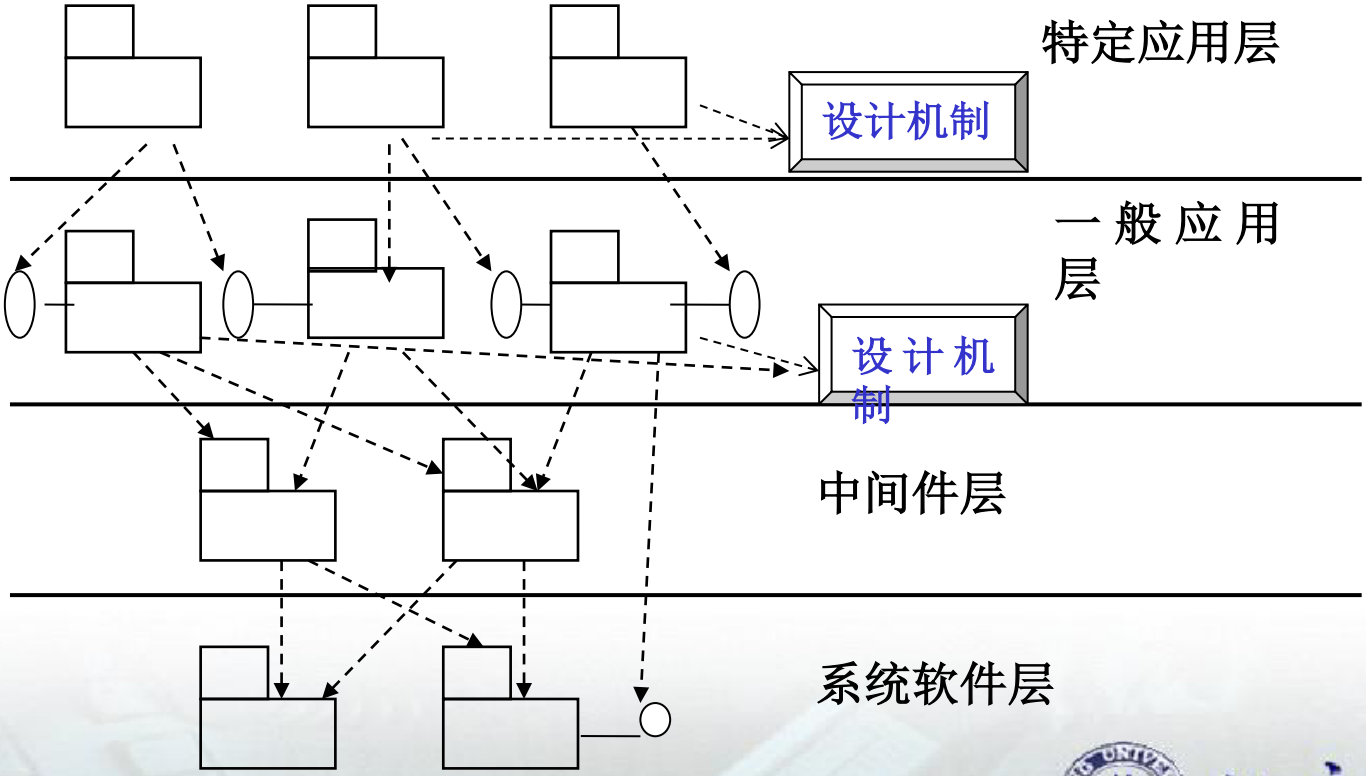
抽象类目的子类型化

以上两个例子是通过使用泛化和协作来创建设计机制的，除此之外，还可以使用其它方法，例如模式（模式是参数化的协作（例如参数化的类）），其中可以通过建立具体类（concrete classes）与参数的关联，来创建具有一般性的设计机制。

在有些情况中，随着用况的细化和设计类的揭示，才能发现一些必要的设计机制。但在RUP的精化阶段，应标识并设计大多数具有一般性的机制。

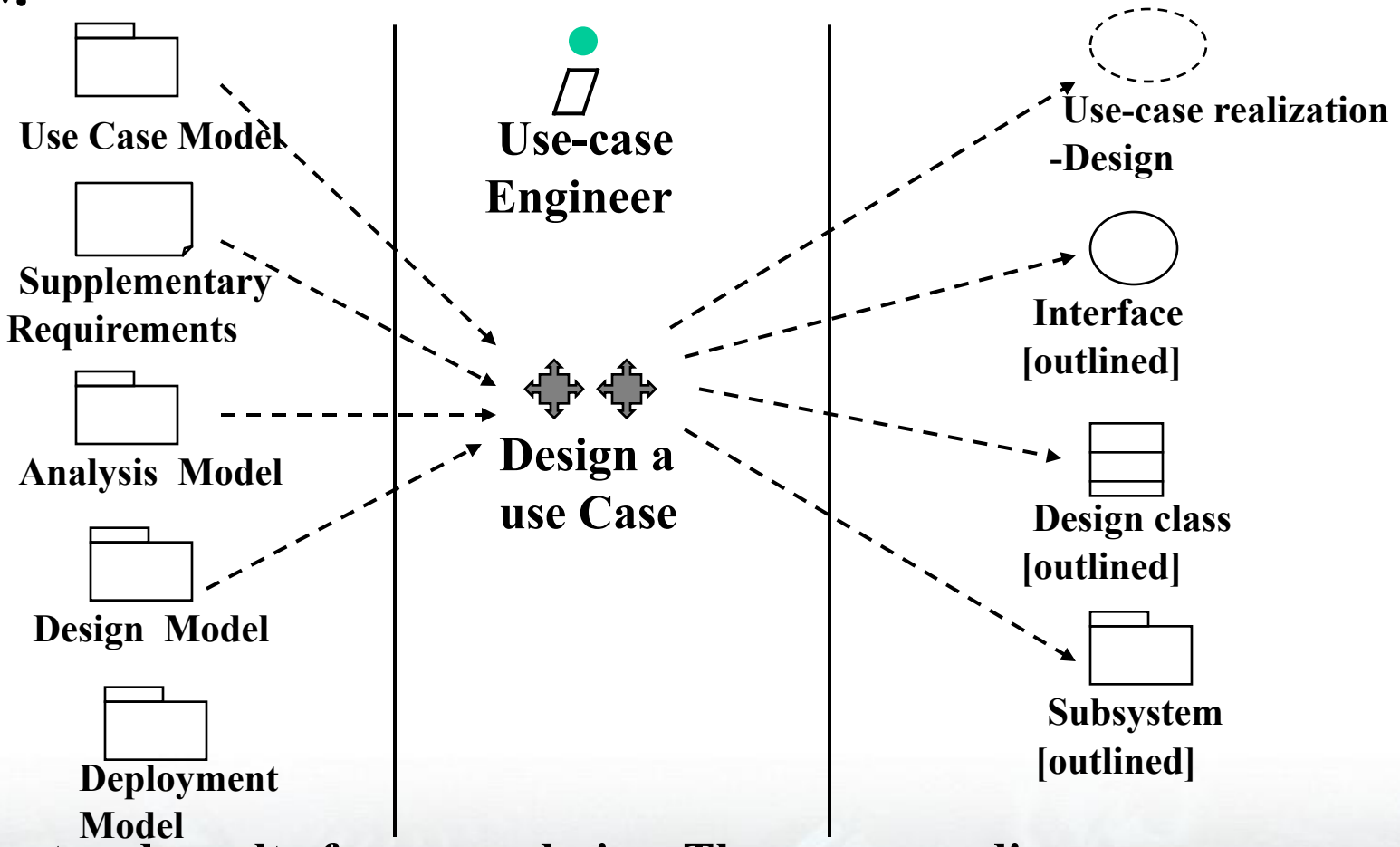
通过一组机制可以解决一些复杂的设计问题，并使构造阶段期间多数情况的细化变得相当简单和直接。

至此,完成了系统体系结构设计，除形成部署模型[概述]和该模型视角下的体系结构描述外，还形成了设计模型中如下形式的设计系统，即形成了系统的顶层设计-子系统结构。



活动2: Use Case 的设计

目标:



The input and result of use-case design. The corresponding use-case realization-analysis in the analysis model is an essential input to this activity.

为了实现以上所示的目标，可以采用以下两种方法。

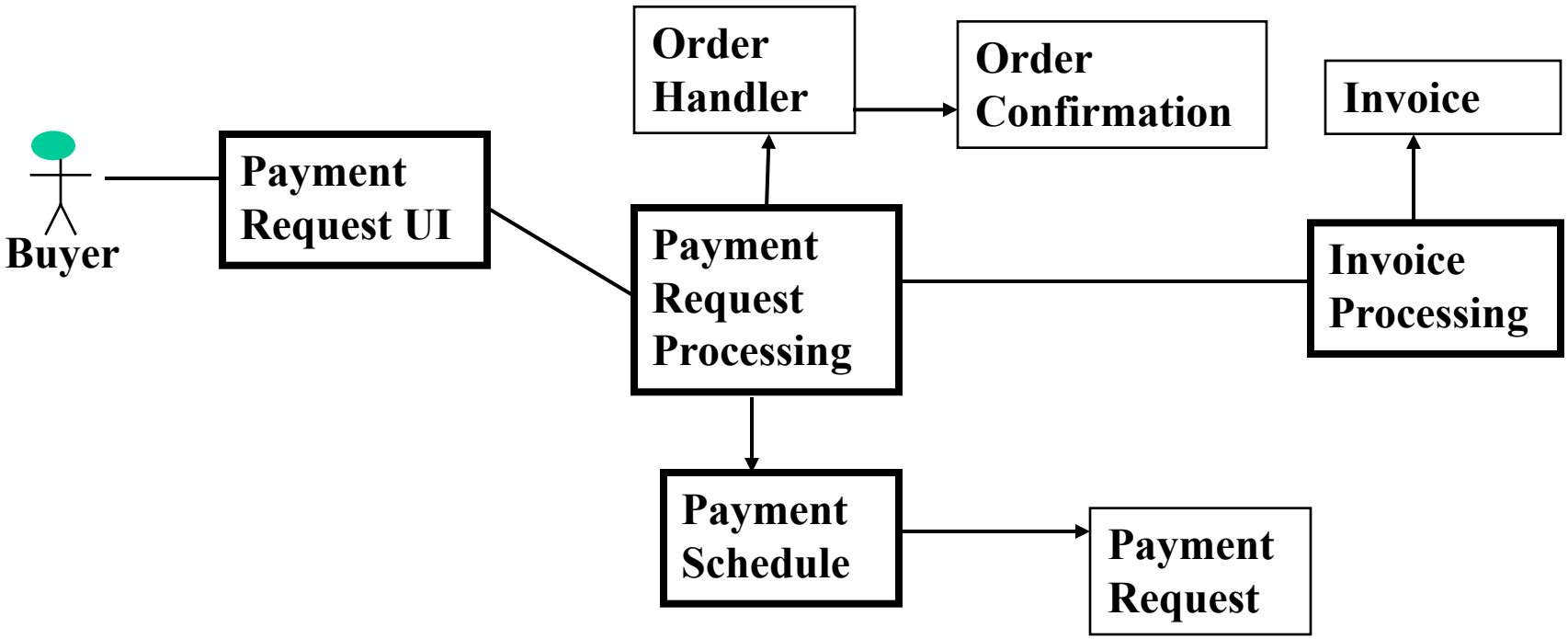
（1）第一种方法：

标识参与用况细化的设计类

首先,基于分析模型，研究相应用况细化[分析]中的分析类，来标识细化分析类所需要的设计类，研究相应用况细化[分析]中的特殊需求，来标识细化这些特殊需求所需要的设计类。这样标识的参与用况细化[设计]的设计类，或在体系结构设计期间已经发现，或在类设计期间予以创建。

继之,基于用况的功能，对每一标识的设计类赋予相应的责任。

最后,为该细化创建一个类图，汇聚参与该用况细化的设计类，并给出类之间的关系。例如：



注：有些设计类已经在任务1中发现，例如主动类 Payment Request Processing 和 Invoice Processing ,它们在不同节点之间传递 trade对象，从发出者到一个接受者，例如对象Invoice（发票）从seller到buyer，使系统 Interbank 得以运行。

描述设计对象交互和接口

在概述细化用况所需要的设计类的基础上，就应描述在该用况中这些设计对象是如何交互的。

这可使用顺序图，其中包括：参与者实例、设计类的对象、以及它们之间传送的消息。

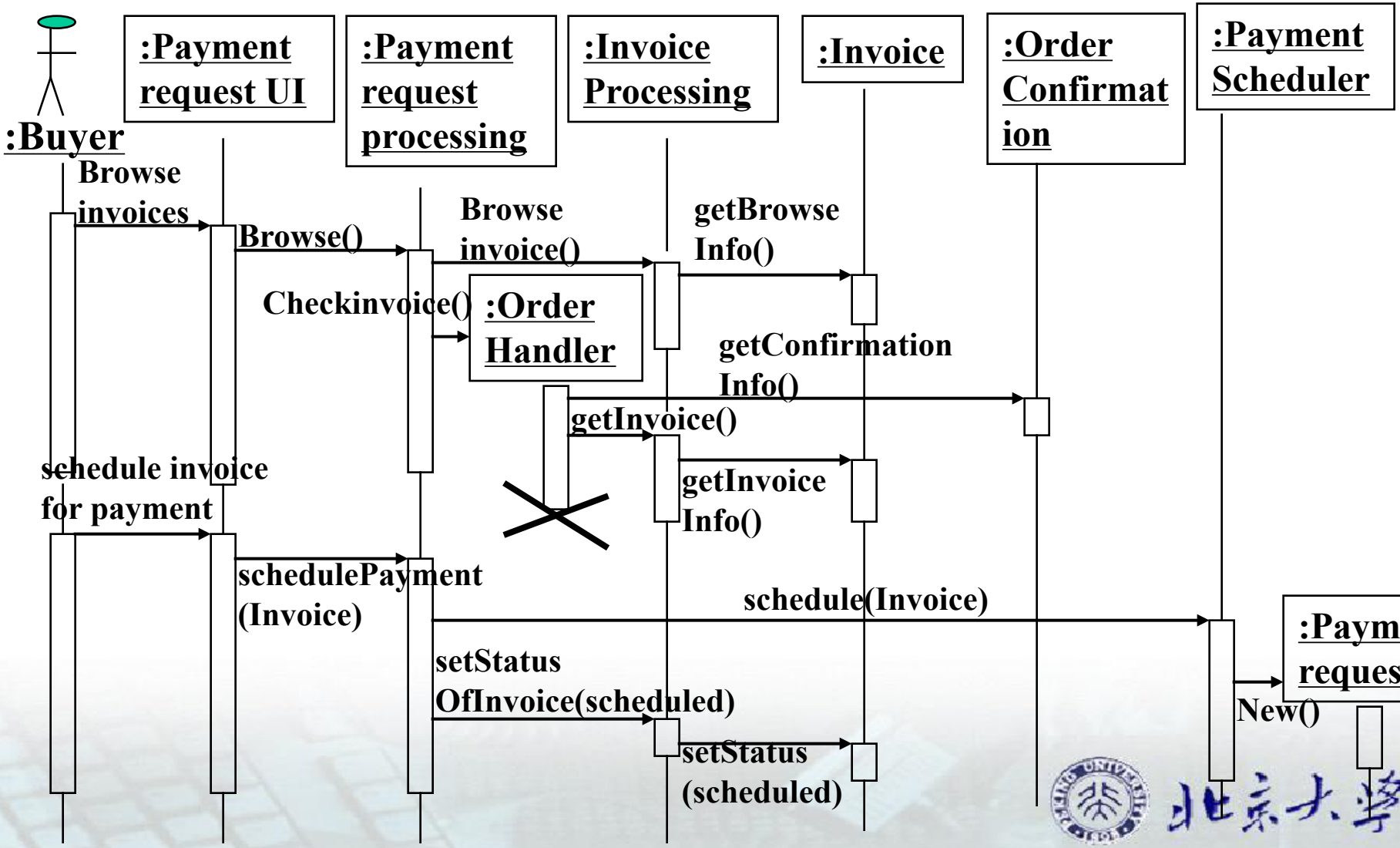
如果该用况存在一些不同的、有区别的流或子流，那么就应该为每一流创建一个顺序图，这有益于细化得更清晰，有益于抽取具有一般性、可复用的交互。

首先，应研究对应的用况细化[分析]，确定是否为了描述设计对象之间的交互而需要一些新的设计类。例如，为了描述设计类之间要求的消息序列，可能就需要填加一些新的设计类；而且在某些情况里，当要把用况细化[分析]的一个通讯图转化为初始的、相对应的一个顺序图，也需要增加这样的设计类。

继之，从该用况的一个流开始，一次遍历一个流，描述该用况细化中所要求的设计类实例与参与者实例之间的交互。在大多数情况里，可自然地发现对象在交互中的位置。

最后，详细描述交互图。其中在多数情况下可能会发现一些新的可选路径。这样的路径可以用该交互图的标号或用它们自身的交互图予以描述。

Example: Sequence Diagram for the design objects that perform part of the Pay Invoice Use Case realization.



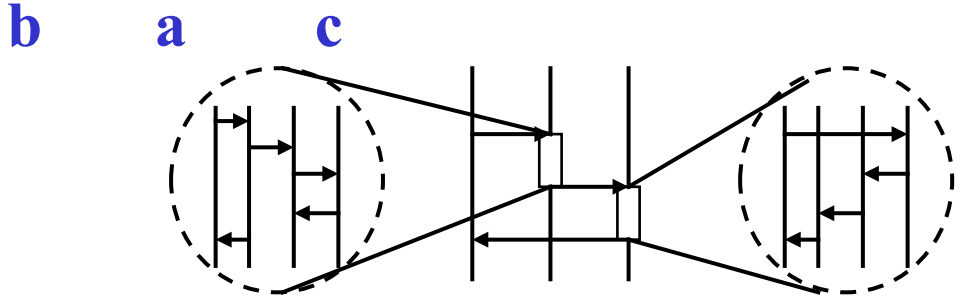
另外，当要增加更多的信息时，经常需要讨论一些新的例外，这些例外并没有在需求捕获和分析时予以思考，包括：

- 由于超时而导致节点或连接的的暂停。
- 人和机器参与者提供的错误输入。
- 中间件、系统软件或硬件所产生的错误消息等。

第二种方法：

标识参与用况细化的子系统和接口

在自顶向下开发期间，有时在进行子系统和其接口的设计之前，可能需要捕获一些有关子系统和接口的需求；另外，在一些情况里可能很容易给出子系统的另一设计，以替代一个子系统和它的特定设计。针对以上两种情况，就可能需要在层次体系的不同层上以子系统来描述一个用况细化的设计。例如：



使用参与use case细化的子系统及其接口对use case的设计

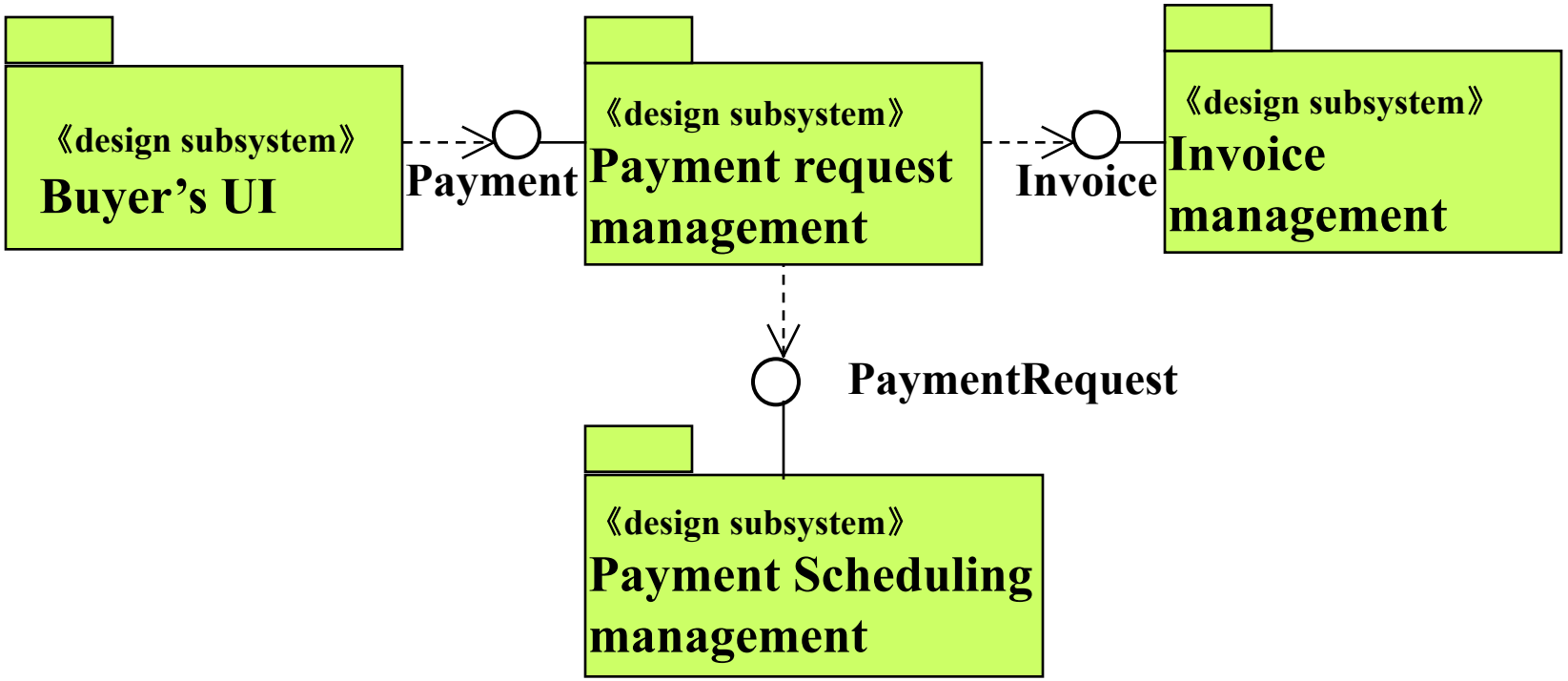
图（a）表示子系统的生命线和它们的消息。图（b）和（c）表示子系统的设计，并显示设计中的元素如何接受消息和发送消息。图（a）可以在图（b）、（c）之间设计。

可见，使用参与用况细化的子系统和/或它们的接口可以给出用况的另一种设计。其中，为了标识在细化用况中所需要的子系统：

❶应研究对应的用况细化[分析]， 标识包含这些分析类的分析包，继之标识可跟踪到这些分析包的设计子系统。

❷应研究对应用况细化[分析]的特殊需求，标识细化这些特定需求的设计类，继之标识包含这些类的设计子系统。

最后，开发一个类图中，汇聚参与该用况细化的子系统，给出这些子系统之间的依赖，并给出用况细化中所使用的接口。例如：



A class diagram that includes subsystems, interfaces, and their dependencies, involved in the first part of the Pay Invoice use case.

当已经勾画了细化用况所需要的那些子系统之后，就应描述子系统的交互，即在系统层上描述所包含的类对象是如何交互的，其中可以使用顺序图，包含参与交互的参与者实例、子系统、以及它们之间消息传送。

与描述设计对象交互相比，在描述子系统的交互图中，生命线表示的是这时标识的每一子系统，而不是设计对象；一个消息指向一个接口的操作，而不是指向另一个子系统，以区分该消息使用子系统的哪一个接口。

以上是用况设计的另一种方法，其中使用了参与用况细化的子系统和接口。

最后,还要捕获实现需求

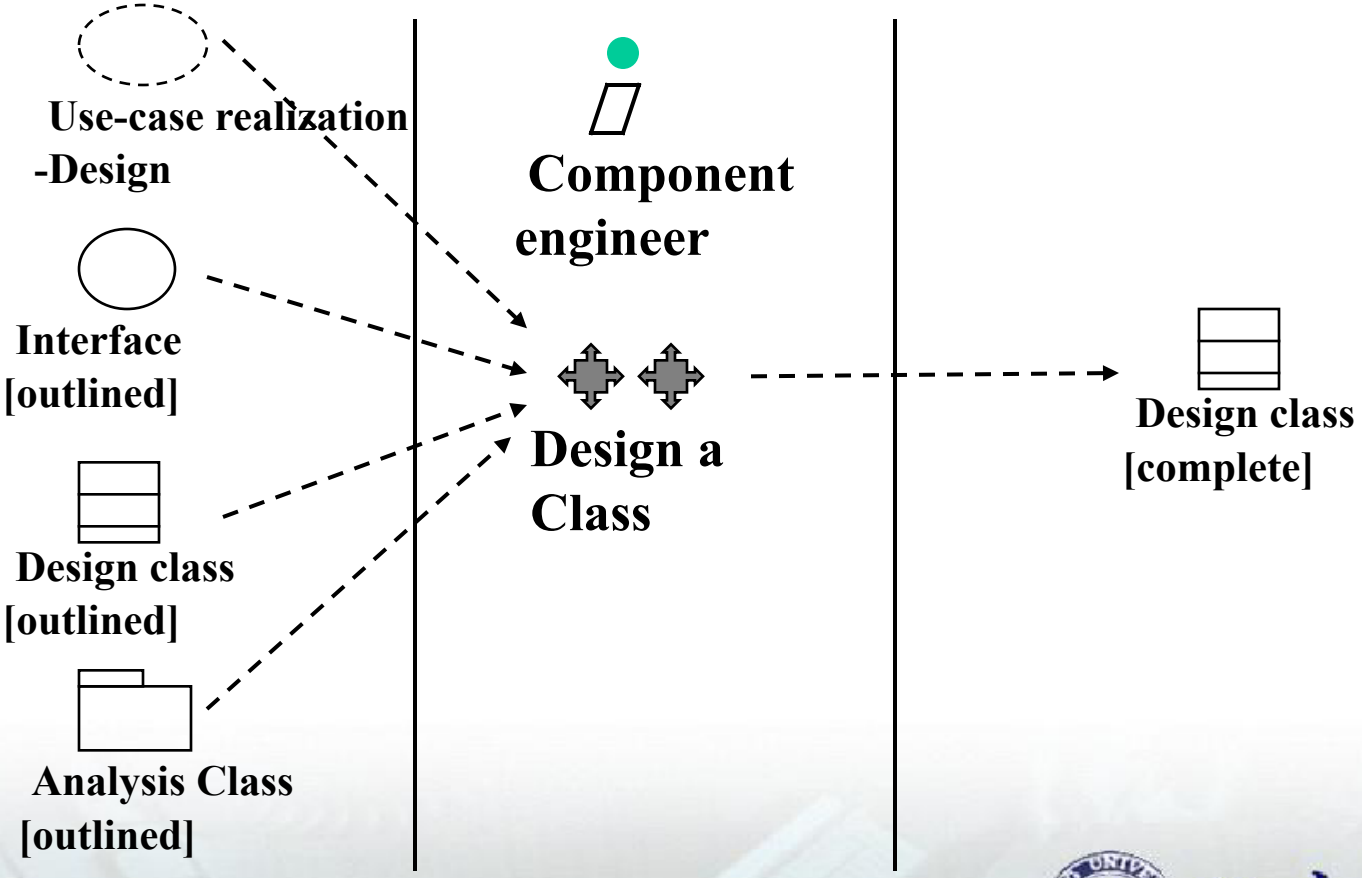
（Capturing Implementation Requirements）

其中,应捕获use-case细化中的所有需求，例如在设计中标识的、应该在实现中予以处理的非功能需求。例如：

an object of the (active) class Payment Request Processing should be able to handle 10 different buyer clients without a perceivable delay for any individual buyer.

活动3: 类的设计

目标：进行类的设计，使之完成在use-case 细化中的角色，并完成针对它的非功能需求。



The input and result of designing a class.

一个类的设计，包含以下方面：

- 类的操作；
- 类的属性；
- 参与的关系；
- 类方法；
- 类的状态
- 对一般设计机制的依赖；
- 与实现有关的需求；
- 所提供的那些接口的细化。

任务1:概括描述设计类（Outlining the Design Class）

即把分析类和/或接口作为给定的输入，来勾画一个或多个设计类。 其中，

❶当给定的输入是一个接口，那么就可以简单的、直接的指定一个提供该接口的设计类。

❷当给定的输入是一个或多个分析类，那么所使用的方法就要依赖分析类的衍型（stereotype）：

- 如果输入的是表示永久信息的实体类，其设计经常隐含地需要使用一种特定的数据库技术；
- 如果输入的是边界类，其设计需要使用一些特定的接面技术；

- 如果输入的是控制类，其设计是一件很辣手的问题。由于控制类有时封装了一些对象的协作，有时封装了业务逻辑，因此往往需要考虑以下问题：

  - 分布问题，即如果控制次序需要在不同节点之间予以分布和管理，那么为了细化该控制类，就可能需要把一些类分别分布到不同的节点上。

  -执行问题，即对这样控制类的细化，可能就没有什么理由需要给出一些不同的设计类。但是往往需要使用由相关的边界类和/或实体类所得到的设计类，来细化这一控制类。

  -事务问题，即这样的控制类经常封装事务，因此对它们的设计应结合现在使用的任一事务管理技术。

任务2：标识操作

一般应依据分析类来标识设计类所提供的、所需要的操作，其中需要使用程序设计语言的语法（syntax）来描述所标识的操作。

❶分析类的一个责任常常隐含了一个或多个操作。另外，如果已经描述了责任的输入和输出，那么就可以使用这些输入和输出，来勾画操作的形式参数和结果值。

❷对于分析类的特殊需求，经常可能需要结合设计模型中某一具有一般性的设计机制或技术（例如数据库技术）予以处理。

❸对于分析类的接口，其中的操作也需要由相应的设计类予以提供。

❹对于参与use-case 细化[设计]中的设计类，应通过走查用况细化，一是研究在该细化图中和事件流的描述中包含该类和它的对象，二是发现它们的角色。在此基础上，为其设计一些支持该类在不同用况细化中所扮演角色的操作。

任务3：标识属性

使用程序设计语言的语法给出属性描述。

一个属性规约了设计类的一个特性，该属性通常由该类的操作所隐含的和要求的。

在标识分析类的属性中，应：

- 考虑设计类所对应的分析类的属性，其中这些分析类的属性隐含了该设计类有关属性的需求。
- 通过程序设计语言对可用的属性类型进行约束。其中当选择类型时，尽量复用已有的一个类型。
- 一个单一的属性实例，不能被多个设计对象所共享。如果希望共享的话，就应该把该属性定义为一个设计类。

- 如果设计类的属性理解起来相当困难，那么就应该对其中的一些属性进行分离，成为该设计类自身拥有的一些类。
- 如果一个类具有很多或复杂的属性，则可以使用一个专门类图，显示属性的各个组件以及它们之间的关系。

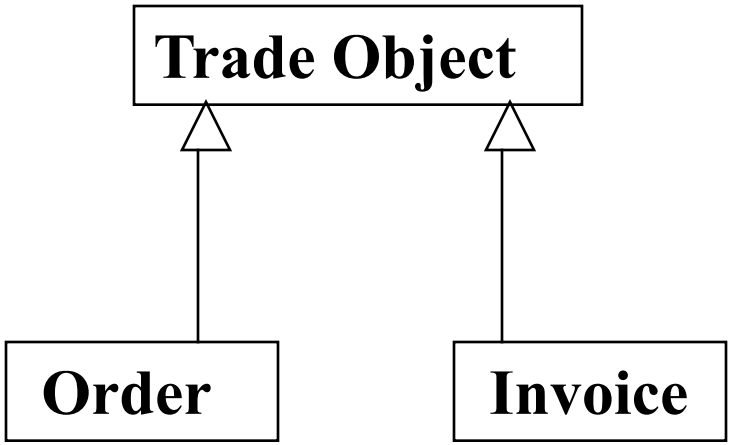
任务4：标识关联和聚合

在标识并精化关联和聚合中，应：

- 考虑对应的分析类所具有的关联和聚合。有时，在分析模型中的这些关系，隐含了所涉及的设计类对关系的需要。
- 在所使用的程序设计语言的支持下，精化关联的多重性、角色名、关联类、定序的角色、限定角色以及N元关联等。例如：在生成代码时，角色名可能成为设计类的属性；或一个关联类可能成为另外两个类之间新的类，因此在“关联类”和另外两个类之间需要一个新的、具有适当多重性的关联。
- 考虑使用该关联的交互图，来精化关联的导航性。另外，设计对象之间消息转送方向，隐含了它们类之间关联的导航性。

任务5：标识泛化

基于分析模型中分析类之间的泛化，可以发现设计模型中的很多泛化。例如：

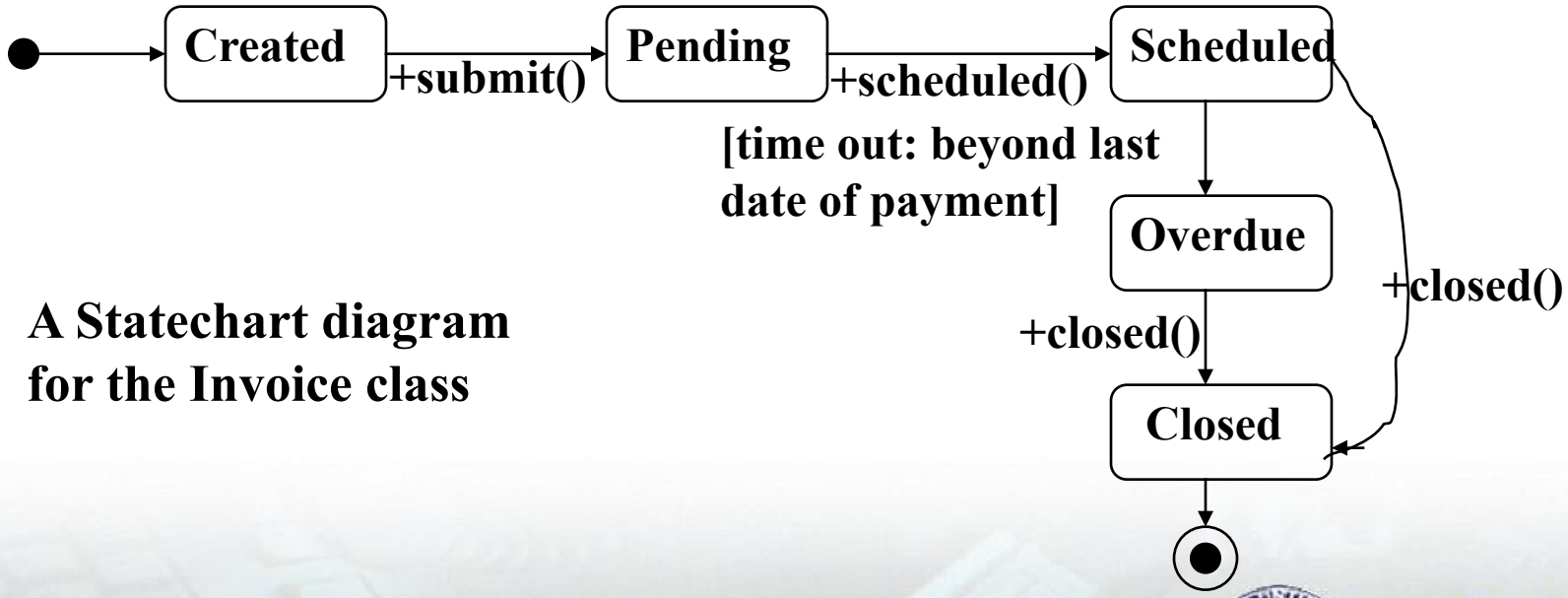


任务6：描述方法

在设计期间，对方法的规约可以使用自然语言，或适当地使用伪码。

任务7：描述状态

有些设计对象是受状态控制的，即它们的状态确定了在它们接受一个消息时的行为。在这样情况下，使用一个状态图来描述一个对象的不同状态转移是有意义的。例如：类“发票”的状态可设计为：



A Statechart diagram for the Invoice class

其中：

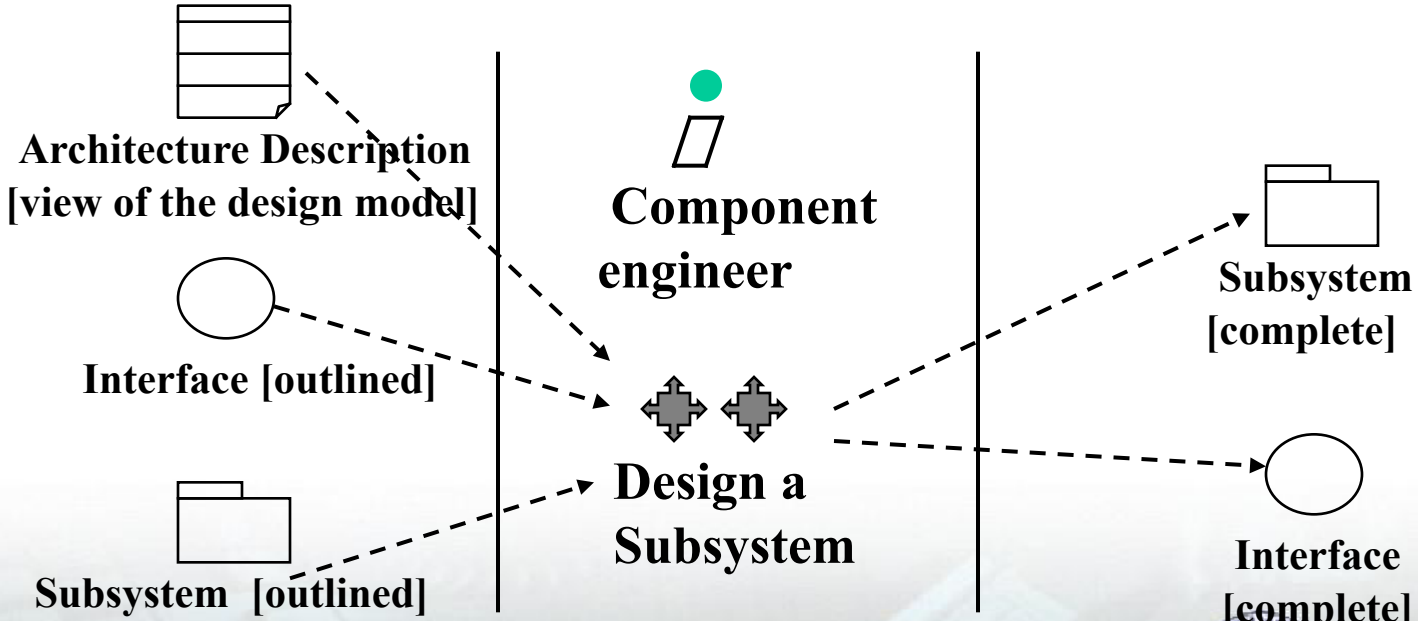
- 当一个seller希望buyer来支付一个定单（order）时， 则invoice 为创建状态（ created）；
- 接之，当该 invoice交给 buyer时，其状态变为 pending；
- 当 buyer决定支付时，该 invoice的状态又变为 scheduled；
- 接之，如果该 invoice予以支付了，则状态变为 closed。

一个状态图对相应设计类的实现来说是一个有价值的输入。

活动: 子系统设计

该活动的目标是：

- 确保子系统尽可能独立于其它子系统或它们的接口。
- 确保子系统提供正确的接口。
- 确保子系统实现了（**fulfills**）它的目标，即给出了该子系统提供的那些接口所定义的操作的细化。

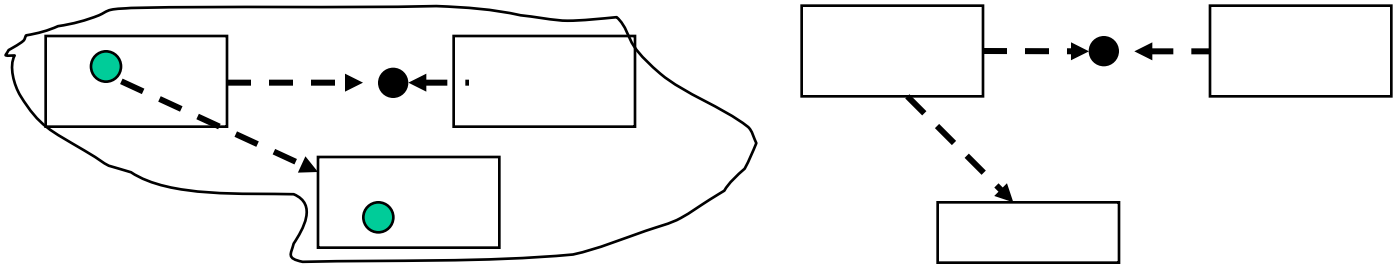


The input and result of designing a subsystem.

任务1：维护子系统依赖

--定义其他子系统所包含的元素与该子系统中元素之间的关

--如果一个子系统为其它子系统提供了一些接口，那么就应当说明这些子系统如何通过接口与该子系统发生依赖。



其中，最好是依赖一个接口，而不依赖一个子系统。因为一个子系统可能会用另一设计子系统所替代，但不需要替换该接口。

还要考虑重新安排所包含的、过多依赖其它子系统的类，以此尽量减少对其它子系统和/或接口的依赖，以实现子系统之间的低耦合。

任务2：维护子系统所提供的接口

一个子系统一般提供了一些接口，由这些接口所定义的操作需要支持该子系统在不同用况细化中所扮演的所有角色。尽管这些接口已经予以勾画，但当设计模型演化时和当用况设计时，它们也需要予以精化。因为在不同用况细化中，可能要使用一个子系统和它的接口，由此对这些接口提供了进一步的需求。

任务3： 维护子系统内容

当对一个子系统的接口所定义的操作给出了一个正确的细化，则该子系统就完成了它的目标。尽管该子系统的内容已予勾画，但当设计模型演进时，也可能需要予以精化。与其相关的一般问题有：

- 对于由该子系统提供的每一接口，在该子系统中存在一些设计类或其它子系统，它们还提供了这一接口。
- 为了说明一个子系统的内部设计怎样细化它的接口或use cases，可能要使用该子系统包含的元素来创建一些协作，以验证该子系统应当包含那些元素。

RUP设计小结

1）RUP的设计方法，由三部分组成：

①给出用于表达设计模型中基本成分的四个术语，包括：子系统、设计类、接口和用况细化[设计]；

②规约了设计模型的语法，指导模型的表达；

③ 给出了创建设计模型的过程以及相应的指导。

2)RUP设计的突出特点:

❶使用了一种公共的思想，来思考该设计，并是可视化的。

❷可以获得对有关子系统、接口和类的需求，为以后的实现活动创建一个合适的输入,即为系统的实现，创建一个无缝的抽象，在一定意义上讲，使实现成为设计的一个直接的精化-填加内容，而不改变其结构。这样就可以在设计和实现之间，使用代码生成技术，反复不断地实现.

❸可以对实现工作进行分解，成为一些可由不同开发组尽可能同时处理的、可管理的部分;

（注：这一分解不可能在需求获取或分析中完成。）

❹可以在软件生存周期的早期，捕获子系统之间的主要接口。这有助于不同开发组之间思考有关体系结构问题并合理使用接口，以提高设计质量.

3）设计的主要结果是系统的设计模型，它尽量保持该系统具有分析模型的结构，并作为系统实现的输入。

包括以下元素：

① 设计子系统和服务子系统，以及它们的依赖、接口和内容。

其中，可以依据分析包来设计上面两层（即特定应用层和一般应用层）的设计子系统。有关设计子系统之间的依赖，有些是基于所对应的分析包的依赖而设计的；有关子系统的接口，有些是依赖分析类而设计的。

② 设计类（其中包括一些主动类），以及它们具有的操作、属性、关系及其实现需求。

一般地，在进行设计类的设计时，分析类作为它们的规约，特别地有些主动类是基于分析类并考虑并发需求而设计的。

③ 用况细化[设计]。

它们描述了用况是如何设计的，其中使用了设计模型中的协作。一般地，在进行用况细化[设计]设计时，用况细化[分析]作为它们的规约。

④ 设计模型视觉下的体系结构描述，其中包括对一些在体系结构方面有重要意义元素的描述。

如以前指出的，在进行设计模型视觉下体系结构方面具有意义的元素设计时，分析模型视觉下的那些在体系结构方面有意义的元素作为它们的规约。

4）RUP的设计还产生了部署模型，描述了网络配置，系统将分布于这个配置上

部署模型包括：

- 节点, 它们的特征以及连接；
- 主动类到节点的初始映射；

部署模型视觉下的体系结构描述，包括对那些在体系结构方面具有重要意义元素的描述。

5）设计模型和分析模型的简要比较

分析模型	设计模型
❶ 概念模型, 是对系统的抽象, 而不涉及实现细节；	❶ 软件模型, 是对系统的抽象,而不涉及实现细节；
❷ 可应用于不同的设计；	❷ 特定于一个实现；
❸使用了三个衍型类： 控制类、实体类和边界类；	❸ 使用了多个衍型类, 依赖于实现语言；
❹ 几乎不是形式化的；	❹ 是比较形式化的；
❺开发的费用少（相对于分析是1:5）	❺开发的费用高（相对于设计是5:1）
❻ 层少Few layers；	❻ 层多；

❼动态的，但很少关注定序方面；❼ 动态的，但更多关注定序方面；

❽概括地给出了系统设计，包括系统的体系结构；❽ 表明了系统设计，包括设计视觉下的系统体系结构

❾整个软件生存周期中不能予以修改、增加等；❿整个软件生存周期中应该予以维护；

❿为构建系统包括创建设计模型，定义一个结构，是一个基本输入；❿ 构建系统时，尽可能保留分析模型所定义的结构

6)设计阶段的活动：

序号	输入	活动	执行者	输出
1	用况模型、补充需求、分析模型、体系结构描述$_{分析模型角度}$	体系结构设计	体系结构设计者	子系统$_{概述}$、接口$_{概述}$、设计类$_{概述}$、部署模型$_{概述}$、体系结构描述$_{设计、部署模型角度}$
2	用况模型、补充需求、分析模型、设计模型、部署模型	设计用况	用况工程师	用况$_{实现-设计}$、设计类$_{概述}$、子系统$_{概述}$、接口$_{概述}$
3	用况$_{实现-设计}$、设计类$_{概述}$、接口$_{概述}$、分析类$_{完成}$	对类设计	构件工程师	设计类$_{完成}$
4	体系结构描述（从设计模型角度）、子系统$_{概述}$、接口$_{概述}$	设计子系统	构件工程师	子系统$_{完成}$、接口$_{完成}$

其中：活动1 把分析模型的分析包变为设计模型的子系统。活动2中的用况里的各个流应该各对应一个顺序图。

7) 对实现的影响

由于设计模型和部署模型是以后实现和测试活动的基本输入，因此要强调的是：

- 设计子系统和服务子系统是由实现子系统予以实现的，这些实现子系统包括一些构件，例如源代码文件、脚本以及二进制、可执行的构件等。这些实现子系统可跟踪到设计子系统。
- 设计类将由文件化构件予以实现的，它们包括源代码。一般地，一些不同的设计类在一个单一的文件化构件中实现，尽管这依赖所使用的程序设计语言。另外，当要寻找可执行的构件时，将要使用那些描述“权重”处理的主动类。

- 在规划实现工作时，将要使用用况细化[设计]，以产生一些“构造”（**Bulid**），即系统的一个可执行的版本。每一个构造将实现一组用况细化或部分用况细化。
- 在节点上部署构件，形成分布系统时，将使用部署模型和网络配置。

附： 下面仅对核心工作流中的实现和测试作简单介绍。

1、RUP的实现

RUP实现的目标是：基于设计类和子系统，生成构件；对构件进行单元测试，进行集成和连接；并把可执行的构件映射到部署模型。其主要活动以及其输入/输出为：

序号	输　入	活动	执行者	输　出
1	设计模型、部署模型、体系结构描述$_{\text{设计模型、部署模型角度}}$	实现体系结构	体系结构设计者	构件$_{\text{概述、体系结构描述}_{\text{实现模型、部署模型角度}}}$
2	补充需求、用况模型、设计模型、实现模型$_{\text{当前建造}}$	集成系统	系统集成者	集成建造计划、实现模型$_{\text{连续的建造}}$
3	集成建造计划、体系结构描述$_{\text{实现模型角度}}$、设计子系统$_{\text{已设计}}$、接口$_{\text{已设计}}$	实现子系统	构件工程师	实现子系统$_{\text{建造完成}}$，接口$_{\text{建造完成}}$
4	设计类$_{\text{已设计}}$、接口$_{\text{由设计类提供}}$	实现类	构件工程师	构件$_{\text{完成}}$
5	构件$_{\text{完成}}$、接口	完成单元测试	构件工程师	构件$_{\text{已完成单元测试}}$

其中：

① 实现模型

该模型描述了设计模型中的元素是如何用构件实现的，描述了构件间的依赖关系，并描述如何按实现环境来组织构件。

可见，构件是设计模型中元素的一种实现，是对这样元素的物理封装，要把它映射到部署模型的节点上。

② 实现子系统

实现子系统是由构件、接口和其它子系统组成的。其中，接口用于表示由构件和实现子系统所实现的操作。在这一阶段可以使用设计时的接口。

③ 实现模型视角下的体系结构描述

实现模型视角下的体系结构描述，包括对由实现模型分解的子系统、子系统间的接口、子系统间的依赖以及关键构件的描述。其中，相应设计子系统中的每个类和每个接口，都要由实现子系统中的构件实现。

④ 实现类

该活动对每一个涉及源代码的文件构件，根据相应的设计类来生成其代码，为设计类提供操作的方法，其中应使构件提供的接口与设计类提供的接口相一致。

在实施RUP的实现中，涉及集成计划的开发问题。在增量开发中，每一步的结果即为一个构造。在一个迭代中，可能创建一个构造序列，即构造集成计划。

2、RUP的测试

RUP的测试包括内部测试、中间测试和最终测试，其主要活动为：

序号	输　　入	活动	执行者	输出
1	补充需求、用况模型、分析模型、设计模型、实现模型、体系结构描述模型的体系结构角度	计划测试	测试工程师	测试计划
2	补充需求、用况模型、分析模型、设计模型、实现模型、体系结构描述模型的体系结构角度、测试计划策略、时间表	设计测试	测试工程师	测试用况 测试过程
3	测试用况、测试过程、实现模型被测试的建造	实现测试	构件工程师	测试构件
4	测试用况、测试过程、测试构件、实现模型被测试的建造	执行集成测试	集成测试者	缺陷
5	测试用况、测试过程、测试构件、实现模型被测试的建造	执行系统测试	系统测试者	缺陷
6	测试计划、测试模型、缺陷	评价测试	测试工程师	测试评价

特别是，当细化阶段中体系结构基线变为可执行时，当构造阶段中系统变为可执行时，以及当移交阶段中检测到缺陷时，都要进行测试。

其中：

① 测试模型

主要描述系统测试者和集成测试者如何实施对在实现中可执行构件的测试，并描述如何测试系统的特殊方面(如用户接口、可用性、一致性以及用户手册是否达到目的等)。因此，测试模型是用况测试、过程测试和构件测试的一个集合体。

② 测试用况描述测试系统的方式。一般描述如何测试用况(或部分)，包括输入、输出和条件。

③ 测试过程描述怎样执行一个或几个测试用况，也可以描述其中的片段。

④ 测试构件用于测试实现模型中的构件。测试时，要提供测试输入，并控制和监视被测构件的执行。用脚本语言描述或编程语言开发测试构件，也可以用一个测试自动工具进行记录，以对一个或多个测试过程或它们片段进行自动化。

⑤ 测试计划描述测试策略、资源和时间表。测试策略包括对各迭代进行测试的种类、目的、、级别、代码覆盖率以及成功的准则。

⑥ 缺陷描述系统的异常现象。

⑦ 评价测试描述在一次迭代中对测试用况覆盖率、代码覆盖率和缺陷情况(可绘制缺陷趋势图)的评价。其中，为了度量需要把评价结果与目标进行比较。

其中，在活动2中，应对每个建造都要设计相应的集成测试用况、系统测试用况和回归测试用况；在活动4中，应对每次迭代中的各个建造都要执行集成测试，当集成测试满足当前迭代计划中的目标时，要进行活动5。

关于RUP的结束语

RUP是一种软件开发过程框架，基于面向对象符号体系给出了有关软件开发过程组织及实施的指导。该框架体现了三个突出特征，即以用况驱动、以以体系结构为中心以及迭代、增量式开发。

RUP和UML是一对“姐妹”，构成了一种特定软件开发方法学的主体。UML作为一种可视化建模语言，给出了表达对象和对象关系的基本术语，给出了多种模型的表达工具，而RUP利用这些术语，定义了需求获取层、系统分析层、设计层、实现层，并给出了实现各层模型之间映射的基本活动以及相关的指导。