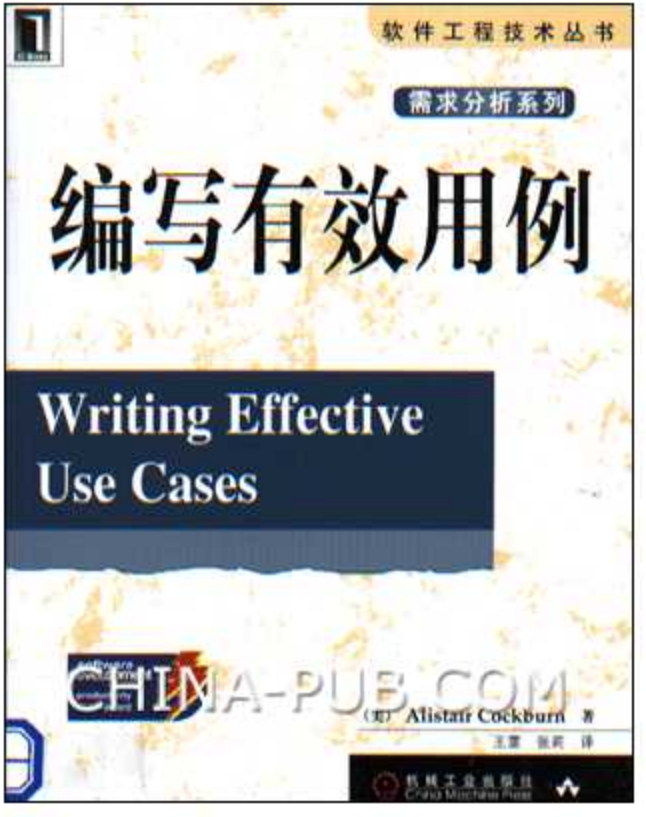

软件工程技术丛书
需求分析系列
编写有效用例
Writing Effective Use Cases
CHINA-PUB.COM
(加) Alistair Cockburn 著
王雷 张莉 译
机械工业出版社

目　录

第三部分 对忙于编写用例的人的提示

附 录

第1章 引　言

用例是什么样子？

为什么不同的项目组需要采用不同的用例编写风格？

在什么地方使用用例有利于做需求收集工作？

编写用例之前，需要做哪些准备工作？

为了帮助读者理解用例的概念，本书将在深入讨论用例具体细节之前，先对上述疑问进行简单解答。

1.1 用例是什么（梗概）

用例是代表系统中各个项目相关人员之间就系统的行为所达成的契约。用例描述了在不同条件下，系统对某一项目相关人员的请求所作出的响应。提出请求的项目相关人员被称为主执行者（*primary actor*）。主执行者通过发起与系统的一次交互来实现某个目标。系统对任一执行者所作出的响应，要保证所有项目相关人员的利益不受侵犯。根据执行者作出的请求和请求涉及的条件，系统将执行不同的行为序列，每一行为序列称之为一个场景（*scenario*）。一个用例是多个不同场景的集合。

虽然可以用流程图、顺序图、Petri网或程序设计语言来表示用例，但是从根本上说，用例是文本形式的。通常情况下，它们是作为人与人之间，尤其是没有受过专门培训的人员之间互相交流的一种手段。因此，简单的文本通常是编写用例的首选形式。

用例，作为一种编写形式，能够在项目组中激发对目标系统的讨论。项目组可以用用例来记录实际需求，当然也可以不用。另一个项目组可能会用用例来记录他们的最终设计结果。上述活动既适于支持大到整个公司系统，也适于支持小到软件应用程序的一个片断。重要的是，对于不同规模的系统，尽管各项目组对用例指定的严格程度和技术细节的深度要求各不相同，但是编写用例的基本规则却是相同的。

如果用用例来记录一个组织的业务过程，那么被讨论的系统（*System under Discussion, SuD*）是指组织本身。项目相关人员是指公司的股东、客户、供应商和政府管理部门。主执行者包含公司的客户，也可能还包含客户的供应商。

如果用用例来记录一个软件的行为需求，那么被讨论的系统是指计算机程序。项目相关人员是指使用该程序的人、拥有该程序的公司、政府管理部门和其他一些计算机程序。主执行者是指坐在计算机屏幕前的用户或者另一个计算机系统。

一个编写良好的用例应该具有很好的可读性。它由多个句子组成，所有句子都采用同一种语法形式——一个简单的执行步骤。通过执行这些执行步骤，执行者或者获得一定结果或者向另一个执行者传递信息。想学会如何阅读用例是很容易的，无需太多时间。

然而，要学会编写一个好的用例却不容易。编写者必须掌握三个概念，这三个概念适用于整个用例和用例中的每一个句子。初看起来，强调这三个概念的重要性显得有点多余，但要自始至终用好这三个概念却不是一件容易的事。一旦开始编写用例，就会感到困难重重。这三个概念是：

- 范围（*scope*）：真正被讨论的系统是什么？
- 主执行者（*primary actor*）：谁有要实现的目标？
- 层次（*level*）：目标的层次是高，还是低？

下面将给出一些用例实例。用例的组成部分将在下一章介绍。首先，要记住下面总结的这些定义：

- 执行者（*actor*）：任何具有行为的人或物。
- 项目相关人员（*stakeholder*）：对被讨论系统(SuD)的行为有特定兴趣的人或物。
- 主执行者（*primary actor*）：启动与被讨论系统的一次交互活动，从而达到某一目标的人或物。
- 用例（*use case*）：规定被讨论系统行为的契约。
- 范围（*scope*）：界定被讨论的系统。

2

- 前置条件和保证（*precondition and guarantee*）：在用例执行之前和之后必须满足的条件。
- 主成功场景（*main success scenario*）：一切顺利的情况。
- 扩展（*extension*）：场景执行过程中出现的不同情况。
- 扩展中的编号是指在主成功场景中不同情况发生时所处的执行步骤号码（例如，步骤4a和步骤4b是指主成功场景中步骤4的两种不同情况）。
- 当一个用例引用另一个用例时，被引用的用例加下划线。

第一个用例描述了一个人准备通过万维网（Web）购买股票的情况。为了表示可以通过一次处理完成的目标，标记该用例处在“用户目标级”(*user-goal level*)，并用“海平面”符号（  ）来表示。第二个用例描述了一个人设法获得汽车交通事故的赔偿金，这个目标要花费较长的时间，不能通过一次处理完成。为了表示这一特点，标记该用例处于“概要级”(*summary level*)，并用“高出海平面”符号（  ）来表示。这些符号将在第5章中详细介绍，在内封前插页中已经总结了所有的符号。

第一个用例描述了一个人与一个程序（PAF）的交互活动，该程序运行在与万维网相连的工作站上。“黑盒”符号（  ）表示所讨论系统是一个计算机系统。第二个用例描述了一个人与一个公司的交互活动，用“建筑物”符号（  ）来表示公司。符号的使用完全可以根据个人的喜好来选择，但对范围和层次的标记却不是。

3

下面是用例1和用例2。

用例1  通过万维网购买股票 

主执行者：购买者
范围：私人顾问/金融包(Personal Advisors/Finance package,简称PAF)
层次：用户目标

项目相关人员和利益：

购买者——购买股票，并希望所买股票能自动被加到PAF记录中。

股票代理商——希望得到全部的购买信息。

前置条件：用户已经打开PAF。

最小保证：有足够的登录信息，以便当出现问题时，PAF能够检测到问题，并要求用户提供更详细的信息。

成功保证：远程web站点认可此次购买事件；日志和用户记录被更新。

主成功场景：

1. 购买者选择通过万维网来购买股票。
2. PAF从用户那里得到所用站点的名称（如E*Trade、Schwab等）。
3. PAF与该站点建立网络连接，并保持控制权。
4. 购买者在该站点上浏览并购买股票。
5. PAF截取站点的响应信息，并更新购买者的记录。
6. PAF向用户显示更新后的记录情况。

扩展：

2a. 购买者要使用一个PAF不支持的站点：

2a1. 系统从购买者那里获取新建议，带有取消用例的选项。

3a. 在设置过程中，网络发生故障：

3a1. 系统向购买者报告错误，并建议他退回到前一步。

3a2. 购买者或者退出此用例，或者重新再试。

4a. 计算机系统崩溃或者在交易过程中被关掉：

4a1.（这时，我们该怎么办？）

4b. Web站点没有及时认可此次购买活动，而是把它推迟处理：

4b1. PAF把这次推迟事件记入日志，设置一个时钟，定期向购买者询问结果。

5a. Web站点没有返回关于购买情况的必要信息：

5a1. PAF把缺少信息的事件记入日志，要求购买者更新存有疑问的交易。 4

用例2 汽车交通事故索赔

主执行者：索赔者

范围：保险公司（MyInsCo）

层次：概要

项目相关人员和利益：

索赔者——获得尽可能多的赔偿金。

MyInsCo——赔偿尽可能少的赔偿金。

保险部门——确保事件处理过程全部符合规定。

前置条件：无。

最小保证：MyInsCo将索赔申请和所有活动记录到日志中。
成功保证：索赔者和MyInsCo就赔偿金额达成协议；索赔者按协议获得赔偿金。
触发事件：索赔者提交申请。
主成功场景：
1. 索赔者提交申请，申请中包含证明事故属实的资料。
2. 保险公司验证索赔者保险单的有效性。
3. 保险公司指派代理人去调查事件是否属实。
4. 保险公司确定事件的所有细节是否在保险范围内。
5. 保险公司对索赔者作出赔偿，然后关闭文件。
扩展：
1a. 提交的资料不完整：
    1a1. 保险公司要求索赔者提供遗漏的资料。
    1a2. 索赔者提供遗漏的资料。
2a. 索赔者的保险单无效：
    2a1. 保险公司拒绝索赔申请，通知索赔者，并将此事记录在案，然后终止此次处理过程。
3a. 当时找不到代理人：
    3a1. （这种情况下，保险公司该怎么办？）
4a. 事故与基本保险条例不符：
    4a1. 保险公司拒绝索赔申请，通知索赔者，并将此事记录在案，然后终止此次处理过程。
4b. 事故与附带保险条例不符：
    4b1. 保险公司开始就保险金的数额问题与索赔者进行谈判。

本书中的大多数用例来自于实际项目，在选用用例时做得很慎重以免对其过多地润色（只是在必要时，给一些用例添加范围和层次标识）。希望读者见到的例子是在实际项目中被真正采用的实例，而不仅仅是在课堂上具有吸引力的示例。人们在多数情况下没有时间去把用例写得正式、完整、漂亮，而只是尽可能将其写得“充分”，这就足够了。在这里之所以要展示这些真实的例子，是因为即使在本书的指导下，你自己也很少去创建完美的用例。实际上，多数情况
5 下，就连我也不会写出完美的用例。

用例3是由挪威中央银行的Torfinn Aas为他的同事、用户代表和他自己写的一个用例。此用例描述了如何在不致于丢失数据的情况下修改一个表格的过程。作者给这个用例附加了业务语境，以表明计算机应用程序如何在一个工作日期间运行。这很有实际意义，因为这样就不用再单独写一个文件来描写业务过程。这样既不会影响任何人对用例的理解，又能给相关人员提供必要的信息。

用例3 对运到的包装箱进行登记

RA（Receiving Agent）——“接收代理”
RO（Registration Operator）——“登记员”

主执行者：RA
范围：夜间接收登记软件系统
层次：用户目标
主成功场景：
1) RA收到运输公司（TC）送来的包装箱（包装箱标记（ID）,带有包标记（ID）的包），并打开它。
2) RA验证包装箱标记是否与TC登记的标记相符。
3) RA可能要在传送人员的表单上签字。
4) RA在系统中登记包装箱的到达信息，存储以下数据：
RA 标记
日期，时间
包装箱的标记
运输公司
<人名? >
#包（连同包的标记? ）
<估计的值? >
5) RA从包装箱中取出包，放到车上，送往RO。
扩展：
2a) 包装箱标记与运输公司的标记不相符。
4a) 响起火警，使登记中断。
4b) 计算机瘫痪。
把钱放在桌子上等待计算机重启。
可变情况：
4' 用与不用个人标记（ID）。
4" 用与不用估计的值。
5' RA把包留在包装箱中。

6

1.2 你的用例不能作为我的用例

用例是一种编写形式，可应用于多种情况，例如：
- 用来描述一个业务工作过程。
- 用来集中讨论未来系统的需求问题，而不是需求描述。
- 作为系统的功能性需求。
- 将系统设计结果文档化。
- 可能应用在小型的、集中的工作组中，也可能应用在大型的或分散的工作组中。

每种情况下所提倡的编写风格都会稍有差别。下面是用例编写的主要派生形式，分别适用

于不同的编写目的：

- 一个集中的工作组在收集需求时，或一个大型工作组在讨论未来的需求时，编写用例的风格比较“随意”；相反，一个大型的或分散的工作组或是正规化的工作组在收集需求时，编写用例的风格就会“完整正式”。随意的形式“简化”了用例模板，这样用例的编写就会很快（详情待续）。用例1到用例3都是“完整正式”的例子，它们采用了完整的用例模板以及按执行步骤编号的方案。非正式（随意）的例子参见用例4。
- 业务过程人员编写业务用例来描述业务操作；而硬件或软件开发人员编写系统用例来描述需求。设计人员可能会另外编写一些系统用例来记录他们的设计结果，或者用来进一步分解子系统的需求。
- 根据当时所需视图的层次，用例编写者在编写用例时有以下几种选择：描写一个经多次处理才能达到的目标，或称为概要目标；描写一个经一次处理就可以完成的目标（或称为用户目标）；描写用户目标的一部分（或称为子功能）。恰当地表达出某个用例究竟描写的是哪种情况是非常重要的，因此我的学生们给出两种不同的“梯度”对此加以描述：一种是以与海平面的相对高度（海平面以上、海平面和海平面以下）来描述；另一种是以颜色（白色、蓝色和靛青色）来描述。
- 任何一个为将要设计的新系统（不管是业务过程还是计算机系统）编写需求的人，都会编写黑盒用例——这种用例不关心系统的内部细节。同时，业务过程设计者编写白盒用例来描述公司或组织机构的内部过程如何运作。技术开发人员编写白盒用例来记录他们将要设计系统的运行环境；或者编写白盒用例来记录他们已设计系统的工作情况。

7

用例编写形式可以应用于如此多的情况，一方面让人感到高兴；但另一方面，也会给人造成迷惑。几个人凑在一起时，常常会发现彼此对用例编写的某些方面有着不同的看法，这是由于大家编写用例的目的互不相同而造成的。此外，在工作中也可能会遇到需要综合运用以上几种形式的情况。

要找到一种通用的方法来讨论用例，而同时又要突出所有这些不同情况各自的特点，在本书中，这个问题自始至终都是困扰我们的一个难题。目前所能采用的最好方法就是：首先概括地提出问题，随后通过例子本身来说明情况。

接下来，可以用本章的例子测试一下自己的理解情况。用例1、用例3和用例5是为系统需求目的而编写的，因此它们是完整正式的、黑盒式的、系统类型的用例，属于用户目标级。用例4大致相同，但它比较随意，不是完整正式的。用例2是为了记录业务过程而编写的一个语境设置用例，它是完整正式的、黑盒式的用例，属于概要级。

用例各种格式间的最大区别是怎样“雕琢”它们。考虑下面几种非常不同的情况：

- 小组人员正在为一个重要的大型项目开发软件。他们认为正规的形式所带来的额外花费是值得的，因此决定：(a)用例模板需要更长、更详细一些；(b)编写小组所有成员要采用相同的编写风格，以减少二义性和误解；(c)用例审查要更严格、更仔细地进行，以避免遗漏和二义性。由于项目不能容许错误的存在，因此他们还决定在用例编写方面也减少容许出现的偏差（各人依情况不同，程度各异）。
- 一个三到五人的小组正在建造一个系统，对这个系统来说，最坏的情况也不过是使用上

的不舒适，这可以通过一个电话很容易弥补。他们认为所有形式上的东西都只不过是在浪费时间、精力和金钱。因此，他们选择(a)一个简单的模板；(b)在编写风格上允许有更多的差异；(c)用例审查的容忍度可大可小。用例编写中的错误和遗漏会被其他项目机制捕获，甚至可能在小组成员间以及小组成员和用户间的谈话中捕获。他们可以容忍在文档交流中存在更多的错误，因此，他们的编写形式更随意，不同人员所采用的风格差异更大。

以上两种情况都是对的，这是由各个项目本身的特点所决定的。这是我作为一个方法学研究者在过去五年中学到的最为重要的一课。当然，多年来我们一直在说："没有放之四海而皆准的真理"，但是方法学研究者们至今仍难以将这句话转变成一个实实在在的建议。

错误的做法是，在不需要十分精确和严格定义的情况下仍然耗费项目组大量的时间和精力来编写用例。正如Jim Sawyer在一个email讨论中写到的："只要你没有感觉到模板使你深陷于无穷尽的细化而不可自拔，以至于使你的设计空间被一点点吞噬……。但如果这种情况开始发生，我建议丢弃那些小的明显的错误，然后重新描述用例情节，甚至把草稿写在餐巾纸上。"

我逐渐得出如下结论：只有一个用例模板是不够的。至少要有两个用例模板：一个是非正式的（或称随意的），在要求不严的项目中使用；另一个是完整正式的，在要求严格的项目中使用。任何项目都要根据自己的实际情况对其中之一加以调整。下面两个用例就是分别用这两种风格编写而成的同一个用例。

用例4 买东西（非正式版本）

请求者发起一个请求，并把这个请求送给他（或她）的批准者。批准者首先检查预算中是否还有资金，然后核实货物的价格，接着完成提交请求，并将请求发送给买者。买者查阅仓库目录，找出最好的货物供应商。认证者验证批准者的签名。买者完成订购请求的各项工作，向供货商发出PO（订单）。供货商把货物发送给接收者，并得到发货收据（这一点超出了本系统的设计范围）。接收者记录交货情况，并把货物发送给请求者。请求者设置请求已被满足标志。

在获得货物前的任意时刻，请求者都可以修改或取消请求。取消请求意味着把此请求从任何执行处理中取消（从系统中删除它吗？）。降低货物价格不影响对其进行的处理过程；提高价格则需要将请求重新发回给批准者。

用例5 买东西（完整正式版本）

主执行者：请求者
语境中的目标：请求者通过系统买东西，并得到所买的东西。不包括付款方面的内容。
范围：业务——整个购买机制，包括电子的和非电子的，正如人们在公司中所见到的一样。
层次：概要
项目相关人员和利益：

请求者：希望得到他/她订购的东西，并且操作要简单。
公司：希望控制花费，但允许必要的购买。
供货商：希望得到任何已发货物的货款。
前置条件：无
最小保证：每一个发出的订单都已经获得有效认证者的许可。订单具有可跟踪性，以便公司只对收到的有效货物开账单。
成功保证：请求者得到货物，修改预算，记入借方。
触发事件：请求者决定买东西。
主成功场景：
1. 请求者：发起一个请求。
9
2. 批准者：检查预算中的资金，检查货物的价格，完成提交请求。
3. 买者：检查仓库的存货，找出最好的供货商。
4. 认证者：验证批准者的签名。
5. 买者：完成订购请求，向供货商发出PO（订单）。
6. 供货商：把货物发送给接收者，得到发货收据（这一点超出了本系统的设计范围）。
7. 接收者：记录发货情况；向请求者发送货物。
8. 请求者：设置请求已被满足标志。

扩展：
1a) 请求者不知道供货商和货物价格：不填写这些内容，然后继续。
1b) 在收到货物之前的任意时刻，请求者都可以修改或取消请求:
　　如果取消，则把这个请求从执行处理中取消。（从系统中删除吗？）
　　如果降低价格，则不影响其处理过程。
　　如果提高价格，则把请求送回批准者。
2a. 批准者不知道供货商或货物价格：不填写这些内容，留待买者填写或返回。
2b. 批准者不是请求者的经理：只要批准者签名仍然可行。
2c. 批准者拒绝申请：送回给请求者，要其修改或删除。
3a. 买者在仓库中找到货物：将存货先发出，并从申请者要求的总购买量中减去已经发出的这部分货物量，然后继续。
3b. 买者填写在前面活动中没有填写的供货商和价格信息：请求重新发回给批准者。
4a. 认证者拒绝批准者：发回请求者，并且将此请求从执行处理中取消。（这样做意味着什么？）
5a. 请求涉及到多个供货商：买者创建多个PO（订单）。
5b. 买者将多个请求合并：相同的过程，但是用被合并的请求标记PO（订单）。
6a. 供货商没有按时发货：系统发出没有发货警告。
7a. 部分发货：接收者在PO（订单）上做部分发货标记，然后继续。
7b. 多个请求PO（订单）的部分发货：接收者给每个请求分配货物数量，然后继续。
8a. 货物不对或质量不合格：请求者拒绝接收所发送的货物。（这意味着什么？）

8b. 请求者已经离开公司：买者同请求者的经理进行核实，或者重新指派申请者，或者返还货物并取消请求。

技术和数据变动列表：无

优先级：多种

发行版本：几个

响应时间：多样

使用频率：3/天

主执行者的渠道：网络浏览器、邮件系统或类似系统

次要执行者：供货商

次要执行者的渠道：传真、电话或汽车

未解决的问题：

什么时候从系统中删除被取消的请求?

要取消一个请求需要哪些权限?

谁能修改一个请求的内容?

请求中需要保留哪些修改历史记录?

当请求者拒绝已经发送的货物时，会发生什么情况?

申请和订货在运作上有什么不同?

订购如何参考和使用内部存货?

10

通过以上举例希望大家明白，只是简单地说“我们写这个项目的用例”没有说明多少问题，而且任何只是简单地说“写用例”这样的建议或过程定义都是不完整的。在一个项目中有效的用例在另一个项目中就不一定有效。应该更详细地说明用例是正式的，还是非正式的，模板中哪些部分和格式是必要的，以及用例编写者之间在风格上允许有多大的差异。

关于项目间差异的容许程度问题在《*Software Development as a Cooperation Game*》(Cockburn, 2001)中进行了全面的讨论。我们不需要进行全面的讨论，就可以学习如何编写用例。我们确实有必要把用例的编写技巧同用例质量和项目标准区别开来。

“技巧”是人们在构建用例时所应用的一种瞬间的思想或动作。本书主要关心技巧问题，即如何构思、如何组织语句、以何种流程进行工作。用例编写技巧很大程度上与项目的规模无关，这倒是件值得庆幸的事情。因此，一旦掌握了用例编写技巧，即可将其应用于各种规模的项目中。

“质量”是指如何判别一个写完的用例是否与其目标相符。本书描述了我们所知道的关于如何编写用例的每个部分、整个用例、以及如何根据不同目的编写用例的最佳方法。但是，最后需要指出，评价用例质量的方法与用例编写的目的、准确性以及所选择的正规程度有关。

“标准”是指在编写用例时项目的参与者就某些问题达成了一致意见。本书中讨论了几种不同的合理标准，以展示不同的模板、不同的句子和标题风格。本书中还提出了几种具体的建议，但是最终我还是提倡根据组织或项目建立或修改这些标准，并决定如何强制性地执行这些标准。

本书中的大部分内容涉及一个最重要的问题——编写准确的需求。在下面的这个真实报告
11 中，Steve Adolph顾问描述了使用用例来发现需求而不是用文档方式记录需求。

◆ Steve Adolph: 在新领域中“发现”需求

用例通常是作为一种捕获需求和对已知的功能性需求进行建模的方法而被使用。人们发现叙述形式的需求描述比传统的商品列表形式更易于理解。人们实际上知道系统应该做什么。

但是，如果没有人知道系统应该做什么怎么办？某个过程的自动化通常会改变该过程。近来，由于胶版印刷技术——直接制版和直接印刷技术的发展——的发明，打印工业经历了一次最大的变革。在此之前，建立打印印刷是一个劳动密集型的、多工序的过程。直接制版和直接印刷使得产业化的印刷工作就如同向字处理器提交打印文档一样简单。

如果你作为负责这个直接制版系统工作流管理的分析师，你应该怎样为一个全新的系统收集需求呢？你可以首先找出现有系统中的用例，然后识别出系统的执行者和服务。但是，这仅仅让你得到现有系统的信息。还没有人对新工作进行过尝试，因此所有的领域专家都和你一样在学习新系统。你在设计新软件的同时，也在设计新过程。祝你好运！你怎样才能从如此洁白的新雪地上发现一点蛛丝马迹呢？审查现有的模型，然后提出如下疑问：“什么东西发生了改变？”答案恰恰是“所有的东西”。

通过编写用例来记录需求时，已经有人创建出了系统的一个版本。这时只是简单地对这个版本进行描述，以便使所有人都能清楚地理解它。然而，在“发现”需求的过程中，却是你在创建这个版本。

把用例作为一种“抛砖引玉”的工具。问这样一个问题“如果使用新技术，这个用例中哪些步骤对实现用例规定的目标不再有任何价值？”。创建一个新故事来帮助执行者达到目标。目标还是一样的，但其中一些曾参与活动的执行者已经不复存在或者已经发生了改变。

使用“深入浅出”方法。创建一个广义的高层模型把想像中的系统工作情况描绘出来。要尽量保持模型的简单性，因为这毕竟是一个新领域。先描绘出主成功场景可能是一种什么样的情况。然后，与先前的领域专家一起走查这个场景。

接下来，深入到一个用例的细节当中，考虑此用例的其他不同场景。一个很有帮助的事实是人们能够很容易地理解一个故事，能够在一定基础上大量补充被遗漏的需求。逐个研读用例的每一步，然后考虑如下问题：“如果用户想要一个硬拷贝的而不是数字拷贝的试样，那该怎么办？”这比试图在头脑中组装出一个系统如何进行工作的完整模型要容易得多。

最后，回到表面。既然已经对用例的内部细节进行了深入了解，那么哪些地方作了改动呢？根据所作的改动相应地修改模型，然后深入到另一个用例的细节当中。

我的经验是利用用例来“发现”需求会得到高质量的功能需求。它们具有较好的组织
12 结构，并且也更加完整。

1.3 需求和用例

如果把用例作为需求来编写，那么请谨记以下两点：

- 用例确实是需求。不必将用例转变成行为需求的其他形式。如果编写恰当，用例可以准确地对系统必须要做什么进行详细的描述。
- 用例不是所有的需求。用例不详细地描述外部接口、数据格式、业务规则和复杂公式。用例只是需要收集的所有需求中的一部分（可能有1/3），虽然这一部分是非常重要的一部分，但毕竟仅仅是“一部分”。

任何组织收集需求都是为了满足自身的需要。甚至制定了关于如何描述需求的标准。但在任何一个标准中，用例仅仅是所有需求文档中的一部分。

下面的需求大纲是比较有用的大纲之一。它是通过对Suzanne Robertson和Atlantic Systems Guild 发表在他们的网站和《*Managing Requirements*》(Robertson和Robertson,1999)一书中的模板修改而来。他们的模板以其完备性而令人生畏，因此将其简化成下面大纲中所示的形式。本书以此模板作为指导方针。就我所见过的大多数项目而言，这个模板仍然过大，因此可以在需要时对它进一步简化。且不管它的规模如何，其价值在于：它提出了许多有意义的问题，如果没有这个模板，这些问题是不会被发现的。例如，“人们为应对系统崩溃而做的备份是什么？”，“哪些政治因素影响了需求？”。

虽然我遇到的许多人从来没有见过需求大纲，但使交付的需求标准化并不是本书的责任。这一问题留给读者去考虑。本书的主要目的是阐明用例在整体需求中的位置，此外，希望大家明白用例并不是全部需求，而只是描述需求中的行为部分，即必需的功能。

一个可行的需求大纲

第1章 目的和范围
- 1a. 整体范围和目标是什么?
- 1b. 项目相关人员（谁关心？）
- 1c. 什么在范围之内，什么在范围之外?

第2章 使用的术语/词汇

第3章 用例
- 2a. 主执行者及其总体目标
- 2b. 业务用例（操作概念）
- 2c. 系统用例

第4章 采用的技术
- 4a. 这个系统有什么技术需求?
- 4b. 这个系统会与哪些系统发生交互，其需求是什么?

第5章 其他需求
- 5a. 开发过程
  - Q1. 哪些人是项目参与者?

Q2. 项目的价值反映在哪些方面（简单、及时、迅速或灵活）？
Q3. 用户或出资人希望得到什么反馈或项目可见性？
Q4. 什么是可以买到的，什么是我们必须要创建的，我们在哪些方面是有竞争的？
Q5. 还有什么其他的过程需求（如测试、安装等）？
Q6. 项目运行依赖哪些条件？

5b. 业务规则
5c. 性能
5d. 操作、安全、文档
5e. 使用和可用性
5f. 维护和可移植性
5g. 还未解决的问题和推迟解决的问题

第6章　人工备份、法律性、政治性和组织性问题

Q1. 为系统操作所作的人工备份是什么？
Q2. 有什么法律性和政治性的需求？
Q3. 这个系统完成后对人们的影响是什么？
Q4. 有哪些培训需求？
Q5. 对人类环境有哪些假设和依赖性？

需要强调的是，用例仅仅是需求文档中第3章的内容，而不是所有的需求——它们仅仅是行为需求，并且是所有的行为需求。业务规则、词汇表、性能目标、过程需求和许多其他方面的东西都不属行为需求之列，它们需要在自己特定的章节中进行介绍(见图1-1)。

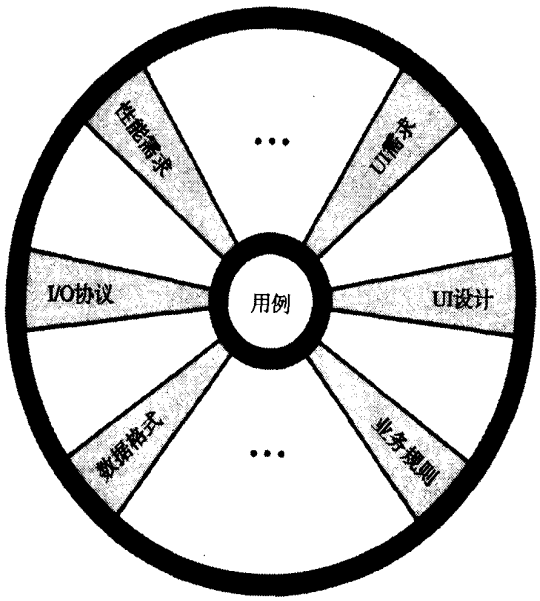


图1-1　“轮轴和轮辐”需求模型

1.3.1 用例作为项目连接结构

用例把许多其他需求细节连接在一起，为连接需求中不同部分的信息提供了一个框架，用例还有助于用户概要信息、业务规则和数据格式需求等内容的交叉连接。

在需求文档之外，用例有助于组织项目计划信息，如发布时间、小组、优先权和开发状态。此外，用例还有助于设计小组跟踪某些结果，尤其是用户界面的设计和系统测试。

虽然所有这些需求没有被包含在用例中，但它们都与用例有关。如图1-1所示，用例是轮子的轴心，其他信息作为轮辐，分别指向不同的方向。正是由于这个原因，人们才认为用例是需求的核心元素，甚至是项目开发过程中的核心元素。

1.4 用例的增值点

用例之所以被广泛采用的主要原因是，用例详细地描述了系统被使用时的行为细节，使得用户能够明白新系统到底是什么样的。这有利于用户尽早对系统运行的细节作出判断：是欣然接收还是拒绝(“你的意思是我们必须这样做？”)。然而，这只是用例的作用之一，而且可能不是其最重要的作用。

用例的首要的增值作用体现在对系统目标的描述，此时用例被冠以系统目标之名，并被收集整理成一个列表。这个列表声明了系统可以做什么，揭示了系统的范围以及创建系统的目的。它成为项目不同的项目相关人员之间互相交流的一个工具。

用户代表、主管、资深开发人员和项目经理会对目标列表进行仔细的研究，并由此对项目的费用和复杂性作出初步的估算。然后，他们之间互相协商先开发哪些功能、如何组建开发小组。这个列表可以作为一个进行复杂性、经费、时间和状态测量的框架，它收集了项目生命周期中各种各样的信息。 14 ~ 15

用例的第二个增值作用体现在对异常情况处理的描述上。此时用例编写者集中讨论主成功场景中可能发生的所有异常情况，并组织成一个列表，同时开始记录系统相应的应对措施。通过该表，开发小组能够发现一些开发人员自己或需求提供方以前没有考虑到的意外情况。

在写用例写烦了的时候，仍然要坚持直至遇到失败的情况。在编写错误处理文档时，经常能够发现新的项目相关人员、系统、目标和业务规则。而当我们考虑异常情况处理方法时，总是要和业务专家聚在一起，或通过电话来商量以决定系统在此种情况下该如何运作。

如果没有这些离散的用例步骤和对失败情况所作的集中讨论，那么许多错误情况就不能在程序员编写代码段之前被发现。如果错误在程序员编写代码时才被发现，这对于发现新的功能和业务规则来说就太晚了。那时，业务专家通常已经走了，时间也已经十分紧迫，因此，程序员就只能想当然地按自己的想法编写代码，而不再去努力寻求更理想的解决办法。

人们只要写出一段用例，就能通过少量的编写而节省大量的时间，并从用例中获得好处。人们只要坚持作错误处理，就能通过早期发现那些难以捉摸的需求而节省大量的时间。

1.5 合理安排你的精力

节省你的精力。或者至少应该有效和合理地安排你的精力。如果在刚开始写用例时就深入到所有的细节，就不能及时有效地在主题层面上考虑问题。而如果在开始时写出一个大纲，然后只写出每个用例的实质性内容，则能：

- 给项目相关人员一个修改和深入考虑用例优先级的机会。
- 使得工作可以分解到多个小组，提高工作的并行性和效率。

人们常说“给我高层视图”、“只把大纲给我”或“我们打算在以后添加细节”，其含义是
16 “目前的工作可以不必太精细；我们可以在以后把更多更细的内容添加进去”。

“精确度”（*precision*）是指对所关心内容的表达程度。当声明“一个客户想要租一个录像带”时，并没有使用很多的词汇，但确实给读者传达了很多的信息。当给出未来系统所要支持的所有目标的一个列表时，尽量用最少的词语向项目相关人员提供最多的信息。

精确度和准确性不是一回事。如果有人说“π是3.141592”，他的说法已经采用很高的精确度，但远远没有达到准确性。如果说“π约为3”，则采用的精确度很低（小数位的个数很少），但是该说法却是准确的。对于用例也是同样的道理。

最终为每个用例添加细节，增加其精确度。如果在刚开始时对目标的低精确度描述恰恰就是错误的（不准确的），那么在实现高精度的描述中所投入的所有精力就白费了。在为了使用例全面细化而投入更多的精力之前，最好保证目标列表的正确性。

根据所需精力的多少和每个阶段后的停顿，我们将编写用例所需的工作量划分为四个精确度等级：

1) 执行者和目标（*actor and goal*）：列出系统所支持的执行者及其目标。审查这个列表的正确性和完整性。划分目标的优先级，并将其分配到各小组和不同的发布版本中。至此，可获得具有初级精确度的功能需求。

2) 用例概述和主成功场景（*use case brief or main success scenario*）。对于所选出的需要进一步细化的用例，勾画出其主成功场景。审查这些草图以确保系统能够真正表达我们所关心的项目相关人员的利益。这达到了功能需求的二级精确度。与下面两级不同，这些内容是相当容易被勾画出来的。

3) 失败情况（*failure condition*）。完成主成功场景，并集中讨论所有可能发生的失败情况。在想出系统如何处理这些失败情况的办法之前，首先拟定一个完整的失败情况列表。设想出系统处理失败情况的方案要比仅仅列出这些失败情况需要更多的精力。如果一开始就立即着手写失败情况处理方案，则常常会在列出所有的失败情况之前就耗尽了所有的精力。

4) 失败情况处理（*failure handling*）。写出系统应该如何对这些失败情况作出反应。这经常是棘手的、累人的工作，但同时也是令人惊喜的工作。说它令人惊喜是因为多数情况下，在编写失败情况处理方案的过程中，对于一个模糊的业务规则的疑问会得到澄清，或者一个失败处理会突然展示出系统需要支持的新执行者和新目标。

多数项目都时间紧迫、精力有限。因此，合理安排用例编写精确度应该是一个项目优先考虑的事情。极力建议大家按上述顺序进行工作。

1.6 先用一个系统使用叙述热身

系统使用叙述（*usage narrative*）是一个处于运行状态的用例实例——一个极为具体的执行者使用系统的例子。它不是用例，并且在许多项目中，它不被记入正式需求文档中。然而，它是一个非常有用的工具，对它进行描述和编写是完全值得的。 17

在刚启动一个新的项目时，开发人员或业务专家可能缺乏用例编写方面的经验，也可能还没有设想出系统详细的操作情况。为了了解系统的具体情况，先勾画一个插图（*vignette*），即先勾画出一个执行者日常生活中的某些瞬间的情况。

在这个叙述中，编造一个虚构但具体的执行者，然后试图去捕捉这个人的心理状态——他为什么想要这样做，或者什么条件驱使他那样去做。对于所有的用例编写，不必写得太多。简明扼要的几句话所能表达的信息量常常是惊人的。在这个具体的事例中，捕获从开始到结束整个过程中的运作情况。

简明性很重要，因为读者可以很快地了解整个故事。细节和动机或者情感内容也是重要的，它们使得每一个读者（包括需求审核人员、软件设计人员、测试用例编写人员以及培训材料作者）都能清楚如何优化系统以使用户获得更多的益处。

系统使用叙述易于阅读，篇幅也很小，但它能很容易地、循序渐进地把读者引入到用例当中。下面是一个例子。

系统使用叙述：获得“快速兑现”

玛丽在上班的路上顺便带她的两个女儿去日托中心。她开车到达ATM机，把她的卡插入读卡口，输入PIN号码，然后选择“快速兑现”（FAST CASH），并输入金额$35。ATM机输出一个$20和三个$5的现金，外加一个显示她取出$35后账号余额的收据。每完成一次快速兑现交易，ATM机就重置其屏幕，这样玛丽就可以开车离开而不用担心下一个司机会进入她的账号。玛丽喜欢快速兑现的原因是它避免了许多减慢交互速度的问题。她到这个特殊的ATM机取钱的原因是这个ATM机能够兑换出$5的现金钞票，她要用$5来付日托的费用，此外，她不必下车就可以使用这个ATM机。

人们编写系统使用叙述是为了帮助设想正在使用的系统的行为。同时也可用它作为写用例和进入细节之前的热身活动。有时，某些小组在所有用例的开头或者只在所描述的某个用例前增加一段叙述。例如，一个小组写道：他们把用户、分析员和需求编写人员聚集到一起，来模仿叙述中的情节，以便帮助他们确定系统的边界，并建立起系统使用情况的一个一致观点。

叙述不是需求；它为更详细、更广泛地描述需求进行了铺垫。叙述锁定了用例。用例本身是一个“没有水分”的叙述——一个公式——用例中执行者的名称是通用的，而叙述中使用的名称是真实的。 18

1.7 练习

需求文件

1-1 需求文件概要中哪些部分对用例来说是敏感的，哪些部分对用例来说不是敏感的？同其他人讨论此问题，思考为什么会得出不同的答案。

1-2 设计一个合理的需求概要，使其可用于以HTML连接的企业内部网中。注意子目录结构和时间戳的约定（为什么会需要时间戳的约定？）

使用叙述

1-3 为ATM写两个使用叙述。它们什么地方与叙述实例不同，为什么会不同？这些不同之处对于要设计此系统的设计者来说有多重要？

1-4 编写一个关于一个人去一家新开的录像带出租商店的使用叙述，侧重于租借原版“*The Parent Trap*”录像带的叙述。

1-5 编写一个关于你的当前项目的使用叙述。让另一个人也来对同一个情况编写一个使用叙述。进行比较并讨论。它们为什么不同？对于这些不同，你所关心的是什么——是操作容许的差异还是不能忽视的重大差异？

第一部分

用例体部分

第2章 用例是规范行为的契约

被讨论系统（SuD）是在不同项目相关人员间制定契约的一种机制。用例构成了契约中的行为部分。用例中的每个句子都有其存在的价值，即它们各自描述了一项行为，该行为维护或增进了某些项目相关人员的利益。有的句子也可能描述两个执行者之间的一种交互，或者描述系统为了维护项目相关人员的利益所必须采取的一个内部动作。

首先，仅从捕获（具有某种目标的）执行者之间的交互行为的角度来考察一个用例。当把这点搞明白之后，可以进一步扩充讨论的内容，直到用例能被用作项目相关人员间协调各自利益的契约。第一部分称为“**执行者和目标**”概念模型，第二部分称为“**项目相关人员和利益**”概念模型。

2.1 具有目标的执行者之间的交互

2.1.1 执行者具有目标

假设有一个职员负责处理电话服务请求（在图2-1中，这个职员是主执行者）。当有电话打进来请求服务时，该职员就产生了一个目标：让计算机注册并启动这个请求。

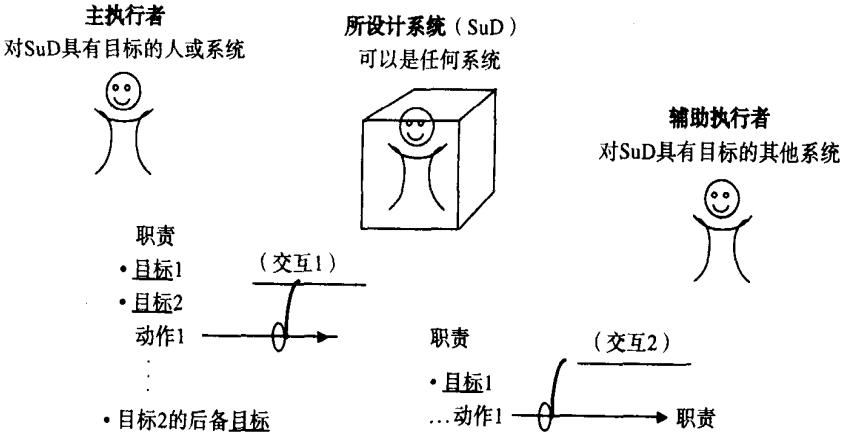


图2-1 一个具有目标的执行者请求另一个执行者履行职责

在本例中，系统也有一个职责：注册和启动这个服务请求。（实际上，系统有责任维护所有项目相关人员的利益，职员（主执行者）只是项目相关人员之一。目前只将讨论集中在向主执行者提供服务这一系统职责上。）

为了实现其职责，系统制定出一些子目标。系统可以内部实现一些子目标，但其他子目标

需要借助“辅助执行者”（*supporting actor*）来实现。这些辅助执行者可以是打印子系统，也可以是其他组织，如合作单位或政府代理等。

辅助执行者通常会履行自己的承诺，并将子目标的完成结果交付给被讨论系统(SuD)。而SuD与外部执行者进行交互。它按一定次序陆续实现其子目标，直到最终交付其职责，即它所承诺的服务。

交付服务承诺是一个最高目标，它通过子目标来实现。子目标可以进一步分解为子-子目标，子-子目标又可以进一步分解，如此可以无限细分下去。如果想把执行者的行为分解得足够精细，那么这样的子-子-(……子)目标就没有一个可能的终点。正如爱尔兰讽刺作家兼诗人Jonathan Swift在“诗，一部狂想曲”中写到的那样（这首诗的内容不是关于用例的，但是用来形容用例也非常形象）：

> 博物学家由此发现，
> 一只跳蚤被小跳蚤啮食，
> 小跳蚤又被更小的跳蚤噬咬，
> 如此往复，永无止日。

可能，要编写出好的用例最困难的就是控制“啮食跳蚤的跳蚤”，也就是写作中的子-子目标。关于这一点，本书其他章节有更多的描述，如第5章“三个命名的目标层次”，第7章的准则6“包含‘合理’的活动集”，以及第20章的提示6“正确地得到目标层”等。

执行者和目标概念模型是很容易使用的，因为它同样适用于业务系统和计算机系统。执行者可以是单个人员、组织或者计算机。执行者和目标概念模型可以用来描述由人、公司和计算机组成的混合系统；也可以用来描述一个软件系统（该系统由另一个需要人类辅助执行者参与的计算机系统驱动，或由需要计算机系统参与的一个组织驱动，或由个人驱动）。执行者和目标概念模型是一个有用的通用模型。

23～24

2.1.2 目标可能失败

如果在记下请求的过程中计算机崩溃了，那么这个通过电话为客户服务的职员该怎么办？如果系统不能交付它的服务承诺，该职员必须建立一个后备目标——在这种情况下，可能是使用纸和铅笔。职员还有一个主要的工作职责，即必须有一个计划以防系统不能履行其职责。

相似地，系统可能会在执行某个子目标时遭遇失败。可能主执行者传送了错误的数据，也可能是一个内部失败，或者辅助执行者没能交付他所承诺的服务。那么，系统该如何运作呢？这是所讨论的系统行为需求中一个真正有意义的部分。

在某些情况下，系统能够修复错误，使正常的行为序列得以继续。而另一些情况下，它只能放弃这个目标。例如：到ATM提取现金时，如果试图提取的现金额超过了有权提取的金额，那么提取现金的目标就会失败。如果ATM与网络计算机的连接断掉了，目标也会失败。不过，如果只是输入了错误的个人密码，那么系统会再提供一次重新输入正确密码的机会。

强调目标失败和失败反应是用例通常能够进行良好的系统行为描述和出色的功能需求描述的原因之一。已经做过功能分解和数据流分解的人认为这是他们从用例中能够获得的最重要的好处。

2.1.3 交互是复合的

最简单的交互是发送一条消息。当两个人在大厅中相遇，一个人对另一个人说“你好，珍妮，”——这是一个简单的交互。在过程化程度设计中，这样的简单交互相当于一个函数调用，如print(value)。在面向对象的程度设计中，这样的交互相当于一个对象向另一个对象发送一条消息，如objectA->print(value)。

一个消息序列（*sequence of messages*）或场景（*scenario*）是一个复合的交互。假设珍妮走到一个饮料机前，想买一桶价值80美分的饮料，珍妮放入一张1美元的钞票，机器告知珍妮必须放入与饮料刚好等价的钱币。珍妮与机器间的交互如下：

1) 珍妮插入1美元的钞票。
2) 珍妮按“可乐”键。
3) 机器告知“必须放入与饮料刚好等价的钱币”。
4) 珍妮咒骂了一句，然后按“退币”按钮。
5) 机器退回价值1美元的硬币。
6) 珍妮收起硬币（然后嘟囔着走开）。

25

可以对一个活动序列进行压缩，把它作为一个单独的步骤（“珍妮试图通过一个自动售货机购买一桶可口可乐，但是机器要求必须放入与饮料刚好等价的钱币”）来对待，然后又可以进一步把它插入到一个更大的活动序列当中：

1) 珍妮去公司银行，取了一些钱。
2) 珍妮试图通过一个自动售货机购买一桶可口可乐，但是机器要求必须放入与饮料刚好等价的钱币。
3) 珍妮（只好）漫步到一个咖啡厅，并从那里买了一桶可口可乐。

因此，交互可以根据需要被折叠或分解，就像目标能大能小一样。场景中的每一步都捕获一个目标，因此每一步都可以被展开，而自成为一个单独的用例。看来交互也是“跳蚤上有跳蚤”，正如目标中又有子目标一样。

令人高兴的是，通过对目标和交互进行折叠，可以在一个很高的层次上来表示系统的行为。而通过对它们进行一点一点地逐步展开，也能够对系统行为尽可能进行精确的刻画。用例常常被看作是一个可以不断展开下去的故事。用例编写者的工作就是把故事编写得情节连贯、条理清楚，以使读者可以很舒适地在故事情节中切换、穿梭。

机敏的读者也许会发现，本书使用“序列”（*sequence*）一词时显得非常不严密。在许多情况下，交互过程没有必要非得按照一个特定的顺序发生。要买一桶价值80美分的饮料，可以放入8个10美分的硬币，也可以放入3个各价值25美分的硬币和1个5美分的硬币，也可以采用其他组合形式（读者如果有兴趣，可以把其他形式列举出来）。至于哪个硬币先被放入是无关紧要的。

正式地讲，“序列”一词用得并不确切。正确的数学用语应该是“偏序”（*partial ordering*）。然而，“序列”说起来比“偏序”要短[⊖]，而且意义也比较接近，因而编写用例的人更容易接受

⊖ 这里的“短”是指，在英语原文中，术语“序列”（sequence）比术语“偏序”（partial ordering）要短。——译者注

“序列”的说法。如果有人提出关于并行消息的问题，完全可以说，“好啊，那就写一点关于这方面的内容吧”，接下来再看他们到底能写出什么样的东西。根据我的经验，几乎不必接受什么训练，人们就可以成功地写出很清晰的描述。因此，本书中一直采用“序列”的说法。参见第5章中用例22“损失登记”，这个用例的活动序列比较复杂一些。

如果有人试图创建一种正式的语言来编写用例，那么他就会很容易地陷入到困境当中。许多语言设计者不是强迫编写者列出所有可能的次序，就是发明出复杂的符号以允许事件以任意次序排列。由于编写用例是为了给其他人而不是计算机来阅读，因此编写者应该感到很幸运。他们只需简单地写成“购买者放入80美分，可以是5美分的硬币、10美分的硬币或25美分的硬币的任意排列组合形式”，别人一般很容易地理解其含义。

活动序列很适合于描述发生在过去的交互过程，因为“过去”是已经完全确定的。要描述发生在将来的交互过程，需要有一个所有可能出现的活动序列组成的集合，每一个可能的活动序列就代表在现实世界中将来可能发生的一种情况。如果一个人陈述昨天他要求老板给他涨工资时的情景，可描述为：

26　“今天，我跟老板进行了一场严肃的交谈：我说‘……’，她说‘……’，我说‘……’，等等。”

但是，若是谈将来的事情，则必须这样描述：

“我对即将要与老板进行的下次交谈真的感到很紧张。”

“为什么？”

“我打算要求老板给我涨工资。”

“那你打算怎么跟老板说？”

“唔，首先我打算说‘……’，然后，如果她说‘……’，我就以‘……’作为应对，但是如果她说‘……’，那我就试着这样说‘……’，等等。”

相似地，如果要告诉另一个人如何买苏打饮料，要这样说：

“首先，把你的钱准备好。”

“如果你准备的钱正好，那就把它放入售货机，按‘可口可乐’按钮。”

“如果你准备的钱稍多了一点，把它放入售货机后，等一会，看它是否能找给你一些零钱。如果它把零钱找给你……”。

要描述将来的一次交互过程，就必须要对不同的情况进行处理，创建出一个交互序列集。对于每一个交互序列，说明它发生的条件，然后描述交互的发生情况，最后给出交互产生的结果。

可以把一个交互序列集折叠成一个单一的陈述，即“首先走过去，从机器中买一桶饮料”，或者说“于是你要求你的老板给你涨工资”。就序列而言，可以把这些陈述折叠成简洁、高层的描述，也可以根据需要把它们展开成详细的描述。

至此，我们已经了解到用例包含了到达一个目标可能出现的所有场景的集合。为了更加完善，需要加入以下内容：

- 与同一个主执行者的同一个目标有关的所有交互过程。

- 用例由触发事件启动后开始执行，直到目标被实现或者被取消而结束，系统通过本次交互完成它的职责。

2.1.4 用例聚集场景

主执行者有一个目标；系统应该帮助主执行者实现这个目标。一些场景描述了目标被实现的情况；而另一些场景以目标被取消为结束。每一个场景都包含一系列的执行步骤，以表明动作和交互过程是如何被一步步展开的。用例把所有的场景聚集到一起，显示了一个目标获得成功或遭遇失败的所有可能的途径。

对此，一个有意义的比喻是通过一个“条形裤”图案（见图2-2）来阐明的。裤子上的条形带指明了把所有场景聚拢到一起的目标。在这两条裤腿中，一条裤腿代表成功结束的那些场景集合，另一条裤腿代表以失败而告终的那些场景集合。 27

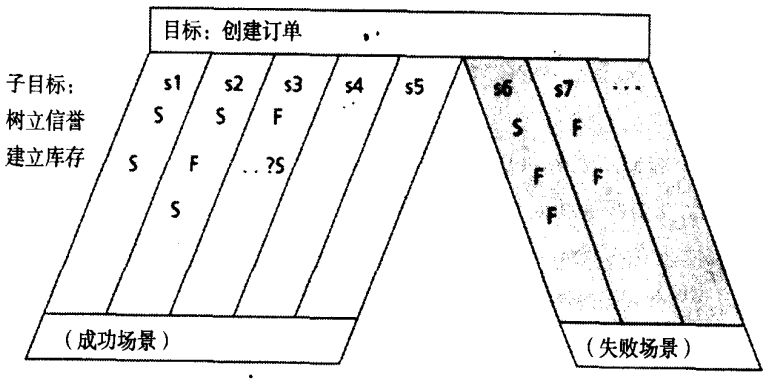


图2-2 条形裤：成功场景和失败场景

图中，无论是代表成功的裤腿还是代表失败的裤腿，其上每一个条纹都对应一个场景。在此把成功裤腿上的第一个条纹称为“主成功场景”，其余的条纹则代表其他最终能够成功结束的场景——其中，一些场景通过可选的成功路径实现；一些场景在中间过程中经历了失败，但又从失败中恢复过来，最终成功结束。失败裤腿上的所有条纹在执行过程中都遭遇到了失败，可能在某段时间内，暂时从失败中恢复过来，但最终还是以失败结束。

实际上，并不需要对每一个用例都从头至尾单独地进行描述。这是一个很糟糕的编写策略，因为这样做既乏味，又容易导致内容冗余，此外还很难维护。条形裤图案很有帮助，它使读者铭记每个用例都有两种结束方式；主执行者的目标将所有的场景捆绑在一起；每一个场景都是对目标成功或失败情况的一种简单描述。

在图2-3的条形裤中增加了一些内容以表示子用例，子用例嵌入在为之命名的用例中。一个顾客想要“创建订单”，顾客的子目标之一是“树立信誉”。这个子目标比较复杂，它可能会成功，也可能会失败：这是一个被折叠成为单个步骤的用例。步骤“顾客树立信誉”的条形带在另一个条形裤集中。包含该步骤的条纹或场景表明子目标是成功还是失败。图2-3的场景1和场景2中，子目标是成功的。在场景3和场景7中，子目标是失败的。但是，在场景3中，“树立信誉”的下一个子目标是成功的，场景最终是成功结束的。而在场景7中，第二次尝试也失败了，整个

28 用例以失败而告终，没能完成订单的创建。

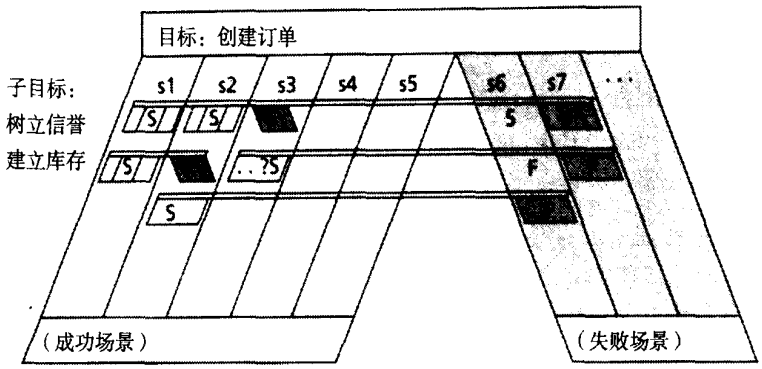


图2-3　条形裤表现子目标

图2-3中，子用例上显示小条纹的方法表明，外部用例并不关心子用例在达到其最终状态过程中究竟做了什么，它只关心子用例最终的结果：是成功还是失败。外部用例（或者说调用用例）只是在定义子用例的步骤是成功还是失败的基础上建立。

从条形裤图案中我们可以得出如下一些原则：

- 一些场景成功结束；一些场景则以失败告终。
- 用例将所有的场景聚集到一起，无论是成功的场景还是失败的场景。
- 每一个场景是对一个条件集合和一个结果的直接描述。
- 用例包含场景（裤子上的条纹），场景中的执行步骤中又会包含子用例。
- 场景中的步骤不关心子用例具体用到了哪个条纹，而只关心子用例最终是成功还是失败。

本书在写作过程中始终贯彻这些原则。

2.2　具有利益的项目相关人员之间的契约

执行者和目标模型解释了如何编写用例中的句子，但是没有涉及如何描述所讨论系统的内部行为方面的内容。出于这个原因，执行者和目标模型需要基于“用例是具有利益的项目相关人员之间的契约”这个观点而进一步被扩展，扩展后的模型称为“项目相关人员和利益”（*Stakeholders and Interests*）概念模型。模型中“项目相关人员和利益”部分指明了哪些内容需要被包括在用例之中，哪些内容不需要。

被讨论系统在项目相关人员之间实施一个契约，用例详细地描述了契约中的行为部分。系
29 统运行时并不是所有的项目相关人员都在场。主执行者通常是在场的，但也不是总在场。其他项目相关人员不在场，因此称之为“幕后执行者”（*offstage actor*）。系统执行动作来满足这些幕后执行者的利益，包括收集信息、执行确认检查以及更新日志等（见图2-4）。

ATM必须保留一个日志来记录所有交互过程，以备发生争议时项目相关人员的利益能得到保护。它也记录其他信息，以便能够判断在失败之前交易究竟进行到何种程度。ATM和银行系

统在交付现金之前，要先核实账号持有人是否有足够的存款，以确保ATM不会向客户提供比他在银行中实际存款数额还要多的现金。

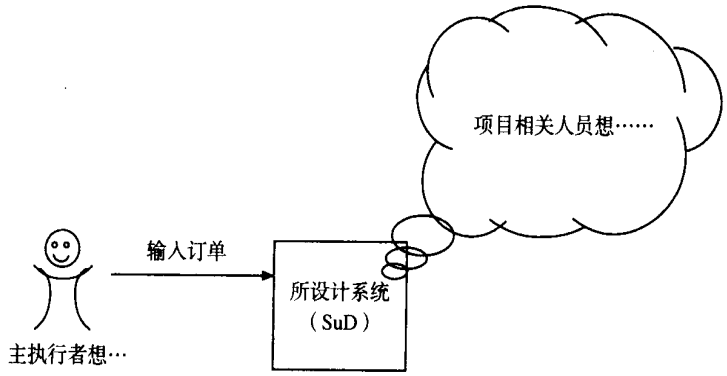


图2-4 SuD为主执行者提供服务，同时维护幕后项目相关人员的利益

用例，作为规范行为的契约，捕获了与满足项目相关人员利益相关的所有行为，并且仅限于此。

要认真地完成一个用例，有必要列出所有的项目相关人员，然后根据用例中的操作命名他们的利益，声明如果用例成功完成将对每一个项目相关人员意味着什么以及靠什么来保证他们能从系统中得到他们所希望得到的利益。把这些搞清楚之后，就可以开始编写用例中的执行步骤，确保从用例被触发开始直到其结束期间所有这些不同的利益都能得到满足。如此，就能知道编写用例时从哪里开始到哪里结束，以及用例中应包含什么，不应包含什么。

许多人并没有如此认真地编写用例，也往往能幸运地完成任务。好的用例编写者在写随意型用例时，往往是在自己的头脑中做这些练习。他们可能会忽略某些东西，但是被忽略的部分在进行软件开发时会有办法重新捕获得到。这种做法在许多项目中都进行得很顺利，但是有时却会招致巨大的损失。参见4.1节“项目相关人员”中关于忘记某些利益的故事。 30

为了满足项目相关人员的利益，需要描述三种行为：

- 两个执行者之间的交互（为了促进一个目标）。
- 确认（为了保护项目相关人员）。
- 内部状态变化（代表项目相关人员）。

项目相关人员和利益模型只是对编写用例的总体过程做了一点微小的改动：列出所有的项目相关人员和他们的利益，并用这个列表进行仔细的检查以确保用例体中没有任何内容被遗漏。然而，这个微小的改动却能对用例的质量产生重大的影响。

2.3 图形模型

这一节是专门为那些喜欢建立抽象模型的人而编写的。不属于这类人的读者，完全可以放心地跳过这一部分。

正如所指出的那样，用例描述了具有利益的项目相关人员之间行为的契约。我们按照被选中的项目相关人员的操作目标来对其行为进行组织。这里“被选中的项目相关人员”是指那些想要系统帮他们做些事情的人，我们称其为主执行者。用例名是主执行者的目标，它包含描写契约中相应部分所需的所有行为。

系统有责任通过执行一系列动作来满足意见一致的项目相关人员的一致利益。系统要执行的动作是下面三类之一：

- 两个执行者之间的交互，交互过程中可能会有信息的往返传递。
- 为了维护某一项目相关人员的利益所进行的确认。
- 一次内部状态变化，也是为了维护或增进某一项目相关人员的某一利益。

场景由执行步骤组成。在成功场景中，项目相关人员的所有（一致）利益都通过系统有责任实现的服务而得到满足。在失败场景中，那些利益都根据系统的承诺而得到保护。当项目相关人员的所有利益都得到满足或得到保护时，场景就结束了。

三个请求目标被实现的触发事件是主执行者启动与系统的一次交互，主执行者利用某个媒介启动那次交互，或者根据时间或状态进行启动。

31 图2-5至图2-8以统一建模语言(UML)的形式描述了本章中所描述的用例模型。

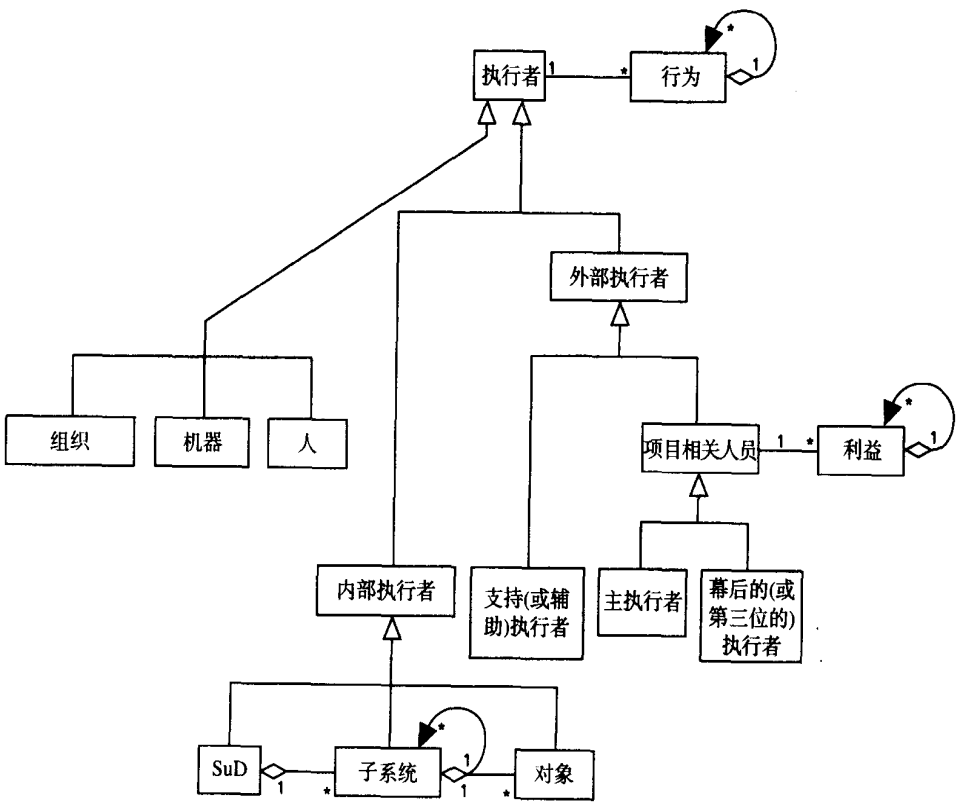


图2-5　执行者和项目相关人员

项目相关人员具有利益。执行者具有行为。主执行者也是项目相关人员。

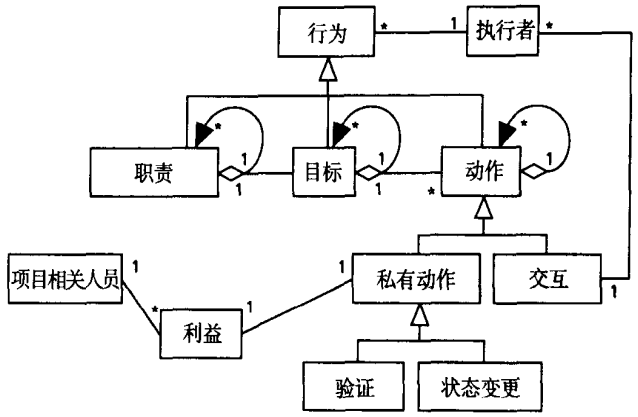


图2-6 行为

面向目标的行为由职责、目标和动作组成。
我们所写的私有动作是指那些增进或保护了项目相关人员利益的行为。
交互将执行者间的动作连接起来。

32

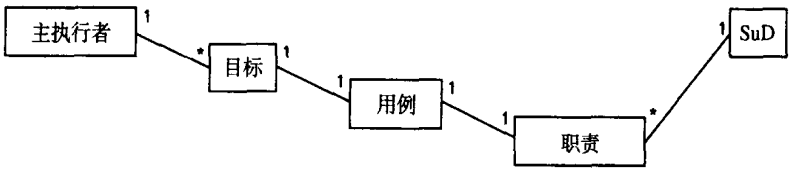


图2-7 用例是职责的激发者

用例通过激发SuD的职责来捕获主执行者的目标。

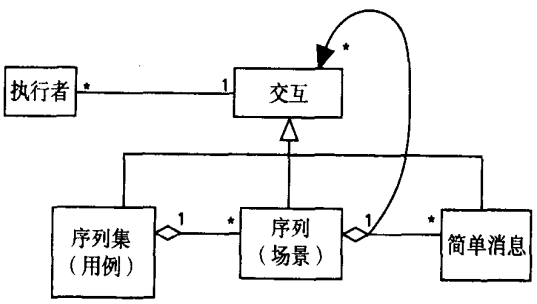


图2-8 作为组合的交互

有N个执行者参加一次交互。交互可以分解为用例、场景和简单消息。
出于方便，再次使用“序列”这个术语。

在这里要宣传一点真实的事情。我们如果不是几年来使用一个基于模型的工具在多个项目中对此模型进行测试，我们真不知道如何调试这个模型。换句话说，它可能包含一些不易被觉察的错误。本书中把它包含进来，是为了提供给那些希望进行试验性研究的人，或许他们想创建一个类似于此的基于模型的工具。

33

第3章 范　围

范围（*scope*）一词用来描述项目开发人员负责的设计工作的边界，以便与应由其他人负责的设计工作或已经完成的设计工作相区别。

要弄清楚一个项目的范围，或甚至只是明确一次讨论的范围，也是很困难的。Rob Thomsett顾问推荐了一个用来跟踪和管理范围讨论的很好的小工具——“内/外”列表（*in/out list*）。这个工具极其简单，但非常有效。既可用它来控制普通会议的讨论范围，也可用它来控制项目的需求。

创建一个由三列组成的简单表格。左边一列可以包含任何内容；其余两列分别以“内”和“外”标识。每当对一个主题是否应被包含在讨论范围之内产生疑问时，就把它加入表格，并询问它是在项目“内”，还是在项目“外”。正如Rob所描述的和我们所见到的，结果令人吃惊：人们对于一个主题到底是属于项目内还是项目外往往持有截然相反的观点。Rob说有时需要上报项目决策委员会来决定一个特定的主题到底应在工作范围之内还是在工作范围之外。“内”和“外”的决策可能会在项目进度上导致若干“工作-月”的差别。建议读者在下一个项目或下一次会议上试试这个技术。

表3-1是一个“内/外”列表的例子，这是为“购买请求跟踪系统”创建的。

表3-1　“内/外”列表

主　题	内	外
以任意形式开发票		外
产生请求报告（请求可能由卖主、分部或人发起）	内	
将请求合并成一个PO	内	
部分发货，延迟发货，错误发货	内	
所有新的系统服务，软件	内	
系统中的任何非软件部分		外
识别任何可用的已存在软件	内	
申请	内	

在需求或用例编写活动刚开始时使用“内/外”列表，以区分工作范围之内的东西和工作范围之外的东西。每当讨论偏离轨迹和本不该有的需求出现在讨论内容中时，就应参考一下“内/外”列表，并随着工作的进展对表格内容进行修改。

与被讨论系统的功能范围和设计范围相关的主题都可以使用“内/外”列表。

3.1　功能范围

功能范围是指系统要提供的服务，它最终应被用例所捕获。然而，在项目刚刚启动时，我

们并不能确切地了解项目的功能范围。在识别用例的同时也在决定项目的功能范围——这两个任务是密不可分的。“内/外”列表对此非常有帮助，因为它使我们能够划定一个边界来区分范围内外的东西。此外，还有两个很有帮助的工具，分别是“执行者-目标”列表（*actor-goal list*）和“用例简述”（*use case briefs*）。

3.1.1 执行者-目标列表

执行者-目标列表列举了系统支持的所有用户目标，展示了系统功能方面的内容。执行者-目标列表与“内/外”列表有着一定的差别：“内/外”列表既显示范围之内的东西也显示范围之外的东西；而执行者-目标列表只包括系统真正支持的服务。表3-2是“购买请求跟踪系统”的执行者-目标列表。

表3-2 执行者－目标列表的示例

执 行 者	任务级目标	优 先 级
任何人	检查请求	1
授权者	改变权限	2
购买者	改变卖主契约	3
请求者	发起请求	1
	改变请求	1
	取消请求	4
	做请求已被满足的标记	4
	拒绝发送货物	2
认可者	完成请求的提交	2
购买者	完成订购请求	1
	指定卖主后启动PO	1
	未发货警告	4
授权者	验证认可者签名的有效性	3
接收者	注册发货	1

该列表由三列组成。左边一列是主执行者（具有目标的执行者）的名称；中间一列是各执行者针对该系统的目标；第三列是优先级，或该系统在此版本中将支持此目标的可能值。在整个项目过程中将不断地更新此表，以保持该表能实时地反映系统的功能边界状态。

有些人在列表中加入额外的列——触发事件（*trigger*），以便识别由时间而不是由人触发的用例。还有人加入其他列，如业务优先级（*business priority*）、开发复杂度（*development complexity*）和开发优先级（*development priority*）等，由此他们可以将业务要求同开发成本分离开来，以得出开发优先级。

执行者-目标列表是用户代表、金融投资人和开发小组间进行谈判的出发点。它的重点在项目的规划和内容上。

3.1.2 用例简述

低精确度情况对合理安排精力和工作进度很重要。执行者-目标列表在描述系统行为方面精

确度是最低的，但它对从整体上把握系统很有帮助。下一个精确度等级是“主成功场景”或“用例简述”。

用例简述是对用例行为所作的一个包含2～6句话的描述，它仅提及了最重要的活动和失败情况。它提示读者用例中正在发生什么事情，对估计工作复杂度很有用。通过商业性的成品构件来构造系统的开发组利用这个描述来选择构件。一些项目小组（例如那些有着相当好的内部交流机制和能与用户频繁进行讨论的小组）在描述需求时从未写过比用例简述更详细的东西；他们把其余的需求置于频繁的讨论、原型和经常性发布的递增版本中。

35～37

可以将用例简述做成一个表格，或作为执行者-目标列表的扩充，或在初稿中直接将其作为用例体的一部分。表3-3是用例简述的一个例子，对此，要感谢Navigation Technologies 的Paul Ford、Steve Young和Paul Bouzide。

表3-3　用例简述的示例

执 行 者	目　标	简　述
生产人员	修改行政区格	生产人员向参考数据库中加入行政区域元数据（行政等级、货币、语言代码、街道类型等）。源数据的联系信息被分类。这是更新参考数据的一个特殊情况。
生产人员	准备数字绘图源数据	生产人员在准备与操作数据库进行合并时，将外部的数字数据转换成一种标准格式，并对之进行验证和校正。数据被分类并存储到数字源库中。
生产人员和实地工作人员	将共享数据的更新事务提交给操作数据库	工作人员将堆积的更新事务实施到操作数据库上。非冲突事务被提交给操作数据库。应用程序的语境与操作数据库是同步的。已提交的事务从应用程序语境中被删除掉，以保证操作数据库的一致性，冲突事务则通过手工方式或交互方式解决。

3.2　设计范围

设计范围是系统的区域——如果软件占据空间的话，可以说成是“空间区域”。它是开发人员负责设计和讨论的系统的集合，包括硬件系统和软件系统；它是集合的边界。如果我们要设计一个ATM，那么放在盒子中的硬件和软件都要由我们产生，即盒子以及盒子中的所有东西都要由我们来设计。而要与盒子进行交互的计算机网络却不用我们来设计——它处在设计范围之外。

38

此后，当只写“范围”（*scope*）一词时，指的是“设计范围”（*design scope*）。这是因为功能范围由执行者-目标列表和用例就可以充分地进行定义，而设计范围则是每个用例中都关心的一个主题。

正如下面这个故事所阐明的，作者和读者对一个用例的设计范围达成共识是很重要的——正确的做法就应如此。错误的做法所付出的代价是使花费增加一倍或更多，而契约将导致灾难性的后果。用例的读者必须很快能明白什么是系统边界之内的内容。仅仅靠用例名和主执行者并不能很明白地表明这一点。不同规模的系统甚至会出现在同一个用例集合中。

通常，编写者会认为系统的范围是显而易见的，以至于他们觉得没有必要去提及。然而，当有多个编写者多个读者时，用例的设计范围就毫无明显性可言了。可能有的编写者认为范围是整个公司（参见图3-1），有的编写者认为是公司里所有的软件系统，有的编写者认为是这个新客户/服务器系统，还有人认为只是客户系统或只是服务器系统。读者在没有关于其真正含义的任何线索的情况下大惑不解，甚至对文档产生错误的理解。

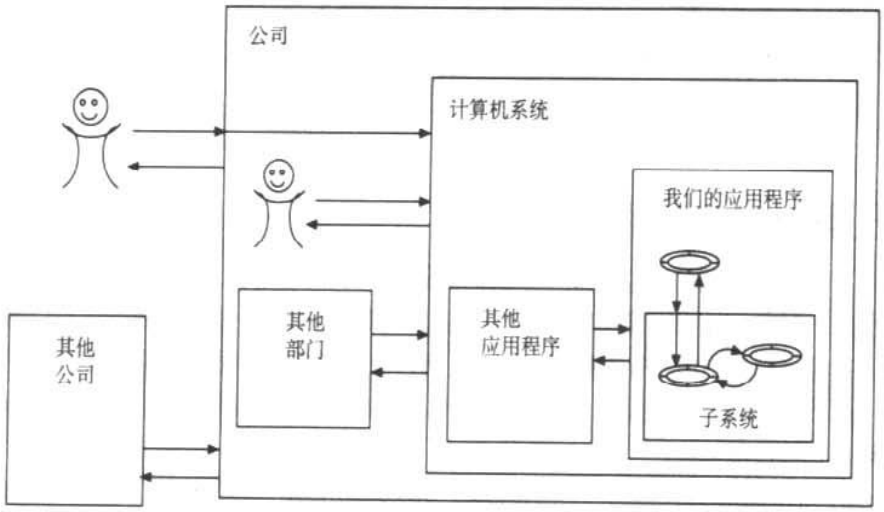


图3-1 设计范围的大小是任意的

那么，怎么做才能消除这种误解呢？

惟一的答案是：*列出每一个用例的设计范围，对最重要的范围使用专用的名称*。更具体一点，假设MyTelCo正在设计NewApp系统，该系统包括一个搜索子系统。设计范围名称如下：

- 企业（即*MyTelCo*）。在陈述主执行者的目标时，我们将讨论整个组织或企业的行为。用组织名（MyTelCo）来标识用例的范围域，而不要仅仅用“公司”这个词。如果正在讨论的是一个部门，那就用部门名称。业务用例是在企业范围内编写的。
- 系统（即*NewApp*）。这是我们将负责建立的硬件或软件部分。系统之外是将与系统进行交互的所有的硬件、软件和人员。
- 子系统（即*搜索者*）。已经打开主系统，并打算讨论它的某一部分是如何工作的。

◆ 一个简短而真实的故事

为了帮助构建一个固定时间、固定花费的大型系统投标，我们正在走查一些设计样例。我提到打印机，然后开始讲述它的功能。IS专家大笑。他说，“你这个PC人士简直是在损我，”他接着说，“你以为我们就用这个小小的激光打印机打印我们的发票？我们有一个巨大的打印系统，包括一连串的打印机、批量式I/O以及各种需要的设备。我们成箱地打印发票！”

我惊呆了，“你的意思是打印机不在系统范围之内？”

“当然不在！我们将用我们已经有的打印系统”。

实际上，我发现针对打印系统有一个非常复杂的接口。我们的系统就是准备一个存放着要打印内容的磁带。晚上，打印系统会读磁带并将其上的内容打印出来。同时还要准备一个应答磁带来描述打印工作的执行结果，包括不能打印内容的出错记录。第二天，我们的系统会读出这些结果，然后注明哪些内容没有被正确打印出来。与磁带联系的接口的设计工作是很重要的，而且与我们普经期望的完全不同。

打印系统并不是由我们来设计，而是由我们来使用。因此，它不在我们的设计范围之内。(正如3.3节中所述，它是一个辅助执行者)。如果我们没有发现这个错误，我们所编写
39 的用例就会把它包含到设计文档当中，从而转去创建一个比所需功能要多的系统。

3.2.1　用图标来突出设计范围

考虑一下在用例标题的左边附加一个图标的做法，以便在读者开始阅读用例之前向其传递设计范围信息。目前，还没有能够管理这些图标的工具，但是画一些这样的图标可以减少混淆。在本书中，我们采用适当的图标来标注每一个用例以便使读者可以很容易地了解其范围。

在读下面的列表时，请谨记这一点：黑盒用例不涉及被讨论系统的内部结构，而白盒用例
40 涉及被讨论系统的内部结构。

- 业务用例把企业作为其范围。它的图标是一个建筑物图形。如果将整个企业作为一个黑盒，就将建筑物图标设为灰色（ ）；如果所谈论的是组织中的部门和人员，那就将其设为白色（ ）。
- 系统用例将计算机系统作为其范围。它的图标是一个盒子。如果将计算机系统作为一个黑盒，就将盒子图标设为灰色（ ）；如果要揭示系统构件是如何工作的，就将其设为白色（ ）。
- 构件用例是关于被设计系统的子系统或构件的描述。其图标是一个螺钉（ ）。具体示例请参见用例13～用例17。

3.2.2　设计范围示例

下面提供三个在不同范围内对系统进行描述的例子。

(1) 企业—系统的范围

假设我们在为电话公司*MyTelCol*工作，该公司正在设计一个新的系统*Acura*，以处理服务并更新订单。Acura包括一个与服务器相连的工作站。服务器与一个运行着旧系统*BSSO*的主机相连。BSSO只是一个隶属于主机的终端，不允许对它作任何修改；只能使用它已有的接口。

Acura的主执行者包括客户、职员、各种各样的经理和BSSO(很显然BSSO不在设计范围之内)。

让我们来找出一些系统必须支持的目标。最明显的目标是“增加一个新服务”。我们认为这个用例的主执行者是公司职员，他代表客户提出这一请求。然后我们坐下来写一些用例。

紧接着会提出的问题是“被讨论的系统到底是什么呢？”我们很自然地会想到下面这两个系统：

- *MyTelCo*。我们所感兴趣的问题是“从用户的角度来看，从发起请求开始，到实现请求，

再到提交请求，MyTelCo提供的完整形式的新服务究竟应是怎样一种情况？”这个问题会引起两方面的兴趣：公司经理希望了解新系统如何响应外界请求；而实现人员则希望了解新系统运行时的环境条件。

我们将在企业范围内（🏠）编写这个用例，范围域标注为MyTelCo，所编写的用例中不会提及公司内部的人员(不会有职员、部门和计算机)。由于这种用例是关于业务的，因此常常被称为*业务用例*（business use case）。

- *Acura*。我们所感兴趣的问题是“Acura一方面提供与职员和顾客的接口，一方面提供与BSSO系统的接口，在这种情况下，其服务以什么样的形式出现呢？”这是设计者最为关心的用例，因为它对设计者所要创建的系统进行了确切的描述。我们将在系统范围内（🔍）编写这个用例，范围域标注为Acura。它可以随意地提及职员、部门以及其他计算机系统，但是不能提及工作站和服务器子系统。

41

我们将创建两个用例。为了避免相同的信息被重复两次，在一个更高的层次上编写企业用例（风筝符号），该用例描述MyTelCo如何响应用户的请求，并满足请求，甚至可能要因此而收取费用。企业用例的目的是为了描述新系统周围的语境。然后，我们详细地描述了一个用户目标级的用例，它以Acura作为其范围，处理请求所需时间大约为5分钟到20分钟。

用例6 🏠 增加新服务（企业）

主执行者：顾客
范围：MyTelCo
层次：概要
1. 顾客打电话给MyTelCo,请求一个新的服务……
2. MyTelCo提供……，等等……

用例7 🔍 增加新服务（Acura）

主执行者：代表外部客户的职员
范围：Acura
层次：用户目标
1. 顾客打电话过来，职员与顾客讨论请求。
2. 职员在Acura找到顾客。
3. Acura显示顾客当前的服务包……，等等……

没有必要在Acura工作站或Acura服务器范围内编写任何用例，因为我们对此并不关心。此后，设计组中某些成员可能选择以用例的形式来编写Acura子系统的设计文档。那时，他们将会编写两个用例，一个用例在Acura工作站范围内，另一个用例在Acura服务器范围内。根据我们的经验，不必编写这两个用例，因为采用其他的技术就已经足以记录子系统的体系结构。

42

(2) 一个应用程序对应多台计算机

下面的情况虽不多见却有着相当的难度。下面以MyTelCo的例子为基础来创建一个用例。

Acura将会逐渐取代BSSO。新的服务请求被放入Acura，然后用BSSO作一些修改。慢慢地，Acura将会发挥越来越大的作用。这两个系统必须共存，并且要互相同步。因此，必须要为这两个系统编写用例：Acura是全新的，对BSSO需作一些修改以与Acura保持同步。

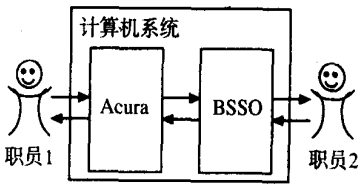
在这种情况下的一个难题是必须要有四个用例，两个为Acura编写，两个为BSSO编写。对每一个系统来说，在其两个用例中，一个用例是以职员作为主执行者，另一个用例是以计算机系统作为主执行者。编写这样四个用例是必须的，无法回避，但是人们常常会被搞糊涂，因为它们看起来有很大的冗余性。

为了记录这种情况，我们首先编写一个概要级的用例，把这两个计算机系统共同作为该用例的范围。这样就有机会记录它们之间的交互情况。在此用例中，我们引用了包含各个系统的需求的具体用例。第一个用例是白盒类型的用例（以白盒符号标注）。

由于该情况很复杂，因此在用例描述中包含了每个用例的范围示意图。

用例8　输入和更新请求（联合系统）

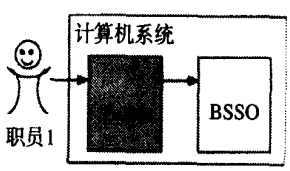
主执行者：代表外部客户的职员
范围：计算机系统，包括Acura和BSSO（见图）
层次：概要
主成功场景：
1. 职员向Acura中添加新服务。
2. Acura把新服务请求通知给BSSO。
3. 一段时间以后，职员更新BSSO中的服务请求。
4. BSSO把更新后的服务请求通知给Acura。



这四个子用例都是用户目标级的用例，以海平线符号标记。尽管它们都是系统用例，但却是为不同的系统编写的——因此相应的用例图也是针对不同系统的。在每一幅图中，我们用圆
43 标注主执行者，而SuD以阴影方式显示。此时，用例是黑盒，因为它们是新任务的需求。此外，在给它们使用的动词名称上也有微小的差异，我们用“通知”一词来表明一个系统与另一个系统保持同步。

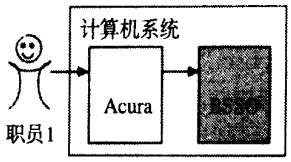
用例9　添加新服务（进入Acura）

主执行者：代表外部顾客的职员
范围：Acura
层次：用户目标
……紧接着是用例体……



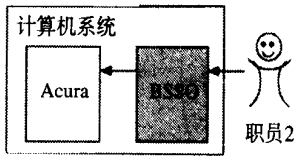
用例10 通知新服务请求（BSSO中）

主执行者：Acura
范围：BSSO
层次：用户目标
……紧接着是用例体……



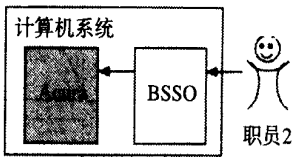
用例11 更新服务请求（BSSO中）

主执行者：代表外部顾客的职员
范围：BSSO
层次：用户目标
……紧接着是用例体……



用例12 通知更新后的服务请求（Acura中）

主执行者：BSSO
范围：Acura
层次：用户目标
……紧接着是用例体……



如果你正在使用UML用例图，那么可以绘出概要级的用例而无需把它们写出来。但那样做仍然不会减少对这四个用户目标级用例所产生的困惑，因此仍然需要认真地标记它们的主执行者、范围和层次，可能仍然需要在用例中绘制出范围图。

这个表示两个系统之间交互情况的图是以UML的风格给出的。上面部分表明BSSO在Acura充当主执行者的用例中充当辅助执行者的角色,而在另一个用例中充当主执行者的角色。在下面的图中，BSSO和Acura 的角色关系正好相反。

这幅图非常清晰地表明了这四个用例之间的关系，但这幅图是不标准的，因为它显示了一个系统的用例对另一个系统的用例的触发情况。

但仍不能消除困惑。为此，我们考虑绘制图3-3所示的非标准的用例图以便说明这两个系统之间的连接关系。这幅图虽然简单，但很难维护。应该以最有利于与读者进行交流的形式绘制这些图。

(3) 基本用例

让我们在最大的范围边界，看一下一个开发小组用用例来记录其设计框架的方法。他们一开始就为他们的框架做了一个长达18页、附满图例的描述。他们认为阅读这个描述非常困难，因此决定以用例作为描述技术来进行测验。

44

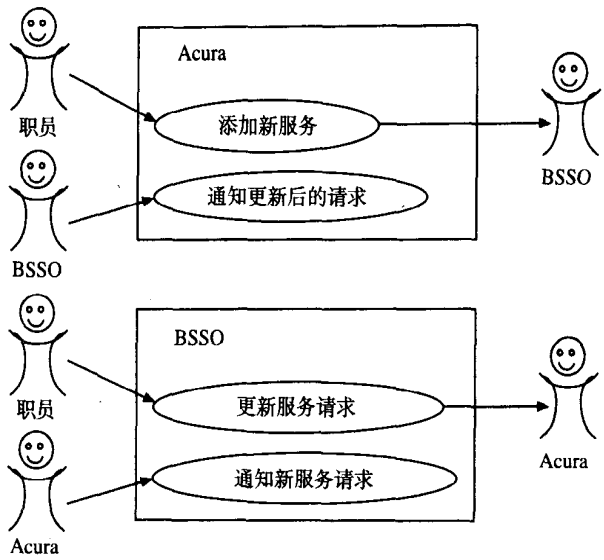


图3-2　Acura-BSSO的用例图

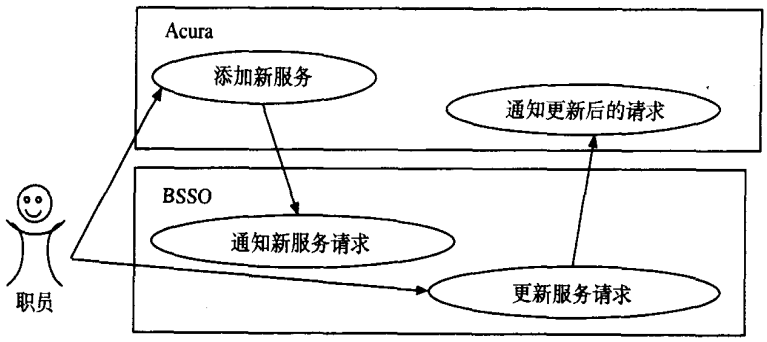


图3-3　Acura-BSSO的组合用例图

这个小组在这个任务上花费了一个星期的时间。首先，他们草拟了40个用例以确保他们捕获到了框架必须处理的所有请求。通过使用扩展和数据可变情况列表，他们对这40个用例进行了调整，最后修改为6个用例。

多数读者会因为没有参与过这样的业务而感到难以理解这些用例。但是，有些读者可能是技术编程人员，也许正在寻求编写其设计文档的方法，因此本书把这些用例包含进来，以展示这个小组是如何记录内部体系结构以及如何利用可变情况列表的。我们发现如果给出问题的复杂度，这些用例将相当容易阅读。

总体描述：

总体结构必须能够处理并发任务。为了做到这一点，它必须支持线程和资源锁定。这些服务由并发服务框架（Concurrency Service Framework, CSF）来处理。客户对象使用CSF来保护代码中的临界区域免受因多线程而导致的非安全存取。

用例13 资源的串行存取

主执行者：服务客户对象
范围：并发服务框架（CSF）
层次：用户目标
主成功场景：
1. 服务客户请求资源锁给予它特殊的资源存取权。
2. 资源锁把控制权返回给服务客户以使它能够使用资源。
3. 服务客户使用资源。
4. 服务客户通知资源锁它对资源的使用已经完成。
5. 服务客户结束后，服务锁处理善后事宜。
扩展：
2a. 资源锁发现服务客户对资源已经具有存取权
    2a1. 资源锁对请求实施资源锁转换策略（用例14）。
2b. 资源锁发现资源已经分配给他人去使用：
    2b1. 资源锁实施兼容性策略（用例15）以便使服务客户对资源也享有存取权。
2c. 资源锁定的持续时间限制不为0：
    2c1. 资源锁启动锁定计时器。
3a. 在客户通知资源锁完成对资源的使用之前，锁定计时器计数时间到：
    3a1. 资源锁向客户进程发送一个异常信号。
    3a2. 失败！
4a. 资源锁发现服务客户上的锁计数非0：
    4a1. 资源锁减少请求的引用个数。
    4a2.成功！
5a. 资源锁发现资源当前未被使用：
    5a1. 资源锁实施存取选择策略（用例16），给处于挂起状态的服务客户赋予存取权。
5b. 锁定计时器仍然在运行：
    5b1.资源锁取消锁定计时器的运行。
技术和数据可变列表：
1. 特殊的存取请求可以是：
    • 独占存取
    • 共享存取
2c. 资源锁的锁定超时时间可以通过以下因素指定：
    • 服务客户
    • 资源锁定政策
    • 全局缺省值

45 ~ 46

用例14　实施资源锁转换策略

主执行者：客户对象
范围：并发服务框架（CSF）
层次：子功能
主成功场景：
1. 资源锁验证请求的存取方式是独占存取。
2. 资源锁验证服务客户已经具有共享存取权限。
3. 资源锁验证没有服务客户等待更新其存取权限。
4. 资源锁验证没有其他的服务客户在共享资源。
5. 资源锁赋予服务客户对资源的独占存取权。
6. 资源锁增加服务客户的锁的个数。

扩展：
1a. 资源锁发现请求的存取方式是共享存取：
　　1a1. 资源锁增加服务客户上的锁的个数。
　　1a2. 成功！
2a. 资源锁发现服务客户已经具有独占存取权限：
　　2a1. 资源锁增加服务客户的锁的个数。
47　　2a2. 成功！
3a. 资源锁发现有另一个服务客户在等待更新其存取权限：
　　3a1. 通知服务客户不能赋予他所请求的存取权限。
　　3a2. 失败！
4a.资源锁发现有其他的服务客户正在使用资源：
　　4a1.资源锁令服务客户等待获得资源存取权限（用例17）。

用例15　实施存取兼容性策略

主执行者：服务客户对象
范围：并发服务框架（CSF）
层次：子功能
主成功场景：
1. 资源锁验证请求的存取方式是共享存取。
2. 资源锁验证当前对资源的所有使用都是共享存取。

扩展：
2a. 资源锁发现请求的存取方式是独占存取：
　　2a1. 资源锁令服务客户等待获得资源存取权限（用例17）（进程的执行由锁服务策略恢复）。

2b. 资源锁发现资源正在被以独占方式使用：

2b1. 资源锁令服务客户等待获得资源存取权限（用例17）。

可变情况：

1. 兼容性准则可以改变。

用例16 实施存取选择策略

主执行者：客户对象

范围：并发服务框架（CSF）

层次：子功能

主成功场景：

语境目标：资源锁必须决定哪一个等待请求（如果有的话）应该获得服务。

注意：这个策略是一个可变点。

1.资源锁选择等待时间最长的请求。

2. 资源锁将存取权限赋予被选中的请求，途径是使其进程进入可运行状态。

扩展：

1a.资源锁发现没有处于等待的请求：

1a1. 成功！

1b. 资源锁发现有一个请求正等待从共享存取更新为独占存取：

1b1. 资源锁选中这个等待更新的请求。

1c. 资源锁选中一个请求共享存取的请求：

1c1. 资源重复第一步直到遇到一个独占请求。

可变情况：

1. 选择顺序准则可以改变。

用例17 令服务客户等待获得资源存取权限

主执行者：客户对象

范围：并发服务框架（CSF）

层次：子功能

主成功场景：

使用者：CC2,4资源锁定：

1. 资源锁将服务客户进程挂起。

2. 服务客户等待直到再次被恢复运行。

3. 服务客户进程被恢复运行。

48

扩展：

1a. 资源锁发现一个等待超时已经被指定：

　　1a1.资源锁启动计时器。

2a. 等待计时器时间到：

　　2a1. 通知服务客户其请求的存取权限不能得到。

　　2a2. 失败！

技术和数据可变列表：

1a1. 锁的等待超时时间可以通过下列因素指定：

- 服务客户
- 资源锁定策略
- 全局缺省值

3.3 最外层用例

在本章的“企业—系统范围”这一小节中，我们建议编写两个用例，一个用例针对被设计系统，另一个用例针对比被设计系统更广的范围。现在，关于这一点可以说得更具体一些：对于每一个用例，找到用例仍能适用的最外层设计范围，在该范围内编写一个概要级的用例。

用例总是在一个设计范围内进行编写。通常，可以找到一个更广的设计范围，而主执行者仍然处于范围之外。如果不断扩大该范围，可以找到一个临界点，一旦越过这个临界点，主执行者就会被包含在范围之内。这个临界点就是*最外层范围*（*outermost scope*）。最外层范围有时
49 是企业，有时是部门，有时仅仅是计算机。通常，计算机部在计算机安全用例中是主执行者，市场部在广告用例中是主执行者，顾客在主系统功能用例中是主执行者。

一般地，整个系统中只有2个到5个最外层用例，因此并不是所有的用例都要编写两遍。最外层用例之所以会这么少是因为，每一个这样的用例都是通过把同一个设计范围内具有相似目标的主执行者合并到一起而创建出来的，它们把与这些执行者有关的所有低层用例都汇聚到了一起。

之所以大力提倡编写最外层用例是因为，编写这样的用例花费的时间很少，并能为用例集提供极好的语境信息。最外层用例显示了系统最终如何使其最外部用户收益；同时还为浏览系统的行为提供了一个内容列表。

下面看一下MyTelCo及其Acura系统的最外层用例。

MyTelCo决定让基于Web的用户直接访问Acura以减轻职员的负担。Acura负责报告职员的销售业绩。一些人还必须为客户和职员设置安全存取级别。有四个用例：**增加服务（由客户执行）**、**增加服务（由职员执行）**、**报告销售业绩**、**管理安全存取权限**。

必须以Acura作为SuD的范围编写所有这四个用例。需要为这四个用例中的每一个用例都找到最外层用例。

客户很显然是在MyTelCo范围之外，因此有一个以客户作为主执行者，MyTelCo作为范围的最外层用例。这个用例将会在概要级编写，把MyTelCo作为一个黑盒来显示，响应客户的请求，提交服务等。实际上，本章中的用例6“增加新服务（企业）”已经勾画出了这个用例的轮廓。

职员在MyTelCo范围之内。“增加特性（由人员）”的最外层范围是所有的计算机系统。这个用例将汇集职员与计算机系统的所有交互活动。我们希望所有职员的用户目标级用例以及一些子功能用例（如*Log In*和*Log Out*），都被包括在这个最外层用例中。

“报告销售业绩”以市场部作为其最终的主执行者。最外层用例的范围是“服务部”，它描述了市场部为了设置业绩奖金、报告销售业绩等，而与整个计算机系统以及服务部的交互情况。

“管理安全存取权限”以安全部或IT部作为其最终的主执行者，以“IT部”或整个计算机系统作为其最外层设计范围。此用例涉及到安全部利用整个计算机系统来设置和跟踪安全问题的所有方式。

应注意的是，这四个最外层用例涉及安全、市场、服务和客户方面，使用了Acura运行的所有途径。即使有100个低层用例需要编写，对Acura系统来说，这四个用例就足以覆盖所有的方面。 50

3.4 使用范围确定的工作产品

假设你们正在为目标系统定义功能范围，正在进行集中讨论时，并正在不断地在白板上的几个工作产品间移动。在白板的一端，放着“内/外”列表用来对作出的范围决定进行跟踪（“不，鲍勃，我们认为新的打印系统在范围之外——或我们需要重新审查一下‘内/外’列表中的这一条目吗？”）。你们已经有了执行者-目标列表。你们还有一张设计范围图，它显示了将与被讨论系统进行交互的人员、组织和系统。

你发现你在考虑这些产品的同时也在不断地对它们进行完善，逐步明确希望新系统所要做的事情。你认为你已经知道设计范围是什么，但是对“内/外”列表的一点改动却在改变着系统的边界。现在，你有了一个新的主执行者，目标列表也改变了。

最终你可能会发现需要第四个条目：一个对新系统的构想陈述（*vision statement*）。构想陈述汇集了全部讨论内容。它有助于首先决定哪些东西在范围内，哪些东西在范围之外。

当完成这些事情以后，就获得了四个工作产品，它们共同决定了系统的范围：

- 构想陈述
- 设计范围图
- “内/外”列表
- 执行者-目标列表

希望通过这个简短的讨论能让读者了解这四个工作产品是密不可分的，在建立工作范围的过程中，可能会对这四个产品进行修改。

3.5 练习

设计范围

3-1 对于下面的用户故事片段，请至少命名5个系统设计范围："……詹妮站在银行的*ATM*机前，天色很黑。她已经输入她的*PIN*，正在寻找确认按钮……"。

51 3-2 为ATM画一个多范围示意图，包括硬件系统和软件系统。

3-3 就个人而言，你正在为什么样的系统编写需求说明？它的边界是什么？哪些东西在其范围之内？在其范围之外有哪些它必须与之进行交互的对象？包含它的系统是什么？在包含它的系统之外，它必须与哪些对象进行交互？请给包含它的系统命名。

3-4 为"私人顾问/财政"（PAF）系统绘制一个多范围示意图（见练习4-4）

3-5 为一个Web应用程序绘制一个多范围示意图，在这个Web应用程序中，用户的工作站通过Web与公司的Web服务器相连，公司的Web服务器隶属于一个遗留主机系统。

52 3-6 阐述企业范围白盒业务用例和企业范围黑盒业务用例之间的区别。

第 4 章
项目相关人员和执行者

项目相关人员是指契约的参与者。执行者是指任何具有行为的事物——正如一位学生所说的："它必须能够执行一个条件语句"。执行者可能是一个人、一个公司或组织、一个计算机程序或一个计算机系统——硬件、软件或软硬件兼备的系统。

请从以下方面入手来寻找执行者：

- 系统的项目相关人员（*stakeholder*）
- 用例的主执行者（*primary actor*）
- 被设计系统（*system under design, SuD*）本身
- 用例的辅助执行者（*supporting actor*）
- 内部执行者（*internal actor*）——所讨论的系统（SuD）内的构件

4.1 项目相关人员

项目相关人员是对用例的行为具有特定利益的人或物。

当然，每个主执行者都是一个项目相关人员，但是一些项目相关人员尽管有权关心系统的行为，却从来不与系统进行直接的交互。这样的例子有：系统的拥有者、公司的董事会和调控主体（如内部税收服务和保险部门）。

学生们给那些在用例的执行步骤中从未直接出现过的项目相关人员起了一些别名："幕后的"（*offstage*）、"第三位的"（*tertiary*）或"沉默的"（*silent*）执行者。

加强对这些"沉默执行者"的注意可以大大提高用例的质量。他们的利益在系统执行的检查和确认中、在创建的日志中、以及在系统执行的动作中得以体现。业务规则之所以要被编制成文档，正是因为系统必须代表项目相关人员执行这些规则。用例必须体现出系统是如何保护这些项目相关人员的利益的。下面的故事说明了忘记项目相关人员的利益会付出怎样的代价。

◆ 一个简短而真实的故事

一个公司在其新系统售出几个拷贝后的头一年中收到了一些系统修改请求。一切看起来都很正常，直到有一天，他们参加了一个用例培训课程，并被要求对最近发布系统中的项目相关人员及其利益进行集中讨论。

令他们吃惊的是，他们发现在集中讨论的过程中，他们正在对最近收到的修改请求中的条目提出讨论。显然，在开发系统时，他们完全忽视了一些项目相关人员的某些利益。时隔不久，这些项目相关人员就发现系统没有为他们提供完全的服务，因此修改请求开始蜂拥而至。

自此，领导者坚决要求尽早考虑项目相关人员及其利益以避免再次发生这种代价昂贵的失误。

我们发现通过询问项目相关人员及其利益，能较早地识别出一些非常重要的需求，否则这些需求很可能会被忽视。要做到这一点不用花费很多时间，但它却使后续工作中少了许多麻烦。

4.2　主执行者

用例的主执行者是请求系统提供一项服务的项目相关人员。对系统来说，它有一个目标——系统通过执行操作可以实现该目标。主执行者经常（但不一直）是触发用例的执行者。

用例开始执行通常是由于主执行者发送了一个消息、按了一个按钮、敲了一个按键，或者以其他方式激发了用例所描述的活动序列。然而，有两种常见的情况，主执行者不是用例的激发者。第一种情况是当一个公司职员或电话接线员代表他人激发用例时；第二种情况是当用例由时间触发时。

公司职员或电话接线员通常是为最终主执行者（*ultimate primary actor*）（即真正关心用例的人）提供技术上的方便。通过技术转换，很可能最终主执行者会利用Web或自动话务系统直接激发或触发用例的执行。例如，一个顾客正在通过电话接入一个请求。而在该系统的Web设计方案
54 中，她可以直接输入她的请求（比如通过*Amazon.com*网站）。

相似地，市场或审计部门可能会坚持主张让职员来操作用例。使用例运行并不是职员的目标；用例为市场部经理提供了技术上的方便。在略微不同的情况下，市场部经理可能会自己来运行用例。

这些天我们写道："销售代理代表顾客"或"职员代表市场部经理"来捕捉系统用户代替他人来操作的信息。这样做就能使我们明白用户界面和安全清理功能是设计给职员用的，但关心操作结果的人却是顾客或市场部经理。

另一个关于非操作人员触发者的例子是时间。在每日午夜和每月月底运行的用例都不是由职员来触发的。在这种情况下，很容易明白主执行者是当时任何关心用例运行的项目相关人员。

关于用户和最终主执行者对应关系的主题足以让我们讨论很长时间。但是，我们建议不要在这方面花费太多的时间，也不要在酒吧中争论此事。当开始考查用户界面设计时，开发小组将会（或者说应该）投入大量的精力来研究真正用户的特点。当审查需求时，将会发现知道每个用例的最终主执行者（即真正关心该用例的人）是非常有意义的。

如果有人提出这样一个敏锐的问题"如果我们执行这一步骤时找到的主执行者是错误的，那其危害性将会有多大呢？"答案是："没多大"，我们将在下面的章节中阐述这一点。

4.2.1　主执行者为什么有时是不重要的（而有时又是重要的）

主执行者在需求收集工作刚开始时和系统将要发布之前一段时间内是重要的。而在这两个

时间点之间的那段时间里，他们是相当不重要的。

在用例编写开始时

列举出主执行者有助于在短时间内（即转瞬即逝的时间段内）能对系统在整体上有一个全局的把握。通过集中讨论来指定所有执行者的目的是为了通过集中讨论来捕捉到所有的目标。真正感兴趣的是这些目标，但是如果直接对它们进行集中讨论，则有许多目标会被遗漏掉。通过对主执行者进行集中讨论，能建立起一个运作情况的大致框架。然后，通过仔细研究这个框架，就能得到一个理想的目标列表。

所找到的主执行者的数目稍微多一些并没有什么坏处，因为最坏的情况也不过是对同样的目标创建两次。当仔细检查执行者和目标以便对工作结果进行优化时，将会发现并删除这些重复项。

但是，即使认真地执行了这两个集中讨论，也很可能不会找到系统需要支持的所有目标。新的目标会在编写用例中的“失败-处理”步骤中出现，但这是在早期阶段不能改变的一件事情。我们所能做的是，通过首先列举出所有主执行者的方法尽量捕捉到所有的目标。 55

充足的主执行者列表还具有其他三个优越性：

- 它使我们把注意力集中到使用系统的人身上。在需求文档中，我们记录下我们期望中的主执行者是哪些人，以及关于他们工作情况的描述和他们的基本背景和技能。这样做是为了使用户界面和系统设计者能保证系统最大程度地满足他们的要求。
- 它建立起了执行者-目标列表的结构框架，我们可利用此列表来分解开发工作和划分优先级。
- 它可用来将一个大的用例集合分解成多个包，以便分配给不同的设计小组。

在用例编写和设计过程中

一旦开始详细地开发用例，主执行者就开始变得无关紧要了。事实是，随着项目的不断推进，用例编写者将发现一个用例可能会被多种类型的执行者所使用。例如，任何职位比职员高的人都可以接听电话，并与客户进行交谈。因此，编写者在对主执行者进行命名时，通常会采用一种越来越一般化的方式，使用角色名如*Loss Taker*、*Order Taker*或*Invoice Producer*等。导致出现这样的用例“Invoice Producer产生发票……,”或“Order Taker取到发票……”（这些都不太有启发性）。

可以采用多种方法来处理这个角色片断，每种方法都各有优缺点。哪种策略都没有明显的优越性，你必须选择其一。

可选策略1。根据主执行者担当的角色来对它们进行分解。创建一个“执行者-角色”表，在表中列举出在任一用例中充当主执行者的所有不同的人和系统，以及他们担当的所有角色。在主执行者域使用角色名称。使用执行者-角色表将用例与现实世界中的人和系统对应起来。

这种策略能使编写者不必顾及错综复杂的工作头衔，而只专注于继续用例的编写工作。有些人或许是用户界面设计者或软件打包人员，他们会利用执行者-目标表来将用例同其最终用户对应起来。可选策略1带来的一个不便之处是需要单独维护和阅读一个列表。

可选策略2。在用例部分前的某个地方，书写“经理可以执行职员可以执行的任何用例，并且经理还可以执行其他更多的用例。地区经理可以执行经理可以执行的任何用例，并且地区经

理还可以执行其他更多的用例。因此，每当我们写主执行者是（例如）职员时，应该理解为任
56 何职位比职员高的人——本例中，经理和地区经理都可以执行该用例。”

这种做法比执行者-角色表容易维护，因为它变化的可能性不大。这种方法的缺点是，人们需要花更多的时间来互相提醒当职员作为主执行者时，经理也可以运行该用例。

利用这两种策略，人们都可以获得足够好的结果。但要说到值不值的问题，我们会采用第二种策略，因为这样可以少去编写、审查和维护一项工作产品。

有一点要注意，用例模板的主执行者域随着时间的推移会变得毫无价值。这是正常的，不必为此而担忧。

设计之后，准备实施系统时

在系统发布之前，主执行者又变得十分重要了。需要一个关于所有人及其要运行的用例的列表。如下所示：

- 将系统打包成单元以便安装到不同的用户机器上。
- 为每一个用例设置安全级别（网络用户、内部用户、检查员等）。
- 为不同的用户组创建培训材料。

4.2.2 执行者和角色

执行者（*actor*）一词暗示了动作中的个体（*individual*）。有时，在一个用例中，它指一个个体。有时，它指担当某个给定角色的一类个体的通称。

假设Kim是MyTelCo的一个客户，Chris是一个职员，Pat是一个销售经理。他们都可以参与“创建订单”。使用执行者的说法，可以说Kim、Chris和Pat是用例“创建订单”的主执行者。还可以说客户、职员和销售经理是“创建订单”的允许主执行者。可能说“销售经理可以执行职员可以执行的任何用例”，这是比较好的表达方法。

使用角色的说法，可以说Kim、Chris和Pat是个体执行者。他们中的任何一个人都可以担当客户的角色，但是只有Chris和Pat可以担当职员的角色，只有Pat可以担当销售经理的角色。由此，我们说驱动用例“创建订单”的角色是订购者（*Order Taker*），客户、职员和销售经理可以担当订购者的角色。这种表达方法比前一种更精确一些，有些人偏向于采用这种方法。然而，在用例领域是没有标准的，各种方法都有利有弊。

值得指出是，你应该使用你的开发组喜欢的任何方法。在前面这一小节“为什么说主执行者是不重要的（和重要的）”中，描述了为什么这件事不应给你造成太大压力的原因，以及如何去处理由此所导致的一些情况的方法。同时，执行者是已被工业界所接受的一个术语，它能够
57 很好地表达其含义，这就是在本书中使用这一术语的原因。

统一建模语言（UML）图和执行者/角色规格说明

UML提供了一种空心箭头来表示一个执行者是另一个执行者的特化（*specialization*）（参见附录A的图A-6，“改正后的终止大交易”）。

UML提供这个符号的好处是，它能使你清晰简洁地表示出“经理能做职员所能做的任何事情”这一个含义——你只要在职员和经理之间画一个这样的箭头，使箭头指向职员，箭尾指向经理就可以了。

这个符号的坏处是，对许多人来说，很容易将它理解成恰恰相反的含义。他们没有将经理理解为一类特殊的职员或将职员理解为一类特殊的客户，而这恰恰是图中所要表达的含义（图中真正想声明的是一个能做另一个所能做的任何事情）。他们理解成经理不仅仅是一个职员。这种情况还不会导致太大的问题，但是必须要正确对待。

特化箭头在执行者-角色问题的主要部分不会有很大的帮助。销售职员和审计职员的用例集之间有交叠部分，但是却不能在他们之间画特化符号，因为他们中任何一个都不能做另一个所能做的所有事情。由此，又回到了执行者-角色的讨论中。

4.2.3 刻画主执行者的特点

只有一个执行者列表对设计者来说是没有多少帮助的，设计者必须知道用户具有什么技能，然后才能保证设计出的系统行为和用户界面能够满足用户的要求。创建“执行者概况表”（*actor profile table*）的开发组声称他们对软件如何满足最终用户的需要有着更好的了解，因为在开发过程中，他们面前摆放着最终用户的技能信息。

最简单的执行者概况表只有两列，如表4-1所示。有些表也把执行者的其他名称（或者说别名）列举出来。执行者概况表的其他形式在“*Software for Use*”（Constantine and lockwood,1999）一书中有所讨论。

表4-1 执行者概况表的示例

名称	概况：背景和技能
客户	街上的任何人，能使用触摸显示屏，但是不能熟练自如地操作GUI。可能有阅读障碍，也可能有近视或色盲等
处理退换货的职员	连续使用该软件进行工作的人。熟悉指法的熟练用户。可能希望个性化界面
经理	偶尔访问的用户，习惯于使用GUI但对系统某些特殊功能不太熟悉。缺乏耐心

58

4.3 辅助执行者

用例中的辅助执行者是指为被设计系统提供服务的外部执行者。它可以是一个高速打印机、一个网络服务或者是必须进行一些调查然后把调查结果反馈给我们的人(例如，验尸官的办公室，它向保险公司提供一个人的死亡证明)。我们过去常常称辅助执行者为“次要执行者（*secondary actor*），但是人们发现这个术语容易引起混淆。现在越来越多的人使用“辅助执行者（*supporting actor*）”一词，它更加合理一些。

我们识别辅助执行者的目的是为了识别系统将要使用的外部接口以及这些接口间所采用的协议。从而得到系统的其他需求，比如数据格式和外部接口等（参见图1-1）。

一个执行者可以在一个用例中是主执行者，而在另一个用例中是辅助执行者。

4.4 被讨论系统

被讨论系统本身也是一个执行者——一个特殊的执行者。我们通常引用被讨论系统是通过其名称，例如Acura，或者我们称其为“系统”、“被讨论系统”、“被设计系统”或SuD。它在用例的设计范围域内被命名或描述。

虽然SuD是一个执行者，但它不是任何用例的主执行者或辅助执行者。第3章的3.2节“设计范围”中对此进行了详尽的讨论。

4.5 内部执行者和白盒用例

多数情况下，我们把被设计系统作为一个黑盒对待，我们看不到盒子里面的东西。内部执行者被有意地回避掉，这使我们能合理地使用用例去命名一个还没有被设计出来的系统的需求。

有时，我们想以用例的形式来记录系统中各部分是如何相互协作来正确地提交一个行为。当我们在记录业务过程的时候，如第1章中用例5“买东西（完整正式版本）”和第5章用例19“处理申请（业务）”，就使用的是这种做法，。我们也可能在描述一个大的、多计算机系统的设计时使用这种方法，如第3章中的用例8“输入和更新请求（联合系统）”。在这些情况下，系统中的构件就被描述为执行者。

当我们考查系统的内部，命名构件及其行为时，我们把系统作为一个白盒来对待。关于编写用例的所有方法仍然适用；不同的是，我们同时对系统内部执行者和外部执行者的行为进行讨论。在白盒用例中，不仅仅有两个执行者，因为除了外部执行者，系统中的构件也被表示出
59 来了。

把白盒用例作为被设计计算机系统的行为需求来编写，这种情况极为少见，而且通常这也是一种错误的做法。

4.6 练习

执行者和项目相关人员

4-1 识别出自动售货机的一个用例，在该用例中，主执行者是售货机的拥有者。

4-2 假设需要为一台新的ATM创建需求文档。试判断下面的列表中哪个是项目相关人员、主执行者、辅助执行者、被设计系统或根本就不是一个执行者（或者同时具备上述角色中的多个角色）：

ATM

客户

ATM卡

银行

前台
银行所有者
服务人员
打印机
主银行计算机系统
银行出纳员
银行抢劫犯

4-3 ATM是一个更大系统中的一个构件。实际上，它是几个较大系统中的一部分。针对一个这样的包含系统重复前面的练习。

4-4 “私人顾问公司”（一个假想的公司）推出了一个新产品，该产品使得人们能够审查自己的金融投资策略，例如退休、教育基金、土地和股票等。这个产品“私人顾问/金融（PAF）”以CD的形式发行。用户完成安装以后，可以运行各种各样的金融场景以便了解如何使其金融前景得到优化。PAF还可以询问许多税务包，如应用于税务法规方面的Kiplinger Tax Cut。私人顾问公司正在争取Kiplinger的同意以便允许直接进行交易。他们还在就基金和股票在Web上的直接买卖等事宜，同许多其他共有基金和Web股票服务商（如Vanguard 和E*Trade）进行协商以期达成一致意见。公司认为拥有一个基于“使用即付款”的网络化版本的PAF也不失为一个很好的想法。

指定和识别PAF的执行者、主执行者、辅助执行者、被设计系统和包含系统（一个将PAF作为其构件的系统）。

60

第5章
三个命名的目标层次

场景中的目标和交互都可以被展开成更精细、更小粒度的目标和交互。这是很常见的，在日常生活中，我们能很好地处理这种情况。下面两段将阐明目标是怎样由子目标和子-子目标组成的。

我希望得到这份销售合同。为了达到目的，我必须要请经理出去吃饭。为了请经理吃饭，我必须获得一些现金。为了获得现金，我必须从这台ATM中取钱。为了从ATM中取钱，我必须让它接受我的身份验证。为了让ATM接受我的身份验证，我必须让它读取我的ATM卡。为了让ATM读取我的ATM卡，我必须找到插卡孔。

我想找到Tab键，这样我就能将光标置于地址域内，这样我就能输入我的地址信息，这样我就能将我的个人信息输入到报价软件中，这样我就能得到报价，这样我就能买到一张汽车保险单，这样我就能使我的车获得驾驶许可，这样我就能开车了。

在日常生活中，无论这些情况多么平常，在编写用例时都会产生疑惑。针对每一个句子，编写者都面临这样一个问题“我应该描述哪个层次上的目标呢？”

给目标层次命名很有帮助。下面几节描述了我们认为很有用处的几个目标层次名称和图标，以及在需要时如何找到目标层次的方法。图5-1描述了所采用的名称和可视化比喻。

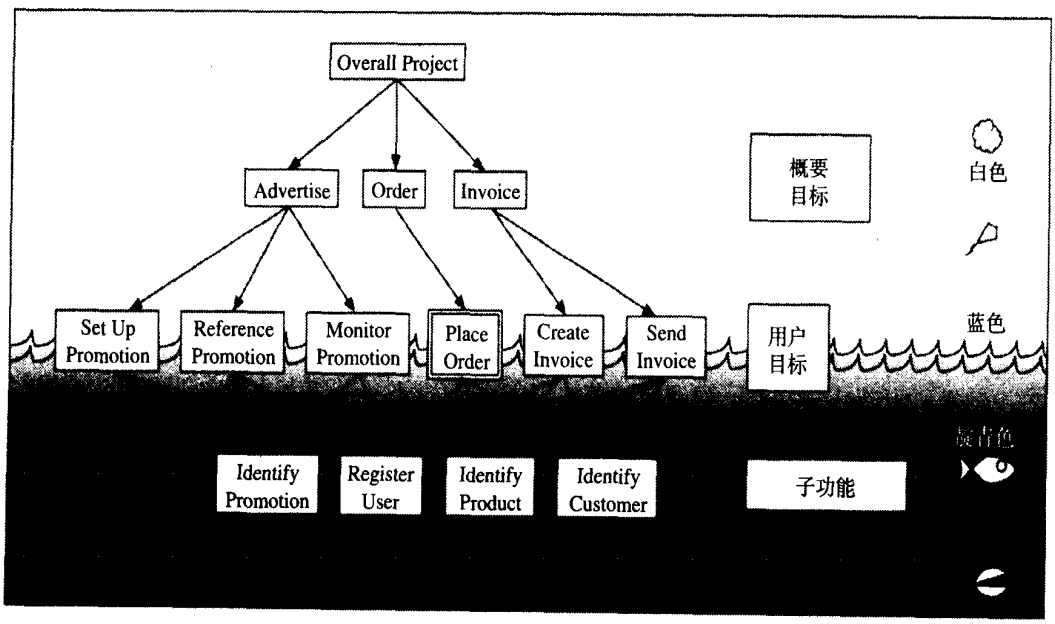


图5-1 用例层次

用例集展示了一个层次化的目标结构——一个不断发展的故事。

5.1 用户目标（蓝色，海平面）

我们最感兴趣的是用户目标（*user goal*）。它是主执行者努力使工作得以完成的目标，或是用户使用系统的目标。它相当于业务过程工程中的“基本业务过程”。

用户目标陈述了这样一个问题“这个操作执行完以后，主执行者会高兴地离开吗？”对职员来说，意味着“你的工作情况是否取决于你今天做了多少工作？”再比如工间休息测试：“做完这件事之后，我能获得工间休息？”在多数情况下，用户目标应通过一人、一次处理完成测试（2～20分钟）。

“在线拍卖中完成一次购买”和“登录”都不能算是一个用户目标。在线拍卖要持续几天的时间，不能通过“一次处理完成”的测试。即使连续登录42次也不能完成这个人的工作任务或达到使用该系统的目的。

“注册新客户”和“买书”可以是用户目标。注册42个新客户对一个销售代表来说具有重要意义。书的购买活动可以一次处理完成。

到目前为止，一切看起来都很简单。但是，当面对一个白板上的很多短语，或者一个由于某种原因看起来不太正确的用例时，一切又将变得不确定起来。我们发现，无论是采用颜色还是采用海拔高度对用户目标进行描述，大多数人都能确定自己的方位。

颜色梯度是从白色，到蓝色，再到靛青色，再到黑色（在本书中用灰色的阴影表示）。用户目标是蓝色的。更大范围、更高层次的目标，例如“在线拍卖中完成一次购买”和“获得交通事故赔偿”是白色的。更小范围、更低层次的目标是靛青色的。黑色表明目标层次太低，以致为它编写一个用例将是不合适的。比如“击tab键”就是一个很好的例子。 62

海平面比喻的思想是：天空延伸至海平面以上很高的地方，海水延续到海平面以下很深的地方，但是天空和海水的交界却只有一个——海平面。关于目标也是同样的道理。在用户目标之上和之下都各自存在着许多目标层次，但是用户目标才是需要被编写的重要目标。因此，海平面（波浪）与用户目标相对应。云朵或风筝表示高于海平面；鱼或蛤表示低于海平面。

系统的价值是通过它对海平面目标的支持来判断的。下面就是一个这样的目标：

> 你是一个职员，正在值班。电话铃响了，你拿起电话。打电话的人说：“……”，你转向计算机。当时，你的头脑中想的是你需要实现G。你跟计算机和客户进行了一段时间的交互，最后实现了G。你从计算机旁走开，说，“再见！”，然后，挂上电话。

G是蓝色或者海平面用户目标。在实现G的过程中，你实现了一些其他低层次的（靛青色）目标。打电话的人的头脑中可能有一个更高层次的目标，实现G只是实现那个更高层目标过程中的一步。那个人的更高层目标是白色的。

海平面/蓝色用户目标极为重要，因此值得花大力气去理解它们，并使它们内在化。系统所支持的用户目标列表就是关于系统功能的一个最短的概述——这是进行优先级划分、发布、分组、评估和开发的基础。

在海平面以上和以下的层次上都有可能要编写用例。把大量的低层目标和用例想像成是“海平面以下的水”，会为我们分析问题提供许多方便，因为那就意味着我们不必真正去编写和阅读它们。

◆ 一个简短而真实的故事

我曾经收到一个长达100多页，写满了用例的文档，这些用例都是靛青色的，或者说在海平面以下。那个需求说明文档是如此之冗长乏味，以至于它对编写者和阅读者都没有带来什么帮助。那个人后来又发给我一份经过改写的文档，这次他用6个海平面用例代替了那100多个靛青色用例，并告诉我每个人都发现这样更容易理解和处理。

5.1.1 蓝色的两个层次

通常，蓝色用例之上会有一个白色用例，之下会有几个靛青色用例。然而，它有时会引用
63 另一个蓝色用例。这只在一种情况下发生，但这种情况却经常发生。

假设我在某次出差时，路过一家录像租借商店。我想“恰好走到这了，我最好现在顺便注册一下。”于是我走进商店，要求“建立会员关系”。这是我的用户目标，即蓝色用例。下个星期，我拿着我的会员卡走进去“租借一个录像”。我的这两个用户目标是在不同的日子里执行的。

但是，可能你租借时的情形跟我的情形不同。你走进这家录像商店想“租借一个录像”。职员问“你是会员吗？”你说，“不是”，于是在你第一次租借录像的过程中，职员会让你“建立会员关系”。“建立会员关系”是“租借一个录像”中的一个步骤，尽管这两个目标都是蓝色目标。

在我的记忆中，“给顺便路过的人注册”是惟一一种我见到过的一个蓝色用例被包含在另一个蓝色用例当中的情况。当被问及此事时，我半开玩笑地说，它们两个都在水平面上，但是“租借一个录像”坐在一个如图5-1所示的浪尖上，而“建立会员关系”却在浪底。

5.2 概要层次（白色，云朵○，风筝🪁）

概要层次目标（*summary-level goal*）[⊖] 包含多个用户目标。在描述系统时，它们有如下三方面的功能：

- 显示用户目标运行的语境。
- 显示相关目标的生命周期顺序。
- 为低层用例（包括白色用例和蓝色用例）提供一个目录表。

概要用例在颜色梯度上是白色的。白色用例包含白色、蓝色，偶尔甚至会有靛青色的执行步骤（“登录主机”是在白色用例中很常见的一个靛青色目标）。我们发现区分白色的不同层次没有什么用处，但是，偶尔也有人会说“那个用例真的是白色的，白得深入云端”。按照海平面比喻的说法，我认为多数概要层用例“像一个刚刚高出海平面的风筝”，而其他概要用例则是“深入云端”。

⊖ 在以前的写作中，我采用过“决策”（Strategic）和“概要”（Summary）两个词。近来发现“概要”一词更清楚一些，因此，在本书中采用了该词。

概要用例通常需要执行几个小时、几天、几个星期、几个月、甚至几年。下面是一个运行时间很长的用例的主场景，其目的是为了把分散在几年中的用例维系在一起。我们将注意到其图标强调了此用例是把公司作为一个黑盒来对待的，并且目标层次是相当白的（深入到了云端）。加下划线的句子表示低层次用例。注意名称后的符号“+”。添加“+”后缀是一种表示概要层用例的可选方法，这一点将在5.4节中进行进一步阐述。 64

用例18 操作保险单 +

主执行者：客户
范围：保险公司（MyTelCo）
层次：概要（白色）
步骤：
1. 客户得到一个保险单的报价。
2. 客户买了一个保险单。
3. 客户针对这个保险单提了一个申请。
4. 客户关闭这个保险单。

本章中的其他白色用例是
- 用例19“处理申请（业务）”。
- 用例20“评估工作补偿申请”。
- 用例21“处理申请（系统）”。

5.2.1 重温最外层用例的内容

前面，我们曾建议为正在设计的系统编写一些最外层用例。下面是找到这些用例的一个更为详细的过程描述。

1) 以一个用户目标作为开始。
2) 提问“这个目标对主执行者AA（最好在组织外部）提供什么服务？”执行者AA是我们想要收集的用例的最终主执行者。
3) 找出最外层设计范围S，使得AA仍然在S之外。给S命名。通常，我能找到三个这样的最外层设计范围：
   - 公司
   - 组合软件系统
   - 被设计的具体软件系统
4) 找出最终主执行者AA在设计范围S中的所有用户目标。
5) 找出执行者AA对系统S具有的概要目标GG。
6) 为执行者AA对系统S的目标GG编写概要层用例。这个用例将一些海平面用例维系在一起。

即使在规模最大的系统中，通常也仅有4～5个这样的最高层用例。它们概括了3～4个最终
65 主执行者（AA）的利益。

- 对公司而言：客户
- 对组合软件系统而言：市场部
- 对软件系统本身而言：安全部

这些最外层用例对于在总体上将工作联系起来很有帮助，基于以上原因，我们极力推荐编写最外层用例。然而，最外层用例不能为开发组提供被创建系统的功能需求——那些内容包含在用户目标（蓝色）用例中。

5.3 子功能（靛青色/黑色，海平面以下/蛤）

子功能层次（*subfunction-level goal*）的目标是指那些在实现用户目标时可能会被用到的目标。只有当迫不得已时才把它们包含进来——需要它们是因为可读性方面的考虑，或者因为有许多其他目标用到它们。子功能用例的例子有“找到一个产品”、“找到一个客户”以及“保存为一个文件”。具体的例子请参见本章用例23（非同寻常的靛青色用例）“查找……（问题声明）”。

子功能用例是在海平面以下的，靛青色的。有些子功能用例简直深得触到了海底。那些黑色用例意味着“这个功能层次太低了，请不要把它扩展为一个用例。”（“它甚至不游泳……它是一个蛤！”）。我们认为给这些超低层用例起一个特殊的名称是一种明智的做法，因为这样一来，当有人写这样的功能时，你可以指出不必为它们编写用例，它们的内容应该被融合到另一个用例当中。

蓝色用例中有一些靛青色步骤，而靛青色用例中有一些颜色更深的靛青色步骤，如后面图5-2中所示。此图还表明，为了给目标语句找到一个更高层的目标，应回到“执行者为什么要做这件事？”这样一个问题，这个“如何做/为什么做”技术在5.5节中有更详细的讨论。

注意，即使最深的海水，最底层的子功能用例，*在*系统之外也会有一个主执行者。但不必在这一点上花费过多的精力，除非偶尔将其作为内部设计的议题来谈论，或看上去缺少一个主执行者时，才有必要考虑子功能用例的主执行者。适用于用例的所有规则对子功能用例同样适用。很可能，一个子功能会与引用它的高层用例具有相同的主执行者。

5.3.1 目标层次总结

至此，关于目标层次有三个重要的注意事项：

- 把较多的精力投入到对海平面用例的考查上，它们是重要的用例。
66 • 编写一些最外层用例来为其他用例提供语境。
- 不要在“是否把系统需求规格说明语句中你最喜欢的那个措辞用作用例的标题”上面小题大做。

把它用作用例的标题并不意味着“它是最重要的需求”，不把它用作用例的标题也不意味着“它不重要”。我见到过这种情况，有人因为自己最喜欢的需求仅仅被当作用例中的一个步骤，

而没有被提升为一个被独立描述的用例而感到沮丧。

千万不要为这种事烦恼。关于目标模型需要注意的一点是，把一大段复杂的文本编写成一个独立的用例，或把一个小的用例折叠成高层用例中的一个步骤时，所做的改动相对来说是很小的。所写的每一个句子都代表一个目标，每一个目标都可以被展开成一个独立的用例。我们不能仅通过阅读文档就判断出哪个句子被展开，哪个句子没有被展开（除非有链接可跟踪）。这是好的，因为它保证了文档在经历小的改动时的一致性。惟一要保证具有自己独立用例的目标是那些蓝色目标。

5.4 利用图标来突出目标层次

在第3章3.2.1节“用图标来突出设计范围”中，给出了一些图标，将其置于用例标题的左侧很有用处。由于目标层次和标题一样令人迷惑，因此我们在标题的最右端放置了一个目标层次图标。这是除了用例模板中列出的内容外，又额外添加的一项内容。读者（和编者）可以在其帮助下了解层次信息。

为了与海拔高度命名法保持一致，我们区分出了五个海拔高度。在多数情况下，只需要使用中间的三个就足够了。

- 将高度概括的（非常白的）用例标以云朵。在极为少见的偶然情况下，你发现用例中某些步骤本身就是白色目标，这时使用云朵图标。如果不能添加图标，那就在用例名称末尾添加一个加号（+），如用例18。
- 将概要（白色）用例标以风筝。这适用于大多数执行步骤为蓝色目标的概要用例。同样，如果不能添加图标，就在用例名称末尾添加一个加号（+）。
- 将用户目标（蓝色，海平面）用例标以波浪。如果不能添加图标，就在用例名称末尾添加一个感叹号（!）或什么也不添加。
- 将子功能（靛青色）用例标以小鱼。这对多数靛青色用例来说是适用的。如果不能添加图标，就在用例名称末尾添加一个减号（-）。
- 对于一些子功能（黑色）是不必编写用例的。使用蛤图标来标记需要合并到其调用用例当中的用例。

67

有了这些图标，就可以对设计范围和目标层次进行标注了。只要工具提供支持，甚至可以在UML用例图上标注这些图标。可以立即开始使用后缀表示法。如果你的模板中已经包含了设计范围和目标层次域，可以把它们用作冗余标记。如果你的模板中没有这些域，请把它们添加进去。

5.5 找出正确的目标层

找出正确的目标层是关于用例的一个最棘手的问题。注意下面两个指导原则：

- 找出用户目标。
- 对每个用例执行第3步到第10步。

5.5.1　找出用户目标

在所有的目标层中，只有一个目标会从其他目标中脱颖而出。

你正在描述一个系统——一个业务系统或者一个计算机系统。你关心某人对系统的使用情况。那个人现在想从系统中获得某些东西。获得这个东西以后，她就可以继续去做其他的事情。她现在想从你的系统中得到什么呢？

这个层次有多个名称。在业务过程建模中，它被称为“基本业务过程”（*elementary business process*）。在法语中，它是系统的“*raison d'être*”。在用例中，它是用户目标。

问这样一个问题，“这是主执行者现在真正想从系统中得到的东西吗？”对多数用例的初稿来说，答案是“不是”。多数初学者在草拟用例时描述的是海平面以下的目标，却以为这些目标处在海平面层次以上。为了找到更高层的目标，试问自己下面这两个问题之一：

- 主执行者真正想要的是什么？
- 执行者为什么做这件事？

答案可能就是执行者的目标，但仍需要反复提出这个问题，直到你能肯定为止。有趣的是，尽管关于用户目标的测试是主观的，但人们却能很快对此达成共识。有经验的人在用户目标这
68 一点上的答案是出奇得相似，看起来就像是一个固定的概念。

5.5.2　提升和降低目标层次

用例中的交互步骤描述了一个过程是如何完成的；用例名称表明了该过程的意义所在。在决定一个交互步骤处于哪一个目标层次上才合适时，可以用这种“如何做/为什么做（How/Why）”的关系（见图5-2）。

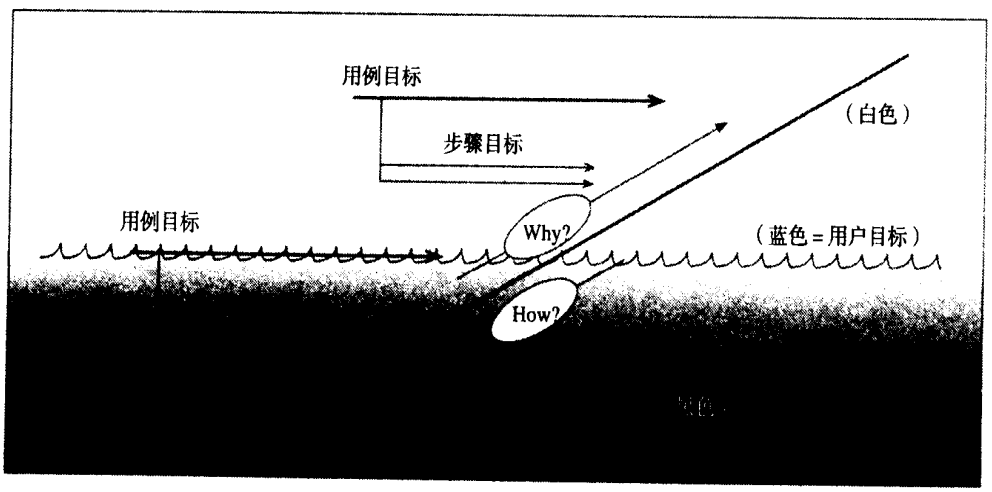


图5-2　通过问“为什么”的问题来转换层次

为了提升一个或几个交互步骤的目标层次，思考这样一个问题，“执行者为什么做这件事？”，答案可能就是较高一层的目标。

考查用例的长度也是判断目标层次的一种方法。多数编写得好的用例一般有3步～8步。几乎所有多于11步的用例都可以被裁短而不影响其表达。这些数字似乎有一些魔法般的奥妙，我想可能是因为，对于一个要通过10个以上中间步骤才能完成的过程来说，人们感觉是不能容忍或难以想像的。至今我们还未见过一个合理的反例能证明交互步骤数的多少对用例质量不太重要。

无论出于什么原因，都可以用这个观察方法来提高用例的编写质量。如果你写的用例中包含了10个以上的交互步骤，那可能是涉及了用户界面的细节信息或是执行步骤层次太低了。

- 删除用户界面细节信息。显示执行者的目的，而不是他的动机。
- 提升目标层次，通过问“为什么”的问题找到较高一层的目标层次。
- 将某些交互步骤合并。
- 将你的用例与5.6节和第19章“修复错误”中的编写范例进行对比。

69

5.6 一个较长的编写实例：“处理申请”的多层次示范

我要感谢加利福尼亚州Novato的“消防员基金保险公司”中的人员，是他们允许我把用例19～用例23作为编写范例包含进来。这些用例都是由申请处理领域中的专职人员编写而成，同时在用例创作过程中发挥关键作用的还有来自IT部门的业务分析专家和来自技术部门的工作人员。领域人员对系统的使用情况有深入的了解，而这是IT人员所猜想不出来的，但IT人员可以帮助领域人员将用例编写得尽量准确。他们把领域知识、合作精神和技术观点紧密地结合在他们的工作当中。

编写小组包括Kerry Bear、Eileen Curran、Brent Hupp、Paula Ivey、Susan Passini、Pamela Pratt、Steve Sampson、Jill Schicktanz、Nancy Jewell、Trisha Magdaleno、Marc Greenberg、Nicole Lazar、Dawn Coppolo 和Eric Evans。他们的合作证明：在编写需求时，没有软件知识背景的领域专家可以和IT人员共同开展工作。

我们用了5个用例来说明我们至此所讨论的内容，尤其是设计范围和目标层次。这些用例也展示了用例中交互步骤和扩展的良好的写作风格。我们在每个用例前面都加了一段说明，说明应引起注意的要点和内容。

这些以云朵层次、白盒业务用例开始的用例集表明了与处理申请有关的业务过程。通过它们，可以观察这些目标是如何逐步进入低层次，以及系统范围是如何从“公司操作”缩小到“所有计算机系统”，再缩小到“被设计系统”。加下划线的短语表明引用到了其他的用例。用例模板被稍微修改了一些，这样主成功场景就可以与顶层更加接近，便于更快地阅读。

关于用例19的说明。SuD是公司的业务运作。注意计算机系统甚至没有被提到过。企业将利用用例去定位它的业务过程，并寻找一种运用计算机来使业务运作过程更加便利的途径。此时，用例只是初步被勾画出一个概要。跟通常一样，主成功场景看起来很琐碎，实际就应如此。它表明了在最成功的情况下，事情是如何进行的！失败情况的发生以及公司如何利用该信息来提高IT对运作的支持能力是其中很重要的部分。另外，不能忽视项目相关人员。

70

用例19　处理申请（业务）

范围：保险公司业务运作。
层次：业务概括
发布情况：将来
状态：草稿
修订情况：当前
使用语境：申请调节者处理申请。
前置条件：损失发生。
触发事件：保险公司接到一个申请报告。
主成功场景：
1. 知情当事人（即报告方）到保险公司登记损失。
2. 职员收到申请，分配申请给申请调节者。
3. 被指定的申请调节者：
   进行调查
   评估损失
   设置追偿权
   就申请进行谈判
   解决申请并关闭申请

扩展：
待写。
成功保证：申请被处理并关闭。
最小保证：无
项目相关人员和利益：
负责出售保险单的保险公司分部
购买保险单的保险公司客户
设置市场行为的保险部门
因某种受保行为而造成损失的申请人
保险公司申请分部
未来客户

关于用例20的说明。这是另一个业务用例，在此用例中，SuD仍然是公司的业务运作。但是，与用例19相比，此用例的目标层次较低。它显示了调节者的工作可能会持续几天、几个星期或几个月。因为它包含许多一次处理完成的活动，因此是一个风筝层次的概要用例。

编写者没有直接提及计算机，只是命名了调节者的目标。开发组必须发挥想像力，来构想计算机在这个过程中会提供怎样的帮助。这个用例是他们构想时的素材。

71　添加第7步是出于对保险部门利益的考虑。

用例20 ⌂评估工作补偿申请 ⌕

范围：保险公司业务运作⌂。
层次：白色（概要，单用户目标层次之上）
使用语境：申请调节者完成对损失事实的全面评估。
主执行者：申请调节者
前置条件：有文件资料
触发事件：有文件资料
主成功场景：
请注意：进行评估前，已经很理想地完成调查工作，调查的深浅因申请不同而各异。
1. 调节者审查并评估医检报告、抵押文件、保险期限和其他辅助文档。
2. 调节者根据确定伤残百分比率的司法程序，评定永久性伤残等级。
3. 调节者对应付的永久性伤残赔偿金额进行汇总，计算预付款和应付的抵押利息以得到申请总值。
4. 调节者决定最终结算范围。
5. 调节者检查追偿权以确保它们同结算范围相一致。
6. 如果超出了他/她的职权范围，则调节者寻求结算和追偿权增加认证。
7. 调节者将文件归档。
8. 调节者在必要时向有关当事人发送信件和/或文档材料。
9. 调节者继续进行与结算活动相关的文件归档工作。

扩展：……
发生频率：每个申请都被评估；一天中可发生几次。
成功保证：申请被评估，结算范围被确定。
最小保证：完成额外的调查和医学验证，直到申请准备再次被评估以做出结算的条件。
项目相关人员和利益：

申请者——希望获得最大结算。
调节者——希望有最低的合法结算。
保险公司——一样。
被保险者的律师（原告和被告）。
保险分部，和州主管办公室（每一个州都有一个独立的管理部门来监督结算的行政利益和公证性），希望公正，照章办事。

待解决的问题：在编写业务规则时要陈述相应的司法问题。

72

关于用例21的说明。对于此项目中的大多数人员来说，这个系统用例看起来很含糊，似乎没有什么用处。但是，在其编写工作上花费时间仍然是值得的，主要体现在以下几个方面。

首先，它把一些用户目标用例紧密地连接在一起，显示了它们如何与业务方针相配套。它描述了关闭、清除和获得一个申请的过程，而该项目中的许多人员对这些内容都不太了解。尽

管最后三步没有为编程人员做什么工作，但它们是处理申请事务情节中的一部分，对每位读者来说，它们提供了很有用的语境信息。

第二，它在正式文件中加入了一些业务规则，开发小组中有些人对这些规则一无所知。在以前，开发小组要花费三个工作小时试图去猜测这些业务规则。一旦编写了这个用例，就会在这个主题的讨论上节省很多时间。

对于像公司管理人员和新雇员这样的新读者来说，这个用例可以作为业务简介和目录表。管理人员可以看到关键过程被包含了进来；新雇员可以了解公司的运作过程，然后深入到用户目标用例中。扩展*a1很有意义，因为它引出了一个失败处理用例，该用例不能由申请调节者来编写，而是要由技术小组来编写。

用例21　处理申请（系统）+

范围："系统"是指所有组合计算机系统。
层次：概要（白色）
发布情况：第1个版本
状态：为审查做好准备
修订情况：当前
使用语境：客户想获得事故赔偿。
主执行者：客户
前置条件：无
触发事件：客户报告一个申请。
主成功场景：
1. 客户向职员报告了一个申请（通过书面、电话或传真）；
2. 职员找到客户的保险单，在系统中登记损失情况，然后指定一个调节者。
3. 调节者调查这个申请，用附加信息修改这个申请。
4. 调节者随时输入过程记录。
5. 调节者修正条目，并随时留出一些钱。
6. 调节者收到一些文档，包括申请整个生命周期的票据，然后输入票据。
7. 调节者评估这个申请的损失，并在系统中归档谈判过程。
8. 调节者支付并关闭系统中的申请。
9. 系统在申请关闭6个月后清除该申请。
10. 时间期限过后，系统将申请归档。

扩展：
*a. 任意时刻，系统崩溃：
　　*a1. 系统小组检修系统。
1a. 提交的数据不完整：
　　1a1. 保险公司要求补充遗漏信息。

1a2. 申请人提供遗漏信息。

1a2a. 申请人没有在规定时间内提供信息：

1a2a1. 调节者关闭系统中的申请。

2a. 申请人拥有的保险单不合法：

2a1. 保险公司拒绝该申请，通知申请人，然后修改申请，关闭申请。

3a. 当时找不到代理：

3a1.（这种情况下我们该怎么办？）

8a. 申请人通知调节者新的申请活动：

8a1.职员重新打开申请，返回步骤3。

技术和数据可变列表：

发生频率：有文件资料。

成功保证：申请被关闭，支付和归档。

最小保证：申请被关闭，但可能以后会被重新打开。

项目相关人员和利益：

公司——做最小的正确补偿结算。

客户——获得最大的补偿。

保险部门——确保程序正确。

业务规则：

数据描述：将在其他用例中定义。

UI连接：有文件资料。

待解决的问题：申请归档的时间间隔是多少?

关于用例22的说明。这是我们所见过的最复杂的用例之一。它展示了为什么用例应该用自然语言散文形式编写。

第一个复杂的原因来自于顺序。通过电话与一个疯狂的客户进行交谈的职员必须能以任何顺序输入信息，并且还要努力遵循一个标准的问题顺序。同时，计算机使用被输入的信息来做它所能做的任何处理工作，例如抽取客户记录、指定申请序号和调节者。编写者针对这个用例至少写了4个完整的版本，努力想做到清晰，以显示正常的工作路径以及计算机在异步方式下工作的情况。很可能在第7和第8个修订版中，他们还能发现一些更好的东西，但是他们感觉到回报越来越少，再花费更多的力气已没有多少实际意义，于是就到此为止了。 74

这个用例多次激活“查找……”用例，每次的查询内容和查询原则都不相同。开发小组想出了一种具有创新性的方法来避免重复书写查找过程中的标准步骤：匹配列表、排序原则、重新排列、重新查找、未找到搜索项等。为此，要求读者在本章结束时做一下练习5-4中的题目。

在处理扩展*a.“电力系统故障”时产生了意想不到的新的需求问题。它引入了中间保存的概念。突然进行中间保存表明职员以后还将再次进行查找。这对编写者来说是个意外，而且由此引入了存储和查询临时保存的损失问题——这令开发小组感到更加意外。它都以失败条件6b结束，失败条件6b处理临时保存损失的超时问题，它使编写者面临这样一个十分细节性的问题：

“非法临时进入的损失不能被提交，因为它缺少关键信息；但也不能被删除，因为它通过了最小进入准则；那么对此该采取怎样的业务规则呢？”在做出决定之前，开发小组揣摩难以接受的可选方案——不进行中间保存删除损失。

扩展1c显示了失败中的失败情况。编写者可以把这种情况单独写成一个用例，但是编写人员认为这样会在用例集中引入过多的复杂性：新的用例必须能被跟踪、审查和维护。因此采取了另一种方法，即将其作为扩展的扩展。当然采用扩展并非都为了此目的，但这也是原因之一。

扩展2-5a表明了这种方法的灵活性。条件可以在步骤2到步骤5中任何一步发生。那么该如何书写它呢？是在每次发生时吗？那看起来很浪费精力。解决的办法是只编写扩展2-5a和扩展2-5b，这样会感到很清晰。

用例22　损失登记

范围：“系统”是指申请捕获计算机系统。
层次：蓝色（用户目标）
发布情况：版本2
状态：已经过审查
使用语境：全面捕获损失
主执行者：职员
75
前置条件：职员已经登录。
触发事件：职员已经开始录入损失信息。
成功保证：损失信息被捕获和保存。
最小保证：什么也没发生。
项目相关人员和利益：同前
主成功场景：
为了提高职员的工作速度，一但获得所需要的数据，系统就应该异步地进行工作，职员可以根据当时的需要以任意顺序输入数据。据推测，下面的顺序是最常见的一种情况。

1. 职员输入被保险人的保险单号或者事故的名称和日期。系统自动填写保险单信息，以表明申请与保险单相一致。
2. 职员输入基本的损失信息。系统验证不存在其他的竞争申请，然后为该申请分配一个申请号码。
3. 职员继续输入与申请种类相关的具体的损失信息。
4. 职员让系统从其他计算机系统中提取其他赔偿金信息。
5. 职员选择并指定一个调节者。
6. 职员确认他们的操作已经结束；系统保存和触发发送给代理的认可信息。

扩展：
*a. 在捕获损失的过程中电力系统发生故障：
　*a1. 系统立即自动保存（可能在某个事务提交点，待解决的问题）。

*b. 申请不归我们公司处理：

*b1.职员通知系统输入申请“只是为了做记录”，之后或者继续或者终止损失处理。

1a. 发现保险单信息与被保险者的信息不相符：

1a1. 职员输入正确的保险单号码或被保险者的姓名，让系统填充新保险单索引信息。

1b. 使用查询细节，系统没有找到保险单：

1ba. 职员回退到损失，输入可用数据。

1c. 职员在初始保险单匹配后，改变保险单号码、损失数据或申请种类：

1c1. 系统对更改进行验证，利用正确的保险单信息来填充损失，确认并指明申请与保险单相匹配。

1c1a. 系统不能确认保险单是否匹配：

1c1a1. 系统向职员发出警告信息。

1c1a2. 职员利用保险单的查询细节查找保险单。

1c2. 系统警告职员要重新评估赔偿金额。

1d. 职员想重新启动一个被中断、被保存或者需要完成的损失：

1d1. 职员利用损失的查询细节查找损失。

1d2. 系统打开它进行编辑。

76

2-5a. 职员修改以前输入的申请信息，但不输入与种类相关的数据：

2-5a1. 系统根据职员输入的申请种类提供损失中相应的种类相关数据。

2-5b. 职员更改以前输入的申请种类，并且在一些与种类相关的域中有数据存在：

2-5b1. 系统警告职员这些数据的存在，并要求职员或者取消更改或者以一个新的申请种类继续。

2-5b1a. 职员取消更改：系统继续处理该损失。

2-5b1b. 职员坚持新申请种类：系统将种类相关的数据置空（保留所有基本的申请的数据）。

2c. 系统检测到可能存在重复的申请：

2c1. 系统列表显示损失数据库中可能的重复申请。

2c2. 职员从列表中选择浏览一个申请。这步操作可能会重复多次。

2c2a. 职员发现申请重复：

如果该申请没有完成标志，职员可打开申请列表中重复的申请进行编辑（基于职员的安全权限信息）。职员可以删除以前保存在文件中的任何数据。

2c2b. 职员发现申请没有重复：职员返回到损失，并完成它。

2d. 最初的重复申请检查完成之后，初步的损失信息被更改：

2d1. 系统再次执行重复申请检查。

2e. 在完成步骤2到6的操作之前，职员可以随时对损失进行保存（一些保存原因可能仅仅是出于心里安慰，或由于某种原因职员必须中断输入，比如，申请必须由更高层的调节者来处理，并要立即传送给调节者）。

2e1. 职员让系统保存损失以备将来完成。

4-5a. 在职员对内容进行审查之后，申请种类或损失描述（见业务规则）被更改：

4-5a1. 系统警告职员对内容重新进行评估。

6a. 在没有完成最小信息的情况下职员认可他/她已经完成：

6a1. 系统警告职员在没有损失日期、被保险者姓名、保险单编号和负责处理的调节者的情况下，他不能接受这个损失：

6a1a. 职员或者决定继续输入损失，或者决定在没有标记为完成的情况下进行保存。

6a1b. 职员坚持在没有输入最小信息的情况下退出：系统放弃所有的中间保存信息并退出。

6a2. 系统警告职员它在没有申请域（申请种类、损失日期、保险单编号或被保险者姓名）的情况下不能分配申请号码：系统向职员指示要求输入的区域。

77

6b. 超时：职员对损失进行了临时保存，打算返回；系统认为已到接受并将该损失记入日志的时候，但是还没有输入负责处理的调节者：

6b1. 系统分配一个默认的调节者（见业务规则）。

发生频率：？？

业务规则：

*. 何时将被保存的损失送入主系统（时机）？

1. 保存损失（要求能再次找到它）所需的最少域是：……

2. 申请编号一旦被系统指定就不能更改。

3. 手动输入申请编号的业务规则——需要吗?

4. 损失描述由两个域组成，一个是自由的形式，另一个是下拉菜单。

5. 系统应该知道如何根据保险单前缀查找赔偿金额。

6. 为了确认损失完成所需的域是：……

6b. 默认调节者规则是：……

所用到的数据描述：

保险单的查询细节、保险单索引信息、初始损失信息、与申请种类相关的损失信息、附加信息。

损失的查询细节、重复申请检查原则、可能的重复申请列表、列表中的一个申请。

UI连接：有文件资料。

拥有者：Susan和Nancy

关键审查者：Alistair,Eric...

待解决的问题：

多长时间进行一次自动保存?

代理通知卡——打印地点和方式，等等。

项目开发组认为编写几乎相同的用例“查找客户”和“查找保险单”是很愚蠢的。因此，他们创建了一个可供所有编写组成员使用的通用机制。

任何“查找……”形式的句子，例如“查找客户”或“查找产品”，都意味着用例“查找……”

会被调用，并暗示“……”将会被具体的东西所代替。每个用例都需要自己的查找、排序和显示准则，因此编写者要把数据和查找限制放入不同的、超链接表中。这个例句将被读作使用客户查找细节来查找客户。

有了这种巧妙的规则，“查找……”的逻辑细节就可以只写一次，却可以在许多相似但不相同的语境中使用。这让开发者感到高兴，因为他们知道所有的查找者都会使用相同的机制，而他们可以只开发一个通用的算法。 78

希望读者能试着编写该用例，因此在这里暂时不给出这个开发小组的解决方案。在看解决方案之前先做一下练习5-4，解决方案将在14.2节“参数化用例”中讨论。

用例23 查找无论什么（问题陈述）

在练习中补充。

5.7 练习

目标层次

5-1 Jenny正站在银行的ATM机前。天色很黑。她已经输入了她的PIN，正在找确认键。为Jenny分别设计一个白色的、蓝色的和靛青色的目标。

5-2 至少列举10个ATM的各种主执行者对ATM所具有的目标，并标注这些目标的目标层次。

5-3 列举练习4-4中描述的PAF软件包的所有主执行者的概要和用户目标。识别出最高层、最外层执行者-范围-目标组合。

5-4 **“查找……”**。为“查找……”编写一个用例，以用户打算定位某个内容作为触发事件。这个用例应该允许用户输入查找和排序信息。它还应该能处理可能出现的所有情况，并且在成功情况下，以计算机识别出调用用例在下一步中指定的要使用的“……”作为结束。 79

第6章
前置条件、触发事件和保证

6.1 前置条件

用例的前置条件（*precondition*）声明了启动该用例之前系统必须满足的条件。由于该条件由系统负责实施，并要求确保为真，因此在用例执行过程中，不必再对该条件进行检查。一个常见的例子是“用户已经登录并通过了身份确认”。

通常，前置条件是指该条件已经通过其他用例的执行进行了设置。如我们说“创建订单”依赖于“已经登录”这个前置条件，则应去查找是哪个用例设置了这个前置条件（这里应寻找“登录”用例）。就个人而言，我通常会创建一个更高层的用例，在该用例中包括“创建订单”和“登录”两个子用例，这样读者就可以清楚地看到这两个用例间的相互关系。在这个例子中，可以将其定义为概要用例：“使用应用软件”。于是可得到如下结构（这里对模板进行了删减以便只显示相关部分）。请注意这三个用例的目标层次。

用例：使用应用软件

> **层次：**概要（白色）
> **前置条件：**无
> 1. 职员登录。
> 2. 职员创建订单。

用例：登录

> **层次：**子功能（靛青色）
> **前置条件：**无
> ……

用例：创建订单

> **层次：**用户目标（蓝色）
> **前置条件：**职员已经登录。
> ……

并不是所有人都像我一样习惯于借助高层用例来体现低层用例间的相互关系。这样，就只能看到前面的例子中“登录”和“创建订单”两个用例。因此，必须能够从编写内容中推断出前置条件的正确性，并对之加以实施。

下面，我们继续通过这个例子来阐明一个用例在执行过程中某一步设置条件，然后在某个子用例中依赖该条件的情况。同样，子用例假定条件为真，不再对其正确性进行检查。

用例：创建订单

层次：用户目标（蓝色）
前置条件：职员已经登录
1. 职员识别客户，系统提取客户记录。
2. 职员输入订单信息。
3. 系统计算费用。
...

用例：计算费用

层次：子功能（靛青色）
前置条件：客户已被建立，系统中有该客户的信息；订单内容已知。
1. 系统计算订单的基本费用。
2. 系统计算客户折扣金额。
...

这个例子说明了子用例如何依赖调用用例所捕获的信息。“计算费用”的编写者对可用信息提前进行了声明，然后，在描写用例执行情况的过程中，就可以直接引用这些信息了。

机警的读者可能会对“计算费用”这个用例产生怀疑。这里将其声明为靛青色，但是到目前为止，从编写的内容中还难以判断它是否可以单独成为一个用例。作为编写者，如果不希望揭示系统与用户间复杂的交互过程或重要的失败情况，就可以重新把它归类为黑色（蛤为使用图标）。这是将文本重新合并到“创建订单”并彻底删除“计算费用”的标志。 82

将前置条件作为用例被打开时现实世界所处状态的一个简单断言来编写。下面是几个这样的例子：

前置条件：用户已经登录。
前置条件：客户已经通过身份确认。
前置条件：系统已经找到了客户的保险单信息。

在编写前置条件时通常易犯的一个错误是，把经常是正确的但不是必须的条件写入前置条件。

假设我们正在编写用例“索要保险赔偿金总额”，主执行者是申请人。我们可以假设在索要保险金总额之前，申请人至少已经提交了一个申请或账单。然而，情况并不总是这样；系统不能保证这个假设的正确性，实际上这不是一个必要条件。申请人应该能在任何时候索要他们的保险金总额，因此诸如“申请人已经提交了一个账单”这样的前置条件是错误的。

6.2 最小保证

最小保证是系统向项目相关人员作出的最低承诺，尤其是在主执行者的目标不能被满足的情况下。当然，在目标被满足的情况下它们仍然成立，但是当主要目标被放弃时它们就成为人们真正关心的事情。多数情况下，两个或更多的项目相关人员必须在最小保证中被提及，例如可以有用户、提供系统的公司，还可能有政府管理部门。

不必在最小保证中不厌其烦地列举出用例的所有失败情况。失败的情况多种多样，而且千差万别。所有的失败条件和处理方案都已在扩展区进行了描述，要使两个列表保持同步既麻烦又容易出错。在模板中加入最小保证的目的是为了声明系统的承诺。

最常见的最小保证是“系统将其执行进度情况记入日志”。为失败的事务处理做日志并不是一个显而易见的需求但却是至关重要的。在需求描述中，系统日志经常会被遗忘，后来又被程序员重新发现。然而，它们对系统拥有者和用户来说都是十分重要的。一旦正常的运行条件得到满足，系统就可以利用这些日志继续某项事务的处理；项目相关人员利用日志来解决纷争。
83 用例编写者应该能通过调查项目相关人员的利益或集中讨论失败条件而发现这些需求。

最小保证被写成一些简单的断言形式，无论用例如何被执行，这些断言最终都应为真。它表明每一个项目相关人员的利益都得到了满足。

最小保证：只有收到付款以后才启动订单。

最小保证：如果没有捕获到最小信息，那么不完整的请求就被丢弃，不对这个请求进行任何记录。如果最小信息被捕获（见业务规则），那么不完整的请求就被保存并记入日志。

在目标遭遇失败的情况下，项目相关人员认可他们的利益得到了保护，这是最小保证是否成功/失败的测试标准。

6.3 成功保证

成功保证（*success guarantee*）说明了用例成功结束后项目相关人员的哪些利益得到了满足，用例可以通过执行主场景获得成功，也可以通过执行可选路径获得成功。成功保证通常是作为最小保证的添加内容：最小保证被满足以后，并且一些附加条件为真；附加条件中至少包括用例标题中声明的目标。

像最小保证一样，成功保证被编写为一组简单的断言，这些断言应用于用例成功运行结束时，以表明每个项目相关人员的利益都得到了满足。下面举几个恰当的例子：

成功保证：申请人按照商定的金额得到赔偿，申请被关闭，解决方案被记入日志。

成功保证：文件将被保存。

成功保证：系统为客户创建一个订单，收到付款信息，并将订单请求记入日志。

项目相关人员认可他们的利益得到了满足，这是成功保证是否成功/失败的测试标准。

找出成功保证的最好方法是问这样一个问题，“在用例成功结束时，什么事会使项目相关人员感到不高兴呢？”这个问题通常很容易回答。然后写出答案的反面回答。如果要参见一个实例，请做练习6-4，然后阅读附录B中的讨论。

6.4 触发事件

触发事件（*trigger*）指明了启动用例的事件。有时，用例执行过程的第一步紧接着触发事件发生，有时触发事件就是用例中的第一步操作。到目前为止，还没有见到有说服力的证据可以说明一种形式可以适用于任何情况，也没有发现在选择使用这两种方法时会产生迷惑的情况。 84
读者应培养自己的风格或项目风格。

设想一个ATM系统只有当用户插入他/她的信用卡时才会被激活。此时，说触发事件是当有人决定去使用ATM时是没有意义的。触发事件“客户插入信用卡”也是用例的第一步。

用例：使用ATM

触发事件：客户插入信用卡。
1. 客户插入信用卡，卡上记录着银行ID、银行账号和加密的PIN等信息。
2. 系统确认……

现在，设想一个终日坐在显示很多图形符号的工作站前的职员要运行一个截然不同的应用程序。触发事件是客户通过电话提出一个具体的请求，该事件可以用上述两种情况中的任意一种进行描述。下面用第二种形式加以说明。

用例：将抱怨记入日志

触发事件：客户通过电话发出抱怨。
1. 职员唤起应用程序。
2.系统调出职员最近使用过的抱怨列表。
...

6.5 练习

最小保证

6-1 写出从ATM机中提取现金的最小保证。

假设我们正在编写用例“索要保险赔偿金总额”，主执行者是申请人。我们可以假设在索要保险金总额之前，申请人至少已经提交了一个申请或账单。然而，情况并不总是这样；系统不能保证这个假设的正确性，实际上这不是一个必要条件。申请人应该能在任何时候索要他们的保险金总额，因此诸如“申请人已经提交了一个账单”这样的前置条件是错误的。

6.2 最小保证

最小保证是系统向项目相关人员作出的最低承诺，尤其是在主执行者的目标不能被满足的情况下。当然，在目标被满足的情况下它们仍然成立，但是当主要目标被放弃时它们就成为人们真正关心的事情。多数情况下，两个或更多的项目相关人员必须在最小保证中被提及，例如可以有用户、提供系统的公司，还可能有政府管理部门。

不必在最小保证中不厌其烦地列举出用例的所有失败情况。失败的情况多种多样，而且千差万别。所有的失败条件和处理方案都已在扩展区进行了描述，要使两个列表保持同步既麻烦又容易出错。在模板中加入最小保证的目的是为了声明系统的承诺。

最常见的最小保证是“系统将其执行进度情况记入日志”。为失败的事务处理做日志并不是一个显而易见的需求但却是至关重要的。在需求描述中，系统日志经常会被遗忘，后来又被程序员重新发现。然而，它们对系统拥有者和用户来说都是十分重要的。一旦正常的运行条件得到满足，系统就可以利用这些日志继续某项事务的处理；项目相关人员利用日志来解决纷争。
83 用例编写者应该能通过调查项目相关人员的利益或集中讨论失败条件而发现这些需求。

最小保证被写成一些简单的断言形式，无论用例如何被执行，这些断言最终都应为真。它表明每一个项目相关人员的利益都得到了满足。

最小保证：只有收到付款以后才启动订单。

最小保证：如果没有捕获到最小信息，那么不完整的请求就被丢弃，不对这个请求进行任何记录。如果最小信息被捕获（见业务规则），那么不完整的请求就被保存并记入日志。

在目标遭遇失败的情况下，项目相关人员认可他们的利益得到了保护，这是最小保证是否成功/失败的测试标准。

6.3 成功保证

成功保证（*success guarantee*）说明了用例成功结束后项目相关人员的哪些利益得到了满足，用例可以通过执行主场景获得成功，也可以通过执行可选路径获得成功。成功保证通常是作为最小保证的添加内容：最小保证被满足以后，并且一些附加条件为真；附加条件中至少包括用例标题中声明的目标。

像最小保证一样，成功保证被编写为一组简单的断言，这些断言应用于用例成功运行结束时，以表明每个项目相关人员的利益都得到了满足。下面举几个恰当的例子：

成功保证：申请人按照商定的金额得到赔偿，申请被关闭，解决方案被记入日志。

第7章
场景和步骤

用例集是一个可以不断展开下去的故事，由主执行者需要实现的目标组成。每一个用例都显示了系统完成目标或放弃目标相互交织的情节。故事的主线表示为一个主成功场景，一些场景片段作为主线的扩展。每一个场景或场景片段都从一个触发条件开始，这个触发条件指明了什么时候场景开始执行，并一直执行到完成目标或放弃目标。就像我们所看到的，目标大小虽然各不相同，但我们可以使用相同的编写格式在不同的场景级别上描述各种大小不同的目标。

7.1 主成功场景

我们经常通过这样一种方式来向别人解释一些事情，即首先采用易于理解的描述，然后再补充说："好了，实际上，还要复杂一点。当这样或那样的事情发生的时候，真正要发生的是……"

人们很擅长使用这种解释方法，在我们编写那些将成为用例的交叉情节时，也使用了这种方法。我们首先编写一个自顶向下的描述，这个描述中包含一个容易理解的相当典型的场景，在该场景中，主执行者完成了目标，所有项目相关人员的利益都被满足了。这个场景就是主成功场景（main success scenario）。其他的成功场景和所有错误的处理，都会在主成功场景的扩展中进行描述。

7.1.1 常见的环境结构

主成功场景和所有场景扩展都可以包含在由以下元素组成的结构中：

- 场景执行的条件。对于主成功场景，条件就是前置条件加上触发事件。对于扩展场景，条件就是扩展条件（扩展条件可能带有步骤编号或者就写在条件生效的场景中）。
- 完成的目标。对于主成功场景，目标就是用例的名称，也就是满足项目相关人员的利益。对于一个扩展场景，目标是完成用例目标，或者是在处理扩展场景后重新进入主成功场景。
- 执行步骤集。这些活动构成了场景的主体。在每一个场景和场景片段中，这些活动都遵循相同的规则。
- 结束条件。主成功场景结束时完成的目标。一个场景的片段可能随着目标的完成或放弃而结束。
- 作为场景片段的、可能的扩展集。主成功场景的扩展写在用例模板中的扩展部分。扩展场景的扩展可以嵌入到扩展体中，或者写在扩展体里面，或者就写在扩展体的后面。

下面是从用例1中摘录出的两部分，摘录这两部分的目的是说明它们的环境结构的相似之处。

以下是一个主成功场景：

用例：通过万维网购买股票

前置条件：用户已经处于PAF打开状态。
触发事件：用户选择“购买股票”：
1. 用户打算通过万维网来购买股票。
2. PAF从用户那里得到所用站点的名称（如E*Trade、Schwab等）。
3. PAF打开到此站点的网络连接，并保持控制权。
4. 用户在该站点上浏览并购买股票。
5. PAF截取站点的响应信息，并修改用户的证券。
6. PAF向用户显示更新后的证券。

对用例的一个扩展：
3a. 在设置过程中，网络发生故障：
　3a1. 系统向用户报告错误，并建议他退回到前一步。
88　3a2. 用户或者退出此次操作，或者重新再试。

在本章中，我们详细讨论场景的主体，它是由执行步骤构成的。

7.1.2 场景主体

每个场景或片段被描述为由不同执行者完成目标的活动序列。我说的“*序列*”（*sequence*）是为了简便，实际上，可以使用注释来说明并行的活动、顺序不确定的活动、可重复的活动、甚至可选的活动。

就如在2.2节“具有利益的项目相关人员之间的契约”中的解释一样，下面的任何一步都要描述

- 两个执行者之间的交互（“顾客输入地址”）。
- 为了保护项目相关人员利益的确认过程（“系统确认PIN密码”）。
- 满足项目相关人员利益的内部变化（“系统从余额中扣除一定数量的金额”）。

有一个典型的主成功场景的例子，来自用例22“*损失登记*”。注意步骤1、步骤3、步骤5到8是交互，步骤4是一个确认，步骤2和步骤9是内部更改。

1) 职员输入被保险人的保险单号或者事故的名称和日期。
2) 系统自动填写保险单信息，以表明申请与保险单相一致。
3) 职员输入基本的损失信息。
4) 系统验证不存在其他的竞争申请，然后为该申请分配一个申请号码。
5) 职员继续输入与申请种类相关的具体的损失信息。
6) 职员让系统从其他计算机系统中提取其他赔偿金信息。
7) 职员选择并指定一个调节者。

8) 职员确认他/她的操作结束。

9) 系统保存和触发发送给代理的认可信息。

每一个执行步骤显示了一个简单、主动的活动。我喜欢用它来描述一场足球比赛：球员1将球传给球员2；球员2运球；球员2将球传给球员3。

一旦你掌握了以上三种执行步骤的编写技巧，你就可以很好地调整你的编写风格。可以在任何用例中的每一部分，无论是在主成功场景、扩展场景、业务场景或是系统场景中，无论是在高层用例或是低层的用例中，都使用相同的风格来编写执行步骤。 89

7.2 执行步骤

执行步骤是对用例的补充，并且都有统一的语法形式，在一个简单活动中，执行者完成任务或向另外的执行者发送信息。

用户输入名称和地址。

在任何时候，用户可以要求退款。

系统确认名称和账号都存在。

系统根据费用更新用户的余额。

由于步骤通常一个紧接着下一个，因此时间限制常被忽略。

就如同你已经看到的用例，在写执行步骤时，经常会有很多小的不同。然而一旦你选择了一种编写方式，就要在每个步骤中都坚持使用相同的风格。

7.2.1 准则

准则1：使用简单的语法

句子的结构应该非常简单：

主语……谓语动词……直接宾语……前置短语。

例如：

系统……从账户余额中扣除……一定数量……。

这就足够了。我提出这个准则是因为很多人有时遗漏了主语，这样人们就不知道谁是活动的执行者（谁在带球）。假如句子没有好的结构，所描述的情节就会变得难以理解。

准则2：明确地写出“谁控制球”

踢足球的场景是一个很好的例子。有时候，球员1将球传给球员2；球员2运球，然后将球传给球员3。有时候，球上粘了一些泥，一个球员就把球擦干净。

在同一个场景中使用相同的结构。在每一个步骤中，一个执行者“控制球”。这个执行者就是句子的主语——第一个执行者，它或许是句子中的第一个词或者是第二个词。“足球”是执行

者与执行者之间传递的信息和数据。

控制球的执行者会做以下三件事之一：自己踢球，将球传给别人，或者将球上的泥擦掉。
90 大多数情况下，在步骤结束的时候，另外一个执行者控制足球。问问自己，“在句子结束的时候，谁控制足球？”不论什么时候，这个问题都应该有明确的答案。

准则3：从俯视的角度来编写用例

初级的用例编写者（特别是那些正在编写用例文档的程序员）在描述场景的时候，经常从系统内部来看待外部的世界，而且总是从系统内部来描述系统。他们可能写出这样的句子：“读取ATM卡和PIN号码，并从账号余额中扣除一定数量。”

而从俯视的角度来编写用例，则是：

用户插入ATM卡并输入PIN。
系统从账号余额中扣除一定数量。

一些用例编写者喜欢使用另外一种风格，这种风格有一定的优点，它描述了执行者和他们的职责：

用户：插入ATM卡并输入PIN。
系统：从账号余额中扣除一定数量。

注意：在这两种风格中，表达的信息是一致的。

准则4：显示过程向前推移

每一个步骤完成的进度是与用例目标的高或低相关。在一个概要或白色用例中，一个步骤可能完整地完成一个用户目标。在子功能用例中，一个步骤可能只完成少部分的工作。如果我们看到像“用户按下Tab键”这样的步骤，那我们看到的可能是一个靛青色（或黑色）用例，或者是用例编写者选择了一个过小的活动来描述。

选择过小的步骤将导致用例文档变得很长。假如一个用例有13到17个步骤，这说明每一个步骤完成的工作过少。如果将一些小的步骤合并，可以使用例的可读性更好，更加清晰，而且保持了原有的基本信息。在我所见到的好用例中，很少见到在主成功场景中有多于9个步骤。

为了找到具有较高层次目标的步骤，可以提出这样的问题“为什么执行者要做这件事情？”（如5.5.2节“提高和降低目标层次”中所述）。可能需要多次地问这个问题才能得到满意的答案，而这个问题的答案可能就是步骤实现的目标。以下是一个通过问为什么来提高目标层次的例子：

用户按下Tab键

91 为什么用户要按下Tab键呢？是为了将焦点移到地址框中。

为什么她要将焦点移到地址框中呢？因为在系统开始工作前，她要输入用户名和地址。

哦！她想让系统完成一些工作（这可能是用例本身），为了使系统工作，她必须输入她的用户名和地址。

因此，一个清楚地描述过程向前移动的主动语态句子是：

用户输入用户名和地址。

准则5：显示执行者的意图而不是动作

通过操纵系统的用户界面来描述用户的动作，这是在编写用例时常见的一种严重错误，它使得编写的目标处于一个很低的层次。我把它叫做“界面细节描述”（*interface detail description*）。界面细节描述使得需求文档质量变差，主要表现在以下三个方面：文档变长、脆弱、过分的限制。

- 文档长度增加使得阅读更困难，维护的费用更高。
- 人机交互界面的描写可能并不是实际需求，而仅仅是作者在某个时刻想像中的用户界面。
- 人机交互界面在某种意义下是脆弱的，因为在系统设计的时候，很小的变化就使得用例文档变得无效。

由用户界面设计师来完成用户界面的设计是一个有效的方法，这样就允许在用例中只描述用户执行的意图。对具体动作的描述属于设计的任务，而不应该出现在功能需求文档中。

在需求文档中，我们只关心界面所要达到的意图，即它表达用户的意图，总结在执行者之间传递的信息。在Larry Constantine和Lucy Lockwood的“*Software for Use*”一书中，有一部分内容是讨论这个主题的。他们使用“基本用例”（*essential use case*）这个名词来命名海平面系统用例（sea-level system use case），该用例描述了界面所要达到的意图。

通常情况下，当一些连续步骤的数据都在同一方向上运动时，就可以将这些步骤合并成一个步骤。下面是一个有缺陷的例子，以及对它的修改。

修改前：

1) 系统要求用户输入名字。
2) 用户输入名字。
3) 系统要求输入地址。
4) 用户输入地址。
5) 用户点击“确定”。
6) 系统显示用户的简介。

92

修改后：

1) 用户输入名字和地址。
2) 系统显示用户的简介。

如果传递三个以上数据元素，你可能更喜欢将每个数据元素写在表格中的单独一行。下面两种方式都可以很好地表达问题。由于第一种方式更容易编写和阅读，因此在你第一次设计用例时，采用这种方式能够工作得更好。如果你想提高可描述性和可测试性，使用第二种方式会更好一些。

可接受形式1：

1) 用户输入名字、地址、电话号码、保密信息、紧急联系电话。

可接受形式2：

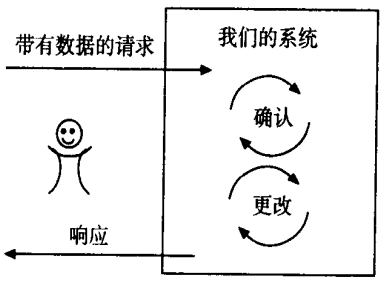
1) 用户输入
—名字
—地址

—电话号码
—保密信息
—紧急联系电话

准则6：包含“合理”的活动集

Ivar Jacobson在用例中描述一个步骤是为了表现一个事务（*transaction*）。在下面的句子中，他捕获了四个相互作用的部分（见图7-1）：

1) 主执行者向系统发送请求和数据。
2) 系统验证请求和数据。
3) 系统更改内部状态。
4) 系统向执行者返回结果。



93

图7-1　一个事务由四个部分组成

你可以把每个部分作为一个单独的执行步骤，也可以用不同的方式合并其中的几个部分，还可以将四个部分合并成一个执行步骤。哪种方式最好，是由每个部分的复杂程度和执行过程中在什么地方会发生自然的中断来决定的。

有五种形式供你考虑。这五种形式都是正确的，尽管我认为形式1比较复杂，不易于阅读。当每部分都很简单时，我喜欢形式2，但我发现在这个例子中，这些部分的确有些太长。形式3是我最喜欢的，形式4也不错。我认为在形式5中，执行步骤太小，使得场景太长，也不容易控制。但形式5确实有它自己的优点，即每个步骤都分别是可测的单元，这种形式可能更适合正式的开发过程。

形式1：

1) 用户输入订购号码。系统发现这个号码与本月的中奖号码匹配，将用户和订购号码注册为当月的中奖者，发电子邮件给销售经理，祝贺客户，并告诉他们如何领取奖金。

形式2：

1) 用户输入订购号码。
2) 系统发现这个号码与本月的中奖号码匹配，将用户和订购号码注册为当月的中奖者，发电子邮件给销售经理，祝贺客户，并告诉他们如何领取奖金。

形式3：

1) 用户输入订购号码。

2) 系统发现这个号码与本月的中奖号码匹配。

3) 系统将用户和订购号码注册为当月的中奖者，发电子邮件给销售经理，祝贺客户，并告诉他们如何领取奖金。

形式4：

1) 用户输入订购号码。

2) 系统发现这个号码与本月的中奖号码匹配。

3) 系统将用户和订购号码注册为当月的中奖者，发电子邮件给销售经理。

4) 系统祝贺客户，并告诉他们如何领取奖金。

形式5：

1) 用户输入订购号码。

2) 系统发现这个号码与本月的中奖号码匹配。

3) 系统将用户和订购号码注册为当月的中奖者。

4) 系统发电子邮件给销售经理。

5) 系统祝贺客户，并告诉他们如何领取奖金。

94

准则7："确认"而不是"检查是否"

三种类型的执行步骤之一是系统验证一些业务规则是否满足。通常，人们说系统检查某个条件。在活动的描述中，用"检查"这个动词并不好。这样说并没有让过程明显地向前发展，它并不是真正的目的，它使得检查的结果是不确定的。你不得不这样写："如果检查通过了……"，然后写："如果检查没有通过……"。

我们使用"问为什么"的技术来找到一个更好的短语。为什么系统要检查条件？答案：是为了确认、验证或确保某些事情。这些都是很好的可以表示目标的动词。我们将"系统检查密码是否正确"替换为：

系统验证了密码正确。

"如果"这个词的出现就会让你回想起这个准则。在任何时候，当你看见"如果（条件）……就……"这样的句子，看一看这个句子的前一句，可能会发现"检查"一词。用"验证"来代替第一个句子中的"检查"，再将第二个句子变为没有"如果"的简单句。下面举一个例子：

修改前：

2) 系统检查密码是否正确。

3) 如果密码正确，系统向用户提供有效的操作。

修改后：

2) 系统确认密码正确。

3) 系统向用户提供有效的操作。

注意，在修改后，用例描述了一个成功的场景。这就会让读者问下一个问题，“如果密码无效时，应该怎么办？”读者将转到扩展部分，寻找“密码无效”的扩展。这就让用例保持了一个一致的风格，使得用例更加容易阅读和评审。

准则8：可选择地提及时间限制

大多数步骤都直接跟着上一步。有时，你也可能会这样描述事情

95　在步骤3和步骤5之间的任何时候，用户将……

或者

当用户……，系统将……

确实需要在用例中插入时间限制的时候，你可以放心地这样做。通常情况下，时间限制是很明显的，并不需要显式地提出。

准则9：习惯用语：“用户让系统A与系统B交互”

你可能会遇到下面这种情况。正在设计的系统A要从系统B中获取信息、或者与系统B有一个交互过程。只有当主执行者指明时间是恰当的，才应该这样做。我们不能这样写：“用户点击获取按钮，这个时候系统从系统B获取数据。”应该详细地描述交互的细节。

我们可以使用两个步骤：

4) 用户通知系统从系统B获取数据。
5) 系统从系统B获取后台数据。

然而，虽然上述写法可以被接受，但它们显得笨拙和冗余。更好的写法如下：

4) 用户命令系统从系统B获取基本数据。

通过编写方式的转变，我们指明是由用户控制执行的时间，并说明了“球”是从用户到系统A，再到系统B。上面的写法还指明了三个系统的职责。用户怎样初始化活动的细节是不明确的，而他们就应该是那样。

准则10：习惯用语：“循环执行步骤x到y，直到条件满足”

有时，我们想把那些能够重复的步骤做上标记。幸运的是，我们只是写没有严格格式要求的文章，而不是写形式化的程序。我们只需要写出哪一步或哪几步需要重复执行。

如果只有一个步骤需要重复，可以将表达重复执行的短语放在步骤描述中：

用户选择了一个或多个产品。
用户在各种各样的产品目录中寻找，直到他发现所需要的产品。

如果多个步骤需要重复，可以把表达重复执行的语句放在这些步骤的前面或者后面。为了让场景更容易阅读，我把表达重复执行的语句放在了这些步骤的后面。其实，每一种方法都是

有效的。

1) 顾客提供账号或者名字和地址。
2) 系统查出顾客的爱好信息。
3) 用户选择一个商品，并做上购买的标记。
4) 系统将这个商品加入顾客的“购物车”中。
   顾客重复步骤3～4，直到顾客指明他/她完成了选购。
5) 顾客购买所有在购物车中的商品。

96

注意在表达重复的语句前面我们不需要加上序号，也不需要用任何句子来说明这是一个表达重复的语句。这两种写法都会使描述显得混乱，使得场景难以理解。

除了“循环执行步骤x到y，直到条件满足”这种形式，另外一种形式是“步骤x到y可以循环执行”。我发现把它放在重复的步骤前面，也是一种很好的方式。

1) 顾客登录。
2) 系统显示可用的产品和服务。
   步骤3～5可以重复执行。
3) 用户选择要购买的产品。
4) 用户确定付款方式。
5) 用户输入目的地址。
6) 用户指明购物行为结束。
7) 系统初始化订单，包含了选购的产品信息、付款方式和目的地址的信息。

7.2.2 编号或不编号

步骤编号使步骤更清晰，并在扩展部分给出引用的位置。然而，编号的维护是困难的。当我们插入或删除一些步骤的时候，如果没有好的工具来完成自动编号，重新对步骤编号是一项乏味的工作，特别是对相同的用例。写一个好的用例，步骤可以有编号也可以没有，因此它只涉及到个人的喜好问题，或者只涉及到某个特定项目范围内的规范。

我调查过一些小组，他们曾经尝试过两种编写的方法，最终他们选择了对段与段之间的句子进行编号的方式。因此，编号方式也成为我标准的编写方式。也有另外一些人喜欢在简单的段落结构中使用简单的编写方式，因为他们并不需要关心转移的问题。

为了强调两种方式的等价性和选择的个人性，我提供了非正式用例和完整正式用例两种模板。

在非正式用例的模板和Rational统一过程的模板中，段落都是缺省的提示（用在第14章的用例32“管理报告用例”中）。如果你需要，就可以对步骤编号，就像例子中的一样。我通常都使用编号，甚至是当与用例相关的其他事情都是非正式的时候也是如此，因为我发现这样做使得检测行为更加容易。

97

完整正式的用例格式需要编号。

7.3 练习

执行步骤

7-1 选择一个用例，采用两种方式（即界面细节描述和意图描述）来写出用例中的一个场景。并讨论两者之间的不同。

7-2 写出任务级别用例“使用快速现金选项退回货币”的主成功场景。

7-3 在PAF系统中，分别写出一个策略级别用例的主成功场景和一个任务级别用例的主成功场景。

7-4 你的下级送给你下面的文档。用迄今为止你所学的知识，写出评论和修改意见。

用例：登录

这个用例描述用户登录到订购系统的过程。它也对不同类别的用户设置访问权限。

事件流：

基本路径：

1. 当用户开始应用程序，用例开始。
2. 系统显示登录界面。
3. 用户输入用户名和密码。
4. 系统验证登录信息。
5. 系统设置访问权限。
6. 系统显示主界面。
7. 用户将选择一个功能。
8. 当用户没有选择退出，就循环。
9. 如果用户选择下订单，完成下订单。
10. 如果用户选择返回产品，完成返回产品。
11. 如果用户选择取消订单，完成取消订单。
12. 如果用户选择显示订单状态，完成显示订单状态。
13. 如果用户选择发送目录，完成发送目录。
14. 如果用户选择登记不满意见，完成登记不满意见。
15. 如果用户选择运行销售报告，完成运行销售报告。

条件语句结束（end if）

16. 用户将选择一个功能。

循环结束（end loop）

17. 用例结束。

第8章 扩　展

我们知道一个用例应该包含所有成功的和失败的场景，并且我们已经了解如何编写主成功场景。现在，需要一种方法来加上所有其他场景。

我们可以单独地写出每一个场景。如图2-2中所示条纹裤的比喻很适合这种方法。有时，人们还在提倡这种方法，一些工具也强制编写者以这种方式来编写场景。然而，根据第2章的描述，该方法的维护是一个噩梦。场景的任何一个变化都导致了在其他包含相同文字的场景里都必须做一份拷贝。

另一种可以选择的方法是在整个文本中使用条件语句："如果密码正确，系统……，否则系统……"。这样相当合理，有一些人这样编写用例。然而，读者要阅读这些条件语句会很困难，特别是当一个条件句中又嵌套了一个条件句。读者往往在两个条件分支后，就失去了用例行为的线索，而大多数的用例包含了很多分支点。

第三种方法是将主成功场景从开始到结束，按照时间的顺序写出来，然后在每个分支点写出场景的扩展。这种方法可能是三种方法中最好的一种。

8.1 扩展的基础

扩展通常是这样的，在主成功场景下，对于因为特别条件而出现行为分支的每个地方，写出分支的条件以及处理分支的步骤。大多数扩展以重新与主成功场景汇合而结束。

扩展实质上是一个从主用例中被拆分的用例。扩展开始于一个与它相关的条件。它包含了一个执行步骤的序列，该序列描述了在这个条件下发生了什么。扩展以完成或放弃扩展目标作为结束。使用这种方法，为了处理多个条件（*if*）和转移（*but*），可能会遇到扩展中又包含扩展的情况。

这些扩展最可能出现在系统需求中。开发组通常对主成功场景很了解。而错误处理通常使用并不为开发人员所了解的业务规则。需求分析人员必须研究正确的系统响应，而这样的研究经常会引入新执行者、新用例或新扩展条件。

这里有一个相当典型的讨论来证明我的观点。

"假定有一个突发的网络错误。我们应该做什么？"

"系统记录这个错误。"

"分析错误。嗨，如果网络性能下降，当这个错误再次发生，将会发生什么情况？"

"噢，我想我们必须增加一个'系统在网络错误后重启'的用例。系统将恢复，取得日志，完成或放弃这个事务。"

"是的，但是如果日志被破坏，将发生什么？系统打算怎么处理？"

"我不知道。就让它作为一个未解决的问题，我们继续。"

上面的对话发现了一个新用例和一个需要研究的业务处理策略。不要被假象所欺骗，认为讨论这些扩展是没有必要的。如果项目组的程序员在编程时遇到这些问题，我们才去寻找新的业务规则，并去研究它，这是很浪费时间的。需求阶段是收集这些规则的最好时机。

我建议将工作分为三个阶段：

1) 集中讨论，发现你和同事能够想到的每种可能。

2) 根据8.2节中的准则，评估、删除以及合并所有建议。

3) 详细讨论，并找到处理每一种情况的方法。

8.2 扩展条件

扩展条件（*extension condition*）就是在一些条件下系统会完成不同的动作。由于使用了*扩展*（*extension*）这个词，而不是*失败*（*failure*）或*异常*（*exception*），因此我们能够包括成功和失败两种条件。

100 以下是两个片断分别来说明扩展中成功和失败的路径。

示例1：

……

4. 用户要求系统保存迄今为止的工作。

……

扩展：

4a.系统自动检测中间保存的要求：

4a1……

4b.保存失败：

4b1……

可以用这样的方式来阅读上面的例子，“除了用户要求系统保存迄今为止的工作外，系统可能会自动检测中间保存的要求。在这种情况下……（做一些事情）。（来自于用户请求或来自于自动保存的）保存在执行过程中可能失败。在这种情况下……（做另外一些事情）。”

示例2：

……

3. 系统浏览文档，根据拼写字典检查每一个单词。

4. 系统检测到拼写错误，加亮单词，向用户显示可选单词，供用户选择。

5. 用户选择一个单词来代替错误的单词。系统根据用户选择的单词来代替加亮的单词。

……

扩展：

4a. 系统检测整个文档没有拼写错误：

4a1. 系统通知用户，终止用例。

5a. 用户保持原有的拼写：

5a1. 系统忽略那个单词并继续。

5b. 用户输入一个不在列表中的新单词：

5b1. 系统重新验证新单词的拼写，返回步骤3。

8.2.1 集中讨论所有可能的失败和可选择的过程

集中讨论是很重要的，它可以在编写怎样处理扩展条件以前收集到大部分可能出现的扩展条件。集中讨论很容易引起疲劳，特别是在讨论中要记录如何处理扩展。想想看，仅仅解决一 101
个扩展就花费了大量精力，那么要解决三到四个扩展时，你就会精疲力尽。这就意味着你不可能继续考虑其他扩展条件，即使它们也是应该考虑的。

另一方面，如果集中讨论所有可能发生的成功和失败情况，那么你将得到一个列表。这个列表就像一个脚手架一样，指导你接下来几个小时或几天的工作。你可以在午餐时间或通宵按照列表来工作，也可以回来从你停止的地方重新开始。

集中讨论所有可能的情况，这些情况中的场景可能失败，或通过其他方式获得成功。仔细考虑下面所有的条目：

- 一种可选择的成功路径（职员使用便捷的代码）。
- 主执行者操作错误（无效密码）。
- 主执行者无任何操作（等待密码超时）。
- “系统确认”这个短语的每一次出现，都暗示了将会有一个处理系统失败的扩展（无效的账号）。
- 从辅助执行者那里得到不恰当的响应或没有响应（响应超时）。
- 作为正常功能的一部分，检测所设计系统的内部错误，并处理（现金分配阻塞）。
- 必须处理不期望和异常的内部错误，并产生一个外部可见的结果（发现崩溃的事务日志）。
- 必须检测系统关键性能的失败（回答没有在5秒内完成）。

许多人集中讨论从场景开始到结束的步骤，以保证能够尽可能覆盖所有情况。这样做的时候，你会很吃惊地发现，你所考虑的可能出现错误的情况有很多。在练习8-1中，你有一个机会，可以尽可能地找出失败的情况。答案在附录B中，可以检查一下你是否找全了。在我的课程中，有两个学生，John Colaizzi和Allen Maxwell，重复了三次才完全找出了我在该题答案中写出的失败情况。你做得怎么样？

准则11：用“检测到什么”的方式来编写条件

应该写出系统检测什么，而不仅仅写出发生了什么。不要写，顾客忘记了密码。系统不可能检测到顾客忘记了密码。可能是顾客离开了，或者心脏病发作，或者正忙于安抚一个哭叫的小孩。在这种情况下，系统能检测到什么呢？在用户没有任何动作的情况下，这意味着系统只能检测到等待时间超过了限制。应该写，等待用户输入密码的时间超时或密码输入的时间超时。 102

扩展条件经常是一个描述系统检测到了什么的短语。有时也可以用一个句子。我喜欢在条件后面加一个冒号（:），使读者不会误认为它是一个执行步骤。这个小小的约定减少了很多错误。这里有一些例子：

无效密码：

网络崩溃：

顾客没有响应（超时）：

现金不适当地投放：

如果你使用编号的步骤，那么在条件被检测出的地方也要写出步骤的编号，并在编号后面加上一个字母（例如4a）。顺序和字母是无关的，没有规定说4b一定要在4a后面。这就允许我们自己决定在一个执行步骤后附加多少扩展条件。

2a. 资金不足：

2b. 网络崩溃：

如果条件发生在多个步骤中，而且你认为指明条件在哪几个步骤中发生是很重要的，那么可以简单地将步骤列出：

2-5a.用户突然退出：

如果条件多次出现，可以使用星号（*）来代替步骤编号。在编号的条件之前列出带星号的条件。

*a.网络崩溃：

*b.用户没有结束操作就离开了（超时）：

2a.资金不足：

2b.网络崩溃：

错误是发生在用户输入数据时，还是之后，或是发生在系统验证数据的时候呢？不要被这样的问题所困扰。一个人可能认为错误发生在上述两个步骤中，但这样的争论是不值得的。我通常把这种情况放到系统验证步骤中。

如果在编写时没有加上步骤编号，你就没有办法明确地指出条件发生的步骤。因此，要在条件描述中提供足够的信息使读者知道什么时候条件可能出现。在每一个条件前，空一行或空一格，并在条件上加一些标记，比如：使用楷体字，使条件更突出。参见第14章中的用例32，“管理报表”。

103

◆ 一个真实的、令人不快的小故事

在我参加的一个重大项目中，发生了一个与好的集中讨论扩展条件相反的事情。

就像很多开发人员一样，我们根本没有去考虑这样一种情况，万一程序所访问的数据库中的数据是错误的，将会发生什么。我们每一个小组都寄希望于别的小组将处理这个问题。你能猜到结果怎么样吗？在第一次交付软件的一个星期后，一个高级副总裁想要看看他所关心的客户使用新销售设备的情况。他打开新系统，查询这个大客户。系统告诉他，“没有发现数据。”可以用一个词来描述他当时的反应，“激动”（但不是因为高兴而激动）。

很快，所有高级职员就集中开了一个紧急会议，讨论如何处理数据库错误。我们发现在数据库中只要有一个错误的数据单元，系统就会报告“一些数据丢失”。更重要的是，我

们没有意识到一个问题，“在检测到错误的内部数据时系统怎样处理”。这样的问题实际上是外部需求的一部分。

我们重新设计系统，使它可以尽量处理不完整的数据，并且使它既能传递有效结果，也能传递数据丢失的消息。

这个教训教育我们要考虑内部错误，比如发现丢失的数据。

关于集中讨论列表

集中讨论列表中包含的内容将多于你最后使用的内容，这种现象是正常的。讨论的目的是尽量找出系统将在这个用例中遇到的所有情况。然后，你会减少列表的内容。

从另一方面来说，你的列表也可能不完全。你可能在编写扩展场景时，或在用例内部增加新的验证步骤时，又想到了新的失败条件。不要担心，虽然需要尽量在集中讨论的阶段完善列表，但也可以在任何时候增加列表的内容。

8.2.2 扩展列表的合理化

合理化是为了使扩展列表尽可能的短。理想的扩展条件列表是所有的而且仅是必须由系统处理的情况。回想一下，我们阅读一个长的需求文档是很困难的，并且维护那些冗余的描述也是很困难的。通过合并扩展条件，可以缩短你的文档，并相应减少了读者的阅读量。

在集中讨论之后，应该产生一个短小和简单的主成功场景以及一个需要考虑的、长长的条件列表。仔细考虑列表，清除那些系统不需要处理的条件，合并那些对系统来说等价的条件。请使用下面两个标准：

104

- 系统必须能够检测到条件。
- 系统必须完成条件的检测。

在删除那些不可检测的条件前，尝试一下它们是否可以变为可测的条件。删除“用户忘记插入ATM卡”等条件，因为系统不能检测到这个条件。可以将不可测的条件“用户忘记密码”改写为系统可测的条件“密码输入超时”。

下一步，合并等价条件。你可能会这样写：“卡被划坏了；读卡器出现故障；不是ATM卡”。从需求角度来看，ATM机对上述三种情况的处理是相同的：“退出卡并通知用户”。因此，可以试着将这些条件合并。如果你不能找到一个简单明确的短语来表达所有这些条件，可以使用复合的条件：“卡不可读或不是ATM卡”。

8.2.3 逐层合并失败

作为合并等价条件的一部分，从低层用例中提取等价的失败情况，合并到高层用例中。低层失败的合并（*rolling up*）是我们在高层上避免扩展条件爆炸的一种方法。

考虑一下，例如，你正在我们的PAF包上工作，编写了一个“更新投资”的用户目标用例。假定最后的步骤中有一步是：

用例：更新投资

……

7. 用户使用PAF保存工作。

8. ……

这个例子中调用了一个“保存工作”用例，该用例包含了以下的条件：

用例：保存工作

……

扩展：

3a. 文件已经存在（用户不想覆盖）：……

3b. 没有找到目录：……

4a. 磁盘空间不够：……

4b. 文件写保护：……

105 ……等等……

“保存工作”用例以成功或失败结束，然后回到“更新投资”用例步骤7的末尾。成功的时候，继续执行步骤8。但是，如果失败了，会出现什么情况？我们应该怎样写扩展7a？“更新投资”用例的读者并不在意为什么保存失败——所有的失败都是等价的——因此在“更新投资”用例中，仅仅写出一个扩展来描述发生了什么。

用例：更新投资

……

7. 用户使用PAF保存工作。

8. ……

扩展：

7a. 保存失败：

7a1. ……不管发生了什么，接下来……

合并这个失败的最佳之处是，甚至在最高层的用例中，也只用了一个词就恰当地为该层报告了失败的情况。由于高层用例中的失败报告也是高层的，因此繁忙的主管就可以抽出时间来阅读它们。

8.3 扩展处理

在最简单的情况下，一个基本步骤序列处理一种情况。通常，扩展是一个小型的用例。触

发事件是扩展条件；目标是完成用例目标或者覆盖所有可能出现的失败。用例主体是执行步骤集和这些步骤可能的扩展。在用例中，扩展以完成或放弃它的目标作为结束。用例与扩展的这种相似不是偶然的，也证明了可以非常方便地以流水方式记录一个复杂的扩展。

应该从检测条件之后的执行步骤开始编写扩展条件的处理。不必重复检测扩展条件。应该继续编写你的场景，就像编写主成功场景时一样。遵守所有前面讨论过的有关目标级别、动词格式和句型的准则。你可以一直写到步骤重新回到主成功场景或用例失败为止。 106

典型的场景片段以下面四种方式之一结束：

修改和替换进入扩展中的步骤。扩展处理结束时，就像完成了原来步骤一样。

3. 用户激活网站的URL地址。
4. ……

扩展：

3a. 没有可用的URL地址：

3a1. 用户搜索网站。

3b. ……

系统给执行者另一个选择。扩展处理结束时，故事回到被扩展步骤的开始。注意在下面例子中，系统将重新验证口令：

3. 用户输入口令。
4. 系统验证口令。
5. ……

扩展：

4a. 无效口令：

4a1. 系统通知用户，重新输入口令。

4a2. 用户重新输入口令。

4b. ……

由于彻底失败而结束用例。

3. 用户输入口令。
4. 系统验证口令。
5. ……

扩展：

……

4c. 输入无效口令的次数超过限制：

4c1. 系统通知用户，结束用户会话。

5a. ……

扩展行为是一条完全不同的成功路径。

3. 用户……

4. 用户……

5.

扩展：

3a. 用户运行个人宏来完成处理过程：

107　　3a1. 用例结束。

在前两种情况下，不需要说明扩展以后发生了什么，因为答案对读者来说很明显，被扩展的步骤将重新开始或继续执行。在后两种情况下，通常只需要说明用例失败或用例结束，因为这些扩展步骤表明了系统在一定程度上满足了项目相关人员的利益。

大多数扩展没有说明扩展以后的情景应该怎样。通常，这是很明显的，并且在每个扩展后面写上“转到步骤N”会使得整个文档难以阅读。偶尔，也会出现不明显的情况，当故事可能会跳到主成功场景的其他部分时，就需要在扩展的最后一步写上“转到步骤N”。

在本书的不同编写样例中，可以找到所有不同类型的例子。

准则12：条件处理的缩排方式

当我们使用编号方式时，应该缩排那些指明如何处理条件的执行步骤，并在字母后重新从编号1开始。执行步骤遵循前面给出的准则。

扩展：

2a. 资金不足：

2a1. 系统通知顾客，要求输入一个新的数量。

2a2. 顾客输入新的数量。

仅仅在本书出版之前，Software Futures CCH 的Volker Schaberg描述了一个简化的编号格式：在这种格式中，直接以一个点和一个数字开始，省略了前面的数字和字母。上面的例子修改后如下所示：

扩展：

2a. 资金不足：

.1 系统通知顾客，要求输入一个新的数量。

.2 顾客输入新的数量。

这种格式的优点是当编号变化时，例如步骤2变为步骤3，对扩展处理步骤的重新编号变得相当简单。

当使用整齐的散文（无编号）格式时，也应该对执行步骤缩排，或者分段。Rational统一过
108　程模板对扩展条件有一个专门的命题标准，在标题下面写出执行步骤。

8.3.1　失败的嵌套

在扩展处理场景的片段中，你可能面对一个新的分支条件，而且可能是一个失败条件。如

果你使用了本书的缩排编写方式，只需要同前面的扩展一样，简单地再次缩排，继续命名条件，继续编写场景。

有时，缩排和编号会变得很复杂，以至于你决定将扩展分离出作为一个完整的用例。大多数曾经写信给我的人都赞成在有第三个缩排时这样做。

这里有一个例子，来自用例22，“遗失登记”。

6a. 职员在没有完成最小的必要信息之前决定退出：

6a1. 系统警告职员在没有输入日期、姓名或保险号码、或调节人的姓名情况下，不能退出。

6a1a. 职员选择继续输入遗失信息。

6a1b. 职员将输入的信息保存为“临时”报告，退出。

6a1c. 职员在没有输入最小的必要信息之前，坚持退出：

系统丢弃所有保存的临时信息，退出。

在这个例子中，注意编写者没有在最后一行加编号。如果将最后一行编号为6a1c1，编写者会觉得这使扩展很混乱，不如直接写一段文字可读性更强。

通常，创建新用例的代价是很高的，以至于人们尽可能不将一个扩展分离出作为一个用例。对于上面的例子，一致的意见应该是在分离出一个用例之前，尽量使用缩排方式。

8.3.2 从扩展中创建新用例

将扩展分离为新的子用例，仅仅需要确定主执行者的目标是什么，给出新用例的级别（可能是子功能级），在新级别中为新用例命名，为新用例创建模板，并填写新用例的细节。

第14章中的用例32“管理报告用例”是一个例子。这个用例曾包含这样一个步骤：

用户能够在任何时候保存或打印报告。

这个步骤有一个扩展集，描述了不同的可选择情况和失败情况，但是这些情况持续增长：没有命名的报告，重名（是否覆盖），用户中途放弃保存，等等。最后，编写者决定创建保存报告用例。 109

在原来的用例中，仍然必须处理新的子用例可能失败的情况，因此你的文档可能需要指明成功和失败的条件。

从理论上和实际花费的观点来看，将扩展移到它自己的用例中或恢复回来，都是简单的事情。用例模型允许我们将扩展看作是一个简单的判断。将保存报告扩展移出管理报告用例是没有问题的，而且在文本编辑器中，也只需要花几分钟时间就能将它们重新移回去。

然而，创建一个用例的花费并不这么简单。新用例必须被分类、跟踪、规划、测试和维护。对于项目组，这些工作的代价是很高的。

将扩展保持在用例中通常是更经济的方法。在以下两种情况下，你可以为扩展创建新用例：

- 扩展在多个地方使用。将扩展转变为用例意味着它可以在同一个地方被跟踪和维护。在理

想情况下，这是在海平面级别下创建用例的惟一原因。

- 扩展使得用例难以阅读。我发现两页左右的用例文档和三级缩排是可读性的极限。（我的用例文档要比大多数人的用例文档短，因此你的用例文档的长度可能会长一些。）

8.4 练习

扩展

8-1 集中讨论并列出在ATM系统操作中可能出现的错误。

8-2 集中讨论并列出在PAF扩展处理系统中，第一个用户目标用例中可能出现的错误（见练习4-4）。

8-3 写出“提取现金”用例，包括失败处理，使用条件语句。使用场景扩展重写用例。比较两者。

8-4 找一篇采用不同格式编写的需求文档，其中一个格式是用例格式。它是怎样捕获失败条件的？你喜欢哪一种格式？你从中学到了什么？

110 8-5 写出PAF系统完整的用例，填写失败处理步骤（PAF系统见练习4-4）。

第9章
技术和数据的变化

扩展说明了系统所完成的目标是不同的，但有时需要表达“有多种不同方法来完成相同目标”。系统所完成的目标是相同的，但怎样做可能不同。这通常是因为技术的变化或输入数据的不同。应该将这些变化写到“技术和数据变化”列表中，而不是写到扩展部分中。

示例1：

你的系统必须对退回货物的顾客归还货款。你写了这样一个执行步骤：

7. 归还退货顾客的货款。

退款可能通过支票或电子资金转账来支付，或在下一次购买时替顾客支付。因此，你可以加入：

技术和数据变化列表：

7a. 通过支票、电子资金转账来支付，或在下一次购买时替顾客支付。

示例2：

你正在定义一个新的ATM系统。技术已经进步到可以通过银行卡、眼扫描或指纹来识别顾客。你写到：

主成功场景：

……

2. 用户标识自己、银行、账号。

……

技术和数据变化列表：

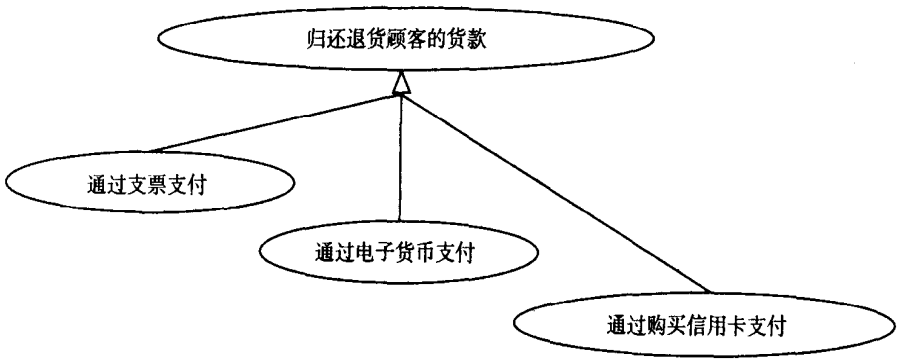
2a. 用银行卡、眼扫描或指纹来识别。

这些变化对这个用例来说不是扩展。在一些低层的用例中，每个变化都有自己的扩展，你可能从来不写它们。每个变化对你的成本和工作计划都有明显的影响，因此需要捕获和跟踪它们。你可以在技术和数据变化列表中写出可能的情况。

技术和数据变化列表不包含步骤。如果你在列表中包含了条件和执行步骤，你就没有正确地使用这个列表。

技术和数据变化列表的例子在用例13“资源的串行存取”中。

如果决定使用UML用例图，那么你可以为一个基本步骤创建一个空的、一般性的基用例，为每个变化创建一个具体的用例。空的基用例说明要做什么，而不是怎么做。每个具体用例定义自己的步骤来解释怎样做。UML的符号在附录A中解释。图9-1是一个示例。



112

图9-1　在UML中使用具体化方式表现技术变化

第10章
连接用例

10.1 子用例

一个执行步骤可以是一个简单的步骤或者另外一个用例的名称。例如：

用户保存报告

上面的步骤没有强调和注释，说明这个步骤是一个简单的原子步骤。如果像下面的例子来写执行步骤：

用户保存报告
用户**保存报告**（用例35 保存报告）

上述的写法指明了有一个用例叫做*保存报告*。这种风格非常自然，编写时没有必要多做解释。一旦告诉读者有下划线（来自于超链接）或*楷体字*的活动表示调用了另外一个用例，该用例扩展了活动的描述，即使是一个非正式的用例阅读者也能理解。[⊖] 这种编写用例的方式相当受欢迎，可以按需要合并或展开活动。而且，这也是连接用例的最简单方式。这种方式也不用多花时间来学习，也不占空间来描述。

10.2 扩展用例

有时，可能在两个用例之间需要另外一种连接，这种连接很像*扩展*机制。考虑这样一个例子：

> 你正在设计一个新的文字处理程序，叫做Wapp。主要的用户活动是键盘输入。然而，用户可能突然决定改变缩放尺寸或字体，运行拼写检查程序，或者进行许多与键盘输入无关的操作。实际上，你希望键盘输入活动与发生在文档中的任何其他动作都无关。
>
> 更重要的是，你希望不同的软件开发小组在对这些服务提出新建议的同时，不会因为新建议而导致基用例的修改。你希望能够在没有大变动的情况下扩展需求文档。

下面描述了上述情况的特征：

- 有一个主活动，主活动可以被中断。
- 主活动可以被多种方式中断，并且不能控制中断。

这和系统有一个主菜单的情况是有区别的，主菜单列出了用户可以选择的系统服务。主菜单控制着用户的选择，然而在我们的例子中，主活动不进行控制，而是被其他的活动中断。

⊖ 在UML的术语中，其他用例是被包含的关系。我发现初级编写者和非正式的读者更喜欢说一个用例引用或调用其他用例。不过，使用什么术语取决于你自己。

在这个实例中，不希望基用例显式地表示出所有的中断用例。如果每个人或小组在加入一个新的中断用例的时候都修改基用例，这样将会产生非常头疼的维护问题。每次的修改可能使用例变得混乱，或者每次修改都需要更新版本，重新评审，等等。

可以考虑使用与描述场景扩展相同的机制，但在这里是创建新用例。新用例称为扩展用例（*extension use case*），它除了是独立的用例之外，其他都与场景扩展相同。就像场景扩展一样，扩展用例从一个条件开始，在基用例中该条件可能满足的地方被引用。应将所有的条件都放到模板的触发事件（*trigger*）部分中。

为了解释清楚，我们从字处理程序*Wapp*的用例中摘录出一部分。图10-1显示了在UML图中*Wapp*的情况。我特别使用了一个特殊形式的连接符（一个钩子）来指明用例扩展（或者钩到）另外一个用例。详细说明参考附录A。

114

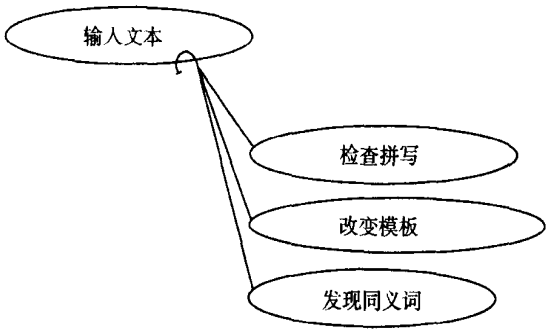


图10-1　扩展用例的UML图

用例：编辑文档

主执行者：用户
范围：Wapp
级别：用户目标
触发事件：用户打开应用程序。
前置条件：无
主成功场景：
1. 用户打开文档并编辑。
2. 用户输入或修改文本。
……
……用户保存文档并退出应用程序。

用例：检查拼写

主执行者：用户
范围：Wapp

级别：子功能！
前置条件：一个文档被打开。
触发事件：当文档被打开并且用户选择了打开拼写检查程序的情况下，在编辑文档中的任何时候。
主成功场景：
……等等……

115

用例：发现同义词

主执行者：用户
范围：Wapp
级别：子功能！
前置条件：一个文档被打开。
触发事件：当光标处于一个词的上面并且用户选择了打开分类词汇词典程序的情况下，在编辑文档的任何时候。
主成功场景：
……等等……

用例：改变文档模板

主执行者：用户
范围：Wapp
级别：子功能！
前置条件：一个文档被打开。
触发事件：当文档被打开且用户选择了该功能的情况下，在编辑文档的任何时候。
主成功场景：
……等等……

10.2.1 什么时候使用扩展用例

只有在必须的情况下才创建扩展用例，因为这些用例不容易被人们所理解，也不容易维护。在下面两种情况下可以使用扩展用例。

最常见的一种情况是当有很多的用户可能使用的异步服务或中断服务的时候，并且这些服务不应该影响基用例。通常，这些服务将由不同的小组来开发，并且这些服务一般与压缩打包软件（比如字处理软件）一起出现。

另外一种情况是为一个已经锁定的需求文档编写附加文档的时候。就像在S.R.A.的Susan Lilly写给我的信中提到的情况：

在你参加的项目中，开发过程是一个迭代的过程，并且有多个结束点。你有一个基准需求作为一个结束点。在一个后来的结束点上，你扩展了一个基准用例，增加了新功能或附加功能。但你不能影响基准用例。

在基用例没有锁定的情况下，扩展是脆弱的。改变基用例就会破坏在扩展用例中涉及到的条件，因此在这样的情况下，要小心地使用扩展用例。练习10-2就是关于这种扩展用例类型的
116 一个练习。

Alan Williams曾提出了一个实用的原则，该原则用来帮助人们判断一个用例是否应该调用一个子用例，还是第二个用例是否应该从基用例扩展：

> 如果触发事件包括了基用例应负责的事件，即基用例知道什么时候/在那里/为什么第二个用例应该开始，那么基用例应包括/调用/引用另外一个用例。
>
> 如果触发事件包括了第二个用例应负责的事件，即第二个用例知道什么时候/在那里/为什么它应该开始，那么第二个用例应扩展基用例。

注意当基用例调用第二个用例时，基用例指定第二个用例在什么时候和什么地方执行。被引用的用例不包含基用例的名称。当第二个用例扩展基用例时，基用例不用指定，实际上也不用知道关于扩展（中断）用例的任何事情。扩展用例指定它所中断的用例，并且决定在什么情况下执行用例。

10.3 练习

扩展用例

10-1 在PAF系统的用户目标用例（见练习4-4）中找到一个条件，这个条件需要产生一个子用例。写出子用例，并把它连接到用户目标用例，注意成功和失败的情况。

10-2 考虑一个ATM系统的情况，这个ATM不是你开户银行的ATM，而是另外一家银行的ATM系统。写一个银行之间取消请求的子用例，并连接到你以前的关于归还现金的用例中。使用两种方式来实现：将子用例作为基用例的引用，或作为一个扩展用例。与同伴讨论你喜
117 欢哪一种方式，并阐述原因。

第11章
用例格式

11.1 供选择的格式

11.1.1 完整正式的用例格式

在本书的大部分例子中都使用了我所喜欢的格式，这种格式是完整正式的：

- 单列文字（不是一个表格）
- 步骤编号
- 没有条件语句
- 扩展部分的编号规则是数字和字母的组合（例如：2a，2a1，2a2等等）

与这种正式的格式相比，还有一些其他的格式，例如非正式格式、两列格式、还有Rational统一过程模板，这些格式在接下来的章节中介绍。然而，当我把内容相同，但格式不同的用例给一个开发组看了后，他们几乎都选择了单列、步骤编号的版本，因此我一直使用，并向别人推荐这种格式。下面是一个基本的模板，世界上很多项目组已经在不同的文字处理器上实现这种模板，比如：Lotus Notes、DOORS、Word、Access等等。

用例24 完整正式的用例模板<名字>

<用例名应该是一个用主动语态动词短语来表示的用例目标>
使用语境：<目标较长的描述，如果需要，还包括触发条件>
范围：<设计范围，在设计时将系统作为一个黑盒来考虑>
级别：<概要、用户目标、子功能三者之一>
主执行者：<主执行者的角色名称或主执行者的描述>
项目相关人员和利益：<用例中的项目相关人员和关键利益的列表>
前置条件：<我们所希望的，周围环境已经达到的状态>
最小保证：<在所有退出操作前，如何保证得到必须的信息>
成功保证：<目标完成时环境的状态>
触发事件：<什么引发了用例，可能是时间事件>
主成功场景：
<在这里写出从触发事件到目标完成以及清除的步骤>
<步骤编号#><动作描述>

扩展：

<在这里写出扩展，每次写一个扩展，每一个扩展都指向主场景中的特定步骤>

<被改变步骤><条件>：<动作或子用例>

<被改变步骤><条件>：<动作或子用例>

技术和数据变化列表：

<在这里写出场景中因技术或数据变化而引起的可能分支>

<步骤或变化编号#><变化列表>

<步骤或变化编号#><变化列表>

相关信息：

<项目所需要的所有附加信息>

11.1.2　非正式的用例格式

相对于完整格式，下面的例子是一个非正式的用例，来自Grady Booch的设计草稿。我在原稿上加了主执行者、范围和级别，使它成为一个非正式格式的完整例子。注意在第二段中仍然有扩展。

用例25　实际登录（非正式版本）

主执行者：用户

范围：应用程序

级别：子功能

在登录前，系统要求用户输入用户名和密码。系统验证提交的用户名是否存在，密码是否有效。然后用户被授权使用所有其他提交者的命令。

如果一个提交者的用户名是管理员，那么该用户被授权使用所有提交者和管理员的命令。

120　如果用户名不存在，或者用户名和密码不匹配，就拒绝用户登录。

11.1.3　单列表格格式

一些人喜欢将场景步骤放在表格中。虽然我认为表格中的行使用例变得不清楚，但有的人经常选择表格方式。表格11-1就是单列表格的模板。

表11-1　用例的单列表格格式

用例#	<用例的名称应该是一个用主动语态动词短语来表示的用例目标>
使用语境	<如果需要，写出语境的详细陈述>
范围	<在设计时将系统作为一个黑盒来考虑>
级别	<概要、用户目标、子函数三者之一>

（续）

主执行者	<主执行者的角色名称或描述>	
项目相关人员和利益	项目相关人员	利益
	<项目相关人员名称>	<项目相关人员取得的利益>
	<项目相关人员名称>	<项目相关人员取得的利益>
前置条件	<我们所希望的，周围环境已经达到的状态>	
最小保证	<在所有退出操作前，如何保证项目相关人员的利益>	
成功保证	<如果目标完成时满足项目相关人员的利益>	
触发事件	<触发系统启动用例的动作>	
描述	步骤	活动
	1	<在这里写出从触发事件到目标完成以及清除的步骤>
	2	<...>
	3	
扩展	步骤	分支动作
	1a	<引起分支的条件>
		<活动或子用例名称>
技术和数据变化		
	1	<变化列表>

121

11.1.4 双列表格格式

Rebecca Wirfs-Brock发明了一种会话（*conversation*）方式的表格，这种方式的特征是使用了两列，左边一列是主执行者的动作，右边一列是系统动作。会话方式主要用于设计用户界面，因此它们包含了更详细的用户移动。

你可以用双列表格的形式来编写用例。这样很清晰，但往往太长了，甚至可能超过3页（参见用例36寻找解决方案——修改前）。通常，在适当的目标级别中，直到我们将文档减少到3至9个步骤时，人们才会发现文档是如此简单和清晰，根本不需要列表。

正如《*Software for Use*》一书中所描述的，Constantine和Lockwood在捕获基本用例（*essential use case*）中的用户界面需求时，采用了这种会话格式。不同的是，在基本用例中所有的用户移动（界面细节）都被省略了，因此用例描述很短，也很准确。

使用双列格式来捕获行为需求的困难是没有地方描述其他的支持执行者。一个解决方法可能是再加一列，但是我从没有听说有人建议过这样做，也没有见到有人这么做过。这是因为会话和基本用例都是为了获得用户界面需求，而不是整个系统的行为需求。

考虑了所有的这些问题，很多人仍然觉得双列格式很有吸引力。如果你喜欢，可以实践一下，看看Wirfs-Brock、Constantine和Lockwood是如何处理的。表格11-2是用双列格式描述的场景片段。

表11-2　双列表格

用　户	系　统
输入订单号	
	检测订单号和当月的中奖号码是否一致 登记用户和订单号作为当月的中奖者 发送电子邮件通知销售经理 祝贺用户，指导用户如何得到奖金
122 退出系统	

11.1.5　RUP格式

Rational统一过程（RUP）中的模板与完整正式格式很类似。步骤编号是可选的。扩展另外有自己的标题，扩展又叫做可选流程（*alternate flow*）。本书中的所有用例都可以用这种格式很好地描述，尽管标题的编号有一些混乱，但是这种格式还是很吸引人而且容易学会。下面是它的一个基本格式：

1. 用例名称

1.1. 简要描述

……文本……

1.2. 执行者

……文本……

1.3. 触发事件

……文本……

2. 事件流程

2.1. 基本流程

……文本……

2.2. 可选流程

2.2.1. 条件1

……文本……

2.2.2. 条件2

……文本……

2.2.3. ……

3. 特殊需求

3.1. 平台

……文本……

3.2. ……

4. 前置条件

……文本……

5. 后置条件

……文本……

6. 扩展点

……文本……

Rational软件公司把下面的用例作为一个例子发给了我。通常，这个用例会被画在某个工具集的用例图或其他一些图中。我发现用例有很强的自解释性，我想读者也会发现这一点。简单自然段和编号的步骤都很适合于描述用例。因此我只在标题上加了两个图标来保持本书中例子的一致，而在模板中没有加入任何其他的域。

第14章中的用例32“管理报表用例”也使用了RUP模板。 123

用例26 登记课程

1 **用例名称**：登记课程

1.1 简要描述

这个用例允许学生登记本学期需要学习的课程。在学期开始的阶段，学生也可以修改或删除他所选择的课程。课程目录系统提供了本学期开设的所有课程列表。

本用例的主执行者是学生。课程目录系统是这个用例的一个执行者。

2 **事件流程**

当学生从主窗口中选择“维护计划”活动，用例开始执行。[参考用户界面原型中的屏幕布局和字段]

2.1 基本流程

2.1.1 创建计划

2.1.1.1 学生选择“创建计划”活动。

2.1.1.2 系统显示空白的计划窗口。[参考用户界面原型中的屏幕布局和所需字段的域模型]

2.1.1.3 系统从课程目录系统中检索出有效的课程列表。[如何选择和显示？文字？下拉列表？]

2.1.1.4 学生从有效课程列表中选择四门主课和两门选修课。课程选择完毕，学生选择“提交”。[在项目术语表中定义“主课数目”和“选修课数目”。必须恰好等于4门或2门吗？或者是“不少于4门……”，等等]

2.1.1.5 在这一步对已选的每门课程执行“加入已选择课程子流程”。

2.1.1.6 系统保存计划。[什么时候更新整个计划？立即？晚上（进行批处理）？]

2.2 可选流程

2.2.1 修改计划

2.2.1.1 学生选择“修改计划”活动。

2.2.1.2 系统检索和显示学生当前的计划（例如本学期的计划）。[该计划是只在本学期有效吗？]

2.2.1.3 系统从课程目录系统中检索本学期所开设的所有课程。系统向学生显示检索

到的课程列表。

124 2.2.1.4 学生就可以通过增加或删除课程来修改所选课程。学生从有效课程列表中选择增加的课程。学生也可以从当前的计划中选择任何想要删除的课程。当修改完毕，学生选择“提交“。

2.2.1.5 在这一步对已选的每门课程执行“加入已选择课程”子流程。

2.2.1.6 系统保存计划。

2.2.2 删除计划

2.2.2.1 学生选择“删除计划”活动。

2.2.2.2 系统检索和显示学生当前的计划。

2.2.2.3 学生选择“删除”。

2.2.2.4 系统提示学生确认这次删除。

2.2.2.5 学生确认这次删除。

2.2.2.6 系统删除计划。[在什么情况下，释放学生所占的空间？]

2.2.3 保存计划

在任何情况下，学生可以通过选择“保存“来保存一个计划，不必提交它。当前的计划被保存，但学生不能再加入已选的课程。在计划中，所选的课程被标识为“被选择”状态。

2.2.4 加入可选课程

系统确认学生符合条件，并且可选课程有效。然后，系统将学生加入已选择课程中。在计划中，所选课程被标识为“已登记”状态。

2.2.5 条件不满足或课程满员

如果在“加入可选课程”子流程中，检测出学生没有满足所需条件，或要选择的课程已经满员，则系统显示错误信息。学生可以选择其他课程或取消本次操作，在这种情况下，用例重新开始。

2.2.6 没有发现计划

如果在“修改计划”或“删除计划”子流程中，系统不能检索到学生的计划，系统显示一个错误信息。学生确认该错误，用例重新开始。

2.2.7 课程目录系统不可用

系统在一定次数的连接尝试后，如果仍不能连接到课程目录系统，系统将显示一
125 个错误信息。学生确认该错误信息，用例终止。

2.2.8 课程登记结束

如果在学生选择“维护计划”的时候，本学期的课程登记已经结束，系统显示一个信息，用例终止。学生在本学期的课程登记结束后就不能登记课程了。

3 特殊需求

在本用例中没有特殊的需求。

4 前置条件

4.1 登录

在这个用例开始之前学生必须先登录系统。

5 后置条件

在本用例中没有后置条件。

6 扩展点

在本用例中没有扩展点。

11.1.6 条件语句格式

程序员不可避免地想在文本中使用条件语句。他们认为，使用条件语句毕竟比学习怎样写扩展更加容易，例如：

如果订购号与中奖号相同，则<所有中奖的业务活动>，否则告诉顾客这不是一个中奖号码。

如果在用例中只有一个条件语句，则我同意这样做。实际上，在用例模型中并没有排除“if...then...else”语句。而且，甚至在只有两个条件语句的情况下，用例就变得很难理解，更不用说有3个、4个和5个条件语句的时候了，甚至也可能在一个条件语句中又嵌套了另一个条件语句。

当人们坚持使用条件语句时，我就让他们使用条件语句，然后请他们回来报告编写用例的经验。每个使用条件语句的人在很短时间内就得出了结论，使用条件语句后用例变得很难阅读，所以他们又转回来使用扩展格式。因此，在编写格式上我强烈建议“不要在你的场景中使用条件语句”。

11.1.7 Occam格式

如果要为你的用例构造形式化的编写模型，首先关注一下Occam语言，它是由Tony Hoare发明的。使用Occam语言，可以比我所知道的其他语言更容易地描述你所需要的交互、并行和可选的动作顺序。我不知道Occam怎样处理异常，异常处理在编写扩展时是需要的。 126

在Occam语言中，你可以这样写

```
ALT
    替换动作1
    替换动作2
    TLA（结束替换动作）
PAR
    并行动作1
    并行动作2
    RAP（结束并行选择）
OPT
    可选动作
TPO
```

如果你想要创建或使用一种形式化的语言来描述用例，可以首先采用用例22“损失登记”

作为第一个测试用例，这个用例中包含了并行、异步、异常和协作处理等多种活动。我想这个例子会显示出自然语言描述这些活动的好处，而且很容易理解。

11.1.8　图形方式

用例详细说明了执行者为了达到目标而进行的活动和交互作用。有一些图形可以表达这些细节，例如顺序图、协作图、活动图和Petri网。如果使用图形来描述用例，你仍然可以利用本书中介绍的大部分思想。

图形符号有两个可用性方面的问题。第一，最终用户和业务执行者不可能熟悉这些符号，也不会有耐心来学习这些符号。使用图形符号使你不能很好地与一些有价值的人员进行交流。

第二，图形不能完全表示出你所需要的意思。我见过的一些CASE工具通过交互图来实现用例，并迫使设计者将一些相关的文字信息隐藏在一个与交互箭头相关联的弹出式对话框中。这让用例很难阅读——用户不得不双击每一个箭头来查看藏在里面的信息。在我主持的一次讨论中，用例设计者和读者都选择了不使用CASE工具支持，他们认为使用简单的字处理工具来写文档比用CASE工具来画图更好。

127　下一部分讨论一种图形方式，这种方式并不适于用例。

11.1.9　UML用例图

UML中的用例图是由椭圆、箭头和一些“小人”图符组成，它并不是用来捕获用例的。椭圆和箭头显示了用例的打包和分解，而不是用例的内容。

回想一下，用例描述一个目标。它由场景组成，场景又由许多执行步骤组成，每个执行步骤完成一个目标。因此这些步骤也可以被展开成一个用例。我们有可能把用例目标画成一个椭圆，把每个执行步骤也画成一个椭圆，然后把它们用箭头连起来，并在箭头上标明“包含”关系。这样可以将最高层的用例分解为最低层的用例，但却产生了一张奇形怪状的图来表示整个行为的分解。

这样的椭圆图遗漏了用例的必要信息，例如，哪个执行者执行了活动，哪个执行者关心活动的步骤。这些信息是非常有用的，而且应该被保存下来。参见提示24：“大的图画恶作剧”，附录A的A.1节“椭圆和‘小人’图符”。

我的观点是不要用一个椭圆来代替用例的文字。在一次演讲中，一个学生问我：“你什么时候开始写用例的文字？是不是从分解到叶级别的椭圆图开始？”

答案是用例本身就是文字的，任何图形都只是为了帮助读者找到他们所要阅读的文字。

很多人发现顶层用例图（顶层用例图表示了外部执行者和用户目标用例）是很有用的。它提供了一个语境图（*context diagram*），类似于人们已经使用了很久的语境图。用例图的价值除此之外就没有什么了。更多的讨论见附录A。

11.2　影响用例书写格式的因素

在1998年的OOPSLA大会上，12个资深的用例设计者和教师聚集在一起，讨论一些常见的

编写用例时容易混淆的地方和困难，以及是什么使得人们写出的用例各不相同。大会的组织者Paul Bramble收集了这些意见并将意见分类。用例在不同情况下可能会不同，你不用担心这个问题，这是大家都无法避免的。

你可能发现在下面所列的情况中，有些情况组合起来迫使你不能按期望来工作。耐心一点，宽容一点，你就可以写出符合要求的用例。在列出了所有这些影响因素后，我们可以找到一个一致的问题答案："怎样写出可读的用例？"

128

矛盾的因素：业务环境、社会作用、不同文化

你想要描述用例，但是可能遇到下面的情况或争论（我不是想平息这样的争论，而是要你认识到并不是只有你会遇到这样的情况）：

我们总是按照另外一种方式去做事情。

在不同的文化背景中，你可能发现：

在小组中存在偏见。

存在不同的工作文化，在这些文化的影响下，人们只不过在"以不同的方式做事"。

写用例的人和读用例的人使用不同的词汇表。

理解层次

在不同的时间、地点，不同的人对同一件事情会有不同的理解。你可能会修改推荐的用例编写格式，这可以根据：

现在你知道多少

- 关于领域
- 关于用例

你知道当前阶段是处于软件生命周期的哪个阶段。

你是否需要确定工程的内容或花费。

你现在是需要广度的研究还是深度的研究。

- 暗中或慢慢地分析现状。
- 人们都强调他们所知道的内容。
- 时间计划、知识深度与领域知识的对立。

项目相关人员的要求

你应注意下面这些方面：

用户是读者，也是用例的使用者，他们关心用例的顶层描述。

IT公司的技术人员是编写者或实现者，他们关心细节描述。

不同的开发组可能想通过服务小组来表达他们对自己所关心用例的不同观点，或者不同的开发组对于是建立一个完整模型还是部分模型产生分歧。

是否还涉及其他不同的读者？如果有，他们是谁？

经验与格式

129

经验（*experience*）：每个小组中都有用例的初学者，但是他们很快就会成为"有经验"的设计者。有经验的人知道一些捷径；初学者则希望有清晰的方向和一致的指令。

格式（*formality*）：不管是对有经验的设计者还是初学者，领导者或部门传统的工作方式都需要有一种正式（或非正式）的书写格式。

覆盖面

用例的覆盖面依赖于开发组的组成、编写技巧、成员之间的交流以及他们是需要覆盖整个问题，还是只需要与读者交流信息。

覆盖面的变化可能是由于：

- 领域问题专家（他们关心的问题可能集中在某一个方面）
- 编写者人数
- 读者人数
- 实现者人数
- 业务人员是否知道他们需要什么
- 按照一个通常模式来工作的决定
- 开发组成员的地理位置分布

一致性

用例内容一致性的要求可能会与客户需求的不一致性和不确定性冲突：

- 需求的易变性
- 格式的一致性

复杂度

用例的复杂度涉及下面几个方面：

- 对用例完整性的要求
- 描述整个问题领域
- 多角度的描述
- 简化系统视图
- 表达的简化

130

- 细化用例表达
- 范围窄的视图和范围宽的视图
- 问题的复杂度
- 加入技术细节的要求（特别是在问题很难解决的时候）

下面的因素影响了系统的复杂度：

- 分析停滞（系统的复杂度超过了分析师的分析能力）
- 执行者的数目
- 功能点的数目
- 系统的种类
- 用户系统的简单性
- 实时系统
- 嵌入式系统（必须有容错能力）

冲突

需求是为了解决用户冲突，但需求的不确定性又掩饰了冲突。

完整性

妨碍用例完整性有下面几个因素：

- 工程的需求不完整。
- 设计者没有接触过用户（用户不是你的顾客）。

目标与任务——完成什么与怎样完成

用户经常细化需求而不是系统的使用方法。

系统环境可能和使用方法冲突。

活动和任务描述了在系统会发生什么，而不是为什么发生。

资源

写出好的用例是需要时间的，但项目的时间是有限的。在编写项目的用例中就必须引入管理机制。

其他因素

其他影响用例的因素有：

- 工具需要/支持
- 不明确的目标
- 将需求分块，并分别进行分析
- 无约束的设计与将要进行的设计的级别
- 抽象用例与具体用例
- 可描述性
- 合作的积极性

131

这个列表有些与众不同。尽管本书中大部分内容适用于所有的情况，但你可以参考这个列表来决定采用多少形式化的东西，是否现在少做一些，而将来多做一些，设计到什么程度或怎样来安排设计的阶段，在深入编写用例之前应该对问题理解得多广泛或多精确。

11.3 五种项目类型的标准

你参加一个项目，和其他人一起阅读了本书，因此你们已经了解了本书的内容。现在的问题是“我们应该采用什么标准？”。答案依赖于你们扮演什么角色，你们的技术如何，还有当前阶段的目标是什么。请将你的答案和列出的因素进行比较。

在这部分，我推荐五种情况下的书写标准。你会注意到每一种标准都是介于非正式格式和完整正式格式之间的标准。这五种情况是

1) 了解需求，甚至包括用例根本不在最后的需求文档中使用的情况。

2) 业务过程建模。

3) 设计和量化系统需求。

4) 在一个短期、高强度的项目中编写功能需求。

5) 在一个长期、大型项目中，在增量式开发开始时编写详细的功能需求。

你应该根据实际情况使用这些标准。经过思考后，你可能决定根据公司的要求或当时的要求，或根据本书给出的原则对这些标准进行一些调整。

在下面的例子中，有一个叫做MyCo的公司，要开发一个新系统Acura以代替旧系统BSSO。
132 我这样做是想提醒你不要使用“公司”和“系统”这样不明确的词，而是使用它们实际的名称。

11.3.1 需求了解阶段用例

当你的最终目的是发现需求时（请重新阅读第1章中Steve Adolph关于“在新领域‘发现’需求”的报告），使用这个模板。记住你的需求文档可能不是以用例的格式来书写的。而这些格式都可以很容易地通过用例进行转换，因此可以采用用例这种灵活的工作方式来完成需求分析。

用例27 需求了解用例模板——Oble a New Biscum

范围：Acura
级别：海平面
使用语境：The quibquig needs to oble a new biscum once a dorstyp gets nagled.（在操作语境中关于目标的文字描述。）
主执行者：quibquig（或所有主执行者）
项目相关人员和利益：Qubquig,MyCo,...（适当的人和物）
前置条件：在用例被触发前必须具备什么条件。
触发事件：The quibquig selects the obling function（可以是任何事件）。
主成功场景：
...A paragraph of stuffstuff describing the quibquig successfully obling a biscum with Acura...actors do this,that,and the other.
发生频率：Yay many times a day(一天中有多次)
未解决问题：……在这里填写一些好想法……

这个模板是非正式的。在模板中保留项目相关人员和利益来提醒我们关于他们的所有需求，但不包括系统的保证。

你的用例通常作为一个黑盒，大多数都是用户目标级别。你可以用一个较高层的用
133 例来描述语境，但是你最好不要经常使用海平面级别以下的用例。

11.3.2 业务过程建模用例

用例28 业务过程用例模板——Symp a Carstromming

范围：MyCo操作
级别：概要

使用语境：在操作语境中关于目标的文字描述
主执行者：所有主执行者
项目相关人员和利益：适当的人和物
最小保证：所有保证
成功保证：所有保证
前置条件：在用例被触发前必须具备什么条件
触发事件：可以是任何事件
主成功场景：
1. ……动作步骤……
2. ……
扩展：
1a. ……扩展条件：
  1a1. ……动作步骤……
发生频率：所有时刻
未解决问题：……在这里填写一些好想法……

这个模板是用于重新设计新软件的业务或过程。阅读这些用例的人往往是一些高层人员、部门经理和高级执行官，因此尽量使这些用例可读性强，减少细节的内容。步骤的编号突出了活动的时间顺序。一定要在扩展中描述错误的处理，这样才能揭示重要的业务规则。

顶层或最外层用例就是一个黑盒，表示了公司与外部合作伙伴的交互。这些用例既可以作为评测业务过程的详细定义，又可以作为白盒用例的语境。与系统（比如上例中的“*MyCo*操作”，相互作用的白盒用例表示整个组织的活动，表示人和部门合作实现整个组织的任务。你可以采用从云朵到海平面的目标级别。在模板中，我为这个例子选择了一个风筝级的概要目标。 134

11.3.3 确定系统需求用例规模

用例29 确定系统需求用例规模模板——Burble the Tramling

范围：Acura
级别：概要
使用语境：在这里填写前置条件或正常用途的条件
主执行者：任何人
……在这里用几句话来描述执行者如何在主成功场景中成功地完成工作……
……在这里用几句话来描述可选的路径和处理……
发生频率：多久
未解决问题：……在这里填写一些好想法……

当你为估计系统大小和结构而分析需求时，使用这个模板。以后，你可以细化需求直到完成一个完整的用例。如果你的项目具有某些特征（参见3.1节中的“用例摘要”小节以及第22章中的提示19：“认识到错误的开销”），可以选择从非正式用例开始设计。

作为一个中等精度的前期工作，这个模板是非正式的，并且它所讨论的系统可以是系统，也可能是一个业务。该用例目标可以是任何级别的，包括子功能级别，因为项目的大小很大程度上依赖于子功能用例的复杂性。在前面的例子中，我使用了Acura和用户目标。参见本章中用
135 例25“实际登录（非正式版本）”

11.3.4 短期、高强度的项目用例

当你需要去编写需求，但项目实施时间很短时，你可以使用上面的模板。由于时间和经济的原因，需要避免编码的开销和编写完整模板；因此你可以使用非正式的格式。但是你还是必须了解前置条件、保证条件和扩展。在这种情况下，我假定你会增加项目开发组内部的交流，参考第22章中的提示19：“认识到错误的开销”。

用例30　高强度项目用例模板——Kree a Ranfath

范围：Acura
级别：用户目标
使用语境：
主执行者：
项目相关人员和利益：
最小保证和成功保证：
前置条件：
触发事件：
主成功场景：
……用来描述执行者在主成功场景中完成目标的一段文字……
扩展：
……用来描述所有可选的路径和处理的一段文字……
发生频率：
136 **未解决问题**：……

11.3.5 详细功能需求用例

用例31　用例名称——Nathorize a Permion

范围：Acura
级别：用户目标

使用语境：
主执行者：
项目相关人员和利益：
最小保证：
成功保证：
前置条件：
触发事件：
主成功场景：
1. ……
2. ……
扩展：
1a. ……
1a1. ……
发生频率：
未解决问题：……

当你收集行为需求，并且需要完整正式的用例格式中的所有信息时，可以使用上述模板。这种模板可能应用在大型、关键、昂贵的项目中；或者应用在有固定预算的项目中；或者应用在开发组分布在不同地方，在增量式开发的开始阶段需要扩充和检查以前设计的量化用例时；或者这样做是你的习惯。

你设计的系统可能是任何系统，执行者可能是任何人，目标可能有任何要求。在样例模板中，我再一次使用了Acura和用户目标级别。

11.4 总结

所有这些不同的用例格式都表达了大致相同的基本信息。本书中的所有建议和指导都可以用于每种格式。不要为你应该在项目中采用哪种格式而烦恼，只需要选择一种既适合于编写者又适合于读者的格式。

137

11.5 练习

条件语句

11-1 重写下面的用例，去掉条件语句，在适当级别和可选场景或扩展中使用表示目标的短语。

完成清洁火花塞服务

条件：火花塞脏了或顾客要求服务。

1. 打开引擎罩。
2. 找到火花塞。

3. 用保护材料覆盖挡泥板。
4. 取下火花塞。
5. 如果火花塞有裂缝或破了，换上新的火花塞。
6. 清洁火花塞。
7. 清洁每个火花塞的间隙。
8. 根据需要调整间隙。
9. 测试火花塞。
10. 装上火花塞。
11. 连接点火线到对应的火花塞。
12. 检查引擎性能。
13. 如果正常，转到步骤15。
14. 如果不正常，执行指定的步骤。
15. 清洁工具和设备。
16. 清理汽车上的油渍。
17. 填写必须的文件。

138　**结果：**引擎运转正常。

第二部分

经常讨论的主题

第12章
什么时候才算完成

当满足如下要求时，才算完成任务：

- 已经命名了与系统相关的全部主执行者及其用户目标。
- 捕获了系统的全部触发条件，既包括用例触发条件，也包括扩展条件。
- 编写了所有用户目标用例以及必要的概要用例和子功能用例。
- 每个用例描述足够清晰，使得：
  —投资方确认他们能够区分这个用例是或不是实际上要提交的用例。
  —用户确认用例表达了他们的要求，或者能接受这些用例作为系统的行为。
  —开发人员确认可以实现所有用例表示的功能。
- 投资方确认用例集覆盖了他们所有的需求。

全部主执行者及其用户目标

通过全部主执行者及其用户目标，可以确定系统的实现范围。由于除了需要人头脑中的系统构想外，没有其他信息来源可与主执行者及用户目标的列表进行比较，因此我们无法知道怎样对其进行处理。只能凭猜测来处理，因而有必要采用集体讨论的方式对主执行者及其目标列表进行多次检查。

全部触发条件

这些触发条件表示系统边界的微调。系统必须对它们进行响应。在用例中，一些触发事件以用例事件形式出现，例如，用户把卡插入槽中、顾客对保险合同增加或减少条款、用户选择更新或安装软件等；另一些触发事件则作为场景扩展条件，例如，用户按取消按钮，系统检测电源中断，等等。

有一个方法可以对系统触发条件进行重新检查，即通过识别系统中所有具有生命周期的元素并重新审查它们的生命周期，寻找所有改变它们状态的事件。

概要用例和子功能用例

概要用例建立了用户目标用例的环境。它们回答了一个经常被问及的问题："所有这些（用户目标）用例如何组装在一起？"事实上，每个用户用例都处于其上层用例中，如此一级一级上升，直到单个根用例。根用例是一张目录表，不再有层次结构。新读者会发现有一个单一的根用例很有用处，他们能够从这里出发对系统中每个用例进行访问。

子功能用例为用户目标用例提供支持。只有被多个其他用例调用或者分离一个复杂行为时，子功能用例才有存在的必要。

用例的确认

只有投资方和应用专家阅读这些用例并认为用例表达了他们所有的需求；系统开发人员阅读这些用例并认为他们能够根据用例描述开发符合需求的系统时，才能认为这些用例合格。但

这是非常艰难的挑战，是对编写需求的挑战。

12.1 关于“正在完成”

短语“正在完成”给人的印象是在开始设计任务之前，就应该从头到尾编写所有用例。须知情况并非如此。参考17.1节“用例在项目组织中的作用”，其中对开发一个可以部分交付用例的项目开发计划进行了讨论。还可以参考《*Surviving Object-Oriented Projects*》(Cockburn, 1998)中对于增量式开发的详细描述；文章“VW-Staging”(*http://members.aol.com/acockburn/papers/vw-staging.htm*)对于增量和迭代式开发的简短讨论。

不同的项目组根据其不同的实际情况，通常采用不同的策略。一些项目组也许是为了对项目投标，一开始就草拟了所有用例。但他们必须清楚这样编写的用例在项目过程中需要被不断地修改。一些项目组则只草拟系统执行者和用户目标，按适当的增量开发方式，开始用例的详细设计。还有一些项目组花6～9个月时间创建用例，直到这项工作快结束时才开始讨论所有其他需求。还有的项目组在每开始一轮工作之前才编写必需的用例。当然，上述每种策略都有其
142 可取之处。

第13章 扩展到多个用例

在对大量用例进行处理时，可以采用两种行之有效的技术：对每个用例进行简单说明，或者把用例分为多个簇。

13.1 简单描述每个用例(低精度表示)

给每个用例单独命名是非常有用的，这也是把用例名作为第一级精度的原因。好的用例名字集合便于操作整个用例集，特别是为评估、规划和跟踪提供了便利。可以把所有的用例名放入一张电子表格中，利用电子表格功能对其分类、排序，统计各种我们感兴趣的用例特性。在1.5节“合理安排你的精力”和17.1节“用例在项目组织中的作用”中都讨论了这种方法。

第二级精度是每个用例的简介，即用两句到三句话对每个用例进行概括。简介也可以放入电子表格或表单中，有利于审查（见表3-3“用例简介的例子”）。另外，用例简介也可以作为系统和项目组织工作的概述。

当需要从整体上浏览用例集时，这种粗略的描述方法就显得非常有用。另外，当需要对用例集进行分簇时，也能节省不少时间。

13.2 创建用例簇

如果使用基于文本的工具（如LotusNotes、电子表格和字处理程序），可以借助给每个用例加注标记，这种常用技术形成用例簇。如果使用UML工具，可以将用例簇作为包。现有三种常用且有效的分簇技术，分别描述如下：

按执行者分类

最容易想到的方法是对用例集按它们所属的主执行者分簇。但是，对于一个含有80～100个用例的系统，有可能出现主执行者过多或过多的用例集中于某一个主执行者的情况，也有可能出现对一些用例来说主执行者重复过多的情况。因此，这种方法其效率不高。

按概要用例分类

在用例生命周期中，或在一个大项目生命周期的某个阶段中，一些用例从一开始建立就自然地处于某个簇中。这些相关用例处于同一概要用例中。如果这时还没有编写概要用例，则可能在对用例分簇前，还必须创建、修改或删除自然分簇的某些信息。在一个项目中，系统维护着一本虚拟的顾客支票簿，我们把所有改变支票簿的用例称为“支票簿用例簇”。这个簇中所有用例由同一个开发组进行开发，开发组的成员可以按一种便于项目经理操作的方式保持开发进度一致。

按开发组和版本号分类

对用例集按负责开发设计工作的开发组和版本号进行分簇，有利于简化项目跟踪工作。项目跟踪工作很自然地变成问这样一些问题："用户的每个用例簇是否按时完成？"。但是，这种分簇方法通常只用于大型项目开发。

按主题域分类

如果一个项目有超过100个用例，人们在描述用例时通常会自动按主题对其进行分类，这样很容易弄清某个主题所涉及的领域。例如，有的项目可能用顾客信息、宣传、发货、广告等作为主题，而另一项目可能用订单、运送、跟踪、传送和账单作为主题。在不同的主题域下，通常会有子项目。每个主题域也可能包含20～100个用例。

虽然跟踪240个用例是非常困难的，但是跟踪15～20个簇还是能够做到的。对于一个大项目，可以首先按主题域分成3～6个包含20～100个用例的簇，然后再对每个簇按开发组和版本号分为子簇。最后，根据这些用例簇中的用例数目，使用相关用例或者概要用例簇来管理整个工作的进度。

144

第14章 CRUD和参数化用例

14.1 CRUD用例

到目前为止，对如何组织创建一个实体、检索一个实体、更新一个实体以及删除一个实体这类用例，一直没有一致的意见。我们把这些基于数据库操作的小用例称为CRUD（Create，Retrieve，Update，Delete）用例。但现在面临的问题是，这些小用例是一个大的管理实体用例的组成部分呢，还是它们相互独立？

原则上，由于每个小用例都表达了单独需求，甚至还可能由不同的人在不同安全级别上执行，因此它们是独立的。但它们打乱了整个用例集，使需要跟踪的用例成倍增加。

对如何最有效管理CRUD用例存在不同的观点。S.R.A公司的Susan Lilly认为分离的CRUD用例便于追踪，主执行者可以安全地调用不同功能。但是，我倾向于先使用单个管理实体用例对其进行处理，这样可以减少用例簇数目。如果编写用例的复杂度增加，则像8.4节中“从扩展中创建新用例”小节描述的一样，再对其进行分裂。然后用电子表格跟踪用户对系统数据和功能的访问。在实际使用中，两种方法都可以使用，到目前为止没有足够的证据表明哪个方法好或不好。

以下用例由Empower IT公司的John Colaizzi 和Allen Maxwell编写。[⊖] 开始时，他们用上述两种方法编写用例，随后却决定把各种小用例合并为一个概要级管理用例。到最后，由于用例复杂度增加，又从中分离出存储子用例。这些用例展示了如何把自己的用例编写风格和给定用例模板结合起来。下面是他们用 Rational统一过程模板，对步骤和扩展进行了编号：

用例32 管理报表用例

1 概述：

报表操作包括创建、存储、删除、打印、退出和显示，管理报表用例对所有这些操作以及如何使用这些操作进行了描述。由于该用例精度级别很低，有时需要利用其他用例来完成目标。在“特殊需求”一节中列出了帮助完成目标的用例。

1.1 执行者：

用户(主要的)。

文件系统：用户使用的典型PC文件系统或网络文件系统（次要的）。

1.2 触发条件

⊖ 从作者那里得到允许可以采用。

用户在浏览器中选择报表操作。

1.3 事件流

1.3.1 基本事件流——打开、编辑、打印、存储和退出报表。

a. 用户在浏览器中通过单击来选择报表，然后选择“打开”菜单项(打开操作也可以在浏览器中通过双击报表完成)。

b. 系统在屏幕上显示报表内容。

c. 用户使用指定报告规格用例对报表布局进行设置。

d. 重复步骤c和d直到用户满意。

e. 使用退出报表用例退出报表。

f. 在步骤c之后的任何时候，用户都能使用存储报表用例对报表进行存储，也可以使用下面打印报表可选事件流打印报表。

1.3.2 可选事件流：

1.3.2.1 创建新报表

a. 用户在浏览器中单击鼠标右键，从弹出菜单中选择“创建新报表”菜单项。系统将用缺省名字创建新报表，并且将报表名状态设置为“未设定”，报表状态为“修改”。

b. 用例流程从基本事件流中步骤b开始继续执行。

1.3.2.2 删除报表

a. 用户在浏览器中单击报表，然后选择删除。

b. 系统打开报表(如果报表已经打开，则使其变为当前报表)，然后由用户对删除报表进行确认。

c. 报表关闭，清除相关资源。

146

d. 系统从报表列表中清除该报表记录，并从存储介质上删除报表数据。

1.3.2.3 打印报表

a. 用户在浏览器中单击报表，然后选择打印，或者选择当前报表的打印选项进行打印（使用这个用例的报表在基本事件流中正在被编辑或显示）。

b. 用户选择打印机并设定打印机参数（通过操作系统提供的打印对话框等），或者选择“打印到文件”选项。

c. 系统加载报表数据及其格式，然后向操作系统提交打印报表作业或者打印报表到指定文件，然后关闭报表。

1.3.2.4 复制报表

a. 在浏览器中单击报表，然后选择复制。

b. 系统提示输入新文件名，并验证新文件名是否存在。

c. 重复步骤b直到输入有效的（不存在的）报表文件名，或者选择覆盖已存在文件，或者取消复制操作。

d. 系统将报表按指定文件名存储为新报表。

e. 如果复制选用已存在文件名，则删除已存在报表。

1.3.2.5 重命名报表

a. 在浏览器中选择重命名报表，并选择重命名命令。

b. 输入新文件名，验证文件名的有效性(如不能与原文件名相同、文件名是否存在及文件名字符是否有效等)。

c. 重复步骤b直到系统接受新报表名或用户取消此次操作。

d. 系统用新报表名对报表列表进行更新。

1.3.3 特殊需求

1.3.3.1 平台：

平台必须能显示报表并能进行控制，并且支持必要的人机交互界面。

1.3.4 前置条件

系统中存在数据元素并且当前被选中。 147

1.3.5 后置条件

1.3.5.1 成功后置条件（注意：这是成功保证）

系统等待与用户交互。报表被加载显示，或用户退出（关闭）报表。用户请求对报表所作的所有修改进行存储，并作了硬拷贝。另外，对报表列表进行必要更新。

1.3.5.2 失败后置条件(注意：这是最小保证)

系统等待用户操作，系统可能处于如下状态：

报表已经被加载。

报表列表仍然有效。

1.3.6 扩展点：

无

用例33 存储报表用例

1 概述：

用例描述报表的存储过程，由管理报表用例和退出报表用例调用。

1.1 执行者：

用户（主要的）。

文件系统：用户使用的典型PC文件系统或网络文件系统（次要的）。

1.2 触发条件：

用户选择管理报表用例或退出报表用例（该用例包含在管理报表用例中）中的操作，对此用例进行调用。

1.3 事件流：

1.3.1 基本事件流——存储新报表

a. 当用户选择“存储报表”菜单项时，用例启动。

b. 系统检测到报表名状态为“未设定”，然后提示键入新报表名，系统验证报表列表中是否存在该报表名，如不存在，将新报表名加入报表列表中。

c. 如果用户取消存储操作，用例结束。

d. 系统使用报表信息更新报表列表。如果报表还没有设定，则系统创建惟一报表文件名，并将它存入文件系统。

e. 报表设为“未修改”状态，名字状态设为“设定”状态。

148 f. 用例结束并显示报表内容。

1.3.2 可选事件流

1.3.2.1 可选子事件流——报表名存在—覆盖

a. 系统发现新输入的报表名在报表列表中，提示用户是否覆盖已存在报表。如果用户选择覆盖文件，系统则使用已经存在的文件名和报表条目，然后从基本事件流步骤d继续执行。

1.3.2.2 可选子事件流 ——报表名存在—取消

a. 系统发现新输入的报表名在报表列表中，提示用户是否覆盖已存在报表。如果用户选择取消，用例显示报表并结束。

1.3.2.3 可选子事件流 ——报表另存为……

a. 用户选择“报表另存为”菜单项。

b. 用户键入新报表名，系统验证报表列表中是否存在该报表名。如果名字不存在，继续向下执行。如果名字在列表中存在，系统提示用户是否覆盖报表文件。如用户选择不覆盖，用例从步骤b继续执行。

c. 如果名字不存在，用例从基本事件流中步骤d继续执行。

1.3.2.4 可选子事件流——存储报表名存在—覆盖

a. 系统在报表列表中发现新输入报表名，提示用户是否覆盖报表文件。如果用户选择覆盖报表文件，系统用报表名存储新报表文件，并从基本事件流中步骤d继续执行。

1.3.2.5 可选子事件流 ——存储报表名存在—取消

a. 系统在报表列表中发现新输入报表名存在，提示用户是否覆盖，用户选择取消，用例显示报表并结束。

1.3.2.6 可选子事件流——存储已存在报表

a. 用户对当前报表选择“存储报表”命令(当前报表指已经存取过，并在报表列表中存在)。

b. 系统在报表列表中找到当前列表，更新必需的信息，并保存报表特征到报表文件。

c. 将报表状态设为“未修改”。

d. 用例显示报表并结束。

1.3.3 特殊需求

没有

1.3.4 前置条件

系统中存在数据元素并且当前被选中。一个报表作为当前报表被显示，并且报表状态为“修改”状态。

1.3.5 后置条件

1.3.5.1 成功后置条件(注意：这是成功保证)

系统等待用户交互。报表被加载且在屏幕上显示。根据特定的存储操作要求，按报表名对报表列表进行更新。报表状态处于“未修改”状态。报表名处于“设定”状态。 149

1.3.5.2 失败后置条件(注意：这是最小保证)

系统等待用户交互。

报表被加载显示。

报表状态为“修改”，报表名字状态与用例开始时一致。

报表列表仍然有效(有时报表存储失败时，报表列表可能被清除)。

1.3.6 扩展点：

没有

14.2 参数化用例

在提取系统用例时，偶尔会遇到这种情况：一些用例大致相同。例如，*搜索客户用例*、*查找商品用例*、*查找促销品用例*等。对于这类用例，可能只由一个开发组建立一种通用搜索机制，而由其他人员进行调用。

编写6个大致相同的非正式用例也许没什么问题。但无论怎样，详细描述6个相似的用例都是一件非常烦琐的工作，而且用例编写人员不必花太多时间就可以找到一种简化的办法。下面通过第5章中的用例23“*搜索……(问题描述)*”来阐述这种方法。

通过这个用例，能观察到：

- 用例每个步骤目标的命名类似于编程语言中的子过程调用。
- 用例是由人而不是由计算机使用。

另外，应注意到，搜索任何东西都必须采用相同的逻辑步骤：

1) 用户指定待搜索对象。

2) 系统搜索，产生可能匹配的对象列表。

3) 用户选择，或重新排序结果列表，或重新搜索。

4) 系统找到(或没有找到)对象。

各种搜索的不同之处在于：

- 被搜索对象的名字。
- 被搜索对象的可搜索属性（搜索域）。
- 怎样显示被搜索对象（按顺序显示对象值）。
- 如何选择排序机制（排序标准）。

150

我们可以创建一个参数化用例，为了便于操作，可以给每个参数起一个别名。在此，我们

选择调用“搜索……”这个用例。为了便于读者理解，在这里做一点提示，记账员搜索客户其实就是调用“搜索某样东西”这个用例对客户进行搜索。当然聪明的读者跳过这点提示，理解也不会有问题。

参数化子用例与参数化用例一样，它们的所有别名都必需包括同样的属性，即搜索域、显示值和排序标准。必要时，还应该指明编写者应详细说明哪些属性。

对于别名数据值，我们定义三个精度级别，第一精度级别通常是用例文本中的描述，即数据别名，例如“客户信息”；第二精度级别描述与数据别名相联系的域列表，通过它们列出所有收集到的信息，如名字、地址、白天电话、夜间电话等；第三精度级别对各属性进行详细的定义，包括属性长度、有效性检验标准等。

对于这三类精度级别，只把第一类精度级别的数据放入用例。对数据的描述、搜索、排序标准，则放入另一单独页中，在用例步骤中用超文本连接指示。

这样处理后，上面例子看起来如下：

记账员使用客户详细情况来搜索客户。

“客户详细情况”页指示搜索域、顺序显示域和排序标准等。用例使用者只需单击它就可以了解相关的详细情况。通过这种机制，能让用例的读者、编写者以及软件开发人员非常容易且快速地相互理解。

到此，“搜索……”用例看起来像下面的样子：

1) 用户指出对象搜索特征域。
2) 系统查找所有匹配对象并在列表中显示相关的显示值。
3) 用户使用排序标准对结果进行重新排序。
4) 用户选择感兴趣对象。

通过使用这种技巧，调用的用例没有被搜索、排序、重排等（低层）细节操作所扰乱，仍保持整齐有序。公共的搜索行为被局部化，并且只需编写一次。对所有需要搜索服务的人员，其过程是一致的，对于那些为了编写程序而需要知道细节的人员，也能够找到详细的说明。事实上，对软件开发小组而言，他们非常乐意只拿到一个搜索机制的规格需求说明，这样他们也不必去考虑所有搜查本质上是否一致。

到此为止，对完成第5章练习5-4已给出足够提示。特别要注意其保证条件和扩展点，因为这两点对调用者是非常重要的。

151

第15章
业务过程建模

本书所讲内容既可用于软件系统设计，也可用于业务过程设计。对于任何系统，包括向外界用户提供服务同时保护其他用户利益的业务系统，用例都可以进行描述。文本用例的可读性特别有利于业务建模。本书中业务用例的例子如下：

- 用例2，“汽车交通事故索赔”。
- 用例5，“买东西（完整正式版本）”。
- 用例18，“操作保险单”。
- 用例19，“处理申请（业务）”。
- 用例20，“评估工作补偿申请”。

15.1 建模与设计

我们经常说：“使用用例对业务过程进行重组”，其可能意味着：

“在重组前，通过用例对其原系统文档化。”

“通过用例创建符合设计要求的外部行为需求。”

“重新设计后，使用用例对新过程文档化。”

事实上，所有这些含义都应该是对的，并且都值得关注。读者可以按自己的意愿去理解其中的一个。

但是通常在谈论用例时，我总是说业务过程建模或文档化，而不是业务过程重组或设计。因为用例仅仅是对过程文档化，不能代表过程重组或设计。在创造设计时，设计者需要经历一个思维跳跃的过程，但用例不能告诉他们怎样去做。通常，每个层次文档所描述的是下个层次设计必须满足的行为需求(实际上，我们却说“这次设计满足这些行为需求”)。

引入新技术经常会改变业务过程。它们分别是从面向技术的核心业务，从新业务过程到技术，以及从技术直接驱动，对业务过程进行重组。这些方法中的任何一种方法都是可行的。

15.1.1 从核心业务

在这种自上而下的分析方法中，正如“*Reengineering the Corporation*”一书（Hammer，1984）中所描述的那样，首先应该仔细识别组织中的核心业务。在这步工作结束时，应该弄清楚：

- 在组织行为中的项目相关人员。
- 该组织必须满足其需求的外部主执行者。
- 该组织必须响应的触发事件。

• 该组织提供的服务，以及对项目相关人员的成功结果。

注意，上面的内容并没有指出该组织如何工作，而只有设定其行为边界条件的信息。通常，它们也是用例的边界信息：项目相关人员与利益、主执行者与目标、以及成功保证。

业务过程设计可以采用业务黑盒用例进行描述，公司或组织作为设计中的一个系统（如图15-1所示）。

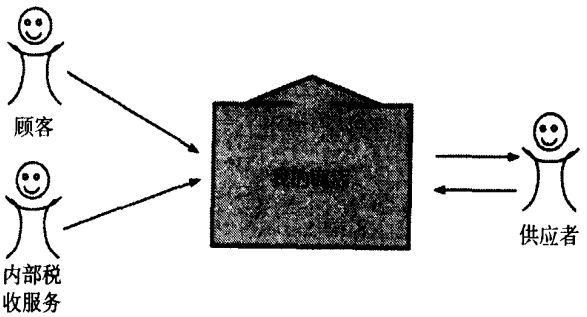


图15-1　核心业务黑盒

在这个阶段，可以充分利用当前的新技术，创建新资源组和新过程。当今社会，计算机系统成为公司主要的动态存储和动态通信渠道。在“*Reengineering the Corporation*”这本书中，给出了许多不同改革行动导致不同业务设计的例子，当然它们的效率也不一样。结果是产生一个新的公司或组织设计（如图15-2所示）。

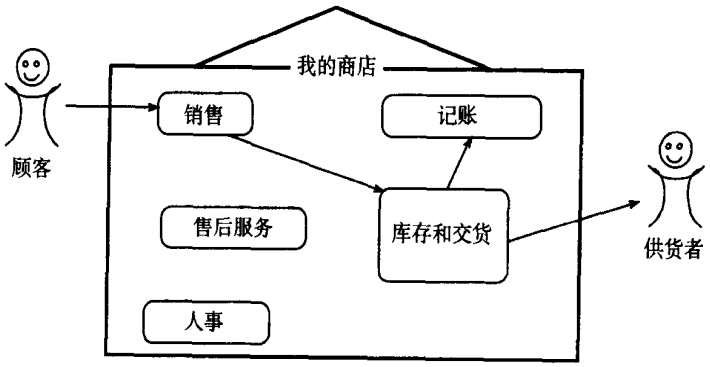


图15-2　白盒用例中的新业务设计

过程革新的结果，就是用白盒用例对系统文档化。这些用例展示了人与各部门（也可能是计算机）之间的交互作用，以及这些作用所产生的组织外部可见行为。

另外，开发真正完备的白盒用例，应该与任何完整用例集或业务过程模型一样，考虑系统失败及异常处理等情况。当然，在此过程中可以选择使用新技术，或不使用新技术来满足系统需求。

15.1.2　从业务过程到技术

这是一种比较折衷的观点，在建模过程中，保持系统核心目的不变，但可以根据实际需要，

重新定义适合新技术的业务过程。可以采用一些新技术，如新软件应用程序或移动设备等。同时需要为技术革新指定边界条件。

如图15-3所示，在建模过程中，用白盒用例来描述系统，它包含新拟定的业务过程，但不具体指明所用的新技术。在这种情况下指明所采用的新技术，就像在一个系统用例中描述用户接口一样，是不妥当的。参考第5章中的用例21“处理申请（系统）”。

154 ~ 155

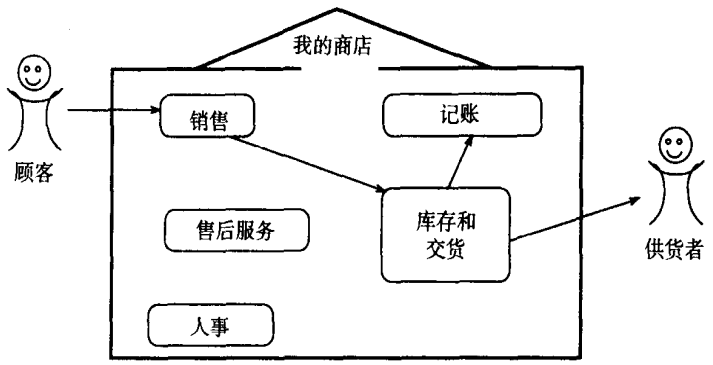


图15-3 白盒用例中的新业务设计（又一次）

原理上，计算机能做的工作，人同样能够完成。但是建模小组应该弄清楚，现有的工具（例如功能齐全的计算机或一批掌上电脑和收音机）是如何改善业务过程的。

到此，设计者知道他们的革新应该支持哪些过程。革新完成后，他们应该把表示新系统的黑盒用例融入到没有提及技术的白盒用例中。这些系统用例即是系统设计的需求，如图15-4所示。

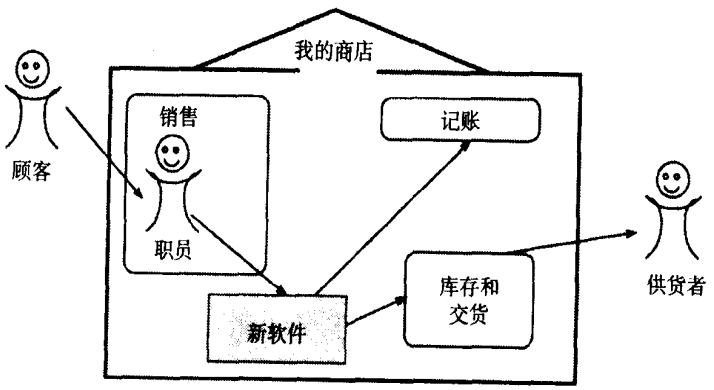


图15-4 黑盒系统用例中的新业务过程

到此为止，建模过程在纸面上看起来很完美，但它花费了许多金钱和时间。由于技术进步越来越快，压力越来越大，通常没有时间以这种方式对业务过程建模。但是应该注意到，应用专家（即在某领域是专家又从事业务工作的人）能够在头脑中对新业务建模，从而节省时间和金钱。下面将讨论这种建模方法，即第三种建模方法。

156

15.1.3 从技术到业务过程

首先，召集一批应用专家，他们可能热心于改进团队的技术和工作习惯。并确保在系统涉及的每个业务领域都能找到两个代表。

然后开始进行协商，技术专家将指出系统那些能力会改善业务过程。通常，与开发小组相比，他们会提出更多建议（这正好拓宽了技术人员的视野）。

让应用专家按他们所设想的系统去设计黑盒用例。这些用例不考虑用户界面，只描述系统应该做什么，即尽可能有效满足业务过程中主执行者的需求。同时，扩展到那些将会涉及的、各种关键的或很少讨论到的业务规则。应用专家必须和同事一起讨论，弄清楚系统行为的细节。另外，在编写过程中，他们也应该编写一些概要级用例和业务用例来展示整个需求之间的联系。

换句话说，作为系统需求实践的一部分，应产生一份简略的业务过程文档。讨论主执行者同新系统在不同环境下怎样进行交互的同时，应用专家已经在他们头脑中对新系统进行了建模。实践表明，这是一种非常行之有效的方法。

- 用例3“对运到的包装箱进行登记”：说明了系统行为文档怎样结束对业务过程细节的描述，怎样完成异常条件处理。
- 用例21“处理申请（系统）”：使用概要（风筝级）用例展示系统的业务过程语境。

15.2 连接业务用例和系统用例

业务用例与系统用例具有相同的特征，因此编写和评审用例的方法对两者都适用。在业务用例中说明的东西，也会在系统用例中说明。这形成了系统用例和用户用例之间的合作。但这
157 样带来了两个坏消息。

第一：编写者和读者经常把二者弄混，可能把系统行为放入业务用例中，也可能把业务操作归于系统用例。如果能够商量着去做将会有所帮助。但通常编写者和读者不会认识到这样做的重要性。使用系统用例的读者批评业务用例所处层次太高，但却没有认识到提供系统详细的行为细节不是业务用例应该做的；业务用例编写者偶尔把系统行为细节写入其中，结果导致业务主管对这类有详细细节行为的文档失去了兴趣。

为了减少这种混乱，应该在用例模板中写明用例范围及级别，让用例编写者依据此规则编写，同时让读者了解这些规则。如果能够，尽量对用例使用图标。对两者使用不同的模型和用完全不同的数字进行编号(如一组从1000开始对用例编号，另一组从1开始对用例编号)。同时，编写一些可以直接使用和可视化的构件。这样就可以既充分利用这种合作关系，又不会让人混淆。

第二：完全且正确地连接系统和用户用例不太值得。通常，业务用例编写者应对业务过程到系统使用（通常没有描述）进行描述。而在描述日常生活中客户如何使用新系统之前，用例编写者已经花光时间、财力、精力以及热情。而系统用例编写者有时为了保持一致，会在业务过程中加一两句，但是他们通常不愿意重写一个包含新系统功能的业务用例。

这样就在系统用例和业务用例之间形成了空隙，即系统用例和业务用例之间不一致。FirePond公司的Rusty Walters对此评论如下：

在完整的业务用例展开为系统用例方面，我有一些成功的经验。根据我的经验，通常把业务用例分为三个级别：开始是少数几个黑盒、云朵级业务用例；然后很快转换为白盒、云朵级用例；最后展开为白盒、风筝级业务用例。

然而，在此过程中，看不到业务用例和系统用例之间清晰的连接。

企图寻找从业务过程到系统需求的自动映射方法是不明智的。就如在15.1节“建模与设计”中描述的一样，我认为自动派生系统需求是不可能的。

一些人经历过其中的麻烦，但我没有。在公司里，我认识的大部分人，一旦他们知道怎么去做，就能跳过上述人为的空隙，而将低层业务用例和概要层系统用例联系起来。但是，我必须指出，花必要的人力和物力编写将业务用例和系统用例完全连接起来的最终用例集还是物有所值的。可以对两种方法进行预算跟踪，在目标满足时，活动就结束。 158

重新考察第5章中的用例19“赔偿处理（系统）用例”。系统用例真正从业务用例中提出，但是，与开始于单独业务过程重组活动不一样，专门编写它们是为了向系统需求组提供相应语境。

在下面论述中，RirePond公司的Rusty Walters阐述了他在业务过程用例方面的经验：

◆ Rusty Walters：业务建模和系统需求

受益于早先曾经阅读过你的书，我通过以前的尝试能够对问题进行合理规划，并且能利用新的知识。

阅读前的经历：

在阅读这本书之前，我从事过产品组中几个应用程序功能需求文档的编写工作。

在一个应用程序开发中，我们编写各个层次的系统用例，包括概要级、用户级和子功能级。我们主要集中在系统功能方面。对建模后的结果也非常满意，这个结果非常易于理解。同时，我们也觉得没有开发业务用例来展示整个语境的必要，系统概要级用例就包括了我们全部的需求。

在另一个应用程序开发中，虽然还是同样的开发组负责用例建模，情况却完全不同。回顾起来，我明白问题症结在于，不同组的成员从不同角度接近系统。我从业务过程到技术进行建模，有的人却从技术到业务过程进行建模。毫无疑问，在两组设计人员中，每个用例的设计范围不清晰。

在从业务过程到技术进行建模的组中，他们从不编写系统用例；而在从技术到业务进行建模的组中，他们也从不编写业务用例。相反，由于两组都希望直接利用对方编写的用例，因此难免会发生正面冲突和相互指责。由于当时对界定用例范围层次没有远见并且缺少必要的理解，用例模型变得相当混乱。直到最后，整个小组对用例模型仍不满意。事实上，整个小组都知道这样有问题，但却不知道症结在哪里。

阅读后的经历：

正如我在一个试图理解和文档化开发过程的小组中发现的，从核心业务到技术进行建模似乎不会导致什么混乱。

159 我们一起讨论业务过程及其内部工作方式(不包括软、硬件系统)时，每一个人都很清楚。而混乱出现的区域确实与业务和部门的范围有关系。

我们从业务中很上层的黑盒（云朵级）用例开始，大家对这些用例都很清楚，甚至包括那些从事底层建模的人。然后，如本书中所说的一样，很快写出白盒（云朵级）概要用例。当我们决定讨论下一级用例时，设计域（即我们是讨论整个业务，还是讨论某个部门）引发的混乱出现了。这个问题也与如何创建后续用例有关。在一个特殊的例子中，当我们认识到那些应该在调用用例中实现、而不应该作为当前用例的一部分后，删除了上面两个步骤。同时开发组还不打算把业务用例展开为系统用例需求。

虽然在会议期间很难注意和修正整个过程，但会后，对问题域的理解会相对容易一些。在文档化会议结果的过程中，我使用设计域语境图，用图标明每个用例的领域和层次。如
160 果图足够简单，就会给读者留下深刻印象，并直接浮现在读者脑海中。

第16章
遗漏的需求

“编写执行者的目的，是仅用别名来表达需要传递的数据细节”，就如客户信息用名字和地址表示一样，这固然是一个好的建议。然而，对程序员来说，这没有提供软件开发所必需的详细信息。程序设计人员和用户界面设计者需要准确地知道地址意味着什么，地址包含哪些域，每个域的长度，以及地址、传真号、电话号码的验证规则，等等。所有这些信息包含在需求的其他部分中，但不在用例中表示。

用例只是需求文档的“第3章”——行为需求，它们不包括系统性能需求、业务规则、用户界面设计、数据描述、有限状态机行为、优先级以及其他相关信息。

系统开发人员通常会问，“这些需求放在哪里呢？”。虽然用例没有包含它们，但它们肯定以文档形式存在于某个地方。

事实上，在这些信息中，下列信息可以作为用例相关信息附在用例上：

- 用例优先级
- 期望的发生频率
- 性能要求
- 交付日期
- 次要执行者
- 业务规则（可能）
- 未解决的问题

可以根据不同项目调整这张表的内容，包含该项目认为重要的方面。

在许多情况下，一张简单电子表或表单就能很好地捕获相关信息。许多人在项目开始处，用一张电子表格来概括用例信息全貌，并在表格最左端标明用例名，然后在其他各列中填充如下信息：

- 主执行者
- 触发条件
- 交付优先级
- 复杂度评估
- 可能的版本
- 性能要求
- 完成状态
- 其他

这里还没考虑软件开发人员所需要的数据需求，这和系统行为需求一样重要，其中包括要执行的域校检等。

16.1 数据需求的精度

与其他工作一样，利用“精度”级别对数据需求进行分类，同样有利于合理安排你的精力（见1.5节“合理分配你的精力”），通常把数据需求分为三个精度级别：

- 信息别名
- 域列表或数据描述
- 域的细节和域校检

信息别名

我们书写“记账员收集客户信息”，或“记账员收集客户的名字、地址和电话号码”的时候，同时期望对*名字*、*地址*和*电话号码*在其他地方做进一步扩展。

别名适合放在用例中。如果在用例中对此描述得过多会减慢需求收集的速度，会使用例变长并难以阅读，同时还不便于对数据需求做进一步修改（也就是说，数据需求的改变变得非常敏感）。另外，还可能导致许多其他用例引用同样的信息别名。

基于以上原因，可以通过许多工具中的超文本链接或支持编号交叉链接的工具，把用例和
162 数据需求细节进行分离，并建立从用例到相关*域列表*的链接。

域列表

在某种程度上，需求编写者不得不对别名表达的每层意思进行讨论，如客户姓名由两部分（即姓和名）组成，还是由三部分组成（或更多）呢？世界上有许多地址表示格式，那么地址真正需要表示的是什么呢？对编写者来说，可以从这里将数据描述扩展到第二级精度。这项工作可以和用例编写同时进行，也可以在用例写完后进行，甚至可以与用户接口设计者一同进行。

对第二级精度的数据进行处理，可以采用许多策略和方法。这里阐述两种，其余可参考“*Software for Use*”（Constantine和Lockwood，1999）和“*GUIs with Glue*”(Hohmann)，当然也可以使用自己在这个领域内的经验。

- 第一种方法是，为每一个别名保留一个单独条目，如客户姓名有三个域：名字、中间名和姓。随着时间推移，用域的细节及相关检验条件对这个条目不断进行细化，正如下节“*域的细节和校验*”中描述的那样，直到包含了这些域的所有细节。
- 第二种方法就是把名字、地址和电话号码一起写在单一用例上。这样做意味着，对这三部分信息统一输入和显示是非常重要的。同时这对用户接口设计者也非常有用，他们能够设计子屏幕或多个域来把这些信息一齐显示在不同地方。因此，需要在工具中为“名字、地址和电话号码”创建一个单独条目，然后列出名字、地址以及电话号码所需的域，但不对列表做进一步扩展。

这两种方法的不同之处在于，第二种方法中把别名信息放在一个域列表中，当需要对信息做进一步细化时，不能在原条目中进行扩展，而必须为每一域重新创建单独条目。

无论选择哪一种方法，都期望第二级精度信息随着项目向前推进和开发组对数据需求进一步了解而改变。可以让不同的人定义数据的第二和第三精度级别，甚至可以手工让第二级精度数据与用例本身分开。

域的细节和域校验

对于程序设计人员和数据库设计者而言，他们需要的是“客户名字由多少个字符组成？”，“对数据丢失采取什么限制措施”等这类问题的答案。要回答这些问题，需要域类型、域长度及域校验等相关信息。

一些项目组把这些问题叫做数据需求或外部数据格式。一些人则用超文本链接方式或数据 163
库，把它们放入根据域定义分类的单独条目中；还有一些人则把用户界面数据的细节直接写入接口需求和设计文档中。

无论怎么做，我们都应该注意到：

- 必须把域的细节和域校验放入第三级精度中。
- 用例不是进行扩展的地方。
- 相关信息必须与用例建立连接。
- 域的详细说明可能随时间改变，应独立于用例详细说明。

16.2 从用例到其他需求的交叉链接

数据格式（即数据详细说明）不是用例的一部分，但是用例指明了数据的需求，因此能够在用例和数据描述之间建立超链接；同样，复杂的业务规则也不适合写入用例，但却可以在包含它们的实体之间建立一个连接。如图16-1所示，这种车轮状连接使得用例处于需求文档的中心，而其他非功能性需求处于其四周。

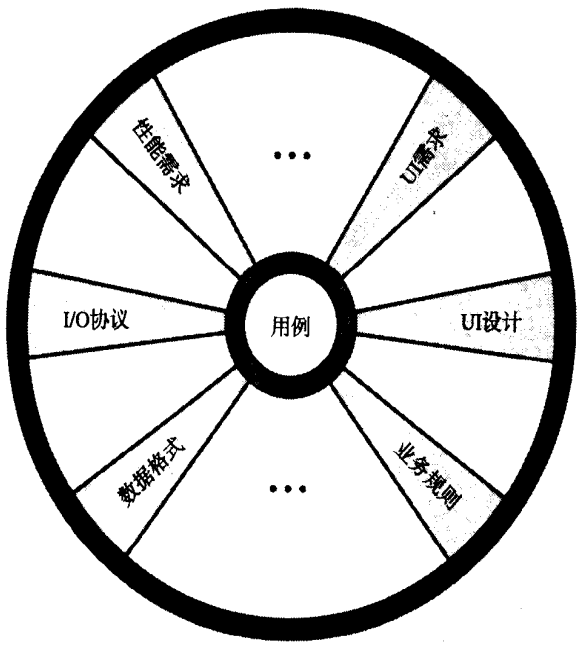


图16-1 翻新图1-1，“轮轴和轮辐”需求模型 164

注意不要把不符合用例格式的需求放入用例中，用例最适合于捕获执行者之间的交互行为。一次偶然机会，我听到有人抱怨，通过用例很难对磁带合并操作或编译器的需求进行描述，对此我表示完全赞同，当然对这样的需求最好用代数或表格方式进行描述。

只有部分需求适于用例形式描述。但令人高兴的是，执行者相互作用这部分需求是系统的
165 核心，并能和许多其他需求建立连接。

第17章
用例在整个过程中的作用

17.1 用例在项目组织中的作用

用例为管理者指明应提交给用户的系统功能。用例的标题指明主执行者的需求，同时系统也必须支持这些需求，而用例描述则说明系统需要什么功能以及将提供什么服务。

17.1.1 通过用例标题进行组织

在早期项目开发中，通常创建一张用例表，表格中的第一、二栏分别写上用例名和用例主执行者，第三栏由投资方填写每个业务过程用例的优先级，第四栏写上由开发组评估的实现功能的困难和复杂度。这种方法体现了执行者目标列表自然演变的过程（参见25页“执行者目标列表”小节）。

Kent Beck在其“规划游戏”（*Extreme Programming Explained: Embrace Change*，1999）中，对此提出了较好的办法。首先，由投资方对每个用例优先级进行评估，由开发组估算开发费用。这样投资方看到开发费用后，就可以对优先级进行调整，当然也可以不进行调整。按照这种思想，可能会两次填写规划表。

根据需要，可以对规划表增加其他栏目，如用例开发优先级、用例版本号以及开发小组等等，如表17-1所示：

表17-1 规划表的示例

执行者	任务级目标	业务需求	技术难度	优先级	UC#
普通	检验请求	最高	工作量大	1	2
授权者	改变授权	高	简单	2	3
买主	改变卖主内容	中等	简单	3	4
请求者	初始一个请求	最高	中等	1	5
	改变请求	最高	简单	1	6
	取消请求	低	简单	4	7
	标记提交请求	低	简单	4	8
	拒绝提交商品	低	未知	4	9
批准者	批准完整的请求	高	复杂	2	10
买主	请求订购	最高	复杂	1	11
	初始化供应商PO	最高	复杂	1	12

（续）

执行者	任务级目标	业务需求	技术难度	优先级	UC#
	没交货警告	低	中等	4	13
授权者	验证供应者的签名	中等	难	3	14
接收者	登记交货	最高	中等	1	15

在整个项目开发过程中，可以非常容易地对这张表进行审核和操作。另外，可以对每个用例进行评估，指定用例开发组，以及对每个用例、每个版本的工作进行跟踪。

下面是一个真实的小例子，它采用规划表进行规划、评估、划分优先级，并尽可能减少用例集。采用这种方式工作的好处如下：

- 用例表清晰显示系统对整个业务的价值。
- 用例名列表为基于优先级的工作和时间表提供了依据。

168

◆ 一个真实的小故事

开发人员接到一个任务：决定软件下一个版本需要支持哪些业务过程。她要面对7页长的执行者目标列表，并要对其价值、复杂度、触发条件等进行评估。她与执行主管一起，将其整理为半页需求。然后写出这些业务过程主成功场景，并与执行主管一起从大约半页系统级目标出发，整理出执行步骤列表。得到这些信息后，她再与各部门接触。然后，她得到了一个清晰的、能最大限度满足各部门人员需求的业务过程图。她和主管一起指定了4个在以后6个月内开发的用例。

17.1.2　跨版本处理用例

一般来说，软件每一个版本对应一套完整用例集，但也有例外。例如“订购产品”用例就涉及各种各样特殊处理，并且它们的实现跨越整个过程。一个典型处理过程是：

- 在版本1中实现简单用例。
- 在版本2中增加高风险处理。
- 在版本3中增加客户推荐处理。

对上述过程，可以采用两种方法：编写一个总用例，在每个版本中实现其相关部分；或者编写三个用例。两种方法都可以使用，但各自都有缺陷。

一些小组把用例分成多个单元，每个单元可以在一个版本中完全实现，例如他们首先编写基本的产品订购用例，然后编写高风险处理的产品订购用例，最后编写增加客户推荐处理的产品订购用例。他们或者重复编写用例描述，用楷体/斜体[⊖]表示新增内容；或者在第一个用例基础上扩充编写第二个用例，在第二个用例基础上扩充编写第三个用例。分拆用例简化了用例跟

⊖ 如果翻译成中文，采用楷体表示新增内容；如果保持英文原文，采用斜体表示新增内容。——编辑注

踪；但跟踪用例集却变为原用例集的三倍，同时扩充用例的方法使用例编写变得更复杂。

另一些人（比如我本人）可能更在意用例的可读性，因此他们可以容忍在一段时间内只实现用例的某一部分。可以使用高亮方式（黄色或楷体/斜体[⊖]）表明先实现的部分，请参考“用例5被表明的部分”。许多项目都以这种方式进行开发，并且运行得还可以。尽管如此，这种方法并不像我们想像得那么好。 169

一种可行的折衷方案是开始就编写完整的、多个版本的用例，然后在需要修订时在用例上写版本号，并将在这次修订中计划增加的功能分离出来。下一次增补时，对用例中已实现的部分用普通文本方式书写，用**黑体**书写将要增加的部分。采用这种策略，用例集数目仍然成倍增长，但却容易控制。如果开始时有80个用例，那么开发组的感觉就好像是，整个用例集为原来80个用例加上许多在某些方面功能有所增强的用例之和。但是用例集的增加，是以一种局部化和易于控制的方式进行。

每种方法都有缺陷，那么到底应该采用哪种方法工作呢？我们注意到所有方法都预先假定系统组织和实现采用增量式方法，在现代软件开发项目中，增量式开发逐渐成为了标准。详细讨论请参见《*Surviving Object Oriented Projects*》（1998）和论文“*VM Staging*”（http://members.aol.com/acockburn/papers/vwstage.htm）。

17.1.3 交付完整场景

◆ 一个短而真实的集成实例

一个大项目的集成测试经理向我请教增量式开发中可能发生什么危害。由于开发组基于系统特性进行开发，而不是通过用例。因为没有程序所支持的“故事”，而仅仅有一堆机制，所以测试组无法对提交的程序进行测试。基于这种情况，项目领导最终决定重写项目计划，以便有一个完整的“故事”贯穿增量式开发工作。

经常碰到这种情况，在同一时间不是所有用例都能交付。但是，每次交付都必须包括一个完整场景。用例说明了用户需要实现什么，并列出了所涉及的很多场景。交付的软件必须包含其中一些完整的场景，否则就不要交付这些功能。

交付的规划和设计必须与最终用户使用的功能集保持一致，这些功能应是来自用例的完整场景。同时，功能测试者要对用例兼容性进行测试。只有在一个完整可用的用例场景中才能按照上述过程进行实施。 170

17.2 从用例到任务或特征列表

一个保持良好交流的开发组，可以通过用例制定软件设计任务。如果不这样，即使是小的

⊖ 如果翻译成中文，采用楷体表示先实现的部分；如果保持英文原义，采用斜体表示先实现的部分。——编辑注

项目管理也很难保证用例到设计任务的映射，以及保持映射的及时性。

下面来自我同事的电子邮件说明了这个过程。

> 最近两周我们两人拜访了客户，提交了需求，并和开发组建立了工作关系。我们一直把精力集中在用例上，因此我们确信，用例对设计者和开发者已足够精确。另外，这里有份描述系统特征是否属于某个范围的域文档，该文档与你提供的域文档样例很类似。原文非常小，但是我们总是收到索要“传统”需求文档的请求。因此我们对其进行了少量扩充，以便保持其对细节的描述最少。
>
> 在我们和软件设计人员的第一次会议中，他们想要评审“需求”。他们所指的“需求”是域文档中的描述。我们弄清楚了软件设计人员是如何被这份文档所困扰的，由于我们希望以用例为中心，因此该文档缺少一些应有的细节。随后，我们又花了三天时间重新修订域文档，你将在附件中看到这份文档。我希望它能作为用例的补充。如果希望对软件开发文化有更深的影响，并帮助公司完成从传统需求到基于用例需求的转变，我们必须使用用例名作为一个特征集，并把场景中每一步作为一个特征。而我们现在是逐字逐句地从用例文本中拷贝每一步，并把它们放入域文档中相应用例的标题下。
>
> 我们面临的、最终导致做双倍维护工作的问题是：从用例场景中拷贝出的文本不能很好地代表某些内在特征。尽管我们可以从场景中拷贝文本，但也经常需要做少量修改或增加一些内容。

这仅仅是开始。软件设计人员需要从这些有编号的需求段落或特征列表开始工作，但这些需求段落或特征列表仅仅是简单的描述。在另一个类似的故事中，首先编写了“详细需求文档”或有编号段落的文档，然后以它们为基础编写用例。（这好像又回到了以前的过程，但情况的确如此）。

在用例中描述的系统行为流程和单个软件设计者的设计任务之间存在一个根本矛盾。设计者对用例中单个特征或单个条目进行设计，也可能在多个条目之间交互工作。完成的工作可能
171 是“撤消机制”、“事务日志机制”或者“屏幕搜索框架”等。由于他们不参与系统行为流程，因此无法使用用例语言来描述他们的工作。他们顶多能说“我正在进行用例5中描述的屏幕搜索部分的工作”。

同时，软件投资方希望从整个流程的角度来观察应用软件，并希望能提交给用户一个有价值的东西。他们对单个设计任务不感兴趣。

保持用例文档和设计任务列表的同步，是非常耗时和辛苦的事情，像我的同事上面描述的那些工作在一个项目中可能要反复多次。当然，我们应尽量避免这种重复工作。到目前为止，对于不超过50人的项目，我通常不把用例分成多个条目。这也许是由于我在项目中特别强调人员交流与合作，以及我有过类似的经历。我们能够在头脑中对用例进行划分，或者用黄色标记出来，而后不用耗费太多精力就能把关键的用例组成任务列表，进行调度。

另一个替代方法是同时产生两份文档，并努力保持对其同步更新。如果采用这种策略，就需要把用例文本分成多个部分，每部分指派给单个开发组或单个开发人员。这样，每部分都将是一个能被指派、跟踪及检验的程序特征、机制或设计任务。对整个软件开发的详细评估是所

有设计任务评估之和，项目跟踪也由每个设计任务的开始和完成所组成。

下面是一个用例到工作列表转换的例子。

用例34 获得折扣

语境目的：购物者有一个装商品的购物车，当向其中加入折扣时，了解这会怎样影响车内商品的价格。

范围：商业软件系统

级别：子功能级

前置条件：购物车内必须有商品。

成功保证：折扣有价值，加入到购物车中，减少了车中商品价格。

最小保证：如果没完成，没获得折扣，或没加入到购物车中。

主执行者：购物者（任何网上用户）

触发条件：顾客选择了折扣。

主成功场景：

1. 购物者选择了折扣。
2. 通过向顾客显示折扣价值信息并询问一系列问题，系统决定折扣的价值。显示的信息和问题取决于购物者对前面问题的回答。为了提高折扣业务上的可行性，问题和信息路径预先给定。 172
3. 系统记载导购信息和折扣信息。
4. 购物者重新查看商品及其价格，考虑是否购买。
5. 购物者把折扣放入购物车中。
6. 系统把导购信息和折扣信息加入购物车中。
7. 系统显示购物车中所有商品和折扣，同时根据折扣重新计算总价格。

购物者可以根据需要多次重复上述过程来获得不同价值的折扣，并根据意愿将其放入购物车中。

扩展：

2a. 在折扣放入购物车之前的任意时刻，顾客都可以返回修改前面问题的答案。

5a. 即使购物者没有把折扣放入购物车中，系统也保留导购信息。

5b. 购物者如果想让折扣用于购物车中一种特定商品，那么购物者应该在购物车中指定一种可以打折的商品。

表17-2是由此产生的工作列表。

表17-2 “获得折扣”任务列表

索 引	特 征	业务需求	版 本
EC10	获得折扣	必需	1.0
EC10.1	提供让购物者把折扣放入车中的功能。	必需	1.0

（续）

索　引	特　征	业务需求	版　本
EC10.2	通过UI形式，提供显示和引导顾客收集折扣信息，并确定其价值的功能。	必需	1.0
EC10.3	提供通过外部折扣系统（站点）来决定折扣价值的功能。顾客相关的折扣信息可以传到外部系统，外部系统估算其价值，并返回该值和其他重要特性。	必需	1.0
EC10.4	提供一种显示折扣概貌（包括其价值）的功能。	必需	1.0
EC10.5	为顾客提供增加和放弃折扣的功能。当增加折扣到购物车时，顾客能够让它与购物车中单个商品或所有商品联系在一起。	必需	1.0
EC10.6	提供把折扣考虑在内，重新计算购物车中所有商品价格的功能。	必需	1.0
EC10.7	提供通过返回折扣提问/回答处理过程来重新编辑折扣的功能。	必需	1.0
EC10.8	提供从购物车中删除折扣，并重新计算商品总价的能力。	必需	1.0
EC10.9	通过预置触发机制，提供记录折扣信息和步骤的功能。	必需	1.0

173

17.3　从用例到设计

用例仅提供了设计所需要的所有黑盒行为需求。这些需求仅描述系统行为，而不对设计者进行任何限制。这些只是帮助设计者利用他们掌握的技巧，完成符合系统需求的“好”设计。需求与设计都不是要去满足用例。

从用例到设计的转变有很多方法，既有好的方法，也有不好的方法。不好的方法如下：

- 设计不通过用例分组。
- 盲目依靠用例，导致功能分解了设计（面向对象的工作组和构件设计组对这点最为关心）。

好的方法如下：

- 在一些设计技术中利用各种场景。
- 利用用例命名域建模中所需的概念。

174

首先让我们看看不好的转换。

设计不通过用例分组

设计任务不与用例单元整齐地对应。这样，一项设计任务产生的业务对象或者行为机制可能同时用于多个用例。后期实现的用例可能包含一些重要信息，这些重要信息涉及早期实现的任务设计。当设计者在后面工作中遇到这些信息时，就意味着需要改变后期设计。

有三种办法处理这种情况。第一种办法，让设计者浏览所有用例，寻找可能涉及他们设计任务的关键信息。对于小项目，一般能做到。谁管理项目，谁就要预先了解这些信息。

第二种办法，让设计者浏览所有用例，寻找可能对设计有重大影响的“高风险”条目和关键功能。然后创建满足这些关键功能的设计，并尽可能保证其他功能对设计不会有太大影响。

第三种办法，也是我比较欣赏的方法，即了解软件在其生命周期上的缓慢改变过程。开发组尽可能使设计的每个版本都可以实际运行。这样，开发组会了解到，导致设计改变的新需求将在下一个阶段的什么时间出现。

当然，这种方式可能让某些开发人员感到不满意，特别是那些熟悉数据库设计的开发人员。在许多情况下，对一个表增加新域，并重新优化数据库，其代价非常昂贵。从经济角度来看，应该要求设计者一开始就识别出系统需要用到的所有属性。然后针对整套属性集，构造并交付各种实现级别的软件，如20%、40%直到100%完成。

然而，在现代采用增量式开发方法的情形下，对一个类或表增加某个属性，其工作非常少，并且代价很低，以至于开发者可以先实现一个类或实体急需的部分。这种开发方式的结果是，类和构件在任一时间只是“完成”一组给定功能集。而随着新功能的实现，“完成”的定义也会发生改变。

◆ 一个真实的小故事

这事发生在某个项目领导人身上，他抱怨，即使应用程序实现了顾客所有的需求，他也没有通过一个20%功能级的“完成”评估。我花了很长时间才发现，他是从传统数据库设计角度说这句话的，事实上，他们的项目却以增量方式进行开发。

以后还要对此进行讨论。如果没有大的经济风险，我建议开发组应按照“完成指定功能集”模型来开展工作。

175

用例与功能分解

如果使用结构化分解技术，对用例进行功能分解可能是有用的。但如果采用面向对象设计方法，则需要注意下面的内容。

17.3.1 面向对象设计者特别注意

采用功能分解或功能层次结构形成的用例集，展示了在行为需求通信方面的有效性，并且使编程变得易于理解，系统行为也可以清晰地传递到更高层。这种方法特别适合那些注重系统行为的开发人员。

这种功能分解可能有益于需求，但却不意味着对软件设计有利。数据和行为的封装已经表明，它有利于简化软件设计过程和软件维护。但却没有证据表明，它有利于需求收集和需求交流。根据我的经验，这种封装方法与基于功能结构的方法一样不好。换句话说，需求收集得益于功能分解，而软件设计得益于数据和行为的构件化。

设计人员应该阅读用例，并对此进行讨论和思考，然后进行有用的抽象。这是设计人员而不是用户的工作，

这样做的一个危害是：没有经验或粗心的设计人员根据对需求文档的功能分解来创建类，并简单地把每个用例设计为一个类、对象或构件。经验表明，这是一种不好的方法。许多OO专

家对此都持明确的反对意见。

一个解决方法是，不要把“用户目标”用例放入所属的类中，因为用户目标具有上下连贯的语义，捕获的是整个事务的处理过程。对这种问题，封装可能是一个解决办法。然而，子功能用例很少有这种特性，它们通常是属于不同类的部分功能。

对于OO设计人员，如果想要直接对域建模，而不关心软件所支持的功能，则存在相反的危害。这些设计人员忽略了功能需求的作用。而用例告诉了域建模人员应该考虑领域的哪些方面。缺少了这些信息，域建模人员可能在与系统无关的方面花费过多时间。用例为合理的域模型提供边界条件。这方面可参考“An Open Letter to Newcomers to OO”（*http://members.aol.com/humansandt/papers/oonewcomers.htm*）。

很明显，在任何情况下，用例都是按不同的组织模式来划分整个系统，而不是对象，因此在设计时，必须考虑这一点。

176 现在让我们看看好的消息。

利用场景设计

设计程序时，把用例看作身边的场景，这种方法特别适合于“职责驱动的设计”[⊖]，这是一种通过场景逐步推进的设计方法。可以显示设计什么时候完成，以及对各种场景的处理，因此用例也适用于其他设计技术。

利用用例命名域概念

用例特别强调所涉及的域对象的名字，考虑如下用例短语：

系统产生“发货单”，填充“发货单项”的价格，加上税费和运输费，计算商品总额。“发货单脚注”说明了交货条件。

看到这些短语（发货单、发货单项和发货单脚注）以及它们的属性（价格、税费、运输费和总额），毫不费力就会明白它们的含义。虽然这些不是最后设计所必需的，但是却是业务对象一个好的开始集。我发现许多项目组直接从用例概念得到设计草图，从此处着手就能很好地把握设计，并细化设计。

17.4　从用例到用户界面设计

Larrg Constantine 和Lucy Lockhood 在《*Software for Use*》中，以及Luke Hohmann在《*GUIs With Glue*》中，写了很多关于用户界面的设计，比我写的更好。然而，在用例编写期间，许多项目组经常会问，“怎样把与用户界面无关的用例转化为实际的用户界面设计呢？”

在开发组中，应指派有丰富经验的员工来创建用户友好的界面。他们首先要阅读用例，创立一个系统界面表示方式，该表示方式应保持用例的步骤，同时尽量减少用户的工作。这样，他们的用户界面设计才能满足用例给定的需求。该设计还应该由用户和程序设计人员进行审核。

⊖ 请阅读K.Beck和W.Cunningham的文章“A Laboratory for Object-Oriented Thinking”，*ACM SIGPLAN*，或者R. Wirfs-Brock、B. Wilkerson和L. Wiener的专著《*Designing Object-Oriented Software*》。也可以访问*http://c2.com/cgi/ wiki?CrcCard*或*http://members.aol.com/humansandt/papers/crc.htm*

用例编写人员发现这种方式非常有用，可以假定正在对屏幕输入数据或填充表格，然后去发现需要输入什么信息，以及输入数据时是否要受某种顺序限制。确保这些表示方式是作为应用专家审查任务的指示，而不是作为需求的一部分。

描述用户界面设计不是编写需求文档，但随着开发的进程，使用用户界面设计实例来扩展 177
需求文档是很有用处的。这些设计信息从文字（抽象的）和可视化（具体的）两方面解释系统的行为，因此增加了需求文档的可理解性。

用户界面设计可分为高、中、低三个精度级别：

- 用户界面的低精度描述是一种屏幕导航图，用状态图或有限状态机进行描述。每种状态是用户将会遇到的屏幕的名称，有限状态机显示了什么用户事件导致从一种屏幕状态向另一种屏幕状态转移。
- 中精度描述可看成一幅画或缩小的屏幕快照。把这些放到用例结尾处，以便读者能查阅指定的设计。
- 高精度描述列出每个屏幕所有的域类型、长度及有效性验证机制，它们不属于需求文档。

17.5 从用例到测试用例

用例为系统提供了现成的功能测试描述，大多数测试组肯定希望有机会使用用例进行测试工作。在开始阶段，用例对简化工作能够提供一些帮助。更进一步，在需求阶段用例就会为测试组提供一套测试方案。最好的情况是测试组也能够帮助编写用例。

在正规的开发组中，测试组应把用例分成多个测试集，并编写计划来确定触发不同路径的每个测试设置。然后，构造用于测试这些设置的测试用例。此外，还将设计不同数据来测试不同数据组合。最后还将设计系统执行性能和负载测试。最后两项测试任务不能直接从用例派生出来。

所有这些工作在通常情况下都是测试组的任务。表17-3和表17-4由Pete McBreen提供（http://www.mcbreen.ab.ca/papers/TestsFromUseCases.html）。开始部分为用例，其后为测试用例集。这个例子可以作为测试组工作参考，将它与自己的工作习惯对应起来。

注意，Pete采用项目相关人员与利益来帮助识别测试用例，以及测试用例怎样包含特定的测试值。

用例35 订购商品，产生发货单（测试用例）

语境：客户创建商品订单，系统产生发货单，并发送被订购商品。
最小保证：万一失败，商品不分配给客户，客户账户信息保持不变，该事务写入日志文件。
成功保证：商品分配给客户，产生发货单（使用客户发货规则），采购清单送到销售部。 178
主成功场景：
1. 用户选择商品及购买数量。
2. 系统分配客户要求的数量。

3. 系统得到经过确认的发货认证。
4. 客户指定发货目的地。
5. 系统发送采购指令到销售部。

扩展：

2a. 指定商品库存数量不够：
　2a1. 客户取消订单。
2b. 商品缺货：
　2b1. 客户取消订单。
3a. 客户信用差（连接到对这种异常情况的认证测试用例）。
4a. 无效的发货地址：？？

对于扩展中的每一项，最少应有一个测试用例。如果要完全覆盖各种情况，则需要更多测试用例对数据值进行测试。表17-3和表17-4中首先列出了主成功场景的测试用例，因为它能很好地显示系统在大多数情况下如何工作。通常，这些测试用例在其他已知扩展条件和恢复路径之前产生。

表17-3　主成功场景测试（好信用）

初始系统状态/输入	客户Pete有好的信用，订购单价为10美元的商品1一件，商品1库存数量为10件。
期望系统状态/输出	商品1库存数量为9，生成发送指令，产生发货单，记录该事务。

179

表17-4　主成功场景测试（坏信用）

初始系统状态/输入	客户Pete有坏的信用，订购单价为10美元的商品1一件，商品1库存数量为10件。
期望系统状态/输出	商品1库存数量为9，发送指令指示进行现金交易，记录该事务。

17.6　实际用例编写

每个开发组都应该形成并制定一套工作习惯。下面将介绍一种我比较欣赏的工作方式——分工合作过程，它是我最推崇的工作方式。Andy Kraus，后来在IBM公司工作，非常清晰地描述了他的开发组与大量不同的用户组协同工作的经验。相信你能从这份报告中得到一些有用的启示。

17.6.1　分工合作过程

开发组与个人相比，有两个优点：第一，便于集中讨论；第二，便于对研究的问题达成共识。然而分开工作却能编写更多文件。基于这个原因，我比较喜欢的过程是：当需要集中研讨或对问题达成共识时，让开发人员以整个组的方式工作，其余时间则一个或两个人一起工作。下面是整个过程，首先是大纲，然后是详细说明。

1) 制定一份粗略的系统功能图：

- 对系统采用的叙述方式达成共识（开发组）。

- 对应用领域达成共识，并集中讨论系统主执行者和系统目标（开发组）。
- 编写系统描述（个人）。
- 收集各种系统描述（开发组）。

2) 制定详细的用例视图。

- 集中研讨用例编写（开发组）。
- 对用例编写格式达成共识（开发组）。
- 编写用例（个人）。
- 审核用例（个人）。
- 审核用例（开发组）。

第1阶段：制定粗略的系统功能图 180

第1阶段分4个步骤完成。

第1步：对系统采用的叙述方式达成共识（开发组）。

整个开发组聚在一起弄清楚系统应该描述什么及怎样对系统进行描述（回顾1.6节“先用系统使用叙述做热身”）。首先，每个人写一份系统叙述，内容可能相同，也可能不同。然后，开发组阅读并讨论这些系统描述，对应该描述什么、怎样描述及其长度、以及哪些细节应包含、哪些细节不需要包含等达成一致。这可能需要花费数小时。最后，整个开发组对要构造的系统形成一个清晰认识。

第2步：对应用领域达成共识，并集中讨论系统主执行者和系统目标（开发组）。

开发组应花费足够时间找出系统总体目标、范围、主执行者。然后，建立一个直观的描述、一个输入输出列表、一个设计范围图、一个主执行者及项目相关人员列表以及一个最重要的初始用户目标集列表。这些条目都互相关联，因此讨论某项时可能会牵扯到对其他各项的理解。采用这种方式，可以在同一时间建立起所有条目。如果开发组认为他们已经弄清楚所要建立的系统，那么这可能会花几个小时到一天时间；如果他们还不知道，则可能需要数天来完成这项工作。最后，要对讨论域的范围、建立什么系统以及谁是关键执行者达成共识。

第3步：编写系统描述（个人）。

开发组分散工作，分别对系统所需功能写出使用描述。开发人员单独编写系统描述，然后和一个同伴进行交换，或在一个几人小组内传阅。最后把结果送到整个开发组。

第4步：收集各种系统描述（开发组）。

开发组共同讨论描述的内容。回答这个问题，“这是我们想要建立的系统吗？”。可能会对系统描述进行多次讨论，甚至重新编写系统描述，直到开发组成员都认为系统叙述正好描述了他们想要的系统。

到此为止，第1阶段工作完成，团队应向每个投资方发送一份系统叙述，它显示新系统草图（低精度），应该包括如下各项：

- 系统构想陈述
- 列出哪些在领域中和哪些不在领域中（包括功能和设计范围）
- 在执行环境中的系统草图 181
- 关键的主执行者列表

- 项目相关人员及其相关利益列表
- 最重要的用户目标列表
- 系统描述集（至少半页）

第2阶段：制定详细的用例视图

第2阶段分五个步骤完成。

第1步：集中研讨用例编写（开发组）。

首先提出一份需要编写的用例详细列表。然后，列出在系统生命周期中可能遇到的主执行者，以及所有能够想到的针对主执行者的用户目标。由开发组采用适当技术对其审核、集中讨论。当然也可以分组进行这步工作。

处理庞大的、不同种类开发组的一个有用技术是将它分成3到4人的工作小组。通常，系统有几个领域或兴趣组，每个组都需要用到相关知识，因此，工作小组应至少从每个领域组中抽取一人，使得每个工作小组都包含讨论所需的各方面知识，也便于小组成员相互了解。小组比大组行动迅速。同时这种分组方法使得工作小组的讨论能同时覆盖系统多个领域。

如果要采用分组的方法构造所有主执行者和用户目标列表，那么还需要把各个小组组织起来同时把各组的结果也汇集起来，作为整体重新对该列表是否能完成和是否能接受进行评审。最后，开发组将得到一份临时的、所有需要编写的用户目标用例集。当然，随着时间推移，还会发现新的用户目标。

开发组公布主执行者及用户目标的列表。与此同时，还可能对用例开发优先级、复杂度评估及开发时间等作出一些附加讨论。

第2步：对用例编写格式达成共识（开发组）。

在这一步中，整个开发组首先一起编写用例样本（或者每个人先编写，然后把每个版本放在一起综合）。开发组讨论用例编写级别和风格、用例模板、项目相关人员及其利益、最小保证等。到这步结束时，应该形成一个初步的用例编写标准。

第3步：编写用例（个人）。

在这个阶段，整个开发组按专业重新分成小组，每个小组由2到4人组成，并为每个专业小组选择用例。

花费几天或数周时间，各小组独自或成对编写用例（我没有发现更大的分组有利于用例的编
182 写工作）。然后在小组内传阅，不断改进草稿直到编写正确。然后编写概要用例。在用例编写过程中，小组成员肯定需要对某些用例进行分解，创建子功能用例，增加一些主执行者和新的目标等。

在这步工作中，对每个用例指定两个联系人，甚至可以将其中一人指定为主要编写者，这是非常有用的。在编写过程中，会出现很多关于业务规则的问题，这实际上是系统真正需求与原来约定之间矛盾造成的。这种工作方式便于一个人回答相关人员对业务规则的询问，同时，另一位人员还能对即将开始的用户界面以及用户目标是否在正确的层次进行复检。

第4步：审核用例（个人）。

用例编写人员可以通过电子方式，或直接通过纸面方式，传阅用例草稿。有趣的是，使用纸面方式有特别的好处。因为纸上保留了每个员工的评论，编写者可能只需对用例做一遍编辑，便可集众家之长。一个开发组说，他们曾试图采用在线建议的方法，但是由于用例要进行反复修改，

根据一个人的建议修改后，还需要查看这些修改是否满足其他人的建议，因此他们放弃了这种方式。但无论在任何情况下，传阅用例草稿的双方都应该对用例编写级别和业务规则进行检查。

编写者把用例发送给系统开发人员和系统应用专家进行审核。技术人员确定用例是否包含实现所需要的各种细节（数据描述和用户界面设计除外），应用专家应确定系统需求是否确实如此，以及业务过程是否真正以这种方式工作。

第5步：审核用例（开发组）。

最后应该有一个开发组对用例进行审核，软件设计人员、业务专家、应用专家、用户界面设计人员都参加审核。用例编写完成后，开发组根据项目以及项目的审核政策和审核机制建立一个他们认为最好的用例草稿。编写者应确保每一步都是可理解的、正确的，并且足够详细便于实现。所有这些工作可以通过正式审核、非正式审核、用户审核及开发人员审核来进行。

一旦系统用例草稿通过了用户和技术专家的详细检查，用例就达到了第一个正式的基本要求。然后将开始系统设计。此后对用例的修改应只限于修复错误，而不应该改变用例的表达形式。

很快就会看到编写用例草稿和完成用例之间的差别，在完成用例编写时，编者应该：

- 指定所有扩展条件。
- 仔细考虑业务策略与失败处理的联系。
- 验证对项目相关人员利益的保护。
- 验证用例是否仅描述了实际需要的所有需求。
- 确保每个用例对用户和应用专家是可读的，以及开发人员清楚地知道所要实现的系统。 183

17.6.2 用例需要的平均时间

草拟一个用例需要数小时，而跟踪扩展处理则需要数天时间。一个10人的开发组在一周内可编写140个用例摘要（平均每人每天2.8个）。然后，他们大约要花4周时间对这些用例摘要进行整理并增加其他需求。因此加上相关需求，总计每个用例大约需要两个工作周。一个开发组在不同项目中花费的平均时间大约为每用例3到5个工作周，这样设计出的用例具有极高的质量。

17.6.3 从大型团队中收集用例

有时，你发现自己正在与一个庞大的、不同类型的、非技术的应用专家团队进行工作，这是极富挑战性的工作。下面的文章摘录于Andy Kraus编写的报告，这份报告发表于“*Object Magazine*”[⊖]，它基于编写者同25个不同领域的专家会谈推动用例的成功经验。

◆ Andy Kraus：从庞大的不同地位的团队那里收集用例

不要忽略对会议场所的选择。

在数周时间内你将居住在这里，在会议期间要完全“拥有”它。……如果不断从一个地方移到另一个地方，会面临严重的后勤问题，应尽量呆在一个地方。

⊖ 摘自A. Kraus编写的报告“Use Case Blue”（*Object Magazine*，1996年5月（63～65页）。SIGS出版，纽约。）得到复印许可。

没有适合的人就找不到正确的用例。

与会者及其发表的观点越多越好……，我们需要这些实实在在的分析家、办公人员、检察员、数据操作员、23个部门的监督人员以及许多非办公人员的经验和头脑中的想法。如果不事先知道真正的系统执行者，我们就不可能预计哪些用户应该参加哪个讨论会。同时，让这么多人找一个共同时间也将是一大难题。我们通过与各个领域代表一起将用例划分不同阶段并确定后面会议的临时时间表，解决了这个问题。

与小团队相比，大团队更难一致。

应该尽最大努力去解决面临的问题，在这些问题中，最重要的是组织问题。随着事态发展，有时被迫要组织8～25人参加的会议，来讨论他们提出的各种问题，这都是由于必须
184 与庞大的团队一起工作造成的。但是，只有通过这种同时有各方代表参加的会议，我们才能消除由成员采用不同方法和不同操作过程造成的需求细微差别。

每次与用户开会不要超过半个工作日。

我们的经验表明，无论是对会议中讨论内容的估计，还是对我们处理这些内容能力的估计，我们都过于自信。实际上，与获得这些信息相比，对它们进行处理需要同样多时间。另外，大量烦琐的日常工作、管理工作、规划以及下一次会议的准备工作都可能要花上半天时间。

与管理者同舟共济。

在各种信息获取过程中，管理者和项目经理都应在场。

在提取用例时负责系统结构设计的人应在场。

系统结构专家会为开发过程提供专业知识。例如，领域专家会对过程的转移提出专家意见，并且一旦过程发生变化就能对其进行跟踪。

如果让支持取代原系统的人参加会议，就有一个达到目的的好机会。

大量组织和信息服务部门的人员参加会议，他们能对外部接口的需求以及当前系统的历史现状等方面提供深入的见解。

没有什么可以代替用例涉及到的“真正用户”。

我们必须确保实际的行政人员、分析人员、调查人员、数据操作人员以及监督人员参加会议。

使用抄写员。

快速、准确记录的重要性怎样强调都不过份。在开始几次会议中，我们使用了几个抄写员，事实表明这非常有用。

及时显示已建立用例来加速用例提取过程。

对用例进行交互式开发，并由一个监督者记录到可翻动的图纸上，一个抄写员输入到字处理程序中。然后把这些用例图挂到会议室墙上。尽管在可翻动的图纸上处理用例很不方便，但却收到了意想不到的效果。由于挂图时无意的懒惰，使用例混乱无序。开发新用例时，这种无序的状态迫使开发人员必须浏览前面开发的用例。这种反复有利于人们熟悉已存在的用例，从而加快了相似用例的开发速度。

一项工作的标题不是一个执行者。

让我们的用户接受执行者“角色”不同于工作标题是相当困难的。经过一天的艰难讨论，初步拟定了一份临时执行者列表，但对此大家好像并不满意。那么该怎样帮助他们认识清楚呢?

应用程序不关心你是谁，只关心你戴的帽子。

在计算机系统中扮演一个执行者的某个角色很像“戴帽子”，为了帮助用户弄清角色这个概念，我们买了许多棒球帽，并在每一个帽子上刺上执行者名字。第二天开会时，帽子被排成一排放在会议室前面的操作员桌子上，一旦提取用例开始，操作员就拿起一顶帽子，为发出询问的人带上一顶帽子，结果非常令人满意。通过这种做法，帮助用户理解了角色的含义。不管工作的标题是什么，当他们使用系统时，就必须戴一顶帽子（即扮演一个特定角色）。

185

在执行者初始编制过程中，寻找遗漏的执行者。

……进入抽取用例过程几周后，我们才认识到遗漏了一些用例——用于处理监督系统用户的用例。尽管尽了最大努力来确保参与者的广泛性及普遍性，但没有监督人员出现在讨论会上，更有趣的是，为他们工作的人也没有想到监督用例。

“日常工作用例”促进了获取用例的讨论。

……我们的用户在谈论用例时，总是缺少用例的“语境”，因此我们让他们写出（在较高的层次）日常活动中完成某些任务所遵循的步骤，例如停止操作等。在这时我们让用户写出使用系统的步骤，这种方式能让用户不去考虑“系统的用处”。在第3天我们花了一天时间开发日常工作用例，第4天就获得了20个用例。

不要急于求成。

像其他创造性活动一样，用例提取有它自己的高峰和低谷，在创造性活动中，企图急于求成只会适得其反。

期望陷入困境；讨论的催化剂。

“提示”——例如在讨论中使用投影机和与主题相关的传单，或将相关的问题放在一起，事实证明这是有效的催化剂，它们能活跃讨论。通常在讨论时，无意中会引入带有争议的主题和观点，我们发现当人们反驳他们所不支持的观点时，通常能迅速和清晰地表达出自己的观点。

用例提取是一项社会活动。

感情被伤害，意见被否定，以及为了在以后会议中保护自己，参与者偏袒一方，等等。因此，应花些时间消除成见，使彼此成为朋友。最后，我们变得非常融洽，无论在用例讨论会还是制定项目决议时，大家都积极发表自己的观点，并且互相尊重和支持。

标准的“描述符”有利于简化过程。

……标准描述有利于新系统的属性按特定规则进行划分，例如人、地方、位置等。描述符集也为信息提供了一致的表达方式 … 通过对描述符命名、分类和扩充，使它们能广泛用于会议讨论以及随后的用例细化。同样，标准的系统职责、成功和失败场景，允许我们

将主要精力集中于异常处理，而不是从一个用例到另一用例复制冗余信息。

构造、维护、显示前置条件列表。

当工作进行到一定阶段，有必要开一次“阅读前置条件”的讨论会。通过阅读使已经考虑过的前置条件最小化。

做一个最小主义者。

186 尽可能使你的用例模板最小。

第18章

用例概述和极端编程

极端编程(Extreme Programming，XP)是一种较浅的方法学，在本书中采用一种比另一本书（“*Extreme Programming Explained*”，Beck，1999）更简单的行为需求编写形式。采用XP，应用和业务专家与软件开发人员坐在一块，因此，开发组不需要编写软件详细需求，而只记录下“用户的故事”作为有约束力的备忘录以便将来能围绕这些功能讨论需求。

一个XP的用户故事，简略地说，可能就像在第3章的表3-3“用例简述实例”中描述的用例概述，也可能类似于第17章的表17-2“获得折扣”工作列表的一个系统特征。

每个XP的用户故事都必需足够详细，以便业务和技术人员理解其含义并估计其开发时间。同时它们应该是足够小的一部分工作，以便开发人员在3周之内能够完成设计、编码、测试，并可以使用。一旦用户故事满足了这些标准，就应对它做一个摘要，以便开发组进行管理。通常把它编写为一张索引卡。

当工作进展到一个用户故事时，设计人员只需简单地拿着索引卡来找到业务专家，询问详细需求。因为业务专家应该保证有足够的时间进行这种交流，所以这种交谈将一直延续，直到所有功能实现。

在另一种少见的场合中，一个配有专职用户的精心组织的小开发组可以把应用描述或用例概述作为需求。只有了解需求的用户和系统开发人员在一起时，这种方法才是可行的。在设计期间，设计人员可以直接与有需求的用户合作。就像XP工作故事一样，如果双方都能履行约定好的备忘录，这个条件就可以满足，XP也是可行的。但是，在大多数项目中，这些约定条件没法满足。因此最好还是把应用描述作为用例编写开始之前的练习，并把用例概述作为项目概要的一部分。

第19章
错误改正

在编写用例时，最常见的错误是遗漏句子主语、假定用户接口及目标编写过于详细等。下面是有关这些错误的例子。这一章的目的不是进行测验，而是希望通过这些例子提高发现错误的能力。

前面的例子很短，最后那个例子比较长，前面的小例子来自于一个实际项目。

19.1 没有系统

修改前

用例：提取现金

范围： ATM系统
级别： 用户目标级
主执行者： 储户
1. 储户插入ATM卡，并键入密码。
3. 储户按“取款”按钮，并键入取款数目。
4. 储户取走现金、ATM卡并拿走收据。
5. 储户离开。

修改提示

用例展示了主执行者（即储户）所做的一切，但没有显示系统行为。让人吃惊的是，人们经常编写这类用例。用例评审员对此回复到，“我没看到系统做任何事情，我们可以肯定这个设计太匆忙了”。

对用例的修改是指出这项活动中所有的执行者。

修改后

现在闭着眼睛都应该能写出一个ATM用例。

用例：提取现金

范围： ATM系统
级别： 用户目标级
主执行者： 账户持有者
1. 通过读卡机，储户插入ATM卡

2. ATM系统从卡上读取银行ID、账号、加密密码，并用主银行系统验证银行ID和账号。
3. 储户键入密码，ATM系统根据上面读出的卡上加密密码，对密码进行验证。
4. 储户选择快取，并键入取款数量。
5. ATM系统通知主银行系统，传递储户账号和取款数量，并接收返回的确认信息和储户账户余额。
6. ATM系统输出现金、ATM卡，显示账户余额的收据。
7. ATM系统记录事务到日志文件。

19.2 没有主执行者

修改前

下面是从ATM上取款用例的一部分。

用例：提取现金

范围：ATM系统
级别：用户目标级
主执行者：储户
1. 收集ATM卡，键入密码。
2. 收集取款事务类型。
3. 收集提取金额。
4. 验证账户上是否有足够储蓄金额。
5. 输出现金、收据和ATM卡。
6. 复位。

修改提示

本用例严格从系统角度进行编写，显示了ATM系统所有的行为，但没有涉及主执行者行为。这样编写的用例难于理解、验证及修改。在一些情况下，执行者行为信息的遗漏经常会导致用例的重写。对用例的修改是直接指出每个执行者和相应动作。 190

修改后

用例与19.1节中的用例一样。

19.3 过多的用户接口细节

修改前

用例：买东西

范围：采购应用系统
级别：用户目标级
主执行者：顾客
1. 系统显示输入ID及密码屏幕。
2. 顾客键入ID和密码，然后按OK。
3. 系统验证顾客ID及密码，并在屏幕上显示个人信息。
4. 顾客键入姓名、街道地址、城市、州、邮编、电话号码，然后按OK。
5. 系统验证是否为老客户。
6. 系统显示可用商品列表。
7. 顾客选取需要购买的商品及数量，完成时按DONE。
8. 系统通过库存辅助系统验证购买商品是否有足够库存。
……等等。

修改提示

这种错误在用例编写中是最常见的。编写者大量描述用户接口，使得用例变成了用户手册，而不是需求文档。过多的接口细节虽然对用例本身没有什么影响，但这样不利于用例的可读性，并使系统需求难以把握。

对用例的修改是：寻找一种描述用户意图的方法，而不是提出特定的解决方案。当然这有时需要做一些创造性工作。

修改后

用例：买东西

范围：采购应用系统
级别：用户目标级
主执行者：顾客
1. 顾客使用ID和密码进入系统。
2. 系统验证顾客身份。
3. 顾客提供姓名、地址、电话号码。
4. 系统验证顾客是否为老顾客。
5. 顾客选择购买商品及相关数量。
6. 系统由库存系统验证购买商品是否有足够库存。
……等等。

191

19.4 过低的目标级别

修改前

用例：买东西

范围：采购应用系统
级别：用户目标级
主执行者：顾客/用户
1. 顾客使用ID和密码进入系统。
2. 系统验证顾客身份。
3. 顾客提供姓名。
4. 顾客提供地址。
5. 顾客提供电话号码。
6. 顾客选取商品。
7. 顾客确定购买商品数量。
8. 系统验证顾客是否为老顾客。
9. 系统打开到库存系统的连接。
10. 系统通过库存系统请求当前库存量。
11. 库存系统返回当前库存量。
12. 系统验证购买商品的数量是否足够。
……等等。

修改提示

显而易见，这个用例非常冗长。与上节中的用例相比，我们不能说它对用户接口的描述过细。但是肯定想缩减这些用例，使整个过程更清晰。缩短步骤如下：

- 对数据项进行合并（3～5步），将这些分散的步骤放到一步来完成。“如何概括顾客所提供的信息呢？”答案是：“个人信息”，这是概括从客户收集的各种信息的好别名。当然这种别名太模糊，所以应该在其域列表上进行间接指示。域列表可以进行扩充，但这丝毫不影响用例。
- 把所有处于同一方向上的信息合并为一步（3～7步）。可是这种方法不一定很好，比如顾客选取商品及其数量信息和顾客个人信息有时完全不同，所以用例编写者需要把它们分开，单独作为一步。当然这与个人喜好有关，我喜欢把同一方向的信息放到单一步骤中。当然如果这些信息过于分散，或扩充需求需要它们分离，我会再把它们分成多步。
- 寻找一个更高层次的目标（8～11步）。“系统为什么需要做这些事呢？”那么可以用一句话来回答，即“系统试图通过仓库存储系统来验证购买商品是否有足够库存”。如上所述，如果用这个高层目标来捕获这些需求，则将使系统描述更清晰、更简明。

192

修改后

用例：买东西

范围：采购应用系统
级别：用户目标级
主执行者：顾客/用户
1. 顾客使用ID和密码进入系统。
2. 系统验证顾客身份。
3. 顾客提供个人信息（包括姓名、地址、电话号码），选择购买商品及数量。
4. 系统验证顾客是否为老顾客。
5. 系统使用仓库存储系统验证购买商品是否足够。
……等等

19.5　目标和内容不符

下面是对第7章练习7-4的提示，练习要求改正“登录”用例中的错误。
如果还没有做，请试图寻找如下三处错误：
- 用例体与用例名及概述中说明的意图不符。实际上，在这个用例中，用例体至少融合了两个用例。
- 用户接口细节描述过多。
- 用例文本中使用程序设计语言，而不使用自然语言。

193　如果不想做这个工作，请参考附录B中的讨论和解决方案。

19.6　用户接口描述过多的改进实例

FirePond 公司很友好地允许我使用下面用例修改前和修改后的两个版本。修改前版本共有8页，其中6页描述了主成功场景及类似的内容。修改后版本只有原来的三分之一长，保留了原来的基本信息，但删除了用户接口部分的约束信息。

请仔细阅读主成功场景部分，考虑在不影响原来需求的情况下，怎样能使这个长用例更让人满意？特别要注意用户接口设计的描述，同时，还应该看一些扩展部分，但没必要全部仔细阅读。我删除了其中的部分扩展，尽管这样，但仍足够让你领略使用这样长用例的困难。那么怎样对其进行简化呢？

在此，有必要对用例标题做一下说明。随着信息技术语言的市场化，说顾客“选购商品”被认为是不够严谨的。当面对纷繁复杂的商品世界时，顾客应该是根据他所处“情景”“寻找一个解决方案”，而不是单纯“选购商品”。尽管我非常愿意把用例重命名为“选购商品”，但这不

是我的职责，况且“寻找一种解决方案”这个标题在编写和阅读它的地方，都被认为是正确的，所以它被保留了下来。

修改前

用例36 寻找一种解决方案——修改前

范围：Web 系统
级别：用户目标级
主执行者：购物者——想寻找所需商品的顾客或代理人
主成功场景：

执行者的动作	系统的反应
1. 当购物者访问电子商务网站时，用例开始。	2. 系统收到购物者访问网站的类型信息。 3. 系统请求建立购物者的身份（调用“启动身份建立过程”用例）。如果系统在这里不能马上建立购物者身份，那么在为购物者存储解决方案之前，必须建立相应身份。 4. 系统为购物者提供下列选项：创建新方案或打开现存方案。
5. 购物者选择“创建新方案”。	6. 系统显示第一个问题来确定购物者的需求和兴趣。
7. 购物者反复增加商品到购物车中； 8. 存在确定需求和兴趣的问题； 9. 购物者回答问题。	10. 根据购物者对前面问题的回答，系统将提供不同数量和类型的问题来确定购物者的需求及兴趣，同时伴随显示一些诸如商品信息、性能价格比以及商品比较等相关信息。
11. 购物者回答最后一个问题。	12. 在询问有关购物者需求及兴趣的最后一个问题后，系统将显示推荐商品系列列表及相关信息，如商品信息、性能价格比信息、商品比较信息和价格信息等。
13. 购物者选取一个商品系列。	14. 系统显示确定商品样式的第一个问题。
15. 存在确定推荐商品样式的问题； 16. 购物者回答问题。	17. 根据购物者对前面问题的回答，系统将显示确定购物者有关商品样式需求及兴趣的问题，以及一些诸如商品信息、性能价格比、商品比较信息和价格等相关信息。
18. 购物者回答最后一个问题。	19. 在询问购物者所需样式的最后一个问题后，系统将显示推荐商品样式及相关信息，如商品信息、性能价格比信息、商品比较信息和价格信息等。
20. 购物者选择商品样式。	21. 系统确定标准商品样式选项，然后提出第一个决定商品主要选项的问题。

194

22. 存在确定推荐商品选项的问题： 23. 购物者回答问题。	24. 根据购物者对前面问题的回答，系统将显示确定购物者有关商品主要选项需求及兴趣的问题，同时显示一些诸如商品信息、性能价格比、商品比较信息和价格等相关信息。
25. 购物者回答最后一个有关商品选项的问题。	26. 在询问确定所需商品选项的最后一个问题后，系统显示所选择的样式及相关选项，由购物者确认。
27. 购物者浏览商品，选择所喜欢的商品，将其放入购物车中。	28. 系统在购物车中增加商品和导购信息。 29. 系统显示购物车中所有商品及总体概况信息。
30. 重复增加商品各步骤，直到购物者选择好了所有商品。	31.
32. 购物者请求对购物车中的商品提出个性化建议。	33. 系统显示用来确定建议内容的第一个问题。
34. 存在确定建议内容的问题。 35. 购物者回答问题。	36. 根据购物者对前面问题的回答，系统将提供有关建议内容的问题来确定购物者需求及兴趣，同时显示一些诸如商品信息、性能价格比、商品比较信息和价格等相关信息。
37. 购物者回答最后一个问题。	38. 在询问有关建议内容最后一个问题后，系统将产生并显示建议内容。
39. 购物者浏览建议并选择打印。	40. 系统打印购物者建议。
41. 购物者要求存储其方案。	42. 如果购物者身份没有建立，则调用“启动身份建立过程”用例来建立身份。 43. 系统提示购物者方案识别信息。
44. 购物者键入方案识别信息，并存储方案。	45. 系统储存方案，并使之与购物者建立联系。

扩展：

*a. 在寻求方案的过程中，如果购物者在系统预先确定的超时时间内没有任何活动，系统将提示购物者并询问是否继续。如果购物者在规定时间内（30秒）对询问没有响应，用例结束；否则，购物者继续进行方案寻找过程。

*b. 在询问/回答过程中，购物者可以从任何问题处返回，修改前面问题的答案，然后继续进行。

*c. 在推荐商品产生后，购物者随时可以查看性能计算信息是否适合他们的需求。系统将完成计算，并显示这些信息。同时，购物者将从他们离开的地方继续进行方案寻找过程。

*d. 在询问/回答过程中，系统与商品目录系统进行交互来返回可用商品信息，或与过程计划系统交互来返回调度信息。商品可用信息和调度信息可用来对所显示商品选择信息进行过滤，也可以在寻找方案过程中用来向购物者显示商品的可用性。通过“启动返回商品可用

性”和“返回调度信息”用例完成。

*e. 在询问/回答过程中，系统显示有关商品生产厂家的连接信息。购物者选择相关连接，系统可以把相关商品信息或方案信息传递到与此连接的相关网站的最适当位置或显示适当的内容。当购物者在生产厂家的站点上完成操作时，可以返回到原先的离开点，也可能带回来一些需要系统确认的商品需求。通过“启动浏览商品信息过程”用例完成。

*f. 在寻找方案过程中，购物者可以请求联系，通过“启动联系请求”用例与系统进行联系。

*g. 在询问/回答过程中，系统建立获得市场数据的触发点，通过它，系统能够捕获导购信息、商品选择以及对外部市场进行分析的询问/回答过程。

*h. 在寻找方案过程中的一些预设点上，系统可以产生一个直接获得解决方案的线索。使用“启动产生线索”用例完成。

*i. 在任何点上，购物者都能够退出电子商务应用软件：

如果购物者已创建了一个新方案或当前方案已被修改，系统将提示购物者是否要存储方案。调用“启动存储方案”用例完成。

*j. 新方案已被创建或当前方案已被改变，任何时候购物者都可以调用“启动存储方案”用例对方案进行存储。

1a. 如果购物者访问了相关商品供应商的网站，并已建立了商品需求，供应商网站允许用户登录回电子商务系统，并接着完成方案建立过程。

1a1. 购物者带着商品需求信息及相关识别信息登录电子商务系统。

1a2. 系统收到商品需求信息和相关用户识别信息。

1a3. 系统确定购物者处于方案过程中哪一步，然后建立一个询问开始点，继续寻找过程。

1a4. 基于开始点，从步骤5或步骤12或步骤18继续寻找方案过程。

3a. 购物者想使用预先存储的方案：

3a1. 购物者选择调出方案。

3a2. 系统显示为购物者存储的方案列表。

3a3. 购物者选取想要的方案。

3a4. 系统加载已选方案。

3a5. 从步骤26继续开始。

23a. 购物者希望改变某些推荐选项：

{创建一个“选择选项”用例，因为对正常的流程有两种选择：系统会显示所有选项，甚至包括相互不兼容的选项。如果购物者选择了不兼容的选项，系统应显示提示信息，告诉购物者如何配置商品使这些选项兼容。}

……当购物者需要改变选项时：

23a1. 购物者选取需要改变的选项。

23a1. 系统显示可用的兼容选项。

23a1. 购物者选择想要的选项。

26a. 购物者想改变购物车中商品的数量：

197

26a1. 购物者从购物车中选择商品并改变其数量。

26a2. 系统根据购物者以及对问题的回答等相关信息，考虑各种因素，如折扣、税收、小费、购物者及相关因素，重新计算总价格。

26b. 购物者想在购物车中增加商品折扣：

26b1. 参考“获得折扣”一节。

26c. 购物者想调用现存方案

26c1. 系统为购物者显示现存方案列表。

26c2. 购物者选择想要的方案。

26c3. 系统调用已选方案。

26c4. 从步骤26继续。

26d. 购物者用一些可行的付款方案对购物车中的商品进行付款：

26d1. 购物者选择对购物车中商品付款。

26d2. 系统基于购物者对前面问题的回答，提出问题以便确认推荐付款计划；系统调用“启动获得银行利率过程”用例与银行系统进行交互来得到银行信贷利率许可。

26d3. 购物者选择付款方案。

26d4. 系统根据前面的回答，显示一系列问题以便确认付款方案细节。

26d5. 购物者查看付款方案并选择同意。

26d6. 调用“启动放置付款订单过程”用例，系统把付款方案订单传给银行系统。

修改提示

编写者采用两栏描述的格式，对执行者及其行为进行分离。他们没有可视化地描述任何用
198 户接口设计，而采用了问/答模式，因此，在用例中描述了这些问题和答案。

我修改的第一步就是去除这种多栏格式，创建简单的单栏格式，这样在查阅时不用来回翻页。

然后，在用例中寻找假想的用户接口以及那些能够提升的用户目标。在本用例中，关键语句为“系统将根据以前的回答显示不同问题数目和类型”。这种说法虽然没有明显地提到类似鼠标单击之类的事情，但的确在系统中基于用户对问题的回答设定了用户接口。我设想了一些完全不同的用户接口看它是如何影响用例编写，所以就推想了一个不需要任何输入的用户接口，用户只需要单击图标就行。由此，我捕获用户真正的意图，然后消除了对用户接口的依赖。就像下面将看到的那样，用例长度将会大大缩短。

另外，我检查了每一段描述的目标级别，看看哪些步骤的目标太低了，是否可以提升。

在实际开始工作前，先讨论一个问题，即对设计者而言，单栏格式是否能清晰地表达出系统职责。Rusty的评价是肯定的。实际上他更愿意这样，这是因为：

- 单栏格式短小且易于阅读。
- 所有真实需求仍然在那里被清晰地列出。
- 设计受到的限制更少。

修改后

用例37 寻找可能的解决方案——修改后

范围：Web 软件系统
级别：用户目标级
前置条件：无
最小保证：没有动作，没有采购
成功保证：购物者准备购买商品，系统记录商品的选择和导购动作，记录购物者特征。
主执行者：购物者（任何在网上冲浪的人）
触发条件：购物者选择寻找一种方案。
主成功场景：
1. 购物者初始化新方案寻找过程。
2. 系统通过显示相关信息以及询问一系列问题来确定购物者需求和兴趣，从而帮助购物者选择商品类、样式及样式选项。在屏幕上显示的信息和问题取决于购物者一路上给出的答案。系统根据已编入程序的选择路径选择相关问题，以便向购物者推荐符合其兴趣的产品。显示的相关信息包括产品信息、性能价格比、商品比较信息及价格信息等。 199
3. 系统在日志文件中沿路径记录导购信息。
4. 购物者选择最终产品配置。
5. 购物者把商品加入购物车中。
6. 系统把选择的商品和导购信息放入购物车中。
7. 系统显示购物车内所有商品及概貌信息。
购物者重复上面各步，导购并选取合适的商品，加入到购物车中。
扩展：
a. 在任何时候，购物者可以调用“要求与系统联系”用例。
1a. 根据Web网站的所有者与发送请求的计算机的所有者之间达成的协议，发送请求的计算机可以保留购物者类型信息和请求信息：
1a1. 系统从web请求中抽取用户信息和导购信息，并加入到已记录的信息中，然后从询问/回答过程中的某步开始。
1a1a. 来自其他站点的信息无效或不可理解：
系统则尽自己最大所能，把所有达到的信息记录到日志文件，尽可能继续。
1b. 购物者想要继续一个以前存储的不完整方案：
1b1. 购物者调用“建立身份”用例，并选择方案。
1b2. 系统调用方案，并返回到上次离开系统时存储方案的位置。
2a. 购物者选择绕过部分或全部问题序列：
2a1. 提示购物者从基于（或不）个人特征的推荐商品中选择。
2a2. 系统记录选择信息到日志文件。
2b. 在增加商品到购物车之前的任何时候，购物者可以返回并修改前面任何问题的答案以便寻找新的推荐商品，或者选择商品。

2c. 在增加商品到购物车之前的任何时候，购物者能够要求查看可用性测试/性能计算（例如，按当前配置能否拖起某种重量的拖车）。系统执行计算并显示答案。

2d. 购物者经过web站点所有者预设的、产生销售线索（动态业务规则）的位置时，系统调用“产生销售线索”用例。

2e. 系统要求购物者能识别自己：

200 购物者调用“建立身份”用例完成。

2f. 建立系统与其他已知系统（商品信息、过程计划）的交互能力，这些系统可能影响商品有效性和商品选择：

2f1. 系统与其他已知系统（商品信息、过程计划）交互，得到必要信息。（调用“启动返回商品有效性”和“返回调度信息”用例完成）。

2f2. 系统使用这些信息来过滤商品或显示选项有效性。

2g. 提示购物者选择到相关生产厂家站点的连接：

购物者调用“查看其他站点”完成。

2h. 让系统与已知的客户信息系统交互：

2h1. 系统从客户信息系统调用“返回客户信息”用例。

2h2. 根据这些信息，系统从询问/回答过程中的某点开始寻找方案。

2i. 购物者想知道银行信贷利率，因为它影响商品选择过程：

购物者调用“获得银行信贷利率”用例完成。

2j. 购物者指示已选择购买商品时，系统与已知的银行账户系统建立交互：

2j1. 系统调用“返回账单历史记录”用例。

2j2. 系统利用购物者账单记录来影响商品选择过程。

2k. 购物者决定改变推荐的商品选项

系统应允许购物者根据他的意愿修改多个选项，并沿着路径显示有效选项，对不兼容选项提出警告。

5a. 购物者决定不把商品加入购物车中：

系统为以后保留导购信息。

7a. 购物者想改变购物车的商品：

系统允许购物者改变商品数量，移去商品或者回到选择过程以前的位置。

7b. 购物者要求存储购物车中的商品内容：

7b1. 系统提示购买者输入名字和ID，并存储。

7b1a.购物者身份不存在。

购物者调用“建立身份”用例。

7c. 购物者在订单中加入折扣

系统调用“获得折扣”用例。

7d. 购物者决定购买购物车中的商品：

购物者调用“获得付款方案”用例。

7e. 当购物车未存储时，购物者退出电子商务系统：

7e1. 系统提示购物者输入名字和ID，并存储。

7e1a. 购物者决定不存储退出：

系统记录导购日志，并结束购物者会话。 201

7f. 购物者选取购物车中的商品，希望从存储清单中了解同类（新的或旧的）商品的有效性：

7f1. 购物者调用“从存储清单找到同类商品”用例。

7f2. 购物者用存储清单中同类商品替换购物车中的商品：

7f2a. 购物者不想要存储清单中的同类商品：

系统保留购物者感兴趣的原购物车中的商品，停止对存货清单进行查找。

7f3. 系统确保购物车中的商品能与库存清单中的某项商品匹配。

调用用例：“Web销售”用例，“出售相关商品”用例。

未解决的问题：

扩展中的2c。什么问题是合法的？与什么样的系统相连？接口的需求是什么？ 202

第三部分

对忙于编写用例的人的提示

第20章 对每个用例的提示

提示1：每个用例都是一篇散文

前言中曾提到："就像写散文一样，全部困难在于既要采用单调的写作方式，又要富有完美的表达能力"。

FirePond公司的Russell Walters曾说过：

> 我认为以上叙述明确地抓住了问题所在。这个问题最容易给人带来误解，而且可能也会给用例编写者带来最大的启示。然而，我并不能肯定编写者能够自己明白这个启示，至少在本书出版以前是如此。我从事用例概念的研究已有四年之久，但是当有幸和你合作之后，我才把它作为一个基本的问题来理解。即使这样，直到我有机会分析和回顾了你帮助重写的前后两种版本的用例（第19章的用例36）后，思路才逐步清晰起来。经历四年多等来这个启示，真是一段不短的时间！所以，如果这本书仅能让读者明白一件事情的话，我希望它就是通过"编写有效用例"意识到了这个基本问题。（经许可采用。）

这个提示提醒你将注意力集中于文字上而不是图画上，同时帮助你了解将要遇到的写作风格。

提示2：使用例易于阅读

要使需求文档短小简明，而且易于阅读。

有时我希望有位八级的英语老师走到我身边，对我说，

> 在现在时态中使用主动动词。不要使用被动语态，要使用主动语态。句子的主语在哪里？只写真正是需求的东西，不要提及与需求无关的东西。

这些都可以使你的需求文档简明而且易于阅读。下面一些习惯会让你提前达到这一目的：

1) 让问题短小、切题。长用例满足大的需求，但很少有人喜欢阅读。
2) 从头开始，用一条主线贯穿始终。顶部是一些对全局有重要意义的用例。从这里分离出用户目标和最终的子功能用例。
3) 用动词短语来给用例命名，这些动词表明了用例所要达到的目的。
4) 从触发事件开始一直继续，直到目标实现或者被取消，并且系统完成所有与这次事务处理有关的记录。
5) 用完整的主动语态句子来描述所要完成的子目标。
6) 确保每步中执行者及其意图是可见的。
7) 突出失败条件，并使恢复动作是可读的。最好是不必为每个步骤编号，人们就能清楚地

知道下一步该做什么。

8) 将可选的行为放在扩展部分，而不是用例主体的条件语句中。

9) 只在非常必要的情况下生成扩展用例。

提示3：仅用一种句型

在编写用例的每个执行步骤时，只采用一种句型。

- 现在时态的句子。
- 在主动语态中用主动动词。
- 描述执行者成功到达的目标，这些目标推动了整个过程的前进。

以下是一些例子：

用户插卡并输入PIN。
系统确认该客户具有权限而且有足够的金额。
PAF截获从Web站点返回的响应，更新用户资料。
职员用“搜索失物细目”来寻找一件失物。

206 采用完整正式的或非正式的模板，在业务用例、系统用例以及概要用例、用户用例和子功能用例中使用这种句型。在主成功场景和扩展场景部分也是一样的。请掌握这种句子风格。

扩展的条件部分拥有不同的语法形式是很有用的，这样不会使它和执行步骤产生混淆。使用一个句子片段（或者可能是一个完整的句子），最好（但不是总是）使用过去时态。用冒号（:）代替句点作为条件的结束。

等待输入PIN超时：
密码错误：
文件没有找到：
用户中途退出：
数据提交失败：

提示4：“包含”子用例

如果没有人告诉你做别的事情，很自然地你将会做的事情就是写一个步骤来调用低层目标或用例，就像下面这样：

职员用“搜索失物细目”来寻找一件失物。

在UML的术语中，调用用例就包含了子用例。要做的事如此明显，以至于就像没有作者和老师鼓励人们使用UML扩展关系和特化关系（见附录A）一样，也没人提及这件事情。

最重要的经验规则就是，经常在用例中使用包含关系。遵循这条规则的人说，与那些将包含关系与扩展关系和特化关系混用的人相比，他们和读者之间产生的混淆较少。参见10.2节中的“什么时候使用扩展用例”小节。

提示5：谁控制球

有时人们采用从系统本身看待外部世界的角度来编写用例，这时使用被动语态。这样就产生了诸如“信用限制得以输入”这样的句子，句中并没有提及谁是输入者。

应该按从上向下的角度，以观察和记录景物的方式来编写用例。或者是以剧本中宣布哪个演员将要出场的方式来编写用例。或者假设在一段时间内你正在描述一场足球比赛，1号球员得到了球，运球，然后传给2号球员，2号球员将它传给3号球员；等等。

让句子中第一个或第二个单词是执行此项操作的执行者名字。无论情况怎样，都要确保清楚地知道谁控制球。 207

提示6：正确地得到目标层

- 回顾5.5节“找出正确的目标层”，进行了全面的讨论。
- 确保使用用例的目标层（概念、用户或子功能）正确地标记用例。
- 周期性地回顾前面的内容，以确定你知道目标的“海平面”在哪里，以及这些步骤在海平面之下（或之上）多远。回顾一下如何测试海平面目标：
  —由一个人，在一个地方，一次完成（2到20分钟）。
  —执行者一完成操作，就可以愉快地离开。
  —执行者（如果是雇员）在完成许多工作后可以要求加薪。
- 回顾一下，大部分用例在主成功场景中有3到9个步骤，每个步骤的目标层是否恰好在用例目标层之下。如果你的步骤多于9步，可以寻找一些步骤在下面的地方合并：
  —在一排中，同一个执行者在好几个步骤中都控制球的地方。
  —在描述用户操作的地方。这些操作通常是用户界面的操作，同时这也违反第7章中的准则5“显示执行者的意图，而不是动作”。
  —那些在两个执行者之间有很多简单来回操作的地方。问一下自己，是不是那些来回操作并没有真正试图去完成更高一级的事情。
- 问“为什么用户/执行者进行这个操作？”，得到的答案将是下一个更高的目标层。可以用这个目标合并你的步骤。图20-1展示了每一步的目标怎样对应不同层次的用例目标。

提示7：不考虑GUI

确定你所写的每一步恰好抓住了执行者的真实意图，而不仅仅是操作用户界面的动作。这个建议可以用在编写功能需求的时候，因为这很清楚，这时编写用例就是为了文档化用户界面本身。

在需求文档中，描述操作界面的用户动作有三个缺点：

- 不必要地加长了文档。
- 需求变得脆弱，意味着用户界面设计中的小改动都会引起“需求”（实际上根本不是需求）的变动。
- 它侵占了UI设计者的工作机会，应该相信他们能做得更好。

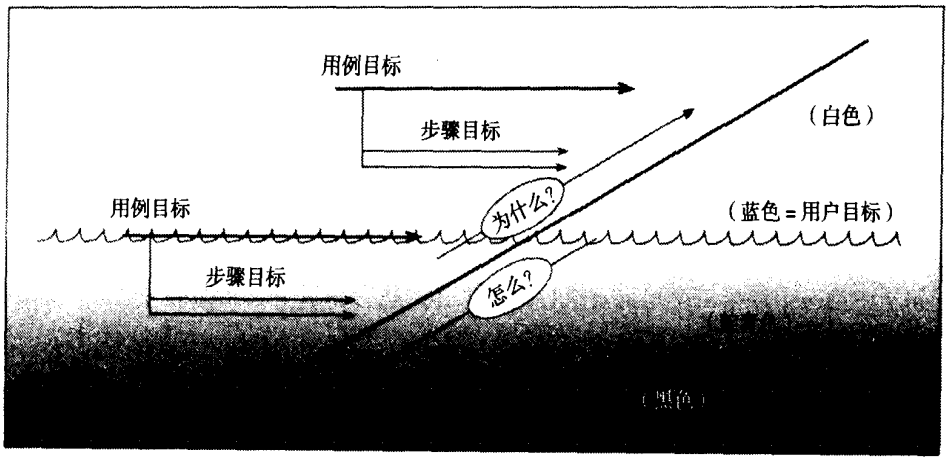


图20-1　问“为什么”来提高层次

本书中的大部分用例都可以作为很好的例子。以下摘自第5章中的用例20“评估工作补偿申请”，便是其中之一。

1) 调节者审查并评估报告，……
2) 调节者根据……评定伤残等级。
3) 调节者对应付的伤残赔偿金进行汇总，……。
4) 调节者决定最终结算范围。

以下是一个告诉你不该做什么的例子：

1) 系统显示有用户名和密码域的登录界面。
2) 用户键入用户名和密码并按确认。
3) 系统确认用户名和密码。
4) 系统显示包含功能选择的主界面。
5) 用户选择一项功能并按确认。

这很容易就陷入了描述用户界面操作中，所以一定要小心。

提示8：两个结局

每个用例都有两个可能的结局：成功和失败。

记住，当一个执行步骤调用一个子用例时，被调用的用例可能成功或者失败。如果是在主成功场景中调用，则将失败处理放在扩展中。如果调用来自一个扩展，则将成功和失败的处理均放在同一个扩展中（作为例子，参见第5章中的用例22“损失登记”）。

208 ~ 209

对于目标的成功与失败，你实际上有两个职责：一个是确定对每个被调用用例的失败都进行了处理；另一个是确定用例满足了每一个项目相关人员的利益，特别是在目标失败的情况下。

提示9：项目相关人员需要的保证

用例不是仅仅记录了主执行者和系统之间公共的可见交互操作。如果用例仅仅完成了这些操作，那么它不是一个可接受的行为需求，而仅仅是文档化了用户界面。

系统执行项目相关人员之间契约上的协定。其中一个项目相关人员便是主执行者，其他需要保护自己利益的人没有列出来。用例描述了系统在不同的环境下怎样通过用户驱动的场景来保护所有人的利益。用例描述了系统提供给这些人的保证。

花些时间为每个用例中的项目相关人员和主执行者命名。你应该发现2到5个项目相关人员：主执行者，公司的所有者，也可能有管理部门，或者其他人。可能测试或维护人员也对使用用例感兴趣。

通常，对于大多数用例来说，项目相关人员都是相同的，而且多数情况下他们的利益在整个用例中也是非常相似的。不用花多少力气就能把他们的名字和利益列出来。以下是他们一些典型的利益：

- 在用例名中说明主执行者的利益，通常是想得到一些东西。
- 公司的利益通常是，保证主执行者不能够免费拿走一些东西，或者为她所得到的东西付费。
- 管理部门的利益通常是，确认公司能够证明它遵循了准则，并且保留某种类型的记录。
- 项目相关人员的一个典型利益是，从事务处理当中的失败恢复机制中受益，那就可能需要更多的记录。

确保主成功场景和它的扩展瞄准了项目相关人员的利益，这并不需要花费很大力气。从主成功场景开始阅读用例文本，看看那些利益是否都已考虑到。你会惊奇地发现漏掉其中之一是多么经常的事。最常见的是，编写者没有想到在主成功场景中会产生失败，因而没有留下记录或恢复信息。检查一下全部的失败处理是否保护了所有项目相关人员的利益。

通常，一个新扩展条件就展示了主成功场景中一个遗漏的确认条件。有时，需要完成的确认条件太多了，以至于编写者可能将这一系列确认检查转移到另一个单独的编写区域，也可能创建新的业务规则。

210

Roshi公司的Pete McBreen写信告诉我，他的开发组首次对他们自己已经发布的系统列出了项目相关人员的利益。他们在列表中发现了他们的软件运行第一年中所有改动的需求。虽然他们已经成功地建立并发布了这个系统，但并没有满足特定项目相关人员的特殊要求。项目相关人员指出了这件事，当然跟着就有了改动需求。让这个开发组感兴趣的是，如果他们早一点写下项目相关人员及其利益的话，那些改动需求本来是可以避免的（起码是可以部分避免的）。结果，现在Pete极力倡导在用例中捕获项目相关人员的利益。进行这种检查只需占用很少的时间，但它却清楚地揭示了项目的时间花费。

用例模板的保证部分记录了用例怎样满足这些利益。具有很好个人交流的开发组在不太关键的、低规格的项目上，可以不用花费太大力气去编写这些保证。要将更多精力放在更关键的项目上，这些项目潜在的损失和由于误解带来的花费是比较高的。无论怎样，在这两种情况下，开发组都应该至少在头脑中检查一下用例的成功和失败出口。

在编写主成功场景之前先编写保证条件是一个好的策略，因为你在第一遍编写时就会考虑

必要的保证，而不是在后来发现它们时再返回来对文本进行修改。

2.2节“具有利益的项目相关人员之间的契约”和6.2节“最小保证”，都在这个问题上有更详细的讨论。

提示10：前置条件

用例中的前置条件表明了用例的可运行条件。系统必须保证前置条件为真。编写前置条件是为了在以后用例编写中不用再对它们进行检查。

通常有两种情况可以给出前置条件。最普通的一种情况是用户登录并被确认。另外一种情况是一个用例通过前面的用例获得激活条件，期望前面的用例已经建立它所能依赖的特殊条件。这种情况的一个例子是，用户在前一个用例选择或部分选择一种产品，而下一个用例在它处理过程中使用这些选择信息。

无论何时看到一个前置条件，我就知道存在一个建立前置条件的高层用例。

提示11：对用例进行通过/失败测试

简单的通过/失败测试让我们知道什么时候已经将部分用例正确地编写完毕。表20-1列出了
211 我的一些发现。对表中所有的问题都应该回答“是”。

表20-1　对用例进行通过/失败测试

域	问　题
用例标题	1. 是用主动动词短语命名主执行者的目标吗？ 2. 系统能完成这个目标吗？
范围与层次	3. 这些域填写了吗？
范围	4. 用例是把范围内的系统作为一个“黑盒”对待吗？（如果是系统需求文档，那么答案是“是”；如果用例是白盒业务用例，那么答案可能是“不”。 5. 如果对范围内的系统进行设计，设计者是只设计范围内的每件事，而不管范围之外的事吗？
层次	6. 用例的内容与所描述的目标层次匹配吗？ 7. 目标是在所描述的目标层次上吗？
主执行者	8. 他/她/它有行为吗？ 9. 他/她/它有目标与被讨论系统的服务承诺矛盾吗？
前置条件	10. 它们是强制的吗？它们能在某些地方被所讨论的系统设置吗？ 11. 在用例中它们真的从来不需要检查吗？
项目相关人员及其利益	12. 他们被指名了吗？系统一定能满足他们所描述的利益吗？（用法随规范性和灵活性而定）
最小保证	13. 所有项目相关人员的利益被保护了吗？
成功保证	14. 所有项目相关人员的利益被满足了吗？
主成功场景	15. 它有3～9个步骤吗？ 16. 它能从触发事件到交付成功保证运行吗？ 17. 在执行过程中，它允许适当地改变吗？

212

（续）

域	问 题
在任何场景中的每个步骤	18. 它描述了一个成功目标吗？ 19. 在它完成后，整个过程明显地向前移动了吗？ 20. 是否清楚地指出哪个执行者正在操作目标？——谁在“踢球”？ 21. 执行者的意图是否清楚？ 22. 该步骤的目标层是否低于整个用例的目标层？它是恰好比用例目标层低一点吗？ 23. 你确定该步骤没有描述系统用户接口的设计吗？ 24. 在该步骤内所传递的信息是否清楚？ 25. 它的“确认”与“检测”条件相匹配吗？
扩展条件	26. 系统能够并且必须发现和处理它吗？ 27. 它是系统确实需要的吗？
技术与数据变动列表	28. 你能否确定这不是主成功场景的一个普通行为扩展？
整个用例的内容	29. 对投资者和用户：“这是你们想要的吗？” 30. 对投资者和用户：“在交付时，你们能识别出这是你们所要的吗？” 31. 对开发者：“你们能实现它吗？”

第21章
对用例集的提示

提示12：一个不断展开的故事

对于一个开发项目，在最上层一定有一个被称为*使用ZZZ系统*这样的用例。这个用例不会比执行者及其最高层目标的目录表更复杂。对任何首次浏览该系统的人来说，它是一个起点。它是可选的，因为它没有多少故事的线索，但是许多人喜欢看到一个阅读的起点。

这种顶级用例调用了*最外层的用例*，这些用例表明了系统最外层主执行者的概要目标。对一个公司的信息系统而言，通常会有外部客户、市场部、IT部门或者安全部门。这些用例显示了那些定义系统的“海平面”用例之间的内在关系。对多数读者而言，“故事”开始于它们其中之一。

最外层用例展开为用户目标或海平面用例。在用户目标用例中，设计范围就是将要设计的系统。各个步骤表明了为实现用户的直接目标，执行者和系统之间的相互作用。

如果子用例很复杂或是在多个地方使用，则将海平面用例的一个步骤展开为水下（更深层或子功能）用例。子功能用例的维护费用很高，所以只有在必须的情况下才使用它们。通常必须为*寻找客户*和*寻找产品*之类的用例创建子功能用例。有时，会将深层用例的一个步骤展开为更深层的深层用例。

将用例集看成是一个不断展开的故事，其价值在于：将复杂的写作部分移到它自己的用例中，或者将简单的子用例合并到调用它的用例中，这使工作变得简单了。原则上，每个执行步骤都按照它们自身的理由，展开成一个用例。参见10.1节“子用例”。

提示13：业务范围和系统范围

设计范围容易引起混淆。人们对系统的确切边界有着不同的观点，特别是当他们不清楚你是在写一个业务用例，还是在写一个系统用例的时候。

业务用例的设计范围是业务运作，它所涉及的执行者在组织外部，完成与组织相关的目标。业务用例通常不涉及技术，因为它只涉及如何进行业务运作。

系统用例的设计范围就是要设计的计算机系统。它所涉及的执行者完成与计算机系统相关的目标，它涉及到技术问题。

业务用例通常以白盒方式来编写，它描述公司中个人和部门之间的相互作用。而系统用例通常以黑盒方式来编写。因为大多数业务用例的目标是描述公司现在和未来的设计，而系统用例的目标是为一个新设计形成需求，所以这样做通常是合适的。业务用例描述了业务的内部情形，而系统用例描述了计算机系统外的情形。

如果你正在设计一个计算机系统，你就必须既收集业务用例，又收集系统用例。业务用例

描述了系统功能的语境以及该系统在业务活动中的位置。

为了减少混乱，我们通常标明用例的范围。可以考虑使用图标来区分业务用例和系统用例（参见3.2节中的“使用图标突出设计范围”小节）。也可以考虑在用例本身所包含的系统内部放置一幅系统的图画（参见第3章中的用例8“输入和修改申请（联合系统）”）。

提示14：核心价值和变化

虽然人们一直在发明新用例格式，但经验丰富的编写者一致认同核心格式的价值。在1999年TOOLS USA会议上发表的两篇文章（Firesmith, 1999; 和Lilly, 1999）中描述了在用例写作过程中容易犯的大约12个错误。那些文章中的错误和修改反映了核心价值。

核心价值

所基于的目标。用例围绕主执行者的目标和各种不同执行者所要实现的子目标，包括所讨论的系统。用例的每句话都描述了一个要实现的子目标。

216

从系统外部的观察角度。通过用例来描述执行动作，就像从高处鸟瞰景色一样，或者像为剧本中的人物命名一样，而不是采用“由内向外看”的方式来编写用例。

可读性。用例或任何规格说明最终都是供人们阅读的。如果人们读不懂它们，那么用例就没有达到它的核心目标。可以通过牺牲一些精确性甚至准确性，并加强交流，来增强用例的可读性。事实上，一旦牺牲了可读性，人们将不会读懂你的用例。

用于多个目标。用例是一种行为描述形式，在项目的不同时间可能用于不同的目标。例如，它们被用于

- 提供黑盒功能需求。
- 为一个组织的业务过程再设计提供需求。
- 文档化一个组织的业务过程（白盒方式）。
- 帮助获得用户或项目相关人员的需求（当开发组以其他形式编写最终需求时，这种需求会被用户或项目相关人员抛弃）。
- 描述将要创建和运行的测试用例。
- 文档化系统的内部（白盒方式）。
- 文档化设计行为或设计构架。

黑盒需求。作为功能规格说明时，所讨论的系统通常被看作是黑盒。曾经试图编写白盒需求（猜测系统内部看起来是什么样子的）的项目组报告说，这导致用例难以阅读，不受欢迎，而且很脆弱，即随着设计进程而改变。

在主成功场景之后的选择路径。在主成功场景后放置选择过程的初衷是想保持这个创建最易读用例的方式。将分支情况放入文本主体中可能会让人难以读懂。

不要太费精力。继续将时间耗于用例不会增加它们的价值。首次创建的很多用例可能只有它价值的一半。增加扩展后可能会增加价值，但是过不了多久，对句子中用词的修改将不再能增进交流。此时，你就应该将精力转入别的事情，例如外部接口、业务规则等，这些都是需求的一部分。当然，这种减少的比例随项目的关键程度而变化。

217

适当的改变

在保证核心价值的同时，能够发现一些可接受的改变。

编号的步骤与简单的分段。一些人将每一步都编了号，以便在扩展部分供他们参考。另一些人以简单的分段方式来编写，并将可选部分放在相似的段落形式中。这两种方法看起来都很好。

非正式与完整正式。有时候把大量精力放在细化功能需求是合适的。而有时候，甚至是在同一项目中，这么做就是浪费精力。（参见1.2节“你的用例不能作为我的用例”和1.5节“合理安排你的精力”）。就某种程度而言，我无法推荐哪一种方法，经常是非正式和完整定义的方法都正确。不同的编写者有时偏爱其中一种，每种方法都有自己的运作方式。比较一下，第11章中的用例25“*实际登录（非正式版本）*”，和第1章中的用例5“*买东西（完整正式版本）*”。

有或没有用例的优先级业务建模。一些开发组在编写系统功能需求之前，喜欢建立或修改业务过程。其中，有些组选择用例来描述业务过程，有些组则选择另外一些业务过程建模形式。从系统功能需求的角度看，选择哪种业务过程建模方式都不会有太大的差别。

用例图与执行者-目标列表。一些人采用执行者-目标列表去描述要开发的用例集，另一些人则偏向于采用用例图。用例图描述了主要执行者及其用户目标用例，这与执行者-目标列表的目标是相同的。

白盒用例与协作图。白盒用例和UML协作图几乎等价。可以将白盒用例看作是文本化的协作图。两者的区别在于协作图不能描述构件内部的动作，而用例却可以。

不合适的改变

主成功场景中的条件语句。如果在用例中仅有一个行为分支的话，那么将分支放在主文本中会简单一点。然而，用例有很多分支，人们就会找不到事情发展的主线。使用过条件语句的
218 人们报告说，他们很快就转向了使用主成功场景后面跟扩展的形式。

用顺序图代替用例文本。一些软件开发工具声称支持用例，因为它们能提供顺序图。顺序图也显示了执行者之间的相互关系，但是

- 它们不能捕获内部动作（需要这些动作来表明系统怎样维护项目相关人员的利益）。
- 它们难以阅读（它们使用专业化术语，并且占用了很多空间）。
- 在执行者之间狭窄的空间内，几乎不可能放下必要的文本描述。
- 许多工具强迫编写者将文本隐藏在弹出式对话框后面，使人们很难弄清事情的主线。
- 许多工具强迫编写者每次从用例的始端出发，独立地编写每个可选路径。这种重复性工作令人厌烦，很容易出错，而且给读者增加了阅读难度，他们必须检查每个变化中出现了什么不同的行为。

顺序图不是一个很好的用例表示方式。坚持用它的人是为了获得工具提供的自动服务：交叉引用、向前向后的超级链接、以及能在全局范围更改名称的能力。虽然这些功能都很好（这正是目前使用的文本工具所欠缺的），但是许多编写者和读者都一致同意，为这些好处不值得去牺牲易读性和易写性。

功能规格说明中的GUI。有一个编写需求的小技巧，能使用户接口不要随着所需要的功能而确定下来。这种技巧不难而且值得一学。大多数人都强烈反对在用例中描述用户界面。参见19.6

节“一个用户接口描述过多的高级用例”和“*Designing Software for Use*”（Constantine和Lockwood, 1999）。

提示15：用例集中的质量问题

对整个用例集我只有三个有关质量的问题：

- 每个用例都是来源于对最高层到最低层目标的展开吗？
- 对每个主执行者在最外层的设计范围内都可能存在一个与设置语境相关的、最高层的用例吗？
- （对于项目相关人员和用户）“这就是所要开发的全部内容吗？”

219

第22章 处理用例的提示

提示16：仅仅是第3章（第4章在哪儿呢？）

用例只是整个需求收集工作中的一小部分，是需求文档的“第3章”。它们是这项工作的重点，在连接数据定义、业务规则、用户接口设计、业务领域模型等中起着核心作用。然而它们并非需求的全部，仅仅是其中的行为部分。

这一点必须反复强调，因为使用用例已经形成了一种趋势，以至于一些开发组想方设法在需求的每一部分都使用用例。

提示17：首先向广度上努力

首先从广度上而不是从深度上，做到逐步精确。工作随着细化（见图22-1）逐步加大，这将有助于分配你的精力(见1.5节“合理安排你的精力”)。应按以下顺序工作：

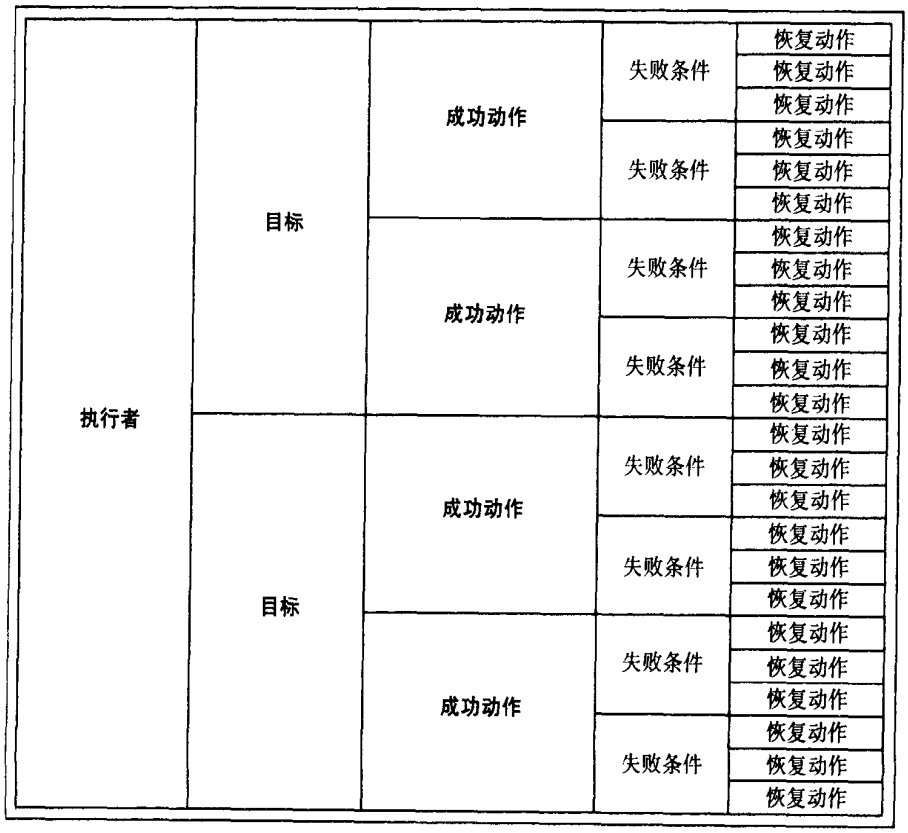


图22-1 工作随着细化而增加

1) 主执行者。把收集所有的主执行者作为了解整个系统的第一步，尽量简明。很多系统非常庞大以至于你很快就会丢失每件事的线索，所以在一个地方了解整个系统，即使只有一小段时间，也是好的。集体研讨这些执行者将帮助在第一轮就获得大部分目标。
2) 目标。列出所有主执行者的目标可能是你在一个视图中了解整个系统目标的最后一次机会。花费足够多的时间和精力，尽可能完整和正确地列出这张表格。下一步将涉及更多的人和工作。与用户、项目相关人员和开发人员一起审查这张表格，以便他们都同意表中的优先关系，并且了解所开发的系统。
3) 主成功场景。主成功场景通常简短而明了。它指明了系统所要交付的内容。在考虑所有可能出现失败情况之前，确保编写者已展示出系统是怎样工作的。
4) 失败/扩展条件。在考虑怎样处理扩展条件之前，应先找到它们。这是一个靠集体研讨的活动，与研究和编写扩展处理步骤不同。产生的扩展条件列表可以作为编写者的工作单，这样编写者在时断时续地编写用例时，不用担心停下来会找不到已经写到何处。那些试图一列出扩展条件的名字就想马上完成的人，通常永远不会完成失败情况列表。当他们完成部分失败情况列表后就已经筋疲力尽。
5) 恢复步骤。虽然这是构造用例中的最后一步，但是奇怪的是，恢复步骤通常不会发现新用户目标、新执行者和新失败条件。为了编写它们，编写者被迫要去面对一些经常被忽略的业务规则问题，因此这是用例编写过程中最艰难的部分。当我在完成恢复步骤中发现了一个很少用的业务策略、新执行者或新用例时，我感到所有的努力都有了回报。 221 ~ 222
6) 数据域。虽然形式上超出了编写用例的工作范围，但是相同的开发人员常常被派去将数据名（例如“客户信息”）扩展为数据域列表（参见16.1节“数据需求的精度”）。
7) 数据域详述和检测。有时，其他人会在用例编写者复查用例时完成数据域详述和检测。通常IT技术人员编写数据域详述，而IT业务分析家或者甚至用户编写用例。这代表了细化到最后的数据格式。再次声明，尽管这些详述和检查超出了用例体系的范围，但是最后必须完成它们的编写工作。

提示18：12步秘诀

1) 寻找系统的边界（语境图，输入/输出表格）。
2) 集体研讨并列出主执行者（执行者简介表）。
3) 集体研讨并列出主执行者对系统的目标（执行者-目标列表）。
4) 编写覆盖以上各项的最外层概要级用例。
5) 重新考虑和修订具有战略意义的用例，通过增加、删减来合并目标。
6) 选取一个用例进行展开，或者写一段叙述来熟悉材料。
7) 将项目相关人员、利益、前置条件和保证填入表中，再对它们检查两次。
8) 编写主成功场景，对比利益和保证来检查它。
9) 集体研讨并列出可能的失败条件和可选的成功条件。
10) 写出在每个扩展中执行者和系统应该做什么。
11) 列出所有需要有自己空间的子用例。

12) 从顶部开始重新调整用例。适当地增加、删减和合并用例，再次检查完整性、可读性和失败条件。

提示19：认识错误的代价

降低用例质量所带来的开销取决于你的系统和项目。一些项目在写需求文档时不需要质量保证，因为它们已经在用户和开发者之间进行了很好的交流：

223 “克莱斯勒综合补偿”项目组在建立软件来支付所有克莱斯勒公司的花销时，使用了“极端编程”方法（Beck, 1999），他们从来没有编写过比用例简介更深入的东西。他们只写了一些被他们称为“故事”而不是用例的东西，并把它们记在索引卡上。每一个“故事”都确实是需求专家和开发者交谈中的一个承诺。更重要的是这14个组员都在两个相邻的房间里工作，有很好的组内交流。

应用专家和开发人员的交流越好，那么由于用例模板省略部分所带来的损失就越少。人们可以进行简单的讨论，然后直接解决问题。

如果你正在进行分布式开发，或在多协议者的开发组中工作，或者在一个大项目组中工作，或者正在开发一个有严格期限的系统，那么质量不过关所带来的损失就会很高。如果将系统功能正确地写下来是很重要的，那么你需要密切注意项目相关人员及其利益、前置条件和最低保证。

了解你的项目在整个范围中的位置。不要被小型、非正式系统中的小错误搞得筋疲力尽。但是如果错误的后果非常严重，就需要认真对待。

提示20：喜欢蓝色牛仔服

“如果写的太少而不是太多，通常带来的危害也会小一点”，这句话听起来很奇怪吧。如果有疑问的话，就少写几行，使用层次较高的低精度目标，并且采用简单的故事形式。这样，就有了一个短小而可读的文档，这也意味着人们在阅读中会有很多疑问，于是就会提问。从那些问题中就能发现遗漏的信息。

与之相反的策略带来了失败：如果编写了一个100行，或是很详细的底层用例，则很少或者没有人在阅读时感到不便，你也就切断了而不是放开了项目组中的交流。喜欢编写太低层的目标是程序员的一个通病，因此经常会发生这类错误。

◆ 一个真实的小故事

在一个拥有50个人、1500万美元的成功项目中，我们仅仅写了简单的一小段主成功场景和一些失败情况。这样使我们有了很好的交流。每个需求编写者都和2～3个程序设计人员编为一组，他们都坐在一起，或者每天要见好几次面。

224 实际上，提高组内交流的质量对每个项目都有好处。上面故事中所描述的开发组策略就来自于*Survivng Object-Oriented Projects*（Cockburn, 1998）中的整体多样性范型。

提示21：处理失败情况

用例的一个很有价值的地方就是定义了所有的扩展条件。在很多项目中，程序员在写到以下的代码时，都会有一个停顿：

```
if(条件)
  then(做该操作)
else ...?
```

程序员停下来苦思冥想这个else...，“我想知道系统想在这儿做什么？需求对这种异常情况却只字未提，我也不知道去问谁。哦，好了……”于是他很快地在程序中加入了一些东西：

```
else（做那些事情）
```

对“else”的处理应该放在需求文档中。通常，它包括了重要的业务规则，我经常看到应用专家聚在一起，或是把他们的同事召集起来，强调系统在这种情况下应该做什么。

找出失败条件和编写失败处理常常引出新的执行者、新的目标和新的业务规则。通常对这些失败的处理是很敏感的，需要进行一些研究，或者说它们改变了系统的复杂度。

如果你仅仅习惯于编写成功场景，那么在下一个用例中试着找出失败情况及其处理。你可能会为你的发现感到惊奇和激动。

提示22：前期和后期的工作标题

工作标题在项目开始和结束时都是很重要的，但在中间时不是这样。

在项目的开始，要收集系统需要满足的所有目标，并将它们放入一个可读的结构中。将精力集中于工作标题或是受系统影响的社会角色，可以使你进行有效的集中讨论，并且给目标命名时就有一个良好的开端。一旦手头有了一长串目标列表，工作标题也能提供一个分簇方案，使得复查和优化已命名的目标更加容易。

可以使用工作标题来描述不同工作技术和工作风格。标题信息也可以体现用户界面的设计。

一旦人们开始开发用例，将会引起关于如何处理角色重叠的讨论。主执行者使用的角色名变得很一般化（例如，“命令发出者”），或者与真正使用系统的人差别很大，以至于管理人员开始提醒用例编写者，应存在一个真正完成目标的执行者。

225

系统一旦开始使用，工作标题再次变得重要。开发小组必须：

- 为特定用户指定权限，去更新或者可能只是阅读每种信息。
- 根据人们对工作标题的熟练程度和每组将要使用哪些用例，来准备新系统的培训材料。
- 将各种分开实现的用例放在一起，打包整个系统以供使用。

提示23：执行者扮演角色

“执行者”是指使用该系统的人的工作标题，也指使用系统时那个人所扮演的角色（假定使用者是一个人）。以哪种方式使用这个术语并不重要，因此不必花太多的精力区分二者。

重要的部分是目标，它表明了系统将要做什么。在系统生命周期里可能会协调和重新安排

是谁完成系统目标。当你发现仓库管理员也可以是售货员时，你可以：

- 在用例中写下主执行者是“售货员或仓库管理员”（UML发烧友：从两个执行者分别各画一个到椭圆的箭头）。
- 写下“仓库管理员在处理这个用例时可能充当售货员的角色”（UML发烧友：从仓库管理员画一个到售货员的泛化箭头）。
- 生成一个“命令发出者”作为主执行者，并指出“售货员和仓库管理员在使用这个用例时都扮演了命令发出者的角色”（UML发烧友：从售货员和仓库管理员分别各画一个到命令发出者的泛化箭头）。

以上的方法都没有错，可以选择其中任何一种你所能找到的方式与读者交流。

记住一个人可以充当多个角色，一个人在做一件工作就是在扮演一个角色。一个工作标题中的人可以扮演很多角色，甚至可以扮演该工作标题中的多个角色。记住用例中最重要的部分不是主执行者的名字而是目标。记住制定一个开发组使用的命名规则是很有用的，以便他们能坚持使用工作标题。

回顾4.2节中的，“为什么说主执行者是不重要的（和重要的）”小节和本章中提示22“前期
226 和后期的工作标题”小节，去发现执行者是怎样变为角色名，又是怎样返回对应的执行者名。

提示24：大的图画恶作剧

不知什么原因，自从Jacobson的第一本书《面向对象的软件工程》（1993）问世以来，许多人在用例编写时将精力集中在小人和椭圆上，却忽略了用例最基本的文本形式。强大的CASE工具行业早已提供了图形工具，但却没有提供文本建模工具，这些图画抓住了人们的这种注意力，提供工具使得用例中图画的数量达到了极限。这在OMG的统一建模语言标准中是不正确的，这些标准是由文本用例领域中具有丰富经验的人制定的。我怀疑是强大的CASE游说团影响了OMG的努力。“UML仅仅是一个工具之间交互的标准”，这是我在很多场合中听到的解释。因此，每个椭圆后面的文本不知为什么没有成为标准的一部分，而仅是由编写者决定的局部问题。

不管什么原因，现在的情况是：许多人认为椭圆就是用例，即使它们根本没传递什么信息。经验丰富的开发者对此持讽刺的态度。我要感谢Andy Hunt和Dave Thomas，他们使用像图22-2一样的示意图对“用例图使需求变得容易”的观点进行了嘲弄（摘自1999年的*The Pragmatic Programmer*）。

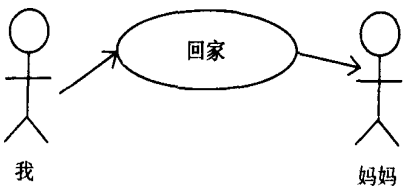


图22-2　“妈妈，我想回家。”

认识到椭圆不可能代替文本是很重要的。用例图（有意地）省掉了排序、数据和接收执行者。它可以被用作：

- 用例的目录表。
- 系统的语境图，显示执行者寻求他们可变的和重叠的目标，也可能是系统得出了辅助执行者。
- 一幅“大图画”可以显示高层用例和低层用例的关系。

正如第21章的提示14“核心价值和变化”中所说，这些都很好。要记住用例的基本形式是文本，因此那些椭圆只是文本的补充，而不能代替文本。图22-3和表22-1展示了语境图的两种表示方式。列表显示出同样的执行者和目标，为了清晰，在此重复使用了公共用例。

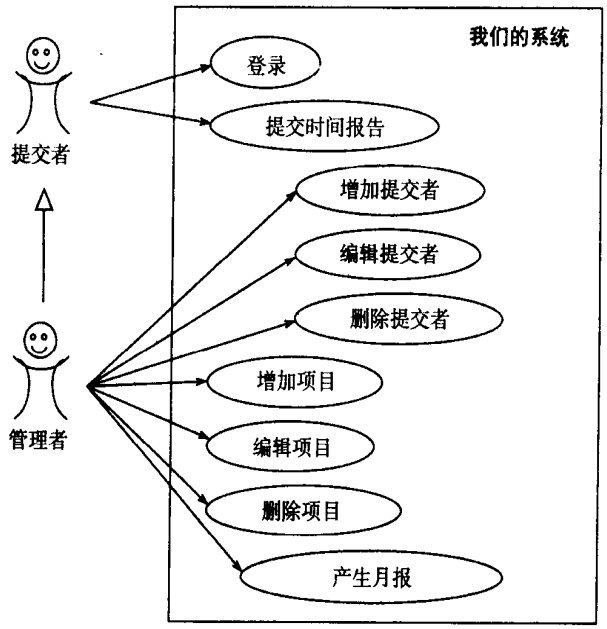


图22-3 椭圆图形式的语境图

（摘自Grady Booch的手稿并进行了适当修改。）

表22-1 语境图的执行者-目标列表

执 行 者	目 标
提交者	登录
	提交时间报告
管理者	登录
	提交时间报告
	增加提交者
	编辑提交者
	删除提交者
	增加项目
	编辑项目
	删除项目
	产生月报

提示25：大型工具的争论

遗憾的是，现在市场上还没有任何工具能很好地支持用例。许多公司都声称要支持用例的文本形式或图形形式。然而，没有一个工具包含与2.3节“图形模型”所描述的图形模型相类似的元模型，并且许多工具都不易于使用。结果使用例编写者面临一个艰难的选择。

LOTUS NOTES。这个工具是我比较喜欢的，LOTUS NOTES还没有用例的元模型，但是支持协同工作、超链接、通用模板、文档历史、在用例集间快速地查看和编辑以及很容易创建可分类视图。这些都是重要的优点。LOTUS NOTES也允许将扩展的数据描述保存在同一数据库中，而显示在不同视图中。当模板更新时，数据库中的所有用例也都被更新。这种模板很容易建立，使用起来也很方便。我曾经使用了LOTUS NOTES和项目相关人员一起审查了关于固定资产项目投标的200多个用例。

和别的纯文本工具一样，LOTUS NOTES的缺点是反复为每个步骤和扩展编号，从而使编辑用例变成了一件很麻烦的事情。超链接最后作废了，人工插入的向后链接很快也作废了。因为在超链接中没有自动的向后链接，所以你不知道是哪个高层用例调用了你正在查看的用例。

LOTUS NOTES最吸引我的地方是它使用起来很方便，并且提供了加注解的执行者-目标列表方式，这种方式可以动态地生成用例集视图。只要编写一个新用例，视图里立刻就会显示它的存在。视图同时也是一个有内容的超链接、一个执行者-目标列表和一个过程跟踪表。我喜欢从优先级、实现、完成状态和标题的角度，或者从主执行者或主题域、层次和标题的角度来审查用例。

具有超级链接的WORD字处理程序。通过使用超级链接，WORD字程序终于可以支持用例了。将用例模板放入一个模板文件。将每个用例放入使用该用例模板的文本文件中，这样在用例之间生成链接变得很容易。千万别更改文件名！用例编写者对WORD都很熟悉，用它来写东西也很顺手。

WORD也具有LOTUS NOTES的所有缺点。更严重的是，它们没有提供任何方式列出所有用例，按交付情况或状态进行分类，并通过点击打开用例。这就意味着必须构造和维护一个独立的概要列表，而这些列表很快就没用了。由于对模板没有全局性更新机制，因此多模板的版本可能会花费更多的时间。

关系数据库。我曾经听说并见到了一些将执行者、目标和步骤模型放入关系数据库中的尝试，例如微软的ACCESS。这种想法很自然，但是开发出的工具却很难使用，这些工具通常将用
229 例编写者又带回了WORD。

需求管理工具。专门的需求管理工具（例如DOORS或Requisite Pro）正在逐步普及。它们提供自动的向前和向后超链接，并且采用基于文本的需求描述方式。缺陷是，我所了解的工具没有一个支持主成功场景和扩展的模型，而这些模型是用例的核心。我看到采用这些工具生成的用例没有几个是很长的，并有很多排版的缩进、编号、行数，这使用例难以阅读（回忆一下第20章中的提示2“使用例易于阅读”和本章中的提示20“喜欢蓝色牛仔服”）。如果你正在使用这种工具，一定要想办法使整个情节清晰。

CASE工具。CASE工具的优点是，支持元模式中对任何实体的全局性更改和自动的向后链

接。然而，正像前面所说，它们都是围绕方框和箭头而建立的，很少使用文本。顺序图无法代替文本用例，但是许多CASE工具只不过是为文本输入提供了一个对话框。我已经看到很多开发小组放弃了CASE工具，转向使用字处理工具。

上述情况使你很难作出选择。愿你好运。

提示26：使用标题和简介的项目计划

回顾17.1节“用例在项目组织中的作用”，目的是使用增加和减少用例来跟踪项目进展，并且举出一个将执行者-目标列表作为一个项目-计划构架的例子。下面是提示。

用例计划表。将执行者和目标填入表中最左边的两列，在接下来的几列中依次填入所需的下列信息：业务价值、复杂度、实现、开发组、完整性、性能需求和外部接口等。

借助这个表，开发组就可以协商每个用例的实际开发优先级，讨论业务需求与技术复杂度、业务依赖关系和技术依赖关系，然后提出一个开发顺序。

交付部分用例。正如17.1.2节“跨版本处理用例”中所描述的，在一个特殊版本中经常会只交付一部分用例。许多开发组只是简单地使用高亮色或黑体文本来标明哪一部分用例先交付。应该在项目计划表中注明在第一个版本中哪些用例要交付，在最终版本中哪些用例需要全部交付。 230

附　录

附录A
UML的用例

统一建模语言（UML）定义了人们想要的图标。它没有确定用例内容或编写风格，但是提供了许多让人们讨论的复杂东西。应将精力放在学习写出清晰的文字上。如果你喜欢图形，首先要了解关系的基础，然后制定一些简单的标准来保持图形清晰。

A.1 椭圆和“小人”图符

当你走到白板前要画出人们怎样使用一个系统的图画时，很自然地会用“小人”图符来代表人，用“椭圆”和“方框”图符来代表调用的用例。将“小人”图符命名为执行者，“椭圆”图符命名为用例。图中的信息和在执行者-目标列表中的信息是一样的，只是表现方式不同。图形可以被用作一张目录表。

迄今为止，所有的一切都很好，很正常。

当你和你的读者认为这些图形定义了系统的功能需求时，麻烦就来了。一些人迷恋于这些图形，认为它们可以使复杂的工作变得简单（如图22-1所示）。他们试图在图形中获得尽可能多的信息，甚至可能希望再也不需要写文字。以下是这种情况的症状。

在我最近的课上，一个人展开了一张卷着的图，有几英尺长。上面画满了椭圆和指向四面八方的箭头，将所有的包含、扩展和泛化关系搅成了一团（当然仅仅靠箭头上的一小段文字来区别）。他想知道他的项目是否正确地使用了所有关系，但他并没有意识到他这样做是不可能真正地知道他的系统要做什么的。

另一个人骄傲地指出他怎样“修改”了图中的明显不足，即没有标明子用例的调用顺序。通过使用UML中的顺序关系，他加入了很多箭头来指示一个子用例与另一个用例之间的先后关系。结果当然是生成了极其复杂的图，这种图比等价的文本占用了更多的空间，而且难以阅读。为了解释一句古老的谚语，他可能要在那张难以读懂的图中又加入很多比较容易读懂的句子。

图形是一个二维的助记工具，它适用于一个认知目的，即要突出用例的相互关系。应该将图形用于这个目的，而不是去代替文本。

抱着这个目的，让我们看一下UML中的每个关系。

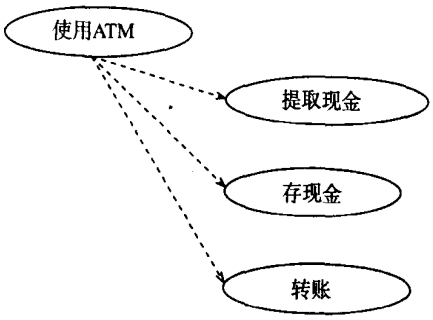
A.2 UML的包含关系

如果基用例中的一个操作步骤调用了被包含的用例名，那么这个基用例就包含了一个被包

含用例。这是在高层用例和低层用例之间一种常见和明显的关系。被包含用例描述了比基用例更低层的目标。

操作步骤中的动词短语可能就是一个潜在的子用例名。如果不将这一步的目标放到它自己的用例中，那么它只是简单的一步操作。如果将目标放到它自己的用例中，按我的说法就是这一步调用了子用例，或者按UML1.3的说法就是它包含了被包含用例的行为。在UML1.3之前，一般说法是它使用了低层用例（这种说法现在已经过时）。

从（高级）基用例到被包含用例的虚线箭头表明：基用例“知道”这个被包含的用例，如图A-1所示。



234

图A-1 包含关系的画法

准则13：将高层目标画得高一点

通常可以将图中的高层目标画得比低层目标高些。这有助于减少目标层次的混乱，而且会使读者感觉很直观。这样做时，从基用例到被包含用例的箭头通常都指向下方。

UML允许你改变每个元素的图形表示。我发现许多人在手工画图时，简单地把从基用例到被包含用例的箭头画成了实心箭头（虚线箭头太麻烦了）。这很好，现在你可以修改它。但是在一个图形程序中，你可能需要使用与该程序一起的箭头风格。

对大多数程序员来说，包含关系很明显来源于程序设计语言中的子程序调用。这不是什么问题，也不丢脸，而是我们在编程序和日常生活中都很自然地使用这种机制。有时还需要参数化用例参数、传递功能参数、甚至有返回值（参见第14章“两种特殊用例”）。不过要记住，用例的目标是和人沟通，而不是和CASE工具或编译器沟通。

A.3 UML的扩展关系

*扩展用例*通过重新指定一个用例和定义中断基用例的条件来扩展一个基用例。基用例不为扩展用例命名。由于新中断用例的加入，因此每次更新基用例都会产生大量的维护工作。如果你有很多用例要中断基用例，而又不想有大量维护工作的话，采用扩展关系是一种很有用的方法（参见10.2节“扩展用例”）。

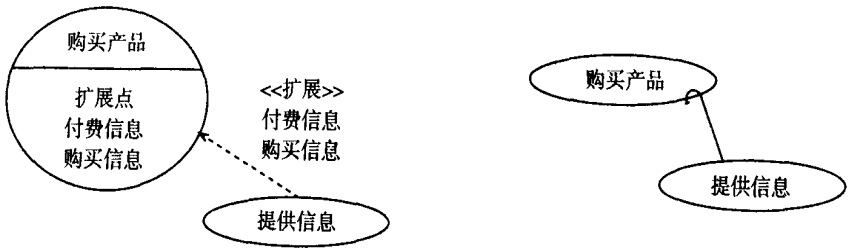
从行为上说，扩展用例指明了基用例中的某些内部条件和一个触发条件。基用例中的各种

活动都一直向前进行直到触发条件产生，在发生点上，活动在扩展用例中继续进行。当扩展用例完成后，行为返回它离开基用例的地方。

Rebecca Wirfs-Brock精辟地将扩展用例看作是基用例的补丁（程序员应该想起类似的程序补丁！）。另一些程序员将它看作是伪程序指令的文本说明，即*come-from*语句。

在用例中编写扩展条件时，我们会很自然地使用扩展形式。一个扩展用例就是在它上面带有转入转出处理的扩展条件（参见10.2节“扩展用例”）。可以将它看作是一个场景的扩展，这种扩展产生它自己的用例并拥有自己的空间。

UML对扩展关系的缺省画法是从扩展用例到基用例画一条虚线箭头（与包含关系的画法一样），并在旁边标上“《扩展》”的说明。我的画法是从扩展用例向基用例画一个钩子，如图A-2所示，用来强调包含关系和扩展关系之间的不同。 235



图A-2 扩展关系的画法

图A-2a显示了UML中扩展关系的缺省画法（该例子摘自Fowler于1999年所写的专著“*UML Distilled*”）。图A-2b显示了“钩子”连接符。

准则14：将扩展用例画得低一些

扩展用例通常比它扩展的用例层次低，所以在图中它应该放得低一些。在扩展关系中，事实上是低层用例知道高层用例，因此，“箭头”或“钩子”应该从扩展用例向基用例上画。

准则15：使用不同形状的箭头

UML故意没有确定连接用例符号的箭头风格。任何关系都能用一个开叉的箭头和一小段说明该关系的文字画出来。这种思想的出发点是考虑到不同的工具提供商或项目小组可能想要定制箭头的风格，UML标准不应该阻止这种做法。

不幸的是人们简单地使用相同的箭头表示所有关系，这使得图形难以阅读。读者必须仔细研究那些小段文字以确定是哪种关系，而以后没有任何简单的可视化线索来帮助他识别这些关系。这种情况再加之缺少其他绘图规定，使得许多用例图确实难以理解。

鉴于以上原因，应多花点时间为三种不同关系建立不同的箭头风格：

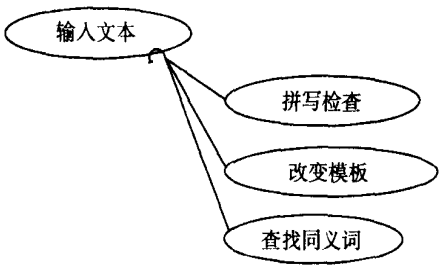
- 包含：使用缺省的开叉箭头，这是最常用的一种箭头。
- 泛化：UML中标准的泛化箭头是三角箭头，使用这种箭头。 236

- *扩展*：可以创造一个全然不同的形状。我使用从扩展用例到基用例的钩子形状。读者会发现它很容易辨认，不会和其他UML符号冲突，同时它暗示了自己是一个通过钩子与基用例相连的扩展用例。无论用什么形状的箭头，都应使扩展连接符与图中的其他符号有所区别。

A.3.1 正确地使用扩展关系

扩展用例（最初的讨论出现在第10章的“什么时候使用扩展用例”小节中）最常出现的场合是当有许多用户想启动中断基用例的异步服务时。通常，它们由不同的小组开发。这种情况出现在收缩包装的软件包的构造过程中，如图A-3所示。

其他通用的场合是对一个锁定的需求文档编写补充材料时。在一个增量式开发的项目中，可能在每次交付后锁定需求，然后通过*扩展*在锁定的用例中增加功能。



图A-3 扩展一个基用例的三个中断用例

A.3.2 扩展点

最初发明扩展的原因是在开发过程中无法再改变以前系统的需求文档这个实际情况。在使用用例开发早期的电话系统中，经常在业务中增加异步服务，于是使用了*扩展*关系。在安全锁定需求文档的情况下，新开发组可以在基用例的适当位置增加新的异步服务需求，而不用改变
237 原始系统需求中的任何一行。

然而，引用其他用例中的行为会产生一些问题。如果没有使用行号，怎么能找到扩展行为发生的地点呢？如果使用行号，在用例被重新编辑过且行号改变时将会发生什么情况？

回忆一下，行号实际上就是行的标号。既然这样，它们不必是数字的或顺序的。它们的存在是为了增加可读性，以便扩展条件有一个引用点。然而，行号通常是顺序的编字，这意味着它们会随着时间而改变。

引入*扩展点*是为了解决这些问题。*扩展点*在基用例中是一个可见的公共标号，通过别名确定用例在某个时刻的行为（技术上，它可指一系列位置，但这段时间我们先不考虑这个问题）。

引入可见的公共扩展点带来了新问题。基用例的编写者需要知道在哪里进行扩展，只要有人想在一个新地方进行扩展，编写者就必须回去修改基用例。但是回顾一下：*扩展*的本意就是为了避免修改基用例。

你必须处理以上问题。我发现定义公共扩展点带来的麻烦比它们的价值更大。我宁愿用文

本来描述扩展用例在基用例的什么地方开始，而不用别名来描述，正如下面的ATM例子所示。

如果你确实使用了扩展点，那么不要在图中将它们表示出来。它们会占用椭圆的大部分空间，控制读者的视线，模糊许多重要目标的名称（参见图A-2）。它们所表示的行为也不要在图中显示出来，这会引起更多的混乱。

关于扩展点还有一些事情。如果需要在基用例中通过扩展用例来增加行为，扩展点的名字允许按你所需的数目在多处被调用。例如，当增加扩展用例——*使用其他银行的ATM*时，对ATM用例可能需要这样。扩展用例需要说明：

在完成事务之前，系统得到来自用户交纳附加服务费的许可。

……

当完成所要求的事务后，系统从用户的账户中扣除附加服务费。

当然，也就能说这些。

238

A.4 UML的泛化关系

一个用例可以特化（*specialize*）另一个更普通用例（更普通用例泛化（*generalize*）特殊用例）。（特化）子用例应该属于和（普通）父用例相似的种类。UML1.3更确切地指出，“用例间的泛化关系表明子用例包含了父用例中定义的所有属性、行为序列和扩展点，并且参与父用例中所有的关系”。

A.4.1 正确地使用泛化关系

一个很好的测试词汇是*普通或类属*（*generic*），也就是属于“某一种类的”。当你说“用户采取了某种行动”时要注意，这时你可以考虑泛化。

以下是“*使用ATM*”用例中的一个片段。

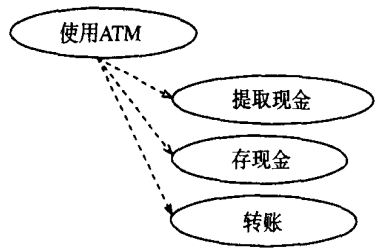
1) 用户插卡和输入PIN。
2) ATM验证用户的账户和PIN。
3) 用户执行下面任一事务：
   提取现金
   存现金
   转账
   核对余额
   用户执行这些事务直到选择了退出为止
4. ATM退卡。

用户在第3步中做了什么？通常的回答是“完成了一个事务”。这里用户可以完成4个事务。“普通的”和“某种”暗示了我们：出现了一般的或泛化的目标——完成一个事务。在纯文本版本中，我们没有注意到，我们正在用例中使用泛化关系；我们只是简单列出了用户能做的操作

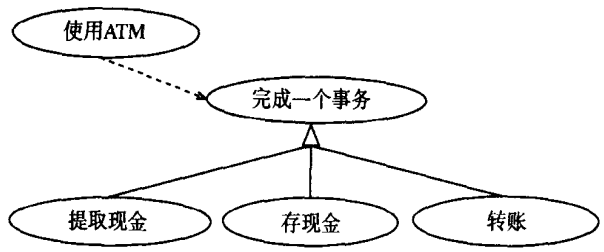
或事务的类型，然后完成。然而，在UML中，这就是要画出泛化箭头的信号。

实际上，我们有两个选择。我们可以忽视整个泛化关系而仅仅包含特殊操作，如图A-4a所示。或者我们也可以为完成一个*ATM*事务生成泛化用例，并将特殊操作作为它的特化，如图A-4b所示。

选择你喜欢的做法。用文字的方式来编写时，我不创建泛化用例。在这种方式中很少将文本放入普通的目标，所以没有必要为此而创建一个新的用例页。然而，在图形方式中，没有办
239 法来表示做以下事务之一，所以你必须找出并且命名泛化目标。



a)



b)

图A-4 泛化关系的画法

将一个被包含用例集转换为泛化动作的特化。

准则16：将泛化目标画得高一点

在图中常常将一般目标画得高一点，并且将三角形箭头朝上指向用例底部而不是旁边。图A-4解释了如何操作。

A.4.2 泛化的危害

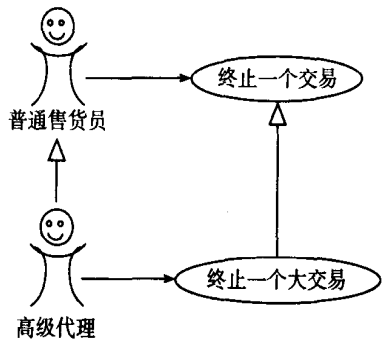
当把执行者的特化和用例的特化混在一起时要小心。常用的避免方法是特化的执行者使用特化用例。为了解释清楚，图A-5试图去表达一种非常普通的观点，即“一个售货员可以终止任何交易，除了那些需要特殊的售货员（高级代理）来终止的超过了一定限制的交易”。然而，它实际上表达了相反的意思。

回顾4.2节“主执行者”中，特化执行者可以执行普通执行者能执行的任何用例。因此，售货员是一个泛化了的高级代理。对许多人来说，这看起来与直觉印象不太一致，但它是正式认

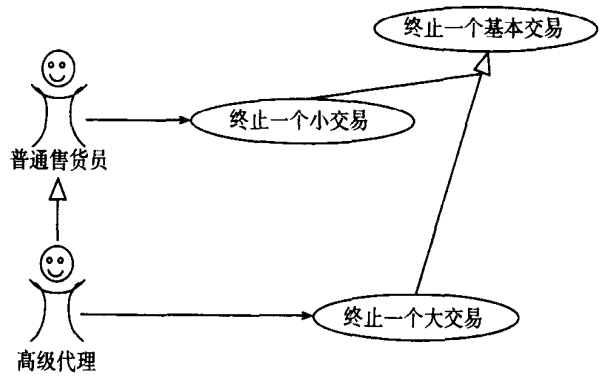
可的和正确的。

另一个特化看起来非常自然，那就是终止一个大交易是终止一个普通交易的特殊情况。然而，UML的规则是特化用例能够在任何地方替代普通用例。因此，图A-5说明一个普通售货员能终止一个大交易！ 240

正确的画法如图A-6所示，但是当你看到图形时可能会问：终止一个小交易真是终止一个基本交易的特化吗，还是它的扩展？由于在文本用例方式下工作就不会给任何人带来这种难题和不必要的困惑，因此我将这个问题作为一个练习留给感兴趣的读者。



图A-5 泛化的危害——终止大交易



图A-6 改正后的终止大交易

总的来说，关于泛化关系的问题：它到底表示子类还是特化行为，即应包含哪些属性和选项，这些在专业人员内还没有达成共识。由于用例是对行为的描述，因此对它们特化行为的意义没有标准的理解。

如果你确实使用了泛化关系，建议你将泛化用例的行为置空，就像上面完成一个事务那样。然后特化用例将提供所有的行为，你必须警惕刚才描述的那个陷阱。 241

A.5 从属用例与子用例

在UML1.3规范中的扩展文本部分，作者描述了两种鲜为人知的用例间关系，一种没有图标

对应部分，另一种没有在对象约束语言中描述，仅仅写入了说明文本中。这就是从属用例和与它相反的上级用例。

这些关系的目的是让你能够显示系统构件用例如何一起工作来完成一个大系统用例。在这个奇怪的轮回中，并不显示各构件本身；它们的用例就是它们本身，并不占用空间。就好像你想画一张匿名的协作图（一种特殊类型的功能分解），你希望以后再用一种适当的协作图来解释。根据UML规范，

> 描述一个模型元素的用例以后会被细化为一系列小用例，每一个小用例描述了包含在前者中模型元素的一种服务……注意，虽然用例没有显示包含者元素的结构，但是这是因为用例仅仅描述元素提供的功能。一个高级用例的从属用例互相合作来完成上级用例。这种相互合作定义为协作，可能出现在协作图中。

在用例规约的说明文本中介绍这些特殊关系的意图是不明确的，我不打算去解释它们。提到了这件事仅仅是因为我在本书中用到了术语“子用例”，一些人可能会围着问，“Cockburn的子用例和UML的从属用例之间的关系是什么呢？”

我使用子用例这个术语是指一个层次较低的目标。通常，高层用例将调用（包含）子用例。我习惯对高层用例和低层用例说“从属”和“上级”，但自从UML1.3使用这两个词以来，我就换用了别的词。我的经验是人们对术语“调用用例”和“子用例”不会发现任何不妥之处，甚至对首次编写和阅读用例的人来说它们的含义也是很清楚的。

A.6 用例图的画法

如果你建立并遵循一些简单的画图规则，你会发现用例图使你和读者的沟通更容易。不要把一堆像老鼠窝一样的箭头交给读者，然后期望他们能明白你的意思。复习准则13到准则16，
242 对理解不同的用例关系会有所帮助。还有另外两个画图准则也会有所帮助。

准则17：语境图中的用户目标

在语境图中，不要显示任何比用户目标层更低的用例。毕竟，该图的目的是为被设计系统提供语境和一张目录表。如果你以画图的形式来分解用例，则应将分解的用例放在几张单独的页上。

准则18：将支持执行者放在右边

我发现将所有的主执行者放在系统框的左边，将右边留给支持（辅助）执行者，是很有用的。这减少了主执行者和辅助执行者之间的混乱。一些人从来不在他们的图中画辅助执行者，则他们可以将主执行者放在两边。

A.7 代之以编写基于文本的用例

如果你花了很长时间学习和注意图形和关系，那你就学错了地方。你应该把精力放在编写

一些易读的文章上，在这样的文章中用例之间的关系很直观，你不会理解为什么有其他人被牢牢束缚在用例的图形和关系上。

许多用例专家都采纳了这种观点。与下面的事件联系起来，读者可能认为我有些自私，但我仍然想强调这个建议的重要性。非常感谢IBM公司欧洲对象技术实践部的Bruce Anderson在1998年OOPSLA会议上关于用例的讨论中所提出的意见。在那次会议上，与会者提出了一系列关于包含和扩展之间差异的问题以及关于场景和椭圆数目爆炸所带来的麻烦。Bruce说他的开发组没有陷入场景爆炸的困境中，也没混淆各种关系。一个人问到，为什么其他所有人都关心“场景爆炸和怎样使用扩展”，而他却不。Bruce回答说“我就是按照Alistar说的办法去做了”，那就是把时间花在将文本写清楚上，不要使用扩展，不要关心图形。

将大量时间花在UML中的“木偶”、“椭圆”和“箭头”上的那些人所遇到的问题，不会困扰那些简单地将文本用例写好的人。当你展开一个故事时，关系就自然会出现。只有在详细描述时，这些关系才会成为问题。因此，咨询专家在文本和UML中获得的经验越多，他们就会越赞成这种观点。

附录B
部分练习题答案

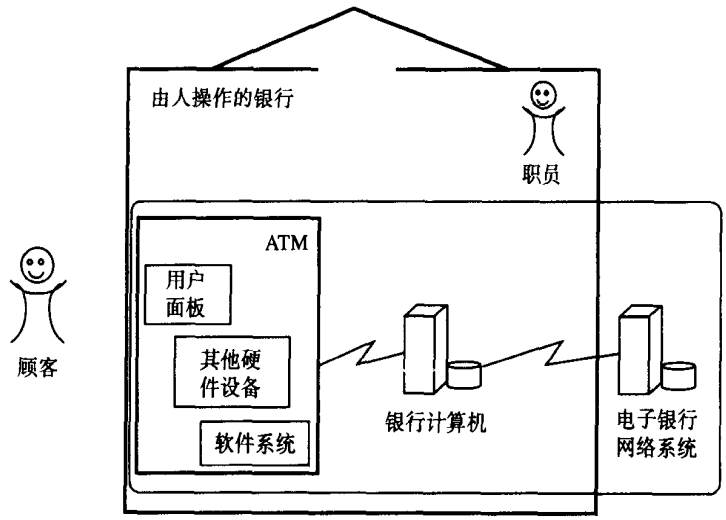
第3章练习题

练习3-1

据推测，这可能是在描述**临近区域**的设施情况，也可能是在描述一套**电子互连的工业系统**。从更小的范围考虑，可能是在描述**银行建筑**和**照明系统**的设计情况。还有可能是在描述一个新的**银行计算机系统**和ATM，或仅仅是ATM。甚至也有可能是在讨论一个新型**键盘**的设计或新型**回车键**的设计。单从这个故事片断中无法确定被讨论的是哪个系统。

练习3-2

同样，单从这个故事片断中无法确定被讨论的是哪个系统。



图B-1 ATM的设计范围

第4章练习题

练习4-2

回忆一下成功/失败测试。执行者必须能够执行一个相当于条件语句效用的行为。主执行者有一个目标，即调用系统承诺的服务。

ATM：当前讨论的系统。

顾客：主执行者和项目相关人员。

ATM卡：不是执行者。它没有充足的行为（这是指“非智能金属卡”；有嵌入芯片的“智能

卡”可以胜任）。ATM卡实际上仅仅是一个数据信封，只是为顾客提供快速、固定的键入服务。

银行：就设计的目的而言，它不是执行者，它是一个包含ATM的系统。

前台：就设计的目的而言，它不是执行者，它是当前所讨论系统的一个构件。

银行所有者：项目相关人员，可能不是主执行者。

服务人员：主执行者。

打印机：就设计的目的而言，它不是执行者，它是当前所讨论系统的一个构件。

银行计算机控制系统：辅助执行者。如果能想像出一种情况，它激发了与ATM的一次交互，那它也可以是主执行者。

银行出纳员：取决于工作分配。谁清空和补充现金？如果答案是“补充员”或“服务人员”，那就永远不会生成一个以银行出纳员为主执行者的用例。如果答案是“银行出纳员”，那么银行出纳员就是一个主执行者。

银行抢劫犯：取决于设计范围和设计者的想像力。只有在有人建议注意“偷盗ATM”情况时，我才会考虑关于银行抢劫犯的正规用况（通常，它不是顾客用例的一种扩展情况）这会激发移动探测机制的产生。根据目标的描述方式，最终可能会产生一个以银行抢劫犯作为主执行者的用例（但这个用例永远不会成功结束），也可能只不过为顾客用例添加更多的扩展情况。

练习4-3

答案要根据所选择的包含系统而定（见图B-1）。

ATM：就设计的目的而言，它不是执行者。现在它是当前所讨论系统的一个构件。

用户：仍然是主执行者和风险投资者。

246

ATM卡：不是执行者，原因如练习4-2（有相同的放弃原因）。

银行：参见图B-1。如果你选择将“由人操作的银行”作为包含系统，它就是你当前讨论的系统。如果你选择“电子银行网络”，那么它就可能是一个执行者（依据你能否证明它所调用的电子银行系统服务是正确的来确定）。

前台：就设计的目的而言，它不是执行者，它是一个构件。

银行所有者：根据所选择的包含系统和所提出的服务目标的不同，它或者会被当做银行的一部分，因此不是主执行者；或者被当做银行的主执行者，但可能不是电子银行系统的主执行者。

服务人员：如果是外部雇用的服务人员，那么他就是主执行者；如果他是银行的员工，而且你将银行看作是当前所讨论系统，那么他就是一个构件。

打印机：就设计的目的而言，它不是执行者，它是一个构件。

银行计算机系统：现在它是包含系统（它们中的任何一个）的构件。

银行出纳员：或者是（银行中的）构件，或者可能是电子银行系统中的主执行者。

银行抢劫犯：与练习4-2相同。

第5章练习题

练习5-1

- 概要（白色）：请某人出去吃饭（这个答案在ATM案例中有点牵强）。
- 概要（白色）：使用ATM。

- 用户目标（蓝色）：从ATM中取款  。
- 子功能（靛青色）：输入PIN。
- 子功能（黑色）：找到确认按钮。

247 **练习5-2**

执 行 者	目 标	级 别
服务人员	使ATM处于工作状态	概要
	运行ATM的自检程序	用户目标
银行职员	存储现金	用户目标
	补充供应	用户目标
顾客	使用ATM	概要
	提取现金	用户目标
	存钱	用户目标
	转账	用户目标
	检查余额	用户目标

第6章练习题

练习6-1

寻找最小保证的最简单方法是问，“怎么做会使令项目相关人员不高兴？”项目相关人员就是顾客、银行和监督机构。

如果顾客得不到现金，他们就会不高兴，但这不是最小保证中所期望的内容。让我们假设他们没有得到现金，在这种情况下，如果在交易中顾客要借款，他们将会不高兴。事实上，在任何时候，如果登记的借款金额超出了他们实际收到的现金数目，顾客就会感到不高兴。他们希望得到所有事务的日志，以便他们能够保护自己免受欺诈。

如果顾客实际得到的现金比银行登记的借款金额多，银行就会很不满意。银行也需要一个日志来保护自己，这可能是一种特殊的日志，它能指出事务进行到什么程度以便发生灾难性的故障时可以找出错误的原因。

监督机构的目的是确保各项准则都能被贯彻实施，因此它最感兴趣的是生成一个记录了所有事务的完整日志。

因此，最小保证是借贷金额等于实际发放金额，并且要生成一个小型日志，以便在发生灾难性故障时可以掌握事务的处理进度。同样，每个事务都要被记入日志。

练习6-4

成功保证是借贷总量等于发放总量（不是需要总量——检查异常条件！），返回卡，重置机
248 器，生成事务日志。

第7章练习题

练习7-1

下面是从ATM中提取现金的界面细节描述。发出100个这样的用例会令读者不高兴。作为深入的描述，看一下练习7-2的答案。

1) 顾客通过读卡机运行ATM卡。
2) ATM读取银行ID和账号。
3) ATM询问顾客是用英语还是西班牙语。
4) 顾客选择英语。
5) ATM要求顾客输入PIN号并按确认键。
6) 顾客输入PIN号并按确认键。
7) ATM列出顾客所要进行的活动列表。
8) 顾客选择“提取现金”。
9) ATM询问顾客取多少，必须是5美元的倍数，然后按确认键。
10) 顾客输入一个5美元的倍数，然后按确认键。
11) ATM通知顾客账户所在的主银行系统，提取顾客所要求数额的现金。
12) 主银行系统接受提取请求，告诉ATM新的余额。
13) ATM发放现金。
14) ATM询问顾客是否要收据。
15) 顾客回答要。
16) ATM开出了显示新的余额的收据。
17) ATM将事务写入日志。

练习7-2

下面是快速提取现金的流水线形式，表现了执行者的意图。

1) 顾客通过读卡机运行ATM卡。
2) ATM从卡中读取ID和账号，在主机上验证其有效性。
3) 顾客输入PIN。ATM验证其有效性。
4) 顾客选择快速提现模式，并输入提取数额，该数额应为5美元的倍数。
5) ATM通知顾客账户所在的主银行系统，提取出所需数额，并收到包含新余额的确认通知。
6) ATM发放现金、卡、和显示新余额的收据。
7) ATM将事务记入日志。

249

练习7-4

例子中包含三种错误。首先，这个用例不是关于登录的，尽管用例名和描述中都这样声称。它是关于使用订单处理系统的。真正的用例是一个“风筝”层次的概要用例。前六步是关于登录的，但与该用例处于完全不同的目标层次上，应该将其单独划分出去。一旦这样做了，将发现用户登录到系统后将永远出不来!

“当用户没有选择退出循环、结束条件和结束循环”是程序的结构，它对用户复查用例毫无意义。连续的条件语句使编写的用例显得混乱。这些步骤描述了用户界面设计。所有这些都应该是固定不变的。

“当……用例开始和当……用例结束”是一些老师建议的惯用风格。它们没有什么特别的错误。

它们仅仅是装饰，不是必要的。大多数人很自然地认为用例从第一步开始到停止写作时结束。

另一点要提到的是短语“用户……然后使用设置订单”。短语中的“使用”是指UML中的包含关系（以前称为使用关系）。它使文章混乱而不是清晰，所以最好用“用户……设置订单”。要尽可能沿用项目组关于如何引用其他用例所建立起来的规范。

最后，分离出两个用例：“风筝”层用例（使用订单处理系统）和子功能层用例（登录）。可以自己独立地创建登录用例。注意在引用其他用例时用下划线标出。

用例38 使用订单处理系统

主成功场景：

1. 用户登录。
2. 系统显示可用的功能，用户选择其中之一执行：

设置订单
取消订单
获取状态
发送目录
登记投诉信息
运行售货报表

3. 重复以上动作直到用户选择了退出。
4. 当用户选择退出时从系统注销该用户。

250

第8章练习题

练习8-1

以下是一个失败条件的例子。一般来说，同学们能举出两倍于此的例子。注意所有的条件都是可检测的，都必须能进行处理。你是怎样做的呢？

读卡机坏了或卡被划坏。
该卡不能在此银行使用。
不正确的PIN。
顾客没有及时输入PIN。
ATM死机。
主机死机或网络瘫痪。
账户上余额不足。
顾客没有及时输入现金额。
不是5美元的倍数。
需要的现金量太大。

在执行事务时网络瘫痪或主机死机。
取款机中现金不足。
发放时出钞口被现金卡住。
收据条用完了或被卡住了。
顾客没有从取款机中取到现金。

练习8-5

用例39 通过万维网购买股票

主执行者：买主/用户
范围：PAF
层次：用户目标
前置条件：用户已经打开PAF。
最小保证：存在足够的日志信息，PAF能检测错误，并要求用户提供详细信息。
成功保证：远程站点已经对购买活动进行了确认；PAF登录，用户记录被更新。
主成功场景：
1. 用户选择在网上买股票。
2. PAF得到了要使用的站点的名称（E*Trade, Schwab,等等）。
3. PAF打开与站点的连接，保持控制。
4. 用户从Web站点浏览并购买股票。
5. PAF拦截来自站点的响应，更新用户资料。
6. PAF显示用户的新资料。

251

扩展：
2a. PAF不支持用户希望使用的站点：
2a1. 系统要求用户提供新的建议，或取消此用例。
3a. 在设置过程中，网络发生故障：
3a1. 系统向用户报告错误，并建议他退回到前一步。
3a2. 用户或者退出此用例，或者重新再试。
4a. 计算机系统崩溃或者在交易过程中被关掉
4a1. （这时，我们该怎么办？）
4b. 站点没有及时认可此次购买活动，而是把它推迟处理：
4b1. PAF把这次推迟事件记入日志；设置一个时钟，定期向用户查询结果。
4b2. (参见更新存有疑问的交易)
5a. 站点没有返回关于购买情况的必要信息：
5a1. PAF把信息缺少事件记入日志，要求购买者更新存有疑问的交易
5b. 在记录更新操作过程中，磁盘损坏或磁盘已满：
5b1. 重新启动，PAF检测日志中不一致信息，要求用户更新存有疑问的交易。

第11章练习题

练习11-1

用例40 执行清洁火花塞服务

前置条件：车进入车库，引擎仍运转。
最小保证：顾客被告知车子有更大的问题；车没有修。
成功保证：引擎工作良好。
主成功场景：
1. 打开引擎罩，并用保护性材料覆盖挡泥板。
2. 移去火花塞。
3. 擦掉火花塞上的油污。
4. 清洁并调节缺口。
5. 测试并证实火花塞工作。
6. 替换火花塞。
7. 连接点火线调正塞子。
8. 测试并保证引擎正常工作。
9. 清洁工具和设备。
10. 从挡泥板上移走保护性材料；擦去车上的任何油渍。

扩展：
4a. 塞子裂了或磨坏了：换一个新的火花塞。
8a. 引擎仍然不能正常工作：
  8a.1 诊断坏引擎（UC23）。
252 8a.2 通知用户车出现了更大的问题（UC41）。

附录C
术 语 表

主要术语

执行者（ACTOR） 带有行为的事物（能够执行一个条件语句）。它可能是一个机械系统、计算机系统、人、组织或这些的组合。

外部执行者（*external actor*）是在被讨论系统之外的执行者。

项目相关人员（*stakeholder*）是一个外部执行者，它有资格要求系统保护它的利益，为满足它的利益需要系统采取一些特殊的动作。不同的用例拥有不同的项目相关人员。

主执行者（*primary actor*）是要求系统实现某个目标的项目相关人员。通常但不是总是由主执行者发起与系统的交互。主执行者可能有一个中介来启动双方的交互或可能在某些事件发生时自动触发这种交互。

支持执行者（*supporting actor*）**和辅助执行者**（*secondary actor*）是与被讨论系统（SuD）的一个目标对应的系统。

幕后的执行者（*offstage actor*）**或第三位的执行者**（*tertiary actor*）是非主执行者的项目相关人员。

内部执行者（*internal actor*）或者是被讨论系统（SuD）的子系统，或者是被讨论系统的活动构件。

交互（INTERACTION） 一条消息、一个交互操作序列或一些交互序列的集合。

场景（SCENARIO） 场景是一个操作和交互的序列，它们在特定条件下发生，在表示上不使用条件语句或分支结构。

具体场景（*concrete scenario*）是一个定义了所有细节（执行者名称以及所涉及的其他值）的场景。它等同于用过去式描述一个包括了所有细节的故事。

应用叙述（*usage narrative*）或**叙述**（*narrative*）是一个揭示各种执行者动机和意图的具体场景。它被用作阅读或编写用例的准备活动。

在编写需求时，有时为执行者和数据值要使用诸如“客户”和“地址”之类的占位符来编写场景。当有必要将这些场景与具体场景区分开时，可以称这些场景为**一般场景**（*general scenario*）。

贯穿用例的一条**路径**（*path*）或一个用例的**执行过程**（*course*）是一般场景的同义词。

主成功场景（*main success scenario*）是一个完整地描述从触发到完成过程的场景，它包括实现目标的成功交付和开始后的所有记录。即使成功的路径不止一条，它也应是一个典型的、范例性的场景。

可选过程（*alternate course*）是作为主成功场景扩展的其他场景或场景片段。

执行步骤（*action step*）是场景中的一个编写单元。通常，一条语句仅描述一个执行者的行为。

场景扩展（SCENARIO EXTENSION） 从另一个场景中的特殊条件开始的一个场景片段。

扩展条件（*extension condition*）指定了不同行为发生的环境。

扩展用例（*extension use case*）是由一个特殊条件启动而中断其他用例的一个用例。被中断的用例称作**基用例**（*base use case*）。

扩展点（*extension point*）是基用例行为中某一点的一个标记符或别名，在这一点上，扩展用例可以中断基用例。为了使扩展用例能搜集所有相关的扩展行为（这些行为可以在一系列条件下中断基用例），扩展点可能定义了基用例中的一系列位置。

子用例（*sub use case*）是在场景的一个步骤中调用的用例。在UML中，称调用用例为**包含**（*include*）了子用例的行为。

用例（USE CASE） 用例表达了系统项目相关人员之间契约的行为部分。它描述了在各种条件下系统的行为和交互，就像系统代表一个项目相关人员（如主要执行者）响应请求一样，并显示主要执行者的目标是成功还是失败。用例将与主执行者目标有关的场景收集起来。

254

用例类型

焦点（FOCUS） 焦点不是集中在业务上，就是集中在新系统上：

业务用例（*business use case*）一词是一种简短的表示方式，它表明了用例强调业务操作而不是计算机系统的操作。在任何目标层上都可能编写业务用例，但是它们应被限定在企业或组织的范围内。

系统用例（*system use case*）一词是一种简短的表示方式，它表明了用例强调计算机系统或机械系统的操作而不是业务操作。可以在任何范围内和任何目标层上编写系统用例，包括业务范围。企业范围内的系统用例突出了被讨论系统在企业行为中的作用。

规范性（FORMALITY） 花费多少精力，采用什么精确度，规范性用于下述场合：

用例**简介**（*brief*）是一段用例纲要。

非正式用例（*casual use case*）采用一种简单的如实描述的文字段。它有可能忽略一些与用例有关的项目信息，很可能没有正式用例描述得那么严格。

完整的正式用例（*fully dressed use case*）按照一个完整的模板编写，定义了执行者、范围、层次、触发事件、前置条件和模板头信息的其余部分，加入了项目注释信息。

层次（LEVEL） 目标的高低：

概要级用例（*summary-level use case*）是要花费一段时间（可能是几周、几个月或几年）来完成多用户目标的一个用例。它的子用例可以是任何层次的用例。它通常的图形化表示是一朵云（  ）或一只风筝（  ）。“云朵”图符适用于包含云朵或风筝层步骤的用例。“风筝”图符适用于包含用户-目标步骤的用例。

用户-目标用例（*user-goal use case*）满足对主执行者有价值的一个特殊的或直接的目标。通常一个主执行者坐下来，在2～20分钟内就可以完成这个目标（如果主执行者是计算机，需要的时间更短），然后主执行者可以处理其他事情。各个步骤处于用户-目标级或更低的层次中。用户-目标用例可以采用“波浪”图符（ ）来表示。

子功能用例（*subfunction use case*）满足用户-目标用例或另一个子功能的部分目标。它的各个步骤都是低层次的子功能。它可以采用一条鱼（ ）或一只蛤（ ）的图符来表示。“蛤”图符表明用例所处的层次太低，根本不应该写出来。

255

范围（SCOPING） 被讨论系统有多大或有多小：

企业范围（*enterprise scope*）表明正在讨论的系统是一个组织或企业。用例模板的范围部分可以用组织、业务或企业名称来填充。根据黑盒用例还是白盒用例，分别采用灰色建筑（ ）和白色建筑（ ）以图形化的方式来标记用例。

系统范围（*system scope*）表明被讨论系统是一个机器/硬件/软件系统或应用程序。用例的范围部分用系统名称来填充。根据黑盒用例还是白盒用例，分别用灰盒（ ）或白盒（ ）以图形化的方式来标记用例。

子系统范围（*subsystem scope*）表明用例中被讨论系统是应用程序的一部分，可能是一个子系统或构架。用例的范围部分用子系统名称来填充。用例的图形化的表示是一个螺栓（ ）。

可见性（VISIBILITY） 用例中的哪些实体是可见的：

黑盒用例（*black-box use case*）不描述被讨论系统内部的任何构件。它通常用在系统需求文档中。

白盒用例（*white-box use case*）描述被讨论系统中构件的行为。它通常用在业务过程建模中。

图形

协作图（COLLABORATION DIAGRAM） 在UML中，协作图与顺序图表示了相同的信息，但采用的形式不同。执行者放置在图的周围，执行者之间带号码的箭头表示交互。仅仅通过箭头的数目来表示时间。

顺序图（SEQUENCE DIAGRAM） 在UML中，顺序图在顶部占用几列空间来放置执行者，列间的箭头表示交互，随着时间向每页的下方滑动。这对于以图形化方式展现一个场景很有用处。

用例图（USECASE DIAGRAM） 在UML中，用例图表示了外部执行者、系统边界、用椭圆表示的用例以及用来将执行者与椭圆或椭圆与椭圆连接起来的箭头。它主要用作语境图和目录表。

256

附录D
参考文献

本书参考图书目录

Beck, Kent. *Extreme Programming Explained.* Reading, MA: Addison-Wesley, 2000.

Cockburn, Alistair. *Surviving Object-Oriented Projects.* Reading, MA: Addison-Wesley, 1998.

Cockburn, Alistair. *Software Development as a Cooperative Game.* Boston: Addison-Wesley (due 2001).

Constantine, Larry, and Lucy Lockwood. *Software for Use.* Reading, MA: Addison-Wesley, 1999.

Fowler, Martin. *UML Distilled.* Reading, MA: Addison-Wesley, 1999.

Hammer, Michael, and James Champy. *Reengineering the Corporation, Reprint Edition.* New York: HarperBusiness, 1994.

Hohmann, Luke. *GUIs with Glue* (in preparation as of July 2000).

Robertson, Suzanne, and James Robertson. *Mastering the Requirements Process.* Reading, MA: Addison-Wesley, 1999.

Wirfs-Brock, Rebecca, Wilkerson, Brian, and Wiener, Lauren. *Designing Object-Oriented Software.* Upper Saddle River, NJ: Prentice-Hall, 1990.

本书参考文章目录

Beck, Kent, and Ward Cunningham. "A Laboratory for Object-Oriented Thinking," *ACM SIGPLAN* 24(10):1-7, 1989.

Cockburn, Alistair. "VW-Staging," at *http://members.aol.com/acockburn/papers/vwstage.htm.*

Cockburn, Alistair. "An Open Letter to Newcomers to OO," *http://members.aol.com/humansandt/papers/oonewcomers.htm.*

Cockburn, Alistair. "CRC Cards," at *http://members.aol.com/humansandt/papers/crc.htm.*

Cunningham, Ward. "CRC Cards," at *http://c2.com/cgi/wiki?CrcCards.*

Kraus, Andy, and Michael Dillon. "Use Case Blue," *Object Magazine*, SIGS Publications, May 1996.

Lilly, Susan. "How to Avoid Use Case Pitfalls," *Software Development* 8(1):40–44, 2000.

McBreen, Peter. "Test Cases from Use Cases," at *http://www.mcbreen.ab.ca/papers/TestsFromUseCases.html.*

有用的在线资源

Web上有大量的信息。你可以从这几个网址开始。

http://www.usecases.org

http://members.aol.com/acockburn

http://www.foruse.com

http://www.pols.co.uk/usecasezone/

索　引

索引中的页码为英文原书的页码，与书中边栏的页码一致。

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

X

对用例进行通过/失败测试

所有问题都应该有一个肯定的答案。

域	问　题
用例标题	1. 是用主动动词短语命名主执行者的目标吗？ 2. 系统能完成这个目标吗？
范围与层次	3. 这些域填写了吗？
范围	4. 用例是把范围内的系统作为一个“黑盒”对待吗？（如果是系统需求文档，那么答案是“是”；如果用例是白盒业务用例，那么答案可能是“不”。 5. 如果对范围内的系统进行设计，设计者是只设计范围内的每件事，而不管范围之外的事吗？
层次	6. 用例的内容与所描述的目标层次匹配吗？ 7. 目标是在所描述的目标层次上吗？
主执行者	8. 他/她/它有行为吗？ 9. 他/她/它有目标与被讨论系统的服务承诺矛盾吗？
前置条件	10. 它们是强制的吗？它们能在某些地方被所讨论的系统设置吗？ 11. 在用例中它们真的从来不需要检查吗？
项目相关人员及其利益	12. 他们被指名了吗？系统一定能满足他们所描述的利益吗？（用法随规范性和灵活性而定）
最小保证	13. 所有项目相关人员的利益被保护了吗？
成功保证	14. 所有项目相关人员的利益被满足了吗？
主成功场景	15. 它有3～9个步骤吗？ 16. 它能从触发事件到交付成功保证运行吗？ 17. 在执行过程中，它允许适当地改变吗？
在任何场景中的每个步骤	18. 它描述了一个成功目标吗？ 19. 在它完成后，整个过程明显地向前移动了吗？ 20. 是否清楚地指出哪个执行者正在操作目标？——谁在“踢球”？ 21. 执行者的意图是否清楚？ 22. 该步骤的目标层是否低于整个用例的目标层？它是恰好比用例目标层低一点吗？ 23. 你确定该步骤没有描述系统用户接口的设计吗？ 24. 在该步骤内所传递的信息是否清楚？ 25. 它的“确认”与“检测”条件相匹配吗？
扩展条件	26. 系统能够并且必须发现和处理它吗？ 27. 它是系统确实需要的吗？
技术与数据变动列表	28. 你能否确定这不是主成功场景的一个普通行为扩展？
整个用例的内容	29. 对投资者和用户：“这是你们想要的吗？” 30. 对投资者和用户：“在交付时，你们能识别出这是你们所要的吗？” 31. 对开发者：“你们能实现它吗？”