PEARSON Prentice Hall

UML精粹 第3版

标准对象建模语言简明指南

(美) Martin Fowler 著

徐家福 译

清华大学出版社

ISBN 7-302-10850-1

9 787302 108504 >

定价：29.00元

http://www.pearsoned.com

UML精粹

第3版

标准对象建模语言简明指南

(美) Martin Fowler 著

徐家福 译

清华大学出版社

北京

图书在版编目(CIP)数据

UML 精粹：标准对象建模语言简明指南：第 3 版/(美)福勒(Fowler，M.)著；徐家福译.—北京：清华大学出版社，2005.5

书名原文：UML Distilled：A Brief Guide to the Standard Object Modeling Language

ISBN 7-302-10850-1

Ⅰ.U… Ⅱ.①福… ②徐… Ⅲ.面向对象语言，UML—程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字(2005)第 036610 号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦

http://www.tup.com.cn　　**邮　　编**：100084

社 总 机：010-62770175　　**客户服务**：010-62776969

责任编辑：薛　慧

印 刷 者：北京鑫丰华彩印有限公司

装 订 者：三河市春元印刷有限公司

发 行 者：新华书店总店北京发行所

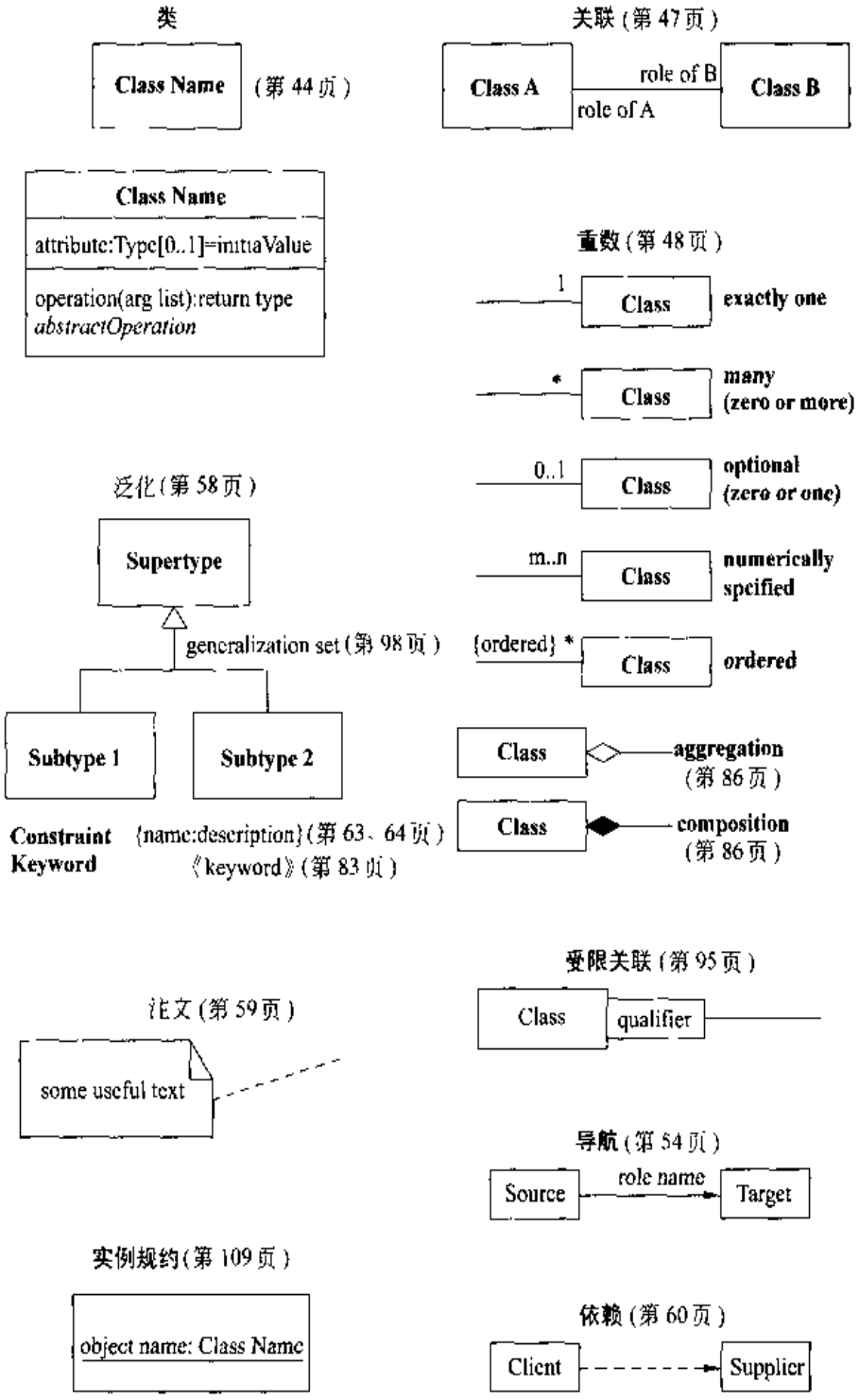
开　　本：185×230　**印张**：16.75　**字数**：186 千字

版　　次：2005 年 5 月第 1 版　2005 年 5 月第1 次印刷

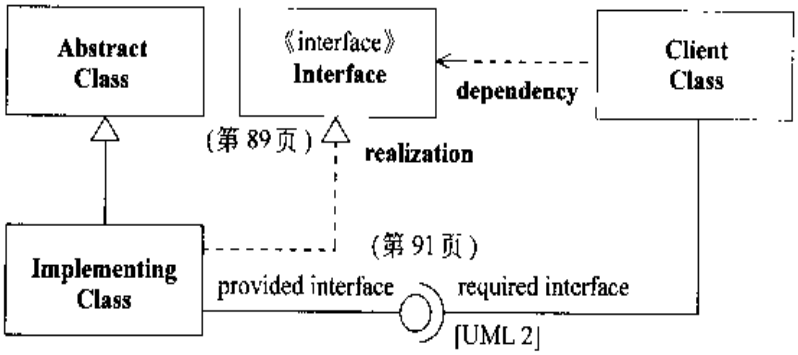
书　　号：ISBN 7-302-10850-1/TP・7215

印　　数：1～5000

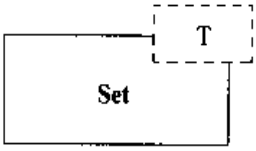
定　　价：29.00 元

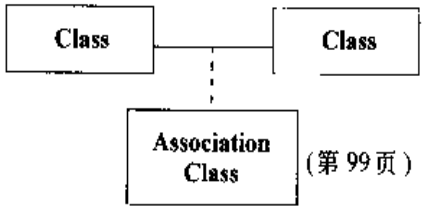

类
Class Name
(第 44 页)
Class Name
attribute:Type[0..1]=initiaValue
operation(arg list):return type
abstractOperation
泛化(第 58 页)
Supertype
generalization set (第 98 页)
Subtype 1
Subtype 2
Constraint {name:description}(第 63、64 页)
Keyword 《keyword》(第 83 页)
注文(第 59 页)
some useful text
实例规约(第 109 页)
object name: Class Name
关联(第 47 页)
Class A
role of B
role of A
Class B
重数(第 48 页)
1
Class
exactly one
*
Class
many
(zero or more)
0..1
Class
optional
(zero or one)
m..n
Class
numerically
specified
{ordered} *
Class
ordered
Class
aggregation
(第 86 页)
Class
composition
(第 86 页)
受限关联(第 95 页)
Class
qualifier
导航(第 54 页)
Source
role name
Target
依赖(第 60 页)
Client
Supplier

类图



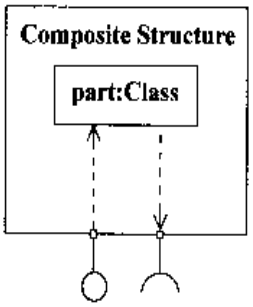
模板类 (第 103 页)





(第 99 页)

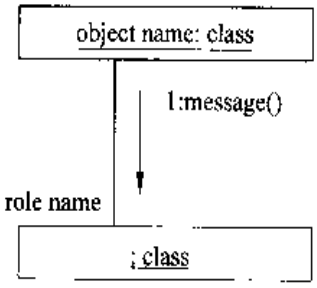
bound element (第 104 页)



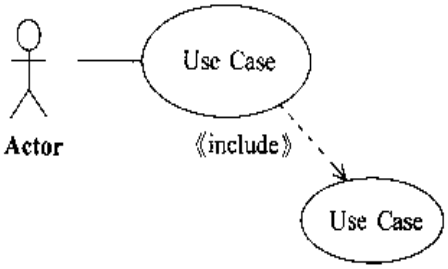
(第 164 页)

(第 167 页)

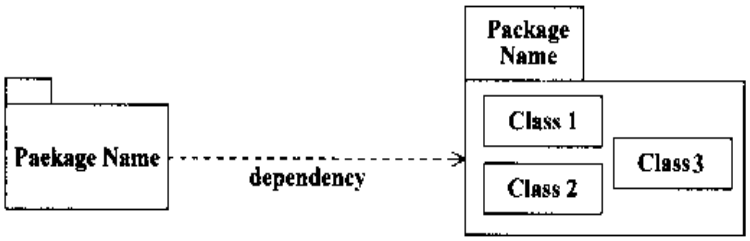
通信图 (第 160 页)



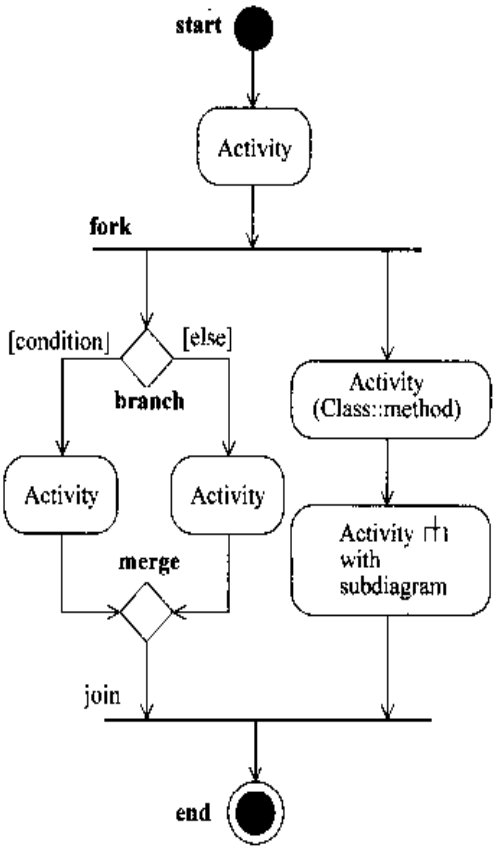
用案图 (第 122 页)



包图第 111 页

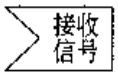


活动图第 143 页



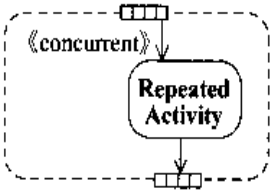
流终（第 155 页）

时间信号

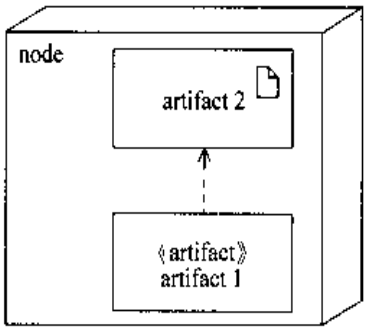


信号
（第 150 页）

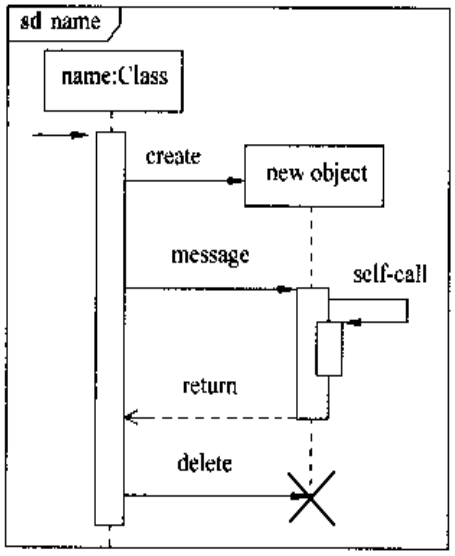
展开区域（第 154 页）

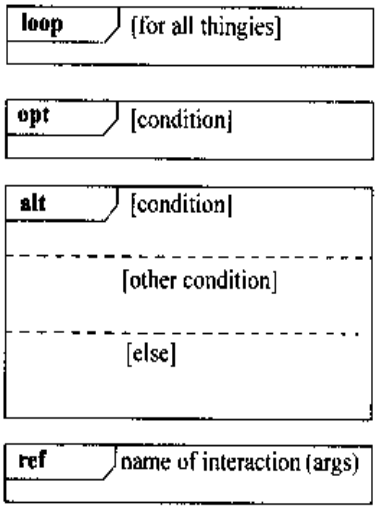


部署图（第 119 页）



顺序图（第 68 页）





（第 73 页）

synchronous（第 79 页）

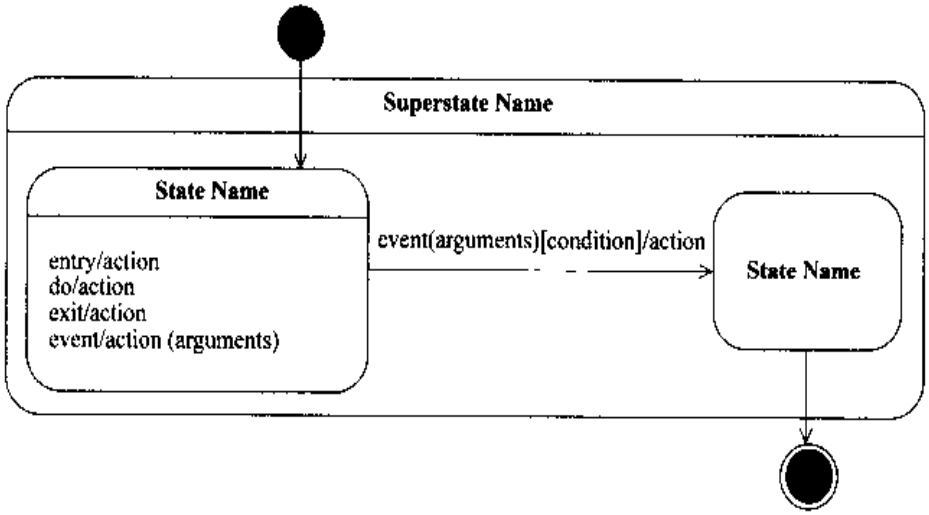
asynchronous[UML>=1.4]

asychronous[UML<=1.3]

*:iteration message()
[condition] message()

[UML 1]（第 76 页）

状态图（第 131 页）



第3版译序

古今中外，举凡传世之作，莫不目标明确，逻辑谨严，内容精辟，言简意赅，尤有作者见地，力戒材料堆积，人云亦云。Matin Fowler 所著《UML 精粹》一书自1997年首版问世以来，深受读者欢迎，堪称书中上乘。数年来，内容与时俱进，不断更新，继第2版之后，去岁九月，第3版出，览读之余，感其特点如次：

一、内容以 UML 2.0 为基础，引进若干新图型，如交互概观图、定时图、复合结构图。此外，对类图、顺序图、状态图、活动图等均有更新。

二、继续遵循"俭省原则"，力求以极小篇幅，讲述重要内容，而且理气兼备，有血有肉。对类图之基础部分、顺序图、用案等尤为精到，其余内容，亦称简明扼要。

三、作者观点明确，有其独到见地。在 UML 用作草图绘制语言、蓝图绘制语言、编程语言三种使用方式中，作者特别推重用于草图绘制，并认为，旨在嫁接活动图与顺序图之交互概观图之构思未臻佳境。

四、作者使用对话文体，讲述技术内容，实用与基理并重，辅以典型实例，便于加深理解，文笔幽默清新，颇能引人入胜。

余自今岁五月阅读原书，六月启译，虽数易其稿，然限于水平，错讹欠妥之处，恳请业界贤达，不吝赐正。

第 2 版之中译本于 2002 年出版以来，曾接不少读者来函，特别是老友周锡令教授提出有益建议，特致谢意。

清华大学出版社老友张兆琪编审、薛慧编辑大力支持，于此均表谢忱。

徐家福

甲申夏月于金陵不阿斋

第 2 版译序

统一建模语言 UML 乃软件设计与需求规约语言。论述语言之优劣，有用户、设计、实现等观点。这些观点既有区别，又有联系。UML 问世以来，褒贬不一，但其应用广泛，成效显著，实为颇具代表性之建模语言。

Martin Fowler 与 Kendall Scott 合著《UML 精粹》一书乃介绍 UML 上乘之作。出版数载，刊印 10 余次，为国际 IT 业界之畅销读物，究其因，实非偶然。研读之余，感其数善存焉。

一曰精粹提炼。UML 为“三友”Booch，Jacobson，Rumbaugh 各自方法之融合，内容丰富。作者爬梳剔精，区分主次，对多数用户经常使用之成分（如用案、类图、交互图等）讲深讲透，对其他重要成分（如状态图、活动图、物理图等）亦予以简明介绍，选材精益求精，达其精粹之境。

二曰学以致用。本书名为 UML 精粹，本身与过程无关。然读者如不熟谙其语境之过程，对语言成分亦难理解深透。故作者特在书首（第 2 章）介绍开发过程概要，俾读者能了然语言成分用于何时何处。此外，作者结合示例，讲解语言成分，理论联系实际，易学易用。并

在书末 UML 与编程一章(第 11 章)就健康护理之“汇总病人资料系统”,讲述自身使用 UML 之经验,学用结合,学以致用。

三曰经验结晶。作者积多年软件开发、特别是软件建模之经验,介绍语言成分,重在讲清道理,谙其底蕴,比较得失,提出个人见解,分析演化,融会而贯通之。此书实乃作者经验之结晶。

四曰文笔清新。作者文笔朴实无华,顺畅清新,逻辑谨严,由表及里,层次分明,引人入胜。览读是书,诚为一大享受。

谨向有志于 UML 之软件人员及广大计算机工作者推荐此书。

余搁笔翻译逾 40 年。今夏在京与旧友新知研讨 UML,览者均感此书内容形式均佳,值得译成中文,建议由余执笔。返宁后揣摩再四,遂定启译,晨夕用笔,兼月而初稿成。虽经四易其文,难免疏漏欠当,恳请业中贤达,不吝赐正。

北京航空航天大学周柏生教授,北京大学邵维忠教授、王立福教授等在术语译名与内容释疑等方面均鼎力襄助。清华大学出版社老友张兆琪编审、薛慧编辑等大力支持,于此均致谢忱。

徐家福

辛巳仲秋于金陵不阿斋

第3版前言

自古以来，最有才干的建筑师以及最杰出的设计师都确信“俭省定律”(law of parsimony)。不管它是陈述成一种悖论(“少即是多”)，还是陈述成一种公案(“禅心即本心”)，其智慧是永恒的。将任何事都简化到本质，使形式与功能相谐。从金字塔到悉尼歌剧院，从冯·诺依曼体系结构到UNIX及Smalltalk，最佳建筑师与设计师们都力求遵循这一普遍的永恒原则。

认识到利用奥卡姆剃刀(Occam's Razor)剃须的意义，当我在设计与阅读时，总在寻找一些遵循俭省定律的项目和书籍。因此，我对你现在正在阅读的这本书赞赏有加。

你可能起先对我的上述评论感到惊异。我经常接触到定义统一建模语言UML的冗长而难懂的规范。这些规范使工具卖主得以实现UML并使方法学学者得以应用UML。七年来，我主持了一些大型国际标准化组，制定了UML 1.1与UML 2.0的规范以及其间若干次较小的修订。这段时期，UML在表现能力与精确程度上已臻成熟，但是，由于标准化过程，也增加了不必要的复杂性。遗憾的是，标准化过程乃是因其“按委员

会设计"的折中妥协而不是因其俭省典雅而闻名于世的。

熟悉规范的晦涩难解之细微末节的 UML 专家能从 Martin 对 UML 2.0 的精粹提炼中学到什么呢？可以学到很多。一开始，Martin 就巧妙地将一种庞大复杂的语言简化成一个颇重实效的子集，后者在实践中已被证明效果不错。他摒弃了对该书的前一版增补篇幅的简易途径。由于这种语言的发展壮大，Martin 一直坚持其寻求"UML 最为有用的部分"的目标，并且所讲述的也正是这一部分。他所指的这一部分就是帮助你做 80%工作的 UML 中神话般的 20%部分。俘获并驯服这匹在逃的野兽乃是一项了不起的成就！

更令人钦佩的是，Martin 是以一种精彩动人的对话文体来实现这一目标的。通过与我们共享他的意见及趣闻轶事，他使本书成为一本有趣的读物，并且使我们联想到构建与设计系统应是既有创见又富有成效的。如果我们追求俭省公案到尽善尽美的地步，就应发现，用 UML 对项目建模就犹如在小学或中学中发现指画班与绘画班那样令人愉快。UML 应是我们创造性的一支照明杖，以及"精确指明系统蓝图以致第三方可以索价并构作系统"的一台激光器，后者乃是对任何一种真正蓝图语言的酸性检验。

因此，虽然这可能是一本小书，它却不是一本平凡的书。你可以从 Martin 的建模途径学到的东西几乎和你从他对 UML 2.0 的解释所学到的东西一样多。

在和 Martin 一道工作，以增进本次修订所阐明的 UML 2.0 的语言特征的选取与纠错上使我感到愉快。我们必须记住，所有有生命的语言(自然语言与人工语言)都必然发展或消亡。Martin 的关于新特征的选择连同你的爱好以及其他实践者的爱

好，乃是 UML 修订过程中的重要部分。它们使这种语言保持生气勃勃，并帮助它通过市场中的自然选择而发展。

在模型驱动开发成为主流以前还有很多挑战性的工作要做，但是我却因像这样一本清晰阐明 UML 建模基本原理并进行实际应用的书籍而感到鼓舞。我希望你和我一样能从中获益，并利用新的深刻了解来改进自己的软件建模实践。

Cris Kobryn
U2 伙伴的 UML 2.0 提交组主席
Telelogic 公司首席技术专家

第1版前言

在开始制订统一建模语言时，曾希望能产生一种表示设计的标准方式，它不仅能反映最佳工业实践，还能帮助打消软件系统建模过程的神秘。我们相信，使用标准建模语言定能鼓励更多开发人员在构作软件系统之前对系统建模。UML的迅速广泛采用表明软件开发界人士确已深知建模的好处。

UML的创建本身为一迭代与渐进过程，这和大型软件系统的建模颇为相似。最终结果成为一标种准，它建立在面向对象界很多个人和公司提出的想法和所做贡献的基础之上，并且也反映了这些想法与贡献。我们开始致力于UML，但很多别人帮助我们臻于成功。对他(她)们的贡献特致谢意。

创建并商定一种标准建模语言本身为一重要挑战。如何教育软件开发界，并以一种既易理解又按软件开发过程的方式来介绍UML亦为一重要挑战。在这本容易被人误解、且更新到反映UML最新变动的小书中，Martin Fowler遇到了更多的挑战。

Martin不仅以一种清晰而友善的文体介绍了UML的关键方面，而且还清楚阐明了UML在开发过程中所

起的作用。阅读中,我们享受到了 Martin 从 12 年以上的设计与建模经验中所得到的大量宝贵的建模见识和智慧。

结果是一本引导成千上万开发人员进入 UML、并促使他(她)们更热衷于以这种当前标准建模语言进一步开拓很多建模好处的书。

谨向任何一位有兴趣于首先览读 UML、并对它在开发过程中所起的关键作用有一概貌了解的建模人员或开发人员推荐此书。

Grady Booch
Ivar Jacobson
James Rumbaugh

序

在我的人生中有很多方面都是幸运的。我的一次鸿运是于1997年在合适的地点以合适的知识撰写了本书的第1版。溯及那时，面向对象建模的混乱天地刚刚开始在统一建模语言(UML)的旗帜下统一起来。从那时起，UML已成为不只是对象的标准而已是软件图示建模的标准。我的幸运是，这本书成为一本最流行的UML书，销售量超过25万册。

好！这对我来说固然不错，但你是否应该购买这本书呢？

我要强调的是，这是一本篇幅简短的书。它不指望给出这些年来日益发展的UML每一方面的细节。我的意图是去发现UML中最为有用的部分，并且只对你讲述这一部分。虽然一本篇幅较大的书会使你了解更多的细节，但也会占用你更多的阅读时间。而时间是你对一本书的最大投资。为了保持这本书篇幅不大，我曾经花时间去选择最佳的部分，这样你就不必去自行选择了。(令人难过的是，“更小”并不总是意味着“更便宜”，产生一本高质量的技术书是有明确固定成本的。)

购买本书的理由之一是要开始学习UML。由于这

是一本小书,它会使你很快抓住 UML 的基本要点。掌握了这些内容以后,就可以通过一些篇幅较大的书(如《用户指南》[Booch, UML user]或《基准手册》[Rumbaugh, UML Reference])来研究 UML 的更多细节。

本书亦可用作 UML 的最常用部分的方便参考书。虽然本书并未涵盖 UML 的所有部分,但比起很多别的 UML 读物,它却是一本颇为轻便、易于携带的书。

它也是一本作者有自己见地的书。我已经在对象领域工作了很长一段时间,因而,对什么工作起来得心应手、什么不然,有明确的想法。任何一本书都反映作者的意见,我也不打算隐瞒自己的看法。因此,如果你在找寻带有客观性的东西,你就要去尝试别的书。

虽然很多人曾经告诉我,本书是一本关于对象的良好导论,但这却非我的撰写本意。如果你要找寻一本面向对象设计的导论,我推荐 Craig Larman 的书[Larman]。

很多对 UML 感兴趣的人都利用工具。本书集中考虑 UML 的标准以及通常用法,并未涉及各种工具所支持的细节。虽然 UML 消解了"前 UML"表示法的巴别(Babel)塔,但在绘制 UML 图时,仍然在工具示明什么和允许什么之间有一些令人烦恼的差异。

在本书中我没有过多谈到模型驱动体系结构(MDA)。虽然很多人把这两者(指 UML 和 MDA——译注)看作一回事,但很多开发人员利用 UML 而对 MDA 却并无兴趣。如果你要多学一些 MDA,我就应该开始先用这本书使你能了解到 UML 的梗概,然后再转去阅读一本更为专门的 MDA 的书。

虽然本书的要点是 UML，我也增加了少许像 CRC 卡那样对面向对象设计有价值的别的技术材料。UML 只是你用对象取得成就所需要的一部分，而我认为，向你介绍一些别的技术，还是重要的。

在这样一本简短的书中，不可能涉及到有关 UML 如何与源代码联系的问题，特别是，尚无一种标准方式去进行这样的对应。但是，我要指出一些实现 UML 片段的常用编码技术。我的代码例子是用 Java 与 C# 书写的，这是因为我发现这些语言通常是最被广泛了解的语言。不要假定我喜欢 Java 和 C#，我用 Smalltalk 写的代码太多了！

为何对 UML 操心？

我们使用设计图示法已经有了一段时间。对我来说，其主要价值在于交流与理解。一个好的图往往能帮助你交流设计想法，特别是在你想要避免很多细节时。图也可以帮助你理解一个软件系统或一个业务过程。作为试图明白一些事情的开发组一方，图既能帮助你理解又能帮助你把这些理解在开发组内交流。虽然设计图示法不是（至少还不是）正文编程语言的接替，它们却是一个有益的助手。

很多人相信，在将来，图示技术在软件开发中将起统率作用，我却对此比较怀疑，但是，鉴别这些表示法能做什么，不能做什么，则肯定是有用的。

在这些图示法中，UML 的重要性源于它在面向对象开发界

的广泛使用与标准化。UML 已经成为不只是面向对象界内部的主要图示法,而且在非面向对象界也是一种流行的技术。

本书的结构

第 1 章是对 UML 的一个简介:UML 是什么,对不同的人它所具有的不同含义,以及它源于何处。

第 2 章谈软件过程。虽然软件过程完全独立于 UML,我认为重要的是,了解过程以便看出像 UML 那样语言的来龙去脉。特别是,重要的是了解迭代开发的作用,对面向对象界的很多人来说,迭代开发是一种基本的过程途径。

我是围绕 UML 内的各种图型来组织本书的其余部分的。第 3 章和第 4 章讨论 UML 两个最有用的部分:类图(核心)与顺序图。即使本书很小,我相信,利用我在这几章中所讨论的技术,你可以从 UML 中得到最大的好处。UML 是一头日益增长的大型野兽,但是你却不需要它的全部。

第 5 章谈论类图的一些比较次要但仍有用的部分的细节。第 6 章到第 8 章表述进一步显露系统结构(structure)的三种有用的图:对象图、包图以及部署图。

第 9 章至第 11 章示明另外三种有用的行为(behavioral)技术:用案、状态图(虽然正式称作状态机图,它们一般称为状态图)以及活动图。第 12 章至第 17 章非常简短,它们描述了一些不太重要的图,对这些图我只提供了一个快速的例子与解释。

本书前后插页综述图示法最常用的部分。我常听人说,它们

是本书最重要的部分。可能你会发现，在你阅读本书某些其他部分时引用它们十分方便。

第3版的变动

如果你有本书的先前几版，也许你想要知道不同之处是什么，更重要的是，是否应该购买新版。

促使我撰写并出版第3版的主要动力是UML 2的出现。UML 2增加了很多新材料，包括一些新图型。即使是熟悉的图也有不少新的图示法，例如，顺序图中的交互架构。如果你想要知道发生了什么，而又不想费力去读完规范（我肯定不会建议你这样做！），本书应该给你一个好的概述。

我已趁此机会完全改写了本书的大部分内容，从正文到例子都使之进入最新状态。我吸收了很多那些过去几年来在讲授与使用UML中所学到的东西。因此，虽然这本超薄的UML书的精神未曾触动，但很多话语都是新的。

几年来我努力工作以使本书尽可能成为内容最新的书。由于UML经历了变动，我尽最大努力与之齐头并进。本书是以UML 2草案为基础的。该草案已于2003年6月被有关委员会接受。在那次投票和更正式的投票之间不太可能出现进一步的变动，因此，我感到，对于付印我的这一修订本来说，UML现在已经足够稳定。任何进一步更新的信息我将在我的网站（http://martinfowler.com）上公布。

致谢

多年来,很多人都居于本书成功的一方。我首先要感谢的是 Carter Shanklin 和 Kendall Scott。Carter 是 Addison-Wesley 的编辑,是他建议我写这本书的。Kendall Scott 帮助我整理先前两版,仔细检查了正文和图形。他们一起在不可想象的短期内完成了难以完成的第 1 版的出版工作,而且保持了人们对 Addison-Wesley 所期待的高质量。他们还在 UML 看来什么都未稳定的早期保持摆脱变动的困境。

Jim Odell 是我事业早期很多方面的导师与指路人。他还深人参与到解决持有已见的方法学学者们在技术问题和个人问题上的分歧并达成共同标准。他对本书的贡献是深邃的,而且也是难以估量的。我敢说,他对 UML 也同样如此。

UML 是一个标准的产物,但我对标准体却有点过敏。因此,为了了解正在进行什么,我需要有一个密探网,他们能使我了解到各种委会员的全部秘密策划的最新动态。如果没有包括 Comrad Bock, Steve Cook, Cris Kobryn, Jim Odell, Guus Ramackers 以及 Jim Rumbaugh 等这样一些密探,我可就完了。他们全都给了我有益的指点并回答我一些无聊的问题。

Grady Booch, Ivar Jacobson 以及 Jim Rumbaugh 以“三友”(Three Amigos)而闻名。虽然几年来我曾经对他们开过一些玩笑,他们对本书却给了我很多支持与鼓励。永远不要忘记,我的刺激通常都源于珍视的赏识。

审阅人是一本书质量的关键。Carter 告诉过我，审阅人永不嫌多。本书前几版的审阅人是 Simmi Kochhar Bhargava，Grady Booch，Eric Evans，Tom Hadfield，Ivar Jacobson，Ronald E. Jeffries，Joshua Kerievsky，Helen Klein，Jim Odell，Jim Rumbaugh 以及 Vivek Salgar。

第 3 版也有一组优秀审阅人：

Conrad Bock
Andy Carmichael
Alistair Cockburn
Steve Cook
Luke Hohmann
Pavel Hruby
Jon Kern
Cris Kobryn
Craig Larman
Steve Mellor
Jim Odell
Alan O'Callaghan
Guus Ramackers
Jim Rumbaugh
Tim Seltzer

所有这些审阅人都花时间阅读书稿，其中每人至少发现一个令人尴尬的可笑错误。谨向他们全体致以诚挚的谢意。剩下来的任何错误完全是我的责任，发现错误后，将在网站 martin fowler. com 的图书部分勘误表页上公布。

设计并撰写 UML 规范的核心组是 Don Baisley，Morgan Björkander，Conrad Bock，Steve Cook，Philippe Desfray，Nathan Dykman，Anders Ek，David Frankel，Eran Gery，Øystein Haugen，Sridhar Iyengar，Cris Kobryn，Birger Møller-Pedersen，James Odell，Gunuar Övergaard，Karin Palmkvist，Guus Ramackers，Jim Rumbaugh，Bran Selic，Thomas Weigert 以及 Larry Williams。如果没有他们，我就没有什么值得大写特

写的了。

Pavel Hruby 开发了一种绝妙的 Visio 模板。我在 UML 图中用到很多该模板;你可以在 http://phruby.com 处找到相关信息。

很多人在网上与我联系,亲自提出建议和问题,并指出错误。我未能和他们所有的人取得联系,但我的感谢也是诚挚的。

我钟爱的技术书店是位于马萨诸塞州伯林顿的 SoftPro,该店的店员让我在那里度过了很多小时,浏览书架以发现人们是如何实际使用 UML 的,而且当我在那里时还为我提供上等的咖啡。

对第 3 版,组稿编辑是 Mike Hendrickson。Kim Arney Mulcahy 负责项目进度、版式设计以及绘图。Addison-Wesley 的 John Fuller 是生产编辑,而 Evelyn Pyle 与 Rebecca Rider 负责本书的文字编辑与校对。我对他们均致谢意。

在我坚持不懈写书时,Cindy 一直和我在一起。那时她在花园中耕耘以取得收益。

我的父母使我有良好的教育开端,从此,其他一切都涌现而出。

Martin Fowler
Melrose, Massachusetts
http://martinfowler.com

目 录

第 3 版译序 1
第 2 版译序 3
第 3 版前言 5
第 1 版前言 9
序 11
为何对 UML 操心? 13
本书的结构 14
第 3 版的变动 15
致谢 16
第 1 章 引言 1
何谓 UML? 1
UML 的使用方式 2
UML 发展简史 9
图示法与元模型 12
UML 图 14
何谓合法 UML? 17
UML 的含义 19
UML 并非足够 19
何处着手使用 UML 21

何处找寻更多资料 22
第 2 章 开发过程 23
迭代过程与瀑布过程 24
预见性计划制订与适应性计划制订 28
敏捷过程 31
Rational 统一过程 32
过程适配项目 33
UML 适配过程 36
需求分析 37
设计 38
文档 40
理解遗产代码 41
选择开发过程 42
何处找寻更多资料 42
第 3 章 类图：基础部分 44
特性 46
属性 46
关联 47
重数 48
特性的程序解释 50
双向关联 53
操作 56
泛化 58

注文与注释 59
依赖 60
约束规则 63
何时使用类图 66
何处找寻更多资料 67
第 4 章 顺序图 68
参加者的创建与删除 73
循环、条件等 73
同步调用与异步调用 78
何时使用顺序图 79
第 5 章 类图：高级概念 83
基词 83
职责 85
静态操作与静态属性 85
聚合与组合 86
寻出特性 88
接口与抽象类 89
只读与冻结 93
指引对象与值对象 93
受限关联 95
分类与泛化 96
多重分类与动态分类 97
关联类 99

模板(参数化)类 103
枚举 105
主动类 105
可见性 106
消息 107
第 6 章 对象图 108
何时使用对象图 109
第 7 章 包图 110
包与依赖 112
包面 115
包的实现 116
何时使用包图 117
何处找寻更多资料 118
第 8 章 部署图 119
何时使用部署图 121
第 9 章 用案 122
用案的内容 123
用案图 126
用案级别 128
用案与特征(或情节) 128
何时使用用案 129
何处找寻更多资料 130
第 10 章 状态机图 131

内部活动 134
活动状态 134
超态 136
并发状态 136
状态图的实现 138
何时使用状态图 141
何处找寻更多资料 141
第 11 章 活动图 143
动作的分解 146
分划 148
信号 149
权标 151
流与边 151
饰针与转换 152
展开区域 154
流终 155
汇合指明 156
此外尚有更多内容 157
何时使用活动图 158
何处找寻更多资料 158
第 12 章 通信图 160
何时使用通信图 162
第 13 章 复合结构 164

何时使用复合结构 166
第 14 章 构件图 167
何时使用构件图 169
第 15 章 协作 170
何时使用协作 173
第 16 章 交互概观图 174
何时使用交互概观图 174
第 17 章 定时图 176
何时使用定时图 178
附录 UML 各个版本间的变动 179
UML 的修订 179
《UML 精粹》中的变动 181
从 UML 1.0 到 UML 1.1 的变动 182
类型与实现类 182
完整与不完整判别元约束 183
组合 183
永恒与冻结 184
顺序图上的回送(返回) 184
术语"角色"的使用 184
从 UML 1.2(及 1.1)到 UML 1.3(及 1.5)的变动 185
用案 185
活动图 186
从 UML 1.3 到 UML 1.4 的变动 187

从 UML 1.4 到 UML 1.5 的变动 188
从 UML 1.x 到 UML 2.0 188
类图：基础部分(第 3 章) 189
顺序图(第 4 章) 189
类图：高级概念(第 5 章) 189
状态机图(第 10 章) 190
活动图(第 11 章) 190
参考文献 191
图索引 195
汉英对照术语索引 198
英汉对照术语索引 217

第1章

引　言

何谓 UML?

统一建模语言 UML 是由单一元模型支持的一组图示法。这些图示法有助于表述与设计软件系统，特别是采用面向对象(OO)方法构作的软件系统。这是一个稍微简化的定义。事实上，UML 却多少因人而异。这一方面是源于 UML 本身的历史；另一方面，是源于人们关于构作有效软件工程过程所持的不同观点。因此，本章的大量工作是通过解说人们看待与使用 UML 的不同方式来为本书的后续部分进行准备。

图示建模语言在软件工业中已经长期存在。其产生与发展的基本动力在于对促进有关设计的讨论来说，编程语言的抽象级别还不够高。

虽然图示建模语言存在已久，但在软件工业中对其作用却颇有争议。这些争议直接影响到人们如何看待 UML 的作用。

UML 是由对象管理组(OMG)管理控制的一种较为开放的标准。OMG 是由各家公司组成的一个开放联合组织。组建 OMG 的目的是构作支持互操作性(特别是面向对象系统的互操作性)的标准。也许,OMG 以 CORBA(公共对象请求代理体系结构)标准而享有盛名。

UML 是在经过统一 20 世纪 80 年代后期与 90 年代初期成长起来的很多面向对象图示建模语言而诞生的。自从它在 1997 年出现,便将特定的传统巴别(Babel)塔抛入历史。这是一项令我以及很多别的开发人员都深感欣慰的贡献。

UML 的使用方式

UML 在软件开发中作用的实质是人们不同的使用方式。这些不同的使用方式传承于其他图示建模语言。它们导致了关于应如何使用 UML 这一问题的长期而艰巨的争论。

为了解决这一纷争,Steve Mellor 和我独立提出了 UML 三种使用方式的分类法:用作草图绘制语言,用作蓝图绘制语言以及用作程序编制语言(即,编程语言,程序设计语言)。迄今,至少依我的偏见看,这三种方式中最为常用的是将 UML 用作**草图绘制语言**(UML as sketch)。在这一用法中,开发人员使用 UML 帮助系统中某些部分进行交往。和对蓝图绘制语言一样,你可以按正向工程方向或逆向工程方向来使用草图绘制语言。**正向工程**(forward engineering)在编写代码前绘制 UML 图,而**逆向工程**(reverse engineering)则根据现有的代码来绘制 UML 图,以助于

理解代码。

绘制草图的实质是选择。对正向工程中的草图绘制，你草拟出要编写的代码中的问题，通常要和你的项目组中的同事进行讨论。你们的目的是，使用草图去帮助交流你们要进行工作的想法和取舍，并不涉及要编制的所有代码，而只是那些打算先交给同事去做的或者想在开始编程以前能看到的一些设计段的重要问题。像这样的讨论会可能很短。十分钟的会议讨论几小时的编程，或者一天的会议讨论两周的迭代开发。

对于逆向工程，你利用草图去阐明系统的某一部分如何工作，并不去示明每一类，只是阐明那些在深入到代码以前有意义以及值得讨论的问题。

由于草图绘制是相当非形式的和动态的，你就必须按合作方式快速进行。因此，常用的介质是白板。草图在文档中也有用，在这种情形下，重点是进行交流，而不是苛求完备。用于绘制草图的工具是一些轻便的制图工具。往往人们并不过于遵循 UML 的严格规则，在一些书(像我的别的书)中所示的 UML 图都是草图，其重点在于有选择地交流而不是完备的规约。

与此截然不同的是，**用作蓝图绘制语言的 UML**(UML as blueprint)却强调完备。在正向工程中，蓝图是由设计人员开发的，设计人员的任务是为编程人员编写代码构作一个详细的设计规约，这种设计规约应该足够完备，所有的设计选定都要安排好，编程人员应能按此编程。这是一项相当简单的工作，无须太多的思考。设计人员和编程人员可以是同一个人，但通常设计人员则是更高层的开发人员，他为一组编程人员进行设计。这一途径是受到其他形式的工程的启发。在那些工程中专业工程师创作工

程图,再把这些工程图转交给建造公司去建造。

蓝图绘制可用于所有细节,或者,设计人员可对一特定领域绘制蓝图。通常的途径是,设计人员开发蓝图级的模型直到子系统的接口,然后,让开发人员产生出实现那些细节的细节。

在逆向工程中,蓝图的目的在于表达代码的详细信息,或者是用纸文档,或者是作为交互图形浏览器。蓝图可以用开发人员较易理解的图示形式来示明类的每一细节。

与草图相比,蓝图需要更多的先进工具,以处理任务所需的细节。专业化的 CASE(计算机辅助软件工程)工具就属于这一类工具,即使术语 CASE 已经成为一个令人讨厌的词,而且目前卖主已打算避免使用。正向工程工具支持绘图,并用一仓库支持它保存信息。逆向工程工具读源代码,并据此解释以进入仓库且生成图。既可进行正向工程又可进行逆向工程的工具称为双向(round-trip)工具。

有些工具使用源代码本身作为仓库并使用图作为代码上的图形视见区。这些工具和编程联系十分紧密,并且往往和编程编辑程序相集成。我喜欢把这些工具想象成无行程(tripless)工具。

蓝图和草图之间的界线有些模糊。但是,我认为区别在于,草图是故意不完备的,要突出重要的信息,而蓝图却打算使内容广泛,并往往具有这样一个目的,即将编程简化成一种简单且相当机械的活动。简言之,草图是探究性的,而蓝图是定义性的。

随着你在 UML 中的工作越来越多以及编程日益机械,变得明显的是,编程应该自动化。的确,很多 CASE 工具进行了某种形式的代码生成,后者使系统的相当一部分构建工作自动化。可是,最终你达到的是,整个系统可以用 UML 来指明,并且把 UML

用作程序编制语言(UML as programming language)。在这种环境下,开发人员绘制UML图,后者直接编译成可执行的代码,而UML成为源代码。显然,UML的这种用法特别要求使用先进工具(而且,正向工程与逆向工程概念对这种方式已不具任何意义,这是因为UML和源代码相同的缘故。)。

模型驱动体系结构与可执行UML

当人们谈到UML时,他们也往往谈到**模型驱动体系结构**(MDA)[Kleppe et al.]。本质上说,MDA是UML用作程序编制语言的一种标准途径。与UML一样,该标准是由OMG管理控制的。通过产生一种符合MDA的建模环境,卖主能够创建也可以在其他遵从MDA的环境中工作的模型。

往往在谈到MDA的同时也谈到UML,这是因为MDA使用UML作为其基本建模语言的缘故。但是,在使用UML时,并无须使用MDA。

MDA将开发工作分为两个主要方面。建模人员通过创建一个**与平台无关的模型**(Platform Independent Model, PIM)来表示一个特定的应用。PIM是一个与任何特定技术无关的UML模型。而后,使用工具将PIM转换成一个**平台特定的模型**(Platform Specific Model, PSM)。PSM是一个标定为特定执行环境的系统模型,这时,别的工具接受PSM,并为该平台生成代码。PSM可以是UML模型,但也不一定是UML模型。

因此,如果要利用MDA构作一个仓库系统,就要从创建一个仓库系统的PIM开始。如果这时要求该仓库系统在J2EE和.NET上运行,

就要利用一些卖主工具,分别创建两个 PSM,一个平台一个。这时,别的工具便为这两个平台生成代码。

如果由 PIM 到 PSM 再到最终代码的过程完全自动化,UML 便是编程语言。如果其中任何一步是由人手工进行的,UML 便是蓝图绘制语言。

Steve Mellor 长期活跃在这一工作领域,并且近来使用了**可执行 UML**(executable UML)这一术语[Mellor and Balcer]。可执行 UML 与 MDA 类似,但使用了稍微不同的术语。类似地,你也是从一个和 MDA 的 PIM 等价的与平台无关的模型开始,不过,下一步是利用一个模型编译程序把 UML 模型一步转换成一个可部署的系统;因此,便用不着 PSM。正如术语编译程序(compiler)所暗示的,这一步是完全自动化的。

模型编译程序是基于可复用的本型。**本型**(archetype)表述如何接受一个可执行的 UML 模型,并将它转换到一个特定的编程平台。因此,对仓库例子来说,你要购买一个模型编译程序和两个本型(J2EE 和 .NET),将每个本型在可执行的 UML 模型上运行,这样就有了仓库系统的两个版本。

可执行 UML 并不使用 UML 全集标准,UML 的很多构造都认为不是必需的,因而都未使用。这样,可执行 UML 就比 UML 全集简单。

所有这一切,看起来都不错,但其现实程度如何?依我看,这里有两个问题。第一个是工具问题。执行这一任务的工具是否足够成熟。这一问题因时而变。的确,在我写这本书时,这样的工具并未广泛使用,并且我也未曾看到它们当中多数在起作用。

一个更为基本的问题是 UML 用作编程语言的整体看法。依我看,将 UML 用作编程语言仅在如下情形才是值得的,即它会导致比使用其他编程语言更加明显富有成效的结果。基于过去我工作过的各种图示

开发环境，我未能确信它是如此。即使它更加富有成效，仍然需要拥有一大批用户，才能使它成为主流，这本身就是一大障碍。像很多Smalltalk老用户一样，我认为Smalltalk比当前一些主流语言更富有成效，但是，由于目前Smalltalk只是一种置入堡垒的语言，我未看到很多项目在使用它。为了避免出现Smalltalk的结局，UML必须是更为幸运的，即使它是优越的。

围绕UML用作编程语言的一个有趣的问题是如何为行为基理建模。UML提供三种行为建模方式：交互图，状态图以及活动图。所有这三种方式都有支持其编程的人。如果UML用作编程语言真的得到普及，那么，看出上述技术中哪一种成为成功的技术，将是很有意义的。

人们看待UML的另一方式就是用于概念建模和用于软件建模之间的幅度。很多人都熟悉UML用于软件建模。在这种**软件视面**(software perspective)中，UML成分相当直接地映射到软件系统中的成分。正如我们将要看到的，映射绝不是规范的，但当我们使用UML，便要谈到软件成分。

对于**概念视面**(conceptual perspective)，UML表示研究领域中概念的表述。这里，我们并不谈到软件成分，甚至在构作谈论特定领域词汇时也是如此。

关于视面并无严格规则：它一出现，就有多种用法。有些工具将源代码自动转换成UML图，将UML图看作源代码的另一种外貌。如果你利用UML图去努力理解一群会计师的资产池(asset pool)术语的各种含义，那你便更多地进入思维的概念架构中。

在本书的前几版中,我将软件视面分成规约(接口)视面与实现视面。实践中我发现很难在这二者之间画出一道严格界线。因此,感到不值得如此小题大做,进行区分。不过,我总是在图中倾向于强调接口,而不强调实现。

UML 的这些不同的使用方式引起了很多关于"UML 图的含义为何以及 UML 和外界的关系为何"的争议。特别是,它影响到 UML 和源代码之间的关系。有些人主张 UML 用于创建和用于实现的编程语言无关的设计,另一些人则相信,与语言无关的设计是牛一般的蠢事。

观点上的另一区别是什么是 UML 的精髓。依我看,UML 的多数用户,特别是草图绘制人员,把图看作是 UML 的精髓。然而,UML 的创建者们却把图看成次要的,他们认为元模型是 UML 的精髓,图只是元模型的表现,这一看法对蓝图绘制人员以及 UML 的编程用户来说,还是有意义的。

因此,当你阅读任何涉及 UML 的读物时,重要的是要了解作者的观点。只有如此,才能认识到 UML 激起的(往往是激烈的)争论的意义。

说到这一步,我需要澄清我的偏见。几乎所有时间,我都将 UML 用于草图绘制。我发现 UML 草图对正向工程和逆向工程都是有用的,并且在概念视面和软件视面中也都有用。

我不是一名正向工程师的详细蓝图的热烈爱好者,我相信,详细蓝图很难做好,而且它会使全部开发工作放慢。对某一级子系统接口绘制蓝图是合适的,但即使那时,当开发人员实现跨接口交互时,你也该指望改变那些接口。逆向工程师蓝图的价值与工具如何工作有关。如果它是用作动态浏览器,则可能很有帮

助；如果它生成大量文档，它所做的一切就是在砍树。

对于将 UML 用作编程语言，我认为这是一种好想法；但怀疑这样做的意义。我并不确信，对大多数编程任务而言，图示形式要比文字形式更富有成效；并且即使如此，对一种语言来说，也很难被广泛接受。

由于我的偏见，本书过多地着重于 UML 用于草图绘制。幸运的是，这对一本简明入门读物来说，却是有意义的。在这样小篇幅的书中，我不能平等对待 UML 的其他使用方式，却是其他可平等对待 UML 三种使用方式的书的一本好的引导读物。建议你将本书看作一本引导读物，并在需要时转去阅读其他书籍。如果只对草图感兴趣，本书很可能就是你的全部需要。

UML 发展简史

我承认自己是一名历史爱好者。我钟爱的轻松读物就是好的历史书。当然，我也知道，这并非所有人的娱乐方式。我在这里谈历史的原因是，我认为，在很多场合，如果不了解 UML 如何发展到这一步，也就难以理解 UML 现今居于何处。

20 世纪 80 年代，对象概念开始离开实验室，朝向“现实”世界迈出最初的步伐。Smalltalk 趋于稳定，成为人们可用的平台，并且诞生了 C++ 。那时，许多人都开始思考面向对象的图示设计语言。

面向对象图示建模语言的重要书籍出现于 1988 年到 1992 年间。领衔人物包括 Grady Booch[Booch, OOAD]，Peter Coad

[Coad, OOA], [Coad, OOD]; Ivar Jacobson (Objectory) [Jacobson, OOSE]; Jim Odell [Odell]; Jim Rumbaugh (OMT) [Rumbaugh, insights], [Rumbaugh, OMT]; Sally Shlaer and Steve Mellor [Shlaer and Mellor, data], [Shlaer and Mellor, states]; 以及 Rebecca Wirfs-Brock (Responsibility Driven Design) [Wirfs-Brock]。

这些作者中的每一位当时都非正式地领导着一个喜爱各自想法的实践人员小组。所有这些方法都很类似,但也含有不少往往令人讨厌的细微差异。相同的基本概念以不同的表示出现,从而引起客户混淆。

在那一段令人兴奋的时期,有人谈到标准化问题,也有人忽视。OMG 的一个小组试图考察标准化,但得到的只是来自所有方法学学者的一封公开抗议信。(这使我回想起一个古老的玩笑。问题:一名方法学学者与一名恐怖主义者的区别何在?答案是,你可以和恐怖主义者协商。)

最初促成 UML 的重大事件是 Jim Rumbaugh 离开通用电气公司,与 Rational 公司(现在是 IBM 的一部分)的 Grady Booch 联合。Booch/Rumbaugh 联盟一开始就被看成是可以占有大部分市场的联盟。Grady 和 Jim 宣称:"方法战已告结束——我们是赢家。"这基本上表明,他们要去实现 Microsoft 方式的标准化。一些别的方法学学者建议组织一个反 Booch 的联盟。

到 OOPSLA'95,Grady 和 Jim 已经准备好一份关于他们合并方法的公开表述:统一方法(Unified Method)文档 0.8 版本。更为重要的是,他们宣布,Rational 软件公司已经收购了 Objectory 公司,因而 Ivar Jacobson 要加入统一小组。Rational

公司盛宴庆祝0.8草案的发布。(宴会的高潮是Jim Rumbaugh首次在公众面前一展歌喉;我们都希望这也是最后一次。)

下一年出现了进一步公开的联合过程。曾经居于旁观地位的OMG此时起了积极作用。Rational公司不得不吸取了Ivar的想法,并和别的伙伴度过了一段时间。更为重要的是,OMG决定担当主角。

这时,重要的是认识OMG为何介入。方法学学者们和图书作者们一样,喜欢认为他们自己是重要的。但是,我并不认为,OMG甚至听到过图书作者们的喧嚷。促使OMG介入的乃是工具厂商的喧嚷,他们担心由Rational公司控制的标准会给Rational公司的工具不公平的竞争实惠。因此,厂商们敦促OMG在CASE工具互操作性的旗帜下有所作为。由于OMG是全力研究互操作的,所以这一旗帜是重要的。想法是,创建一个允许CASE工具自由交换模型的UML。

Mary Loomis和Jim Odell成为这支创始工作队的主席。Odell明确表示,他准备将他的方法让位给标准,但不要Rational公司强加的标准。1997年1月,各种组织提交了关于方法标准的建议方案,以便于互换模型。Rational公司和其他组织合作,发布了UML文档的版本1.0作为它们的建议方案,这是一件首次回答统一建模语言这一名称的非同寻常的事。

经过一段短期的斡旋,合并了各种建议方案,OMG采用了随之产生的1.1版作为一个正式的OMG标准。后来又作过几次修订。修订版1.2完全是修饰性的,修订版1.3比较重要。修订版1.4增加了一些关于构件与侧图的详细概念。修订版1.5增加了作用语义。

当人们谈到 UML 时，他们主要将 UML 归功于 Grady Booch，Ivar Jacobson 以及 Jim Rumbaugh，并将这三人看成 UML 的创建者。通常将他们称为三友(Three Amigos)，尽管一些能言善辩者喜欢省去第二个词(Amigos)的第一个音节。虽然三友对 UML 颇多建树，但我认为把他们看成举足轻重的首要人物，多少有些不公。UML 图示法首先是在 Booch/Rumbaugh 的统一方法中形成的。从那时起，大量工作都是在 OMG 委员会的领导下进行的。在这些稍后的阶段中，Jim Rumbaugh 是三友中惟一一位主要责任人。我的看法是，UML 委员会工作进程中的各位成员才算得上 UML 的主要功臣。

图示法与元模型

UML 以其当前的状态定义了一种图示法和一种元模型。**图示法**(notation)是你在模型中看到的图示材料；它是建模语言的图示语法。例如，类图图示法定义了如何表示类、关联以及重数等项目和概念。

当然，这就导致关联、重数甚至类的精确定义为何的问题。通常的用法举荐非形式定义，但很多人要求比非形式定义更为严格的定义。

严格的规约与设计语言的想法在形式方法领域最为盛行。在这些技术中，设计和规约都是用谓词演算的派生内容表示的。这样一些定义在数学上是严格的，并且没有歧义。不过，这些定义的价值却绝不是绝对的。即使可以证明一个程序满足一个数

学规约，也无法证明这个数学规约满足系统的实际需求。

大多数图示建模语言的精确程度都不高；其图示法的魅力在于直观，而不是形式定义。整体上说，这似乎并无大害。这些方法可以是非形式的，但很多人仍然发现它们有用，并且也认为它们有用。

不过，方法学学者们正在在不牺牲其有用性的前提下寻求提高严格性的方法。其中之一是定义一种元模型。**元模型**(meta-model)是一种用以定义语言概念的图(通常是类图)。

图1.1是UML元模型的一个片段，它示明特征之间的联系。(这一片段只是展示一下元模型的风格，我甚至未打算对它进行解释。)

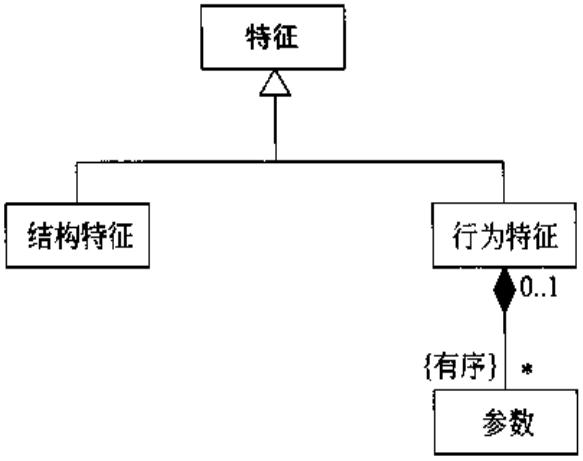


图1.1 UML元模型的片段

元模型对建模图示法的用户有多大影响？这一问题的答案主要取决于使用方式。草图绘制人员通常不太留意元模型，蓝图绘制人员应该予以更多的关注。对于将UML用作编程语言的人们来说，元模型就极其重要了，因为元模型定义了编程语言的

抽象语法。

介入 UML 发展进程的很多人主要对元模型感兴趣,原因是元模型对于 UML 的用法以及编程语言都很重要。图示法的问题往往居于第二位,如果你打算熟悉标准文档本身,那么,记住图示法就很重要了。

随着你深入到 UML 的更为详细的用法之中,就会认识到,你所需要的比图示法更多。这就是 UML 工具如此复杂的原因所在。

在本书中我不是严格的。我喜欢传统的方法途径并且主要靠直觉。对于由主要倾向于用作草图绘制的作者所写的这本小书来说,那就是自然的了。如果你需要更严格一些,应该去阅读更为详细的巨著。

UML 图

UML 2 表述了列于表 1.1 中的 13 种正式图型(diagram type),并在图 1.2 中进行了分类。虽然这些图型是很多人探讨 UML 的手段,并且本书也是据此安排的,UML 的作者们却不把图看作 UML 的主要部分。因此,图型并不特别精确,往往你可以将一种图型的成分合法地用于另一种图。UML 标准指明某种成分典型地画于某种图型上,但这并不是一种规定。

表 1.1 UML 的正式图型

图 名	章节	目 的	联 系
活动(activity)	11	过程行为与并行行为	UML 1 中
类(class)	3,5	类、特征与关系	UML 1 中
通信(communication)	12	对象间交互,着重连接	UML 1 协作图
构件(component)	14	构件的结构与连接	UML 1 中
复合结构(composite structure)	13	类的运行时刻分解	UML 2 的新图型
部署(deployment)	8	制品在结点上的部署	UML 1 中
交互概观(interactive overview)	16	顺序图与活动图的混合	UML 2 的新图型
对象(object)	6	实例的样形	UML 1 中非正式图型
包(package)	7	编译时刻层次结构	UML 1 中非正式图型
顺序(sequence)	4	对象间交互,着重顺序	UML 1 中
状态机(state machine)	10	事件如何改变生命期中对象	UML 1 中
定时(timing)	17	对象间交互,着重定时	UML 2 的新图型
用案(use case)	9	用户在系统中如何交互	UML 1 中

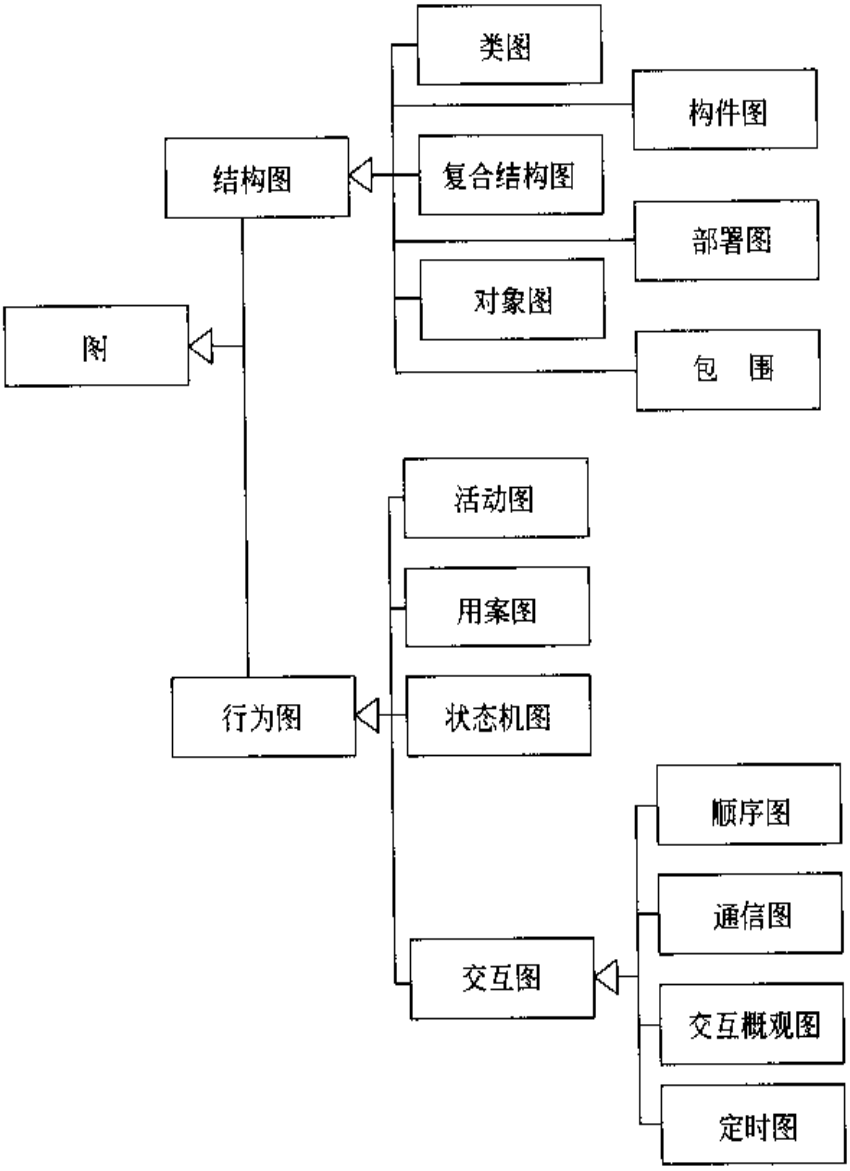

类图
构件图
结构图
复合结构图
部署图
对象图
图
包 围
活动图
用案图
行为图
状态机图
顺序图
通信图
交互图
交互概观图
定时图

图 1.2 UML 图型分类

何谓合法 UML?

骤然一看,这应该是一个待回答的简单问题:合法 UML 就是在规范中定义为组成良好的 UML。然而,在实践中,答案却稍为复杂一些。

这一问题的一个重要部分就是 UML 是否具有描述性规则或指定性规则。具有**指定性规则**(prescriptive rule)的语言是由正式机构管理控制的,这个机构陈述了语言中什么合法、什么不合法以及该语言中言辞的含义。具有**描述性规则**(descriptive rule)的语言却是由人们通过考察如何在实际中使用语言来理解其规则的。编程语言趋于具有指定性规则,后者是由标准委员会或大厂商制定的。像英语这样的自然语言却趋于具有描述性规则,其含义是约定俗成的。

UML 是一种非常精确的语言,因而,你可以指望它具有指定性规则。但是,在其他一些工程学科中,UML 往往被看作是蓝图的软件等价物,而这些蓝图并不是指定性的表示。没有一个委员会谈到结构工程制图方面什么是合法的符号,类似于对自然语言那样,图示法是习惯被接受的。就是真有一个标准机构,也不可能做这种事,这是因为领域中的人不可能遵循标准机构所声称的任何事;这正如向法国人询问法语学会一样(法语学会曾经制定了一些语言规则,但法国人一般是置之不理的——译注)。此外,UML 相当复杂,以致标准往往有多种解释,甚至审阅本书的 UML 领导者们也会不同意 UML 标准的解释。

这一问题对我写这本书以及对你使用 UML 都很重要。如果你要了解一张 UML 图,重要的是要认识到,了解 UML 标准并非问题的全局。在工业界,人们普遍遵循惯例,在特定项目中也是如此。因此,虽然 UML 标准可以是 UML 的主要信息来源,它却并非惟一的信息来源。

我的态度是,对大多数人来说,UML 具有描述性规则,UML 标准是关于"什么是 UML 的含义"的最具影响力的材料,但它并非惟一的材料。我认为,这一点对 UML 2 来说,确切无疑。UML 2 引进了一些表示上的习惯,这些习惯要么和 UML 1 的定义相抵触,要么和 UML 的习惯用法相抵触,同时对 UML 还增加了更多的复杂性。因此,在本书中,当我发现这些情况(标准及习惯用法)时,我将力图对 UML 进行概括说明。当必须进行区分时,将使用**习惯用法**(conventional use)一词来表示在标准中没有,但我认为是广泛使用的。对于那些符合标准的内容,我将使用**标准**(standard)或**规范**(normative)二词来表示。("规范"一词是标准人们用来表示"你必须和标准一致"的陈述。因此,"非规范的 UML"成为"按照 UML 标准,严格不合法"的一种别致的说法。)

当你考察一张 UML 图时,应该记住 UML 中的一个通则,即对一个特定的图来说,任何信息均可被**隐抑**(suppressed)。要么一般地,"隐蔽所有属性";要么特定地,"不示明这三类"。这两种隐抑都可能出现。因此,在一张图中,你一定不能在信息阙如时进行任何推断。如果没有重数,就不能对其值进行推断,即使 UML 元模型有一默认(如对属性默认[1])。如果在图中未曾看到信息,这可能出自默认,也可能出自隐抑。

说到此，有一些一般的习惯，如多值性是集合。在正文中，我将指出这些默认习惯。

如果你是一名草图绘制人员或一名蓝图绘制人员，不必更多强调有合法的 UML，更为重要的是，要对你的系统有一个好的设计。我宁愿有一个用不合法的 UML 书写的好的设计，而不愿有一个用合法的 UML 书写的蹩脚的设计。显然，既好又合法当然最好，但是将精力用于追求有一个好的设计，要比为 UML 的隐晦费解而担心更好。（当然，在将 UML 用作编程语言时，就必须是合法的，否则，程序就会运行不正常。）

UML 的含义

UML 的棘手问题之一是，虽然规范相当详尽地表述什么是组成良好的 UML，但并未过多地谈到在 UML 元模型的精良天地之外 UML 的含义。将 UML 映射到任一种特定编程语言的形式定义并不存在。你不可能一看到一个 UML 图，就能准确地说出其等价代码。但是，对这种代码却能得出一个粗略印象。实际应用中，这就足够了。开发小组往往据此形成其特定的习惯，而你有必要在使用中熟悉它们。

UML 并非足够

虽然 UML 提供了有助于定义应用的各种图，但这绝不是你

所要用的所有有用图的完备清单。在很多场合,别的图也可能有用。如果没有适用于你的目的的 UML 图,你就应毫不犹豫地去使用那些非 UML 的图。

图 1.3 是一个屏幕流图,它示明一个用户接口上的各种屏幕以及你如何在它们之间移动。我已经见过并使用这些屏幕图多年,所看到的只是关于它们的含义的极为粗略的定义。在 UML 中却没有任何一项这样的内容。但是,我已发现,它是一种很有用的图。

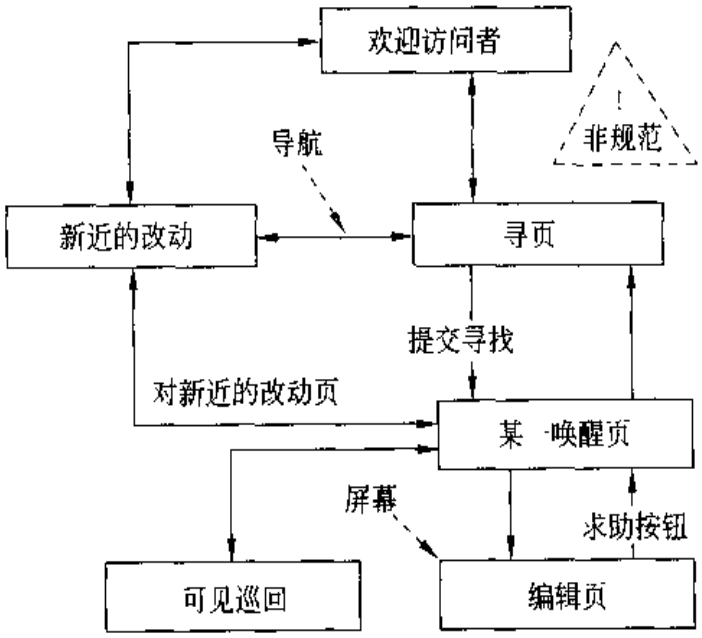


图 1.3 唤醒部分的非形式屏幕流图(http://c2.com/cgi/wiki)

表 1.2 示明另一个令人喜爱的构造:选定表。选定表是示明复杂逻辑条件的好方法。你可以用一张活动图来做这件事,但是,一旦遇到简单情形以外的情形,选定表就更为紧凑,更为清晰。此外,已经出现了多种形式的选定表。表 1.2 把整个表分成

两部分：双线上的条件部分和双线下的结果部分。每一列示明条件的特定组合如何导致一组特定的结果。

表 1.2 选定表

重要客户	X	X	Y	Y	N	N
优先订单	Y	N	Y	N	Y	N
国际订单	Y	Y	N	N	N	N
费用(美元)	150	100	70	50	80	60
超级代理	*	*	*			

在各种书籍中你将会碰到上述各类情况。不必犹豫去试用看来对项目合适的技术。如果这些技术工作得很好，就使用它们，否则，就把它们扔掉。(当然，这同样也是对 UML 图的建议。)

何处着手使用 UML

没有人了解并使用 UML 的全部，即使是 UML 的创建者们也是如此。大多数人都使用 UML 的一个小的子集，并据以工作。你应该寻求为你以及你的同事们工作的 UML 子集。

如果你开始用 UML，我建议首先将注意力集中到基本形式的类图及顺序图，这些是最常用的图型，并且依我看，它们是最有用的图型。

一旦领会了这些图型，就可以开始使用更为高级的类图图示法，并看看别的图型。对它们进行试验，并考察它们对你的帮助如何，不要有顾虑删掉看来对你的工作无用的任何东西。

何处找寻更多资料

本书不是一本 UML 的完备的定义性基准读物,更谈不上面向对象分析与设计了。好多话以及好多值得阅读的内容在书中都没有。在我讨论个别论题时,也提到一些你应该从中获得更为深入信息的其他书籍。这里是一些关于 UML 及面向对象设计的一般书籍。

关于所有推荐的书,你可能需要核查一下它们是针对 UML 的哪一版本写的。2003 年 4 月前,出版的书都未涉及 UML 2.0,这不足为奇,因为当时 UML 2.0 刚刚杀青。我所建议的都是一些佳作,但是我说不上它们是否或何时将要更新到 UML 2.0 标准。

如果你是一位对象技术的新手,我向你推荐我当前钟爱的导引书[Larman]。该书作者坚定的职责驱动的设计方法是值得采用的。

关于 UML 的结语是,你应该去看正式的标准文档,但要记住,那些文档是为方法学学者们在其私用工作室中使用而写的。关于 UML 标准的更加易于消化的改写本,可看[Rumbaugh, UML Reference]。

关于面向对象设计的更为详尽的推荐资料,你可以从[Martin]书中学到很多好的东西。

我还建议,关于超出基本内容的材料,可阅读一些讨论模式的书。现今方法战已经结束,模式(第 35 页)则是分析与设计的很多有趣材料出现的地方。

第2章

开发过程

如已提及，UML 源于一组面向对象分析与设计方法。在某种程度上，所有这些方法交织成一种图示建模语言，并带有一种旨在描绘如何进行开发软件的过程。

在趣的是，当 UML 形成时，各种从业人员发现，虽然他们可以在建模语言上达成协议，却断然不能在过程上达成协议。因此，他们同意把过程协议的事留待以后去讨论，而将 UML 限定为一种建模语言。

本书名为 UML 精粹，所以我有把握不管过程。但是，我相信，不了解建模技术如何适应过程，建模技术也就不会有丝毫意义。使用 UML 的方式在很大程度上取决于使用的过程的风格。

因此，我认为，重要的是先来讨论过程，使你能看到使用 UML 的环境。我不准备详细讨论任何特定过程，只想给你足够的信息以看出这一环境并指出到哪里才能找到更多的资料。

当你听到人们讨论 UML 时，往往也听到他们谈论 Rational 统一过程(RUP)。RUP 是一种你可以使用 UML 的过程，或者，

更严格地说,RUP 是一种过程框架。但是,除了 Rational 公司各种人员的普遍介人以及“统一”之名外,RUP 和 UML 并无任何特定联系。UML 可用于任何过程。RUP 是一种流行的方法,在第 32 页上要对它进行讨论。

迭代过程与瀑布过程

关于过程的最大争议之一是瀑布风格与迭代风格之间的争议。这些名词往往被错用,特别是,由于迭代被看作一个时尚词语,而瀑布过程似乎是穿上了一条彩格裤子。因此,很多项目都声称进行迭代开发,但实际做的却是瀑布开发。

迭代过程与瀑布过程之间的本质区别是如何将一个项目分解成小块。假设有一个项目,预计将要花一年时间,那么让项目组走开一年,等项目完成后再回来,这样做,很少有人会感到适意的。因此需要有某种分解,以便人们可以对付问题并跟踪进展。

瀑布风格是基于活动来分解项目的。为了构作软件,你必须进行一些活动:需求分析、设计、编码以及测试。这样,为时一年的项目就可能有两个月的分析阶段,接下去有四个月的设计阶段,后接三个月的编码阶段,再后接三个月的测试阶段。

迭代风格根据功能子集来分解项目。拿一年来说,你将项目分解成几次三个月的迭代。在第一次迭代中,取四分之一需求,并对之进行软件生命全期工作:分析、设计、编码、测试。(一般认为,软件生命全期除了分析、设计、编码、测试外,还包含维护——译注。)在第一次迭代结束时,就有了一个系统,该系统实现了所

需功能的四分之一。然后，再进行第二次迭代，这样在六个月结束时，就有了一个可以实现二分之一功能的系统。

当然，上面是一种简化的描绘，但它却反映了这两种风格的本质区别。自然，在实践中，有些搀杂会渗入过程之中。

对瀑布开发而言，通常在相邻两个阶段之间，有一种例行的亲手交接形式，但往往会有回流。在编码中，会发生这样的事，它使你重新访问分析与设计。你绝对不应假定，在编码开始时，整个设计已经结束。不可避免的是，在后续阶段中会有必要重新访问分析与设计选定。不过，这些回流是一些异常情况，应该尽可能使之减为极少。

对迭代而言，在真正迭代开始前，你通常看到某种形式的探索活动。至少说，这一活动将会使你能看到需求的高层综合外观；这至少对于把需求分解成接下去的一些迭代来说已经够用。在这种探索中还可能出现一些高层次的设计选定。另一方面，虽然每次迭代都应生出备产的集成软件，但是，却往往并不能完全达到这一地步，而需要一段稳定期，以排除一些最终的差错。还有像用户培训等活动则留到最后进行。

在每次迭代结束时，你还不能立即把系统投入生产，但是，这个系统应该已具备生产质量了。不过，往往可以每隔一定时间就把系统投入生产一次，这样做的好处是，可以早一些从系统得到裨益，并得到较高质量的反馈。在这种情况下，你常常听到具有多次发布（release）的项目。每次发布分解成若干次迭代（iteration）。

迭代开发也曾以多种名称出现：渐进式、螺旋式、演化式、喷泉式。不同的人对这些命名的含义会有些区别，但这些区别并未

得到广泛认同,而且也不如迭代/瀑布二分法那样重要。

你可以有一些混合方法。[McConnell]描绘了**阶段提交**(staged delivery)生命期,其中先按瀑布风格进行分析与高层设计,随后把编码与测试分解成几次迭代。这样一个项目就可能会有四个月的分析与设计,接下去再有四次两个月的系统迭代构作。

过去几年中软件过程的大多数作者(特别在面向对象界)都不喜欢瀑布风格。原因很多,最基本的是,很难回答项目是否真正遵循了瀑布过程。很容易凭借前期阶段就宣布胜利而隐蔽日程表的差错。通常你实际能回答项目是否真正遵循瀑布过程的惟一办法就是生产出经测试的集成软件。通过反复如此进行,如果出现差错,迭代风格会给你较好的预警通知。

单凭这一点理由,我就极力推荐项目不要采用纯瀑布风格。如果不用更纯粹的迭代风格,也应该采用阶段提交风格。

面向对象界长期以来支持迭代开发,并且有把握地说,几乎介入构作 UML 的每一个人都至少支持某种形式的迭代开发。我的工业实践的感觉是,瀑布开发仍然是较为常用的风格。理由之一就是所谓的"拟迭代开发"(pseudoiterative development):人们声称在做迭代开发,但事实上却在做瀑布开发。这种情况的普遍迹象是:

- "我们做一次分析迭代,接下去做两次设计迭代,……"
- "这次迭代的代码有很多错误,但是我们将在最后去排除它。"

特别重要的是,每次迭代产生经测试的集成代码,后者尽可能接近于生产质量。测试和集成是最难评价的活动,因此,重要

的是，不要有允许调整的活动，像在项目结束时再调整那样。测试应该是，那些未被安排交付的任何迭代均能在无本质额外开发工作的前提下得以交付。

关于迭代的一种常用技术是利用**时间框定法**(time boxing)。这种方法迫使各次迭代的时间长度固定。如果出现你不能构作你打算在一次迭代中要构作的全部功能，你就必须决定从这次迭代中搁置某一功能：你一定不能搁置迭代日期。使用迭代开发的很多项目在整个项目中使用相同的迭代长度；这样，便会得到构作的定时节律。

我喜欢时间框定法的原因是，人们通常对搁置功能感到困难。通过定时搁置功能，他们在大型交付上就能较好地在搁置日期与搁置功能之间进行明智的选择。在迭代中搁置功能对帮助人们学到什么是实际的优先需求来说，也是起作用的。

关于迭代开发最为普遍考虑的问题之一是“重做”(rework)问题。迭代开发显式地假定在项目较后的迭代中将要重写并删除现有的代码。在诸如制造业的很多领域，“重做”看来是一种浪费，但软件却不像制造业，往往对现有代码重写比对蹩脚设计的代码打补丁(此处“补丁”即“补绽”之意——译者注)更为有效。一些技术实践可以在很大程度上使重做更有效。

- **自动回归测试**(automated regression test)使你能迅速查出在改动时可能引进的任何缺陷。测试框架 xUnit 组是一项构作自动单元测试特别有价值的工具，从原始的 JUnit http://junit.org 起，现在有了通向几乎任何可想象到的语言的港口(见 http://www.xprogramming.com/software.htm)。一个好的经验性法则是单元测试代码的

大小应该和生产代码大致相同。

- **结构改组**(refactoring)是改变现有软件的一种规范技术[Fowler, refactoring]。结构改组利用对代码库施行一系列保持行为不变的小变换来进行工作,很多这些变换均可自动进行(见 http://www.refactoring.com)。
- **连续集成**(continuous integration)使项目组协调以避免痛苦的集成循环[Fowler and Foemmel]。其核心是一个完全自动的构作过程,当项目组任何成员往代码库中检测代码时,该过程即可自动开始工作。要求开发人员天天进行检测,这样,一天可以进行多次自动构作。构作过程包括运行一大块自动回归测试,以迅速发现任何前后矛盾之处从而易于排除它们。

所有这些技术实践近来已由极端程序设计[Beck]所普及,尽管它们先前已被使用,并且不管你是否使用 XP 或任何别的便捷过程,它们都是可以并且应当被使用的。

预见性计划制订与适应性计划制订

瀑布开发经久不衰的一个原因是软件开发中对预见性的渴望。最令人感到沮丧的是,对于构作一项软件所需的费用以及时间没有一个明确的概念。

预见性方法指望通过在项目中早点进行工作,来对后来要完成的工作有一个较深的了解,从而可以对项目的后继部分有准确的估计。对**预见性计划制订**(predictive planning),一个项目有两

个阶段。第一阶段提出计划，这是难以预见的，但第二阶段则在很大程度上是可以预见的，这是因为这时计划已经安排就绪的缘故。

这未必是泾渭分明的。随着项目的进行，会逐渐有更多的预见性，并且即使有了一个预见性计划，事情仍然会出岔。你只能指望一旦安排好一个扎实稳妥的计划，其实际的偏离不太显著而已。

然而，关于是否很多软件项目总是可以预见的这一问题却颇有争议。问题的核心是需求分析。软件项目中复杂性的重要根源之一就是了解软件系统需求的困难。大多数软件项目都要经历明显的**需求折腾**(requirements churn)：在项目的往后阶段改变需求。这些改变动摇了预见性计划的基础。你可以通过早点冻结需求而不许改动来抵制需求改变，但这样就会冒提交一个不再满足用户要求的系统的风险。

这一问题引起两种截然不同的反应。一种途径是更多地致力于需求过程本身。这样，你就可以得到一组更为精确的需求，从而会使折腾减少。

另一派却主张需求折腾是不可避免的。对多数项目来说，要想需求稳定到足以使用预见性计划，那是太难了。其原因或是由于展望软件可以做什么是极其困难的，或是由于市场条件迫使一些不可预见的改动的缘故。这一思想派别主张**适应性计划制订**(adaptive planning)，因此把预见性看成一种假象。不用虚幻的预见性来愚弄自己，而应该面对不断改变的现实，采用这样一种计划制订途径，即把软件项目中的变动看作是经常出现的事。控制这些变动，以使项目能提交最好的软件；但是虽然项目是可控

制的,它却不是可以预见的。

预见性项目与适应性项目之间的区别在人们谈论如何进行项目时以各种方式显现出来。当谈到一个项目进展良好时,是指它是按计划进行的,这是预见性的思想方式。在适应性环境中,你却不能说"按计划",原因在于计划总在改变。这并不意味着适应性项目不要制订计划;通常它们都很有计划,只不过是把计划看作是评价改动结果之基准,而不作为对未来的预测。

对预见性计划,你可以制订一个固定价格/固定范围的契约,这一契约准确地表述应该构作什么、价格多少以及提交日期。对适应性计划而言,这样的固定却是不可能的。可以固定预算与提交时间,却不能固定将要提交的功能。适应性契约假定,用户将与开发组合作再行定期对所需构作的功能进行评估,并且如果进展过于缓慢,则项目将予以取消。这样,适应性计划制订过程可以是固定价格/变动范围的。

自然,适应性途径较不可取,这是因为任何人都希望项目中有较多的预见性。然而,预见性取决于一组准确、精确以及稳定的需求。如果你不能稳定你的需求,预见性计划就建基于沙滩之上,而变动严重致使项目偏离方向。因此有如下两点重要的劝告:

1. 在具有准确而精确的需求且确信不会大动以前,不要制订预见性计划。
2. 如果不能得到准确、精确、稳定的需求,就使用适应性计划制定方式。

预见性与适应性提供了生命期的选择。适应性计划绝对要求迭代过程。预见性计划制订两种风格(迭代与瀑布)都可以做,

当然采用瀑布风格或阶段提交风格更容易看出来它是如何工作的。

敏捷过程

过去几年中，人们对敏捷过程很感兴趣。敏捷是一个范围很广的词，它涵盖了很多过程，这些过程共享由《敏捷软件开发宣言》(http://agile Manifesto. org)所规定的一组价值和原则。这些过程的例子是：极端程序设计(XP，Extreme Programming)、喧嚣(Scrum)、特征驱动开发(FDD，Feature Driven Development)、晶体(Crystal)以及动态系统开发方法 DSDM (Dynamic Systems Development Method)。

从我们的讨论来看，敏捷过程是强适应性的过程。它们也是很面向人的过程。敏捷方法假定，项目成功中最重要的因素是介入项目的人的素质以及他们如何在人际上良好地协同工作。他们使用什么过程和什么工具完全是次要的。

敏捷方法倾向使用时间框定的短小迭代，通常是一个月或更短的迭代。因为敏捷方法不太注重文档，所以它们不屑于将 UML 用于蓝图绘制。大多数是将 UML 用于草图绘制，少部分人主张将它用作编程语言。

敏捷过程倾向是低仪节(ceremony)的。高仪节的或者重量级的过程在项目进行中有很多文档和控制点。敏捷过程认为，仪节使得修改或使用天才们的工作更难了。因此，敏捷过程往往表现为轻量级(lightweight)的过程。重要的是要认识到，仪节欠缺

乃是适应性与面向人的结果,而不是一个基本特征。

Rational 统一过程

虽然 Rational 统一过程(RUP)与 UML 无关,但人们往往同时谈论两者。因此,我认为在这里还是应该简单介绍一下 RUP。

虽然 RUP 称作过程,但实际上它是一个过程框架,提供了一个讨论过程的词汇表和松散的结构。当你使用 RUP 时,需要做的第一件事就是选择一个**开发例案**(development case):即准备用于项目中的过程。开发例案可以是多样化的,因而,不要假定你的开发例案将要和任何别的开发例案很相像。选择一个开发例案先需要一位很熟悉 RUP 的人,他能针对特定项目之要求来裁剪 RUP。要么就是从一开始就有一组不断发展的成包开发例案。

不管是什么开发例案,RUP 本质上是一个迭代过程。瀑布风格和 RUP 的基理是不匹配的,不幸的是,遇到这样的项目——使用瀑布风格的过程却乔装成 RUP——并不罕见。

所有的 RUP 项目都应遵循四个阶段:

1. **初始**(inception) 对项目进行最初估算。通常在初始阶段决定是否要拨出足够的款项以供细化阶段使用。
2. **细化**(elaboration) 识认项目的基本用案并在迭代中构作软件,以得出系统的体系结构。细化结束时,应该对需求有了较好的概念,并且有了一个概要的工作系统,它可以用作开发的种子。特别是,应该已经发现并解决了项

目的主要风险。

3. **构作**(construction)　继续构作过程，开发足够要交付的功能。
4. **转接**(transition)　包括各种未曾在迭代中进行的后阶段的活动。这些活动可能包括往数据中心的部署、用户培训等。

各个阶段之间稍有模糊，在细化与构作之间更是如此。对某些人来说，转移到构作阶段即是可以转移到预见性计划方式之时。对另一些人来说，这只表示对需求及体系结构有了一个粗略看法，这种看法可以持续到项目的其余部分。

有时，RUP 也称作统一过程(Unified Process，UP)。希望利用 RUP 的词汇和总体风格但不利用 Rational 软件公司特许产品的机构通常就是这样做的。你可以把 RUP 想象成是基于 UP 的 Rational 公司产品的待售品，或者把 RUP 和 UP 看成同义。不管是哪一种理解，总会找到同意你意见的人。

过程适配项目

各个软件项目彼此大有差别，进行软件开发的方式取决于很多因素：构作的系统的种类、使用的技术、开发组的大小与成员分布、风险的性质、失败的后果、开发组的工作作风以及组织机构的文化。因此，不应指望有一个对所有项目都适用的万能过程。

因此，你总须将一过程适配你的特定环境。你需要做的首要工作之一是观察项目，并考察什么过程看来对它接近适配。这就

应该给出要考察的过程的短小列表。

然后,应该考察的是,为了使过程适配项目需要做哪些适应工作。对此,须稍加小心。很多过程在你用它工作以前是难以完全对它做出正确评价的。在这种情形下,往往值得在你学会利用特别好的过程以前,用它作两次迭代。随后,就可以开始对它进行修改。如果一开始就比较熟悉一个过程是如何工作的,可以从开始就对它修改。请记住,通常总是开始少修改一些,随后再增加,要比开始多修改,随后再把一些东西扔掉,更为容易一些。

不管在开始时对你的过程有多么确信,重要的是在继续进行中要学习。的确,迭代开发的一大好处就是它支持过程经常改进。

每次迭代结束时,要进行一次迭**代回顾**(iteration retrospective)。项目组人员聚到一起,考察工作进行得如何以及如何改进。如果是短小的迭代,这样的聚会两小时就已足够。做这件事的一个好的方式是列出一张带有三项的表。

1. **保留**(keep) 已经工作得不错的事,你确信你要继续做下去。
2. **问题**(problem) 工作得不好的那些方面。
3. **尝试**(try) 旨在改进过程的那些变动。

在首次迭代回顾后,你就可以这样来开始每次迭代回顾,即检查上次聚会的讨论项目,并查看事情是如何变化的。不要忘记保留事情表:重要的是要掌握正在工作的那些事情的动态。如果不这样做,就可能丧失对项目的洞察能力,并可能对成功的实践不再予以留意。

在一个项目结束或一次重要交付时,你可能希望考虑一次更

为正式的**项目回顾**(project retrospective),这可能持续两天;细节请见 http://www.retrospectives.com/及[Kerth]。最烦恼的一件事就是组织机构一贯不能从自身经验中学习,而屡屡以犯代价昂贵的错误而告终。

模　式

UML告诉你如何表示面向对象设计。模式则考察过程的结果:样例设计。

很多人评论项目有问题,原因是项目参与人员不熟悉更有经验的人所熟知的设计。模式表述通常做事的方法,并由设计中重复进行论题的人收集。这些人把每一论题拿来并加以表述,使得别人可以阅读模式,并看出如何运用模式。

让我们来看一个例子。比方说,在你的便携式计算机上运行一个过程中的对象,这些对象需要与运行在另一过程中的别的对象进行通信。另一过程也许也在你的便携式计算机上,也许在别的地方。你不想让系统中的对象操心如何找寻网上的其他对象或者执行远程过程调用。

你能做的就是为这个远程对象在你的局域过程中建立一个代理对象,这个代理对象与远程对象具有相同的接口。你的局域对象利用通常处理过程中的消息发送来与代理对象交谈。这时,代理对象负责把消息传送给实在对象,而不管该实在对象位于何处。

代理是网络或其他地方常用的技术。人们有很多使用代理的经验,如了解如何使用代理、代理能带来什么好处、代理的局限以及如何实现代理。像本书这样的方法书不讨论这方面的知识,只讨论如何能对代理绘图。虽然代理有用,但讨论涉及代理的经验则更有用。

20世纪90年代初,有些人开始取得这方面的经验,他们组织了一个

志在写模式的团体，这些人主办会议并出版书籍。

由这个小组撰写的最著名的模式书就是[Gang of Four]，这本书详细讨论了23个设计模式。如果要了解代理，该书用了10页篇幅论述论题，给出各个对象如何一同工作的细节、模式的益处和局限、常用的变形以及关于实现的指点。

模式的内容比模型丰富得多。模式还须包括为什么它是这个样子的理由。人们常说，模式是问题的一个解。模式必须清晰地识认问题，并阐明为什么用它来求解问题，以及阐明它在何种条件下工作，在何种条件下不工作。

模式是重要的，这是因为它们是在对语言或建模技术的基本原理理解之上的下一阶段。模式给你的是一系列解决方案，还指出什么是一个好的模型以及如何着手构作模型。模式是通过例子来讲授的。

当我开始工作时，我就奇怪为什么事情都必须从头做起。为什么没有一些手册来指明如何做一些通常的事？模式团体正在着手制作这些手册。

现在已有很多模式书籍问世，并且它们质量甚殊。我所钟爱的书是[Gang of Four]、[POSA1]、[POSA2]、[Core J2EE Paterns]、[Pont]以及拙著[Fowler，AP]和[Fowler，P of EAA]。你也可以看一下模式主页：http://www.hillside.net/patterns。

UML 适配过程

当人们看待图示建模语言时，通常总是把它们想象成是在一个瀑布过程的环境中。瀑布过程通常有一些文档，作为分析、设

计、编码各个阶段之间的亲手交接材料。图示模型往往构成这些文档的主要部分。的确，20 世纪 70 年代与 80 年代的很多结构化方法关于这样一些分析与设计模型已有不少讨论。

不管是否使用瀑布方法，你仍然在进行着分析、设计、编码以及测试等活动。你可以运行一个具有若干次一周迭代且每周有一个小瀑布的迭代项目。

使用 UML 不一定要开发文档或提供复杂的 CASE 工具。很多人只是在聚会时才在白板上画出 UML 图，以助于沟通想法。

需求分析

需求分析活动包括试图了解软件工作的用户和客户要求系统做什么。这里的一些技术可以派得上用场：

- 用案　用案表述人们如何与系统交互。
- 从概念视面绘制的类图　这种图可能是一种构作领域精确词汇表的好方法。
- 活动图　活动图可以示明组织机构的工作流，示明软件与人的活动如何交互。活动图还可以示明用案的环境以及复杂用案如何工作的细节。
- 状态图　如果一个概念具有有趣的生命期、具有各种状态以及改变状态的事件，状态图则可能有用。

进行需求分析时请记住，最重要的是与用户及客户的交流。通常，他们都不是软件业人士并且不熟悉 UML 或任何别的技术。即使如此，我也已经有了对非技术人士使用这些技术的成功经验。要记住，重要的是要尽量少用图示法，不要引进软件实现

所特有的任何东西。

不论何时都要准备背离 UML 的规则,如果它能帮助你更好地进行交流的话。在分析过程中使用 UML 的最大风险是领域专家对你所绘的图不能完全理解。一个了解领域的人不理解的图比无用更坏,它所做的一切是使开发组产生一种错误的自信感。

设计

做设计时,可以使你的图具有更多的内涵。可以更多地利用图示法并使图示更加准确。一些有用的技术是:

- 软件视面的类图　这些图示明软件中的类以及它们如何相互联系。
- 常用案况的顺序图　一种有用的方法是从用案中挑选出最为重要且有趣的案况,并利用 CRC 卡或顺序图以勾勒出软件中发生的事件。
- 包图　示明软件的大型组织。
- 具有复杂生命史的类的状态图。
- 部署图　示明软件的物理布局。

一旦编写出软件,同样上述很多技术可以用来制作软件文档。这样就可以帮助人们发现他们绕过软件的方式,如果他们必须对之工作而且又不熟悉代码的话。

对瀑布生命期,就要把这些图和活动作为各个阶段的部分。阶段结束(end-of-phase)文档通常包括关于那一活动的一些合适的 UML 图。瀑布风格通常是指把 UML 用作蓝图。

在迭代风格中，UML 图既可用于蓝图方式又可用于草图方式。对蓝图方式，分析图往往是在构作功能图前在迭代中构作的。每次迭代并不是从头开始，而是修改现有的文档，突出新迭代中那些变动。

蓝图设计通常在迭代中总是早做的，并且可以根据该次迭代的不同的功能片分段进行。此外，迭代是指改动现有的模型，而不是每次都要构作新模型。

按草图方式使用 UML 意味着一个更为易变的过程。一种途径是，在迭代开始时，花两天功夫对该次迭代草拟出设计。在迭代中的任何时刻，你还可以进行短暂的设计聚会。在开发人员开始对付一项非平凡的功能时，举行一次半小时的快速会议。

对蓝图来说，你总指望代码实现是按蓝图进行的。蓝图的一次改动就形成一次偏离，它需要绘制蓝图的设计人员的复查。草图通常较多地看作是设计的初次删减。如果在编码中人们发现草图不完全正确，他们就应该自由改动设计，实现人员必须自己判断是否这一改动需要进行更广泛的讨论，以了解其全部后果。

对蓝图我所担心的一件事就是，根据我的观察很难使蓝图正确无误，即使对一位好的设计人员来说也是如此。我常常发现我自己的设计在编码过程中不能保持完好无缺。我还发现 UML 草图有用，但并未发现它们可以看成是绝对的东西。

在两种方式中，探索一些设计选择是有意义的。通常是在草图方式中探索选择，这样可以迅速生成并改变选择。一旦挑选出一个待运行的设计，你或者就利用这个草图，或者将它细化为蓝图。

文档

一旦构作了软件,你就可以利用 UML 来帮助制作有关所做工作的文档。为此,我发现 UML 图能使你对系统有一个全局的了解。然而,在做这件事时,应强调我并不相信产生整个系统细图的效用。今引用 Ward Cunningham [Cunningham]的一段话如下:

> 精心选择并书写良好的备忘录可以轻易取代传统上全面的设计文档,后者除了个别地方外,很少显现出是杰出的。把那些地方提升,……,而忘却其余(第 384 页)。

我相信,详细文档应该根据代码生成——比方说,像 JavaDoc 那样。你应该书写附加文档以突出重要概念,把这些文档看作在读者进入到基于代码的细节以前的第一步材料。我喜欢把它们构造成散文式的文档,短小到花喝一杯咖啡的功夫就可以读完,利用 UML 图来帮助讨论。我宁愿把 UML 图作为草图以突出系统中最为重要的部分。显然,文档作者需要决定何者重要以及何者不重要。作者对此事胸有成竹比要求读者去做这件事远远为好。

包图使系统有一个好的逻辑路图,这种图帮助我理解系统的逻辑片段以及其间的依赖关系,并使之置于控制之下。示明高层物理形象的部署图(见第 8 章)在本阶段也可能证明是有用的。

在每个包内部,我喜欢看到一个类图。我并不示明每个类上的每一操作,只示明一些重要特征,这些特征帮助我了解在哪里

有些什么。这个类图用作一个图示目录。

类图应该由少量交互图支持，这些交互图示明系统中最重要的交互。此外，选择性在这里也是重要的；请记住，在这种文档中，全面性乃是易懂性的敌人。

如果一个类具有复杂的生命期行为，我就绘制一个状态机图（见第 10 章）来描绘它。只是在该行为充分复杂时才画状态机图，但我发现，这种复杂的行为并不是经常出现的。

我的书中往往包括一些按文化程序风格书写的重要代码。如果包括一个特别复杂的算法，我就考虑使用活动图（见第 11 章），但这只是在它能比用代码使我有更多的了解时才这样做。

如果我发现一些重复产生的概念，就使用模式（第 35 页）来获取基本思想。

文档制作的一项最重要的事就是你未曾采用的设计选择以及未采用的理由。这一点往往是最容易被人遗忘但又是最有用的、你能提供的外部文档片段。

理解遗产代码

UML 可以帮助你按两种方式勾勒出许多崎岖不平的不熟悉的代码。对一些关键事实构作的草图可用作一种图示记录机制，以帮助你在学习中获取重要信息。包中关键类的草图及其关键交互可有助于弄清楚正在进行的是什么事。

你可以用一些现代工具对系统的关键部分生成细图。不要利用这些工具去生成大量书面报告；而是在探究代码本身时，利用它们去钻入关键方面。一种特别令人愉快的能力是生成一个

顺序图,以看出多个对象在处置复杂方法中是如何合作的。

选择开发过程

我坚决支持迭代开发过程。如同我在本书中先前所说的:应该只对那些你希望取得成功的项目使用迭代开发。

也许这有点油腔滑调,但是随着我的年龄渐长,关于利用迭代开发,就愈加敢作敢为了。好!这是一种重要的技术,一种可以用来及早暴露风险并对开发取得较好控制的技术。它和没有管理截然不同。虽然公正地说,我应该指出,有些人曾经这样使用过(这里指的是"没有管理"——译注)。它需要很好地制订计划。但它却是一种扎实的方法,每一本面向对象开发的书由于充分的理由都鼓励使用迭代开发。

当你听到我是敏捷软件开发宣言的作者之一时,不应感到奇怪,我是敏捷方法的一名狂热爱好者。我也对极端程序设计有过不少正面经验,并且肯定你应该很认真地考虑它的实践。

何处找寻更多资料

讨论软件过程的书总是常见的,而敏捷软件开发的腾起导致出现很多新书。总的说来,我所钟爱的讨论过程的书是[McConnell]。作者在书中宽广而实用地涵盖了软件开发包含的很多问题以及一个长的有用实践表。

出自敏捷团体的书[Cockburn, agile]与[Highsmith]提供了一个好的综述。关于按敏捷方式应用 UML 的很多良好建言，请看[Ambler]。

最流行的敏捷方法之一是极端程序设计(XP)，你可以通过 http://xprogramming.com 与 http://www.extremeprogramming.org 等网站来对它进行钻研。XP 已经引发出很多书出版。这就是为什么我把它称作前任轻量级方法学的原因。通常的起点是[Beck]。

虽然[Beck and Fowler]是就 XP 写的，它对制订迭代项目计划给出更多的细节，其中很多内容也由别的 XP 书籍所涵盖，但是如果你只对计划制订感兴趣，这本书乃是一个好的选择。

关于 Rational 统一过程更多的资料，我所钟爱的导引读物是[Kruchten]。

第 3 章

类图：基础部分

如果某人在一条黑暗的胡同中向你走来，并说："嗨，要不要看一个 UML 图？"那个图就可能是一个类图。我看到的大多数 UML 图都是类图。

类图不仅使用广泛而且也属于最大范围的建模概念。虽然其基本成分人人需要，但其高级概念却鲜为人用。因此，我把关于类图的讨论一分为二：即基础部分（本章）与高级概念（第 5 章）。

一个**类图**（class diagram）表述系统中各个对象的类型以及其间存在的各种静态关系。类图也示明类中的特性和操作以及用于对象连接方式的约束。UML 使用**特征**（feature）一词作为涵盖类之特性与操作的一般术语。

图 3.1 示明一个简单类模型。对任何一位已和订单打过交道的人来说，不会使他感到吃惊。图中的方框是类，它又分成三个隔开的部分：类名（黑体），属性以及操作。图 3.1 也示明类间两种关系：关联与泛化。

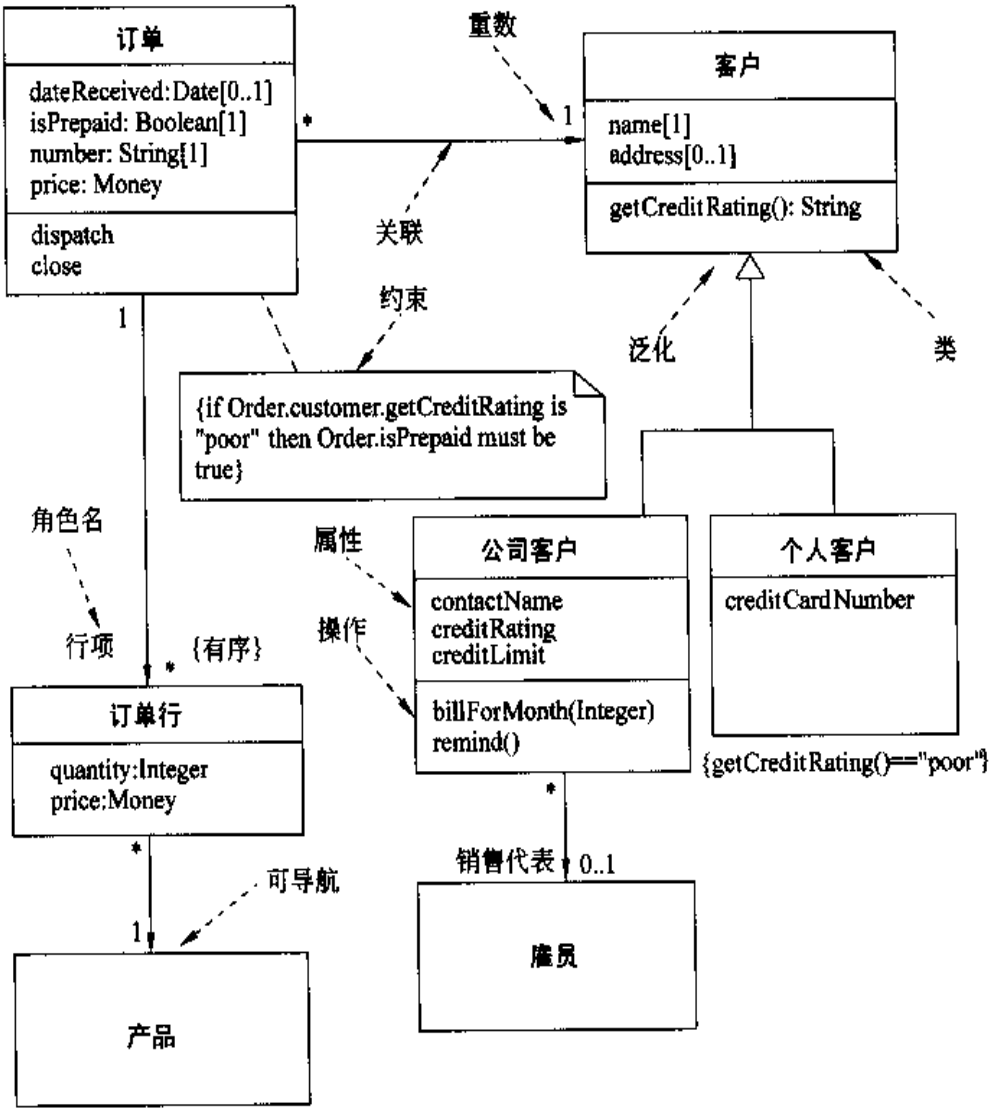

订单
dateReceived:Date[0..1]
isPrepaid: Boolean[1]
number: String[1]
price: Money
dispatch
close
重数
客户
name[1]
address[0..1]
getCreditRating(): String
关联
约束
泛化
类
{if Order.customer.getCreditRating is "poor" then Order.isPrepaid must be true}
角色名
属性
公司客户
contactName
creditRating
creditLimit
billForMonth(Integer)
remind()
个人客户
creditCardNumber
{getCreditRating()=="poor"}
行项
{有序}
操作
订单行
quantity:Integer
price:Money
销售代表
0..1
可导航
雇员
产品

图 3.1　简单类图

特性

特性(property)表示类的结构特征。作为第一级近似,你可以把特性想象成对应于类中的域。正如我们将要看到的,现实是比较复杂的,但却是一个合适的起点。

特性是一个单一的概念,但它出现在两种截然不同的表示中:属性及关联。虽然这两者在图中看起来很不一样,实际上它们却是一回事。

属性

属性(attribute)图示法把特性表述成类框中的一行正文。属性的全形是:

可见性　名:类型　重数=默认{特性串}

它的一个例子是:

-名:String[1]="Untlitled"{readOnly}

只有名是必要的。

- 可见性标记指明属性是公用(+)或私用(-)。在第 106 页中我将要讨论别的可见性。
- 属性名——类如何指称属性——大致对应于编程语言中的域名。
- 属性的类型指明何种对象可置于属性中。你可以把它想象成编程语言中的域类型。

- 我将在第 48 页中解释重数。
- 默认值是新近创建的对象的值，如果在创建中该属性未曾指明的话。
- {特性串}使你能指出属性的附加特性。在本例中我用{readOnly}指出客户不能修改这一特性。如果这一部分缺少，通常可以假定该属性是可以修改的。随着讨论的进行，我将要表述一些别的特性串。

关联

表示特性的另一种方式是标作关联。可以在属性上示明的大多数同样信息都可以在关联上出现。图 3.2 与图 3.3 分别示明用两种不同的图示法表示的同样特性。

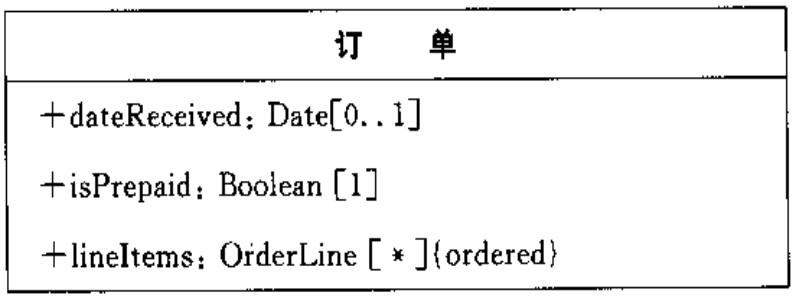


图 3.2　订单特性的属性表示

关联(association)是两个类之间的一条实线，方向是从源类到目标类。特性名连同其重数置于关联的目标端，关联的目标端连接到表示特性类型的类。

虽然大多数同样的信息出现在两种表示中，有些项却不同。特别是，关联在线的两端均可示明重数。

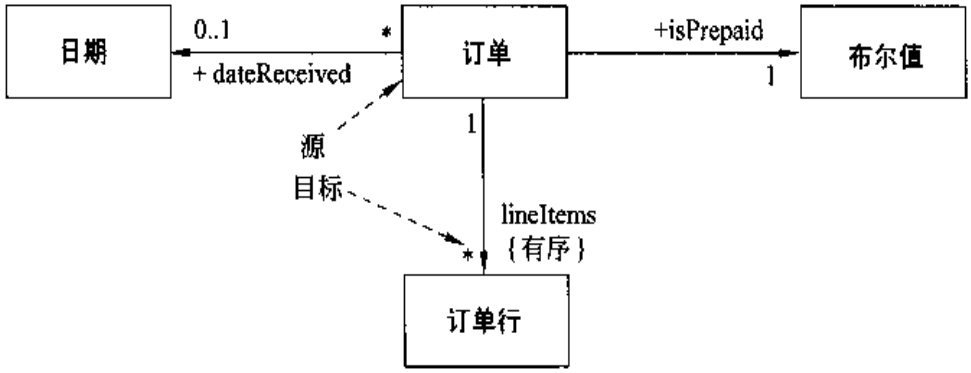


图 3.3 订单特性的关联表示

既然对相同的东西可采用两种表示，那么显而易见的问题是，为什么应该使用这种表示或另一种表示？一般说来，我倾向对小事(如日期或布尔值，一般说，值类型(第 94 页))使用属性，对较为重要的类(如客户与订单)使用关联。我也倾向于对图中导致使用关联的重要的类使用类框，对该图不太重要的东西使用属性。这种选择更多的是看强调什么而不在乎所附的含义。

重数

特性的**重数**(multiplicity)指出可以具有该特性的对象数目。你将要看到的最常见的重数是：

- 1(一份订单必须正好具有一名客户。)
- 0..1(一名公司客户可以有也可以没有单个的销售代表。)
- *(一名客户不一定发出一份订单，并且一名客户可以发出的订单数是没有上限的——零份或更多份。)

更为一般地，重数用一个下界和一个上界来定义，例如对凯

纳斯特纸牌游戏者，上下界为 2..4。下界可为任何正数（此处宜改为正整数——译注）或零；上界为任何正数（宜改为正整数——译注）或 *（代表无限制）。如果下界和上界相同，可以只用一个数；因此，1 和 1..1 是等价的。因为 0.. * 是常见的情形，故简记为 *。

在属性中，将遇到涉及重数的各种术语：

- **任选**(optional)　意味着下界为 0。
- **强行**(mandatory)　意味着下界为 1 或可能更大。
- **单值**(single-valued)　意味着上界为 1。
- **多值**(multivalued)　意味着上界大于 1，通常是 *。

如果我有一个多值特性，我就宁愿用复数名。

根据默认，多值重数中的成分构成一个集合，因此，如果你向客户要订单，这些订单不会依任何次序返回。如果关联中的各份订单的次序有意义，就必须在关联端添上{ordered}。如果你要允许复本，则添上{nonunique}（如果要明显表述默认，则可用{unordered}及{unique}）。也可以看到面向组(collection)的名，如{bag}表示无序、非惟一。

UML 1 允许间断重数，如 2,4（意思是 2 或 4，如小型货车时代以前的车）。间断重数并非常见，因此，UML 2 已将它们删去。

属性的默认重数是[1]。虽然这在元模型中为真，你却不能假定，在一个缺少重数的图中一个属性的重数是[1]，这是因为图可以隐抑重数信息的缘故。因此，我宁愿显式声明为重数[1]，如果它是重要的话。

特性的程序解释

和对 UML 中任何别的成分一样,不是只有一种方法能解释代码中的特性。最常见的软件表示是你的编程语言中的域和特性。因此,图 3.1 中的 OrderLine 类就对应如下的 Java 程序段:

```
public class OrderLine...
    private int quantity;
    private Money price;
    private Order order;
    private Product product.
```

在像 C# 这样具有特性的语言中,它就对应于:

```
public class OrderLine...
    public int Quantity;
    public Money Price;
    public Order Order;
    public Product Product;
```

请注意,属性典型地对应于支持特性的语言中的公用特性,但对应于不支持特性的语言中的私用域。在不具特性的语言中,你可以看到通过访问者(获取、置送)方法暴露出的各个域。只读属性没有置送方法(对域而言)或者置送动作(对特性而言)。注意,如果你对特性不给出名,普通就用目标类的名。

使用私用域是图的一种很着重实现的解释,而一种比较面向

接口的解释可能集中于获取方法而不是所附的数据。在这种情形下，可以看出来，OrderLine(订单行)的属性对应于如下方法：

```
public class OrderLine...
    private int quantity;
    private Product product;
    public int getQuantity(){
        return quantity;
    }
    public void setQuantity(int quantity){
        this.quantity=quantity;
    }
    public Money getPrice(){
        return product.getPrice().multiply(quantity);
    }
```

在这种情形，价格没有数据域，它却有算出值。但是，就 OrderLine 类的客户而言，它看起来和域一样。客户无法区分何者是域，何者是算出值。这种信息隐蔽乃是封装的本质。

如果属性是多值的，即指所考虑的数据是一个组。因此，订单类就指一组订单行。因为重数是有序的，这个组也必须是有序的(如 Java 中的表或 .NET 中的 IList)。如果那个组无序，严格说，它就应该没有有意义的序。因之，就用集合来实现，但是很多人还是把无序属性用表来实现。有些人利用数组，但是 UML 却指一个无限制的上界，因此，我几乎总是把组用作数据结构。

多值特性产生一种不同于单值特性的接口(在 Java 中)：

```
Class Order {
    private Set lineItems=new HashSet();
    public Set getLineItems() {
      return Collections. unmodifiableSet(lineItems);
    }
    public void addLineItem(OrderItem arg){
      lineItems. add(arg);
    }
    public void removeLineItem(OrderItem arg){
      lineItems. remove(arg);
    }
...
```

在大多数情形,不必指派一个多值特性,而是用添加方法与删除方法进行更新。为了控制其 LineItems 特性,订单须控制该组的成员;因此,不应该分发无保护的组。在这种情形,我使用一个保护代理来对组提供只读包装。你也可以提供一个不可更新的迭代体或者制作一个复本,这对客户修改成员对象已无问题,但客户不应直接改动组本身。

因为多值属性指的是组,在类图上你几乎永远看不到组类。你只好在组本身的非常低层的实现图上来示明它们。

你应该非常害怕那些除了一组域及其访问者外什么都没有的类。面向对象设计几乎提供能进行内容丰富的行为的对象,因此,它们不宜只是对别的对象提供数据。如果你在利用访问者重复调用数据,这就预示,某一行为应该移往具有数据的对象。

这些例子也进一步表明,在 UML 和代码之间并不存在严格的对应,但却有一种相似。在一个项目组内部,组的习惯将会导

致一种较为紧密的对应。

不管特性实现为域还是实现为算出值，它都表示对象总能提供的东西。你不应利用特性来对短暂的关系制作模型，例如对一个在方法调用中作为参数传递且只用于该交互范围内的对象。

双向关联

迄今我们所看到的关联称为单向关联（unidirectional association）。另一种常见的关联是如图 3.4 所示的双向关联。

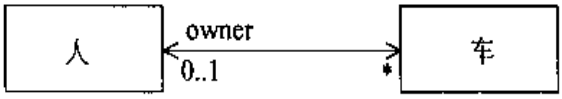


图 3.4　双向关联

双向关联（bidirectional association）是一对连接在一起互为其逆的特性。车（Car）类具有特性 owner：Person［1］。人（Person）类具有特性 cars：Car［*］。（请注意，我是如何按特性类型的复数来命名 cars 特性的，这是一种常见的但非规范的习惯。）

它们之间的逆连接指的是，如果遵照两个特性，你就应该返回到包括起点的集合。例如，如果从一个特定的 MG Midget 开始，寻找车主，然后又查看车主的其他车，这位车主的车的集合中就应该包含我从它开始的 Midget。

作为用特性来标记关联的另一种选择，很多特别是具有数据建模背景的人都喜欢用一个动词短语来标记关联（图 3.5），以便

这种关系可以在句子中使用。这是合法的,并且你还可以对关联添上一个箭号以避免歧义。大多数对象建模人员都喜欢使用特性名,这是因为它更好地对应于职责与操作的缘故。

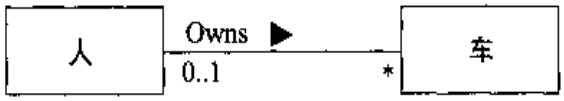


图 3.5 关联的动词短语命名

各人都有对关联命名的方式。只有当命名有助于增强理解时,我才会选择对关联命名。我见到太多关联具有"有"或"关系到"这样的名。

在图 3.4 中,位于关联两端的**导航箭号**(navigability arrow)使关联的双向性明显易见。图 3.5 没有箭号;UML 使你能将这种形式或者用于指明双向关联,或者用于你并未指出导航的时候。在你要使双向关联清晰时,我喜欢使用图 3.4 中的双头箭号。

用编程语言实现双向关联往往有点微妙,这是因为你必须确信两个特性保持同步的缘故。利用 C#,我使用如下代码来实现一个双向关联:

```
class Car...
  public Person Owner {
    get {return _owner;}
    set {
      if (_owner!=null) _owner.friendCars().Remove(this);
      _owner=value;
      if (_owner!=null) _owner.friendCars().Add(this);
```

```
        }
    }
    private Person _owner;
    ...
  class Person...
    public IList Cars{
      get {return ArrayList.ReadOnly(_cars);}
    }
    public void AddCar(Car arg){
      arg.Owner=this;
    }
    private IList _cars=new ArrayList();
    internal IList friendCars(){
      //should only be used by Car.Owner
      return _cars;
    }
...
```

首要的事是让关联的一方——如果可能的话，单值方——来控制这种关系。为此，从端（人）需将其封装的数据渗往主端（车）。这便对仆类增添了一个使用不便的方法，这一方法其实是不该在那里的，但语言具有细粒度的访问控制时则为例外。我曾经在这里利用过“朋友”的命名习惯来向 C++ 点头致意，其中主人的设置者确实是一位朋友。像很多特性代码一样，这是一种漂亮的书面材料，这就是为什么很多人都喜欢使用某种形式的代码生成程序来生成代码的原因。

在概念模型中,导航不是一个重要问题,因此,在概念模型上并未示明任何导航箭号。

操作

操作(operation)是类知道要去施行的动作。操作最明显地对应于类上的方法。一般说来,不去指明那些只是处置特性的操作,原因是,通常那些操作都是可以推断出来的。

UML 全集的操作语法是:

可见性　名(参数表):回送类型{特性串}

- *可见性*标记是公用(+)或私用(-);其他标记见第106 页。
- *名*是串。
- *参数表*是关于操作的参数的表。
- *回送类型*是回送值(如果有的话)的类型。
- *{特性串}*指出用于所给操作的特性值。

参数表中的参数按类似于属性的方式标记,其形为

方向　名:类型=默认值

- *名*、*类型*、*默认值*与关于属性所述者相同。
- *方向*指出参数是否为输入(入)、输出(出)或既输入又输出(入出)。如果未示明方向,则假定为入。

关于账务操作的一个例子是:

+balanceOn (date : Date) : Money

对概念模型,不应利用操作来指明类的接口,而是利用操作

指明类的主要职责。也许可以利用几个词来概括 CRC 卡的主要职责(第 80 页)。

我常发现，把改变系统状态的操作和不改变系统状态的操作加以区分是有益的。UML 把从一个类取得值而不改变系统状态的操作(换言之，无副作用的操作)定义为**恒态操作**(query)，可以对这样的操作标以特性串{query}。把改变状态的操作称为**改态操作**(modifier)，也称作命令。

严格说来，恒态操作与改态操作之间的区别是它们能否改变可观察到的状态[Meyer]。可观察到的状态是从外面可以看出的状态。更新快速缓存的操作会改变内态，但对从外面可观察到的现象来说，则是没有作用的。

我发现，突出恒态操作是有益的，原因是，你可以改变恒态操作的执行顺序，而不改变系统行为。通常的习惯是试图写一些操作，使改态操作不回送值，这样，便可以确信，回送值的操作是恒态操作。[Meyer]将此称作“改态操作(命令)-恒态操作分隔原理”。如果始终这样做，有时就会感到不便，但是，你应该尽可能多地这样做。

你有时看到的其他术语是获取方法与置送方法。**获取方法**(getting method)从一个域回送一个值(仅此而已)。**置送方法**(setting method)把一个值送往一域(仅此而已)。客户从外面不可能了解恒态操作是否是获取方法或者改态操作是否是置送方法。了解获取方法与置送方法完全是类内部的事。

操作和方法的另一区别是，**操作**(operation)是对对象提出的事(过程说明)，而**方法**(method)却是过程体。二者当你有多态时就显得不同。如果有一个具有三个子类型的超类型，每个子类型

都撤销超类型的 getPrice 操作,你就有一个操作和四个实现它的方法。

通常人们把操作和方法混用,但有时加以精确区分也是有益的。

泛化

泛化(generalization)的一个典型例子是涉及业务的个人客户和公司客户。个人客户和公司客户二者既有区别,又有很多类似之处。可以把类似之处放入一个通用客户类(超类型),它以个人客户类及公司客户类为其子类型。

这一现象也属于建模的不同视面之不同解释。从概念视面看,可以说公司客户类是客户类的一个子类型,如果公司客户类的所有实例按定义也是客户类的实例的话。这时,公司客户类便是一种特殊的客户类。关键的意义是,关于客户类所谈的每一件事——关联、属性、操作——对公司客户类也同样成立。

对软件视面来说,明显的解释是继承:公司客户类是客户类的子类。在主流的面向对象语言中,子类继承超类的所有特性,并且可以撤销超类的任何方法。

有效利用继承的一个重要原理是**置换性**(substitutability)。在任何一段要求有客户类的代码内部,应能将客户类置换成公司客户类,并且每件事都应进行顺利。本质上说,这表示,如果在我有一客户类的假定下写代码,我就可以自由使用客户类的任何子类型。公司客户类可以对别的客户类的某些命令作出不同的响

应(利用多态)，但命令发布者却不必担心这一区别(关于这方面的更多资料，请看[Martin]书中的 Liskov 置换性原理(LSP))。

虽然继承是一种强有力的机制，它却带来了不少负担，这些负担对于实现置换说来，并非总是必要的。Java 早期有一个这方面的好例子，那时很多人不喜欢内建(built-in)的向量(Vector)类的实现，希望用较为轻便的构造来取代它。然而他们能够产生一个可置换 Vector 的惟一方法就是对它构作子类，这便表示要继承不少不希望有的数据和行为。

很多别的机制可以用来提供可置换的类。因此，很多人喜欢区分子类型构作(或接口继承)与子类构作(或实现继承)。一个类是一个**子类型**(subtype)，如果它可以置换其超类型而不管是否使用继承。子类构作(subclassing)则用作常规继承的同义词。

还有很多别的机制可以使你有子类型构作而无子类构作。例如实现接口(第 89 页)以及很多标准设计模式[Gang of Four]。

注文与注释

注文是图中的注释。注文可以单放也可以用一条虚线将它和其所注释的成分相连接(图 3.6)。注文可在任何一种图中出现。

虚线有时使人感到不便，原因是你不能把它的端点放准。于是，通常习惯是把一个很小的开口圆放在虚线的线端。有时，在图成分上有一个直接插入的注释也是有益的。你可以在正文前附以两个短横线(--)来做这件事。

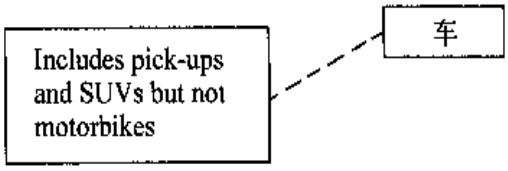


图 3.6 用作图成分上的注释的注文

依赖

如果改动一个成分(供应方(supplier))的定义可引起另一成分(客户方(client))的改动,则称在这两个成分之间存在一种**依赖**(dependency)。对类来说,存在依赖有各种原因:一个类把一个消息发送给另一类;一个类以另一个类作为其数据部分;一个类提到另一类作为操作参数。如果改动一个类的接口,那么,送往该类的任何消息就可能不再有效。

随着计算机系统的发展,你就越来越担心控制依赖。如果依赖失控,对系统的每一改动将引起越来越多的事情需要改动,因而导致广泛的涟漪效应(ripple effect)。涟漪越大,改动任何事情就越困难。

UML 使你能绘出各种成分间的依赖。当要示明一个成分中的改动如何引起其他成分改动时,就要使用依赖。

图 3.7 示明可以在多层应用中发现的一些依赖。*效益窗*类(用户接口或**表象**(presentation)类)依赖于*雇员*类:它是一个**领域对象**(domain object),用以获取系统的本质行为(在本例的情形,即业务规则)。这便表示,如果*雇员*类的接口改动,*效益窗*类就可

能需要改动。

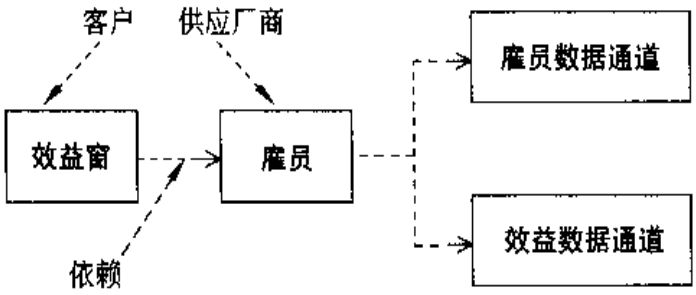


图 3.7　依赖一例

这里重要的事是，依赖只是单向的，从表象类到领域类。这样，我们就知道，可以自由改动效益窗，而对雇员或任何别的领域对象不会产生任何影响。我发现，严格把表象和领域分开的基理（表象依赖于领域，反之不然）已成为我所遵循的一条重要规则。

这个图上第二件值得注意的事是在效益窗类到两个数据通道类没有直接依赖，如果这两个类改动，雇员类可能需要改动。但如果只改动雇员类的实现而不改动其接口，改动就在那里停止。

UML 有很多种依赖，每一种都有特定的语义和关键词。我在这里扼要讲述的基本依赖是我发现最为有用的依赖，而且我通常都不带关键词地使用它们。为了增加更多的细节，你也可以加一个合适的关键词（表 3.1）。

基本依赖不是传递关系。**传递**（transitive）关系的一个例子是“较大胡须关系”。如果 Jim 的胡须比 Grady 大，且 Grady 的胡须比 Ivar 大，则可以推知，Jim 的胡须比 Ivar 大。像置换这样一种依赖是传递的，但在多数情形，如图 3.7 中所具有的，直接依赖与间接依赖之间却有明显区别。

表 3.1 所选的依赖基词

基 词	含 义
《call》	源调用目标中的操作
《create》	源创建目标的实例
《derive》	源由目标导出
《instantiate》	源是目标的一个实例(请注意,如果源是一个类,这个类本身就是类的一个实例,亦即,目标类为一元类)
《permit》	目标允许源访问目标的私用特征
《realize》	源是由目标定义的规约或接口的一个实现(第89页)
《refine》	精化指出不同语义层之间的一个关系。例如,源可以是一个设计类,而目标是相应的分析类
《substitute》	源可以置换目标(第58页)
《trace》	用于追踪诸如需求到类或者一个模型中的改动如何连接到别处的改动这样的事
《use》	源要求目标为其实现

很多UML关系都蕴含依赖。图3.1中从订单到客户的可导航关联的意思是,订单依赖于客户。子类依赖于超类,反之则不然。

你的通则应是使依赖减为极少,特别是,在它们跨越系统的大区域时。应特别谨慎提防循环,因为它们可以引起循环改动。对此我并不特别严格。我不在意紧密相关的类之间的相互依赖,但是,我却力图在更广的层次上消除循环,特别是一些包间的循环。

试图示明一个类图中的所有依赖是徒劳无功的练习；依赖太多，改动也多。选择并示明各个依赖仅当它们和你要交流的特定题目直接有关时才做。为了了解并控制依赖，最好对包图不要使用依赖(第110页)。

我对类使用依赖的最常见的情形是在阐明瞬时关系，比方说，把一个对象作为参数传递给另一对象时，你可以看到它们和《parameter》、《local》、《global》等基词一起使用。也可以在UML 1模型中的关联上看到这些基词，这时，它们指明的是瞬时的连接，而不是特性。这些基词在UML 2中却没有。

可以通过查看代码来确定依赖，因而，工具对于进行依赖分析是理想的。获取一项工具来求工程师所用的依赖图之逆乃是利用UML这一小部分的最有用的方式。

约束规则

绘制类图时做的很多事都是指明约束。图3.1指明，一份订单只能由单独一名客户发出。这个图的另一含义是，把每一行项均看作是独立的：在订单中，你说“40件棕色装饰品，40件蓝色装饰品，40件红色装饰品，”而不说“120件东西”。此外，这个图说明的是，公司客户有信用界限，而个人客户没有。

关联、属性以及泛化的基本构造对于指明重要约束起了很大作用，但是，它们不能指明每一个约束。这些约束仍然需要去获取。类图就是做这件事的好地方。

UML允许使用任何东西来表述约束。惟一的规则就是要把

约束置于两个花括号({ })之间。你可以使用自然语言、编程语言或 UML 的一种基于谓词演算的形式的对象约束语言(OCL)[Warmer and Kleppe]。利用形式的表示法可以避免有歧义的自然语言所引起的错误解释的风险。可是,它却引出由于作者和读者不真正了解 OCL 所引起的错误解释的风险。因此,除非你有一些对谓词演算感到舒服的读者,我愿意推荐使用自然语言。

你可以随意先安放名再后接一个分号来对约束命名(不许乱伦:夫妇一定不能是同胞兄妹)。

按契约设计

按契约设计是 Bertrand Meyer 开发的一种设计技术[Meyer]。这种技术是他开发的 Eiffel 语言的一个重要特点。但是,按契约设计并非 Eiffel 专用,它是一种可用于任何编程语言的重要技术。

断言是按契约设计的核心。**断言**(assertion)为一布尔陈述。除了有错情况,它恒为真;在有错时,它才为假。独特的是,只在排错时才查核断言,在生产性执行时并不查核断言。的确,程序从来不应假定要查核断言。

按契约设计使用了三种特定的断言:后置条件(post-condition)、前置条件(pre-condition)以及不变式(invariant)。前置条件与后置条件用于操作。**后置条件**是操作执行后"事情就该如此"的一种陈述。例如,如果我们定义了一个数的"平方根"操作,后置条件应呈如下形式,即,输入=结果×结果,这里的结果是输出,输入是输入值。后置条件是"用以表达我们做什么而不表达我们如何做"的一种有用方式。换言之,它是把接口和实现分开的一种有用方法。

前置条件是在操作执行前,我们指望事情如何的一种陈述,对平方

根操作，可以定义一个前置条件"输入>=0"。这一前置条件说的是，对一个负数施行"平方根"操作是一个错误，并且这样做的结果是未定义的。

骤然一看，这似乎是一种不好的想法，因为应该在某处核查，以确保"平方根"启用正常。重要的问题是，谁来负责做这件事。

前置条件明确了这一点，即由调用者负责核查。如果没有这样明确的职责陈述，则会要么核查过少（由于双方都假定对方负责），要么核查过多（双方都核查）。核查过多是一件坏事，因为它导致很多重复核查代码，从而会显著增加程序的复杂性。明确由谁负责有助于降低这一复杂性。调用者通过如下方法可以减少忘记核查的危险，即通常在排错和测试时都要核查断言。

根据前置条件和后置条件的定义，可以看出异常（exception）一词的深刻定义。异常发生在启用操作时，其前置条件满足，但该操作不能回送使后置条件满足的结果。

不变式是关于类的断言。例如，账务类可以有一个不变式，即

balance==sum(entries. amount())

这一不变式对该类的所有实例总为真。这里"总"的含义是，"只要对象有一个可在其上启用的操作"。

本质上，这就表示，不变式是加在与给定类所有公用操作相关的前置条件和后置条件之上的。在方法执行中，不变式可以变成假，但它在任何别的对象可对接收者做任何事时它就应该恢复成真。

断言对子类构作可以起到独特的作用。继承的危险之一是，你可能重新定义一个子类的操作，它们和超类的操作不相容。断言减少出现这种情况的机会。类的不变式和后置条件必须用于所有子类。子类可以选择加强这些断言，但却不能减弱它们。另一方面，前置条件却不能加强，但可以减弱。

乍一看来,这有些古怪,但它对于允许动态绑定却是重要的。你应该永远把子类对象看成就像超类的一个实例(按置换性原理)。如果一个子类增强了它的前置条件,则当它用于子类时,超类操作就可能失败。

何时使用类图

类图是 UML 的支柱,因此,你将发现你自己一直在使用类图。本章涵盖基本概念,第 5 章讨论很多高级概念。

类图的麻烦在于其内容太丰富,又不可避免地要使用它们。下面是一些使用提示:

- 不要试图使用对你可用的所有图示法。从本章的简单材料(类、关联、属性、泛化以及约束)开始,仅当需要时才从第 5 章中引进别的图示法。
- 我发现,概念类图(即从概念视面绘制的类图)在探究业务语言时很有用。为此,你必须努力工作,以使软件离开讨论并保持图示法很简单。
- 不要对每件事都绘制模型,而要集中考虑关键方面。少量使用至今的图要比很多被人遗忘而遭淘汰的模型更好。

用类图最大的危险是,你可能全神贯注于结构而忽略行为。所以,在绘制类图以了解软件时,总以连同使用某种形式的行为技术为宜。如果进行得好,你将发现你自己经常在这些技术之间进行交换。

何处找寻更多资料

第 1 章提到的所有 UML 的一般书籍都详细论述了类图。依赖管理是一些较大项目的关键点。关于这一题目的最好的书是[Martin]。

第4章

顺 序 图

交互图(interaction diagram)表述各组对象如何依某种行为进行协作的模型。UML 定义了若干形式的交互图,其中最常见的是顺序图。

典型的是,一个顺序图获取单个案况的行为。这种图示明一些样例对象以及在用案内部这些对象之间传递的消息。

为了开始讨论,我将考察一个简单案况。我们有一份订单并着手对它施行一道计算价格的命令。为此,订单需要查看其上的所有行项并决定这些行项的价格,这是以各个订单行的产品定价规则为基础的,对所有行项都这样做好后,订单需要计算一个基于顾客规则的总折扣。

图 4.1 是一个顺序图,它示明该案况的一种实现。顺序图通过每名参加者下方的垂线(生命线)以及各个消息依次向下的顺序来示明交互。

顺序图的一件漂亮的事是,我几乎对图示法无须解释。你可以看出一份订单实例把 getQuantity 和 getProduct 消息发送给订

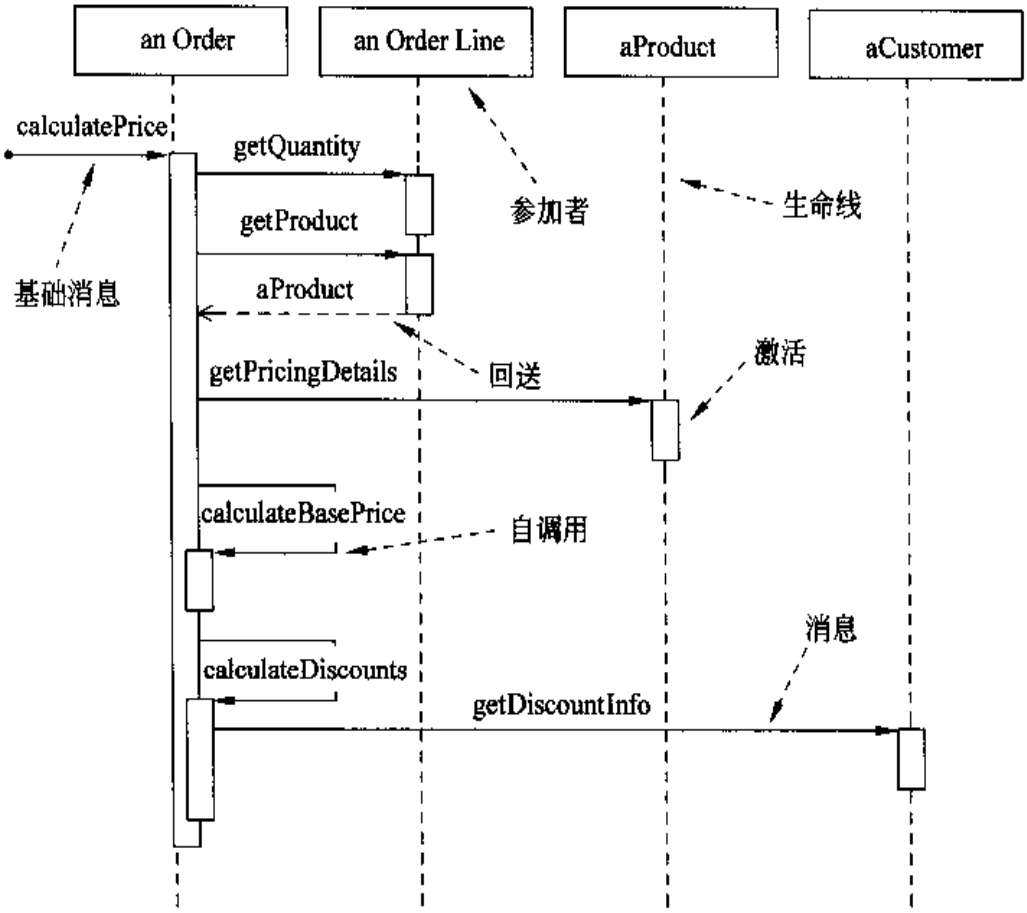


图 4.1　集中式控制顺序图

单行，也可以看出我们如何来示明订单施行自己的方法以及该方法如何把 getDiscountInfo 送给客户的一个实例。

然而，这张图并未能很好地示明每一件事。getQuantity，getProduct，getPricingDetails 和 calculateBasePrice 这一列消息对订单上的每一订单行都需施行，而 calculateDiscounts 却只要启用一次。你却不能从这张图上分辨出这一点，虽然往后我将要引进更多的表示来对之处理。

多数时间,你可以像在 UML 1 中那样,把交互图中的参加者想象成对象。但是,在 UML 2 中,参加者的作用就大大复杂,而对它进行全面阐述已经超出本书范围。因此,我利用一个在 UML 规范中未曾正式使用的词**参加者**(participant)。在 UML 1 中,参加者是对象,它的名是带下划线的,但在 UML 2 中,正如我在这里所做的,为了示明它们,无须加下划线。

在这些图中,我曾经利用 anOrder 样式对参加者命名。多数时间,这样工作很好。比较完整的语法是 name:Class,这里的名和类都是任选的,但是,如果你利用类,就必须保留冒号(示于第 75 页的图 4.4 就利用这一样式)。

每一条生命线都有一个激活框,它示明在交互中参加者何时在起作用。这对应于栈上一个参加者的方法。在 UML 中激活框是任选的,但我发现,它们在阐明行为中极为有用。我的一个例外是在设计聚会中探究设计的时候,这是因为在白板上画激活框不方便的缘故。

命名往往对图中各个参加者相互呼应有用。调用 getProduct 示为回送 aProduct,它是同名,因此,是同一名参加者,原因是 aProduct 就是 getPriceDetails 调用所送往的。请注意,我只对这一调用使用了回送箭号,这样做为的是示明对应。有些人对所有的调用都使用回送,但是我却喜欢在能增添信息时才去使用,否则,它们只会把事情弄混。即使在这种情形(即在能增添信息的情形),你可能也要省去回送,以避免读者混淆。

第一个消息并无参加者发送,原因是它来自一个不确定的源。这个消息称为**基础消息**(found message)。

关于这一案况的另一途径，让我们来看一下图 4.2。基本问题仍然相同，但是参加者协同实现的方式却很不同。订单要求每一订单行计算其自身的价格，订单行本身进一步把这一计算递交给产品，请注意，我们是如何示明参数传递的。类似地，为了计算折扣，订单启用客户上的一个方法，为此，它需要来自订单的信息，客户对订单作一次再入调用(getBaseValue)以获取数据。

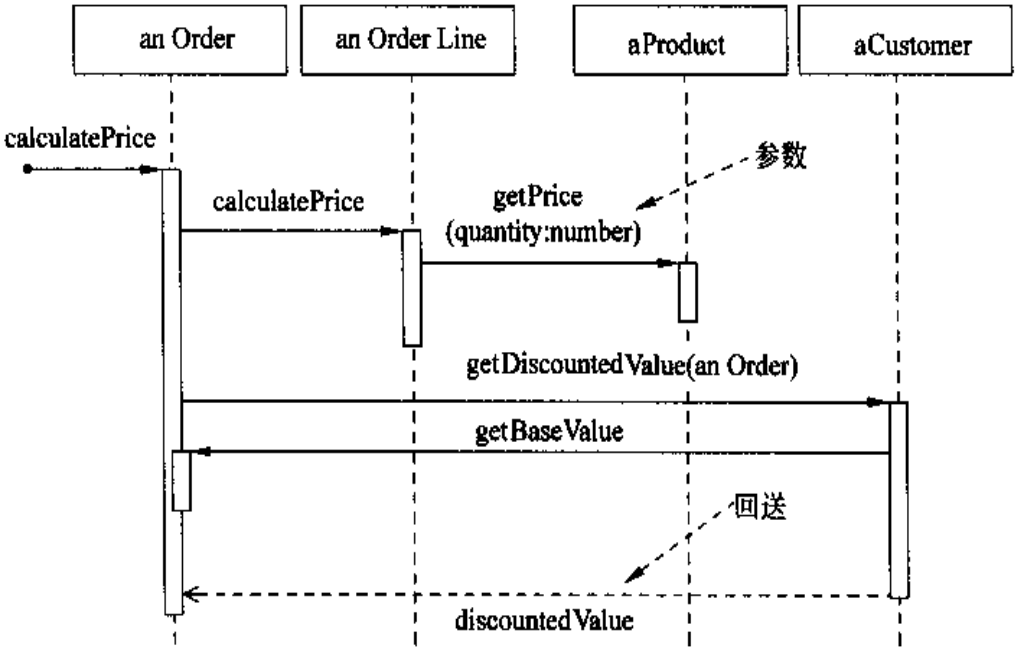


图 4.2　分布式控制顺序图

关于这两张图要注意的第一件事是，顺序图如何明晰地指出参加者交互方面的区别，这是交互图的最大优点。交互图并不擅长于示明诸如循环与条件行为等算法细节，但却能使参加者之间的调用一目了然，并对“什么参加者正在进行什么处理”给出一个确实好的描绘。

要注意的第二件事是两种交互的方式迥异。图 4.1 是**集中式控制**,其中一名参加者进行所有处理,其他参加者只提供数据。图 4.2 利用**分布式控制**,其中把处理分摊给多名参加者,每人只处理算法的一小部分。

两种方式各有利弊。大多数人(特别是对象新手)更习惯使用集中式控制。在很多方面,集中式控制是比较简单的,原因是所有的处理都在一处进行;相比之下,对分布式控制,你会感觉到频繁交往于各个对象之间以找寻程序。

虽然如此,像我这样执著于对象的人却强烈爱好分布式控制。好的设计的主要目标之一就是使变动的影响局部化。数据与访问该数据的行为往往一同变动,因此,把数据和使用该数据的行为一同放在一处,就成为面向对象设计的首要规则。

此外,借助于分布式控制,你可以创建更多使用多态而不使用条件基理的机会。如果产品定价算法随产品类型而异,分布式控制机制就允许你利用产品的子类来控制这些变化。

一般说来,面向对象风格是利用许多带有小方法的小对象,这些小方法使我们有很多便于撤销与改变的接插点。这一风格对惯于使用长过程的人来说是很容易弄不清的;的确,这一改变乃是面向对象的**风范转移**(paradigm shift)的核心。这是非常难教的事。看来,真正弄懂它的惟一办法就是在具有强分布式控制的面向对象环境中工作一段时间。很多人在感到这种风格有意义时,会豁然开朗,这时他们的头脑重新兴奋起来,并且开始认为分散性控制实际上更容易一些。

参加者的创建与删除

顺序图示明某种关于创建与删除参加者的特别图示法(图 4.3)。为了创建一名参加者,你直接往参加者框画一消息箭号。如果使用一个构造符,这里的消息名便是任选的,但是,我通常在任何场合都标以“new”。参加者一经创建,便立即工作(像恒态命令那样),你就紧接在参加者框之后开始激活。

删除参加者是用大写×指出的。朝向×的消息箭号指出,一名参加者显式地删除另一名参加者;生命线端的×示明参加者自身删除。

在废区回收的环境下,不要直接删除对象,但仍然值得利用×指出何时不再需要一个对象并准备对它回收。它对关闭操作也是合用的,借以指出该对象不再可用。

循环、条件等

顺序图的一个常见问题是如何示明循环行为与条件行为。首先要指出的是,这并非顺序图之所长。如果你要示明这样一些控制结构,采用活动图或者甚至代码本身会更好。把顺序图看作各个对象如何交互的形象化表示而不是一种对控制基理的建模方法。

说了那一点之后,这里是要用的图示法。循环及条件二者均

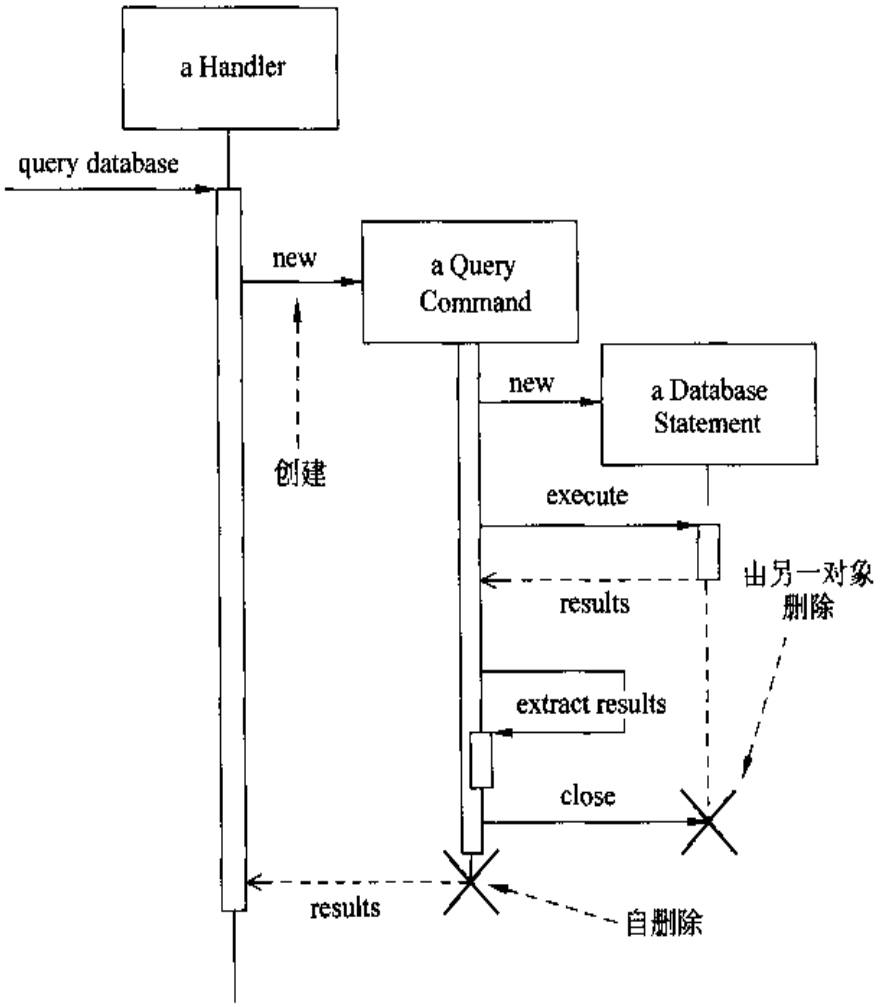


图 4.3 参加者的创建与删除

利用**交互架构**(interaction frame),它是一种标记顺序图片段的方法。图 4.4 示明一个基于下述伪码的简单算法:

```
procedure dispatch
    foreach (lineitem)
        if (product.value> $ 10k)
```

```
            careful.dispatch
        else
            regular.dispatch
        end if
    end for
    if (needsConfirmation) messenger. confirm
end procedure
```

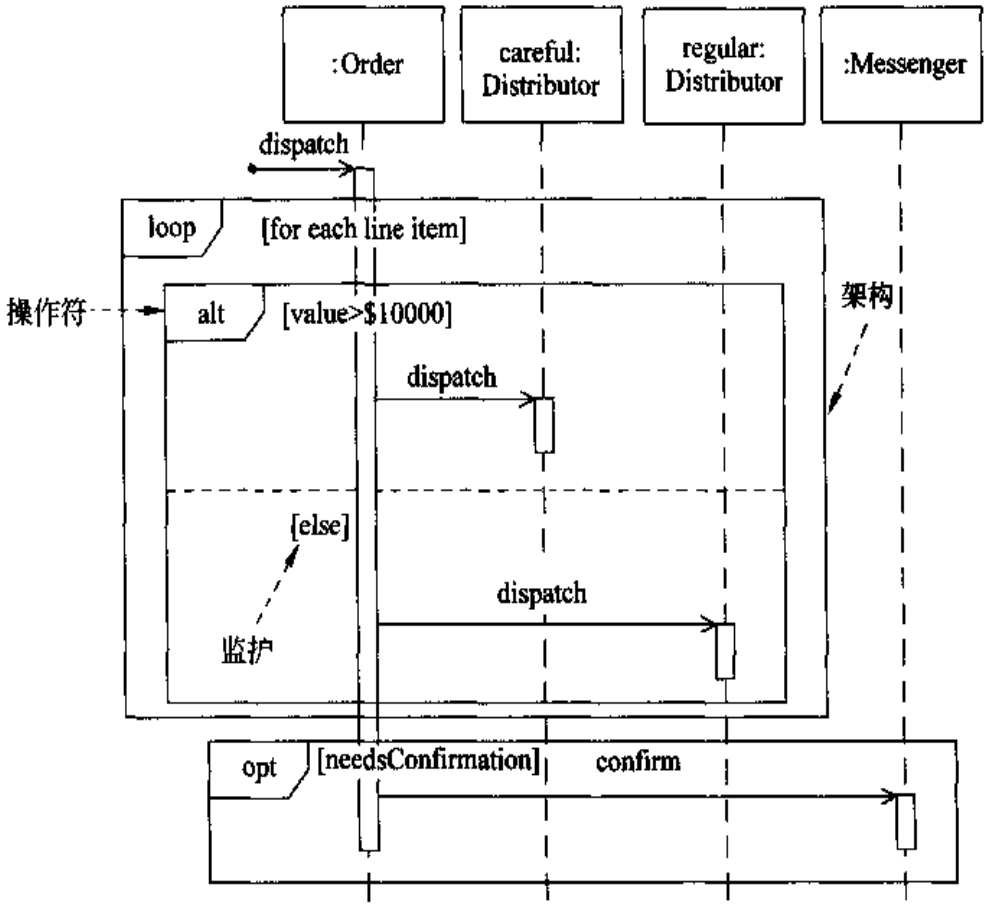


图 4.4、交互架构

一般说来,架构包含分成若干片段的一个顺序图的某一区域。每个架构有一个操作符,每一片段有一名监护(表 4.1 中列出了交互架构的一些常用操作符)。为了示明循环,对单一片段利用 loop 操作分量(似应为操作符——译注),并把迭代基置于监护之下。对条件基理而言,可以利用操作符 alt 并把条件置于每一片段上,只有监护为真的片段才执行。如果只有一个区域,就有一个 opt 操作符。

表 4.1 交互架构的常用操作符

操作符	含 义
alt	供选多重片段;只有条件为真的片段才执行(图 4.4)。
opt	任选;片段仅当所提供的条件为真时才执行。与只有一条路线的 alt 等价(图 4.4)。
par	并行;每一片段都并行运行。
loop	循环;片段可执行多次,监护指出迭代基(图 4.4)。
region	临界区域;片段只能有一个线程对它立即执行。
neg	否定;片段示明一次无效交互。
ref	指引;指引在别的图上定义的一个交互。把架构绘成涵盖交互中所含的各条生命线。你可以定义参数和回送值。
sd	顺序图;如你愿意,用它来围绕整个顺序图。

交互架构是 UML 2 中的新成分。因此,你可以看到,UML 2 以前所准备的一些采用不同方法的图。也有一些人不喜欢架构而喜欢较老的习惯。图 4.5 示明这样一些非正式的回转。

UML 1 使用了迭代标记和监护。**迭代标记**(iteration marker)是一个附于消息名的 *,你可以在两个方括号之间加上

一段正文以指出迭代基。**监护**(guard)是置于两个方括号间的条件表达式,并指出消息仅当监护为真时才发送。虽然这些图示法在 UML 2 的顺序图中已经不用,在通信图上它们仍是合法的。

尽管迭代标记和监护都是有用的,但它们也有缺点。监护不能指出一组监护(如图 4.5 上的两名监护)彼此互斥。这两种图示法只能在具有单个消息发送时工作,而当有来自单个激活的几条消息位于同一个循环或条件块之内时则不能很好地工作。

为了解决上述问题,现已成为流行的一种非正式的习惯是使用带有循环条件或自调用表示变形上的监护的**伪消息**(pseudomessage)。在图 4.5 中,我示明了这一点而不带消息箭号。有些人则带消息箭号,但是,去掉消息箭号却有助于强调这并不是一个实在的调用。有些人还喜欢在伪消息的激活框上加灰度。如果你有别的行为,就可以利用激活之间的别的标记来示明。

虽然我发现激活很有用,它们在分派(dispatch)方法的情形并未增加过多的信息,在那里,除了发送一个消息外,在接收者的激活内部其他任何事均未发生。我已在图 4.5 上示明的通常习惯是,对那些简单的调用去掉激活。

UML 标准并未指明传递数据的图示设备,而这是用消息名及回送箭号中的参数来示明的。一些**数据蝌蚪**(data tadpole)遍布于很多方法中借以指出数据的移动。很多人对 UML 仍然喜欢使用它们。

总之,虽然各种方案都可以对顺序图增添条件基理表示,但是,我并未发现,它们要比代码或者至少伪代码工作得更好。特别是,我发现交互架构太繁重,致使图的要点难于明晰。因此,我喜欢伪消息。

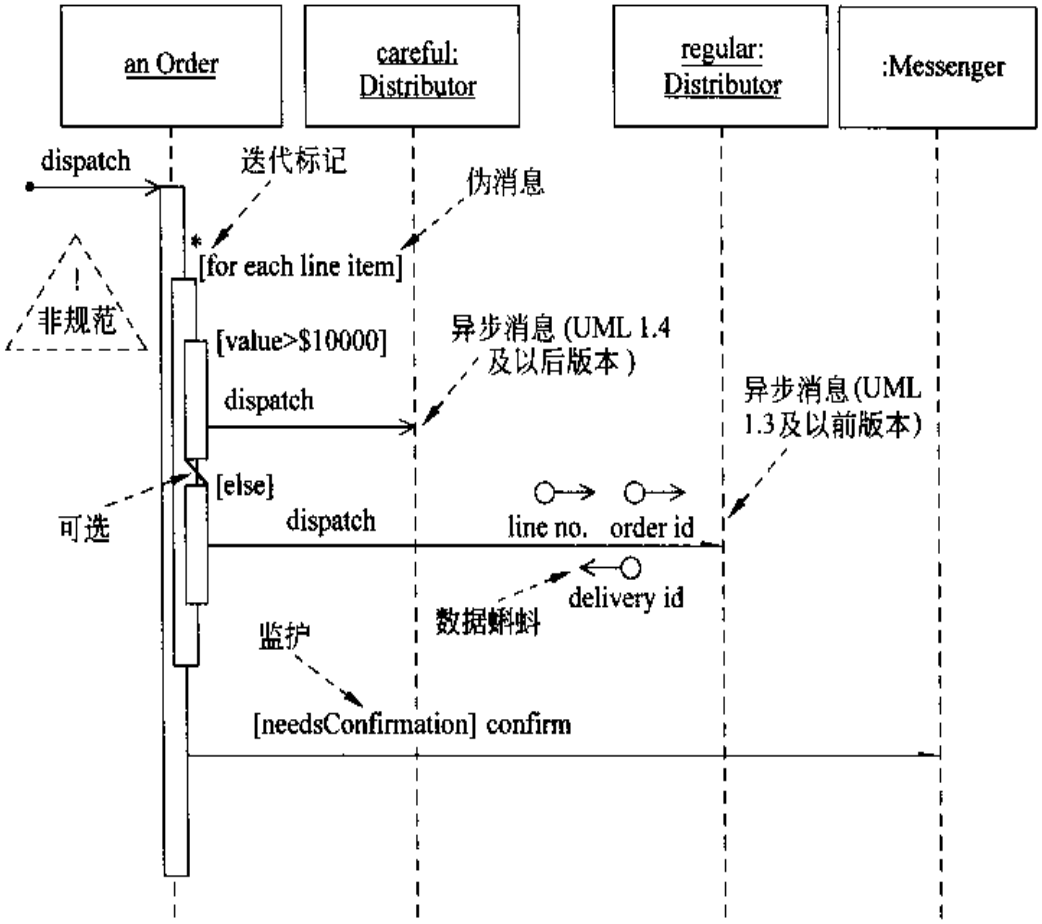


图 4.5 对控制基理的较老习惯

同步调用与异步调用

如果你是一位高度警觉的人,将已经留意到图 4.4 和图 4.5 中的箭头与稍早所用的箭头不一样。那种细微的区别在 UML 2 中非常重要。在 UML 2 中实心箭头(—▸)示明同步消息,而实线

箭头(—>)则示明异步消息。

如果调用者发送一个**同步消息**(synchronous message),它就必须一直等到消息发送成功,例如激活一个子例程。如果调用者发送一个**异步消息**(asynchronous message),它就可以继续处理,无须等待响应。在多线程应用及面向消息的中间件中可看到一些异步调用。异步性给出更好的响应性并减少时间连接,但是却较难排错。

箭头的区别是微妙的,实在是有点过于微妙。这也是在 UML 1.4 中所引进的一种向后不匹配的改动。在此之前,异步消息是用半实线箭头(—>)指明的(如在图 4.5 中)。

我认为,这样箭头上的区别过于微妙。如果你要突出异步消息,我倒愿意推荐使用老式的半实线箭头,它能使你的目光更好地瞄准重要的区别。如果你正在阅读一个顺序图,就要提防从箭头上所做的关于异步的假定,除非你确信,作者是有意加以区别的。

何时使用顺序图

当你要考察单个用案内部若干对象的行为时,就应使用顺序图。顺序图擅长于示明对象间的协作;它们却不大擅长于示明行为的精确定义。

如果要考察跨用案的单个对象的行为,就使用状态图(见第 10 章)。如果要考察跨用案或跨线程的行为,就考虑活动图(见第

11章)。

如果要迅速探究多个供选交互,则使用CRC卡更好,原因是,那样可以避免不少绘制及擦去工作。举行一次CRC卡聚会探讨设计选择非常方便,而后再使用顺序图来获取往后要提到的任何交互。

其他形式的有用交互图是用以示明连接的通信图以及用以示明时间约束的定时图。

CRC 卡

在针对好的面向对象设计想出的最有用的技术之一是探究对象交互,这是因为它集中考察的是行为而不是数据。20世纪80年代后期由Ward Cunningham发明的类-职责-协作(Class-Responsibility-Collaboration)图(简称CRC图)经受了时间的考验,已经成为进行这项工作(指的是探究对象交互——译注)的一种极为有效的方法。虽然CRC图不是UML的组成部分,但在熟练的对象设计人员中却是一种颇为流行的技术。图4.6给出了CRC卡的一个样例。

为了利用CRC卡,你和你的同事们聚集在一张桌子周围,取几个不同的案况,用卡片进行模拟。当被激活时,就举起卡片,移动它们以看出它们是如何发送消息的,并向四周传递。这种技术几乎无法在书中表述,但却是容易演示的。对它最好的学习方法就是由一位亲身经历过的人进行示范。

CRC想法的一个重要部分是识认职责。职责(responsibility)是一个短句,它概括了一个对象应做的事;例如,对象施行的一个动作,对象

图 4.6　CRC 卡样例

持有的某种知识，或者对象进行的某些重要选定。目的是，你应能取任意一个类并利用少量职责对它概括。这样做，能帮助你更清楚地构想各个类的设计。

第二个字母 C 指**协作者**(collaborators)，即这一类需要与之协作的别的类。这使你能对各类之间的连接有某一想法——仍然是居于高层的。

CRC 卡的一个主要好处是，它们鼓励开发人员之间的活跃讨论。当你正在检查一个用案，要弄清楚各个类是如何实现这一用案时，本章的交互图绘制起来就嫌慢。通常需要考察供选情况。而用图的话，绘制或改动供选情况所花的时间就会长得令人难以忍受。利用 CRC 卡，可以通过收集整理卡片并来回移动来对交互建模，这就使你能迅速考察供选情况。

当你这样做时，便形成一些职责概念，并把它们写在卡片上。思考职责是重要的，这是因为它使你摆脱作为哑数据持有者的类，并使项目

组成员便于理解每一类的高层行为的缘故。一项职责可以对应一个操作、一个属性,或者,更可能的是对应一堆未确定的属性和操作。

我看到人们所犯的一种常见的错误是,生成很长的低层次的职责表。这实在是未能抓住要点。职责应能轻而易举地容身于一张卡片之内。我总要对具有三项以上职责的卡片提出质问。问一下你自己,是否应该把类分解,或者把职责合并成一些高层陈述而叙述得更好一些。

很多人强调角色扮演的重要性。这里项目组中每人扮演一类或多类角色。我从未见过 Ward Cunningham 做过角色扮演,并且我发现,这样做会碍事。

市面上有一些论述 CRC 的书,但是我发现,它们从未真正触及到这一技术的核心。Ward Cunningham 和 Kent Beck 合写的原始文章是[Beck and Cunningham]。要想学到 CRC 卡和关于设计中的职责更多的东西,可以看一下[Wirfs-Brock]。

第 5 章

类图：高级概念

第 3 章中表述的概念对应类图中的基本图示法。那些是首先要了解并熟悉的概念，这是因为在类图构作中 90%的工作都要用到它们。

但是，类图技术已衍生出许多用于高级概念的图示法。我发现，我并非一直使用它们，但当感到它们合适时，用起来比较方便。我将对它们逐一进行讨论，并指出使用中的一些问题。

你可能会发现，本章有些艰涩。好消息是在首次览读本书时，你可以安全地跳过本章，以后再转回来阅读。

基词

对图示语言的指责之一是你必须记住一些符号的含义。由于符号太多，用户很难记住所有符号的含义。因此，UML 往往试图减少符号数目，而使用一些基词。如果你发现，需要这样一个

建模构造,它在 UML 中没有,但和 UML 中的某一构造类似,你就使用这个现有的 UML 构造,但要标以一个基词,以指明稍有区别。

接口就是一个这样的例子。**UML 接口**(第 89 页)是一个只有公用操作、不具方法体的类。这对应于 Java、COM(构件对象模块)和 CORBA 中的接口。因为它是一种特殊的类,于是便用具有基词《interface》的类图符来指明。**基词**(keyword)通常是示为两个双重尖括号之间的正文。作为基词的另一选择,你可以使用特定图符,但这时你又撞上了人人都要记住其含义的问题。

有些基词(如{abstract})是示于两个花括号之间的。至于从技术上看,哪些基词应置于双重尖括号之间,哪些基词应置于花括号之间,实际上从来都是不清楚的。幸运的是,如果你用错,只有严格的 UML 讨厌鬼才会留意或在意。

有些基词非常普遍,往往可以缩写:《interface》可缩写成《I》,{abstract}可缩写成{A}。这样一些缩写非常有用,特别是在白板上,但并不标准。因此,如果你要使用它们,就要确信,你会有地方把它们的含义讲清楚。

在 UML 1 中尖括号主要用于**衍型**(stereotype)。在 UML 2 中,对衍型是非常贴切定义的,而表述何者是衍型何者不是衍型已经超出了本书的范围。然而,由于 UML 1 的关系,很多人利用术语衍型,其含义和基词相同,虽然这已不再正确。

衍型用作侧图的部分。**侧图**(profile)是为了特定目的(如业务建模)用一组协调的衍型进行扩张过的 UML 的一部分。侧图的全部语义超出了本书的范围。除非进入严格的元模型设计,你不可能自己来创建侧图。更可能的是,使用一个已为特定建模目

的创建的侧图。但幸运的是，使用侧图并不要求了解关于侧图是如何联系到元模型的那些骇人听闻的细节的。

职责

往往在类图中的类上示明职责(第 80 页)是方便的。示明职责最好的方法是在类中自己的隔间里(图 5.1)给出注释行。如果愿意，可以对隔间命名，我通常并不这样做，这是因为出现混淆的可能性很小的缘故。

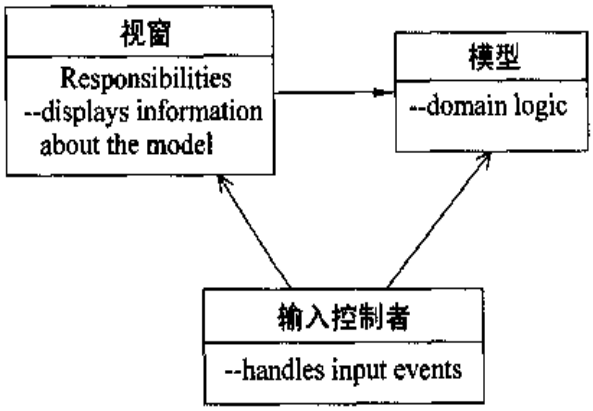


图 5.1　在类图中示明职责

静态操作与静态属性

UML 把施于类而不是施于实例的操作或属性称为**静态**(static)操作或属性。这等价于基于 C 的语言的静态成员。在类

图上静态特征是带下划线的(见图 5.2)。

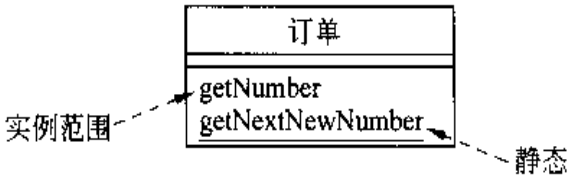


图 5.2 静态图示法

聚合与组合

UML 中最常见的混淆之一是聚合与组合。这是很容易加以解释的。**聚合**(aggregation)是整体-部分关系。就如同说,一辆车以引擎和车轮为其两部分。它听起来很好,但困难的事是什么是聚合和关联的区别。

在 UML 以前,人们通常对什么是聚合与什么是关联比较模糊。不管模糊与否,它们总是和任何别的关系不相容。因此,很多建模人员认为聚合重要,即使他们所依据的理由不同。于是,UML 包含了聚合(图 5.3),但几乎没有任何语义。正如 Jim Rumbaugh 所说,"把它想象成一种建模安慰剂"[Rumbaugh, UML Reference]。

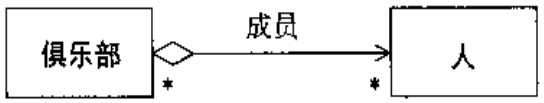


图 5.3 聚合

除聚合外，UML 具有另外定义的**组合**(composition)特性。在图 5.4 中，点的一个实例可以是多边形的部分或者也可以是圆心，但不能二者兼是。通则是，虽然一个类可以是多个其他类的成分，任一实例却必须只能是一个拥有者的成分。类图可以示明可能拥有者的多重类，但是，每一实例却只能有单独一个对象为其拥有者。

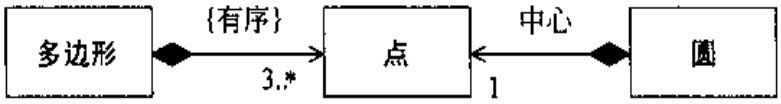


图 5.4　组合

你将注意到，在图 5.4 中我并没有示明逆重数。在很多情形，都和这里一样，它是 0..1，它的仅有的另一个可能的值是 1，这是针对其中成分类设计成只可能有一个别的类为其拥有者的情形。

这一条“非共享”规则对组合来说是关键的。另一个假定是，如果删除多边形，便应自动确保这个多边形所拥有的任何点也都被删除。

组合是示明按值拥有的特性、值对象(第 93 页)的特性，或者对其他特定成分具有强烈并稍有排它的拥有性等特性的好方法。聚合是完全没有意义的。因此，我建议，你在自己的图中略去聚合。如果在别人的图中见到聚合，你就要深入挖掘以找出它们的含义。不同的作者和开发组使用聚合的目的可能很不同。

导出特性

导出特性(derived property)是可以根据其他值计算出来的特性。当我们考虑一个日期范围(图 5.5),就可能想到三个特性:起始日期、结束日期以及这段期间的天数(即长度)。这些值是有联系的,于是我们可以把长度看成是从其他两个值导出的。

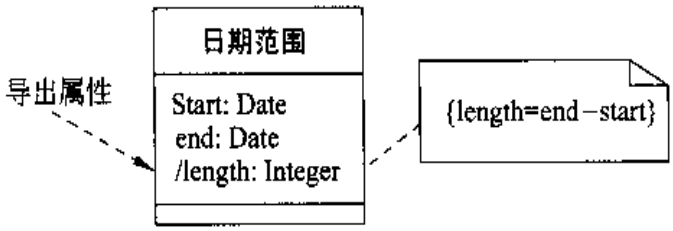


图 5.5 时间区间中的导出属性

软件视面中的导出可以按两种不同的方式来解释。你可以利用导出指出一个算出值与一个储存值之差。在这种情形,就要把图 5.5 解释成起始日期与结束日期都是储存值,长度是算出值。虽然这是一种通常的用法,我对它却并不很热衷,原因是,它对日期范围的本质揭露过多。

我喜欢的想法是,它表示各个值之间的一个约束。在本例的情形,我们说,这三个值之间保持有一种约束,至于这三个值当中何者是算出值并不重要。这时把什么属性标为导出属性的选择是随意的并且是完全不必要的,但有用的是,可以帮助人们想起约束。这种用法对概念图也是有意义的。

导出也可以用于利用关联图示法的特性。这时,你只用一个

“/”来标记名。

接口与抽象类

抽象类(abstract class)是不能直接被初启的类。而你却可以初启子类的一个实例。典型的是，一个抽象类具有一个或多个抽象操作。**抽象操作**(abstract operation)是不具实现的操作，它是纯粹的说明，因而客户可定绑于抽象类。

指出 UML 中抽象类或抽象操作最常用的方法是将名用斜体表示。也可以使特性成为抽象，只要指出它是一个抽象特性或访问者方法即可。斜体字在白板上很难写，因此，可以利用标号：{abstract}。

接口是一个不具实现的类，亦即，接口的全部特征都是抽象的。接口直接对应于 C# 与 Java 中的接口，并且在其他有类型的语言中也是常见的用法。用基词《interface》标记接口。

针对接口，类有两种关系：即提供与需要。如果一个类可置换一接口，则称该类**提供一个接口**(provides an interface)。在 Java 和 .NET 中，一个类可以通过实现接口或实现接口的一个子类型来做这件事。在 C++ 中，通过对接口类构作子类来达到这一目的。

如果一个类为了进行工作而需要接口的一个实例，则称该类**需要一个接口**(requires an interface)。本质上说，这是对接口的一个依赖。

图 5.6 基于 Java 中不多的几个类示明了作用中的这些关系。

我可以写一个具有一列行项的订单类。因为我在利用一个表,订单类就依赖于表的接口。假定它使用了 equals、add、get 几个方法。当对象相连时,订单实际上将利用数组表的一个实例,但无须知道这是为了利用那三个方法,这是由于它们全都是表接口的组成部分的缘故。

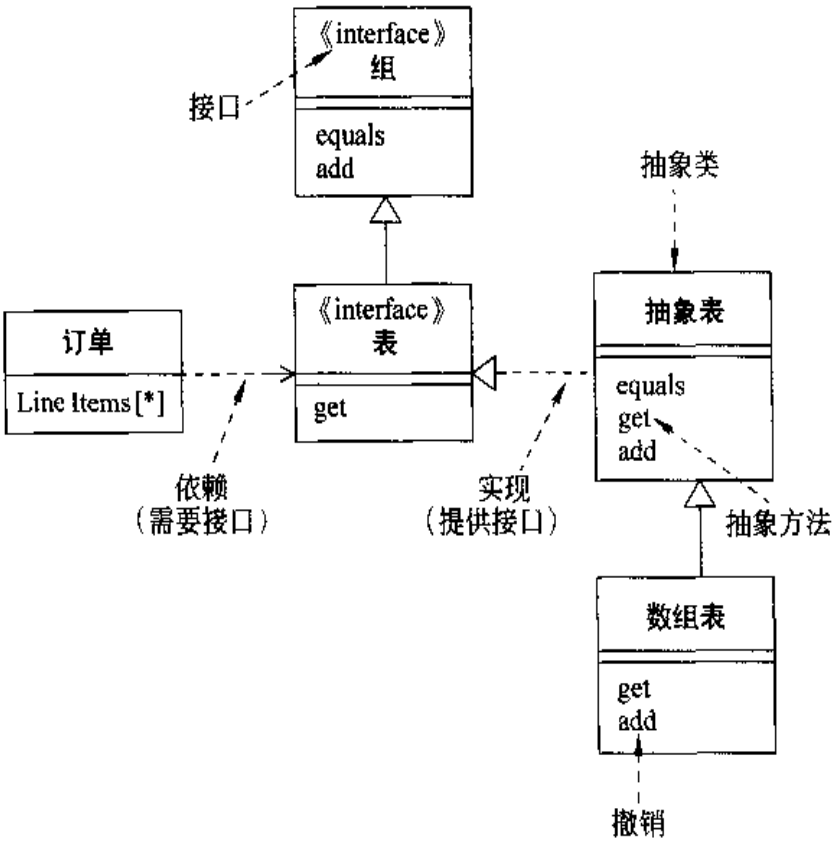


图 5.6 接口和抽象类的 Java 例子

数组表本身是抽象表类的一个子类,抽象表提供了某些(并非全部)表行为的实现,特别是,get 方法是抽象方法,因此,数组表实现了 get 但也撤销了抽象表上一些别的操作。在这种情形,它撤销了 add,但愉快的是,继承了 equals 的实现。

为什么我不只避免这一点，而要订单直接使用数组表呢？通过利用接口，使我以后在需要时能比较容易改动实现。另一种实现则可以提供性能改进、某些数据库的交互特征或其他好处。通过对接口而不是对实现编程，我避免必须改动所有代码，如果我需要表的不同实现的话。你应该一直试图这样来对接口编程，即总是利用你能用的最一般的类型。

我也应该指出其中一点有实效的好处。当编程人员使用这样的数组时，他们通常是用如下说明来初启这个类组的：

```
private List lineItems=new ArrayList();
```

请注意，这就严格地引进了一个由订单类到具体数组表类的依赖。理论上说，这是一个问题，但人们在实践中并不为此担心。因为行项的类型说明为表，订单类并无别的部分依赖于数组表类。如果要改动实现，需要担心的只有这一个初始化代码行。非常普遍的是，在创建中指向具体类一次，以后只使用接口。

图 5.6 的全部图示法是表述接口的一种方法。图 5.7 示明一种更为紧凑的图示法。数组表类实现表类以及用球形图符示明类组一事往往称作从中发出的小连珠。订单类需要一个表接口一事是用托座图符来示明的。联接相当明显。

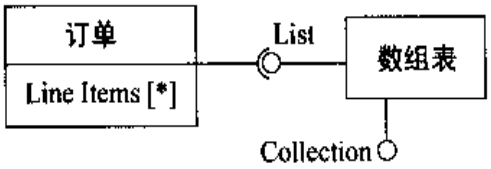


图 5.7　球形-托座图示法

UML 已经使用了连珠图示法一段时间，但托座图示法对

UML 2 还是新引进的(我认为,它是我所钟爱的一种图示法方面的增添)。可能你将会看到如同图 5.8 样式的老图,其中依赖代替了托座图示法。

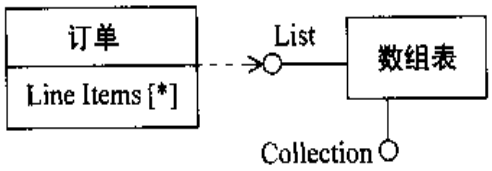


图 5.8 带连珠的较老依赖

任何类都是一个接口和一个实现的结合体。所以,我们往往会看到通过它的一个超类的接口来使用对象。严格说来,对超类利用连珠图示法不能认为是合法的,这是因为超类是一个类,不

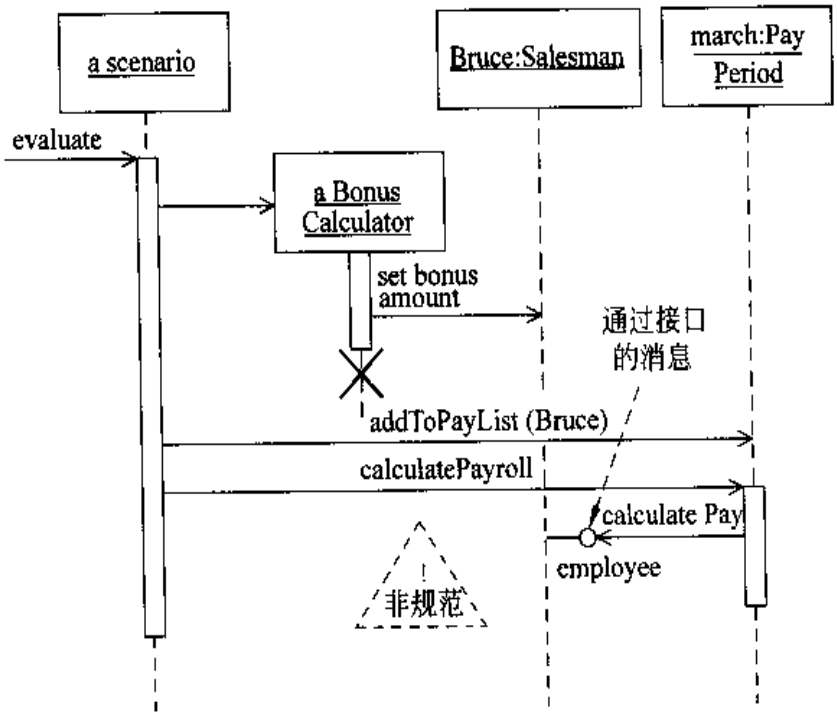


图 5.9 顺序图中多态的连珠示明法

是一个纯粹的接口。但是，为清晰起见，我就变通使用这些规则。

除了在类图上外，人们已发现连珠图示法用于别处。交互图的一个长期难解的问题是，它对多态行为未能提供出很好的形象化表示。虽然它不是规范用法，你却可以沿着图 5.9 的路来指出这一点。这里我们可以看出，虽然有一个销售人员实例，它是由 Bonus Calculator 这样用的，付款期(Pay Period)对象只能通过其雇员(Employee)接口来使用销售人员(Salesman)(也可以用通信图来玩同样的把戏)。

只读与冻结

在第 46 页上我表述了{readOnly}基词。你使用这一基词标记这样一个特性，它只能由客户读，却不能更新。与它类似但又不同的是 UML 1 中的基词{frozen}。如果一个特性在对象的生命期中不能改变，则称这一特性是**冻结的**(frozen)。这样的特性有时称为不可改变的特性或永恒特性。虽然冻结已在 UML 2 中取消，{frozen}却是一个很有用的概念，因此，我愿意继续使用它。除了用以标记冻结个别特性外，也可以把{frozen}基词用于类，以指出其所有实例的所有特性都已冻结(我已听说短期内冻结可能要恢复使用)。

指引对象与值对象

关于对象所说的一件共同的事就是它们都有身份。这是对

的,但它并不如此简单。在实践中,你会发现,对指引对象,身份是重要的,但对值对象,身份并不这样重要。

客户是一个**指引对象**(reference object)。在这里,身份非常重要,这是因为你为了表示现实世界中的一名客户,通常只需要一个软件对象。指引一客户对象的任何对象是通过一个指引或一个指引元来实现的。指引该名客户的所有对象将指引同一个软件对象。只有这样,对一名客户的改动对该名客户的所有用户才都可用。

如果你对一名客户有两个指引,并且希望看出二者是否相同,通常要比较它们的身份。复本可能是不允许的。如果允许复本,则希望尽量少用,或许这是为了归档或跨网络复现的缘故。如果制作了复本,则要对其整理分类,以使改动同步。

日期是一个**值对象**(value object)。表示现实世界中同一个对象往往有多个值对象。例如,表示 1-Jan-04 有几百个对象,这是正常的。这些全都是可换用的复本。新的日期经常会创建和消亡。

如果有两个日期并希望看出它们是否相同,则无须查看它们的身份,而只需查看它们表示的值。这通常是指,必须写一个相等测试操作符,后者对日期要进行年、月、日(或其内部表示)的测试。通常指引 1-Jan-04 的每一对象都有自己的专用对象,但你也可以共享日期。

值对象应是不可改变的。换言之,不能取 1-Jan-04 的一个日期对象而把这同一个日期对象变为 2-Jan-04。你应该创建一个新的 2-Jan-04 对象,而来使用它。理由是,如果日期共享,就会不可预测地更新另一对象的日期。这一问题称为**取别名**(aliasing)。

随着时间的推移，指引对象和值对象之间的区别就更为清楚。值对象是类型系统的内建值。现在可以用你自己的类来扩张类型系统。因此，这一问题要求有更多的思考。

UML 使用**数据类型**(data type)概念，它示明为类符号上的一个基词。严格说来，数据类型和值对象不同，理由是，数据类型不能有身份，值对象可以有身份，但不能把它用于相等关系。虽然 Java 的原始数据类型和日期都是值对象，但前者是数据类型，后者则否。

如果突出它们是重要的话，在联系有一个值对象时，就使用组合。也可以对值类型使用一个基词，我见到的常用基词是《value》或《struct》。

受限关联

受限关联(qualified association)是关联数组、映像、散列以及词典等不同称呼的编程概念的 UML 等价语。图 5.10 示明一种表示订单(Order)类和订单行(OrderLine)类之间关联的方法，它使用一个限定符。该限定符说的是，关于一份订单，对产品(Product)的每一实例，可有一个订单行。

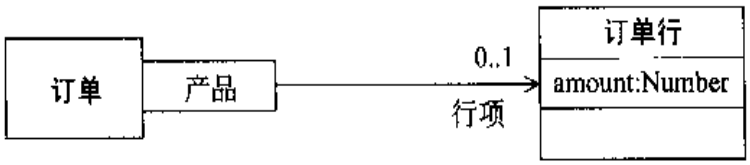


图 5.10　受限关联

从软件视面看,这一受限关联就指一个依照

```
class Order...
    public OrderLine getLineItem(Product aProduct);
    public void addLineItem(Number amount,Product
                                forProduct);
```

的接口。

这样,对一个给定行项(Line Item)的所有访问都要求有一项产品为其变元。提出一种实现,后者利用一个带键的值数据结构。

通常人们会混淆受限关联的重数。在图 5.10 中,一份订单可有很多行项,但受限关联的重数乃是在限定符方面的重数。因此,这张图说的是,一份订单对每项产品只有 0..1 个行项。重数 1 指出,对产品的每一实例必须有一个行项。*号指出,对每项产品有多个行项,但对行项的访问则是以产品作为下标的。

在概念建模中,我利用限定符构造只是示明依照"订单上每项产品一个订单行"的设计思路的约束。

分类与泛化

我常听人把子类型构作称为"是一(is a)"关系。我敦促你要提防这种思考方式。问题是,"是一"这一短语可有不同的含义。

考察如下短语:

1. Shep 是一只牧羊犬。
2. 牧羊犬是一只犬。

3. 犬是一动物。

4. 牧羊犬是一属(Breed)。

5. 犬是一种(Species)。

现在试图把这些短语加以组合。如组合短语1和2，便得："Shep是一只犬"；组合2、3得"牧羊犬是一动物"；组合1、2、3得"Shep是一动物"。到目前为止，都很好。现在试1和4，便得"Shep是一属"。组合2、5，得"牧羊犬是一种"。这就不太好了。

为什么我可以组合其中某些短语，而不可以组合别的短语？理由是，有些是**分类**(classification)(对象Shep是牧羊犬类型的一个实例)，有些是**泛化**(generalization)(牧羊犬类型是犬类型的一个子类型)。泛化是传递的，分类则不然。我可以组合后接泛化的分类；但反之则不然。

我提出这一点是使你对"是一(is a)"谨慎对待。使用它可能引起子类构作不合适以及职责混淆。在这种情形，子类型构作更好的检验是如下一些短语："犬是动物的种类"和"牧羊犬的每一实例都是犬的一个实例"。

UML利用泛化符号示明泛化。如果你需要示明分类，可利用一个带基词《instantiate》的依赖。

多重分类与动态分类

分类指的是对象及其类型之间的关系。主流编程语言都假定，一个对象只属于一类。但是，对分类的考虑还有更多的选择。

在**单一分类**(single classification)中，一个对象属于单一一

类,它可以继承其超类。在**多重分类**(multiple classification)中,一个对象可以表述为若干类型,它们不一定都是用继承来联结沟通的。

多重分类和多重继承不同。多重继承说的是,一种类型可以有多个超类型,但对每个对象必须定义单独一个类型。多重分类允许一个对象有多种类型,并无须为其特定目的而定义特定类型。

例如,考察一个区分为子类型男人或女人、医生或护士、病人或非病人的人(person)(见图5.11)。多重分类允许一个对象具有依任何准许的组合指派给它的各种类型中的任何一种类型,无须对所有合法的组合都定义类型。

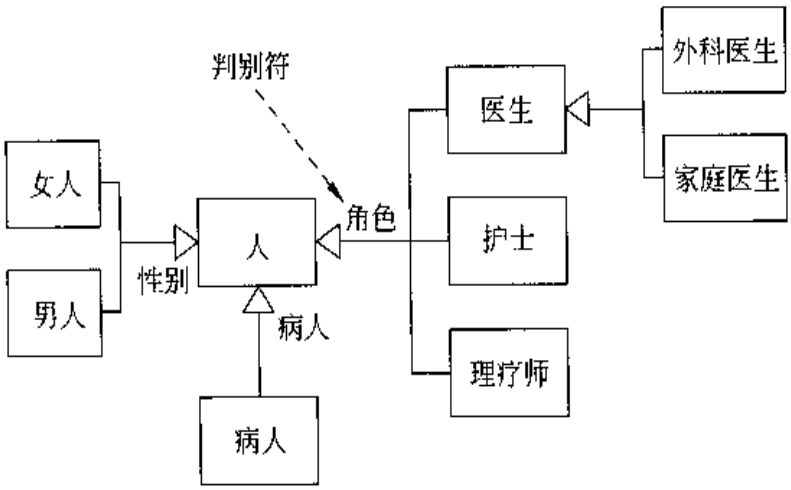


图5.11 多重分类

如果使用多重分类,就需确信,你已清楚哪些组合是合法组合。UML 2是通过把每一泛化关系置入**泛化集**(generalization set)中来做这件事的。在类图上,把泛化箭头标以泛化集的名,这在UML 1中称作判别符。单一分类对应于不具泛化集名的单一

泛化。

按照默认，各个泛化集互不相交：超类的任一实例只可以是那个泛化集中一个超类的实例。如果把泛化卷缩成一个单箭号，它们就必须全都是同一个泛化集的部分（如图 5.11 中所示）。要不然，也可以有多个带相同正文标号的箭号。

为阐明起见，请留意图中各个子类型的如下合法组合：（女人，病人，护士），（男人，理疗师）（女人，病人）以及（女人，医生，外科医生）。组合（病人，医生，护士）是非法的，原因是它包含了角色泛化集中的两种类型。

另一个问题是一个对象能否改动它的类。比如，当一个银行账户已透支，这实质上已改动了它的行为。特别是“提款”、“关闭”等操作都要撤销。

动态分类（dynamic classification）允许对象在子类型构作结构以内改动类；**静态分类**（static classification）则否。对静态分类，类型和状态要分开；动态分类则结合了这些概念。

你是否应该使用多重动态分类？我相信，它对概念建模来说是有用的。但是，对软件视面，它和实现之间的距离跃度太大，在大多数 UML 图中，你将要看到的只是单一静态分类，因此，单一静态分类应该是你的默认。

关联类

关联类（association class）使你能对关联添加属性、操作以及其他特征（如图 5.12 中所示）。从该图可以看出，一个人可参加

多个会议。我们需要保持关于该人警觉程度的信息;为此,可以对关联添上属性“专注”。

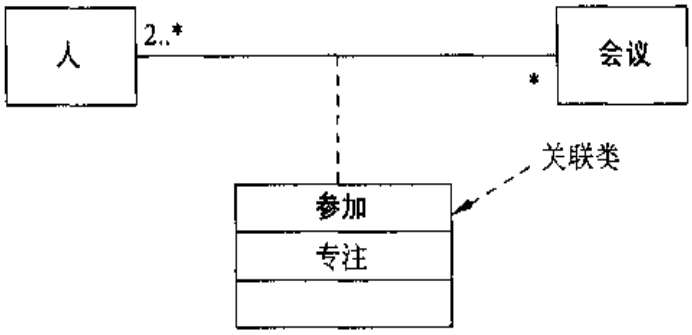


图 5.12 关联类

图 5.13 示明表示这一信息的另一方式,使参加独立地成为一个全类。请注意,重数是如何因之而移动的。

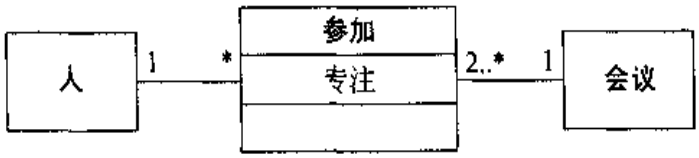


图 5.13 关联类的全类提升

你从关联类得到什么好处以补偿必须记着补充图示法?关联类添加了一个补充约束,即在任何两个参与对象之间只能有关联类的一个实例。我感到需要有一个例子。

看一下图 5.14 中的两个图。这两个图的形式很相似。但是,可以想象在同一个契约中担任不同角色的一家公司,但难以想象在同一技艺上有多种本领的人;的确,你也可能把它看作错误。

在 UML 中,只有后一种情形才算合法。对于人和技艺的每一组合只能有一种本领。图 5.14 中的上图不允许一家公司在一

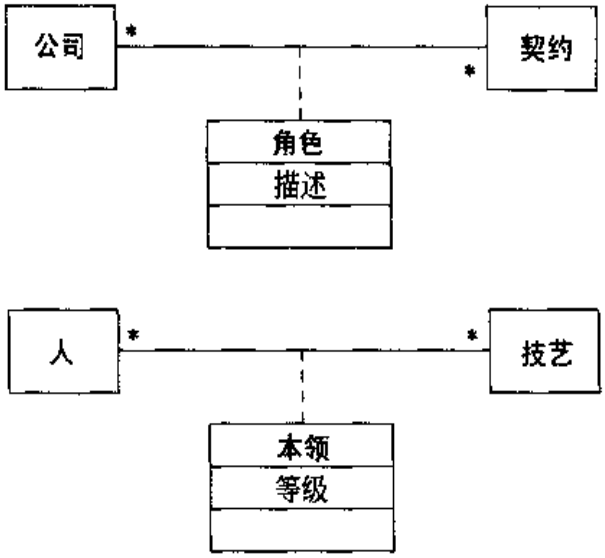


图 5.14　关联类的微妙(也许角色不应为关联类)

个契约上担任多个角色。如果你需要允许这一点，就必须依照图 5.13 的图风使角色成为一个全类。

实现关联类并非非常明显。我的建议是把关联类实现成犹如它在那里是一个全类，但要提供一些方法以便获得由关联类连接的各个类的信息。因此，对图 5.12，我会看到人(Person)上的如下方法：

```
class Person
    List getAttendances()
    List getMeetings()
```

循此途径，人(Person)的一名客户可以左右会议上的人；如果他们需要细节，便可抓住参加本身。如果你做这件事，记住要加强约束，使得对任何一对人和会议，只有一个参加对象。你应该核查是哪一个方法创建了参加。

你往往发现图 5.15 中那样一种带有历史信息的构造。但是,我发现,创建一个补充类或关联类会使模型难于理解,并且使实现形成依某一特定方向的往往不合适的偏离。

图 5.15 用于时态关系的类

如果我有这种时态信息,则在关联上利用基词《temporal》(见图 5.16)。该模型指出,一个人在某一时间只能为一家公司工作,但在一段时间,一个人却可以为几家公司工作。这便使人联想起依照下述代码的一个接口:

```
class Person...
    Company getEmployer();//get current employer
    Company getEmployer(Date);//get employer at a given date
    void changeEmployer(Company newEmployer,Date
    changeDate);
    void leave Employer(Date changeDate);
```

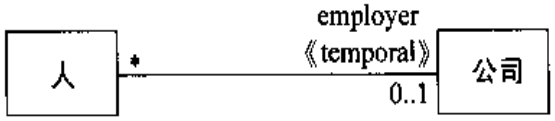


图 5.16 用于关联的基词《temporal》

《temporal》基词并非 UML 的部分,我在这里提它的理由有二:第一,它是在我的建模事业中若干场合发现有用的概念。第二,它示明你如何可用基词来扩张 UML。在 http://martin

fowler.com/ap2/timeNarrative.html 处，你可以读到关于这方面更多的资料。

模板(参数化)类

有些语言(最值得注意的是 C++)具有**参数化类**(parameterized class)或**谓模板**(template)。(在不远的将来，Jave 及 C# 中即将包含模板。)

这一概念对于强类型语言的组显得最为有用。循此，一般可以通过定义一个模板类 Set 对集合定义行为：

```
class Set <T> {
    void insert (T newElement);
    void remove (T anElement);
    ⋮
}
```

这样做了以后，就可以使用这一一般定义构作元素更为特定的 Set 类：

```
Set<Employee> employeeSet;
```

在 UML 中使用图 5.17 中所示的图示法来说明模板类。图中的 T 是一个类型参数占位符。(你可以有多个占位符。)

参数化类的一次使用(如 Set <Employee>)称为一个**导出**(derivation)。可以按两种方式来示明导出。第一种方式反映 C++ 的语法(见图 5.18)。在尖括号中表述导出表达式，其形式为

〈参数名::参数值〉。如果只有一个参数,通常用法往往省略参数名。另一种图示法(见图 5.19)加强了和模板的连接并使你能对束元重新命名。

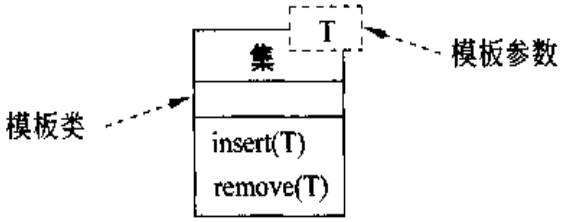


图 5.17 模板类

图 5.18 束元(形式 1)

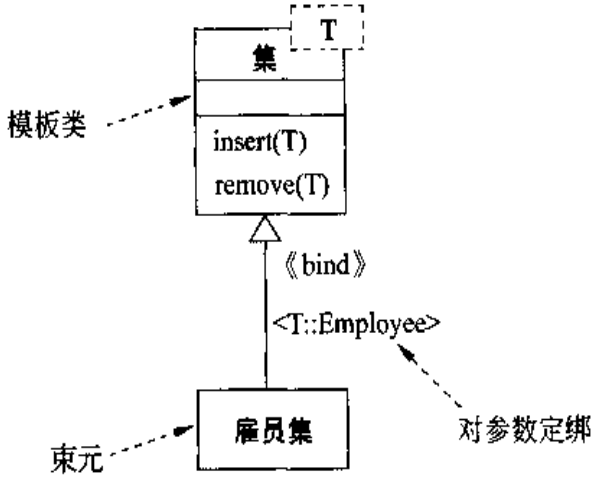


图 5.19 束元(形式 2)

《bind》基词是一个精化关系上的衍型(stereotype)。这种关系指出,EmployeeSet 将和 Set 的接口相符。你可以把 EmployeeSet 想象成 Set 的一个子类型。这和实现类型特定的组的其他方式相适应,这个组是用来说明所有合适子类型的。

但是,使用导出和子类型构作不同。不允许对束元增添特征,束元完全是由模板指明的;你只能对它增添限制类型的信息。

如果要增添特征，就必须创建一个子类型。

枚举

枚举(enumeration)(图 5.20)用来示明固定的一组值，这些值除符号值外，别无任何特性。

图 5.20　枚举

它们是用带有基词《enumeration》的类来示明的。

主动类

主动类(active class)所具有的各个实例执行并控制其自身的控制线程。方法调用可在客户线程或主动对象的线程中执行。主动类的一个好的例子是命令处理程序，它从外面接收命令对象，然后在自身的控制线程内执行命令。

主动类的图示法从 UML 1 到 UML 2 已有改变(如图 5.21 所示)。在 UML 2 中主动类在两边具有附加的垂直线；在 UML 1 中，它具有粗边界，并称作主动对象。

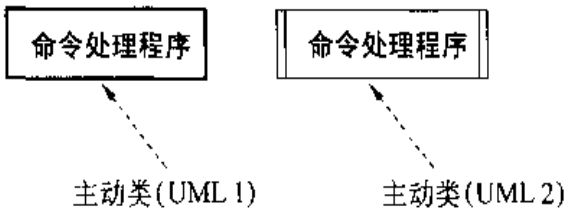


图 5.21 主动类

可见性

可见性(visibility)是一个这样的论题,它原则上简单,但却有复杂的微妙。简单的概念是,任何类皆有公用成分与私用成分。公用成分可由任何别的类使用;私用成分只能由拥有它的类使用。但是,每种语言都有自己的规则。虽然很多语言都使用"公用"、"私用"、"受护"等术语,但语言不同,含义各异。它们区别不大,但却会引起混淆。特别对于我们这样使用多种语言的人更是如此。

UML 试图把这一问题处理得不致陷人骇人的纠纷。本质上说,在 UML 内,可以对任一属性或操作标以一个可见性指示符。也可以使用你喜欢的任何标记符,并且其含义和语言有关。可是,UML 提供四种可见性缩写:+(公用)、-(私用)、~(包)、#(受护)。这四个级别的可见性用于 UML 元模型内并在其中定义,但它们的定义和一些别的语言略有不同。

当使用可见性时,利用正在赖以工作的语言中的规则。当在别处看到 UML 模型时,要留神可见性标记符的含义,并意识到那些含义如何随语言而变。

大多数时间我不在图中画出可见性标记符；仅当我需要突出某些特征的可见性差别时，才使用它们。即使那时，我多半会选择＋和－，它们至少是容易记住的。

消息

标准 UML 在类图上并不示明任何关于消息调用的信息。但是，有时我已在像图 5.22 那样常见的图中看到。这些图在关联两旁添加有箭号。这些箭号都标以一对象发送给另一对象的消息。因为不需要对它发送消息的类的关联，也可以加上一个依赖箭号以示明各不关联的各类之间的消息。

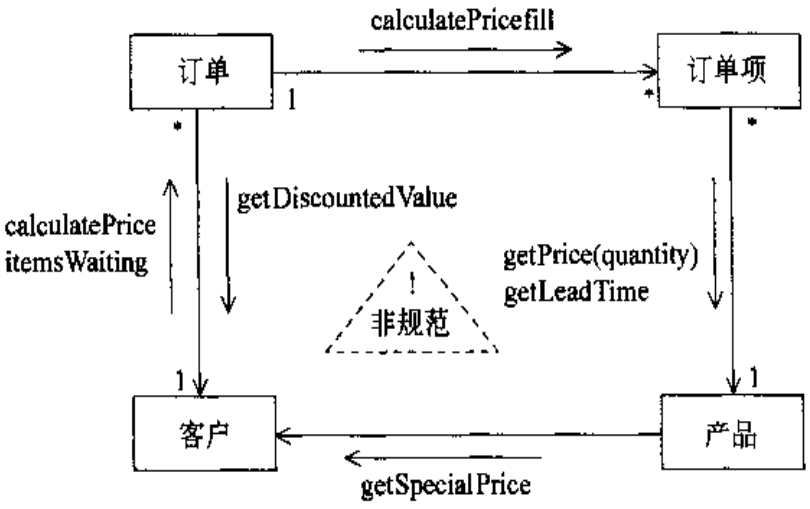


图 5.22　带消息的类

这种消息信息跨越多个用案，因此，与通信图不同，为了示明顺序图，就不包括它们了。

第 6 章

对 象 图

对象图(object diagram)是在一个时间点上系统中各个对象的一个快照。由于对象图示明的是实例而不是类,所以它有时亦称实例图。

你可以使用一个实例图去示明各个对象的一个样例构形(见示明一组类的图 6.1 以及示明一组相关对象的图 6.2)。在对象间可能的连接比较复杂时,实例图很有用。

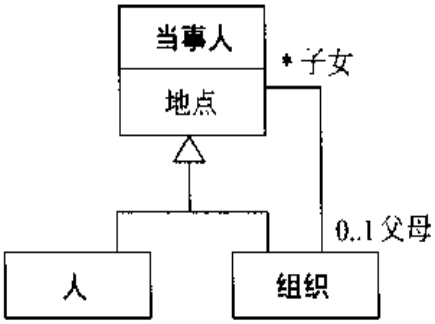


图 6.1 当事人类组合结构的类图

可以说图 6.2 中的成分是实例,这是因为名是带下划线的。

每个名的形式是实例名:类名。名的两部分都是任选的。因此，John、:Person 和 aPerson 都是合法名。如果只用类名，就必须包括分号。如图 6.2 中所示，可以示明属性和连接的值。

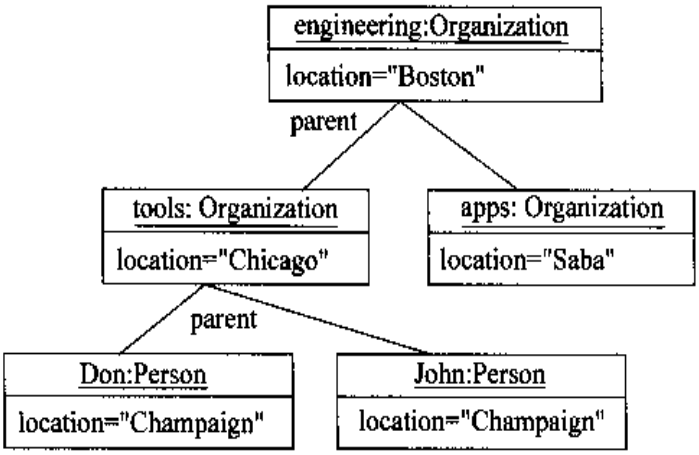


图 6.2　当事人类样例实例的对象图

严格说来，对象图的成分是实例规约而不是真正的实例。理由是，让强制属性阙如或者示明抽象类的实例规约都是合法的。可以把实例规约(instance specification)看成部分定义的实例。

另一种看待对象图的方式是作为不带消息的通信图(第 160 页)。

何时使用对象图

对象图对于示明连接在一起的对象的例子是有用的。在很多场合，可以使用类图精确定义一个结构，但这个结构仍然难以理解。在这些场合，几个对象图例就可以使全局改观。

第7章

包　图

类表示构组面向对象系统的基本形式。虽然类是出奇地有用，但为了构组拥有几百个类的大型系统，还需要有别的构造。

包(package)是一种这样的聚组构造，使你能取 UML 中的任一构造，将其成分聚组在一起，以构成更高层的单位。其最通常的用法是聚组类，而这就是我在这里表述的方式，但要记住，你也可以将包用于 UML 的任何别的小块。

在一个 UML 模型中，每一类是单一一个包的成员。包也可以是别的包的成员，因此，这就给你留下一个分层结构，其中高层的包分成若干子包，对子包依此进行，直到全是类的底层为止。包可以既包含子包又包含类。

按照编程术语，包对应于 Java 中的包和 C++，.NET 中的名空间这样一些聚组构造。

每个包表示一个**名空间**(namespace)，这指的是，每个类在拥有它的包中必须有一个惟一的名。如果我要创建一个称为日期(Date)的类，而在系统(System)包中已有了一个日期类，只要把

它放入另一个包中，就可以有我自己的日期类了。为清晰起见，我可以利用一个**全受限名**(fully qualified name)，亦即，示明拥有它的包结构的名。在 UML 中用双冒号示明包名，于是，日期可以是 System::Date 与 MartinFowler::Util::Date。

如同图 7.1 那样，图中的包是用凸耳小框来示明的。你可以只示明包名或者也示明内容。在任何地方，都可以使用全受限名或正规名。用类图符示明内容，使你能示明类的所有细节，甚至示明包内的一个类图。只列出包名在下述情况也是有意义的，即当你所需要做的全部事情只是指明哪些类在哪些包中。

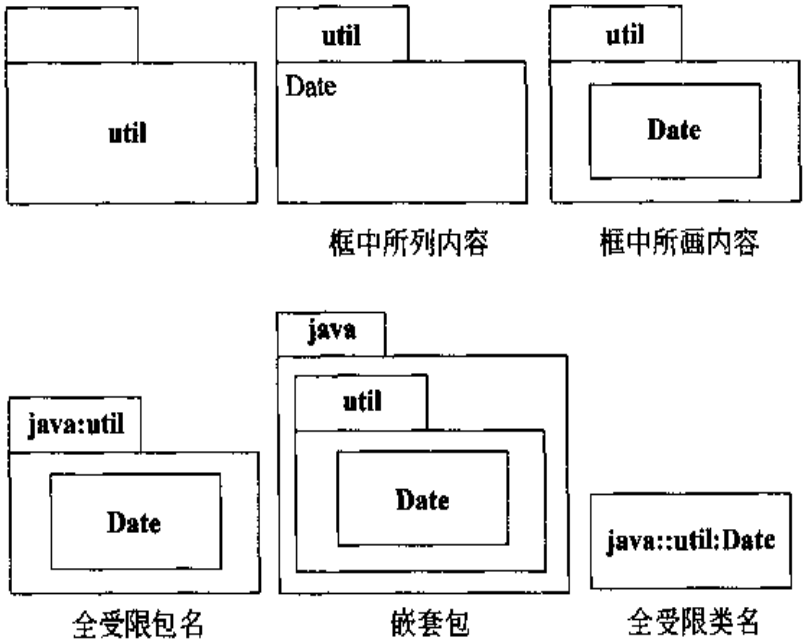


图 7.1 包在图中的示明方式

经常看到的是一个标以如同 Date(来自 java. util)那样信息的类而不是它的全受限形式。这是 Rational Rose 公司用得很多的一种图风，它并非标准中的部分。

UML 允许包中的类为公用类或私用类。公用类是包接口的组成部分,可为别的包中的类使用;私用类则是隐蔽的。关于成包构造之间的可见性规则随编程环境而异;你应该遵从所在编程环境的习惯,即使它的含义与 UML 的规则有所偏离。

这里一项有用的技术是,通过只移出与包的公用类有关的操作的一个小的子集来缩小包的接口。为此,可以通过对所有的类都给予私用可见性(于是,只有从同一个包中别的类才能看见它)以及对公用行为添上额外的公用类。这些额外的类称为**虚包类**(Facades)[Gang of Four]。这时把公用操作委派给包中更为隐蔽的伙伴。

如何选择哪些类放入哪些包中呢?这实际上是一个非常难以处理的问题,它需要高明的设计技艺来回答。两种有用的原则是共同封闭原则(Common Closure Principle)与共同复用原则(Common Reuse Principle)[Martin]。共同封闭原则说的是,一个包中的各个类应该是由于类似的原因而改变。共同复用原则说的是,一个包中的各个类应该一起被复用。聚组各类入包的很多理由都和包间的依赖有关。这个问题将在下面讨论。

包与依赖

包图(package diagram)示明包及其依赖。我在第 60 页上引进了依赖概念。如果你有用于表象的包和用于领域的包,当表象包中任一类对领域包中的任一类均有一依赖时,就有了一个从表象包到领域包的依赖。这样,包间的依赖就概括了其内容之间

的依赖。

UML有很多种依赖，每一种都有特定的语义和衍型。我发现，比较容易的是，从不带衍型的依赖开始，仅当需要时才使用更为特别的依赖，这一点我几乎未曾做过。

在大型系统的介质上绘制包图可能是一件使你得以控制系统大型结构的最重要的事。理想的是，这个图应根据代码库本身生成，以便能看出系统中在那里真正有些什么。

高明的包结构有清晰的依赖流程，这是一个难以定义但往往又是容易认识的概念。图7.2示明一个企业应用的样例包图，它是一个结构良好且具有清晰流程的包图。

你常常可以识认一个清晰流程，这是因为所有的依赖都按单一方向走向的缘故。虽然这是结构良好系统的一个好的标志，图7.2中的数据保管员包却示出一个经验性规则的例外。数据保管员包用作领域包和数据库包之间的一个隔离层，它是保管员模式[Fowler,P of EAA]的一个例子。

很多作者宣称，依赖中不应有循环（无循环依赖原则[Martin]）。我并不把它看作一条绝对规则，但却认为应使循环局部化，特别是，不应有跨层的循环。

包中依赖越多，包接口就越需要稳定，原因是接口中的任一变动将波及到对它依赖的所有的包（稳定依赖原则（Stable Dependencies Principle)[Martin]）。于是，在图7.2中，资产领域包比租赁数据保管员包更需要稳定。你会发现，更稳定的包往往会有更高比例的接口和抽象类（稳定抽象原则（Stable Abstraction Principle)[Martin]）。

依赖关系不是传递的（第61页）。为了看出这件事对依赖重

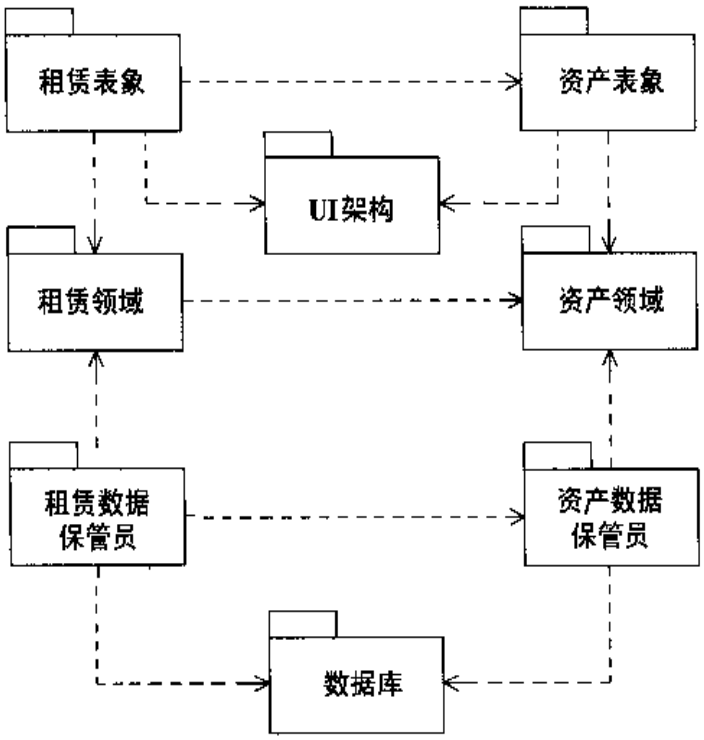


图 7.2 企业应用包图

要,再看一下图 7.2。如果资产领域包中有一类变动,在租赁领域包中的类就会有变动。但是,这一变动并不必波及到租赁表象(仅当租赁领域变动其接口时才会波及)。

有些包要在很多地方使用,因而要对它绘制出所有的依赖就会出现混乱。在这种情形,习惯是在包上使用《global》这样的基词。

UML 包也定义了一些构造,它们允许由其中一个包向另一个包移入类或其中一个包中的类与另一个包中的类进行合并,而利用带有表述这件事的基词的依赖。但是,关于这类事情的规则在很大程度上随编程语言而变。总的说来,我发现,依赖的一般概念在实践中极为有用。

包面

如果你考察图 7.2，便会认识到，这个图有两种结构。一种是应用中的层次结构：表象领域、数据保管员以及数据库。另一种是主题域结构：租赁与资产。

通过像图 7.3 那样把这两面分开，可能会使之更为明朗。对这个图，你可以清晰地看出每一面。但是，这两面都不是真正的包，原因是，你不能对单一一个包指派类（必须从每一面挑选一个）。这一问题反映出编程语言中层次名空间的问题。虽然像图 7.3 中那样的图不是标准的 UML 图，它们往往在阐明复杂应用的结构中非常有用。

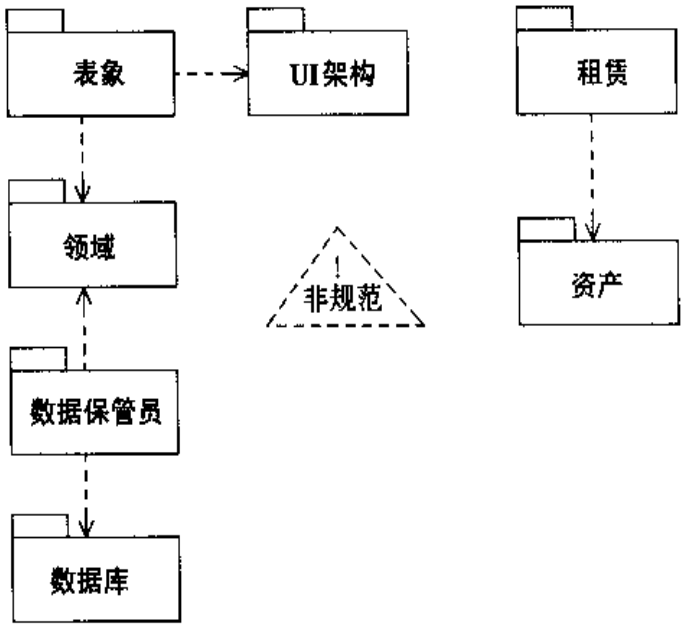


图 7.3 分隔图 7.2 为两面

包的实现

你往往会看到这样的情形,一个包定义了一个可由若干别的包来实现的接口,就如同图 7.4 那样。在这种情形,实施关系指出,数据库通道包定义了一个接口,而别的通道包提供了实现。在实践中,这就表示,数据库通道包包含了这样的接口和抽象类,它们完全是由别的包来实现的。

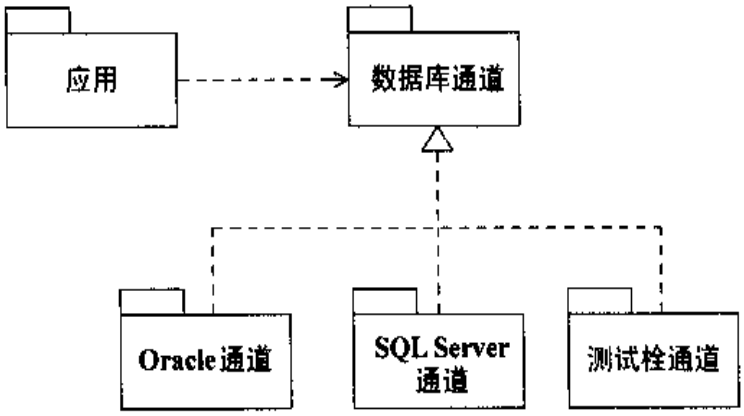


图 7.4 由别的包实现的包

很常见的是,接口及其实现分别在不同的包中进行。的确,客户包往往包含一个由别的包去实现的接口:这是和我在第 89 页上讨论的所需接口相同的概念。

设想我们要提供某一用户接口(UI)控制的开关机制。我们想要它对很多不同的东西工作,例如,加热器和光。UI 控制需要调用加热器上的方法,但不希望这些控制对加热器有依赖关系。

采用下述方法就可以避免这种依赖，即在控制包中定义一个接口，该接口随后是由希望与这些控制工作的任何一个类来实现的（如图 7.5 所示）。这是隔开接口（Seperated Interface）模式[Fowler,P of EAA]的一个例子。

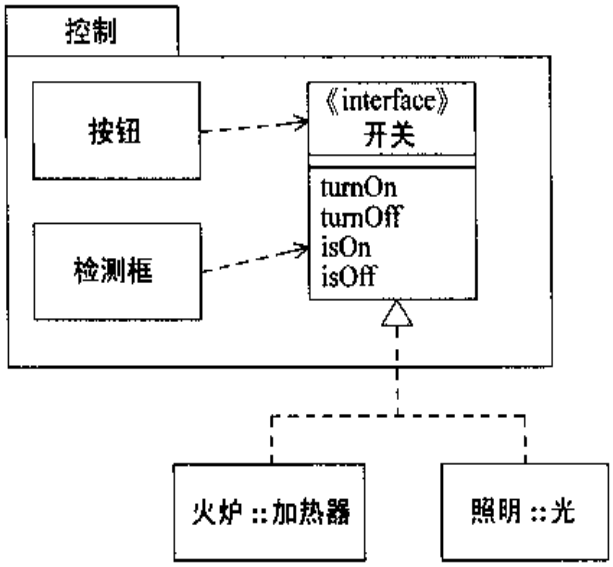


图 7.5 在客户包中定义所需接口

何时使用包图

我发现，对大型系统要了解系统主要成分之间的依赖时，包图极为有用。这些图很好地对应于通常的编程结构。绘制包和依赖的图有助于你保持对应用依赖的控制。

包图表示一种编译时刻的聚组机制。关于示明在运行时刻如何来组合各个对象，请使用复合结构图（第 164 页）。

何处找寻更多资料

据我所知,关于包及其使用的最佳讨论是[Martin]。Robert Martin 长期以来对依赖几乎是病态般的着迷,并撰写了一些关于"如何留神依赖,因而使你能对之控制并使之极少出现"的很不错的资料。

第8章

部 署 图

部署图(deployment diagram)通过揭示"哪些软件片段运行于哪些硬件片段上"来示明系统的一个物理布局。部署图实在很简单,因此,本章是一个短章。

图8.1是一个部署图的简例。图上的主要项是用通信路径连接起来的各个结点。**结点**(node)是可作为软件宿主的东西。结点可依两种形式出现。**设备**(device)是硬件,它可以是一台计算机或者是与系统相连接的一个较为简单的硬件。**执行环境**(execution environment)是软件,它本身可以是其他软件的宿主或者包含其他软件,操作系统或集装箱进程都是例子。

结点含有一些**制品**(artifact),它是软件的物理体现,通常是文件。这些文件可以是可执行的(如.exe文件、位串、DLL、JAR文件、汇编语言程序或脚本),或数据文件、配置文件、HTML文档,等等。在一结点内列出制品示明在运行系统中该制品就部署在该结点处。

你既可以将制品示明为类框又可以在结点中列出其名来示

明制品。如果将制品示明为类框,则可以添加一个文档图符或基词《artifact》。你可以用标记值来标记结点或制品,以指出关于结点的各种有意义的信息,如厂商、操作系统、位置或任何其他你所喜爱的信息。

往往会有施行相同逻辑任务的多个物理结点。你或者可用多重结点框或者把物理结点数描述成一个标记值来示明这一点。

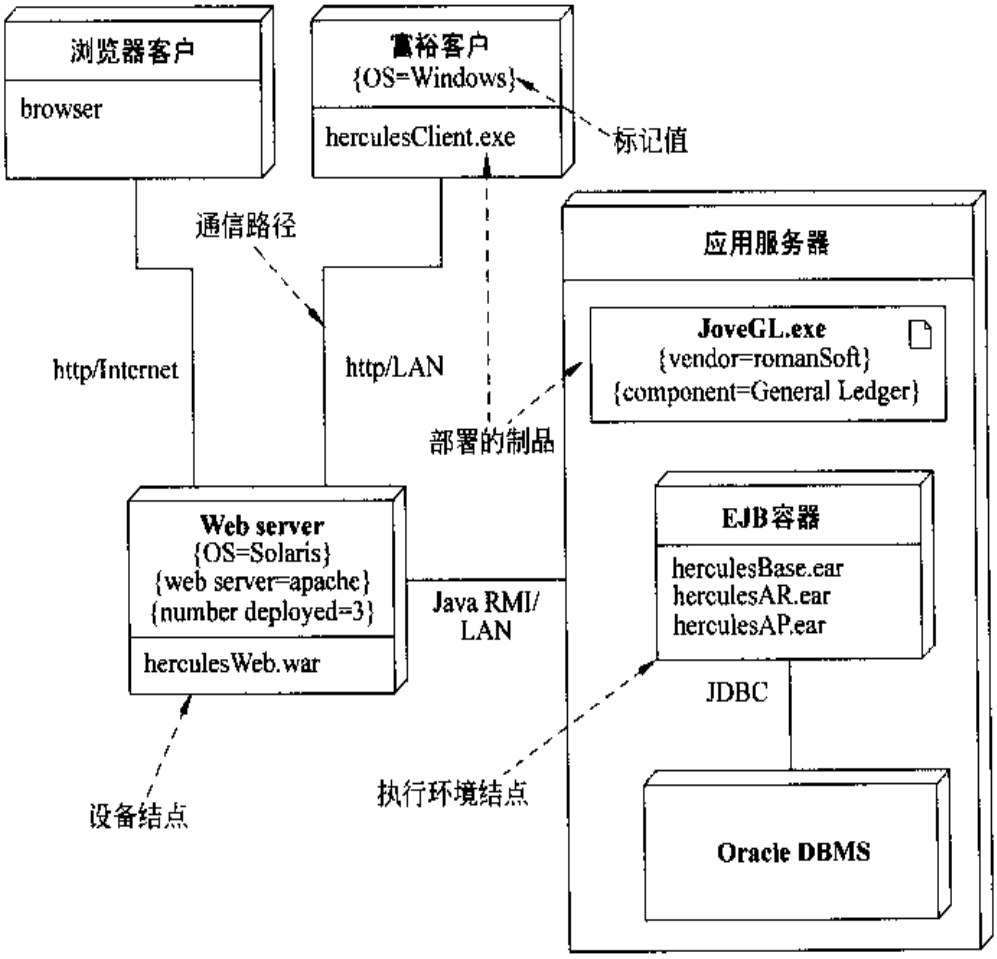


图 8.1 部署图例

在图 8.1 中我使用部署标记数来指出三个物理 Web 服务器，但对此并无标准标记。

制品往往是一个构件的实现。为了示明这一点，可以在制品框中使用一个标记值。

结点间的通信路径指出各部分是如何通信的。你可以把这些路径标以关于你使用的通信协议的信息。

何时使用部署图

不要因本章简短而使你认为不应使用部署图。部署图对示明“何者部署于何处”是很有用的，因此，任何复杂的部署都可以很好地使用部署图。

第9章

用　案

用案是获取系统功能需求的一种技术。用案通过表述系统的用户和系统本身之间特有的交互而工作，提供了如何使用系统的一种陈述。

我不正面来表述用案，而发现比较容易的是悄悄地从背后去接近用案，而且从表述案况开始。一个**案况**(scenario)是一系列表述用户和系统之间一次交互的步骤。这样，如果有一个基于网络的联机存储，就可以有一个"购物"案况，它可以说成：

> 客户浏览了目录，并向购物篮子里添加一些要买的物品。当客户希望付款时，他给出送货和信用卡信息，并对这一销售予以确认。系统核查信用卡许可并立即对销售确认，紧接着发一份电子邮件。

这一案况是可能出现的一种情况。但是，核查信用卡许可可能失败，这将是一种不同的案况。在另一种情形，你可能有一位常客，无须要求他提供送货和信用卡信息，而这是第三个案况。

所有这些案况各不相同，但却相似。其相似性的本质是，在

所有这三个案况中，用户有相同的目标：购物。用户并不见得总是成功的，但目标却保持不变。这一个用户目标乃是用案的关键。一个**用案**(use case)是由一个共同的用户目标联系在一起的一组案况。

按照用案的说法，用户指称为参与者。**参与者**(actor)是用户相对系统而言所扮演的角色。参与者中可包含客户、客户服务代表、销售经理以及产品分析人员。参与者施行用案。一名参与者可以施行很多用案，反之，一个用案可以有多名施行它的参与者。通常，你有很多客户，于是很多人都可以是客户参与者。此外，一个人可以作为多名参与者，例如一名做销售服务代表工作的销售经理即是如此。参与者不必是人。如果系统对另一个计算机系统履行一种服务，那另一个系统就是一名参与者。

参与者(actor)实在不是一个准确术语。角色(role)要好得多。显然，这里有一个从瑞典语的误译，可是，参与者却是用案界使用的术语。

用案是UML的一个重要部分已众所周知。但是，令人吃惊的是，在很多方面，UML中用案的定义却颇少。关于应该如何获取用案的内容，UML中竟毫无表述。UML表述的是用案图，而用案图只示明各个用案如何彼此相关。但是，用案的几乎全部价值都在于内容，图的价值并不大。

用案的内容

没有书写用案内容的标准方式。不同的格式工作于不同的

场合。图 9.1 示出一种常见的使用样式。你开始从各个案况中选出一个案况作为**主成功案况**(main success scenario)。用案体从写一列带编号的步骤的主成功案况开头,随后,就取别的案况,把它们写成**扩张**(extension),用主成功案况的变形来表述扩张。扩张可以是成功(用户实现了目标)(如 3a 中),也可以是失败(如 6a 中)。

购物

目标级:海级

主成功案况(MSS):

1. 客户浏览目录并选择要买的东西
2. 客户去结账
3. 客户填写送货信息(地址,第二天或第三天发货)
4. 系统提出包括送货在内的全部价格信息
5. 客户填写信用卡信息
6. 系统核定购买许可
7. 系统立即确认销售
8. 系统向客户发确认电子邮件

扩张:

3a: 客户是常客

    1. 系统显示当前的送货信息、价格信息以及票据信息
    2. 用户可以接受或取消这些默认,返回到 MSS 的第 6 步

6a: 系统不准许信用卡购物

    1. 客户可以再输入信用卡信息或取消

图 9.1 用案一例的正文

每一用案有一名主参与者，它请求系统提供服务。主参与者是具有用案试图满足的目标的参与者，通常是(并非总是)用案的发起人。在施行用案时，可以有别的参与者也和系统通信，这些参与者称为次参与者(secondary actor)。

用案中的每一步骤应是参与者和系统之间交互的一个元素。每一步骤应是一个简单陈述并应清楚地示明谁在施行这一步骤。步骤应示明参与者的意图，而不是参与者所进行的技术内容。因此，在用案中并未表述用户接口。的确，书写用案通常是在设计用户接口之前进行的。

用案内部的扩张对条件进行命名，该条件是与主成功案况(MSS)中所表述之交互不同的交互，并说明其不同点。扩张的开始处要对检查条件的步骤进行命名并提供条件简短表述，在条件之后，是一些带编号的步骤，其样式与主成功案况中的相同。通过表述在何处返回到(如果返回的话)主成功案况来结束这些步骤。

用案结构(拟订)是一种讨论提供主成功案况以外的供选情况的惯用手段。对每一步骤，问，这一步骤如何能另法进行？特别是，什么可能走错？通常是在陷于构想后果以前，先想出所有的扩张。这样，你就可能会想到更多的状况，而将之转移给少许蠢人，以后又必须拣回。

用案中一个复杂的步骤可能是另一用案。按照 UML 术语，我们说，第一个用案**包含**(includes)第二个用案。如何示明正文中所包含的用案并无标准方法，但是，我发现，使我们联想到超文本链接的下划线工作得很漂亮，并且在很多工具中其实就是一个超文本链接。因此，在图 9.1 中，第一步包含用案“浏览目录并选

择要买的东西”。

所包含的用案对于会弄乱主案况的复杂步骤以及在若干用案中重复出现的步骤,可能有用。但是,不要使用功能分解把用案分解成一些子用案、子子用案,这样的分解除了浪费很多时间外,别无任何意义。

除了案况中的各个步骤外,可以对用案添加一些别的常用信息:

- **前置条件**(pre-condition) 表述在系统允许用案开始以前,系统应确保为真的条件。对于告知编程人员在代码中哪些条件无须核验一事,前置条件是有用的。
- **保证**(guarantee) 表述在用案结束时,系统将要保证的事。成功的保证在成功案况后成立;极少的保证在任何案况后均成立。
- **引发装置**(trigger) 指明启动用案的事件。

当你考虑添加元素时,要持怀疑态度。做得过少比做得过多更好。此外,要竭力使用案简明易读。我已发现,详细的长用案不仅难读,甚至会使目的受挫。

在用案中你需要的细节量取决于该用案中的风险量。常常,初期仅对少量关键元素需要细节;其他则可以仅在实现之前再增添。无须写下所有的细节,口头通信往往很有效,特别是在一个需要迅速看到运行代码的迭代循环中更是如此。

用案图

像我在前面说过的,UML 关于用案的内容未曾涉及,而提供

了一种示明其内容的图格式(如图 9.2)。虽然这种图有时有用，它却不是强制性的。在你的用案工作中，不必过多地致力于图，而要全神贯注于用案的正文内容。

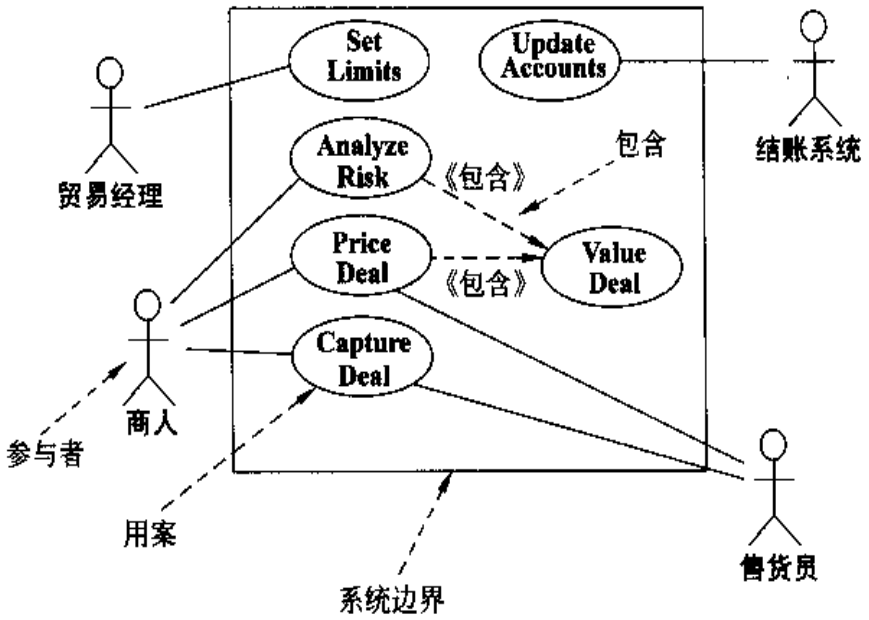


图 9.2 用案图

构想用案图的最佳方式是一个关于用案组的内容图表。它也和关于结构化方法的语境图类似，原因是，它示明系统边界以及与外界的交互。用案图示明参与者、用案以及其间的关系：

- 什么参与者施行什么用案
- 什么用案包含别的用案

除了简单的包含关系以外，UML 还包括了用案间的一些别的关系，如《extend》。我坚决建议，你略去它们。我已经看到很多场合，其中项目组在使用别的用案关系中大大误时受挫，因而，致力于使用别的用案关系简直是一种浪费。而要全神贯注于用案的正文表述，那才是这种技术的真正价值所在。

用案级别

针对用案的一个共同问题是,由于着重考虑用户和系统之间的交互,而可能忽略如下情况,即改动业务过程可能是处理问题的最佳方法。你往往听到人们谈论系统用案与业务用案。这些术语并不准确,但一般用法是,**系统用案**(system use case)是与软件的交互,而**业务用案**(business use case)讨论的是一种业务如何响应客户或事件。

[Cockburn, use cases]中提出一种用案级别框架。核心用案处于"海级"。**海级**(sea-level)用案典型地表示主参与者与系统之间的一次离散交互。这样的用案把一件事交给主参与者去做,并且往往只需两分钟到半小时即可完成。包含在海级用案之内的用案是**鱼级**(fish level)用案。更高的是**风筝级**(kite-level)用案。它示明海级用案如何适应更广的业务交互。风筝级用案通常是业务用案,海级与鱼级用案则是系统用案。你的大多数用案应是海级用案。我喜欢在用案顶端指出级别(如图 9.1 所示)。

用案与特征(或情节)

很多方法使用系统的特征(极端程序设计称特征为用户情节)以助于表述需求。通常的问题是,特征与用案如何相互联系。

特征是系统赖以分块对迭代项目制订计划的好办法,其中每

次迭代交付一些特征。用案提供参与者如何利用系统的描述。因此，尽管这两种技术都表示需求，其目的则异。

虽然你可以直接表述特征，很多人发现先开发用案，再生成特征表，是有益的。特征可以是整个一个用案、用案中的一个案况、用案中的一个步骤，或者一个变形行为，例如把另一个已经贬值的方法添加到你的财产计值中，后者在用案描述中是未曾示明的。通常，特征比用案的粒度更细。

何时使用用案

用案是帮助了解系统功能需求的一项重要工具。早先应构作第一遍用案。更详细的用案版本应该仅在开发那个用案之前产生。

重要的是要记住，用案表示系统外观。据此，不要指望在用案与系统内部的各个类之间有任何联系。

我见到的用案越多，似乎用案图的价值就越小。对用案，要全神贯注的是正文而不是图。纵然 UML 对用案正文未有任何论述，包含技术中全部价值的依然是正文。

用案的一大危险是，人们把它做得太复杂，以致不知所措。通常，做得过少比做得过多危害要小。对多种情况，两三页的用案正好。如果做得过少，至少还有一个短小易读的文档，后者乃是发问的起点。如果做得过多，任何人对它将难以阅读，难以理解。

何处找寻更多资料

用案原来是由 Ivar Jacobson 在[Jacobson, OOSE]中普及推广的。

虽然用案已经存在了一段时间,但对它却鲜有标准。UML 未曾谈及用案的重要内容,而仅对其远非重要的图进行了标准化。因此,你可以发现关于用案的各种不同意见。

但是,过去几年,[Cockburn,use cases]一书已成为这一论题的标准读物。在本章中,我已经遵循了该书的术语和指点。极好的原因是,在过去我们发生争执时,通常我总是最终以同意 Alistair Cockburn 的意见而告结束。他也维护着网站 http://usecases.org。[Constantine and Lockwood]关于从用案推出用户接口提供一个令人信服的过程;参见 http://foruse.com。

第 10 章

状态机图

状态机图(state machine diagram)是表述系统行为的一种熟悉的技术。自从 20 世纪 60 年代以及最早的面向对象技术采用状态机图表述行为以来,各种形式的状态(机)图已经盛行一时。在面向对象方法中,对单一一类画一个状态机图以示明单个对象的生命期行为。

人们谈到状态机时,巡航控制器和售货机都是不可或缺的例子。我对它们有点厌烦,决定通过一座哥特式城堡中的秘柜控制器作为例子。在这座城堡中,贵重物品保存在一个隐秘的保险箱中。而要找到保险箱的锁,必须从烛台上移开伪装的蜡烛,但是,只有关上房门时,锁才会被发现。一旦看到锁,就能插入钥匙,以打开保险箱。为了额外保险起见,我要确保仅当我先把这一蜡烛归还原处,才能打开保险箱。如果一名小偷忽略了这一防备,我就会放出一个令人恐惧的怪物来吞食他。

图 10.1 示明一个控制器类的状态机图。这一控制器类指挥我的独特的安全系统。状态图以该控制器对象创建时的状态开

始,在图 10.1 中为等待(Wait)状态。该图将它指作**初始伪态**(initial pseudostate),初始伪态并不是一个状态,但有一个指向初态的箭号。

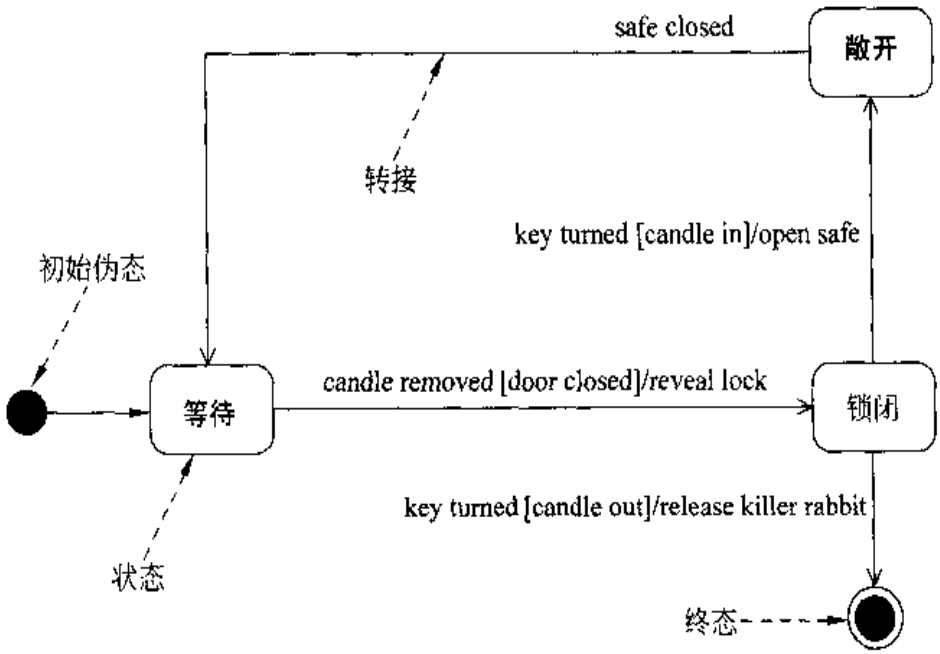


图 10.1 简单状态机图

这个图示明,控制器可以有三种状态,即等待,锁闭,打开。该图也给出了控制器赖以改变状态的规则。这些规则以转接的形式出现,它们是连接各个状态的线段。

转接(transition)指出由一种状态到另一种状态的运动。每一转接有一个由三个部分组成的标记:引发装置识别标志[监护]/活动。所有这三部分都是任选的。引发装置识别标志通常是一个单一事件,它引发了状态的可能改变。监护(如果有的话)为一布尔条件,在出现转接时,这一布尔条件必须为真。活动是在转接中执行的行为,它可以是任何一个行为表达式。引发装置

识别标志的全形可以包含多个事件和参数。因此，在图 10.1 中，你读到从等待状态向外的转接为：在等待状态时，在门是敞开的情况下移开蜡烛，你看到锁，并转为锁闭状态。

转接的所有三个部分都是任选的。活动阙如表示在转接中你什么事都没有做。监护阙如表示，如果事件发生，你总要进行转接。引发装置识别标志阙如非常罕见，但也会出现。它表示立即进行转接。这种情况多见于活动状态，关于这一点，我马上就要谈到。

当在某一状态下发生一事件，你只能从它转接一次。因此，如果你以同一事件进行多次转接，如在图 10.1 的锁闭状态那样，监护就必须互斥。如果事件发生，而没有转接生效，例如，等待状态下的保险箱锁闭事件或者门敞开时的蜡烛移开事件，便略去这一事件。

终态表示状态机完成，即表示控制器对象的删除。因此，如果万一有人不小心而陷入我的陷阱，控制器对象终止，于是我就需要把兔子关入笼内，擦擦地板，重新启动系统。

请记住，状态机只能示明对象直接观察或激活什么。于是，虽然你估计我在保险箱敞开时会往保险箱中放东西，或者从保险箱中拿东西，我却不能把这些行为放到状态图上，原因是，控制器不能辨别它们。

当开发人员谈到对象时，他们往往把对象的状态指为对象各个域中所有数据的组合。但是，状态机图中的状态却是更为抽象的状态概念。本质上说，不同的状态意味着对事件不同的反应方式。

内部活动

状态可以无须转接而利用**内部活动**(internal activity)(往状态框内置事件、监护、活动)来对事件做出反应。

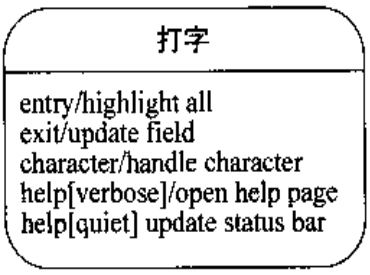


图 10.2 以一正文域的打字状态示明的内部事件

图 10.2 示明一个带有字符和帮助事件的内部活动的状态,正如你可以在一个 UI 正文域上所找到的。内部活动与**自转接**(self-transition)类似。自转接是循环回到相同状态的转接。内部活动的语法遵循事件、监护、过程等相同的基理。

图 10.2 还示明两个特别活动,即进入活动与退出活动。**进入活动**(entry activity)是在你进入一状态时所执行的活动;**退出活动**(exit activity)是在你离开状态时所执行的活动。但是,内部活动并不能引发进入活动与退出活动,这就是内部活动与自转接二者的区别。

活动状态

在我迄今所表述的状态下,对象在进行工作以前,一直是在安静地等待下一事件。但是,你可以有一些状态,在这些状态下,

对象正在做一项进行中(不可间断的)的工作。

图 10.3 中的搜索状态就是这样一个**活动状态**(activity state)。进行中的活动标以 do/;因此,又称之为**进行活动**(do-activity)。一旦搜索完成,就施行无活动的转接(如显示新硬件的转接)。如果在活动中发生删除事件,进行活动则非正式停止,又回到更新硬件窗状态。

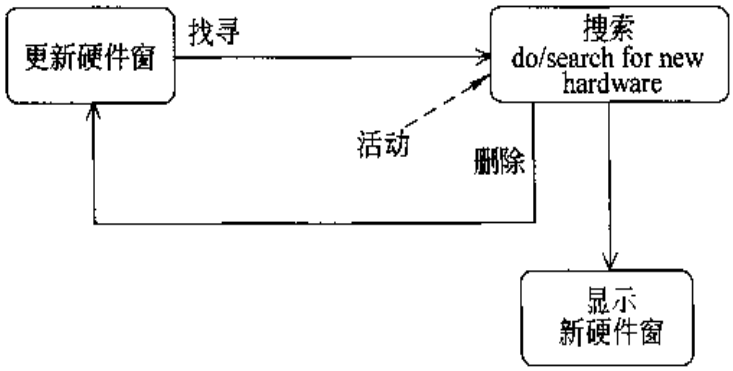


图 10.3　带活动的状态

进行活动与正常活动都表示施行某一行为。二者的主要区别是,正常活动"瞬时"出现并且不能由正常事件中断,而进行活动可以耗费有限时段,并且可以被中断(如图 10.3 中的情形)。瞬时的含义随系统而异。对硬件实时系统,可以是少许机器指令执行时间;但对桌面软件,则可以是数秒钟。

UML 1 对正常活动使用**动作**(action)一词,而活动只用于进行活动。

超态

往往你会发现,若干状态共享共同的转接与内部活动。在这些情形,你可以使它们呈一些子态,并将所共享的行为换成超态(如图 10.4 所示)。如果没有超态,你就必须在登录连接细节状态内对所有这三个状态都画一个删除转接。

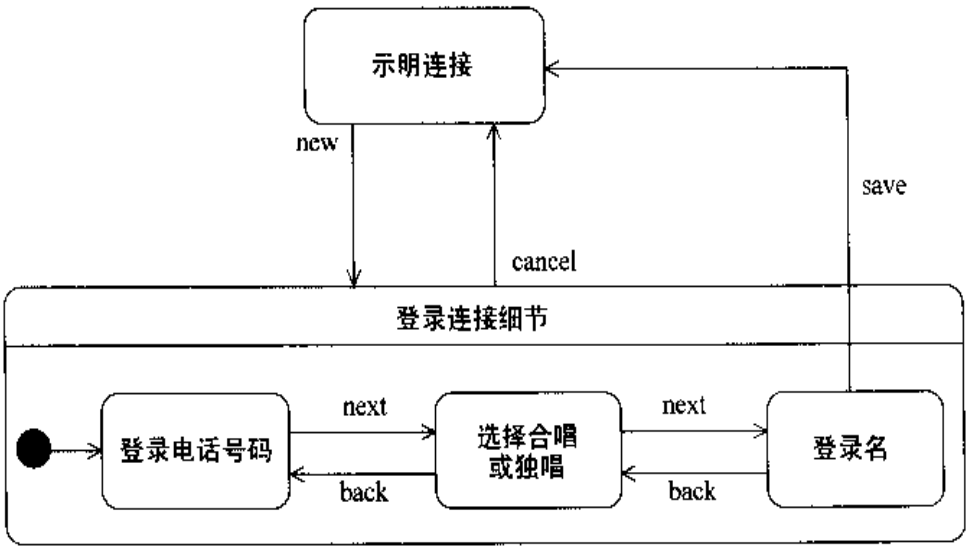


图 10.4 带嵌套子态的超态

并发状态

可以把一些状态拆分成几个并发运行的正交状态图。

图 10.5 示明一个乏味的简单闹钟，它既可以播光碟又可以做收音机并示明现时或闹时。

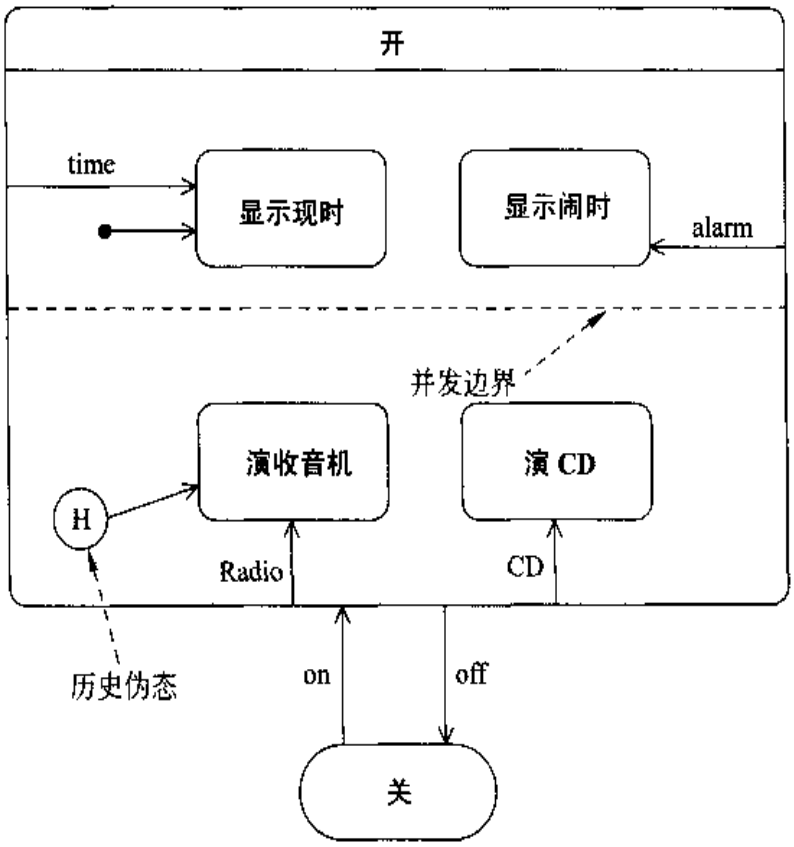


图 10.5　并发正交态

光碟/收音机和现时/闹时选择都是正交选择。如果要将它用一个非正交状态图来表示，那将是一个凌乱不堪的图（如果有更多状态的话，图将变得难以控制）。把两种行为领域分开来放入不同的状态图，可以使这种情形更为清晰。

图 10.5 也包含了一个**历史伪态**(history pseudostate)。这表示，当闹钟开关置为开时，收音机/光碟选择回到钟关时的状态。发自历史伪态的箭号指明在没有历史时首次所呈的状态。

状态图的实现

可以按三种主要方式来实现状态图:即嵌套开关、状态模式以及状态表。最直接处理状态图的方法是嵌套开关语句(如图 10.6)。这种方法虽然直观,但即使在这一简单情形,也显得冗长曲折。这种方法也很容易失控。因此,即使对简单情形,我也不乐意使用。

状态模式(state pattern)[Gang of Four]建立状态类的一个层次来处理各个状态的行为。图中每一状态有一个状态子类。对每一个只是转交状态类的事件,控制器都有一些方法。图 10.1 的状态图就生成一种由图 10.7 的各个类所表示的实现。

层次顶端是一个抽象类,它实现所有无所作为(do-nothing)的事件处理方法。对每一具体状态,你只是重载一些特定事件方法,对该方法状态有转接。

状态表(state table)获取状态图信息作为数据。因此,图 10.1 最终可以表示成一个表(见表 10.1)。这时或者构作一个利用运行时刻状态表的解释程序,或者构作一个基于状态表以生成各个类的代码生成程序。

显然,状态表不是只工作一次,而是在任何时间只要你有一个状态问题,就可以利用它。运行时刻状态表也可以无须重新编译(在有些语境中它是很方便的)而进行修改。在你需要时,状态模式是比较容易组合的,虽然它对每一状态,需要一个新类,但在

```
public void HandleEvent (PanelEvent anEvent){
  switch (CurrentState){
    case PanelState. Open:
      switch (anEvent){
        case PanelEvent. SafeClosed:
          CurrentState=PanelState. Wait;
          break;
        }
        break;
      case PanelState. Wait:
        switch (anEvent){
          case PanelEvent. CandleRemoved:
            if (isDoorClosed){
              RevealLock();
              CurrentState=PanelState. Lock;
            }
            break;
        }
        break;
      case PanelState. Lock:
          switch (anEvent){
            case PanelEvent. KeyTurned:
              if (isCandleIn){
                OpenSafe();
                CurrentState=PanelState.Open;
              }else{
                ReleaseKillerRabbit();
                CurrentState=PanelState. Final;
              }
              break;
          }
          break;
    }
  }
```

图 10.6　处理图 10.1 中状态转接的嵌套开关方法(用 C# 语言实现)

每种情形,只要写少量代码。

这些只是极少的几种实现,但应能使你对如何实现状态图有一概念。在每种情形,实现状态模型都导致标准款式的代码;因此,通常最好是用某种形式的代码生成程序来做这些事。

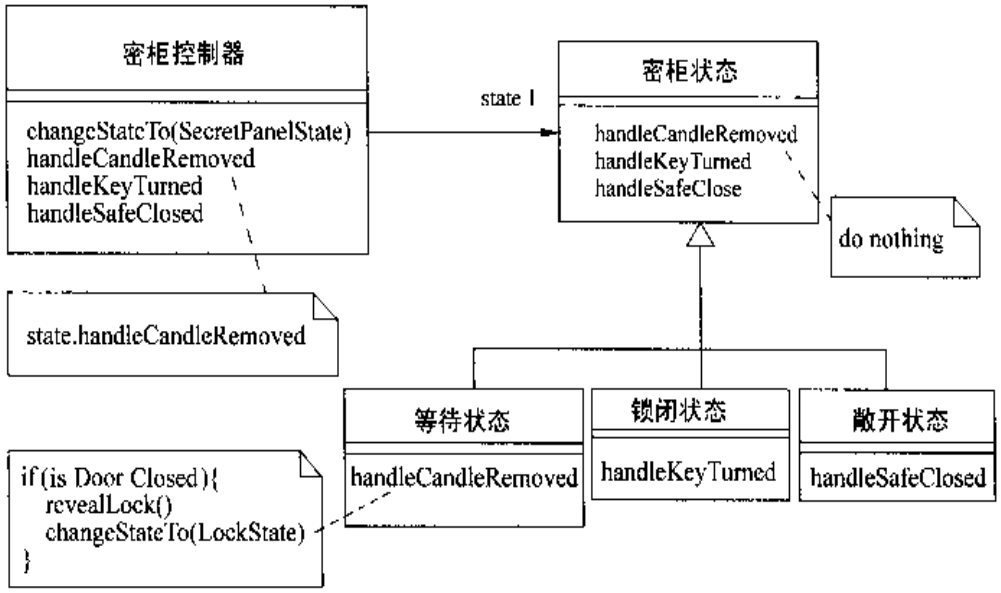


图 10.7 图 10.1 的状态模式实现

表 10.1 图 10.1 的状态表

源态	目标态	事件	监护	过 程
Wait	Lock	Candle removed	Door closed	Reveal lock
Lock	Open	Key turned	Candle in	Open safe
Lock	Final	Key turned	Candle out	Release killer rabbit
Open	Wait	Safe closed		

何时使用状态图

状态图擅长于表述跨用案的对象行为。状态图不太擅长于表述包含若干协作对象的行为。就其本身而论，把状态图和其他技术进行组合是有用的。例如，交互图（见第 4 章）擅长于表述单个用案中若干对象之行为，而活动图（见第 11 章）擅长于示明若干对象和用案的活动的一般顺序。

不是人人都发现状态图自然。留神人们是如何用状态图工作的。可能你的开发组并未发现状态图对其工作方式有用，这并不是一个大问题。和平时一样，你应该记住，结合使用为你工作的各种技术。

如果你果真使用状态图，就不要打算对系统中每一个类都绘制状态图。虽然这种途径（指对系统中每一个类都绘制状态图）往往是由注重形式的求全主义者们使用的，却几乎总是浪费人力和物力。只对那些表现出有趣行为的类才使用状态图，在其中构作状态图有助于了解正在进行什么。很多人发现 UI（用户接口界面）和控制对象具有值得用状态图刻画的行为。

何处找寻更多资料

《用户指南》[Booch，UML user]和《基准手册》[Rumbaugh，UML Reference]都有更多论述状态图的资料。实时系统设计

人员倾向使用很多状态图,因此,不足为奇的是,[Douglass]有很多关于状态图的论述,其中包括关于如何实现状态图的资料。[Martin]书中包含论述状态图的各种实现方法的很好的一章。

第 11 章

活 动 图

活动图是一种表述过程基理、业务过程以及工作流的技术。在很多方面，它们所起的作用与流程图类似，但是，与流程图表示法的主要区别是，活动图支持并行行为。

在 UML 的各种版本上已经看到活动图变动最大，因此，不足为奇的是，对 UML 2，活动图又有了显著的扩充和改动。在 UML 1 中，活动图被看作状态图的特例。这给工作流建模人员带来不少问题，而工作流建模却是很适宜使用活动图的。在 UML 2 中，就去掉了前述联系（指将活动图看成状态图的特例）。

图 11.1 示明活动图的一个简例。我们在**初始结点**（initial node）动作处开始，随后做**接订单**动作，完成后，便遇上一个**分岔**。**分岔**（fork）有一个入流和几个并发的出流。

图 11.1 说的是，**接订单**、**供货**、**开发票**以及它们的后继动作都是并行发生的。本质上这意味着它们之间的顺序无关紧要。我可以按订单供货，开发票，交付，然后再收款；或者也可以开发票，收款，按订单供货，然后再交付：你了解这一情况。

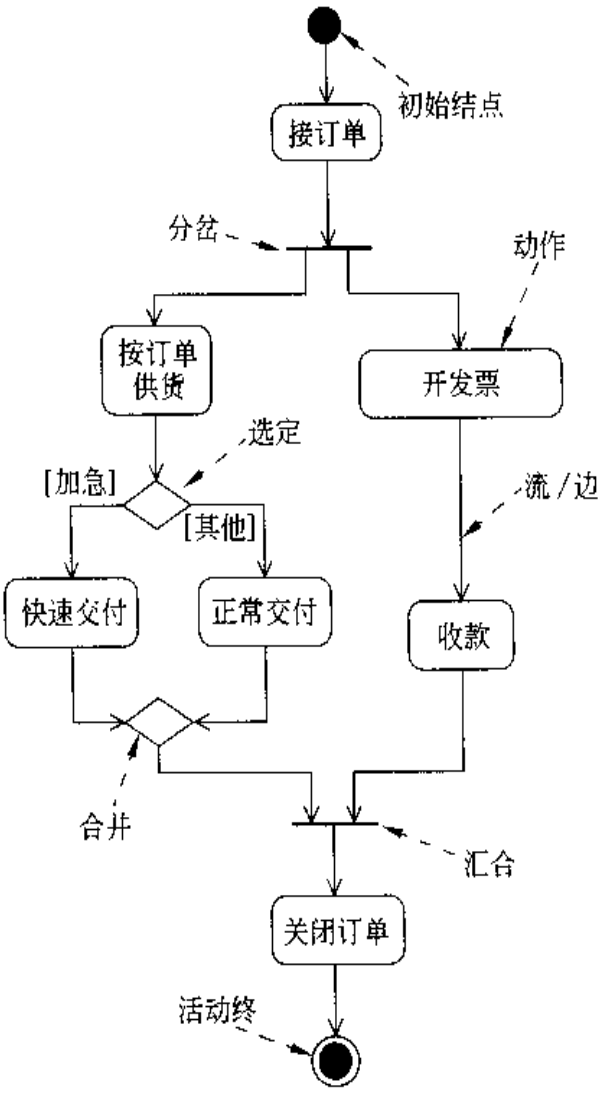


图 11.1 简单活动图

我也可以交替进行这些动作。从存储器中取第一行内容,开发票,取第二行,把发票放入信封,等等。或者其中有些事也可以同时做:一只手开发票,另一只手到存储器中去拿东西。根据这个图,任意顺序都是正确的。

活动图允许进行这一过程的任何人来选择做事的顺序。换言之，图仅仅说明应遵循的主要定序规则。这一点对业务建模来说是重要的，这是因为业务过程往往是并行出现的。对于并发算法，它也有用。在并发算法中各个独立的线程可以并行工作。

当你有了并行性，就需要进行同步。在货物交付及付款以前，不能关闭订单。在关闭订单动作前用一个**汇合**(join)来指明这一点。对一个汇合，仅当其所有的入流均已到达，才能处理出流。因此，仅当收款及交付都做完后，才能关闭订单。

在 UML 1 中，由于活动图为状态图的特例，为了平衡分岔和汇合，UML 1 有一些特殊规则。对 UML 2，这样的平衡就已不再需要。

你将注意到，活动图上的结点称为动作，而不称为活动。严格说，一项活动指的是一系列动作，因此，图中示明的一项活动是由若干动作构成的。

条件行为是用选定和合并来表述的。**选定**(decision)(UML 1 中称为分支(branch))有单个入流和多个受监控出流。每一出流有一监护，监护是置于两个方括号之间的一个布尔表达式。每当到达选定时，只能取一个出流。因此，监护间必须互斥。使用监护[else]指明，如果选定上所有别的监护均为假时，则应使用[else]流。

图 11.1 中，在按订单发货后，就有一个选定。如果是一份加急订单，则快速交付；否则，进行正常交付。

合并(merge)具有多个入流和单个出流。合并标志着由选定开动的条件行为的结束。

在我的图中，每一动作有单个入流和单个出流。在 UML 1

中,多重入流有一个隐式合并,亦即,如果引发了任一个流,你的动作就要执行。在 UML 2 中,这一点变成有一个隐式汇合;因此,仅当引发所有的流,动作才执行。由于这一改动,我建议,你对动作只使用单个入流和出流,并显式示明所有的汇合与合并,以免混淆。

动作的分解

可以把动作分解成子活动。我可以取图 11.1 中的交付基理,将它定义为它自己的活动(图 11.2)。这时就可以将它称为一个动作(图 11.3)。

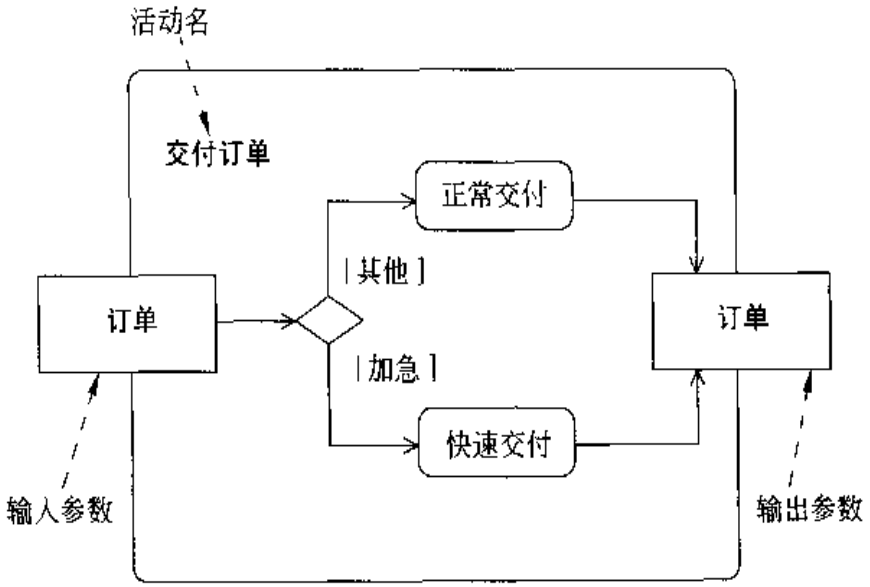


图 11.2 辅助活动图

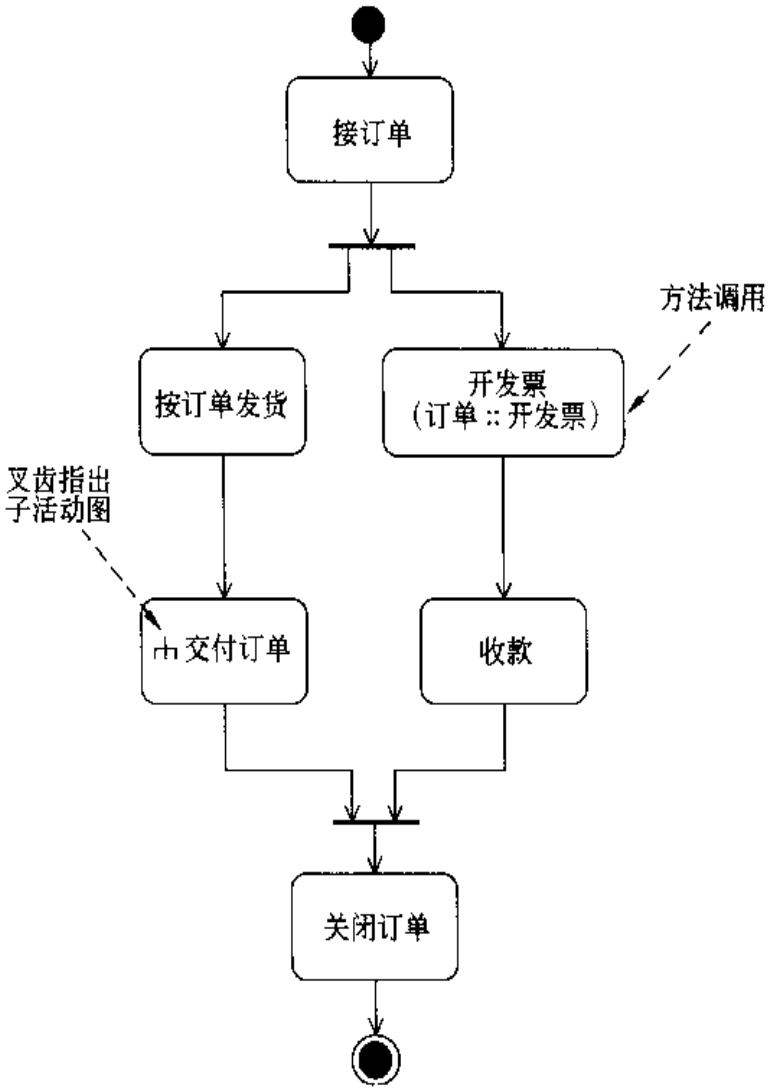


图 11.3　图 11.1 的活动修改成实施图 11.2 中的活动

动作可以实现为子活动或类上的方法。可以用一个叉齿符号来示明子活动。可以示明一个语法为*类名*::*方法名*的方法调用。如果所实施的行为不是单个方法调用，也可以往动作符号中写一段代码。

分划

活动图告诉你发生什么,但未告诉你谁做什么。在编程中,这指的是,图并未表达每一活动由什么类来负责。在业务过程建模中,这并未表达机构的哪一部门施行什么动作。这并不一定是一个问题;往往,有意义的是,要把注意力集中在做了什么,而不

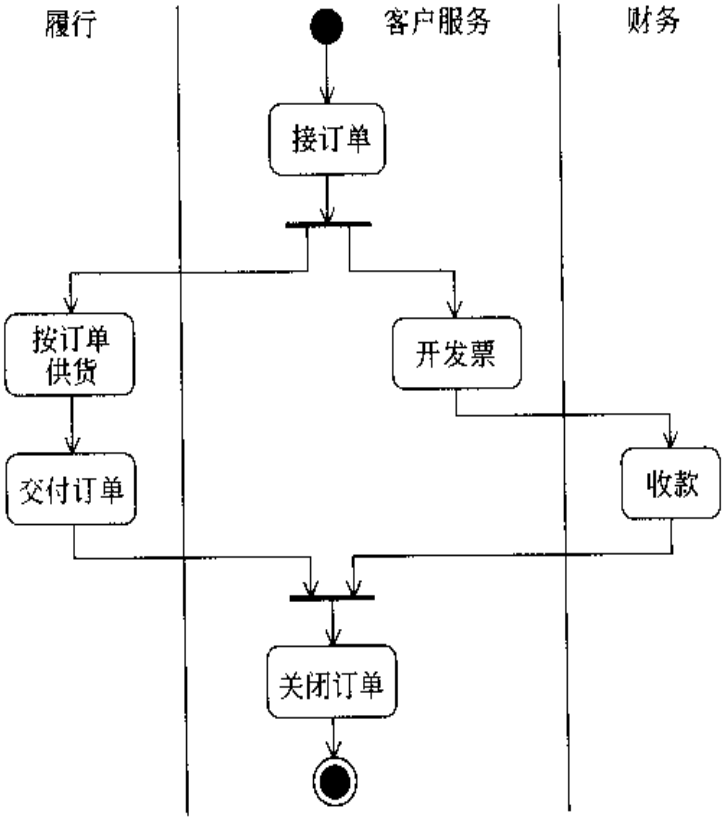


图 11.4 活动图上的分划

在谁做行为的那一部分。

如果要示明谁做什么，就要把一个活动图分成几个**分划**(partition)。分划示明一个类或一个机构单位施行哪些动作。图 11.4 给出一个简例，指明订单处理中所含的各个动作如何划分成不同的分划。

图 11.4 的划分是一个简单的一维划分。这种方式往往称为**泳道**(swim lane)。由于明显的理由，泳道是 UML 1.x 中所用的惟一形式。在 UML 2 中，可以使用二维的格，于是，游泳比喻对水就不再成立。你也可以对每一维按行或列进行层次式划分。

信号

在图 11.1 的简例中，活动图有明确定义的起点，它对应于程序或例程的启用。动作亦可对应于信号。

由于时间的推移，出现**时间信号**(time signal)。这样的信号可以指一个财政周期中的月终或实时控制器中的每一微秒。

图 11.5 给出一个监听两个信号的活动。**信号**(signal)指明活动接收来自外部过程的一个事件。这便表示，活动不断监听信号，而图则定义活动如何反应。

对图 11.5 的情形，在我的航班离港前两小时，我需要开始收拾行李。如果我很快就收拾好，在出租车到来之前，仍然不能离开。如果在收拾好之前，出租车就已来到，它就必须等我把行李收拾好后，我们才能出发。

除了接收信号以外，也可以发送信号。当我们必须发送消

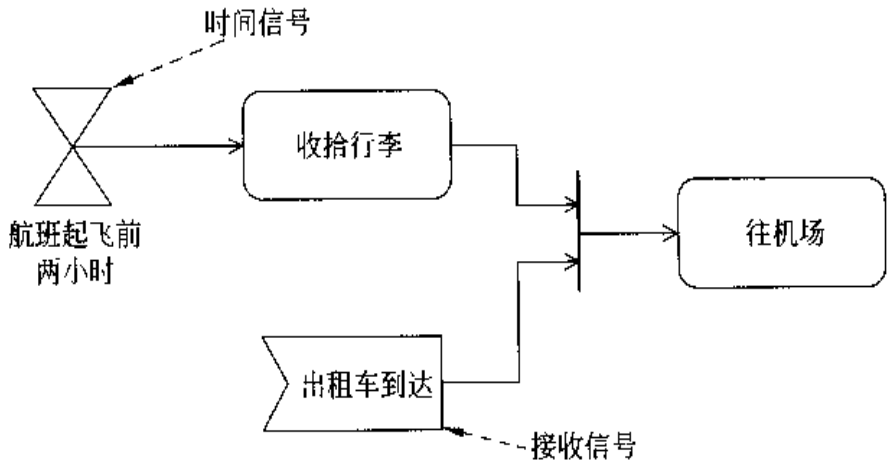


图 11.5 活动图上的信号

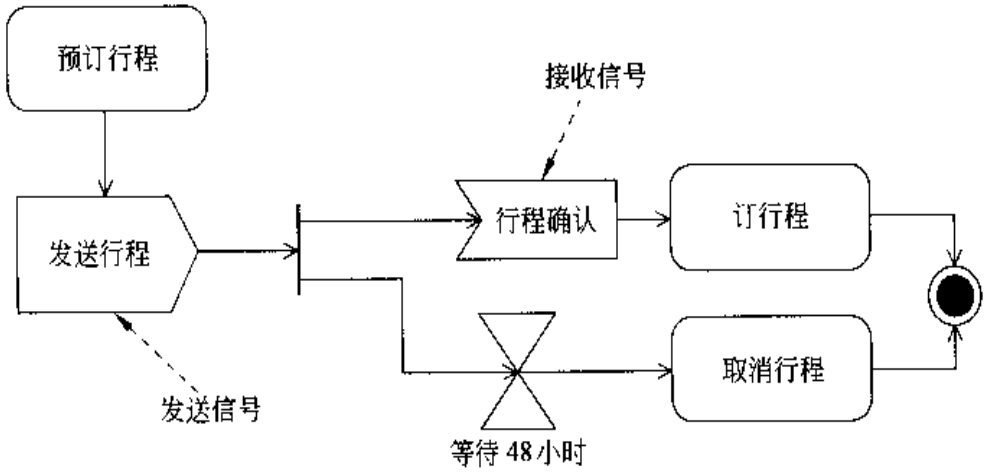


图 11.6 发送信号与接收信号

息,并在继续工作以前等待答复时,这是有用的。图 11.6 是一个成语"暂停"的好例子。请注意,两个流在比赛:首先到达终态的那一个流是赢家,且终止另一个流。

虽然接收通常只是等待一个外部事件,我们也可以示明一个流流入接收中。那就表示,在该流引发接收之前,不能监听信号。

权标

如果你敢于进入 UML 规范的超凡深度，你将发现，规范中的活动一节对**权标**(token)及其生产与消耗有很多讨论。初始结点创建一个权标，随后就把它传给下一动作，执行之，然后再将权标传给下一个。在分岔处，进入一权标，分岔则在其每个出流上均产生一个权标。反之，在汇合上，当每一个入权标抵达，在所有权标均出现在汇合处之前，什么事都未发生；在所有权标均已出现在汇合处时在出流上才产生一个权标。

可以用硬币或跨图移动的计数器来使权标形象化。处理更为复杂的活动图例时，权标往往更容易使一些东西形象化。

流与边

流(flow)与**边**(edge)是 UML 2 中用来表述两个动作之间连接的同义词。最简单的一种边就是两个动作之间的简单箭号。如果喜欢的话，也可以对边命名，但大多数时间，一个简单的箭号即已足够。

如果你安排路线有困难，就可以使用连接符，它使你可以不必去画一条整个距离的连线。利用连接符时，必须成对使用，一个用于入流，一个用于出流，而且这两个连接符要用相同的标号。我倾向在任何可能的情况下都要避免使用连接符，因为它们影响

控制流的形象化。

最简单的边所传递的权标除了用以控制该流外别无任何含义。但是,也可以沿着边传递对象;这时除了携带数据外,对象也可以起权标的作用。如果你沿着边传递一个对象,可以通过在边上放一个类框或者利用动作上的饰针来示明这一点。饰针还有更多的奥妙,下面即将对它进行表述。

图 11.7 中所有的方式都是等价的;对于你打算通信的东西,哪种方式表述得最好,你就应该用哪种方式。大多数时间,简单的箭号即已足够。

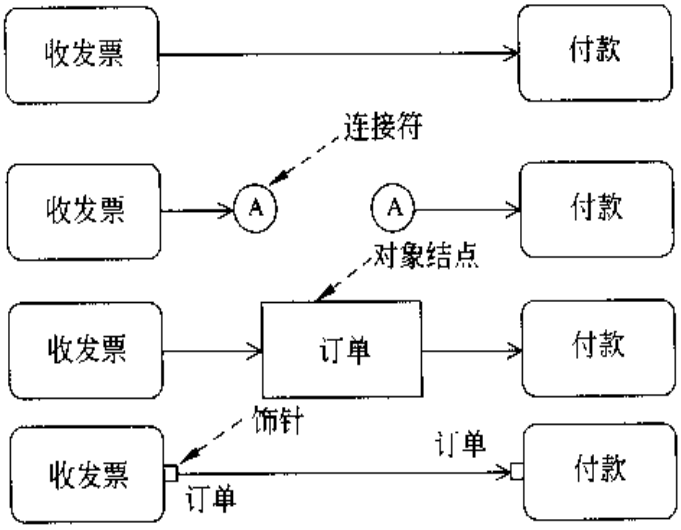


图 11.7 示明边的四种方式

饰针与转换

正如方法可以有参数一样,动作亦然。你并不需要在活动图

上示明参数信息，但如果希望的话，可以用**饰针**(pin)来示明这些信息。如果你在分解一个动作，饰针就对应于所分解的图上的参数框。

当你严格地绘制一个活动图时，就必须确保，出向动作的输出参数要和另一动作的输入参数匹配。如果它们不匹配，则可以指明一个**转换**(transformation)(见图 11.8)，使其转成匹配。转换必须是一个没有副作用的表达式：本质上说，它是一个在输出饰针上的查询，为输入饰针提供一个合适类型的对象。

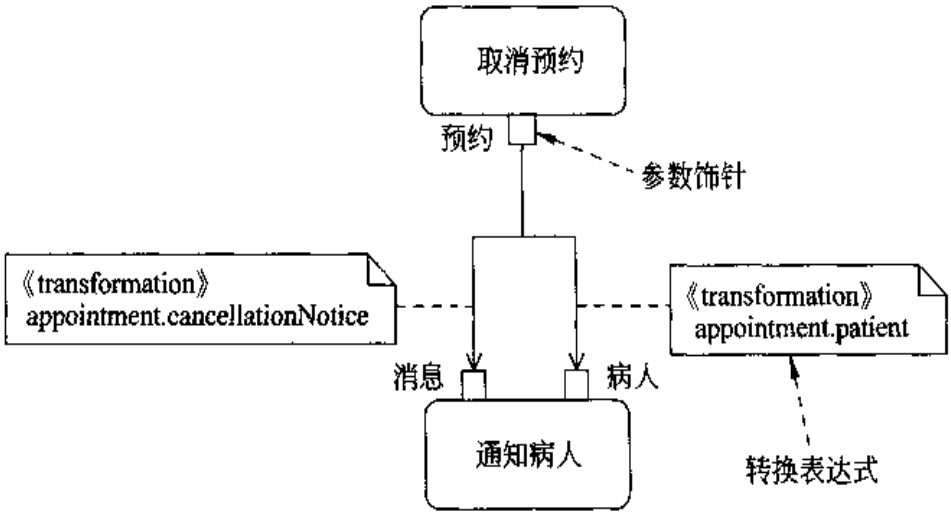


图 11.8　流上的转换

无须在活动图上示明饰针。当你打算查看各种动作所需的以及所产生的数据时，使用饰针就最好不过了。在业务过程建模中，可以利用饰针示明由动作所产生及消耗的资源。

如果使用饰针，安全的是，要示明同一动作的多个入流。饰针图示法补充了隐式汇合，而 UML 1 中没有饰针，因此，不会与先前的一些假定混淆。

展开区域

对活动图,你往往会遇上这样一些场合,其中一个动作的输出引发另一动作的多次启用。有几种示明这种情况的方法,但最好的一种方法是利用展开区域。**展开区域**(expansion region)是活动图上标出的一个区域,其中的动作对一个组中每一项均要发生一次。

在图 11.9 中,选择主题动作生成一个主题表作为输出。表中每一元素就成为写文章动作的一个输入权标。类似地,每一个审文章动作生成单篇文章,添加到展开区域的输出表中。当展开区域中所有权标最终都在输出组中时,展开区域便对这个表生成单一一个权标,传给出版业务通讯。

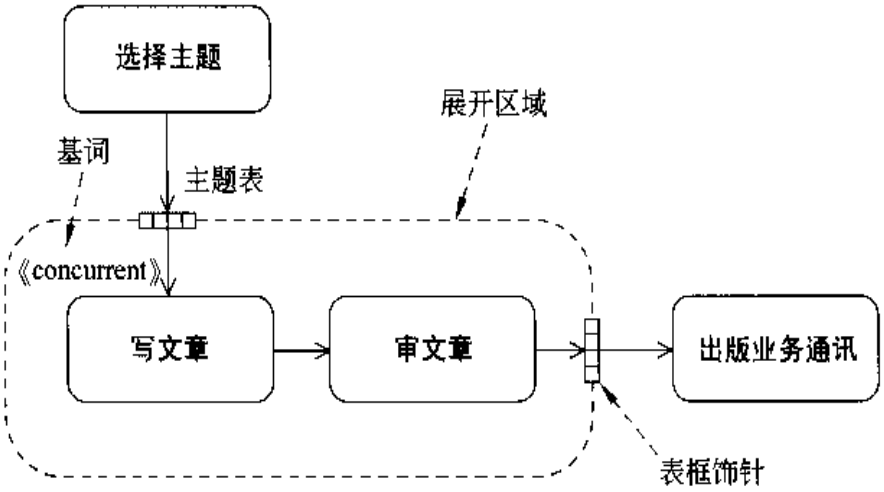


图 11.9 展开区域

这时，输出组中的项目数和输入组中的项目数相同。但是，输出组中的项目数也可以少于输入组中的项目数，在这种情形，展开区域起了过滤器的作用。

在图 11.9 中，所有的文章都是并行写并行审的，这是用基词《concurrent》来标记的。也可以有一个逐次展开区域。逐次展开区域必须一次完全处理一个输入元素。

如果只有一个需要多次启用的动作，就使用图 11.10 的简略表示。简略表示假定了并发展开，这是因为那是最常见情况的缘故。这种图示法对应于 UML 1 中的动态并发概念。

图 11.10　展开区域中单个动作的简略表示

流终

一旦得到多重权标（如在展开区域中那样），你往往会得到一些流，它们甚至在活动作为整体并未结束时就已停止。**流终**(flow final)指明在整个活动并未终止时，一个特定流的终结。

图 11.11 通过把图 11.9 的例子修改成允许文章被拒用的情形来示明这一点。如果一篇文章被拒用，权标就被流终摧毁。与活动终不同的是，这时活动的剩下来的部分仍可继续。这种方法允许把展开区域用作过滤器，在那里输出组要小于输入组。

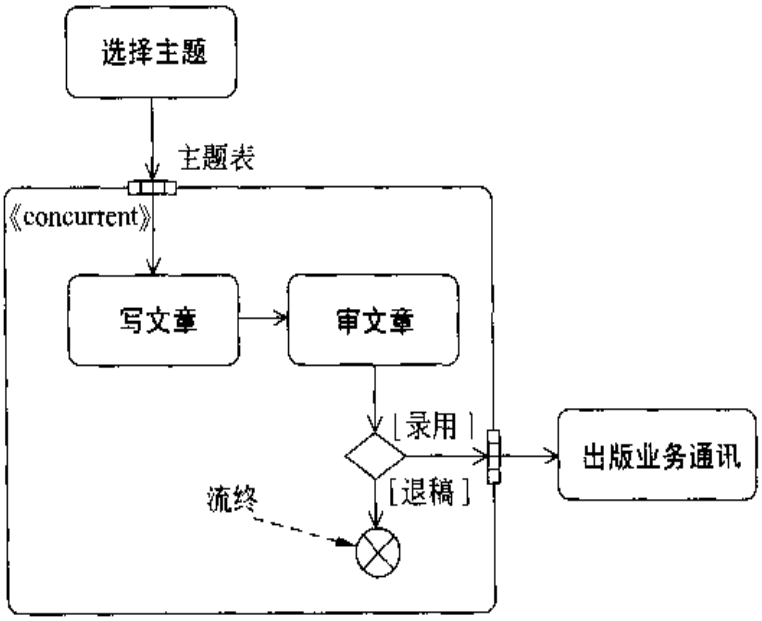


图 11.11 活动内的流终

汇合指明

根据默认，当汇合的所有入流均已抵达汇合，它就把执行传给其出流。(或者，更为形式的说法是，当在每一个输入流上都已有一权标抵达，它就往输出流上发出一权标。)在有些情形，特别是，当你有一个带多重权标的流时，有一个更富有内容的规则，则是有用的。

汇合指明(join specification)是一个附于汇合的布尔表达式。每当一个权标抵达汇合，就对汇合指明求值，如果它为真，就发出一个输出权标。因此，在图 11.12 中，当我选择了一种饮料或投入一硬币，机器就对汇合指明求值。仅当我已投入足够的钱，机

器才能吐出饮料解我的渴。像这种情形，如果你想要指出在每一个输入流上均已收到一个权标，就要对流加上标号，并在汇合指明中使用标号。

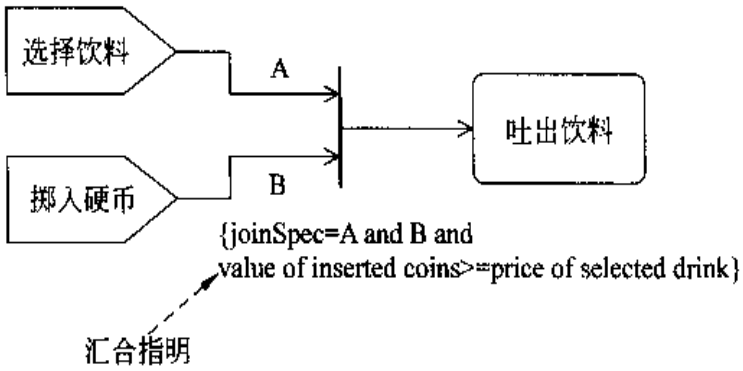


图 11.12　汇合指明

此外尚有更多内容

我应该强调的是，本章仅仅对活动图的讨论开了个头。与 UML 的很多内容一样，你可以针对这项技术单独写一本书。的确，我认为，对一本实际深入到 UML 图示法中以及如何使用这种图示法的书来说，活动图例是一个很合适的题目。

重要的问题是，活动图如何才能得到广泛使用。目前活动图还不是最为广泛使用的 UML 技术，而且它们的流建模前辈们在当时也并不很受爱戴。关于按这种方式来表述行为，图示技术尚未广泛流行。另一方面，在一些要求受到压抑的团体中，却有感到标准技术会有助于达到这一要求的迹象。

何时使用活动图

活动图最大的优点是,它们支持并鼓励并行行为。这使它们成为工作流建模和过程建模的一项重要工具。的确,对 UML 2 的推动很多都来自介入工作流的人们。

你也可以利用活动图作为遵照 UML 的流程图。虽然这允许你按照严格遵循 UML 的方式构作流程图,但却难于令人十分兴奋。原则上说,你可以利用分岔与汇合表述并发程序的并行算法。虽然我未曾更多地涉足并发界,我并未看到在那里利用活动图的人们的很多实证。我想,原因是并发程序设计的大部分复杂性在于避免竞争数据,而活动图对之却没有多大帮助。

做这件事(指绘制活动图)的主要好处是伴随将 UML 用作编程语言的人而来的。在这种情形,活动图代表用以表示行为基理的一种重要技术。

我常常看到用于表述用案的活动图。这一途径的危险性是,往往领域专家不易领会活动图。如果这样,你最好去用通常的正文形式。

何处找寻更多资料

虽然活动图总是相当复杂的,并且对 UML 2 来说,甚至更为复杂,但迄今尚无一本深入表述活动图的书。我希望有朝一日这

一空白会得到填补。

各种面向流的技术在风格上都与活动图类似。较为人知(还难说是“众所周知”)的一种技术是佩特里(Petri)网,对它而言,http://www.daimi.au.dk/PetriNets/是一个好的万维站点。

第 12 章

通 信 图

通信图(communication diagram)是一种着重阐明交互中各个参加者之间的数据连接的交互图。与顺序图依垂直方向对每一参加者画一条生命线并示明消息顺序不同,通信图允许自由放置参加者,允许画一些连线来示明各个参加者如何相连,并利用编号示明消息顺序。

在 UML 1.x 中,这些图称为**协作图**(collaboration diagram)。这一名称一直为人所用,而我怀疑,要经过一段时间之后,人们才会习惯使用新名。(这些图和协作[第 170 页]不同,所以,名才改动。)

图 12.1 示明一个与图 4.2 中相同的集中式控制交互的通信图。用通信图可以示明各个参加者如何连接起来。

除了示明作为关联实例的连接外,也可以示明瞬时连接,后者只在交互的语境中发生。在这种情形下,从订单到产品的《local》连接是一个局部变量;其他的瞬时连接是《parameter》,《global》。这些基词在 UML 1 中使用,但在 UML 2 却取消了。

由于它们有用，我指望它们在习惯用法中仍然保留下来。

图 12.1 中的编号方法简单易懂而且常用，但其实并不是合法的 UML。要成为合法的 UML，就必须像图 12.2 那样，使用嵌套十进制数编号方案。

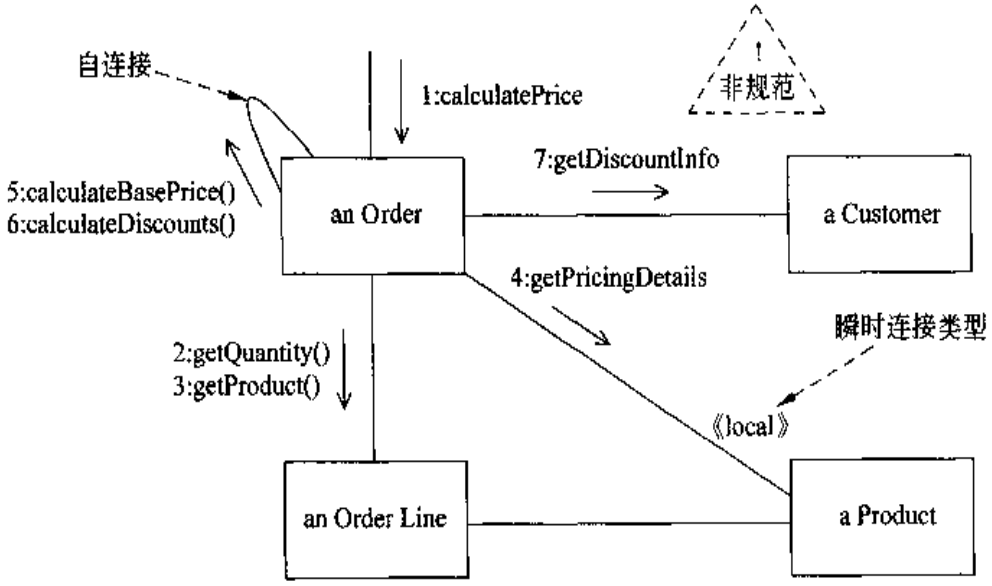


图 12.1　集中式控制通信图

使用嵌套十进制数的理由是解决自调用的歧义。在图 4.2 中，可以清楚地看到在方法 calculateDiscount 内部调用 getDiscountInfo，但是，利用图 12.1 中的无层次编号，你就不能分辨出究竟是在 calculateDiscount 内部还是在从头至尾的 calculatePrice 方法内部调用 getDiscountInfo。嵌套编号方案解决了这一问题。

纵然不合法，很多人还是喜欢无层次编号方案。嵌套号数可能把图弄得很乱，特别是在多层嵌套时，会导致像 1.1.1.2.1.1 这样的顺序号数。在这些情形，拯治歧义比歧义本身更坏。

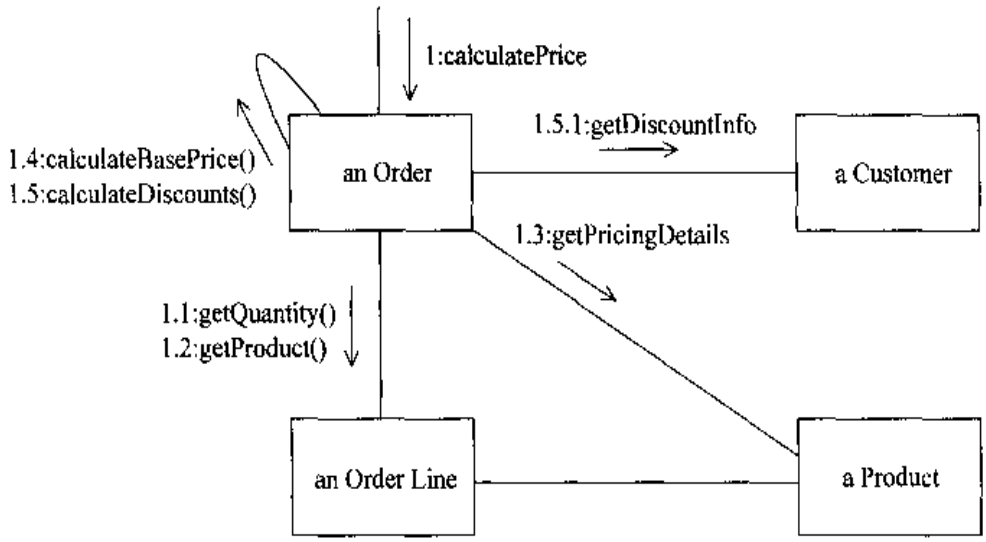


图 12.2 具嵌套十进制数编号的通信图

除了数以外，你也可能在消息上看到字母；这些字母指明不同的控制线程。因此，消息 A5 和 B2 就应该在不同的线程中；消息 1a1 和 1b1 就应该是在消息 1 内并发嵌套的不同线程。你也会在顺序图上看到线程字母，尽管这未能形象地表达并发性。

通信图对控制基理并没有任何精确图示法。它们使你能利用迭代标记与监护(第 76、77 页)，但并未使你能完全指明控制基理。对创建和删除对象，没有特殊图示法，但基词《create》和《delete》却是常见的惯用词。

何时使用通信图

关于通信图的主要问题是，何时使用通信图而不使用更为常用的顺序图。这种选择更多地依赖于个人爱好：有些人喜欢这一

种图而不喜欢另一种图。与其他方面相比，往往个人爱好对选择的动力更强。总的看来，大多数人似乎都喜欢顺序图。这一次，我是站在多数人一方。

一种更为合理的态度是，要强调调用顺序时，用顺序图更好；要强调连接时，通信图更好。很多人发现通信图在白板上比较容易改动，因而，使用通信图是探究供选方案的好办法，尽管在那些情形，我往往喜欢 CRC 卡。

第 13 章

复合结构

UML 2 中一个重要的新特征是层次地把一个类分解成一个内部结构的能力。这就使你能把一个复杂对象分成若干部分。

图 13.1 示明一个带有所供接口与所需接口的 TV 观众类。我已用两种方式对它示明：利用球座图示法和内部列表法。

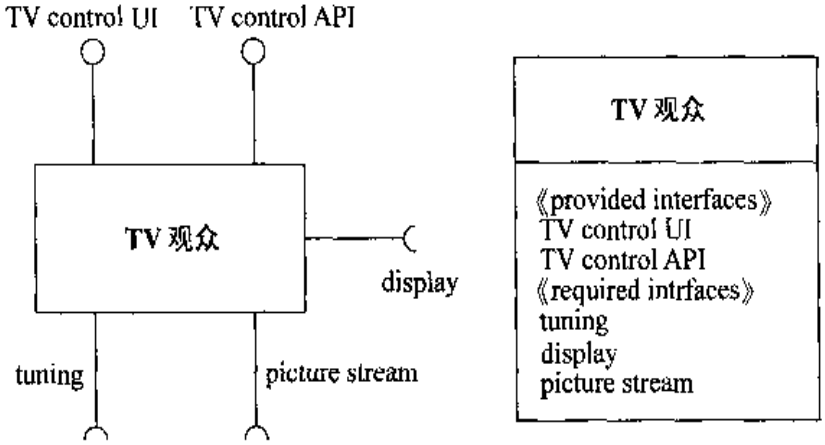


图 13.1　示明 TV 观众及其接口的两种方式

图 13.2 示明这一个类如何在内部分解成两部分以及哪些部分支持并需要哪些不同接口。每一部分按名：类的形式命名，其中两个成分（名和类）都是任选的。这两部分不是实例规约，因而，它们写成黑体，而非下划线。

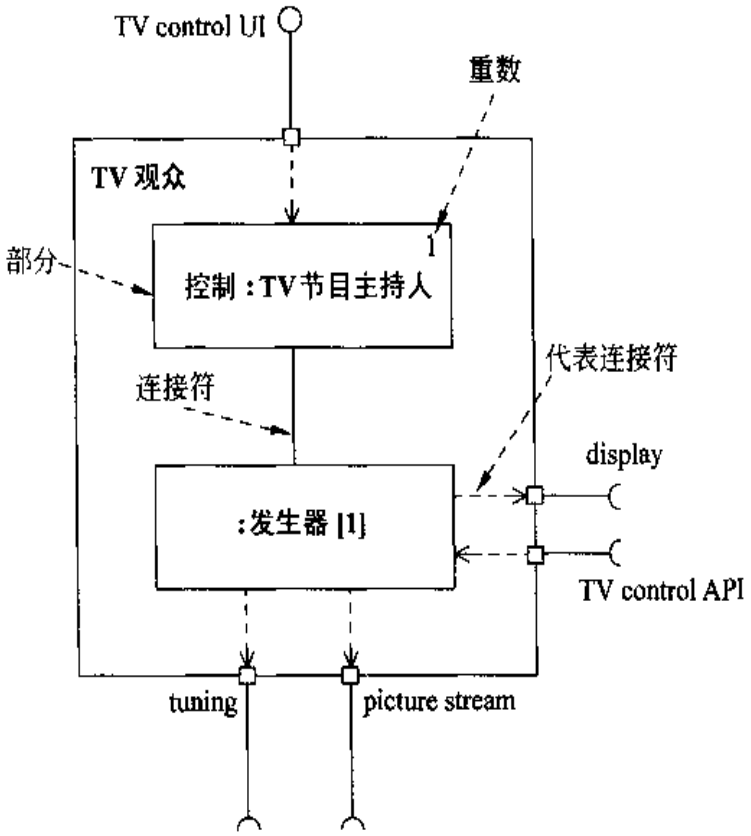


图 13.2　构件的内部观(Jim Rumbaugh 推荐之例)

你可以示明一个部分有多少实例出现。图 13.2 说的是，每一 TV 观众包含一个发生器部分和一个控制部分。

为了示明实现接口的那一部分，可从该接口画一个代表连接符。类似地，为了示明需要接口的那一部分，示明一个连往该接口的代表连接符。也可以用一条简单的线（像我在这里所做的那

样)或者用球座图示法(第 91 页)来示明各部分之间的连接符。

你可以对外结构添加一些港口(图 13.3)。港口使你能把所需接口与所供接口聚组成构件与外界的逻辑交互。

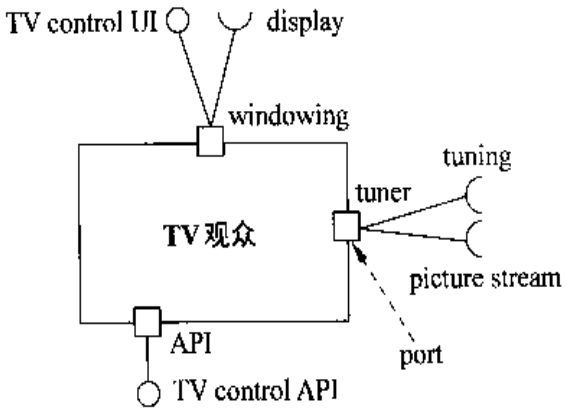


图 13.3 具有多个港口的构件

何时使用复合结构

复合结构是 UML 2 的新成分,尽管有些较老的方法已有某些类似的概念。思考包和复合结构之间区别的好方法是,包是编译时刻的聚组而复合结构则示明运行时刻的聚组。就其本身而论,复合结构自然宜于示明构件以及如何把构件分解为部分,因此,这种图示法大部分是用于构件图的。

由于复合结构是 UML 的新成分,要知道它们在实践中如何有效,尚为时过早;UML 委员会的很多成员都认为,复合结构图将成为对 UML 的一种很有价值的增添成分。

第 14 章

构 件 图

面向对象界一直广为流传的一项争论就是构件和正常类之间的区别为何。我不想在这里来解决这一争论，但是可以向你指明 UML 用以区分它们的图示法。

UML 1 对构件有一特别符号(图 14.1)。UML 2 去掉那个图符，但允许用一个形状类似的图符附于类框。或者，也可以利用基词《component》。

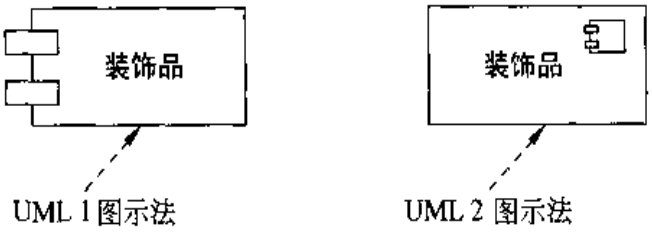


图 14.1 构件图示法

除了这一图符外，构件并未引进我们未曾见过的任何图示法。各个构件通过所实现的接口(即所供接口)及所需的接口(即

所需接口)彼此相连,往往是和类图一样,利用球形-托座图示法(第 91 页)。也可以利用复合结构图对构件进行分解。

图 14.2 示明一个构件图例。在该例中,收银机可以利用销售消息接口与销售服务器构件相连。由于网不可靠,设立了一个消息队列构件,于是收银机在网可用时就与服务器交互,在网不可用时,就与队列交互;当网变为可用时,队列再与服务器交互。因此,消息队列既提供销售消息接口与收银机交互,又要求该接口与服务器交互。服务器分解成两个主要构件。事务元处理程序实现销售消息接口,计账驱动程序与计账系统交互。

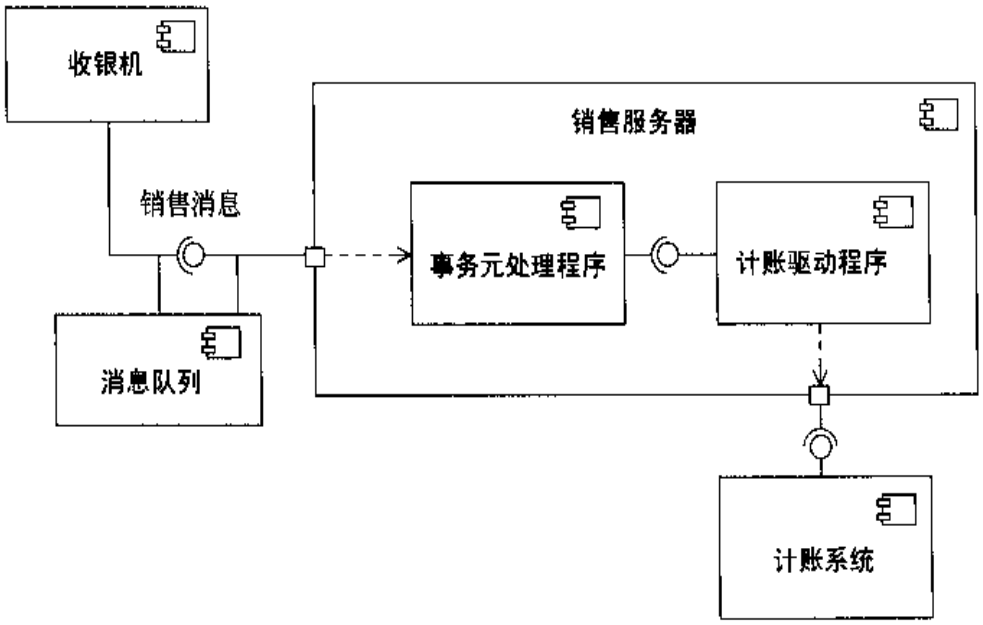


图 14.2 构件图例

正如我已说过,"什么是构件"这一问题是一个争论不休的题目。我发现一段颇为有益的陈述是:

构件不是一种技术。技术人士似乎发现对此难以

理解。构件是涉及客户如何与软件相联系的一种构造。客户希望一次购买所需软件的一部分(即一个软件片),并能使之升级,正如他们可以使音响系统升级一样。客户希望一些能和老的软件片无缝工作的新的软件片,并能根据他们自己的日程表而不是制造商的日程表来升级。他们希望能结合使用来自不同制造商的软件片并能使之匹配。这是一项很合理的要求,只是难以满足。

(Ralph Johnson, http://www.c2.com/cgi/wiki?DoComponentsExist)

要点是,构件代表可以独立购买与升级的软件片。因此,把一个系统分成若干构件既是一个技术抉择,又是一个销售抉择,对此,[Hohmann]是一本极好的入门书。它还提醒你要留意粒度过细的构件,原因是,构件过多,就难以处理,特别是,当出现版本过滥时,会有"动态链接库(DLL)灾难"。

在 UML 的较早版本中,构件用来代表物理结构,例如,DLL。那种表示现在已不再合法。为了表示物理结构,现在可以使用制品(第 119 页)。

何时使用构件图

当你将系统分成若干构件,并希望借助接口或者把构件分解为一些更为低层的结构示明其间相互关系时,就要用构件图。

第 15 章

协　　作

与本书其他各章不同，本章并不对应于 UML 2 中一种正式的图。标准把协作作为复合结构的部分来讨论，但其实二者的图很不同，并且把它用于 UML 1 中和复合结构并无任何联系。因此，我感到，最好是作为单独一章来讨论协作。

让我们来考察拍卖概念。在任何一项拍卖中，有一个卖主、一些买主、很多物品以及一些销售报价。我们可以用类图（图 15.1）来表述这些要素，也许还有一些交互图（图 15.2）。

图 15.1 不是一个很正规的类图。开始，把这个图的周围环以一个代表拍卖协作的虚椭圆。其次，协作中的所谓类并不是类而是随着协作的应用将被认知的**角色**（role），因此，它们的名不是大写的。通常，看到一些实在接口或对应于协作角色的类，但是，你不一定要有它们。

在交互图中，对参加者所用的标记和用案中稍有不同。在协作中，命名方案是：*参加者名/角色名：类名*。和通常一样，所有这些成分都是任选的。

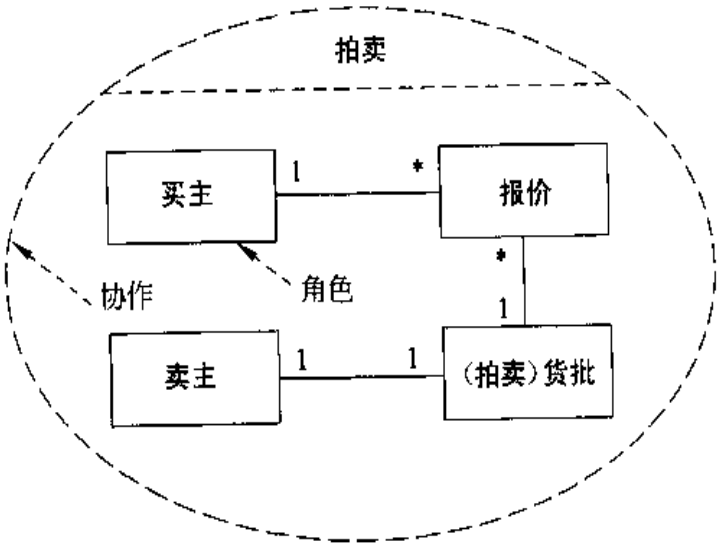


图 15.1 带角色类图的协作

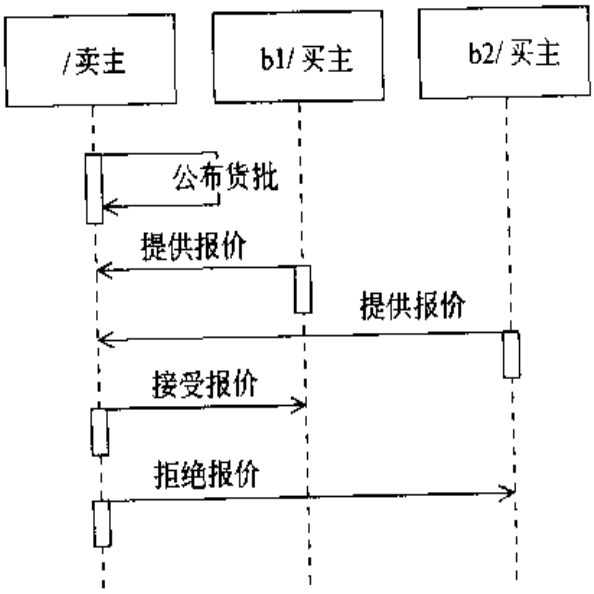


图 15.2 拍卖协作顺序图

当你利用协作时,可以通过把一个协作出现放在一个类图上来表示(如在图 15.3 中)。这个类图乃是由应用中的一些类组成的类图。从协作到那些类的连接指明这些类如何扮演在协作中

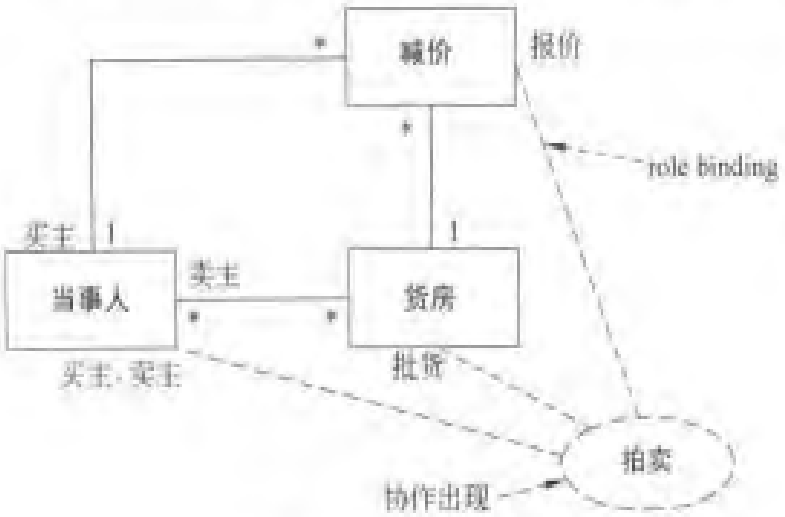


图 15.3 协作出现

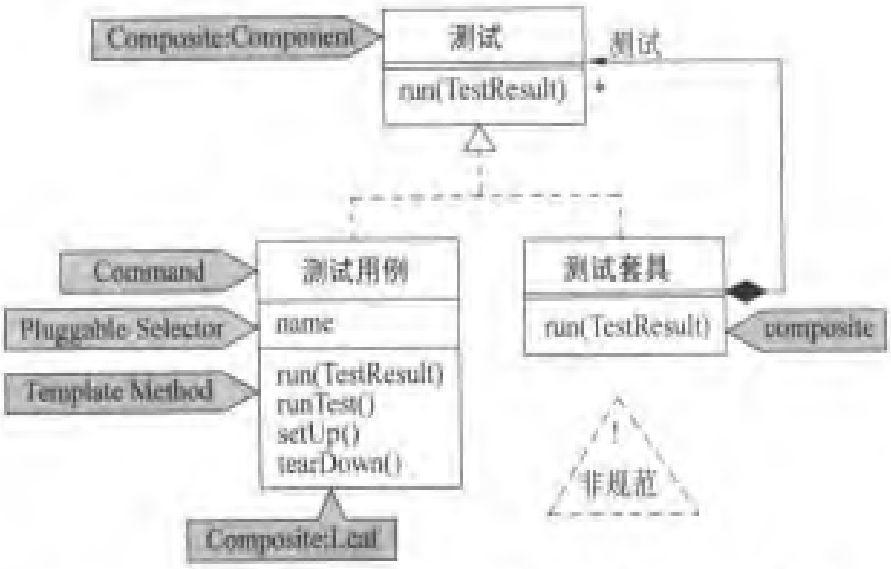


图 15.4 示明 JUnit (junit.org)中模式使用的非标准方式

定义的各个角色。

UML 建议，你可以利用协作出现图示法来示明模式的使用，但是，却很少有模式作者这样做。Erich Gamma 开发了另一种美妙的图示法(图 15.4)，图的各个成分均标以模式名或模式：角色的组合。

何时使用协作

在 UML 1 中就有了协作，但是，我承认，我很少使用它们，即使在我写模式时也是如此。在角色由不同的类扮演时，协作的确提供了一种把交互行为聚组成块的方式。但实际上，我并未发现它们是一种使人不得不采用的图型。

第 16 章

交互概观图

交互概观图是将活动图与顺序图嫁接在一起的图。你可以把交互概观图想象为活动图(其中把活动换成一些小顺序图)或者想象为利用示明控制流的活动图图示法的分解过的顺序图。不管哪种方式,它们总是使人感到有点古怪的混合。

图 16.1 示明一个简例,其图示法在活动图及顺序图两章中已经司空见惯。在这一个图中,我们希望产生并编排一个订单综述报告。如果客户是外部的,就从 XML 取得信息;如果是内部的,就从数据库中取得信息。小顺序图示明这两种选择。一旦取得数据,就编排报告;在这种情形,并不示明顺序图,但只是用一个指引交互构架来指引它。

何时使用交互概观图

交互概观图是 UML 2 的新成分。对它们在实践中的成功程

度进行评价尚为时过早。我对交互概观图并不热衷，这是因为我认为它们结合了两种方式，而实际上并没有结合好的缘故。取决于何者服务于你的目的更好，你或者画活动图，或者画顺序图。

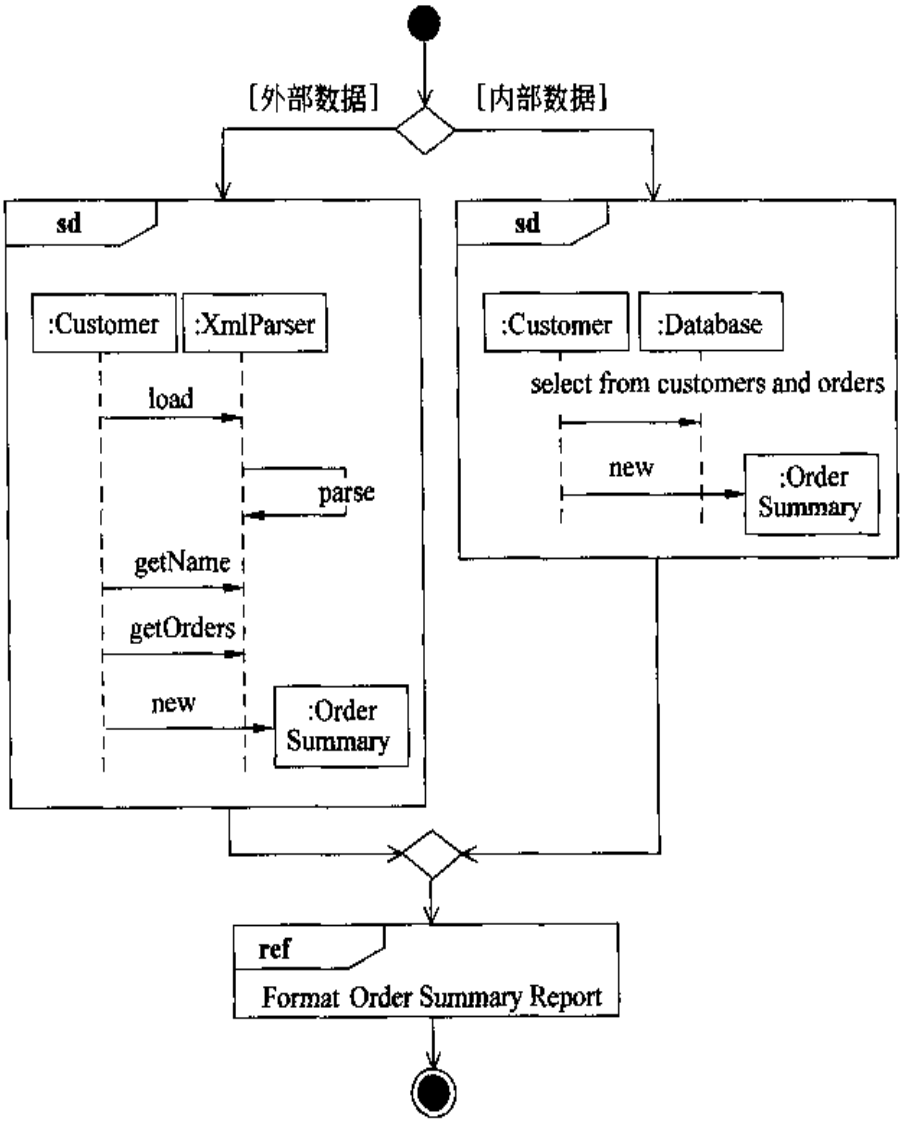


图 16.1　交互概观图

第 17 章

定 时 图

离开中学后，在转到计算领域以前，我开始是从事电子工程工作。因此，当我看到 UML 把定时图定义为它的一种标准图时，就感到有某种令人痛苦的熟悉感。定时图在电子工程中已经长期存在，似乎从来不需要 UML 的帮助去定义定时图的含义。但是，由于它们是在 UML 中，则值得简单一提。

定时图是交互图的另一种形式，其中着重点是对约束定时：或者是对一单个对象，或者更有用的是，对一群对象。让我们拿一个关于咖啡壶的基于抽吸装置与加热板的简单案况来看。设想一条规则，它说的是，在抽吸装置打开和加热板打开之间必须至少要隔十秒钟。储水容器变空时，抽吸装置则要关闭，而加热板继续保持加热不能超过十五分钟。

图 17.1 与图 17.2 是示明这些定时约束的两种选用方式。两个图示明相同的基本信息。主要区别是图 17.1 通过把一条水平线移往另一条水平线来示明状态改变，而图 17.2 则保持相同的水平位置，但用一个十字形示明状态改变。在只有少数状态时

（如在本例的情形），图 17.1 的方式工作得更好；在有很多要处理的状态时，图 17.2 则更佳。

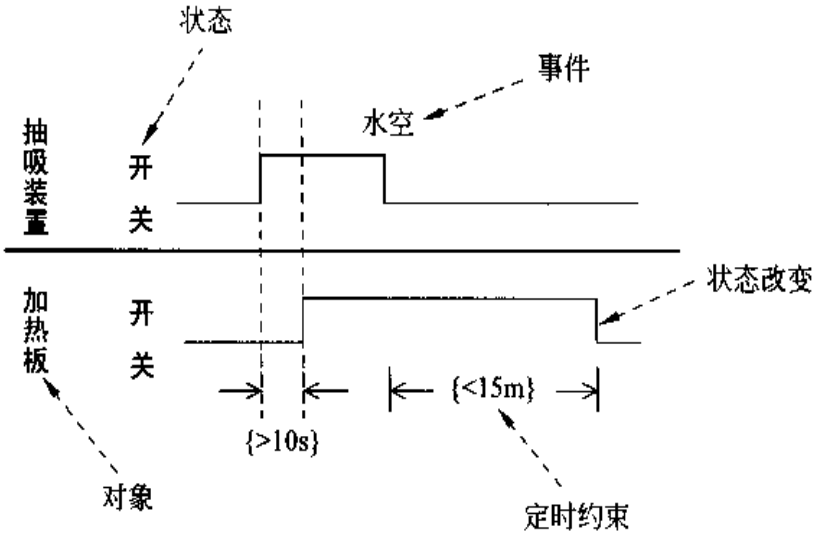


图 17.1　用线段示明状态的定时图

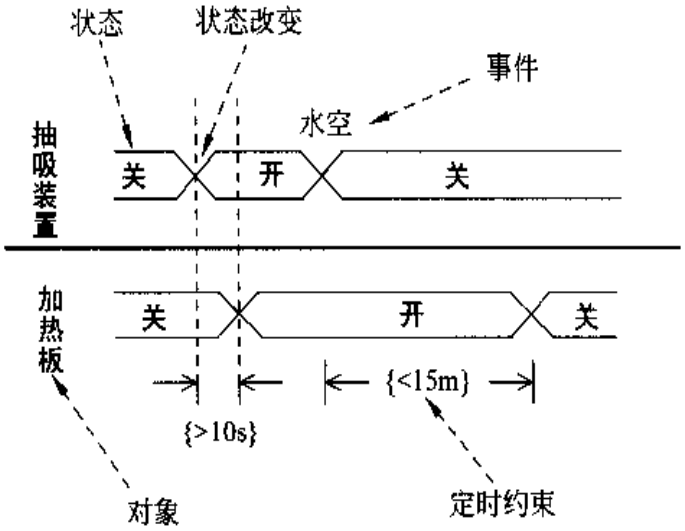


图 17.2　用区域示明状态的定时图

在[>10s]的约束上所用的虚线是任选的。当你认为它们有助于明确定时约束哪些事件时,就使用它们。

何时使用定时图

定时图用以示明不同对象上状态改变之间的定时约束。硬件工程师对这些图特别熟悉。

附录

UML 各个版本间的变动

当本书第一版在书架上出现时，UML 处于版本 1.0。看来，好多内容已经稳定，并且它是在 OMG 认可的过程中。此后，已有不少修订。在本附录中我描述了自版本 1.0 以来曾经出现的一些重要的变动以及这些变动是如何影响本书内容的。

本附录概述了这些变动。因此，如果你有本书的稍前印本的话，可以使它跟上时代。我对本书已作了一些变动以跟上 UML 的现状。所以，如果你有稍后印本的话，它表述的就是它在印刷时的情况。

UML 的修订

UML 最早的公开发布是统一方法 0.8 版本。它是在 1995 年 10 月举行的 OOPSLA 会议上发布的。这是 Booch 和 Rumbaugh 的工作，原因是 Jacobson 在此之前尚未加入 Rational

公司。1996 年 Rational 公司发布了 0.9 和 0.91 两个版本,其中包含了 Jacobson 的工作。在 0.91 版本以后,他们更名为 UML。

Rational 公司及一组伙伴于 1997 年 1 月把 UML 1.0 版本提交给 OMG 分析与设计工作队。此后,Rational 公司合伙以及其他一些提交者把他们的工作进行组合,于 1997 年 9 月提交了一份关于 OMG 标准的建议书并作为 UML 的 1.1 版本。到 1997 年年底,这一建议书由 OMG 采纳。但是,在一阵最黑暗的混乱中,OMG 称此标准为 1.0 版本。因此,那时的 UML 既是 OMG 的 1.0 版本又是 Rational 的 1.1 版本,不要和 Rational 1.0 版本相混。实际上,人人都称该标准为 1.1 版本。

从那时起,UML 中有了许多进一步的发展。1998 年出现了 UML 1.2,1999 年出现了 UML 1.3,2001 年出现了 UML 1.4,2003 年出现了 UML 1.5。在 UML 1.x 各个版本之间的大部分变动都是对 UML 所作的较为深奥晦涩的考虑。只有 UML 1.3 作了一些引人注目的变动,特别是关于用案与活动图。

随着 UML 1 系列在继续,UML 开发者们将目光瞄准到藉 UML 2 对 UML 作一次重大修订。首次建议邀请(REP)于 2000 年发出,但直到 2003 年,UML 2 才开始实际稳定。

UML 的进一步的发展几乎肯定是会出现的。UML 论坛(http://uml-forum.com)通常是一个找寻更多信息的好去处。我也在我的站点(http://martinfowler.com)上保存一些 UML 信息。

《UML 精粹》中的变动

随着这些修订的继续进行，我已经打算用一些接踵而来的重印机会来修订《UML 精粹》一书以跟上 UML 的形势。我也想趁此机会校正错误并进行澄清。

跟上时代步伐的变化频仍的时期是在《UML 精粹》第 1 版时，那时为了跟上新出现的 UML 标准，往往在各次印本间都须进行更新。第 1 次到第 5 次印本都是基于 UML 1.0 的。这几次印本之间对 UML 的任何变动都是细微的。第 6 次印本则考虑到 UML 1.1。

第 7 次到第 10 次印本基于 UML 1.2；第 11 次印本首次使用 UML 1.3。基于 1.0 以后的 UML 各个版本的印本在封面上都有 UML 的版本号数。

第 2 版的第 1 次到第 6 次的印本都是基于版本 1.3。第 7 次印本是最先考虑到版本 1.4 的轻微变动的。

第 3 版开始藉 UML 2 来更新本书(见表 A.1)。在本附录的其余部分，我对 UML 中从 1.0 到 1.1，从 1.2 到 1.3，从 1.x 到 2.0 的重大变动进行了概述。我并不讨论出现的所有变动，而只讨论那些我在《UML 精粹》中谈到的或者那些代表我应该在《UML 精粹》中讨论的重要特征的变动。

我继续遵循《UML 精粹》之精神：讨论 UML 的关键成分，理由是，这些成分影响现实世界项目中 UML 的应用。与往常一样，选择和建议都是我自己的。如果在我说的内容与正式 UML

文档之间有任何冲突,应以 UML 文档为准。(但是,请让我知道,以便能进行改正。)

表 A.1 《UML 精粹》与相应的 UML 版本

《UML 精粹》	UML 版本
第 1 版	UML 1.0—1.3
第 2 版	UML 1.3—1.4
第 3 版	UML 2.0 开始

我也趁此机会指出先前几次印本中的重要错误或遗漏。谨向对我指出这些问题的读者致谢。

从 UML 1.0 到 UML 1.1 的变动

类型与实现类

在《UML 精粹》的第 1 版中,我谈到视面以及这些视面如何改变人们绘制模型与解释模型(特别是类图)的方式。现在 UML 考虑到这一点,这是通过声明类图上所有的类都可以特化成类型或实现类。

实现类(implementation class)对应于你所处的软件开发环境中的类。**类型**(type)就更为模糊,它表示"较少囿于实现"的抽象。这可以是 CORBA 类型、类的规约视面或概念视面。如有必要,也可以加上衍型以进一步区分。

对特定的图,可以说所有的类都遵循一特定的衍型。这是根

据某一特定视面绘图时应做之事。实现视面要使用实现类，而规约视面与概念视面则要使用类型。

利用实施关系指出一个实现类实现一个或多个类型。

类型和接口不同。接口指望直接对应于 Java 或 Com 样式的接口。因此，接口只有操作，而无属性。

对实现类，只能使用单一静态分类，但对类型却可以使用多重分类与动态分类。（我这样假定，是因为主要的面向对象语言都遵循单一静态分类。如果有一天使用了支持多重分类或动态分类的语言，上述限制就全然不适用了。）

完整与不完整判别元约束

在《UML 精粹》的以前几次印本中，我谈过，关于泛化的{complete}约束指的是，超类型的任一实例也必须是该分划内的一个子类型的实例。而 UML 1.1 却规定，{complete}指的是，该分划内所有的子类型均已被指明，这两种说法并不是一回事。关于这一约束的解释，我已经发现有很多不一致的地方，因此，对它应谨慎从事。如果你希望指出，超类型的任一实例应是某一子类型的一个实例，那么，我建议使用另一约束以免混淆。当前，我使用的是{mandatory}约束。

组合

在 UML 1.0 中，利用组合指的是，连接是永恒的（或冻结的），至少对单值构件是如此。这一约束不再是定义的组成部

分了。

永恒与冻结

UML 定义了约束{frozen}以规定关联角色的永恒性。像当前对它所定义的那样,似乎它不适用于属性或类。根据我的实践,目前我使用术语冻结而不使用术语永恒。愉快的是,我已把这一约束用于关联角色、类以及属性。

顺序图上的回送(返回)

在 UML 1.0 中,顺序图上的回送(返回)利用实线箭头(⟶)代替实心箭头(➞)以资区别(见先前的印本)。这是一件烦神的事。原因是,这一区别太微妙,并且容易遗忘。UML 1.1 对回送(返回)利用破折箭号,我很满意。理由是,这使它大为明显。(由于我在《分析模式》[Fowler,AP]一书中使用过破折回送(返回),似乎对我是有影响的。)你可以使用 enoughStock := check()的形式对回送(返回)取一个名以备后用。

术语"角色"的使用

在 UML 1.0 中,术语**角色**(role)主要指的是关联上的方向(见先前的各次印本)。UML 1.1 把这一用法称作**关联角色**(association role)。还有一种**协作角色**(collaboration role),它是类的实例在协作中扮演的角色。UML 1.1 对协作大加强调,看

来，似乎“角色”的这种用法有可能变成主要用法。

从 UML 1.2(及 1.1)到 UML 1.3(及 1.5)的变动

用案

用案的变动包含用案间的新关系。UML 1.1 有两种用案关系：《uses》与《extends》，两者都是泛化的衍型。UML 1.3 提供了三种关系：

- 《include》构造是依赖的一种衍型。这表示，一个用案的路径包含在另一用案中。典型的是，这出现在少量用案共享一些公共步骤时。所包含的用案可以分解出公共行为。源于 ATM(自动取款机)的一个例子可以是取款和转账都使用有效客户。这取代了《uses》的通常用法。
- 用案泛化(generalization)指出，一用案是另一用案的变异。因此，我们可以有一个取款用案(基底用案)与另一个处理因资金短缺而被拒绝提款的用案。这一拒绝提款用案可以处置为一个特化提款用案的用案。(你也可以把拒绝提款处理为提款用案内的另一案况。)像这样的特化用案可以改变基底用案的任何方面。
- 《extends》构造是依赖的一种衍型。这提供了比泛化关系更为受控的扩张形式。这里基底用案说明一些扩张点。扩张用案只能在这些扩张点处改变行为。因此，如果排队购物，可以有一个购物用案，它带有一些扩张点，以取得送

货信息与付款信息。这时该用案便可以对常客扩张。对此,这一信息可以按不同方式取得。

在旧关系与新关系之间有些混淆。大多数人按照使用UML 1.3的《includes》一样的方式来使用《uses》。因此,对他们而言,可以说《includes》取代了《uses》。大多数人按以下两种方式使用UML 1.1的《extends》,即按照UML 1.3的《extends》的受控形式以及作为UML 1.3泛化风格的一般重置。因而,你可以想象UML 1.1的《extends》分解成UML 1.3的《extends》与泛化。

现在,虽然这一解释涵盖了我所见过的UML的很多用法,它却不是使用旧关系的严格正确方式。不过,很多人并不遵循严格用法,我其实也不希望在这里进行全面讨论。

活动图

当UML到达1.2时,关于活动图的语义还有不少未决问题。因此,UML 1.3包含了关于这些语义的相当多的严格化工作。

关于条件行为,现在你可以对行为合并以及行为分支使用菱形选定活动。虽然,为了表述条件行为,分支与合并均非必要,但是,越来越通常的方法是示明它们,只要你可以把条件行为括以括号。

现在同步线条指的是分岔(在分解控制时)或汇合(在控制同步到一起时)。但是,你对汇合不能随意添加条件。而且,必须遵循匹配规则,以确保分岔和汇合匹配。本质上说,这表明,每一分岔需有一相应的汇合,它把由该分岔开始的各个线程汇合起来。

也可以使分岔与汇合嵌套，虽然在各个线程直接从一个分岔进入另一分岔（或从一个汇合进入另一汇合）时，可以消去图上的分岔（或汇合）。

仅当所有进入线程均已完成时，才激活汇合。但是，对来自分岔的线程可以有一个条件，如果该条件为假，为了汇合目的，就把该线程看作完成。

多重引发装置特征不再出现。在其位置，可以在一个活动中有动态并发（在活动图内用一个 * 指明）。这一活动可并行启用多次；所有的启用都必须在任何外出转接以前完成。这不精确地等价于（虽然灵活性较差于）多重引发装置以及匹配同步条件。

这些规则使活动图的灵活性降低，但它们却可以确保活动图的确是状态机的特例。活动图与状态机之间的关系是 RTF (Revision Task Force)（修订工作队）中一个有争议的问题。

从 UML 1.3 到 UML 1.4 的变动

UML 1.4 中最明显的变动是增加了**侧图**(profile)。侧图使你能把一组扩张汇集到一个连贯协调的集合中。UML 文档包含几个侧图样例。连同侧图，还有一个涉及定义衍型的较好的形式体系，并且模型成分现在可有多个衍型；在 UML 1.3 中，它们只能有一种衍型。

往 UML 中增加了**制品**(artifact)。制品是构件的一种物理表现形式。因此，比方说，Xerces 包为一构件，而在我的磁盘驱动器上 Xercesjar 的所有那些复本都是实现 Xerces 构件的制品。

在 UML 1.3 以前,UML 元模型中并无处理 Java **包可见性**(package visibility)的任何成分。现在有了,符号是"~"。

UML 1.4 还使交互图中的实线箭头(—>)标记异步,这是一种使用相当不便且向后不匹配的变动。那种标记使得少数包括我在内的人瞠目以对。

从 UML 1.4 到 UML 1.5 的变动

这里的主要变动是往 UML 中增加了动作语义,这是使 UML 成为一种编程语言的必要一步。这样做了之后,就使得人们可以据此继续工作,而无须等待完整的 UML 2 的出现。

从 UML 1.x 到 UML 2.0

UML 2 代表对 UML 所发生过的最大变动。所有各种事情都随此次修订而改变,很多变动也影响到《UML 精粹》。

在 UML 内部,对 UML 元模型有了深刻变动。虽然这些变动并不影响《UML 精粹》中的讨论,它们对有些团体来说,却是很重要的。

最明显的一个变动是引进了一些新的图型。以前对象图与包图是广泛绘制的,但并非正式图型,现在它们是了。UML 2 把协作图改名为通信图。UML 2 还引进了一些新的图型:交互概观图、定时图以及复合结构图。

很多变动并未触及《UML 精粹》。我没有讨论诸如状态机扩张、交互图中的门以及类图中的幂类型等构造。

因此，对本节来说，我只讨论进入《UML 精粹》书中的那些变动。这些或者是对本书以前各版中讨论的事情的变动，或者是本版开始讨论的新的事情。由于变动相当广泛，我按本书各章进行组织。

类图：基础部分(第 3 章)

属性和单向关联现在对相同的特性概念基本上采用不同的图示法。删去了间断重数(如[2,4])。删去了冻结特性。增加了一列常用的依赖基词，其中有些是 UML 2 新的基词。基词《parameter》和《local》已被删去。

顺序图(第 4 章)

这里最大的变动是关于为了处理迭代控制、条件控制以及其他各种行为控制对顺序图增加的交互架构图示法。这种图示法使你能在顺序图中很完全地表示算法，尽管我并不确信，这些表示要比代码更清楚。在顺序图中删去了旧的迭代标记和消息上的监护。生命线头不再是实例，我将它们称作**参加者**(participant)。UML 1 的协作图在 UML 2 中改名为通信图。

类图：高级概念(第 5 章)

现在，衍型的定义更为精练贴切。因此，现在我把双尖括号

中的词称为基词,其中只有一部分是衍型。对象图中的实例现在是实例规约。类现在可以需要接口也可以提供接口。多重分类利用泛化集将各个泛化聚组成组。不再用特殊符号来画构件。主动对象不用粗线而改用双重直线。

状态机图(第 10 章)

UML 1 把短暂动作和长留活动分开。UML 2 把二者均称为活动,并对长留活动使用术语"**进行活动**"(do-activity)。

活动图(第 11 章)

UML 1 把活动图看成状态图的特例。UML 2 切断了这一连接。因此,取消了分岔与汇合的匹配规则,而这些规则 UML 2 活动图是必须遵循的。因此,它们最好是根据权标流而不是根据状态转接来理解。出现了一组新的图示法,其中包括时间与接收信号、参数、汇合指明、饰针、流转换、子图齿耙、展开区域以及流终等。

一个简单但不便的变动是,UML 1 把活动的多重入流看作隐式合并,而 UML 2 却把它们看作隐式汇合。为此,我建议在绘制活动图时,利用显式的合并或汇合。

泳道现在可以是多维的,因而一般称之为分划。

参考文献

[**Ambler**] Scott Ambler, *Agile Modeling*, Wiley, 2002.

[**Beck**] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.

[**Beck and Fowler**] Kent Beck and Martin Fowler, *Planning Extreme Programming*, Addison-Wesley, 2000.

[**Beck and Cunningham**] Kent Beck and Ward Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," *Proceedings of OOPSLA 89*, 24(10): 1—6. http://c2.com/doc/ oopsla89/paper.html.

[**Booch, OOAD**] Grady Booch, *Object-Oriented Analysis and Design with Applications*, *Second Edition*. Addison-Wesley, 1994.

[**Booch, UML user**] Grady Booch, Jim Rumbaugh, and Ivar Jacobson, *UML User Guide*, Addison-Wesley, 1999.

[**Coad, OOA**] Peter Coad and Edward Yourdon, *Object-Oriented Analysis*, Yourdon Press, 1991.

[**Coad, OOD**] Peter Coad and Edward Yourdon, *Object-Oriented Design*, Yourdon Press, 1991.

[**Cockburn, agile**] Alistair Cockburn, *Agile Software Development*, Addison-Wesley, 2001.

[**Cockburn, use cases**] Alistair Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2001.

[**Constantine and Lockwood**] Larry Constantine and Lucy Lockwood,

Software for Use, Addison-Wesley, 2000.

[**Cook and Daniels**] Steve Cook and John Daniels, *Designing Object Systems: Object-Oriented Modeling with Syntropy*, Prentice-Hall, 1994.

[**Core J2EE Patterns**] Deepak Alur, John Crupi, and Dan Malks, *Core J2EE Patterns*, Prentice-Hall, 2001.

[**Cunningham**] Ward Cunningham, "EPISODES: A Pattern Language of Competitive Development." In *Pattern Languages of Program Design 2*, Vlissides, Coplien, and Kerth, Addison-Wesley, 1996, pp. 371-388.

[**Douglass**] Bruce Powel Douglass, *Real-Time UML*, Addison-Wesley, 1999.

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

[**Fowler, new methodology**] Martin Fowler, "The New Methodology," http://martinfowler.com/articles/newMethodology.html.

[**Fowler and Foemmel**] Martin Fowler and Matthew Foemmel, "Continuous Integration," http://martinfowler.com/articles/continuousIntegration.html

[**Fowler, P of EAA**] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

[**Fowler, refactoring**] Martin Fowler, *Refactoring: Improving the Design of Existing Programs*, Addison-Wesley, 1999.

[**Gang of Four**] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[**Highsmith**] Jim Highsmith, *Agile Software Development Ecosystems*, Addison-Wesley, 2002.

[**Hohmann**] Luke Hohmann, *Beyond Software Architecture*, Addison-Wesley, 2003.

[**Jacobson, OOSE**] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

[**Jacobson, UP**] Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, *The Object Advantage: Business Process Reengineering with Object Technology*, Addison-Wesley, 1995.

[**Kerth**] Norm Kerth, *Project Retrospectives*, Dorset House, 2001.

[**Kleppe et al.**] Anneke Kleppe, Jos Warmer, and Wim Bast, *MDA Explained*, Addison-Wesley, 2003.

[**Kruchten**] Philippe Kruchten, *The Rational Unified Process: An Introduction*, Addison-Wesley, 1999.

[**Larman**] Craig Larman, *Applying UML and Patterns*, 2d ed., Prentice-Hall, 2001.

[**Martin**] Robert Cecil Martin, *The Principles, Patterns, and Practices of Agile Software Development*, Prentice-Hall, 2003.

[**McConnell**] Steve McConnell, *Rapid Development: Taming Wild Software Schedules*, Microsoft Press, 1996.

[**Mellor and Balcer**] Steve Mellor and Marc Balcer, *Executable UML*, Addison-Wesley, 2002.

[**Meyer**] Bertrand Meyer, *Object-Oriented Software Construction*. Prentice-Hall, 2000.

[**Odell**] James Martin and James J. Odell, *Object-Oriented Methods: A Foundation (UML Edition)*, Prentice-Hall, 1998.

[**Pont**] Michael Pont, *Patterns for Time-Triggered Embedded Systems*, Addison-Wesley, 2001.

[**POSA1**] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter

Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996.

[**POSA2**] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Archtecture Volume 2: Patterns for Concurrent and Networked Objects*, Wiley, 2000.

[**Rumbaugh, insights**] James Rumbaugh, *OMT Insights*, SIGS Books, 1996.

[**Rumbaugh, OMT**] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenzen, *Object-Oriented Modeling and Design*, Prentice-Hall,1991.

[**Rumbaugh, UML Reference**] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[**Shlaer and Mellor, data**] Sally Shlaer and Stephen J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press, 1989.

[**Shlaer and Mellor,states**] Sally Shlaer and Stephen J. Mellor, *Object Lifecycles: Modeling the World in States*. Yourdon Press, 1991.

[**Warmer and Kleppe**] Jos Warmer and Anneke Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1998.

[**Wirfs-Brock**] Rebecca Wirfs-Brock and Alan McKean, *Object Design: Roles Responsibilities and Collaborations*. Prentice-Hall, 2003.

图索引

图 1.1 UML 元模型的片段 …… 13
图 1.2 UML 图型分类 …… 16
图 1.3 唤醒部分的非形式屏幕流图 …… 20
图 3.1 简单类图 …… 45
图 3.2 订单特性的属性表示 … 47
图 3.3 订单特性的关联表示 … 48
图 3.4 双向关联 …… 53
图 3.5 关联的动词短语命名 … 54
图 3.6 用作图成分上的注释的注文 …… 60
图 3.7 依赖一例 …… 61
图 4.1 集中式控制顺序图 …… 69
图 4.2 分布式控制顺序图 …… 71
图 4.3 参加者的创建与删除 … 74
图 4.4 交互架构 …… 75
图 4.5 对控制基理的较老习惯 …… 78
图 4.6 CRC 卡样例 …… 81
图 5.1 在类图中示明职责 …… 85
图 5.2 静态图示法 …… 86
图 5.3 聚合 …… 86
图 5.4 组合 …… 87
图 5.5 时间区间中的导出属性 …… 88
图 5.6 接口和抽象类的 Java 例子 …… 90
图 5.7 球形-托座图示法 …… 91
图 5.8 带连珠的较老依赖 …… 92
图 5.9 顺序图中多态的连珠示明法 …… 92
图 5.10 受限关联 …… 95
图 5.11 多重分类 …… 98
图 5.12 关联类 …… 100
图 5.13 关联类的全类提升 … 100
图 5.14 关联类的微妙(也许角色不应为关联类) …… 101
图 5.15 用于时态关系的类 … 102
图 5.16 用于关联的基词《temporal》…… 102
图 5.17 模板类 …… 104
图 5.18 束元(形式 1) …… 104

图 5.19 束元(形式 2) …… 104
图 5.20 枚举 …… 105
图 5.21 主动类 …… 106
图 5.22 带消息的类 …… 107
图 6.1 当事人类组合结构的类图 …… 108
图 6.2 当事人类样例实例的对象图 …… 109
图 7.1 包在图中的示明方式 … 111
图 7.2 企业应用包图 …… 114
图 7.3 分隔图 7.2 为两面 …… 115
图 7.4 由别的包实现的包 …… 116
图 7.5 在客户包中定义所需接口 …… 117
图 8.1 部署图例 …… 120
图 9.1 用案一例的正文 …… 124
图 9.2 用案图 …… 127
图 10.1 简单状态机图 …… 132
图 10.2 以一正文域的打字状态示明的内部事件 …… 134
图 10.3 带活动的状态 …… 135
图 10.4 带嵌套子态的超态 … 136
图 10.5 并发正交态 …… 137
图 10.6 处理图 10.1 中状态转换的嵌套开关方法(用 C# 语言实现) …… 139
图 10.7 图 10.1 的状态模式实现 …… 140
图 11.1 简单活动图 …… 144
图 11.2 辅助活动图 …… 146
图 11.3 图 11.1 的活动修改成实施图 11.2 中的活动…147
图 11.4 活动图上的分划 …… 148
图 11.5 活动图上的信号 …… 150
图 11.6 发送信号与接收信号 …… 150
图 11.7 示明边的四种方式 … 152
图 11.8 流上的转换 …… 153
图 11.9 展开区域 …… 154
图 11.10 展开区域中单个动作的简略表示 …… 155
图 11.11 活动内的流终 …… 156
图 11.12 汇合指明 …… 157
图 12.1 集中式控制通信图 … 161
图 12.2 具嵌套十进制数编号的通信图 …… 162
图 13.1 示明 TV 观众及其接口的两种方式 …… 164
图 13.2 构件的内部观(Jim Rumbaugh 推荐之例)…165
图 13.3 具有多个港口的构件 …… 166
图 14.1 构件图示法 …… 167
图 14.2 构件图例 …… 168
图 15.1 带角色类图的协作 … 171
图 15.2 拍卖协作顺序图 …… 171

图 15.3 协作出现 …… 172
图 15.4 示明 JUnit (junit.org) 中模式使用的非标准方式 …… 172
图 16.1 交互概观图 …… 175
图 17.1 用线段示明状态的定时图 …… 177
图 17.2 用区域示明状态的定时图 …… 177

汉英对照术语索引

A

案况集(scenario sets) 123

B

Beck, Kent, CRC 卡(Beck, Kent, CRC crads) 80～82

Booch, Grady, UML 历史(Booch, Grady, UML history) 9～12

版本修订(UML)(revisions by versions (UML))

从 0.8 到 2.0,一般历史(from 0.8 through 2.0, general history) 179～180

从 1.0 到 1.1(from 1.0 to 1.1) 182～185

从 1.2 到 1.3(from 1.2 to 1.3) 185～187

从 1.3 到 1.4(from 1.3 to 1.4) 187～188

从 1.4 到 1.5(from 1.4 to 1.5) 188

从 1.x 直至 2.0(from1.x through 2.0) 188～189

包(packages)

定义(definitions) 110

共同封闭与复用原则(Common Closure and Reuse Principles) 112

面(aspects) 115

名空间(namespaces) 110

全受限名(fully qualified names) 111

实现(implementing) 116

依赖(dependencies) 112～114

包含关系(include relationships) 125

包图(package diagrams) 15～16

UML版本变动(UML version changes) 187～188

基本原理(basics) 110～112

设计(design) 38

使用时间(times to use) 117

文档(documentation) 40～41

资料源(resources) 118

保证(guarantees) 126

本型(archetypes) 6

边(edges) 151～152
UML 用作编程语言(programming languages,UML as) 2,4～5
MDA(模型驱动体系结构)(MDA (Model Driven Architecture)) 5～7
价值(value) 7
逆向工程(reverse engineering) 2～5
正向工程(forward engineering) 2～5
标记,迭代(markers,iteration) 76
标准用法(standard use) 18
表象类(presentation classes) 60
并发状态(concurrent states) 136～138
不变式(invariants) 64～65
部署图(deployment diagrams) 15～16
结点(nodes) 119～120
设备(devices) 119～121
设计(design) 38～39
使用时间(times to use) 121
执行环境(execution environments) 119～120
制品(artifacts) 119～121

C

CASE(计算机辅助软件工程)工具(CASE (computer-aided software engineering) tools) 4
UML 历史(UML history) 9～12
Coad, Peter, UML 历史(Coad, Peter, UML history) 9～10
Cockburn, Alistair 用案(Cockburn, Alistair,use cases) 130
CORBA(公共对象请求代理体系结构)(CORBA (Common Object Request Broker Architecture)) 2
CRC(类-职责-协作)卡(CRC (Class-Responsibility-Collaboration) cards) 80～82
Cunniugham, Ward, CRC 卡(Cunniugham, Ward,CRC cards) 80～82
参加者,顺序图(participants, sequence diagrams) 68～74
参与者(actors) 123,170～171
仓库系统,与平台无关的模型及平台特定的模型(warehousing systems,platform indendent model and platform) 5
操作与方法之对比(operations versus methods) 5
操作符,交互架构(operators, interaction frames) 57～58
侧图(profiles) 84～85
UML版本变动(UML version changes) 187
超态(superstates) 136
抽象类,类和接口的关系(abstract classes, relationship of classes to interfaces) 89～93

初始结点动作(initial node actions) 143～144
初始伪态(initial pseudostate) 131
传递关系(transitive relationships) 61
词典,见受限关联(dictionaries, *See* Qualified associations)

D

DSDM(动态系统开发方法)(DSDM (Dynamic Systems Development Method)) 31
代理对象(proxy objects) 36
单向关联(unidirectional associations) 53
单一分类(single classification) 97～98
实现类(implementation class) 182～183
单值属性(single-valued attributes) 49
导出特性,类图(derived properties, class diagrams) 87～88
导航箭号(navigability arrows) 54～56
迭代(iterations) 24～28
时间框定(timeboxing) 27
迭代标记(iteration markers) 76
迭代回顾(iteration retrospective) 34
迭代开发过程(iterative development process) 24～28
定时图(timing diagrams) 15～16
基本原理(basics) 176～178
动态分类(dynamic classifications) 99
数据类型(data types) 182～183
动作(actions)
UML版本变动(UML version changes) 188
展开区域(expansion regions) 154～155
冻结特征(frozen property) 93,184
断言(assertions) 64～66
子类构作(subclassing) 65
对象图(object diagrams) 15～16
使用时间(times to use) 109
对象约束语言(OCL(Object Constraint Language)) 64
多值属性(multivalued attributes) 49
多重分类(multiple classifications) 97～99
数据类型(data types) 182～183

E

Eiffel 编程语言(Eiffel programming language) 64

F

FDD(特征驱动开发)(FDD(Feature Driven Development)) 31
发布(releases) 25
泛化(generalizations) 44～45
UML 版本变动(UML version

changes) 185
与分类之对比(versus classifications) 96～97
集(sets) 98～99
类特征(class properties) 56
方法(methods)
动作的实现 (implementation of actions) 146～147
与操作之对比(versus operations) 57～58
非循环依赖原则(Acyclic Dependency Principle) 113
分岔(forks) 143～144
UML 版本变动(UML version changes) 186
分划,活动图(partitions, activity diagrams) 148～149
分类(classifications)
动态与多重(dynamic and multiple) 97～99
与泛化之对比(versus generalization) 96～97
实现类(implementation classes) 182～183
数据类型(data types) 182～183
分析模式(analysis patterns) 184
分支(branches) 145
风筝级用案(kite-level use cases) 128
复合结构图(composite structure diagrams) 15～16
基本原理(basics) 164～166
使用时间(times to use) 166

G

改态操作(modifiers) 57
隔开接口(Separated Interface) 117
工程,正向(engineering, forward)
UML 用作编程语言(UML as programming languages) 5
UML 用作草图绘制语言(UML as sketches) 2
UML 用作蓝图绘制语言(UML as blueprints) 3～4
公共对象请求代理体系结构(CORBA)标准(Common Object Request Broker Architecture(CORBA) standards) 2
公用成分(public elements) 106
供应方/客户方(suppliers/clients) 60
共同封闭与复用原则(Common Closure and Reuse Principles) 112
构件图(component diagrams) 15～16
基本原理(basics) 167～169
使用时间(times to use) 169
构作,RUP 项目(construction, RUP projects) 33
关联,类特性(association, class

properties) 47～48
单向(unidirectional) 53
受限(qualified) 95～96
双向(bidirectional) 53～55
永恒还是冻结(immutability versus frozen) 184
关联类(association classes) 99～103
关联数组,见受限关联(associative arrays, *See* Qualified associations)
关系(relationships)
包含(include) 125～127
抽象类与接口(abstract classes to interfaces) 89～93
传递(transitive) 61
时态(temporal) 102
规范用法(normative use) 18

H

海级用案(sea-level use cases) 128
合并(merges) 145
恒态操作(queries) 57
后置条件,按契约设计(post-conditions, design by contract) 64
回顾(retrospectives)
迭代(iteration) 34
项目(project) 34
汇合(joins) 145
UML 版本变动(UML version changes) 186～187
指明(specifications) 156～157
活动,退出(activities, exit) 134
活动图(activity diagrams) 15～16
UML 版本变动(UML version changes) 186～187,190
边(edges) 151～152
动作,展开区域(actions, expansion regions) 154～155
动作的分解(decomposing actions) 146～147
分划(partitions) 148～149
汇合(joins) 145
指明(specifications) 156～157
基本原理(basics) 143～146
流(flows) 151～152
佩特里网(Petri nets) 159
流终(flow final) 155～156
权标(tokens) 151
使用时间(times to use) 158
饰针(pins) 152～153
信号(signals) 149～150
需求分析(requirements analysis) 37～38
转换(transformations) 152～153
资料源(resources) 158～159
活动状态(activity state) 134～135
获取方法(getting methods) 57

J

Jacobson,Ivar(Jacobson,Ivar)
UML 历史(UML history) 10,12
用案(use cases) 130
基础消息(found messages) 70
基词,类图(keywords, class diagrams) 62～63,83～84
极端程序设计(XP)(Extreme Programming(XP))
技术实践(techinal practices) 28
敏捷开发过程(agile development process) 31
资料源(resources) 42～43
集成,连续(integration,continuous) 28
计划制订,适应性还是预见性(planning, adaptive versus predictive) 28～31
计算机辅助软件工程(CASE)工具(computer-aided software engineering (CASE) tools) 4
UML 历史(UML history) 9～12
监护(guards) 76
渐进开发过程,见迭代开发过程(incremental development process, *See* Iterative development process)
交互概观图(interactive overview diagrams) 15～16
交互架构(interaction frames)
操作符(operators) 76～77
循环与条件(loops and conditionals) 74～76
交互图(interaction diagrams)
CRC 卡(CRC cards) 80～82
参加者(participants) 68～72
基本原理(basics) 68～72,174～175
设计(design) 38～39
使用时间(times to use) 79～80,174～175,178
顺序图(sequence diagrams) 68～72
同步消息与异步消息(synchronous and asynchronous messages) 79
循环与条件(loops and conditionals) 74～78
角色,见参与者(Roles, *See* Actors)
阶段提交开发过程(staged delivery development process) 26
接口(interfaces) 84
和类的关系(relationship to classes) 89～93
结点(nodes) 119～120
结构改组(refactoring) 28
进入活动(entry activities) 134
进行活动(do-activities) 135
晶体,敏捷开发过程(crystal, agile development process) 31
静态分类(static classifications)

与动态分类之对比(versus dynamic classifications) 99
实现类(implementation classes) 182～183
聚合(aggregation) 86～87

K

开发过程(development processes)
DSDM(动态系统开发方法)(DSDM (dynamic systems development method)) 31
FDD(特征驱动开发)(FDD(feature driven development)) 31
Rational 统一过程(RUP)(Rational Unified Process(RUP)) 32～33
迭代(iterative) 24～28
过程适配项目(fitting processes to projects) 33～35
极端程序设计(XP)(Extreme Programming(XP)) 28, 31～32, 42～43
阶段提交(staged delivery) 26
敏捷(agile) 31
敏捷软件开发宣言(Manifesto of Agile Software Development) 31
瀑布(waterfall) 24～28
轻量级(lightweight) 31
选取(selecting) 42
资料源(resources) 42～43
开发例案(development cases) 32
可复用本型(reusable archetypes) 6
可见性(visibility) 106～107
可执行 UML(executable UML) 5～6
客户方/供应方(clients / suppliers) 60
扩张(extensions) 124

L

Loomis, Mary, UML 历史(Loomis, Mary, UML history) 11
类, 见类特性(classes, *See* Class properties)
表象(presentation) 60
抽象(abstract) 89～93
导出(derivation) 103～104
动态数据类型(dynamic data types) 182～183
泛化(generalizations) 44～45
关联(association) 99～103
静态分类还是动态分类(static versus dynamic classifications) 99
静态数据类型(static data types) 182～183
类-职责-协作(CRC)卡(Class-Responsibility-Collaboration (CRC) cards) 80～82
模板(参数化)(template

(parameterized)) 103～104
实现(implementation) 182～183
属性(attributes) 85～86
子类构作(subclassing) 65
类的导出(derivation of classes) 103
类的静态操作(static operations of classes) 85～86
类的职责(responsibilities of classes) 85
类特性,参见类(class properties, *See also* Classes)
程序解释(program interpretation) 50～53
导出(derived) 87～88
冻结(frozen) 93
泛化(generalization) 58～59
关联(associations) 47～48
关联,受限(associations, qualified) 95～96
关联,双向关联(associations, bidirectional associations) 53～56
关联,永恒还是冻结(associations, immutability versus frozen) 184
基本原理(basics) 44～48
只读(read-only) 90
重数(multiplicity) 48～49
属性(attributes) 46～47
类图(class diagrams) 12,15～16
UML 版本间之变动(UML version changes) 189～190
操作(operations) 56～58
抽象类(abstract classes) 89～93
与对象图之对比(versus object diagrams) 108～109
泛化(generalization) 58～59,96～97
分类(classifications) 96～97
动态与多重(dynamic and multiple) 97～99
关联类(association classes) 99～103
基词(keywords) 83～85
静态操作与静态属性(static operations and attributes) 85～86
聚合与组合(aggregation and composition) 86～87
可见性(visibility) 106～107
模板(参数化)类(template (parameterized) classes) 103～104
设计(design) 38
使用时间(times to use) 66
特性,见类特性(properties, *See* Class properties)
文档(documentation) 40～41
消息(messages) 107
需求分析(requirements analysis) 37～38
依赖(dependencies) 60～64
约束规则(constraint rules) 63～64

值对象(value objects) 93～95
职责(responsibilities) 85
指引对象(reference objects) 93～95
主动类(active classes) 105～106
注释(comments) 59～60
注文(notes) 59～60
着手使用 UML(starting with UML) 21
资料源(resources) 67
类图中的注释(comments in class diagrams) 59～60
类型,见数据类型(Types, *See* Data types)
类-职责-协作(CRC)卡(Class-Responsibility-Collaboration (CRC) cards) 80～82
类特性(properties of classes)
程序解释(program interpretations) 50～53
导出(derived) 87～88
冻结(frozen) 93
关联(associations) 47～48
受限(qualified) 95～96
双向关联(bidirectional associations) 53～56
基本原理(basics) 44～48
只读(read-only) 93
重数(multiplicity) 48～49
属性(attributes) 46～47
历史伪态(history pseudostate) 137
连续集成(continuous integration) 28
连珠图示法(lollipop notation) 91～93
领域对象(domain objects) 60
流(flows) 151～152
流终(flow final) 155～156
佩特里网(Petri nets) 159
螺旋开发过程,见迭代开发过程(spiral development process, *See* Iterative development process)

M

MDA,模型驱动体系结构(MDA(Model Driven Architecture)) 5～7
Mellor,Steve(Mellor.Steve)
UML 历史(UML history) 10
可执行 UML(executable UML) 6
Meyer, Bertrand, 按契约设计(Meyer, Bertrand,design by contract) 64
枚举(enumerations) 105
描述性规则, UML(descriptive rules, UML) 17
敏捷开发过程(agile development processes) 31
资料源(resources) 42～43
敏捷软件开发宣言(Manifesto of Agile Software Development) 31

名空间(namespaces) 110
模式(patterns)
定义(definition) 35～36
隔开接口(separated interface) 117
使用(using) 173
状态(state) 136～142
模型编译程序(model compilers) 6

N

内部活动，进入与退出(internal activities, entry and exit) 134
内部活动，退出活动(internal activities, exit activities) 134
逆向工程(reverse engineering)
UML 用作编程语言(UML as programming languages) 4～5
UML 用作草图绘制语言(UML as sketches) 2～3
UML 用作蓝图绘制语言(UML as blueprints) 3,8

O

Odell, Jim, UML 历史(Odell, Jim, UML history) 10～11
OMG(对象管理组)(OMG(Object Management Group)) 1～2
MDA(模型驱动体系结构)(MDA (Model Driven Artichitecture)) 5～7
UML 的控制管理(control of UML) 2
UML 版本修订(revisions to UML versions) 179～180
UML 历史(UML history) 9～12
OO(面向对象)编程(OO(object-oriented) programming) 1
风范转移(paradigm shift) 72

P

PIM(与平台无关的模型)(PIM(platform independent model)) 5
PSM(平台特定的模型)(PSM(platform specific model)) 5
佩特里网(面向流的技术)(Petri nets (flow-oriented technique)) 159
喷泉开发过程，见迭代开发过程(Jacuzzi development process, *See* Iterative development process)
平台特定的模型(PSM)(platform specific model(PSM)) 5
瀑布开发过程(waterfall development process) 24～28

Q

前置条件(pre-conditions)
按契约设计(design by contract) 64
用案(use cases) 130
强行属性(mandatory attribute) 49
轻量级开发过程(lightweight

development processes) 31
情节,见用案的特征(stories, *See* Features of use cases)
球座图示法(ball and socket notation) 91~93
取别名(aliasing) 94
全受限名(fully qualified names) 111
权标(tokens) 151

R

Rational 统一过程(RUP)(Rational Unified Process(RUP)) 23,32~33
阶段(phases) 32
开发例案(development cases) 32
资料源(resources) 42~43
Rebecca Wirfs-Brock,UML 历史(Rebecca Wirfs-Brock,UML history) 10
Rumbaugh,Jim(Rumbaugh,Jim)
UML 历史(UML history) 10~12
复合结构(composite structure) 165
聚合(aggregation) 86
RUP(Rational 统一过程)(RUP(Rational Unified Process)) 23,32~33
阶段(phases) 32
开发例案(development cases) 32
资料源(resources) 42~43
任选属性(optional attributes) 49
软件开发过程,见开发过程(software development processes, *See* development processes)
软件视面,UML(software perspectives, UML) 7~8

S

Shlaer,Sally,UML 历史(Shlaer, Sally, UML history) 10
Smalltalk(Smalltalk) 7
三友(Three Amigos) 12
散列,见受限关联(hashes, *See* Qualified associations)
设备(devices) 119~120
设计(design) 38
时间框定(timeboxing) 27
时间信号(time signals) 149
时态关系(temporal relationships) 102
实例规约(instance specifications) 109
实现类,数据类型(implementation classes,data types) 182~183
事件开关(event switches) 137
饰针(pins) 152~153
受护成分(protected elements) 106
受限关联(qualified associations) 95~96
束元(bound elements) 104
数据蝌蚪(data tadpoles) 77
数据类型(data types) 95
动态分类与多重分类(dynamic and

multiple classifications) 182～183
实现类(implementation classes) 182～183
属性(attributes)
类(classes) 84～86
类特性(class properties) 46～48
强行(mandatory) 49
双向工具(round-trip tools) 4
双向关联(bidirectional associations) 53～55
顺序图(sequence diagrams) 15～16
CRC 卡(CRC cards) 80～82
UML 版本变动(UML version changes) 189
参加者(participants) 68～74
回送(返回)(returns) 184
基本原理(basics) 68～72
集中式控制与分布式控制(centralized and distributed control) 69～72
交互图(interation diagrams) 68
使用时间(times to use) 79～80
同步消息与异步消息(synchronous and asynchronous messages) 79
协作(collaborations) 170～172
循环与条件(loops and conditionals) 74～78
着手使用 UML(starting with UML) 21
顺序图绘制的集中式控制(centralized control of sequence diagramming) 69～72
顺序图制作的分布式控制(distributed control of sequence diagramming) 69～72
私用成分(private elements) 106
四人组(Gang of Four) 36～37
搜索状态(searching state) 135

T

特性的重数(multiplicity of properties) 48～49
条件(conditionals) 74～78
选定与合并(decisions and merges) 145
通信图(communication diagrams) 15～16
基本原理(basics) 160～162
使用时间(times to use) 162～163
同步消息(synchronous messages) 79
统一方法文档(Unified Method documentation) 10
统一建模语言，见 UML(Unified Modelling Language, *See* UML)
图(diagrams)
包(package) 15～16
UML 版本变动(UML version

changes) 187～188
基本原理(basics) 110～112
设计(design) 38
使用时间(times to use) 117
文档(documentation) 40～41
资料源(resources) 118
部署(deployment) 15～16
结点(nodes) 119～120
设备(devices) 119～120
设计(design) 38
使用时间(times to use) 121
执行环境(execution environments) 119～120
制品(artifacts) 114
定时(timing) 15～16
基本原理(basics) 176～178
对象(object) 15～16
使用时间(times to use) 109
分类(classifications) 16
复合结构(composite structure) 15～16
基本原理(basics) 164～166
使用时间(times to use) 166
构件(component) 15～16,167～169
观点(viewpoints) 8
活动(activity) 15～16
UML 版本变动(UML version changes) 186～187,190
边(edges) 151～152
动作,展开区域(actions,expansion regions) 154～155
动作的分解(decomposing actions) 146～147
分划(partitions) 148～149
汇合(joins) 145
汇合,指明(joins,specifications) 156～157
基本原理(basics) 143～146
流(flows) 151～152
流,佩特里网(flows,Petri nets) 159
流终(flow final) 155～156
权标(tokens) 151
使用时间(times to use) 158
饰针(pins) 152～153
信号(signals) 149～150
需求分析(requirements analysis) 37～38
转换(transformations) 152～153
资料源(resources) 158～159
基本原理(basics) 14～16
交互(interaction)
CRC 卡(CRC cards) 80～82
参加者(participants) 68～74
基本原理(basics) 68～72,174～175
设计(design) 38
使用时间(times to use) 174～175

顺序图(sequence diagrams) 68～72
同步与异步(synchronous and asynchronous messages) 79
循环与条件(loops and conditionals) 74～78
交互概观(interavtive overview) 15～16
类(class) 15～16
UML 版本变动(UML version changes) 189～190
操作(operations) 56～58
抽象类(abstract classes) 89～93
与对象图之对比(versus object diagrams) 108～109
泛化(generalizations) 58～59,94～95
分类(classifications) 96～97
分类,动态与多重(classification, dynamic and multiple) 97～99
关联类(association classes) 99～103
基词(keywords) 62,83～85
静态操作与静态属性(static operations and attributes) 85～86
聚合与组合(aggregation and composition) 86～87
着手使用 UML(starting with UML) 21
可见性(visibility) 106～107
模板(参数化)类(template (parameterized) classes) 103～104
设计(design) 38～39
使用时间(times to use) 66
特性(见类特性)(properties (*See* Class properties))
文档(documentation) 40～41
消息(messages) 107
需求分析(requirement analysis) 37～38
依赖(dependencies) 60～63
约束规则(constraint rules) 63～64
值对象(value objects) 93～95
职责(responsibilities) 85
指引对象(reference objects) 93～95
主动类(active classes) 105～106
注释(comments) 59～60
注文(notes) 59～60
资料源(resources) 67
类型(types) 16
类型,UML 版本变动(types, UML version changes) 188
起点(starting point) 21
缺点(shortcomings) 19～21
顺序(sequence) 15～16
CRC 卡(CRC cards) 80～82

UML 版本变动(UML version changes) 189
参加者(participants) 68～74
回送(返回)(returns) 184
基本原理(basics) 68～72
集中式控制与分布式控制(centralized and distributed control) 69～72
交互图(interaction diagrams) 68～72
使用时间(times to use) 79～80
同步消息与异步消息(synchronous and asynchronous messages) 79
协作(collaborations) 171
循环与条件(loops and conditionals) 74～78
着手使用 UML(starting with UML) 21
通信(communication) 15～16,160～163
用案(use case)
基本原理(basics) 126～127
需求分析(requirement analysis) 37～38
状态机(state machine) 15～16
UML 版本变动(UML version changes) 190
并发状态(concurrent states) 136～138
超态(superstates) 136
初始伪态(initial pseudostate) 131～132
活动状态(activity status) 134～135
基本原理(basics) 131～133
内部活动(internal activities) 134
实现(implementing) 138～140
使用时间(times to use) 141
需求分析(requirement analysis) 37～38
转接(transitions) 132
资料源(resources) 141～142
图示法(notation)
定义(definitions) 12～14
连珠(lollipop) 91～93
球座(ball and socket) 91～93
图示建模语言(graphical modeling languages) 1
退出活动(exit activities) 134

U

UML(UML)
标准,合法使用还是非合法使用(standards,legal versus illegal use) 17～19
定义(definition) 1～2
含义(meaning) 19

历史(history) 9～12
描述性规则(descriptive rules) 17
软件视面与概念视面(software and conceptual perspectives) 7
适配过程(fitting into processes) 35
习惯用法(conventional use) 18
指定性规则(prescriptive rules) 17
资料源(resources) 22
UML 版本修订(UML revisions by versions)
从 0.8 到 2.0,一般历史(from 0.8 through 2.0,general history) 179～180
从 1.0 到 1.1(from 1.0 to 1.1) 182～185
从 1.2 到 1.3(from 1.2 to 1.3) 185～187
从 1.3 到 1.4(from 1.3 to 1.4) 187～188
从 1.4 到 1.5(from 1.4 to 1.5) 188
从 1.x 直至 2.0(from 1.x through 2.0) 188～189
UML 的概念视图(conceptual perspectives of UML) 7
UML 精粹,书版与相应 UML 版本(UML distilled, book editions and corresponding UML) 181～182
UML 图,见图与特定图形(UML diagrams, *See* Diagrams and specific diagram types)
UML 用作编程语言(UML as programming language) 2,4～5
MDA(模型驱动体系结构)(MDA (Model Driven Architecture)) 5
价值(value) 7
逆向工程(reverse engineering) 2～5
正向工程(forward engineering) 2～5
UML 用作草图绘制语言(UML as sketches) 2,3,9
逆向工程(reverse engineering) 2～5
正向工程(forward engineering) 2～5
UML 用作蓝图绘制语言(UML as blueprints) 2,3
逆向工程(reverse engineering) 2～5,8
正向工程(forward engineering) 2～5,8
UP(统一过程),见 RUP(UP (Unified Process), *See* RUP)

W

伪消息(pseudomessages) 77～78
文档(documentation) 40～41
稳定抽象原则(Stable Abstractions Principle) 113
稳定依赖原则(Stable Dependencies Principle) 113

X

XP(极端程序设计)(XP(Extreme Programming))
技术实践(technical practices) 28
敏捷开发过程(agile development process) 31～32
资料源(resources) 42～43
习惯用法(conventional use) 18
系统用案(system use cases) 128
项目回顾(project retrospective) 34
消息(messages) 107
基础(found) 70
类图(class diagrams) 107
伪消息(pseudomessages) 77～78
异步与同步(asynchronous and synchronous) 79
协作(collaborations) 170～172
角色(roles) 170～171
使用时间(times to use) 173
顺序图(sequence diagrams) 171
协作图,见通信图(collaboration diagram, *See* Communication diagrams)
信号(signals) 149～150
虚包类(facades) 112
需求分析(requirements analysis) 37～38
需求折腾(requirements churn) 29
喧嚣(scrum) 31
选定(decisions) 145
循环(loops) 74

Y

衍型(stereotypes) 84
演化开发过程,见迭代开发过程(evolutionary development process, *See* Iterative development process)
业务用案(business use cases) 128
依赖(dependencies) 60～64
UML版本变动(UML version changes) 185
包(packages) 112～114
基词(keywords) 62
资料源(resources) 67
仪节,敏捷过程(ceremony, agile processes) 31
遗产代码(legacy code) 41～42
异步消息(asynchronous messages) 79
引发装置(trigger) 126
映射,见受限关联(Maps, *See* Qualified association)
泳道,见分划(swim lanes, *See* Partitions)
用案(use cases)
MSS(主成功案况)(MSS(main success scenairo)) 124
UML 版本变动(UML version changes) 185～186

案况集(scenario sets) 123
包含关系(include relationships) 125
参与者(actors) 123
级别(levels) 128
扩张(extensions) 124～126
使用时间(times to use) 129
特征(features) 128～129
业务(business) 128
资料源(resources) 130
用案的特征(features of use cases) 128～129
用案图(use case diagrams)
基本原理(basics) 126～127
需求分析(requirement analysis) 37～38
用户情节,见用案的特征(user stories, *See* Features of use cases)
用户指南(User Guide) 141
鱼级用案(fish-level use cases) 128
预见性计划制订与适应性计划制订之比较(predictive planning, versus adaptive planning) 28～31
元模型(meta-models)
UML 版本变动(UML version changes) 188
定义(definitions) 12～14
约束(constraints) 145～146
规则(rules) 63～64
完整/非完整(complete/incomplete) 183

Z

展开区域(expansion regions) 154～155
正向工程(forward engineering)
UML 用作编程语言(UML as programming languages) 2～5
UML 用作草图绘制语言(UML as sketches) 2
UML 用作蓝图绘制语言(UML as blueprints) 3
执行环境(execution environments) 120
值对象(value objects) 93～95
只读特性(read-only property) 93
指定性规则(prescriptive rules) 17
指引对象(reference objects) 93～95
制品(artifacts) 119
UML 版本变动(UML version changes) 187
置换性(substitutability) 58
置送方法(setting methods) 57
主成功案况(main success scenario) 124
主动类(active classes) 105～106
转换(transformations) 152～153
转接(transitions) 33,132,136
状态(state) 139
状态表(state tables) 138～140

状态机图(state machine diagrams) 15～16
UML 版本变动(UML version changes) 190
并发状态(concurrent states) 136～138
超态(superstates) 136
初始伪态(initial pseudostate) 131～132
活动状态(activity status) 134～135
基本原理(basics) 131～133
内部活动(internal activities) 134
实现(implementing) 138～140
使用时间(times to use) 141
需求分析(requirement analysis) 37～38
转接(transitions) 131～133,137
资料源(resources) 141～142
状态图,见状态机图(state diagrams, *See* State machine diagrams)
子活动(subactivities) 146～147
子类构作(subclassing) 65
断言(assertions) 64～65
子类型(subtypes) 59
自动回归测试(automated regression tests) 27
组成良好的 UML(well formed UML)
定义(definition) 19
合法的 UML(legal UML) 17～19
组合(composition) 86～87
UML 版本间之变动(changes between UML versions) 183～184

英汉对照术语索引

A

Abstract classes, relationship of classes to interfaces(抽象类,类和接口的关系) 89～93

Actions(动作)

expansion regions(展开区域) 154～155

UML version changes(UML 版本变动) 188

Active classes(主动类) 105～106

Activities, exit(活动,退出) 134

Activity diagrams(活动图) 15～16

actions, expansion regions(动作,展开区域) 154～155

basics(基本原理) 143～146

decomposing actions(动作的分解) 146～147

edges(边) 151～152

flow final(流终) 155～156

flows(流) 151～152

Petri nets(佩特里网) 159

joins(汇合) 145

specifications(指明) 156～157

partitions(分划) 148～149

pins(饰针) 152～153

requirements analysis(需求分析) 37～38

resources(资料源) 158～159

signals(信号) 149～150

times to use(使用时间) 158

tokens(权标) 151

transformations(转换) 152～153

UML version changes(UML 版本变动) 186～187,190

Activity state(活动状态) 134～135

Actors(参与者) 123,170～171

Acyclic dependency principle(非循环依赖原则) 113

Aggregation(聚合) 86～87

Agile development processes(敏捷开发过程) 31

resources(资料源) 42～43

Aliasing(取别名) 94
Analysis patterns(分析模式) 184
Archetypes(本型) 6
Artifacts(制品) 119
  UML version changes(UML 版本变动) 187
Assertions(断言) 64～66
  subclassing(子类构作) 65
Association classes(关联类) 99～103
Association, class properties(关联,类特性) 47～48
  bidirectional(双向) 53～55
  immutability versus frozen(永恒还是冻结) 184
  qualified(受限) 95～96
  unidirectional(单向) 53
Associative arrays, *See* qualified associations(关联数组,见受限关联)
Asynchronous messages(异步消息) 79
Attributes(属性)
  class properties(类特性) 46～48
  classes(类) 84～86
  mandatory(强行) 49
Automatedregression tests(自动回归测试) 27

B

Ball and socket notation(球座图示法) 91～93
Beck, Kent, CRC cards(Beck, Kent, CRC 卡) 80～82
Bidirectional associations(双向关联) 53～55
Blueprints, UML as(UML 用作蓝图语言)
  forward engineering(正向工程) 3～4
  reverse engineering(逆向工程) 4,8
Booch, Grady, UML history(Booch, Grady,UML 历史) 9～12
Bound elements(束元) 104
Branches(分支) 145
Business use cases(业务用案) 128

C

CASE (computer-aided software engineering) tools(CASE (计算机辅助软件工程)工具) 4
  UML history(UML 历史) 9～12
Centralized control of sequence diagramming(顺序图绘制的集中式控制) 69～72
Ceremony,agile processes(仪节,敏捷过程) 31
Class diagrams(类图) 12,15～16
  abstract classes(抽象类) 89～93
  active classes(主动类) 105～106

aggregation and composition(聚合与组合) 86～87
association classes(关联类) 99～103
classifications(分类) 96～97
dynamic and multiple(动态与多重) 97～99
comments(注释) 59～60
constraint rules(约束规则) 63～64
dependencies(依赖) 60～64
design(设计) 38
documentation(文档) 40～41
generalization(泛化) 58～59,96～97
keywords(基词) 83～85
messages(消息) 107
notes(注文) 59～60
operations(操作) 56～58
properties(*See* Class properties)(特性,见类特性)
reference objects(指引对象) 93～95
requirements analysis(需求分析) 37～38
resources(资料源) 67
responsibilities(职责) 85
starting with UML(着手使用 UML) 21
static operations and attributes(静态操作与静态属性) 85～86
template (parameterized) classes(模板(参数化)类) 103～104
times to use(使用时间) 66
UML version changes(UML 版本变动) 189～190
value objects(值对象) 93～95
versus object diagrams(与对象图之对比) 108～109
visibility(可见性) 106～107
Class properties, *See also* Classes(类特性,参见类)
associations(关联) 47～48
associations, bidirectional associations(关联,双向关联) 53～55
associations, immutability versus frozen(关联,永恒还是冻结) 184
associations, qualified (关联,受限) 95～96
attributes(属性) 46～47
basics(基本原理) 44～48
derived(导出) 87～88
frozen(冻结) 93
generalization(泛化) 58～59
multiplicity(重数) 48～49
program interpretation (程序解释) 50～53
read-only(只读) 90
Class-Responsibility-Collaboration (CRC) cards(类-职责-协作(CRC)卡) 80～82

Classes, *See* Class properties(类,见类特性)
abstract(抽象) 89～93
association(关联) 99～102
attributes(属性) 85～86
Class - Responsibility - Collaboration (CRC) cards(类-职责-协作(CRC)卡) 80～82
derivation(导出) 103～104
dynamic data types(动态数据类型) 182～183
generalizations(泛化) 44～45
implementation(实现) 182～183
presentation(表象) 60
static data types(静态数据类型) 182～183
static versus dynamic classifications(静态分类还是动态分类) 99
subclassing(子类构作) 65
template(parameterized)(模板(参数化)) 103～104
Classifications(分类)
data types(数据类型) 182～183
dynamic and multiple(动态与多重) 97～99
implementation classes(实现类) 182～183
versus generalization(与泛化之对比) 96～97
Clients / suppliers(客户方 / 供应方) 60
Coad, Peter, UML history(Coad, Peter, UML 历史) 9～10
Cockburn, Alistair, use cases 130
Collaboration diagram, *See* Communication diagrams(Cockburn, Alistair, 用案) 130 (协作图,见通信图)
Collaborations(协作)
roles(角色) 170～171
sequence diagrams(顺序图) 171
times to use(使用时间) 173
Comments in class diagrams(类图中的注释) 59～60
Common Closure and Reuse Principles(共同封闭与复用原则) 112
Common Object Request Broker Architecture(CORBA) standards(公共对象请求代理体系结构(CORBA)) 2
Communication diagrams(通信图) 15～16
basics(基本原理) 160～162
times to use(使用时间) 162～163
Component diagrams(构件图) 15～16
basics(基本原理) 167～169
times to use(使用时间) 169
Composite structure diagrams(复合结构图) 15～16

basics(基本原理) 164～166
times to use(使用时间) 166
Composition(组合) 86～87
changes between UML versions(UML版本间之变动) 183～184
Computer-aided software engineering (CASE) tools(计算机辅助软件工程(CASE)工具) 4
UML history(UML历史) 9～12
Conceptual perspectives of UML(UML的概念视面) 7
Concurrent states(并发状态) 136～138
Conditionals(条件) 74～78
decisions and merges(选定与合并) 145
Constraints(约束) 145～146
complete/incomplete(完整/非完整) 183
rules(规则) 63～64
Construction, RUP projects(构作,RUP项目) 33
Continuous integration(连续集成) 28
Conventional use(习惯用法) 18
CORBA (Common Object Request Broker Architecture)(CORBA(公共对象请求代理体系结构)) 2
CRC (Class-responsibility- Collaboration) cards(CRC(类-职责-协作)卡) 80～82
Crystal,agile development process(晶体,敏捷开发过程) 31
Cunniugham, Ward, CRC cards (Cunniugham,Ward,CRC卡) 80～82

D

Data tadpoles(数据蝌蚪) 77
Data types(数据类型) 95
dynamic and multiple classifications(动态分类与多重分类) 182～183
implementation classes(实现类) 182～183
Decisions(选定) 145
Dependencies(依赖) 60～64
keywords(基词) 62
packages(包) 112～114
resources(资料源) 67
UML version changes(UML版本变动) 185
Deployment diagrams(部署图) 15～16
artifacts(制品) 119～121
design(设计) 38～39
devices(设备) 119～121
execution environments(执行环境) 119～120
nodes(结点) 119～120
times to use(使用时间) 121
Derivation of classes(类的导出) 103

Derived properties, class diagrams(导出特性,类图) 87～88
Descriptive rules, UML(描述性规则,UML) 17
Design(设计) 38
Development cases(开发例案) 32
Development processes(开发过程)
agile(敏捷) 31
DSDM(Dynamic Systems Development Method)(DSDM(动态系统开发方法)) 31
extreme programming(XP)(极端程序设计(XP)) 28,31～32,42～43
fitting processes to projects(过程适配项目) 33～35
FDD(Feature Driven Development)(FDD(特征驱动开发)) 31
iterative(迭代) 24～28
lightweight(轻量级) 31
Manifesto of Agile Software Development(敏捷软件开发宣言) 31
Rational Unified Process(RUP)(Rational 统一过程(RUP)) 32～33
resources(资料源) 42～43
selecting(选取) 42
staged delivery(阶段提交) 26
waterfall(瀑布) 24～28
Devices(设备) 119～120
Diagrams(图)
activity(活动) 15～16
actions, expansion regions(动作,展开区域) 154～155
basics(基本原理) 143～146
decomposing actions(动作的分解) 146～147
edges(边) 151～152
flow final(流终) 155～156
flows(流) 151～152
flows, Petri nets(流,佩特里网) 159
joins(汇合) 145
joins, specifications(汇合,指明) 156～157
partitions(分划) 148～149
pins(饰针) 152～153
requirement analysis(需求分析) 37～38
resources(资料源) 158～159
signals(信号) 149～150
times to use(使用时间) 158
tokens(权标) 151
transformations(转换) 152～153
UML version changes(UML 版本变动) 186～187,190
basics(基本原理) 14～16
class(类) 12,15～16

abstract classes(抽象类) 89～93
active classes(主动类) 105～106
aggregation and composition(聚合与组合) 86～87
association classes(关联类) 99～103
classifications(分类) 96～97
classifications, dynamic and multiple (分类,动态与多重) 97～99
comments(注释) 59～60
constraint rules(约束规则) 63～64
dependencies(依赖) 60～63
design(设计) 38
documentation(文档) 40～41
generalizations(泛化) 58～59,94～95
keywords(基词) 62,83～85
messages(消息) 107
notes(注文) 59～60
operations(操作) 56～58
properties(*See* class properties)(特性(见类特性))
reference objects(指引对象) 93～95
requirement analysis(需求分析) 37～38
resources(资料源) 67
responsibilities(职责) 85
starting with UML(着手使用 UML) 21
static operations and attributes(静态操作与静态属性) 85～86
template (parameterized) classes(模板(参数化)类) 103～104
times to use(使用时间) 66
UML version changes(UML 版本变动) 189～190
value objects(值对象) 93～95
versus object diagrams(与对象图之对比) 108～109
visibility(可见性) 106～107
classifications(分类) 16
communication(通信) 15～16,160～163
component(构件) 15～16,167～169
composite structure(复合结构) 15～16
basics(基本原理) 164～166
times to use(使用时间) 166
deployment(部署) 15～16
artifacts(制品) 114
design(设计) 38
devices(设备) 119～120
execution environments(执行环境) 119～120
nodes(结点) 119～120
times to use(使用时间) 121

interaction(交互)
basics(基本原理) 68～72,174～175
CRC cards(CRC 卡) 80～82
design(设计) 38
loops and conditionals(循环与条件) 74～78
participants(参加者) 68～72
sequence diagrams(顺序图) 68～72
synchronous and asynchronous messages(同步消息与异步消息) 79
times to use(使用时间) 174～175
interavtive overview(交互概观) 15～16
object(对象) 15～16
times to use(使用时间) 109
package(包) 15～16
basics(基本原理) 110～112
design(设计) 38
documentation(文档) 40～41
resources(资料源) 118
times to use(使用时间) 117
UML version changes(UML 版本变动) 187～188
sequence(顺序) 15～16
basics(基本原理) 68～72
centralized and distributed control (集中式控制与分布式控制) 69～72
collaborations(协作) 171
CRC cards(CRC 卡) 80～82
interaction diagrams(交互图) 68～72
loops and conditionals(循环与条件) 74～78
participants(参加者) 68～74
returns(回送(返回)) 184
starting with UML(着手使用 UML) 21
synchronous and asynchronous messages(同步消息与异步消息) 79
times to use(使用时间) 79～80
UML version changes(UML 版本变动) 189
shortcomings(缺点) 19～21
starting point(起点) 21
state machine(状态机) 15～16
activity status(活动状态) 134～135
basics(基本原理) 131～133
concurrent states(并发状态) 136～138
implementing(实现) 138～140
initial pseudostate(初始伪态) 131～132
internal activities(内部活动) 134

requirement analysis（需求分析）37～38
resources(资料源) 141～142
superstates(超态) 136
times to use(使用时间) 141
transitions(转接) 132
UML version changes(UML 版本变动) 190
timing(定时) 15～16
basics(基本原理) 176～178
types(类型) 15
types, UML version changes（类型，UML 版本变动）188
use case(用案)
basics(基本原理) 126～127
requirement analysis（需求分析）37～38
viewpoints(观点) 8
Dictionaries, *See* qualified associations(词典，见受限关联)
Distributed control of sequence diagramming(顺序图制作的分布式控制) 69～72
Do-activities(进行活动) 135
Documentation(文档) 40～41
Domain objects(领域对象) 60
DSDM（dynamic systems development method)(DSDM(动态系统开发方法))31
Dynamic classifications(动态分类) 99
data types(数据类型) 182～183

E

Edges(边) 151～152
Eiffel programming language(Eiffel 编程语言) 64
Engineering, forward(工程，正向)
UML as blueprints(UML 用作蓝图绘制语言) 3～4
UML as programming languages(UML 用作编程语言) 5
UML as sketches(UML 用作草图绘制语言) 2
Entry activities(进入活动) 134
Enumerations(枚举) 105
Event switches(事件开关) 137
Evolutionary development process, *See* iterative development(演化开发过程，见迭代开发过程)
Executable UML(可执行 UML) 5～6
Execution environments(执行环境) 120
Exit activities(退出活动) 134
Expansion regions(展开区域) 154～155
Extensions(扩张) 124
Extreme programming(XP)(极端程序设计(XP))

agile development process(敏捷开发过程) 31
resources(资料源) 42～43
techinal practices(技术实践) 28

F

Facades(虚包类) 112
FDD(feature driven development)(FDD(特征驱动开发)) 31
Features of use cases(用案的特征) 128～129
Fish-level use cases(鱼级用案) 128
Flows(流) 151～152
flow final(流终) 155～156
Petri nets(佩特里网) 159
Forks(分岔) 143～144
UML version changes(UML 版本变动) 186
Forward engineering(正向工程)
UML as blueprints(UML 用作蓝图绘制语言) 3
UML as programming languages(UML 用作编程语言) 2～5
UML as sketches(UML 用作草图绘制语言) 2
Found messages(基础消息) 70
Frozen property(冻结特征) 93,184
Fully qualified names(全受限名) 111

G

Gang of Four(四人组) 36～37
Generalizations(泛化) 44～45
class properties(类特征) 56
sets(集) 98～99
UML version changes(UML 版本变动) 185
versus classifications(与分类之对比) 96～97
Getting methods(获取方法) 57
Graphical modeling languages(图示建模语言) 1
Guarantees(保证) 126
Guards(监护) 76

H

Hashes, *See* Qualified associations(散列,见受限关联)
History pseudostate(历史伪态) 137

I

Implementation classes, data types(实现类,数据类型) 182～183
Include relationships(包含关系) 125
Incremental development process, *See* Iterative development(渐进开发过程,见迭代开发过程)
Initial node actions(初始结点动作)

143～144
Initial pseudostate(初始伪态) 131
Instance specifications(实例规约) 109
Integration, continuous(集成,连续) 28
Interaction diagrams(交互图)
basics(基本原理) 68～72,174～175
CRC cards(CRC卡) 80～82
design(设计) 38～39
loops and conditionals(循环与条件) 74～78
participants(参加者) 68～72
sequence diagrams(顺序图) 68～72
synchronous and asynchronous messages(同步消息与异步消息) 79
times to use(使用时间) 79～80,174～175,178
Interaction frames(交互架构)
loops and conditionals(循环与条件) 74～76
operators(操作符) 76～77
Interactive overview diagrams(交互概观图) 15～16
Interfaces(接口) 84
relationship to classes(和类的关系) 89～93
Internal activities, entry and exit(内部活动,进入与退出) 134
Internal activities, exit activities(内部活动,退出活动) 134
Invariants(不变式) 64～65
Iteration markers(迭代标记) 76
Iteration retrospective(迭代回顾) 34
Iterations(迭代) 24～28
timeboxing(时间框定) 27
Iterative development process(迭代开发过程) 24～28

J

Jacobson, Ivar(Jacobson, Ivar)
UML history(UML历史) 10,12
use cases(用案) 130
Jacuzzi development process, *See* Iterative development process(喷泉开发过程,见迭代开发过程)
Joins(汇合) 145
specifications(指明) 156～157
UML version changes(UML版本变动) 186～187

K

Keywords, class diagrams(基词,类图) 62～63,83～84
Kite-level use cases(风筝级用案) 128

L

Legacy code(遗产代码) 41～42
Lightweight development processes(轻量

级开发过程) 31
Lollipop notation(连珠图示法) 91～93
Loomis, mary, UML history (Loomis, Mary,UML 历史) 11
Loops(循环) 74

M

Main success scenario(主成功案况) 124
Mandatory attribute(强行属性) 49
Manifesto of Agile Software Development (敏捷软件开发宣言) 31
Maps, *See* Qualified association(映射,见受限关联)
Markers,iteration(标记,迭代) 76
MDA(model driven architecture)(MDA,模型驱动体系结构) 5～7
Mellor,Steve(Mellor,Steve)
  Executable UML(可执行 UML) 6
  UML history(UML 历史) 10
Merges(合并) 145
Messages(消息) 107
  asynchronous and synchronous(异步与同步) 79
  class diagrams(类图) 107
  found(基础) 70
  pseudomessages(伪消息) 77～78
Meta-models(元模型)
  definitions(定义) 12～14
  UML version changes(UML 版本变动) 188
Methods(方法)
  implementation of actions(动作的实现) 146～147
  versus operations(与操作之对比) 57～58
Meyer, Bertrand, Design by Contract (Meyer,Bertrand,按契约设计) 64
Model compilers(模型编译程序) 6
Modifiers(改态操作) 57
Multiple classifications(多重分类) 97～99
  data types(数据类型) 182～183
Multiplicity of properties(特性的重数) 48～49
Multivalued attributes(多值属性) 49

N

Namespaces(名空间) 110
Navigability arrows(导航箭号) 54～56
Nodes(结点) 119～120
Normative use(规范用法) 18
Notation(图示法)
  ball and socket(球座) 91～93
  definitions(定义) 12～14
  lollipop(连珠) 91～93

O

Object diagrams(对象图) 15～16
times to use(使用时间) 109
OCL(object constraint language)(对象约束语言) 64
Odell, Jim, UML history (Odell, Jim, UML历史) 10～11
OMG(Object Management Group)(OMG(对象管理组)) 1～2
control of UML(UML的控制管理) 2
MDA (model driven architecture) (MDA(模型驱动体系结构)) 5～7
revisions to UML versions(UML版本修订) 179～180
UML history(UML历史) 9～12
OO (object-oriented) programming (OO(面向对象)编程) 1
paradigm shift(风范转移) 72
Operations, *versus* methods(操作,与方法之对比) 57～58
Operators, interaction frames(操作符,交互架构) 74～76
Optional attributes(任选属性) 49

P

Package diagrams(包图) 15～16
basics(基本原理) 110～112
design(设计) 38
documentation(文档) 40～41
resources(资料源) 118
times to use(使用时间) 117
UML version changes (UML版本变动) 187～188
Packages(包)
aspects(面) 115
Common Closure and Reuse Principles (共同封闭与复用原则) 112
definitions(定义) 110
dependencies(依赖) 112～114
fully qualified names(全受限名) 111
implementing(实现) 116
namespaces(名空间) 110
Participants, sequence diagrams(参加者,顺序图) 68～74
Partitions, activity diagrams(分划,活动图) 148～149
Patterns(模式)
definition(定义) 35～36
Separated Interface(隔开接口) 117
state(状态) 136～142
using(使用) 173
Petri nets(flow-oriented technique) (佩特里网(面向流的技术)) 159
PIM(platform independent model) (PIM(与平台无关的模型)) 5
Pins(饰针) 152～153

Planning, adaptive *versus* predictive(计划制订,适应性还是预见性) 28～31
Platform specific model(PSM)(平台特定的模型(PSM)) 5
Post-conditions, Design by Contract(后置条件,按契约设计) 64
Pre-conditions(前置条件)
  Design by Contract(按契约设计) 64
  use cases(用案) 130
Predictive planning, versus adaptive planning(预见性计划制订与适应性计划制订之比较) 28～31
Prescriptive rules, UML(指定性规则, UML) 17
Presentation classes(表象类) 60
Private elements(私用成分) 106
Profiles(侧图) 84～85
  UML version changes(UML 版本变动) 187
Programming languages, UML as(UML 用作编程语言) 2,4～5
  forward engineering(正向工程) 2～5
  MDA (Model Driven Architecture) (MDA(模型驱动体系结构)) 5～7
  reverse engineering(逆向工程) 2～5
  value(价值) 7
Project retrospective(项目回顾) 34
Properties of classes(类特性)
  associations(关联) 47～48
    bidirectional associations(双向关联) 53～56
    qualified(受限) 95～96
  attributes(属性) 46～47
  basics(基本原理) 44～48
  derived(导出) 87～88
  frozen(冻结) 93
  multiplicity(重数) 48～49
  program interpretations(程序解释) 50～53
  read-only(只读) 93
Protected elements(受护成分) 106
Proxy objects(代理对象) 36
Pseudomessages(伪消息) 77～78
PSM(platform specific model)(PSM(平台特定的模型)) 5
Public elements(公用成分) 106

Q

Qualified associations(受限关联) 95～96
Queries(恒态操作) 57

R

Rational Unified Process(RUP)(Rational 统一过程(RUP))
  development cases(开发例案) 32
  phases(阶段) 32
  resources(资料源) 42～43

Read-only property(只读特性) 93
Rebecca Wirfs-Brock, UML history (Rebecca Wirfs-Brock,UML 历史) 10
Refactoring(结构改组) 28
Reference objects(指引对象) 93～95
Relationships(关系)
abstract classes to interfaces(抽象类与接口) 89～93
include(包含) 125～127
temporal(时态) 102
transitive(传递) 61
Releases(发布) 25
Requirements analysis(需求分析) 37～38
Requirements churn(需求折腾) 29
Responsibilities of classes(类的职责) 85
Retrospectives(回顾)
iteration(迭代) 34
project(项目) 34
Reusable archetypes(可复用本型) 6
Reverse engineering(逆向工程)
UML as blueprints(UML 用作蓝图绘制语言) 3,8
UML as programming languages(UML 用作编程语言) 4～5
UML as sketches(UML 用作草图绘制语言) 2～3
Revisions by versions(UML)(版本修订(UML))
from 0.8 through 2.0, general history (从 0.8 到 2.0,一般历史) 179～180
from 1.0 to 1.1(从 1.0 到 1.1) 182～185
from 1.2 to 1.3(从 1.2 到 1.3) 185～187
from 1.3 to 1.4(从 1.3 到 1.4) 187～188
from 1.4 to 1.5(从 1.4 到 1.5) 188
from 1.x through 2.0(从 1.x 直至 2.0) 188～189
Roles, *See* Actors(角色,见参与者)
Round-trip tools(双向工具) 4
Rumbaugh,Jim(Rumbaugh,Jim)
aggregation(聚合) 86
composite structure(复合结构) 165
UML history(UML 历史) 10～12
RUP (Rational Unified Process) (RUP (Rational 统一过程))
development cases(开发例案) 32
phases(阶段) 32
resources(资料源) 42～43

S

Scenario sets(案况集) 123
Scrum(喧嚣) 31

Sea-level use cases(海级用案) 128
Searching state(搜索状态) 135
Separated Interface(隔开接口) 117
Sequence diagrams(顺序图) 15～16
basics(基本原理) 68～72
centralized and distributed control(集中式控制与分布式控制) 69～72
collaborations(协作) 170～172
CRC cards(CRC 卡) 80～82
interation diagrams(交互图) 68
loops and conditionals(循环与条件) 74～78
participants(参加者) 68～74
returns(回送(返回)) 184
starting with UML(着手使用 UML) 21
synchronous and asynchronous messages(同步消息与异步消息) 79
times to use(使用时间) 79～80
UML version changes(UML 版本变动) 189
Setting mathods(置送方法) 57
Shlaer, Sally, UML history(Shlaer, Sally, UML 历史) 10
Signals(信号) 149～150
Single classification(单一分类) 97～98
implementation class(实现类) 182～183
Single-valued attributes(单值属性) 49
Sketches, UML as(UML 用作草图绘制语言)
forward engineering(正向工程) 2～5
reverse engineering(逆向工程) 2～5
Smalltalk(Smalltalk) 7
Software development processes, *See* Development processes(软件开发过程,见开发过程)
Software perspectives, UML(软件视面, UML) 7～8
Spiral development process, *See* Iterative development process(螺旋开发过程,见迭代开发过程)
Stable Abstractions Principle(稳定抽象原则) 113
Stable Dependencies Principle(稳定依赖原则) 113
Staged delivery development process(阶段提交开发过程) 126
Standard use(标准用法) 18
State diagrams, *See* state machine diagrams(状态图,见状态机图)
State machine diagrams(状态机图) 15～16
activity status(活动状态) 134～135
basics(基本原理) 131～133
concurrent states(并发状态) 136～

138
implementing(实现) 138～140
initial pserdostate(初始伪态) 131～132
internal activities(内部活动) 134
requiremens analysis(需求分析) 37～38
resources(资料源) 141～142
superstates(超态) 136
times to use(使用时间) 141
transitions(转接) 131～133,137
UML version changes(UML 版本变动) 190
State tables(状态表) 138～140
Static classifications(静态分类)
implementation classes(实现类) 182～183
versus dynamic classifications(与动态分类之对比) 99
Static operations of classes(类的静态操作) 85～86
Stereotypes(衍型) 84
Stories, *See* Features of use cases(情节,见用案的特征)
Subactivities(子活动) 146～147
Subclassing(子类构作) 65
assertions(断言) 64～65
Substitutability(置换性) 58
Subtypes(子类型) 59
Superstates(超态) 136
Suppliers/clients(供应方/客户方) 60
Swim lanes, *See* Partitions(泳道,见分划)
Synchronous messages(同步消息) 79
System use cases(系统用案) 128

T

Temporal relationships(时态关系) 102
Three Amigos(三友) 12
Time signals(时间信号) 149
Timeboxing(时间框定) 27
Timing diagrams(定时图) 15～16
basics(基本原理) 176～178
Tokens(权标) 151
Transformations(转换) 152～153
Transitions(转接) 33,132,136
state(状态) 139
Transitive relationships(传递关系) 61
Trigger(引发装置) 126
Types, *See* Data types(类型,见数据类型)

U

UML(UML)
conventional use(习惯用法) 18
definition(定义) 1～2
descriptive rules(描述性规则) 17
fitting into processes(适配过程) 35

history(历史) 9～12
meaning(含义) 19
prescriptive rules(指定性规则) 17
resources(资料源) 22
software and conceptual(软件视面与概念视面) 7
standards, legal *versus* illegal use(标准,合法使用还是非合法使用) 17～19
UML as blueprints(UML 用作蓝图绘制语言)
forward engineering(正向工程) 2～5, 8
reverse engineering(逆向工程) 2～5,8
UML as programming language(UML 用作编程语言) 2,4～5
forward engineering(正向工程) 2～5
MDA (Model Driven Architecture) (MDA(模型驱动体系结构)) 5
reverse engineering(逆向工程) 2～5
value(价值) 7
UML as sketches(UML 用作草图绘制语言) 2,3,9
forward engineering(正向工程) 2～5
reverse engineering(逆向工程) 2～5
UML diagrams, *See* Diagrams and specific diagram types(UML 图,见图与特定图型)
UML distilled, book editions and corresponding UML(UML 精粹,书版与相应 UML 版本) 181～182
UML revisions by versions(UML 版本修订)
from 0.8 through 2.0, general history (从 0.8 到 2.0,一般历史) 179～180
from 1.0 to 1.1(从 1.0 到 1.1) 182～185
from 1.2 to 1.3(从 1.2 到 1.3) 185～187
from 1.3 to 1.4(从 1.3 到 1.4) 187～188
from 1.4 to 1.5(从 1.4 到 1.5) 188
from 1. x through 2.0(从 1. x 直至 2.0) 188～189
Unidirectional associations(单向关联) 53
Unified Method Documentation(统一方法文档) 10
Unified Modelling Language, *See* UML (统一建模语言,见 UML)
UP(Unified Process), *See* RUP(UP(统一过程),见 RUP)
Use case diagrams(用案图)
basics(基本原理) 126～127
requirements analysis(需求分析) 37～38

Use cases(用案)
actors(参与者) 123
business(业务) 128
extensions(扩张) 124～126
features(特征) 128～129
include relationships(包含关系) 125
levels(级别) 128
MSS(main success scenairo)(MSS(主成功案况)) 124
resources(资料源) 130
scenario sets(案况集) 123
times to use(使用时间) 129
UML version changes(UML 版本变动) 185～186
User Guide(用户指南) 141
User stories, *See* Features of use cases(用户情节,见用案的特征)

V

Value objects(值对象) 93～95
Visibility(可见性) 106～107

W

Warehousing systems, platform independent model and platform(仓库系统,与平台无关的模型及平台) 5
Waterfall development process(瀑布开发过程) 24～28
Well formed UML(组成良好的 UML)
definition(定义) 19
legal UML(合法的 UML) 17～19

X

XP(extreme programming)(XP(极端程序设计))
agile development process(敏捷开发过程) 31～32
resources(资料源) 42～43
technical practices(技术实践) 28