

非程序员

【访谈】

1 Roger S. Pressman：CMM不是宗教

【方法】

9 面向对象开发中的本质用例及其职责

28 利用角色扮演和用例卡片进行需求复审

40 针对用户界面设计的用例结构和式样

70 网上商店的模式

80 分析模式学习笔记之四（下）

【过程】

87 《人件》节选



主编：davidqql

审稿：cindgy geng, 闫琪,
xuxu, davidqql

投稿：editor@umlchina.com

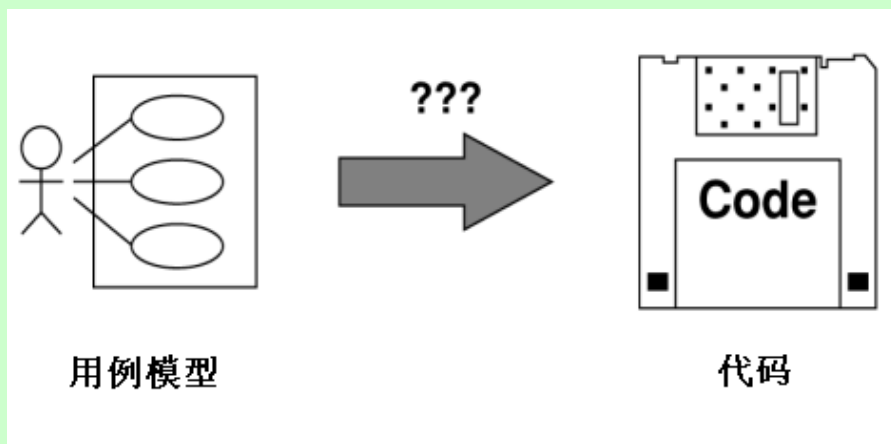
广告：adv@umlchina.com

反馈：think@umlchina.com

2002-3 11

UMLChina 培训

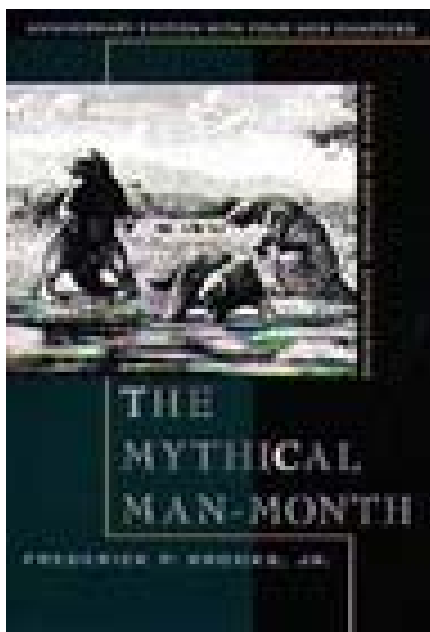
The real thing



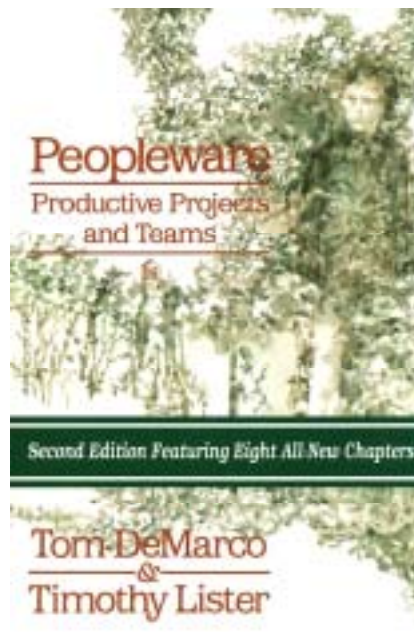
利用 UML 的 20%就可以为 80%的问题建模
-- 《UML 用户指南》，第 32 章

详情请垂询：think@umlchina.com

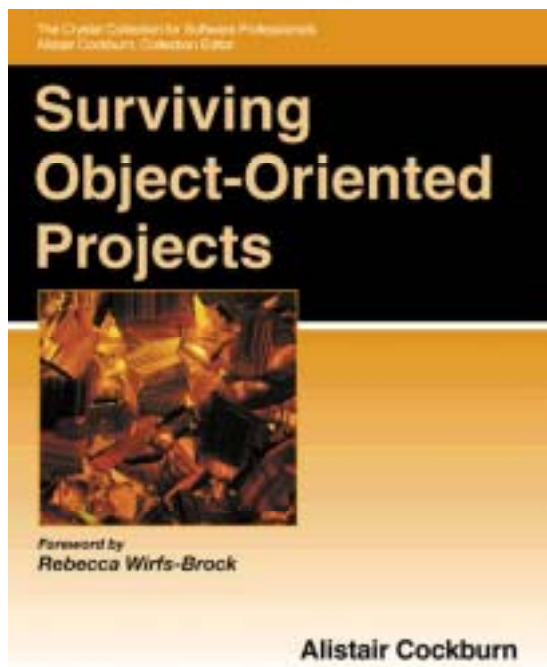
中文译本即将发行！



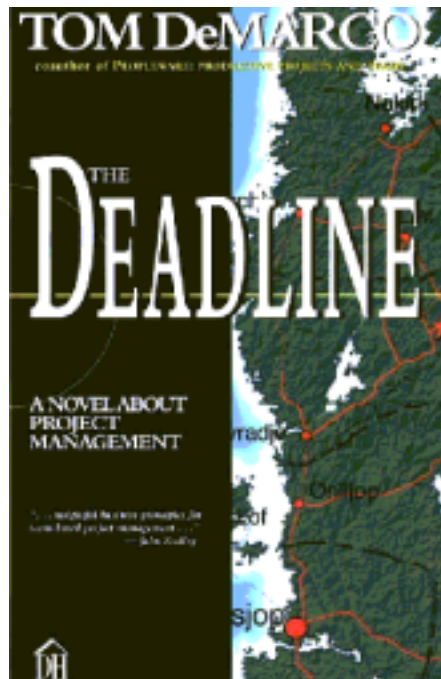
《人月神话》20 周年纪念版，Fred Brooks
翻译：UMLChina 翻译组 Adams Wang
散文笔法，绝无说教，大量经验融入其中



《人件》第 2 版，Tom Demarco 和 Tim Lister
翻译：UMLChina 翻译组方春旭、叶向群
微软的经理们很可能都读过—amazon.com



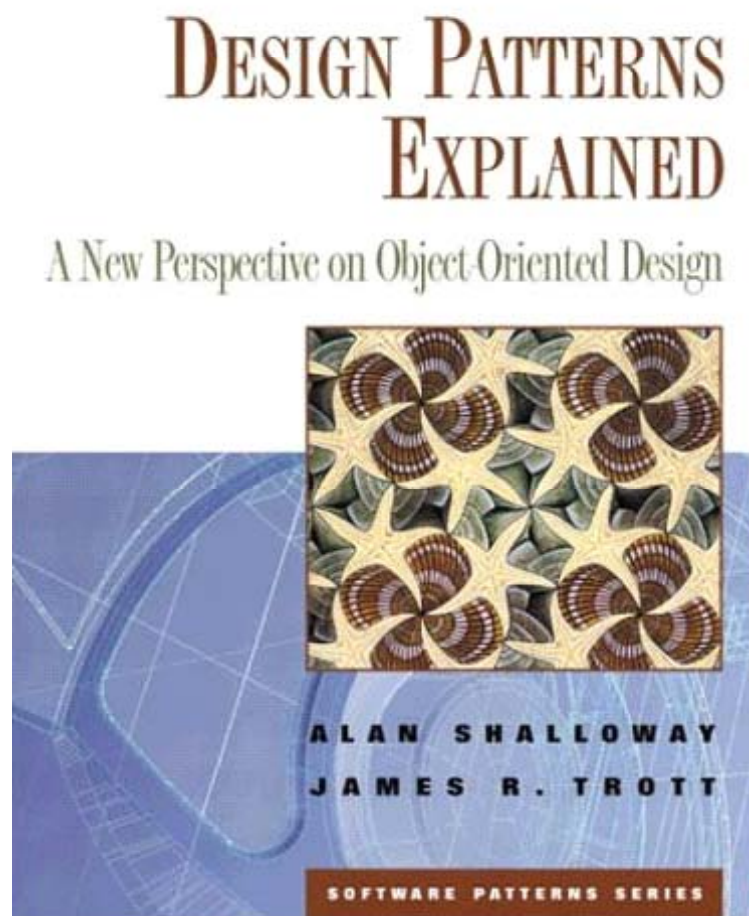
《面向对象项目求生法则》，Alistair Cockburn
翻译：UMLChina 翻译组乐林峰
Cockburn 一向通俗，本书包括十几个项目的案例



《最后期限》，Tom Demarco
翻译：UMLChina 翻译组透明
这本最通俗，这是一本软件开发小说

看不懂《设计模式》？

它的作者推荐《设计模式精解》----中译本即将上市！透明 译



直接学习《设计模式》这本书是非常困难的。作为入门，我推荐 Alan Shalloway 的《设计模式精解》。

--John Vlissides 写于 [UMLChina 答疑板](#)

Roger S. Pressman

CMM 不是宗教



编注：北京时间 2002 年 1 月 10 日晚上 21:30-23:30，Roger S. Pressman 先生作客 UMLCHINA 讨论组的聊天室，畅谈软件工程。

Roger S. Pressman 博士是软件工程领域国际知名的咨询专家和作者，R.S. Pressman & Associates, Inc 的创始人，已有超过 30 年的产业经验，主要从事工程产品软件和系统软件的开发技术工作和管理工作。作为产业的实践者和管理者，Pressman 博士主要从事的是航空航天应用中高级工程和制造的 CAD/CAM 系统的开发，他也从事科学及系统程序设计方面的工作。除了他的产业经验之外，Pressman 博士还是 BridgePort 大学计算机工程系的兼职教授和该大学的计算机辅助设计和制造中心的主任。Pressman 博士是杂志 American Programmer (美国程序员) 和 IEEE Software (IEEE 软件) 的编委，是 IEEE Software 的 Manager (管理员) 专栏的编辑。他还是 ACM、IEEE、Tau Beta Pi、Phi Kappa Phi、Eta Kappa Nu 和 Pi Tau Sigma 的会员。

Pressman 博士著有 6 本书籍，其中最著名的是已经发行了 20 多年的“Software Engineering: A Practitioner's Approach”，此书被全世界广泛使用为软件工程教材，最新版为第五版。第五版影印本由清华大学出版社发行，第四版中文译本“**软件工程--实践者的研究方法**”由机械工业出版社发行。。以下是交流实录。由[钱五哥](#)翻译。[原文链接](#)。[答疑板链接](#)

csliuyb：您认为编程和软件工程的关系是什么？

rogerpressman：软件工程需要过程的约束，写程序相对非正式一些。两种工作的目标是相同的

h_gaofeng：您如何看待 RUP 和 CMM？

rogerpressman：任何过程模型和评估成熟度的相关方法都是有意义的。但是必须避免过分强调过程。每个机构必须自己选择…

umlchina：您如何看待 XP (eXtreme Programming)？

rogerpressman: XP 是优秀的想法，但它并非软件工程的替代品。我认为它是软件工程的一个变种。

h_gaofeng: 如果我们还没有弄清楚需求，但经理命令我们必须开始设计和编码，这时应该怎么办？

rogerpressman: 你必需理解需求。有很多方法来得到需求。一种是原型法，这种方法使经理们满意，也能得到你所需的信息。

Charity_Zhou: 如何为团队或项目选择适当的过程模型？

rogerpressman: 选择适当的过程模型决定于团队成员的知识、项目特征和很多其他因素，参见 <http://www.rspa.com/apm>

vickyday: 您能够在软件规模估计方面给我们一些指导吗？

rogerpressman: 软件规模：有两种方法，一种是 LOC（编注：代码行），目前广为使用但很有争议。另外一种为 FP（编注：功能点），它不便应用于工程性项目，但完全适用于 IT 应用。

qlw5: 争议？

rogerpressman: 我怀疑很多新型语言中很难使用 LOC，如可视化编程，但是如果适当使用，还是很有效的。

vickyday: 但如何更有效地估计呢？

rogerpressman: 为了更有效地估计，你必须用多种方法来估计，然后整合各个估计值。

seeseax: 下一代 UML 是什么？UML 带给我们什么？

rogerpressman: UML 是很重要的分析设计方法，虽然很多项目使用它，仍有很多软件工程从来不用 UML，因为他们担心其前景。在考虑下一代 UML 之前，我希望看到现有方法的广泛应用

Charity_Zhou: 我正在看您的书《软件工程：实践者的研究方法》第五版，有什么忠告吗？

rogerpressman: 关于我的书：很长☺，选择你想看的东西，留心相关网站

xuefg: 您如何看 CMM？

rogerpressman: CMM 是很好的模型，但决非宗教信仰。必须以平常心看待它

jameshwoo: 您认为一个机械工程背景的人可以成为一个好的职业软件工程师吗？

rogerpressman: 很多年前，我还年轻的时候，我第一个工程学位也是机械工程。真是巧合！

qlw5: CMM 是否适用于绝大多数项目，哪种类型的项目最适合？

rogerpressman: CMM 可以应用在各种项目中，但是对于快速响应性和小型项目未必适用。

siltboy: 您怎样看测试在软件工程中的地位？美国公司如何测试它们的软件？

rogerpressman：测试是软件工程中最弱的一环，也是最耗时间的一环。我们需要好的策略和战术

qlw5：新项目可以使用 CMM 模型吗？

rogerpressman：新项目可以使用 CMM 模型

vickyday：Rational 的工具是否对开发过程帮助很大？

rogerpressman：Rational 的工具极为有用，能加深人们对软件工程的了解。但更重要的是软件工程自己要掌握好的实践。

siltboy：对 OO 测试有什么建议？

rogerpressman：OO 测试与传统测试不同，我的书中的第 23 章有较完整的讨论。

mhuang：如何看待应用开发中体系结构或基础架构的重要性？

rogerpressman：体系结构已经成为一种最重要的设计观念。底层结构（Infrastructure）依赖于好的体系结构。

brook0344：您说的"小项目"是指什么样的项目？

rogerpressman：一般来说，小型项目是 2~3 个人、不超过 2 个月的项目，也不是关键项目。

siltboy：请问 OO 集成测试和单元测试的区别是什么？

rogerpressman：你关于 OO 的问题很好，但这超出本次讨论的范围，参见我的书，里面回答了你的问题

netd：能举例说明体系结构（architecture）和底层结构（infrastructure）的区别吗？

rogerpressman：简单的说，体系结构描述了系统中处理和数据部件间的关系，关于它们的通信方式和交互方式；底层结构定义了支持软件工作的系统（例如，操作系统，外部部件等等）。

qlw5：以我的经验，我发现虽然 CMM 是通用的方法，但其实是基于瀑布生命期模型，您认为呢？

rogerpressman：我不同意 CMM 是基于瀑布模型。它定义了所有软件组织中过程的特征。

jameshwoo：您如何看待软件工程教育，特别是在中国？

rogerpressman：软件工程教育具有挑战性。最好有计算机科学背景、实际工作经验并且了解如何团队协作。

csliuyb：目前软件工程的热点研究是哪些？

rogerpressman：目前的软件工程研究集中在基于构件的开发、形式化方法和其他领域。

seeseax：您如何看 PSP？

rogerpressman: Humphrey 关于 PSP 和 TSP 的工作非常杰出，但我坚信软件工程需要大量努力来正确实施 PSP，如果投入够，效果应该很好。

siltboy: 软件复用是否能降低软件复杂性？

rogerpressman: 软件复用应该可以降低软件复杂性，但是需要复用部件有良好的设计、经过良好测试并适用于项目。

qlw5: CMM 需要计划，但只有在项目不经常改变的情况下，计划才会有用。

rogerpressman: CMM 意味着计划是必需的，但是它并不意味着我们不能改进计划。

vickyday: 我想软件工程是一门十分注重实践的学科，如果我要到美国学习软件工程，您有什么好的建议？

rogerpressman: 美国很多大学都有软件工程课程，可以到网上查。

qlw5: 如果变化太多，意味着需要另一个计划，暗示需要新的开发模型，CMM 是 10 年前开发的，为什么我们必须用它呢？

rogerpressman: 我从来没有说必须用 CMM。你要做的是采用 CMM 的精髓并根据公司的实际情况来使用。

wenjiamin: 我想 XP 会导致迷惑，您认为呢？

rogerpressman: 如果 XP 是用来避免规矩，我同意这会造成迷惑。但是如果 XP 被规范地使用，它还是很有效率的。我尤其喜欢 XP 中的配对编程。

vickyday: 如果您是一名教授，您希望从学生身上看到什么？

rogerpressman: 我不再是教授，但以前很长一段时间我曾经是。我期望从学生身上看到勤奋工作和好主意。

qlw5: 我清楚 CMM 基于实践，但 SW-CMM v1.1 已经有 10 年了，虽然精神还在那儿，但实践已经改变了。

rogerpressman: 这么多问题，我意识到 CMM 再中国很"热"。记住：CMM 不是盲目崇拜的宗教，它是可以修订来适用于你的需求的哲学。事实上，CMM 虽然已经超过 10 岁，并不妨碍它成为优秀的哲学。

seeseax: 我正在阅读 Humphrey 关于 PSP 的书，但在周围很少看到有软件过程师在实践 PSP。

rogerpressman: 我已经说过 PSP 和 TSP 需要正式培训，并规范实施。在美国也并没有被广泛使用，我想。

qlw5: 但有些人，好像是 DeMacro 还是 High SmithIII 并不同意 CMM，他们认为 CMM 是过程的负担，您认为呢？

rogerpressman: 虽然 DeMarco 和 Highsmith 都是我的朋友，但我不同意他们的观点。CMM 成为负担的原因是没有经过思考。如果领会了其精髓，它对机构的软件工程工作会有很好地帮助。

javaor：您认为怎样才是好的团队和好的领导？

rogerpressman：Jerry Weinberg 有一本关于团队领导的好书。Becoming a Technical Leader，推荐你看看。

atlas_tao：微软的 COM 在软件工程中的位置？

rogerpressman：MS COM 是基于构件的技术的一种。如果有好的体系结构，所有实现复用的技术都是有意义的。

qlw5：我想，事情在变化，例如：关键计算机资源在 CMMI 1.02 不再是 KP，但在 SW CMM v 1.1 L2 中是。

rogerpressman：事情在变化，一直如此。随着 CMMI 的发展，这些东西会和旧的 CMM 不同，但是其精神是一致的，这只能说明我们从经验中学习并修正我们的方案。

netd：我想软件工程减少而不是消除软件开发中的问题，您如何看这一点？

rogerpressman：我同意，软件工程不是银弹，它减少软件开发中的问题，但不能完全消除问题。

qlw5：以我的观点，对小组织来说，XP 比 CMM 好得多。

rogerpressman：为什么不结合 XP 和 CMM 的优点。这样你会具有好的责任心、创造性、高质量和规则。

qlw5：XP + CMM + adaptive 软件开发，可能是优秀的解决方案。

javaor：也许有一天最终用户能开发系统，而不是软件工程师。

rogerpressman：很可惜，构造复杂系统是困难的。我不认为我们已经解决了全部问题，如果确实如此，在座各位都失业了 J。

vickyday：您怎么看软件行业的产业化？

rogerpressman：多年前，人们就已经讨论"软件工厂"，软件工程的主要因素是人，看上去短期很难实现"产业化"。

qlw5：过程是基于角色好，还是基于活动好？

rogerpressman：我不想区分它们，活动必需分配到角色，角色发挥其职能以完成有效管理。我两个都喜欢。

qlw5：我目前在读一本 Highsmith 写的书，书中说 WBS (Work breakdown structure) 绝对不能用于复杂的新项目，为什么？

rogerpressman：我不同意 highsmith 的这个观点。他认为所有项目都必须自适应，因此 WBS 就没有用了。我不同意。你可以先设计 WBS，随着项目进行而修订它。如果没有这个东西，你就没有方向。

john long: 我们公司正在开始使用面向对象软件测试, 您有什么好的建议?

rogerpressman: 抱歉, 00ST 超出讨论范围, 关于这个话题, 有很多好书, 到 amazon.com 或其他地方看看。

john long: 可是我没有美元啊, 怎么买书呢?

rogerpressman: 对不起! 我想互联网上有很多免费的信息, 可以访问 www.rspa.com/spi 查看关于 00 测试的内容。

seeseax: 您如何看 UXF (UML eXchange Format), 它能象所期望的那样增进协作吗?

rogerpressman: UXF 有帮助, 但首先我们需要更多的人和机构经常性地使用 UML。

qlw5: 但是如果需求变化很大, 静态的 WBS 是没有用的。

rogerpressman: 我同意你的观点, 静态 WBS 没有用处, 随着需求和项目的发展, WBS 应该是自适应的。

qlw5: CMM 是否能用在少于 6 个人或少于 6 个月的项目?

rogerpressman: 可以, 但是必须在不影响进度的前提下使用, 必须自适应。

qlw5: 以我的经验, 小型项目可以比大型项目少一些 SPT0 (CMM 术语, Software Project tracking and oversight, 软件项目跟踪和监督)

rogerpressman: 在某些场合, SPT0 对小型项目更重要。理由: 小的变动就可能让项目无法接受地改期, 因此应该及时了解出现的问题。

javaor: 对从业者有什么建议? 努力工作, 深刻思考。

rogerpressman: 最重要的是, 从业者理解更多的软件工程知识, 现在一般的从业者对好的分析设计和测试方法并不熟悉。

vickyday: ISO9000 2000 版本怎么样?

rogerpressman: 每种过程/质量模型都有其优点和问题。ISO 9000 方法的文档工作量很重, 而且需要持续的审核 (一个好主意)。

seeseax: PSP 和 TSP 对 SPT0 有帮助, 是吗?

rogerpressman: PSP 和 TSP 使用不同的 SPT0 方案。通常, 测量和跟踪问题是个人的责任。

vickyday: 有时, 我感到困惑, 从书中学到的软件过程技术太复杂了, 很难推广, 而且需要很多时间来学习。

rogerpressman: Vickyday, 我同意。软件过程师们很少有时间来学习新东西, 更谈不上严格应用。为了达到成功, 我们必须采用更有效的方法。最好的方法是, 软件工程师学习一些好的方法, 并很好地应用。

软件工程师自己来进行学习和选择。

qlw5：形式化方法是软件工程的未来吗？

rogerpressman：FM 不会占据未来。虽然 FM 非常强大，他们对于多数 SE 来说还是太难了，难以有效实施。

qlw5：如果您设计一个 MIS 系统，您会采用什么方法？UML？

rogerpressman：今天 UML 已经是分析和设计方法的选择，但是它依赖于 MIS 系统的特征。记住，方法是重要地，但规范的分析设计方案更为重要。

qlw5：为什么您会认为 FM 不会象工具，IDE 或其他东西一样被广泛使用呢？

rogerpressman：主要原因是多数 SE 缺乏相关背景，也不愿意学习。请记住 FM (formal methods, 形式化方法) 非常强大，但我个人认为它不会统治 SE 工作。

javaor：系统的相关知识对软件工程师或项目参与者很重要。

rogerpressman：是的！理解系统，把软件放在该环境中，定义合适的外部接口，决定架构。

qlw5：您如何看待编码和详细设计？在一些项目里它们是相同的？

rogerpressman：从现代的观点看，编码和 DD 常常是一件事情。老式观点认为编码是机械的，但今天编码和 DD 通常被合并。

qlw5：你是说现代的设计方法是基于 RAD 的？

rogerpressman：不是，我的意思是从现代的观点看待 SE，设计可以通过多种途径。

netd：软件过程中的沟通是否比文档更重要？

rogerpressman：都重要。今天，我们不是很强调纸面文档，并尝试把文档作成工作产品。会谈（通过有效的会议）和评审非常重要，SE 要求人与人有效沟通。

john long：虽然超出了本次讨论的范围，我还是想问：在系统测试时，如何使用 00 方法？

rogerpressman：技术术语叫 intercall test case design，是系统测试的一部分。我在这里说不清楚，但它们使用用例、行为模型和其他机制。

vickyday：能推荐一些 FP 的材料吗？

rogerpressman：在互联网上有丰富的材料。www.rsps.com/spi 的过程度量与项目管理栏目可以作为一个起点。

qlw5：您如何看 aspect-orientated Programming(AOP)？

rogerpressman：我刚刚开始研究 AOP，所以我保持沉默，直到以后有机会。

mum：大项目和小项目有什么不同？

rogerpressman：有微妙的不同——交流、策划、协调、测量、团队结构、方法学、SQA 和控制。

vickyday：我曾经按照你的书（第 4 版）尝试对一个模块进行 FP（功能点）估计，但发现很难确定因子的值。CMM 的评估师同意我们用 FP 进行规模估计。

rogerpressman：这是个问题，FP 需要对因子进行一些主观的评估。www.fpug.org 有比我的书中更详细的描述。

qlw5：如果我说 SW CMM v1.1 将在 2-3 年内死亡，您同意吗？

rogerpressman：在 SE，啥也不会消亡，它们只是淡出而已：-）。我想 CMM v1.1 会随着 CMMI 的出现而淡出。

netd：rogerpressman，积累软件工程知识的最好方法是什么？

rogerpressman：好书。还有 www.swebok.org 也包括了软件工程的主要知识，也可以到 www.rspa.com/spi 去寻找资料。

qlw5：基于组件的分析和设计与面向对象的分析和设计的联系？

rogerpressman：OOAD 假定（至少在某种程度上）SE 会创建合适的类或从现存本地库中进行访问，CBSE 假定访问现有的商业部件。但从核心上看，区别是微妙的。

javaor：什么叫系统？

rogerpressman：一系列相关元素一起协作以满足客户的需要。客户可以是人，组织，机器或另一个系统-非正式的定义。

rogerpressman：对不起大家，我 15 分钟后有个会，因此只好说再见了。我非常愿意和各位交流，另外请原谅我的拙劣指法。祝各位好运，并深表谢意。

Alistair Cockburn

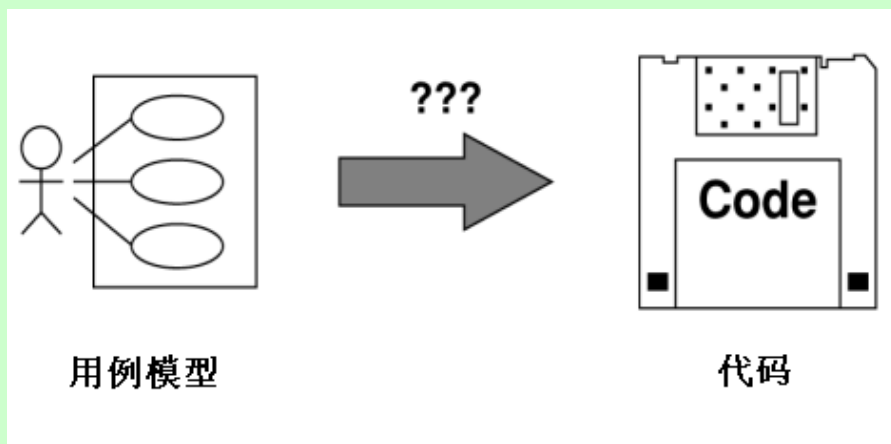


[Alistair Cockburn](#) 世界顶级 OO 专家，[全球软件业最杰出的技术与书籍的奖项 Jolt Productivity Award](#) 的获奖书籍“[Writing Effective Use Cases](#)”的作者。

»»» [答疑板](#)

UMLChina 培训

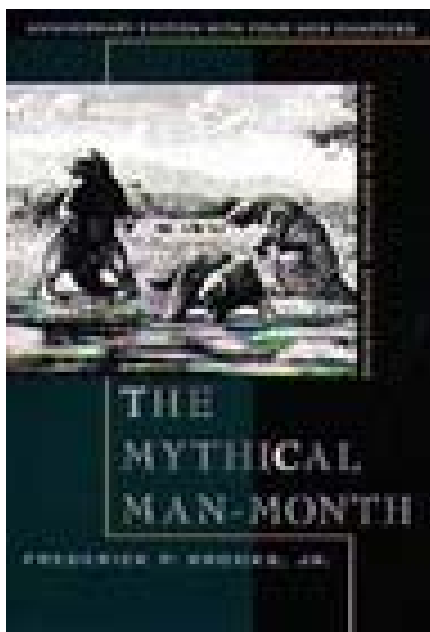
The real thing



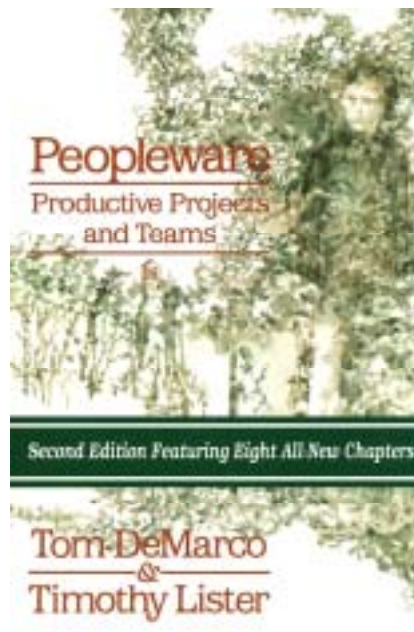
利用 UML 的 20%就可以为 80%的问题建模
-- 《UML 用户指南》，第 32 章

详情请垂询：think@umlchina.com

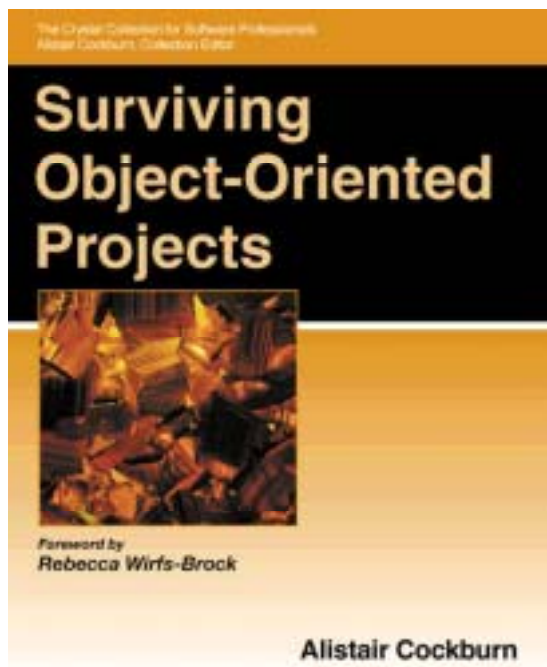
中文译本即将发行！



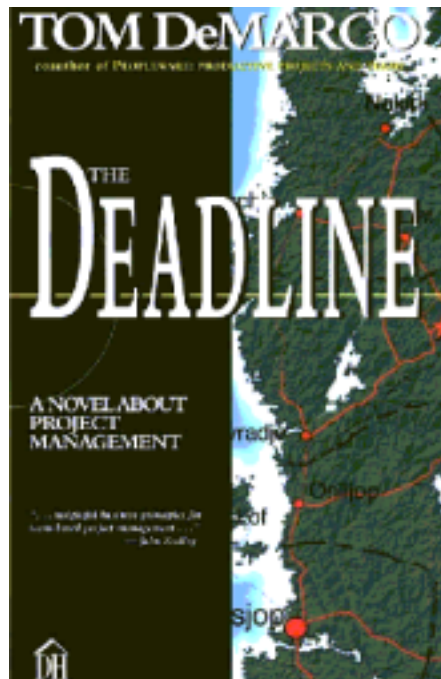
《人月神话》20 周年纪念版，Fred Brooks
翻译：UMLChina 翻译组 Adams Wang
散文笔法，绝无说教，大量经验融入其中



《人件》第 2 版，Tom Demarco 和 Tim Lister
翻译：UMLChina 翻译组方春旭、叶向群
微软的经理们很可能都读过—amazon.com



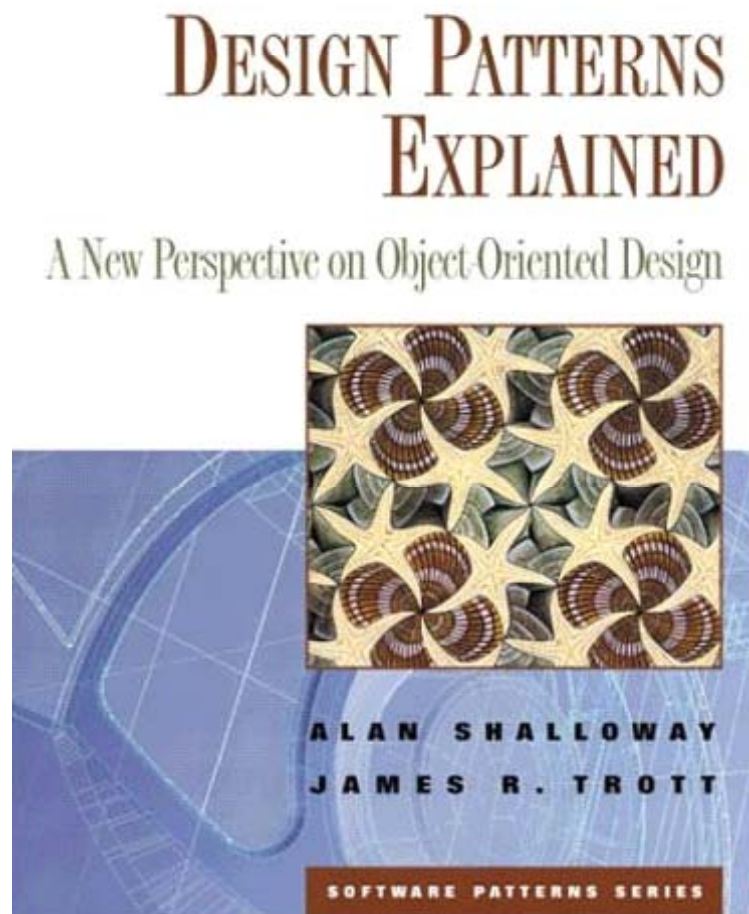
《面向对象项目求生法则》，Alistair Cockburn
翻译：UMLChina 翻译组乐林峰
Cockburn 一向通俗，本书包括十几个项目的案例



《最后期限》，Tom Demarco
翻译：UMLChina 翻译组透明
这本最通俗，这是一本软件开发小说

看不懂《设计模式》？

它的作者推荐《设计模式精解》----中译本即将上市！透明 译



直接学习《设计模式》这本书是非常困难的。作为入门，我推荐 Alan Shalloway 的《设计模式精解》。

--John Vlissides 写于 [UMLChina 答疑板](#)

面向对象开发中的本质用例及其职责

Robert Biddle 著, [zhangxxin](#) 译

【摘要】 本质用例（Essential Use Cases）是一种轻量级的方法，它简单明了，不受技术约束，用于沟通用户意图和系统职责，能够有效地捕捉用户界面的设计需求。在设计过程中，使用本质用例从系统职责中提取的关键词汇可以直接作为对象来使用，具有显著的优点。本文描述如何使用本质用例直接驱动面向对象开发过程，并实现与用户界面进行并行开发的方法。

关键字：用例 面向对象分析 面向对象设计 可追溯性

一、概述

在面向对象软件开发的过程中，用例在捕获需求过程中得到了广泛应用，并受到业界标准建模语言 UML 和建模过程 RUP 的广泛支持。Constantine 和 Lockwood 在《以使用为中心的设计》（Usage-Centered Design）中写道：“本质用例为支持用户界面设计而编写，简单明了，不受技术约束”。在一般过程中，本质用例用来产生用户界面设计，设计完成以后，又分解成几个更具体的一般用例，其中包括更详尽的细节和进一步的界面设计。但是，采用这种方法在用户界面设计完成以后才能进行实质的面向对象开发工作，费时费力，难以取得良好的效果。

在本文中，我们讨论在面向对象的软件开发过程中尝试直接使用本质用例的方法。与一般用例类似，本质用例可以作为面向对象设计的起点，但是，使用本质用例的方法进行沟通更加简洁，且无需指定太多的设计细节，便于需求搜集。

在过去两年中，我们使用本质用例方法作为主要需求搜集工具做了好几个系统的开发。我们发现，本质用例方法非常适用于面向对象软件开发，并且比一般用例具有更加显著的优点。

有三个结果需要介绍：第一，与我们推测的一样，本质用例可以直接驱动面向对象设计；其它的两个是意料之外的结果：一是本质用例提供了从需求转换到面向对象设计的指导，再有，就是保证了本质用例和对象之间的可追溯性。进一步研究发现，因为本质用例反映了用例的基本需求，从而可以由此确定用例的模式，更有效地进行需求搜集。在后文中，我们将详细描述该方法。

本文组织如下：首先介绍本质用例提出的过程，并与一般用例方法进行比较；在第三章中，介绍本质用例的提取方法，以及使用角色扮演的方法检验其正确性和一致性；第四章通过一小段例子介绍如何在设计面向对象系统时使用本质用例；第五章以应用为中心讨论在无人参与的系统中使用本质用例的方法、开发过程以及业务过程设计等主题；最后，在第六章，进行相关工作的讨论，并得出结论。

二、背景

1. 用例

Jacobson 在他 1992 年的著作中写道：“用例是与系统进行对话时行为相关的事务系列的描述。”在最近的 RUP 中，对用例的描述没有实质性的改变，它认为用例是“一系列带变量的动作描述，系统由此对特定用户产生有价值的可见结果”。

从某种程度上说，利用用例思想描述系统与外界的交互过程是行之有效的。

在软件开发的早期，用例着眼于系统交互，有助于提取系统行为，从而捕捉系统需求，确定系统规范。用格式化的语言和图形对系统的交互过程进行描述比较容易，而且易于理解，所以，由此产生的用例技术非常有效，特别适用于在大范围内进行需求搜集和分析工作。

在后续的开发阶段中，用例描述系统的交互过程。而交互过程是系统规约的具体化。在设计和实现过程中，系统规约通过结构来体现。在复审和测试过程中，用例描述测试等系统行为。在设计、实现和复审过程中，它们由同一角色引出，因此，整个开发过程具有**可追溯性**。

用例建立在一系列交互的基础上，一般来说，一个期望的交互过程总伴随与其目标（或子目标）一致的结构，因此，采用用例可以对需求进行有效的划分，通过组合、过滤、设定优先次序等方式进行组织，协助管理整个开发过程。

关于用例的优点以及用例到底是什么的讨论一直没有停止过。其基本概念已经有所发展和变化，尤其对软件开发方面的支持，例如职责描述和本质用例。我们选择本质用例作为工作的重点。

2. 本质用例

本质用例是以使用为中心设计的一部分，由 Larry Constantine 和 Lucy Lockwood 提出。用例具有显著的优点，但也有其局限性：“特别是一般的测试用例，它包括太多难以说明的，甚至是隐含的内建假设，而且用户界面设计的内容也包括在内”，这导致很早就必须进行设计决策，因为其内容成为需求的一部分，难于变更来适应以后的变化。

本质用例克服了上述问题。术语“本质”来源于：本质模型是“通过与技术无关的、理想化的、简单明了的描述来捕捉问题的实质”。Constantine 和 Lockwood 对“本质用例”的定义如下：

“本质用例是使用应用领域或用户语言描述的结构化叙述，由职责的描述或交互过程组成。是一种轻量级的方法，对职责的描述简明扼要、技术无关、可独立实现，从反映系统交互的根本意图和目的角色的用户视角出发来描述交互过程，具有完整、针对性强、定义明确的特点。

本质用例用一种格式化的文本来记录用户和系统的交流过程。这个文本与 Wirfs-Brock 使用的两列表的格式类似。虽然在 Wirfs-Brock 的格式中用“动作”和“响应”两列来讨论用例抽象的级别框架，但它也适用于用户和系统的交互。

在本质用例中相应地采用“用户意图”和“系统职责”作为两列的标识。这样，即便使用用户界面细节描述也能抽象地描述交互的过程。需要注意的是，这个抽象过程并非作为一个整体与用例相关，而是与用例的各个阶段相关。用这种方法得到的本质用例是一系列交互过程，而不是抽象的步骤。

Constantine 和 Lockwood 的例子如图 1 和图 2 所示。图 1 采用一般用例，用“用户动作”和“系统响应”进行描述，图 2 采用本质用例，用“用户意图”和“系统职责”进行描述。可以看到，采用本质用例的描述更抽象，更易于适应具体实现的变化，它更简短而且易于理解。

Jacobson 使用用例支持面向对象的软件设计；Constantine 和 Lockwood 提出了本质用例和更广泛的基本模型框架，以支持界面设计和开发。我们注意到除了面向对象的软件开发外，本质用例实际上是毫无

用处的。换句话说，本质用例的优点只有在面向对象的软件开发领域才能得到体现。

取款	
用户动作	系统响应
插入卡	读取磁卡 提示输入 PIN
输入 PIN	校验 PIN 显示交易菜单
按键	显示总额菜单
按键	提示总额
输入总额	显示总额
按键	退卡
取卡	吐钞
取钞	

图 1 自动取款系统的一般用例描述 (Constantine 和 Lockwood)

取款	
用户意图	系统职责
身份识别	验证身份 提供选择
选择	吐钞
取钞	

图 2 自动取款系统本质用例描述 (Constantine 和 Lockwood)

三、本质用例和需求

在设计过程中，CRC 技术可以在团队活动中提高对设计的理解和设计的有效性，而在用例分析和评估过程中却缺乏这样的技术。我们希望开发这样一种技术。本质用例有许多适应这种技术的特性，因此，我们开始在面向对象软件开发过程中使用本质用例。

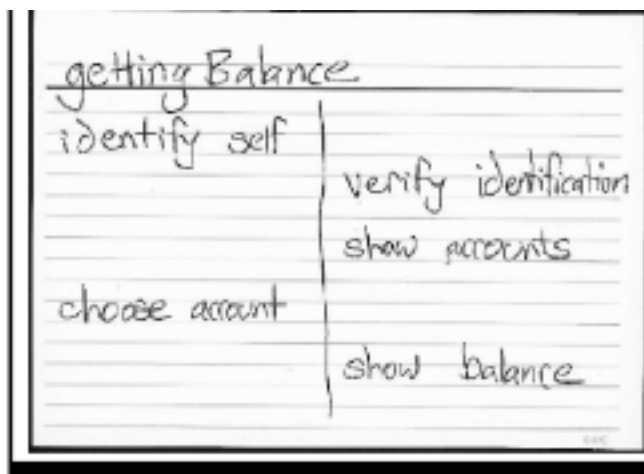


图 3 本质用例卡片样例

与 CRC 技术类似，该技术包括索引卡和角色扮演。经过大家讨论确定用例以后，给每个用例分配一张卡片，在顶部写上用例的名称。将卡片纵向分成两部分，左侧描述用户，右侧描述系统，如图 3 所示。然后，团队中每两人一组来研究人机对话过程，一个扮演用户，一个扮演系统，并记录对话的经过，把结果提交给用例的审查小组进行复审。

本质用例描述系统的人机对话过程，是角色扮演的脚本，可以通过一部分人扮演系统，一部分人扮演用户的方式来模拟系统的人机对话过程。与纯文本的描述不同，这个过程是可视化的，不易引起歧义。Wirfs-Brock 指出，用例可以看做用户和系统的“交谈”过程，有助于系统建模。另外，用例和角色扮演还可以帮助确定系统边界，划清系统内部和外部的界限。

我们在应用领域和学校的开发小组中展开进一步的工作，获得了使用的实际经验。我们发现，抽象的本质用例有很多好处，简短的卡片式描述可以加速分析过程，更多地考虑交互细节，无需早期就在系统交互的具体问题上纠缠不清，被系统的具体实现所困扰而花费大量时间，从而可以集中精力确定系统的本质用例。

本质用例的抽象过程实际上是指导角色扮演的过程，并不是一个真正可实现的对话过程。但是，开发早期的角色扮演实际上是轻量级的，不能判断能否实现。如果需要对抽象元素的具体例子加以明确，那么，把它作为一个可操作的本质用例，可以由此讨论产生一个具体的脚本，描述可能的实现细节。

1. 用例

本质用例的对话并不是指简单的用户动作，而是用户意图。

在角色扮演中，要从角色的视角考虑问题，深入理解其动机和类型，考虑其所处的环境，了解其背景。这样才能准确表现其意图。

从这个意义上说，角色扮演对团队其它成员和听众变得更加重要。对用户意图的充分考虑赋予角色扮演更丰富的内容。更重要的是，复审人员需要更深入详细地考虑用例，评估其一致性，验证是否提取对话中的所有元素。

除此之外，用例着眼于系统与外部世界的交互过程和使用细节。而在大多数系统的开发过程中，系统使用的不确定导致用例实现完全依靠理解能力和创造能力的发挥。本质用例通过对用户意图的理解来确认系统用途，但并不否认用途也可以基于对用户本身的理解进行提取。在这种方式下，用户意图成为用例设计的原动力。

在 RUP 等软件开发过程中，用户界面原型包括对用户本身和用户需求的理解，因此很早就需要进行开发。采用本质用例方法，在理解用户意图的前提下，为用户建立一个清晰的流程设想，同样可以作为系统后期设计的输入。但是，这种方法是轻量级的，不需要产生具体的用户界面，可以支持更快速的开发过程。

2. 系统职责

在本质用例中，采用系统职责代替系统响应进行描述是因为系统职责包括了更多与用户意图相关的内容，并在系统后期设计中产生其它影响。

在角色扮演中，用户角色扮演特定的人，意图可以通过多种方式表达，而系统扮演的未知实体则很难进行发挥。但是必须达到验证、确认人机交互过程的目的。

要深入理解人机对话中的任何一个元素，都需要在目标基础上加以引申。因此，本质用例对用户的描述采用用户意图而不是用户目的。用户意图中除了可以理解的内容外，有一部分我们可以适当地加以引申。

对系统的描述应增加一些针对内部的说明，用以指导下一步的设计。系统职责是描述系统需要做什么，而不是系统实现的细节。这与用户意图的产生动机略有不同，但有助于确定用例和角色扮演。引入用户意图是为了描述系统实现的目标，而引入系统职责是为了描述系统实现的责任。

在本质用例中，某些用户界面与大系统的开发有千丝万缕的联系。在使用本质用例开发用户界面的例子中，系统职责作为人机对话的参与者，主要关心提供给用户的信息内容。但是，在大系统中，系统职责必须包括更广泛的内容。例如，在图 2 所示的银行系统取款的本质用例中，清楚地描述了与用户界面相关的内容，却缺乏与银行财会业务相关内容的描述。

当然，一般用例也有这样的缺点。如果用例只用于描述系统和用户之间的对话，必然难以准确地体现各系统（子系统）之间的联系，比如图 1 的用例也有这样的缺点。

从这个意义上说，着眼于系统职责的本质用例更加实用，因为它并不直接、简单地描述系统与用户的交流过程。在图 4 的本质用例中，我们补充了对系统职责的描述，虽然这些描述不会在具体的对话过程中体现，但是交流过程的完整性和场景的一致性能指导系统的开发。

取款	
用户意图	系统职责
身份识别	验证身份 启动交易服务 提供选择
选择	吐钞 结余
取钞	终止交易服务

图 4 补充后的自动取款系统的本质用例描述

四、本质用例及其设计

面向对象的设计中，“职责”扮演了非常重要的角色，确切地说，它不仅是 CRC 技术的关键概念，同样也是职责驱动设计中的关键概念。

在 CRC 技术中，职责与对象相关联，标识需要解决的问题。在完成职责的过程中，互相协作的对象之间彼此通信，通过不断的优化、迭代，形成一个满意的结构。这样，将职责分配到类，并进行细化，作为一种工具，指导实际系统的开发。

在职责驱动的设计中，职责的内容更广泛更完整。与职责相关联的对象不仅描述了对象维护的知识，还需要描述对象需要完成的动作。在这个意义上，职责强调的是对行为的抽象，而不是具体可能的执行结构的提取。但是，对象可以通过协作对象之间的消息传递完成职责，也可以由此产生更高层次的抽象类。总的来说，在这种设计方法中，职责是在较高层次上进行抽象的分配，而不考虑对象执行的细节。

在 CRC 技术和职责驱动的设计中，关于职责的概念和使用方法看起来非常类似。它们都认为每一个对象都有一组明确的职责，都包括了进行职责分配，指导设计决策的概念。其实，两者之间本来就有一定的继承关系。面向对象设计的基本原理就是：对象是行为和数据的统一体。职责观点沿袭了这一说法，“职责”本身就包括了职责的内容和可以使用的资源。职责还允许进行授权和管理，通过对一系列子职责的调度和管理完成职责，包括了抽象和封装的概念，正如 Wrifs-Brock 等人解释的那样：“职责驱动强调对结构和对象行为的封装，关注类所完成的职责，而到实现阶段再考虑类实现的具体细节。”

本质用例的职责描述系统要做什么，而不是做的具体细节。这与对象封装的观点类似，同样也具有类似的优点。

用例中的职责与设计中的职责一样，都描述行为本身，而不是执行的细节，这就可以将需求和设计连成一个整体来进行描述。

1. 确定系统边界

在实际工作中，需求主要是区分系统做什么不做什么。然而，这个边界非常难以确定，也非常难以与涉及的所有人（包括分析员和投资方等）进行沟通，达成共识。我们发现设计中使用的逼近技术非常适用于这种情况。我们把系统当成一个有明显边界的“黑盒子”，用本质用例的系统职责对系统的行为进行描述。

我们把系统看成由一系列职责组成尚未考虑执行细节的独立系统对象。Jacobson 等人在不同的方向上

提出过类似的观点。采用本质用例、系统职责有助于确定系统边界。如果系统类似一个独立对象，那么用例就采用类似的方法描述，只能从系统的内部访问系统的行为，而交互的过程就类似于一系列能有序管理的参数和返回值。

在 UML 和 RUP 中，通用的用例图可以把这些想法固化下来，如图 5 所示，在图中，我们增加了一个方框，将用户意图和系统职责分离开来，从而明确了系统边界。

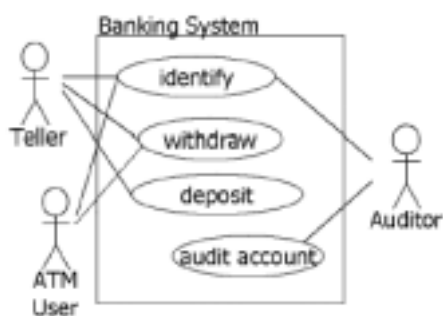


图 5 银行取款系统的用例表示样例

Jacobson 认可这种用法，UML 同样认可这种用法，它指出：“用例可以用一个随意封闭的矩形代表系统的边界或分界”。这种用法与系统对象的提法非常一致，便于讨论和确定系统边界。

2. 用例的职责和设计

对任何系统来说，本质用例的职责与系统对象内部的职责联系紧密。本质用例的职责反映了整个系统的行为，对象的职责协同工作也应该反映相同的行为。

这种方法将系统的需求和设计联系起来。本质用例的职责可以作为系统设计的起点，指导系统的行为设计，保证设计和用例的**可追溯性**。

从一系列本质用例开始设计，引出系统对象，通常通过一系列对象的协作完成职责。另外，在设计中，两者可以通过采用相同语言的方法巩固其一致性。

首先，尽可能采用一致的语言，对一系列描述系统对象职责的本质用例进行设计，然后再确定完成相

同职责的协作类。

在严格的设计过程中，将根据上面的结果产生具体的类。这个过程是一个重新分配职责的过程，将直接影响最终设计方案的确定，要求非常细致。这个阶段的工作属于设计的基本流程，不可省略。需要一定的设计技巧和技术进行职责再分解。现在有一些成熟的技术可以直接使用，例如，在设计早期就可以使用类析取（Extract Class）技术。

但是，我们不提倡这种做法，主要有三个原因：其一，没用利用任何领域模型，使领域模型和设计脱节，造成理解和维护的困难；其二，一个完整系统有很多用例和职责，难以严格地分解；其三，难以兼顾其它地方产生的设计结构。

在 CRC 和职责驱动的设计中，基于基本模型和应用领域模型，产生一组关键对象和类，根据领域知识和设计要求进行初步职责分配，然后着眼于少量的关键用例进行交互和迭代，产生初步设计方案。

在 CRC 和职责驱动的设计中，本质用例没有引起任何流程的改变，但是，它扮演了使用快速原型法进行开发过程中非常有用的角色，为检验设计与需求的一致性提供了有效的方法。

这种方法要从设计初期抓起，从早期用例的职责分配抓起。换句话说，初始的设计职责应该从领域知识中产生。这样，可以与本质用例进行比较，得到早期的反馈，避免以后产生歧义。这样可以更好地指导关键职责设计者的开发工作。

设计一般都会有结果，有许多设计方案可以在新的设计中作为一部分进行继承和重用，例如组件、组件库、框架、设计模式等等，有的甚至本身已经是可执行的，它们对系统设计影响很大。而本质用例则提供最终设计与需求是否一致的有效检验方法。

与 CRC 和职责驱动的设计不同，其它设计与职责本身可以无关，因此，设计和需求的检验显得更加复杂，更加重要，需要对组件和其它结构进行仔细检查。但是，对可执行重用资源的检查并不是对其执行的正确性进行验证，更多的是对其行为模式的认可，也可以说是对其适用性的认可。这样确实对提高重用度具有深远的意义。

用例和设计的职责划分明确以后，问题就迎刃而解了。无论是无意的还是设计活动中的折衷，设计的

不完善性都可能存在。无论采用哪种方式，都需要提升设计。另一方面，设计又可以从某种程度上弥补需求的不足。例如，设计必须基于现有系统的处理活动，哪怕仅是需求中的片言只语，这时就可以重新考察用例，看能否有所改变，能否使用现成的设计资源。

综上所述，引入需求和设计职责的概念提高了两者之间的**可追溯性**，保证需求和设计的一致性。

3. 举例说明

在本节中，我们将通过一个小例子来描述我们的工作。为了说明问题，例子非常简单，但是所用的方法同样适用于更复杂的系统，可以用于顶层设计，也可以用于详细设计，并且也有可用的支撑工具。

这个例子是图书馆系统中的一小部分。领域模型由两个对象类组成：书籍和借阅者。分析的过程如图6所示。在顶层，我们只关注于两个本质用例：借书和还书。

通过对这两个本质用例的进一步分析，我们可以得到迭代1中产生的CRC卡片。在这个步骤中，我们不一定能够分析得到所有的系统对象。例如，要知道可借阅的图书，就必须登记已借阅的图书，必须对作为协作对象的职责进行分配。

系统对象应该明确，能够与用例对应，如果用例太多，产生的对象太大，就会难以执行。从领域模型出发，我们建立了三个类：单独的图书馆系统 Library System 类、书籍 Book 类（每本书一个实例）和借阅者 Borrower 类（每个借阅者一个实例），然后考虑它们的职责分配。

迭代2显示了这个设计过程。在设计中，书籍借阅标志的设定在书籍 Book 类中完成，但是，依然可以通过图书馆系统 Library System 类进行初始化、查询和更新操作。

图书馆系统 Library System 类还有一项职责，就是不但需要“知道有哪些图书”，而且还要能够查询。也就是说，图书馆系统 Library System 类必须对书籍 Book 类进行管理，同时对借阅者 Borrower 类也应有相应的管理功能。

在迭代3中，书籍 Book 类和借阅者 Borrower 类没有改变，增加了一个独立的分类 Catalogue 类，完成“知道有哪些图书”的职责，可以用一个标准的 collection 对象来实现。

需要说明的是，我们在这里基本上只列举在用例中明确描述的职责，实际上，在领域分析的过程中，还会分析产生一些职责，并且对整个过程发生影响。例如，检查书籍是否被借阅的职责就是通过类似领域分析的过程产生的。

如图所示，从需求中提取职责进行设计并不只是一个单向的过程，它可以进行调整和变化。但是，这个过程的描述非常有利于与投资方进行交流，特别是复审阶段的双方沟通。

五、讨论

在本章中，将讨论几个与本质用例相关的面向对象设计主题，包括：

- (1) 无界面系统（实时系统）设计；
- (2) 定制界面系统的设计；
- (3) 迭代和增量开发过程；
- (4) 界面和应用的并行开发；
- (5) 基于业务过程的面向对象设计。

1. 无界面系统（实时系统）设计

在实时系统和软件引擎中，系统只与系统主角交互，并且没有传统的人机界面，那么，能否采用本质用例呢？答案是肯定的。本质用例同样适用于这类系统。

ESSENTIAL USE CASES

borrowingBook

User Intention	System Responsibility
identify self	verify borrower id
identify book	verify that book may be borrowed record book as issued to borrower

returningBook

User Intention	System Responsibility
identify self	verify borrower id
identify book	record book as returned by borrower

CRC CARDS ITERATION 1

System Object	
verify borrower id	
verify that book may be borrowed	
record book as issued to borrower	
record book as returned by borrower	

ITERATION 2

Library System	
verify borrower id issue book to borrower know all borrowers know all books	Borrower Book

Borrower	

Book	
verify that I may be borrowed issue myself to borrower return myself	

ITERATION 3

Library System	
verify borrower id issue book to borrower know all borrowers	Borrower Book Catalogue

Catalogue	
know all books	Book

图 6 基于本质用例的设计样例

在这类系统中，有一个重要概念就是“外部世界（outside world）”。在任何系统中，都需要对系统的内部和外部进行明确的划分，从而确定系统的边界；确定系统交互的原则（无论是人还是机器）；设计交互方式。我们发现，在这样的系统中，本质用例同样有许多优点。在适当的抽象层次上，本质用例摆脱了表达和实现的具体细节的束缚，能够更准确的捕捉用户意图和系统职责的本质，它不受具体系统类型限制，简洁明了，易于理解，便于编辑，方便复审，可以实现分析和设计的无缝连接和平滑过渡。

系统常常有一些附加的非功能性约束，例如响应时间、交换速率、信息容量、可靠性等。因为用例着眼于交互的过程，所以不是发现和确定这类需求的理想工具。但是，它可以帮助记录和管理这类需求。在常规用例方法中，通常采用文字描述的方法，并与用例同步管理，且其管理过程与用例结构无关。本质用例也可以采用类似的方法。在以用为中心的设计中，这类指标往往与本质用例相关，因为它们对高性能的用户界面设计至关重要。

本质用例的系统职责、系统对象的职责以及中间对象的职责之间的可追溯性弥补了系统设计的不足，保证系统满足这类非功能性指标的要求。例如，用例级的系统约束可以分解到对象中，对象的性能又直接影响到用例职责。采用公共任务标签的办法可以调整最终设计，确保满足非功能性指标要求。

2. 定制界面系统

本质用例同样适用于已经定义好界面的情形。在这种情况下，我们先对界面进行分析，然后描述交互过程的本质用例，并在此基础上进行面向对象的设计。这样，从系统交互和职责的本质着眼进行设计的中间系统更稳定，更能够适应界面细节的变化，同样具有本质用例方法的主要优点。

3. 设计业务流程

长期以来，常规用例被认为是业务流程建模的好方法，它主要描述的是业务单元（人）之间的交互，而不是计算机系统和人之间的交互过程。

本质用例与技术无关，它描述用例之中抽象的意图和任务，因此，无论采用何种界面技术，都可以实

现用例。例如，采用相同的一个呼叫中心的用例，既可以支持用桌面应用的方法实现，也可以支持用 Web 的方式实现。还可以支持采用交互的语音提示的方法实现。这就是说，本质用例与采用何种技术途径无关，需要的仅仅是最基本的计算支持。

使用本质用例的成功经验告诉我们：本质用例既适用于软件设计，也适用于业务流程设计，它们在抽象、对话和任务中具有相同的优点。

4. 开发过程

本节主要描述我们建立的软件设计模型，并为模型构建的过程提供参考。非常重要的一点是，从本质用例到系统对象，然后到面向对象的中间设计描述是开发的抽象过程，而不是构建模型的具体步骤，并不是类似于瀑布的过程，先对本质用例进行“分析”，接着确认其“完整性”和“正确性”，然后进行设计等等。

我们希望这个过程是一个增量的、可迭代的过程：先有一个备选用例表和一个初步的领域模型；其中的关键用例首先被细化，按用户意图和系统任务编写到用例卡片中；然后进行初步的设计，划分对象任务，并对用例的措词进一步细化和调整。再细化其它的用例，获得更多的对象模型，这些工作可根据后续工作中情况进行必要的复查和调整。

本质用例有助于软件开发的迭代过程，其抽象层次、用例与模型间的通用词汇决定了迭代开发的过程更易于跟踪，具有更好的一致性，对象和用例之间具有更良好的可追溯性。当设计完成时，所有的用例和对象都使用相同的词汇表，所有的系统任务被设计划分成适当的中间对象，所有的用例都可以被对象模型执行，能够更方便地检验设计和系统任务的一致性。

5. 并行开发

在以用为中心的设计中，使用本质用例同样可以产生用户界面。在这种情况下，基于本质用例采用面向对象设计方法进行中间设计具有更加显著的优点。它可以避免重复工作：它设计的用户界面可以直接在软件设计中重用，无须重新捕捉和进一步细化，使软件设计和界面设计可以同步进行，只要二者同时支持本质用例就可以保证良好的一致性。

这和其它的过程方法（特别是 RUP 过程）略有不同，不需要在早期就完成界面设计，可以在软件设

计阶段进行，支持软件设计和界面设计的同步进行，不会因为界面设计没有完成而延宕工期，可以有效地缩短项目周期，特别是在用户界面设计要求高、数量多的项目中，可以为整个项目节约好几个月的时间。

并行设计特别适合于 Web 系统的开发，前端界面和后台服务软件可以独立设计，并有专门的技术支撑其实现。在这样的并行开发过程中，要求对迭代过程进行有效地管理，协调设计过程中用例的变更和实现，使二者保持良好的一致性和可追溯性。

六、与其它方法的比较

软件开发技术在不断地向前发展，也提出了许多其它的手段和方法。在此，我们着重介绍它们处理需求和设计之间关系的不同方法。

面向对象的软件工程（OOSE）对建立对象模型中对象和用例之间的关系提出了许多有用的建议。Jacobson 等人用系统边界、控制和实体对象实现用例的方法来建立分析模型，但是在分配系统行为和产生类（class）的过程，要求有一个附加的翻译过程。与这种方法相比，我们使用的本质用例方法具有更好的可操作性。

还有一些更简单和直接的方法来组织需求和进行面向对象设计。例如，在第二章中提到，Jacobson 等人介绍用例概念，以及 Lockwood 提出在用例界面设计中使用本质用例方法就是基于这样的思想。还有许多指导用例写作及其在软件开发中具体应用的书籍。例如，Cockburn 最近就出版了一本详细介绍用例写作方法的书籍，并对各种风格的用例进行了介绍。Amour 和 Miller 还讨论作了为需求搜集工作的一部分，如何发现和管理用例。但是，这些工作很少提及用例如何在分析和设计阶段的工作中反映，讨论用例和设计关系的工作在领域模型的开发中完成，把领域模型作为设计和开发的第一步。领域模型固然非常重要，但是，从需求到发现领域模型、领域模型满足需求之间还应该大量的工作需要完成。

大量的文献都提到可追溯性的重要性。Pfleeger 指出，可追溯性包括横向和纵向的可追溯性。纵向的可追溯性是指开发过程中模型内部的关系，横向的可追溯性是指模型与模型之间的关系。加强系统的可追溯性可以有效地提高开发过程的质量和降低维护成本。

我们致力于需求与设计横向可追溯性的工作，将二者的关系直接反映到任务的分配中，成为设计工作

的一部分。

任何软件开发方法都伴随相应的过程。最近经常提到的过程是 Rational (瑞理公司) 统一过程, 即 RUP。RUP 是用例驱动的软件开发过程, 用例连接开发过程的各个阶段。在我们使用的本质用例方法中, 本质用例在开发过程中具有同等重要的地位, 只是在用例格式的选择上有所差异。事实上, 本质用例方法与 RUP 方法中用例的描述没有任何矛盾, 甚至可以认为本质用例是基于前文中指出的 OOSE 方法中分析和设计工作流的细化。

还有一种开放式的软件开发过程, 准确地说, 它是一种软件开发方法的框架, 而我们只关注其过程部分。特别值得关注的是它的开放式工具技术, 为不同职责的软件开发需求提供完整的技术支持。它提供从原始需求到设计模型的技术, 包括: 协作分析技术、CRC 卡片建模技术、委托分析技术、领域分析技术、泛化和继承技术以及对象模型转换技术等。上述所有技术都可以和我们的研究联系起来, 而我们的工作为指导操作和提供可追溯性方面做出了贡献。

最近, 轻量级方法 XP 等大行其道。基于 XP 方法的设计要求功能需求尽量简化, 在真正需要以前, 不轻易增加别的功能。XP 是带有独立“用户故事”的增量开发过程 (与用例类似但完全固化), 通过再分解现存的程序实现用例, 达到增量的目的。我们的方法与 XP 的方法也有所类似, 所有的设计都可以再生。但在 XP 的方法中, 当需要扩展支持新的用例时就需要重新设计, 而在我们的方法中, 设计从完整的系统目标出发, 以实现整个系统为目的, 所有的再分解都维持了对象的任务, 都可以在中间组件之间重新进行分配。

结论

用例适用于各种软件的开发, 本质用例起源于用户界面设计的细化和描述。我们尝试在面向对象的系统开发中使用本质用例, 并在本文中进行了总结。

本质用例能够简单快速地发现需求, 支持通过角色扮演进行沟通, 有助于发现系统边界。它简明扼要, 易于学习, 避开了繁琐的实现细节, 支持快速开发。使用这种方法易于确定用例模式。它还同时支持界面和系统设计, 可以实现二者的并行开发。

本质用例标识系统任务，将需求和面向对象的设计联系起来。在设计中，其职责后发分配协作对象之间的抽象行为。使用本质用例描述需求，采用职责驱动的方法进行设计，可以更好地为设计提供指导，使设计和需求具有良好的可追溯性。

参考文献（见下页）

A version of this paper in English appeared in the Proceedings of the 25th Australasian Computer Science Conference 2002, Melbourne, Australia, Conferences in Research and Practice in Information Technology, Vol. 4. Michael Oudshoorn, Ed., January 2002

UMLChina 保留中译本一切权利



欢迎访问

www.umlchina.com

钱五哥



钱岭（钱五哥），计算机科学博士，专业方向：软件工程。现为 Bell Lab Research China 高级研究员。1995 年起开始参与各种软件开发项目，后来逐渐转入组织/项目过程管理的研究。

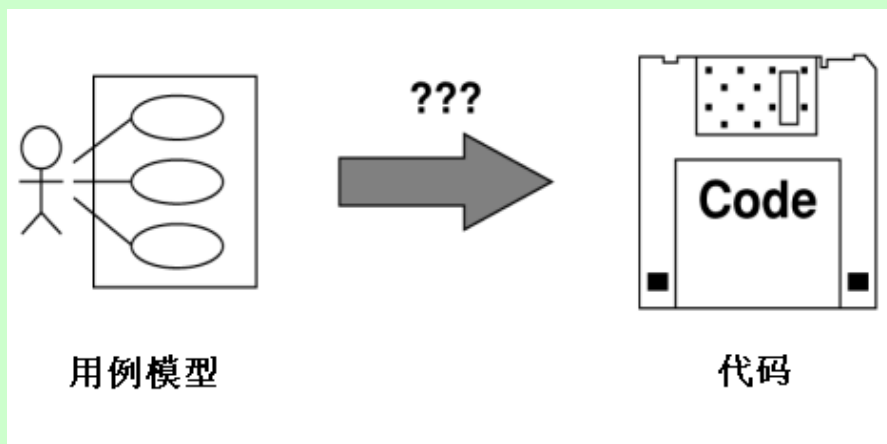
钱五哥

References

- [1] Frank Armour and Granville Miller. *Advanced Use Case Modeling: Software Systems, Volume 1*. Addison-Wesley, 2001.
- [2] Kent Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77, October 1999.
- [3] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *Proc. of OOPSLA-89: ACM Conference on Object-Oriented Programming Systems Languages and Applications*, pages 1–6, 1989.
- [4] David Bellin and Susan Suchman Simone. *The CRC Card Book*. Addison-Wesley, 1997.
- [5] Robert Biddle, James Noble, and Ewan Tempero. Patterns for essential use cases. In *Proceedings of KoalaPLoP 2001*, 2001. Also available as CS-TR-01-2 at <http://www.comp.vuw.ac.nz/Publications/>.
- [6] Robert Biddle, James Noble, and Ewan Tempero. Sokoban: A System Object case study. Technical Report CS-TR-01/11, Victoria University of Wellington, 2001.
- [7] Robert Biddle, James Noble, and Ewan Tempero. Supporting reusable use cases. Technical Report CS-TR-01/10, Victoria University of Wellington, 2001.
- [8] Alistair Cockburn. *Writing effective use cases*. Addison-Wesley, 2001.
- [9] Larry L. Constantine and Lucy A. D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage Centered Design*. Addison-Wesley, 1999.
- [10] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [11] Ian Graham. Task scripts, use cases and scenarios in object oriented analysis. *Object Oriented Systems*, 3, 1996.
- [12] Brian Henderson-Sellers, Anthony Simons, and Houman Younessi. *The OPEN toolbox of techniques*. Addison-Wesley, 1998.
- [13] Ivar Jacobson. *The Object Advantage : Business Process Reengineering With Object Technology*. Addison-Wesley, 1995.
- [14] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
- [15] Ivar Jacobson, Mahnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.
- [16] Dean Leffingwell and Don Widrig. *Managing Software Requirements: A Unified Approach*. Addison-Wesley, 2000.
- [17] Mikael Lindvall and Kristian Sandahl. Practical implications of traceability. *Software—Practice and Experience*, 26(10):1161–1180, October 1996.
- [18] Pete McBreen. When are use cases done? OOTips web site at <http://ootips.org/use-cases-done.html>, January 1998.
- [19] Object Management Group. *Unified Modeling Language (UML) 1.3 specification*. Object Management Group, 2000.
- [20] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, 1998.
- [21] Doug Rosenberg and Kendall Scott. *Use case driven object modeling with UML: A practical approach*. Addison-Wesley, 1999.
- [22] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
- [23] Nancy Wilkinson. *Using CRC Cards - An Informal Approach to OO Development*. Cambridge University Press, 1996.
- [24] Rebecca Wirfs-Brock and Brian Wilkerson. Object-oriented design: A responsibility-driven approach. In Norman Meyrowitz, editor, *Proc. of OOPSLA-89: ACM Conference on Object-Oriented Programming Systems Languages and Applications*, pages 71–75, 1989.
- [25] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object Oriented Software*. Prentice Hall, 1990.
- [26] Rebecca J. Wirfs-Brock. Designing scenarios: Making the case for a use case framework. *The Smalltalk Report*, 3(3), 1993.
- [27] Rebecca J. Wirfs-Brock. The art of meaningful conversations. *The Smalltalk Report*, 4(5), 1994.

UMLChina 培训

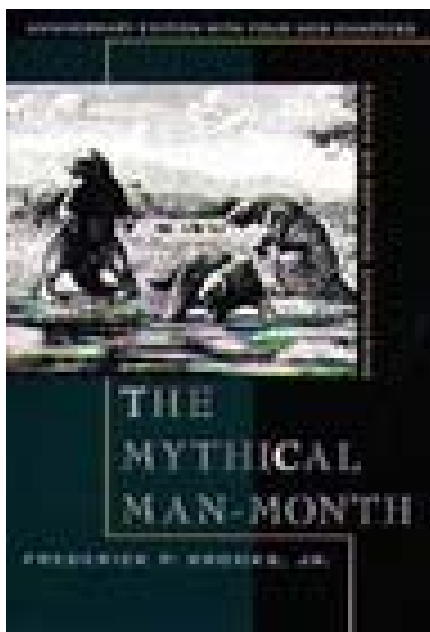
The real thing



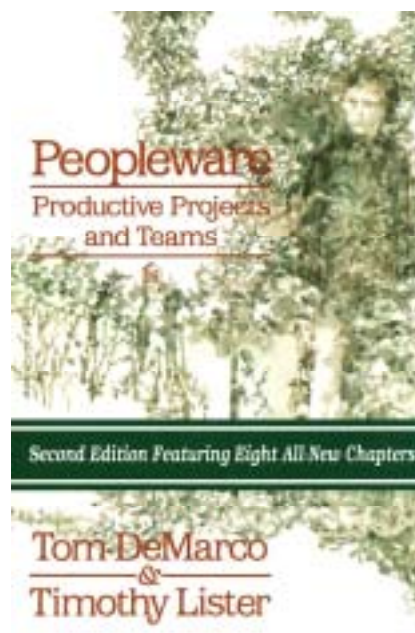
利用 UML 的 20%就可以为 80%的问题建模
-- 《UML 用户指南》，第 32 章

详情请垂询：think@umlchina.com

中文译本即将发行！



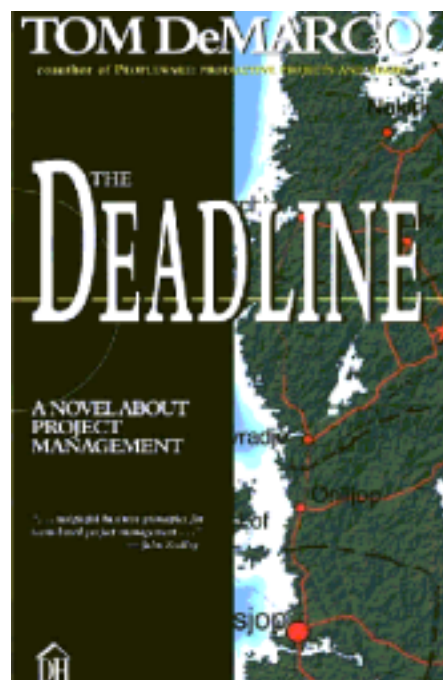
《人月神话》20 周年纪念版，Fred Brooks
翻译：UMLChina 翻译组 Adams Wang
散文笔法，绝无说教，大量经验融入其中



《人件》第 2 版，Tom Demarco 和 Tim Lister
翻译：UMLChina 翻译组方春旭、叶向群
微软的经理们很可能都读过—amazon.com



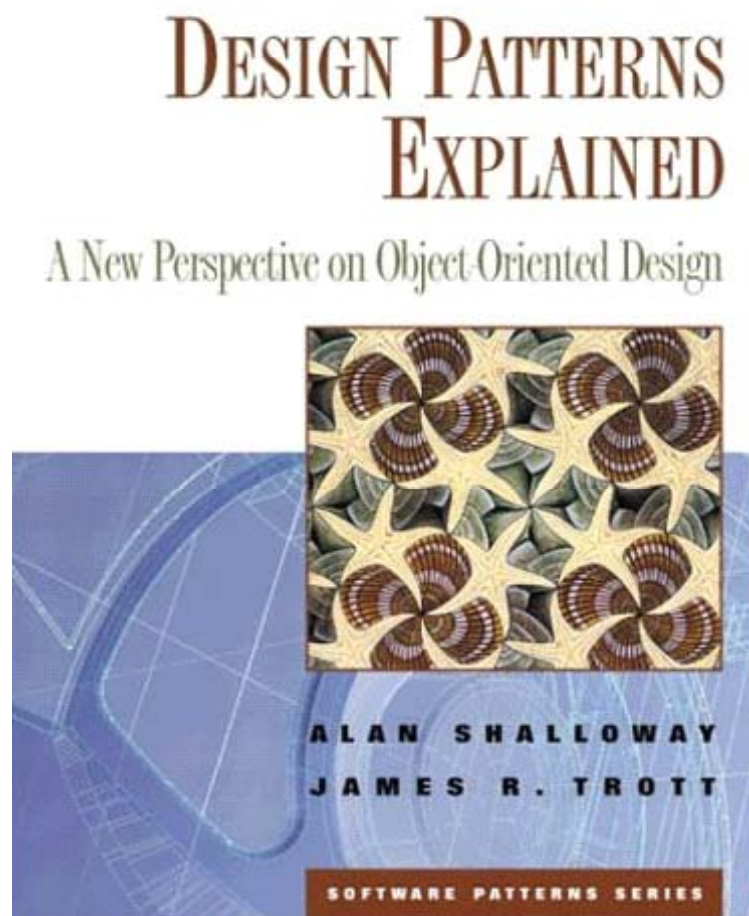
《面向对象项目求生法则》，Alistair Cockburn
翻译：UMLChina 翻译组乐林峰
Cockburn 一向通俗，本书包括十几个项目的案例



《最后期限》，Tom Demarco
翻译：UMLChina 翻译组透明
这本最通俗，这是一本软件开发小说

看不懂《设计模式》？

它的作者推荐《设计模式精解》----中译本即将上市！透明 译



直接学习《设计模式》这本书是非常困难的。作为入门，我推荐 Alan Shalloway 的《设计模式精解》。

--John Vlissides 写于 [UMLChina 答疑板](#)

利用角色扮演和用例卡片进行需求复审

Robert Biddle 著, [caorui](#) 译

摘要

本文提出了一种利用角色扮演和索引卡片进行需求复审的方法，该方法能够更好地指导用例开发过程。这种方法建立在已经成功应用的面向对象设计技术——CRC 卡片技术之上。它使用索引卡片记录本质用例，同时利用角色扮演的方法对用例进行开发和复审。本文详细说明了这种方法，同时描述了这种方法的应用过程。

关键字

FA11 面向对象观点，FB0301 信息需求判定活动，FC28 面向对象分析

简介

我们已经探索出如何完善在面向对象开发方法的前期阶段使用技巧的方法。特别地，我们对如何更好地确定那些捕获需求并驱动设计用例进行了研究。为了确认用例集合是否已经捕获需求，需要对它们进行复审，因此我们关注如何有效对用例进行复审。

用例描述了用户与系统的一系列交互。用例的概念已经在开发过程中得到普遍接受，在软件开发中寻找并使用用例已经是很寻常的事情。但正如面向对象分析方法的其它方面一样，要发挥用例的潜能，需要理解和经验。用例复审过程可以加深新手对用例的理解并增加他们的经验。

据我们所知，还没有专用于用例复审的技巧。我们寻求这样一种方法，它能保证团队成员的积极参与，提供一些操作性较强的指导，同时这种方法应该对于初学者和系统利益相关者开放。另外，这种技术应该是轻量级、完备而灵活的，以保证团队成员能够洞悉目标系统的行为，同时对系统有一致的理解。本文提出一种复审用例的方法，我们认为满足以上各点要求。该方法基于已经获得成功应用的 CRC 卡片技术，同时还使用了索引卡片和角色扮演。

本文是按如下方式组织的：下节提供一些背景知识，讨论面向对象开发中的用例及其作用；同时对我们应改编用于用例的 CRC 卡片设计技术做了一定的回顾。随后几节对方法本身进行描述，总结经验并解释本质用例的应用及推断。最后一节给出了结论。

背景

用例

Ivar Jacobson 等（1992）将用例定义为“与系统对话行为相关的一系列事务”。其基本思想是以用例来表达系统与外部世界间的一系列交互——甚至系统还没有构建出来。这种思想具有不可忽视的作用，有几个原因：

在前期开发阶段，用例可以有助于集中于交互，并借此引出需求或预期系统行为，从而捕捉需求并帮助确定规范。这种技术是十分有效的，因为可以按照容易唤起回忆或想象的形式描述交互，如同叙述或对话一样。

在其后的开发阶段，用例仍然很有帮助，因为它侧重于交互。在这个阶段交互就是系统必须满足的功能规范的具体体现。在设计和实现阶段，必须确认并建立一种能够满足这些规范的结构。在复审和测试阶段，用例可以驱动系统测试行为。

用例以一系列交互为基础，而系统需要的交互通常遵循一致的结构——即在一定程度上面向同一个目标或子目标。这就允许对需求规范分类，这是十分有益的。这种分类对于开发过程中的整体管理有很大帮助，因为用例可以通过选择、分组、过滤、优先级排队等方法来重新组织。

CRC 卡片

在寻找更加方便应用用例的方法时，我们从另一种技术中得到了灵感，在面向对象开发过程的稍后阶段需要这种技术。这种技术就是 CRC（类—职责—协作），它使用卡片和角色扮演的方法，帮助人们将系统设计为一系列协作的对象（Beck 和 Cunningham 1989，Bellin 和 Suchman Simone 1997，Wilkinson 1996）。以索引卡片代表每个类，设计组的成员扮演类的角色，以话语的形式模拟系统的行为。卡片上记录了类的职责和协作者。职责是一种概念的抽象，是类应该知道或做的事情。职责概念被当作一种直观推断试探法来分配系统中的智能。

最初，写作 CRC 卡片只是用于学习设计技术的方法，卡片使得对象的思想更加具体，角色扮演则促进了对对象协作的认识。但现在 CRC 卡片已经是得到广泛承认的技术，而不是仅仅面向初学者。Bellin 和 Suchman Simone（1997）认为 CRC 卡片技术是一个“元认知”过程，该技术具有可操作性强的本性，这有助于在设计阶段思考一些关键问题：

“团队中的每个人承担了一个类的职责，以 CRC 卡片为脚本，扮演设计的系统。这一方法的价值在于，扮演类并指出类应该做什么这一过程能够同灵机一动一样，触发相同的响应。把玩这些卡片能够引发难以预料的洞察力。角色扮演能够如此成功是因为它促进了团队成员积极的参与。”

用例卡片与角色扮演

我们已经使用过一段时间的 CRC 卡片技术，并十分赞赏其效果。在寻求改善用例方法的途径时，我们决定采取类似的方法。其基本想法是使用用例的索引卡片，同时引入角色扮演。

用例描述了与系统的交互，但不止一种方法可以描述这些交互。我们选择用户与系统之间对话的形式。因为对话形式的使用例描述能够构成一部剧本，我们希望这样能够有助于角色扮演。剧本有两个角色，用户和系统。这样用例角色扮演可以只有两个人参与，每人负责剧本的一部分。

我们决定，每张用例卡片只代表单个用例，卡片上要记录用例的名称和对话脚本步骤。为了清楚区别两个角色，用居中的一条竖线将卡片分为两个部分，左侧写用户脚本，右侧写系统脚本，如图 1 所示。这种布局正是 Wirfs-Brock（1993）采用的两栏式格式。这种格式对于功能性需求很适用，因为可以很清楚看出如何使用系统以及系统又是如何运作的。

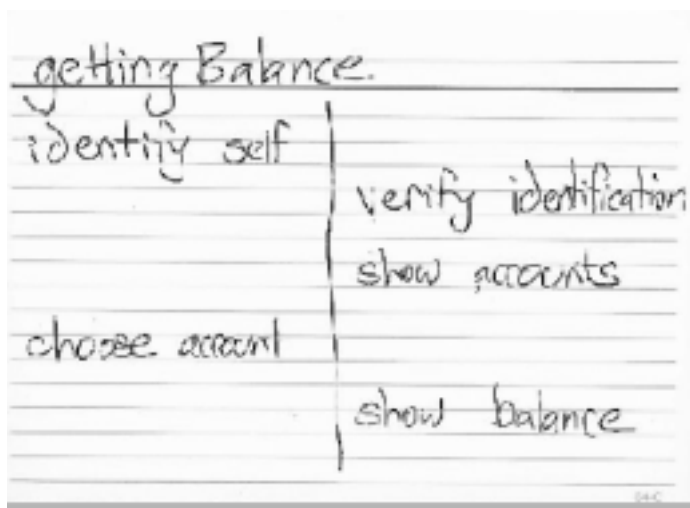


图 1：描述银行系统本质用例的卡片

通常，用例对于陈述非功能性需求没有什么帮助。然而，有时非功能性需求确实与特定的用例相关，这时可以在用例卡片的底部或者背面写下简短说明。

使用用例卡片及角色扮演的方式主要有助于详尽阐述用例的“主体”，确定交互的步骤。在此之前，首先应该识别用例。关于用户与系统交互的各种不同方式的信息等初期工作需要通过背景分析得到。在大型系统开发中，即使是用例识别的工作也可能需要付出大量的代价而且影响范围广泛。我们用各种分析技术识别大量的备选用例，确定它们的优先级，然后选择一些“重点”用例，这些用例代表了最重要的交互，即使是在早期原型中它们也是必需的。

然后，开始在这些重点用例上工作，首先利用卡片和角色扮演详细阐述它们的步骤。如 CRC 卡片方法一样，我们一般推荐 3 到 6 个人的小组来做这些工作。卡片上工作的第一步是写下重点、用例的简短名称。例如，银行系统的重点用例可能名为：存款、检查帐户余额和取款。

下面选择一张卡片并写出对话。选出扮演用户和系统的人，使用探讨的方式进行排练。团队需要集体工作以决定对话的步骤。当某一想法听起来很合理时，演员按脚本将之“表演”出来。团队的其它成员则审查表演，之后再讨论，并在必要时进行改进。文档投影仪的应用越来越广泛，这样卡片可以很容易的投影到大屏幕上，供更大的团队查看。以上过程反复进行，直到整个团队认为对话已经能够描述用户与系统交互的方式。有时可以先放下某一用例而去审核另一用例，晚些时候再回头改善前一个用例。

角色扮演的作用之一就是探索迭代的方式进行用例开发。我们同样利用角色扮演来进行用例复审。这种复审可以由合作开发者、项目其它方面的工作人员、系统利益相关者或系统领域专家来进行。

在利用索引卡片和角色扮演确定了用例的主体后，其详细信息可以记录在需求文档或者 CASE 工具的数据库中。用例可以用于驱动系统设计和复审，在晚些时候用于系统测试和演示。在一些后期活动中，卡片和角色扮演可能还有用途，例如对于某一交互的替代方案进行快速的审查和探讨，卡片和角色扮演直接和轻量的自然本性有很大帮助。

经验

到目前为止，我们已经观察过许多团队使用用例卡片和角色扮演技术。我们在与超过 20 家企业团队合作中使用过该技术，典型情况下，这些团队都包括一些非技术人员，如业务分析人员或基层经理。在教学和指导中我们也介绍了这些方法，特别是在大学的面向对象开发与软件工程课程中。这些经验是鼓舞人心的，在工业领域和教育领域，所有的参与人员都已经熟悉了面向对象系统和 UML 表示的一些基本概念。

我们寻求团队成员的主动和直接参与，结果很令人满意。与 CRC 卡片相同，索引卡片的具体性和角色扮演的行为性结合起来，吸引了团队成员的注意力，同时也要求积极参与。仅仅这一点就有足够的价值，因为它确保在确定需求时整个团队都参与进来了。

与 CRC 卡片一样，索引卡片在另一方面也有所帮助，这与卡片的本性有关：它们既可以分散又可以结为一体，同时只需要很少的投资。卡片可以在桌面上排列、分级，需要的卡片可以抽取出来，在角色扮演时又可以拿着。一张卡片可以很容易被改动或抛弃而不会影响其它卡片，同时可以快速地进行改变。

与卡片相比，角色扮演的优势影响更加深远。团队成员，甚至非开发人员，都能够很好的理解对话。另外，他们很熟悉表演道具，同时乐于怀疑戏剧依赖的基础。人们通常都能够观看角色扮演并在真正理解的基础上想象完成的系统的交互。这是快速原型法的一种形式，它允许快速复审和反馈，允许迭代改良。

Bellin and Suchman Simone (1997) 指出，CRC 的角色扮演方法可以引出出人预料的洞察力，对于用例角色扮演也是这样的。在 CRC 的角色扮演中，这种洞察力常常表现于如何在几个协作的对象间分配智能。在用例的角色扮演中，这种洞察则与如何更好在系统边界进行通信有关。

卡片与角色扮演本身并不总是足够确定用例是否合理。开发者应该清楚，有很多相关因素影响用例。特别地，开发者应该研究用例对功能性需求的建模 (Armour 和 Miller 2001, Rosenberg 和 Scott 1999)，

以及用例得以高效的种种因素（Cockburn 2001）。Wirfs-Brock（1994）讨论了用例怎样很好地模拟“有意义的人机对话”，这与我们的技术有关。角色扮演使人机对话更加生活化，从而易于评价它是否真正有意义。

但是我们必须承认，系统还需要满足非功能性需求（如响应时间、事务处理速度、信息容量、可靠性等）。由于用例侧重于交互，所以在发现非功能性需求时并不理想。这意味着非功能性需求必须使用其他方法来进行确定。但无论如何，用例构造了一个很好的需求复审起点，并且通常会引起对非功能性需求的讨论。

根据经验，确定需求时的一个关键问题就是决定系统的边界，将应该由系统完成的事与不应该由系统完成的事分开。因为牵涉到系统的人包括分析人员和各种利益相关人员，对于边界的区分工作达成一致可能很困难。

在决定系统边界时，用例卡片和角色扮演方法被证明是非常有效的。我们使用一种与设计阶段相似的手段，就是将系统看成“黑箱”。并不说明内部如何工作，但如何使用系统由用例指定。每张用例卡片上的两栏式对话表清晰地显示了用户职责和系统职责，划分两个角色的线就是系统的边界。

在确定系统边界时，用例角色扮演有多方面的帮助作用。在前期的探索式表演中，很快可以看出团队成员对系统应该如何工作持有各种不同意见。人们阅读的是同一份分析文档，但却提出不同的解释，特别常见的是设计人员与业务分析人员或领域专家出现不同理解。及早发现和解决这种差异是很重要的，在确定角色扮演的实际交互序列时比较容易暴露这些差异。

探索式角色扮演同样也能暴露一些对于用户或系统角色的不合理假设。对于用户来说，如果用例指定的动作与可能的实际用户行为不一致，则很容易发现。对于系统来说，如果无法获得重要的信息，则指定的行为无法完成。图3显示了在角色扮演中能够发现异常现象。实际的表演看来比仔细阅读更容易暴露出这些异常问题。

第一幕

用户：

我指出我需要做的工作，系统显示该工作的细节。

系统：

喂！说明系统做什么是我的事情。

停！——这经常只是演员的错误，但也能揭示关于系统边界的迷惑。注意负责表演系统的人如何“保护自己的势力范围”

第二幕

用户：

我指定我要做的工作。

系统：

我显示这项工作的细节，并说明是否有座位。

观众：

等一下，还没有指定座位呢。

停！——这说明在用例中有遗失的信息。在此例中，观众保证演员诚实的表演。

第三幕：

用户：

我说明我要做的工作。

系统：

我显示这项工作的细节。

用户：

我指定我要哪一个座位。除非我不想这样做……，如果我这样做，但没有座位，我就得不断的选择座位，直到找到。要是系统能告诉我哪些座位可用就好了。

停！——在这个例子中，用户已经深入到角色中去了，并且发现了当前版本用例的不足。

第四幕：

用户：

我说明我要做的工作。

系统：

我显示这项工作的细节，并列出的可用座位。

用户：

我选择我想要的座位。

系统：

我将这些座位标记为不可用，并将票打印出来，并且……嗯，也许我应该先算一算需要多少钱并收了钱后再打印这些票？

停！

第五幕：如此等等...

图 3：剧院订票系统用例的角色扮演对话示例，描述角色扮演如何在早期发现用例中的难点

要解决这些异常，可能需要修改用例的步骤，也可能需要以不同的方式来组织用例。这些变化都需要改动其它用例，甚至可能要新建用例。但重要的一点是，可以在早期发现并修正问题，而不是在后期开发才发现从而付出更大的代价。

角色扮演方法也使开发队伍之外的人易于理解用例交互。在给那些事务繁忙的利益相关者演示用例时尤其有用。角色扮演方法快捷、现场感强、易于为人接受。此方法不需要太多投资，能对问题进行讨论，对比几种不同的方案，并可能发现存在的缺陷。

在寻求对用例角色扮演和系统边界确认工作的支持时，我们发现用例图有很大帮助。我们使用的是UML（Rumbaugh 等 1998, OMG, 2000）用例图，它以线条小人表示角色（在UML中表示用户），用椭圆表示用例。我们同样明确地标出系统边界，表示为用例周围的方框，角色与用例之间的线穿越边界，如图4所示。在用例图中表示出系统边界的约定由Jacobson等人（1992）倡导，虽然它是UML用例图的一个可选部分，但它并不是UML的典型特性。这样描述系统边界的好处在于将系统可视化地表示为一个单元，边界线与用例卡片上划分用户与系统的线是一致的。

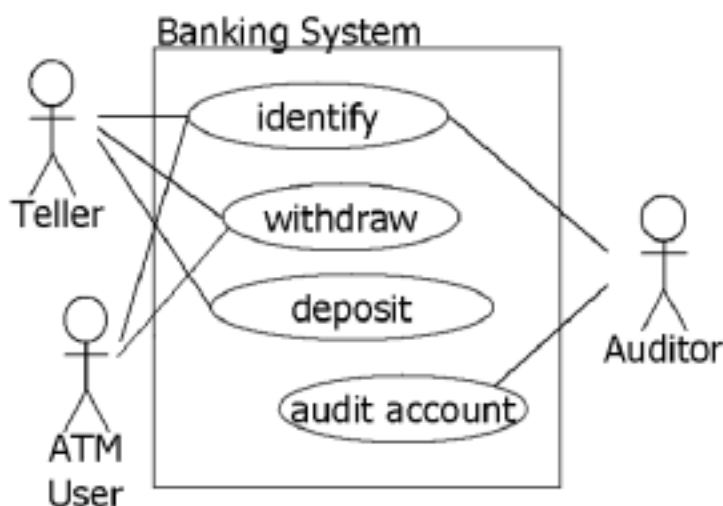


图4：明确表示出系统边界的用例图

本质用例

尽管卡片与角色扮演可以涵盖任何用例，事实上，我们主要将它们应用在精选过的用例上，这些用例称为本质用例。Constantine 和 Lockwood（1997）发展了这种精选方法，它是“以使用为中心设计”（Usage-Centered Design）的一部分，是用于开发用户界面的过程。他们观察到一个重要现象：

“特别地，常规用例一般包含太多针对尚需设计的用户界面的假设，这些假设多数是隐藏或者暗示的。”

Constantine 和 Lockwood 将本质用例定义为“用例的简化和泛化形式”。他们使用“本质”这一词，因为这些用例：“希望通过非技术性的、理想化和抽象化的描述来捕获问题的本质”

本质用例的抽象与用例没有直接关联，而是更多地与用例的步骤相关。从这一点上讲，一个本质用例确实说明了一系列交互，但只是一些抽象步骤的序列。例如，图 5 显示了银行系统的一个普通用例，该用例详细指定了对话中的具体步骤，例如“插入卡片并读磁条”。图 6 则是一个与之等价的本质用例，它以一个更抽象的“确认自己的身份”开始。再有，需要注意的是，普通用例对各角色的标识是“用户行为”和“系统响应”，这无疑强调了实际的具体交互。而本质用例则在用户部分使用“意图”，系统部分使用“职责”，这样更加抽象些，但内容更丰富。

取款	
用户动作	系统响应
插入卡	读取磁卡 提示输入 PIN
输入 PIN	校验 PIN 显示交易菜单
按键	显示总额菜单
按键	提示总额
输入总额	显示总额
按键	退卡
取卡	吐钞
取钞	

图 5：从自动柜员机取现金的普通用例（引自 Constantine 和 Lockwood.）

取款	
用户意图	系统职责
身份识别	验证身份 提供选择
选择	吐钞
取钞	

图 6 从自动柜员机取现金的本质用例（引自 Constantine 和 Lockwood）

在卡片上使用这种形式的用例有很多原因。抽象保证了本质用例的简洁性，这样一张卡片就可以容纳用例描述。同时，这种抽象可以避免对于无关的实现细节的不必要争论，从而加快进度。关注于“意图”和“职责”的方法具有重大意义。

我们的经验证实本质用例确实有简明的优点，同时还具有更深远的影响。正如我们希望的，抽象使得用例适于在索引卡片上使用，同时也避免过早陷入实现细节的讨论。

从更深层次上讲，将用户角色表述为“意图”，而系统角色表述为“职责”的做法具有启发的优点。二者的结合形成的用例对话能够更好的反映用户的真实动机，更好的捕捉系统需求，同时避免不成熟的设计决策。

我们发现使用本质用例需要做一些调整。调整之一，用例角色扮演本身应该是抽象的，以避免过于具体而与特定的接口细节相关。按照我们的经验，这并不是问题，尽管发现团队有时候考虑用例的一些具体条例有助于检查他们的理解。

本质用例起源于用户界面设计。通常的情况下，系统职责仅限于与直接交互相关的职责。暂时并不分析系统行为更深层的职责语义。然而我们发现，在有利的时候可以将这一部分较深层的职责包含进来。例如，图 5 中的用例可能包含维护账户余额一致性和审计的职责。

关注对话中系统部分，如记录其职责，对于系统开发具有更深远的意义。确定用例之后，面向对象开发过程后一步是将系统设计为一系列协作的类和对象。关注于职责正是确定这些关系的主要方法之一，这也是 CRC 卡片的主要用途，它在职务驱动的设计方法中得到了更加细致的应用（Wirfs-Brock 和 Wilkerson 1989, Wirfs-Brock 等，1990）。

本质用例将系统的角色记录为职责的集合。这意味着，当所有的用例都已经确定时，我们也就有了整个系统的职责集合。我们可以将系统看作单个对象，而职责则是这个对象的规范。我们可以设计一系列协作的对象，如同分解系统对象，并将系统职责恰当地分配给系统各部分一样。

CRC 之类的职责驱动设计技术已经阐述了如何分配职责。但是这些技术都是从领域分析得到对象和类关联的职责开始。本质用例交付了正在设计的系统特有的职责集合，从而提供有价值的机会。

从本质上说，系统职责必须与对象集合职责相匹配。这有两个主要含义：首先，CRC 或职责驱动设计方法依据的职责可以来自本质用例，也可以来自领域分析；其次，我们可以在任何时候检查对象职责的集合，以确认它们是否真的符合系统需求。不管是快速评价其它备用设计方案，还是考察其它现有的可重用设计资源，如模式、框架或组件，我们都能够确认这些对象能够满足用例规定的系统职责。前者意味着在设计初始能够有更好的操作指南。后者意味着设计与需求之间具有更好的可追溯性。

相关工作

本节综述对我们的方法有影响的公开出版物。

用例方法是面向对象软件工程（OOSE）引入的一个关键概念（Jacobson 等, 1992）。用例有很多种形式，对我们影响最大的是 Wirfs-Brock 称为“会话”的那一种（“Wirfs-Brock, 1993 及 1994”）。在实践中我们采用的是“本质用例”，它是从用户界面设计中发展起来的（Constantine 和 Lockwood, 1999）。Cockburn（2001）提供了与用例相关的论述和方法的综述。

OOSE 建议如何在系统开发语境中应用用例。用例也是更现代化开发过程的特征，如 Rational（瑞理）统一方法（Unified Method）（Jacobson, Booch, 和 Rumbaugh, 1999）。在 UML（Jacobson, Booch, and Rumbaugh, 1998）等图示语言中也处理了用例。影响我们的工作还有，关于如何利用用例对需求建模的研究（Rosenberg 和 Scott, 1999, 以及 Armour 和 Miller, 2001），极限编程（Beck, 1999）中对“用户故事”（该概念很象用例，但是写得很具体）的应用，以及强调开发者和利益相关者参加复审的重要性。OPEN 技术工具箱（Henderson-Sellers, Simons, 和 Younessi, 1998）提供了一个详尽的调查，包括关于用例和 CRC 卡片的讨论和指南。

CRC 卡片技术，包括卡片本身和角色扮演，对我们的方法影响很大。CRC 最初只是一个教学手段（Beck 和 Cunningham, 1989）。但最近的研究和实践表明，CRC 卡片作为一个通用设计技术有很大好处（Wilkinson, 1996, 以及 Bellin 和 Suchman-Simone, 1997）。另外，CRC 卡片技术和本质用例都大量使用“职责”做为分析指南。我们对这种启发式方法的应用也受到“职责驱动设计”过程的影响。（Wirfs-Brock 和 Wilkerson, 1989, Wirfs-Brock, Wilkerson 和 Wiener, 1990）

做为一种复审技术，我们的方法也反映了其它软件质量保证工作的结果。过去需求检查或者很不正规，或者仅仅是一次明确而正式的“软件检查”（Fagon, 1979）的一部分。我们的方法不是完全正式的，但提供了一个有用的整体结构，类似于 Yourdon（1989）的“排练”，但强调用例而不是设计或实现。

结论

本文提出使用索引卡片和角色扮演进行用例复审的技术，以帮助开发更好的用例。该技术基于面向对象设计中成熟的 CRC 卡片技术。在角色扮演中只有两类角色：用户和系统，这来自于使用对话格式的形式描述用例，如同本质用例中使用的一样。

该技术是轻量级的，并不拘泥于形式，这意味着较低的准入门槛，特别是对初学者和利益相关者来说。这一点很重要，因为它能够保证利益相关者很容易参与到复审过程中来，这无疑能够保证高质量。

我们发现该技术对于暴露用例中潜在的缺陷和异常很有效，在确定系统边界时更是有帮助，因为这种划分正好与角色划分相同。另外，利用本质用例方法对于面向对象设计还有其它好处，它可以带来设计和需求之间更好的可追溯性。

参考文献

- Frank Armour and Granville Miller. (2001) *Advanced Use Case Modeling: Software Systems, Volume 1*. Addison-Wesley.
- Kent Beck and Ward Cunningham. (1989) A laboratory for teaching object-oriented thinking. In *Proc. of OOPSLA-89: ACM Conference on Object-Oriented Programming Systems Languages and Applications*, pages 1-6.
- David Bellin and Susan Suchman-Simone. (1997) *The CRC Card Book*. Addison-Wesley.
- Alistair Cockburn. (2001) *Writing effective use cases*. Addison-Wesley.
- Larry L. Constantine and Lucy A. D. Lockwood. (1999) *Software for Use: A Practical Guide to the Models and Methods of Usage Centered Design*. Addison-Wesley.
- Michael E. Fagan. (1986) Advances in Software Inspection. *IEEE Transactions on Software Engineering*, pages 744-751, July.
- Brian Henderson-Sellers, Anthony Simons, and Houman Younessi. (1998) *The OPEN toolbox of techniques*. Addison-Wesley.
- Ivar Jacobson, Grady Booch, and James Rumbaugh. (1999) *The Unified Software Development Process*. Addison-Wesley.
- Ivar Jacobson, Mahnus Christerson, Patrik Jonsson, and Gunnar Overgaard. (1992) *Object-Oriented Software Engineering*. Addison-Wesley.
- OMG Object Management Group. (2000) Unified Modeling Language (UML) 1.3 specification.
- Doug Rosenberg and Kendall Scott. (1999) *Use case driven object modeling with UML: A practical approach*. Addison-Wesley.
- James Rumbaugh, Ivar Jacobson, and Grady Booch. (1998) *The Unified Modeling Language Reference Manual*. Addison-Wesley.
- Nancy Wilkinson. (1996) *Using CRC Cards - An Informal Approach to OO Development*. Cambridge University Press.
- Rebecca Wirfs-Brock and Brian Wilkerson. (1989) Object-oriented design: A responsibility-driven approach. In Norman Meyrowitz, editor, *Proc. of OOPSLA-89: ACM Conference on Object-Oriented Programming Systems Languages and Applications*, pages 71-75.
- Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. (1990) *Designing Object Oriented Software*. Prentice Hall.
- Rebecca J. Wirfs-Brock. (1993) Designing scenarios: Making the case for a use case framework. *The Smalltalk Report*, 3(3).

Rebecca J. Wirfs-Brock. (1994) The art of meaningful conversations. *The Smalltalk Report*, 4(5).

Ed Yourdon. (1989) *Structured Walkthroughs*, Prentice Hall/Yourdon Press.

版权

Biddle, Noble, Tempero © 2001. The authors assign to ACIS an educational and non-profit institutions a non-exclusive licence to use this document for personal use and in courses of instruction provided that the article is used in full and this copyright statement is reproduced. The authors also grant a non-exclusive licence to ACIS to publish this document in full in the Conference Papers and Proceedings. Those documents may be published on the World Wide Web, CD-ROM, in printed form, and on mirror sites on the World Wide Web. Any other usage is prohibited without the express permission of the authors.

UMLChina 保留中译本一切权利

UMLChina 书籍推荐



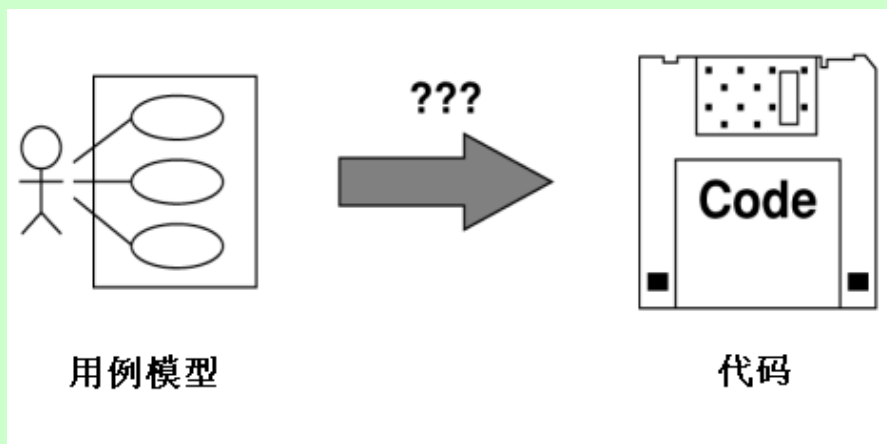
《人月神话》，Frederick P. Brooks, Jr 著，Adams Wang 译，清华大学出版社，预计 2002 年 6 月出版。[作者答疑](#)。[留言评论](#)

“《人月神话》一书被大量地引用，很少存在异议；相比之下，《没有银弹》却引发了众多的辩论，编辑收到了很多文章和信件，至今还在延续。他们中的大多数攻击其核心论点和我的观点--没有神话般的解决方案，以及将来也不会有。他们大都同意《没有银弹》一文中的多数观点，但接着断定实际存在着杀死软件怪兽的银弹--由他们所发明的银弹。今天，当我重新阅读一些早期的反馈，我不禁发现在 1986 年~1987 年期间，曾被强烈推崇的秘方并没有出现所声称的戏剧性效果。”

——Frederick P. Brooks, Jr

UMLChina 培训

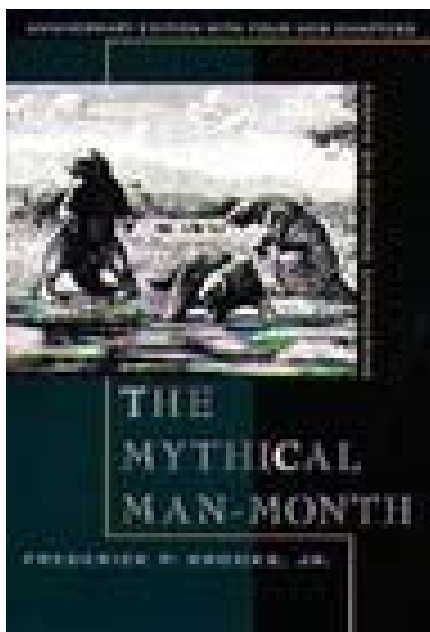
The real thing



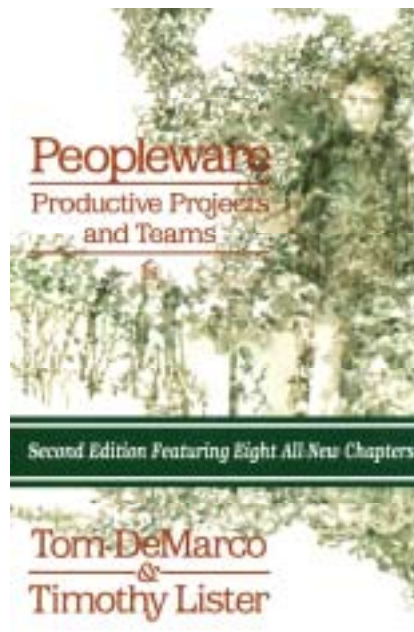
利用 UML 的 20%就可以为 80%的问题建模
-- 《UML 用户指南》，第 32 章

详情请垂询：think@umlchina.com

中文译本即将发行！



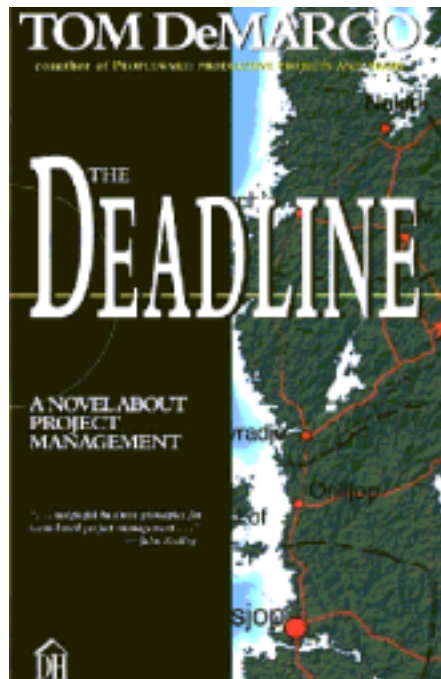
《人月神话》20 周年纪念版，Fred Brooks
翻译：UMLChina 翻译组 Adams Wang
散文笔法，绝无说教，大量经验融入其中



《人件》第 2 版，Tom Demarco 和 Tim Lister
翻译：UMLChina 翻译组方春旭、叶向群
微软的经理们很可能都读过—amazon.com



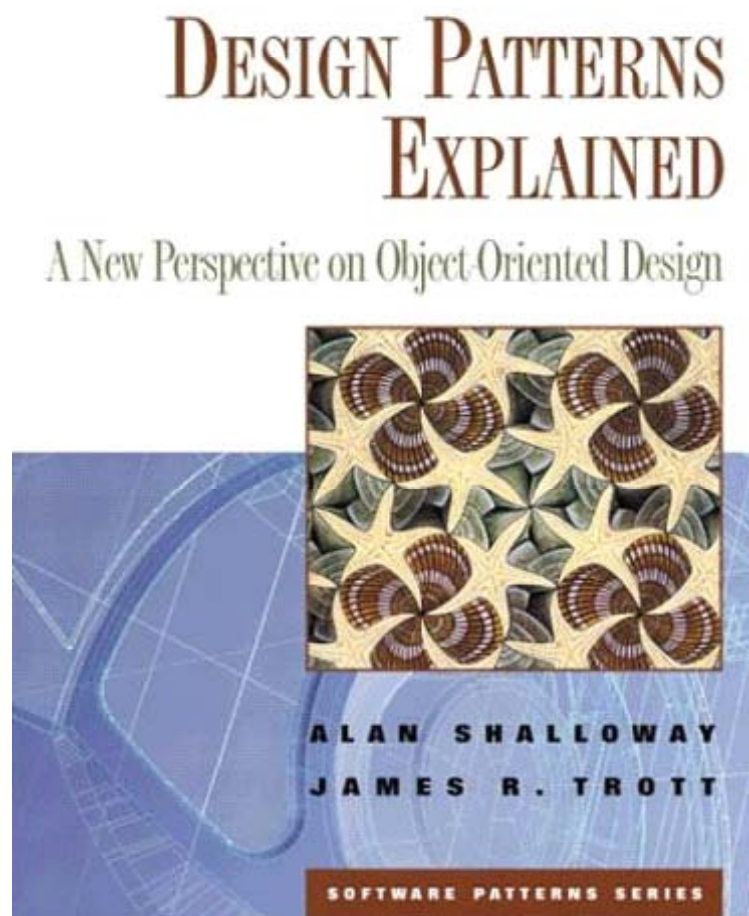
《面向对象项目求生法则》，Alistair Cockburn
翻译：UMLChina 翻译组乐林峰
Cockburn 一向通俗，本书包括十几个项目的案例



《最后期限》，Tom Demarco
翻译：UMLChina 翻译组透明
这本最通俗，这是一本软件开发小说

看不懂《设计模式》？

它的作者推荐《设计模式精解》----中译本即将上市！透明 译



直接学习《设计模式》这本书是非常困难的。作为入门，我推荐 Alan Shalloway 的《设计模式精解》。

--John Vlissides 写于 [UMLChina 答疑板](#)

针对用户界面设计的用例结构和式样

Larry L. Constantine, Lucy A. D. Lockwood 著, zhen_lei 译

摘要

虽然用例广泛应用于面向对象软件工程和用户界面设计，但并没有很好地定义。人们较少关注书写用例所采用的各种描述形式，这些描述针对用户界面设计和软件可用性定义了用例及其相关序列。本文结合实例展示常见的样式，并讨论各自的优缺点。与常规用例及场景进行了比较，本质用例（essential use cases）是服务于以使用为中心设计的用例变体。为了最有效地支持用户界面设计，尤其对于大型复杂项目，本文提出了一种高度结构化形式的用例，并介绍了用例中新的描述元素和关系。本文包括表示部分或灵活的交互序列的方法，业务规则间的关系，同时，本文还澄清了一个经常容易引起混淆的关于扩展的概念，即区分两种完全不同的扩展类型：同步扩展和异步扩展。

引言

自从用例引入支持面向对象软件工程，它的使用增长很快，在开发方法和开发实践中迅速普及。用例被普遍应用的原因部分归功于本身的有效性，它已被证明为一种多用途的工具，适用于分析与开发的很多方面。但另一方面，由于其定义的不严谨，多数开发者说他们正在使用用例，不论其范围、细节、目标、形式、结构、式样和定义上有多大不同，因为几乎所有的事情都可以称作用例。已颁布的特定术语混淆了明显的区别，如场景与用例，这更进一步加深了混乱。作为应用工程与用户界面设计之间以及软件开发和设计间有效的桥梁，用例具有多变的适应性。对需求工程师来说，用例提供简明的用户建模媒介；在用户界面设计者手中，用例成为理解用户需求和引导界面设计的有力工具，对软件工程师，用例指导对象间的通讯设计以满足功能需求。在这些方面的成功依赖于对用户界面设计而不是软件设计的认识。它原本是为支撑软件构件和构件间交互设计的模型，不能自动适应针对组织用户界面构件以及用户与这些构件间的交互的设计。

用例的不明确

一个非常值得注意的用例问题是：尽管几乎完全缺乏精确的定义，用例仍获得广泛应用。所有完全的或主要是描述用例的书籍甚至没有给出一个定义。Jacobson 最初的定义简单、概括但缺乏描述性：

用例是使用系统部分功能的特定方法。[一个用例]包括一个完整的执行者（Actor）和系统间交互过程。

部分由于不精确的定义，部分由于用例在不同应用和不同目的中使用的混淆和混乱，许多用例，包括一些已发表的用例，混合了分析与设计、业务规则与设计目标、内部与接口描述，并带有无意义的旁白，试图包含所有成份。混乱如此之深以至于最无约束的杂烩也被称为用例。

实例 1：

顾客在饭店预定一个房间。如果有可用房间，饭店会尽可能接受预定。当顾客到达时，由注册职员负责接待。该职员会将客户提供的细节与已有记录进行比较。有时顾客在到达前没有进行预约。一些顾客希望无烟房间。[Roberts 等., 1998: 68]

在这里，用例是什么？其目的是什么？是预定一个房间还是获得一个房间？或者是为顾客进行登记？谁是用户以及他们扮演什么角色？是顾客或者雇员还是电话转接员完成预定？要设计什么样接口的？

为了保持信誉，UML（傲慢的名字，统一建模语言）的开发者们一致采用同样的段落，试图将定义的范围缩小。例如，一个用例是：

一系列特定的动作，包括变化序列和出错序列，系统、子系统或者类可以通过与外部执行者交互进行执行[Rumbaugh 等, 1999: 488]。

这个小组的其它工作 [Jacobson, Booch, 和 Rumbaugh, 1999: 41; Kruchten, 1999: 94]将这个定义限定为这个短语：“那些对特定执行者获得一个可观察的结果。”

不管增加的定义多么微小，当前的“官方”定义实际上从 Jacobson 最初重于使用的定义转移到“系统为中心”的观点：重点放在系统如何工作而不是用户完成的或所希望的用途。我们的观点是，这种从内到外的透视方法，虽然精致，但实际上导致在界面设计中使用用例时出现问题。

在一个流行的 UML 介绍中，Fowler 提供了一个稍微接近以使用为中心的用例定义：

用例是用户和计算机系统间的典型交互... [用例]捕获用户可见的功能... [并且]完成一个独立的用户目标。

尽管如此，大部分情况下，用例定义为不与用户相关，而与“执行者”相关——与系统交互的外部代理。对软件设计而言，执行者包括与正在开发的系统打交道的其它外部系统，但从界面设计角度出发，当然应该只与人类用户相关。

许多设计人员明白，在某种程度上，用户界面设计的时候，需要考虑的往往不是用户本身，而是他们在系统中扮演的角色。

UML 认可的“执行者”在这里并不适用，因为它导致一种尴尬和根本无意义的形式，如下所述：

一个执行者是用户在系统中扮演的某种角色。

所有其他领域的论文中，执行者不是一个角色，而是有着明显的区别。将角色称为执行者的文字游戏特别使用户和客户感到混淆，同时也在设计人员意识里造成模糊概念，使他们下意识地混淆角色的职责和角色本身。因为这个原因，我们更愿意将由人类扮演的角色称为“执行者”，也就是系统的用户。为了避免与 UML 词法的混淆，我们采用系统执行者——其它系统——与扮演角色的用户进行区别。因此，一个角色由一个用户和系统间的联系构成，并被定义为一组特征需要、期望、行为和职责。

具体用例和本质用例

在通常的应用中，用例描述外部用户（或系统执行者）通过一个特定的界面与系统进行交互。由于它们采用具体的术语进行表示，这些用例就是具体用例。例如，考虑“回收物品”用例的开始[*Jacobson et al., 1992: 157*]。

实例 2：

当顾客按下面板上的“开始按钮”，一系列过程开始。这个面板中内置传感器，因此系统被激活。

这类用例不适合用户界面设计，原因很简单，其中包含对特定用户界面和交互设计的假定。

为了书写这种形式的用例，你必须定义好用户界面，至少是一部分。为什么使用回收机的用户要按一个按钮而不是触摸屏幕？为什么顾客必须完成将物品放入回收机以外的动作？这种用例中的隐含假定不必要地限制了设计，并且，在实际中经常导致低质量的用户界面，这些界面中常常伴随不必要的传统复杂交互过程。

正是这种循环论证，促使我们设计一种截然不同的用例，并且开发一种系统过程，服务于创建和使用

这种针对用户界面设计的用例。好的用户界面设计工具的秘诀是，将关注的重点由交互过程转向用户意图，从复杂转向简单，将重点放在简单化，用简洁的语言描述用例的实质，而不是采用带有特定细节和多种交互过程的复杂用例。我们用“本质用例”描述这种模型[Constantine, 1994b; 1995; Constantine 和 Lockwood, 1999]，因为它包含最初由 McMenamin 和 Palmer[1984]引入的实质模型的含义，这就是对一个问题进行抽象、泛化、与技术无关的描述。

我们将本质用例定义为：

简单、孤立、完整、有意义，并且很好定义的任务，该任务属于外部用户与系统相关的特定角色的职责，包含在完成这个任务过程中，用户的意图和系统的响应采用与实现无关的应用领域或外部用户所熟悉的词汇进行抽象的、与技术无关的描述。

并非只有我们认识到在用例建模中需要这种“目的为中心”的方法，以及在用例构造过程中将重点转移到抽象。Kaindl [1995]和 Cockburn [1997; 2000] 建议将目标合并到用例中，Graham [1996]讨论了抽象用例的价值。统一的观点正在形成，如果不是在细节上，至少是在概念上：最近，也在加入到统一过程中[Lee 和 Xue, 1999]，甚至统一过程的创建者们已经认识到具体用例的局限性和本质形式的优点[Jacobson 等., 1999: 164]。本文的意图是，通过增加一些关于用例的清晰、精确、有深度的讨论，帮助沟通软件工程和应用工程。为达到这个目标，我们将探究不同的描述样式及其在用户界面设计中的价值，我们还将介绍结构化用例的概念，这种用例使用高度结构化的方式组织和描述，以加强在用户界面设计和与其它开发过程集成中的应用。

图示法

我们必须对本文采用的特定图示法进行一些辩护性说明。虽然我们不知道 UML 占有统治地位，我们想将图示符号与之保持一致已经太迟了。虽然 UML 对它包含的许多模型的图示上给予了很多关注，但对于用例，几乎没有详细说明确定的图示符号，也就是说，实际上，定义什么是用例，用例关于什么的规范上几乎没有详细说明确定的图示。对这个关键问题的疏忽无疑是造成混淆什么是、什么不是用例的原因。

像所有应用专家和用户界面设计人员一样，我们也敏锐地觉察到，信息的表现形式很大程度上影响信息的有效性。图示实际上是模型化的用户界面。我们曾长时间争论，对于创建和说明模型的人而言，可视化形式的图示是显著提高效率还是降低效率[Constantine, 1994a; Constantine 和 Henderson-Sellers, 1995a; 1995b; Page-Jones, Constantine, 和 Weiss, 1990]。尽管在这些问题上，UML 包含一些观点，即使没有得到承认，我们

关于图示功能的大量讨论和建议好像被当作了耳边风。现代建模者被迫在不规范的图示法和带有各种缺陷的图示法中进行选择。（这超出了本文的范围，参见[Henderson-Sellers, 1999; Henderson-Sellers 和 Barbier, 1999; Henderson-Sellers 和 Firesmith, 1999]）。

关于用例图示符号有两方面需要说明：一是关于用例中用到的符号，这是一种用来定义用例的式样，应用于用例描述体中；二是关于用例间的关系如何用图形形式进行可视化，通过用例将整个任务结构模型化。这两部分将分别进行说明。

以使用为中心的设计

以使用为中心的设计过程

本质用例在用户界面设计的有序过程中得到发展，虽然我们的主要目的不是讨论方法论，关于过程确实有值得一提的地方。被称为“以使用为中心”的设计突出了使用是中心的事实（而不是用户本身）。虽然我们的方法具有概念更为广泛的用户为中心设计理念的特征，我们称之为以使用为中心的设计，目的是希望将注意力集中于与用户界面设计相关的用户身上，并且强调作为任务模型的用例的关联。

以使用为中心的设计是一种灵活的可扩展的“工业级”过程，可以应用于商业与工业，问题的范围可以从小规模 WEB 站点设计到 18 个月的工业自动化设计，该项目包括 6 个人的设计组和 17 个软件开发人员。它可以用于软件设计和基于 WEB 的应用设计，它可以适用于不同的应用，如银行、出版、制造、教学以及设计消费型电子产品和工业自动化工具。

如图 1 所示，以使用为中心的设计是一个模型驱动过程，采用三种主要的抽象模型：用户角色模型、任务模型、内容模型。用户模型捕获并组织特定用户和被设计系统间的特定关系，任务模型采用本质用例的方式表示用户在用户角色的位置上所关心的与系统相关的事情。内容模型表现用户界面的内容和组织，而不管其表现形式和行为。这三个核心模型结构化并与领域模型互相依赖。领域模型可能采用数据模型或对象模型，或者仅是一个词汇表，这由项目的范围决定。（可选模型捕获实际环境和当系统发布时的可选内容，该模型与核心模型同时开发，但该问题超过了本文的范围，有兴趣的读者可以咨询 Constantine 和 Lockwood [1999]）

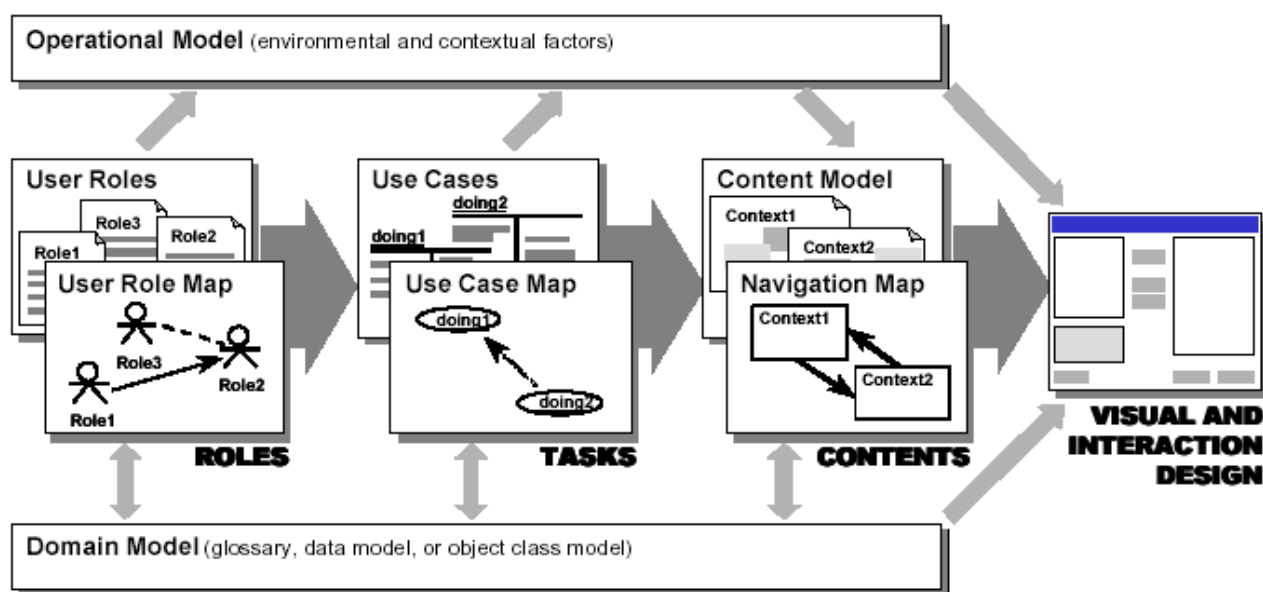


图1 以使用为中心的设计主要模型间的逻辑关系

三个模型中，每个都包含两部分：一个描述集合和一个描述中的关系图。因此，用户角色间的联系由一个用户角色图表示。用例之间的关系模型化为用例图（对应于 UML 中的 Use Case Diagram，但不完全相同）。内容模型表现用户与系统交互在不同情况下的内容（或视图）。导航图表现这些不同交互之间的互相连接关系。内容模型和导航图一起，有时叫做抽象原型[Constantine, 1999]。（逻辑原型或逻辑用户界面设计，在统一过程中[Jacobson 等, 1999: 161ff]本质上是相同的概念，反映了我们与 Jacobson 及其同事们在 90 年代初期的工作）

概念上，最终的可视和交互的设计经常是在纸上书写的原型形式，是基于抽象的原型，也就是内容和导航模型。这些模型从任务模型中获得，而任务模型本身依赖于用户角色模型。尽管如此，在实际中，随着建模者注意力的任意转移，角色模型，任务模型和内容模型基本上是同时建立的，这取决于分析和解决问题的过程。通过为设计者对问题的不连贯但是逐步发展的理解提供框架，模型有助于组织和系统化建模过程。这种同时进行模型化处理的过程比传统过程更为灵活和有效。传统过程有比较严格的次序，不管是采用迭代生命周期还是采用瀑布生命周期。

以使用为中心的设计是“由外及里”的方法，在这种方法中，内部系统设计支持用户界面，而用户界面支持外部用户的需求。因为这个原因，本质用例被当作基础模型，具体用例，如果需要的话，被看作是导出的模型。实践中，支持系统执行者的用例可以从具体用例中直接开发，同时本质用例也支持用户角色。

以使用为中心的设计包含一个代理过程，可以使之与任何软件开发过程相适应。例如，关键元素可以

简洁地对应于“轻量级”的 WISDOM 方法。另一方面，与统一过程的“重量级”结构相适应的考验更具挑战性。其它情况下，统一过程通过将建立用户界面的原型作为需求过程的一部分，将事情的因果倒置，如同将马车放到马的前面。采用原型进行用户界面设计是解决方案的一部分而不是问题定义的一部分。增强可用性的一个关键是服从于表示需求的用户界面细节，使其独立于实现。统一过程中的问题是难以克服的，尽管如此，经过适当的改进和重新定义，它可以适应于以使用为中心的设计实践。具体的实现细节超出了本文的讨论范围。

任务建模，场景和用例

任务建模是以使用为中心设计中的关键活动，用户角色是有用的结构，但用户角色模型本身没有意义，它首先是作为连接用例的标识和构造的桥梁，其次作为用户和影响界面设计工作环境的信息载体。内容模型是另一个桥梁，连接用例与用户界面设计。角色模型与内容模型都使设计过程变得容易，并且有效地促进用户界面系统化开发，但在某些情况下，它们多少可以省略，而任务模型则是必不可少的。

当然，用例和本质用例仅仅是许多可用的任务建模方法之一，这些方法范围很广，从研究人员和学者采用的严密的高度结构化方法，到不规范的电影剧本式的非规范场景。场景被广泛用于一般设计与用户界面设计，特别是捕获和描述任务[Carroll, 1995; Rosson 和 Carroll, 2000]。虽然一些作者[Booch, 1994; Graham, 1996; Wirfs-Brock, 1993]曾采用这个术语，把它多少等同于用例，但在通常的设计中，使用场景与用例有很大不同。场景通常被扩展为描述故事，它们倾向于丰富的、现实的、具体的、特定的，通常为实现逼真而充满无用的细节。

例如，一个场景“Ian Smith 通过我们基于 Web 新的技术支持系统获得 HyperCam 软件的安装过程帮助”可能如下：

例 3

现在是凌晨两点，Iam 不能正确安装其新的 HyperCam 软件，他打开浏览器，并指向 www.camerahype.com，看到欢迎页，并等待相应的主页出现。他向下滚动该页并点击技术支持链接。在出现的表格中，他键入他的姓名，查找与 HyperCam 一同寄来的顾客 ID，并将它填到表格中。他点击提交按钮。他浏览技术支持页，最终点击了一个表示安装帮助的古夫图像，这将他带入安装帮助页，在那里他填写事件报告表，表格提交后，他对系统给出的建议不满意，于是他来到“与我们联系”页面，发了一个 Email 信息。

这就是长期以来流行的场景，带有传统的 HCI 和用户界面设计特有的真实感。与之对照，用例倾向于去掉冗余，并且较少包含有趣的描述，用例中的细节被去掉，“变量”或类名称代替了大部分文学描述，如同在这个具体用例中“获得安装帮助”中一样。

例 4

当顾客到达“顾客登录页”时，用例开始。在这里，顾客在表中键入他/她的姓名和顾客 ID 并提交。系统会显示技术支持主页，主页上有一个问题目录列表，顾客点击表中的安装帮助，系统提供“事件报告”表，顾客完成并提交该表，系统显示建议的解决方案。

与之相对照，类似地，“获得某种问题的帮助”的本质用例更为简练。

例 5

用户意图	系统响应
确认自己是用户	
	显示帮助选择列表
选择帮助项	
	需求描述
描述问题	
	提供可能的解决方案

正如这些例子表明，场景、传统（具体）用例和本质用例（抽象用例），处于不同的抽象层次，一个场景包含用户实际执行的几个用例的组合。

例如，在这个例子中，场景包括“用户登录”、“获得某个问题的帮助”、“发 Email”用例的实现。与惯例不同，UML 的建立者将场景声明为通过一个用例具体实现的单线程过程。我们更倾向于采用设计和人机交互通讯中非常普遍的用法，它将一个场景作为一组（一个或多个）用例的实现组合，也就是说，一

个线程贯穿多个用例的实现。

场景由一组用例的实现组成,由扩展的描述或一系列事件或状态表达,构成一个叙述小品,或具有真实感的故事。

在面向对象术语中,用例被“例示”,但这个词对于一般用户而言没有意义。扮演用户角色的人不会“例示”用例,他们“执行”、“实现”或“表现”用例。

在三种任务模型中,本质用例最为健壮,特别是在采用的技术变更的情况下,这很大程度是由于它们用接近于问题本质的方式模型化任务,并且与解决问题的具体设计方案无关。例如,本质用例“获得某种问题的帮助”不需要重写或改变,尽管将来可能将技术支持系统改变为通过电话实现的按键选择,声音回答的界面。与之相比,例3中的场景和例4中的具体用例都不再适合。

用例分解

在以使用为中心的设计中,每个本质用例都代表增值的潜力——任务聚合体中的一小部分。采用本质用例模型化倾向于对任务进行相对细致的分解。支持系统的工作被分割为一个简单用例的集合,这些用例间的关系包括“包含”、“特殊化”、“扩展”等关联形式。有些作者批评[例如, Fowler, 1997; Korson, 1998]对用例进行过度的分解。但在用户的以使用为中心的设计中,分解本身不是目的,也不是构成纯概念的目的树。

对于以使用为中心的设计,在分解为相对小而互相关联的单元后,有许多明显的优点。首先,它引申出较小的简单用例模型,在这些模型中,整个模型的每一部分相对简单,容易理解,并且与相对较少的其它用例相关。第二,整个模型通过复用变得简单,因为相同的元素被分解到分离的用例中,而不是在许多地方多次重复。

当在设计用户界面和内部结构中应用的时候,这种分解的结构促进内部和外部的构件及设计的复用。问题的每一部分仅解决一次,解决方案可以重复使用,只要相应的用例出现。

根据我们的设计经验,在提高可用性时,有一个细微但回报巨大的细节。反映精细、以使用为中心的任务分解而设计的用户界面让用户可以灵活地重组新的任务。采用这种方法,设计者可以开发强大的工具来支持甚至是不可预见的应用和应用模式。

用例叙述式样和用户界面设计

场景、具体用例和本质用例是各种建模式样的实例，并且在不同情况下表现出各自不同的优势。可用性检查和可用性测试经常要求适当详细或特定的场景，这些场景运用多种功能，并且在详细的检查中可以覆盖用户界面的较大部分。非技术性的最终用户经常对现实和特定的场景或者具体用例感到满意。但是对软件工程和内部结构设计，更为传统的具体用例被证明为比较有效。对于指导许多重要的编程和程序设计，本质用例可能过于抽象，而场景又太不规范，其中混合了各种功能、特性和使用线索。

然而，对用户界面设计而言，采用抽象的本质用例是最恰当的。本质用例使设计者建立核心任务结构，对界面设计的细节不必过早地进行假设。我们曾长时间讨论抽象促进创造性的新方法，许多小组最近使用本质用例的经验获得了一系列新的软件专利，这无疑支持了这一论点。

模型中的语言和结构

因为语言影响整个模式，因此书写式样、格式、用词、甚至用例描述中采用的语法都能够影响为设计用户界面所建立用例的价值。当然，这一点针对软件结构设计同样正确，但与其它面向对象的软件工程课题相比，我们这里关心的是可用性与用户界面设计。

给出真正好的用户界面设计非常困难，并且实际设计问题经常很复杂，设计者需要所有可获得的帮助。一些书写用例的式样促进好的用户界面设计，但同时其它一些样式与之无关或甚至起阻碍作用。

用例在用户界面设计过程中所起的作用越是重要，其书写形式和式样就越显得重要。如果在以使用为中心的设计中用例模型直接驱动用户界面设计，那么描述式样和表现形式就显得极为重要。

普通的描述方式

不管场景、用例和本质用例的总体区别，我们已知道用例的书写式样对设计者和设计结果的质量有很大的影响。这一节，我们描述一些常用的用例书写式样。

用例的过程或描述体，在 UML 中指事件流程，包含一个给定用例的定义。用例描述可以书写为多种式样，采用多种不同的方法。（关于用例描述式样及其相关结果的一个讨论，请参阅[Cockburn, 2000]）

连续描述

许多作者和建模者偏爱连续的无格式的描述，这种形式最初由 Jacobson 和他的同事提出，以下是最近

从 Web 上得到的一个例子：

例 6

当顾客从可选交易菜单上选择了“提取现金”后，现金提取交易开始执行。顾客从可选账户中选择账户，然后从可选金额中选择要提取的金额。系统检查是否有足够的现金，如果没有足够的现金，将报告一个错误，这将启动“失败交易处理”过程。如果有足够的现金，系统将顾客的信用卡号、PIN、所选择的提取账户和金额发送给银行，银行将核准是否同意交易。如果该交易获得同意，机器将数量正确的现金支付给顾客，并提供收据。如果该交易没有获得同意的原因是 PIN 不正确，“PIN 错误”部分被执行。其它原因导致的交易未获得同意将启动“失败交易处理”。机器会通知银行，已获准的交易是否完成，如果完成，银行将从用户账户中扣除相应的金额。

这种书写方式存在很多问题，用户和系统间没有明确的界限。描述混淆了内部需求和外部需求，并且在内部视角和外部视角之间来回切换。包含问题本质的基本元素（如：“机器支付正确数量的现金”）与暗含的用户界面设计混在一起（如：“顾客从菜单”中选择金额）。由于缺乏结构，迫使读者在整个文字中追踪问题的实质，对用户界面设计而言，重要的部分淹没在无关的叙述中。

数字序列

另一种常用的书写式样是采用一系列数字，例如，对上述用例使用另一种描述方式如下：

例 7

提取现金

- 1、 当客户插入 ATM 卡时，用例开始。系统读取并检验卡上的信息。
- 2、 系统提示输入 PIN，客户输入 PIN，系统验证 PIN。
- 3、 系统提示客户希望进行的交易类型，客户选择“提取现金”。
- 4、 系统提示输入金额，客户输入金额。
- 5、 系统提示输入账户类型，客户选择账户类型。
- 6、 系统 ATM 网络连接，验证客户的 ID、PIN 以及需要的金额。

- 7、 系统询问客户是否需要收据，只有当系统中有可供打印单据的纸时，才执行本步骤。
- 8、 系统询问用户是否已取出信用卡，客户取回信用卡。（这是一个安全步骤，确保用户将卡取回）
- 9、 系统支付现金。
- 10、 系统打印收据。
- 11、 用例结束。

这种描述的优点很容易看出来：分离为独立的步骤使整体浏览用例和发现交互的特征变得容易。但是，它包含有连续式样的一些问题。尽管分割为独立的步骤，每个单独的步骤中仍混合了用户和系统动作。这个描述展现了系统为中心的视图：除了一个例外，其它步骤都以系统动作开始。

例 6 和例 7 也揭示了用例中常用的描述式样，它们罗嗦，充满冗余。尽管用例开始和结尾不言而喻的，许多作者仍不厌其烦地写上，“当... 时用例开始”或是声明“用例结束”。

分离描述

因为结构很少或没有结构，为了能够清楚地说明，连续和序列描述式样需要反复说明视点或焦点（系统做这个，用户选择那个，系统完成什么事）。这导致过多的语言，尽管如此，系统内部和外部的界限仍不清楚，只能通过仔细阅读整个说明获得。

这个问题的简单的解决方案是将用户和系统完全分开。虽然也有其他人针对其它形式的任务模型提出类似的方法，但对用例来说，这种分开的方法最初由 Wirfs-Brock [1993]提出。Wirfs-Brock 将具体用例的描述分为两列，用户动作模型和系统响应模型，在这种式样中，边界表示的用户界面很明显，不需要反复说明哪一部分描述用户，哪一部分描述系统。例如，另一个版本的“提取现金”用例，采用分离描述。

例 8

用户动作	系统响应
将卡插入 ATM	

	读卡 提示输入 PIN
输入 PIN	
	验证 PIN 显示选择菜单
选择账号	
	提示输入金额
输入金额	
	显示金额
确认金额	
	返回卡
取卡	
	如果可能支付现金

这种方式如此优越，任何与项目相关的人都能读懂，以至于很难找出它不适用于什么情况。当然，没有什么理由可以阻止用例书写者给动作和响应加上编号，这样效果会更好。

伪码

有些用例作者在用例描述中采用各种结构化英语或伪码，例如，经常用到如下结构：

```
until customer_done repeat
```

```
if valid_user_code then do...end_do else do...end_do end_if
```

虽然这样表示看似更为精确，并且可能对软件工程师更为熟悉，但对最终用户，这种方式几乎是不透明的，而且这种精确性可能使交互过程的真实性变得模糊。

根据我们的经验，用例的描述越象代码，实际上它是代码的可能性就越大。编程是必要的和卓越的工作，但它属于解决问题而不是描述问题。在对问题建模上，本质用例是主要的任务模型。

交互图

一些设计者采用图形化模型，如 UML 中的序列图和合作图，代替常用的描述文字来定义和模型化用例。虽然较好（但有争议）地适应于软件工程的目标，这也是当初设计的初衷，交互图不适用于用户界面设计。它们使设计者投入过多精力考虑软件对象间的消息传递，而不是将注意力集中在外部用户的角度和任务的本质特征。像伪码一样，它们导致过早地关心设计，但对非技术用户和客户而言，其结构与说明更加难懂。

前提和完成状态（Pre- and post-conditions）

我们坚持在用例中不采用软件编程的原则，特别是在那些引导用户界面设计的情况下，但有一个重要的例外，这就是前提和完成状态的使用。它们在编写用例叙述时经常用到。例如下例[Schneider 和 Winters, 1998]：

例 9：

订购

前提：合法用户已登录到系统

基础路径：

- 1、当顾客选择订购时，本用例开始。
- 2、顾客输入姓名与地址。
- 3、如果顾客仅输入邮政编码，系统自动给出其所在城市。
- 4、顾客输入希望购买的商品编码。

- 5、系统显示该商品的描述和价格。
- 6、系统保存用户输入的每个定单。
- 7、顾客输入信用卡信息。
- 8、顾客提交信息。
- 9、系统验证信息，保存订购信息，并将付款信息提交结算系统。
- 10、 如果付款被确认，订购被标记为“确认”，用户会得到一个订购 ID，用例结束。

可变路径：

步骤 9 中如果输入信息出现错误，系统提示用户输入正确信息。

完成状态：订单被保存，并标记为确认。

虽然我们起初反对这种来自编程、看似对任务建模有干扰作用的方法，但我们发现前提和完成状态对实现支持以使用为中心设计的目标有很大的帮助。一方面，前提说明后，最终用户会感到满意一些。例如，上例中如果缺少前提，用户或顾客就会反复抱怨用例无法工作或它不完整，因为没有用户登录系统的步骤。当然，许多作者可能试图将这个步骤作为用例的第一步，如例 7 和例 8 中那样，但这种做法有一个缺点，就是在每一个依赖这个步骤的用例中，都要重复描述，而这样会导致描述真正的任务结构出现偏差（例如，在美国的大多数 ATM 机上，用户不需要在每个独立交易前都插入信用卡）。在例 6 中，前提隐含存在于描述的句子中（“提款交易开始……”），这显然不如明确说明来得清楚。

前提还为表示用例的执行顺序提供了一种非过程方法。如“登录系统”是一个已描述好的用例，那么，前提“合法用户采用‘登录系统’进行登录”就恰当地确定了用例间的顺序。用例间的这些联系对用户界面设计者和软件开发者同样透明并且有用。与纯粹主义者观点相反，这个实践过程不是将过程模型引入到面向对象中，而仅仅是表示实质上存在的任务执行过程。实际上，它们作为其它用例的前提或完成状况的用例，提供了直接的逻辑上的工作流建模方法。而这种任务结构常被用例建模者忽略。（一个合适的 CASE 工具可以支持、跟踪或强调工作流关系）

抽象

用例描述可以采用不同层次的抽象方式来书写，在我们自己的设计工作中，有时把一个式样称为“适

当的模糊”。其中忽略了许多有用的细节，特别是在任务模型初期。例如，在一个数控机床应用中，在设计早期，一个用例形式可能如下：

例 10

用户意图	系统反应
输入设定参数	
	显示设置
确认设定	
	完成工具设定

在这个层次上的抽象不仅忽略了实际用户界面的具体细节，也忽略了任务本身的一些内容。例如，可能有很多独立参数需要采用各种表格形式输入，并且有很多约束条件。“给用户显示设置”这个信息中可能隐含复杂的，不可缺少的用户交互动作。尽管如此，这种“适当的模糊”在建模中非常有用。它可以避免脱离主题，过早地陷入到细节。在许多情况下，模糊的基础信息也蕴涵着可能的解决方案。例如，输入参数是一种方法，尽管可能的输入方式非常复杂。实际上，这种“适当的模糊”引导一个小组实施的解决方案足以让他们的法律顾问确定他们为软件革新专利的拥有者。

有两个简单而重要的原因使我们在本质用例中忽略在一般用例中存在的大量词汇。一是，语句和结构会引入大量无意义的单词，仅仅是降低信噪比，对分析和设计人员，常常使得提出对用户界面设计有用的信息变得困难。第二，实际存在但不必要的精细而冗余的结构，如果真正变成设计单元，会使用户界面不必要地变得复杂。

一个简单而又说明问题的实例是（例 2）的回收机的用例，其中描述的第一个用户步骤就是没有必要的。按开始按钮对用户来说没有意义。将回收物品放入机器中就足够了。实际上，将用例描述缩减为本质用例的形式，可以引导设计者大大缩减用户的动作。由于在先前的设计中，太多不必要的步骤通过用例带入到交互设计中，某个为工业自动化设计的软件工具，采用本质用例进行重新设计后，完成一个编程任务所需的步骤缩减了一半。实际上，本质用例在工作流重组和过程重组中也被证明为是有效的工具。因为它

们揭示了任务的核心特征。

任务目标和用户意图

本质用例的描述分成用户意图和系统响应。在本文的前一部分，我们在用例中引用了面向目标方法中的内容，但在用户界面设计中，目标与意图的区别虽然微妙但非常重要。一个目标是一个系统期望的结束状态，它采用静态术语描述该状态和作用对象的特征。例如，在宾馆预订应用中，宾馆雇员的一个目标可以描述为“顾客登记到可用房间”。与之对照，意图是动态的，表示方向或趋势，而不是一个结束状态。目标是目的地，意图是旅程，在这里用户的旅程——交互过程——很大程度上依赖于界面和交互过程的设计。目标是静态的，注意力放在宾语或名词上，而意图是动态的，重点放在行为和动词上，暗中允许交互被分解或转换。一个意图可能实现一个目标，也可能很多意图最终为实现一个目标。

明确地讲，目标与意图都可以认为是目标为中心的用例建模和用户界面设计的一部分。可以超出用户的直接意图，在整个项目的范围内考虑应用的目标。在许多应用中，独立的用例，是任务结构的一部分，就象是用户在系统中扮演一个角色时所具有的工作、职业和任务职责。本质用例的目标和功能是用例的一部分，并且在其中表现出来。

在本质用例模型中，目标和子目标的整个树状结构，如同任务与子任务，通过用例间的交互联系表现出来，这些交互联系要么存在于用例描述中，要么在与目的相关的部分单独列出。

结构化的本质用例

许多模型驱动的设计过程中，模型的质量和特征决定结果的质量和特征。因为这个原因，我们开发了一种高度结构化的用例，主要根据我们自己的工作和我们客户在以使用为中心的设计实践，但也结合其它一些想法，包括前面提到的一些注释形式。这种结构是用于组织用例的不同部分，建立一个有序的易于理解的框架。并且确保用例的相关部分都被定义或至少被考虑到。

与那些非规范化的示例相比，结构化本质用例是连接需求分析和用户界面设计间以及设计及实现间坚固的桥梁。作为管理复杂事物的一个工具，在处理真正的大型问题时，也提高了本质用例的有效性。在处理较小而相对简单的问题，并且由一个小规模但联系紧密的团队设计和实现时，不规范或相对非结构化的本质用例确实适用。但是，随着项目中用例数量的增加，就需要更加系统化、复杂化和结构化的建模和沟通。

没有一个固定的定律告诉我们何时选择非规范用例，何时采用结构化用例，这个问题取决于项目如何组织实施，以及要解决的问题的规模。但是，通常情况下，在多于一个的用例时，采用非规范用例建模就开始出现问题。另外，任务分析人员、设计人员、开发人员间沟通主要采用模型的情况下，需要更为规范的形式。图 2 揭示了结构化基础用例的组织结构，用例被分为三个主要部分：标识、关系和过程。标识部分给用例一个唯一的标记并解释其目标及功能。联系部分标出与本用例相关的其它用例，过程是用例的描述体，定义交互过程的动态部分。以上部分将分别讨论。

Identification	
ID	Name
Contextual Purpose	Supported Roles
Relationships	
Specializes	Extends
Resembles	Equivalent
Process	
Preconditions	
User Intentions	System Responsibilities
Asynchronous Extensions	Asynchronous Extensions
(steps)	(steps)
Post-conditions	
Business Rules	

图 2 本质用例框架结构示意图

标识与目的

标识符仅仅是一系列数字或其它永久的唯一标识码。在小问题中可能不需要这种代码，但大型项目需要他们帮助跟踪用例和相关的模型与文档。

本质用例的名称反映其功能或直接目标。直接目标表示用户单一、独立的意图。对意图与目标进行区别会引出描述和定义用例上的差别。

我们已发现及物动词的现在分词——表示对直接宾语的连续动作，可以最好地描述用户的意图。例如，从 ATM 中取现金或为到达顾客登记房间。采用形容词限定直接宾语同样重要，这样可以与应用中其它类似或相关的对象进行区别。通常，当问题的粒度增大，增加对宾语的限制会更安全。特别要避免的是，同一个用例名称可以被理解为几个不同的意图。例如，删除块在同时包括删除程序块（在编辑程序时）和删除执行时间标记块（在代码调试时）的应用中就不是一个可接收的名称。适当的模糊与危险的二义性的界限依赖于手中的特定问题。

不确定命名有一个有用的变体，我们称之为案例族，其名称表示一组相关用例的集合。例如，下载模型/块/程序到模拟器/控制器需要 6 个独立的用例描述，也可以由一个简单的定义合适地概括。案例族可能指明针对不同对象的同一操作，或是一个对象上的多个操作。例如，输入/修改父母背景信息。案例族在具有大量用例的应用建模早期或针对那些时间非常有限的项目特别有用。

通过试验不同的用例名称拼写方法，我们现在倾向于采用带下划线的小写字母，单词间用空格隔开。下划线使得从整个描述中很容易找到用例名称。将单词分开书写便于编制用例索引及方便查询。用户也发现这种格式比先前采用大写，没有分隔的式样要友好。在联机文档中，下划线也可以作为超链接直接指向该用例的定义。

关联目标部分包括对大目标的描述或该用例的相关关系。例如，在宾馆预订系统中，浏览空房间的关联目标可能是：“为了完成顾客注册，必须找到一个合适的房间分配给顾客。”

相关角色部分列出与本用例相关的角色，进一步限定该用例的本质特征。例如，浏览空房间的相关角色包括：接待员、接待主管、自注册顾客。

关系

关系部分标明与该用例相关的其它用例，在以使用为中心的设计中，用例间可能有如下关系：

包含——某个用例包含或使用其它用例。

特殊化——某个用例是另一个普遍用例的特殊变体。

扩展——某个用例通过引用可变过程或异常处理扩展另一个用例。

另外，我们发现其它，在某些场合有用的用例。

相似 —— 一个用例在某些不确定情况下与其它用例类似。

相同 —— 一个用例等同于另一个，可以作为替代品。

相似关系，这种关系经常在用例建模的开始阶段使用，当关系的实质还不清楚时，它提供了一个发现用例间关系的方法。例如，查看存款帐户和查看支票帐户可能是独立的两个用例，在搞清楚是否某个过程可以同时适用于两种情况或是需要两个完全不同的方法之前，可以将它们标记为相似。

相同关系，对那些从用户角度看属于完全不同的意图，但可以用同一个定义概括的用例可以采用相同关系标记。这个结构可以使这种情况下，采用一种设计满足不同用户的需要变得容易。如果没有这种关系，建模者就会被迫为仅有一点微不足道的差别定义两个用例。

在定义部分，关联的用例列在一起，标记上“特殊化”、“扩展”、“组合”和“相同”。这里包含关系被忽略，因为在使用例名称或描述中包括对包含用例的引用。

例如，通过目录浏览空房间可能包括如下关系：

扩展：登记顾客到房间，查看房间使用情况

特殊化：浏览房间

关系中的关联反映在用例图中，将在本章后面讨论。

过程

用例的过程部分由前提开始，结束状态终止。这两种状态可以包括与其它用例明确的或不明确的关系。用例的描述体分为用户意图和系统响应，虽然在例 5、例 6 中我们采用两列进行描述，但这种方法有时浪费显示空间并且显示困难。特别对有复杂步骤的用例。另一种方式是采用不同的字体和印刷体，区分系统响应与用户意图。

例 11

用户意图

系统响应

- 1、显示标准测试设置
- 2、选择测试设置

- 3、显示测试设置
- 4、可选（改变测试设置）
- 5、确认测试设置
- 6、测试并提供报告
- 7、可选（打印测试报告）
- 8、打印并确认

虽然看上去不如分开两列，但这种方式采用文字编辑器很容易实现。

结构化精炼描述中的元素式样

在描述用户意图和系统响应的过程中，首选采用自然语言和用户所在应用领域常用的词汇。由于本质用例是用户需求和用户界面设计、设计和应用间起到桥梁作用的关键模型，所以要求必须提供一定的技术结构和说明。这些技术元素包括对象、其它用例和表示传统交互作用、序列、扩展的惯用结构。

对象

理想情况下，不论是具体模式还是实质模式，用例模型都是依次开发的对象模型或其它形式的数据模型，如图 1 所示(cf. [Nunes 和 Cunha,2000])。

用例描述的步骤可以明确或隐含地针对对象、它们的方法及属性。例如，“测试结果”可以是定义的术语或类对象，打印是它的一个方法。可以使用斜体或引号来强调被定义为术语的对象或数据。一个用例的整个过程可以对应于对象的一个方法，如同在包含改变测试设置的用例中那样。

被包括的用例

描述过程可以包括明确引用其它过程或交互过程一部分的用例。如“例 11”中第 4 步，用例名称在“做”之前，该用例启动其它用例，这样书写提高了可读性，特别是对于最终用户阅读或审查。

条件交互

条件交互作用以“可选”表示，后面的括号中列出可选动作。如上面例 11 中的第 4 步，内容可以是

条件性的，条件交互将在下面与扩展一起讨论。

部分序列

许多传统的用例描述方法的一个缺陷是，他们不能提供明确的方法来表示任务模型中非常普遍的一种情况。在大部分已出版的叙述中，包括所附带的例子，都是完全排序的，顺序间的制约作用非常强：首先用户做什么，然后系统会如何，而后用户再做什么等等。在这种叙述中，有一些动作可能是可选的，但序列表现的却不是这样。

虽然有些分析员通过重申和采用条件表达式的方法进行了程序化的补充，序列的可选性和灵活性问题很少被模型化。对这个问题的忽视是一个很大的缺陷，因为许多任务可以采用多种顺序完成，并且允许用户通过不同的动作序列完成任务是影响可用性的一个重要因素。以前，许多类似的情况被描述成有严格序列要求的，而实际上可以看出作用过程实际上是灵活的。可以相信，在一般粒度的问题中，尽管用例中没有说明，最终的可视化交互设计可以允许需要的灵活性。但是，在大规模的项目中，分析与设计间隐含的重要信息丢失的可能性大大增加。这些信息应该存在于用例叙述中的固定位置。

在我们的惯例中，短语“按任意顺序”，后面跟随一系列行为或目的，用逗号分开并放置在大括号内。如：

例12

“按任意顺序”{标明部件号；标明数量}

这个惯例鼓励用户界面设计者仅当序列必须固定时才采用严格的顺序进行描述。这种标记法已经证明对于用户和非技术人员是容易的和自然的，不需要培训就可以掌握。这种标记法被设计成一种在记者中流行的速记形式，它一目了然，足以省略“按任意顺序”。

扩展

“扩展”标记自从用例诞生的那一天起就成为用例模型中的一个核心概念[Jacobson 等., 1992]。扩展是一个截然不同的用例，包含表示变化、条件或异常交互的过程，这种交互过程包括在基本情况的主“事件流”中，在某些情况下会改变或扩展。基本情况是指一般期望的交互过程；扩展是一个例外或变化。

扩展的主要优点是它使本质用例的描述过程简单化，通过直接分为几种特殊或不常见的用例过程，保持用例叙述过程的清晰，使得在正常条件下，用户操作界面简洁明了。另外一个明显优势在于，由于扩展

可以修改或扩展多个其它用例，扩展在模型中以及设计与实现中支持复用。

但是，扩展为方法论者和实践者带来许多问题。有关结构的一个尴尬问题是，扩展指向被它扩展的基础用例，而不是人们希望的另一方向。换句话说，在基础用例中，扩展用例不可见，在图中，箭头指向所扩展的所有用例。虽然这种惯例对许多人来讲是个缺陷，但在另一方面也是个优势。扩展允许发现和修改，而不必回到基础用例中对描述进行修改。这样，最初可以只考虑正常的交互情况，可以稍后考虑例外处理。

虽然连一些面向对象的领导者和用例分析员陷入对扩展确切语法的争论，我们已经采用了一种完全实际的方法。因为它们简化了任务模型，由此可以产生简化的用户交互过程，以使用为中心的设计直接使用扩展，不管有多少语法的不精确或是理论上的缺陷。

然而，随着设计规模的扩大，会出现一些问题。随着设计词汇和用例的增加，在中等规模的问题中，有益于快速设计的“有意的粗略”变成了一种障碍。所以，最近我们重新考虑了扩展的定义以及它们如何在用例和用例图中使用。其中一个问题是，扩展最初的定义实际上是指特殊或可替换的两种不同程序的合并。在第一种情况下，扩展用例只可能在使用过程的某些特定情况下使用，如例 11 所示，用户可以选择或不选择修改测试设置。很显然，这不属于正常或主要程序部分，将它看作运行一个标准测试的独立用例应该更准确。但是，在选择已定义过的测试之后和确认测试开始之前，用户可以执行扩展用例，更改测试参数。

同步扩展

第一种“扩展”定义为“同步”是因为它与被扩展的用例同步。实际上，同步扩展就是条件包含。实际上，是在某种情况下，被交互过程包含或在特定位置使用的用例。在定义时，它必须在基础用例中可见，因为描述中必须说明在什么位置发生扩展。

异步扩展

最初的扩展定义包含的另一种情况是指在可变的或异常情况下，无法知道扩展发生的具体位置。这种异步扩展的操作好像是对主交互过程的中断。例如，例 13 中的用例，用户可能在任何时候重新启动系统，取消测试并且从新加载初始设定。同样地，系统应该在任何时间作出反映并提示用户发生一个无法恢复的例外。因为这种扩展在交互的整个过程都可能发生，它们被合理地分离并出现在描述体的顶端，例如：

例13

用户意图

系统响应

异步扩展

可选，在任意地方发生{重启动测试}

可选，在任意地方发生{报告灾难性错误}

- 1、 显示标准测试设置
- 2、 **选择测试设置**
- 3、 显示测试设置
- 4、 **可选（改变测试设置）**
- 5、 **确认测试设置**
- 6、 测试并提供报告
- 7、 **可选（打印测试报告）**
- 8、 打印并确认

与其他人一样，Cockburn [2000]，将各种扩展放在用例描述的结尾，但我们发现应将异步扩展作为序言放在用户、管理者及非技术人员期望发现它们的地方。就好比说，“在我们开始讲故事之前，请注意我们可能被突发事件随时打断。”在对非技术人员展示用例时，我们应该省略用例中“异步扩展”的标题。

用例图

用例图模型化用例间的关系，表现正在设计的系统所支持的所有任务。正如前面所说的，UML的Use Case Diagram中的问题使我们不能简单地完全采用它。真实的应用项目中，典型的大型复杂模型在用UML表示时，阅读和理解都十分困难。将执行者（用户角色）与用例关系集成在一个模型中，导致令人困惑的大量连线仅仅为了包含所有微不足道的方面，这种情况可以在书、文章以及教程中发现。一个合适的CASE工具可以允许选择执行者或用户角色并且重点查看相关用例，但在文章或静态图中，我们应该忽略它们。

表现用例

依据Jacobson，我们一直使用一个椭圆表示一个用例，但最近我们发现，这种惯用方法在学术著作或教科书中用例名称短而且简单时还够用，而在许多大规模复杂的实际问题中就不再适用。具体来讲，椭圆形对于长的名称就不如其它几何形状。其它两种仍在使用的传统用法值得考虑用来替代椭圆形。一种是将名称放在用例符号外，就像Jacobson最初做的那样。另一种是用一种可以被称为“椭圆角”的复合图形

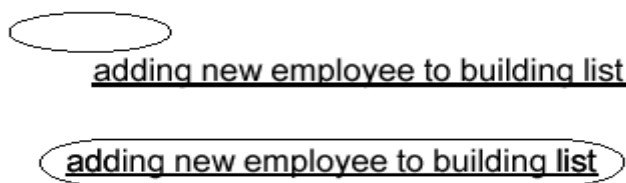


图3 表示用例的可替代图形

前面谈到用例族，在图中表示成图形族，从图中可以看出实际包含的一系列用例。

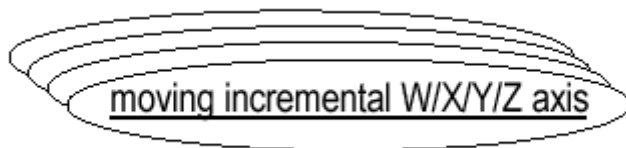


图4 用例族在用例图中的表示方法

表示关系

由于在 UML 中由一条线表示的关系类型不是很清楚，必须沿着一条线到底去寻找最近的标注来弄清楚两个用例间的关系。UML 中关系标注（《and》）简直就是一种图形噪音，使图形看上去更难解析清楚。

随着经验的积累，特别是大规模复杂用例的应用，我们不断修改以使用为中心设计的标注来提高准确性，使交流变得更容易。为了提高在复杂模型中的可阅读性，我们使用不同类型的线来代表不同的相互关系，对复杂的交流使用标签（如图 5 和图 6）。

对图示来说，如何区别同步扩展和异步扩展是一个具有挑战性问题。“扩展”关系仅为异步扩展保留。考虑到异步扩展存在于对基础用例扩展的用例族中，箭头方向仍然由扩展指向被扩展的基础用例。

因为同步扩展仅仅是条件包含，在图形上它们应表现为包含的某种变体。我们倾向于采用一种更为自然的表现方法模型化这种关系，在箭头尾部增加表示包含的菱形，这种在结构图示法中准确标注关系的方法已经在全世界使用 25 年了[Yourdon 和 Constantine, 1975]。不幸的是，UML 乱用了菱形标记，使之代表聚合。这种采用不填充的菱形代表聚合，填充的菱形代表类别聚合（对大多数用户来说感到困惑）使得这个符号变得意义含糊、不具有指导性[Henderson-Sellers 和 Barbier,1999]。

对于条件包含，考虑到其它提议，包括在箭头的尾部加上问号注释或集合注释，如 0..1.我们现在采用带有标注的实线。

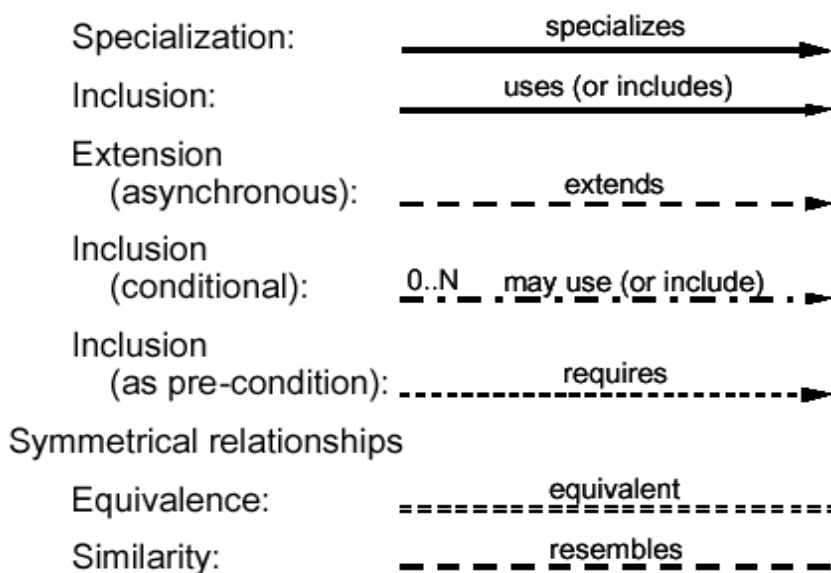


图5 用例图中的关系

图 6 所示例子为使用这些标注的用例图。

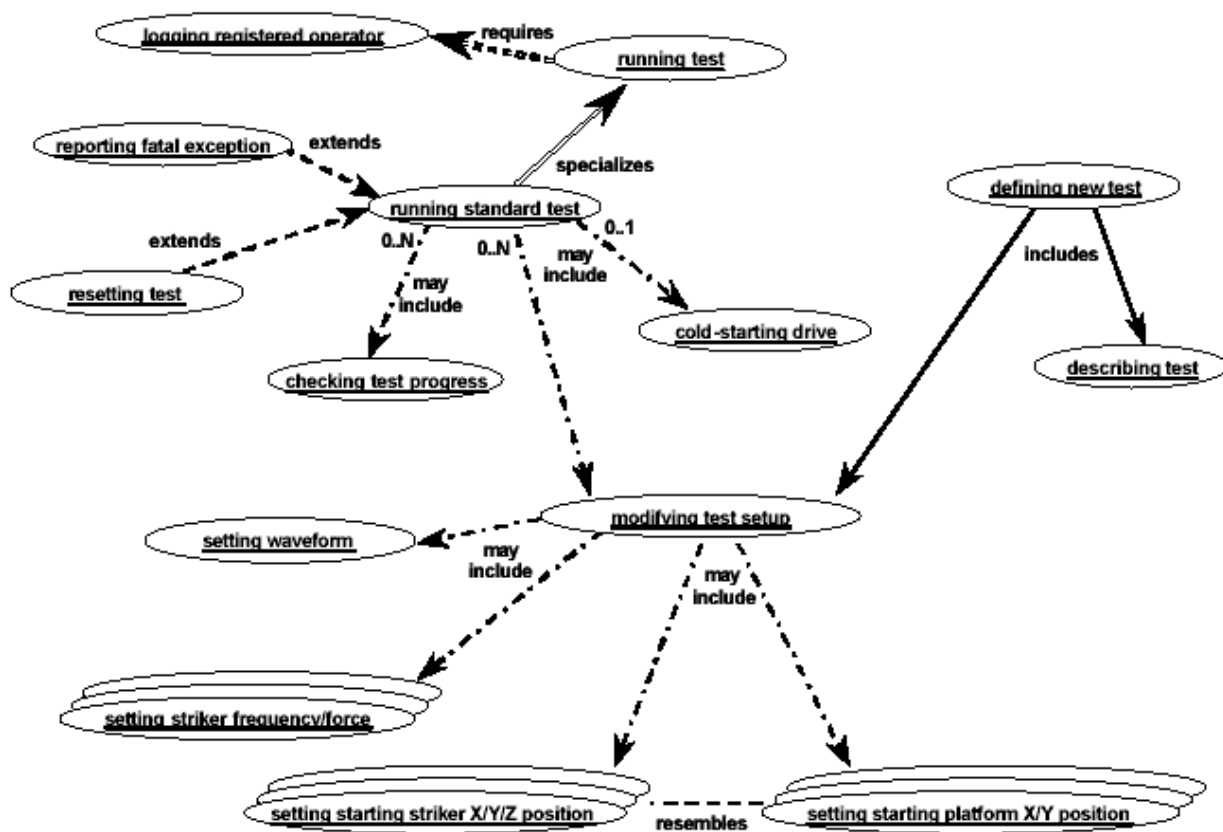


图6 用例图举例

业务规则

考虑到功能上的要求，几乎所有的应用都部分定义了通常所说的“业务规则”。业务规则——或者简单讲就叫规则——就是指可以满足所属业务流程的政策、条款或条件[Gottesdiener,1999]。这些规则定义了操作关系或控制信息以及规定在一个系统（有些情况下是在更大一些的组织中）如何操作。虽然业务规则在用例及面向对象的学术及理论工作方面还没有突出表现，但它在实际应用中起着非常关键的作用。因为他们描述了业务过程将如何进行，这些术语虽然在业务词汇中不常使用，却在技术领域用到。举例说明，一个规则可能规定在一个化学品程序中不同容器被清空，然后装入其它成分的顺序。或者有一个规则要求密码必须包含一系列数字及字符并且必须至少每 90 天更换一次。

虽然业务规则本身不是功能要求，它们通常在用例中与功能有紧密的联系。规则可以应用于用例，或与用例的一部分或整体相关。很明显，规则与用例的联系对用户界面设计有直接及重要的影响。举例说明，要求更新密码的规则，就有通知用户何时需要更新密码的功能并且可以在登录后提供修改密码的功能。

为确保准确的设计及实施，并更容易满足需求，业务规则应该与采用的分析与设计模型有特别准确和直接的联系。另一方面，业务规则应包含一系列不同的需求，这些需求如何分散在不同的分析及设计模型中就会被隐藏、丢失或者重复。为解决这个复杂问题，在“业务规则模型”中将规则分别列出。业务规则可以通过用例注释或用例图上通过名称和/或标识符对规则索引。

举例说明，一个业务规则要求在同一时期对于没有余额的账户不能批准贷款限额的增加。尽管从理论上讲，这两个功能不需要顺序上的相互关联。用例查阅账户余额，由此成为用例所授贷款限额增加的前提条件。例如这样一种注释“规则#27，动态审查贷款增加”是与两个用例间的关系相联系，在用例图中由关系连线表示，在前提条件中列出的格式是：

审查用户账户余额（规则#27 动态审查贷款的增长）

对那些应用于整个用例的业务规则，在过程中应保留一条说明。严格来讲，如果相关的业务规则没有包含或者以某种方法与用例模型相联接，那么这个用例模型作为说明材料就是不完整的。

建议

大量的经验证明，对于界面设计，一个合适的用例描述格式可以促进有效的设计过程，简化项目内部的交流，从而使设计质量得到相应的提高。人们已认识到针对用户界面设计的任务建模，简明而目的明确的用户例最具优势。我们建议采用高度结构化和系统化的用例描述任务模型，这对于大规模及复杂项目尤其重要。结构化本质用例所提供的丰富而准确的表达模型不但满足用户界面设计师和软件工程师的要求，而且也容易得到最终用户的理解与认可。这种经过精良准确组织的用例提高了模型化程度和易交流性。它包含以下内容：（1）将用户意图与系统响应明确分开，使系统边界更加明确。（2）将同步扩展和异步扩展明确分开表示。（3）采用表达部分序列过程的术语。（4）直接与相关的业务规则相联系。这种结构一方面避免了连续描述中的不规范与不准确，另一方面，避免了不正确的模型的使用，如内部行为图更适合于面向对象设计及程序设计而不是用户界面设计。

参考文献

Ahlqvist, S. (1996a) "Objectory for GUI Intensive Systems." Kista, Sweden: Objectory Software AB.

Ahlqvist, S. (1996b) "Objectory for GUI Intensive Systems: Extension." Kista, Sweden: Objectory Software AB.

- Bjork, R. C. (1998) "Use Cases for Example ATM System.",
http://www.cs.gordonc.edu/local/courses/cs320/ATM_Example/UseCases.html
- Booch, G. "Scenarios," (1994) *Report on Object Analysis and Design*. 1 (3): 3-6.
- Carroll J. M. (ed.) (1995) **Scenario-Based Design**. NY: Wiley, 1995.
- Cockburn, A. (1997) "Structuring Use Cases with Goals," *Journal of Object-Oriented Programming*, Sep/Oct, 1997, pp. 35-40, and Nov/Dec, 1997, pp. 56-62.
- Cockburn, A. (2000) **Writing Effective Use Cases**. Boston: Addison-Wesley.
- Constantine, L. L., & Lockwood, L. A. D. (1999) **Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design**. Boston: Addison-Wesley.
- Constantine, L. L. (1995) "Essential Modeling: Use Cases for User Interfaces," *ACM Interactions*, 2 (2): 34-46, April.
- Constantine, L. L. (1994a) "Mirror, Mirror," *Software Development*, 2 (3), March 1994. Reprinted in Constantine, L. L. **Constantine on Peopleware**. Englewood Cliffs: Prentice Hall, 1995.
- Constantine, L. L. (1994b) "Essentially Speaking," *Software Development*, 2 (11): 95-96, November 1994. Reprinted in Constantine, L. L. **Constantine on Peopleware**. Englewood Cliffs: Prentice Hall, 1995.
- Constantine, L. L., and Henderson-Sellers, B. (1995a) "Notation Matters. Part 1: Framing the Issues," *Report on Object Analysis and Design*, 2 (3):25-29, September-October 1995.
- Constantine, L. L., and Henderson-Sellers, B. (1995b) "Notation Matters. Part 2: Applying the Principles," *Report on Object Analysis and Design*, 2 (4):25-27, November-December 1995
- Fowler, M. (1997) **UML Distilled: Applying the Standard Object Modeling Language**. Reading, MA: Addison-Wesley.
- Gottesdiener, E. (1999) "Business Rules as Requirements." *Software Development*, 7 (12), December.
- Graham, I.(1996) "Task Scripts, Use Cases and Scenarios in Object-Oriented Analysis," *Object-Oriented Systems* 3 (3): 123-142.
- Henderson-Sellers, B. (1999) "OML: Proposals to Enhance UML," in J. Bézivin and P.-A. Muller (eds.) **The Unified Modeling Language: Beyond the Notation**. Berlin: Springer-Verlag.
- Henderson-Sellers, B., and Barbier, F. (1999) "Black and White Diamonds." in R. France and B. Rumpe (eds.) **The Unified Modeling Language: Beyond the Standard**. Berlin: Springer-Verlag. *Inf. Software Technol.*, 41 (2): 139-156.
- Henderson-Sellers, B., and Firesmith, D. G. (1999) "Comparing OPEN and UML." *Inf. Software Technol.*, 41 (2): 139-156.
- Jacobson, I., Booch, G., Rumbaugh, J. (1999) **The Unified Software Development Process**. Reading, MA: Addison-Wesley.
- Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1992) **Object-Oriented Software Engineering: A Use Case Driven Approach**. Reading, MA: Addison-Wesley.
- Korson, T. (1998) "The Misuse of Use Cases," *Object Magazine*, 8 (3): 18-20.
- Kruchten, P. (1999) **The Rational Unified Process: An Introduction**. Reading, MA: Addison-Wesley.

- Lee, J., and Xue, N. (1999) "Analyzing User Requirements by Use Cases: A Goal-Driven Approach," *IEEE Software*, 16 (4): 92-101, July/August.
- McMenamin, S. M., & Palmer, J. (1984) *Essential Systems Analysis*. Englewood, Cliffs, NJ: Prentice Hall.
- Nunes, N., and Cunha, J. (2000) "WISDOM - Whitewater Interactive System Development with Object Models," this volume.
- Page-Jones, M., Constantine, L. L., and Weiss, S. (1990) "Modeling Object-Oriented Systems: A Uniform Object Notation." *Computer Language*, 7 (10), October.
- Roberts, D., Berry, D., Isensee, S., and Mullaly, J. (1998) *Designing for the User with OVID*. New York: Macmillan.
- Rosson, M. B., and Carroll, J. (2000) "Scenarios, Objects, and Points-of-View in User Interface Design," this volume.
- Rumbaugh, J., Jacobson, I., and Booch, E. (1999) *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley.
- Schneider, G., and Winters, J. P. (1998) *Applying Use Cases: A Practical Guide*. Reading, MA: Addison-Wesley.
- Texel, P. P., and Williams, C. B. (1997) *Use Cases Combined with Booch OMT UML*. Upper Saddle River, NJ: Prentice Hall.
- Wirfs-Brock, R. (1993) "Designing Scenarios: Making the Case for a Use Case Framework," *Smalltalk Report*, November-December.
- Wirfs-Brock, R. (1994) "The Art of Designing Meaningful Conversations," *Smalltalk Report*, February.
- Yourdon, E., and Constantine, L. L. (1975) *Structured Design (First Edition)*. New York: Yourdon Press.

词汇表

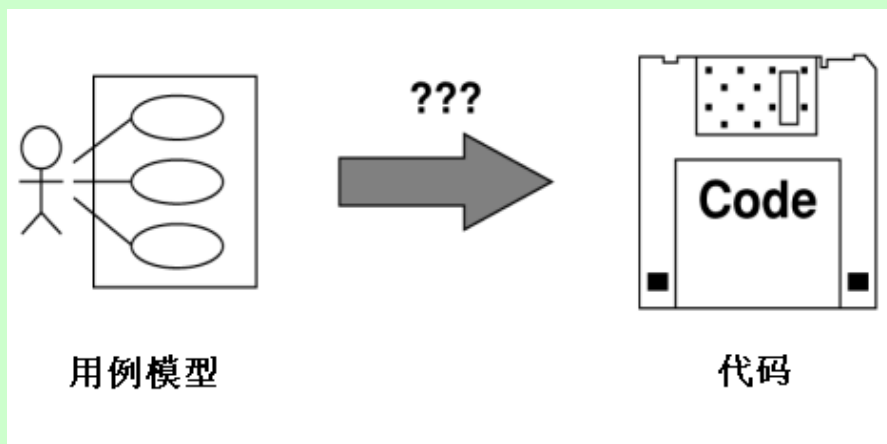
- 1、Use Case 用例，在有些文献中为“用况”，但 Rational 的中文版 RUP 中采用“用例”
- 2、Essential Use Case 本质用例。不采用《统一开发过程》（周伯生等译，机械工业出版社）中的翻译方法（p124），该书翻译为基本用例，容易引起混淆。
- 3、Concrete Use Case 具体用例。
- 4、User Interface Design 用户界面设计
- 5、Notation 图示法
- 6、actor 在本文翻译为“执行者”，但在有的著作中被翻译为角色

(c) Copyright 2000, L. L. Constantine and L. A. D. Lockwood, all world rights reserved. Translated with the authors' permission. Reprinted from M.van Harmelen, ed., *Object-Modeling and User Interface Design: Designing Interactive Systems* (Addison-Wesley, 2001; ISBN: 0201657899). Additional information and materials on use cases and usage-centered design can be obtained on the Web at <http://foruse.com/>.

UMLChina保留中译本一切权利

UMLChina 培训

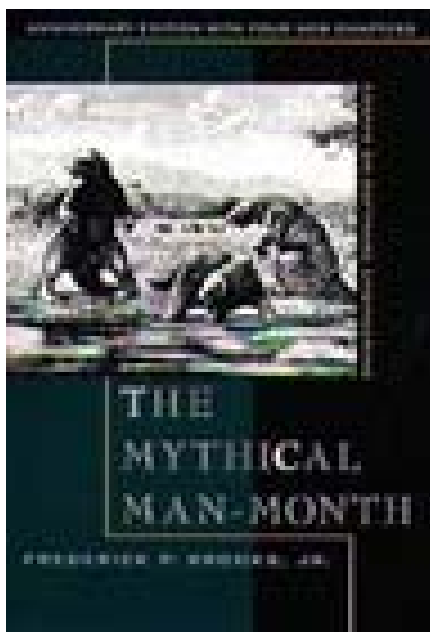
The real thing



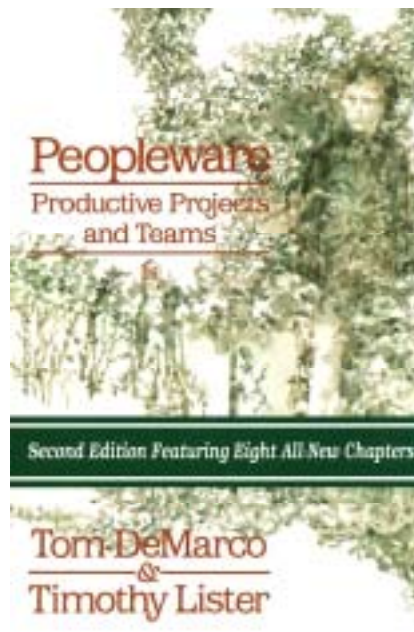
利用 UML 的 20%就可以为 80%的问题建模
-- 《UML 用户指南》，第 32 章

详情请垂询：think@umlchina.com

中文译本即将发行！



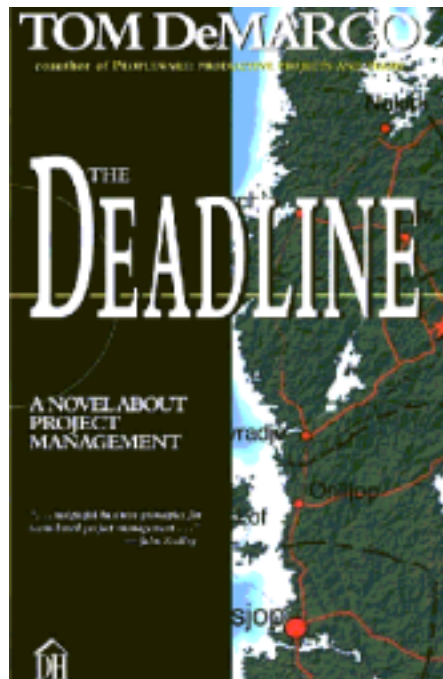
《人月神话》20 周年纪念版，Fred Brooks
翻译：UMLChina 翻译组 Adams Wang
散文笔法，绝无说教，大量经验融入其中



《人件》第 2 版，Tom Demarco 和 Tim Lister
翻译：UMLChina 翻译组方春旭、叶向群
微软的经理们很可能都读过—amazon.com



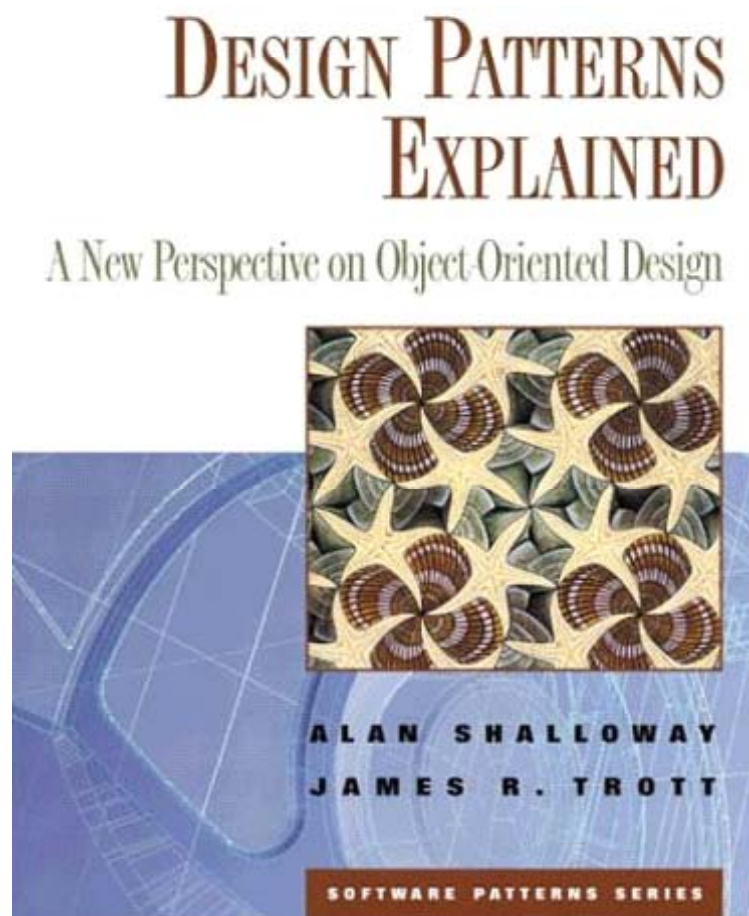
《面向对象项目求生法则》，Alistair Cockburn
翻译：UMLChina 翻译组乐林峰
Cockburn 一向通俗，本书包括十几个项目的案例



《最后期限》，Tom Demarco
翻译：UMLChina 翻译组透明
这本最通俗，这是一本软件开发小说

看不懂《设计模式》？

它的作者推荐《设计模式精解》----中译本即将上市！透明 译



直接学习《设计模式》这本书是非常困难的。作为入门，我推荐 Alan Shalloway 的《设计模式精解》。

--John Vlissides 写于 [UMLChina 答疑板](#)

网上商店的模式

Eduardo B. Fernandez 著, [Huang Yin](#) 译

摘要

网上购物已经变得很普遍,许多网站都为此提供了方便的用户界面。为了支持不同种类的导航视图,网上商店需要良好的基础结构。目录模式和购买流程模式是网上商店基础结构的一部分:目录模式描述了如何组织网上商店的商品信息,购买流程模式描述了在网上购买商品所必需的步骤。我们还将展示在网上商店中如何结合应用这两种模式。

介绍

网上商店之所以变得如此普遍,是因为它们给予商家和客户更多的机会。在这一类应用中可以发现很多模式,我们将其分为三类:

- 用于导航视图的模式
- 用于构筑界面的模式
- 用于基础结构的模式

前两类模式在附录参考书中有介绍[Lya98, Lya99, Pal, Ros00]。好的基础结构可以提高服务器的响应速度,使导航视图和界面具有较好的可扩展性和灵活性。这种类型的模式应该是清晰、灵活、有效、可扩展的,并且独立于另两种类型。

在本文中,我们将介绍建立网上商店所必需的两种基础结构模式:目录模式和购买流程模式。目录模式用一种有效灵活的方式来组织商品信息,并为客户决策提供帮助机制。购买流程模式的目标是简化并提供一个方便有效的购买流程。

这些模式都是语义分析模式(Saps)的范例[Fer00a],一个简单的语义单位可以应用于不同的情况。它们也是一种合成模式:它们的组件在其中有自己的值。这两种模式的结合可以用于定义一个网上商店的

框架。

目录模式

意图

目录模式组织网上商店所出售的商品信息

环境

网上商店是公司出售各种商品的地方。在服务器上还有用于提供流程结构、权限表、库存以及其他相关功能的应用。

问题

网上商店出售不同的商品，而这些商品有时是根本没有关联的，例如书和食物。一个重要的问题是：如何组织商品的信息，为客户提供一个友好的网上向导，从而提高网站对客户的吸引力？

约束

- 有大量不同种类的商品供销售
- 商品及其描述时常变化
- 如果查询和选择商品不方便，客户将不会再访问网站，基础结构应该支持这些不同的功能
- 应该将相关的商品介绍给客户来吸引其购买
- 目录的制作通常使用特别的样式，并且不能重复使用，这导致了时间和精力的重复和浪费。

解决方案

定义一个目录类作为收集商品的中心点。将不同关注点的特征，例如商品的细节描述，和相关的商品分成单独的类。利用观察者（Observer）模式来发现改变并关注客户的变化。

类图

目录是一类商品的集合。每个商品至少归属于某个目录。**商品**类定义了一种供销售的商品，购买者购买的是一种商品，而不是商品的个体。这个类包括了每个商品的基本属性，特别是代表某种特性的状态属

性，例如，一个新的商品。导航视图可能将特殊的商品放在屏幕上的显著位置，以便与一般商品相区分，从而让客户了解[Lya98]。

ProductInfo 类提供了更多的商品细节信息。其中可能包括同一种商品的不同类别、不同品牌的比较，或是提供一个最好的性价比。例如对电视机，可能就要提供几个不同品牌的比较信息。作为旅游中介，可能就要给客户id提供附加的选择，例如让客户能够选择较近的机场，这样可以使飞机票更为便宜。**SimilarProduct** 类则提供相似商品的链接。

通过电子邮件的方式将商品（信息）的修改通知客户，让他们能了解到新的或是感兴趣的商品（信息）。**ProductObserve** 类用于检测变化并通知客户。注意这些类都是观察者（Observer）模式的一种类型。**Notification** 类保证将修改的记录通知客户。

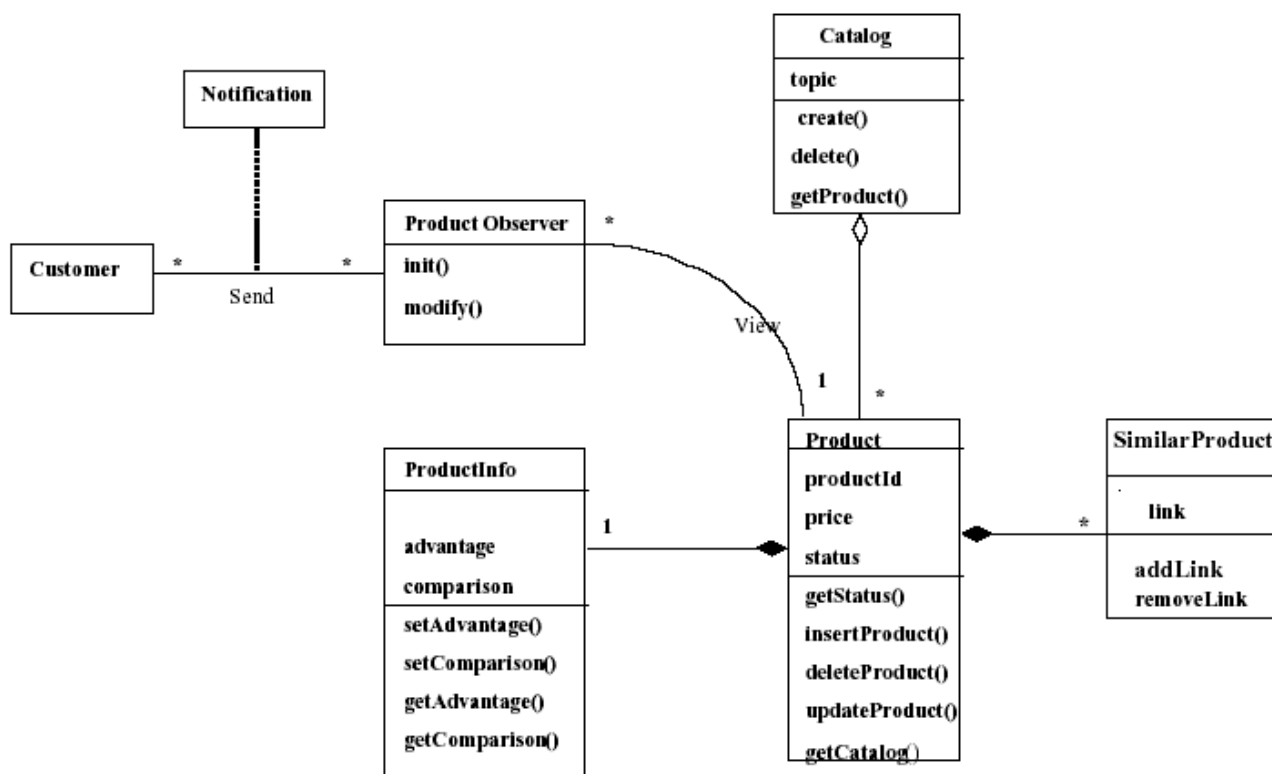


图 1.目录模式的类图

动态描述

图 2 中的序列图显示了当商品信息修改时触发的协同操作。这与标准的观察者（Observer）模式序列相似。

结论

这种模式有以下好处：

- 它提供了一种描述商品方便、有用的基础结构。导航类可以加入其中，利用有吸引力的方式显示商品信息。
- 它可以复用。这种模式适用于各种不同的网上商店，不管是大商场还是个人商店。
- 它可以容易地和其他模式结合使用，例如个性化模式[Pal, Ros01]，我们下面要介绍的购买流程模式，或是库存模式[Fer00b]。

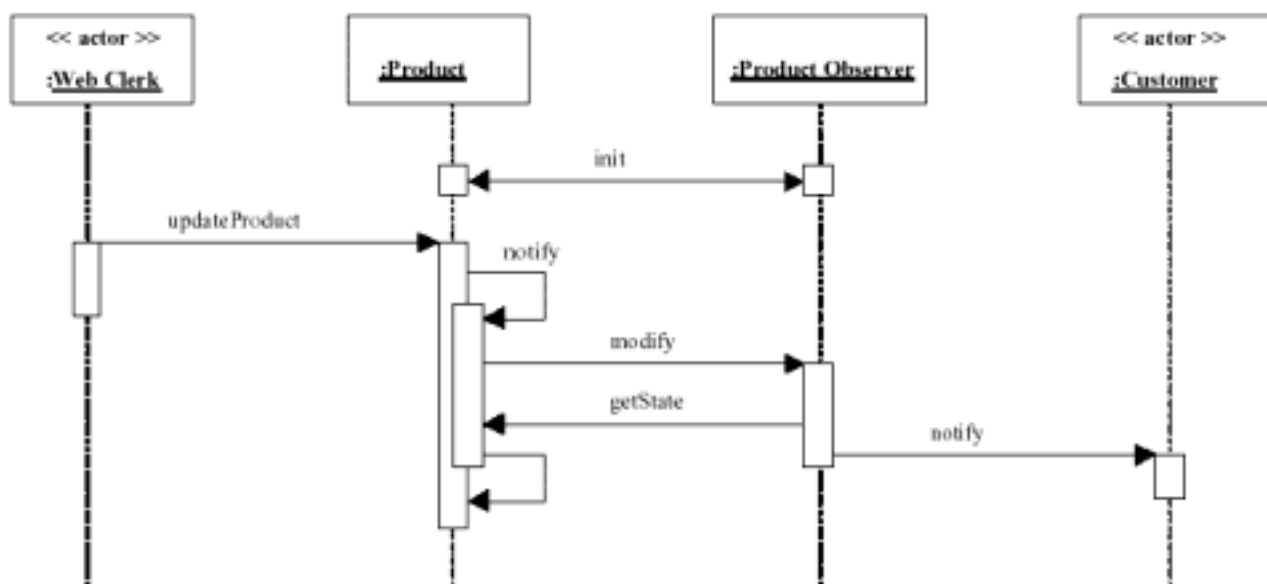


图 2.修改商品（信息）的序列图

其他结论包括：虽然基本模式涉及商品类型（的概念），但某些站点只是出售一些个别的商品，例如旧车网站、拍卖网站。既然 Product 类作为个体商品的集合，那么我们需要一个附加类来指出商品的类型。

已知的用途

该模式可以适用于很多应用，例如：

- 网上书店。每一个网上书店都利用目录来组织书籍，如计算机、医学、历史和文学。商品就是书。商品信息包括每一本书的评论及描述。状态属性指出特殊的交易情况，新书或是畅销书。如果一些客户对某一类的数据感兴趣，不管什么时候，只要新到这类书或是特殊的交易情况

有效，系统都将用电子邮件通知客户。

- 音乐商店，鞋店，酒店.....

很多的 Web 应用服务，例如 IBM 的 Websphere 的商务套餐都体现了目录模式。[ibm]

相关模式

目录模式包括容器/上下文模式[Coa97]。股票经纪通过跟踪大量股票中的商品来实现这个模式[Fer00b]。其他经常和目录模式一起使用的模式是搜索模式[Lya99]。

购买流程模式

意图

为网上商店选择及购买商品的流程建模。

关联

网上商店包括很多商品。客户可以在同一次会话中选择和购买不同的商品。客户寻找自己感兴趣的商
品，选择他想买的东西放入购买列表中。他可以随时检查和更改他所购买的商品。有一些顾客还可以在网
站上登记并可以得到不同的待遇。网上商店还包括目录，成员表，运输以及其他的应用程序。

问题

购买流程必须有定义良好的步骤，以便告诉客户他现在正处于流程中的那一步。现在问题是：怎样用
精确的方式来描述购买流程？

约束

- 客户有可能在购买流程的步骤中感到很迷惑。我们要尽可能地让购买流程对于客户而言清晰而且方便。很明显，这要依靠那些显示给客户的特殊网页。在这里，我们所考虑的是使这项工作更为方便的后台结构。
- 我们需要向客户显示他现在正处于购买流程中的什么位置。
- 因为新的商品或是商业模式而可能需要改变流程的步骤。这可能暗示新的实体或新的步骤。

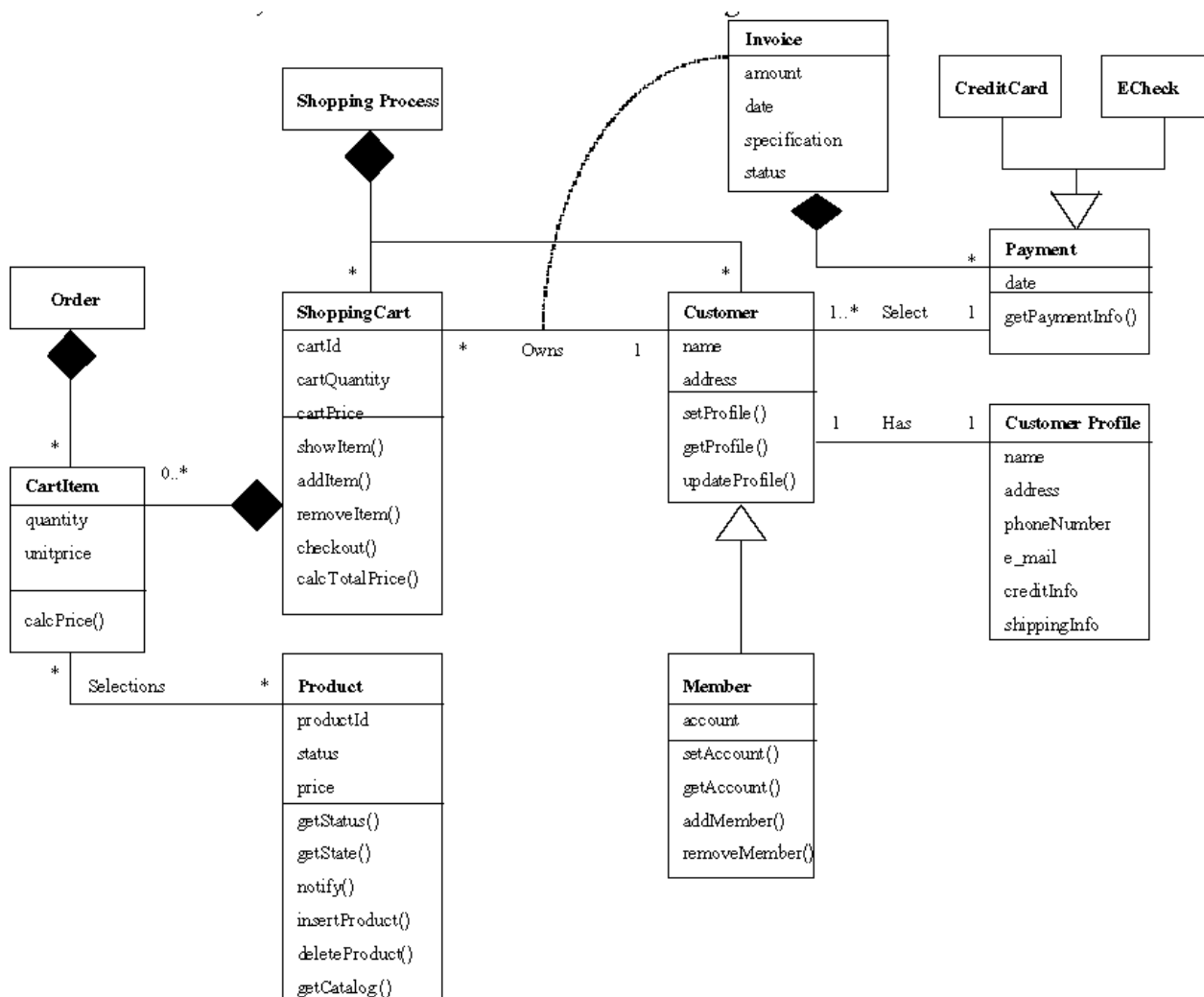


图 3.购买流程的类图

动态描述

这种模式包括了以下交互（图 4）：

- 选择：当客户对同一种商品进行选择时，产生一个新的购物车项目（cartItem）对象，并将其加入到购物车中。
- 结帐：当客户结帐时，计算所选择商品的金额，用户账单和送货信息被重置，随之产生收据。

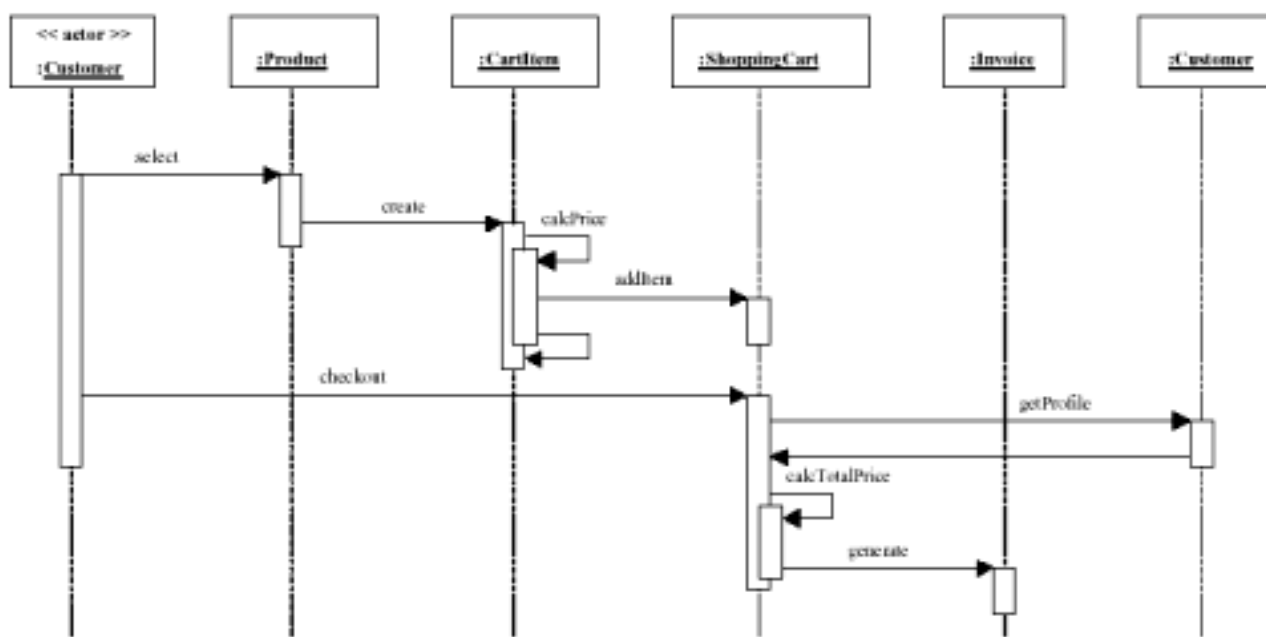


图 4.购买和支付商品的序列图

结论

使用该模式的好处如下：

- 该模式描述了一种抽象的购物流程。它被应用于不同的网上商店，从卖鞋的到卖软件的。
- 该模式为因特网上建立商店提供了公用的元素。
- 该模式可以简化因特网上的购物流程，部分或所有的步骤在向导中显示给用户。
- 对于会员用户可以不需要重新进入。
- 在总体结构中容易加入授权（功能）。例如，仅有流程管理员才有权变更流程。

其他需要注意的地方：

- 该模式并没有提供出错处理的机制，例如对错误的信用卡、错误的付款地址或送货地址的处理等，这些必须要包含在一个特殊的模块中。
- 这是一个通用的网上商店模型，当用于一些特殊的网上商店的典型时，可能需要修订。
- 对有效性而言，这个模式需要低级的基础结构来支持，例如数据库系统。

- 在模式中定义的安全约束必须被低级别的机制所执行，例如数据库授权、文件访问权限、加密和其他一些机制。

已知用途

大多数的网上商店使用购物车的概念，在基础结构上和该模式相似。相似模式的特殊模型和代码可参见附录[Bar96]， [Bol00]和[Kob01]。

相关模式

该模式可以用于支持网络用户的两种模式：

- 动态配置（Dynamic Configuration）模式[Lya98]，该模式帮助用户从不同选项中进行选择并验证其选择。
- 显式流程（Explicit process）模式[Ros00]，该模式帮助用户理解购买流程。

导航模式，如新闻[Lya98]和广告[Ross00]，可以利用这两种模式作为其基础结构的一部分。

订单和送货模式[Fer00]，可以补充本模式中的运送细节。付款的细节处理参见 [Fow97]以及[Hay96]。购物车，客户相关类以及付款类都是各司其职的基本模式。这里所介绍的两种模式的抽象版本的结合是购物框架的基础。

感谢

我们要感谢我们的领导 Gustavo Rossi 有见地的建议，他的建议使这篇文章增色不少。

参考文献

[Bar96] C. Baron and B. Weil, "Implementing a web shopping cart", *Dr. Dobbs Journal*, September 1996, 64-69 and 83-85.

[Bol00] G. Bollinger and B. Natarajan, "Build an e-commerce shopping cart", *Java Pro*, June 2000, 38-50.

[Coa97] P.Coad, "Object models: Strategies, patterns, and applications" (2nd Edition), Yourdon Press, 1997.

[Con99] J. Conallen, *Building Web Applications with UML*, Addison-Wesley, 1999.

[Fer99] E.B.Fernandez and X.Yuan, "An analysis pattern for reservation and use of entities", *Procs. of Pattern Languages of*

Programs Conf. (PLOP'99), <http://jerry.cs.uiuc.edu/~plop/plop99>

[Fer99a] E.B.Fernandez, "Coordination of security levels for Internet architectures", *Procs. 10th Intl. Workshop on Database and Expert Systems Applications*, 1999, 837- 841.9

[Fer00] E.B.Fernandez, X.Yuan, and S.Brey, "An analysis pattern for the order and shipment of a product", *Procs.of Pattern Languages of Programs Conf. (PLOP'2000)*, <http://jerry.cs.uiuc.edu/~plop/plop2k>

[Fer00a] E.B. Fernandez and X. Yuan, "Semantic Analysis patterns", *Procs. Of 19th Int. Conf. on Conceptual Modeling, ER2000*, 183-195.

[Fer00b] E.B. Fernandez, "Stock Manager: An analysis pattern for inventories", *Procs. Of PLOP 2000*,
<http://jerry.cs.uiuc.edu/~plop/plop2k>

[Fow97] M. Fowler, *Analysis patterns -- Reusable object models*, Addison- Wesley, 1997

[Hay96] D.Hay, *Data model patterns-- Conventions of thought* , Dorset House Publ., 1996.

[ibm] IBM Corp., Web Sphere Commerce Suite, <http://www-4.ibm.com/software/webservers/commerce>

[Kob01] C. Kobryn, *Modeling components, patterns, and frameworks with UML*, Notes for tutorial, *Software Development 2001* West, April 2001.

[Lya98] F. Lyardet, and G. Rossi, "Patterns for dynamic websites", *Procs. PloP'98*, <http://jerry.cs.uiuc.edu/~plop/>

[Lya99] F. Lyardet, G. Rossi, and D. Schwabe: "Patterns for adding search capabilities to web information systems", *Proceedings of EuroPLOP'99*, <http://www.argo.be/europlop/index.html>

[Pal] M. Palmer, "A Personalization design pattern for dynamic websites", <http://objectdesign.com>

[Ros00] G. Rossi , F. Lyardet, and D. Schwabe, "Patterns for e-commerce applications", *Procs. EuroPLOP'2000*,
<http://www.coldewey.com/europlop2000/>

[Ros01] G. Rossi, D. Schwabe, J. Danculovic, and L. Miaton, "Patterns for personalized web applications", *Procs. EuroPLOP'2001*,
<http://www.hillside.net/patterns/EuroPLOP/>

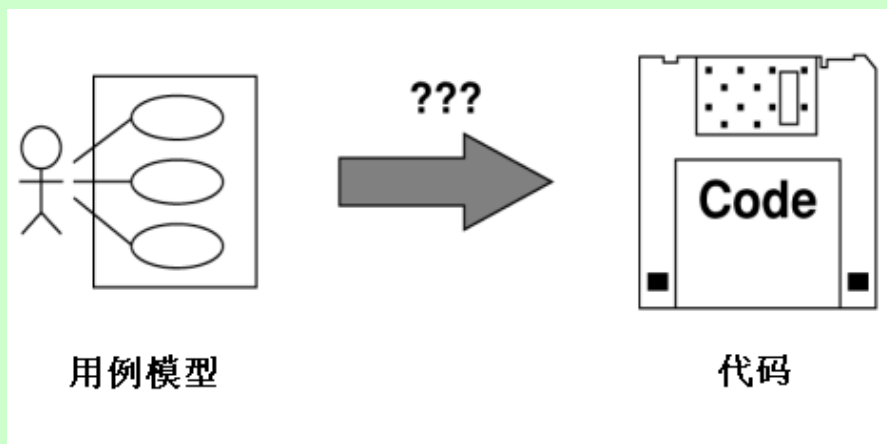
[Sch98] D. Schwabe, and G.Rossi: "An object-oriented approach to web-based application design", *Theory and Practice of Object Systems (TAPOS)*, October 1998.

Copyright (c) 2002 by Eduardo B. Fernandez. All international rights reserved

UMLChina 保留中译本一切权利

UMLChina 培训

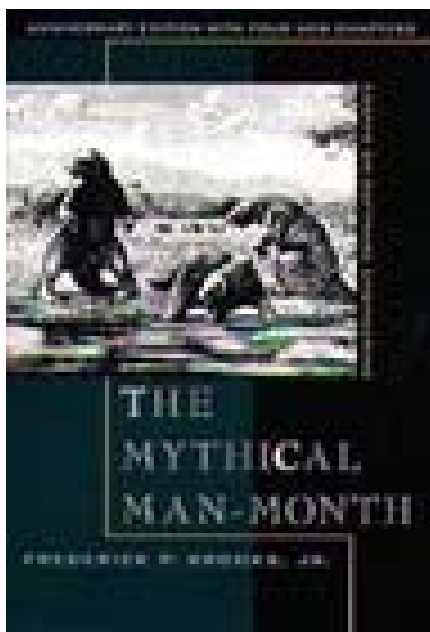
The real thing



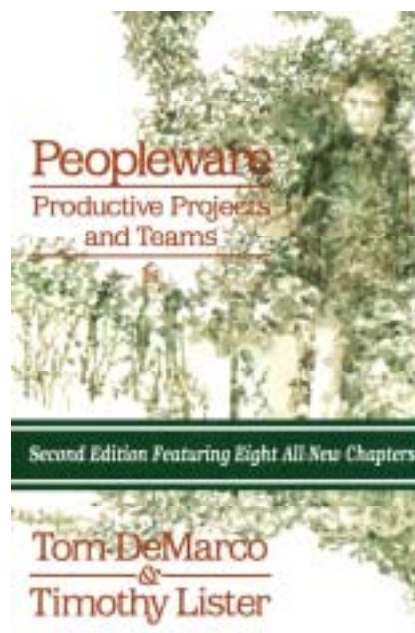
利用 UML 的 20%就可以为 80%的问题建模
-- 《UML 用户指南》，第 32 章

详情请垂询：think@umlchina.com

中文译本即将发行！



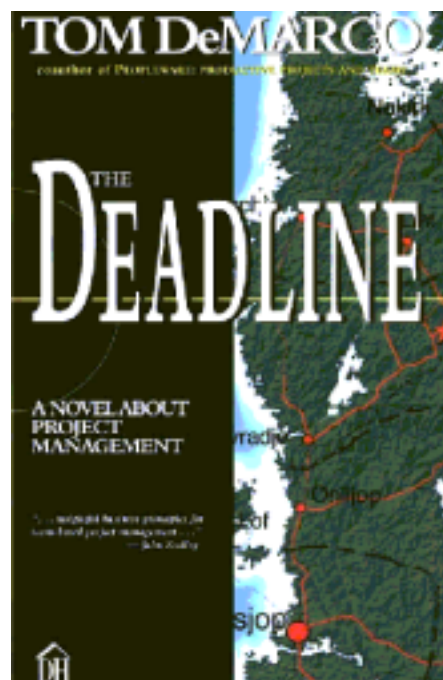
《人月神话》20 周年纪念版，Fred Brooks
翻译：UMLChina 翻译组 Adams Wang
散文笔法，绝无说教，大量经验融入其中



《人件》第 2 版，Tom Demarco 和 Tim Lister
翻译：UMLChina 翻译组方春旭、叶向群
微软的经理们很可能都读过—amazon.com



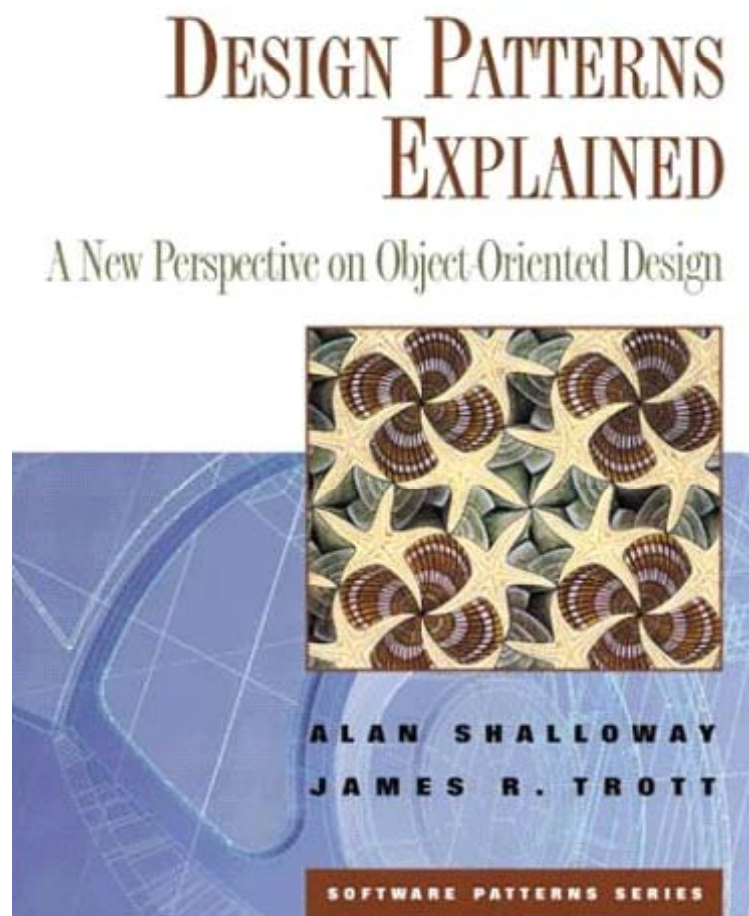
《面向对象项目求生法则》，Alistair Cockburn
翻译：UMLChina 翻译组乐林峰
Cockburn 一向通俗，本书包括十几个项目的案例



《最后期限》，Tom Demarco
翻译：UMLChina 翻译组透明
这本最通俗，这是一本软件开发小说

看不懂《设计模式》？

它的作者推荐《设计模式精解》----中译本即将上市！透明 译



直接学习《设计模式》这本书是非常困难的。作为入门，我推荐 Alan Shalloway 的《设计模式精解》。

--John Vlissides 写于 [UMLChina 答疑板](#)

《分析模式：可重用对象模型》

学习笔记之四：企业财务分析中的观察和测量（下）

[Windy.J](#)

1.2.5 维数合并（Dimension Combinations）

第三种很常见的测量方式/种类是按维数合并，很容易想像，例如，统计整个东北地区的销售额，这时需要提供的信息是维数和测量的类型，然后对东北地区下属的各个子区域的同类型测量对象进行合并，也就是说，按维数合并是在某个维数上针对某种测量类型的合并。

合并的计算方法一般来说是累加，但也不完全如此，例如，求平均值的操作就不是。

因此我们可以在测量协议的子类中增加一个类型：Dimension Combination，首先必须指明是针对哪一个 Dimension 进行合并，然后对这种类型的协议增加一个约束（constraint），让它的输入参数为 0（因为输入参数一般跟输出的类型一致）；对这种测量的创建过程也跟前面提到过的差不多一样，只是将查找输入参数替换成查找 Dimension 的下属对象就可以了。

可以针对每个 Dimension 和 Phenomenon Type 定义一种 Combination；还要提到的是，可以针对每个 Phenomenon Type 的 Combination 定义比较测量（Comparative Calculations）。

1.3 范围（Range）

在判断某个数值是否落在某两个数值之间时最常见的是要用到范围（Range），范围一般由一对数值组成，而对范围的检查也需要同时检查它们两个。而 Range 正是用一个单独的对象对这对数值进行整体表达，并提供一般的相关方法，例如检查某个数值是否落在它自己的区间里。

它的基类非常简单，包括两个表示上下限的属性 start 和 end；同时再提供一个 include 方法，来判断范围。

我们可以用任何支持比较的数据类型作为范围的数据类型（在这里，可以用到 c++ 中的模板类/参数化类的概念，在 UML 表述中写成 `Range<T>` 的方式，如果你用的语言也支持的话）；例如 `Range<number>`，`Range<date>`；而实际上有许多的人，包括作者自己，一般都只用到这两种常用的数据类型，并引入了术语 `Number Range` 和 `Date Range`。

更复杂一点的 `Range` 可能还设置了一种排序准则，就是将区间里的各种对象按某种方式排序的方法，表现为一种方法（function）；或者是包装了这种方法的对象。

对于开区间，例如 `>6`，我们可以用 `null` 值代表一个无限值；然后在实现判断的时候会稍稍显得复杂一点，不过对用户来说，他们不会看到这种复杂性，从而觉得接口很友好。还有一种办法是提供一些额外的 `Range` 创建方法例如 `Range.greatThan()`。还要注意的，如果范围支持的数据类型是连续的而不是离散的（实数对于整数就是连续的），例如“`>6`”的整数范围可以将下限设为 7 来表达，而同样的实数范围你大概不想设为 6.00000000000001），那么最好再提供一对 `bool` 值来辅助表达上限和下限是否包括在 `Range` 中。

`Range` 类非常适合采用参数化类的方式，如果语言支持的话；如果语言不支持，可能需要建立在基类，或者为特殊的类生成子类；如果需要对上下限的对象进行其他的操作，那么向下转型将会是一场非常痛苦的经历。

我们习惯上用起始值（`start` 和 `end`）来表示 `Range`；也可以加上长度（`length`）；如果一定要以 `start` 和 `length`，或者 `end` 和 `length` 来表达也是可以的，甚至可以三个都用，不过要定义这三个值之间的约束条件。

什么时候会用到 `Range` 模式？作者说，他一直都在使用，用 `Range` 比用一对数值好多了，而且编码也非常容易。而笔者也曾参照 `Range`，实现了一个根据范围查比例的类，例如从 1 到 100 对应的比例是 0.1，101 到 200 对应的比例是 0.2，等等，感觉非常好用。

下面是一些用 Java 给出的代码示例（Sample code），是作者最喜欢的 `Date Range`。

```
class DateRange...

    public DateRange (Date start, Date end) {

        this (new MfDate(start), new MfDate(end));

    }

    public DateRange (MfDate start, MfDate end) {
```

```
        this.start = start;

        this.end = end;

    }

class DateRange...

    public MfDate end(){

        return end;

    }

    public MfDate start() {

        return start;

    }

    public String toString() {

        if (isEmpty()) return "Empty Date Range";

        return start.toString() + " - " + end.toString();

    }

    public boolean isEmpty() {

        return start.after(end);

    }

}
```

Range 最关键的方法：includes()方法。

```
class DateRange...

    public boolean includes (MfDate arg) {

        return !arg.before(start) && !arg.after(end);

    }

}
```

为开区间和空白范围提供的构造函数。

```
class DateRange...

    public static DateRange upTo(MfDate end) {
```

```
        return new DateRange(MfDate.PAST, end);
    }

    public static DateRange startingOn(MfDate start) {
        return new DateRange(start, MfDate.FUTURE);
    }

    public static DateRange EMPTY = new DateRange(new MfDate(2000,4,1), new MfDate(2000,1,1));
```

对范围进行比较的方法也非常有用。

class DateRange...

```
    public boolean equals (Object arg) {
        if (! (arg instanceof DateRange)) return false;
        DateRange other = (DateRange) arg;
        return start.equals(other.start) && end.equals(other.end);
    }

    public int hashCode() {
        return start.hashCode();
    }

    public boolean overlaps(DateRange arg) {
        return arg.includes(start) || arg.includes(end) || this.includes(arg);
    }

    public boolean includes(DateRange arg) {
        return this.includes(arg.start) && this.includes(arg.end);
    }
}
```

其他一些可能有用的方法，例如两个范围的差（返回一个范围的下限和另一个范围上限之间的范围）

class DateRange...

```
public DateRange gap(DateRange arg){  
    if (this.overlaps(arg)) return DateRange.EMPTY;  
  
    DateRange lower, higher;  
  
    if (this.compareTo(arg) < 0) {  
        lower = this;  
        higher = arg;  
    }  
    else {  
        lower = arg;  
        higher = this;  
    }  
  
    return new DateRange(lower.end.addDays(1), higher.start.addDays(-1));  
}  
  
public int compareTo(Object arg) {  
    DateRange other = (DateRange) arg;  
  
    if (!start.equals(other.start)) return start.compareTo(other.start);  
  
    return end.compareTo(other.end);  
}
```

还有两个范围是否相邻。

class DateRange...

```
public boolean abuts(DateRange arg) {  
    return !this.overlaps(arg) && this.gap(arg).isEmpty();  
}
```

还有一个范围是否落在另一个范围里面。

class DateRange...

```
public boolean partitionedBy(DateRange[] args) {  
    if (!isContiguous(args)) return false;  
    return this.equals(DateRange.combination(args));  
}  
  
public static DateRange combination(DateRange[] args) {  
    Arrays.sort(args);  
    if (!isContiguous(args)) throw new IllegalArgumentException("Unable to combine date  
ranges");  
    return new DateRange(args[0].start, args[args.length - 1].end);  
}  
  
public static boolean isContiguous(DateRange[] args) {  
    Arrays.sort(args);  
    for (int i=0; i<args.length - 1; i++) {  
        if (!args[i].abuts(args[i+1])) return false;  
    }  
    return true;  
}
```

1.4 使用结果框架（Using the Resulting Framework）

这一章讲述的模式是第三章的扩展，主要讲述了几种计算测量，以及定量的测量和定性的测量如何通过 Range 结合起来。本章的内容反映了一种尝试，那就是试图用医疗保健行业（Health Care）中适用的模式来解决企业金融分析的问题；同样，这些扩展也可同样反馈回原来的医疗保健行业。通过这样的扩展和反馈，作者希望更多有用的模式可以结合起来，从而可以得到更广泛的应用。

第五章讲述的是对象识别的问题，最常见的是用 Name（名称）来识别对象，但任何有商业意义的属性都不适合作为对象的唯一标识，因为商业规则是不稳定的。关于这点可以参考 Scott Ambler 的文章“Mapping Objects to Relational Databases”，笔者就不再详细介绍了。

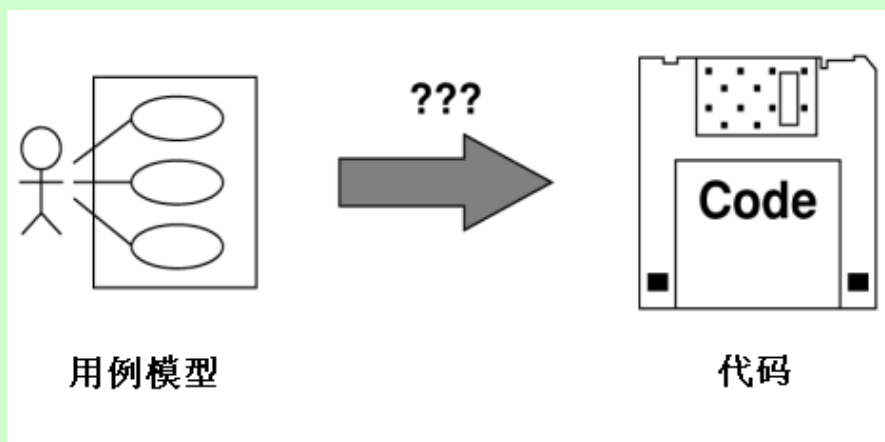
对于对象相等的判断，可能会根据具体情况有所不同，例如，是判断 ID 相等呢还是判断其他的属性相等，这一点是需要注意的。

第六章和第七章讲述的是 Accounting（记账/帐务）模式，对这个模式 Fowler 先生后来进行了改写，最新的英文原文在 www.martinfowler.com，地址是 <http://martinfowler.com/apSUPP/accounting.pdf>，大家可以先去下载来看看，下次我再把笔记发过来。

对于模式的理解，如果有不同的意见，欢迎大家发邮件给我，windy.j@163.com；例如在“前言和介绍”里面有一个 Mental Modal，当时我没有找到合适的译文，翻成了“精神模型或理论模型”，后来 sciman 给我来信说翻译成“概念模型”比较贴切，谢谢 sciman。

UMLChina 培训

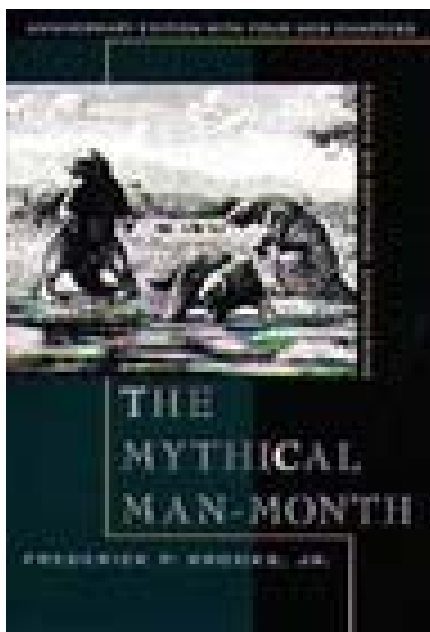
The real thing



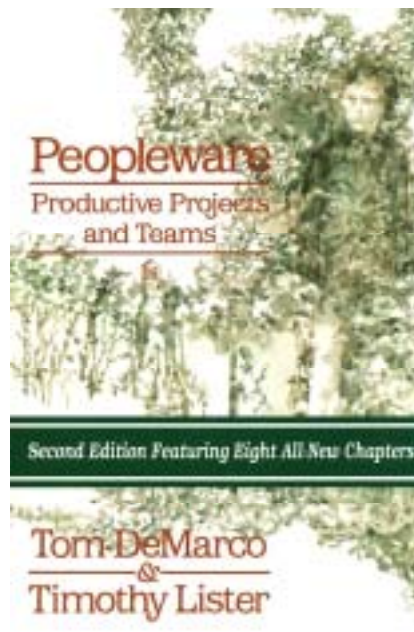
利用 UML 的 20%就可以为 80%的问题建模
-- 《UML 用户指南》，第 32 章

详情请垂询：think@umlchina.com

中文译本即将发行！



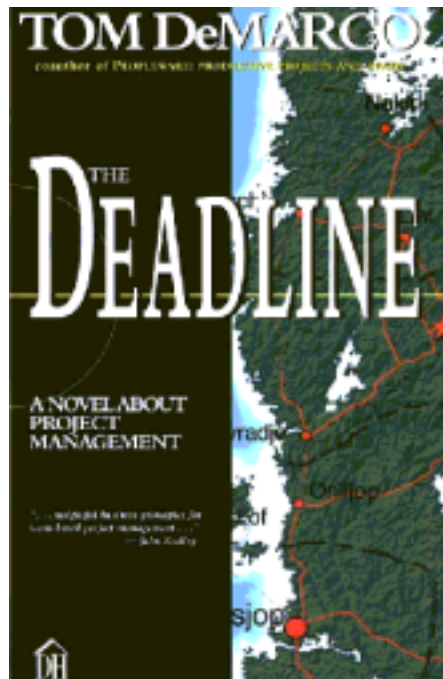
《人月神话》20 周年纪念版，Fred Brooks
翻译：UMLChina 翻译组 Adams Wang
散文笔法，绝无说教，大量经验融入其中



《人件》第 2 版，Tom Demarco 和 Tim Lister
翻译：UMLChina 翻译组方春旭、叶向群
微软的经理们很可能都读过—amazon.com



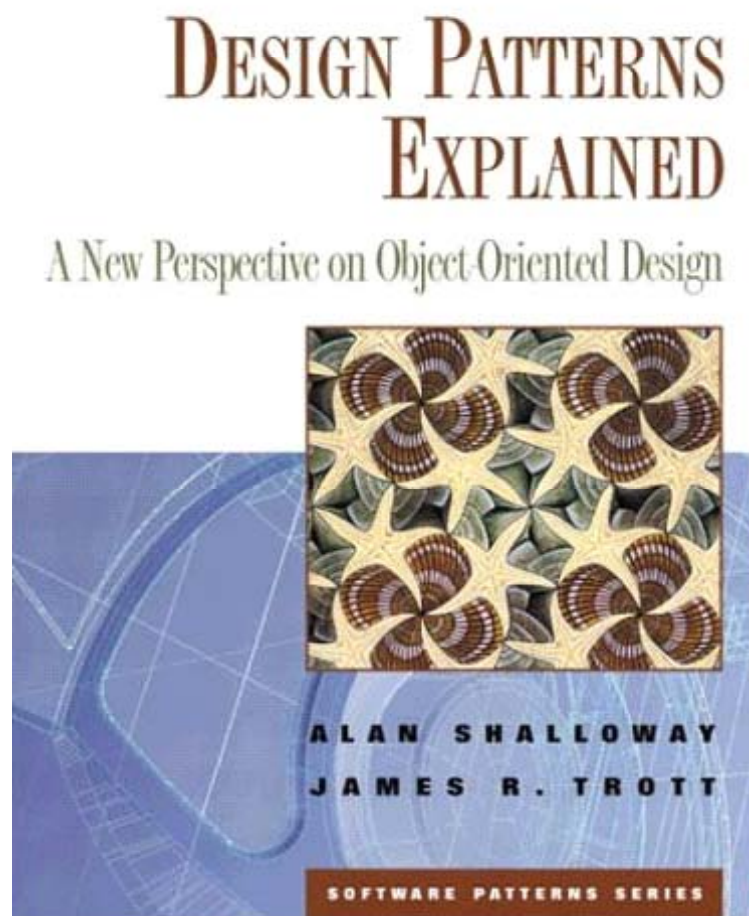
《面向对象项目求生法则》，Alistair Cockburn
翻译：UMLChina 翻译组乐林峰
Cockburn 一向通俗，本书包括十几个项目的案例



《最后期限》，Tom Demarco
翻译：UMLChina 翻译组透明
这本最通俗，这是一本软件开发小说

看不懂《设计模式》？

它的作者推荐《设计模式精解》----中译本即将上市！透明 译

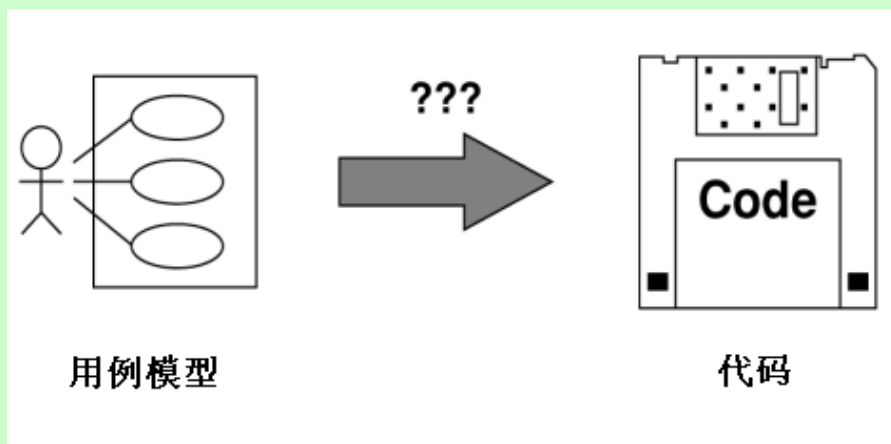


直接学习《设计模式》这本书是非常困难的。作为入门，我推荐 Alan Shalloway 的《设计模式精解》。

--John Vlissides 写于 [UMLChina 答疑板](#)

UMLChina 培训

The real thing

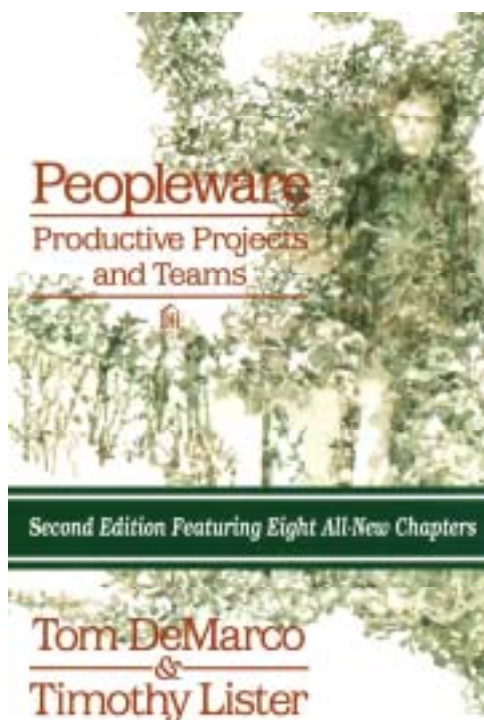


利用 UML 的 20%就可以为 80%的问题建模
-- 《UML 用户指南》，第 32 章

详情请垂询：think@umlchina.com

《人件》节选

Tom Demarco 和 Tim Lister 的 “Peopleware : Productive Projects and Teams” (人件：高效的项目和团队) 第一版于 1987 年出版，专门讨论了软件开发和维护的团队管理问题，向传统的管理方法提出了挑战，推崇人本管理思想，给予软件工人自由和信任。和《人月神话》一样，该书现在已经成为软件团队管理的经典之作。1995 年，Larry Constantine 出版了一部题为 “Constantine 论人件” 的论文集，扩大了 “人件” 概念，把那些使用软件的人也包括进来。1999 年 2 月，《人件》第二版出版，增补了 8 章新内容。这些增补的内容视角更加宽广，对比较大型的组织中的团队如何运作进行了探索。《人件》的中文译本即将发行，译者为 UMLChina 翻译组的方春旭、叶向群，以下是翻译草稿的片段。



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。

-- 《人月神话》20 周年纪念版第 19 章

Frederick P. Brooks, Jr

2 生产三明治，出售三明治

开发从来就与生产不同，但开发经理及其同僚常常让自己的思考方法定型于完全从生产环境中引伸出来的管理哲学。

花几分钟时间设想一下，你是当地一个快餐店的经理，对你来说要争取尽可能诚实地遵守下列有效的产品度量标准：

- 排除错误。使机器（人就是你的机器）尽可能顺畅地运转；
- 严厉地对付那些上班时打发时间的人；
- 对待工作人员就像对待可拆换的机器零件一样；
- 优化稳定状态（甚至不考虑怎么做可以提高速度，或者什么因素有可能使一切停止。）；
- 使过程标准化，一切按书中的说法去做；
- 消灭实验——（因为）那是需要老总们花钱的。

如果你在经营一个快餐店（或者在从事任何一种生产环境下的工作）这将是一个合理的方法，但你经营的并不是这类业务。将“生产奶油三明治，出售奶油三明治”这种方法用到以脑力劳动中为中心的开发领域中将会导致重大失误，这样做只能使你的下属士气衰减，并且使他们的注意力从手头实际问题中移开。这种管理风格将使你的工作转向以突发性安排为主的运转状态。

要有效地管好你的知识工人，你需要把上面列出的度量标准几乎都反过来用，我们推荐的相反方法将在下文中阐述。

错误的限额

对大多数脑力劳动者来说，偶尔犯一个错误是自然的，这也是他们正常工作的一个组成部分，但大多数著作却把工作中的差错与罪过联系在一起，这是一种需要我们做出痛苦转变的态度。

对一些软件经理来说，我们引入一种策略，我们把这种策略称作交互设计，产生这种思路是由于设计在本质上就是一个不断更正缺陷的过程，他们应该被扬弃而不是持续修补。这种失败的结果在设计活动中应该是预料之中的，对这种失败的结果所付出的努力是用小的代价换来清爽的开始。出乎我们预料的是很

多经理感到这将是向他们的老板提出一个不可接受的政治问题：“我们怎么能把公司花钱生产的产品扔掉呢？”，他们似乎相信即使从长远来说将要付出更大的花费也要将这种带有缺陷的版本挽救回来。

要形成这样一种氛围：不要因为一些小小的错误就让人产生防备心理，他们不去尝试那些弄不好就有可能被炒掉的事情。当你试图使开发过程系统化时，当你实施强硬的方法以至于员工不允许犯任何关键策略性决定唯恐他们不正确时，你也在鼓励这种防备心理的工作态度，在不允许出错的管制下，一般水平的技术也许通过谨慎地试验可以掌握。然而，这种团队社会将遭受更惨的命运。

相反的方法将是鼓励人们犯适当的错误，做到这一点，你可以通过询问你的下属偶尔在他们走入死胡同同时他们做了什么，并且让他们确信他们懂得了“无”并不是最好的答案。当人们在喘气的时候，他们应该得到祝贺——祝贺他们付出代价的那一部分。

管理：给大汉（尤指四肢发达，头脑简单的人——译者注）下定义

管理是非常复杂的事情，难以对它简单地定义。在伦敦的一次专业协会的会议上，我们遇到了一个高级经理，那些细微差别就不存在了。

他用这样的话概括他全部的看法：“管理就是踢公驴”。这种说法就等同于这样的观点：经理给出他的所有想法，而他手下的人仅仅按照他的命令去做就可以了。这个翻版，也许能让奶油三明治的生产正常地运转，但没有做任何努力使员工们用脑子工作而不是用手工作。在这种环境中工作的每一个人已经将他们的脑子当作齿轮一样在运转了。你也许可以“踢”人，迫使他们动起来，但你不能迫使他们去创造、去发明、去思考。

即使在背后“踢”人能够推进短期内的生产力，但从长远的角度来看，这样做是没有用的：对任何一个工人来说，如果老是感到自己的主动性不够，并且总是由他的老板来做“补充”，再没有比这种事更让他泄气的了。

关于这种管理方法的最悲哀的事情是它几乎总是多余的。你很少需要执行官的度量器跟着你的人员工作，他们大多数热爱自己的工作，你甚至可以偶尔让他们工作得更少，从而更有意义地完成工作（关于这一思想详见第3章）。

人力商场

在一个生产环境里，把人视为机器的部件是很方便的。当一个部件用坏了，可以换另一个。用来替换

的部分与原来的部件是可以互换的。你可以或多或少地订购这样的部件。

许多开发经理采用了类似的态度，他们竭尽全力地使自己确信没有人能够取代自己。由于害怕一个关键人物要离开，他强迫自己相信项目组里没有这样的关键人物，毕竟，管理的本质是不是取决于某个个人的去留问题？他们的行为让你感到好像有很多人物储备在那里让他随时召唤，说“给我派一个新的花匠来，他不要太傲慢。”

我的一个客户领着一个极好的雇员来谈他的待遇，令人吃惊的是那家伙除了钱以外还有别的要求。他说他在家中时经常产生一些好主意但他家里的那个慢速拨号终端用起来特别烦人，公司能不能在他家里安装一条新线，并且给他买一个高性能的终端？公司答应了他的要求。在随后的几年中，公司甚至为这家伙配备了一个小的家庭办公室。但我的客户是一个不寻常的特例。我惊奇的是有些经理的所作所为是多么缺少洞察力，很多经理一听到他们手下谈个人要求时就被吓着了。

——TDL

一个老板非常明显地表露出被他部下的个性吓倒的神态正好是这样一个缺少洞察力的经理的例子：他有一个非常聪明的员工，每年大部分时间花在回访客户的路上，因此花去费用的数目也就较高。该员工的开支分析报告显示，他在吃饭上的花费与其他人不协调，他比其他人的餐饮费用多一半，在一次公开的例会上，老板因一时愤怒竟然污辱那位员工，说他是一个“食物罪犯”。其实那个员工的总费用并没有超出标准；无论他在饮食方面的额外消费多少，他都从其他方面节省回来，此人并没有比其他人花费更多，只是与别人不同罢了。

3 维也纳正在等着你

几年前，在南加州参加一个大项目的过程中，在我与项目经理之间爆发了一场战争。

一开始他就将项目成功的希望和疯狂的时间表压到下属肩上，在项目进行期间两人离婚（可直接追溯到的原因是由于经常加班）和一个员工的孩子吸毒（可能是由于在过去的一年里孩子的父亲太忙未能尽到做父亲的责任），最后，他们又神经质地将一个测试组长停职了。

在他继续这些可怕的行程时，我开始认识到在他这种奇怪的工作方式中，此人一直在吹牛，你也许怀疑除了那两个离婚和一个自杀的人之外，至少在他的眼里看来，项目将能取得完全的成功。

——TMD

每当谈起“聪明地工作”这一话题时，一个很普遍的感觉就是现实世界中所有的管理就是为了在同样人员花费的条件下使人们更努力、更长时间地工作。经理们总是对他们投入的人力不停地吹着产品数量的号角，其诡计就是在从这些人身上榨取更多的价值。

西班牙的管理学理论

很久以前，历史学家们从各种不同的价值理论中形成了一个抽象的结论：对一个理论来说，西班牙理论坚持认为地球上只有一个固定数量的价值，因此通向积累财富的道路就是学会从土地或者从人身上更有效地榨取财富。于此同时英国也有一个理论，这个理论认为价值可以通过天才和技术创造出来。因此英国产生了工业革命，而西班牙人则转起了车轮，开始开垦土地，形成了印第安新世界，他们从海外运回大量的黄金，他们所有努力带来的却是通货膨胀（太多的金钱追逐太少的有用货物）。

在各地的经理人中西班牙理论的幽灵依然活着，而且还很活跃，每当他们谈到生产力的时候，你就能看到这一点。

生产力的含义应该是在一个小时内取得更多的成果，但一种非常普遍的情形是，一切都用一小时的工钱榨取更多的利润作为追求目标，两者之间有很大的不同。

持西班牙理论的经理梦想通过不付钱要求工人加班加点的简单手段将生产力水平提升到一个新的高度，他们将一周所要做的任何工作安排在 40 小时之内，而不是按工人实际上投入的 80 小时或 90 小时来安排。

这不是准确的生产力定义——这更像是欺骗——但对美国的经理人来说这是一种艺术境界。他们威逼或哄骗他们的下属延长工作时间，他们给下属的印象是交货日期是多么重要（即使这个日期可能是武断地给出来的，地球不会因为你的项目推迟一个月完成而停止转动）。他们欺骗手下人接受毫无希望实现的进度表，让他们带着愧疚的心理牺牲一切来迎接最后期限，竭尽所能地使手下人更努力更长时间地干活。

现在从家庭的角度说句话

虽然你的员工在办公室里也许看起来“工作时间更长、更努力”，但从他们家里却传来完全不同的信息，家中传来的消息是“生活已离你而去，你的佣人已不再为你服务，你的小宝宝没人抱了，你的配偶刚开始把目光投向别人，生活上，所谓旋转木马的游戏只剩下最后一轮了，只有最后一次敲锣的机会了，如果你把你的生活全部用到 COBOL 上……”。

但是你知道，当实情说出之后，

你可以得到你想要的或者你就这样活到老，

当你正走在半路时，你正要被踢出局

当你认识到这一点……维也纳正在等着你？

——“陌生人” Billy Joel

用 Billy 的话来说，维也纳正在等着你，那是你人生之路的最后一站。当你到达那里时，一切都完了。如果你认为你的项目组成员对如此沉重的话题一点也不担心，请你再想想。你的下属很清楚上帝赐予每个人的生命都是短暂的，他们也非常清楚在他们这种无聊的工作之外还有更重要的事情在等着他们。

权衡

从短期来看，这种权衡可能对你有利，但从长远的角度来看它将被淘汰。

让你发疯的孩子慢下来，放下电话离开一会，就这样，你努力放松一两天，当你认识到……维也纳在等着你？

就像大多数持西班牙理论的经理们对不付钱加班视而不见一样（这些人总是计算一周的 40 小时而不考虑人们实际投入了多少时间），在工作时间内也照样是视而不见。你在任何人的时间表上看不到这一点，时间花在打电话上，还是用来闲谈，或者干脆就是用来休息，没有人能真正工作超过 40 小时，至少不是连续处在创造性工作所需要的紧张状态之中。

加班就像跑步：在马拉松赛跑中，到只剩下最后几百码时，对那些还有剩余能量的人来说，快跑从某种意义上来说是有用的，但如果你在第一英里的时候就开始快跑，你却是在浪费时间。试图使你的下属过多地加班加点，最终的结果只能是失去他们对你这个经理的尊重。

最好的员工以前已经经历过这些事情的所有过程，他们非常懂得保持沉默，并且在经理咆哮着要求在四月前一定要完成工作的时候他们转动着眼珠，然后在可能的情况下他们会在上班时间寻找补偿，结束每周实际工作量为 40 小时的投入，最好的员工会用这种方法做出反应。还有另一些就是那些迷恋工作的人。

迷恋工作的人

迷恋工作的人加班加点，不需补偿。

他们将工作过多的小时，虽然他们的工作效率可能会降低。当给他们施加足够压力时他们会主动采取损害个人生活的做法。甚至对最迷恋工作的人来说，迟早会传递这样的信息：

放慢一点，你做得很好了，

**你不可能把什么事都提前做好，虽然今晚走在完成任务的边缘会非常浪漫，但是当你认识到……
维也纳在等着你？**

一旦把这种思想融会贯通，对项目来说，组员已不见了。为了获得并不重要的东西（工作），一个人牺牲了更重要的价值（家人、爱情、家庭、青春），这是破坏性的，它会使得一个曾经丧失理智地做出牺牲的人寻找报复。他不会走到老板的办公室理智、心平气和地解释将来的情况会改变——他只能做到两相抵消的另一种情况，歇火。一条路或者另一条路，但他已经离开了。

工作狂主义是一种病态（心理），但它与酒徒不一样，酒徒只是极少数人的不幸。工作狂更像庸俗的冷酷：每个人都有一场偶尔的较量。我们在此写的这些内容的目的不是为了讨论它的根源和根治方法，而仅仅是为了告诉你，经理人，如何处理你的工作狂主义，如果你把他们剥削得所剩无几，用典型的西班牙理论的方法，你最终将失去他们，无论你是多么努力地需要他们把所有的时间都投入到项目中，你不能让他们以个人生活为代价做到这一点，损失一个好人是不值得的，谈到这一点已超过了狭隘的工作狂主义，进入到更加复杂的具有深刻意义的主题：生产力。

生产力：赢了小的战斗输了大的战役

下次当你听某人谈到生产力时，仔细听一听说话的人是否用了周转一词，很大的可能性是他或她没用这个词，多年来，听到的关于生产力的讨论或看到的数以百计的关于这方面的文章中，我们从来没有见到过一个专家在谈到相关主题时用到周转这个词。然而是什么感觉它可能使得人们讨论一个问题而没讨论其它的内容呢？考虑一下一些组织提高生产力的典型做法：

- 强迫人们加班加点
- 产品开发的机械化过程
- 在产品质量上的妥协（关于这一点详见下一章）

- 生产过程的标准化

.....

.....

年薪

在过去的一年中，我在一个项目中做了一些咨询工作，该项目进展得很顺利，项目经理知道她能够按项目进度表按时提交产品。她被召到管理委员面前做项目进度汇报，她说她可以保证她的产品在最后期限 3 月 1 号准备好，这一确切时间是按最初估计确定的。高层经理详细讨论了这一意料之外的好消息，然后第二天又把她叫来，他们解释说，由于她可以按时在 3 月 1 号完成，所以最后期限被改到了 1 月 15 号。

——TRL

一张能够真正按时完成的进度表对那些西班牙理论的经理们来说是没有价值的，因为它没有给工作人员带来压力。更好的做法是有一张毫无希望、不可能按时完成的时间表，它可以榨取工人们更多的劳动。

很可能在你的职业生涯中，你已经见识过一个或多个西班牙理论的经理。

对他们的短见付之一笑是很好的，但让你自己不上钩却不是那么容易。

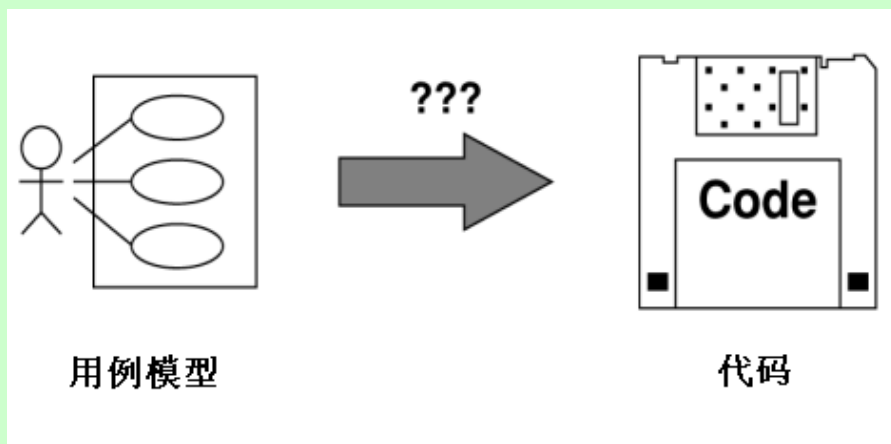
我们每个人都有过一次或多次不得不使用这种给下属施加压力让他们更努力地工作这种短期策略，为了做到这一点，我们不得不忽视他们效率的降低，不得不接受因此产生的营业额的结果，忽视负面影响是容易做到的，但不那么容易做到的是心里牢记这样一条永恒的真理：

人们在受到重压的时候不是工作得更好，只是工作得更快。

为了工作得更快，他们不得不牺牲产品的质量和他们对自己工作的满意度。

UMLChina 培训

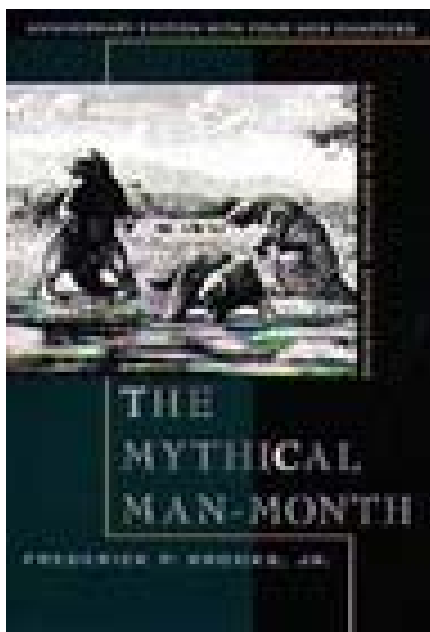
The real thing



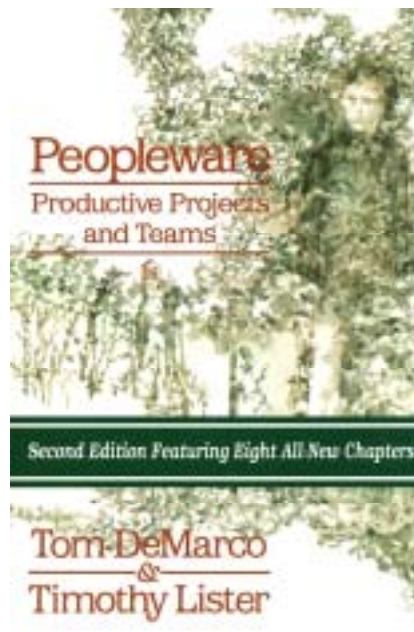
利用 UML 的 20%就可以为 80%的问题建模
-- 《UML 用户指南》，第 32 章

详情请垂询：think@umlchina.com

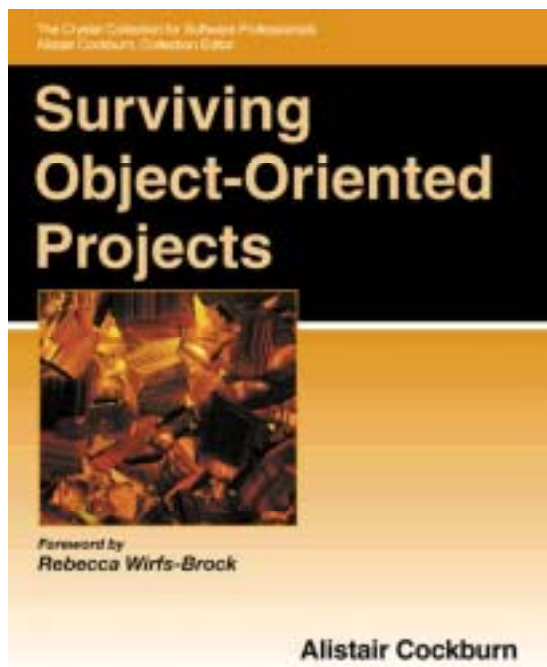
中文译本即将发行！



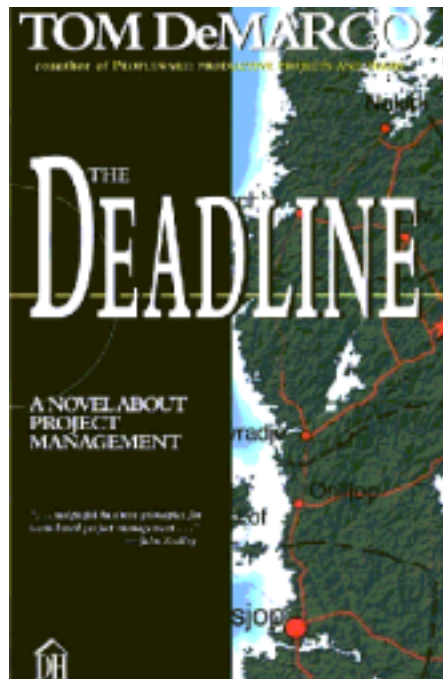
《人月神话》20 周年纪念版，Fred Brooks
翻译：UMLChina 翻译组 Adams Wang
散文笔法，绝无说教，大量经验融入其中



《人件》第 2 版，Tom Demarco 和 Tim Lister
翻译：UMLChina 翻译组方春旭、叶向群
微软的经理们很可能都读过—amazon.com



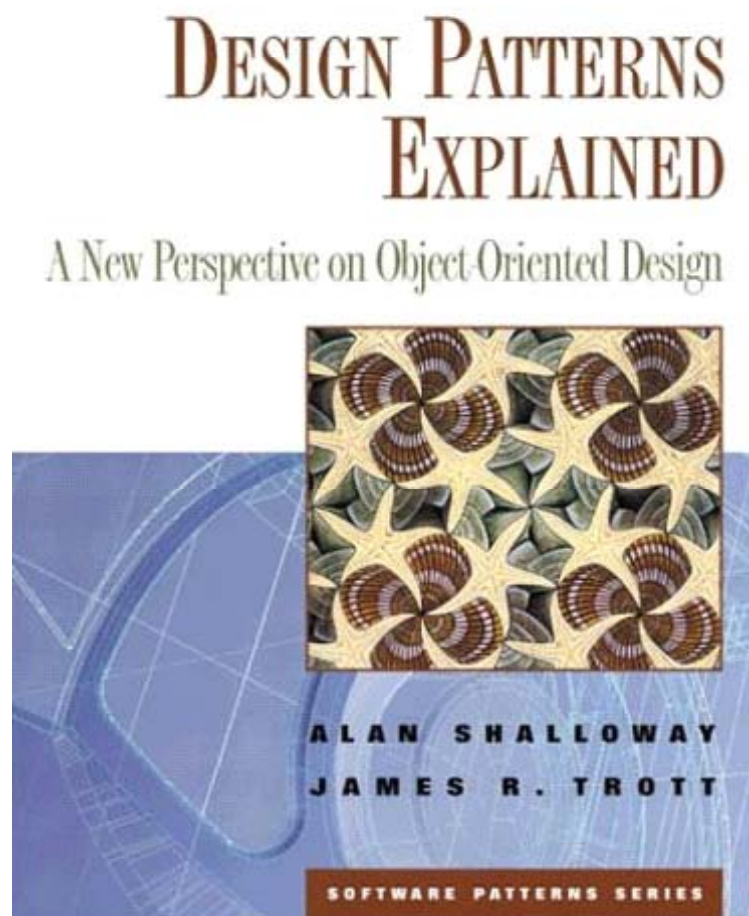
《面向对象项目求生法则》，Alistair Cockburn
翻译：UMLChina 翻译组乐林峰
Cockburn 一向通俗，本书包括十几个项目的案例



《最后期限》，Tom Demarco
翻译：UMLChina 翻译组透明
这本最通俗，这是一本软件开发小说

看不懂《设计模式》？

它的作者推荐《设计模式精解》----中译本即将上市！透明 译



直接学习《设计模式》这本书是非常困难的。作为入门，我推荐 Alan Shalloway 的《设计模式精解》。

--John Vlissides 写于 [UMLChina 答疑板](#)



关于需求，设计，构造，测试，维护，
配置管理，管理，过程，工具，质量...翻译
或原创均可。([详细>>](#))

UMLChina在线测试--目前最高分40000--VRGL, 杨应勇、纪海军

一个BBS系统，用户A发贴，用户B回贴，如果用户A在注册时设定了“回贴邮件通知”，用户B的回贴内容将作为电子邮件发送给用户A。对这个BBS系统，Actor是

A 用户A，用户B

B 发贴用户，回贴用户，电子邮件

C 发贴用户，回贴用户

D 用户

☐ A ☐ B ☐ C ☐ D

确定

已经连续答对8题，得分：0

UMLChina 在线测试