# 大象 – ThinkinUML

# 面向对象

## 面向过程与面向对象

* 以下为人们采用不同方法来认识和描述这个世界时的工具，它们都只是表征而不是本征：

1. 面向过程

结构化程序设计、DFD图、ER模型、UC矩阵等

1. 面向对象

继承、封装、多态 、复用等

* 面向对象不仅仅是解决复用问题的，而是一种认知论，它能够构造出更为复杂的系统来解释越来越复杂的现实世界。
* 面向过程认知：

世界不是孤立的，相互联系、相互影响、相互作用，由小构大，因果关系。

发展瓶颈：过程不再稳定、结果不能再能预设，牵一发而动全身。

面向对象认知：

世界是彼此分割的，通过外界条件使得它们相互间产生联系。

## 抽象

* 现实世界和对象世界之间的鸿沟
* 跨越鸿沟，需要

1. 一种把现实世界映射到对象世界的方法；
2. 一种从对象世界描述现实世界的方法；
3. 一种验证对象世界行为是否正确反映了现实世界的方法。

# 建模语言 —— UML

## OOD: 面向对象设计

* 从设计中推导对象
* Booch86、GOOD（通用面向对象设计）、HOOD（层次化面向对象设计）、OOSE（面向对象结构设计）

## OOA： 面向对象分析

* 从需求中推到对象
* (Booch，OOSE、Martin/Odell，OMT、Shlaer/Mellor) 🡪 UM (统一方法)

## UML（统一建模语言）

* UML： Unified Modeling Language
* UML本身不包含软件方法，仅是一种语言
* 语言 = 基本词汇 + 语法

1. UML的基本元素（词汇）： 元模型，如用例、类等
2. UML的语法： 视图（view）

* 统一： 使用统一的语言覆盖整个软件过程
* 可视化： 让隐晦的含义可视化
* 从现实世界至业务模型 🡪 从业务模型至概念模型 🡪 从概念模型至设计模型

1. 从现实世界至业务模型 【需求阶段】
   1. 对现实世界抽象：人、事、物、规则
   2. 建立模型： 有什么人【中心】，什么人做什么事【过程】，什么事产生什么物【结果】，中间有什么规则【控制】
   3. UML中的参与者（actor）元模型 🡪 现实世界中的人



* 1. UML中的用例（use case）元模型 🡪 现实世界中的事



* 1. UML中的业务场景(business scenario)和用例场景(use case scenario) 🡪 现实世界中的规则
  2. UML中的业务对象模型（business object model） 🡪 现实世界中的物

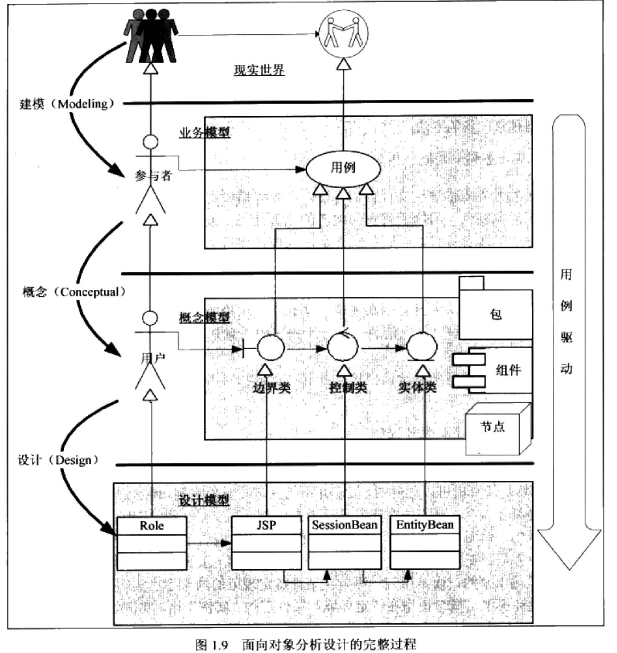
1. 从业务模型至概念模型 【开发阶段】
   1. UML通过被称之为概念化的过程（Conceptual)来建立适合计算机理解和实现的模型，这个模型称为分析模型（Analysis Model)。分析模型介于原始需求和计算机实现之间，是一种过渡模型。分析模型向上映射了原始需求，计算机的可执行代码可以通过分析模型追溯到原始需求；同时，分析模型向下为计算机实现规定了一种高层次的抽象，这种抽象是一种指导，也是一种约束，计算机实现过程非常容易遵循这种指导和约束来完成可执行代码的设计工作。
   2. 绘制分析模型最主要的元模型有：

* 边界类（boundary）: 任何一件事物都分为里面和外面，外面的事物与里面的事物之间的任何交互都需要有一个边界。边界决定了外面能对里面做什么“事”。静态的。对应现实世界中的“事”。
* 实体类（entity）: 用于表述业务模型中的业务实体，包涵了表示计算机逻辑关系的控制信息如包、组件等，是业务实体实例化结果。静态的。对应现实世界中的“物”。
* 控制类（control）: 表述原始需求中的动态信息，即业务模型中的业务场景和用例场景中的步骤和活动。边界类和实体类之间、边界类和边界类之间、实体类和实体类之间不能够直接访问，需要通过控制类来代理访问要求。对应现实世界中的“规则”。

1. 从概念模型至设计模型
   1. 设计模型是建造零部件，组装汽车的过程。
   2. 概念模型中的边界类 🡪 设计模型中的操作界面或者系统接口。
   3. 概念模型中的控制类 🡪 设计模型中的计算程序或控制程序，如工作流、算法体等。
   4. 概念模型中的实体类 🡪 设计模型中的数据库表、XML文档或者其他带有持久化特性的类。
   5. 概念模型到设计模型转化过程遵循的规则：

* 软件架构和框架：规定了实现类必须实现的接口、必须继承的超类、必须遵守的编程规则等。例如，采用J2EE架构时，Home和Remote接口是必需的。
* 编程语言：各类编程语言有不同的特点，例如在实现一个界面或者一个可持久化类时，采用C++还是Java作为开发语言会有不同的设计要求。
* 规范或中间件：如果决定采用某个规范或采用某个中间件时，实现类还要遵循规范或中间件规定的那些必需特性。

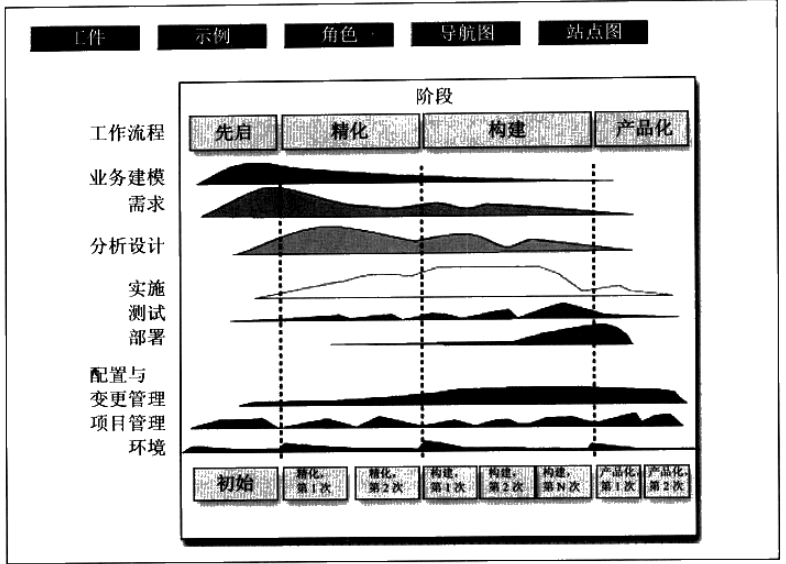
## 面向对象分析设计完整过程



# 建模方法 —— RUP

## RUP（统一过程）

* RUP：Rational Unified Process，统一过程
* RUP归纳和集成了软件开发活动中的最佳实践，是一个采用了面向对象思想，使用UML作为软件分析设计语言，并且结合了项目管理、质量保证等许多软件工程知识综合而成的一个非常完整和庞大的软件方法。
* RUP定义了软件开发过程中最重要的阶段和工作（4个阶段和9个核心工作流），定义了参与软件开发过程的各种角色和他们的职责，定义了软件生产过程中产生的工件，并提供了模板，最后采用演进式软件生命周期（迭代）将工作、角色和成果物串在一起，形成了统一过程。



## RUP与UML

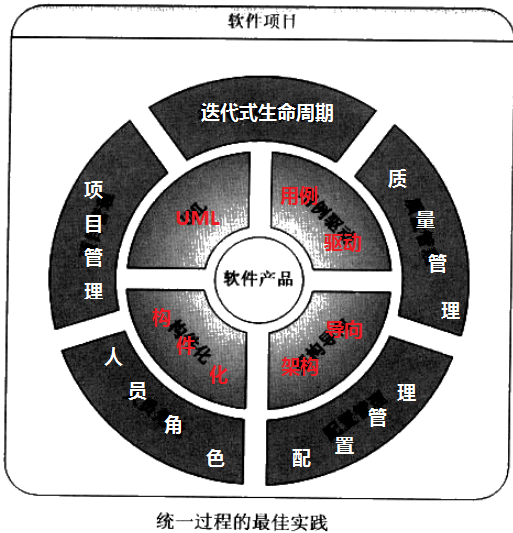
* UML是一种语言，用来描述软件生产过程中要产生的文档，RUP则是指导如何产生这些文档以及这些文档要讲述什么的方法。
* RUP和UML不是一体的，只是软件方法和建模语言的一个完美结合。要站在**软件过程**的角度，先了解软件项目如何做的，然后再从UML中寻找需要的工具，将软件过程中的要求记录下来。

## RUP与软件工程

* RUP（统一过程）是一种重量级方法，追求稳定，长期战略，适用于大中型软件产品开发，但实施过程成本高。
* 敏捷开发方法，如XP（极限编程）方法，追求在变化中用最快速的办法适应变更，用小的管理成本保障软件管理，适用于中小型软件产品开发，但稳定性差。

## RUP最佳实践

* 长久以来，人们期望软件开发能够像其他工业产品一样，可以单独生产标准零部件，然后按照要求来组装它们，用较少的投入完成最终的软件产品。现在随着面向对象的发展，基于架构的、构件式的软件开发模式己经成为软件开发的主流，这代表着软件从“手工业”向“工业”的转化取得了重大的进步。
* 对于软件产品来说，最佳实践来自两个方面：一方面是技术类的，如设计模式：另一方面的是过程类的，如需求方法、分析方法、设计方法等。
* RUP（统一过程）集成了很多过程类的最佳实践，这些最佳实践中包括用例驱动、架构导向、构件化等。另外，统一过程不仅仅集成了软件过程的技术方面的内容，还集成了大量的管理方面的内容，涉及到了软件工程的方方面面。



# 建模相关概念

## 建模（Modeling）

* 建模是指通过对客观事物建立一种抽象的方法用以表征事物并获得对事物本身的理解，同时把这种理解概念化，将这些逻辑概念组织起来，构成一种对所观察的对象的内部结构和工作原理的便于理解的表达。
  + 如何建?

依赖于方法论、认识论、世界观。面向过程和面向对象两种不同的软件方法将导致不同的建模结果。

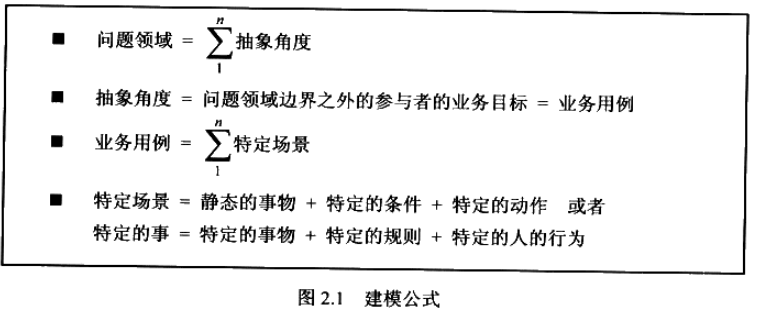
抽象角度的不同决定了建模方向的不同，抽象确定后，相应模型能够建立起来。

做需求的时候，首要目标不是要弄清楚业务是如何一步一步完成的，而是要弄清楚有多少业务的参与者？每个参与者的目标是什么？参与者的目标就是抽象角度，即用例。

* + 模是什么？

抽象角度（目标）确定后，接下要做的事情便是找出那些能够满足这一目标的事物。

采用过程化的描述方式，但是描述这个过程化的场景并不是最终目的，而是为了找出场景当中贡献于场景目标的那些事物，以及这些事物是如何贡献于这个场景的。

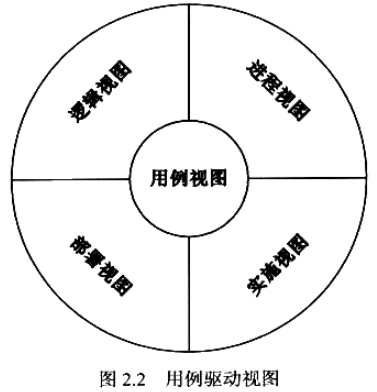


## 用例驱动

* 用例 == 抽象角度
* 要解决问题领域，就要归纳出所有必要的抽象角度，即用例
* 为归纳出的这些用例描述可能的特定场景
* 找出实现特定场景的事物、规则和行为

**实现用例是必须的工作**

* RUP中，一个用例就是一个分析单元、设计单元、开发单元、测试单元、部署单元等



* 逻辑视图（一个）

以图形方式说明关键的用例实现、子系统、包、类，它们包含在架构方面具有重要意义的行为，即建模公式中的那些人、事、物、规则是如何分类组织的。

* 进程视图（一个）

以图形方式说明了系统中进程的详细组织结构，其中包括类和子系统到进程和线程的映射，即建模公式中的那些人、事、物、规则是如何交互的，它们的关系如何，是常说的分析设计试图。

* 部署视图

以图形方式说明了处理活动在系统中各节点的分布，包括进程和线程的物理分布，即建模公式中的那些人、事、物、规则是如何部署在物理节点（主机、网络环境）上的。

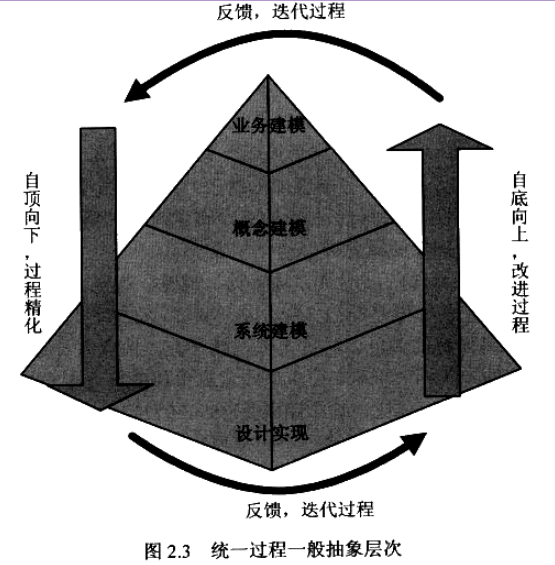
* 实施视图

其作用是获取为实施制定的构架决策，包括以下内容：

* 列举实施模型中的所有子系统
* 描述子系统如何组织为层次和分层结构的构件图
* 描述子系统间的导入依赖关系的图解
* 为个人、团队或分包商分配实施工作
* 估算要开发、修改或删除的代码数量
* 阐明大规模复用的理由
* 考虑发布策略

## 抽象层次

* 抽象层次越高，具体信息越少，但概括性越强；反之，具体信息越丰富，结果越确定，但概括性越弱。
* 抽象的两种方法：
* 自顶向下（由简到繁）：从头开始认识一个事物
* 自底向上（由繁到简）：在实践中改进和提高认识
* 软件开发过程中，主体上应当采用自顶向下的方法，用少量的概念覆盖系统需求，再逐步降低抽象层次，直到代码编写；同时应当辅以自底向上的方法，通过总结在较低抽象层次的实践经验来改进较高层次的概念以提升软件质量。

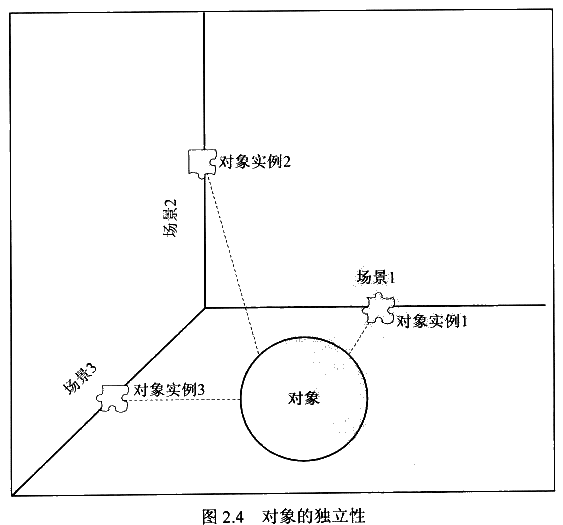


## 视图

* 用于组织UML元素，表达出模型某一方面的含义。
* 属性是事物的视图，每个视图都向观察者展示了目标事物的一个方面；只有将必要的属性都用视图展示出来，观察者才能真正理解这个事物。
* 视角是人们观察事物的角度，不同的观察者（客户、系统分析员、架构师、设计师、开发人员、测试人员、项目经理等）观察的角度不同，但观察者感兴趣的只是他那部分信息。因此在展示信息时选择适当的视角并展示给适当的观察者是十分重要的。
* 建立模型的目的是向相关的人展示将要生产的软件产品，不同的视图去展示软件这些不同的方面——静态的、动态的、结构性的、逻辑性的等——才能够说建立了一个完整的模型。
* UML中定义了用例图、对象图、类图、包图、活动图等不同的视图，从不同的方面描述了一个软件的结构和组成，所有这些视图的集合表达了一个软件的完整含义。
* **建模的主要工作就是为软件绘制那些表达软件含义的视图来完整的表达软件的含义。**

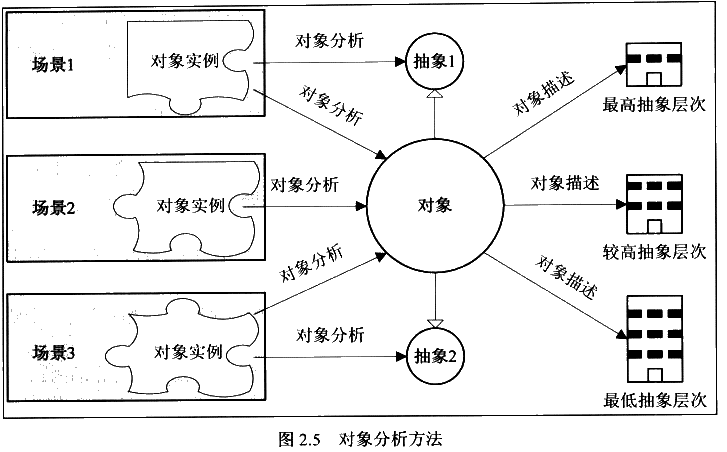
## 面向对象分析（OOA）方法

* 一切都是对象
* 对象都是独立的、离散的
  + 对象不是因为场景而存在的，场景中的对象只是对象“映射”到该场景中的一个侧面，称为**对象实例**



* + 面向对象分析方法：从不同的场景中分析对象实例，获得对象的多个侧面，再通过归纳整理这些对象的侧面抽象出对象的一般特性，这个一般特性是与场景无关的、独立于场景的，是对象的可抽象能力和可扩展能力。
  + 采用面向对象分析方法时，在需求、分析、设计过程中，所得到的任何一个有名字的东西，不论是用例、类、包、组件等都是独立于那个场景的，不要将对象局限在那个场景中。
* 对象都具有原子性
  + 无论什么时候、无论对象规模有多大，在分析过程中都应该将对象视为一个不可分割的原子
  + 在分析过程中，对象总有一个边界，永远也不应该打破边界去窥探对象的内部。如果将对象比喻为鸡蛋，蛋壳就是对象的边界，在分析对象的过程中，我们对它的所有理解都是来自蛋壳。
  + **面向接口编程**：将分析过程中得到的所有对于对象的认识附加在对象边界，在实现这个对象之前不理会其内部的细节。
* 对象都是可抽象的
  + 对象有着很多个不同的方面。一般来说，对象参与一个场景时会展现出某一个方面。总可以将对象的某一个方面抽象出来，让其作为对象的一个代表来参与场景交互。通常这种抽象会以接口来命名。在分析过程中，得到的任何一个对象都有特定的方面可作为抽象。因为对象总是从场景分析中得到的，它在场景中肯定展现了一个方面。
  + 对象所具有的方面，或者说对象所参与的场景越多，对象越有抽象价值，反之则越没有抽象价值。因此在分析过程中，应当关注于那些参与了很多场景的对象，它们往往是分析设计中的重点以及成败关键。
* 对象都有层次性
  + 对象是有着抽象层次的。层次越高，其描述越粗略但适应能力越广；层次越低则描述越精确但适应能力越下降。在分析过程中，应当根据问题领域的复杂程度设定多个抽象层次，在每个层次上使用适合的抽象程度的对象描述。这将有助于显著地减少分析的难度和工作量。
  + 不论是在需求、分析还是设计过程中，都应当具备抽象层次的观点。从需求到设计的过程己经是几个不同的抽象层次，笔者要说的是，在其中的一个阶段，例如需求阶段，仍然可以再多分几个抽象层次来说明。具体分多少抽象层次应视问题领域的复杂程度而定。
* 面向对象分析方法总结

独立性、原子性、抽象性、层次性是面向对象分析时应当遵循的一些原则和方法。



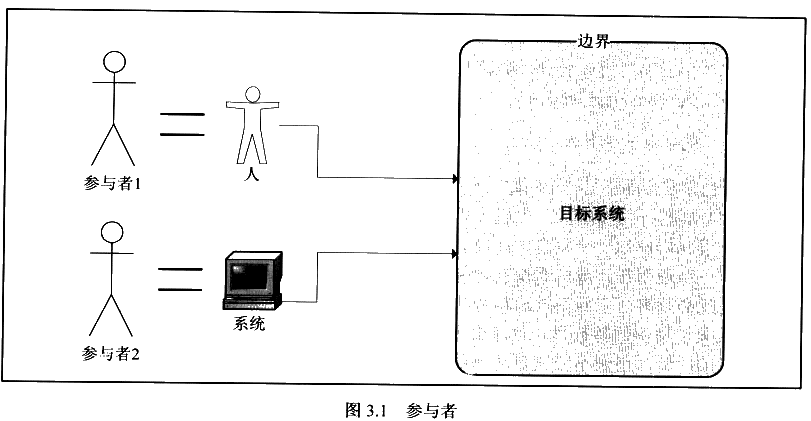
# UML核心元素（词汇）

## stereotype（版型）

* 版型是UML的一种扩展手段
  + UML已经定义的版型：用例有业务用例、业务用例实现等版型；类有接口、边界类、实体类、控制类等版型；参与者是一种特殊类的版型；包有子系统、组织结构、模块等……
  + UML自定义版型：如定义文档、开发小组等能够表达某种含义的版型来辅助建模。

## actor（参与者）

* UML建模以人为本，从寻找抽象角度开始
* 基本概念
  + 参与者是在系统之外与系统交互的某人或某事物



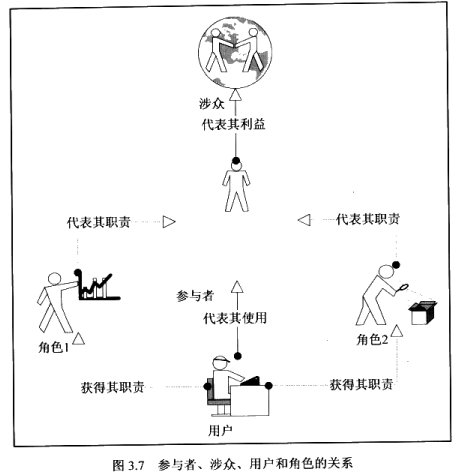
* + 参与者位于系统边界之外（“谁”作为主角，是参与者）

场景：小王到银行去开户，向大厅经理询问了办理手续，填写了表单，交给柜台职员，拿到了银行存折。

* + - **谁**对系统有着明确的目标和要求并且主动发出动作？
    - 系统是为**谁**服务？

场景分析：第一个问题的答案是小王有着明确的目标：开户，并且主动发出了开户请求的动作；第二个问题的答案是系统运作的结果是给小王提供了开户的服务。小王是参与者。

* + 参与者可以非人
* 场景：每天自动统计网页访问量，生成统计报表，并发送至管理员信箱。
* 场景分析：计算机在没有“外力”的情况下，将保持等待或循环任务状态，因此必须有“东西”发出指令或动作，计算机才会做出相应的反应。任何需求都必须至少有一个启动者，如果找不到启动者，那么可以肯定的说这不是一个功能性需求（如：客户提出建立的系统界面要很友好，在每个页面上都要有操作提示）。场景中描述的需求的启动者是一个计时器，因此计时器是这个需求的参与者。
* 发现参与者
  + 参与者一定是直接并且主动地向系统发出动作并获得反馈的，否则就不是参与者。
* 业务主角
  + 是参与者的一个版型，特别用于定义业务的参与者，在需求阶段使用。
  + 业务主角的特殊性在于，它针对的是业务人员而非计算机用户，在查找业务主角时必须抛开计算机，没有计算机系统这些业务人员也客观存在。
  + **要建设一个符合客户需要的计算机系统，首要条件是完全彻底地搞清楚客户的业务，而不是预先假设已经有了一个计算机系统，再让客户来假想需要计算机系统帮他们做什么。**
  + 建立业务模型、查找业务用例都必须使用业务主角，而不是普通参与者。
* 业务工人
  + 参与者是“主角”，业务工人是“配角”；参与者在边界外，主动发出动作，业务工人在边界内，被动参与业务，完成参与者的业务目标
  + 业务工人不需要建立业务模型，在在参与者的业务模型中出现，是业务模型中的非常重要的一部分，经常出现的地方是领域模型和用例场景。
* 参与者与涉众的关系
  + 涉众，也称为干系人，是与要建设系统有利益关系的一切人和事，涉众的利益要求会影响到系统的建设。
  + 参与者是涉众代表，通过对系统提出的要求来获得他所代表的涉众的利益。
* 参与者与用户的关系
  + 用户是指系统的使用者，或系统的操作员。
  + 用户是参与者的代表（实例或代理）。
  + 并非所有的参与者都是用户，但一个用户可以代理多个参与者。
* 参与者与角色的关系
  + 角色是参与者的职责，是从众多参与者的职责中抽象出相同的那一部分，将其命名而形成一个角色，一个角色代表了系统中的一类职责。
  + 角色一般适合用在概念阶段的模型里，以表达业务的逻辑理解。
  + 使用角色可以灵活化系统。
  + 一个用户可以代理多个参与者，因此一个用户可以拥有多个职责，也是就可以被指定多个角色。
* 参与者的核心地位
  + 参与者是涉众的代表，它代表涉众对系统的利益要求，并向系统提出建设要求；参与者通过代理给其他用户或将自身实例化成用户来使用系统；参与者的职责可以用角色来归纳，用户被指定扮演哪个或哪些角色因此来获得参与者的职责。



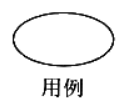
* + 参与者的核心地位还体现在，系统是以参与者的观点来决定的，参与者对系统的要求，对系统的表述完全决定了系统的功能性。
* 检查点
  + 检查发现的参与者是否正确
    - 是否您己找到所有的参与者？也就是说，是否您己经对系统环境中的所有角色都进行了说明和建模？虽然您应该检查这一点，但是要到您找到并说明了所有用例后才能将其确定。
    - 每个参与者是否至少涉及到一个用例？删除未在用例说明中提及的所有参与者，或与用例无通信关联关系的所有参与者。
    - 您能否列出至少两名可以作为特定参与者的人员？如果不能，请检查参与者所建模的角色是否为另一角色的一部分。如果是这样，您应该将该参与者与另一参与者合并。
    - 是否有参与者担任与系统相关的相似角色？如果有，您应该将他们合并到一个主角中。通信关联关系和用例说明表明参与者和系统是如何相互关联关系的。
    - 是否有两个参与者担任与用例相关的同一角色？如果有，您应该利用参与者泛化关系来为他们的共享行为建立模型。
    - 特定的参与者是否将以几种（完全不同的）方式使用系统？或者，他使用用例是否出于几个（完全不同的）目的？如果是这样，您也许应该有多个参与者。
    - 参与者是否有直观名称和描述性名称？用户和客户是否都能理解这些名称？参与者的名称务必要与其角色相符。否则，应对其进行史改。

## 用例

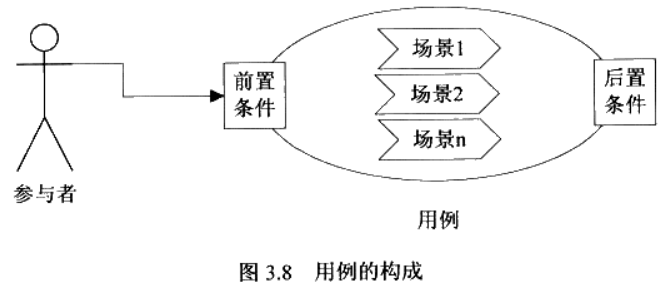
**用例通过“外力“将其它的独立的对象关联起来。**

### 基本概念

* + 官方定义：用例定义了一组用例实例，其中每个实例都是系统所执行的一系列操作，这些操作生成特定主角可以观测的值。



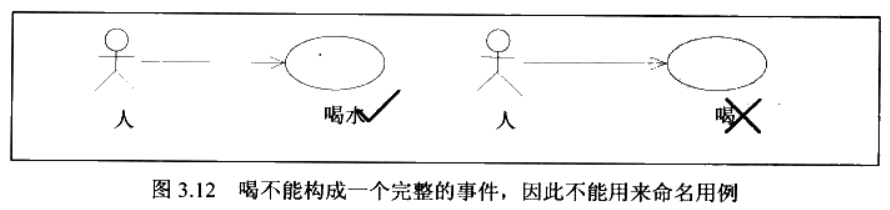
* + 一个用例就是与参与者交互的，并且给参与者提供可观测的有意义的结果的一系列活动的集合，用例的来源是参与者对系统的期望。
  + 对于一件事情，要完成这件事情，需要做一系列的活动；而做一件事情可以有很多不同的办法和步骤，也可能会遇到各种各样的意外情况，因此这件事情是由很多不同情况的集合构成的，在UML中称之为用例场景。
  + 一个场景就是一个用例的实例。
  + 用例的启动是需要条件的，启动用例的前提称之为**前置条件**，用例执行完后获得的结果称为**后置条件**。
  + 一个完整的用例由参与者、前置条件、场景、后置条件构成。



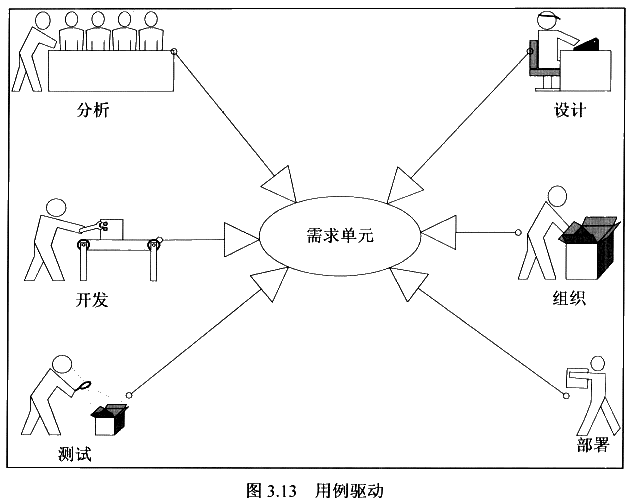
* + 一个系统的功能性是由一些对系统有愿望的参与者要做的一些事情构成的，事情完成后就达成了参与者的一个愿望，当全部参与者的所有愿望都能够通过用例来达到，那么这个系统就被确定下来了。
  + 捕捉功能性需求，是用例的作用。

### 用例的特性

* + 用例是相对独立的
    - 不需要与其他用例进行交互就能独自完成参与者的目的，“功能”上是完备的。
    - 用例本质体现了系统参与者的愿望，不能完整达到参与者愿望的不能称为用例。
  + 用例的执行结果对参与者来说是可观测的和有意义的
    - 进程监控功能模块是系统的一个必需组成部分，但是它在需求阶段却不应该作为用例出现，因为对参与者来说是不可观测的
    - 登陆系统是一个有效的有意义的用例，但输入密码却不是一个有意义的用例，因为单纯的输入密码并不是参与者的意图，参与者通过登陆系统可以获取身份认证和授权。
  + 这件事必须由一个参与者发起，不存在没有参与者的用例，用例不应该自动启动，也不应该主动启动另一个用例。
    - 用例存在的原因是要满足参与者的愿望
  + 用例必然是以动宾短语形式出现的
    - 用例必须有一个动作和动作的受体



* + 一个用例就是一个需求单元、分析单元、设计单元、开发单元、测试单元，甚至部署单元。
  + 用例驱动：一旦决定了用例，软件开发工作的其它活动都以这个用例为基础，围绕着它进行

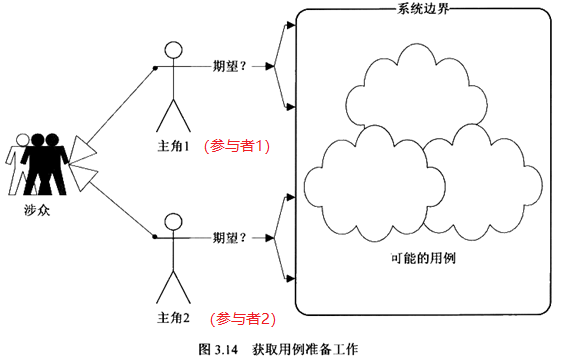


### 用例的粒度

* + 在ATM机取钱的场景中，取钱、读卡、验证账号、打印回执单等都是可能的用例，取钱包含了后续的其它用例，取钱粒度更大一些，其他用例的粒度则要小一些。
  + 在项目过程中根据阶段不同，使用不同的粒度
    - 业务建模：用例的粒度以每个用例能够说明一件完整的事情为宜。即一个用例可以描述一项完整的业务流程。这将有助于明确需求范围。
    - 概念建模（用例分析阶段）：用例的粒度以每个用例能描述一个完整的事件流为宜。一个用例描述一项完整业务中的一个步骤。这个阶段需要采用一些面向对象的方法，归纳和抽象出业务用例中的关键概念模型并为之建模。
    - 系统建模：用例视角是针对计算机的，因此用例的粒度以一个用例能够描述操作者与计算机的一次完整交互为宜。
  + 用例粒度的划分依据最标准的方法是以该用例是否完成了参与者的某个完整目的为依据的。
  + 用例粒度太小，造成的后果是需求因为过于细碎和太多而无法控制；用例粒度太大，造成的后果是需求因为过于模糊而容易忽略细节。一般来说，一个系统的业务用例定义在多于10个，少于50个之间，否则就应该考虑一下粒度选择是否合适了。
  + 不论粒度如何选择，必须把握的原则是在同一个需求阶段，所有用例的粒度应该是同一个量级的。
  + 粒度选择的问题本质上还是因为边界认定不同而产生的。

### 用例的获得

* + 发现用例的前提条件是发现参与者，而确定参与者的同时就确定了系统边界。

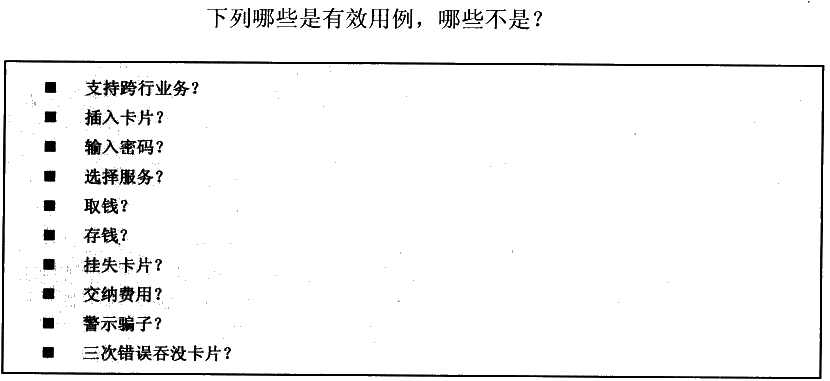


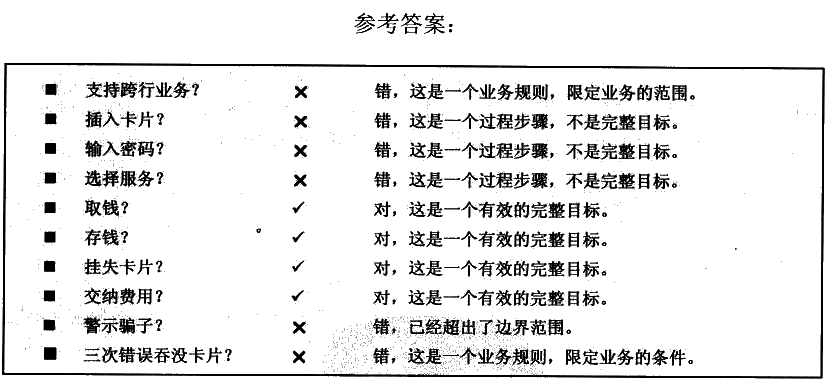
* + - 主角（参与者）位于系统边界之外的
    - 主角（参与者）对系统有着明确的期望和明确的回报要求
    - 主角（参与者）的期望和回报要求在系统边界之内
  + 接下来，可以开始对主角，即业务代表进行**访谈**。访谈时请不要试图让业务代表为你描述整个业务流程，也不要涉及表单填写一类的业务细节，甚至你可以不关心业务规则，更不要试图让业务代表理解将来的计算机系统会如何工作。你只需要让业务代表从他自己的本职工作出发来谈谈他的期望，并时时提醒和引导那些喜欢一讲什么事情就深入到细节当中去的客户。可以通过以下问题引导业务代表，这些问题对用例获取来说己经足够了。
    - 您对系统有什么期望？
    - 您打算在这个系统中做些什么事？
    - 您做这些事的目的是什么？
    - 您做完这件事希望有一个什么样的结果？
  + 对访谈结果的分析应当确保：

*简单地用纸和笔记录下业务代表的访谈结果，从结果中找出用例。不要指望客户和你一样对什么是用例了如指掌，也不要期望客户能有条有理分层分次地把他对系统的期望表达出来。从客户也许语无伦次，也许杂乱无章的谈话中找出主角期望的真实和有效目标是你的工作。你应当清楚，主角想做和要做的事情不一定是他真实的目标，也许只是他做事情的一个步骤。比如客户或许会说我首先做......，然后做......，最后做......，你需要从冗长的谈话中为客户总结出他的真实目标来；另外，主角对系统的期望也不一定是一个有效的事件，也许真的只是一个愿望，比如客户会说我期望界面能漂亮一此，你需要告诉客户他的期望将是一件可以做的事情，而不仅仅是一个主观愿望。不同主角对同一目标可能会有不同的表达，例如客户甲说我希望能把我这些文件保存下来以供将来查询，而客户乙说我要能查看我之前工作过的所有工作记录。或许甲和乙口中的文件和工作记录就是同一件事情，你应当去伪存真，求同存异，而不是简单地就分为两个用例；还有，不同主角的目标可能会相互重叠，呈现出一种交集的状态。你应当小心求证，是否这些主角所谈的都只是某个完整目标的一个部分？如果这样，应当合并成一个用例，并假定这两个主角在这个用例中只是担任业务工人的角色而不是真正的主角。或者这些主角所谈的是有交叉的部分，但的确是两个不同的目标。如果这样，应当就是两个用例。至于交集的部分，需要在概念模型中去提取公共的业务单元。*

* + - 一个明确的有效的目标才是一个用例的来源
    - 一个真实的目标应当完备地表达主角的期望
    - 一个有效的目标应当在系统边界内，由主角发动，并具有明确的后果。
  + 重新访谈应考虑调整以下策略：
    - 调整系统边界和主角
    - 扩大或缩小系统边界
    - 变更主角
  + ATM实例

|  |
| --- |
| *客户代表说：我希望这台ATM能支持跨行业务，我插入卡片输入密码后，可以让我选择是取钱还是存钱；为了方便，可以设置一些默认的存取金额按钮；我可以修改密码，也可以挂失；还有我希望可以交纳电话费、水费、电费等费用；为了安全起见，ATM上应当由警示小心骗子的提示条，还有摄像头；如果输入三次密码错误，卡片应当被自动吞没。* |





### 用例和功能的误区

* + **用例并不是功能**，虽然用例的定义中用例是用来捕获功能性需求的，但是有一个前提条件，即这个功能性需求是从参与者的角度出发的。功能实际描述得是输入 -> 计算 -> 输出。
  + 区分用例和功能的出发点
    - 这个事物是什么？ 【结构性观点，事物的客观存在】
    - 这个事物能做什么？ 【功能性观点，事物可利用的价值】
    - 人们能够用这个事物做什么？ 【使用者观点，事物对使用者的意义】
  + 在软件项目前，软件还是一种不存在得事物，不能从结构观点去描述它，也不能从功能观点去描述它，最好的方法就是从使用者的观点去描述它。
    - 不能从功能观点去描述它的解释：若从功能观点出发，采用功能分解方式来获取需求的方式，因为**缺少了使用者的上下文**，功能很可能就变成了对使用者无用的，或者使用者不知道怎么用的东西。
  + 功能是脱离使用者的愿望而存在的，描述的是事物固有的性质；用例是描述使用者愿望的，描述的是使用者对系统的使用要求。
  + 功能是孤立的，描述的是一个个点：输入 -> 计算 -> 输出，必须额外加上一个顺序的过程把点串起来才能完成一个系统性的工作；用例是系统性的，非常明确地去达成一个特定的目标，量身定制用户的需求。
  + 用例可以解释为一系列完成一个特定目标的“功能”的组合，针对不同的应用场景，这些“功能”体现不同的组合方式。并且，不是先有了这些“功能”才来组合成某个场景，而是先有了场景，才分解出“功能”。这里的“功能”之所以打引号，是因为在UML里是没有功能这个词的，实际上从场景分解出来的是对象，这些对象通过消息相互交流而完成场景。

### 目标和步骤的误区

* + 一个用例是参与者对目标系统的一个愿望，一个完整事件。为了完成这个事件需要经由很多步骤，但这些步骤不能够完整的反映参与者的目标，不能够作为用例。
  + 在做需求时用例要体现参与者的完整目标，如果错误地使用步骤作为需求用例，将无法准确地描述参与者如何使用系统，也就无法准确地捕获需求。
  + 步骤也可以作为用例，在概念建模阶段，由于需求已经捕获，在对需求进行分析时，实际上已经进入了用例的内部，意味着边界已经改变，导致参与者也在改变。通常参与者已经变成了原来的业务工人，完整目的也变为更改后参与者的目的，但现在参与者的所有活动都处于该用例的上下文环境之内。

### 用例粒度的误区

* + 分不清目标和步骤，导致用例粒度过于细小，使得系统分析没有抽象的余地。
  + 边界不清，参与者混乱，导致在同一个需求阶段中的用例粒度大小不一，抽象层次产生错误导致程序结构混乱。

### 业务用例（business use case）

* + 业务用例是用例版型中的一种，专门用于需求阶段的业务建模。具有普通用例的所有特征。



* + 业务用例用于描述客户现有业务的，面对的问题领域就是没有将来的计算机系统参与的、目前客观存在的业务领域。
  + 业务模型的参与者是业务主角，看到的边界是业务边界，而非系统边界。

### 业务用例实现（business use case realization）

* + 也称为业务用例实例，是用例版型中的一种，专门用于需求阶段的业务建模。



* + 业务用例实现是业务用例的一种实现方式，一个业务用例有多种实现方式，他们的关系类比编程上的接口和实现类的关系。业务用例实现的意义在于表达了同一项业务的不同实现方式。

### 概念用例

* + 实际应用中很少用到，UML中也没有为其预定版型，下面为自定义的概念用例版型。概念用例用于概念模型。



* + 作为概念模型中的核心元素，概念用例用来获取业务用例中的核心业务逻辑，这些核心的业务逻辑揭示了业务模式，成为业务架构的重要指导。
  + 概念用例是从业务用例到系统用例过渡时非常重要的指导。尤其对于复杂业务来说，缺少了它，系统用例的产生就会显得突兀和生硬。

### 系统用例

* + UML中没有为系统用例定义版型，实际其为经常说的用例，可以直接称为用例。



* + 系统用例是用来定义系统范围、获取功能性需求的。是软件系统开发的全部范围，是得到的最终需求。

### [系统]用例实现

* + [系统]用例实现是[系统]用例的一种实现方式，用例有多种用例实现方式。



* + 现实情况中绝大部分项目都是在做完用例模型后，直接开始进入数据库表设计、类设计等。但很多时候不清楚这些数据库表和类是依据什么出来的，仅凭经验。其实**用例实现正是连接起用例模型和系统实现之间的桥梁**。

## 边界

* 基本概念
  + 边界在UML图符里的定义只是一个简单的矩形框，矩形框的四个边决定了边界的内外。



* + 边界本质上是面向对象方法的一个很重要的概念，与封装的概念师出同源。面向对象里，任何一个对象都有一个边界，外界只能通过这个边界来认识对象，与对象打交道，而对象的内部则是一个禁区。
  + 需求出来之前，我们必须先设想一个边界，这个边界的大小是不确定的，随着需求的明确，边界也逐步变得明朗。
  + **参与者、用例、边界的矛盾**
    - 需求的确定依靠参与者和用例；
    - 参与者和用例的明确的前提条件是边界的确定，但此时边界并没有被确定。

**需求就是在不断地调整这个矛盾的过程中逐步明确进而更加确定边界的。这个调整过程不可避免地会导致参与者和用例的变化。所以需求过程是一个动态的过程，不可能一蹴而就，也因此统一过程需要迭代，而不能采用瀑布方法。**

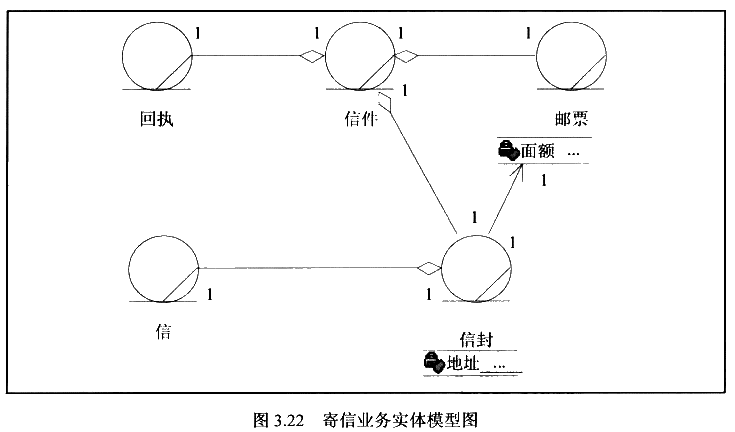
* 边界决定视界
  + 收集需求和开发软件的过程，像是盲人摸象，为了更接近真相，我们能够做的就是不断变换边界，改变视界，从更多的侧面去描述同一个信息，以求最大程度地符合真实的需求。
* 边界决定抽象层次
  + 自顶向下：通过逐步缩小边界进而影响到可以观察到的事物，也决定了抽象层次，使得我们的分析粒度可以有条不紊地逐步细化。
  + 自底向上：设置边界时由小到大
  + 不论哪种方式，边界总能帮助很好地把握当前的抽象层次，忽略掉那些边界外的杂音，专心地把当前边界内的问题搞清楚。
* 灵活使用边界
  + 边界应用于软件设计：
    - 软件设计面临着很大的信息量，既要实现需求，又要保证性能，要具有扩展能力，还要友好易用。
    - 将实现需求的任务交给分析模型，在这个边界内只考虑需求实现；将扩展能力交给框架设计，在这个边界内专心设计灵活的框架；在框架的约束下把分析模型转化成设计模型。

## 业务实体

* 基础概念
  + 业务实体是类（class）的一种版型，特别用于在业务建模阶段建立领域模型。

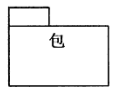


* + 业务实体是业务模型中非常重要的一个因素，它为问题领域中的关键概念建立概念化的理解，是人们认识问题领域的重要手段。
  + 参与者和用例描述了在问题领域中达到什么样的目标；业务实体就描述了使用什么来达到业务目标以及通过什么来记录这个业务目标。
  + 业务实体是问题领域这幢楼房的砖瓦和石头。
  + 官方定义：业务实体代表业务角色执行业务用例时所处理或使用的“事物”。
    - 业务实体是来自现实世界的，在建模的问题领域里一定能够找到与它对应的事物，并且这个事物是参与者在完成其业务目标的过程中使用到的或创建出来的。
    - 业务实体一定是在分析业务流程（用例场景）的过程当中发现的。意味着业务实体必须至少被一个业务用力场景使用或创建，对业务用例场景没有贡献的事物，即使它是客观存在的，也不应当为它建模。
    - 业务实体作为类的一个版型，具有对象的所有性质，包括属性和方法，同时也具有对象的独立性，即业务那实体只应当包含它本身固有的特性，而不能包含外界是如何使用它的信息。
* 业务实体的属性
  + 属性是用来保存业务实体特征的一个记录，业务实体的属性集合决定了它的**唯一性**。
  + 通常情况下业务实体的属性可以很容易地从它所对应的现实事物中找到，但是一个事物通常有非常多的属性，在建模的时候，不需要全部列出，只需要把与这个场景直接关联的那些属性列出。
  + 很多时候属性是可分的，是一个复杂的业务对象。一般来说，如果只有一个对象可以直接使用这个属性，或者只能通过对象才能访问到这个属性，它就应当作为一个属性存在；否则就应当把它单独建模成一个业务实体。这也是面向对象方法中的封装原则的应用。
* 业务实体的方法
  + 方法是访问一个业务实体的句柄，它规定了外部可以怎么来使用它。
  + 方法就是外部能够使用这个业务实体的全部信息。
  + 一个业务实体可能有很多使用方法，不需要将全部可能得方法都定义出来，只关心那些与这个场景有直接关系得那些方法。
  + 与业务实体得属性一样，业务实体得方法也同样是面向对象方法中得抽象视角的体现。
* 获取业务实体
  + 首先要建立业务用例场景；
  + 然后从业务用例场景中逐个分析动词后面的名词，它们就是业务实体的备选对象；根据对象对业务目标是否有贡献这个筛选条件从备选列表中挑选出符合的对象，组成初始的业务实体；
  + 最后**分析这些业务实体之间的关系，并决定哪些应当单独建模，哪些应当作为属性**。
  + 为业务实体之间的关系建模，为他们之间的交互建模，就得到了问题领域的领域模型，如下为寄信这个业务场景的领域模型中的静态视图

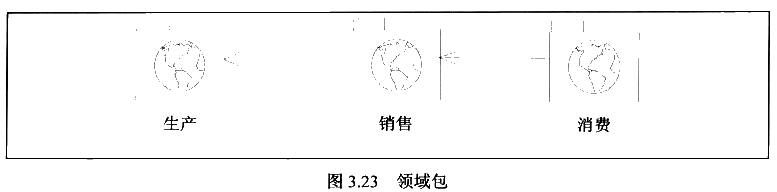


## 包

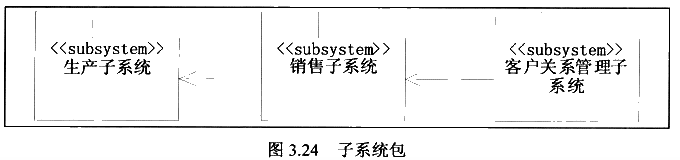
* 基础概念
  + 包是一种容器，如果文件夹一样，它将某些信息分类，形成逻辑单元。使用包的目的是为了整合复杂的信息，某些语义上相关或者某方面具有共同点的信息都可以分包。



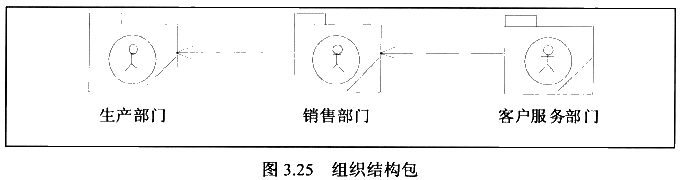
* + 包是UML中常用的元素，它最主要的作用就是容纳并为其它元素分类，它可以容纳任何UML元素，如用例、业务实体、类图等，也包括子包。
  + UML认为好的分包具有高内聚、低耦合的性质
    - 如果将所有元素分为三个包A、B、C，那么被分入同一个包中的那些元素应当是相互联系紧密，甚至不可分割的。同时这些元素又具有某些相同的性质，使得包可以抽象出一些接口来代表包内事物与包外事物的交互，以避免包外的事物频繁地直接访问包内元素。这时称A、B、C三个包具有**高内聚**的性质。
    - 包的最理想的情况是修改A、B、C三个包中任意一个包的元素，其他的任何一个包中的内容都不受到影响。这时称A、B、C三个包之间无依赖关系或**松耦合**关系，它们之间可以保持消息通信。
    - 如果实际情况难以做到完全解除依赖关系，那么至少应当保证包之间的依赖关系不会被传递。如B依赖于A，C依赖于B，当A修改导致B要做出修改时，C不会受到影响。如果做不到这一点，当一个包发生变动时将会引起大范围的连锁反应。
    - 包之间的依赖关系应当是单向的，应当尽量避免双向依赖和循环依赖。
* UML中常用的包的版型：
  + 领域包（Domain Package）：领域包用于分类业务领域内的业务单元，每个包代表业务的一个领域，领域包视图可用于展示这些业务领域的高层次关系。



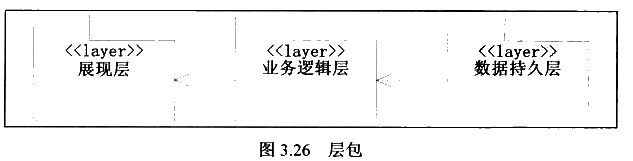
* + 子系统（Subsystem）：用于分类系统内的逻辑对象并形成子系统，子系统包视图可用于展示系统的高层次逻辑结构关系。



* + 组织结构（Organization unit）：用于分类业务领域中的组织结构，它可以直接用来表述企业的组织结构



* + 层（layer）：用于分类软件中的层次，层可以用于展示软件的架构信息



## 分析类

* 基础概念
  + 官方定义：分析类用于获取系统中主要的“职责簇”。它们代表系统的原型类，是系统必须处理的主要抽象概念的“第一个关口”。如果期望获得系统的“高级”概念性简述，则可对分析类本身进行维护。分析类还可产生系统设计的主要抽象——系统的设计类和子系统。
    - 分析类代表系统中主要的“职责簇”，意味着分析类是从功能性需求向计算机实现转化过程中的“第一个关口”；
    - 分析类可以产生系统的设计类和子系统，意味着计算机实现是可以通过某种途径“产生”出来的，而不是凭经验出来的。
  + 分析类是从业务需求向系统设计转化过程中最为主要的元素，它们在高层次抽象出系统实现业务需求的原型，业务需求通过分析类逻辑化，被计算机所理解。分析类是需求实现的第一步，虽然在统一过程中分析类被定义为一种过渡类型，意味着它不是一个强制过程。但是笔者在自己的工作经验中认识到，分析类对于系统分析和设计的重要性远远超出过渡类型所能发挥的作用。
* 边界类（boundary）
  + 边界类是一种用于对系统外部环境与其内部运作之间的交互进行建模的类。这种交互包括转换事件，并记录系统表示方式（例如接口）中的变更。在从需求向实现的转换过程中，任何两个有交互的关键对象之间都应当考虑建立边界类。



* + 对现实世界来说，边界类的实例可以是窗口、通信协议、打印机接口、传感器、终端等，在计算机世界里，边界类也可以是一个消息中间件、一个驱动程序、一组对象接口甚至任意的一个类。总之，不论是现实世界还是计算机世界里，当我们打算对A对象和B对象之间的交互进行建模时，边界类都可以充当这一载体。下面来看一些边界类的常用场景。
    - 参与者与用例之间应当建立边界类

用例可以提供给参与者完成业务目标的操作只能通过边界类暴露出来。例如，参与者通过一组网页、一组Windows窗口、一个字符终端或者是一只鼠标来使用用例的功能，上述的东西都可以称为用例的边界类。

* + - 用例与用例之间如果有交互，应当为其建立边界类

一个用例如果要访问另一个用例，直接访问用例内部对象是不好的结构，这样将导致紧耦合的发生。而边界类可以隔离这种直接访问，其作用相当于一个门面模式。在最终实现时，用例之间的边界类可以演化为一组API、一组JMS消息或是一组代理类。

* + - 如果用例与系统边界之外的非人对象有交互，例如第三方系统，应当为其建立边界类

这通常是因为异构系统、异构数据、访问权限、安个通道等原因。在具体实现时，边界类可以演化为中介和通信协议，中介的例子如网关、通信中间件、代理服务器、安全认证服务器、WebService、SOA组件等；通信协议的例了如HTTP、FTP、SSL、RMI,SOAP等。

* + - 在相关联的业务对象有明显的独立性要求，即它们可能在各自的领域内发展和变化，但又希望互不影响时，也应当为它们建立边界类

例如生产计划和客户服务计划都来源于销售记录和客户关系记录，但是当销售记录和客户关系记录发生变化时，生产计划和客户服务计划对此产生的回应是不一样的。这时在销售记录和客户关系记录与生产计划及客户服务计划之间加入边界类或许就是一个好主意。在实现时，边界类可以转化为一组接口来为这些对象解耦。

* + - 最后，从架构角度上来说，边界类主要位于展现层。边界类的获取对架构设计中的展现层有着重要的指导意义。
  + 一个好的边界类应该具备以下特点
    - 边界类应该有助于提高系统的可用性。
    - 边界类应该尽可能地保持在较高的层次（如概念层次）上。
    - 边界类应该合理封装介于系统与主角之间的交互。
    - 如果主角改变他们为系统提供输入的方式，边界类就应该是唯一需要改变的对象，如果系统改变为主角提供输出的方式，边界类就应该是唯一需要改变的对象。
    - 边界类必须“知道”其他对象类型（例如控制对象和实体对象）的需求，以便它们能够得以实施，并相对于“系统内部元素”保持其可用性和有效性。
* 控制类（control）
  + 控制类用于对一个或几个用例所特有的控制行为进行建模。控制对象（控制类的实例）通常控制其它的对象，因此它们的行为具有协调性质。控制类将用例的特有行为进行封装。



* + 控制类来源于对用例场景中行为的定义，换言之，控制类来源于对用例场景当中动词的分析和定义，包括限制动词的描述
  + 在提取控制类的时候，要认真考察用例场景中的行为，如果这些行为在执行步骤、执行要求或者执行结果上具有类似的特征，应当考虑进行适当的抽象，例如合并或者抽取超类。同时，也要考察这些行为是否对要建设的系统产生影响而进行一些取舍。
  + 理论上：在UML定义中，认为控制类主要起协调对象的作用，例如从边界类通过控制类访问实体类，或者实体类通过控制类访问另一个实体类。同时也认为不必强制使用控制类，例如边界类也可以直接访问实体类。
  + 实践中：应当强制使用控制类，应为这是一种好的程序结构。
    - 未使用控制类，如C/S结构应用模式、网页+数据库应用模式，通常都是两层架构应用。这并非好的应用模式，业务逻辑代码要么与显示混在一起（网页里充满了处理代码），要么与数据逻辑混在一起（大量的存储过程）。
    - 好的应用模式，在边界类和边界类、边界类和实体类、实体类和实体类之间都默认加入控制类，将相关的处理逻辑放到控制类里去，哪怕控制类只有一个操作。
  + 在设计阶段，控制类可以被设计为SessionBean、COM+、Servlet、Java类、C++类等设计类。
  + 从架构角度上来说，控制类只要位于业务逻辑层。控制类的获取对架构设计中的业务逻辑层有着重要的指导意义。
* 实体类（entity）
  + 实体类用于对必须存储的信息和相关行为建模的类。实体对象（实体类的实例）用于保存和更新一些现象的有关信息，例如事件、人员或者一些现实生活中的对象。实体类通常都是永久性的，它们所具有的属性和关系是长期需要的，有时甚至在系统的整个生存期都需要。



* + 实体类源于业务模型中的业务实体。很多时候可以直接把业务实体转化为实体类，但是出于系统结构优化的需要，一些业务实体可以在后续的过程中被分拆、合并。
  + 在设计阶段，实体类可以被设计为Entity Bean、POJO、SDO、XML Bean等设计类甚至是一条SQL语句。
  + 从架构的角度上来说，实体类主要位于数据持久层，实体类的获取对架构设计中的数据持久层有着重要的指导意义。
* 分析类的三高

分析类是从业务需求向系统设计转化过程中最为主要的元素，它们在高层次抽象出系统实现业务需求的原型，业务需求通过分析类被逻辑化，成为可以被计算机理解的语义。分析类的抽象层次有三高的特点，正是因为这些特点，使得分析类成为比设计类“更好用”的元素：

* + 高于设计实现

在为需求考虑系统实现的时候，可以不理会复杂的设计要求，比如设计模式的应用、框架规范的要求等，而专心地为从需求到实现搭建一座桥梁。以实体类为例，一个实体类可以被设计成Entity Bean，也可以被设计为POJO，不论是哪一种设计实现，都要遵循相关的规范，实现特定的接口等。这些复杂的要求在为需求考虑系统实现的时候就成为一些杂音，要处理的信息越多，越容易分散注意力。而使用分析实体类的话，就不需要顾忌实现问题，专心解决需求问题。

* + 高于语言实现

在为需求考虑系统实现的时候，可以不理会采用哪一种语言来编写代码，也就可以排除特定语言的语法、程序结构、编程风格和语言限制等杂音，而能专注在需求实现上。例如，Java不允许多继承。如果分析时连实现语言的细节也要考虑进去，就会浪费很多时间。而对于分析类来说，只需要表示出类的职责即可，不必理会实现语言的约束。

* + 高于实现方式

在为需求考虑系统实现的时候，可以不考虑采用哪一种具体的实现方式。例如安全认证，可能的实现方式有LDAP、CA认证、JAAC等，如果在进行需求分析时就开始考虑这些实现方式，一方面会付出过多的精力，另一方面考虑过多的具体细节相反会扰乱需求实现的分析工作。

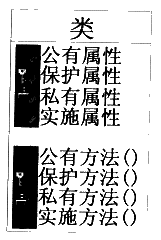
如果用对分析类，我们只需要用一个认证控制类代表系统需要这样一个程序逻辑来完成需求即可，至于实现方式则可以先放下不谈。

可以看到，一方面由于分析类的抽象层次较高，基本上停留在“概念”阶段，相对于设计实现、语言实现、实现方式这些较低抽象层次的工作来说，需要考虑的信息量要少得多，而能够让分析工作专注在实现需求。因为相对于设计模式、编程风格这些因素来说，忠实地实现需求才是项目成功的第一位。

另一方面，也由于分析类的抽象层次较高，概括能力就很强，也就比设计和实现要稳定。在一个演进式的软件生命周期里，维护稳定的分析类比维护易变的设计类要投入更少的精力，更容易获得一个稳定架构来指导整个软件的开发。

## 设计类

* 基础概念
  + 设计类是系统实施中一个或多个对象的抽象；设计类所对应的对象取决于实施语言。设计类用于设计模型中，它直接使用与编程语言相同的语言来描述。到了这个阶段，设计类已经直接映射到实现代码了，因此设计类依赖于实施语言。另一方而，设计类来源于前期的系统分析，在统一过程中，类不是凭空想象出来的，它们可以一一映射到前期系统分析的成果物上。从这个观点出发，分析类的重要性就能够体现出来。分析类为设计类中所需要的界面、逻辑和数据提供了非常好的抽象基础，设计类可以非常容易和自然地从分析类中演化出来。



* + 在面向对象的原则里，应该假定类有具体类和封装体，即使实施语言并不支持这样做。虽然设计类取决于实现语言，实现方式各不相同，作为统一语言，UML还是为设计类的概念进行了定义：**设计类由类型、属性和方法构成。设计类的名称、属性和方法也直接映射到编码中相应的class、property和method。**
* 类
  + 类对对象进行定义，而对象又实现用例。
  + 类的来源可以是用例实现对系统所需对象的需求，这是为实现业务需求而定义的；也可以是任何以前已开发的对象模型，即现有的系统模块、采用的软件框架、第三方产品等。
  + 类说明了对象是什么，同时也就决定了对象拥有什么属性，具有什么方法。
  + 类是对对象某一方面特征的归纳和抽象，而对象则是类实例化的结果。
  + 在从分析模型向设计模型转化的过程中，可以把分析类认为是需求分析过程中得到的对象，进而抽象出具体的类。在实际的工作中，设计类的获得很多时候可以参照某个软件框架的指导或某个规范的要求。例如采用J2EE作为软件架构时，servlet、session bean、entity bean等就是必须遵守的规范，根据规范我们可以从分析类中抽象出设计类。
* 属性
  + 属性是对象特征，属性同时表明了对象的唯一性。
  + 属性名称是一个名词，描述与对象有关的属性的角色。
  + 创建对象时，属性可以具有初始值。
  + 在为对象建立属性时需要考虑的一个指导原则是**只有对象单独具有的特征才能建模为属性。否则，应该使用与类（其对象代表特征）的关联关系或聚合关系对特征进行建模。**
* 方法
  + 原则上，访问对象或影响其他对象的属性或关系的唯一途径就是方法，**直接访问和修改对象的属性是不提倡的。**
  + 对象的方法由它的类进行定义。绝大多数情况下，类定义的方法都是由实例化后的对象来执行的，即这些方法为对象方法；但有时候方法也可以由类来执行，这种方法称为类方法。
  + 方法的作用是访问和改变对象的属性，有时候方法仅仅封装了算法，执行该方法不会改变对象的属性。
  + 在面向对象中，需要注意的原则是**一个对象的属性只应该由它自己的方法来改变**。
* 可见性
  + 类的属性和方法都有相似的可见性定义，在UML中，可见性被归纳为以下四类：
    - **公有**：除了类本身以外，属性和方法对其他模型元素也是可视的。公有可见性应该尽量少用，公有意味着将类的属性和方法暴露给外部，这与面向对象的封装原则是矛盾的。暴露给外部的内容越多，对象越容易受影响，越容易形成高耦合度。
    - **保护**：属性和方法只对类本身、它的子类或友元（取决于具体语言）是可视的。保护可见性是默认的可见性；它保护属性和方法使其不被外部类使用，防止行为的耦合和封装变得松散。
    - **私有**：属性和方法只对类本身和类的友元（取决于具体语言）是可视的。私有可见性可以用在不希望子类继承属性和方法的情况下。它提供了从超类对子类去耦的方法，并且减少了删除或排除未使用继承操作的需要。
    - **实施**：属性和方法只在类本身的内部是可视的（取决于具体语言）。实施可见性最具限制性；当只有类本身才可以使用操作时，使用这种可见性。它是私有可见性的变体。

## 关系

在UML中，关系抽象出对象之间的联系，让对象构成某个特定的结构。

### 关联关系（association）

* + 使用一条直线表示，如A关联B：
  + 描述了不同类的对象之间的结构关系，它在**一段时间内**将多个类的实例连接在一起，这与依赖关系是不同的，依赖关系通常表示两个实例之间的临时关联关系。描述了某个对象在一段时间内一直“知道”另一个对象的存在。
    - 单向“知道”：，A“知道”B，但B“不知道”A，**箭头读“知道”**
    - 双向“知道”：
  + 特别的：

在用例模型中，单向关联关系用于连接参与者和用例，箭头由参与者指向用例，表示参与者“知道”用例的存在，而用例是“不知道”参与者的。

### 依赖关系（dependency）

* + 使用一条带箭头的虚线表示：，A依赖B，**箭头读“依赖”**
  + 依赖是一种特殊的关联关系：依赖关系除了“知道”其它对象存在，还会“使用”其它对象的属性和方法，其它对象的修改会导致这个对象的修改。
    - 单向“依赖”：，A依赖B，B对A造成影响
    - 双向“依赖”：杜绝，一种非常不好的结构

### 扩展关系（extends）

* + 使用一条带箭头的虚线加版型<<extends>>来表示：，A扩展出B。
  + 特别用于在用例模型中说明向基本用例中某个扩展点插入扩展用例。
  + 一般来说，扩展用例是带有抽象性质的，它表示了用例场景中的某个“支流”，由特定的扩展点触发而被启动。所以严格来说扩展用例应当用在概念用例模型中，通过分析业务用例场景抽象出关键的可选核心业务而形成扩展用例。不过，在业务模型当中使用也是可以接受的，它可以更显式地表示出一个复杂业务用例的各个“分支”。
  + 与包含关系不同的是，**扩展表示的是“可选”**，而不是“必需”，这意味着即使没有扩展用例，基本用例也是完整的；如果没有基本用例，扩展用例是不能单独存在的；如果有多个扩展用例，同一时间用例实例也只会使用其中的一个。
  + 在建模过程中，我们使用扩展关系可能基于以下理由：
    - 表明用例的某一部分是可选（或可能可选）的系统行为。这样就可以将模型中的可选行为和必选行为分开。
    - 表明只在特定条件（有时是例外条件）下才执行分支流，如触发警报。
    - 表明可能有一组行为段，其中的一个或多个段可以在基本用例的扩展点处插入。所插入的行为段（以及插入的顺序）将取决于在执行基本用例时与主角进行的交互。
    - 表明多个基本用例中都有可能触发一个可选的分支流。从这个意义上说，扩展用例也代表了多个用例的可复用部门。

### 包含关系（include）

* + 使用一条带箭头的虚线加版型<<include>>来表示：，A包含B
  + 特别用于用例模型，说明在执行基本用例的用例实例过程中插入的行为段。
  + 包含用例总是带有抽象性质的，基本用例可控制与包含用例的关系，并可依赖于执行包含用例所得的结果，但基本用例和包含用例都不能访问对方的属性。从这种意义上讲，包含用例是被封装的，它代表可在各种不同基本用例中复用的行为。因此，与扩展用例一样，包含用例也应当用在概念用例模型中，通过分析业务用例场景而抽象出关键的必选的核心业务而形成包含用例。同样，在业务模型中使用也是可以接受的，它可以显式地表示出那些可复用的业务过程。
  + 与扩展用例不同的是，**包含用例表示的是“必需”**而不是“可选”，这意味着如果没有包含用例，基本用例是不完整的，同时如果没有基本用例，包含用例是不能单独存在的。
  + 在建模过程中使用包含关系可能基于以下理由：
    - 从基本用例中分解出这样的行为：它对于了解基本用例的主要目的并不是必需的，只有它的结果才比较重要。
    - 分解出两个或更多个用例所共有的行为。

### 实现关系（realize）

* + 使用一条带空心箭头的虚线表示：，A实现B，**箭头读“实现”**。
  + 在用例模型中连接用例和用例实现，说明基本用例的一个实现方式。
  + 实现所代表的含义是，基本用例描述了一个业务目标，但是该业务目标有多种可能的实现途径，每一种实现途径可以用用例实现（或称用例实例）来表示，而用例实现与基本用例之间就构成了实现关系。换言之，每个实现途径都实现了基本用例的业务目标。

### 精化关系（refine）

* + 使用一条带箭头的虚线加版型<<refine>>来表示：，A精化B
  + 特别用于用例模型，一个基本用例可以分解出许多更小的关键精化用例，这些更小的精化用例更细致地展示了基本用例的核心业务。精化关系用来连接基本用例和精化用例，说明精化用例是由基木用例精化得来的。
  + 精化关系也可以用于模型与模型之间，表示某个模型是通过精化另一个模型而得来的。
  + **与泛化关系不同的是，精化关系表示由基本对象可以分解为更明确、精细的了对象，这些子对象并没有增加、减少、改变基本对象的行为和属性，仅仅是更加细致和明确化了。在泛化关系中，基本对象被泛化成为了对象后，子对象继承了基本对象的所有特征，并且子对象可以增加、改变基本对象的行为和属性。**
  + 另一方面精化关系仅仅用于建模阶段，在实现语言中是没有精化这一语义的。泛化则等同于实现语言中的继承语义。

### 泛化关系（generalization）

* + 使用一条带空心箭头的直线表示：，A继承自B，**箭头读“继承”**。
  + 泛化关系可用于建模过程中的任意一个阶段，**说明两个对象之间的继承关系**。继承意味着祖先的定义（包括任何特征，如属性、关系或对其对象执行的操作）对于后代的对象也是有效的。泛化关系是从后代类到其祖先类的关系。
  + 不赞同在用例之间使用泛化关系，尽管UML认为它是合法的。原因是用例带有原子特性，每个用例都应当是独一无二的。用例描述了参与者完成一个目标的整个过程，如果采用泛化关系，很难描述子用例继承了基本用例的什么。过程？还是业务实体？如果仅仅为了将用例之间的可复用部分或用例的可扩展部分描述出来，那么使用包含关系和扩展关系就足够了。

### 聚合关系（aggregation）

* + 使用一条带空心菱形箭头的直线表示：，A聚合到B上，或B由A组成。
  + 聚合关系用于类图，特别用于表示实体对象之间的关系，**表达整体由部分构成的语义**。
  + 与组合关系不同的是，**聚合关系中的整体和部分不是强依赖的**，即使整体不存在了，部分仍然存在。

### 组合关系（composition）

* + 使用一条带实心菱形箭头的直线表示：，A组合成B，或B由A构成。
  + 组合关系用于类图，特别用于表示实体对象关系，**表达整体拥有部分的语义**。例如母公司拥有许多子公司。
  + 组合关系是一种强依赖的特殊聚合关系，如果整体不存在了，则部分也将消亡。

## 组件

* 基础概念
  + 组件是系统中实际存在的可更换部分，它实现特定的功能，符合一套接口标准并实现一组接口。组件代表系统中的一部分物理实施，包括软件代码（源代码、二进制代码或可执行代码）或其等价物（如脚本或命令文件）。



* + **建模过程中，通过组件这一元素对分析设计过程中的类、接口等进行逻辑分类**，一个组件表达软件的一组功能。例如一个网站有用户注册和用户维护两个目标功能，通过对网站需求的用例分析和设计，得到许多类和接口，这些类和接口实现网站的用户管理。出于构件化的需要，把那些紧密合作的类和接口组合起来实现一组特定的功能，形成一个组件。
  + 一个类可能被分派给多个组件以完成该组件的功能，当组件被编译或打包成一个物理文件时，每个组件都拥有这个类的一个拷贝或者引用该类的途径。
  + 现实中有时人们定义一个组件仅仅是为了组织代码，那用包来表示不就足够了吗？按照笔者的理解，一个组件应当具有完备性、独立性、逻辑性和透明性。
  + 一个组件应当是一个独立的业务模块，有着完备的功能，可独立部署，一个组件可以看成是一个完备的服务。从SOA架构的观点来看，一个SOA服务与其他服务是没有依赖关系的，服务与服务之间仅仅保持着松耦合的通信关系。如果组件之间有着依赖关系，那么定义组件就没有什么实用意义了，因为组件不能够独立存在。所谓构件化开发就是像搭积木一样建设系统，很难想象积木块之间有着千丝万缕的依赖关系还能够“自由”地搭建系统，
* 组件具备的特性
  + 完备性

组件包含一些类和接口，一个组件应当能够完成一项或一组特定的业务目标（或说功能）。从调用者的观点看，它不需要调用多个组件来完成一个业务请求。

例如我们将组件A定义为用户注册，那么我们应该在组件A中包含所有实现用户注册的必需的类和接口，在任何时候，仅通过组件A就可以注册一个用户而无须访问组件外的其他类；而组件B定义为用户维护组件，我们就应当在组件B中包含所有实现用户维护的必需的类和接口，使用者可以通过组件B完成维护用户的功能而无须访问组件外的其他类。

* + 独立性

组件应当是可以独立部署的，与其他组件无依赖关系，最多仅保持关联关系。例如可以把组件A部署到服务器l,把组件B部署到服务器2，虽然组件A和B都共同使用用户数据，但是A与B之间无依赖关系。也就是说，组件与组件之间应当是松耦合关系。

* + 逻辑性

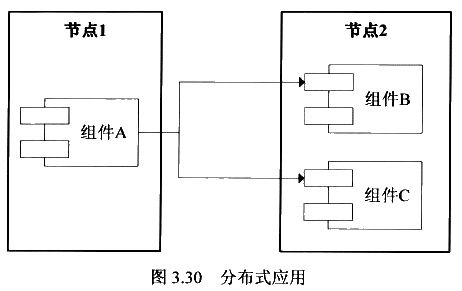
组件是从软件构件设计的观点来定义的，并非从需求中可以直接导出。组件建立在系统分析和设计的基础上，对己经实现的功能进行逻辑划分。组件的定义是为了规划系统结构，将一个复杂的系统分解为一个个具有完备功能的，可独立部署的小模块。这些小模块可大可小，从理论上说，可任意选择一部分功能定义一个组件。

* + 透明性

组件的修改应当只涉及组件的定义以及组件中所包含的类的重新指定，而不应当导致类的修改。例如当一个组件的功能变化时，它所包含的类可能从原来的类A、类B、类D变成类B、类C、类D，但是类ABCD都不应当被修改。

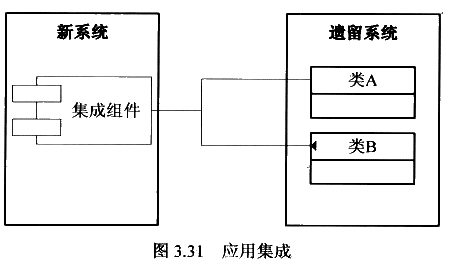
* 使用组件
  + 分布式应用

在分布式应用的情况下，系统的功能可能被部署在**异构环境**下，一个业务目标可能需要经历两个甚至多个节点才能完成。这时需要将实现业务目标的那些类和接口规划成一些组件，每个组件完成这个业务目标中的一部分功能。这些组件可被独立部署在不同的节点上，相互之间通过既定的通信协议交互来完成业务目标。



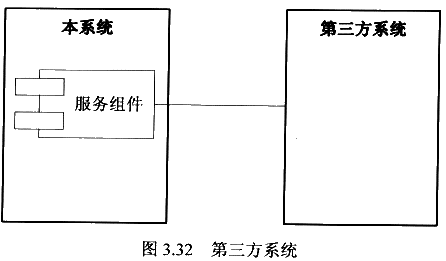
* + 应用集成

在应用集成项目中，经常面临新业务和遗留系统问题。新业务需要调用遗留系统的功能，但是又不能修改遗留系统。原因可能是修改遗留系统的代价高昂，也可能是结构差异导致新旧系统无法直接通信。不管什么原因，为了保证遗留系统能够被集成到新系统中，一个解决方案就是在新系统中规划出一些组件，这些组件所拥有的接口完成遗留系统的功能。新系统中的其他业务模块与这些组件交互，而这些组件则拥有遗留系统的代码或者通过某种方式（代理模式、适配器等）使用遗留系统。



* + 第三方系统

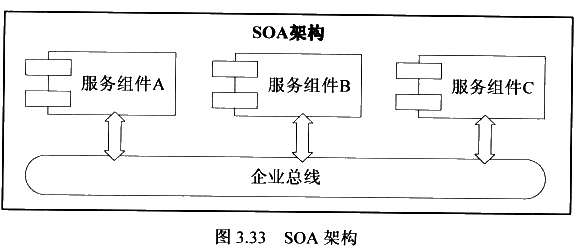
如果在建设的项目中，有第三方系统要访问本系统，出于松耦合的考虑，让第三方系统直接使用或者说把本系统中的类直接暴露给第三方系统是很糟糕的设计。因此，有必要将本系统要提供给第三方系统使用的功能定义成一系列组件，让第三方通过组件来访问本系统。在这些组件中，除了包含本系统的实现类外，还可以根据实际情况通过提供这些实现类的代理、适配器、消息中间件等手段来解耦第三方系统对本系统的依赖。



* + SOA服务

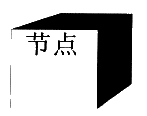
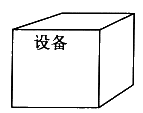
SOA（Service Oriented Architecture）面向服务的架构是目前新兴的软件架构，有人说SOA是下一代软件的发展趋势。它将系统结构划分为粗粒度的服务组件SCA，每个服务组件都遵循一系列的标准和规范，通过标准的通信协议与其他服务交互，服务与服务之间是松耦合的。

在SOA中，系统分析、设计、开发都以服务为主，每个服务都具有上述组件的所有特点。实际上组件的概念非常类似于SOA的服务。如果要开发一个SOA架构的应用系统，那么开发SOA服务的过程实际上就是定义组件的过程。在SOA架构下，系统功能由一个个的服务向外部暴露，也就是说，系统被定义成一个个的组件。这些服务是松耦合的，它们之间通过**企业总线**交互以完成业务功能。



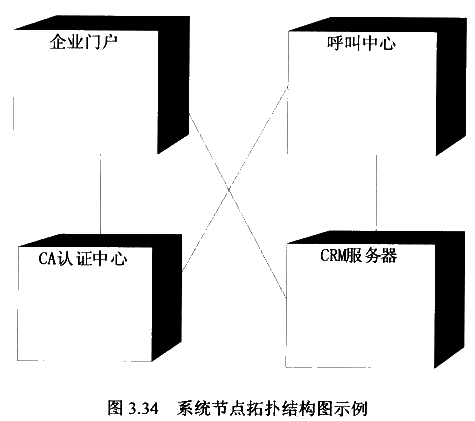
## 节点

* 基础概念
  + 节点是带有至少一个处理器、内存以及可能还带有其他设备的处理元素。在实际工作中，一般说来服务器、工作站或客户机都可以称为一个节点。
  + 节点是应用程序的部署单元。节点元素特别用于部署视图，描述应用程序在物理结构上是如何部署在应用环境中的，是一种包括软、硬件环境在内的拓扑结构描述。

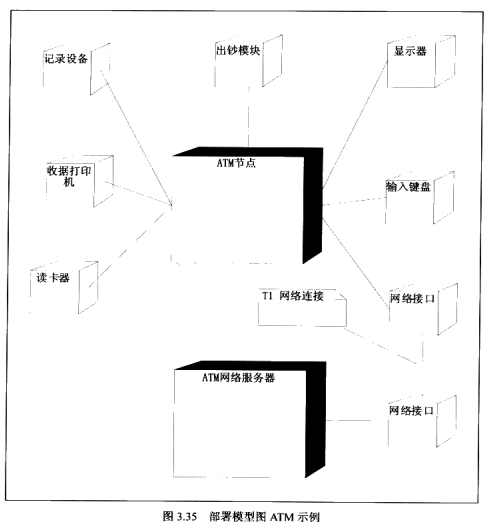
* 应用环境
  + 分布式应用环境

在分布式应用环境中，通常会有多于一个的服务器、处理设备或者中间件。所开发出的应用程序会部署到这些不同的服务器或处理节点上，通过描述这些服务器之间的调用和依赖关系以表达应用环境的拓扑结构。



* + 多设备应用环境

如果应用环境中包括多种硬件设备，为了表达这些硬件设备的结构，应当使用节点元素来绘制部署视图。



# UML核心视图（语法）

UML语言 = UML元素 + UML视图

UML视图将基本元素组织在一起，形成有意义的句子。

## 静态视图

表达静态事物。只描述事物的**静态结构**，而不描述事物的动态行为。

包括用例图、类图、包图等

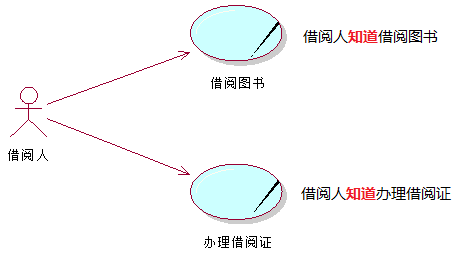
### 用例图

元素组成：参与者、用例、边界、关系等

用例视图采用参与者和用例作为基本元素，以不同的视角展现系统的功能性需求。用例视图是了解系统的第一个关口，人们通过用例视图得知一个系统将会做什么。对客户来说，用例视图是他们业务领域的逻辑化表达；对建设单位来说，用例视图是系统蓝图和开发的依据。

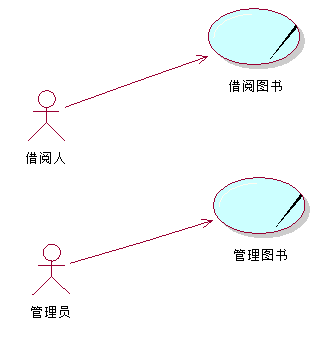
#### 业务用例视图 【展示系统功能性需求】

* 业务用例视图 = 业务主角 + 业务用例
  + 从业务主角视角展示



这个视角有利于向业务主角确认其业务目标是否都已经齐全，以此来检查是否有遗漏的业务用例没有发现。

* + 从业务模块视角展示



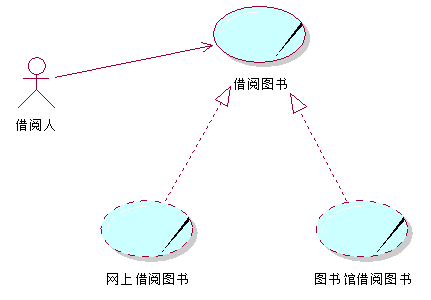
这个视角有利于从业务的完整性角度出发，检查完成某个业务的所有业务主角和业务用例是否已经齐全，以此来检查是否有遗漏的业务用例没有发现。

* + 从其它视角展示

例如，可以从部门的视角绘制一个部门所参与的全部业务用例视图，或者从一个重要业务实体的生命周期角度，例如一份文件来描述文件从产生到销毁的整个生命周期过程中涉及的业务主角和业务用例视图。

在建模过程中，每当需要展现某个方面的视角时，都可以将获取到的业务主角和业务用例用用例视图展现出来，不要拘泥于某几个固定的形式。

#### 业务用例实现视图 【展现系统功能性需求的实现途径】



实际工作中，业务用例实现视图通常省略不绘，原因是一般一个业务用例只有一个实现途径，但是绘制业务用例实现视图是有用的，是符合软件工程**需求可追溯原则**的好的做法。

#### 概念用例视图

对业务用例进行分析，分解出**关键概念**的用例，使用概念用例视图展示。

概念用例和业务用例之间的关系一般包括扩展<<extends>>、包含<<include>>和精化<<refine>>。

#### 系统用例视图

#### 系统用例实现视图