第 24 期（2003 年 4 月）

Ivar Jacobson

X 非程序员 -Programmer

软 件 以 用 为 本

投稿：editor@umlchina.com
反馈：think@umlchina.com

http://www.umlchina.com/

数据建模工具与 BPM 日趋融合

[2003/2/21]

企业建模并非易事，这并不奇怪。首先，现在的企业业务更加复杂，而管理企业的人却被要求在很少的条件下做很多事情。然而，企业建模工作困难还有另一个原因，这就是有许多不同的模型都可以用来描述企业系统。

在这些模型之中，近些年来为大家关注的主要是数据设计模型、对象模型、系统模型和业务流程模型。

然而，模型的种类如此繁多却不一定是件好事。当数据设计者、业务策划人和系统设计者使用完全不同的语言讨论同样的问题时，使用技术实现业务目标就会陷入混乱。

虽然 UML 并不适合于任何的建模问题，但是 UML 得到日益广泛的接受，并给建模工具领域带来了更多的凝聚力。通向不同工具高度结合的道路可能刚刚开始，未来还不明朗，但是有迹象表明，业务模型和数据模型正在越走越近。期望“以少做多”正在逐渐影响数据建模工具市场，业务流程建摸又重新成为其关注焦点。

不同的是，那些新的建模工具强调的是不同的方面，而这些方面今后可能会成为一个问题。Sybase 的 PowerDesigner 9.5 和 Quest Software 的 QDesigner 可以支持 UML 的九种图和元数据集成。Embarcadero 公司已经开始提供 ER/Studio 数据设计工具与 Describe 和 DT/Studio 应用建模工具的多种集成。Computer Associates 公司的 AllFusion Erwin 数据建模工具 4.1 版本具备管理多种模型的能力，并允许不同的数据库管理员和开发者基于同样的模型开展工作。

所有新的数据建模工具的共同之处是更侧重于业务流程建模。

为什么会有这种变化呢？当前企业业务的发展要求在能够提高效率的同时尽可能的降低成本。正如纽约 Popkin 软件公司的咨询服务部门主管 Chuck Faris 所说，“我们的目标是增加资产，同时节约时间，提高价值”。

寻求降低成本，提高效率的企业组织正在努力理解其业务流程，主要是一个“孰重孰轻”的问题。而理解这些流程的最好方式是为其建立模型。

Faris 说，“人们正在开始进行实际的数据设计流程，而不是正在下决定。有些人起步甚至比别人更早”，人们从企业模型的角度来谋求问题的解决。各个公司都在努力寻找提高整体竞争力和全局管理能力的途径。

从一开始，Popkin 的系统架构工具就把数据和结构化的业务流程集成在一起。现在，它也可以支持 UML。那些与系统架构相关的业务流程模型通过一个通用的、多用户的数据库共享数据，这意味着数据建模工具也可以作为业务流程建模工具来应用，反之亦然。

“业务流程带来数据集成的观点，” Faris 解释说，“基于业务效率的考虑，从业务流程出发的思路改变了数据模型。即使你能设计一个十分完美的数据模型，但只有当你从业务流程的角度来看待数据的时候，你才算是完成了这个模型。”

企业应用的有力支撑

其他的数据建模工具供应商早已开始支持业务流程建模。

例如旧金山的 Embarcadero 公司就声称早在 1996 年，它就开始把其 ER/Studio 产品作为一个业务元数据建模和物理数据库设计工具了。设计和建模解决方案负责人 Greg Keller 说，“当我们开始做的时候，这种工具传统上是纯粹的数据设计驱动的”。而让 ER/Studio 不同的就是，它能够在一个工具中提供两个方面。从 Describe 产品问世以来，Embarcadero 公司已经在跨企业开发团队的建模应用方面更进一步。Describe 在 2000 年 10 月问世，Embarcadero 将它的数据库设计软件和通过收购 Active 软件公司得到的 GD Pro 软件工程工具集成在一起。

ER/Studio 覆盖了一个企业数据的概念设计和逻辑设计。Keller 提到，“对一个数据库的元数据模型来说，它可以被分成不同的格式。” 用户可以用一种方式创建业务模型来存储数据，然后将其付诸实施。

CA 公司已经以其著名产品 Erwin 数据设计师为核心，开发了一系列建模工具套件。过去，“人们进行数据建模，但是不知道如何把它与业务流程相结合。” AllFusion 的品牌经理 Greg Clancy 说，在其负责的品牌下，ERwin 和其他的 CA 建模工具已经捆绑在一起。这些工具是 AllFusion 建模套件的一部分，它还包括一个数据模型验证工具、一个模型管理器和一个组件建模工具。Clancy 说 Erwin 通过其与 AllFusion 流程建模工具的集成来支持业务流程建模。虽然这种工具可以分开销售、独立使用，Clancy 还是强调应当把这些工具作为一个整体来看待。

数据建模的领先企业 Sybase 继续为其 PowerDesigner 产品不断升级。并且业务建模是其主要侧重点。他们通过业务流程模型从而在 PowerDesigner9.0 版中增加了业务流程建模能力。高级产品经理 David Dichmann 强调业务流程模型允许你在控制型图表中使用非技术业务术语。

Sybase 认为 IT 项目的失败通常并不是有 bug 的代码造成的，而是由于技术与公司目标或者业务运作方式不符造成的。这就是业务流程模型背后的推动力。有了它，Dichmann 说“信息技术就可以极方便的与企业业务进行有效和高效的交流。它为业务自身提供了再工程的方法。”这是如何完成的？业务流程模型帮助用户理解其业务究竟做什么，并帮助他们形成流程文档和优化流程，从而通过业务流程自动化确保业务取得成功。

Quest 软件公司的 QDesigner 除了某些调整之外，在很大程度上是和 Sybase 的 PowerDesigner 一样的产品。QDesigner 的产品市场经理 Darin Pendergraft 解释道，“允许人们以可视化的方式支持业务工作和功能，以及他们如何进行整合。”如果一个用户正在浏览模型应用，他只会把焦点集中在这个应用上。然而，如果一个人正在关注模型业务，那他就可以覆盖多个应用。Pendergraft 还指出，“业务流程模型帮助人们看到即将用到什么东西，它是如何有利于业务的以及该业务还有哪些其他方面的需求”。

为迎接黄金时代而准备

任何转变都需要时间，但是纽约的哥伦比亚大学的 Arthur Langer（信息技术与继续教育指导和课程发展计划主席）指出向业务流程建模转变的周期并不像多数人认为的那样长。“人们往往花费的更多的时间去想，而不是去做。”他提醒到。

一些 Langer 的从前学生请他去评估纽约的 13 频道 WNET TV 的信息系统的结构。在那里，与 WNET 的 CTO Ken Devine 一道，Langer 意识到 WNET 的未来是结合 IT 管理和工程技术。

WNET 是美国公共广播公司的最大节目提供商之一，如果让 WNET 同时跟踪 30 到 40 个生产单位往往需要很大花费。“我们从来没有从通常的帐面上得到过我们的利润。” WNET 的 Devine 说。

有多年使用 System Architect 工具的经验和工具应用教学经验的 Langer 带领 WNET 完成了提速过程。在他的帮助下，电视台能够在 Devine 的部门中通过结合若干不同的日程表和数据库实施进度跟踪。

“由于我们能够转移数据，并利用数据完成工作，我们必然将进入一个新的天地，我们可以从我们完成业务的方式中获得更多的回报。” Devine 强调到。

WNET 向业务流程建模的成功转换的关键在于获得对数据的控制。“Langer 告诉我们，控制数据、审查数据和转移数据是非常必要的工作。” Devine 解释道，“我从来都不知道这里面还存在一个工程的问题”。

在使用 System Architect 工具之前，WNET 的网络环境是在 Novell 网上运行的 Groupwise。“技术的能力和集成都不够，” Langer 提到，电视台有许多苹果公司生产的机器和其它特殊设备。“但是这些东西的一个核心就是数据，”Langer 说，“能够在所有的数据需求部门之上集成数据是非常必要的”。

在过去的两年里，电视台在其数据查询中看到了更多的效率。“这是 PBS 系统中唯一一个有所成长的电视台。” Devine 提到，“系统中的其他人正在走下坡路，我们相信我们成长的一个理由（并且我们的捐助者们也相信）是挖掘数据的能力；现在，他们处理数据的技术越来越成熟，这将会让他们的业务更为成功。”

System Architect 工具使 WNET 做到数据库逆向工程、工作流和数据图表化、前向工程和数据库设计。这个电视台还使用 System Architect 创建了一个互联系统，允许它使用不同的名字，如“customer”和“cust”，在多个系统中具有多个数据域。

Devine 指出:“业务流程建摸是使我们成长并向前发展的绝对关键”。Langer 把业务流程建模作为基础来看待,他认为“即使你其它的活动都取得了成功,但是如果你没有业务流程工具的话,所有的成功都将毁于一旦。”

位于亚特兰大的国际技术咨询公司也在寻求一个综合的工具解决方案来支持组织内部的技术人员和非技术人员共同协作去创建系统。其公司总裁 Donald Clayton 说:“PowerDesigner 9.5 软件提供了这些能力,并且能够在多种平台上支持 UML 图,也就是说国际技术公司不需要购买附加的未集成的工具。”

医疗数据服务中的应用

将数据建模和业务过程建模相联系对于医疗数据服务来说是很重要的。负责医疗护理行业垂直市场应用开发的 CTO,Jeff Moyers 说,“将数据建模与业务流程建模相结合对于医疗数据服务而言是非常重要的。为了开发应用去解决业务问题,你首先应当了解问题就是什么?”

他说:“理解问题最简单的方法就是对问题建模。”

医疗数据服务使用 Embarcadero 公司的 ER/Studio 工具来建立模型并为应用构造不同的数据库。其组织结构使用三个主要的、非常大的模型来划分其工作的主要方面。Moyers 说,他是依靠类似 ER/Studio 的建模工具给他可以工作的结构并使他更容易的管理数据库之间的关系和许多工作的顺序。

通过比较多种产品,医疗数据服务选择了 ER/Studio,因为其用户界面和逻辑表达层更加出色。此外,公司发现这个工具易于使用。Moyers 说,“其它工具的功能没有如此强大,或者其它工具的功能即使强大,和 ER/Studio 比起来,界面也不友好。在效率与易用性之间进行权衡,才选择了 ER/Studio。”

那么 Moyers 究竟需要功能多么强大的建模工具呢?“我需要它可以在任何时间去做我想做的任何事情”,他说,“只有到你需要的时候,你才发现你没有足够的能力。能满足你所需要的就是最好的。”

Moyers 需要的是能够建立大型的模型,能够方便的检查这些模型,容易的重构数据库,能够以最小的成本和努力尽可能快的、尽可能好的去实现这些模型。

Moyers 相信,方便的浏览一个结构是 ER/Studio 的闪光点之一。“当你处理数据库中的复杂关系时,非常关键的是你能够方便的、容易的、快速的浏览模型,并到达你想看的结构。”,他说。Moyers 也非常欣赏 ER/Studio 逻辑模型与物理模型的分离、模型的全局可视化表示以及其海量数据建模的能力。

尽管医疗数据服务认识到业务流程建摸的重要性,但是他们没有依赖工具去进行业务流程建摸。Moyers 解释道,这是因为他的公司是一个小公司。“这将花费时间去准备、花费时间去实现并且要让其正确的运行也需要时间。如果我投入时间,我确信我将获得好的回报,但是我现在没有时间投入到上面。”

显然,将业务流程与数据建模相结合是极具潜力的发展方向。

(自 adtmag,June 摘译,不得转载用于商业用途)

业内第一个全面的用例开发平台

[2003/2/29]

SteelTrace 将 Catalyze 与 Rational 产品集成，两者的集成为业内提供了第一个全面的用例开发平台。

SteelTrace 公司 2003 年 2 月 20 日宣布，该公司用例驱动的需求及过程获取软件 Catalyze，和 Rational 公司市场领先的需求管理工具 RequisitePro 实现了集成，新集成后的 Rational RequisitePro 产品，结合早期发布的 Rational Rose，为业内提供了针对用例开发的全面解决方案。

集成后的解决方案允许在 SteelTrace Catalyze、Rational Rose 和 Rational RequisitePro 三个工具之间以增量迭代的方式实现信息同步。SteelTrace Catalyze 工具通过图形或文本的方式来描述一个用例的事件流、场景、步骤及相关用例模型的细节，来开发一个全面而完整的用例需求模型。Catalyze 中的用例能够与 Rational RequisitePro 中的用例保持同步，作为一个正式的需求进行管理。而 Rational RequisitePro 工具通过跟踪需求的变化、属性及其它需求间的联系来管理这些需求。集成后的产品在用例需求文本描述（Rational RequisitePro 工具）、用例需求模型（SteelTrace Catalyze 工具）、UML 可视化模型（Rational Rose 工具）之间提供了一个语义上沟通的桥梁，用例行为被转换为对交互对象的描述并最终生成一个可以实现的应用。

“集成后的产品具备了市场所需求的完整的用例需求开发能力”，SteelTrace 的 CEO Mark Melville 说，“通过这个完整的应用解决方案，组织可以弥补在商业和技术概念方面的鸿沟，保证开发出一个满足用户商业需求的、成功的应用解决方案。”

“Ctalyze、Rational RequisitePro 和 Rational Rose 三个工具的集成，为 Rational 用户提供了一个强有力的用例驱动开发解决方案”，Rational 软件公司负责需求管理解决方案的主管 Kurt Bittner 说。“这个组合在一起的解决方案为业内提供了唯一一个使用用例来驱动项目成功的全面需求管理平台。”

关于 SeelTrace 公司

SteelTrace (http://www.steeltrace.com)，通过开发商业过程和需求工具，来保证用户需求的准确和系统能够按时交付。公司的两个主要产品 Catalyze(TM) Professional 和 Catalyze(TM) Enterprise 可以用来帮助商业过程工程师、项目经理、商业分析人员及系统分析人员获取，查看和管理用户需求和系统定义。公司成立于 2001 年 2 月，总部位于爱尔兰的首都都柏林，在英国的伦敦、美国的华盛顿和旧金山设有分支机构。SteelTrace 公司是一个通过网络进行管理（netdecisions）的公司集团。

（卓锐 摘译，不得转载用于商业用途）

面向服务架构（SOA）的原则

[2003/3/20]

Web service 已经不再是新婚的娘子。众多企业都已经创建各种实验性 Web Services 项目，事实证明，这项新兴的分布式计算技术确实能够降低集成和开发的成本。另外，一些关键的 Web Services 标准纷纷制定，强安全（robust security）和管理方面的产品也陆续问世。对于志向远大的企业来说，他们已经在考虑下一步了。

对大多数公司来说，下一步要考虑的不再是点对点的应用，而是 Web services 在企业间以及业务伙伴间更为宽广的应用。这种技术的变迁需要更松散耦合、面向基于标准的服务的架构。这样一个架构要求对 IT 在组织中的角色有新的观点和认识，而不仅仅是一种实现方法。通过对业务的敏捷反应，企业可以得到实实在在的回报，而要达到这一点，面向服务架构设计师的角色非常关键。除此之外，潜在的回报更是不可胜数－分布计算技术能够保证对业务需求足够灵活的反应，而这种业务上的敏捷正是各公司梦寐以求而目前还遥不可及的。

分布式计算将网络上分布的软件资源看作是各种服务。面向服务架构是一种不错的解决方案。但这种架构不是什么新思想；CORBA 和 DCOM 就很类似，但是，这些过去的面向服务架构都受到一些难题的困扰：首先，它们是紧密耦合的，这就意味着如分布计算连接的两端都必须遵循同样 API 的约束。打比方说，如果一个 COM 对象的代码有了更改，那么访问该对象的代码也必须作出相应更改。其二，这些面向服务架构受到厂商的约束。Microsoft 控制 DCOM 自不必说，CORBA 也只是一个伪装的标准化努力，事实上，实现一个 CORBA 架构，经常都是在某个厂商对规范的实现上进行工作。

Web services 是在改进 DCOM 和 CORBA 缺点上的努力。今天应用 Web services 的面向服务架构与过去不同的特点就在于它们是基于标准以及松散耦合的。广泛接受的标准（如 XML 和 SOAP）提供了在各不同厂商解决方案之间的交互性。而松散耦合将分布计算中的参与者隔离开来，交互两边某一方的改动并不会影响到另一方。这两者的结合意味着公司可以实现某些 Web services 而不用对使用这些 Web services 的客户端的知识有任何了解。我们将这种基于标准的、松散耦合的面向服务的架构简称为 SOA。

SOA 的强大和灵活性将给企业带来巨大的好处。如果某组织将其 IT 架构抽象出来，将其功能以粗粒度的服务形式表示出来，每种服务都清晰地表示其业务价值，那么，这些服务的顾客（可能在公司内部，也可能是公司的某个业务伙伴）就可以得到这些服务，而不必考虑其后台实现的具体技术。更进一步，如果顾客能够发现并绑定可用的服务，那么在这些服务背后的 IT 系统能够提供更大的灵活性。

但是，要得到种强大和灵活性，需要有一种实现架构的新方法，这是一项艰巨的任务。企业架构设计师必须要变成“面向服务的架构设计师”，不仅要理解 SOA，还要理解 SOA 的实践。在架构实践和最后得到的架构结果之间的区别非常微妙，也非常关键。本文将讨论 SOA 的实践，即：面向架构的设计师在构建 SOA 时必须要做的事情。

SOA 的原则

SOA 是一种企业架构，因此，它是从企业的需求开始的。但是，SOA 和其它企业架构方法的不同之处在于 SOA 提供的业务敏捷性。业务敏捷性是指企业对变更快速和有效地进行响应、并且利用变更来得到竞争优势的能力。对架构设计师来说，创建一个业务敏捷的架构意味着创建这样一个 IT 架构，它可以满足当前还未知的业务需求。

要满足这种业务敏捷性，SOA 的实践必须遵循以下原则：

* 业务驱动服务，服务驱动技术

从本质上说，在抽象层次上，服务位于业务和技术中间。面向服务的架构设计师一方面必须理解在业务需求和可以提供的服务之间的动态关系，另一方面，同样要理解服务与提供这些服务的底层技术之间的关系。

* 业务敏捷是基本的业务需求

SOA 考虑的是下一个抽象层次：提供响应变化需求的能力是新的“元需求”，而不是处理一些业务上的固定不变的需求。从硬件系统而上的整个架构都必须满足业务敏捷的需求，因为，在 SOA 中任何的瓶颈都会影响到整个 IT 环境的灵活性。

* 一个成功的 SOA 总在变化之中

SOA 工作的场景，更象是一个活的生物体，而不是象传统所说的“盖一栋房子”。IT 环境唯一不变的就是变化，因此面向服务架构设计师的工作永远不会结束。对于习惯于盖房子的设计师来说，要转向设计一个活的生物体要求崭新的思维方式。如下文所写的，SOA 的基础还是一些类似的架构准则。

SOA 基础

在 IT 行业有两个越来越普遍的发展方向，一个是架构方面的，一个是方法学方面的，面向服务的架构设计师可以从中有所收获。第一个就是 MDA（模型驱动架构），由提出 CORBA 的 OMG 模型提出。MDA 认为架构设计师首先要对待创建的系统有一个形式化的 UML（也是由 OMG 提出）的模型。MDA 首先给出一个平台无关的模型来表示系统的功能需求和 use cases，根据系统搭建的平台，架构设计师可以由这个平台无关的模型得到平台相关的模型，这些平台相关模型足够详细，以至于可以用来直接生成需要的代码。

MDA 的核心就在于在设计阶段系统就已经完全描述，这样，在创建系统的时候，几乎就没有错误解释的可能，模型也就可以直接生成代码。但 MDA 有一些局限性：首先，MDA 假设在创建模型之前，业务需求已经全部描述，而这一点，在当前典型的动态业务环境中几乎是不可能的。第二，MDA 没有一个反馈机制。如果开发人员对模型有需要改动的地方，并没有提供给他们这么一个途径。

SOA 的另一个基础是敏捷方法（AM），其中非常有名的方法是极限编程（XP）。象 XP 这样的 AM 提供了在需求未知或者多变的环境中创建软件系统的过程。XP 要求在开发团队中要有一个用户代表，他帮助书写测试来指导开发人员的日常工作。开发团队中的所有成员都参与到设计之中，并且设计要尽量小并且非形式化。AM 的目标是仅仅创建用户想要的，而不是在一些形式化模型上耗费工作量。AM 的核心思想就在于其敏捷性－处理需求变更的敏捷性。AM 的主要弱点是其规模上的限制，例如，XP 在一个小团队和中型项目中效果不错，但是当项目规模增大时，如果没有一个一致的清晰的计划，项目成员很难把握项目中的方方面面。

从表面看来，MDA 和 AM 似乎是相对立的－MDA 假定需求是固定的，而 AM 恰恰相反。MDA 的中心是形式化的模型，而 AM 恰恰要避开它们。但是，我们还是决定冒险把这些不同方法中的一些元素提取出来，放入到一个一致的架构实践中。

在 SOA 中有三个抽象层次，按照 SOA 的第一条准则：业务驱动服务、服务驱动技术。AM 将业务模型直接和实践连接起来，表现在平台相关的模型之中。MDA 并没有把业务模型和平台无关模型分开来，而是把平台无关模型做为起点。SOA 必须连接这些模型，或者说抽象层次，得到单一的架构方法。我们将从五个视图的架构实现方法来实现这个连接。

SOA 的五视图实现方法

企业架构设计师发现他们的职业非常有竞争力并且值得骄傲，因为他们要从很多方面来通盘考虑 IT 系统。Kruchten（RUP 的开发负责人）将这些方面提取出来，在应用到 SOA 时，我们称为五视图实现方法（five-view approach）。

四个方框表示对一个架构的不同审视方法，分别代表不同的涉众（stakeholder）。弟五个视图，use-case 视图涵盖了其它视图，在架构中扮演的是一个特殊的角色。部署视图将软件映射到底层平台和相关硬件上，是系统部署人员对架构的视图；实现视图描述了软件代码的组织，是从开发人员角度出发的视图；业务分析人员则利用过程视图进行工作，它描述的是软件系统的运行时特性。最后，逻辑视图表示的是用户的功能需求。在 SOA 中，面向服务的架构必须能够以 use-case 视图中的用例将用户连接到服务，将服务连接到底层的技术。

为了表示面向对象的架构是如何工作在这些视图之上，让我们将他们置于 SOA 元模型的上下文之中。SOA 中两个领域存在重叠：由业务模型和服务模型表示的业务领域和由服务模型及平台相关模型表示的技术领域（两个领域共享服务模型）。业务用户通过逻辑视图和过程视图处理粗粒度的业务服务，根据变化的业务需求，按照需要将它们安排在过程之中。另一方面，技术专家的工作是创建并维护服务和地层技术之间的抽象层。表示这些服务的中间模型，起到的是轴心的作用，业务以它为中心进行。

SOA 元模型从 MDA 中继承平台无关模型和平台相关模型，但是添加了 AM 和用户交互以及敏捷的反馈这两部分，后者通过椭圆之间的双向箭头来表现。类似地，元模型通过引入由中心的服务模型提供的中间层抽象解决了 AM 在伸缩性方面的问题。这样，服务模型中的任何需求的变化，都会反映到用户每天的业务处理中。同样，由于底层技术是模型驱动的，技术专家也可以根据这些变化的需求迅速而有效地作出应变。

SOA 实践和过去解决企业架构传统方式的不同之处就在于其对敏捷性的支持。如前所说，SOA 的第三条原则就在于它总在变化之中。这种恒在的变化性环境是 SOA 实践的基石。如图所示，涉众（stakeholders，译者注：RUP 中也有这个词，表示软件开发中涉及到的各种角色如：用户、设计人员、开发人员乃至测试人员等等。）在一个必需的基础上影响到整个架构的变化。在当技术专家在每天的日常工作中不断对变化的业务需求作出响应的这种情况下，设计阶段和运行阶段之间的界限变得模糊起来，很难清晰地分离这两个阶段。

剩下的部分

我们已经为面向服务的架构提供了一个高层次的框架，其中 MDA 和 AM 的元素帮助工具的使用者来创建和维护 SOA。但是，SOA 中还缺少一些内容－那就是软件开发商和专业的服务组织必需提供的。理想情况下，开发商必需提供面向服务的业务流程、工作流以及服务的协调工具和服务；另外，能够以一种敏捷的、平台无关的方式充分反映业务服务的建模工具也是必须的；技术专家必须配备可以从模型中自动生成代码，并在代码变化时更新模型的工具，最后，开发商必须提供支持 SOA 的软件，帮助面向服务的架构设计师以一种可信并且可伸缩的方式创建位于服务和底层技术之间的抽象层次。幸运的是，这方面的产品即将上市。

另外，最重要的就是贯穿本文的自顶而下的 SOA 实现方法了。今天关于 Web services 的大部分思考都是自底而上的：“这是如何创建 Web services 的方法，现在，我们来使用它们集成吧”，对 Web services 技术的这种方法是伟大的第一步，因为它可以惊人地降低集成的开销，这是现在的技术管理人员最乐意见到的了。但当经济进一步发展，IT 走出低谷，企业会寻求 IT 的帮助来提高组织战略意义上的核心价值。使用面向服务的架构，IT 可以提供给企业实现业务敏捷性的这样一个框架。

（自 adtmag，风自由 摘译，不得转载用于商业用途）

短讯：Ivar Jacobson 即将离开 Rational。见 CSDN 新闻>>

UMLChina 公开课

“UML 应用实作细节”

（2003 年 4 月 26-27 日，北京）

现在，UML、RUP、Rose 等概念已经传播得越来越广，书籍、资料也越来越多，决定在开发过程中使用 UML 的开发团队也越来越多。

在开发团队应用 UML 的软件开发过程中，自然会碰到很多细节问题：“我这样识别 actor 和用例对不对？”、“用例文档这样写合适吗？”、“RUP 告诉我该出分析类了，可类怎么得出来啊？”、“先有类，还是先有顺序图啊？”、“类怎样才能和数据库连起来啊？”...许许多多的细节问题，而每一个细节都和背后的原理有关。

本课程秉行 UMLChina 一贯的“只关心细节”的原则，内容完全是由 UMLChina 自行设计，围绕一个案例，阐述如何（只）使用 UML 里的三个关键要素：用例、类、顺序图来完成软件开发。使学员自然领会 OOAD/UML 的思想和技术，并对实践中的误区一一指正。整个过程简单实用，简单到甚至可以只有一个文档，非常适合中小团队。

详情请见>>

让历史告诉未来

——评 Ivar Jacobson 博士的《用例的昨天、今天和明天》（2003.4 版本 A）

IT 之源 张恂

查看评论

概述

2003 年 3 月，著名的电子杂志《The Rational Edge》在 IBM 出巨资收购 Rational 之后的第一期发表了 UML 之父 Ivar Jacobson 博士撰写的《Use Cases——Yesterday, Today and Tomorrow》。这篇文章依次介绍了用例技术的起源、发展和演进过程，并且对大家在实践中困惑较多的用例关系、用例数量、用例与 UML 等问题作了深刻的阐释，同时提出了对扩展/包含用例的改进意见，最后还对用例未来的发展趋势作出了有趣的预测。

这次由用例的发明人亲自撰文、现身说法讲述用例的历史，实在难得。我怀着如获至宝的心情反复研读了原文。现在把我整理、学习这篇文章的体会记录下来，在介绍 Jacobson 博士观点的同时加入我个人的分析和评述，以飨广大读者。

一、用例的起源

1.1 萌芽期（1967-1986 年）

用例的出现源于 Ivar Jacobson 从 1967 年开始在爱立信公司所从事的近 20 多年对大量不同电话呼叫类型建模的工作。当时他把各种不同类型的电话呼叫情况（电话用户对电话交换系统的需求）称为 traffic case，完成所有呼叫则需要交换机具备各种不同的“功能”（function）或特性（feature）。

1986 年春天，在研究如何把 traffic case 映射到功能的过程中，Jacobson 突然产生了灵感，发明了“use case”这个术语。他发现可以利用一种类似继承的机制通过功能来表达 traffic case。当时他把这两者都叫做用例，而 traffic case 就是后来的具体（concrete）或真实用例，功能则变成了抽象用例。

在提交给 OOPSLA’86 的一篇会议论文中，Jocobson 首次提出了用例的概念，可惜当时那篇文章由于特殊的原因没有被录取。在随后的更新版本中，Jacobson 又加入了许多用例建模的重要概念，这篇论文终于在 1987 年的 OOPSLA 大会被录取，用例正式诞生了。（迄今已有 15 年了！）

Jacobson 在发表的用例建模理论中提出，**“用例是由用户和系统在一次对话中执行的一个特殊事务序列。”**为此，Jacobson 还开发了一个独立的模型用于从外部视点描述系统，这就是我们熟知的用例模型，它提供了系统的黑盒视图，代表了系统的功能需求。当时的用例概念模型可以被无缝地翻译成一个新模型，它可以展示每个用例是如何通过明确的块（可能是一个子系统、类、或构件）实现的。当时序列图（sequence diagram）就已经被用来描述用例是如何通过软件块/构件之间的交互来实现的。对每个用例进行独立测试可以保证系统满足了用户的需求。可见，用例已经构成了当时整个开发活动的核心。

Jacobson 提到他早在 1969 年就发明了序列图，所以当他说出“这不足为怪，那时序列图展示其在实践中的价值已经差不多有 20 年了”时，实在令人惊讶——这意味着序列图到现在竟然已有 35 年的历史了！

1.2 成熟期（1987-1992 年）

从 1987 年到 1992 年的 5 年间，Jacobson 通过自己创办的公司 Objectory AB 展开了大量的用例实践，约有 20 多家涉及管理信息、国防（导航、测量、C3I）、电信（POTS、移动）等众多领域的公司纷纷把 Jacobson 发明的以用例内容为核心的 Objectory Process（对象工厂过程）应用于新产品开发。

此时，用例引入了“继承”关系。到了 1992 年（10 年前！）用例已经具备了大部分当代的语法和语义，并且已经对用例（也是像类一样的东西，即“类元”）、用例的一个实现和用例的一段描述进行了严格的区分。

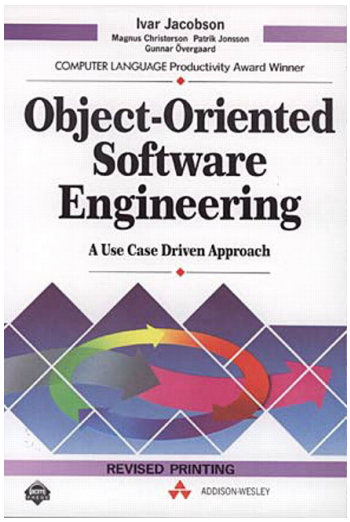
当时的用例描述主要包含下列内容：

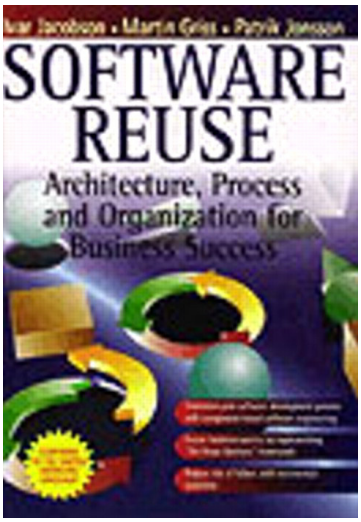
- 用例目的的简要叙述
- 控制流
- 基本流和可选流
- 子流程（可在同一个用例描述中的多个位置上被重用）
- 前置条件和后置条件

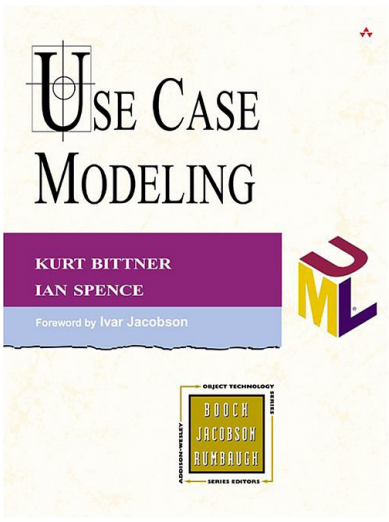
Jacobson 还与同事们创造了“用例驱动式开发”这一术语。整个过程简单说就是，首先找出所有的用例并在需求文档中对其进行详述，然后对每个用例进行分析、设计，最后对所有的用例进行逐个测试。

在这一阶段，用例不仅仅是一种非常有效的需求技术，它还可以跟踪到分析、设计、实现和测试。对于每个用例，在分析和设计阶段都有一个协作（参与对象的视图）与其对应。每个用例都能产生一个测例（test case）集合。用例对于用户界面设计、编写用户手册也很重要。由于用例能够完美地捕捉业务过程，其应用甚至还扩展到了业务建模领域。

在获得了深入理论研究和充分实践经验的基础上，Ivar Jacobson 与其同事一起于 1992 年出版了《Object-Oriented Software Engineering: A Use Case Driven Approach》(Addison Wesley, 1992）一书，奠定了 OOSE 和用例方法的基础，这无疑是一本软件工程史上划时代的里程碑之作。







1.3 发展期（1992 年至今）

1992 年以后用例的发展史大家可能比较熟悉了。1995 年 Jacobson 加入了 Rational 公司，并与另两位 OO 大师 Grady Booch、James Rumbaugh 携手一同领导面向对象建模语言的统一进程，演绎了软件史上的一段佳话。用例和 OOSE 随之融入了 UML，并与 Objectory 过程、Rational 方法结合产生了如今闻名遐迩的瑞理统一过程（RUP）。RUP 的 3 个基本特征就是以用例为驱动，以架构为中心和迭代递增式开发[2]。

用例被广泛接受、迅速采纳的速度令 Jacobson 也感到惊讶：它发表后几乎立即就受到所有方法论学家的欢迎，并在全世界范围内被投入大量应用。这可能源于用例本质上是一个简单、显而易见的想法，它天然地与对象、面向对象思维吻合得很好。

二、用例的今天

在回顾辉煌的历史之后，Jacobson 博士不但对用例应用中一些常见的问题作出了澄清和说明，而且还提出了相应的改进建议。

2.1 如何控制用例的数量

今天 RUP 和 UML 中关于用例的定义主要来源于 1994 年，为了使软件人员准确地把握用例的粒度避免用例数量过多或过少，它强调用例必须为“特定的使用者（Actor）”提供“可度量的价值”。Jacobson 建议：支持单个业务过程的大型系统所拥有的具体用例的数目通常不应该超过 20 个。

2.2 用例与 UML

如今 RUP、UML、用例已经完美地融合在一起，全球成千上万的客户和专家对其进行了充分的实践。

UML 把用例的“使用”（uses）关系改为一般化（generalization，又译泛化）关系，并且增加了《包含》关系。UML 旧版的使用关系实际上源自继承关系，这一点可能出乎许多读者的意料。

Jacobson 博士还向我们推荐了 2003 年刚出版的新书《用例建模》(Kurt Bittner 和 Ian Spence 著，Addison Wesley)，他认为这是“THE book on use cases”(用例专著)，并强烈推荐每一位软件工程和需求开发人员阅读。

2.3 用例的形式化

Jacobson 博士提醒我们，在使用用例的形式化技术时要小心。毕竟，用例模型是用来与客户和用户交流的。他指出，“利用数学对用例进行形式化从来都不是个问题”，他本人早在 1986 年就已经这么做了，而且计算机科学系的学生也都可以做到。

到现在，Jacobson 仍然不太情愿建议系统分析员用一种比简单文本更形式化的方式来描述用例。他建议，应避免用活动图或状态图来详述每个用例的内部状况（注：这其实属于分析任务），尽管用序列图或带有泳道的活动图来描述用例与使用者之间的交互是不错的。

2.4 用例关系

Jacobson 博士对在用例关系的应用中出现的问题进行了澄清，并对扩展/包含抽象用例的处理方式提出了改进。

他认为，一个用例模型通常包含四种用例：

- 具体用例（可被实例化，抽象用例不能）
- 一般化用例（支持用例的重用）
- 扩展用例（向已存在或假定存在的用例添加新的行为而不改变它）
- 包含用例（通过增加行为改变其他用例）

2.4.1 一般化

一般化用例都是抽象用例，它们是对其他具体用例或抽象用例的一般化。一般化用例（父用例）与子用例必须类型相同以保证可替换性。这遵从了 OO 继承的基本原理：由于子与父具有相同的类型，所以可以在使用父实例的任何地方用子实例来代替。

2.4.2 扩展

扩展用例的目的比较特殊，它们的作用是在不改变某个已存在（或假定存在）的用例的前提下为之增添新行为。这些附加的行为可能是必需的，也可能是可选的。

使用扩展的一个潜在问题是创建过深的扩展依赖层次，因此 Jacobson 博士建议永远不要扩展一个扩展（片段）。

对于在描述用例的时候，什么时候用扩展，什么时候用可选路径，Jacobson 建议：只有当扩展用例与被扩展用例完全分离（即它本身是一个独立的具体用例或者是其他用例需要的一个小片段）时，才使用扩展关系。基用例自身必须是完整的，它的正确执行不需要扩展。否则，就应该用可选路径来描述附加行为。

扩展实际上并不仅仅对用例建模有用，它的出现甚至可以追溯到 1978 年，当时 Jacobson 还在爱立信工作，后来他在 OOPSLA’86 的论文中对扩展作了介绍，而直到 1991 年开发人员才接受这一概念。有趣的是，爱立信公司甚至还为此申请了专利，用于对 C++、操作系统、计算机体系结构的扩展。当时的基础设施团队还利用这些方法显著降低了开发成本。

2.4.3 包含

用例有两种重用方式。Jacobson 在 1987 年引入用例的继承关系，并在 1992 年把它的名称改为“uses”，也就是后来 UML 的“一般化”关系。

第 2 种重用方式，即从用例的描述中提取出公共（共享）的事件流，就是现在的《包含》关系。Jacobson 博士多次强调，他很不情愿引入包含这样一种关系，包含关系使用不当容易诱使人们进行功能分解，从而导致对用例的误用。因此他们过去不允许开发人员对用例公共的片段建模，而是用文本对象（text object）来代替。这是一些包含文本段落的可重用对象，可以被多个位置同时引用，很容易保证同步。

Jacobson 说，“事实上，今天一些人误用了用例，把它们用来描述功能（注：指功能分解式的分析）而不是对象，反过来又指责用例概念存在问题”，这真是令人哭笑不得。前些年我在网上也看到不少贴子说用例不是 OO 的，甚至还有国外某些“专家”文章佐证，这令我很纳闷。我们怎么不动动脑子呢？让直觉说话吧——用例明明出自 OO 大师之手，怎么反倒变成不是 OO 的了？

2.4.4 扩展与包含

扩展和包含用例本质上其实非常相似，它们的主要区别在于用例实例中断基用例、执行附加用例的方式。

扩展和包含用例都与基用例相联。在基用例的执行过程中，可能在某种条件下基用例的执行流被中断，转而执行扩展或包含用例（在 UML 中统称为附加用例）的流。当附加用例流执行完毕，控制将返回到基用流原来被中断的那个位置恢复执行。

扩展用例通过引用扩展点（extension point）建立与基用例的联系，扩展点指明了在基用例中的扩展位置。以前在 OOSE 和 Objectory Process 中，扩展点属于扩展用例。在制定 UML1.1 时，James Rumbaugh 建议扩展点应该属于被扩展用例。这是出于封装的考虑：扩展用例不应看到基用例的细节——它只能看见扩展点；因为如果扩展点改变了，只有被扩展用例应该知道新的位置。

2.5 两类扩展/包含用例

1）作为具体用例的扩展/包含

具体用例既可以扩展基用例，也可以被基用例包含。这些用例可以与使用者交互、实例化并提供价值。

2）作为用例片段的扩展/包含

这种类型的用例数量较多，但一般每个用例都很小。这些用例是抽象的，也就是说不能被独立地实例化，它们通常是其他具体或一般化用例所需要的。

Jacobson 博士专门举了一个银行系统的例子（如图 1）。在这张图中有两个具体用例：“执行事务”和“检查事务失败”（管理用例）。每当一个事务失败时，银行要记录这一事件以便其他用例（如“检查事务失败”）能够获得这一信息。

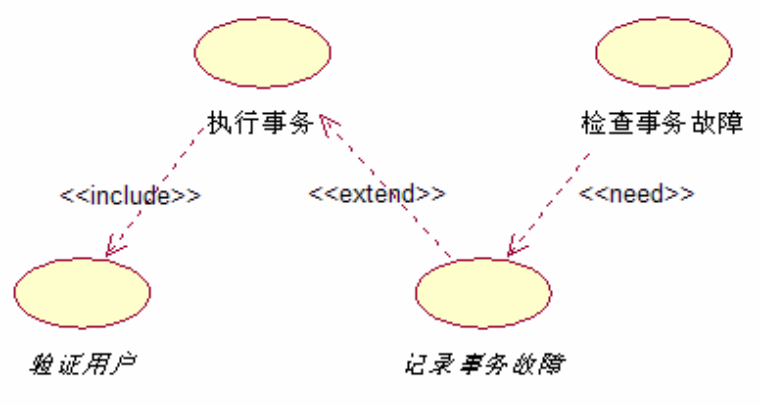


图 1、具体用例与抽象用例（片段）

“记录事务失败”与执行事务本身无关，它只是为了登记其他用例需要的信息，所以它是基用例“执行事务”的扩展。“验证用户”是一个被包含的用例片段。“验证用户”和“记录事务失败”都是抽象的（故用例名称用斜体字表示），他们不能直接被使用者实例化。

值得注意的是，Jacobson 博士强调，用例片段（抽象用例）——无论扩展片段还是包含片段——都不是“真正的”用例。没有对真正的用例和不那么重要的用例片段作出明确的区分，实际上是过去的一个小缺陷，很容易在使用中造成混淆。于是他提出通过补充少量图形元素对 UML 改进的想法。

既然片段不是用例，那么就不应该用用例的语法来表示它。在 UML 中，片段应该是一种带有图标的类元，能提示我们它是一种细小的东西。为此 Jacobson 向我们推荐了一种可能的新表示法，采用“圆点”或类似的小图标来表示扩展/包含片段。

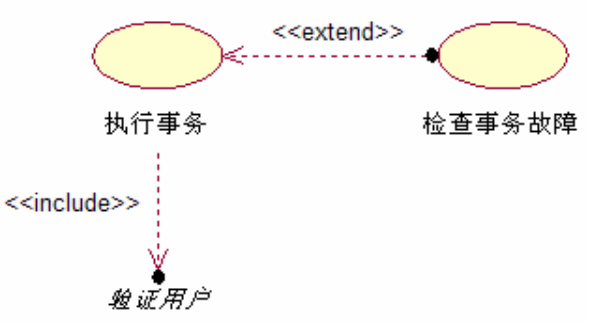


图 2、片段表示法

在图 2 中，“记录事务故障”缩成了一个匿名的圆点（表示扩展片段），并贴附在具体用例“检查事务故障”上面，通过《扩展》关系指向“执行事务”。这表示：当“执行事务”用例激活扩展点时，由小圆点代表的扩展将开始执行。“检查事务故障”用例将利用（需要）“记录事务故障”这一扩展来完成其任务，而后者并不是前者的一部分。注意，我们不能直接让“检查事务故障”扩展“执行事务”，这在语义上是不通的。所以，需要用在具体用例“检查事务故障”上附加扩展片段的方法来处理。

在图 2 中“验证用户”是一个包含片段，它可以是可重用的用例描述或文本对象，并与包含它的用例“执行事务”分离。如果有多个用例“需要”某个扩展片段，那么也可以把它独立地表示成一个类元。所以在这点上，扩展/包含片段在语义上是相似的。

当然，扩展/包含片段也存在重要的区别：一个包含片段（如“验证身份”）将由执行真实用例（即包含它的用例，如“执行事务”）的用例实例来执行它；而一个扩展片段（如“记录事务故障”）将由需要它的用例实例（如“检查事务故障”）之外的用例（如“执行事务”）来执行。

三、用例的未来

在文章的最后部份，Jacobson 博士向我们介绍了用例技术的发展趋势：

1）可以通过 UML 的构造型机制对不同的用例类型进行细分以方便开发人员使用，比如添加管理用例、运营用例、辅助用例等等。

2）进一步澄清模式与用例的关系。在模式与描述特定问题的一般化的、可重用的用例之间存在一种有意思的联系。许多设计模式实际上是实现可重用用例的“模板”。

3）在人机交互（HCI）领域中运用用例。HCI 是一门科学，用例能够在集成软件开发与 HCI 的实践中扮演更加重要的角色。

4）利用用例进行成本估算。早在 1994 年 Magus Christerson 就在 Gustav Karner 硕士论文[1]的基础上提出了基于用例点的项目估算方法，这些经验如今仍具有指导意义。

5）开始重用用例。这是一个很大的话题。**重用业务软件应该从理解用例开始**——既包括业务用例，也包括软件用例。1997 年 Jacobson 与 Martin Griss、Patrik Jonsson 发表的专著《*Software Reuse: Architecture, Process and Organization for Business Success*》（Addison Wesley Longman）对此专门作了深入探讨。

6）让用例情节（scenario）成为一等“公民”。识别、列举用例情节并说明它们之间的依赖是有帮助的。比方说，可以根据情节来制定、跟踪项目迭代计划，建立用例情节到测例的跟踪链。

7）建立用例与 Aspect 的密切联系。Jacobson 博士认为 AOP（Aspect-Oriented Programming）是当今最激动人心的技术新进展之一，他深信 AOP 将为用例带来戏剧性的变化。实际上，扩展用例的目的——在不改变已有系统的情况下添加新的行为——与 aspect 非常相似。用例实现可以实现为 aspect，扩展用例也可以实现为 aspect，扩展点则与 AOP 中的联结点语义上近似。有了 AOP，我们就可以从用例设计直接过渡到用例编程，再到用例测试，显著减少构件方面的工作。事实上，用例就像横切构件的模块，在这些模块上的工作将由新型编程环境（AOP 语言=OOP 语言+aspects）来支持。AOP 使得我们可以无缝地实现用例，两者的结合将极大地改进如今软件开发的方式。

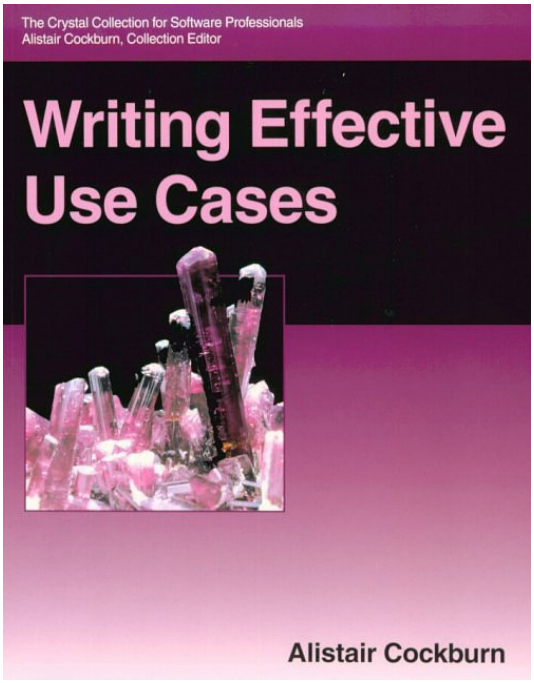
8）使用例成为运行时实体。最令人兴奋的用例展望是——将来用例在运行时环境中也有对应物。能够在操作系统中识别执行用例（实例）将带来很多重要的特性，比如可以更加增量化的方式改变已安装的软件（每次 1 个用例实例），用更小的步骤（终断的用例实例）重启软件系统。

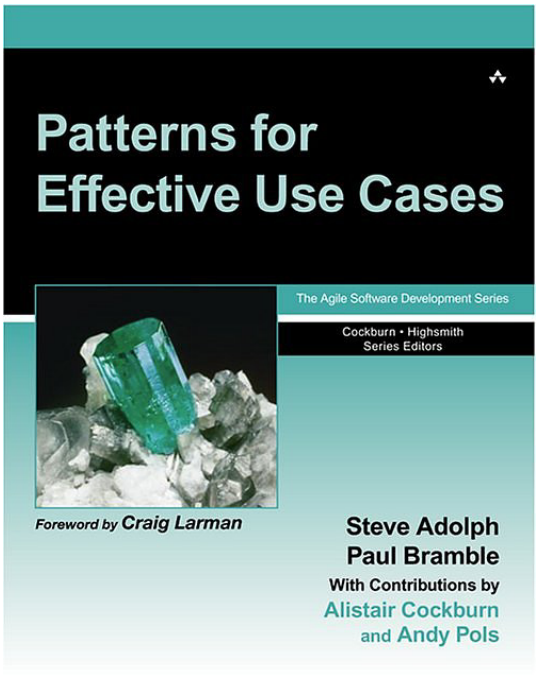
四、体会和结语

读完全文，我有以下几点体会和看法与大家分享：

1）我们怎么强调用例的重要性可能都不为过。老式的结构化功能分析其实并不能满足当今复杂商业软件开发的需要，用例分析不同于功能分解，也就是说很多情况下简单的功能描述是不够的，**我们必须规范、确切地给出功能“使用的例子”，这才是真正的功能需求**。用例桥接客户需要和软件功能的特殊地位和作用使得它不仅仅能用于需求的捕捉和提炼，而且还构成了系统分析、设计、测试乃至整个项目管理的基础。**事实上用例已经成为当代软件开发活动的轴心**。

2）我们知道 Alistair Cockburn 是用例另一支流派的代表人物，他在基于用户目标的用例需求分析方面开展了大量实践，并且也获得了全球范围内开发者的一致好评和认同。可是我发现 Rational 公司和 Jacobson 博士本人有关用例的大量文献很少提到 Cockburn 的名字，这一现象让我感到有些奇怪，我也隐约地感觉到 Jacobson 博士和 Cockburn 在对用例、UML 的看法上存在一些分歧（虽然并无本质上的不同）。事实上，我们不难发现，Cockburn 与 Rational、UML 的用例方法在一些细节上存在着明显的差别。作为一名中国的用例和 UML 实践者，我认为面临这个现实问题，我们有必要在具体实践中将 Cockburn、Jacobson 以及 UML 标准的方法巧妙地结合起来，汲取二者的精华，从中提炼出统一而实用的用例方法。





3）Jacobson 的文章再次说明了一个道理。坦率地讲，用例、UML、OOAD 和 RUP 基本上都不能算什么“新技术”，它们至少都有 10 年以上的发展史。我们总是不断听到业内人士评论、抱怨软件技术发展太快，新技术层出不穷，跟也跟不上，从而得出推论：既然它们是新技术，那么肯定都还不成熟，我们不如观望之或干脆拒绝之，还是用老方法保险。事实到底如何呢？我怀疑还有一种极大的可能性——大量所谓的软件“新”技术其实并不新，**不过是“新”近传进来或者我们刚刚知道罢了**，人家早已经用了 5 年、10 年甚至更长时间了；况且很多看似时髦的新技术背后其实都有着非常深厚的技术渊源和理论基础，我们看到、听到的“新技术”不过是些肤浅的表象。长期的技术传播落后、信息交流不畅和安于现状、自欺欺人的心态阻碍了我们虚心踏实、坚持不懈地学习基础知识，知己知彼、与时俱进，赶上国外先进的软件技术发展水平，与国际接轨。现在我们其实是在“补课”，需要补 CMM 的课（CMM 差不多也是别人 10 多年前的成果），也需要补 UML、用例的课。所以，大家应该协同努力尽早改变这一不合理的现象。

总之，Jacobson 博士的《用例的昨天、今天和明天》无疑是一篇宝贵的学习资料，它带给我们很多的启迪和感悟。它能使我们对用例相关概念的来龙去脉有更为清晰、深刻和完整地了解，便于我们在今后的工作中更好地掌握和应用这一先进技术。相信大家在读了之后也会有相同的感受。

参考文献

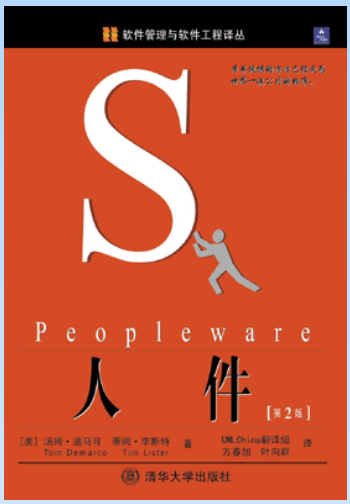
[1]《用例分析技术》（第 2 版），G. 施奈德等著，姚淑珍等译，机械工业出版社，2002.8

[2]《统一软件开发过程》，Jacobson 等著，周伯生等译，机械工业出版社，2002.1

张恂：东南大学工学硕士（软件），OO 软件工程传道者，软件咨询顾问。长期从事软件工程与面向对象技术的应用开发、管理、咨询和推广工作，具有 8 年软件开发的丰富经验和扎实的理论基础。曾先后担任国内著名通信企业大型移动系统研发的软件项目经理、某民营软件公司 CTO。在国内通信软件界较早地引入了 UML、模式、统一过程、配置管理等先进的软件工程理念和方法。2002 年 9 月发表国内第 1 本软件组织模式译著《软件架构：组织原则与模式》。研究兴趣包括 OOAD*UML、需求工程、软件架构、软件模式、敏捷（统一）过程、软件项目管理、质量保证等等。在 IT 之源开设了个人专栏，愿与各位网友切磋交流，共同进步！

征稿

http://www.umlchina.com/xprogrammer/xprogrammer.htm



Tom Demarco，Tim Lister

翻译：UMLChina 方春旭 叶向群

http://www.peoplewarecn.com

人件中文版网站

Frederick P. Brooks, Jr--《人月神话》第 19 章

近年来，软件工程领域的一个重大贡献是 DeMarco 和 Lister 在 1987 年出版的《人件》，我衷心地向我的读者推荐这本书。

Edward Yourdon

《人件》在 1987 年出版后，立即成为最畅销的作品。当人件第一版出版时，我写了一份评论，“我强烈推荐你买一份《人件》给你或你的老板，如果你是老板，那么为你部门的每一个人买一份，并给自己买一份”。这建议在 12 年后的第二版依然有效，并且更加热烈。

Mark A. Herschberg

我只学了办公环境的章节，就辞掉了原来的工作！

Joel Spolsky

我想，微软成功的原因之一就是公司里的所有经理都读过《人件》

Steve McConnell

由于本书第 1 版的赫赫声名，新版的《人件》是我不用看就会决定购买的少数几本书之一。

预订中文译本>>

从组织建模出发开发用例

Victor F.A. Santander 等 著，wnb 译

查看评论

摘要

面向对象的开发规范吸引了许多软件工程领域的支持者参加。目前其主要的成果之一是统一建模语言（UML），一种实现可视化建模的标准。用例图用于获取系统的功能需求。然而，系统开发往往基于一个重要的前提，组织过程已经建立完好。因此，对系统如何完成组织目标需要获取组织需求，以及为什么需要它，有哪些可能的选择，在包含的各个部分中存在何种联系等等。遗憾的是，UML 及其它基于场景的技术难以胜任对组织需求进行建模的任务。需要其它的技术来完成该任务，I*技术恰好可以应用于对组织需求建模的工作。当然，组织需求必然与由用例表达的功能需求有关。本文提出了一些辅助从 I*技术表达的组织模型出发，进行用例开发的需求工程方法原则。

关键词

场景，组织建模，需求工程。

1 介绍

复杂软件系统的开发，由于其对验证和有限预算的易感性，长期以来一直是对软件工程师的重大挑战。目前也提出了一些辅助和支持高质量软件开发的技术、方法和工具[9][12][16][26]。但软件危机在许多方面依然存在。经常可以看到难以真正满足用户需要的软件。其中主要的原因在于：缺乏对软件过程的有效定义，软件需求没有被很好地理解或（与用户）达成共识，采用了不适合的技术以及复杂软件本身所固有的难以表示的特性等等。

软件工程领域一直强调需求工程是软件开发过程中最为关键的活动。应该适当地获得、分析、验证和管理系统需求以满足项目相关人员（涉众）的需要。不完全、不一致的需求文档通常是导致软件产品质量低下，难以满足客户需要的原因。因此，有效地开展需求工程的各项活动并尽力在软件开发的初期阶段消除需求方面可能出现

的问题非常重要[26]。目前需求工程研究提出的方法和技术，都将目标锁定在开发完整的、一致的需求文档方面。我们认为在需求文档细化阶段必须考虑的主要问题包括：考虑软件执行的组织环境中所有相关要素，获取功能性和非功能性需求，确保这些需求尽可能完整是确保文档化的需求能够满足用户需要的基础。

就系统需求的定义来说，可以参考文献上提出的技术。70 年代，开发了诸如 DFD、ER[13]等技术来获取功能需求。最近，作为可行的需求过程的方法，场景、用例、ethnograph（观察法或分类法）、规范方法等被广泛地使用。在这些技术中，我们将讨论重点放在基于场景的技术上。场景可用于描述用户与软件系统的交互。最流行的基于场景的技术是众所周知的用例技术。正因为用例技术的广泛采用，UML 也将其纳入其中。在 UML 中用例主要用于获取系统的功能需求。

由于系统开发的前提是已经建立了良好的组织过程[4][5][27]。因此，需要获取组织需求以定义系统如何完成组织的目标，为什么需要它，有哪些可能的选择，在各个包含的部分中存在何种联系等等。遗憾的是，UML 及其它基于场景的技术难以完成对组织需求进行建模的任务。I*[27]技术恰好可以应用于对组织需求建模的工作。不过，组织需求必然与由用例表达的功能需求有关。我们主张采用的 I*框架技术适合于表达需求获取先期阶段的组织需求，它提供了适合的表达抉择，并提供了基本的建模概念如软目标（softgoal）和目标。

组织需求必然与使用用例技术表达的功能需求有关。通常，由富有经验的需求工程师开发用例。而使用目前文献中表达的开发用例的启发式知识完成系统化的开发是不充分的。的确他们没有考虑有关的如软目标、目标等组织方面的问题。本文提出了一些支持 I*与用例集成的指导策略。描述了一些辅助需求工程师基于组织化 I*模型开发 UML 用例的启发式知识。I*框架可用于增进对系统如何完成组织目标的理解。我们提出的观点可以用如下图 1 表示。

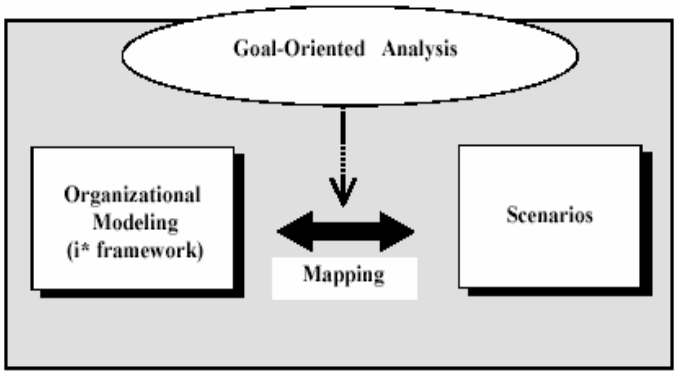


Figure 1. A vision of mapping process between organizational modeling and scenarios.

图 1. 组织建模与用例间映射过程的描述

目标是集成目标驱动的方法论、I*框架和场景，如目标驱动的分析下所示。早期的较好的研究结果可参考文献[6][18][19][20][23]。在本文提出的方案中，由 I*所表达的要素如目标、软目标、资源和任务被用于分析以建立用例的目标。在 I*框架中，角色之间通过相互依赖完成目标和软目标并最终实现任务。我们也需要研究这些要素与系统需求和系统用例有什么样的关系。

本文提出了指导方法和规则，帮助需求工程师从 I*框架所描述的组织模型出发开发（与场景关联的）用例。该工作是文献[24]工作的演进。最初表达的指导被改进为三个良好的定义阶段（第 4 部分，图 5）。这些阶段提供了集成 I*技术与用例的更系统化的方法。另外，也介绍了其它需要考虑的重要的问题，如面向目标分析的应用和用例集成过程的规则（如第 4 部分中指导 1.5 和 2-2 建议步骤）。我们建议所使用的方法有助于需求工程师从 I*发现并定义与用例目标。这些目标分为三个层次（业务、上下文及用户目标）。对这些目标的分析可以帮助需求工程师发现其它用例并确保与目标有关的用例处于适当的抽象层次上。其它相关工作包括改进了 I*模型和可能存在的源自预期系统其它用例的角色间的依赖修改。第 4 部分描述了我们提出的所有建议和指导。

本文如下组织。第 2 部分介绍了表达组织化需求和早期需求的 I*框架的概念。第 3 部分提出了基于场景技术，特别是用例技术的主要概念。第 4 部分，介绍了集成 I*组织化模型和用例图的指导方法和步骤。为表明方法的可行性，我们也描述了如何使用这些指导设计会议调度问题。第 5 部分对本文进行了总结。

2 I* 建模框架

在进行系统开发时，通常需要广泛深入地了解系统的组织环境和目标。I*框架[27]提供了对构成系统需求的原因（为什么）的理解。该技术提供了实现策略性角色关系的建模框架。当我们试图理解组织时，一般由标准建模技术如 DFD、ER、Statechart 获取的信息，主要关注实体、功能函数、流程、状态以及诸如此类的信息。他们难以表达过程（动机、意图、合理性）产生的原因（为什么？）。I*方法的主旨是获取这些高级的概念。因此，I*方法允许对在组织环境中角色的意图和动机进行描述。I*提供了两个模型：策略依赖模型和原理（理性）依赖模型来表示这些涉及的方方面面的问题。I*技术可用于各种应用领域：需求工程、商业过程再工程、组织影响分析，软件过程建模等等。

2.1 策略依赖模型

策略依赖模型主要处理组织角色间的意向性关系。策略性模型是由一个包含节点和连线的集合构成，节点表示角色，两个节点间的连线表示角色间的依赖关系。产生依赖的角色称为依赖者(Depender)，被依赖的角色称为被依赖者(Dependee)。因此，该模型包含角色间关系的集合，这些关系用以获取角色之间的意向和动机。I*框架定义

了角色之间的 4 种依赖关系，分别是：目标依赖、资源依赖、任务依赖和软目标（softgoal）依赖。这些依赖关系分别表示组织环境中不同的意向和动机。在目标依赖关系中，角色需要其它角色完成一个目标，而其本身不用考虑该目标如何被完成。资源依赖关系，角色依靠其它角色提供物理资源或信息。任务依赖关系，角色依靠其他角色实现系列的活动。角色间的软目标依赖依靠其它角色完成模糊的目标。软目标依赖表示的依赖与其它几类不同，它所表示的目标没有或无法进行精确的定义。在需求工程中，软目标代表非功能需求。

在 I*框架中还可以定义角色间依赖的程度：Open、Committed、Critical。角色可提炼成主体、角色和形势。主体是一个具有具体物理表示的角色（人或系统）。角色是对特定环境、领域或职责中社会角色的行为的抽象刻画。Position 是由一个主体驱动的角色的集合。

图 2 用策略依赖模型描述了一个会议调度（Meeting Scheduler）问题。会议发起者（Meeting Initiator）角色与会议参与角色（Meeting participant）之间包含一系列依赖。资源依赖 ExclusionDates(p)和 PrefferedDates(p)表示参与者应该提供关于不适合参会的日期和适合参会的日期的信息。会议发起者依靠这些资源继续进行调度处理。

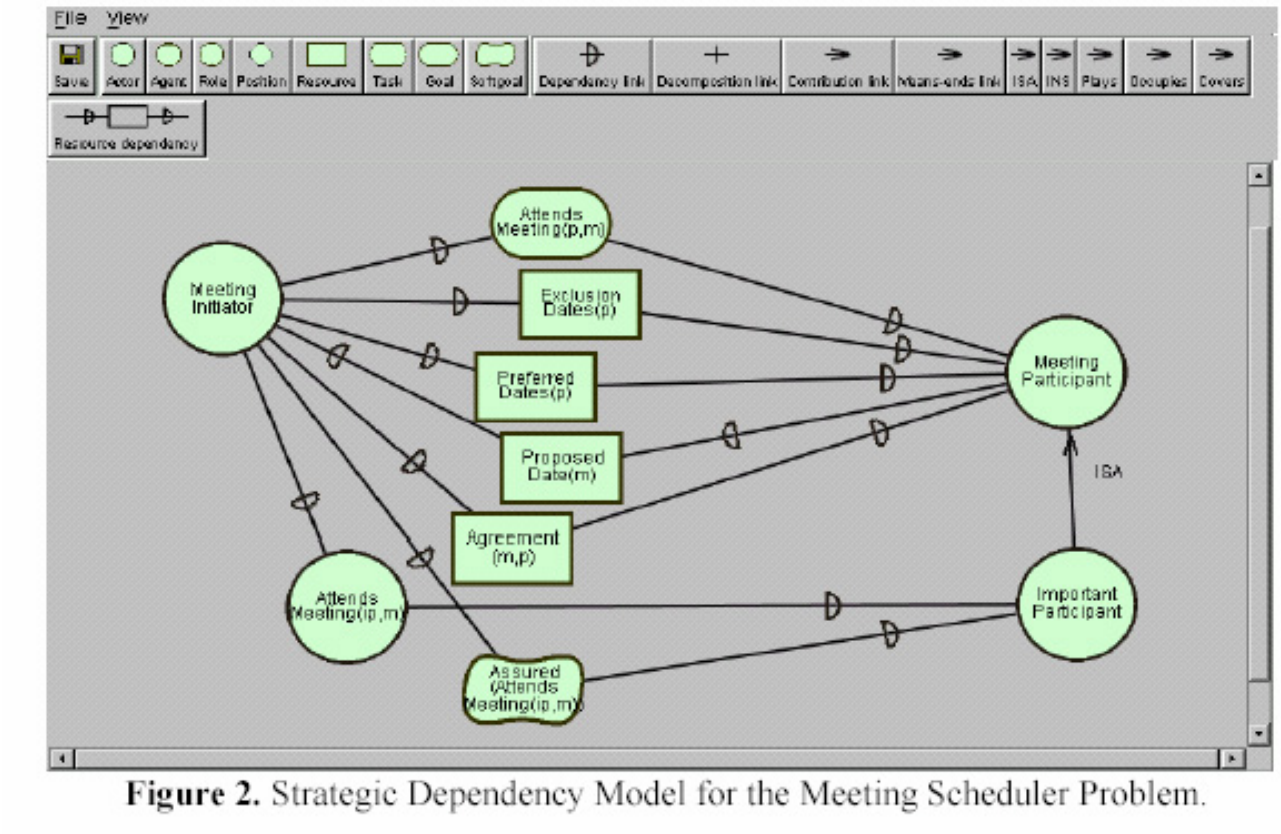


Figure 2. Strategic Dependency Model for the Meeting Scheduler Problem.

图 2. 会议调度问题的策略依赖模型

另一方面，会议参与者根据（依赖）会议发起者提出的日期（ProposedDate(m)），决定可能同意或不同意发起者的时间安排。目标依赖 AttendsMeeting(p,m)考虑会议参与者满足参加会议的目标。在此条件下，会议参加者角色可以选择会议完成的方式。Agreement(m,p)依赖表示了一个会议参加者应该提供显示是否一个参加者不同意提出的日程安排的资源。Assured（AttendsMeeting(ip,m)）软依赖是一个主观评价的参考引用。

2.2 策略原理模型

策略原理模型是策略依赖模型的补充模型。该模型可以对各个角色及他们之间依赖产生的原因进行建模。角色从表面看来与其它角色具有某种依赖关系，实际上，其本身包含的一定目标是产生和保持依赖关系的基础。为了实现该模型，需要分析每个角色与其它角色产生依赖关系的原因。开始分解的好方法是观察被依赖角色如何能够同时满足与其相关的依赖，然后从整体上分析和分解意图和策略性组织原因。节点和连线共同构成该模型。节点在策略依赖模型中根据依赖分类为：目标、资源、任务和软目标。连线可表示为两类连接：方法终点（Means-ends）和任务分解。连线可以如下描述：

方法终点：该连线获取一定的结果终点，这些终点可以是目标、资源、软目标和任务。获取终点的方法被定义为需要结束的任务。

任务分解线：通过分解连接，节点任务与其子部件连接。可以连接的节点类型有四类，对任务的分解通过这些连接类型进行。与完成任务有关的方法和原因的表示可以通过连接分解成更小的任务单元。

图 3 表示了策略原理模型的示例。该模型表达了与调度会议组织环境中角色有关的前提和原因。会议调度角色（Meeting Scheduler），表示软件系统，完成部分的调度会议工作。其他两个角色，会议发起者和参加者，向系统提供并接受系统的信息。会议调度角色处理调度会议任务，该任务可使用任务分解关系分解为以下三个任务：ObtainAvailDates，FindAgreeableSlot 和 ObtainAgreement。这些任务表示了会议调度系统将要完成的工作。同时，组织模型内其它角色被分解表示满足与其他角色依赖的内部原因。

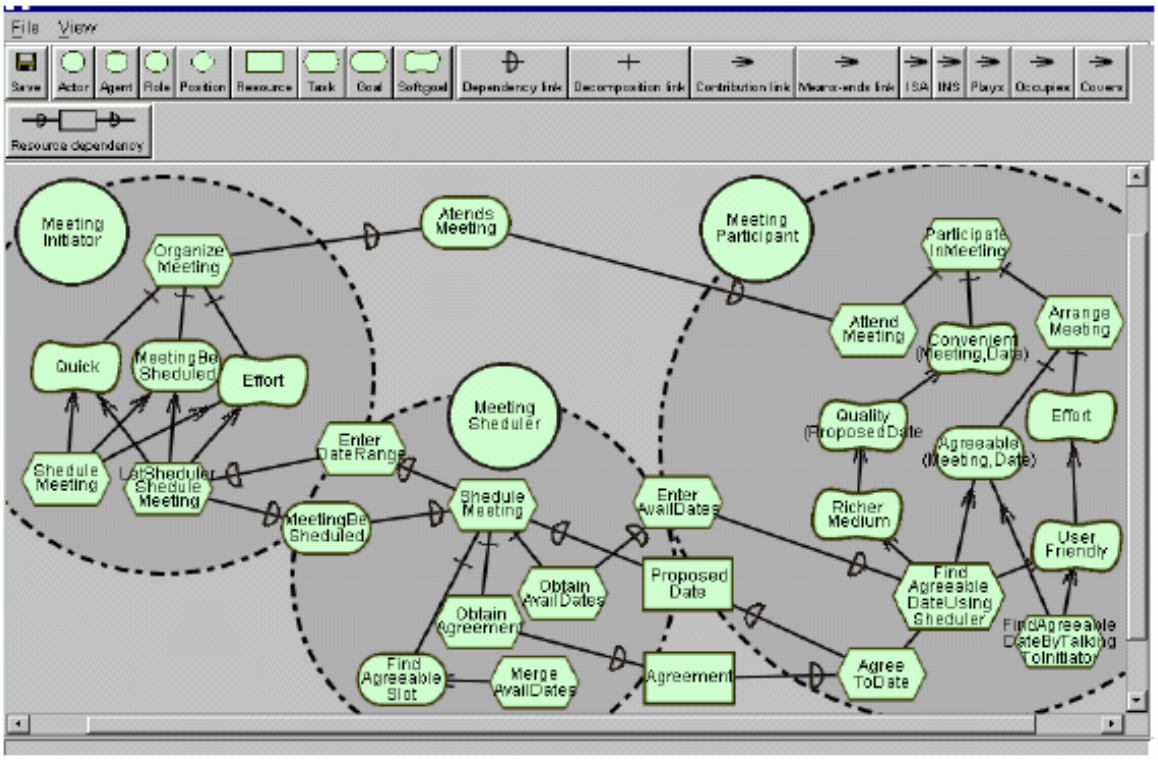


Figure 3. Strategic Rationale (SR) Model to the Meeting Scheduler System.

图 3. 会议调度系统的策略原理模型

3 UML 的用例

在软件工程中，基于场景的技术用于理解、建模和验证用户需求。文献[23][17][16]均提出采用场景获取和验证需求的方法。在这些技术中，用例倍受关注。用例图也被 UML 采用，UML 是可视化建模的标准，是面向对象开发规范中最重大的进步。

UML[2]中的用例通过角色来描述系统的使用情况。角色是与系统进行交互的任何外部元素。用例描述用户与系统交互实现某一个目标时完成的序列。然而，对用例的描述不关心这种交互如何实现。软件工程的后续阶段，例如设计和实现阶段关注交互如何实现的问题。

一个用例可以建立多个场景。场景之于用例就像实例之于类。就是说场景是用例的实例。角色对系统的使用包含几种方式，对这些方式的采用依赖于系统执行环境。这些方式成为可能构成的用例场景。实现用例的基本方式，不存在问题和错误的方式，成为主要场景。在主场景中，实现用例基本功能的执行步骤成功完成。另一方面，其它可能出现的方式及出错情况可以通过辅助场景表示。

辅助场景描述了主场景的可能替代情况，辅助场景可以分开描述或作为主场景的扩展。如果辅助场景更加复杂，包含了几个步骤，分开描述非常方便。应该注意到对场景的定义有可能产生差异。例如，在[17]中，属于场景的范围非常广。场景与软件开发过程一起改进，与最初宏观系统的描述有所差别。另外，场景自然地与需求基线的语言扩展词汇和基本模型视图有关。

UML 允许对用例结构化。UML 中的关系包括：

<<Include>>（包括关系）：当系统中几个用例具有公共功能时，可以将该公共功能放入一个用例中，其它的用例可以通过包含关系共享该用例的功能。通过建立一个公共用例，既避免了重复，又使其它用例在需要时可以使用该公共用例。

<<Extend>>（扩展关系）：该关系适用于包含带有选择或条件顺序的步骤的用例。该步骤可以用一个用例描述，其它用例可以在需要时使用。扩展用例一般用于发生条件或选择行为的情况。

<<generalization>>（泛化用例）：用例中的泛化与面向对象规范中类的泛化相似。意思是子用例可以继承其父用例的结构和行为。子用例通过增加新的结构及行为，或者修改父用例的行为实现自身的特色。

也可以使用用例的泛化机制反映用例与角色的关系。角色 2（如下图所示）与角色 1 形成泛化关系。角色 2 继承了角色 1 的全部结构和行为，并且可以增加新的结构和行为。UML 中包含的描述用例的其它额外的机制可以参考文献[2]。图 4 表示了 UML 中用例的标注方式。

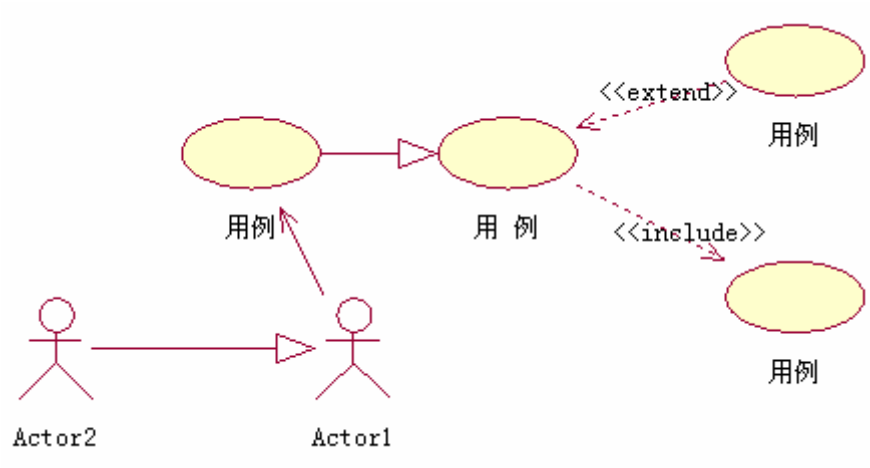


图 4. UML 中的用例标记

用例开发过程始于发现系统角色，并根据这些角色发现与之相关的用例。为每一个角色发现所有可能存在的用例。

第 2 步包含为每一个用例定义基本方法（主场景），并定义可替换的方法（辅助场景）第 3 步校正对用例行为的描述，发现关系（包含关系、扩展关系和泛化关系）。该过程通常采用迭代或增量方式进行。在定义了系统中包含的所有角色和用例后，采用如图 4 的标注符号开发用例图。

在文献[2][25]中描述了一些用于辅助需求工程师进行用例开发的启发式规则。当然，发现和描述用例并不容易，因为在多数情况下，需要需求工程师具有一定的经验。首要的问题是发现哪个角色真正需要从系统中获得服务或功能。

另外一个困难是定义哪个用例真正满足了角色的目标，为角色提供相应的结果。文献[10]中提出了一个辅助该过程的可行方案。Cockburn 认为项目参加者拥有被开发系统有关的目标。这些目标可以具有更高或更低的层次，可以源自于预期系统的用例。

然而，主要的困难在于系统目标如何才能尽早发现。一般需求工程师在完成该工作时，主要运用他们的经验发现这些目标，然后描述用例。既然这样，我们有理由相信可行的方法是，通过 I*组织模型研究目标和其它元素开始用例的发现过程。使用组织模型中定义的信息，有利于用例的开发并且使这个过程更加系统化。我们认为文献[2][25]中描述的辅助用例开发的启发式规则令人难以满意并且使开发过程混乱。另外，在多数情况下，用例的开发没有仔细考虑由组织定义的组织需求。因此，本文提出用例的开发应当开始于对由 I*表示的组织目标和其它要素的分析和考察。

4 从组织建模出发开发用例

前面我们已经描述了 I*框架有利于尽早对业务领域组织模型中存在的关系的理解。在继续进行开发时，需要将关注的焦点放在预期系统的功能和非功能需求上，并对早期需求阶段考虑的问题进行选择。在需求阶段的后期，第 1 步采用用例描述系统的功能需求。一般，UML 中用例的开发没有有效地考虑前面所定义的组织需求。我们认为，从 I*实现的组织建模出发开始用例的设计，将有助于需求工程师建立预期系统的功能需求与组织建模中建立的组织目标之间的关系。另外，通过对组织化模型进行面向目标的分析，可以捕获和映射组织角色到系统用例的目标、意图和动机。对每个与主目标有关的用例，它表示用户要达到的目标，该目标是用例执行的结果。我们认为，用例场景的描述可以从分析组织模型开始，因为组织模型对项目参与者来说是已知和能够理解的。这种方式可以避免与预期系统的需求不一致，模糊和不确定的情况发生。

为了指导集成 I*组织模型与用例的映射过程，我们定义了一些指导原则。这些原则需要按照图 5 所示的步骤进行。图 5 中，第 1、2、3 步表示了发现系统角色、角色的用例及这些用例的场景的行为。集成过程的输入（与步骤 1 相连的矩形框），由 I*框架开发的策略依赖和策略原理模型构成。在第 1、2 步中，输入是策略依赖模型。用例场景的描述（步骤 3）可以从策略原理模型中表达的元素获得。集成过程的结果（与步骤 3 相连的矩形框）是预期系统的用例图和每个用例场景的文本描述。在每个步骤中，提供了辅助需求工程师完成该步骤的指导和规则。这些规则有助于需求工程师完成对策略依赖模型和策略原理模型的面向目标的分析，进而为获取和精练用例提供帮助。

最后提出一些从 I*组织化建模出发开发用例的启发式规则。

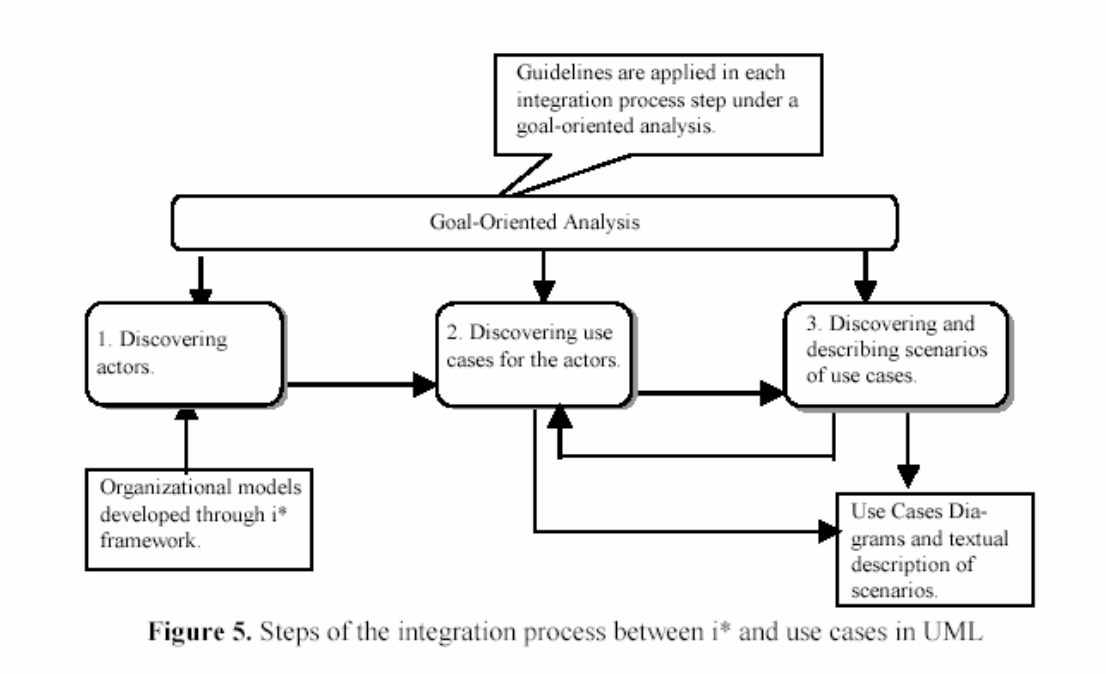


Figure 5. Steps of the integration process between i* and use cases in UML

图 5. I*与 UML 用例集成过程步骤

建议步骤 1：发现系统角色

指导原则 1：I*中每一个角色都是用例角色的潜在候选者，因此要对每一个角色进行分析。例如，我们可以分析图 2 中的会议参加角色。

指导原则 2：对 I*中角色的分析应该在预期系统外部。用例中的角色不应该是软件系统。会议参加角色在系统外部是因为它将与预期系统交互以满足会议调度。

指导原则 3：如果角色放于系统外部，应该保证 I*中的角色是用例图中角色的候选。由此，进行下列的分析是必要的。

指导原则准则 3.1：I*中的角色依赖一定与预期系统的视图有关。例如，I*中的会议参加角色可能被映射到用例角色，考虑与它有关的依赖，刻画它与系统的交互非常重要。会议参加者与会议发起者之间的依赖表明会议参加者有责任参加和提供会议调度系统中用到的信息。

指导原则 4：在组织模型中通过 IS-A 机制描述并且为用例中角色独立地映射的 I*中的角色，将通过泛化关系关联到用例图（应用指导原则 1，2，3）。例如，图 2 中描述的会议参加者与重要参与者间的 IS-A 关系可以在用例图中映射为泛化关系。

建议步骤 2：为角色发现（寻找）用例

指导原则 1：对每个通过步骤 1 获得的角色，应当从被依赖者（Dependee）的角度考察其所有的依赖，以此为角色寻找用例。例如，通过考察在 I*中的依赖关联。一些用例可能与会议参加者角色有关。

指导原则 1.1：评估 I*中与角色有关的目标依赖；通常，I*中多数目标可以被映射到用例的目标上。例如，在图 2 中，会议参加者与会议发起者之间的目标依赖 AttendMeeting(p,m)可以映射到 AttendsMetting 用例中，该用例包含由会议参加者参加会议需要完成的几个步骤。

指导原则 1.2：评估与角色有关的资源依赖；此时，最重要的问题是：如果角色依赖另外的角色获取资源，获取该资源的目标是什么？如果存在多个抽象目标，可能该目标是角色需要的用例目标的候选目标。例如，观察作为被依赖者的会议参加者，包含两个资源依赖 ExclusionDates(D)和 PreferredDates(p)。这些资源可以被映射到会议参加者的 DefineAvailDates 用例中，包含定义有效的会议调度日期的步骤。我们认为获取这些资源主要的目标是定义会议参加者参加会议的有效日期。

指导原则 1.3：评估与角色有关的任务依赖；在此情况下主要的问题是：如果一个角色完成任务依赖其它角色，应该分析是否该任务包含在一组任务中，用于完成角色更抽象的目标。该目标指向那里？例如，在图 3 中，任务依赖 Organize Meeting 与会议发起者角色关联，可以由会议发起者角色的组织会议用例映射。表示了组织会议的系统步骤。

指导原则 1.4：评估与角色有关的软目标依赖；一般，I*中软目标依赖是系统的非功能需求。因此，软目标不会用来表示系统的用例，但是非功能需求与系统用例有关。例如，会议发起者和重要参加者之间的软目标 Assured(AttendsMeeting(ip,m))可以映射到与用例 AtttendsMeeting 有关的非功能需求中。非功能需求表明，需要确保重要参加者参加会议。

指导原则 1.5：考虑改进 I*模型用以检查角色间依赖关系的变化。这些变化可能表示系统的新用例。例如，在图 2 中，对资源 Agreement(m, p)，关系中的依赖者是会议发起角色。在图 6 中，我们处理最近的需求，软件是会议调度者被介入。因此，依赖者变为会议调度角色，它表示会议调度系统。这意味着在图 6 中，会议调度角色依赖会议参与角色同意提出的会议时间。比较来看，如图 2 所示的会议调度的手工过程是会议发起者角色，它依靠关于对会议日期达成的一致。在此情况下重要的是分析目标的目的以及获取 Agreement(m,p)资源。因此，我们可以定义用例 Agreement，该用例包含会议参加者同意提出的时间所几个需要的步骤。

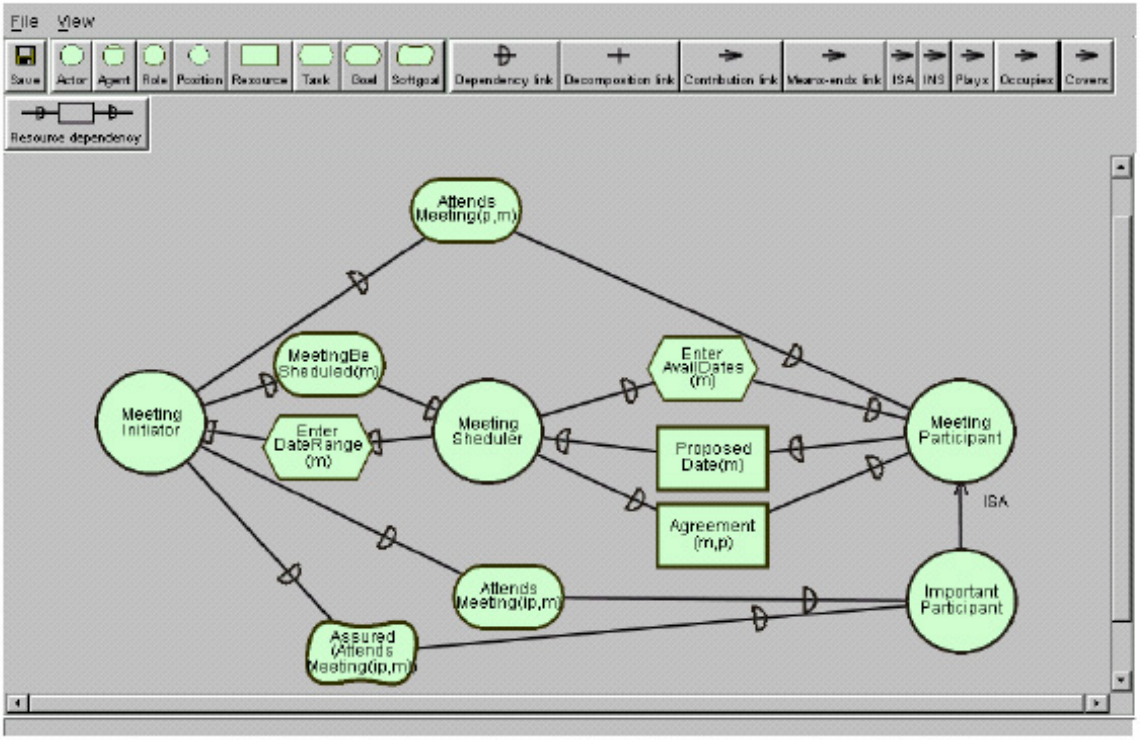


Figure 6. Strategic Dependency Model for the meeting scheduling with computer-based scheduler.

图 6. 基于计算机的调度系统的策略依赖模型

指导原则 2：对特定情况进行分析，根据步骤 1 提出的建议发现角色及其用例，分析处理关于表示预期系统或其部分的 I*框架中各个角色，根据依赖产生用例。例如，图 6 中会议发起者和会议调度系统之间的目标依赖

MeetingBeScheduled 为会议发起者角色指出了用例 ScheduleMeeting 的定义，表达了角色对系统的使用，描述了会议调度过程的细节。

指导原则 3：在提出的目标的层次上（业务、上下文或用户目标）对用例与其主要目标的关系进行分类。此分类有助于确定可能存在的新用例。业务目标表示与业务过程有关的高级意图，在组织环境中组织用户过程。例如目标“在尽可能短的时间内组织会议”。根据上下文，目标表示满足商业目标的一个潜在的对象。例如，“通过软件系统实现调度”。最后，获取用户目标导致为组织角色使用软件系统直接发现相关的功能和价值。示例可以是目标会议参加者希望参加会议。

建议步骤 3：发现和描述用例的场景

指导原则 1：分析策略原理模型中每个角色以及关系，获取可能是产生角色的用例场景描述的信息。例如，分析图 3 的策略原理模型。从会议发起者角色的角度考虑，核实会议调度任务分解成 ObtainAvailDates、FindSuitableSlot、ObtainAgreement。由于软件系统目标是完成会议调度，可以认为这些任务是完成会议调度需要的高级步骤。因此，可以为会议发起者角色定义用例 Schedule Meeting。该用例可能包含关于需要从每个会议参加者、有效的会议日期获取的步骤（主场景描述）；

为了全面改进指导，将它们应用到会议调度问题。回忆图 2 描述的会议调度者的策略依赖模型，它没有考虑会议调度软件的存在。另外，在示例研究中也考虑通过表示会议调度软件的角色的存在对早期的模型进行改进。考虑图 3 中的模型，它表示策略原理模型与图 5 中策略依赖模型的关系。通过这三个模型构成的组织模型用于发现和描述会议调度系统的 UML 用例。

下列步骤根据图 5 提出并应用了适当的规则，我们有：

- 分析图 6，可以发现用于开发用例的候选角色。根据在第 1 步中提出的指导，验证其中一个被分析的角色不满足规则 2。会议调度角色是一个系统，是将要开发的软件。因此，该 I*的角色不能作为一个用例的角色。相应地考虑其它的 I*角色，因为它们的策略依赖涉及与会议调度系统开发有关的方面。因此，候选用例角色包括：Meeting Initiator，Meeting Participant，Important Participant。进一步分析这些候选者，可以发现 Important Participant 是 Meeting Participant 的类型（IS-A 关系）。根据指导原则 4（第 1 步），我们认为该角色是 Meeting Participant 的特例。由此在 Meeting Participant 与 Important Participant 之间建立泛化关系。与 UML 中用例图定义的标记一致。

继续从 I*组织模型中发现用例，按照图 5 定义的过程，为每个角色发现并关联用例。按照第 2 步中的规则，可以校验如下：

对于会议参加者角色（Meeting Participant），注意到该角色是依赖者，可以指出一些来自角色依赖关系的用例（指导原则 1）。首先，分析作为依赖者角色的目标依赖。在图 2 中，验证目标 Attendmeeting(p,m)，它表示需要会议参加者角色参加会议的需要。该目标源于用例 AttendMeeting。完成该目标需要几个相应的步骤。一般来说，该目标是一个用户目标，考虑规则 3 提出的目标层次的定义。用例目标的实现为会议参加者角色带来了相应的结果，允许相同角色参与会议。通常，该用例的主场景的描述将表示系统中其它新的用例的用户目标。

继续分析，从图 2 中得知，有三个资源依赖与 Meeting Participant 角色有关，分别是：ExclusionDates(p)，PreferredDates(p)和 Agreement(m，p)。根据指导原则 1.2，可以确定获取 Agreement(m，p)资源的主要目标是产生一个被调度的每个参与者日期协议。看来在达成该协议过程中，每个参与者同意提出的带有调度约束或持续时间会议时间。当然，协议也可以包含对其它可能采用的日期的分析。换句话说，为了获得时间协议，首先应该在系统与会议参加者角色之间包含几个交互步骤，可以为会议参加者角色定义称为 Agreement 的用例。

我们也应该分析(图 2 中的)资源依赖 ExclusionDates(p)和 PreferredDates(p)。使用指导原则 1.2 可以分析资源 ExclusionDates(p)和 PreferredDates(p) 需要获取由会议参与者角色定义的有效日期有关的更加抽象的目标。该目标来源于会议参与者角色的用例 DefineAvailDates。

为会议发起者角色寻找侯选用例，根据与会议参与者相同的指导（建议步骤 2）。与会议发起者角色相关的依赖有两个。在图 2 中，会议参与者与会议发起者之间的资源依赖 ProposedDate(m)表明会议发起者应该为会议调度提供一个建议日期。考察图 6，注意到依赖 ProposedDate(m)是会议调度系统的一个属性。该资源将从初始化系统的操作发起，根据先前的判定定义并没有与会议发起者之间的交互发生。因此，会议日期将由系统提出，该目的不成为验证一个用例的定义（参考指导原则 1.2 及 2）。

其他与会议发起者角色有关的依赖是任务依赖 EnterDateRange(m)，见图 6。根据指导原则 1.3，可以发现为会议调度提供日期范围的任务非常简单，因为该任务仅仅寻找由先前为会议发起者角色提供的信息。因此，任务 EnterDateRange(m)可能将成为一个描述与会议调度目标有关的用例的步骤。该依赖不是直接来源于对会议发起者的用例。

因为对会议发起者没有其它可用的依赖。将按照指导原则 2 继续进行。分析图 6，考虑会议发起者与会议调度者（软件系统）之间的目标依赖 MeetingBeScheduled。非常清楚，为完成会议调度，会议发起者角色将与系统交互以完成会议调度。因此，可以定义用例 Schedule Meeting 来表示会议发起者对系统的使用情况。在该用例中描述会议调度过程的细节。

在使用了建议指导（步骤 2）后，为会议发起者找到了用例 Schedule Meeting，为会议参加者找到了用例 DefineAvailDates，AttendsMeeting 和 Agreement 用例。下面开始描述主场景和辅助场景以及用例之间的关系（建议步骤 3）。在此，策略依赖模型，主要是策略原理模型被用做描述场景和用例关系的信息资源。

例如，为会议发起者角色发现的用例 Schedule Meeting 表示了会议发起者为完成会议调度而对系统的使用情况。该用例应当包含所有需要的步骤以调度会议，这些步骤从会议发起者向系统提供如调度会议的日程安排这样的信息开始。基于会议发起者所提供日期范围，系统要寻找对所有会议参加者都有效的日期，提供一个有效的日期列表以供选择。该过程产生一个一致同意的日期并与所有参与者确认该日期。对用例 Schedule Meeting，我们可以建立如下的主场景步骤：

用例：调度会议（Schedule Meeting）

角色：会议发起者（Meeting Initiator）

用例目标：完成会议的调度安排工作

主场景：

1 用例开始于会议发起者向系统提供会议日期范围的信息。

2 系统向所有会议参加者发出请求，获得一个基于会议发起者提供的会议日期范围内的有效日期列表（用例 DefineAvailDates 包含在该步骤中）。

3 系统按照参加者提供的有效日期和发起者提供的日期范围寻找一个一致的时间列表。

4 基于一致的时间列表，系统定义一个会议开始日期并与参与者协商。

5 会议发起者期望系统获得调度会议日期的一致许可，包含当所有参与者同意该日期的会议调度过程（用例 Agreement 包含于该步骤中）。

对该用例的描述所用到的信息主要来源于图 3 表示的策略原理模型。该用例中步骤 1 的基本信息是从任务依赖 EnterDateRange 中获得的，建立会议发起者提供日期范围的需要，该要求发生在会议调度期间。考虑建立日期范围的过程非常简单，对该目标不需要建立另外的用例。

步骤 2 和步骤 3 从对任务调度会议的分解中获得（与图 3 的会议调度者有关的）。步骤 2 源于对任务 ObtainAvailDates 的分析。用例 ObtainAvailDates 是被包含的（<<include>>），因为它表示参与者获取有效日期列表需要的步骤。步骤 3 产生于对目标 FindAgreeableSlot 和任务 MergeAvailDates 的分析。该步骤表示系统为会议调度产生的一致同意的时间的列表定义的内部行为。

步骤 4，从对与任务 Schedule Meeting 有关的资源依赖 ProposedDate 的分析获得（图 3）。给定在图 2 至 6 中定义的信息，可以设想 Proposed day 可以被系统发现，先前使用的一些由会议发起者建立和定义的评价标准，作为示例的基础，优先于会议组织。

步骤 5 从为会议调度选择日期以达成一致的系统中获得。该信息源于对任务 ObtainAgreement 的分析，并与资源 Agreement(m，p)关联。先前为系统发现的用例，我们认为获取每个参加者同意提出的日期的目标，需要在每个会议参加者和和调度者之间完成一些交互的步骤。这些步骤应该在用例 Agreement 中描述。基于以上的分析，该用例被包含于步骤 5 中。

我们可以用同样方法描述其他用例，并采纳在该示例研究中所提供的过程和准则。

采用提出的准则应用于示例研究后，我们可以使用 UML 用例图定义如图 7 所示的会议调度系统的用例模型。当发现新的关系时，对发现的用例的描述仍然可以修改和补充。另一个重要的方面是其它用例的开发依赖于需求工程师的经验。然而，对属性的建模范围广阔，其目标是方便理解以及在用户与开发者之间就系统需求的定义达成共识。

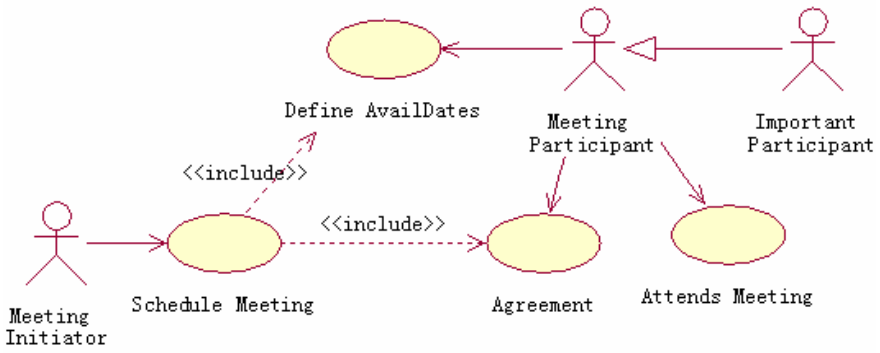


图 7. 会议调度系统的用例图

因此，使用 I*组织模型并按照提出的准则，可以为用例的开发获取有用的信息。正如前面所指出的那样，用例发现和描述可以通过更加详细分析组织模型和启发式规则而得到改进。注意到本文中提出的建议比[24]提出的更全面，更加系统化是非常重要的。主要的提高是采用了面向目标的分析，并以此指导从 I*到用例的映射技术。我们提出的最新的指导版本有助于需求工程师从 I*中与用例有关的目标、资源、软目标和任务中，从如何表示这些元素或关联预期系统的目标的角度发现目标。

最后，我们也考虑了发现新的用例，在 I*角色和特定的用于表达预期软件系统或部分系统 SD 模型中的依赖。

5 结论

本文提出了一些启发式规则，用于寻找使用 I*框架开发的组织模型与 UML 用例集成，该方法显示了良好的生命力。对组织模型和 UML 技术都进行了描述，提出的指导原则部分应用到会议调度模型中。从示例研究开始可以分析和考查在策略依赖模型中存在的信息，策略原理模型可以作为用例开发的基础。另外，更加详细地考察组织模型，使得需求工程师可以获得软件开发的最佳选择以及稳定完成组织目标的用例。在传统的开发方法中，没有有效地考虑用例和场景，以及系统的动机、目标和选择等。组织模型的运用有助于这些活动的开展。

采用本文提出的集成组织模型和用例的方法，一些重要的问题，如系统如何完成组织目标，以及为什么需要它，有哪些可能的选择，在包含的各个部分中存在何种联系等等可以更好解决并提出了与软件系统开发融合的解决方案。

一些相关的工作包括在 Tropos 框架[6][19]和 I*提出需求驱动的开发方法和 pUML 图集成[1]。这些工作认为组织模型是开发更可靠软件的基础，它能够满足用户和组织的实际需要。未来的工作包含以下几个方面：

改进目前工作中提出的指导规则。主要包括：i)更加系统化的描述指导规则，使得指导规则可以帮助需求工程师将从 I*软目标依赖中获得非功能需求[8]，与通过 UML 描述的用例功能需求关联起来；i i)合并 I*和用例的集成/映射过程，结构化 I*（主体，角色和形势）中角色的机制。

研究和分析 Tropos[6][7][19]框架，寻找并发现在该项目中表达的可以改进我们目前工作的解决方法。

将其它的在文献[1]和项目 CREWS[22][23]提出的基于场景的技术与我们提出的方法进行扩展。

研究其它面向目标建模方法[11][21][23][28]作为场景发现和描述的方法。

定义启发式规则用于辅助需求工程师更加有效地分析 I*组织模型，进行场景开发以捕获、分析和文档化系统需求。

参考文献

[1] Alencar, F., Castro, J., Cysneiros, G., Mylopoulos, J., “From Early Requirements Modeled by i* Technique to Later Requirements Modeled in Precise UML”, In Proceedings of the “III Workshop de Engenharia de Requisitos”, pp 92-108, Rio de Janeiro, (2000).

[2] Booch, G., Jacobson, I., Rumbaugh, J., “The Unified Modeling Language User Guide” ,Addison-Wesley (1999).

[3] Brooks, Frederick P., "No silver bullet; essence and accidents of software engineering", Computer, Vol. 20, No. 4, IEEE, pp. 10-19, April, (1987).

[4] Bubenko, J. A., Extending The Scope of Information Modeling, Proc. 4th Int. Workshop on the Deductive Approach to Information Systems and DataBases, Lloret-Costa Brava, Catalonia, Sept. 20-22, 1993, pp73-98.

[5] Bubenko , J, A., Kirikowa, M., “Worlds” in Requirements Acquisition and Modeling, Sweden, (1994).

[6] Castro, J., Kolp, M. and Mylopoulos, J., Developing Agent-Oriented Information Systems for the Enterprise, Proceedings of the Second International Conference on Enterprise Information Systems (ICEIS00), Stafford, UK, July, (2000).

[7] Castro, Jaelson F., Kolp, M., Mylopoulos, J., A Requirements-Driven Development Methodology In: CAISE’01, The 13Th Conference on Advanced Information Systems Engineering, 2001, Interlaken, Suiça. Proceedings of the 13th Conference on Advanced Information Systems Engineering (in Press). Heildelberg, Alemanha: Springer Lecture Notes in Computer Science, 2001.

[8] Chung, L., Nixon, B.A.,Yu, E., Mylopoulos, J., Non-Functional Requirements in Software Engineering (Monograph), Kluwer Academic Publishers, 472 pp, (2000).

[9] Clarke, Edmund M., Jeanette M. Wing et al. Formal Methods: State of the art and future directions. ACM Computing Surveys n. 4, vol. 28. 1996.

[10] Cockburn, A., Writing Effective Use Cases, (Pre-Pub. Draft #3), Humans and Technology, Addison-Wesley, (2000).

[11] Dardene,A., Lamsweerde, V., Fikas, S., Goal-Directed Requirements Acquisition. Science of Computer Programming, 20, pp. 3-50, 1993.

[12] D'Souza , Desmond F., A.C. Wills, Objects, Components and Frameworks with UML: The Catalysis Approach, Addison-Wesley, November, (1998).

[13] DeMarco, T., Structured Analysis and System Specification, Englewood Cliffs, New Jersey: Prentice-Hall, 1979.

[14] Eriksson, Hans-Erik., Penker, Magnus., Business Modeling with UML: business patterns a work, John Wiley & Sons, 2000.

[15] Jacobson, I., Booch, G., Rumbaugh, J., The Unified Software Development Process, Addison-Wesley (1999).

[16] Jacobson, I., Object Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley (1995).

[17] Leite, J.C.S.P, Rossi, G., Balaguer, F., Maiorana, V., Enhancing a requirements baseline with scenarios. In Proceedings of the Third IEEE International Symposium on Requirements Engineering – RE97, pages 44-53. IEEE Computer Society Press, January, 1997.

[18] Mylopoulos, J., Chung, L., and Yu, E., “From Object-Oriented to Goal-Oriented Requirements Analysis,” Communications of the ACM, 42(1), pp. 31-37. January 1999.

[19] Mylopoulos, J., Castro, J., “Tropos: A Framework for Requirements-Driven Software Development” , Brinkkemper, J. and Solvberh, A. (eds), Information Systems Engineering: State of Art and Research Themes, Lectures Notes in Computer Science, Springer-Verlag, June 2000.

[20] Mylopoulos, J., Chung, L., Wang, H.Q. , Liao, S., Yu, E., `Extending Object-Oriented Analysis to Explore Alternatives' IEEE Software.

[21] Potts, C., Fitness for Use: the System Quality that Matters Most, Proc. Third Int’l Workshop Requirements Engineering: Foundations of Software Quality REFSQ’97, pp. 15-28, Barcelona, June, 1997.

[22] Ralyté, Jolita., Rolland, C., Plihon, V., Method Enhancement With Scenario Based Techniques, To appear in Proceedings of CAISE 99, 11th Conference on Advanced Information Systems Enguneering Heidelberg, Germany, June 14-18, 1999, (CREWS Report Series 99-10).

[23] Rolland, C., Souveyet, C., Achour, C. B., Guiding Goal Modeling Using Scenarios, IEE Transactions on Software Engineering, Vol 24, No 12, Special Issue on Scenario Management, December, 1998.

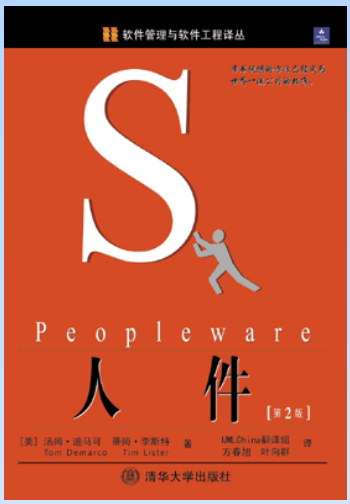
[24] Santander, V. F. A., Castro, J. B., Desenvolvendo Use Cases a Partir de Modelagem Organizacional, III Workshop de Engenharia de Requisitos (WER) , Rio de Janeiro, 13-14 de Julho, 2000.

[25] Schneider, G., Winters, J. P., Applying Use Cases: a practical guide, Addison Wesley, (1998).

[26] Sommerville, Ian., Peter Sawyer, Requirements Engineering: A good practice guide, John Wiley & Sons, 1997, ISBN: 0 471 97444 7.

[27] Yu, Eric, Modelling Strategic Relationships for Process Reengineering, Phd Thesis, University of Toronto, (1995).

[28] Yu, E., “Why Agent-Oriented Requirements Engineering”, Proc. Third International Wokshop Requirements Engineering: Foundations of Software Quality REFSQ’97, pp 171-183, Barcelona, June, 1997.



Tom Demarco，Tim Lister

翻译：UMLChina 方春旭 叶向群

http://www.peoplewarecn.com

人件中文版网站

Frederick P. Brooks, Jr--《人月神话》第 19 章

近年来，软件工程领域的一个重大贡献是 DeMarco 和 Lister 在 1987 年出版的《人件》，我衷心地向我的读者推荐这本书。

Edward Yourdon

《人件》在 1987 年出版后，立即成为最畅销的作品。当人件第一版出版时，我写了一份评论，“我强烈推荐你买一份《人件》给你或你的老板，如果你是老板，那么为你部门的每一个人买一份，并给自己买一份”。这建议在 12 年后的第二版依然有效，并且更加热烈。

Mark A. Herschberg

我只学了办公环境的章节，就辞掉了原来的工作！

Joel Spolsky

我想，微软成功的原因之一就是公司里的所有经理都读过《人件》

Steve McConnell

由于本书第 1 版的赫赫声名，新版的《人件》是我不用看就会决定购买的少数几本书之一。

预订中文译本>>

复杂软件驱动系统的 UCM 与 UML

Daniel Amyot 著，殷建民 译

查看评论

摘要

UCM（Use Case Map）表示法允许以高级因果脚本的方式描述复杂软件驱动系统。通过在抽象组件的结构上添加脚本路径的办法，UCM 提供了系统级的行为与结构的完整视图。本文介绍了 UCM 中的一些与 UML 图相关的有趣特征，还展示了 UCM 如何能够在架构级上将脚本与结构捆绑在一起，如何帮助动态系统可视化，如何支持架构推导，如何帮助在用例与顺序图、活动图、状态图之间的概念缺口上搭建桥梁。

1 引言

复杂软件驱动系统有许多类型，包括面向对象的、基于代理的、实时的和分布式系统。它们具有下面的许多属性——大规模、协同性、分散控制、及时性、可靠性、变化多端特色丰富的功能、运行时组织的流畅性以及系统的升级需求等，这些属性使得它们无论从技术还是从管理复杂性的角度来看都是难以理解的 [12] [25]。这些复杂系统经常用在电信、防卫、宇航和工业控制领域 [6]。

统一建模语言（UML）是一种通用目的建模语言，它可以用于详细说明和构造软件系统（特别是面向对象和基于组件的系统）工件并使其可视化与文档化，也可用于业务建模和非软件系统 [27]。它包括用于各种模型描述与文档化的许多概念和表示符，并且拥有技术和工业团体的坚定支持。

作为 UML 的重要特色，用例（Use Case）被定义为某一特定用户（执行者）看得见具体结果的系统运行动作序列 [27]。在过去几年里，用于脚本和用例的各种表示法，以及基于它们的设计过程，已经非常流行了。例如，“Rational 统一过程”就是一种用例驱动的（Use-case driven）基于 UML 的方法学。在这种方法中，用例将五类模型（需求、分析、设计、实现和测试）捆绑在一起，这种模型描述了系统的局部表示。UML1.3 允许使用九种不同的图描述复杂软件驱动系统及其模型，每一种图提供了特定角度的模型观点，每一种图在语义上必须与所有其它的图一致。本文中，这些图被分为两类。第一类被称作**行为图**（Behavioral diagram），主要着眼于系统的功能和动态方面，它由五种类型的 UML 图组成：

用例图（Use case diagram）：展示了执行者、用例以及彼此之间的关系，以用户的观点描述系统功能。

顺序图（Sequence diagram）：描述了对象之间按前后次序排列的交互模型，它来源于信息序列流程图。

协作图（Collaboration diagram）：展示了系统的总体结构和交互行为。

状态图（Statechart diagram）：展示了特定内容的状态空间、引起状态变化的事件以及产生这种结果的动作等。

活动图（Activity diagram）：从操作方面捕捉系统的动态行为，着眼于内部进程驱动的流程。

第二类被称作**结构图**（Structural diagram），与系统构件及静态特征有较大关系，它包含四类 UML 图。

类图（Class diagram）：捕捉系统的词汇表，展示了系统的各种实体以及彼此之间的总体关系。

对象图（Object diagram）：一个运行系统的“快照”，展示了特定时刻的各种对象实例（带有数据值）以及彼此之间的总体关系。

构件图（Component diagram）：展示了软件构件之间的依赖性。

配置图（Deployment diagram）：展示了运行时刻进程元素的结构以及与之相伴的软件构件、进程和对象。

UML 包含了两类图之间的几种隐含的连接（例如，顺序图和协作图可以使用类图中定义的实体）。但 UML 并没有强调许多系统构件协同工作时（例如，跨越网络的事务处理）出现的大规模行为单元的首要的和紧密的描述方式。

本文描述了一种被称为“用例映射图”（Use Case Map，UCM）的制图技术，作为一种以外在的、可视的方式联接行为与结构的手段。UCM 是第一流的架构实体，它描述了捆绑到底层的、组织化的抽象构件结构的各种职能（responsibility）之间的因果关系。本文试图图解 UCM 怎样帮助在用例模型中的用例和分析设计模型中的其它行为图（顺序图、状态图、活动图）之间的概念缺口上搭建桥梁。UCM 还提供了从行为图中的各种活动到结构图中的各种构件（以及对象）组织之间的鸟瞰图，这将使贯穿系统设计发展全过程的架构推导成为可能。

虽然本文并不打算成为 UCM 表示法的指南（建议有兴趣的读者参看 [8] [11] [13] [28] 以获取更详细的资料），但它图解了 UCM 中对 UML 有潜在帮助的几个特性。第 2 节通过一个简单的电话系统的例子，展示了 UCM 表示法的概貌以及它与用例的关系。第 3 节和第 4 节分别讲述了 UCM 与行为图、状态图的几种关系。第 5 节讨论了 UCM 的现状及其未来的发展。第 6 节是整篇论文的结论。

2 用例映射图（UCM）

2.1 基本表示法

UCM 用于强调一个系统的最实质、最引人注目的、最关键的功能。沿因果路径的各种职能可以是某一构件内部的、也可以是外部可见的。UCM 可以表现特定的脚本，也可以被抽象，并且可以覆盖多个脚本实例。使用 UCM 时，脚本建立在构件间消息交换的上层，所以它们不必捆绑到特定的组织结构上。UCM 提供了以路径为中心的系统功能视图，并且提高了脚本的可重用性的级别。

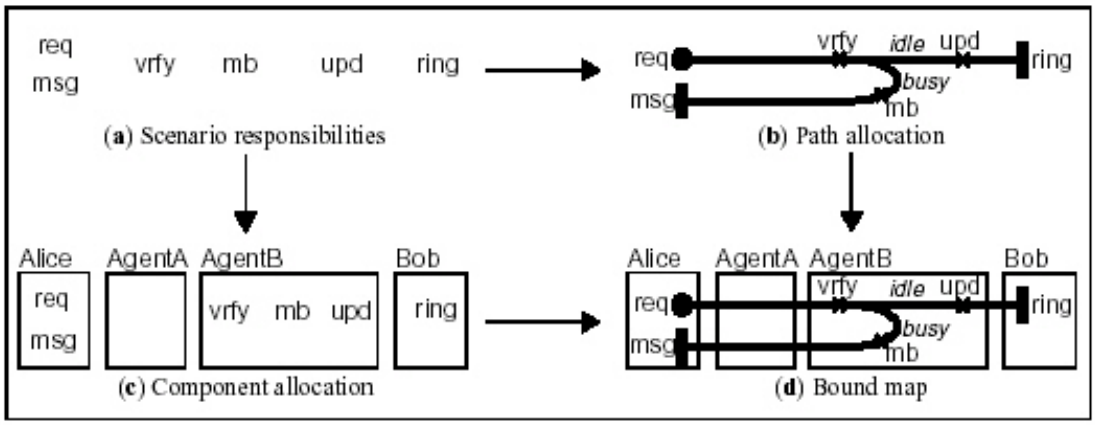


Fig. 1. Use Case Maps construction.

图 1(d)显示了一个简单的 UCM。其中，一个用户（Alice）试图通过代理网络建立与另一个用户（Bob）之间的电话呼叫。每个用户有一个代理负责管理预订的电话功能，比如源呼叫屏蔽（OCS）。Alice 首先通过她的代理同网络发出连接请求（req），这一请求引起被呼代理检查（vrfy）被呼当事人是“空闲”还是“忙”（图中的“条件”是斜体的）。假如空闲，一些状态将被更新（upd），在 Bob 一方，振铃信号将被激活。否则，一个声明 Bob 无法通话的消息将准备好（mb）并被回送给 Alice（msg）.

脚本以一个触发事件和（或）一个初始条件（标号 req 的实心圆）开始，以一个或多个结束事件和（或）后继条件（杠）结束（在我们的例子中是 ring 或 msg）。我们将连接原因与结果的路径称为路由（route）。中间的各种职能（vrfy，upd，mb）沿此路径被激活。可以将职能看作任务或者是被执行的计算功能。在这个例子中，各种职能被分配到不同的抽象构件中（Alice，Agent A，Bob 和 Agent B 等盒子），这些构件可以看作对象、进程、代理、数据库，甚至可以看作角色、执行者或人。我们将这种叠加图称作约束图（bound map）。

UCM 的构造可以用多种方式完成。例如，一开始可以确定各种职能（图 1(a)），虽然在一个如此简单的图中是不必要的。然后将这些职能分配给脚本（图 1(b)）或构件（图 1(c)）。构件可以沿路径发现。最后，两个视图合并形成约束图（图 1(d)）。

图 1(d)是一个相当简单的图，但它以一种紧凑的形式传递了大量的信息，并且允许需求工程师和设计者使用二维信息（结构和行为）选择其系统架构。

2.2 附加表示法

为了进一步介绍 UCM 的符号元素，可以在这个基本用例中增加几项新功能。

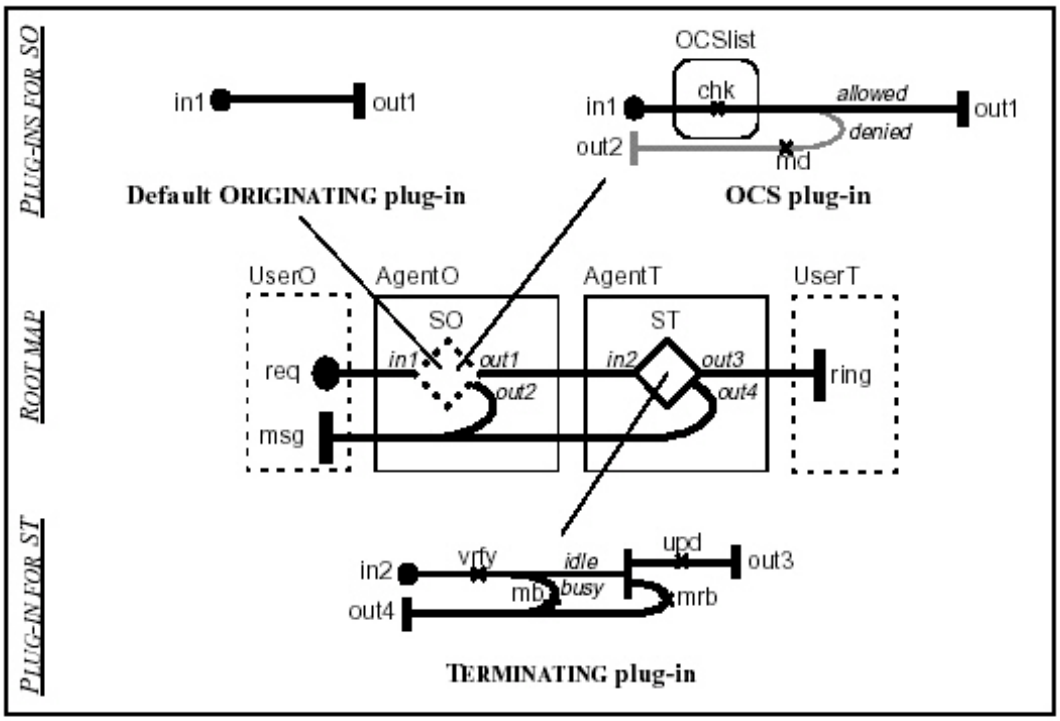


Fig. 2. More complex call connection and new notation elements.

图 2 由图 1 中介绍的实例抽象而来，构件不再涉及 Bob 和 Alice，而是更一般的主叫方和被叫方（无论用户还是代理）。虚的构件称作“槽”（Slot），可以由不同时刻的不同实例来填充。这些实例可以扮演构件的某一特定类的角色。在 [8] [11] 中，Buhr 介绍了一种可处理不同类型的构件和更加丰富的语义（活动进程、被动对象、分组、对象池、中断服务请求、代理、互斥等）的体系化表示法。他还论述了怎样将这些构件与对象类联系起来，由于它们的定义超出了本论文的范围，下面的例子仅仅提供几点直觉的说明，这儿所涉及的构件的自然属性实际上与本论文强调的 UCM 特性并不矛盾。

图 2 的中部展示了图 1(d)中 UCM 的增强版，它表现了与用例实例有关的整个类。这种 UCM 被称作根图，因为它拥有一些含有各种子图（称作插件）的“ 容器 ”（称作桩-Stub），桩有如下两种：

静态桩:表现为一个普通的菱形（见 ST 桩），它们仅包含一个插件程序，因此允许复杂图的层次分解。

动态桩:表现为一个虚的菱形（见 SO 桩），它们可能包含几个插件程序，运用时根据选择方针（通常用预设条件描述）确定具体选择，也可能同时选择多个（顺序地或并行地），尽管这样需要 UCM 图之外的更详细的说明。

输入和输出桩的路径段已经在根图中标明（斜体标号）。虽然它们并不要求以可视的方式显示出来，但它们的存在有助于完成插件与桩的明确链接。例如，主叫方的 SO 桩有两个插件（ORIGINATING 和 OCS）。ORIGINATING 插件的起点 in1 被接到输入路径段 in1，终点 out1 被接到输出路径 out1。为了显示插件和桩之间的清楚的链接关系，图 2 使用了相似的标号。但在通常情况下，名字是不同的，并且这种关系必须明确描述。

OCS 插件显示了一个新的构件（被动对象 OCSlist），它表现为一个屏蔽号码列表，这些号码禁止主叫用户（User O）接通，假如 User O 预订了源呼叫屏蔽服务，则选择 OCS 插件代替 ORIGINATING 插件。在这种情况下，将检查被呼号码是否在屏蔽列表之中（chk），如果呼叫被拒绝，相应的消息被送回主叫方（md）.

TERMINATING 插件在原来的 UCM 的基础上作了一些改进，在被叫方更新（upd）和振铃的同时，返回主叫方一个回铃信号（mrb）。在这儿用一个 AND 分支表示并发性。在 UCM 表示法中，选择性路径（就象 TERMINATING 插件中的 OR 分支、OR 连接）、并发性路径（AND 分支和 AND 连接）、职能共享、例外路径、计时、故障点、错误处理、以及路径之间的同步／异步交互作用等，都有相应的命名，除了个别元素之外。

通过在集成视图中为桩选择插件，我们可以得到一张展开图，它依然包含了多种可能的头尾相接的脚本。一旦桩被定义为路径上的一个关键点，很容易加入新的插件，这些插件在我们的例子中体现了新的功能。现有的图和插件可以进一步分解或使用新的路径及新的静态桩和动态桩进行进行扩张（例如，当加入一个完全不同的服务时）。

2.3 拼版中的 UCM 和用例

UML 用例定义为用例实例的集合，而用例实例则是某一特定执行者看得见具体结果的系统运行动作序列 [22]。用例通常以普通文本的方式描述，虽然有时这种表示法可以被活动图、状态图、前置条件或后置条件等其它行为描述技术所替代。当描述系统和有关的外部执行者之间的交互作用时，用例通常将系统看作内部不可见的黑盒子。由于用例的实现过程体现在行为图中，而在行为图中系统内核被提炼为子构件，因此在用例与实现之间存在一个大的概念缺口，使用现行的 UML 图去研究这么一个缺口和这么大的一个图片经常使人迷惑，因为将不同类型的许多图中的许多详细资料集成在一起需要很高的智力。

图 3（改编自 [12]）显示了现行 UML 拼版的各块，而 UCM 正是缺少的那块。有了 UCM，不再必须将其它块合在一起，不再必须理解这么一张大图。UCM 并不仅仅看作一个附加步骤，更重要的事是，它可以追踪从着眼于系统行为的用例（黑盒）到最终着眼于构件行为的更详细的行为图（玻璃盒）的足迹，因此可以看作一个合理的灰盒（gray box），我们使用“灰盒”这一术语表示某些设计信息是可见的。

另外，UCM 表示法还包含一些表达动态情形的功能，它能够以一种紧凑的方式描述整个系统。首先，构件的动态组织可以从代码结构和配置方面抽象出来，用桩、构件池和系统级的动态职能（这里不作讨论）来表示，其次，随时间变化的脚本模型可以用桩和插件表示，如图 2。

本文并未宣称使用现行的 UML 图和过程在上述概念缺口上搭建桥梁或表示动态情形是不可能的。但是，UCM 在解决这一迷团使“大图”可视化方面似乎具有独特的优势。下一节进一步图解 UCM 中与现有 UML 图有关的几种优势。

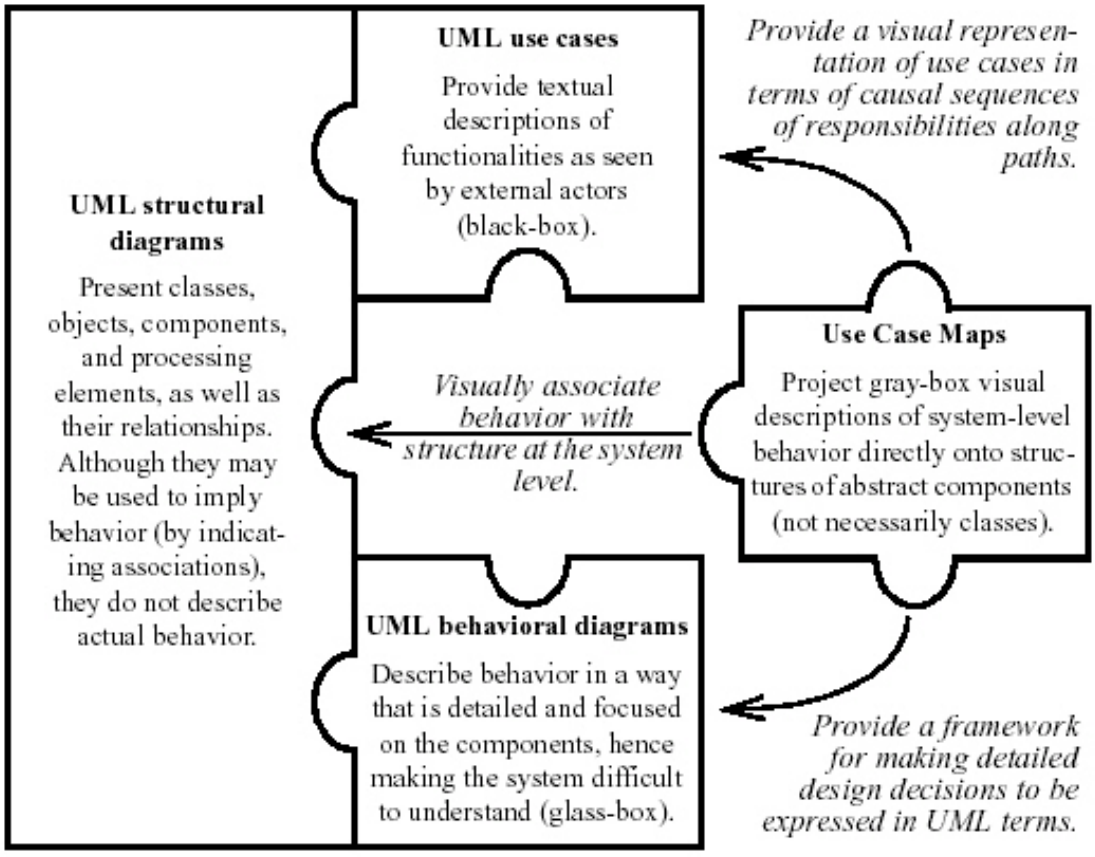


Fig. 3. UCMs as a missing piece of the puzzle.

3 UCM 和行为图

3.1 UCM 和用例图

用例图显示了角色、用例及其关系。它们特别致力于捕捉用例模型中的功能需求或现有功能，但也可用于其它类型的模型。

UML 用例是黑盒描述，而 UCM 更象灰盒，因为它显示了系统的某些细节（例如，抽象构件的拓朴结构、动作的内部流程等）。象用例一样，UCM 与系统外部角色通信时（尤其在起点和终点），使用信号、事例和消息，但当 UCM 与系统内部元素通信时，可能使用其它的通信语义。在这一抽象级别上，不会过早地作出系统详细设计的决定。这一决定要留给使用更恰当的表示法（例如顺序图）的其它模型。

UML 用例图可以使用用例之间的三类关系，也就是包含关系、扩展关系和泛化关系 [22]。UCM 表示法在此基础上作了很大的扩展，它以一种紧凑的方式，全面展示了用例及其关系：

包含关系：通过隔离和封装复杂的细节（以便使用例的实际意义明显化）和提高一致性（通过分解包含在几个基用例之中的行为）的办法，目的是帮助阐明一个用例。

在基用例路径上放置一个静态桩，就可以实现包含关系。这个桩将其细节隐含在它的插件中（内含用例），而这个插件可以在多个桩中重用，因此改进了 UCM 的一致性。包含点的位置在路径上以可视的方式表述，许多静态桩可以用于表现多种包含。例如，图 2 中的 ST 桩和 TERMINATING。

扩展关系：扩展的目标是表明一个用例的某一部分是（潜在地）可选择的，即在一定条件下（有时是异常条件）才执行一个子流程，或者有一系列的行为段，其中一个或几个可以插入基用例中的扩展点。

这种关系在 UCM 术语中用 OR 分支表示，可能有多于两个的选择。OCS 插件的拒绝路径和 TERMINATING 插件的忙路径（图 2）都是它们各自基用例的扩展。UCM 可使用路径标签、颜色、阴影、加粗等可视化标志强调原始的基用例（以便区分来自可选的或异常事件的基本流程），否则会迷失在繁乱的图中。举个例子，OCS 插件的许可路径（粗线）代表基用例，相反的拒绝路径则是一个扩展。UCM 表示法还为异常的、超时的和错误处理的路径提供了其它的可视化标识。

动态桩展示了扩展关系的另一层次。这种桩可能有一个缺省的行为（一个通常包含空路径的插件），它可以被其它插件覆盖。选择某一非缺省插件的条件在选择方针中描述，例如，图 2 中 SO 桩有一个缺省插件（ORIGINATING）。当预订者的 OCS 功能被激活时，OCS 插件取代了缺省插件。

UML 用例明确定义了其它行为可以插入的扩展点。UCM 中无此概念，因为任何路径段都是潜在的扩展（例如，一个 OR 分类）的隐含扩展点，而对于动态桩来说则是外在的扩展点。

用例泛化：当两个或多个用例在行为、结构和目的方面有共同点时，其共享部分随即又可描述为这些子用例所专用的父用例。

拥有公共段和公共目标的各种 UCM 脚本可以被集成为 OR 分支和 OR 连接的结合体，或者更有可能被泛化为一个多分支动态桩。父 UCM 代表了原来的用例图中的公共部分，它包含一些拥有行为分支（插件）的动态桩。子 UCM 被父 UCM 所构造，而它的桩被适当的插件所占据。但是，从多个父 UCM 生成子 UCM 时（多重继承），在定义插件以及子 UCM 作用这些插件的方式之前，需要先将各个父 UCM 集成。

作为一个例子，一个基本呼叫的 UCM 可以看作是图 2 中根图的简化版，其中缺省的 ORIGINATING 插件占据了 SO 桩，而 TERMINATING 占据了 ST 桩。但一个 OCS 呼叫将在 SO 桩中使用 OCS 插件。无论是基本调用还是 OCS 调用，都是它们的父 UCM（根图）的子 UCM。父 UCM 的结构和行为已被子 UCM 继承和修改。

3.2 UCM 和交互图

UML 定义了两类交互图。**顺序图**显示了触发事件沿纵向时间轴（称作生命线）的外在顺序，比较适合于实时说明和复杂脚本。**协作图**显示了实例之间的关系，有助于理解一个给定实例的所有作用，并且有助于程序设计。它们在本质上包含了类似的概念，但以不同方式表达。这一节主要着眼于顺序图。

UCM 能够帮助从（用例模型中的）用例得到（分析和设计模型中的）交互作用图。UCM 并没有明确定义构件之间的消息交换，但消息的构造应使得来自不同构件的职能之间的因果关系明确化。构造消息通常有多种方式，依赖于所用的接口、信道和协议。

图 1(d)中 UCM 抽象而来的基用例的因果关系路径<req，vrfy，upd，ring>,可以作为一个例子。图 4(a)中，这一顺序图被链接到相同的构造层，我们所加入的明确的信道（线）约束了每一消息的潜在发送者与接收者。关于协议与控制的不同决定可以导致多种解决方案。

图 4(b)显示了四个并发实体通过简单协议通信产生直接信息交换的情形。但是，假如两个代理之间使用了更复杂的协议（例如，协商协议），并且这一控制被认为是有差别的，那么就该采纳图 4(c)中的源自相同因果关系路径的顺序图。哪个图是最适当的，依赖于设计决定，这一决定并不在 UCM 级别上作出，它需要在具有进一步追踪能力的适当模型中以文档形式表现出来。

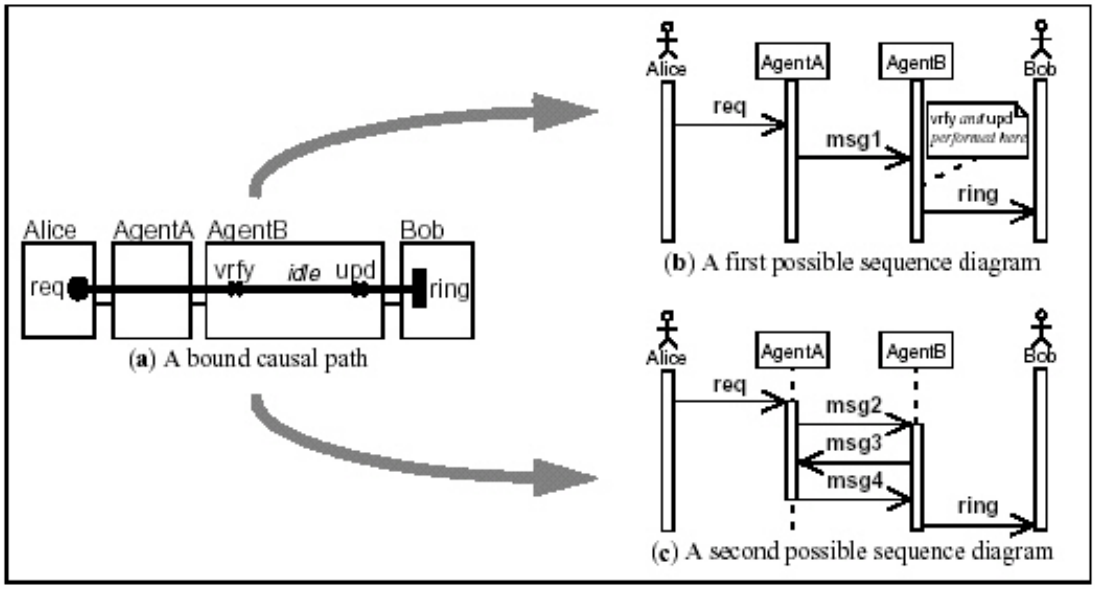


Fig. 4. Admissible sequence diagrams derived from a UCM path.

有几篇论文已经图解了从 UCM 中如何引出有效的消息序列图（MSC）[1] [4] [5]。基本的 MSC 实质上与序列图类似。在 [7] 中作者介绍了一个从 UCM 到高级 MSC 的映射、一种允许 MSC（带有构造序列的）递归构造的表示法、并发性、可选择性、叠代性以及其它的东西（基本 MSC 有着类似的构造，但对于消息来说，子 MSC 是不同的）。

3.3 UCM 和状态图

“状态机”形式上是 Harel 状态图的基于对象的变种 [15]。它混合了 ROOMchart 中定义的几个类似概念，也可看作 ROOM 建模语言的状态图的一个变种 [24]。

有了状态图，焦点直接转到了构件行为。UCM 并不替换这些图，但可以指导其构造。UCM 脚本中的路径段（可能很多）被连接到一个构件，这个构件需要被集成化以便确定其逻辑和状态。与此同时，覆盖不同构件职能之间的因果关系路径的综合需求被提炼为消息交换的形式，穿越构件边界的路径段也可以帮助描述构件接口。状态可能被类图中定义的（也可能是独立定义的）可利用的对象类所影响。需要在 UCM 的抽象构件和构造状态机的对象、进程、模块之间建立一种映射。此外，这一综合过程可以导致许多有效的解决方案。因此，设计决定需要在适当模型中被激发（可能被 UCM 之外的需求）并且要以文档的形式表现出来。

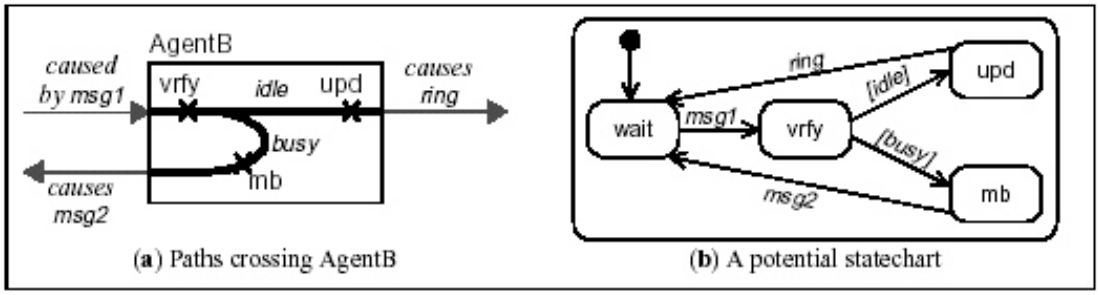


Fig. 5. Potential statechart diagram derived from UCM paths

图 5(a)展示了从图 1(d)穿越构件 Agent B 的 UCM 路径。图 5(b)图示了这一路径的潜在状态图。在这里，职能、保护信息和消息已经分别被映射到状态、条件转换和正常转换。这一特定的例子假定代理有自己的线程，并且在一开始就等待一个特定消息。很明显，不同的假定和需求将导致不同的状态图。

从 UCM 路径到状态图的映射并不总是直接的。例如，图 2 的构件将要求更复杂的状态图以便集成多种插件（Agent O）、集成多种路径起点和终点（Agent O）并且覆盖并发性路径（Agent T）

直接从 UCM 产生状态机要迈出很大的步伐，经常用顺序图作为中间步骤。在顺序图这一级别，将做出与提炼构件之间因果关系有关的决定。状态机还需要集成这些顺序图以便覆盖每个构件所扮演的不同角色。

从 UCM 生成有限状态机的方法，在 [20] 中有所介绍，高级 MSC 用作一个中间步骤。UCP 还可导向其它类型的通信实体。在 [1] [2] [5] 中，作者使用正式的代数语言 LOTOS[18]为基于构件的系统行为建模，而代理行为（对于高级原型）则在 [9] [10] 中生成。

3.4 UCM 和活动图

活动图是状态图的特殊情形，它着眼于内部进程驱动的流程（相互对于普通状态图中的外部事件）。因此，从本质上说，它代表了一个过程或一个业务工作流的状态机。活动图更强调动作顺序和条件而不是执行动作的构件。

活动图享有 UCM 的许多概念（甚至于享有 UCM 的许多表示符）。UCM 中的“职能”类似于“活动”。两种表示法都支持动作序列、可选择性和并发性。起点和终点也有类似的用途。

一个复杂的活动可以被提炼出来成为另一个活动图，正象 UCM 使用桩进行路径分解一样。但桩似乎更通用，桩允许多个输入和输出路径，而且动态桩还准许使用基于某种选择方针的许多插件。在表示复杂系统的动态行为和结构时，UCM 的桩被证明是一个非常有用的创造。

UCM 的优势之一体现在职能与构件之间的连接能力。活动图通常并不以这种方式使用，虽然它支持二者之间的有限程度的映射。活动图可以被可视地化分为“泳道”，每条泳道与其相邻的泳道被其两边的纵向实线分离。每一泳道代表了全部活动中的其中一部分职能，最终可由一个或多个对象实现。每个动作被分配到一个泳道。“转移”可能会跨泳道，但转移路径的路由并不明显。泳道可用最简单的形式解释为构件，它们是一维的，并且不以任何方式（例如，根据它们的相关位置或真实属性）显示构件之间的关系。UCM 提供了包含这一信息的集成化鸟瞰图。这一鸟瞰图，对于理解表现为路径和（动态）职能的行为如何影响和修改动态系统中构件的运行结构，几乎是必须的。

4 UCM 和结构图

4.1 UCM 和基于构件的图

UML **构件图**是被依赖关系所连接的各种构件的图。某些构件也可能以物理包含的方式连接到另外一些构件，这种物理包含体现了构件之间的复合关系。UML **配置图**显示了运行进程元素的结构以及软件构件、进程，还有与其相关的对象。UML **对象图**呈现了运行系统的“快照”，因为它展示了某一特定时刻的对象实例及其关系。

缺省的 UCM 构件表示法，就象 Buhr 和 Casselman 在[8]中所定义的那样，具有高度的概括性，它能够展现这三类 UML 结构图中所发现的许多重要特色。它们可以图解包含关系和其它不同类型的（被动的，主动的，等等）依赖关系，甚至于可以展示运行时刻的实例（无数据）。但是，UCM 的主要优势在于它具有以静态图的方式描述动态结构的能力。借助于桩、构件池，以及具有动态职能的 UCM 路径，构件可被创造或撤清，可以四处移动，可以让其它构件看到，等等。带有这种构件的 UCM 可以用一种外表静态和简练的方式表示复杂的动态结果。如果没有 UCM，则需要用 UML 基于构件的图的许多快照来表述。

在 UCM 路径下，我们并不阻止其它的结构化表述。这种路径可以用在几类基于构件的图的顶部，就象在 UML 或类似的表示法中一样。

4.2 使用 UCM 进行架构推导

UCM 可通过功能（用例）和形式（结构）的连接进行架构选择的早期评估。通过在结构中减弱功能的办法，我们可以将功能与形式同时考虑，也可以分别考虑。前面的 UCM 图图解了相同构件结构的不同路径，UCM 还允许我们重用不同的可选结构中的相同路径。例如图 6(a)象图 4(a)一样重用了相同的因果路径，但是在不同的构件集上。这儿没有涉及通信代理，而是采用不同的依赖机制（例如，通信连接）将职能分配给更加传统的电话构件，如开关和服务节点(SN)。这将依次导向另一个不同集合的有效顺序图和不同的状态机，从而进一步延续设计周期。

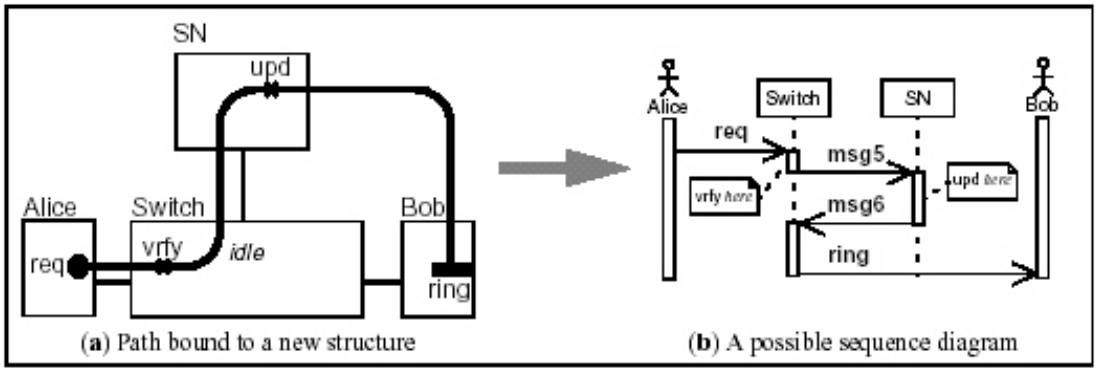


Fig. 6. Admissible sequence diagrams derived from a UCM path.

由于 UCM 路径能够很容易地从结构中减弱，从而改进脚本的可重用性，并且导向行为模型，这种模型可以用于广泛的应用软件。在许多场合，UCM 可以提供有帮助性的可视模型，它可以触发关于系统问题的思考和讨论，并且可以被重用[11]。

还应注意到架构选择的评估是在抽象高层进行的，并没有象在顺序图中那样的关于消息与协议的任何早期约定。当潜在的结构被修改时，修改顺序图需要付出更多的努力并可能造成浪费。

架构推导还需处理系统需求的改进。复杂系统很少由草图建立，相反，它们是通过不断接收新技术和新功能而不断改进的。正如 Velthuijsen 在[29]中展示的那样，新功能的增加并不是单调的，它们能够并且将要改变现有功能的操作。新技术也能够改变一些基于功能的假定。UCM 提供了处理系统改进的非单调特性的一种机制（例如，桩和插件），而且还展示了实践中 UCM 如何辩别“分解”（例如，小规模对象、线程、进程、模块、包等）和“分层”（例如，操作系统、信息栈、网络中间件等）[8][11][12]。这两种体系概念的区分可以帮助处理系统的可升级性和可维护性。

5 讨论

5.1 UCM 的语义和工具

UCM 的语义和良好的规则是以超图的方式定义的[21]。XML 中所表示的 UCM 文本的线性形式[31]，同样已经被定义[3]。这种形式适合于不同工具的输入和文档的生成。有了 XML 文档类型定义，UCM 与即将制定的 UML 标准可以很容易地集成。这些标准包括 XML 元数据交换（XML）[17]、UML 交换格式（UXF）[26]等。

UCM 导航器（一个构造和编辑 UCM 的工具），使用超图语义和规则提供可靠的转换以确保构造正确的图[21]。这一工具支持路径表示和 Bahr 的构件表示，并且使用 XML 形式作为文件格式。它可用来产生嵌套的桩和插件，可以很容易地将职能连接到构件，可支持代理系统和表演模型的扩展表示，可生成支持 PDF 的 PostScript 文档。这一工具可用于三种平台（Linux、Solaris 和 HP-UX）

5.2 UCM 用于正式确认及验证

虽然 UCM 拥有半正式的语义，但能够指导为复杂系统生成更正式的模型和详细说明。最近几年，围绕着从 UCM 引出 LOTOS 详细说明[18]的问题[1][5]，已经作了大量工作。LOTOS 是一种代数语言，它甚至可以在缺少构件结构的情况下使 UCM 中发现的事件序列正规化。这将允许需求、规格说明与设计的正式确认及验证，其中的某些东西正是许多面向对象的 CASE 工具所缺少的。其它目标形式包括 ROOM[7][24]，不久还将包括 SDL[2][19]。还应注意，如何使用 ROOM 模型实现 LOTOS 详细说明，这一工作近来还在进行[1][4]。

UCM 当前被用于几类工程，其中的一些工程处理与动态代理有关的问题[9][10]（在描述复杂的代理关系时，UCM 被证明是强大的）、电话间不受欢迎的交互行为的探测与避免[5][9][10]、功能测试组件的生成、日益涌现的移动电话服务标准的描述等[1][2][4]。

建立执行模型是 UCM 的另一应用。在[23]中，执行变成了路径的一种属性，而不是通常想象的那样，作为整个系统的一种非功能属性。扩展的 UCM 表示法包含了时间戳记、时间的约束、事件分配以及设备进程与任务的联合等。无论是 XML 生成器，还是 UCM 导航器，都支持这些扩展。有关其它类型的非功能需求的集成化问题，也在研究之中。

5.3 UCM 和 UML 的集成

UML 的应用非常广泛，但一般不能有扩展，因为这些扩展可能不被普遍理解、支持和同意。UML 预定义文件（profile）提供了在特定领域使用 UML 而无需扩展或修改 UML 的标准使用方式。预定义文件是一个预定义的集合，它包括常规、标记值、约束、表示图标等，它们共同剪裁 UML 使其专用于某一特定领域或过程（例如，对象过程模板和业务工程模板）。一个预定义文件并不增加新的基本概念以扩展 UML，与此相反，它使标准的 UML 专用于某一特定环境或领域。

在 UML 中集成了 UCM 的概念，就可以通过剪裁适当的预定义文件实现某些扩展。虽然这样并不要求对 UML 标准作任何修改，但有许多最引人注目的 UCM 概念，很难被现行的 UML 图和语义所覆盖。

在 UML 图集中增加 UCM 表示法是另一种显而易见的选择。这种办法虽然看起来简单合理，但也因此增加了冗余，而在一个稍微大一点的 UML 图中就存在一些冗余。

扩展现有的制图技术（例如，活动图）和语义，支持新颖的 UCM 概念，可以认为是第三种选择。

最后，使用 UCM 替换（或改编）一个或多个 UML 图，也被考虑为一个潜在的选择。但由于当前标准和工具的现有投资，这种方法实现起来是很困难的。

最好的和最适当的选择，仍然是一个研究话题。然而，独立于具体选择方式的 UCM 概念的标准化的实现，似乎是非常重要的。假如这种标准化能够在不远的未来产生，UML 必定是一个杰出的候选者。

6 结论

UCM 与现有的 UML 制图技术密切相关，但它帮助填补了用例和行为图之间的概念性（灰盒）缺口。UCM 还展示了关于架构推导的一种人注目的观点，它尤其适合于复杂的和动态的软件驱动系统。在这种系统中，行为从多个构件中不断涌现，通常很难使其可视化。

这篇论文图解了关于 UCM 表示法与运用的一些最重要的概念。虽然 UCM 也允许人们独立运用行为图和状态图，但它在系统级上为二者建立了一种有用的连接。通过使用桩、插件和动态构件，UCM 在设计周期的早期，推进了架构推导。未被连接的 UCM 路径成为可重用的脚本模型，它可以连接到多个底层构件的结构中。虽然 UCM 定义在比消息交换更高的抽象级别上，但它可以指导生成详细的图（例如，顺序图和状态图），甚至于生成正式的详细说明。

UCM 表示法已广泛应用于不同的工程，所以变得更稳定、更健壮。UCM 工具开始涌现，UCM 用户群已初具规模[28]。UML 可以从 UCM 的许多要领中获益。作为 UML 拼版中最佳位置的那一块，UCM 仍然有待进一步阐明。

感谢

我非常感谢 Ray Buhr 和 Luigi Logrippo 在过去六年中关于 UCM 的大量讨论，非常感谢 Tom Gray 和 Darcy Quesnel 对于早期草案所提出的有用建议。本研究工作得到了 FCAR、NSERC 和 CITO 的部分支持。

参考文献

[1] Amyot, D., Hart, N., Logrippo, L., and Forhan, P.: “Formal Specification and Validation using a Scenario-Based Approach: The GPRS Group-Call Example”. In: ObjecTime Workshop on Research in OO Real-Time Modeling, Kanata, Canada, January 1998.

http://www.csi.uottawa.ca/~damyot/wrroom98/wrroom98.pdf

[2] Amyot, D., Andrade, R., Logrippo, L., Sincennes, J., and Yi, Z.: “Formal Methods for Mobility Standards”. In: IEEE 1999 Emerging Technology Symposium on Wireless Communications & Systems, Dallas, USA, April 1999. http://www.UseCaseMaps.org/UseCaseMaps/pub/ets99.pdf

[3] Amyot, D. and Miga, A.: Use Case Maps Linear Form in XML, version 0.12, April 1999. http://www.UseCaseMaps.org/UseCaseMaps/xml/

[4] Amyot, D. and Andrade, R.: “Description of Wireless Intelligent Network Services with Use Case Maps”. In: SBRC’99, 17th Brazilian Symposium on Computer Networks, Salvador, Brazil, May 1999. http://www.UseCaseMaps.org/UseCaseMaps/pub/sbrc99.pdf

[5] Amyot, D., Buhr, R.J.A., Gray, T., and Logrippo, L.: “Use Case Maps for the Capture and Validation of Distributed Systems Requirements”. In: ISRE’99, Fourth International Symposium on Requirements Engineering, Limerick, Ireland, June 1999.

http://www.UseCaseMaps.org/UseCaseMaps/pub/re99.pdf

[6] Booch, G.: Software Architecture and the UML. Slide package, Rational Software, 1998.

http://www.rational.com/uml/img/arch.zip

[7] Bordeleau, F. and Buhr, R.J.A.: “The UCM-ROOM Design Method: from Use Case Maps to Communicating State Machines”. Conference on the Engineering of Computer-Based Systems, Monterey, USA, March 1997. http://www.UseCaseMaps.org/UseCaseMaps/pub/UCM-ROOM.pdf

[8] Buhr, R.J.A. and Casselman, R.S.: Use Case Maps for Object-Oriented Systems, Prentice-Hall, USA, 1995. http://www.UseCaseMaps.org/UseCaseMaps/pub/UCM_book95.pdf

[9] Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., and Mankovski, S.: “High Level, Multi-agent Prototypes from a Scenario-Path Notation: A Feature-Interaction Example”. In: H.S. Nwana and D.T. Ndumu (Eds), PAAM’98, Third Conference on Practical Application of Intelligent Agents and Multi-Agents, London, UK, March 1998, 277-295.

http://www.UseCaseMaps.org/UseCaseMaps/pub/4paam98.pdf.

[10] Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., and Mankovski, S.: “Feature-Interaction Visualization and Resolution in an Agent Environment”. In: K. Kimbler and L. G. Bouma (Eds), Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW’98), Lund, Sweden, September 1998. IOS Press, 135-149.

http://www.UseCaseMaps.org/UseCaseMaps/pub/fiw98.pdf.

[11] Buhr, R.J.A.: “Use Case Maps as Architectural Entities for Complex Systems”. In: Transactions on Software Engineering, IEEE, December 1998, pp. 1131-1155.

http://www.UseCaseMaps.org/UseCaseMaps/pub/tse98final.pdf

[12] Buhr, R.J.A.: “Use Case Maps and UML”. Slide package, Carleton University, December 1998. http://www.UseCaseMaps.org/UseCaseMaps/pub/ucm_umlSlides98.pdf

[13] Buhr, R.J.A.: “Making Behaviour a Concrete Architectural Concept”. In: 32nd Annual Hawaii International Conference on System Sciences (HICSS’99), Hawaii, USA, January 1999.

http://www.UseCaseMaps.org/UseCaseMaps/pub/hicss99.pdf

[14] Hart, N.: Protocol Validation and Implementation: A Design Methodology Using LOTOS and ROOM. M.Sc. thesis, SITE, University of Ottawa, Canada, April 1999.

[15] Harel, D. and Gery, E.: “Executable Object Modeling with Statecharts”. In: Proceedings of the 18th International Conference on Software Engineering, Berlin, IEEE Press, March, 1996, pp. 246-257.

[16] Hurlbut, R. R.: Managing Domain Architecture Evolution Through Adaptive Use Case and Business Rule Models. Ph.D. thesis, Illinois Institute of Technology, Chigago, USA.

http:// www.iit.edu/~rhurlbut/hurl98.pdf

[17] IBM, Unisys et al.: XMI (XML Metadata Interchange) Proposal, OMG document ad/98-10-05, October 1998. http://www.software.ibm.com/ad/features/xmi.html

[18] ISO, Information Processing Systems, Open Systems Interconnection: LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, IS 8807 (1989).

[19] ITU: Recommendation Z.100, Specification and Description Language (SDL). Geneva, 1994.

[20] ITU: Recommendation Z. 120: Message Sequence Chart (MSC). ITU, Geneva, 1996.

[21] Miga, A.: Application of Use Case Maps to System Design with Tool Support. M.Eng. thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1998.

http://www.UseCaseMaps.org/UseCaseMaps/ucmnav/

[22] Rational Software: Rational Unified Process 5.0, Cupertino, CA, 1998.

[23] Scratchley, C. and Miga, A., “A Use Case Map Editing Tool with Performance Prediction Capabilities”. In: ObjecTime Workshop on Research in OO Real-Time Modeling, Kanata, Canada, January 1998.

[24] Selic, B., Gullekson, G., and Ward, P.T.: Real-Time Object-Oriented Modeling, Wiley & Sons, 1994.

[25] Selic, B. and Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. White paper, ObjecTime Ltd., March 1998. http://www.ObjecTime.com/otl/technical/umlrt.pdf

[26] Suzuki, J. and Yamamoto, Y.: “Making UML Models Exchangeable over the Internet with XML: UXF approach”. In: Proceedings of the First International Conference on the Unified Modeling Language (UML ’98), Mulhouse France, June, 1998.

http://www.yy.cs.keio.ac.jp/~suzuki/project/uxf/index.html

[27] UML Revision Task Force: OMG Unified Modeling Language Specification, version 1.3 beta R1, April 1999. http://uml.systemhouse.mci.com/

[28] Use Case Maps Web Page and UCM Users Group, 1999. http://www.UseCaseMaps.org

[29] Velthuijsen, H.: "Issues of non-monotonicity in feature-interaction detection". In: Third Interntional Workshop on Feature Interactions in Telecommunications Software Systems, Kyoto, Japan, October 1995.

[30] Weidenhaupt, K., Pohl, K., Jarke, Matthias, and Haumer, P.: "Scenarios in System Development: Current Practice". In: IEEE Software, March/April 1998, 34-45.

[31] W3 Consortium: Extensible Markup Language (XML) 1.0. W3C Recommendation, 10 February 1998. http://www.w3.org/TR/REC-xml

人件 - Microsoft Internet Explorer

文件(F) 编辑(E) 查看(V) 收藏(A) 工具(T) 帮助(H)

后退 媒体

地址(D) http://www.peoplewarecn.com/ 转到 链接

《人件》— 老板，别把开发人员当成牲口

» 新闻

《人件》提供预订>>

互动出版网 WWW.CHINA-PUB.COM

Cnforyou.com

华储 www.huachu.com.cn

» 相关文章

关注程序员自己的文化——专访Tom DeMarco

别把开发人员当成牲口

Brooks在《人月神话》中的评论

Alan Cooper的评论

办公室空间，下一场革命

《人件》在计算机行业的实践（待续）

人的问题：关于《人件》

Edward Yourdon的评论

开发人员是人吗?

» 各版本封面

英文，日文，德文

首页 购买 留言板

完毕 Internet

把业务对象连接到关系数据库（下）

Joseph W. Yoder、Ralph E. Johnson 等 著，Happy 译

查看评论

类型转换

别名

数据转换

类型翻译

动机

数据库值类型并不总是和对象类型直接对应，例如，一个布尔值也许在数据库存成 T 或者 F，在 Patient 例子中，性别可以是一个属性，以一个名为 Sex 的类存储，男性实例有某些行为，而女性实例有另外不同的行为，在数据库中也许他们的值是 M 和 F，当从数据库读取这个值，M 需要转换成一个 Sex 类的男性实例，F 需要转换成 Sex 类的女性实例。***类型转换***允许对象值和数据库值之间的转换。

问题

如何将一个没有对应数据库类型的对象映射到一个数据库类型，并反之亦然？

特定约束

- 数据库中的值也许不能映射到指定的对象类型上；
- 对象类型和数据库类型之间存在着阻抗不匹配；
- 应用程序中的值可以被其他应用程序在数据库中编辑和存储；
- 对象属性和数据库字段值能够都存成字符串和数字，这样就减小了阻抗不匹配；

解决方案

通过一个***类型转换***器对象把所有值转换成各自的类型，这个对象知道如何处理空值以及其他对象值和数据库值之间的映射。当一个对象从大型多应用数据库被持久化，数据格式会多种多样，这个模式确保了从数据库中获取的数据能适合于对象。

确保从数据库中读取的数据能够为对象工作是很重要的，同时从对象写入数据库的数据遵从数据库规则和维护数据的一致性也是非常重要的。

每个对象属性通过适当的***类型转换***器传递，为特有的应用系统和 DBM 应用必要的数据规则，在域级别应用的数据规则比在接口级别中应用的要高效得多，这通常是由于所有的域对象都使用一组共同的***类型转换***。

想象一下有些老的数据库没有空值（NULL）的概念，它们实际上为“无数据”的情况存储一个空字符串，当读出这个值，返回一个对应用程序表示“空白数据”的空字符串，他依赖于数据库和该应用程序的规则定义。

这个***类型转换***可以在映射代码中直接完成，数据类型将被转换成适当的对象类型，或者反之亦然，通常，一组共通的转换可以抽象出来。例如，一个 Boolean 对象可以映射数据库中的 T 或 F，Timestamps 能够映射到 String 上，一个 NULL 可以映射到一个空字符串。当你有了这些共通的转换，就可以调用适当的例程为转换预处理类型，这些转换例程形成了策略(Strategy) [GHJV95]的一部分。

根据你的需要，实现是多种多样的，下面的例子中，将所有的转换方法放置在一个对象中，这使方法都存在于一处，如果有必要，允许动态地切换转换对象。这样，也能够用一个策略，为不同数据库应用不同转换算法，如果余下的应用中，转换器不再需要，将是一个更清洁的方式。

另一个选择将扩充每个受影响或被使用的基类，每个对象知道如何将他们转换成数据库必须的格式，假设你使用了多个数据库，那么每个方法都需要适应这个格式之间的差异。

还有一个方式，将你所有的转换例程放在 PersistentObject 中，如果你需要映射到一个新的数据库或是转换改变了，就隔离出不得不修改代码的地方。这个方法存在于任何从 PersistentObject 继承而来的对象，类似前面一种选择，当你有多个数据库时，情形是相似的。

所有提到的这些方法都可以独立于所选的持久机制而工作。

实例实现

当你从一个数据库中字典结构的一行值产生属性时，可以简单地使用：

```
attribute := (aRow at: key).
```

这可以完成任务，然而，如果数据库值不能保证符合对象所需，那么你使用该属性的代码会失败，当你应用了***类型转换***，如：

```
attribute := self typeConverter convertToUpperString:
                (aRow at: key).
```

这个属性值将是应用程序所需的，这会先得到负责转换的类，然后把数据库值传递到一个方法中，确保产生一个大写字符串的属性值。

把准备对象的属性值存储到一个数据库的情形是相似的，你可以简单地将属性值放入一个字符流。

```
nextPutAll: (attribute) printString.
```

这也可以完成任务，然而，如果碰巧一个对象不知道 printString，那么这个代码将会失败。should the database … a conditional statement. 当你应用***类型转换***，如：

```
nextPutAll: (self typeConverter prepForSql:
                (attribtute asUppercase)).
```

属性将被转换成适当的格式，如上所述，负责转换的类被得到并且属性将被转换。

类型转换方法通过 PersistentObject 访问，既可以准备存入数据库的值，也可以将数据库返回的行映射成对象属性。initialize:和 insertRowSql:方法（在每个域对象中）展示了***类型转换***的例子。

Protocol for Type Conversion PersistentObject (instance)

这个方法决定哪一个类负责为表类型和对象***类型转换***类型，这个方法可以延伸为，在运行期决定使用哪个数据库或使用哪个类来转换类型。

typeConverter

“返回负责***类型转换***的类”

```
^TypeConverter
```

Protocol for Type Conversion TypeConverter (class)

这些方法为 PersistentObject 提供一致的格式化的数据值，当从对象向数据库转换时（TypeConverter>>prepForSql:），根据数据库需要来决定和格式化类型；当从数据库向对象转换时，数据库的类型也许不是对象/应用程序所要的，在***属性映射方法***中，每个属性经过***类型转换***以确保数据值是正确的。在有些情况下，当数据库不包含数据时，提供一个缺省值。当若干个不同实现的应用程序使用同一个数据库时，数据规则在数据库级别并不需要强制遵从。

convertToBooleanFalse: aString

“从一个字符串返回一个布尔值，非值为默认值。”

```
^'t' = aString asString trimBlanks asLowercase
```

convertToString: aString

“从一个数据库字符串或字符返回一个字符串，缺省值是一个新字符串。”

```
^aString asString trimBlanks
```

convertToNumber: aNumber

“从一个数据库数字返回一个数字，缺省值是 0。”

```
^aNumber isNumber ifTrue: [aNumber asInteger] ifFalse: [0]
```

这个方法把一个对象转换成正确的数据库格式，放在一个字符流中，对象被测定为什么类型后返回适当的格式，形成 SQL 代码放置在字符流中。这为***持久层***数据类型提供了一个共通的格式。本例中缺省的日期格式假设为（本地当前时间：'%m%d%Y'）。如果你不想使用一个完全日期格式，日期的格式应该根据你的数据库修改。注：这些格式类型被 IBM DB2 UDB V5.0 支持。

```
prepForSql: anObject

    “以字符流的形式返回具有适当格式的对象。”

    anObject isNil ifTrue: [^’NULL’].

    anObject isString

        ifTrue:

            [anObject isEmpty

                ifTrue: [^’NULL’]

                ifFalse: [^anObject trimBlanks printString]].

    anObject isNumber ifTrue: [^anObject printString].

    anObject abtCanBeDate ifTrue:

                [^anObject printString printString].

    anObject abtCanBeBoolean ifTrue:

                [anObject ifTrue: [^’T’ printString]

                                ifFalse: [^’F’ printString]].

    anObject abtCanBeTime

ifTrue: [^self databaseConnection databaseMgr

            SQLStringForTime: anObject].

    (anObject isKindOf: PPLPersistentObject)

            ifTrue: [anObject objectIdentifier isNil

                    ifTrue: [^’NULL’]

                    ifFalse: [^anObject objectIdentifier printString]]
```

结论

- 能帮助确保数据一致性；
- 对象类型可以有很多种，和数据库类型无关；
- 可以替从数据库读出的空值赋缺省值；
- 阻止“不理解未定义对象(Undefined object does not understand)”的错误；
- 为应用程序提供增强的 RMA（可靠性、可维护性和可访问性）；
- 转换类型需要花费时间，特别是从数据库读取大量的值时；

相关或交互模式

- 策略[GHJV 95]可以用来实现这个模式；
- 当构造 SQL 代码时，***属性映射方法***调用***类型转换***；
- ***SQL 代码描述***可能嵌入***类型转换***的调用。

已知应用

- Illinois Department of Public Health TOTS 和 NewBorn Screening 项目；
- ObjectShare 的 VisualWorks Smalltalk[OS 95]使用***类型转换***来在数据库类型和对象类型之间转换。VisualAge for Smalltalk 也在他们的 AbtDbm*应用程序中使用***类型转换***。
- GemStone GemConnect 在数据库类型和对象类型之间转换；
- TopLink 也提供***类型转换***。

变更管理

别名

HasChanged

IsDirty

Laundry List

对象管理器

动机

通常使用一个病人管理系统的方法是为病人调出他的记录和为最近拜访的病人增加一条记录。但是有时候病人的地址已改变了，如果发生了实际改变，系统应该只写回病人的地址。

实现这个情况的方式之一是提供一个单独的改变地址的画面，一个单独的更新按钮用来将地址写回数据库，但这很笨拙并需要维护大量代码；更好的方法是，让病人管理系统的用户在必要时编辑地址，并让系统只写回地址对象的变化和其他用户决定要写回的值。

总的来说，一个***持久层***应跟踪所有 PersistentObject 的变更状态，并应确保他们都要写回数据库中。

问题

如何判断一个对象改变了，且需要保存入数据库？防止对数据库不必要的访问，确保用户知道什么时候一些值被改变了并在退出程序前没有被保存是很重要的。

特定约束

- 大多数对象读出来以后从没修改过；
- 保存没有改变的对象是浪费时间；
- 开发人员经常忘记保存一个修改过的对象，而用户更糟；
- 如果对象的每一个修改都写入数据库的，那么，为了取消用户的请求，就需要另一个写入或回滚操作；
- 复杂对象也许只有一个部件修改了，那无需将它所有的值都写入数据库；
- 将没有被修改的对象写入数据库，将难以稽核到底是谁最后修改了这个对象；

解决方案

设置一个***变更管理器***，来跟踪任何 PersistentObject 修改了某个持久属性，无论何时，请求保存对象都需要这个***变更管理器***。

实现它的方式之一是从 PersistentObject 继承一个类，它有一个脏位，一旦一个映射到数据库中的属性值发生改变就被置位。这个脏位通常是一个布尔值实例变量，表明一个对象是否改变。当这个布尔值设定了，一个保存操作调用时，PersistentObject 将保存新值到数据库中，否则，PersistentObject 将忽略写入数据库。

洗衣列表(laundry list)也是一个用来保存数据的模式，它通过将变更存储在一个洗衣篮中，然后你能够控制对它们做什么，另外它还跟踪哪些属性被改变了，从而只保存脏属性。这样，如果一个应用程序改变了一个表的若干字段，而另一个应用改变了其他字段，你能确保只更新你所改变的字段，他也能够有助于应用程序并发修改数据库表。

根据类体系的不同，实现方案有很多种。一种方案是修改属性设置(setter)方法，可以在对象值修改时设定一个标志，也可以把这个对象加入到已变更对象的洗衣列表，这非常简单却很有效。另一个方案可以使用类似于方法包装(method wrapper)[Brant, Foote, Johnson, &Roberts 1998]，在属性设置方法中做同样的事情。还有一种方案，初始化持久属性成为一种依赖性机制，一旦一个属性值被修改了，可以设置脏位也可将对象加入洗衣列表，并删除依赖性；一脏永脏。另外一种实现方法是使用元数据来描述所有的持久属性，无论何时改变了对象的状态，都能够使用元数据来决定对象是否变脏了，方法包装能够使用这个元数据提供类似的服务。

数据访问一般非常昂贵，应该节约使用，通过标志一个对象是否需要写入数据库将显著提供性能。除了性能的考虑，也有助用户界面在退出前提示用户保存，这个功能使用户更容易接受你的应用系统，这能让他们知道自己忘记保存了已变更信息，而系统会提示他们保存。用户就会认为：如果他们不得不重复输入同样的数据，这个系统便很快没有价值了。

变更管理器的另一个特性是提供对象的初始状态或改变状态的记忆功能，这可以通过 Memento[GHJV95]来实现，如果你系统的用户需要回到初始状态，***变更管理器***能够保留初始值，通过调用一个撤销操作完成。同时，你也还可以为你的系统提供多步撤销操作。

如果你有一个对象，它的某个属性是其他对象的一个有序集合，（在本例中，Name 的属性地址可以是 Address 类的 OrderedCollection），当你删除一个 Address 的实例，数据库如何得到删除数据库行的消息？一种方法是将该实例的键值设为 nil，然后，域对象提供一个 isValid 方法，判定删除特定行，这个方法可以实现，但是应用系统程序员不得不用特定的代码处理所有 nil 实例。一个更好的方式是使用一个删除管理器(Deletion Manager)，当用户按下删除按钮（或其他机制），应用系统程序员把删除的实例放置其中，因为每个对象知道如何删除它自己，该实例会从集合中和数据库中被删除。删除管理器是一个 Singleton[GHJV95]对象，它保留被删除的实例，直到用户发出保存或取消操作。

实例实现

本例将展示一个存取器(accessor)方法如何为 Name 类的首名属性在修改值时设置脏位，这个访问者方法在 VisualAge 中缺省生成了附加的 makeDirty 调用，它将将继承的 isChanged 属性设置为真值。

first: aString

```
"保存首名值。"
self makeDirty
first := aString
self signalEvent: #first
    with: aString
```

Protocol for Change Manager PersistentObject (instance)

这些方法为*持久层*提供改变脏标志的功能，这避免了*持久层*不得不向数据库写入没有改变的数据，同时也向 GUI 程序员提供一个测试对象的方法，以便向用户提供是否保存数据的提示信息。

makeDirty

"表明一个对象需要保存入数据库，如前面的例子中，这个方法可以在 setter 方法中调用。"

```
isChanged := true.
```

makeClean

"表明一个对象不需要保存入数据库或者对象没有被改变过。"

```
isChanged := false.
```

结论

- 用户会更乐意接受这个应用系统；
- 不写入没有被改变的数据到数据库，可以保证数据库有更好的性能；
- 当在数据库间融合数据，可以设定该标志，这样这个记录在必要时将插入新的数据库中；
- 相关或交互的模式：

相关或交互模式

- 状态(State)[GHJV 95]是一个使用布尔值标志的替代方法；
- Memento 可以被***变更管理器***用来支持撤销操作；
- 洗衣列表能够跟踪所有改变了的对象；
- 可以使用删除管理器辅助删除复杂对象的一部分；
- 对象管理器[Keller 98-2]是一个非常相似的模式。

已知应用

- 操作系统为虚拟内存使用脏位；
- 大多数 DBMS 在高速缓存中使用脏位[Keller 98-1]；
- 高速缓存一般都使用脏位；
- Illinois Department of Public Health TOTS 和 NewBorn Screening 项目；
- 在 GemStone OODBMS 中，使用一个***变更管理器***来跟踪一个对象何时被改变了，因此可以知道一个对象何时需要保存到服务器。
- ObjectShare 的 VisualWorks Smalltalk[OS 95]使用***变更管理器***指明一个对象值是否被改变过。VisualAge Smalltalk 也在他们的 AbtDbm*应用系统中使用***变更管理器***。VisualAge 为 GUI 构建者提供图形化联接，以提供一个持久机制。

OID 管理器

别名

唯一键值生成器

动机

只要一个对象被持久化，对象的唯一性是很重要的。在一个面向对象系统中，所有对象都是唯一的，所以给定每个对象一个唯一的标识是非常重要的，它通常称作 OID。***OID 管理器***确保为所有对象生成唯一的键值并存储到数据库中。

问题

我们如何确保把每个对象唯一保存在数据库中，而不管是否和其他对象共享相似的状态？

特定约束

- 您不会希望在一个数据库中改变键值和重复键值，数据库管理员认为这是非常糟糕的事；
- 增加 id 有时是人工的，通常需要在表中增加附加的字段；
- 为分布式数据库创建一个唯一键值；
- 为 SQL 代码明确地标识一条记录而创建一个唯一键值。

解决方案

提供一个 ***OID 管理器***，为所有需要存入数据库的对象创建唯一键值，确保所有新创建并需持久化的对象都能得到一个唯一的键值。当一个新对象需要持久化，它将被写入数据库并且有一个唯一的标识生成，这个生成过程要求非常快速且要求确保标识的唯一性。

一种方案是生成随机数，一旦生成一个数，必须检查它是否已经使用。不过当在多个数据库运行时，无法从本地获知，所需时间增加，而且当从多个数据库融合数据时，键值重复的可能性更大了。

另一个方案是在本地表中存放最后使用的数字，并和一个本地且唯一的键值合并起来，这需要每个键值都读取、写入键值表。

一个更好的方法（本例中使用的）是前面一种方法的变种，可以减少将每个键值都写入表的需要[Ambler 97]。当需要一个键值，向一个 singleton 实例请求，如果这个实例中没有任何数字（例如第一次写入），就从一个表读出一个数字段。一旦返回这个数字，它将立即增加一个特定数（应用程序指定），并写回数据库，给下次其他用户访问用。这个数返回给调用对象并增加 1，存储在内存中直到下次需要一个键值。当这个段的数字都用完了，将重新读入一段并重复上面的过程。

可以使用一个键值生成策略来创建这些键，一个数据库或站点可以使用一种算法，而另一个可以使用其他算法，重要的是键值在表中是唯一的，最好在整个数据库或多个数据库中是唯一的。

有些数据库有生成唯一键值的方法，需要强调的是如果你有多个服务器，生成算法不能冲突。你也可以使用一个 TCP/IP 地址和/或硬盘序列号，跟上其他数字，以确保对象标识的唯一性。同时，有些数据库有生成一序列数字的方法，可以用来作为 OID，不管您使用什么算法，确保线程安全是非常重要的。

注：将这个和 Kellers 和 Browns 模式语言中的 Unique Key 模式联系起来。

示例实现

OID 管理器为***持久层***提供域对象的唯一标识，这个标识可以作为数据库的键值。***OID 管理器***维护一张表，表中有一个数，当它不能向***持久层***返回一个合法键值时，***OID 管理器***增加这个数并写回表中。本例中，当它没有一个合法数字时，从表中获取一段数字维护，直到数字超过写回表的数。本例中段的大小为 10（见下面的 increment 方法）。

Protocol for Accessors OIDManager (instance)

这个方法返回一个值，这个值是当需要一段新数字时，从数据库获取的数字段的大小。它可以被应用系统在启动和不想再设为 10 的时候设置。

increment

```
"返回增加的值。"
(increment isNil)
    ifTrue:[increment:=10].
```

```
    ^increment
```

Protocol for Key Generation OIDManager (instance)

这是本模式的核心方法，从表中读出一个字段值，所有用户访问数据库都使用它。当读取一个值，它立即加上 increment 属性中存放的值，这个值缺省为 10，新数值接着被写回到表中以被下一次使用和下一个用户获取。这个值初始读出并存放在属性中，必要时进行操纵以提供唯一数值，这个数将附加上站点（或数据库）的键值，以创建一个唯一的 14 或 15 位数字的数。

readKey

```
    "生成一个唯一键值，使对象存放入数据库中"
    | newKey aQuerySpec aResultSet aMaxKey prep |
    PersistenceObject beginTransaction.
    aQuerySpec := AbtQuerySpec new
              statement: ' SELECT NUM_SEQ FROM SEQUENCE '.
    aResultSet := PersistenceObject databaseConnection
              resultTableFromQuerySpec: aQuerySpec.
    aMaxKey := aResultSet first at: 'NUM_SEQ'.
    newKey := aMaxKey + self increment.
    PersistenceObject
      executeSql: 'UPDATE SEQUENCE SET NUM_SEQ = ', newKey printString.
    PersistenceObject endTransaction.
    prep := self class siteKey * self keySize.
    self lowKey: prep +aMaxKey.
```

```
    self highKey: prep + (newKey -1).
    ^nil
```

Protocol for Key Retriever OIDManager (instance)

这是获取一个键值的方法，检查这个属性，看是否需要通过上面的方法读出一个数，或是仅仅为这个单一实例增加 1 并返回它的值。

getKey

```
    "这是获取一个键值的方法。"
    self currentKey =0 ifTrue: [self readKey].
    self currentKey = self highKey
        ifTrue:
            [self readKey.
            ^self currentKey].
    self currentKey: self currentKey + 1.
    ^self currentKey
```

结论

- 当写回数据库表时总是增加数字将浪费数字值；
- 当写回数据库表时，如果数字增加不足，将会消耗时间；
- 唯一的单一字段键值可以提高性能，并使 SQL 编码更简单、更易抽象。

相关或交互模式

- 一个 ***OID 管理器***是一个单一实例，所有的持久对象在一个共同的地方生成他们的键值；

已知应用

- Illinois Department of Public Health TOTS 和 NewBorn Screening 项目；
- 所有 OODBMS 都使用一个键值生成算法来为保存的对象创建唯一键值；
- PLoP Registraion 使用 Microsoft Access 序列数据库命令来生成唯一键值；
- 在 DCOM 中，每个接口在运行期通过他们的接口标识（IID）来识别。IID 是 DCOM 为接口生成的一个全局唯一标识（GUID）。GUID 有 128 位，由 DCOM API 函数 CoCreateGuid 生成，这个 API 调用依照由 OSF DCE 指定的算法，它使用当前日期和时间、网卡 ID 和一个高频度计数器。基本上，你可以使用工具（例如 Microsoft Developer Studio）来获得这些 GUID 并将他们放入你的 DCOM IDL。
- CORBA2.0 拥有联合的接口仓库——在多个 ORB 之间操作的仓库。为避免命名冲突，仓库为全局接口和操作分配唯一的 ID（CORBA 中称作仓库 ID）。一个仓库 ID 是一个字符串，有三个层次组成，CORBA2.0 定义了两个格式：

i) IDL 名称有一个唯一的前缀，它的部件组成是：一个 IDL 字符串，跟着一个由'/'分隔的标识符列表和一个主辅版本号。通过':'分隔的第一个标识符是一个唯一前缀——java 使用同样的方式。例如：

IDL:JoeYoder/Foo/Bar/:1.0

JoeYoder 是唯一前缀

Foo 是一个模块

Bar 是一个接口

ii) DCE 通用唯一标识符（UUID），它使用 DCE 提供的 UUID 生成器生成——例如 DCOM。它的部件通过':'分隔，第一个部件是 DCE 字符串，第二个是 UUID，第三个是一个版本号（没有辅版本号）。例如：

DCE:100ab200-0123-4567-89ab:1

事务管理器

别名

工作单元

事务对象

动机

保存对象的过程中，允许对象在某些值不能正常保存时，能以某种方式保存是非常重要的，先前存储对象能够被回滚。在病人例子中，你不仅只存储 Patient 对象的属性，还要存储它包含的变化过的 address 对象，如果试图保存任何一个病人的地址对象失败了，你想回滚保存该病人信息时所有的写入数据库操作。一个***事务管理器***提供开始事务、提交事务和回滚事务的支持。

问题

如何向数据库写入一组对象，当任何写入失败时，将没有任何对象写入。

特定约束

- 对象之间通常以某种方式联系起来，如果一个对象没有正常保存，那您就不想保存相关的对象；
- 知道哪些消息是事务，哪些不存在阻抗不匹配；
- 每个向数据库的写入操作都可以设定为一个单一事务；
- 复杂对象能够构造一个复杂的 SQL 语句，它包含事务处理，可以看作一个工作单元。

解决方案

构建一个和其他***事务管理器***类似的***事务管理器***，这个管理器允许开始事务、结束事务、提交事务和回滚事务。***事务管理器***通常映射到 RDBMS 的***事务管理器***，如果它提供的话，所有流行数据库都内嵌了***事务管理器***，最好是使用它们提供的。

为了性能的原因，也许最好缓冲数据，这种情况，***事务管理器***有提交和回滚缓冲值的功能是非常重要的。如果你映射的数据库不支持***事务管理器***，那么你有必要在开始事务时保存对象的初始状态，这样当请求回滚时，初始值能够写回数据存储。

示例实现

没有事务管理，你将使用如下形式：

```
anObject save.
```

这将提交每条发送到数据库的 SQL 语句，如果在保存 anObject 过程中发生了某种错误，你将在数据库中留下部分 anObject，特别当 anObject 是一个复杂对象时。

当你使用事务管理，基本上你将 anObject 构建的所有 SQL 语句包装起来，告诉数据库，要不保存所有，要不什么都不干，这看起来如下所示：

```
beginTransaction

anObject save

anotherObject save

endTransaction
```

这确保在向数据库提交改变前，所有的保存命令都能成功执行。当你的对象是复杂对象，它知道如何保存它的聚合对象，确保将完整的对象写入数据库中。

下面的代码从 PersistentObject 摘录，它们展示了实现事务管理的包装结果，代码（self class beginTransaction）告诉数据库开始和停止一个事务，它依赖于你所使用的数据库和你所用的开发语言。你也许需要扩展事务以处理你的特殊实现，同时，如果你保存的数据库（可能只是一个文本文件）不支持事务，你将不得不开发一个完整的***事务管理器***以支持用户的需要。本例中将所有保存和删除对象操作包装在一个 beginTransaction 和一个 endTransaction 之中。类方法进行数据库联接类的调用，这是 VisualAge 为它支持的数据库提供提交和回滚事务的地方。

Protocol for Public Interface PersistentObject (instance)

save

```
self class beginTransaction.
```

```
    self saveAsTransaction.
    self class endTransaction.
delete
    self class beginTransaction.
    self deleteAsTransaction.
    self class endTransaction.
```

Protocol for Public Interface PersistentObject (class)

```
beginTransaction
    self databaseConnection beginUnitOfWorkIfError:
            [self databaseConnection rollbackUnitOfWork]
endTransaction
    self databaseConnection commitUnitOfWork
rollbackTransaction
    self databaseConnection rollbackUnitOfWork
```

结论

- 复杂对象要不完全保存，要不什么都不保存；
- 半个对象不能维护引用一致性；
- 所有其他写入数据库的应用系统不得不使用***事务管理器***，你也许不得不增加检查，看在你保存时，是否有其他什么人曾修改过数据库；
- 除非数据库中内嵌了事务管理的支持，写一个完整的***事务管理器***非常困难。

相关或交互模式

- ***事务管理器***和事务对象[Keller 98-2]非常相似，事务对象通过在外面构建对象封装事务，这些事务对象提供开始、提交和回滚事务操作。

已知应用

- 大多数数据库类应用系统提供类似的功能或工作单元的定义；
- Illinois Department of Public Health TOTS 和 NewBorn Screening 项目；
- VisualWorks 在他们的 ObjectLens 中提供***事务管理器***；
- VisualAge 也在 ABTDbm*类中提供事务支持；
- GemStone GemConnect 为数据库对象提供事务管理。

联接管理器

别名

当前联接

数据库会话

动机

只要一个持久对象被读出或是写入数据库，必须建立一个和数据库的联接。一个数据库事务需要的典型信息是数据库名、用户名和用户口令，这些信息可以在每次访问时向用户获取，这将导致失败的用户验证，因为大多用户不会知道数据库名或别名。你也可以在启动时获取用户名和口令，并在以后的代码中使用，建立联接信息，这导致在不需要的时候传递了大量多余信息。最好的方式是构造一个***联接管理器***，存放这些全局信息，并在需要访问数据库的任何时候使用。

问题

持久管理器如何跟踪当前联接的数据库和联接用户？

特定约束

- 所有数据库联接需要访问某个建立联接的地方；
- 引用全局变量可以保持代码简洁明了；
- 维护当前数据库会话，而不是为每个事务创建新会话，将更有效且更易调试；
- 在一个地方提供必要的联接信息和操作可以使代码易于理解和维护。

解决方案

创建一个***联接管理器***对象，存放用户数据库联接所需的所有数据值，共同的值通常是数据库会话、登录到系统的当前用户和其他用于稽核、事务和其他类似的全局信息。

当一个应用系统需要为某些信息只提供一份拷贝，可以使用 Singleton 模式[GHJV 95]或者使用一个单一活动实例和一个 Session[Yoder &Barcalow 97]，这个单一活动实例在 Design Pattern Smalltalk Companion[MORE HERE 98]中有描述。Singleton 通常存放在一个单一全局的地方，例如作为一个类变量或是类本身。不幸的是，这个模式在一个多线程、多用户或分布式的应用时，有时会被破坏。某些情况，需要一个真正的 Singleton，而有的情况，每个线程或每个分布的进程可以看作一个独立的应用，它们每个都有自己的 Singleton，但是当应用共享一个共同的全局环境时，这个单一全局区域不能被共享，这需要一个机制允许多重 Singleton，每个应用一个。因此，***联接管理器***是管理任何和所用数据库会话的 Singleton。

联接管理器和***持久层***交互，以让后者得到当前所需的数据库联接。***联接管理器***也和***表管理器***交互，为任何建立的联接提供所需的数据库表名。这样，可以再一次使用策略[GHJV 95]模式来选择适当的数据库和表映射。想象一下你可能将你的值保存到五个数据库中，为了速度的考虑，你将选择一个负荷最小的，再用一个批处理过程以一个主数据库同步所有库。这个分布式的数据库系统通过一个策略模式方法，确保用户能够得到一个理想的性能。

示例实现

基于从***表管理器***得到的信息，***联接管理器***建立和数据库的联接。当***持久层***请求一个联接，***联接管理器***利用数据库部件通过 ODBC、CLI 或其他可能的方式向数据库提供一个接口。一旦当时不能得到联接，或中途中断了，***联接管理器***提供错误块来进行错误处理，并提供错误收集以记录错误，如有可能就进行恢复。

简单的说，当一个域对象说：

```
self databaseConnection
```

就从***联接管理器***返回联接，联接类型和哪个联接（如果使用了多个联接）由***联接管理器***决定。

在我们的例子中，既选择了本地的，也选择了远程的数据库。我们展示了您将如何从 VisualAge 和 ABTDbm* 类交互，以得到本地数据库的联接。

Protocol for Public Interface ConnectionManager (class)

databaseConnection

“检查局部变量，看建立什么联接，或者看返回什么联接。”

```
^self getLocal
    ifTrue: [self localConnection]
    ifFalse: [self remoteConnection]
```

localConnection

“返回一个联接给 PPLPersistentObject，检查别名看是否已经打开了一个联接，如果没有，创建一个，并提供错误处理，处理联接和联接打开时的错误。”

```
| connection |
(connection :=
    AbtDbmSystem activeDatabaseConnectionWithAlias:'Example') isNil
        ifTrue:
        [connection :=
            ((AbtDatabaseConnectionSpec
                forDbmClass: #AbtOdbcDatabaseManager
                databaseName: 'Example')
```

```
                promptEnalbled: false;
                connectUsingAlias: 'Example'
                logonSpec: (AbtDatabaseLogonSpec
                        id: ''
                        password: ''
                        server: '')
            ifError:[:error|PPLErrorCollector dbConnectionError])
        autoCommit: false;
    yourself].
```

“如果开发环境没有使用错误块，需要提供调试器，在运行期错误块设定为调用错误收集器，它为管理器处理所有 dbm 类错误，上面的错误块是特定于联接的。”

```
    connection databaseMgr
    errorBlock: (System startUpClass isRuntime
    ifTrue: [[:error | PPLErrorCollector dbmError:error]]
    ifFalse:[nil]).
^connection
```

结论

- ***联接管理器***为所有持久对象提供联接信息；
- 当联接信息改变时，只需在一个地方修改；
- 它提供一个支持多联接的方式，可以用于从数据库装载一个对象，或平衡数据库的负荷。

相关或交互模式

- 在多线程、多用户或分布环境中，***联接管理器***是 Singleton 的替代方法；

- ***联接管理器***和会话相似，它维护和数据库的会话以及和数据库联接相关的全局信息。
- ***联接管理器***可以使用一个策略来选择当前联接；

已知应用

- 对 VisualWorks，Lens Oracle 框架和 GemStone 的 GemBuiler 分别有 OracleSession 和 GbsSession，他们都存放事务状态和数据库联接的信息，这个***联接管理器***在同样的数据库环境中被任何对象引用；
- Caterpillar/NCSA Financial Model Framework 有一个 FMState 类[Yoder 97]，当作一个 Session 并维护***联接管理器***;
- VisualAge 有 AbtDbmSystem，维护活动联接的轨迹，***联接管理器***调用 VA 类，依次控制物理联接。
- Illinois Department of Public Health TOTS 和 NewBorn Screening 项目。

表管理器

别名

数据字典

表映射

动机

随着时间发展，对象对应的持久存储也许会发生变化，或者对象有多个存储。一个***表管理器***描述了数据库中表和字段的结构，让开发人员远离这些细节，从而做到改变数据库的命名结构却不影响应用开发人员。

问题

一个对象如何知道使用什么表、什么字段，特别是对象需要存入到多个表中时。当使用多个数据库时，又会多出一个数量级的问题。

特定约束

- 将对象映射到关系型数据库，最终需要将对象值映射到数据库表中；
- 当应用系统扩大和修改时，开发人员希望能够快速修改表名；
- 将对象在不同地方，以不同名称映射到同一个表；
- 很容易地从一个数据库切换到另一个；
- 在描述 SQL 代码的地方直接硬编码表名和字段名非常直接明了。

解决方案

提供一个地方，让对象获取必要的表、字段名以存储自身。当一个对象被存储，在***表管理器***中查找它的表名。通过只在一个地方定义名称字符串，更改和测试修改变得非常迅速和高效。

这个模式使用一个 Sigleton[GHJV 95]，将所需的表或字段名称返回给域对象。这个单一实例包含字典结构存放名称，当一个对象发出请求名称的消息，该实例的方法就通过***联接管理器***检查，决定使用那个字典，使对象可以发送相同的消息，而不管它必须访问那个数据库。

当从多个数据源获取数据，这个模式提供动态改变所访问数据库或表的能力。它灵活地减轻了为多个数据库的相同表一遍一遍编写相同的 ***SQL 代码描述***的工作，而且仍然可以访问它们。

表管理器在某种意义上，为表映射存放元数据，这个元数据应用于任何一个需要表名的时候，它可以用来动态生成数据库查询。基本上，一个 Schema 描述了这个映射，每当访问数据库时都需要使用它。

示例实现

表管理器在字典结构中存放表名，在被请求时提供给 SQL 代码，当***表管理器***初始化时，字典结构用表名初始化，然后，根据***联接管理器***，访问适当的字典并返回表名。

当一个域对象说：

```
self class table,
```

TableManager 将为类返回表的名称。

结论

- 使用*表管理器*的一个好处是，当一个表名修改后，代码中只有一个地方需要修改；
- 另一个好处是，可以用不同的名字访问不同数据库的相同表或视图；
- 编写一个*表管理器*包括要有一种方式解释表和字段的映射，这并不是很容易开发或维护的。

相关或交互模式

- 只要有访问数据库的需要，*持久层*和*联接管理器*将调用*表管理器*；
- ***SQL 代码描述***将使用*表管理器*，以将开发者和数据库的变更隔离开；
- *表管理器*是一个 Singleton。
- *表管理器*实际上只是一个用元数据描述数据库结构的方式；
- *表管理器*能够为复杂映射使用一个翻译器或是为动态系统使用元数据。

已知应用

- Illinois Department of Public Health TOTS 和 NewBorn Screening 项目；
- 很多数据库系统都使用数据字典；
- VisualWorks ObjectLens 使用一个 Spec[Foote & Yoder 98]，结合 Schema 来提供表映射。

综合讨论

现在你已经看到所有的模式，你也许会问，“我如何综合起来用他们？”。所有这些模式一起协作提供一个持久对象映射到数据库中的机制。图 2 展示了模式之间是如何交互的。*持久层*为所需持久化域对象的 ***CRUD***（创建、读取、更新和删除）操作提供标准接口，*持久层*使用域对象提供的 ***SQL 代码描述***构建数据库调用，在生成 SQL 代码时，*持久层*和*表管理器*交互以获取正确的数据库表名和字段名。当已经从数据库返回数据值或是将数据写回数据库时，属性值必须映射到数据库字段名，反之亦然，这由*属性映射方法*完成。*属性映射方法*在 SQL 代码生成中进行某些*类型转换*。属性映射和类型映射在*持久层*实例化一个新对象时也会发生。*持久层*将一个对象发生变化的值通过*联接管理器*保存到指定的数据库中，这个变化的值由*变更管理器*管理；*联接管理器*能够和*表管理器*交互，决定使用哪个数据库。*持久层*在需要进行事务处理时提供对*事务管理器*的访问。

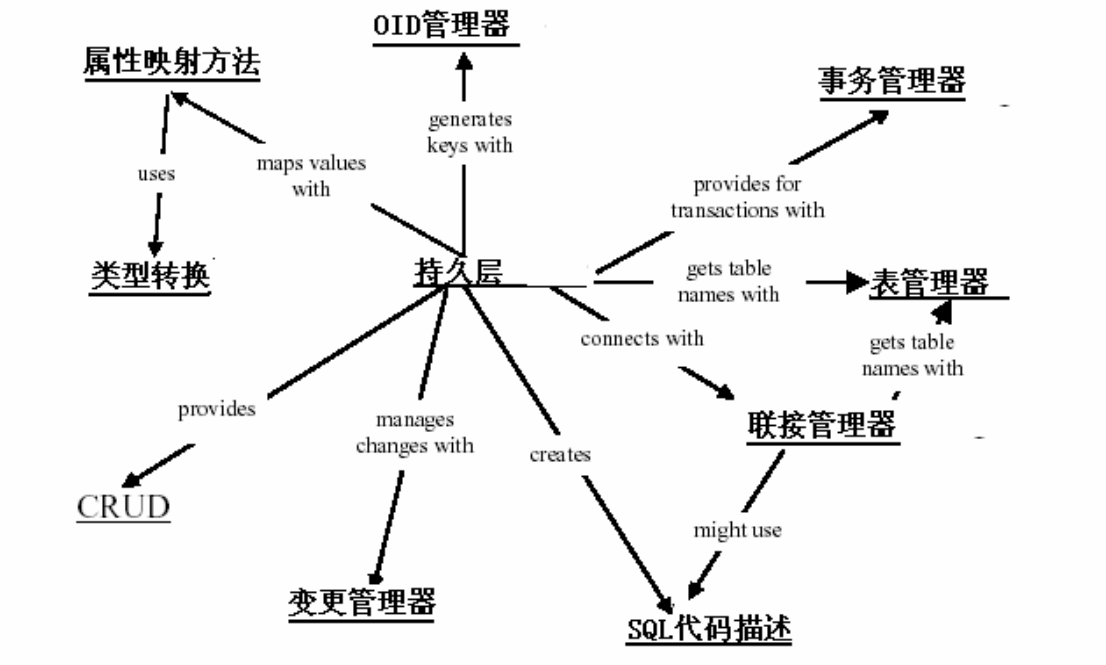


图 2 — 模式交互图

一个类场景图（见图 3）展示在我们例子中，当从数据库装载值时，对象是如何交互的。一个 Name 类可以请求 load，这个消息将被转发到它的超类 PersistentObject。它将从 ConnectionManager 得到一个联接，接着 PersistentObject 将生成 SQL，为完成这个，它需要向 Name 请求这个 SQL，并从 TableManager 得到一些表名映射，一旦生成 SQL，一个 Database Component 调用将返回结果集，接着，对每条从数据库返回的行，创建一个新的 Name 对象，并将这个行的每个字段分配到特定的对象属性中，在映射数据库值到对象属性过程中，数据库类型将被转换到对应的对象类型。一旦完成，PersistentObject 将返回一个已创建的 Name 对象集合。

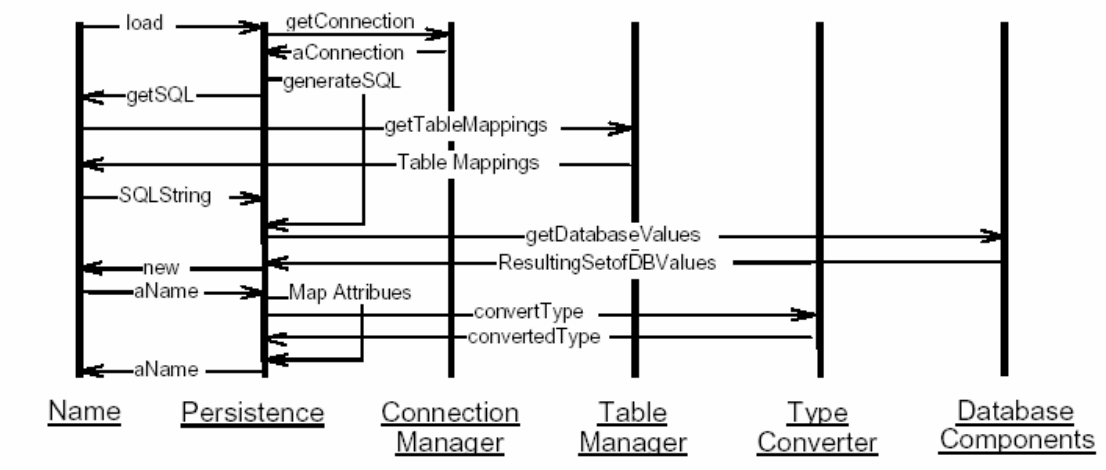


图 3 — 类交互图

参考文献

[Ambler 97]Scott W. Ambler. Mapping Object to Relational Databases. http:// www.AmbySoft.com/mappingObjects.pdf

[Beck 97]Kent Beck. SMALLTALK Best Practice Patterns, Prentice Hall PTR, Upper Saddle River, NJ, 1997.

[Brant &Yoder 96]John Brant and Joseph Yoder. "Reports," Collected papers from the PLoP '96 and EuroPLoP '96 Conference, Technical Report #wucs-97-07, Dept. of Computer Science,Washington University Department of Computer Science, February 1997. http://www.cs.wustl.edu/~schmidt/PLoP-96/yoder.ps.gz.

[BMRSS 96]Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Pattern-Oriented Software Architecture: A System of Patterns, John Wiley and Sons Ltd., Chichester, UK, 1996.

[Foote &Yoder 96]Brian Foote & Joseph Yoder. “Evolution, Architecture, and Metamorphosis,” Pattern Languages of Program Design 2, John M. Vlissides, James O. Coplien, and Norman L. Kerth, eds., Addison-Wesley, Reading, MA., 1996.

[Brown &Whitenack 96]Kyle Brown & Bruce Whitenack. “Crossing Chasms: A Pattern Language for Object-RDBMS Integration,” Pattern Languages of Program Design 2, John M. Vlissides,James O. Coplien, and Norman L. Kerth, eds., Addison-Wesley, Reading, MA., 1996.

[Foote &Yoder 98]Brian Foote & Joseph Yoder. “Metadata,” Submitted to PLoP ’98.
http://wwwcat.ncsa.uiuc.edu/~yoder/Research/metadata.

[Fowler 97-1]Martin Fowler. Analysis Patterns: Reusable Object Models, Addison Wesley, 1997.

[Fowler 97-2]Martin Fowler. Dealing with Roles, Proceedings of PLoP ’97, Monticello, IL, October 1997. http://www.aw.com/cp/roles2-1.html.

[GHJV 95]Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns:Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

[GemStone 96]Gemstone Systems, Inc. GemBuilder for VisualWorks, Version 5. July 1996.
http://www.gemstone.com/Products/gbs.htm.

[GemConn 96]Gemstone Systems, Inc. GemConnect for VisualWorks, Version 5. July 1996.
http://www.gemstone.com/Products/gbs.htm

[Keller 97-1]Wolfgang Keller: Mapping Objects to Tables: A Pattern Language, in “Proceedings of the 1997 European Pattern Languages of Programming Conference,” Irrsee, Germany, Siemens Technical Report 120/SW1/FB 1997.

[Keller 97-2]Wolfgang Keller, Jens Coldewey: Relational Database Access Layers: A Pattern Language, in “Collected Papers from the PLoP’96 and EuroPLoP’96 Conferences” Washington University, Department of Computer Science, Technical Report WUCS 97-07, February 1997.

[Keller 98-1]Wolfgang Keller, Jens Coldewey: Accessing Relational Databases: A Pattern Language, in Robert Martin, Dirk Riehle, Frank Buschmann (Eds.): Pattern Languages of Program Design 3. Addison-Wesley 1998.

[Keller 98-2]Wolfgang Keller. “Object/Relational Access Layers - A Roadmap, Missing Links and More Patterns,” Submitted to EPLoP ’98.

[OE 98]IBM, Corporation. Object Extender for VisualAge. 1998.
http://www.software.ibm.com/ad/smalltalk/about/persfact.html.

[Orfali &Harkey 98]Robert Orfali & Dan Harkey. Client/Server Programming with Java and {CORBA}, 2nd Edition, John Wiley & Sons, 1998.

[OS 95]ObjectShare, Inc. VisualWorks User’s Guide. 1998 http://www.objectshare.com/vw30abt.htm.

[RSBMZ 98]Dirk Riehle, Wolf Siberski, Dirk Baeumer, Daniel Megert, & Heinz Zuellighoven.Serializer, in Robert Martin, Dirk Riehle, Frank Buschmann (Eds.): Pattern Languages of Program Design 3. Addison-Wesley 1998.

[TopLink 97-1]The Object People Inc.: TOPLink Version 4.0 - A White Paper, 1997. http://www.objectpeople.com/.

[TopLink 97-2]The Object People Inc.: TOPLink Version 4.0 - User Manual, 1997.

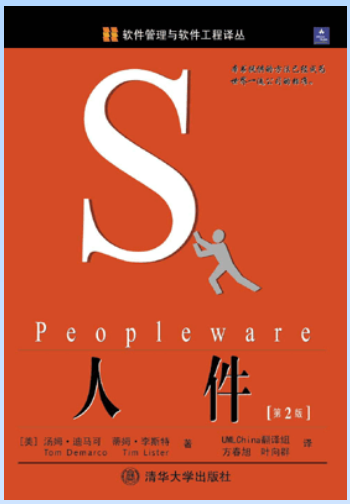
[VA 98]IBM, Corporation. VisualAge Smalltalk. 1998. http://www.software.ibm.com/ad/smalltalk.

[Yoder &Barcalow 97]Joseph Yoder & Jeffrey Barcalow. "Security," Fourth Conference on Patterns Languages of Programs (PLoP '97) Monticello, Illinois, September 1997. Technical report #wucs-97-34, Dept. of Computer Science, Washington University Department of Computer Science, September 1997.URL: http://www-cat.ncsa.uiuc.edu/~yoder/papers/patterns/#YoderBarcalow1997.

[Yoder 97]Joseph Yoder. A Framework to Build Financial Models.
http://www-cat.ncsa.uiuc.edu/~yoder/financial_framework.

[Yoder &Wilson 98]Joseph Yoder & Quince Wilson. A Framework for Persisting Objects to Relational Databases. URL: http://www-cat.ncsa.uiuc.edu/~yoder/Research/objectmappings.

[You+ 95]Joseph Yoder & Quince Wilson. Mainstream Objects, An Analyusis.
http://www-cat.ncsa.uiuc.edu/~yoder/Research/objectmappings.



Tom Demarco，Tim Lister

翻译：UMLChina 方春旭 叶向群

http://www.peoplewarecn.com

人件中文版网站

Frederick P. Brooks, Jr--《人月神话》第 19 章

近年来，软件工程领域的一个重大贡献是 DeMarco 和 Lister 在 1987 年出版的《人件》，我衷心地向我的读者推荐这本书。

Edward Yourdon

《人件》在 1987 年出版后，立即成为最畅销的作品。当人件第一版出版时，我写了一份评论，“我强烈推荐你买一份《人件》给你或你的老板，如果你是老板，那么为你部门的每一个人买一份，并给自己买一份”。这建议在 12 年后的第二版依然有效，并且更加热烈。

Mark A. Herschberg

我只学了办公环境的章节，就辞掉了原来的工作！

Joel Spolsky

我想，微软成功的原因之一就是公司里的所有经理都读过《人件》

Steve McConnell

由于本书第 1 版的赫赫声名，新版的《人件》是我不用看就会决定购买的少数几本书之一。

预订中文译本>>

案例研究：设计一个基于 Web 的服务配送系统

Min Song、Il-Yeol Song 著，林善茂 译

查看评论

摘要

在这篇论文中，我们介绍了一个使用 IBM 的 Net.Commerce 设计一个基于 Web 的服务配送系统的案例研究，以及我们通过这个项目学习到的经验。 采用 UML 符号来介绍设计规约，采用 IDEFIX 符号来介绍数据库方案。我们的设计规约包括架构，使用包图（package diagrams）描述系统构件，使用用例图描述系统功能，处理逻辑采用活动图（activity diagrams），数据库设计方面，我们介绍了详细的数据库设计，对设计和专门针对电子商务系统的用户个性化考虑进行了评论。我们的经验表明电子商务工具仍然缺乏某些功能，如处理订单的撤除，允许个性化的回复，发电子邮件给用户等，但总体上，采用这些工具，能帮助系统设计者有效地开发和维护这些系统。这篇论文的读者，将了解和学习到一种基于 Web 的服务配送系统的典型设计规约和各种技术设计问题。

关键词

电子商务（E-commerce），数据库设计，UML（统一建模语言），Net.Commerce，案例研究

介绍

在这篇论文中，我们介绍了一个基于 Web 的在线服务配送系统的设计规约。一般而言，一个电子商务系统的构建有以下两种方法。第一种方法是使用工具套件的个性化的方法，比如采用 IBM 的 Net.Commerce（Shurety, 1999），该套件最近被扩展并入 WebSphere 商务套件。比方说，商务套件提供工具来创建虚拟的商场的框架，包括目录，注册模板，购物车，订购和结账流程以及一个可个性化定制的数据库。第二种方法是一个单独公司的专家自底而上地开发一个内部系统。在这种情况中，开发者采用 mix-and-match 的工具手工构建一个虚拟的商场。另外，电子商务系统商业模式的数据库必须手工开发。虽然个性化方法加速了开发进程，但是受所用工具的功能的限制。另一方面，尽管自底而上方法与前一种方法相比，需要更长的时间，但是，以集成复杂性为代价，它为个性化提供了一个灵活的环境。关于开发基于 Web 的应用程序的工具和方法的更详细讨论，我们查阅了 Fraterali（1999）的工作。

不论开发者采用个性化的方法或是自底而上方法，了解电子商务系统的结构和处理逻辑，将帮助开发者有效地开发和维护系统。我们的论文基于我们采用个性化方法构建一个基于 Web 的在线服务配送系统的经验。

系统的开发采用 IBM 的 Net.Commerce 框架来实现我们的商务目标和需求。选择 IBM 的 Net.Commerce 系统，有几个重要的理由。第一，Net.Commerce 支持多店面。目前，我们只建立了一个店面，但今后我们的系统将发展到包含多个公司的多店面环境。第二，Net.Commerce 在于其他商务处理系统的联合方面比较灵活。尽管有这些良好的功能特征，我们还是不得不定制 Net.Commerce，以满足某些商务功能的要求，如电子邮件提示和回复处理。在这篇论文中，我们将介绍基于 Net.Commerce 的系统结构和个性化模块。我们将详细描述设计规约，处理逻辑以及基于 Web 的服务配送系统的数据库结构。我们使用 IDEFIX 符号（Song, Evans, and Park, 1995）来介绍相关联方案，使用 UML（Booch, Rumbaugh,, and Jacobson, 1999; Fowler, 1999）来介绍我们的设计规约中的其它部分。

电子商务系统的数据库结构的设计，与 OLTP 系统（Buchner Mulvenna, 1998; Ceri, Fraternali, and Paraboschi, 1999; Lohse and Spiller, 1998; Song and La-Van-Schultz, 1999; Song and Whang, 2000）相比，需要考虑一些不同的地方，比如：

处理多媒体和半结构化（semi-structured）的数据；

将书面的目录转化为标准的统一格式，并净化数据；

支持数据库级别的用户界面（比如，导航，商场布局，超链接）；

规格演进（Schema evolution）（比如，合并两个目录，产品类别，售完产品，新近产品）；

数据演进（比如，修改说明和描述，商品命名，定价）；

处理元数据；

为个性化和定制捕获数据，比如上下文之间的数据导航。

比如，在我们的系统中，我们使用许多可变长度的数据类型来处理半结构化的数据类型。我们的目录是一种格子结构。我们所设计的系统将容易适应今后的演进。在设计考虑的时候，我们尽可能周到地考虑结构的扩展，功能的扩展和系统的容量扩展。理解结构和电子商务处理逻辑能够帮助系统设计者们有效的开发和维护电子商务系统。

这篇论文的结构组织如下：第二部分从整体上介绍系统的架构和系统构件。第三部分介绍设计规约。在这一部分中使用用例图来介绍的需求，采用活动图介绍其处理逻辑，包括个性化定制，目录浏览以及订购商品等功能。第四部分描述了一个详细的数据库设计，并针对该电子商务系统的设计考虑进行了评论。第五部分对论文进行了总结。

架构和总览

架构

在这一部分中，我们简单描述了我们的电子商务系统的架构，该结构很多部分借用了普通的 Net.Commerce 系统。正如图 1 所描述的，我们的系统主要包含四个系统构件：

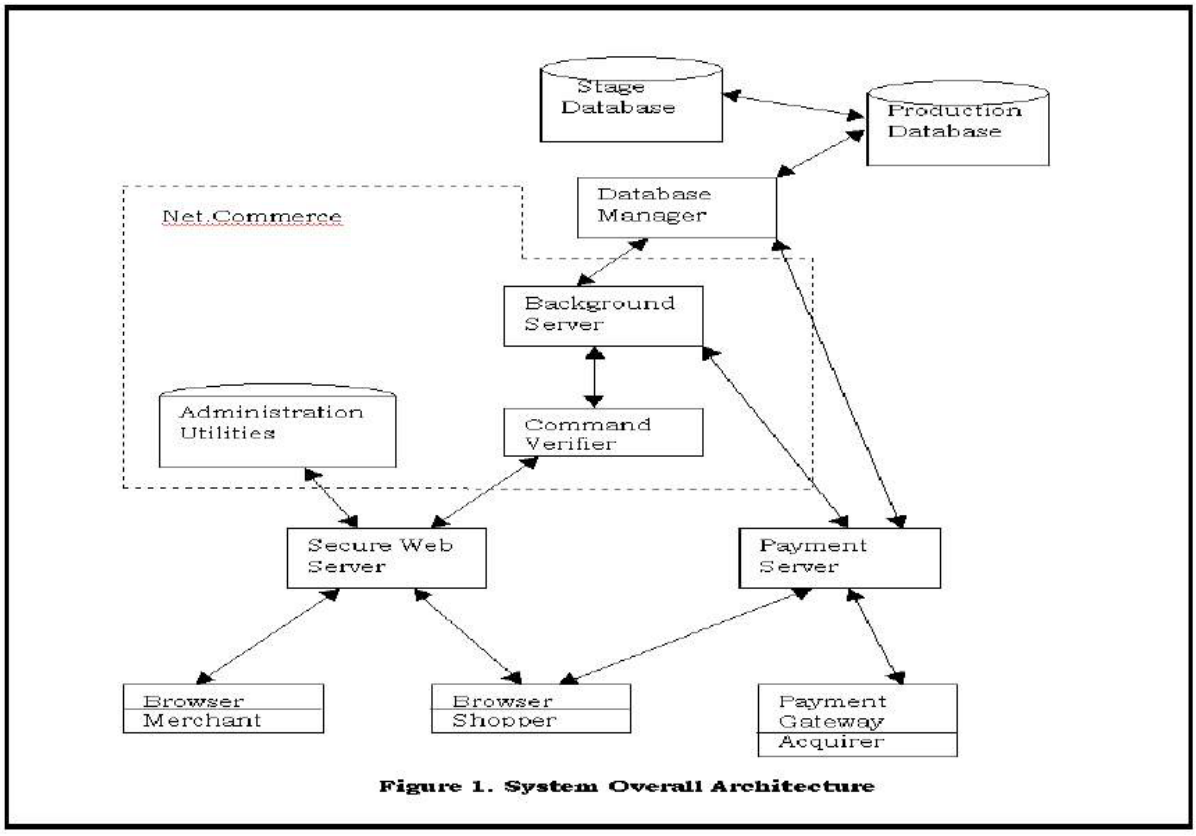


图 1 系统整体架构

安全 Web 服务器

Net.Commerce

数据库管理器

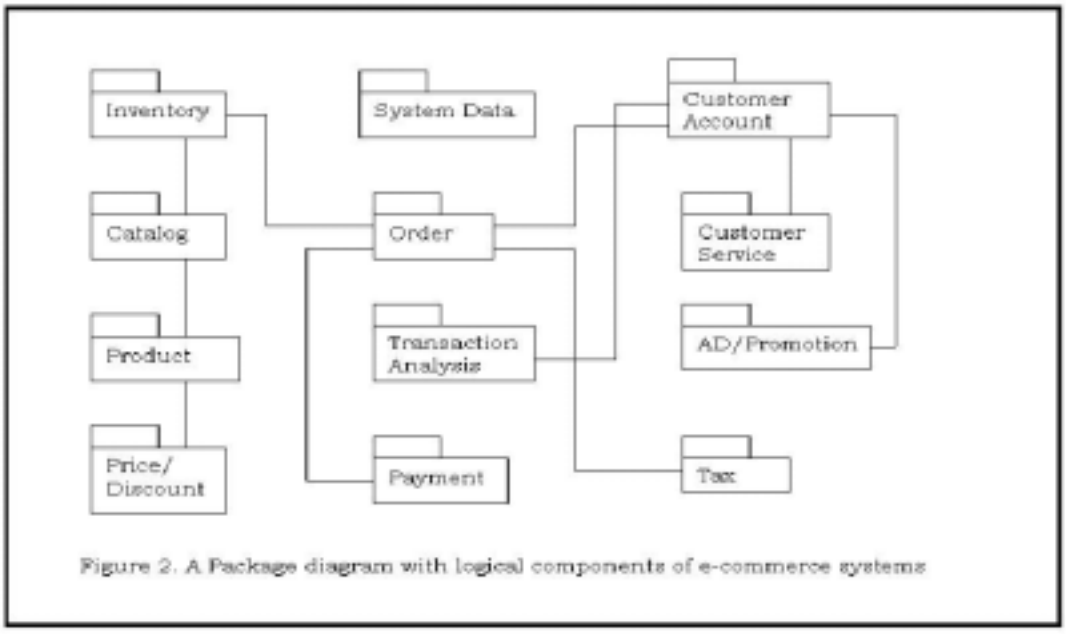


Figure 2. A Package diagram with logical components of e-commerce systems

图 2 电子商务系统逻辑构件包图

包 System Data 中控制着各种系统参数，如服务器的配置和访问控制参数。**包 Order** 保存着从购物车中传过来的当前订单信息，这个包保存着购物车中的商品和款项，客户下的订单，不同邮寄地址的子订单，付款方式，子订单与客户地址本入口的连接等信息。**包 Inventor、Catalog 和 Products** 联系非常密切。Products 保存着卖主信息和产品。Inventor 保存着可供销售的所有商品。Catalog 是个标准化的模板，集成了一个领域中各种卖主特定的商品。对于同一个商品，每个卖主都可以采用不同的专业术语和格式，而 Catalog 保护了真实的商业模式，避免了不同卖主带来的变更。Catalog 在 Products 和 Inventory 之间充当了缓冲器的作用。Catalog 同样也需要一个动态的目录管理器模板,以适应新的卖主和新的产品。这方面没有为今后考虑 :真正的程序实现是针对Catalog 和 Inventory。我们的经验表明，为某个特定领域的 Catalog 开发一个合适的规格是非常耗时间的，而且非常复杂。

包 Tax 处理关于税的类别和代码的信息，以及表示不同状态或税收管辖权限的代码信息。**包 Payment** 和 CyberCash、CyberRegister 相关联，它包含一个能确保在线安全交易的程序。

通过直接在一个数据库表格中记录交易信息的日志，**包 Transaction Analysis** 保存着客户请求和购物信息的踪迹。这个包包含了个性化的数据。

结算服务器

安全 Web 服务器这个构件负责接收用户浏览器的请求，并提交给相应的应用程序。由于客户的私人信息被获取并存储，这个部分对于客户交易的机密性来说，非常重要。为了确保只有授权的用户采用完成系统管理员的任务，Web 服务器在实现上采用了 SSL 协议。

Net.Commerce 构件在整个系统架构中充当了最为重要的作用。这个构件的主要功能是容许系统管理员通过安全的基于 Web 的管理工具来操作系统。另外，它还将验证由 Web 服务器引发的命令，并处理客户的请求。

数据库管理器构件管理着包含整个站点数据和全部商场商务数据的 Net.Commerce 数据库。我们使用 IBM 的 DB2 Universal Database V5 作为数据库管理系统。我们的 Net.Commerce 服务器和 Net.Data 使用 ODBC 来连接数据库。

结算服务器采用 CyberCash 来处理结算业务。我们使用 CyberCash 的方法来处理信用卡交易。首先，我们先要注册 CyberCash CashRegister 服务，然后下载并安装 CyberCash 商业连接工具包（MCK，Merchant Connection Kit）。CyberCash 的功能是获取客户的信用卡信息，并将其写到数据库中。一个后台的服务会定期唤醒来处理新的结算业务。它将连接到 Cyber MCK，发送结算信息，并最终更新数据库。

高层的逻辑构件

在这一部分，我们将从整体上描述电子商务系统的功能构件。从图 2 中显示的包图，可以看到我们系统的逻辑构件。

在 UML 中，一个包（Package）是一个结构，由相互关联的建模元素所组成。图 2 种的每个包，都包含着一个或多个表格，以及相应的处理逻辑。

包 Price/Discount 中包含哪些需要打折扣的价格信息以及哪些商品和款项将为哪些客户群打折扣的折扣率。**包 AD/Promotion** 包括一些与广告，促销和赠券有关的表格。这个包记录着哪些促销与哪些会话（session）相关联，哪些广告在那个会话中显示。**包 Customer Service** 包含着每个客户的反馈数据，还有订单，比如类别，种类，状态和回复等。

系统功能规约

在这一部分中，我们使用用例图来介绍功能需求，采用活动图介绍其处理逻辑。我们展示了客户和类别这两个类的层次关系，来说明我们的电子商务系统的一些独特特征。

客户层次关系

用例分析从识别系统的角色开始。角色是在系统中具有任务的用户组。在我们的系统中，主要的角色是顾客。我们系统的用户可以分为图 3 所示的几种类型。正如图 3 所示，顾客分为游客顾客和注册顾客。注册顾客又进一步细分为普通顾客组和定制顾客组。一个游客顾客可以浏览系统的货物类别，但不允许购买。对于这一限制条件可能有争议。我们做出这个决定，是因为我们的商品比较昂贵，对于偶尔逗留的非注册顾客来说。订购的概率非常小。

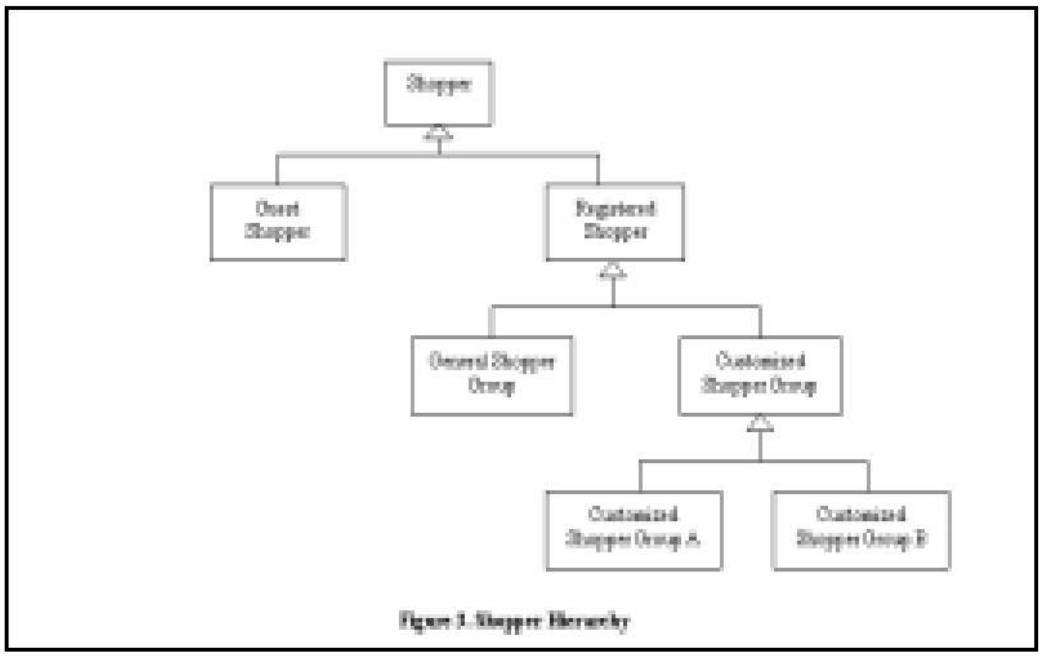


图 3 客户层次关系

出于个性化的目的，我们将注册用户分为普通顾客组和定制顾客组两种类别。普通顾客组没有个性化的功能，而定制顾客组则具有此功能。对于每个定制顾客组，都将分配一个顾客组的 ID 号，根据不同类别的模板和价格，提供不同的个性化的服务。顾客组 ID 号可以分配给一个用户或一个用户组，通过地理位置、商业类别或是由系统管理员设置的任意准则，可以划分一个用户组。例如，我们可以在系统中设置一条准则：来自日本的用户在某段时间范围内将享受某种折扣。

目录层次关系

如图 4 所示，我们的目录系统是一个格子结构。这个结构保存在数据库中，利用这些数据，可以自动生成 Web 页面，显示给顾客。Catalog Home Page 是根目录，Catalog A 和 Catalog B 是子目录。Product A 和 Product B 是我们网站上提供的日用品实例，可以属于多个父目录。许多商品包含着多个与其基类商品不同的款项。

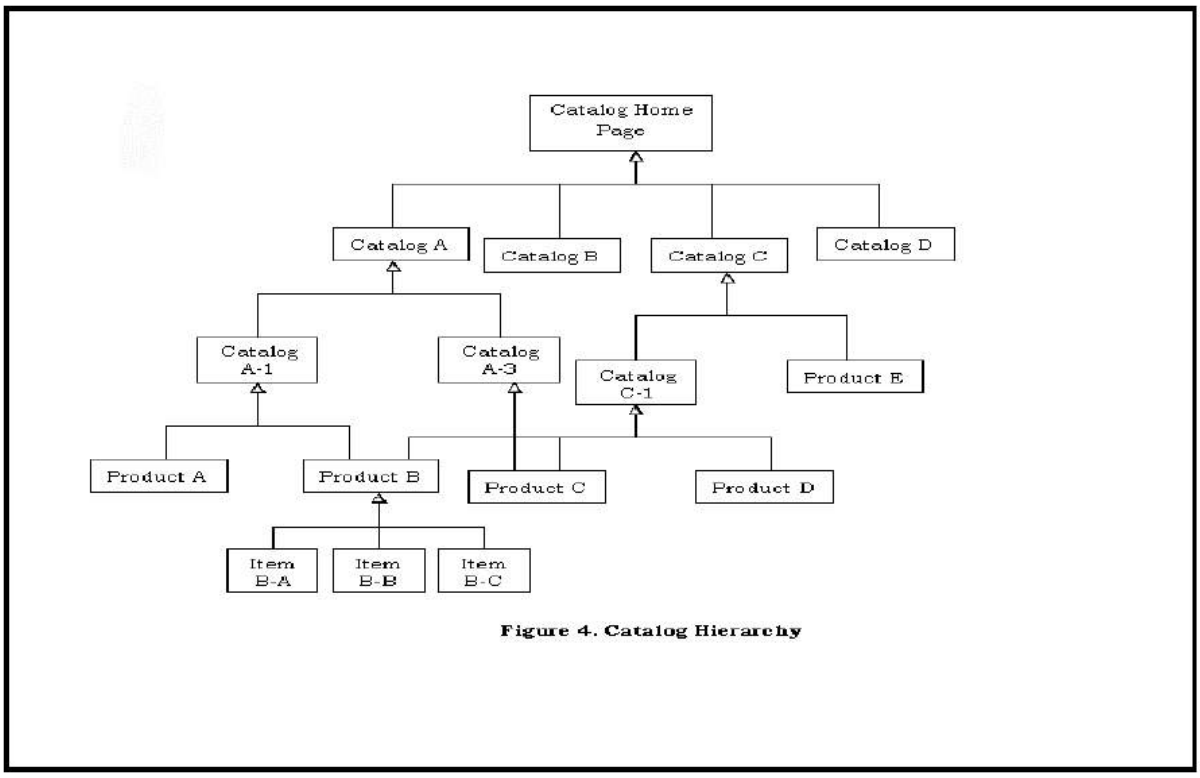


Figure 4. Catalog Hierarchy

图 4 目录层次关系

使用用例分析系统功能

用例分析广泛用于从各种角色的视野角度高层次地捕获系统功能的活动中。在系统的开发阶段，为了识别所有的角色活动和系统需求，在系统设计过程中，我们尝试捕获了一系列与角色相关的用例。图 5 的用例图显示了我们的电子商务系统中的用例。

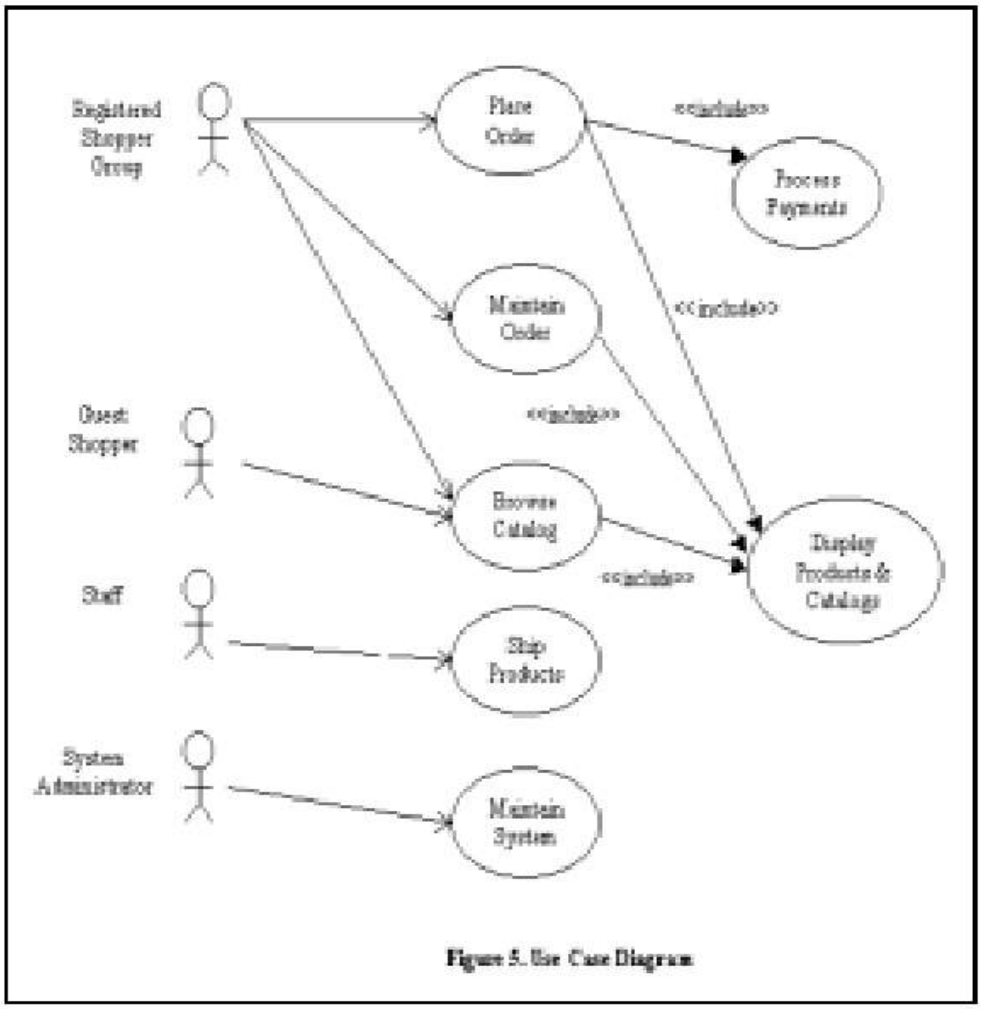


图 5 用例图

我们定义的三个角色为：1）注册顾客；2）游客顾客；3）系统管理员。另外，图 5 中是按照包含（include）的关系来描述用例间的关系。这将减轻描述多个用例中相同的行为的步骤，否则的话，将造成冗余。

当一个注册顾客或游客顾客选择了 Browse Catalog 的时候，Browse Catalog 用例开始了，对于游客顾客，将显示常规的类别，但是对于注册顾客组，将显示个性化的类别。由于 Play Order 用例是每个电子商务系统中必不可少的部分，我们将在下一节中详细介绍。Maintain Order 用例和订单更新-取消相关联。当一个用户完成订购操作，结束购买行为或是没有购买就离开系统时，系统就更新订单的状态和客户的账户信息。当一张订单产生后，某个职员的角色将完成“Ship Product”用例的功能。“Maintain System”用例则与系统管理员的任务有关。一个系统管理员将被卷入到这些活动中，如在产品服务器和补给服务器之间分段复制与传播数据，监视用户流量，检查数据输入，以及加强访问控制等。

使用活动图进行业务逻辑建模

在这一部分，我们采用活动图描述了业务逻辑。活动图捕获用例或系统功能的工作流。尽管活动图不能使对象和行为之间的联系变得非常清楚，但对于我们系统用例的功能建模来说，它是适用的。

浏览目录

图 6 描述了浏览目录用例的处理逻辑。目录浏览这个活动的第一个分支取决于用户是否注册。如果用户是游客顾客，将只提供最少的优惠条件，而且用户不允许下订单。如果用户以前曾注册过，则活动根据顾客是否有合法的顾客组 ID 进一步产生分支。如果顾客有 ID 号，则个性化模块产生，用户将可以看到个性化的目录和价格。如果顾客目前不属于任何一个顾客组，则显示给顾客的是普通模块。不论顾客属于哪个类别，在特定的时间内，所有顾客组看到的同一种商品的价格均相同。

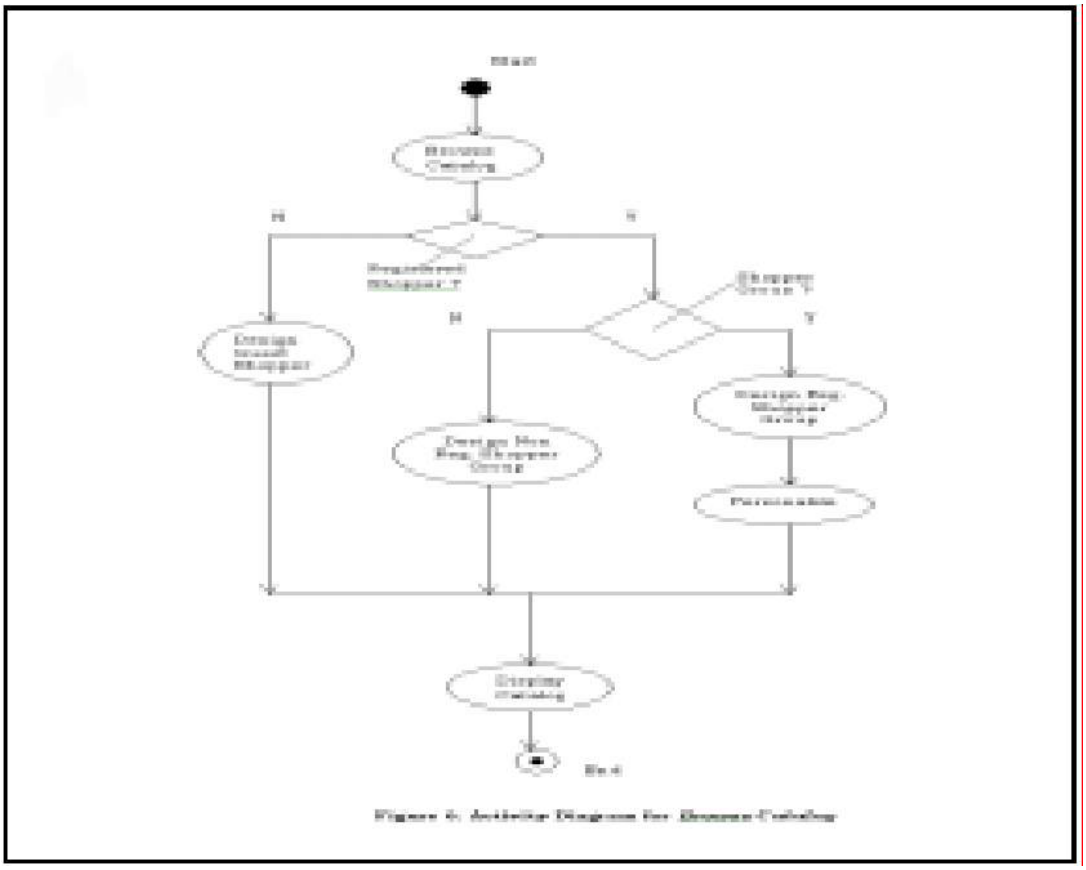


图 6 Browse Catalog 用例业务逻辑

个性化

图 7 描述了与个性化公共用例相关的活动。在创建一个用户组的过程中，活动“根据交易历史分配顾客组 ID”和活动“发送商品到目标用户”扮演了重要的作用。目前，我们区别顾客组的准则是根据其以前记录。当用户取消订单时，我们将这些信息保存在一个非独立的系统中。由于这时我们不能将当前订购系统集成到电子商务系统中，我们只能手动执行这个处理。活动“输入顾客组 ID”根据用户是否具有合法的顾客组 ID 产生分支。如果用户没有，系统显示普通目录。如果用户有一个合法的 ID，则活动根据用户是否成功输入顾客组 ID 进一步产生分支。假如用户忘记了它的顾客组 ID，它可以向系统提供注册时填写的 E-mail 和用户 ID。如果 E-mail 和用户 ID 不符合，用户必须通过电话或其他通讯工具来证实他们的身份。如果 E-mail 和用户 ID 相符，该用户就会收到一个含有分配给他的顾客组 ID 的 E-mail。

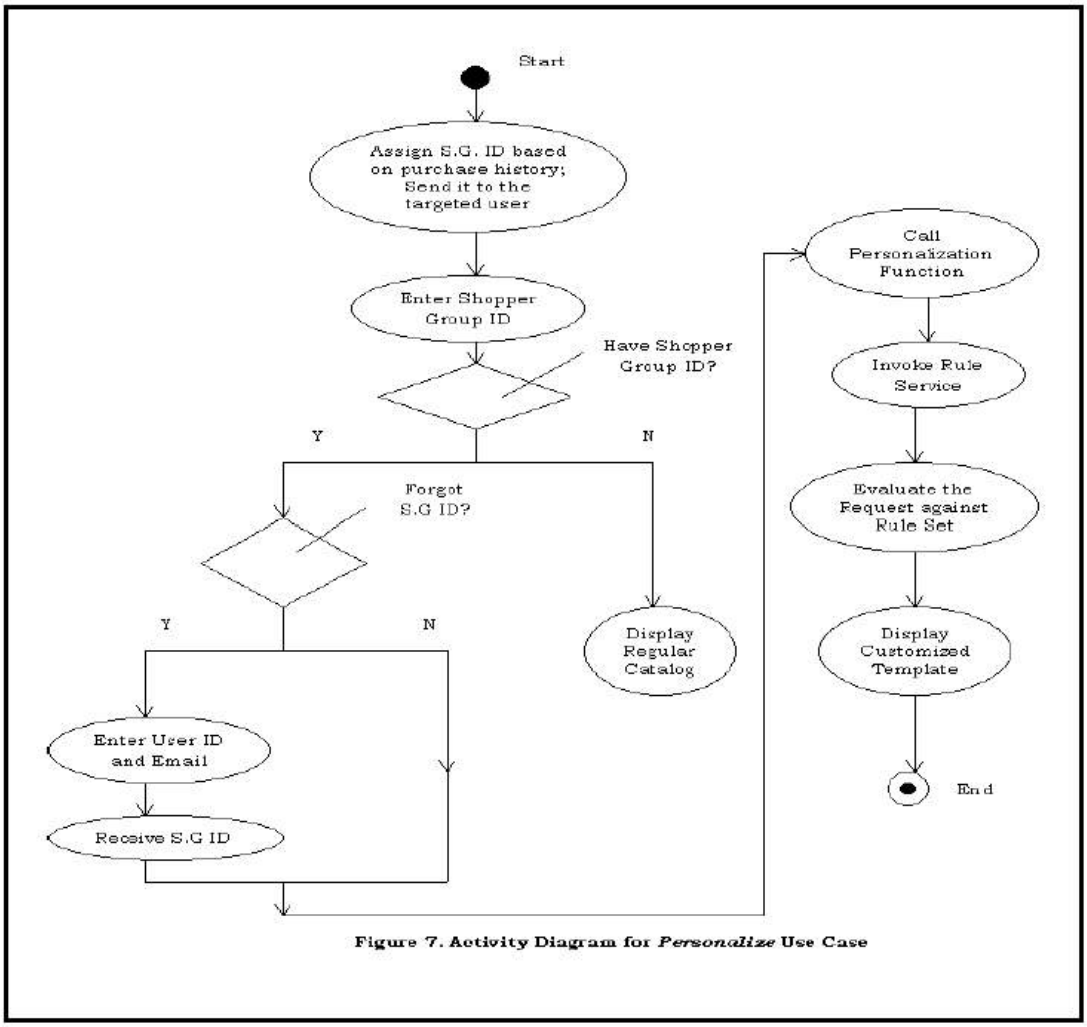


Figure 7. Activity Diagram for *Personalize* Use Case

图 7 个性化用例的活动图

如果用户输入了正确的顾客组 ID，系统将根据个性化模板中预设好的规则产生个性化目录。这个规则用于决定哪些商品和目录在什么情况下分配给哪些顾客组。通常情况下，每个顾客组 ID 至少分配了一个商品或目录。一个顾客组 ID，可以对应多个商品或目录，但是一个会话，只能对应一个 ID。

个性化模板基于一个规则集。利用这些根据我们公司业务政策所产生的规则，个性化模板可以分析一个特定的顾客或者他们的购买行为，并产生一系列商品，推荐给顾客。规则的格式很简单：<if 条件，then 活动>。比如，下面这句话就是个性化规则的例子。

如果顾客对 A 领域的物品感兴趣，则考虑推荐商品 A1。

规则处理器计算条件是否成立。如果条件成立，就完成一个特定的活动。由于维护规则和捆绑规则的处理结果非常复杂，我们采用了 WebSphere 商业套件。该套件提供了面向对象的各种非常灵活的方法。

由于我们的模块不需要数据挖掘、联合过滤等复杂的技术，个性化模块背后的业务逻辑是很直接的。正如 Aggarwal 和 Yu（2000）所指出的：采用明确的方法，如用户反馈或评价，或是采用不明确的方法，如观察用户的购买行为，个性化将会变得有规则。在我们系统的下一个版本中，我们将运用数据挖掘技术，来改进个性化模块。这样，用户以前的购买行为的统计信息将会在个性化中体现出来。

下订单

图 8 显示了“下订单”用例所发生的活动。这个用例介绍了整个电子商务系统中最为关键的部分。“选择订单详细信息”活动根据送货地址与注册时填写的地址是否相同产生分支。如果不相同，用户需要填写送货信息；否则，系统默认根据用户注册所使用的地址来送货。

当送货地址确定后，系统开始计算折扣和税额，并显示订单的分类统计信息。尽管前面在商品价格上有了折扣，我们仍然需要一个折扣的字段。这样，对于某些商品，我们可以在计算完税额后再给顾客一个折扣。

如果用户希望用信用卡支付，系统会提示用户输入信用卡信息。另外，如果用户希望收到发票的话，它可以在“寄送发票”选项上打勾。如果信用卡信息不正确，系统将提示错误信息，这样用户可以采取相应的操作。一旦“提交”活动结束，系统显示订单确认页面。同时，一个包含订单确认内容的 E-mail 将发送到用户注册指定的邮箱中。“更新帐户”的活动与订单状态以及订单历史纪录相关联。如果用户想取消订单，它需要访问我们系统中一个叫做退款的子模块，输入用户 ID，密码以及下订单时的信用卡信息。当取消流程结束时，系统将更新用户的账户信息，相应的一封包含取消确认内容的 E-mail 将发送给用户。

数据库结构

在这一部分中，我们介绍了电子商务事务处理（ECTP，E-commerce Transaction Processing）系统的数据库方案。在图 2 的包图中，我们从整体上介绍了系统构件，图 9 介绍了 ECTP 方案，该方案是在 Net.Commerce（Doemer et al, 1999; Shurety, 1999）的基础上稍稍修改而成的，包含了目录、商品、订单、支付和价格/折扣包等部分。虽然在图 9 中我们想尽量反映现实生活中的电子商务系统，但是在系统实现的时候，实际的方案是复杂多变的。另外，根据底层的商业模式和需要，对于每个实例来说，详细的方案都会不一样。注意，在图 9 采用的是 IDEFIX 标识（Song, Evans, and Park, 1995）。

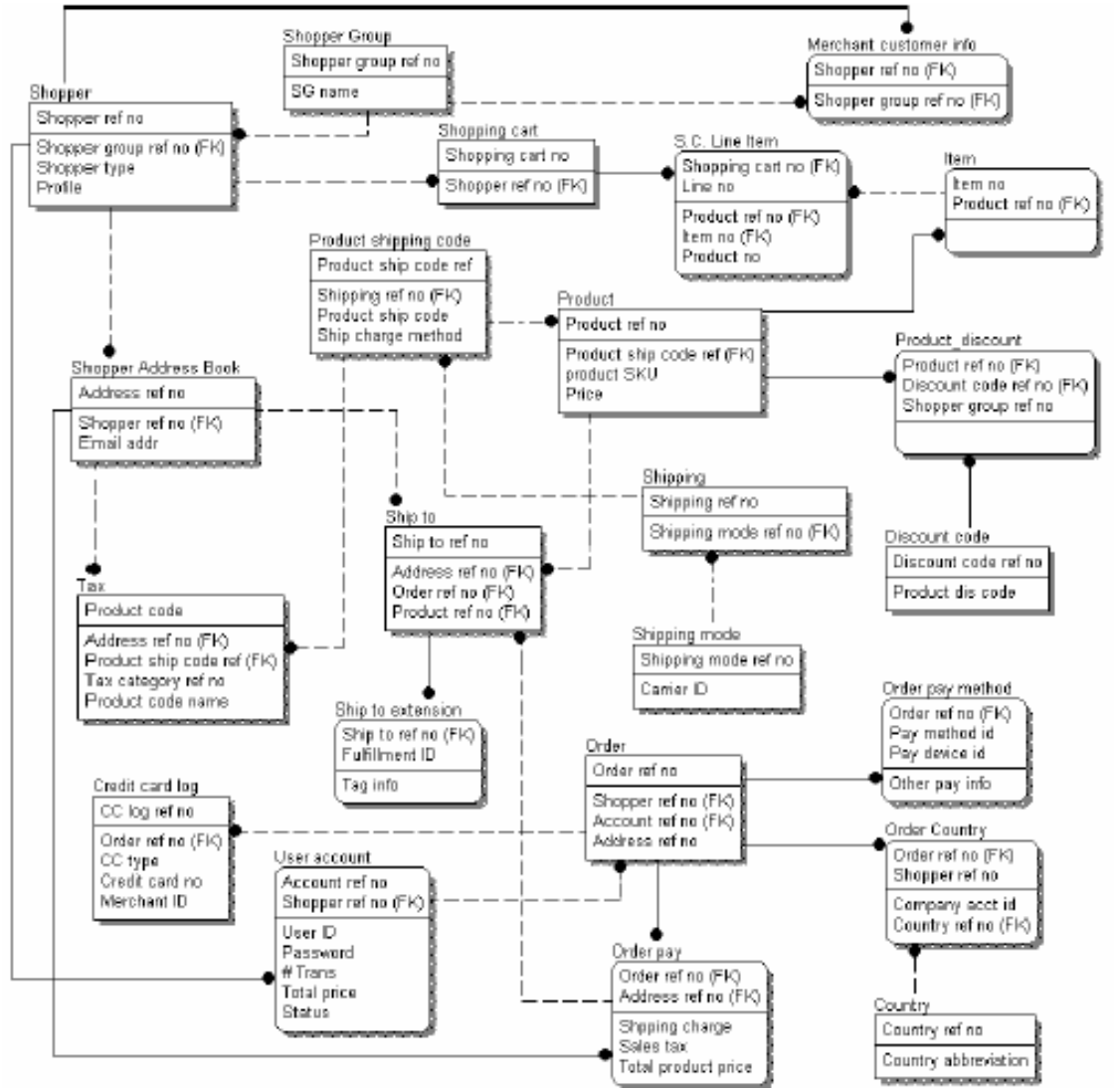


图 8 电子商务事务处理系统的数据库方案

对于多数的表，我们使用系统生成（代理）的主键（PK，Primary Key）来避免数据改变的依赖性。

智能键，有着内在含义，依赖数据而创建。因此，改变数据会带来主键的改变。图 9 中所有表都采用代理键。主键在第一个格，而非主键在第二个格。FK 表示外键(Foreign Key)。

图 9 介绍了 IDEFIX 中最重要的两种关系类型：(1) 一对多确定的关系，用实线表示；(2) 一对多不确定的关系，用破折线来表示，比如：SHOPPER 和 USER ACCOUNT 之间是一对多确定关系，因为表 SHOPPER 的主键 SHOPPER REFERENCE NO#是表 USER ACCOUNT 主键的一部分。SHIPTO 和 PRODUCT 之间是一对多不确定关系，因为表 PRODUCT 的主键 PRODUCT REFERENCE NO#是表 SHIPTO 的外键，但不是其主键的一部分。

由于涉及到参照完整性，下面三种类型的 delete 动作需要考虑：1—) Delete Cascade。2) Delete No Action。3) Delete Set Null。Net.Commerce 系统实现了 delete Cascade，保留了数据库的参照完整性。当数据从一个表中删除时，delete cascade 使得相关表中相同数据也被删除。比如说，对 Order 和 OrderPay 表进行 Delete Cascade 操作，则所有表由于 Delete Cascade 都受影响，也被删除。

第二种类型的删除操作 Delete no action，用于主键被删除但外键的值需要保留的情况。比如，Product 和 ShipTo 就采用这一类型的删除操作，当表 Product 中的 Product Refer NO 被删除时，表 ShipTo 的外键保持不变。这就确保了当表 Product 中的 Product Refer NO 被删除时，相关订单发货信息不会丢失。User Account 和 Order 也采用 Delete No Action，由于只能是注册顾客才能下订单，而且，不能从数据库中删除注册顾客，因此，删除 Shopper Reference Number 时不允许的，然而，当一位顾客搬家时，它可以提供一个新的联系地址信息，一个新的记录被加到表 Shopper Address Book 中。旧的地址信息并没有丢掉，只是加了个标志，作为一个临时地址存放在 StatusFlag 属性中。临时地址将在 60 天后由 Net.Commerce 提供的工具自动删除。

第三类型的删除操作，Set Value to null，用于当外键没有任何意义，但我们不想删除包含外键的行的情况。比如，表 Ship 中的 Tax Product Code 就采用这种删除操作。在这些实例中，如果 Tax Product Code 被删除了，Taxware Sale/Use Tax 系统会将相应产品代码的值设为空。

还有少量用于定制功能的表，如 Credit Card log 表， Ship To Extension 表以及 country 表。在支付的时候，考虑到今后退货的情况，我们需要用到表 Credit Card log 表。在运输特别贵重的物品时，需要贴有公司标签的文件，用于报告完成的服务部分，这种情况下，我们就需要用到 Ship To Extension 表。表 Country 用于保存一个公司的国家名称缩写，该缩写从公司的账户系统的全局换算表转化而来。表 Order Country 用来匹配下订单的用户所对应的国家中，是否设有代理商。

学到的经验

在这一部分，我们总结了采用 Net.Commerce 开发我们的系统时所学到的经验。

在项目计划中考虑定制功能

在我们电子商务套件的最初设计阶段，采用一个由 Net.Commerce 软件提供的“立即可用”的模型。当我们直接用“立即可用”模型来迎合内部用户的需求时，我们遇到了困难，这也促使我们必须定制 Net.Commerce。由于 Net.Commerce 支持在任何领域中使用，因此在理解和定制工具上非常复杂。我们不得不简化某些模块，比如说单店面和订单流程。由于过于复杂，定制工作所需的时间比我们计划的多很多。其中的一个障碍是在早期设计阶段开发组和内部客户意见不一致。比如，我们打算开发一个系统功能：根据某些国家将顾客分块，由这些国家的代理商来联系销售我们的商品。最后决定是不对顾客分块，当他们需要通过我们的系统订购商品的时候，系统跳转到代理商的网站。其他定制的成果包括 E-mail 提示，和根据公司办事流程处理回复。

系统架构方面的信息不足

在软件开发包中，Net.Commerce 系统架构介绍得不是很清楚。在初始的设计阶段，由于这方面的不足，我们在理解系统上遇到了困难。而且，由于缺乏对软件架构的详细理解，我们很难对 Net.Commerce 进行定制，以满足我们商务的需要。另外，Net.Commerce 没有提供关于定制的足够的文档，迫使我们只能求助昂贵的顾问。

开发组之间缺乏交流

我们意识到开发组成员之间的有效交流通道必须改进。几个开发人员同时开发基于 Net.Commerce 的不同部分的 C++类。结果，在写的程序中，我们遇到一些前后矛盾和冗余的情况。另外，在内部客户，项目经理和开发人员之间的交流效率也比较低。

整合已有的系统

在电子商务系统与已有的商务系统，如账户和税额系统整合的时候，我们遇到了困难。由于 Net.Commerce 缺乏自动确认和报告的功能，我们不得不建立一个界面模块来自动处理我们电子商务系统与已有系统之间的事务。

教育相关的所有人员

我们还发现一些成员拒绝接受由于新的电子商务系统所引起的变化，结果造成项目的延迟。加强所有相关人员的教育，使他们从一开始就重视项目的开发。

目录设计，结构与维护

假如商品的类型是变化多样的，对我们来说，建立一个适当的目录结构是非常重要的。尽管我们花了很多时间设计目录和构建目录系统，但是由于缺乏对商品完全的理解，我们不得不经常更新目录结构。这个问题的原因在于系统开发组和内部系统用户之间存在认识的差异。这点再次证实了彻底的需求建模的重要性，它能够使升级和维护费用降到最小。

培训店面管理员

在我们的实例中，商场组负责通过基于 Web 的管理员工具，来管理店面。由于缺乏技术知识，他们害怕使用管理员工具，趋向于依赖开发人员来完成他们的任务。尽管 Net.Commerce 提供了一个叫做 NCADMIN 的基于 Web 的管理工具，但是它对操作人员要求较高，对于一个新手，会导致许多不必要的错误。

处理参照完整性的限制

我们认识到，并不是所有的参照完整性的限制都会自动加强。依赖于关系的语义，不同类型的参照行为，如 DELETE CASACDE，DELETE ACTION，或者是 DELETE SET NULL 必须小心选择。

这个项目总的经验是采用迭代的方法来构建系统。在项目规划阶段，软件的范例和部署应该被集成到方案中。不要过多强调所涉及到的技术的数量，应该强调的是这些技术所带来的影响力以及对业务所带来的价值。项目组成员从项目一开始就应该对创新和规模有深刻的理解。

Net.Commerce 性能全面，功能强大，但是需要有经验的程序员来实现它的功能。但是，它的开发和维护工具，还是非常简陋，它需要更多的图形化界面和向导，在现有的模块中集成进更多易于掌握的组合。

总结

在这篇论文中，我们介绍了一个使用 IBM 的 Net.Commerce 设计一个基于 Web 的服务配送系统的案例研究，我们介绍了技术设计规约以及我们通过这个项目学习到的经验。我们采用包图介绍了架构和系统构件，采用用例图描述系统功能，处理逻辑采用活动图，还介绍了数据库设计。借助于 Net.Commerce 的强大功能和顾问的一些帮助，我们成功开发出了商业需要的系统。

我们还详细介绍了电子商务事务处理系统的数据库方案。大多数现实生活中的电子商务服务配送系统的数据库框架与我们在这篇论文中介绍的相似。了解电子商务系统的结构和处理逻辑，将帮助开发者有效地开发和维护系统。我们的经验表明电子商务工具仍然缺乏某些功能，如处理订单的撤除，允许个性化的回复，发电子邮件给用户等，但总体上，采用这些工具，能帮助系统设计者有效地开发和维护这些系统。

从这个项目中学习到的经验表明：开发一个基于 Web 的系统同样需要典型的系统开发方法。遇到的困难包括对系统架构的彻底理解，与现有系统的整合，彻底的需求建模，相关人员的培训，用户培训，以及对于实体完整性约束的细致处理。

参考文献

Booch G., Rumbaugh, J. and Jacobson, I. (1999) UML User's Guide,Addison Wesley.

Buchner, A and Mulvenna, M. (1998) Discovering Internet Market Intelligence through Online Analytical Web Usage Mining. SIGMOD Record, 27(4): 54-61.

Charu C. Aggarwal and Yu, Philip S. (2000) Data Mining Techniques for Personalization, IEEE Data Engineering, 23(1): 4-9.

Ceri S., Fraternali P. & Paraboschi, S. (1999) Design Principles for Data-Intensive Web Sites. SIGMOD Record, 28(1): 84-89.

Doemer R., Ezers P., Gustavsson P., May C., and Shull G. (1999) Building e-commerce Solutions with Net.Commerce: A Project Guidebook, IBM.

Fraternali P. (1999) Tools and Approaches for Developing Data-Intensive Web Applications: A Survey. ACM Computing Surveys,33(3): 227-263.

Fowler, M. (1999) UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley: Harlow, England.

Lohse, G. L. and Spiller P. (1998) Electronic Shopping. Communications of the ACM, 41(7): 81-88.

Shurety S. (1999) E-Business with Net.Commerce. Prentice Hall.

Song I., Evans M, and Park E. (1995) A Comparative Analysis of Entity-Relationship Diagrams. Journal of Computer and Software Engineering, 3(4): 427-459.

Song I and LeVan-Schultz K. (1999) Data Warehouse Design for Ecommerce Environments. Lecture Notes in Computer Science, 1727: 374-388.

Song I and Whang K (2000) Design of Real-World E-commerce Systems. IEEE Data Engineering Bulletin, 23(1): 23-28.

Treese, G.W. and Stewart, L.C. (1998) Designing Systems for Internet Commerce, Addison Wesley.

人件 - Microsoft Internet Explorer

文件(F) 编辑(E) 查看(V) 收藏(A) 工具(T) 帮助(H)

后退 媒体

地址(D) http://www.peoplewarecn.com/ 转到 链接

《人件》— 老板，别把开发人员当成牲口

» 新闻

《人件》提供预订>>

互动出版网 WWW.CHINA-PUB.COM

Cnforyou.com

华储 www.huachu.com.cn

» 相关文章

关注程序员自己的文化——专访Tom DeMarco

别把开发人员当成牲口

Brooks在《人月神话》中的评论

Alan Cooper的评论

办公室空间，下一场革命

《人件》在计算机行业的实践（待续）

人的问题：关于《人件》

Edward Yourdon的评论

开发人员是人吗?

» 各版本封面

英文，日文，德文

首页 购买 留言板

完毕 Internet

业务规则说明

Windy J 整理

查看评论

1 概述

本文描述了业务规则、政策性指导和业务规则陈述三者之间的关系，并详细介绍了术语，结构性规则，行为规则和衍生规则，对每个类别都提供了相应的样例进行说明。

2 业务规则背景

业务规则最初表现为政策性的陈述（方针，策略，原则，即 Policy），作为企业业务指导的概括性描述。通常没有正式的形式来表达这些陈述。

例如：*银行要为客户提供一个安全的资金存取环境。*

接着，企业的员工需要将其转化为更有实际意义的“做什么”的陈述。这就是业务规则陈述（BusinessRule Statement），然而，这些陈述通常也是类似散文的风格，也就是说这些句子有时候清楚，有时候含糊（或许是故意的），大多数情况下包含了多个概念。业务规则陈述通常作为分析员导出更正式的业务规则的起点。

例如：

当下列情况出现时，客户不能取款：

客户账户余额不足；

客户密码错误；

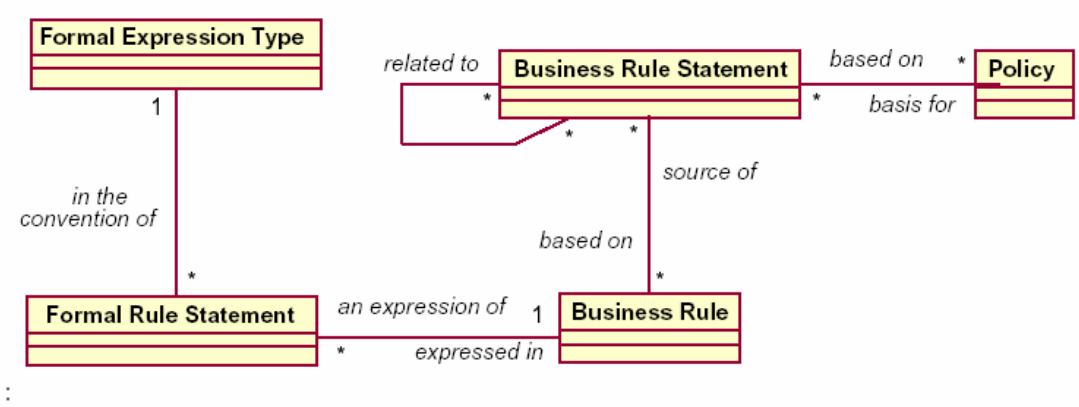


Figure F-1: The Origin of Business Rules
Corresponds to Figure 3 [p. 10]

业务规则（Business Rule），需要进行清晰定义，它有原子性，也就是说它既不能包含多个概念（那是业务规则陈述），也不可被拆分成更细的业务规则（否则信息丢失）。

上面的业务规则陈述将分解成两条业务规则：

如果客户密码错误，客户不能取款。

如果账户余额不足，客户不能取款。

业务规则基于相应的业务规则说明。

可以对业务规则进行正式描述（Formal Rule Statement），一条业务规则可以有多种正式描述，每种正式描述都有一种存在公开标准的描述类型（Formal Expression Type），例如：structured English（结构化英语）, IDEF1X, Oracle's CASE*Method, Object Role Modeling,Ross's notation 等。

用结构化英语描述的业务规则例子如下：

If Accout. AmountAvailable<=0 then

Accout. CanDraw =false

End if

3 业务规则种类

可以把所有的业务规则归为以下四个种类：

- 业务术语（Business Term）的定义；
  - 是表达业务规则的基本元素。它描述了人们怎样思考和谈论事物。
- 事实（Facts）：
  - 建立在术语之上，表达了各术语代表的事物间的关联；
- 约束（Constraints，这里叫 action assertions，行为声明）；
  - 每个企业都在某些方面约束行为。
- 衍生规则（Derivations）；
  - 定义某种形式的业务规则如何转变为其他业务规则，通常表现为不同的形式。

4 业务术语

该部分内容已列入独立的术语表。术语（Term）包括通用术语和业务术语，通用术语是含义自明的，例如*人民币，身份证*等，无须额外说明，而业务术语指该词汇在特定的业务环境（上下文，context）中有特定的含义，所以需要进行定义，并在定义时说明它适用的业务环境，例如，“*个人支票，是指在银行开设活期账户的居民用个人信誉作保证，以支票为支付凭证的一种结算工具。*”它的业务环境是银行业务。

另外，一个术语可能拥有多个同义词，例如，*存款*，*存钱*，在当前业务范围中是同义词；一个术语也可能是多个术语的同义词。

术语可以是类型，例如*银行*，也可以是具体实例，例如*工商银行，招商银行*。

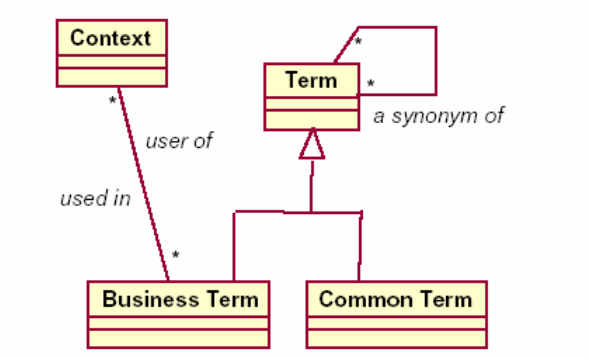


Figure F-3: Terms

5 业务规则详述

从上面可以看到，除了业务术语的定义，业务规则包括结构，行为，衍生三个种类，下面分别详细介绍每一类。

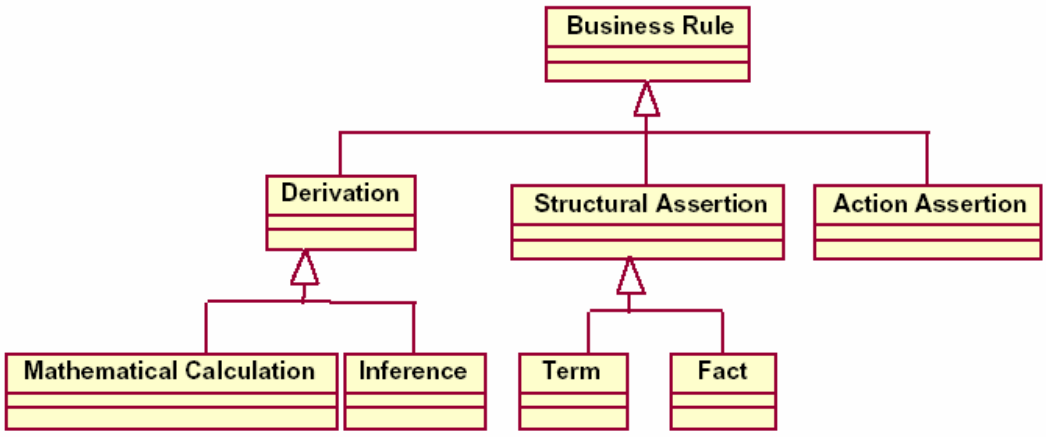


Figure F-2: Business Rule Types

结构规则

结构规则关注业务中的概念和概念之间的关系，是业务的静态部分，经常表现在实体关系图中。

具体而言就是，通过事实（Fact）将术语（Terms）关联起来，形成对业务组成的结构描述。所以，一个事实（Fact）包括两个或两个以上的对象角色（Object Role），并表达了它们之间的关系，在这个 fact 中，每个语义上的对象角色（Object Role）都由术语来担任，例如事实“*每个客户拥有一个或多个活期帐户*”，在这里，*客户*和*活期帐户*是业务术语，*客户*担任*拥有者（Owner）*的角色，*活期帐户*担任*被拥有者*的角色。

同一个术语可以出现在多个事实中，但业务术语至少应该出现在一个事实中，如果没有事实跟某个业务术语相关，说明这个业务术语不属于现有的业务范围。

对同一个事实，至少存在一种定序文本（Text Ordering）的表达方式，也可能存在多种表达方式，但一个定序文本只表示一个事实。一种定序文本一定由一个或多个短语（Phrase）组成，这些短语用来描述每个对象角色（Object Role），在短语中按顺序使用了这些对象角色。

例如：

本银行为每个客户提供现金存取服务。

客户可以通过本银行存取现金。

两个定序文本表达了同一事实的不同视角，而相同的术语在这两种表达方式中都担任了某种对象角色；“*每个客户*”是一个短语，用来描述术语“*客户*”；“*银行*”是术语，“*本银行*”是短语，“*现金存取服务*”是短语，本身也是一个术语。

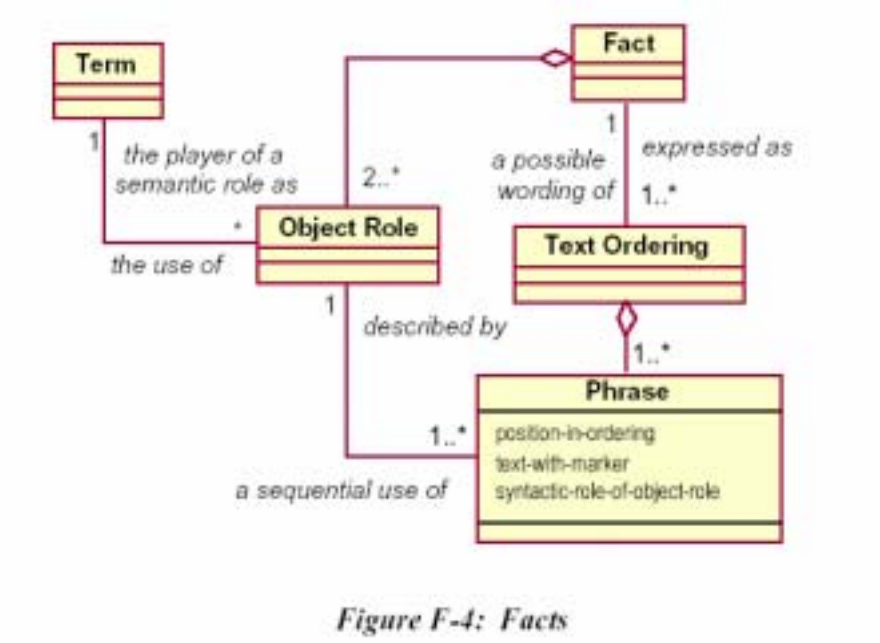


Figure F-4: Facts

事实（Fact）所表达的术语关系又可以分为以下三类：

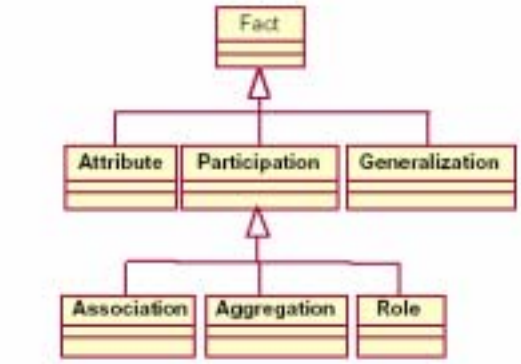


Figure F-6: Kinds of Fact

属性（Attribute），指某个术语作为另一个术语的属性，例如，*客户资料包括姓名，联系电话，联系地址……*（略）；

泛化（Generalization）：指某个或某些术语作为另外一个术语的泛化，例如，*ATM 取款是取款的一种*；

参与，协作（Participation）：由上图可见，参与（协作）关系是术语之间除了属性和泛化之外的关系它又可以分为关联（Association），聚合（Aggregation）和角色（Role）；例如前面的证券公司和客户之间是*关联*的关系，客户和保证金帐户是*聚合*的关系，*在存钱时，客户充当贷方的角色*，此时客户和买方两个术语就是*角色*关系，指一个术语在它与外界的某种交互中担任某种角色，注意，它在另外的交互中可以担任另外的角色，例如，*借款，客户充当借方的角色*。

行为规则

行为规则关注业务的动态行为（活动），尤其对活动能否进行，以及活动可能产生的结果进行约束，所以，在行为规则中，会出现一些“必须（must）”，“不能（must not）”，“需要，最好（should）”，“最好不要（should not）”这样的限定词。

由下图可见，行为规则（Action Assertions）可以跟业务规则（Business Rule）相关联，此时业务规则多是结构规则，也可以是行为规则和衍生规则；但它一定要跟一个或多个业务构成元素（Construct）相关，业务构成元素包括业务规则和业务活动（Action），业务活动指具体的业务行为，例如一次电话查询，一次存款服务，等。

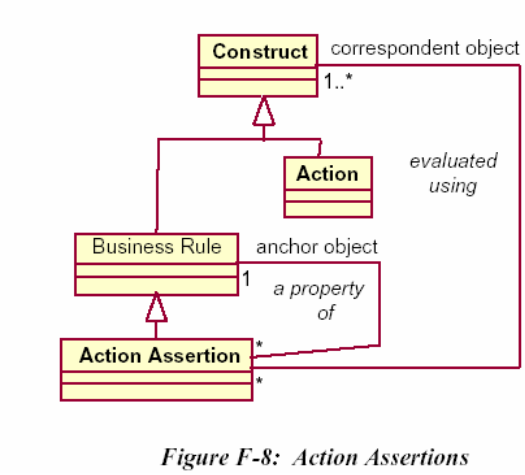
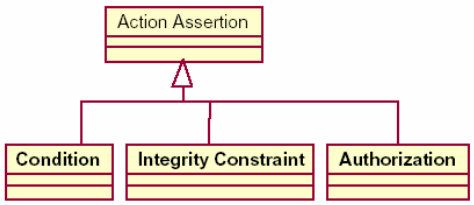


Figure F-8: Action Assertions

行为规则可以按第一种分类方法分为：



条件规则（condition），一般以“如果……那么”的形式出现，表示在满足某种条件下，某个活动可以进行或某个规则有效，例如，*如果活期帐户的人民币可取金额为0，客户不能进行人民币取款*。

完整性约束（Integrity Constraint），表示必须永远为真的声明。例如，

客户必须开户。

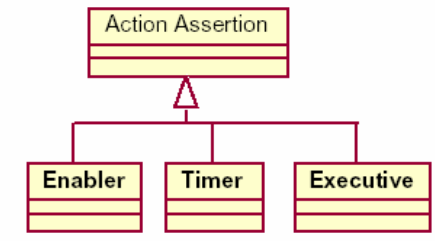
活期帐户只能属于一个客户。

授权（Authorization），表达：

（只有）X 可以做 Y 的授权规则。

X 一般是某个人或某种角色，Y 一般是 Action，例如，*客户可以进行电话查询*；*客户可以进行电话转账*；*只有储蓄柜柜员可以为客户提供柜台存取服务*等。

如果着重于对活动产生的影响，还可以这样分类（这种分类方法和上面的分类是重叠的）或分出更多的类别：



使能规则（Enabler），

定时器规则（Timer）

和执行规则（Executive），

其中：

使能规则指允许某个对象的创建，某个规则成立，或某种活动可以执行，例如*客户可以提前还贷*。

定时器规则指检测满足某种时间要求的条件是否满足。

执行规则指需要或导致某个活动（Action）的执行，强调动作的执行。

例如：

每个月 10 号，银行结算上个月的信用卡帐单。

在上面这个例子里，*每个月 10 号*是一个定时器规则，而*结算信用卡帐单*是一个执行规则。

衍生规则

衍生规则指的是如何从其他基本业务规则计算或推导出其他业务规则的方法（规则）。

计算

计算的例子如下：

如果有以下的两条基本规则：

客户通过 ATM 取款，一天累计不得超过 3 次。

客户通过 ATM 取款，每次不得超过 2000 元人民币。

其中，**客户**是术语；ATM 是业务术语，在术语表中定义；上述规则本身就是对客户取款的一种约束；并且可以衍生出：*客户通过 ATM 取款时，一天累计不得超过 6000 元人民币。*

在这里的计算规则就是：每天*最大累计金额*=每天*最多取款次数*×*每次最高取款金额*。

推导

推导是另外一种衍生方法，例如有以下的基本规则：

1．储蓄柜柜员组拥有现金存取权限。

2．储蓄柜柜员属于储蓄柜柜员组。

3．成员拥有所属组的权限。

可以推导出：

4.储蓄柜柜员拥有现金存取权限。

这里的推导规则是：*1 and 2 and 3→4*

谢谢 PureArch 指正。

参考资料：Defining Business Rules ~ What Are They Really?———Business Rules Group

人件 - Microsoft Internet Explorer

文件(F) 编辑(E) 查看(V) 收藏(A) 工具(T) 帮助(H)

后退 媒体

地址(D) http://www.peoplewarecn.com/ 转到 链接

《人件》— 老板，别把开发人员当成牲口

» 新闻

《人件》提供预订>>

互动出版网 WWW.CHINA-PUB.COM

Cnforyou.com

华储 www.huachu.com.cn

» 相关文章

关注程序员自己的文化——专访Tom DeMarco

别把开发人员当成牲口

Brooks在《人月神话》中的评论

Alan Cooper的评论

办公室空间，下一场革命

《人件》在计算机行业的实践（待续）

人的问题：关于《人件》

Edward Yourdon的评论

开发人员是人吗?

» 各版本封面

英文，日文，德文

首页 购买 留言板

完毕 Internet

在大型过程公司中应用 XP

James Grenning 著，袁峰 译

查看评论

摘要

本文描述了在一家使用传统正规过程的公司中应用了很多 XP 实践开发的一个项目。其中介绍了如何将 XP 建议给管理层、如何开始项目、怎样进展。并列举了实际执行过程中小组在最初 6 个月遇到的一些问题。

关键词

XP、极限编程、软件工程、过程改进、面向对象设计、软件项目管理

1 介绍

本文介绍一个 XP 的实际应用案例，应用背景是一家过去使用大型正规软件开发过程的公司。该公司从事临界安全系统（safety critical system）的开发，下文中简称该公司为 DPS（Defined Process Systems）。DPS 正在开发一个新系统以取代遗留的程序，经过系统工程的努力，系统已经被分解成多个子系统。这时作者参与进来帮助其中一个小组在项目中使用迭代开发、用例以及面向对象设计等技术。该项目的目标是实现一个运行在 Windows NT 上的嵌入式应用、和其它机器共同组成一个网络，相互协作并对外提供服务。下面，作者将按照项目实际的执行顺序向大家详细介绍有关情况。

项目刚刚开始，我就参加了名为“XP Immension I”的培训课程，用了一周时间学习有关极限编程的技术和哲学。受 XP 的影响，我和小组成员一起讨论是否能够将某些 XP 实践应用到我们的项目之中，和 DPS 的标准过程不同，XP 使用的是用例驱动及面向对象的迭代的开发过程。当时 DPS 也决定试试迭代开发和面向对象技术。但这两者肯定是不同的，到底怎么不同呢？不管怎样，我们决定向总工程师申请试一试 XP。

2 当前形势

标准实践也都是针对各种问题量身定制的。举个例子：对于大多数开发人员经验不足的问题。标准实践的做法是让高级开发人员担当评审员的角色，确认低级程序员的工作。他们倾向于在开发中采取正规的评审过程，并由小组严格执行。这方面也已经积累了很多工具，不少步骤都已经自动化。例如：通过网络征询意见和缺陷。

在采取 BDUF（译者注：“大规模前置设计”，指一种在着手进行程序代码的撰写之前，就先按照既定的程序分析、设计、制图、撰写文件等等耗时费力的工作方式。）和阶段包含（phase containment，译者注：指在各个阶段及时发现错误的方法）等过程时，这种做法是无可置疑的。如果不是因为一个问题的存在，我现在就可以结束这篇文章了。那就是：这个过程到处都在给软件开发增加额外的负担。不管设计什么，都要先写设计、在 Rose 里画出来、安排时间召开评审会议、分发评审材料；然后评审人员还必须阅读这些材料、在网上提交意见、召开评审会议、在网上收集意见及缺陷、修正文档并检查。光想想这些，我都快要疯掉了。

更糟的是，所有这些前期（up and front）工作并没有排除集成和部署阶段的 bug。工程师们经常在系统工程和管理层的指挥下忙得团团转，根本就没有机会使用他们的经验，就这样，deadline 和延期的警钟还总是在他们耳边敲响。

开发小组不知道什么时候可以开始项目，这个阶段的需求就像散文一样，除了作者之外，谁也看不懂。

总结一下需要解决的问题吧：过程中的额外工作太多、工期太紧、工程失控、需求是单方定义的（没有获得一致的理解），但项目却需要马上开始。这么多问题，我建议采取某些极端的方法，例如 XP？我相信它会有所帮助。

3 擒贼先擒王

我刚刚说过，DPS 实行的是 BDUF，其过程中有很多的文档、web 页面、评审和建议。如果告诉他们新的开发过程：仅仅在字条上写一些产品功能，不作任何概要设计、马上开始编码……他们肯定会考虑把你送往精神病院！对于那些不熟悉 XP 的人来说，这听起来确实象是在开玩笑。我们怎样才能说服这些反对者呢？

在小组中普及 XP 时，我们认识到当务之急就是要把 XP 的这套理论推销给管理层，和一个好的律师一样，我们准备了一些问题和对答，按照我们的预想，决策层会认为其中一些实践很危险，而且不会在 DPS 中奏效，要知道，文档化的需求已深深扎根于文化之中，决策层肯定会认为我们根本不想写文档，他们会问：代码能代替设计吗？真的可以不进行概要设计吗？重构时出现问题怎么办？代码评审呢，不要了吗？

擒贼先擒王，这是我们确定的原则。Paraphrse Kent 认为，一定要完整地实施 XP，而不是做一些不做一些。这实在是非常伟大的建议，但如果真要按他的要求来做的话，我们的项目可能永远也开始不了。我们需要将一些能够尽早带来成效的实践先应用到项目之中，同时努力避免因没有实施其它实践而带来的损失。这并不是说，我们就挑那些不喜欢的实践搁置起来，我们总是试图实施尽量多的实践，希望通过它们来向反对者推销我们的理论。

首先要清楚项目的目标，我们需要的是操作可靠的可运行的产品、保证高效维护所需要的足够的文档（不要太多，也不要太少），以及具有可读性的源代码。这些要求就象母爱以及苹果派一样，实在是再客观不过了，标准过程同样有这样的要求。

可信赖的工作产品是项目的基本要求。对于临界安全系统来说，这一点尤为重要。提一个很简单的问题：文档怎样才算足够？这并不是废话，在这个项目中，用 XP 完成的目标程序只是由很多小组协同开发的大规模系统中的一小部分，其它小组并没有使用 XP 或短迭代开发，而是使用标准 DPS 过程。这样，在 XP 小组和其它小组之间就存在潜在的不协调。

如果我们张口就说，什么文档都不要了，肯定没有人会理睬我们。我们必须对问题做出合理的解答。首先，为什么要写文档？答案是：必须要有足够的文档支持系统的维护。对我们来说，不仅需要清晰可读的源代码，为了解决前面提到的小组之间的沟通问题，还得要有各种形式的接口文档，另外，还需要专门的文档来定义产品的需求。这些回答和 XP 的书上写的并不一致，但它保证了别人愿意继续听我们说下去。一定要记住一点：XP 并不是反文档，只是在 XP 的思想中，文档都是有成本的，有时不写文档也许成本会更少一些。当然，用传统思想的眼光来看，这实在是太反叛了。

接着，我让小组成员考虑 Kent Beck 的《Extreme Programming Explained》书中有关变更成本的内容，主管、管理者和一些高级技术人员都认为 XP 能够解决他们现阶段开发中的一些问题，但肯定不能完全照搬书上的。那么，要对 XP 做哪些改动呢？

文档和评审肯定是最大的拦路石。这样，按照擒贼先擒王的原则，我们一个个地来回答这些反对意见。

反对意见：在卡片上写“需求”!“我们不能交给测试组一堆卡片吧”，“硬件部的 Bob 也需要这些卡片”，“有人会不小心遗失这些卡片的”。

我注意到标准过程中用到了按照 Alistair Cockburn 格式书写的用例，用的是文本方式，类似于用户故事（user story），但更清楚一些。和我们协同工作的其它小组也有查看用例的需求。既然这样，我们决定在项目中放弃这个 XP 实践，而采取和其它小组一致的用例。

反对意见：为了工程维护阶段的需要，文档是必须的。而且我们需要高级工程师来评审你们的设计，保证你们没有大的错误。

对这一点，我们的回答是这样的：对软件后期的维护者来说，清晰、简单的源代码是最好的，而不是大堆过期的文档。维护者总会去看源代码的，这一点勿庸置疑。因此，应该做到高质量的代码和少量的文档，这样，他们才更容易看懂。XP 要求源代码清晰而有表现力，重构是达到这一点的一种方法，源码就是设计！

反对意见：一个人认为是可读的源码，对另一个人可能并不是这回事。

XP 解决这一点的方法是结对编程。如果两个人结对致力于代码的可读性，第三个人来看这个代码，可读性应该也不错。另外，“集体拥有代码”使得需要的时候，任何人都可以修改源代码。

反对意见：光有代码还是不够。

项目存档时，维护者需要的不仅仅是源代码。而且项目交付或移交给其它小组时，需要一些文档或者说明。对维护者来说，最好的文档就是和项目交付时状态一致的文档。写文档应该保证一点：写下来了的就是不需要再改了的。毫无疑问，维护开始之后，没人会再去理会那些文档。文档中不需要写细节，另外，要保证文档处在足够高的层次上，这样，维护时一些小的改动和修改 bug 不需要改动文档。文档的作用是定位到相应的代码，剩下的细节，应该由高质量、可读性好、简单的源代码来说明。

遵循以上就不会产生庞大的文档。记住，自动的单元测试以及系统中对象实际使用的例程代码就是最好的细节文档。XP 中并不禁止文档。但一定要牢记一点：文档是有成本的，一定要确信有所值然后再写文档。可以在迭代中制定文档的任务，另外，我们还建议，在编译代码时再写文档，而不是一来就写。

反对意见：项目尾期，没有人会去写文档。

是否可以这么理解这句话：在项目中一点点地写文档，并且一遍遍地去修改、重写？听起来这种方法应该更浪费时间一些。事实确是如此。管理层应该坚定立场，将高层次的文档任务安排在项目的收尾阶段进行，这样可以减少开发中浪费的时间。

但反对者根本不买帐，并且提出了最后的反对意见：我还是不相信你，如果设计不好怎么办？那只有等到项目结束才能发现错误了？

对管理层来说，结对编程还并不足够让他们放弃评审过程。但现阶段评审过程最大的问题就是导致开发过程的缓慢。我们可以看看现在的过程：先设计、在 Rose 里写文档、安排评审会议的时间、分发材料、所有人评审材料、召开评审会议、收集意见并解决之，也许还要再次评审，最后写代码，看设计是否有效。毫无疑问，这些工作导致了昂贵的变更成本。另外，一些有益的想法也有可能在这个过程中被否决掉。但是，鉴于这个系统的性质：临界安全系统，管理层不会放弃评审。

这里，我推荐一种进行评审工作的更有效的方法：每月设计评审。在项目的各个阶段，让开发团队一次连续工作一个月，然后进行一个 design-as-built 的评审，这和严格的评审过程不同，不会延缓开发进展。我们在每次迭代中记录重要的变更，由评审小组进行评审；评审员提出的意见作为下一次迭代的输入，类似于用户故事。另外，建议在迭代中花些时间来记录当月的设计。事实证明，这种建议，高层比较容易接受。

4 和 XP 无关

这些实践的目的都是一致的，那就是：构建可预见的更快更好的软件。

我们前面说了很多，大家一致同意使用很多 XP 实践：测试在先、结对编程、短周期迭代、持续集成、重构、计划和客户成员。另外还增加了一些过程和规则：用例、每月设计评审和一些文档。在软件的开发过程上有一些明显的变化，我们也希望能够证实这种变化是一种改进。老板总有他的考虑，他认为 XP 许下了很多诺言：提高产品质量、加快开发步伐、提供更好的可预见性以及更好的工作满意度（smiles per hour）。但他觉得我们是在纸上谈兵。除非我们展示这些潜在利益中的一个给他看看，并且不带来其它负面效应，他才觉得这值得一试。如果能带来更多好处的话，当然更好啦。

软件企业中存在种种问题：质量低劣、交付延期、交付周期太长、程序员过度疲劳等。对于饱受这些摧残的小组领导来说，XP 就象是天籁之音：对测试的重视和结对编程可以保证产品的质量。另外，XP 的迭代特性有助于控制项目的进度并给予管理层必要的回馈。

好了，应该已经说得很清楚了，XP 就是值得期待的一系列技术的集合。

5 探索—开始喽

许多项目都会在前期消耗过多的时间，在瀑布式开发中这一点尤其突出，只有等所有的需求完成之后才可以进行设计工作。而在 XP 中，两周左右的用户需求获取之后就可以开始设计工作了。要在项目后期腾出两周时间是非常困难的，而在项目初期，得到一个月的用户故事之后马上就进行开发，这样可以很容易地节约一个月的时间。用户故事的发现和实现工作可以同时进行。

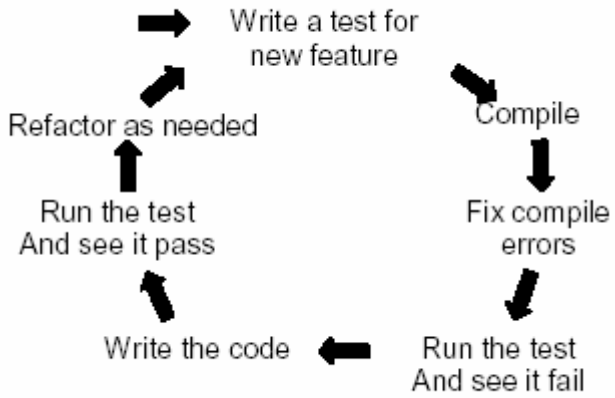
项目顾客（这里是系统工程师）已经写过一份需求文档。我们通读了这份文档，并从中识别出功能需求、记录在卡片上。这是用例识别的过程，其中并不需要理会用例的细节，只要为每个用例取个名字就可以了。几天之后我们就得到了 125 个用例。从中可以很容易地找出那些最重要的用例。

在 XP 中，需要挑选出最重要的用户故事并和程序员进行讨论。虽然我们使用的是用例，但这并不构成妨碍。首先，由用户挑选出有价值的用例并进行细化。我们决定在第一次迭代中先不考虑用例扩展（Alistair Cockburn）。Alistair 方法的用例扩展包括区别于用例正常事件流的特殊情况和变化。在早期的迭代中，尽可能地让产品简单化，假定没有任何特殊情况和错误。在项目初期，必须要保证设计能够而且将会得到改进。否则前期的设计工作将会使你陷入分析的沼泽之中。在明白这一点后，我们在识别功能之后立即着手构建系统。顺便补充一句，我们并不使用用例图，那并没有什么好处。

6 第一次迭代、

第一次迭代时，小组中有三个人：一个客户和两名开发者（包括我）。我们建了一个 CppUnit 并集成到 VC++ 开发环境中。这花不了多少时间，工具都相当好用。我们第一次迭代的主要目标是得到最终产品的一小部分，得到一些经验、掌握一些技巧并收获自信。

客户为我们提供了最重要的用例。他选择了一条主线，忽略了那些包含特殊案例的用例部分（用例扩展）。这一点确实有助于在迭代的每个阶段进行简化并限定范围。这样，设计是由主要的功能驱动的。至少在短期内简化的假定降低了代码的复杂度。这里引用一句话，是 Grady Booch 曾经引用过的 J Gall 的话：“复杂系统总是从简单系统进化而来的”。



（图中对应的翻译： 写对应新功能的测试、编译、使编译通过、运行测试，fail、写代码、运行测试，通过、需要的话，进行重构）

我们确定了第一次迭代要实现的功能，并召开计划会议。我和我的拍档都有 OOD 的经验但却没有 XP 的实战经历（不算我参加 XP immension 培训的那一周）。我们希望在第一次迭代中有一个明确的方针，因此花了一天半通过白板的手段来考虑我们的设计思想。因为我们对 XP 都不熟，不敢确信真的可以不要设计就编码。所以还是决定做一点设计，或许可以称为 LDUF（Little Design Up Front）。我们设计了一组相互协作的对象，觉得它们已经可以满足这次迭代的要求了。在备份了白板上的类图和顺序图之后，OK，可以开始了。哦，对了，我们没有用 Rose，我们是徒手画的图，因为这些肯定是要变动的，我可不想白白浪费时间。这次讨论帮助我们理清了思路，并清楚了要做的事情。

我们要构建的系统中还有定制的硬件部分，而且在开发软件的时候硬件也在开发之中。硬件还不存在，怎样写代码呢？我们定义了接口来模拟硬件的行为并不断改进。这样，真实的硬件便和代码分离开来。但还有一个问题：硬件还不存在，该怎样处理程序逻辑中对硬件的依赖呢？

如果硬件存在的话，代码中可以直接调用硬件提供的 API，否则代码都写不出来。真是这样的吗？需要指出，清晰而富有弹性的软件对实施 XP 是很关键的。软件越依赖于运行环境，就越不容易对其进行独立的测试。因此，如果依赖于硬件的话，弹性就会降低。硬件的变动（如部件更新、成本削减、新技术等等）也会带来麻烦，我不希望这些变动影响到我的程序。

没有硬件意味着要对它进行抽象。可以在开发过程中从程序中发现需要和硬件进行哪些交互。我的一个信条是：接口是为客户端服务的。因此应该由客户端来决定硬件需要进行哪些交互，以及提供这些交互的接口。它带来的好处就是客户端代码可以独立于硬件进行测试。要构建高效的单元测试案例，独立性是关键。听起来这有点象 BDUF 了，但实际上并非如此，因为这里不需要找出所有的接口细节，只要指出接口提供的方法即可。在 C++ 中，接口通过抽象类（这种类中全部是虚方法）来实现；而 Java 中则是用 interface。关于独立性的讨论在数据库模式、GUI 和协议等各领域都不少。要了解这方面的更多细节，可以查看 Robert Martins 的论文《principles and patterns》。

在迭代计划会议和 LDUF 上，我们得到了实现第一次迭代的功能所需要的接口。它们的作用就和真实硬件的插槽类似。我们为每个接口增加模拟。通常说来，这些接口提供封装的数据并将输出写入到文件中。另外，我们还为每个接口写了一个例子。

接着就可以开始写代码了。从 LDUF 手工画的图上可以得到候选的类。按照测试在先的原则，应该先写测试，刚写出来的测试当然是无法编译通过的，因此必须先让它通过编译。运行测试，failed。接下来的工作就是让测试通过，最后进入维护阶段。

我们建立了结对编程的一些原则：测试用例、模拟器和接口类都可以单独完成，其它的部分都必须采用结对编程的方法。在第一次迭代时，我们总共只有两个开发人员，自然就是一对。需要指出，开会是最耽误生产效率的事情，而结对编程非常有意思而且很紧张。在结对编程时我们总是专注在任务上，一个人一跑题，另一个人会马上把他揪回轨道上来，另外，结对编程时两人可以很好地交流工具的使用、学习新的技巧。

我的拍档和我的编程风格相似。如果不是这样的话，双方需要协调出一种一致的风格。目的是让代码看不出来是由很多人写的。为什么？因为这样可以保证代码的易于阅读。我们并不想要那种复杂的代码规范（有这么一份，但从来没人看它），只是希望代码看起来都是一样的。很多人有自己的编程风格，但风格是可以改变的，这并不是问题。在有新人加入时，代码规范已经建立。最基本的编码标准是：“让别人看不出来哪些是原来的代码，哪些是新代码和修改过的代码。”精辟，一句话的编码规范！但现实往往是残酷的，实际上，我们被要求必须遵循代码规范，好吧，按照“擒贼先擒王”的出发点，这还不是我们擒的“王”，我们可以在这一点上妥协，就用这个标准吧：在每个函数前面写上注释块。这其实用处不大，真正管用的是类级别的注释，它们描述类的作用。完全按照上面的标准，经常会在代码中加入很多垃圾，反而混淆了程序员的视线。

7 更多迭代

考虑到在第一次迭代中使用了很多假定，我们想了解系统如何处理那些特殊情况的引入。重构能行吗？能够在保持设计清晰的同时添加功能吗？事实上，一切都很顺利！添加新的功能的时候对代码进行重构，使之更清晰；有时候为了让新功能看起来更自然，也要对代码进行重构，然后继续添加新功能。非常顺利，就和书上说的一样！

另外一个重要假定是系统同时只支持一个单一的用户交互。这样的设计能够支持 50 或 100 个并发交互吗？伸缩性如何？我们决定使用 Active Object 设计模式，这样可以把线程策略和程序逻辑分开。Active Object 实际上就是封装了一个线程的对象。该线程代表程序进行排队。程序逻辑和线程模型分离可以帮我们简化很多工作。我们构建了系统核心支持 50 个用户需要的实例，然后模拟最坏的情况，测试完美通过。度量系统的性能，我们发现即使更糟的负荷，也只需要 CPU 大约 5%的资源就可以对付了，所有的 IO 都已经建立并模拟。这样在开发的初期我们就确认了一点：我们选择的线程模型不会导致负载处理能力不够这个最大的风险。

8 精化设计

我们为每个子系统分配了功能并模拟了它和环境的交互。但新的用户故事会给子系统带来新的功能要求和复杂度。可以看到设计的发展和精化过程，核心设计在第一次迭代之后就不再改变，但系统却越来越大。设计早期，只有十二个类和一些模拟，最后有 100 个类。对于一个小的小组，这个过程花了大概 6 个月的时间。这个系统并不大，但 DPS 过去从来没有这样又快又稳地做出这么清晰而富有弹性的设计。

设计的不断精化大大减轻了开发团队的压力。再也不需要一次为我们不清楚的很多细节作出最佳设计，只要在当前基础上作出最佳设计就可以了。幼稚的设计也不会带来灾难性的后果，自动化测试和重构技术给了我们足够的自信：我们可以持续地精化设计、改善系统。按照这样的过程，我们可以大胆地添加新功能。当然，这并不是说我们可以通过重构把自动售货机变成投票系统。

XP 中并不要求使用面向对象设计（OOD），但 OO 会带来很多好处。例如：“测试在先”的技术要求每次构建、测试一小块程序。用 C 这种面向过程的语言要创建小而独立的模块是比较困难的，而使用 java、C++或者 smallTalk 等面向对象的编程语言则要方便得多。各自独立的类通过接口相互沟通，同时，接口的特性也有助于独立的测试，封装实现了对该部分程序的保护。有关这些知识，可以查阅前面提到的有关论文。

9 项目经理大吃一惊

不仅仅是程序员在为他们的工作感到欣喜，在第四个迭代之后，项目经理说话了："我到现在还只有三份文档！但是，我竟然已经有一小块可以运行的系统了！"除此之外，项目经理还发现了其它好处。一般说来，为了和其它小组协同，项目经理要花很多时间来考虑各个任务的优先级，确定任务之间的依赖关系。而在 XP 小组中，功能上的依赖性基本上都不存在；过去，功能的实现顺序是由写在 BDUF 中的软件内部框架来确定的，而在这里，则是按照用户确定的优先级。这样，XP 小组的开发非常敏捷，并且能够适应其它子系统变化的需求。

10 建设队伍

建设队伍的过程就是增长经验的过程，这个过程是缓慢的。在我们的实践中，每一次迭代可以吸纳一到两名新成员，进来之后并不急着给他们分配任务，在第一次迭代中，他们主要是参与结对编程，这样他们会逐渐了解系统和我们的过程，然后在迭代计划会议上给他们分配任务。我们不会因为给小组增加了新成员就要求提高速度，开发的速度是可以度量但不能预测的。

在 DPS 中，通过高级工程师对普通工程师工作的评审来帮助后者增长经验。而在 XP 目中，同样需要以老带新的途径，这要求小组中至少要有一个有经验的程序员，并不需要给他们什么头衔，或者角色定义，但他的存在是非常必要的，我们需要他们在小组中传播知识和经验，而在和新手的结对编程中，他们同样会学到很多。

11 向其它小组传播经验

你可以把一匹马带到河边，但无法强迫它喝水。同样的道理，仅仅是程序员想实施 XP，而管理层没有兴趣，或者仅仅是管理层想试试 XP，而程序员不配合的话，都无法成功。我遇到过的是第一种情况。

给管理层的忠告：世界上最美妙的事情莫过于拥有思维开放并且乐于尝试新事物的工程师了；要鼓励他们实施各种实践，而不是强迫；给他们提出好的指导、鼓励他们改变现状。千万不要强迫一个不情愿的小组实施 XP，那样还不如重新招募一个愿意尝试 XP 的小组。要说服高层，不要去责怪那些勇于尝试新事物的部下。另外，要寻找一些顾问，当尝试部分 XP 实践时，肯定会出现一些问题，这时候，顾问应该找出问题的症结，并告诉小组怎样应用实践来解决问题。一到两次迭代尝试一种实践，如果效果不明显或者小组成员都不接受的话，OK，可以不采用。还有，保持每次迭代的短周期，这样会经常得到回馈。

给工程师的忠告：像我前面说的那样去推销 XP，可以向管理层申请尝试一下 XP，如果效果不佳的话，一切都好商量。同样，为了得到频繁的回馈，迭代周期要短。

12 回顾

开放的工作环境，这个实践是经常被忽视的，而事实上这很重要，结对编程和协作对工作环境的要求都很高，千万别在这一点上省钱。

自动化测试也是需要的。理想化的情况是：顾客能够按照某种高层次的语言提供测试脚本。事实上，我们有这种脚本语言，但大部分的测试还是工程师书写的。应该把更多时间花在创建测试脚本上，而不是将用例步骤文档化以作为开发过程的输入。测试脚本提供了用户故事中缺少的事件发生的触发点。如果非要先将用例步骤写下来，然后再根据这个来写测试脚本的话，只能说是重复劳动。

迭代周期（两周）越短，回馈越频繁。如果原来没有 XP 的经验，两周可能短了一些。我们选择的是四周的周期，效果不错。但是两周的迭代周期回馈会更多一些。迭代周期的长度经常会有一些小的变动，这很正常。

现在，我们的小组中没有一个人怀念前置设计的方法，大家都很满意新的设计方法带来的弹性。这里我再抛出一个问题：为评审过程中准备的文档仅仅在阶段性评审中有用，对开发成员并没有太大的帮助。那么，大家不妨置疑阶段性评审，它确实有帮助吗？将时间花在什么地方效率更高？

13 信息和问题

要了解更多，请联系 grenning@objectmentor.com

相关知识

感谢以上经验的提供者（他希望自己的名字保密）。同样，感谢 Chris Biegay 和 Jennifer Kohnke 帮助我准备这篇论文。

参考资料

i Kent Beck, Extreme Programming Explained First Edition. Addison-Wesley Publishing, Co. (October, 1999)

ii ibid

iii Cockburn, Alistair, etc

iv Steve McConnell, Rapid Development First Edition.Microsoft Press. (July, 1996)

v Alistair Cockburn, Writing Effective Use Cases (The Crystal Collection for Software Professionals) First Edition. Addison-Wesley Publishing, Co. (October, 2000)

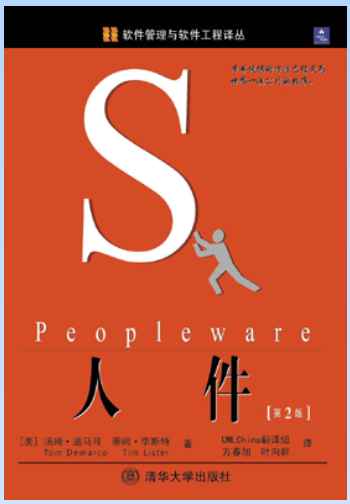
vi Booch, Grady. Object-Oriented Analysis and Design with Applications. Second Edition. Santa Clara, CA. The Benjamin/Cummings Publishing Company, Inc. Second Edition. Addison-Wesley Publishing, Co. (February, 1994)

vii Gall, J. 1986, Systemantics: How Systems Really Work and How They Fail. Second edition. Ann Arbor, MI: The General Systemantics Press (1977)

viii Robert Martin, from Principles of Object-Oriented Analysis and Design class

ix Robert Martin, Principles and Patterns, www.objectmentor.com

x Schmidt, Douglas C. and Lavender, R Greg. Active Object, http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf



Tom Demarco，Tim Lister

翻译：UMLChina 方春旭 叶向群

http://www.peoplewarecn.com

人件中文版网站

Frederick P. Brooks, Jr--《人月神话》第 19 章

近年来，软件工程领域的一个重大贡献是 DeMarco 和 Lister 在 1987 年出版的《人件》，我衷心地向我的读者推荐这本书。

Edward Yourdon

《人件》在 1987 年出版后，立即成为最畅销的作品。当人件第一版出版时，我写了一份评论，“我强烈推荐你买一份《人件》给你或你的老板，如果你是老板，那么为你部门的每一个人买一份，并给自己买一份”。这建议在 12 年后的第二版依然有效，并且更加热烈。

Mark A. Herschberg

我只学了办公环境的章节，就辞掉了原来的工作！

Joel Spolsky

我想，微软成功的原因之一就是公司里的所有经理都读过《人件》

Steve McConnell

由于本书第 1 版的赫赫声名，新版的《人件》是我不用看就会决定购买的少数几本书之一。

预订中文译本>>

《人件》在计算机行业的实践（待续）

UMLChina 摘录整理

查看评论

公司名	业务	员工受培训小时/年	举措	《人件》章节
Xilinx	可编程逻辑器件	不详	技术低迷期不解雇员工。	17 章 自愈系统
Adobe	多媒体处理软件	51	同事间的友谊是这家以图形产品闻名的硅谷公司的代名词：经常举行的有全体职员参加的会议，岗位轮换，周五晚上的啤酒聚会。每五年有三周带薪假期。	22 章 意大利通心粉晚餐 16 章 很高兴在这里
CDW	计算机产品	68	从来没有解聘过雇员。	17 章 自愈系统
高通	通信和信息技术	20	对领取薪水的雇员来说，休病假的制度是荣誉性的。不存在雇员滥用的风险；90% 的雇员期盼病愈后马上上班。公司餐厅为员工们供应寿司，晚上加班，可享用免费晚餐。	22 章 打电话说身体好了
SAS	企业软件	32	每月 300 美元的儿童保育补贴、公司餐厅（有钢琴师）、健身房、还有诊所，这一切都降低了离职率。	34 章 社区的形成

微软	软件	不详	这家巨型公司将最聪明的人材吸引到公司富有田园风味的厂区。厂区内设有垒球场、篮球场以及带淋浴设施的衣帽间。如果你愿意，你也可以免费得到非公司健身房的会员资格。	34 章 社区的形成
思科	网络	40	规模庞大的儿童保育中心看护 400 多名儿童——这在硅谷的同行中很少见。	34 章 社区的形成
思科中国	网络	/	把“看得见风景”的窗口留给员工；允许员工调换岗位；不限制员工的办公地点和时间；帮助员工在家里安装宽带；员工平均每年参加 6 个培训班。	第二部分：办公环境（7-13 章） 2 章 做吉士汉堡，卖吉士汉堡 16 章 很高兴在这里
英特尔	芯片	45	没有专用车位、私人办公室、经理餐厅	7 章 家具警察
SGI	三维图形软件	23	工作四年后可以享受六周带薪假期。同时在公司内还提供按摩、波足桌、咖啡机、享有补贴的自助餐厅，和沙地排球场。	第二部分：办公环境（7-13 章）
SRA	IT 咨询	30	所有新雇员都能够得到 30 股股票。几乎 100%的雇员都认为公司管理层是诚实的。用一位雇员的话来讲,“这里的确以理服人。”	第四部分：培育高生产力团队（18-23 章）

IBM	计算机产品	50	在全球拥有 67 个日间托儿所（在美国境内有 61 个）。自 1984 年以来，公司已在包括子女保育和老年保健补贴在内的家属照护方面花费二亿多美元。	34 章 社区的形成
IBM 中国	计算机产品	/	晚上 6 点以后，公司放音乐，催促员工回家。如果主管同意，员工一周内可以有几天在家办公。家有学龄前儿童的女性员工可以停薪留职，请假一两年在家照顾小孩。	3 章 维也纳在等着你（世上没有加班这回事）
Network Appliance	网络存储	40	雇员行为古怪，他们自己深知这一点：一个标示牌欢迎来访者进入公司总部，而以往的销售活动都是以与真人大小相仿的身着《星际探险》（Star Trek）剧中人服装的公司主管的剪影像拉开序幕的。	14 章 霍恩布洛尔因子（爆米花不够专业） 19 章 黑衣团队
Autodesk	设计软件	50	雇员们用当场颁发 1,000 美元奖金的方法为一位同事带来惊喜。雇员们还可以享受长期的福利：在工作四年后可以享受六周的休假，同性恋配偶与政府承认的配偶享受同样的待遇。一位雇员说：“你怎么能不喜欢为一家每天都能把狗带来上班的公司工作呢?”	25 章 “自由电子”
IDG	计算机媒体	12	办公室增添了一个拱廊；另外一个办公室让编辑人员负责进行聘用面谈。	第二部分：办公环境（7-13 章） 15 章 雇用一个变戏法的人

Acxiom	综合数据服务	38	没有繁杂的行政管理，可以选择弹性工作时间，也可以在家办公。	7 章 家具警察 25 章 “自由电子”
Intuit	软件	不详	谁在度假时还想到老板？Alan Hampton 可能如此，因为雇主 Intuit 公司会为他的休假付费。雇员们在回报社会方面作出了重要的贡献：雇员们每年有 32 个小时的带薪志愿服务时间，去年社区服务的时间达到 7,600 小时。	24 章 混乱和秩序(培训、旅游、会议、庆祝和休养所)
Sun	计算机软硬件	40	95%的雇员采用弹性工作时间；100%的雇员可报销最多一万美元的学费。	16 章 很高兴在这里 25 章 “自由电子”
Electronic Arts	游戏软件	26	雇员们在感到充满压力的创造力下降的时候，为使自己精神放松，便到直径为 81 英尺的迷宫中徜徉。“在像我们这样的增长型公司，我们不缺乏灵感，我们所要做的 就是为找到灵感创造合适的环境。”公司的游戏开发者们也可以 享用免费的卡布其诺咖啡，或者打排球和篮球，以恢复精力。如果想要休息的话，在工作七年后可以享受七周的休假，在工作 12 年后可以享受 12 周的休假。	10 章 脑力时间 vs. 体力时间（顺流）
德州仪器	计算机产品	34	雇员们着装随便，礼仪也如此：从首席执行官开始，每个人都以名字相称。	第四部分：培育高生产力团队（18-23 章）

注：以上有关公司的资料摘自《财富》