Agile软件开发丛书

PEARSON
Addison
Wesley

有效用例模式

Patterns for Effective Use Cases

Foreword by ***Craig Larman***
Series Editor: ***Alistair Cockburn***

[美] Steve Adolph
Paul Bramble 著

ePress.cn车立红 译

UMLChina 审

清华大学出版社

Agile 软件开发丛书

有 效 用 例 模 式

[美] Steve Adolph, Paul Bramble 著
ePress.cn 车立红 译
UMLChina 审

清华大学出版社
北 京

内 容 简 介

用例已经成为对软件需求进行建模的必不可少的方法。软件开发人员不仅要知道用例的基本原则，还要知道用于判断质量和效率的客观标准。本书便提供了所需的客观标准。本书简单易懂，针对实际项目中遇到的常见问题，提供了有效的解决方案。全书讲述了 30 多种有价值的模式，每种模式都有相应的示例，以方便读者度量其用例的质量。本书是第一本提出“借助既有模式编写用例”的专业图书。

本书可作为软件学院及大学计算机相关专业本科生和研究生的教材，也适合专业软件开发人士阅读参考。

EISBN: 0-201-72184-8

北京市版权局著作权合同登记号　图字：01-2002-3039

本书封面贴有 Pearson Education(培生教育出版集团)激光防伪标签，无标签者不得销售。

图书在版编目（CIP）数据

有效用例模式/[美]阿道夫, [美]布朗布尔著；ePress.cn 车立红译, UML China 审.
—北京：清华大学出版社，2003　（Agile 软件开发丛书）
书名原文：Patterns for Effective Use Cases
ISBN 7-302-06733-3
Ⅰ.有…Ⅱ.①阿…②布…③e…④车…⑤U…Ⅲ.①软件工程－系统分析②软件工程－系统设计
Ⅳ.TP311
中国版本图书馆 CIP 数据核字（2003）第 045573 号

出 版 者：清华大学出版社　　地　　址：北京清华大学学研大厦
http://www.tup.com.cn　　邮　　编：100084
社 总 机：010-62770175　　客户服务：010-62776969

文稿编辑：潘旭燕　特约编辑：蒋　芳
封面设计：立日新设计公司
印 刷 者：北京牛山世兴印刷厂
发 行 者：新华书店总店北京发行所
开　　本：170×230　印张：15.75　字数：354 千字
版　　次：2003 年 8 月第 1 版　2003 年 8 月第 1 次印刷
书　　号：ISBN 7-302-06733-3/TP・5019
印　　数：1～4000
定　　价：36.00 元

译 者 序

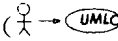
用例的概念在 1986 年由 Ivar Jacobson 正式提出之后被广泛接受，迅速发展。现在，“用例驱动”已经成为统一过程的基本原则。

Alistair Cockburn 等人在探索用例的本质和如何使用用例上，作出了大量的努力，使用例变得更加朴素、更加实用。

由 Alistair Cockburn 担任丛书编辑的 Agile 软件开发丛书，强调适度发挥个人能动性的软件开发技术。该丛书遵循以下两个原则：不同的项目有不同的需要；重点放在技能、交流和团体上能够使项目更有效更灵活。丛书沿着以下 3 条主线进行描述：如何通过使用个人技术提高个人的效率；如何通过使用各种团体技术提高团体的效率；成功方法的特定示例如何作为您自己具体情况的模型。

《有效用例模式》是其中一本编写高质量用例的手册，可以看作是 Alistair Cockburn 的《编写有效用例》（*Writing Effective Use Cases*）的续篇，它提供了一组用于测量用例质量的模式，并给出了一组当用例未能满足模式要求时的改进建议。

本丛书可作为软件学院及高等院校计算机相关专业本科生和研究生的教材和参考书，也适合专业软件开发人士阅读参考。为帮助我国软件专业人士更好地学习，清华大学出版社引进了该书的中、英文版版权，相信会成为广大开发人员最喜爱的读物。

UMLChina（ 

，网址 *http://www.UMLChina.com*）已经决定把本书作为专业训练的指定教材。UMLChina 从 1999 年开始就已经在中国推广用例技术，迄今为止，实践和指导实践用例技术的项目已超过 20 个，深深感到用例技术带来的强烈变化，正如面向对象为分析设计所带来的，用例也是软件开发的一种“返璞归真”。

本书由 ePress.cn 的车立红执笔翻译，UMLChina 的 Windy.J 和 Think 审校。因水平有限，翻译不到位之处请广大读者指正和谅解。

代丛书序：用例——十年风雨[①]

Alistair Cockburn

用例是与外界交互时系统行为的平直叙述。我想你们中的大多数人已经使用或听说过用例。其中一些人可能还听说过关于用例实际上是怎样有用（或怎样无用）的热烈争论。

在过去的十年间，我们对用例的理解已经向前迈进了一大步。本文的目标是为了说明：了解这些发展能让你了解现在人们使用、滥用、或者只是误解用例的方法，以及你自己如何更好地使用用例。为了展示发展的阶段，我把文章分成 4 幕来讨论：史前（比十年前还早）、过去十年、现在和未来。

第 1 幕：史前

Ivar Jacobson 在 1967 年定义爱立信 AXE 系统的构架时开始书写使用场景（usage scenarios），这些使用场景是非形式和粗糙的，目的是大体描述 AXE 系统是如何工作的。使用场景和平时人们写的大型正式结构并不是很像（在这上面也有我的贡献，惭愧……）

在 1980 年代中期，Jacobson 花了很多精力来思考过去十多年的工作方法。他造了一个瑞典语术语 anvendningsfall，大意是“使用情况”（situation of usage）或“用况”（usage case）（译注：现在，“用例”这个中文译法已经被广泛接受，所以本文的翻译也使用“用例”）。当用英文出版时，他发现“usage case”在英语里说不通，所以写作“use case”。如果你不关心“use case”的来源，只需要为每次不用说“usage case”或“anvendningsfall”感到高兴就是了。

Jacobson 最初使用场景时故意写成了非形式的。人们过去——现在还是一样

① 本序由 UMLChina 译。

——不愿意写形式的东西。事实上，有一次我向 Jacobson 询问用例的形式模型时，他答复："我的确有用例的形式模型，但唯一的问题是没有人想要用它。"

但是，用例保持完全非形式导致了其他困难。人们问，"什么是真正的用例？""我怎样才能知道我做对了？""我怎样才能知道什么时候我做完了？""用例很多时，怎样把它们关联起来？"

1992 年，在为 IBM 顾问组构建面向对象的开发流程时，我偶然发现了 Jacobson 发表于 1987 年的文章。像很多人一样，我认识到，这些系统范围行为的叙述自然地补充了作为面向对象设计成果的面向组件的内部叙述。所以我，和其他很多人一样，开始探索"什么是用例"。我也曾经陷入过"太形式"、"太不形式"的困境，案例太少，很难发现一条舒服的中间道路。Jacobson 坚持出版用例方面的书籍和文章，但不知何故混淆也延续了下来。

第 2 幕：过去十年

在 1992 至 1997 年间，大多数人在写用例时避免说这些东西（是）什么。虽然他们声称用例对项目是如何的有用。尽管看起来没有人能够说出一个用例是什么，或者说清用例和场景的区别，但那些基本的、吸引人的想法还是保留了下来：书写短小的、基于文本的、关于系统如何与周围事物交互的叙述，这些交互为用户提供一个价值，同时捕获出现错误时的系统行为。

这个想法过去有用，现在也有用，不管文本的书写是形式的还是非形式的。

在此过程中形成的不同流派中，人们建议和推荐了下面这些基本问题答案的几乎所有排列组合。

- 一个用例是一项需求还是只是一个故事？
- 场景是否只是用例的另一个名字？
- 用例的结构是形式的、非形式的、还是半形式的？
- 用例是否有关联结构，还是只是堆在一起？

对某些人来说，用例只是场景的另一个名字，是某人使用系统的简短描述（Martin Fowler 经常开玩笑说，"我想用例就是瑞典语中的'场景'"），出于这种想法，一个用例没有内部结构，用例的集合也只是堆放在一起。用这种方法书写

用例的人们从他们的努力中获得了很好的回报，他们中的许多人现在依然推荐书写这种非形式的用例。

其他人，特别是研究者和 CASE 工具设计者，认为这些非形式的工作产品是不完备的，需要 CASE 工具中的数学基础和支持。他们发明语言、标记和工具来使用例变得“严格”。这个流派进展并不顺利。人们需要一种比较非正式的媒介来表达他们对系统的早期想法。他们只是不想使用形式的工具。他们不想感觉到在项目生命期中付出更多的劳动。

形式主义者们面临的另一个问题是处理系统必须处理的所有变化。曾经有人问我，“对一个糖果机，我如何说明一个人可以塞进去 3 枚 25 美分的硬币，或者是 15 枚 5 分的硬币，或者是 10 枚 5 分的硬币再加上 1 枚 25 美分的硬币？或者是其他很多种可塞入硬币的组合方法？”

当时我的回答是，现在依然是，“只要写‘人投钱进去’就可以了。”

多次陷入太形式和太不形式的困境之后，我选择了一条中间路线：半形式的结构中的半形式文本。使用这些半形式的结构，我们可以：

- 断言用例确实是需求并且需要一个基本的结构。同时，
- 允许人们在需要的时候书写任何想写的东西。

令人惊奇的是，第二个目标比第一个更重要。

这是我最终得到的半形式结构，它允许人们处理两个相互矛盾的目标。

使用目标组织用例

用例是参与者（actor）动作的文字陈述，这很容易理解。20 世纪 90 年代早期的大问题是：是什么把这些动作连接起来？

近距离聆听了 Ivar Jacobson 的解释和看了他的例子之后，我发现了线索。第一条也是最重要的一条线索是：用例描述了一个参与者试图使用系统达到一个目标。即，如果我们指定其中一个参与者作为主参与者，所有动作都与参与者达到其感兴趣的目标有关。

把用例和参与者的目标连接在一起是如此重要，因为它把书写者的注意力从程序员关心的功能列表转移，回到用户身上：用户真正需要使用这个软件来完成

什么是他们使用这个软件时的目标。如果软件支持这些目标，软件将产生最大的商业价值。

第二条线索是目标有时会失败。无论是签一份合同，从 ATM 提取现金，把 ATM 卡插入读卡器，还是在填写一张在线表单时移动到下一个字段，任何目标都会失败。思考并写出需求文档中的失败处理，能够在项目过程中为设计团队节省一大笔钱。

因此，一个用例的结构分为两部分：每件事都顺利进行时的事件序列，随后是各种目标和子目标失败时的不同小事件序列的描述。

即使有了这些线索，带着目标工作的结果还是有意外。目标有不同的尺度。

在某个时候我的目标可能是赢得一份商业合同。一个更直接的目标（时间跨度更短）是带客户去午餐。更进一步直接的目标（时间跨度又更短）是为午餐准备一些现金。这个目标又带来一个更小的目标：从 ATM 提取现金。在提取现金的时间里的一小段，我的目标是把 ATM 卡插入 ATM 读卡器以便开始取钱的事务。

所有这些目标在同一时间生效。达到（或不能达到）它们中的一个都可以被描述成一个用例。重要的领悟是认识到我们不能用捕获目标的相对尺度来标记用例，否则会使读者产生混乱。很多人在写用例时会工作在多种多样不同的目标层次。如果他们在早期就在用例文本中说明了目标层次，就可以提醒读者：用例书写者在想什么。这减少了许多误解。

对于 ATM 的行为来说，赢得一份合同是非常高的层次，或者说是“高入云端”类型的目标。从 ATM 获取现金是一个尺度非常自然的目标。把卡插入 ATM 是“泥底下”类型的目标。每一个都有资格成为真正的目标，也值得书写一个自己的用例，标记出目标的层次来提示用例的读者希望讨论哪些东西。

当书写一个用例时，我们把目标层次的想法和目标完成/失败的想法结合起来。我们在用例中书写每一步行动步骤作为更小的成功目标。我们写，“顾客把卡插入 ATM”，即和低层次目标“插入卡”相关，假设它成功了。

往下走是备选或扩展的部分，我们写如果行动步骤失败了将会发生什么（“卡堵在槽里：关闭 ATM 机并通知维修公司”）。

加入涉众和利益

参与者和目标在很长一段时间内是令人满意的，但终于在某个特定的地方露出了缺点。为什么我们要在用例中书写外部不可见的行为？为什么我们要在给钱之前记录“银行检查客户余额”，或“最后记录事务的日志”？

我们以前的答案一直是，“噢，这是一份需求文档，如果你不写下这些，你就得不到好的需求。”虽然这个答案可行，但不够令人满意。

有一天我注意到计算机的动作加强了涉众之间的契约，每一个动作既是为了维护这个涉众的利益，也是为了维护其他涉众的利益。

主要参与者只是涉众之一。拥有这个系统的组织是另一个涉众，政府的管理部门可能是另一个涉众，可能还有其他涉众。

这里是一个例子。Jim 走向糖果机。他的目的实际上不是花钱，他真实的目的是想要吃糖，如果糖果是免费的，他会更高兴。糖果机的主人的立场相反：实际上她的目的不是为了发放糖果，如果人们只是往机器里放钱而不要糖她会更高兴。糖果机加强一个简单的契约：Jim 放入钱，机器的主人看着他得到糖果。

在保险业务中的相应契约更加复杂。保单所有人出现了意外；受益人提出索赔；保险公司在允许的范围内付出最少的金钱；保险部门检查事务的操作在政策之内。计算机系统保证所有涉众的利益都被满足。用例的书写就是描述这些是如何完成的。

涉众和利益模型填补了参与者和目标模型的空白。计算机执行的每一个动作和涉众的利益相关。系统接收信息，根据业务规则验证输入值，更新内部状态，或输出数据。主参与者和辅助参与者都照顾到了，就像所有其他在系统操作时不出场的涉众一样。

一位顾问描述他和他的小组第一次为一个系统列出涉众和涉众利益时的情况，此系统最近刚刚交付。他们突然发现大多数变化需求在项目的第一年早就应该产生！换句话说，他们只要在早期为每个用例简单地写下涉众和利益，他们将能够避免各种类型的错误，这些错误就不会等到移交系统以后才显露出来。其他人也报告了类似的经验。一些人说他们在列出涉众和利益时发现了价值，不管他们捕获的需求是如何的随便和非形式。

在这个时候（1997 年），我们的用例理论看起来已经完备了。

反对用例的声音

以上模型被接受的时候，自然也出现了反对的声音。对继续探寻一个完美的形式模型的形式主义者来说，用例依然不是形式的。同时，对另外一些人来说，它又太形式了，需要写太多东西。这些人认为用例是非形式、非结构化的，不是需求。后面这些人的反应有三种：特性列表（feature lists）、故事卡片（story cards）和 task cases。

单个用例可以穿越不同人的编程边界，所以单个程序员的工作分配不是明确可见的。因此一些程序员要求可以被分配到单个程序员的“特性列表”。即使提取和维护这些列表花了很多工作量，使团队的维护负担加倍，它们依然经常被请求。

极限编程（XP）的作者坚持非形式场景的思想，根本不使用任何形式的结构。Kent Beck 创造了术语“用户故事”（user story）来描述此类需求。一个用户故事只是包括一些简短的语句或一些写在索引卡上的句子，用于声明用户所需要的东西。在 XP 中，用户故事不用作需求说明，而是作为将来谈话的航标。因此，这些卡片记录的信息只需要让程序员和客户知道将来讨论什么东西就够了。

Larry Constantine 使用场景说明的书写形式来帮助进行用户界面（UI）的设计。他发现他不需要记录系统的内部处理步骤或它与辅助参与者的交互。他可以成功地使用非常简单的两栏式结构。左边的栏列出用户在每一步试图完成什么。右边的栏列出系统的响应。这种两栏式表提供足够的信息来让人们发明优秀的用户界面。他把它重新命名为 task case，避免与系统说明的用例混淆。

依然存在分歧。一些人想要保持用例简短和非形式，另一些人想要让用例变得详细，给出概要模板和自动链接至其他系统文档。在这些不同流派的用例书写者中，除了使用某些叫用例的东西来捕获行为需求之外，他们之间的共同点不多。

第 3 幕：现在

分歧让我们来到现在，鉴于 Jacobson 说过的他有形式的模型但人们不想跟随。从用例的发展中，你应该注意以下 4 条关键的忠告。

准备多种格式

用例不只有一种模板。一些团队使用短的用例格式，这种格式中，主要的故事展示在一段简单的叙述中，失败处理列在随后的几个段落中。其他团队则需要详细的模板和仔细的书写规范。

在《有效用例模式》（*Writing Effective Use Cases*）一书中，我命名了书写用例的三种详细程度：简短的、非形式的、形式的。

- 简短格式用两到四句话来概述用例。可以放在电子表格的一个单元格中，允许电子表格中的其他栏记录业务优先级、技术复杂度、发布版本号和其他计划信息。
- 非形式格式包括一些文本段落，涵盖以上所提到的条目。
- 完全形式格式的用例的特点是使用长的模板，模板上有涉众、最小保证、业务规则、性能约束等。

由于业界的两种力量，三种格式不可能融合。第一，在用例书写中，多种项目情况和特点总是需要不同的格式。第二，野心勃勃的人们通过制造一些新东西来成名。一旦一个主题稳定下来了，就会有人去寻找能让他进思想名人堂的替代品。

要点是：总是需要多种需求形式和多种级别的格式。

只在某种形式适当的时候使用它

用例格式实际上只是一种书写的格式化方法。一种短文的形式有两部分。第一部分描述一个基本的场景，包括动作和交互。第二个部分给出一组场景，描述不同环境下的不同行为。这种书写格式可以在任何时候描述有变体的行为，例如黑盒需求、业务流程、系统设计说明。

- 作为系统的黑盒需求，用例描述：用户做这个，系统做那个；系统和另一个系统交流；某些事情出错了；系统现在做这些作为替代；等等。它清楚地说明了系统必须完成什么，但没有说系统如何做，也并没有假设系统使用面向对象方法（或其他方法）来设计。

- 作为业务流程，用例描述：顾客这样做，职员那样做，职员把某些东西交给另一个部门，另一个部门做一些别的事情，等等。但事情出错时，某些部门退回一些工作，这些工作转到另一个不同的部门，等等。这些类别的说明对业务人员来说相对容易书写，未受过训练的人也容易读懂，所以在业务流程重组中的使用正在日益增长。

- 作为设计文档，用例描述：用户做这个；第一个组件做一些事；该组件转发一个请求给第二个组件；等等。在不同条件下，组件之间的顺序是不同的，等等。用例很少用在设计文档中，因为有太多种其他方法来写设计文档。

要谨记的事情是用例的格式，也就是，一个行为场景紧跟着备选行为的场景，以适合某些书写要求。如果某种格式适合你，就使用用例；否则，如果不符合需要，把它们丢到一边，不要用这种格式。

了解用例的局限

用例不关心系统设计、界面设计、特性列表或测试，即使许多人想要用例这么做。当然，用例不应该关心这些。用例应该描述行为需求，但这并不能阻止人们寻找“自动”从用例获取设计的方法。用例不说明设计，使这些人在悲伤和欢喜之间徘徊。他们悲伤，因为他们希望得到简化软件开发的魔幻公式；他们高兴，因为他们认识到对黑盒说明的需要。

即使有用例的格式允许，用例也不应该用于描述界面设计。有几个原因，其中两个是：首先，用例是指一种需求文档，而界面设计是设计，由受过训练的人们在被告知系统该做什么之后创建。其次，界面设计是脆弱的、易变的。在用例中书写界面设计意味着文档不得不经常修改，远远超出了一份需求文档应有的修改频率，也远远超出了项目团队（我碰到过的）所能维护的。其他方式的说明，如状态图、Constantine 的 task cases，还有屏幕快照，都在用户界面方面做得更好。

用例和特性列表存在一个基本的不匹配。同样的系统特性可能在多个用例中作为单个条目表示。结果，一些人不得不把用例文本转化成编程任务。对很多人来说，包括我自己，这不是一个问题。他们只需在头脑中、在谈话中或注释中完成这个转化就可以了。这种工作方法避免了双重维护问题。任何既需要用例又需要特性列表的人是在寻找双倍的工作量。

用例不是测试计划或测试用例。用例没有测试用例需要的数据值。但是，既然用例说的是在不同的条件下应该发生什么，因此用例是测试流程的优秀输入。测试员还是必须创建真实的测试用例来和用例匹配。

避免用例的标准错误

用例书写者会犯很多错误。对项目来说，两个最常见也最昂贵的错误包括太多细节和包括用户界面说明。两种错误都使得用例变长，难以阅读、脆弱。

无论教师如何经常警告不要在需求文档中包括界面说明，新人还是很想描述“OK”按钮、特定字段、屏幕、屏幕序列。这样做的原因是人们喜欢具体的概念。虽然事实如此，但用例不适合做这个。用例要求你付出努力，去学习如何用技术中性的方法来书写。例如，“系统显示选项。用户选择一项活动”。这是可以的，你得到的用例更短、更易读、更健壮。

如果用例的主场景很短，许多人似乎会感到羞愧，所以他们把它伸展成 15 甚至 35 步的长度。以我个人来说，我还没有写过一个主场景长于 9 步的用例。9 不是一个魔幻数，只是说我在适当的层次上得到子目标和移除设计说明时，任务就少于 9 步了。用例有时可以短到主场景只有 3 步。

用例的最大价值不在于主场景，而是在于备选行为。主场景可能只占用例长度的 1/4 到 1/10，因为需要处理的备选情况太多了。如果主场景有 35 步那么长，用例会伸展到 10 页纸，这太难阅读和理解了。如果主场景在 3 到 9 步之间，整个用例的长度就会只有 2 到 3 页，当然，这也够长了。

如果你可以避免在用例中包含太多细节和用户界面说明，你的用例会更容易阅读，而可读的用例会真正地得到阅读。冗长难读的用例只会令人厌恶，在以后的项目生命期中被抛到一边。

第 4 幕：不远的将来

用例已经到达了一个静止点，这意味着两件事：这是一种有效的模型，可以用于教学和实践；分化已经开始了。

对于这点，我们可以预言不远的将来的一些事情。

（1）作为一种主流技术，用例将在本科课程中、全世界的标准软件开发和方法学课程中被讲授。

（2）工具厂商和理论家将继续寻求用例所隐含的形式主义。

（3）工具厂商将生产和理论匹配得更好的工具，使交叉关联的用例更容易书写、维护和集成到 CASE 工具套件中。

（4）人们将继续寻找这项当前主流技术的替代品，然后建议各种不用用例来进行软件开发的方法。

（5）一些事情依然不变：人们继续在他们的用例中书写太多的界面细节，不管他们的工具和模板是什么；公司将继续使用用例来避免人们面对面交流。

（6）人们将误用用例，不正确地教授用例，曲解用例，然后把随之而来的混乱归结到用例身上。

为了保证用例有用，你所能做的是学习：

- 如何把设计说明排除在用例之外；
- 在用例书写中何时使用不同级别的用例格式；
- 何时使用另一种格式来表示，甚至像两栏式表格那样简单的格式。

然后，就可以坐下来聆听唾沫横飞的争吵了。

序

自1994年以来，Standish Group每年都发布关于项目成功和失败因素的“Chaos”报告，该报告对数万个项目进行分析。在项目失败专栏中，有30%以上的失败因素与需求问题有很大关系。美国软件质量实验室（National Software Quality Experiment，简称NSQE）于2000年通过研究发现，所占比例最高（为41%）的缺陷与缺乏需求定义和可跟踪性有关。在Chaos研究所确定的关键成功因素“Chaos前10位”中，“高用户参与度”和“清晰的基本需求”是其中的两个关键因素。

显然，要想把工作做好，确定需求是很重要的。但是——一种普遍的错觉——人们通常认为试图在项目的第一个阶段就确定需求，并使其稳定下来的“瀑布”或顺序生命周期便是此问题的解决方法，事实证明并非如此。根据：软件项目的需求很容易发生变化（50%以上的需求会发生变化），而且如果试图在第一步就敲定需求，那么，项目会面临很大的挑战，并且很可能会失败。研究表明，一种可取的方法是，采用逐渐定义和细化需求的迭代生命周期，以及短的时间盒式实现迭代来确定需求。在“确定”所有需求之前，可以快速构建系统的关键部分，这些迭代通过快速反馈和修改周期推动了需求的澄清。

还可以做许多工作来改进需求定义并促进用户的参与。例如，可以使用一种需求语言，它是一种大多数参与人员都感兴趣，强调实现用户的真正目标，将相关需求放在了一起，并且在一个内聚的上下文中对需求进行描述的语言。这种语言就是由Ivar Jacobson于1986年首次提出的“用例”。

尽管用例很受欢迎，但还是有一些教条主义者把它们称为“滥用实例”。他们不仅缺乏创造力，而且多次误用用例，因此才得出了这样的结论。在这种情况下，我非常高兴能够看到有助于编写高质量用例的学习工具的问世，如有影响力的《编写有效用例》（Cockburn 2001），以及 Cockburn 一书的补充读物——这本由 Steve Adolph 和 Paul Bramble 编写的关于用例指导和模式的书籍。模式是学习和共享最佳实践的出色机制，通过学习和应用书中给出的建议，我们都会有很大的收获。20 世纪 90 年代初，我在加拿大不列颠哥伦比亚省温哥华市的一所大学任教，与 Steve 在一个办公室工作。有一天送他回家，我们边走边谈，气氛极为融洽，我们很快就建立了友谊。他构建出色软件的方法和热情给我留下了深刻印

象。20 世纪 90 年代中期，当我就职于 Paul 在亚利桑那州的首府菲尼克斯开设的公司时，与他相识。他提倡把模式作为捕获和培训最佳实践的工具，我赞同他的这种看法。知道他们有着共同的兴趣爱好后，我建议他们在温哥华的 OOPSLA 98 上合作，他们采纳了我的建议。看到这些同事开始合作真是太好了，他们已经在这一主题上进行了长期摸索和思考，付出了艰苦的努力，做了很多教学和研究工作，拥有很多经验，因此，他们的合作必将使我们所有人从中受益。

——Craig Larman
得克萨斯达拉斯
2002 年 5 月
www.craiglarman.com

前　言

用例广泛用于建模，但许多人在编写用例时通常会遇到这样那样的困难。尽管理解了用例的基本概念，但他们发现实际编写用例要比预想的更困难。造成这种情况的原因之一是缺乏判断用例质量的客观标准。许多人都发现清晰地阐述有效用例的特征很不容易。

本书对编写用例时遇到的问题进行了分析。它针对为实际项目编写用例时遇到的具体问题，提供了简单出色且经过验证的解决方案。我们确定了 30 多种模式，专业人员可以用这些模式来评估他们的用例。那些成功的项目很可能会表现出来的质量标志是这些模式的基础。我们的目的是提供与他人讨论和共享这些特性的专业用语，为有效编写和组织用例提供建议，并给出一些评估用例的“诊断方法”，希望使用这些模式成为人们的习惯。讨论用例时，我们希望听到这样的话：“我们有 SharedClearVision（见 4.1 节）吗？该用例有 CompleteSingleGoal（见 5.1 节）吗？”

读者对象

本书的读者对象为：编写和使用用例以及希望了解更多有关用例知识的人。本书假定您已具备了用例的应用知识，且具有一定的编写经验，它是 Alistair Cockburn 的《编写有效用例》一书的延续。如果您不熟悉用例或没有什么经验，我们建议您先阅读 Alistair 的《编写有效用例》。

不管是否基于软件，用例都有助于设计业务过程。本书并不是一本用晦涩难懂、只有技术人员能够理解的语言编写的软件开发书籍。另外，我们知道用例主要是一种软件开发工具，作为软件开发人员，必须把重点放在软件开发上。希望本书能够帮助项目的所有参与人员理解软件开发团体，我们尽量以一种适合所有用例开发人员的方式来说明这一主题。