统一用例方法

版本 A0

IT 之源　张恂
www.zhangxun.com
zhangxun2001@hotmail.com
2004 年 1 月 3 日

摘要

用例是 10 多年来最重要的需求分析技术，在保障全球各类软件的成功开发中发挥了极其重要的作用。鉴于目前用例技术尚存在几种相互竞争的流派，在实践中如何仔细甄别，取长补短，有效地作出选择，成为实践者必须面对的现实问题。本文根据笔者近年来的培训教学和项目咨询经验，详细分析比较了亚克申和寇本这两种主流用例方法之间至少 10 处以上的明显差异，逐一给出消除不一致的建议，并在此基础上提出了统一用例方法（UUCM，Unified Use Case Method）。

关键词：

用例，UML，RUP

一、用例基础

1.1 用例简史

用例技术大体上经历了萌芽、成熟和发展 3 个阶段[12]，最早可追溯到上世纪 60 年代末 UML（统一建模语言）、RUP（Rational 统一过程）之父 Ivar Jacobson（伊瓦·亚克申）博士在著名的瑞典爱立信公司领导程控电话交换机开发时采用的 traffic case（话务案例），1986 年前后亚克申博士在 OOPSLA 大会上发表的论文[4]标志着用例的正式诞生。1992 年，亚克申博士在其名著《面向对象软件工程：用例驱动方法》[3]中正式推出了当时已相当完善的用例方法，用例驱动成为了 Objectory 过程（RUP 前身之一）的核心内容，从此用例在国外软件工程界得以迅速普及，并于 90 年代中后期被 RUP 和 UML 吸收为核心要素，亚克申博士在《统一软件开发过程之路》[4]和《统一软件开发过程》[5]中对此作了精彩的阐述，这些书因为成为了解其用例思想的重要著作。如今，作为必不可少的关键内容，用例技术总是被所有的当代需求工程名著所引用，而“用例”本身也几乎成为功能需求的代名词。（注：考虑到历史上的渊源关系，为了方便起见，以下我们用术语“亚克申方法”或“亚克申用例”来统称 RUP、UML 及其支持者所采用的用例方法，尽管它们彼此之间可能存在细微的差别，而且也不一定全部是由亚克申博士本人所提出或赞成的。）

用例另一支主流派别代表人物 Alistair Cockburn（阿里斯代·寇本）上世

纪 90 代初从亚克申那里学习了用例，随后通过十年认真广泛的实践对其进行了继承和发展。寇本于 1995 至 1997 年间提出了著名的“基于目标的用例”方法[2]，他的方法和思想集中体现在《Writing Effective Use Cases》(以下简称为 WEUC)[1]中，以结构化/半结构化文本用例为中心是寇本方法的一大特色，该书可以说是迄今为止最为详细的一本用例教材，对于指导实践者如何写好文本用例具有很高的价值。

1.2 用例定义

下面，先让我们从什么是用例开始讨论。亚克申定义[7]强调用例是系统执行的一个动作序列（注：这其中也包括与用户的交互），这些动作必须对某个特定的使用者（Actor）产生可观测的、有价值的结果。(注：Actor 实质上是用户所扮演的一种角色)

那么到底什么结果叫“可观测、有价值”呢？虽然两种用例本质上是一致的，但亚克申定义对此没有明说，而寇本定义[1]则更加完善和明确。它首先强调用例是各种系统受益人（Stakeholder，又译“干系人”）之间的一种行为契约（注：行为包括对象的活动、动作和对象之间的交互等），建立契约的目的是为了达成某种目标，因此每一个用例及其名称实际上都应代表一个用户目标，这个目标是否得到真正满足正是判断我们抽取的某个用例是否“有价值”的关键。寇本还点出了要通过用例的具体执行来展现 Actor 的目标是如何实现或失败的，而一个用例其实就是多个在不同条件下执行并可能导致许多不同后续状态的情节（scenario，又译“场景”）的叠加，这就是用例结果的“可观测”。

因此综合起来，我们只要抓住这样几个关键词：**目标、行为契约、行为(事件)序列（动作和交互）、情节、可观测、有价值**，就可以比较准确地描述出用例的本质特征。

1.3 用例的重要性

为什么用例如此重要？一言以蔽之，这是因为**用例是一种普遍存在的客观现实**，而实践证明，**用例技术是迄今为止最为深刻、准确和有效的系统功能需求描述方法**。

功能需求是指系统输入到输出的映射以及它们的不同组合[9]，任何功能必然要通过外部环境与系统之间的交互才能完成，这不正是用例所要反映的内容吗？因此，我们可以在内容和形式上把用例和系统的功能需求等同起来，并且得出推论：**只要是软件，必然都存在用例**（虽然有时候不一定非要用某种具体的用例格式来描述），其中即包含数据流，也包含控制流，既包含消息发送和数据交换（交互），也包括活动/动作的执行以及状态的变迁。这些就是用例的本质（现象背后那个真实的、抽象的“胚”），而各种格式文本、UML 图形（我们至少可以用 4 种 UML 动态图来描述用例）不过是用例的外部表现形式。所以，与其说亚克申博士发明了用例，还不如说亚克申博士早在 20 年前就**发现**了用例这种客观现实，并最终**发明**了用例表示和用例驱动软件过程的方法。

那么，什么情况下不太适合采用用例方法？主要有两种情况：（1）用户很少或没有，接口也很少，如科学计算/仿真软件、杀病毒软件、编译程序、内存管理程序等[9]；（2）功能需求非常简单，非功能需求和约束占主导地位。显然，如

今绝大部分的应用软件、系统软件，尤其像电信、银行、保险、税务、制造业、企业信息化等领域的复杂系统，都是符合用例适用条件的，这从一个侧面反映出用例技术的广泛适用性。请注意，即使在上面两种情况下也并不是说这些软件的用例就不存在了，而只是表明它们的功能需求很简明或不太重要，除用例之外可能还有更加适用的方法。

正确有效的软件需求必须是可测试、可验证的。过去我们描述功能需求可能采用了很多种方法，除严格的形式化方法外，普遍的缺点就是粒度太粗、精度不够，大多停留在受益人要求（request）和特性（feature）这一层[9]。比方说，针对某个“打印报表”功能，通常仅用一段话来描述它的静态输入和输出是不够的，还应该描述出用户打印报表的具体操作流程，它有哪些特殊条件和选项设置，以及它与其他需求的依赖关系等等。很多团队在需求尚未细化到用例这一层次时就开始匆匆编码了，结果往往导致大量需求风险乃至架构风险被隐藏到构造、移交阶段才发现，这必然造成频繁的返工和严重的资源浪费。及时准确地抓住需求契约——用例这一关键，可以帮助我们在不失实用性、灵活性的情况下，有效地避免项目后期大量非正常的需求变化，为进度愈来愈紧的项目赢得宝贵的时间，提高项目的成功率。

1.4 相关译法

借此机会，顺便谈谈对 use case 有关术语翻译的看法。

笔者认为“用例”是目前较好的译法，这个词可能来源于大家熟知的“测试用例”。有人认为把 use case 翻译成“用例”是错误的[11]，理由是：“‘例’是被列举出来以说明某种情况的个别事物，use case 是对一项系统功能使用情况的普遍适应的描述，而不是对个别 actor 或者在个别条件下使用这项功能才适应，它也不是通过举例的方式来描述的”，所以不能叫作“用例”。此种说法不尽全面，而且有些牵强（先不管它正确与否），其实 use case 到底是个别的，还是群体的（普遍适应），取决于我们的视点。虽然对于单个的 scenario 来说，use case 是多个情节的叠加，是一个整体的复合概念，但是我们知道，一个系统的功能必定是可数的、有限的，而每一个功能都可以表示为一个 use case，所以在观察系统提供的所有功能需求的集合这个层面上，use case 又是一个一个可数的个体（“椭圆”），每一个都代表了不同的用户目标，适用于个别的 actor 和个别特定的前置条件。同一个事物既是个体的又是整体的，这种现象并不足怪，例如在 UML 对象-类-类元关系中，通常对象是类的实例，而类又是类元的实例，对类元来说，类、接口、子系统、use case 等等就是一个个个体的概念，类既是其对象实例的集合又是其类元集合的个别元素。可见，**把 use case 的“case”译成“例”并没有错**。

有的地方把 use case 翻译成“用况”，即“使用的情况”之意，意思的确不错（use case 的另一种说法是“使用的方式”）！可我总感觉这个词比较突兀、拗口，类似的还有“用案”，把 scenario 叫作“案况”，大概这些词读起来不太符合大家的习惯（类似地，既然可以叫“用况”，为什么不能叫“用情”呢？），所以现在“用例”的叫法还是越来越多了。

其实“用例”这个译法还有个附带的好处，通过它我们很容易把原本就存在紧密联系的 use case 和 test case（test case 来自于对 scenario 的分析，而 scenario 是用例的一次执行）从中文名称上也方便地统一起来。不过，这里我

们需要做一个小小的改进。中文的“测试用例”到底是指 test case（带定语的名词词组）呢，还是指对用例进行测试（testing the use cases，动宾词组）呢？显然这两者不易分辨，而且若“用例”和“测试用例”两个词同时出现在一个句子或一段话中，常常会让人感觉啰嗦和便扭。为了消除歧义，干脆以后把 test case 都叫做“**测例**”，这样不但比以前的叫法更加简洁明了，而且无论字面上还是语义上都很贴切。当然，用例和测例是不同层面的“例”。

现在市面上 Actor 也有多种译法，常见的包括“参与者、执行者、主角”等等。“参与者、执行者”的问题主要是不准确。首先，“参与者”通常让大家马上想到的词是 participant，而且请注意，一个用例的真正参与者决不是只有外部的 Actor，它们必然还包括系统本身及其内部的各种元素。“执行者”的问题与此类似：一个用例的真正执行者应该是系统本身！因此严格地讲这样译是错误的，兴许叫作“外部参与者”、“外部执行者”才更为恰当。“主角”的译法同样存在着矛盾。如果把 Actor 叫作“主角”，那么 Primary Actor 就应该叫作“主主角”了。看来 Actor 的译法中是不能含有“主”的，那么就剩下“角”了，而 UML 已经有了一个专门术语 role（角色），我们又不能把 Actor 直接叫作“角色”。

目前看来，把 Actor 意译成“使用者”是比较妥当的。在大多数情况下 Actor 的的确确就是用户（确切地说是系统用户所扮演的一种角色），所以我们可以用“使用者”这个词从字面上与“用户”（user）进行区分，但同时又保持两者语义上的联系。我们还可以把为系统服务的 Supporting/Secondary Actor（见下文）叫做“被使用者”（为了简化可以省略“被”字）或“辅使用者”。除了指系统的用户之外，“使用者”还有另一层含义，即 **Actor 是 use case 的使用者**（或被使用者），这种关系在 UML 用例图上应该可视化地表示为它们之间的连线（关联）。这样解释不但说的通，而且更便于不熟悉软件技术的业务人员理解。

当然，我们也不排除将来会找到“use case”、“actor”等术语更好的译法。

二、统一用例方法

2.1 理由

为什么要提出统一用例方法（UUCM），有这个必要吗？

我们发现，虽然寇本用例起源于亚克申用例，但两种用例方法各自经过十多年的发展，彼此之间逐渐出现了一些显著的差异，而且由于商业或其他方面的原因，目前我们尚未看到两者将要融合的明显趋势。两种方法各有优缺点，寇本强调基于目标的文本格式，亚克申用例则更突出 UML 的作用，如下文所示，两者的差别至少有 10 处之多。对于实践者来说，如何处理好这些明显差异，避免使用上的误区，再者能否巧妙地做出取舍，实现熊掌与鱼兼得，这些都是非常现实的在实践中必须面对的问题。

笔者认为采用统一的用例方法，把亚克申和寇本用例两者结合起来，甚至融合其他的用例方法（据说已知的各种用例表示方法多达 18 种以上[2][6]），在理论和实践上均是可行和必要的，这可能是我们目前可以采取的最佳策略。但是，UUCM 本身并不是一种全新的用例方法，而只是一种特定的处理方案。它是在对亚克申、寇本用例方法继承的基础上，试图消除这两种经典方法的不一致和矛盾，并探索可能的优化改进和后续发展，所以 UUCM 起到的作用与亚克申、寇本用例方法相比是次要的和微小的。

以下我们对两种方法的异同进行比较分析，并同时给出 UUCM 的建议。

2.2 层次

明确提出用例的层次和范围划分，是寇本“基于目标的用例方法”的精华所在[2]。显然，每个用例存在的意义是为了完成一定的用户目标。寇本把用例划分为 3 个目标层次：概要层、用户目标层和子功能层，并通过引入巧妙的 Why/How 技术帮助分析者找到合适的目标层次，从而可以有效地把握用例的粒度（真正的用例最终应落实到用户目标层），防止用例情节的爆炸。

亚克申方法及其相关文献在介绍如何有效地控制用例的粒度方面，大体上只有这样两条基本的判定规则：(1)通过判断内容是否有价值可以防止用例过小(例如，“输入发货地址”的粒度就太小，这不是一个真正的用例，相当于寇本的子功能层用例)；(2）通过判断具体内容是否可观测，可以防止用例过大（例如，“用户管理”就比较空洞，这不是一个有效的用例名称，相当于寇本的概要用例)。但如前所述，究竟什么叫“有价值”，什么叫“可观测”，如何把握好这个度，亚克申方法对此语焉不详，给 RUP 和 UML 的使用者带来了不少困惑，而寇本方法恰好出色地回答了这个问题。

UUCM:

通过寇本的分析，我们发现用例有纵向（层次）和横向（范围）之分，这些是非常有价值的概念，它们是对亚克申基础用例方法的丰富和完善，两者是不冲突的，完全可以在 RUP 相关的实践中加以运用。

值得注意的是，我们在实践中应该尤其关注用户目标层用例。寇本引入概要层用例的主要目的是为了包含一个或多个用户目标层用例，为系统提供全局功能视图，提出子功能层用例则是为了表达用户目标层用例的具体实现步骤。虽然有时为了简便，我们也把后两者叫做“用例”，但其实它们都不是真正的用例。

可能是为了防止滥用、误用用例的分解（这很容易引诱人倒退到结构化的功能分解)，亚克申方法没有提及甚至有意回避用例的层次问题。亚克申博士在最近的一篇文章中[12]引入了用例片段的概念，这意味着我们应该把寇本方法中的子功能层用例叫做用例片段以避免名称上的混淆，看来这是今后一种比较好的处理办法。

2.3 范围

关于用例范围的处理，两种方法基本上是一致的，都可分为业务用例、系统用例两种，区别在于寇本强调在格式上用图标或文字显式地表示出每个用例的范围，并且在用例层次划分的基础上提出了最外层（或最外围）用例的概念[13]，这在亚克申方法中是没有的。

UUCM：

用例的本质就是对象之间的一种协作和交互，这些内容是属于需求还是属于设计，关键就看你划定的讨论边界。我们不仅可以用它来描述发生在系统边界上的功能需求（用例的常规定义)，而且还可以同样的方式描述系统内部发生在子系统、构件、接口或类/对象边界上的黑盒交互，不过后者往往涉及到系统设计

范畴，属于用例的特殊用法，在做项目的需求分析时一般不予考虑。

正如寇本所提到的，同样一个用例名称，比如“取款”，可能实质上代表着两个截然不同的用例：一个是 ATM 取款（系统用例），另一个是银行柜台取款（业务用例）。所以在一个用例中明确标记出它的范围，是很有必要的。我们既可以用特定的图标，也可以用专门的格式字段来表明用例的范围，还可以对用例的名称加以修饰以便区分，比如“柜台取款”，“ATM 取款”。

当前的讨论边界（SuD，the System under Discussion）一般比较容易确定，那么如何从用例的范围上判断一个用例是系统用例，还是业务用例呢？（这是一个出现率很高的 FAQ）。有个小窍门：如果某个 SuD 或者用例的范围包含了人以及由人组成的团队、部门、组织的活动，那么针对这个 SuD 写出的用例必然是业务用例；如果该 SuD 仅仅是一些软件、硬件、机电设备或由它们组成的系统，并不涉及到人的业务活动，那么根据这个 SuD 写出来的肯定是系统用例。

由于系统（注意，这里的“系统”是指由软、硬件组成的 IT 系统）往往是业务的一部分，我们还可以得出推论：**对于某个系统用例 suc，通常总是可以找到一个业务用例 buc，buc 的范围要大于或等于 suc 的范围；当这两个用例的范围相等时，buc 就是 suc，或者 buc 的层次要高于 suc 的层次。**

2.4 包含与扩展

由于各方面复杂的原因，如何更好地表示用例之间的关系一直是个争议不断的问题，而且恐怕在将来较长的一段时间内对此还难有定论。我们发现，寇本与亚克申方法在用例的扩展、包含、继承等关系的使用细节上存在着一些明显差异，其中很大一部分原因可能是由于现有文献对用例关系的定义不够精准和完善，导致大家在应用时各取所需、各自解释造成的。如果我们在实践中对这一情况不加以留意，就可能给需求的沟通和交流带来障碍。

2.4.1 一头雾水

同样一对用例之间的关系，在寇本用例模型中是包含关系，在亚克申模型中却可能被表示成扩展关系。例如:

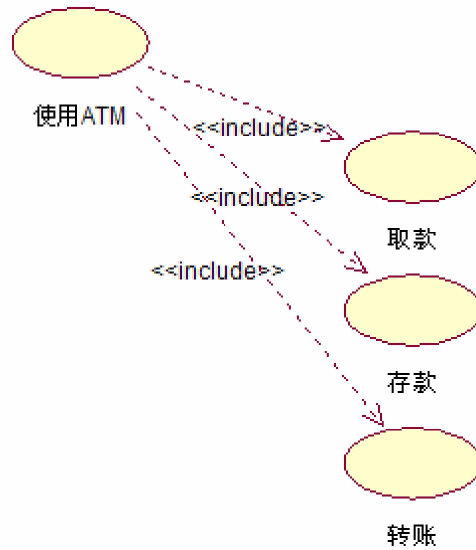

图 1、寇本画法[1]

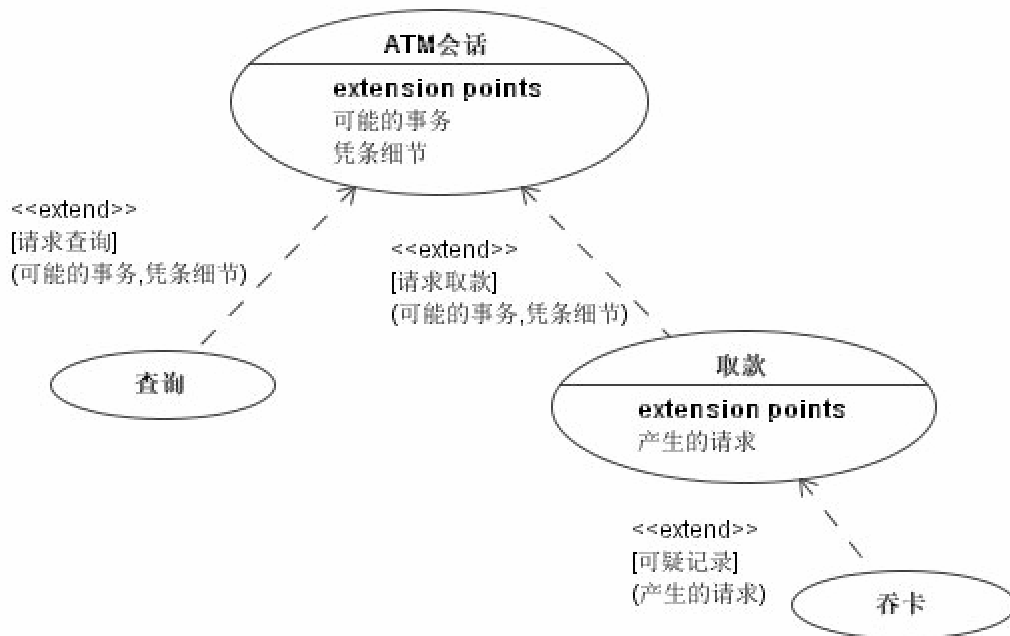

图 2、《UML 参考手册》画法[10]

UUCM:

为什么会有如此差别？为了说明问题，还是让我们先看看这几个用例的文本描述。《UML 参考手册》写道:

基用例 ATM 会话:

显示当天广告

include（识别顾客）
include（验证账户）
-------------------------- <可能的事务>（扩展点 1）
打印凭条标题
-------------------------- <凭条细节>（扩展点 2）
登出

扩展用例取款:
收到取款请求
指定取款数量
-------------------------- <产生的请求>（扩展点 3）
付款

看到这里，我们就不难解释为什么图 2 把 ATM 会话与取款之间的关系画成扩展了（注：这本身可能不符合亚克申扩展用例的原则[4]）。由于在身份验证通过后，顾客具体执行什么操作是不定的，可以是取款、存款、转账、查询等事务中的任何一个，将来甚至还有可能添加（扩展）新的功能，而 ATM 打印的凭条根据用户操作的不同，内容也将有所不同。这些内容的发生都需要具备一定的条件，属于可能的情况，这一点似乎很符合扩展用例的定义。考虑到这些因素，作者于是采用了图 2 的策略，把可能的操作和打印内容都从基用例中抽走。

然而不要忘了，除了扩展用例只有在特定条件下才能被触发之外，把一段内容提取为扩展用例还要求即使在没有该扩展用例插入的情况下，基用例本身的执行也不应受到任何影响。因此，如果我们把上段文字中的两个扩展点拿掉（只看左边的文字）并参照寇本目标用例的标准，可以发现该基用例完全不是一个目标得以完整执行的用例。上述写法是有问题的，它不完整，而且效率不高，似乎模仿了一些编程语言的写法和思路，在实践中不值得推广。

下面再让我们看看寇本版的“使用 ATM”[1]:

1. 顾客插卡，输入 PIN。
2. ATM 验证顾客账户和 PIN。
3. 顾客执行下列任一事务:
   取款
   存款
   转账
   查询
  顾客执行以上事务直到选择退出。
4. ATM 退卡。

在这里，取款、存款等事务操作明确地出现在用例的基本流中，显然它们都是被“使用 ATM”所包含的用例。注意，在寇本用例中每个动作步骤不一定都是顺序执行的，可以有循环，有选择，甚至可以是任意顺序，例如上面的步骤 3。在 WEUC 中寇本还指出，实际上取款、存款、转账等操作都是顾客执行一次交易事务的特例和具体化，所以还可以进一步画成类似图 3 的形式。我们可以看到图 1 其实是图 3 的简化。

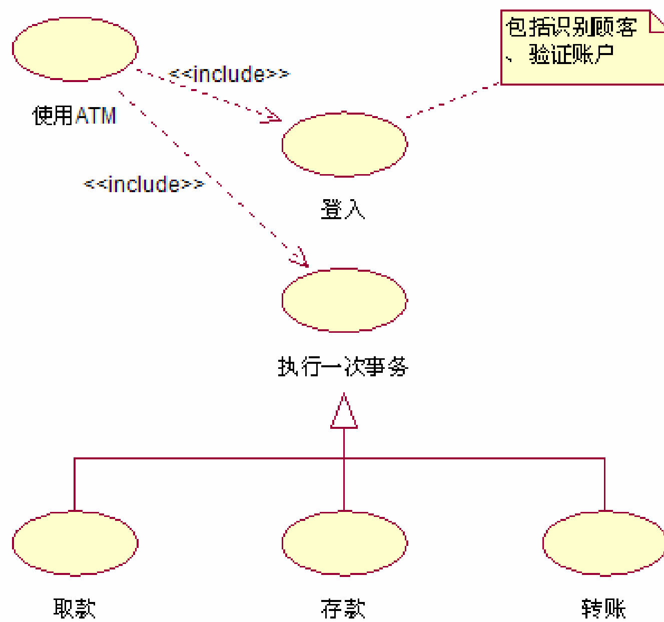

图 3、更为准确的画法

应该如何判定一个用例关系是包含关系还是扩展关系？有些情况还是比较明确的，例如对于“验证身份”和“吞卡”，大家可能都毫无异议，一致认为它们分别是“使用 ATM”的包含用例和扩展用例。但是，一旦出现上例的情况时，就较难处理了。不过，我们通常可以据此判定：如果触发条件中含有基用例负责的事物，即基用例知道附加用例何时、何处、为什么发生，那么基用例应该包含（调用或引用）它；如果触发条件中含有附加用例负责的事物，即附加用例知道它应该何时、何处、为什么发生，那么应该让附加用例扩展基用例[1]。这两条规则还是很管用的。此外，根据笔者经验，还可以参考这样一条简单法则：**凡是在用例基本流中出现的附加用例都应作为包含用例，而在扩展流中出现的附加用例必然是扩展用例。**显然，图 3 符合以上规则。

问题似乎得以解决了。然而，在 UML 2.0 上层结构规范草案（www.uml.org）中，我们却看到了类似图 4 的方案：

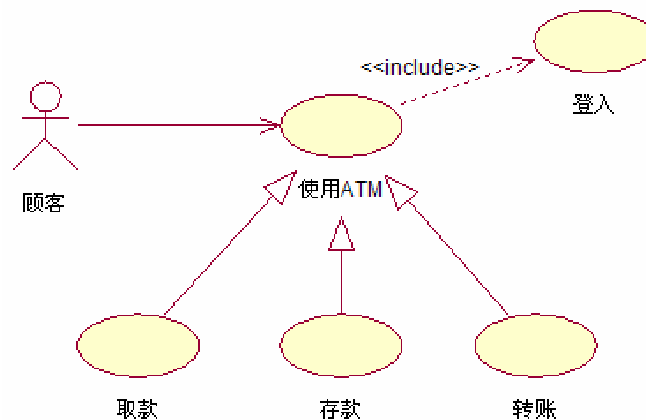

图 4、另一种近似的画法

有人要问了，为什么针对同一段用例文本描述，却存在两种矛盾的 UML 画法，同一个用例“取款”在图 3 中是包含，在图 4 中却是继承，到底哪个是对的？其实，图 4 与图 3 这两种用例模型在整体上所描述的内容是完全一致的，问题就出在用例名称上。分析、比较用例之间的关系，请务必关注用例的内容——动作步骤。我们可以发现，图 4 中的“取款”用例与图 3 中的“取款”用例所包含的实质内容是不一样的，一大，一小。所以，假使我们把它们的名称分别改为“取款交易”和“取款（事务）操作”，并约定前者“大于”后者，就可以在一定程度上消除二义性，这样两个图就近似等价了。不管如何，图 3、图 4 都一致地没有把“取款”、“存款”、“转账”等用例作为基用例的扩展。笔者认为，经过改进，这两种画法目前都是可行的，而图 3 可能更好一些。

关于用例包含、扩展和继承关系方面的更深层探讨，值得写一篇专述文章，在这里不可能展开。在目前众说纷纭的情况下，建议大家像寇本建议的那样，如果实在搞不清两个用例到底是什么关系，不如干脆都先定为包含关系，这样做并不影响后续的系统分析设计和使用，因为**用例本身的内容写得如何才是问题的实质、真正的关键**。总之，**实践者不要指望只通过看几个用例的名字就能准确地判断出它们之间的关系。可靠的依据来自于用例内部，不必在弄清用例的关系上浪费太多时间**，这个问题可以留待学者和研究人员来解决。

2.4.2 sub use case 之争

寇本在 WEUC 中把包含用例称为 sub use case，即 subordinate use case 之意，但其实 subordinate use case（笔者将其译为“分用例”）在亚克申方法中有专门的用途（在亚克申博士早就提出的对大规模系统建模的 SystemOf InterconnectedSystems 模式中，有超系统/超用例和分系统/分用例之分[4]）。而寇本采用的 sub use case 叫法很容易被直译成中文“子用例”，从而造成新的误解，因为我们知道“父子”一说在 OO 方法中通常是用来形容对象类之间的继承关系的，子类是父类的派生类，显然这与包含用例的真实语义不符。

在 UML[10]中，把包含用例、扩展用例统称为附加（additional）用例，被包含、被扩展的用例叫做基用例，在用例的继承关系中则采用父用例、子用例的说法，这样做是妥当的。

UUCM:

建议在实践中尽量回避寇本 sub use case 的说法，可以用“包含用例”或“附加用例”来代称，同时明确约定在中文中凡是提到“子用例”的地方，就是指用例的继承或一般化。

2.5 Actor

寇本用例的 Actor 类型有 7 种之多，包括[1]:

Actor（something with behavior）、

External actor（an actor outside SuD）、

Stakeholder（an external actor entitled to have its interests protected by the system）、

Primary actor（a stakeholder who requests that the system deliver a goal）、

Supporting/secondary actor（a system against which the SuD has a goal）、

Offstage/tertiary actor（a stakeholder who is not the PA）、

Internal actor（either the SuD, a subsystem or an active component of the SuD）。

RUP关于Actor的定义[7]是: Someone or something, outside the system or business that interacts with the system or business。UML的定义是: An actor specifies a role played by a user or any other system that interacts with the subject。

UUCM：

亚克申用例与寇本用例对于 Actor 的定义存在着明显区别。亚克申及 UML 的 **Actor 是系统之外的人或物**，而寇本 Actor 的含义（可能来自早期的亚克申版本）比较笼统，其范围要大得多，有内外 Actor 之分。既然我们确定 Actor 的主要目的之一是为了划定系统（或业务）的边界，那么应该始终把 Actor 当作系统外部的事物，这样才比较妥当，内部 Actor 是不必要的概念。而且，在实际应用中 Actor 即重要也不重要[1]，识别 Actor 的主要目的是为了帮助提取用例，通常不需要对各种 Actor 类型加以如此细致的区分。

另外，寇本关于 stakeholder 的定义与通常的理解也有所差别。寇本认为 stakeholder 是一种外部 Actor（注：这本身与寇本自己对用例的定义明显存在矛盾[1]），这不对。按常理，所有利益相关人包括内部 Actor、外部 Actor 都应该属于 stakeholder 之列，尤其当我们讨论的是业务组织的时候。

2.6 后置条件

与亚克申用例相比，寇本对用例的后置条件进行了细分，提出了最小保证、成功保证等概念，这样做是有意义的。最小保证描述了系统不管在任何情况下，尤其当用例失败、目标未达成时，都应满足的起码条件和应采取的措施。

UUCM:

建议采纳寇本方法的最小保证和成功保证划分。此外，在前置、后置条件（最小保证和成功保证）中除了说明必须满足的条件外，还可以分别说明系统在用例开始前和结束后的状态，包括各种成功和失败状态以及对失败状态的处理。

2.7 动作步骤

寇本用例步骤采用独特的编号方式，基本流采用 1,2,3...顺序编号，扩展流的条件和扩展步骤采用数字、字母间隔的方式，如 1a、1a1、5c、5c3b1 等等，而且还可以使用宏代符*，可以指定任意数目步骤的条件，如 1-9a、2,7-9c 等等，使用起来非常方便。RUP 的基本流、扩展流步骤完全采用自然分节的顺序编号，如 1.1、2.3.4.1 等等，不便于阅读者找到用例具体的引用位置，在指定扩展位置时显得较为麻烦。

另一方面，RUP 用例的每个步骤都可以附上一个名称，这叫做“命名步骤”。

如果一个步骤内容较多，用一个短语标记来概括说明该步骤执行的大致内容，确实比较方便，而且将来需要画用例的活动图时也可以作为快速参考。

UUCM：

建议采纳寇本的步骤编号方式和 RUP 的命名步骤方法。

2.8 文本与 UML

亚克申用例方法与 UML、RUP 三者之间有着天然的紧密联系。用例驱动、可视化建模是 RUP 的两大特征，若用例和 UML 缺席，RUP 也就称不上 RUP 了。在亚克申方法中，除了可以用 RUP 的格式文本描述用例外，还推荐适当地选择利用 UML 用例图、序列图、协作图、活动图和状态图 5 种图示从各个角度来描述用例，可谓手段充足、武器齐备。

寇本代表作 WEUC 的主基调是用结构化/半结构化的文本描述用例。虽然他也提到了 UML，但讨论主要集中在如何正确对待 UML 的用例图和相关 CASE 工具等问题上，对 UML 用于描述动态行为的其他 4 种图的作用和意义强调得不够充分，其实这些图的作用远比用例图要重要得多，也是 UML 强大描述能力之体现。

UUCM:

如前所述，文本用例本质上是对对象交互过程的执行步骤的罗列。用例本身即每个“椭圆”内部的东西才是最为关键和重要的，用例之间的关系相对来说要次要些，在这一点上，可以说亚克申方法和寇本方法的看法是一致的[1][4]。实践中，我们发现很多时候，先写文本系统用例，后照着已有的文字说明画 UML 图比先画图或完全依赖于图形来描述系统用例要容易得多。可见对于软件需求分析，我们应该首先把更多的精力放在写好文本用例上（这正是寇本方法的强项）。

但是，寇本认为“Sequence diagrams are not a good form for expression use cases”[1]，这种说法有失准确和全面。纵观全书，寇本主要是从工具使用的角度来分析的，他在书中对当时那些不那么先进的 CASE 工具颇有微词，认为它们不如文本写作更加便捷和有效。然而，事实上这只是问题的一个方面，UML 工具的缺陷并不能简单地等同于 UML 语言本身的缺陷。正确地使用 UML 及其工具不仅仅是为了获得形象直观、交叉引用、超链接、名称自动变更等一目了然的好处，更重要的一个原因是，通过合适的 UML 图形，比如 SSD（系统序列图），来精确地定义和描述系统事件与作为其响应的系统操作（也就是系统的输入和输出）[8]之间的契约，这正是后续系统分析和设计的起点。序列图在用例分析中其实起到相当关键的作用，在实践中用它来刻画每一个重要的系统用例也是非常普遍的做法。

而且，在分析复杂的业务流程/业务用例时，人们好像更习惯于首先画活动图（可能与人类自身的思维习惯有关），反而不太愿意采用繁琐的文字说明。对于复杂和关键的用例，除了一些必要的文本描述之外，再辅之以 UML 活动图、序列图或状态图进行可视化，是行之有效的好做法，有助于澄清问题本质、迅速抓住要领。对于复杂的用例模型，通过用例图描述用例之间的关系，提供全局浏览视图，也是非常必要的。

用例的 UML 图形与文本描述之间不是谁取代谁的关系，而是相辅相成、优势互补，应该因地制宜地加以运用。不仅如此，同时拥有用例的结构/半结构化文

本和 UML 图形，往往还有助于彼此之间相互比对、确认，能显著提高用例描述和分析的正确性。根据本人经验，把两者结合起来运用效果才是最好的，没有必要过份地强调某一方面。

2.9 黑盒白盒

亚克申博士发明的用例实现（UCR，Use Case Realization）[5]在 RUP、UML 中是一个非常重要的概念，它描述了内部对象如何相互协作共同实现一个用例，理论上每一个用例都应该至少有一个 UCR 与其对应，因而 UCR 在亚克申方法中起到了联接问题域和解决域、贯穿整个软件分析设计过程的关键作用。在寇本方法中，不存在 UCR 这个术语，只有黑盒用例（需求）、白盒用例之分（需求的实现）。

UUCM:

首先，既然谈及基于用例表示的功能需求，那么它就应该是黑盒的、透明的。如果我们看到了系统内部的情况（白盒、不透明），那么这其实已经是需求的一种实现了。所以，我们应该**在需求分析时尽量避免“白盒用例”这种矛盾的说法**。严格地区分 UC 和 UCR，有助于项目团队在实践中消除需求和实现不分的情况。这一点过去在被拉来写需求的程序员当中比较普遍，受习惯性思维的影响，他们往往写到最后就变成写软件设计方案了，这是很糟糕的。

2.10 格式模板

寇本在 WEUC 中一共列出了 5 种主要的用例格式模板：完整型、简易型、单列表式、双列表式和 RUP 样式。在此，我们推荐以完整型（寇本本人最喜欢的）为基础，结合了单列表式和 RUP 样式特点的 UUCM 模板：

<table>
<tr><td colspan="2">用例名称：</td><td></td><td>层次：</td><td>+|!|-</td><td>范围：</td><td></td></tr>
<tr><td colspan="2">简述/背景：</td><td colspan="5"></td></tr>
<tr><td colspan="2">主使用者及利益：</td><td colspan="5"></td></tr>
<tr><td colspan="2">其他受益人及利益：</td><td colspan="5">受益人 1:
受益人 2:</td></tr>
<tr><td rowspan="2">后置条件</td><td>最小保证：</td><td colspan="5">状态 1:
状态 2:</td></tr>
<tr><td>成功保证：</td><td colspan="5"></td></tr>
<tr><td colspan="2">前置条件：</td><td colspan="5">条件 1:
条件 2:
状态 1:
状态 2:</td></tr>
<tr><td colspan="2">触发事件：</td><td colspan="5"></td></tr>
<tr><td colspan="2">基本流：</td><td colspan="5">1. 步骤 1
2. <可选名称>. 步骤 2
…
n. 步骤 n</td></tr>
</table>

	<结束>
扩展流：	1a. 条件 1: 　　1a1. 步骤 1 　　1a2. 步骤 2
扩展点：	名称 1: 位置 1 名称 2: 位置 2
技术和数据变化：	1a. 2a.
非功能需求：	
业务规则：	
备注：	
其他必要字段 ……	

表 1、UUCM 模板 v1.0

双列表式较多地被用来描述用户界面需求，有些人偏爱它，但我们发现双列表不够简洁，比较占空间，而且很多情况并不适用，比如参与者多于两个的情况。

三、结语

寇本用例方法以基于目标的结构化/半结构化文本描述见长，亚克申用例方法更重视 UML 可视化建模和用例驱动过程。两者尽管同宗同源[1]，却在一些使用细节上存在着明显差异，而且各自还在沿着既有轨道继续向前发展。

本文提出的统一用例方法并不是一种全新的方法，UUCM 仅仅是个符号或代称（也许可以有其他更好的名称），它实质上代表了一种解决方案和思路，目的是吸收亚克申和寇本用例方法的长处，消除两者的不一致，从而帮助实践者尽可能规避使用上的误区，发挥用例和 UML 方法“1+1〉2”的联合优势。这既是实践的选择，也是现实的需要。

话说天下 IT 之势，合久必分，分久必合。UUCM 不是句号，而是一个新的起点。出于商业、私人或其他方面的原因，国内外许多技术流派分呈的局面会长久持续下去，这本身是一件好事。不过，对于我们实践者来说，选择实用的工程技术时不应有门派分割的障碍。对于特定的场合、特定的项目，企业人往往只有一个明智的选择，那就是：用最小的成本创造最大的客户价值。所以，对于实用型技术我们完全可以采取拿来主义的态度，防止 guru-locked-in，适用的即是最好的！

致谢

感谢邱嘉文（从他这里我学到了“胚”这个词）、潘加宇、沈备军博士在百忙之中审阅了全文，并提出了宝贵的改进意见！

参考文献

[1] Cockburn A.著,《编写有效用例》(英文版),机械工业出版社,2002 年 7 月。
[2] Cockburn A., *Using goal-based use cases*, JOOP Vol. 10, No. 6 (Sep-Oct), No. 7 (Nov-Dec), 1997.
[3] Jacobson I.等著, *Object-oriented Software Engineering: A Use Case Driven Approach*(修订版、英文版),人民邮电出版社,2003 年 9 月。
[4] Jacobson I.著,程宾等译,《统一软件开发过程之路》,机械工业出版社,2003 年 8 月。
[5] Jacobson I.等著,周伯生等译,《统一软件开发过程》,机械工业出版社,2002 年 1 月。
[6] Hurlbut R., *A Survey of Approaches For Describing and Formalizing Use Cases*, Expertech, Ltd., 1997.
[7] Kruchten P., *The Rational Unified Process An Introduction, Second Edition*, Addison Wesley, 2000.
[8] Larman C.著,姚淑珍等译,《UML 和模式应用:面向对象分析与设计导论》机械工业出版社,2002 年 1 月。
[9] Leffingwell D.等著,蒋慧等译,《软件需求管理:统一方法》,机械工业出版社,2002 年 3 月。
[10] Rumbaugh J.等著,姚淑珍等译,《UML 参考手册》,2001 年 1 月。
[11] 邵维忠等著,《面向对象的系统设计》,清华大学出版社,2003 年 2 月。
[12] 张恂,《让历史告诉未来:评 Ivar Jacobson 博士的〈用例的昨天、今天和明天〉》,IT 之源,2003 年 4 月。
[13] 张恂,《如何写好最外层用例》,IT 之源,2003 年 3 月。

感谢您阅读此文!本文著作权所有人为张恂,保留所有权利;您可以从 www.zhangxun.com、**IT 之源**、UMLChina 上获得本文的最新版本和相关资料;如对本文存在的错误、缺陷和不完善之处有任何意见或建议,请发信到 zhangxun2001@hotmail.com;以上言论仅代表作者本人观点,与作者服务的公司无关;欢迎转载本文电子版,转载时请注明出处并保留所有原始信息;纸质媒体如需转载请与作者联系。