# 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

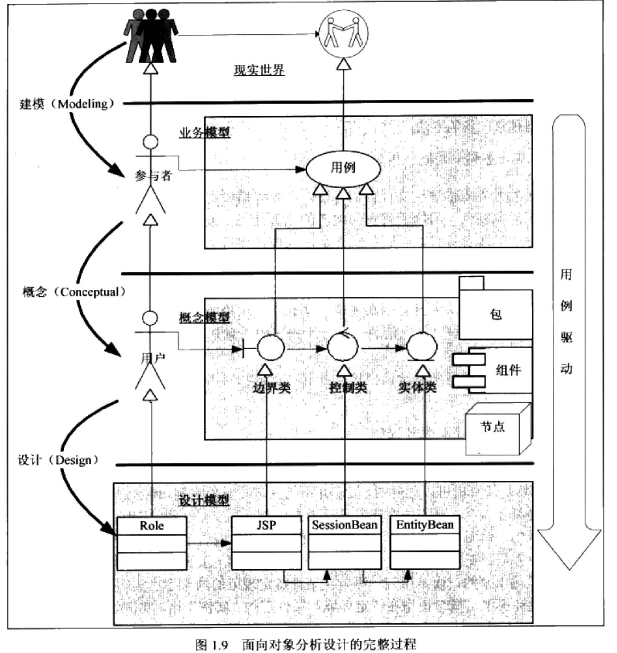
1. 从业务模型至概念模型 【开发阶段】
   1. UML通过被称之为概念化的过程（Conceptual)来建立适合计算机理解和实现的模型，这个模型称为分析模型（AnalysisModel)。分析模型介于原始需求和计算机实现之间，是一种过渡模型。分析模型向上映射了原始需求，计算机的可执行代码可以通过分析模型追溯到原始需求；同时，分析模型向下为计算机实现规定了一种高层次的抽象，这种抽象是一种指导，也是一种约束，计算机实现过程非常容易遵循这种指导和约束来完成可执行代码的设计工作。
   2. 绘制分析模型最主要的元模型有：

* 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。

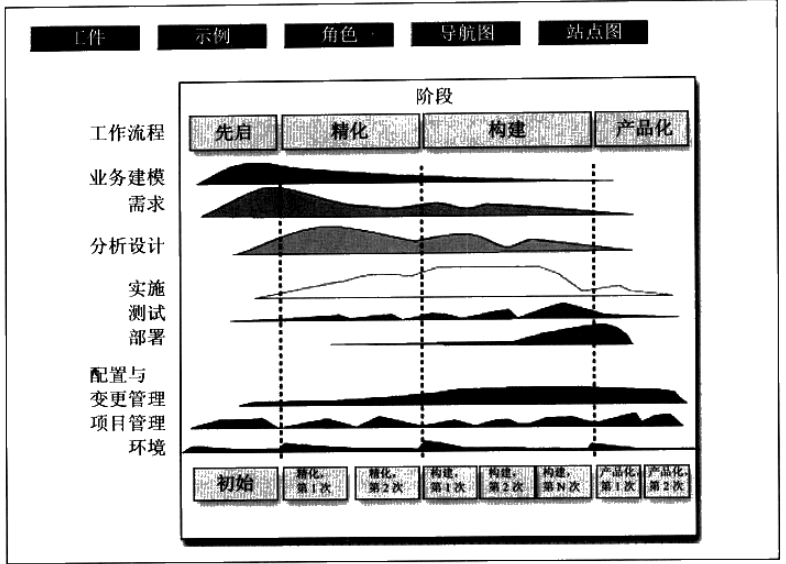
## 面向对象分析设计完整过程



# 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

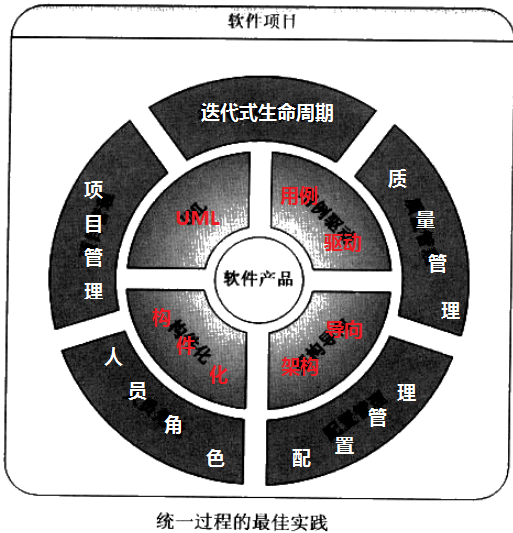
* UML是一种语言，用来描述软件生产过程中要产生的文档，RUP则是指导如何产生这些文档以及这些文档要讲述什么的方法。
* RUP和UML不是一体的，只是软件方法和建模语言的一个完美结合。要站在**软件过程**的角度，先了解软件项目如何做的，然后再从UML中寻找需要的工具，将软件过程中的要求记录下来。

## 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。

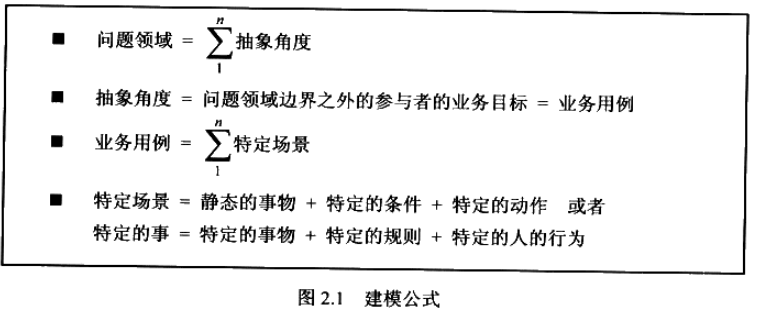
抽象角度的不同决定了建模方向的不同，抽象确定后，相应模型能够建立起来。

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。

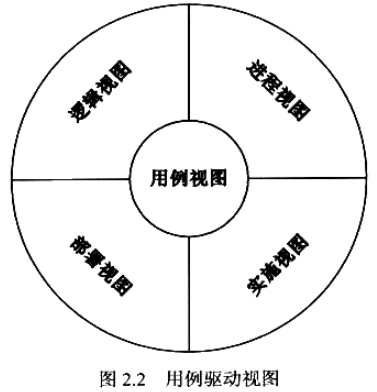


## 用例驱动

* 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。

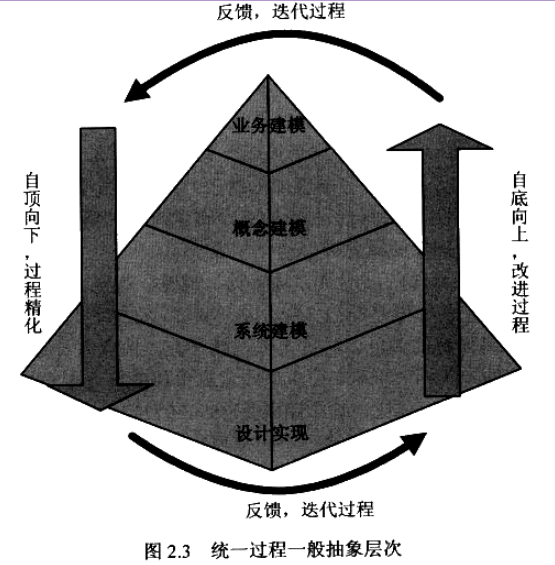
* 实施视图

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。

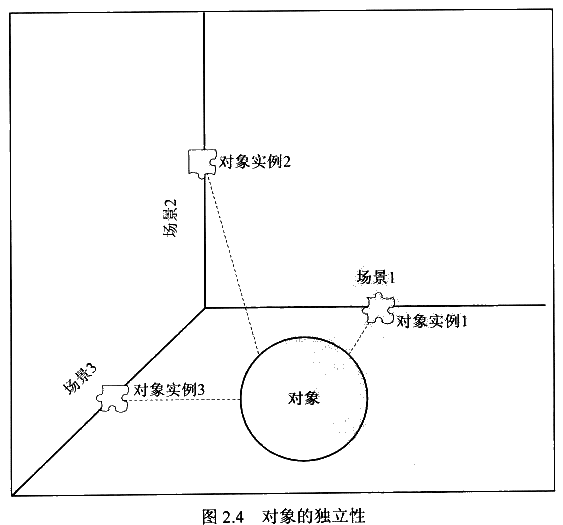


## 视图

* 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。**

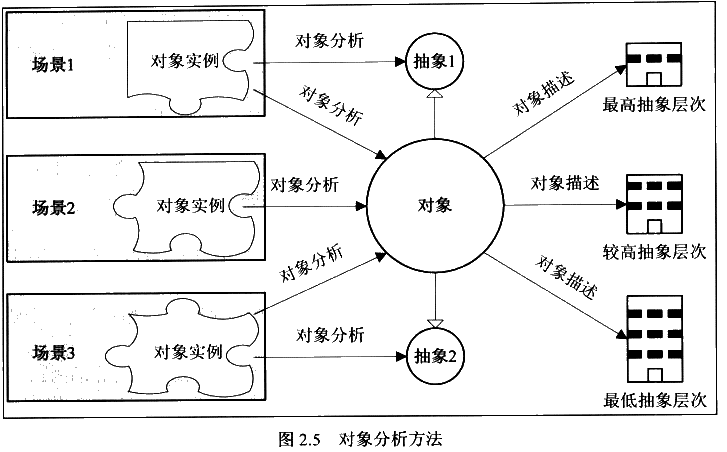
## 面向对象分析（OOA）方法

* 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。



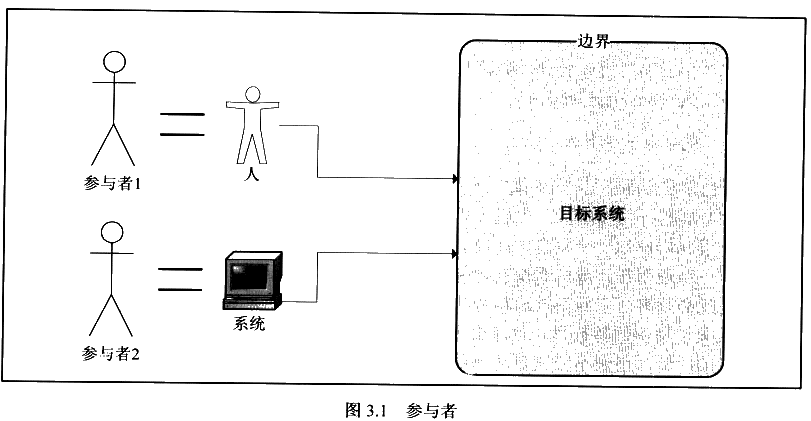
# UML核心元素

## 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



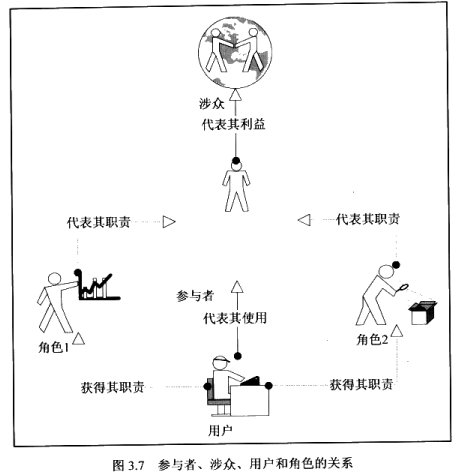
* + 参与者位于系统边界之外（“谁”作为主角，是参与者）

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。

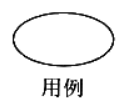


* + 参与者的核心地位还体现在，系统是以参与者的观点来决定的，参与者对系统的要求，对系统的表述完全决定了系统的功能性。
* 检查点
  + 检查发现的参与者是否正确
    - 是否您己找到所有的参与者？也就是说，是否您己经对系统环境中的所有角色都进行了说明和建模？虽然您应该检查这一点，但是要到您找到并说明了所有用例后才能将其确定。
    - 每个参与者是否至少涉及到一个用例？删除未在用例说明中提及的所有参与者，或与用例无通信关联关系的所有参与者。
    - 您能否列出至少两名可以作为特定参与者的人员？如果不能，请检查参与者所建模的角色是否为另一角色的一部分。如果是这样，您应该将该参与者与另一参与者合并。
    - 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。

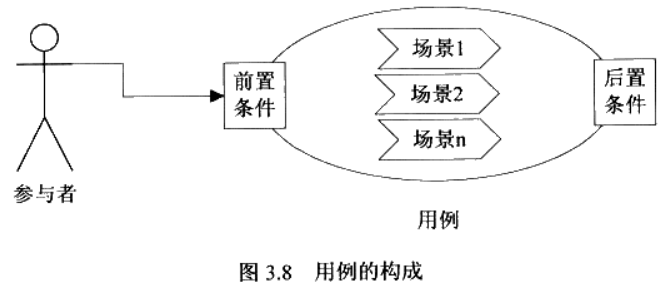
## 用例

**用例通过“外力“将其它的独立的对象关联起来。**

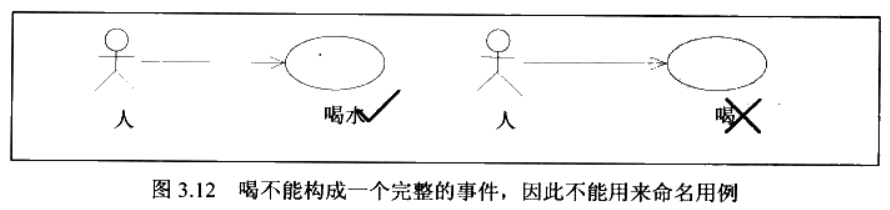
* 基本概念
  + 官方定义：用例定义了一组用例实例，其中每个实例都是系统所执行的一系列操作，这些操作生成特定主角可以观测的值。



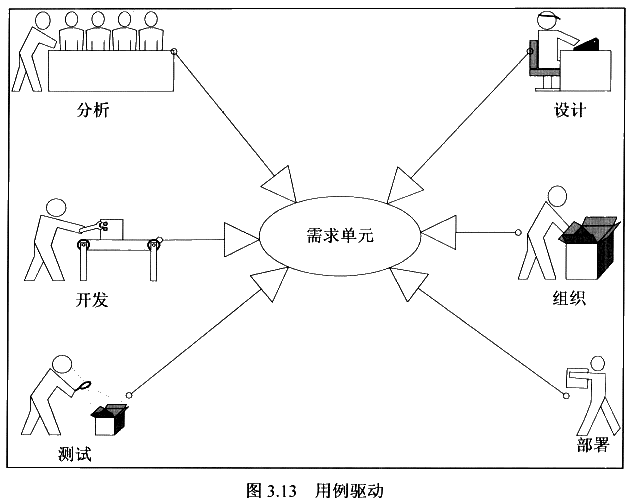
* + 一个用例就是与参与者交互的，并且给参与者提供可观测的有意义的结果的一系列活动的集合。
  + 对于一件事情，要完成这件事情，需要做一系列的活动；而做一件事情可以有很多不同的办法和步骤，也可能会遇到各种各样的意外情况，因此这件事情是由很多不同情况的集合构成的，在UML中称之为用例场景。
  + 一个场景就是一个用例的实例。
  + 用例的启动是需要条件的，启动用例的前提称之为**前置条件**，用例执行完后获得的结果称为**后置条件**。
  + 一个完整的用例由参与者、前置条件、场景、后置条件构成。



* + 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。