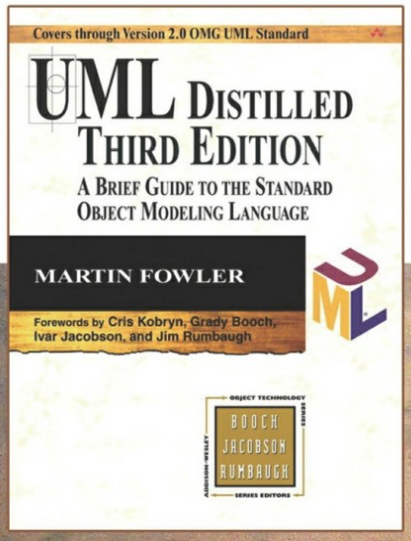
Broadview®
www.broadview.com.cn
PEARSON

传世经典书丛
Eternal Classics

UML精粹：

标准对象建模语言简明指南

（第3版）



[美] **Martin Fowler** 著
UMLChina 译

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

UML精粹：标准对象建模语言简明指南（第3版）

1. 第1章 简介
2. 第2章 开发过程
3. 第3章 类图：基础
4. 第4章 序列图
5. 第5章 类图：进阶概念
6. 第6章 对象图
7. 第7章 包图
8. 第8章 部署图
9. 第9章 用例
10. 第10章 状态机图
11. 第11章 活动图
12. 第12章 通信图
13. 第13章 组合结构
14. 第14章 组件图
15. 第15章 协作
16. 第16章 交互概述图
17. 第17章 时间图
18. 附录A UML版本之间的变化
19. 参考文献
20. 索引
21. 后折页
22. 封底

第1章 简介

1.1 UML是什么

统一建模语言（UML）是一组图形表示法。这些表示法的背后有共同的元模型。UML帮助描述和设计软件系统，特别是使用面向对象（OO）风格建造的软件系统。这个定义多少有点简化了。事实上，对不同的人来说，UML会稍微不同。之所以会这样，有它自己的历史原因，也有人们为了达到高效的软件工程过程采用不同视图的原因。因此，本章的主要任务是，通过解释人们看待和使用UML的不同方式，拉开本书的大幕。

图形建模语言在软件业已经出现很长时间了。背后的基本驱动力就是：编程语言的抽象级别不够高，不方便讨论设计。

尽管事实上图形建模语言出现很长时间了，但是对于它们在软件业中扮演的角色有很多的争议。轮到UML头上，也有一样的争议。

相对来说，UML是比较开放的标准，它由对象管理组织（OMG）控制。OMG是一个由各公司组成的联盟。OMG成立的目的是为了建立支持互操作性的标准，特别是面向对象系统的互操作性。OMG最广为人知的工作成果可能是CORBA（通用对象请求代理架构）标准。

20世纪80年代末到20世纪90年代初，出现了许多面向对象的图形建模语言，UML是这些语言的联合。1997年UML诞生后，这段特定的巴别塔时代成了历史。我和其他许多开发人员一样，深深感谢UML的出现。

1.2 使用UML的方式

关于UML在软件开发中所扮演角色的核心问题是——人们会用不同的方式使用它，其他图形建模语言也存在这种使用上的差别。这导致了一个长期以来的争论：如何使用UML。

为了理清这个问题，Steve Mellor和我分别针对人们使用UML的特征归纳出三种模式：草稿、蓝图和编程语言。目前最常用的一种，是**把UML当做草稿（UML as sketch）**，至少从我的“偏见”看是这样。在这种用法中，开发人员使用UML协助沟通系统的某些方面。在把UML当做蓝图时，你可以从正向工程或逆向工程两个方向使用草稿。**正向工程（forward engineering）**在编写代码之前画UML图，而**逆向工程（reverse engineering）**从已有代码建造UML图，目的是帮助人们理解代码。

把UML当做草稿，其本质是选择性。正向使用草稿，你粗略画出即将要编码的东西，通常要和团队中的一群人讨论。使用草稿的目的是来帮助沟通想法或者展示所要做事情的可选方案。你不会谈论即将要写的所有代码，只会和同事谈论一下重要的东西，或者在开始编程之前将一部分设计可视化。像这样的会议可以很短，如花10分钟讨论需要几小时进行的编程，或者用一天时间来讨论一个需要两周时间的迭代。

在逆向工程中，可以使用草稿来解释系统的某些部分如何工作。你不用展示每一个类，只需展示那些令人感兴趣的、在深入到代码之前值得讨论的类。

因为打草稿是相当非正式和动态的，需要快速地协作进行，所以常用的媒介之一是白板。草稿在文档中也有用，在这种情况下，焦点是沟通而不是完整性。草稿工具是轻量级的画图工具，通常人们不会特别纠结于UML的严格规则。书籍（例如我的其他书籍）中展示的大多数UML图都是草稿。它们强调的是选择性的沟通，而不是完整的规则。

相反，**把UML当做蓝图（UML as blueprint）**就要关心完整性。在正向工程中，思路是由一名设计人员开发蓝图，他的工作是为程序员建造详细设计，然后程序员在此基础上编码。设计应该足够完整，列出所有设计决策，程序员应该能够跟随设计，把编码当做基本不需要思考的、相当直接的活动。设计人员和程序员可以是同一个人，但通常设计人员是一名更高级的开发人员，他为一个团队的程序做设计。这个方法的灵感来自其他形式的工程，即职业工程师创建工程图，移交给建筑公司来建造。

设计人员可以对所有细节画蓝图，也可以对特定的区域画蓝图。一种常见的做法是，设计人员开发蓝图级模型只做到子系统的接口，而让开发人员负责实现细节。

在逆向工程中，蓝图的目标是传达关于代码的详细信息，可以在纸质文档上，也可以在交互式的图形浏览器上。蓝图能够以图形的形式展示类的每个细节，更易于开发人员理解。

比起草稿来，蓝图需要复杂得多的工具，以处理任务所需的细节。特制的CASE（计算机辅助软件工程）工具就属于这一类，不过CASE这个术语已经变得不太好听，厂商目前都避免使用这个词。正向工程工具支持画图并把图放进存储器中保存信息。逆向工程工具阅读源代码，解释源代码后将其放进存储器，并生成图形。像这样可以做正向和逆向工程的工具被称为双程（**round-trip**）工具。

一些工具使用源代码本身作为存储器，把图当做代码的图形视角。这些工具和编程更接近，经常直接和程序编辑器集成。我喜欢把这些工具看做**无转换（tripless）**工具。

蓝图和草稿之间的界限有些模糊，但是我认为，两者的区别基于这个事实：草稿故意画成不完整的，强调重要信息，而蓝图倾向于全面，目的经常是把编程缩减成简单的机械活动。简言之，我的观点是草稿是探索性的，而蓝图是定义性的。

在UML方面做得越多，编程变得越机械，显然，这时编程应该被自动化。事实上，有许多CASE工具是生成某些形式的代码，自动化地建造系统的一个重要部分。最终到达这样一个境界——整个系统可以用UML来表述，进而将**UML作为编程语言（UML as programming language）**。在这个环境下，开发人员画UML图，直接编译为可执行代码——UML变成了源代码。显然，UML的这种用法需要特别复杂的工具。（同样，这个模式下正向和逆向工程的概念就没有意义了，因为UML和源代码是同一个东西。）

模型驱动架构和可执行UML

当人们谈论UML时，他们也经常谈论**模型驱动架构（MDA）**[Kleppe et al.]。本质上，MDA是一种使用UML作为编程语言的标准方法；该标准和UML一样，由OMG控制。通过生产一款遵从MDA的建模环境，厂商可以创建在其他兼容MDA的环境中也能运行的模型。

MDA经常和UML放在一起谈论，因为MDA使用UML作为其基本建模语言。当然，你使用UML时未必要使用MDA。

MDA把开发工作划分为两个主要方面。建模人员通过创建平台无关模型（**Platform Independent Model，PIM**）表达特定应用。PIM是独立于特定技术的UML模型。然后，工具将PIM转化为**平台特定模型（Platform Specific Model，PSM）**。PSM是针对特定执行环境的系统模型。接着会有更进一步的工具接管PSM，为该平台生成代码。PSM可以是UML，也可以不是。

因此，如果你要使用MDA建造一个仓储系统，你会从创建仓储系统的PIM开始。接下来如果你想要这个仓储系统运行在J2EE和.NET上，你会使用某些厂商的工具来创建两个PSM：每个平台一个。然后，进一步的工具将生成两个平台的代码。

如果从PIM到PSM再到最终代码的过程是完全自动化的，我们就把UML当成了编程语言。如果其中有些步骤是手工的，我们就把UML当成了蓝图。

Steve Mellor已经在此类工作上研究很长一段时间了，最近他使用了一个术语：**可执行UML（Executable UML）**[Mellor and Balcer]。可执行UML类似于MDA，但使用稍微不同的术语。类似地，你从一个平台无关模型开始，这个模型等同于MDA的PIM。然而，下一步是使用一个模型编译器把UML模型一步转成可部署的系统；因此，不需要PSM。正如术语编译器所指出的，这个步骤是完全自动化的。

模型编译器基于可复用的架构型。**架构型（archetype）**描述如何把可执行UML模型转换到特定编程平台。因此，对于仓储的例子，你要购买一个模型编译器和两个架构型（J2EE和.NET）。在你的可执行UML模型上运行每一个架构型，就得到了两个版本的仓储系统。

可执行UML不使用完全的UML标准；许多UML构造被认为是没有必要的，因此没有用到。所以，可执行UML比完整的UML更简单。

所有这一切听起来不错，但有多少能成为现实呢？在我看来有两个问题。首先是工具的问题：工具是否足够成熟来完成任务。这一点会随着时间而变化；当然，在我写这些文字的时候，这些工具还没有被广泛使用，并且我也没有看到在开发中使用很多工具。

把整个UML概念性地作为编程语言，是更基本的问题所在。在我看来，只有在比使用另一种编程语言带来显著的更高的生产率时，才值得把UML作为编程语言使用。基于我用过的各种图形开发环境的经验，我不能说服自己那样做。即使它的生产率更高，仍然需要得到超过临界值的用户量，才能成为主流。这本身就是一个大障碍。和许多老的Smalltalk程序员一样，我认为Smalltalk比当前主流语言要高效得多。但Smalltalk现在只是一门利基语言，我没有看到许多项目使用

它。为了避免得到Smalltalk的下场，UML必须更加走运才行，即使它已经很优秀。

一个围绕着UML作为编程语言的有趣问题是：如何建模行为逻辑。UML 2提供三种行为建模方法：交互图、状态图和活动图。这些方法都有对编程的支持。如果UML确实作为编程语言流行起来，看看哪一种技术能够成功也是有趣的。

另一种人们看待UML的方式是使用范围：用于概念建模还是软件建模。大多数人熟悉使用UML做软件建模。基于这个**软件视角（software perspective）**，UML元素会相当直接地映射到软件系统中的元素。正如我们将看到的，映射不是强制性的，但是当我们使用UML时，我们的确是在谈论软件元素。

从**概念视角（conceptual perspective）**看，UML表达对所研究领域概念的描述。在这里，我们不谈论太多软件元素，因为我们正在建造谈论特定领域的一个词汇表。

关于视角，没有硬性的规则；正如它所表现的，用法的范围真的十分广泛。一些工具将源代码自动转为UML图，把UML作为源代码的另一种视图。这种做法更多的是从软件的视角看问题。如果你尝试使用UML图和一帮会计沟通并理解各种资产池术语的含义，你更多地处在概念的思维框里面。

在本书之前的版本中，我把软件视角分离为规格（接口）和实现。在实践中，我发现在这两者之间做严格区分太难，因此我认为不值得再去强调其中的区别。然而，我总是倾向于在我的图中强调接口而不是实现。

这些使用UML的不同方式，导致大家去争论UML图是什么意思，它们和世界上其他事物之间的关系是什么。特别是，它也影响着UML和源代码之间的关系。一些人持这样的观点：UML应该用于创建独立于编

程语言的设计，编程语言用于实现。其他人相信：独立于语言的设计是自相矛盾的，过分强调这一点是愚蠢的。

另一个视角上的不同是：什么是UML的本质。在我看来，大多数UML使用者，特别是UML草稿使用者，把UML的本质看做图形。然而，UML的创造者们认为图是次要的，UML的本质是元模型，图只是元模型的表示。这个观点同样适用于UML蓝图使用者及UML编程语言使用者。

因此无论何时你阅读到任何涉及UML的文字，重要的是理解作者的视角。那样，你才能理解UML经常引起的激烈争论。

说了那么多，我需要清晰表明我的个人倾向。几乎所有时候，我都把UML当成草稿。我发现把UML作为草稿在正向和逆向工程及概念视角和软件视角上都是有用的。

我不是详细正向工程蓝图的粉丝；我相信这太难做好，而且会减慢开发速度。子系统接口级别的蓝图是合理的，但即使这样，你也应该预料到，当开发人员实现跨接口的交互时，可能要改变那些接口。逆向工程蓝图的价值依赖于工具。如果工具被用做动态的浏览器，会非常有帮助；如果工具生成一大堆文档，只会浪费更多的树木。

在我看来，把UML当做编程语言是一个美妙的主意，但我怀疑它很难被广泛接纳。对于大多数编程任务来说，我不太相信图形形式比文本形式更有生产效率，即使确实生产效率更高，一门语言要被广泛接受，也是很难的。

既然我的倾向如此，所以本书更多地聚焦于使用UML作为草稿。幸运的是，这使得本书成了简短的指南。我不能用这样的篇幅的书来描述UML其他模式的用法，但本书可以作为介绍其他书的良好指南。因此如果你对以其他方式使用UML感兴趣，我建议你把本书当做通往其他

你需要看的书的介绍。如果你只对把UML作为草稿感兴趣，本书可能正是你需要的。

1.3 UML诞生史

我承认，我是一个历史迷。我最喜欢的轻松阅读方式是看一本好的历史书。但我也知道不是每个人都有这个爱好。我在这里谈论历史是因为我认为如果不理解UML的历史，在很多方面就难以理解UML。

在20世纪80年代，对象开始离开实验室，迈出走向“真实”世界的第一步。Smalltalk开始有稳定的平台可供人们使用，C++也诞生了。这时，各种人开始考虑面向对象图形设计语言。

面向对象图形建模语言的关键书籍出现在1988年到1992年之间。领导人物包括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和Steve Mellor [Shlaer and Mellor，data]、[Shlaer and Mellor，states]，以及Rebecca Wirfs-Brock（职责驱动设计）[Wirfs-Brock]。

这些作者中的每一位现在都非正式地领导着一组喜欢他们的思想的实践者。所有这些方法都非常相似，然而它们之间包含许多经常会让人恼火的微小差别。同一个基本概念以差别很大的表示法出现，导致了客户产生混淆。

在这个艰难的时期，也有人谈到标准化，但没人理睬。OMG的一个团队尝试标准化，但只收到一封来自所有重要方法学家的公开抗议信。（这让我想起一个老笑话。问题：方法学家和恐怖分子之间的区别是什么？回答：你可以和恐怖分子谈判。）

首次引发创建UML的催化剂事件是Jim Rumbaugh离开GE加盟Grady Booch所在的Rational（现在是IBM的一部分了）。Booch/Rumbaugh联盟从一开始就被视为可以达到市场份额的临界值。Grady和Jim宣称“方法战争结束了——我们赢了”，基本上就宣布了他们打算“以微软的方式”达成标准化。许多其他方法学家建议成立一个反Booch同盟。

在OOPSLA'95大会上，Grady和Jim首次公开描述他们合并的方法：统一方法文档版本0.8。更重要的是，他们宣布Rational Software购买了Objectory，因此，Ivar Jacobson将加入统一的团队。Rational举行了一个庆功派对来庆祝0.8版本草稿的发布。（派对的亮点是Jim Rumbaugh第一次公开演唱；我们都希望这也是最后一次。）

接下来的一年，更加开放的合并过程出现了。大多数时候站在一旁的OMG，现在扮演了一个积极的角色。Rational不得不结合Ivar的思想，还要花时间和其他伙伴沟通。更重要的是，OMG决定担当主角。

在这一点上，认识到为什么OMG介入非常重要。方法学家，例如书籍作者，宁愿认为他们是重要的。但我不认为书籍作者的呼声会被OMG听到。导致OMG介入的是工具厂商的呼吁，他们害怕标准被Rational控制，Rational工具会得到不公平的竞争优势。因此，厂商力促OMG在CASE工具互操作性的旗帜下做些事情。这面旗帜很重要，因为OMG代表的就是互操作性。大家的想法是创建一个UML，它允许CASE工具自由地交换模型。

Mary Loomis和Jim Odell成为工作组最初的负责人。Odell清楚地表明他准备为标准放弃他的方法，但他不要Rational强加的标准。1997年1月，各种组织一起提交了方法标准的建议书，以方便模型的互换。Rational和其他许多组织一起协作，发布了UML文档版本1.0作为他们的建议，这是UML第一次被叫做统一建模语言。

接下来是短回合的扳手腕过程，各种建议书被合并。OMG采纳1.1版本作为官方的OMG标准。稍后又做了一些修订。修订1.2完全是走走

过场。修订1.3更加重要。修订1.4围绕组件和扩展机制添加了许多详细的概念。修订1.5添加了动作语义。

当人们谈论UML时，会把创造者的功劳主要归于Grady Booch、Ivar Jacobson和Jim Rumbaugh，他们一般被称为“三友”（Three Amigos）。不过，搞怪的人喜欢故意读错第二个词的第一个音节。虽然人们将主要功劳给了他们，但我认为把绝大多数荣誉归功于“三友”有些不公平。UML表示法首先成型于Booch/Rumbaugh统一方法。从那时起，很多工作就由OMG委员会领导了。在后来这些阶段中，Jim Rumbaugh是“三友”中唯一做出重要贡献的。我的观点是，UML过程委员会的那些成员才称得上首要的UML功臣。

1.4 表示法和元模型

目前UML定义了表示法和元模型。**表示法（notation）**是你在模型中看到的图形，它是建模语言的图形语法。例如，类图表示法定义了如何表达类、关联和多重性等条目和概念。

当然，这导致了一个问题：关联、多重性甚至类的确切含义是什么。常见的用法给出了一些非正式的定义，但许多人需要更严密的定义。

严密的规格和设计语言的思想在形式方法领域很普遍。在这些技能中，设计和规格用一些谓词演算衍生物来表达。这样的定义在数学上很严密，不允许模棱两可。然而，这些定义的价值不是普适的。即使你能够证明程序满足数学规格，也没有办法证明数学规格满足了系统真正的需求。

大多数图形建模语言没有那么严格；它们的表示法看起来是凭直觉的，而不是正式定义的。从整体上来看，这么做没有太大的害处。这些方法可能是非正式的，但依然有许多人发现它们有用——管用就行了呗。

然而，方法学家还是在寻找在不牺牲好用性的情况下提高方法严谨性的方法。一种方法是定义一个元模型（**meta-model**），即一张定义语言概念的图，通常是类图。

图1.1是UML元模型的一个小片段，展示了特性之间的关系。（这里的摘录只是让你了解元模型长成什么样，我不打算解释它。）

元模型对建模表示法的用户影响有多大？答案更多依赖于使用的模式。草稿使用者通常不太在意，蓝图使用者更在意一些。对那些使用UML作为编程语言的人来说，元模型是至关重要的，因为它定义了语言的抽象语法。

许多参加到持续的UML开发中的人，首先都对元模型感兴趣，因为它对UML作为编程语言使用很重要。表示法的问题经常放在次要的位置，如果你要尝试自行熟悉标准文档，把这一点记在心中很重要。

图1.1 UML元模型的小片段

随着你更加深入和详细地使用UML，你会认识到你需要的东西比图形表示法多得多。这就是为什么UML工具如此复杂的原因。

在本书中，我不注重严密性，更喜欢传统的方法路径，主要是为了唤起你的直觉。像这样的一本薄书，作者又主要倾向于草稿用法，所以本书自然是这样的风格。如果你需要更多的严密性，应该去看更详细的书。

1.5 UML图

表1.1列出了UML 2描述的13种官方图形类型，图1.2展示了这些图的分类。这些图形类型是许多人学习UML的方式，也是我组织本书的方式，但是UML的作者并没有把图看做UML的核心部分，因此图形类型

并不是特别死板的。通常，你能够合法地把来自一种图形类型的元素用在另一种图上。UML标准指明了某个元素通常画在某个图形类型上，但这不是死的规定。

表1.1 UML官方图形类型

图1.2 UML图形类型分类

1.6 什么是合法的UML

乍一看，回答这个问题应该很简单：合法的UML就是在规格中良好定义的东西。然而，在实践中，回答要复杂一些。

这个问题的一个重要地方是，UML的规则是描述性的还是强制性的。**强制性规则（prescriptive rules）**语言由官方控制，官方会说明语言中什么是合法的、什么是不合法的，以及你用这门语言所发表言论的含义是什么。**描述性规则（descriptive rules）**语言要通过看人们在实践中如何使用该语言来理解它的规则。编程语言倾向于由标准委员会或占主导地位的厂商来设定强制性规则，而自然语言，例如英语，倾向于描述性规则，它的含义由习惯设定。

UML是十分精确的语言，因此你可能期望它有强制性的规则。但UML经常被认为是其他工程学科的蓝图的软件等价物，这些蓝图不是强制性的表示法。没有委员会规定结构性工程图中合法的符号是什么；这些表示法通常习惯性地被接受，和自然语言相似。仅有标准不解决问题，因为领域里的人们可能不会遵从标准所说的每一样东西；你问问法国人关于Académie Française的事情就知道了。另外，UML如此复杂，以至于标准经常有多个开放的解释。即使是评审本书的UML领导者也会在UML标准的解释上有不同的意见。

这个问题对我编写本书和你使用UML都是重要的。如果你要理解UML图，重要的是要认识到理解UML标准不是一切。在整个行业和特定项目中，人们确实接受习惯性用法。因此，虽然UML标准是UML的首要信息源，但不是唯一的。

我的态度是，对于大多数人，UML有描述性规则。UML标准对UML含义的影响是最大的，但不是唯一的。我认为对于UML 2尤其是这样，UML 2引进了一些习惯表示法，这些习惯和UML 1的定义或者UML的使用习惯冲突，给UML增加了更多复杂性。因此，在本书中当我发现标准和UML习惯用法时，我会尝试概括出来。当我不得不在本书中做区分时，我将使用术语**习惯用法（conventional use）**来指明某些东西不在标准中，但我认为是广泛使用的。对于某些遵从标准的东西，我将使用术语**标准的（standard）或规范的（normative）**。（规范是标准制定者使用的术语，意思是一个你必须遵从的陈述，这样在标准中才有效。因此，非规范的UML也是一种很好的形式，用来说明某些东西相对于UML标准严格来说是非法的。）

当你查看一张UML图时，你应该牢记——UML的通用原则是特定图上的任何信息都可以被**收起（suppressed）**。这种收起可以是普遍的——隐藏所有属性，或者特定的——不展示这三个类。因此，在一张图中，你绝对不能够因为没有看到某个东西就乱做推断。如果多重性不见了，你不能推断它可能是什么值，即使UML元模型有默认的值，例如属性是[1]。如果你看不见图上的信息，可能是因为它是默认的，也可能是因为它被收起来了。

话是这么说，不过还是有一些通用的习惯，例如多值性质是集合。书中我将指出这些默认的习惯。

如果你是一名草稿使用者或者蓝图使用者，重要的是不要太强调合法的UML。给系统一个好的设计更重要，我宁可有一个好的设计，但用的是非法的UML，而不愿意要合法却苍白的设计。显然，又好又合法固然最好，但你最好把精力放在好的设计上，而不是操心UML的奥

秘。（当然，如果把UML当做编程语言，你使用的UML必须是合法的，否则你的程序就不会正常运行了！）

1.7 UML的含义

关于UML，一个尴尬的问题是，虽然规格详细地描述了UML的确切定义，但对于在UML元模型的纯粹世界之外，UML意味着什么，它没有说太多。UML图如何对应特定的编程语言，也没有正式的定义。你不能看到一张UML图，确切说出等价的代码是什么样子。然而，你可以对代码是什么样子有一个大概的设想。在实践中，这已经够用了。开发团队经常形成他们的本地习惯，你需要熟悉那些正在使用的习惯。

1.8 仅有UML是不够的

虽然UML提供十分可观的各种图形来帮助你定义一个应用，但没法列出你想用的所有图。在许多地方，会用不同的图，如果没有UML图符合你的要求，你不应该犹豫是否使用非UML图。

图1.3是一张屏幕流图，展示了用户界面上的各种屏幕，以及如何在这些屏幕之间进行转换。我看到人们使用这些屏幕流图许多年了。它们的意思是什么，我从来都只看到过非常粗糙的定义；UML里面没有类似的东西，然而我发现它是一种非常有用的图。

图1.3 Wiki的一部分的非正式屏幕流图（http://c2.com/cgi/wiki）

表1.2展示了我喜欢的另一种图表：决策表。决策表是展示复杂逻辑条件的好办法。你可以用活动图代替，但如果你面对的情况比较复杂，决策表会更加精练和清晰。此外，现在已经有许多形式的决策表了。表1.2分为两个部分：双线上面的部分是条件，下面的部分是结果。每一列都展示了一个特定的条件组合是如何导致特定的一组结果的。

你会在各种书中碰到类似这样的技术。如果某项技术看起来适合你的项目，尽管尝试，不要犹豫。如果管用就用，如果不管用，就丢掉。（当然，对于UML图也是同样的建议。）

表1.2 决策表

1.9 何处开始UML

没有人能够理解或使用所有的元素，即使是UML的创造者。大多数人使用UML的一个小子集来工作。你不得不找到适合自己和同事的UML子集。

如果你正在开始，我建议你首先把精力放在基本形式的类图和序列图上。在我看来，它们是最常用的，也是最有用的图形类型。

一旦你掌握了上面这些，就可以开始使用一些更高级的类图表示法，并且可以看看其他的图形类型。实验并看看对你有多大帮助。不要害怕丢弃任何看起来对你的工作没有用的东西。

1.10 更多资料

本书不是一本完整、可靠的UML参考书，更不用说OO分析和设计了。很多其他书籍写了许多值得一读的东西。当我讨论个别主题时，我也会提及你应该看的其他书，以获取更深入的信息。下面会给出一些UML和面向对象设计方面的通用书籍。

对于所有我推荐的书，你可能需要检查它们是针对哪个UML版本写的。在2003年6月以前，还没有使用UML 2的书出版，这不奇怪，因为标准还墨迹未干。我建议的书是好书，但我不能确定它们是否会随UML 2标准的更新而更新及何时更新。

如果你是对象新手，我推荐我当前最喜爱的入门书：[Larman]。作者在设计上使用强烈的职责驱动方法，值得效仿。

UML确切的文字，你应该看官方的标准文档；但是记住，它们是为躲在自己的斗室里赞同UML的方法学家而写的。如果想要看更容易消化的标准版本，看一看[Rumbaugh，UML Reference]。

关于面向对象设计方面更详细的建议，你可以通过[Martin]学习到许多好东西。

我也建议你阅读模式方面的书，这样可以获得超越基础的材料。既然现在方法战争已经结束，模式（**33**页）是大多数关于分析和设计的有趣材料出现的地方。

第2章 开发过程

正如我已经提到的，UML从一堆OO分析和设计方法中发展而来。所有这些都在一定程度上混合了图形建模语言和描述如何开发软件的过程。

有趣的是，当UML成形时，各种玩家发现，虽然他们在建模语言上可以达成一致，但几乎不可能在过程上达成一致。因此，他们同意把关于过程的各种希望达成的协议放在一边，把UML局限为仅仅是建模语言。

本书的标题是《UML精粹》，因此我可以安全地忽略过程。然而，在不了解它们如何在过程中使用的情况下，我不相信建模技术会有什么用。你使用UML的方式在很大程度上依赖于所使用的过程的风格。

因此，我认为首先讨论过程是重要的，这样你可以看到使用UML的上下文。我不打算深入讨论任何特定过程的细节；我只是要给你足够的信息看到上下文，以及指引你到哪里可以发现更多材料。

当你听到人们讨论UML时，你经常会听到他们谈到Rational统一过程（RUP）。RUP是一个过程——或者，更严格地说，是一个你可以使用UML的过程框架。但除了它经常涉及的各种人来自Rational以及名称里有“统一”之外，它和UML没有任何特殊关系。UML可以和任何过程一起使用。RUP是一个流行的方法，31页将讨论它。

2.1 迭代和瀑布过程

关于过程，最大的争论之一发生在瀑布和迭代风格之间。过程的术语经常被误用，特别是迭代被看做时尚，而瀑布看起来很不合时宜时。因此，许多项目声称做迭代开发，但实际上做的是瀑布开发。

两者之间的本质区别是，你如何将一个项目分解为更小块。如果你有一个认为要花一年时间的项目，几乎没有人可以舒服地通知团队离开一年，回来的时候，项目已经做好了。因此，需要做一些分解，人们才能够着手处理问题和跟踪进度。

瀑布风格基于活动来分解项目。为了建造软件，你不得不做某些活动：需求分析、设计、编码和测试。为期一年的项目可能有2个月的分析阶段，然后是4个月的设计阶段，接着是3个月的编码阶段，再接着是3个月的测试阶段。

迭代风格根据功能子集来分解项目。你可能会把一年分解为3个月的迭代。在第一个迭代中，你会处理1/4的需求，并对这1/4做完整的软件生命周期：分析、设计、代码和测试。在第一个迭代结束时，你拥有了一个做了1/4所需功能的系统。然后，你再做第二个迭代，这样在6个月结束时，你拥有了一个完成一半功能的系统。

当然，以上是简化的描述，但它是两者差别的本质。当然，实践中的过程会掺杂一些杂质。

在进行瀑布开发时，每一个阶段之间通常有某些正式的交接，但也经常有回溯。在编码期间，会出现一些情况导致你要回头看分析和设计。当然，在开始编码时，你不应该假设所有设计完成了。在后续阶段，回头再看分析和设计决策是不可避免的。然而，这些回溯是一种异常，应该尽可能地减到最少。

一方面，在进行迭代发开时，在真正的迭代开始之前，你通常会看到某些形式的探索活动。至少，这会让我们得到需求的高级别视图：至少足以让我们把需求打碎，放进接下来要进行的迭代中。在探索期间，也会发生一些高级别设计决策。另一方面，虽然每一个迭代应该生产出准备产品化的已集成软件，但是通常很难严格达到这个地步，因为需要一个稳定化周期来消除最后的bug（漏洞）。另外，一些活动会留到最后做，例如用户培训。

你可能无法在每一次迭代结束时都把系统变成产品，但系统的质量应该向产品看齐。然而，通常你可以固定地隔一段时间就把系统产品化一次；这样做有好处，因为你越早从系统得到价值，就会得到越高质量的反馈。在这种情况下，你经常听说一个项目有多个**发布（release），**而每一个发布又分解到若干**迭代（iteration）**中。

迭代开发有许多名字：能想到的有增量、螺旋、演进和极可意（Jacuzzi）。很多人描述了这些名字之间的区别，但区别没有被广泛接受，和迭代与瀑布之间的分歧相比，也没有那么重要。

你可以使用杂化的方法。[McConnell]描述了**阶段交付（staged delivery）**生命周期，先以瀑布风格做完分析和高级别设计，然后把编码和测试划分到迭代中。这样一个项目可能会有4个月的分析和设计，紧跟着是4个两个月的迭代来建造系统。

在过去几年，大多数软件过程作者，特别是面向对象社区成员，不喜欢瀑布方法。这有许多原因，最基本的原因是，瀑布过程中辨别项目是否真的还在轨道上非常困难。它太容易在早期阶段宣称胜利，却隐藏了进度延误的情形。通常，你可以真正辨别项目是否还在轨道上的唯一方法是生产已测试、集成的软件。迭代风格不断重复这样做，如果某些东西出了偏差，你会更好地得到警示。

仅仅为了这个原因，我强烈推荐项目不要使用纯瀑布方法。如果不能采用更纯粹的迭代技术，你至少也应该使用阶段交付。

OO社区钟情于迭代开发由来已久，可以放心地说，投入到建造UML的人中，有相当多的人至少喜欢某些形式的迭代开发。但是我对行业实践的感觉是，瀑布开发依然是更通用的方法。其中一个原因就是被称为伪迭代开发的东西：人们声称在做迭代开发，但事实上做的是瀑布开发。常见的症状有：

- “我们正在做一个分析迭代，然后做两个设计迭代……”

- “这个迭代的代码bug非常多，但最后我们会清除它。”

特别重要的是，每一个迭代产生已测试、已集成的代码，代码尽可能接近于产品级质量。测试和集成是最难估计的活动，因此重要的是不要把这样的开放式活动放在项目结尾。测试应该做到：任何没有计划要发布的迭代，不用做大量额外的开发工作就可以发布。

迭代的常见技能是使用**时间盒（time boxing）**。时间盒强迫每个迭代有固定长度的时间。如果看起来你不能建造所有在一个迭代期间打算建造的东西，你必须决定从迭代中延迟一些功能而不必延迟迭代的日期。大多数使用迭代开发的项目，在整个项目范围内使用同一迭代长度；这样，你就有了固定节奏的构建。

我喜欢时间盒，因为人们通常难以延迟功能。如果练习定期延迟功能，在大的发布时会更好地在延迟日期和延迟功能之间做出明智的选择。迭代期间延迟功能也能高效地帮助人们学习真正的需求优先级是什么。

关于迭代开发，一个最常见的关注点是返工的问题。迭代开发显式地假设在项目的后续迭代期间你会返工和删除已有代码。在许多领域中，例如制造业，返工被看做浪费。但软件业不像制造业；因此，通常重写已有代码比围着设计不良的代码打补丁要高效。许多技能实践可以大大地帮助你，使返工更加高效。

- **自动化回归测试（automated regression test）**，当你改变一些东西时，它允许你快速检测任何可能引进的缺陷。xUnit家族测试框架是建造自动化单元测试特别有价值的工具。xUnit起源于JUnit（**http://junit.org**），现在已经移植到几乎每一种想得到的语言（参见**http://www.xprogramming.com/software.htm**）。一个好的经验法则是，单元测试代码的量应该和产品代码差不多。

- **重构（refactoring）**是一种有纪律地改变现有软件的技能[Fowler，refactoring]。重构对代码基使用一系列小的、行为保留的变换。这些变换中的许多变换可以自动化（参见**http://www.refactoring.com**）。

- **持续集成（continuous integration）**保持团队同步，以避免痛苦的集成周期[Fowler and Foemmel]。持续集成的核心在于完全自动化的构建过程，无论何时，任何团队成员在代码基中签入代码，构建过程都会自动启动。开发人员每天都要签入，因此自动化构建每天都要做许多次。构建过程包含运行一大块自动化回归测试，这样，任何不一致都可以被快速捕获，所以很容易修正。

所有这些技能实践最近因极限编程（XP）[Beck]而流行起来，虽然它们之前已经被使用，而且不管你是否使用XP或任何其他敏捷过程，都可以用。

2.2 预测性和自适应计划

瀑布还在继续使用的原因之一是：人们希望软件开发可预测。没有什么比对要花多少成本和时间来建造软件没有清晰的想法更加令人沮丧了。

预测性方法指望在项目早期做些工作，以便更好地理解后面不得不做的事情。这样，你可以到达一个点，在那里，项目的后面部分可以以一定程度的准确性来估算。使用预测性计划（**predictive planning**），一个项目有两个阶段。第一个阶段准备好一个计划，这个计划比较难预测，但第二个阶段更加可预测，因为计划已经在实施。

这不是非黑即白的事情。随着项目的进行，你会逐渐获得更多的可预测性。即使你有一个预测性计划，还是会出差错。你只能期望一旦坚实的计划实施，实际和估算的偏离变得不那么重要。

然而，对于是否有许多软件项目是可预测的，有许多争论。这个问题的核心是需求分析。软件项目复杂性的独特来源之一是理解软件系统需求的困难。大部分软件项目经历了重要的**需求剧烈变动（requirements churn）**：项目后期阶段的需求改变。这些变动动摇了预测性计划的基石。你可以通过在早期冻结需求并且不允许变化来防止这个问题发生，但也有风险：交付的系统不再满足用户的需求。

这个问题导致两种非常不同的反应。一个流派主张在需求过程本身上投入更多的精力。这样，你可以得到更准确的需求集，剧烈变动会减少。

另一个流派主张需求剧烈变动是不可避免的，对许多项目来说，确保需求足够稳定以致能使用一个预测性计划是很困难的。原因可能是展望软件能做什么实在太困难，或者市场条件强加了不可预测的改变。这个流派的思想鼓吹**自适应计划（adaptive planning）**，认为可预测性看起来就是错觉。与其用幻想的可预测性愚弄自己，不如面对不断改变的现实，把处理改变看做软件项目的常态。这样，改变得到控制，项目交付所能交出的最好的软件；虽然项目不是可预测的，但项目是可控的。

当人们谈论项目如何进行时，预测性项目和自适应项目之间的区别就会在许多方面浮现出来。如果人们谈论一个项目做得好是因为它按计划进行，那是预测性形式的想法。在自适应环境中，你不可以说“按计划”，因为计划总是在改变。这并不意味着自适应项目不做计划；它们通常做很多计划，但计划只是当做基线来评估改变的结果，而不是对将来的预测。

使用预测性计划，你可以开发一份固定价格/固定范围的合同。这样一份合同确切说明应该建造什么，要花多少钱，何时交付。这样的固定在自适应计划中是不可能的。你可以固定预算和交付时间，但你不能固定交付什么功能。自适应合同假设用户会和开发团队协作，定期重

新评估需要建造什么功能，如果进展太慢，会取消项目。像这样，自适应计划过程可以是固定价格/可变范围的。

自然，喜欢自适应方法的人更少，因为任何人都更喜欢软件项目有更多的可预测性。然而，可预测性依赖于精确、准确和稳定的需求集。如果你不能稳定你的需求，预测性计划就建在流沙之上了，项目脱轨的概率很高。这里针对这种情况给出两点重要的建议。

1．不要做预测性计划，直到你有精确、准确的需求，并确信它们不再有大的改变。

2．如果你不能得到精确、准确和稳定的需求，就使用自适应计划的形式。

预测性和自适应性为选择生命周期提供了原料。自适应计划绝对需要迭代过程。预测性计划两种方式都可以，不过，采用瀑布或阶段交付方法，更容易看到项目如何进行。

2.3 敏捷过程

在过去几年，人们对敏捷软件过程产生了巨大的兴趣。敏捷（agile）这个术语是个口袋，装了许多过程，这些过程共享由敏捷软件开发宣言（**http://agile Manifesto.org**）定义的常见价值和原则集合。这些过程的例子有极限编程（XP）、Scrum、特性驱动开发（FDD）、Crystal和DSDM（动态系统开发方法）。

在我们讨论的术语中，敏捷过程在本性中是非常自适应的，它们也是非常以人为本的过程。敏捷方法假设项目成功的最重要因素是项目中人的素质，以及他们一起工作时人际关系有多好。使用什么过程和工具绝对是次要因素。

敏捷方法倾向于使用短的、基于时间盒的迭代，最常见的是一个月或更短的迭代。因为没有附加重量级的文档，敏捷方法鄙视以蓝图模式使用UML。大多数项目以草稿模式使用UML，还有一些宣称使用UML作为编程语言。

敏捷过程倾向于更少的**仪式（ceremony）**。注重仪式的过程或重量级过程在项目期间有许多文档和控制点。敏捷过程认为仪式使得做出改变更困难，而且有才华的人也不喜欢。因此，敏捷过程经常被刻画为**轻量的（lightweight）**。重要的是认识到缺少仪式是自适应和以人为本的结果，而不是基本性质。

2.4 Rational统一过程

虽然Rational统一过程（RUP）独立于UML，但两者经常被一起谈论。因此我想这里值得谈一些东西。

虽然RUP被称为过程，实际上它是一个过程框架，提供了一个用来谈论过程的词汇表和松散结构。当你使用RUP时，你需要做的第一件事是选择一个**开发案例（development case）**：你打算在项目中使用的过程。开发案例变化的范围很广，所以不要假设你的开发案例和其他开发案例看起来很像。选择开发案例需要有人事先非常熟悉RUP：这样的人可以为特定项目的需要裁剪RUP。或者，有越来越多的开发案例包，可以从中选择一个作为开始。

不管是什么开发案例，RUP本质上是迭代的过程。瀑布风格和RUP的哲学不兼容，但是，可悲的是，使用瀑布风格过程却披着RUP外衣的项目并不罕见。

所有RUP项目应该遵循4个阶段。

1．**初始（inception）阶段**对项目做初始的估计。通常在初始阶段，你决定是否投入足够的资金来做细化阶段。

2．**细化（elaboration）阶段**识别项目的首要用例，并迭代建造软件，以便使系统的架构成型。细化阶段的最后，你应该对需求有好的体会，并且有大致可工作的系统来扮演开发的种子。特别是，你应该发现并解决了项目的主要风险。

3．**构造（construction）阶段**继续建造过程，开发足够发布的功能。

4．**移交（transition）阶段**包含各种后期阶段的活动，这些活动不用迭代去做，可能包括部署到数据中心、用户培训等。

在阶段之间，有相当多的模糊地带，特别是在细化阶段和构造阶段之间。对某些人来说，转到构造阶段说明你可以转移到预测性计划模式。对其他人来说，它只是说明你对需求有了更广视角的理解，并且有了一个你打算在项目剩下部分延续的架构。

有时，RUP被称为统一过程（UP）。通常，希望使用RUP的术语和整体风格但不使用Rational Software授权产品的组织会这样称呼它。你可以把RUP想成基于UP的Rational产品，或者把RUP和UP想成一个东西。两种方式都会有人赞同你。

2.5 为项目裁剪过程

不同软件项目之间有很大差别。软件开发进行的方式依赖于许多因素：你所建造的系统的种类、你所使用的技术、团队的规模和分布、风险的本质、失败的后果、团队的工作风格和组织的文化。因此，你决不应该期望有一个对所有项目普遍适用的过程。

所以，你总是不得不改编过程以适合你的特定环境。你需要做的第一件事是看一看你的项目，考虑哪一种过程看起来接近于适合。结果应该会得到一个短的过程列表，供你考虑。

然后，你应该考虑你需要做什么改变来适合你的项目。你不得不小心一些。许多过程很难完全领会，直到你用它们来工作。在这些情况下，经常值得使用现成的过程做两三个迭代，直到你了解了它如何工作。接着你可以开始修改过程。如果从一开始你就很熟悉过程如何工作，你可以一开始就修改它。记住，从太小的地方开始再添加东西比起从太多地方开始再把东西拿走，通常更容易。

模式

UML告诉你如何表达面向对象设计。相反，模式看的是过程的结果：例子设计。

许多人会说，项目有问题是因为项目成员不知道那些更有经验的人熟知的设计。模式描述常见的做事情的方式，由发现重复设计主题的人收集。这些人针对每一个主题来描述，这样其他人可以阅读模式，并看到如何应用模式。

让我们看一个例子。假设你的桌面上有一些对象运行在一个进程中，它们需要和运行在另一个进程中的其他对象沟通。也许这个进程也在你的桌面上，也许它驻留在别的地方。你不想让你的系统中的对象为发现网络上的其他对象或者执行远程例程调用而操心。

你可以做的是在本地进程内为远程对象创建一个代理对象。代理对象有和远程对象同样的接口。你的本地对象和代理对象交互，使用正常的进程内消息发送。然后，代理对象负责传送消息到真实对象，不管它驻留在何处。

代理是用于网络或其他地方的常见技能。人们有许多使用代理的经验，知道它们可以如何使用，它们会带来什么好处，它们的局限性以及如何实现它们。像本书这样的方法书籍不讨论这种知识；它们所讨论的只是你可以如何画出代理的图，虽然有用，但不如讨论代理相关的经验有用。

20世纪90年代早期，一些人开始捕获这方面的经验。他们组成了对编写模式感兴趣的社区。这些人举办了一些会议，并出版了若干书籍。

来自这个人群的最著名的模式书籍是[Gang of Four]，这本书详细讨论了23种设计模式。如果你要知道关于代理的知识，该书花了10页来讨论这个主题，详细描述了对象如何一起工作、该模式的好处和局限性、常见的变体以及实现提示。

模式不只是模型。模式还必须包含为什么这样做的原因。经常有人说模式是问题的解决方案。模式必须清晰地识别问题，解释为什么它这样解决问题，也要解释模式管用和不管用的环境。

模式很重要，因为它们是理解基本的语言或建模技能以后的下一个阶段。模式带给你一系列的解决方案，也向你展示好的模型是什么样子，以及如何构造模型。模式通过例子来教学。

我开始学习设计时，我纳闷为什么我不得不事事从零做起，为什么没有一本手册向我展示如何做常见的事情？模式社区在尝试建造这些手册。

现在已经出版了许多模式书籍，质量参差不齐。我喜爱的书籍有[Gang of Four]、[POSA1]、[POSA2]、[Core J2EE Patterns]、[Pont]，再冒昧加上我的[Fowler，AP]和[Fowler，P of EAA]。你也可以看一看模式的主页：**http://www.hillside.net/patterns**。

开始时，不管你对使用的过程如何自信，在往下进行的过程中，学习是必要的。事实上，迭代开发的一大好处是支持频繁的过程改进。

在每一个迭代的末尾，举行一次迭代回顾（**iteration retrospective**），团队聚在一起，考虑事情进行得如何，以及可以如何改进。如果你的迭代很短，两三个小时足够了。做这件事情的一个好办法是做一个有三个类目的列表：

1．保持：开展得很好，要确保继续做。

2．问题：开展得不好的区域。

3．尝试：改变你的过程以获得改进。

你可以在第一个迭代之后就开始每一次迭代回顾，通过评审之前的会议提到的条目，看看事情如何改变。不要忘记保持更新要做的事情的列表，重要的是保持跟踪起作用的事情。如果你不做这件事，你会丢失对项目的感觉，以及没有注意到潜在的致胜实践。

在项目的末期或有主要发布时，你可能要考虑一次更正式的、持续两三天的项目回顾（**project retrospective**）；更多细节参见**http://www.retrospectives.com/**和[Kerth]。我感到最恼火的事情之一就是组织总是不能从自己的经验中学习，导致一次又次地犯下昂贵的错误。

2.6 为过程裁剪UML

当人们使用图形建模语言时，通常在瀑布过程的上下文中看它们。瀑布过程通常以文档作为分析、设计和编码阶段之间的交接。这些文档的主要部分经常是图形模型。事实上，来自20世纪70年代和20世纪80年代的许多结构化方法就像这样讨论了大量的分析和设计模型。

不管你是否使用瀑布方法，你依然会做分析、设计、编码和测试的活动。你可以运作一个每周一迭代的迭代项目，每周一个小型瀑布。

使用UML不一定意味着开发文档或者引进复杂的CASE工具。许多人只在开会时在白板上画UML图，以帮助沟通他们的想法。

2.6.1 需求分析

需求分析活动涉及尝试弄清楚软件的用户和顾客需要系统做什么。手头有许多UML技能可以使用，包括：

- 用例，描述人们如何与系统交互。
- 类图，从概念视角画类图，是建造严密的领域词汇表的好办法。
- 活动图，能够展示组织的工作流，展示软件如何与人类活动交互。活动图能够为用例展示其上下文，以及复杂用例的工作细节。
- 状态图，如果一个概念有一个有趣的生命周期，里面有各种状态和改变状态的事件，状态图很有用。

做需求分析时，记住，最重要的事情是与你的用户和顾客沟通。通常，他们不是软件开发人员，不熟悉UML或任何其他技能。即使如此，我已经成功地和非技术人员一起使用过这些技能。为了达到这一点，记住，重要的是保持表示法最小化。不要引进任何软件实现特有的东西。

在任何时候，准备好打破UML的规则，如果这样能够帮助你更好地沟通。在分析中使用UML的最大风险是你画了领域专家不能完全理解的图。不能被了解领域的人们理解的图比无用还糟糕，这样做只会培养对开发团队的不信任。

2.6.2 设计

在你做设计时，你可以在你的图上放更多技术的东西。你可以使用更多的表示法，而且表示法也要更精确。一些有用的技能是：

- 类图从软件视角展示软件中的类，以及它们如何互联。

- 常用场景的序列图。一个有价值的方法是从用例中挑选最重要和最有趣的场景，使用CRC卡或序列图来弄清楚软件中发生了什么。
- 包图用于展示软件的大规模组织。
- 为有复杂生命历史的类创建状态图。
- 部署图用于展示软件的物理布局。

一旦写好了软件，这些技能有许多同样可以用于给软件归档。如果人们不得不在软件上工作，但又不熟悉软件的代码，文档可以帮助他们理解软件。

在瀑布生命周期中，这些图和活动作为阶段的一部分。阶段结束时的文档通常包括该活动合适的UML图。瀑布风格通常意味着将UML用做蓝图。

在迭代风格中，UML图可以用做蓝图或草稿。用做蓝图时，分析图通常会在建造功能的迭代之前的迭代中被制作。每一个迭代不是从头开始，而是修改现有文档，强调新迭代中要做的改变。

蓝图设计通常在迭代早期就做完了，然后，可能会针对本次迭代不同的小块功能一块一块完成。再说一遍，迭代意味着对现有模型做出改变，而不是每次建造新模型。

以草稿模式使用UML意味着过程更加可变。一种方法是在迭代的开始花费两三天草拟出本次迭代的设计。你也可以在迭代期间的任何时候召开短的设计会议，无论何时，开发人员开始处理稍为重要的功能，就举行半小时的快速会议。

以蓝图方式使用UML时，你期望遵循图来实现代码。来自蓝图的改变是一种偏离，需要制作蓝图的设计人员的评审。草稿通常更多地被认

为是设计的第一刀；如果编码期间人们发现草稿不完全正确，他们应该能自如地去改变设计。实现人员不得不判断改变是否需要更广的讨论以理解所有的分歧。

关于蓝图，我的一个关注点是：根据我自己的观察，即使是好的设计人员，把蓝图做对也非常难。我经常发现我自己的设计经过编码以后，很难不做修改。我发现UML作为草稿依然很有用，但我没有发现它们可以是绝对正确的。

不管是两种模式中的哪一种，对于探索设计备选方案都是有意义的。通常最好以草稿模式探索备选方案，这样你可以快速生成和改变备选方案。一旦你挑选了设计来运作，就可以使用草稿或者蓝图详细描述。

2.6.3 文档

一旦你建造了软件，可以使用UML帮助你归档已经做的事情。为此，我发现UML图对整体理解系统很有用。然而，在这样做时，应该强调，我不相信UML可以生成整个系统的详图。引用Ward Cunningham[Cunningham]的话：

小心选出和编写良好的备忘录，可以轻易替换传统的面面俱到的设计文档。后者只有孤立的几个闪光点。提升那些点的地位……忘记剩下的东西。（[Cunningham]384页）

我相信详细的文档应该从代码生成，就像JavaDoc。你写附加文档的目的应该是强调重要概念。把这些看做读者在进入基于代码的细节的第一步。我喜欢使用散文结构，短到一杯咖啡时间读完，使用UML图来帮助阐述讨论。我更喜欢把图作为草稿，强调系统最重要的部分。显然，文档作者需要决定什么是重要的、什么不是，比起读者，作者有更多的知识来做这个工作。

包图适合描绘系统的逻辑路线。包图帮助我理解系统的逻辑块，同时看到依赖并保持依赖受控。展示高级别物理图像的部署图（参见第8章），也证明在这个阶段是有用的。

在每一个包内部，我喜欢看到一张类图。我不展示每个类上的每个操作，只展示帮助我理解内容的重要特性。这张类图扮演图形目录的角色。

应该做一些交互图来支持这张类图，用于展示系统中最重要的交互。再说一次，在这里选择性是重要的；记住，在这种文档中，全面是可理解性的敌人。

如果一个类有复杂的生命周期行为，我就画一张状态机图（参见第10章）来描述它。只有行为足够复杂时，我才会这样做，我发现这种情况不常有。

我经常在文档中放进一些重要的代码，用易读的程序风格写就。如果涉及特别复杂的算法，我会考虑使用活动图（参见第11章），但只有代码让我有更多理解时才这样做。

如果我发现重复出现的概念，我会使用模式（33页）来捕获基本的思路。

文档中最重要的一项内容是你没有采纳的设计备选方案以及没有采纳的原因。这是你可以提供的外部文档中，经常最容易忘记、但最有用的部分。

2.6.4 理解遗留代码

UML可以帮助你通过两三种方式弄清楚一大堆不熟悉的代码。建造关键事实的草稿可以扮演图形注解的机制，帮助你在学习时捕获重要的信息。包中关键类的草稿及它们的关键交互可以帮助澄清所发生的事情。

使用现代的工具，你可以为系统的关键部分生成详细的图。不要使用这些工具来生成大型的纸质报表；相反，在探索代码时使用它们来深入关键区域。一个特别棒的功能是生成序列图，以查看处理复杂的方法时多个对象如何协作。

2.7 选择开发过程

我对迭代开发过程有强烈爱好。正如我已经在本书前面说的：项目要成功，你就应该使用迭代开发。

也许这样说有点大嘴巴，但随着我的年龄的增长，我越来越喜欢使用迭代开发了。做好了的话，这是一种基本的技能，可以用来早点暴露风险以更好地控制开发。迭代开发不等于没有管理，不过我应该公平地指出，一些人就是这样使用迭代开发的。迭代开发确实需要好好计划。它是一种很实在的方法，因此每一本OO开发书籍都鼓励使用迭代开发。

听说我是敏捷软件开发宣言作者中的一员你应该不会感到惊讶，我非常喜欢敏捷方法。我也有过许多关于极限编程的正面经验，当然你应该非常认真地评估它的实践。

2.8 更多资料

关于软件过程的书籍总是常见的，敏捷软件开发的兴起带来了许多新的书籍。整体而言，我最喜爱的通用过程书籍是[McConnell]。作者在书中给出了软件开发涉及的许多广泛和实践性的主题，并且给出了有用的实践列表。

来自敏捷社区的[Cockburn，agile]和[Highsmith]提供了敏捷过程的好的概述。关于以敏捷方式使用UML的建议，参见[Ambler]。

最流行的敏捷方法之一是极限编程（XP），你可以在**http://xprogramming.com**和**http://www.extremeprogramming.org**等网站找到许多东西。XP孕育出许多书，这就是为什么现在我把它称为“从前的轻量方法学”。学习XP，通常的起点是[Beck]。

[Beck and Fowler]虽然是为XP而写的，但给出了迭代项目中计划的更多细节。大部分内容其他XP书也覆盖了，但如果你只对计划方面感兴趣，本书是很好的选择。

关于Rational统一过程的更多信息，我最喜爱的入门书籍是[Kruchten]。

第3章 类图：基础

如果在黑暗的小巷中，某人来到你面前说：“喂，要看一张UML图吗？”这张图可能就是类图。大多数我看到的UML图都是类图。

类图不只是被广泛使用，而且是大多数建模概念的基础。类图的基本元素每个人都需要，但高级概念通常用得较少。因此，我把类图讨论分成两部分：基础（本章）和进阶（第5章）。

类图（class diagram）描述系统中的对象类型，以及存在于它们之间的各种静态关系。类图也展示类的性质和操作，以及应用于对象连接方式的约束。UML使用**特性（feature）**作为一个通用术语，覆盖了类的性质（property）和操作（operation）。

图3.1展示了一个简单的类模型，不出大家所料，用的是订单处理的例子。图中的方框是类，方框被分为三栏：类的名字（粗体）、类的性质和类的操作。图3.1也展示了类之间的两种关系：关联和泛化。

图3.1 一个简单的类图

3.1 性质

性质（property）代表类的结构特性。一开始，你可以把性质粗略看做对应于类中的字段。正如我们将看到的，实际上它相当复杂，不过一开始这样看它是合理的。

性质虽然只是单个概念，但可以以两种十分不同的表示法出现：属性和关联。虽然在图中看起来十分不同，但实际上是同样的东西。

3.1.1 属性

属性（attribute）表示法把性质描述成类方框中的一行文本。属性的完整形式如下:

例如:

只有name是必需的。

- visibility标记意味着属性是公开的（+）还是私有的（-），我将在104页讨论其他标记。
- 属性的name——类如何引用属性——大致对应于编程语言中的字段名。
- 属性的type指明哪一种对象可以被放进属性的限制。你可以把这看成编程语言中的字段类型。
- 我将在47页解释multiplicity。
- default value是在创建期间没有指定值时新创建对象的值。
- {property-string}允许你指出属性附加的性质。在例子中，我使用{readOnly}来表示客户不可以修改该性质。如果没有这个，你通常可以假设该属性是可以修改的。随后涉及时，我将描述其他性质字符串。

3.2.2 关联

表示性质的其他方法是关联。你能够在属性中展示的很多信息同样可以出现为关联。图3.2和图3.3展示了同一性质的两种不同表示法。

图3.2 展示订单的性质为属性

关联（**association**）是一根两个类之间的实线，方向从源类到目标类。性质的名称及多重性放在关联的目标端。关联的目标端链接到性质所属类型的类。

图3.3 展示订单的性质为关联

虽然两种表示法中出现的信息大多数相同，但还是有一些不同之处。特别是，关联能够展示线两端的多重性。

同一个东西有两种表示法，明显的问题是，为什么你应该使用这个或者另一个？一般来说，我倾向于对小的东西使用属性，例如日期或布尔——一般来说，值类型（91页）。对于更重要的类，使用关联，例如顾客和订单。我也更喜欢为图中重要的类使用类方框，这样就要使用关联，对于图中重要性稍差的使用属性。这样的选择更多是一种强调，并没有潜在的含义。

3.2 多重性

性质的多重性（**multiplicity**）指出多少对象可以填充该性质。最常见的多重性是：

- **1**（一张订单必须有且只有一名顾客。）

- **0..1**（一名企业顾客可能有一名销售代表，也可能没有销售代表。）
- *****（一名顾客不需要下订单，但顾客下的订单数量没有上限——零或者更多张订单。）

更常见的是，多重性通过一个下限和上限来定义，例如canasta纸牌游戏有2..4个玩家。下限可以是任何正整数或零；上限可以是任何正整数或*（无限）。如果下限和上限是同一个数，你可以使用一个数；因此，1等同于1..1。因为0..*是很常见的情况，所以可以简写为*。

在属性中，你会碰到各种说明多重性的术语。

- **Optional**（可选的）意味着下限0。
- **Mandatory**（强制的）意味着下限为1，也可能更多。
- **Single-valued**（单值的）意味着上限为1。
- **Multivalued**（多值的）意味着上限大于1，通常是*。

如果我有一个多值的性质，我更喜欢使用复数形式来命名。

默认地，多值多重性的元素形成了一个集合，因此如果你向一名顾客要他的订单，返回订单的顺序是不固定的。如果订单的排序在关联中有含义，你需要添加{ordered}到关联端。如果允许重复，添加{nonunique}。（如果你要显式展示默认值，你可以使用{unordered}和{unique}。）你也可以看到面向集合的名称，例如{bag}的意思是无序、非唯一。

UML 1允许不连续的多重性，例如2，4（含义2或4，在小型货车出现之前，车内的座位数）。不连续的多重性很少见，UML 2将其移除了。

属性的默认多重性是[1]。虽然元模型允许，但是图中如果有属性没有标出多重性，你不能假设多重性为[1]。因为这张图可能把多重性信息隐藏了。因此，如果它很重要的话，我更喜欢显式说明[1]多重性。

3.3 性质的编程解释

和UML里的其他东西一样，用代码来解释性质的方式不止一种。最常见的软件表达是编程语言的字段或性质。因此图3.1中的Order Line类对应于Java中类似这样的东西：

像C＃这样有性质的语言，应该对应于：

注意，在支持性质的语言中，一个属性通常对应于公开的性质，但对于不支持性质的语言来说，属性对应的是私有字段。对于没有性质的语言，可以通过存取器（读取和设置）方法来看到字段。只读属性没有设置方法（针对字段）或者设置动作（针对性质）。注意，如果你没有给性质命名，通常就使用目标类的名称。

使用私有字段是对图的一种非常聚焦于实现的解释。更面向接口的解释把重点放在获取方法上而不是潜在的数据上。在这个例子里，我们可以看到OrderLine的属性对应于如下方法：

在这个例子里，没有价格的数据字段；相反，它是一个计算出来的值。但从OrderLine类的客户关心的角度看，它和一个字段是一样的。客户不能辨别什么是字段，什么是计算出来的。信息隐藏正是封装的本质。

如果属性是多值的，意味着相关数据是一个集合。因此订单类会引用一个订单行的集合。因为这个多重性是有序的，这个集合必须是有序的（如Java中的List或者.NET中的IList）。如果集合是无序的，严格地说，它应该没有有意义的次序，因此用无序集实现，但大多数人也用列表来实现无序属性。一些人使用数组，但UML隐含着上限是无限的，因此我几乎总是使用集合来实现数据结构。

多值性质产生和单值性质不同的接口（在Java中）：

在大多数情况下，你不给多值性质赋值；相反，你通过add和remove方法更新它。为了控制它的订单行条目性质，订单必须控制集合的成员；因此，它不应该将裸的集合传出。在这个例子里，我使用一个保护代理来给集合提供一件只读的外衣。你也可以提供一个不可更新的迭代器或者做一个副本。客户可以修改成员对象，但不应该直接改变集合本身。

因为多值属性意味着集合，你几乎不会在类图中看到集合类。只在非常低级别的实现图中，你才会看到集合类。

你应该非常害怕类中只有字段集合及它们的存取器。面向对象设计希望提供能够做丰富行为的对象，因此不应该只简单地提供数据给其他对象。如果你重复调用存取器存取数据，这是一个征兆：某些行为应该转移到拥有该数据的对象。

这些例子也再次说明这样的事实：在UML和代码之间，没有硬性、快速的对应方式，但它们之间有相似之处。在一个项目团队中，团队的惯例会决定采用哪一种对应方式。

无论性质被实现为字段还是计算值，它都代表了对象总是能够提供的某些东西。你不应该使用性质来建模瞬时的关系，例如在方法调用时作为参数传递的对象仅限于在这次交互中使用。

3.4 双向关联

到目前为止我们看到的关联都叫做单向关联。另一种常见的关联是双向关联，如图3.4所示。

图3.4 双向关联

双向关联是一对性质，它们从两个方向链接在一起。Car类有性质owner：Person[0..1]，Person类有性质cars：Car[*]。（注意我如何命名cars性质，我用性质类型的复数形式，这是常见的做法，但不是规范的习惯。）

它们之间的双向链接意味着如果你跟随两个性质，你应该会回到包含起点的集合。例如，我从特定的MG Midget开始，发现它的拥有者，然后查看拥有者的车，车的集合应该包含作为起点的Midget。

除了用性质给关联加标签，许多人，特别是有数据建模背景的人，喜欢使用动词短语给关联加标签（图3.5），这样，关系可以用一句话来表达。这是合法的，你可以给关联添加一个箭头，以避免模棱两可。大多数对象建模人员更喜欢使用一个性质名称，这样与责任和操作更加对应。

图3.5 使用动词短语命名关联

一些人会用各种方式给每一个关联命名。我选择只有在这样做有助于理解时才命名关联。我看见过太多命名为“有”、“和……有关系”之类的关联。

在图3.4中，从关联两端的**导航性箭头（navigability arrow）**可以明显看出关联的双向本质。图3.5中没有箭头，UML允许你使用这两种形式来表示双向关联。我的偏好是，当你要说清楚这是一个双向关联时，就使用如图3.4所示的双向箭头。

在编程语言中实现双向关联经常有点棘手，因为你必须保证两个性质保持同步。使用C＃的话，实现双向关联的代码如下：

最重要的事情是，如果可能，让关联的一侧——单值侧控制该关系。为此，从属端（Person）需要向主控端泄露其数据的封装。这给从属类添加了一个尴尬的方法，这个方法实际上不应该在这里，除非语言有细粒度的存取控制。在这里，我使用“friend”作为其命名惯例，这是向C++学习，主控的设置器实际上就是友元。就像许多性质代码一样，这是相当公式化的东西，这就是为什么许多人更喜欢通过某些形式的代码生成来产生它。

在概念模型中，导航性不是一个重要的问题，因此我不在概念模型中展示任何导航性箭头。

3.5 操作

操作（operation）是类知道如何执行的动作。最明显的是，操作对应于类中的方法。一般来说，不展示那些简单操纵性质的操作，因为它们通常可以推导出来。

操作的完整UML语法如下：

- visibility标记可以是公开的（+）或私有的（-）；其他标记在104页讨论。

- name是一个字符串。
- parameter-list是操作的参数列表。
- return-type是返回值的类型，如果有的话。
- property-string指应用到给定操作的性质的值。

参数列表中的参数用和属性类似的方式标记，形式为：

- name、type和default value的意思和属性中的意思相同。
- direction指参数是输入（in）、输出（out）还是兼有（inout）。如果没有展示方向，就假设为in。

一个关于账户的操作例子可以是这样的：

对于概念模型，你不应该使用操作来详述类的接口。相反，应该使用它们来指出类的首要责任，也许要使用概述CRC责任的一些词（81页）。

我经常发现辨别改变系统状态和不改变系统状态的操作很有用。UML把**查询（query）**定义为从类中取值而不改变系统状态的操作——换句话说，没有副作用。你可以用性质字符串{query}标记这样一个操作。我把改变状态的操作称为**修改器（modifier）**，也称为命令。

严格地说，查询和修改器之间的区别就在于它们是否改变可观察的状态[Meyer]。可观察的状态是可以从外部觉察的东西。一个更新缓存的操作会改变内部状态，但从外部观察不到。

我发现强调查询很有帮助，因为你可以改变查询执行的次序，而不会改变系统行为。一个通常的惯例是尝试编写操作，使得修改器不返回值；这样，你可以依赖于这样的事实：返回值的操作是查询。[Meyer]把这称为命令 查询分离原则。有时总是这样做显得很尴尬，但在可以这样做时你应该尽可能这样做。

有时你会看到其他术语，例如获取方法和设置方法。**获取方法（getting method）**返回来自字段的值（不做其他事情）。**设置方法（setting method）**把值放进一个字段（不做其他事情）。从外部看，客户应该不能够识别查询是不是获取方法，修改器是不是设置方法。获取和设置方法的知识完全在类的内部。

操作和方法的另一个区别是：**操作（operation）**是对象上可以调用的某些东西——例程声明——而**方法（method）**是例程体。在多态情况下，这两者是不同的。如果一个超类型有3个子类型，每一个子类型都覆盖超类型的getPrice操作，你有一个操作和4个实现它的方法。

人们通常混用术语操作和方法，但有时准确了解它们之间的区别是必要的。

3.6 泛化

泛化（generalization）的一个典型例子是做生意时的个人和企业顾客。它们有差异也有相似之处。相似之处可以放进一个通用的Customer类（超类型）中，Personal Customer（个人顾客）和Corporate Customer（企业顾客）作为子类型。

从不同的建模视角看，这个现象也会有各种解释。概念上，如果根据定义，Corporate Customer的所有实例也是Customer的实例，我们可以说Corporate Customer是Customer的一个子类型，Corporate Customer是一种特殊的Customer。关键思路是我们针对Customer所说的每件事情——关联、属性、操作——对于Corporate Customer来说也是对的。

从软件视角看，明显的解释是继承：Corporate Customer是Customer的子类。在主流OO语言中，子类继承超类的所有特性，并且它是可能会覆盖任何超类的方法。

高效使用继承的一条重要原则是可替换性（**substitutability**）。在任何需要Customer的代码里面，应该能够替换为Corporate Customer，且一切工作正常。本质上，这意味着我编写代码时如果用到Customer，可以自由地使用Customer的任何子类型。因为使用多态，Corporate Customer响应某个命令的方式可能和其他Customer有所不同，但调用者应该不需要操心其中的差异。（关于这方面的更多信息，参见Liskov的替换原则（LSP）[Martin]。）

虽然继承是一种强有力的机制，它也带来了许多达成可替换性未必需要的包袱。一个好的例子是在Java的早期，许多人不喜欢内建Vector类的实现，要用某些更轻量的东西替换它。然而，产生一个可替换Vector的类的唯一办法是做一个子类，这意味着继承许多不需要的数据和行为。

还有许多其他机制可以用于提供可替换的类。因此，许多人喜欢区分子类型化（或接口继承）和子类化（或实现继承）。一个类如果可替换它的超类型，它就是一个子类型（**subtype**），不管是否使用继承。子类化（**subclassing**）用做正规继承的同义词。

还有许多其他机制允许你子类型化但其实并不用子类化。例如，实现一个接口（86页）及许多标准设计模式[Gang of Four]。

3.7 注解符和注释

注解符是图中的注释。注解符可以单独存在，也可以通过虚线链接到所注释的元素（图3.6）。注解符可以出现在任何类型的图中。

虚线有时会很尴尬，因为你不能定位线条确切在哪里结束。因此通常的惯例是在线的末端放上一个很小的开口圆圈。有时，把注释放在图形元素里面是有用的。你可以给文本加上两个半字线作为前缀：--。

图3.6 注解符用做一个或多个图形元素的注释

3.8 依赖

如果改变一个元素——**供应者（supplier）**或**目标**——的定义会导致改变其他——**客户（client）**或**源**——类，两个元素之间就存在**依赖（dependency）**。依赖的存在有各种原因：一个类发送消息给另一个类；一个类拥有另一个类作为其数据的一部分；一个类把另一个类当成操作的参数。如果一个类改变其接口，任何发送给该类的消息可能都不再有效。

因为计算机系统会不断成长，你不得不越来越操心控制依赖的问题。如果依赖超出了控制，对系统的每个改变会产生大范围的波动效应，导致不得不改变越来越多的东西。波动越大，越难改变所有东西。

UML允许你描绘所有类型元素之间的依赖。无论何时，如果你要展示一个元素中的改变会如何改变其他元素，你都可以使用依赖。

图3.7展示了一些你可能在多层应用中会发现的依赖。Benefits Window类——用户界面，或**表示（presentation）**类——依赖于Employee类：一个捕获系统本质行为（在这个例子里就是业务规则）的**领域对象（domain object）**。这意味着如果Employee类改变其接口，Benefits Window可能不得不改变。

图3.7 依赖的例子

这里重要的是依赖只有一个方向，从表示类到领域类。这样，我们知道可以自由地改变Benefits Window，而不会对Employee或其他领域的对象有任何影响。我已发现表示和领域逻辑严格分离，表示依赖于领域，但反之不是，这是一条值得遵从的、有价值的规则。

图3.7中第二件值得注意的事情是，从Benefits Window到两个Data Gateway类没有直接依赖。如果这些类改变，Employee类可能不得不改变，但如果改变仅限于Employee类的实现而不是接口，改变到此为止。

UML有许多种依赖，每一种都有特定的语义和关键词。我在这里列出我发现的最有用的基本依赖，我通常在使用依赖时不带关键词。为了添加更多细节，你可以添加一个合适的关键词（见表3.1）。

表3.1 可选择的依赖关键词

续表

基本的依赖不是可传递的关系。可传递的关系的例子是“大胡子”关系。如果Jim的胡子比Grady多，而Grady的胡子比Ivar多，我们可以推断Jim的胡子比Ivar多。某些种类的依赖，例如替换，是可传递的，但大多数情况下重要的是直接和间接依赖之间的区别，如图3.7所示。

许多UML关系隐含着依赖。图3.1中从Order到Customer的可导航的关联意味着Order依赖于Customer。子类依赖于它的超类，但反之不成立。

你的通用规则应该是最小化依赖，特别是依赖波及系统的大片面积时。特别是，你应该小心避免环状依赖，因为环状依赖会导致环状的

改变。不过对这一点，我不特别严格要求。我不介意紧密联系的类之间存在相互的依赖，但我确实尝试在更广的级别上消除环状依赖，特别是在包之间。

尝试展示类图中的所有依赖是吃力不讨好的；因为有太多依赖而且改变太频繁。只有当依赖直接和你要沟通的特定主题相关时，才有选择地展示它们。为了理解和控制依赖，你最好只在包图上使用依赖（109页）。

我使用类之间依赖的最常见情况是阐述瞬时的关系，例如当一个对象被传递给另一个对象作为参数。你可以看到会使用«parameter»、«local»和«global» 等关键词。你也可以在UML 1模型的关联上看到这些关键词，在这种情况下，这些关键词表示瞬时的链接，而不是性质。在UML 2中，不再有这些关键词。

依赖可以通过查看代码来决定，因此用工具做依赖分析是理想的做法。找到一个工具来做逆向工程，得到依赖图，是这里使用UML最有用的办法。

3.9 约束规则

画类图时，你正在做的大部分事情都是表明约束。图3.1指出一个Order（订单）只可以由一名Customer（顾客）来下。这个图也意味着每个Line Item（订单行条目）是分开考虑的：你说Order上有“40个棕色的装饰品、40个蓝色的装饰品和40个红色的装饰品”，而不是“120个东西”。此外，图中还说Corporate Customer有信用额度，但Personal Customer没有。

关联、属性和泛化的基本构造已经详细描述了许多重要的约束，但它们还不能表达每一个约束。这些约束依然需要捕获；类图是做这件事情的好地方。

UML允许你使用任何东西来描述约束，唯一的规则是你要把它们放进花括号（{}）中。你可以使用自然语言、编程语言或UML提供的形式对象约束语言（OCL）[Warmer and Kleppe]。OCL基于谓词演算。使用形式表示法避免由于含糊的自然语言造成误解的风险。然而，这也引进了由于作者和读者没有真正理解OCL导致误解的风险。因此除非读者熟悉谓词演算，否则我建议使用自然语言。

可选的做法是，你可以通过把名称放在前面，后面跟着冒号来命名约束。例如，{不允许乱伦：丈夫和妻子不能是兄妹、姐弟}。

按契约设计

按契约设计是由Bertrand Meyer [Meyer]开发的设计技术。该技术是他所开发的Eiffel语言的中心特性。不过，按契约设计不是Eiffel独有的，它是一项有价值的技术，可以用于任何编程语言。

按契约设计的核心是断言。**断言（assertion）**是一条决不应该为假的布尔陈述，因此，为假的唯一原因是存在bug。通常，断言只在调试期间检查，不在生产执行期间检查。事实上，程序决不应该假设断言正在被检查。

契约设计使用3种特定的断言：后置条件、前置条件和不变式。前置条件和后置条件应用于操作。**后置条件（post-condition）**陈述操作执行后世界看起来应该像什么样子。例如，我们定义数字上的操作“求平方根”，后置条件的形式为input=result * result，result是输出，input是输入值。后置条件是说我们做什么而不说我们怎么做的一种有用的方式。换句话说，就是将接口从实现分离。

前置条件（pre-condition）陈述我们期望在执行操作前世界是什么样子。我们可以定义“求平方根”操作的前置条件为input >=0。这样一个前置条件说明了在负数上调用“求平方根”是一个错误，这样做的结果没有定义。

乍一看，这看起来是个坏主意，因为我们应该把一些检查放在某些地方，以确保“求平方根”被适当地调用。重要的问题是谁负责这样做。

前置条件显式指出调用者负责检查。没有这个显式的责任陈述，我们可能检查得太少——因为双方都假设对方负责——或太多——双方都检查。太多检查是坏事，因为它会导致许多重复的检查代码，大大增加程序的复杂度。显式指出谁负责，有助于减少复杂度。由于这样一个事实，即断言通常在调试和测试期间检查，调用者忘记检查的危险减少了。

从前置条件和后置条件的这些定义，我们可以看到术语**异常（exception）**的强定义。在满足前置条件时，操作被调用，然而返回值如果没有满足后置条件，异常会发生。

不变式（invariant）是关于类的断言。例如，Account类可能有不变式balance==sum（entries.amount()）。对于类的所有实例来说，这个不变式“总是”为真。在这里，“总是”意味着“无论何时，只要对象身上可以发生操作调用”。

本质上，这意味着不变式添加到与给定类的所有公开操作相关联的前置条件和后置条件上。在方法的执行期间，不变式可以变为假，但当任何其他对象能够对接收者做任何事情时，不变式应该被重建为真。

断言在子类化中扮演了独一无二的角色。继承的危险之一是你可以重定义子类的操作，和超类的操作不一致。断言减少了这样的危险。类的不变式和后置条件必须应用于所有子类。子类可以选择加强这些断言，但不能减弱。另一方面，前置条件不能被加强，但可以被减弱。

乍一看这有点奇怪，但重要的是允许动态绑定。根据可替换性原则，你应该总是能够像对待超类的实例一样对待一个子类对象。如果一个子类加强了其前置条件，超类操作应用到子类时会失败。

3.10 何时使用类图

类图是UML的骨架，因此会发现你自己总是用到它。本章覆盖了类图的基本概念；第5章将讨论许多类图的高阶概念。

类图的麻烦是里面的东西太丰富了，可能会被过度使用。以下是一些提示。

- 不要尝试使用所有可用的表示法。从本章简单的东西开始：类、关联、属性、泛化和约束。只有在需要时才引进第5章的其他表示法。
- 我发现概念类图在探索业务语言上非常有用。为此，你不得不尽量把软件排除在讨论范围之外，保持表示法非常简单。
- 不要为每个东西都画模型；相反，集中于关键区域。有一些经常使用并保持更新的图，比有许多被遗忘的、过时的模型要好。

类图的最大危险是你可能只聚焦于结构，而忽略了行为。因此，当画类图来理解软件时，总是要结合某些形式的行为技术。如果进展顺利，你会发现自己频繁地在各种技术间切换。

3.11 更多资料

我在第1章提及的所有总体讨论UML的书籍都更详细地讨论了类图。对于更大的项目，依赖管理是关键的特性，这个主题的最佳书籍是[Martin]。

第4章 序列图

交互图（**interaction diagram**）描述对象组在某些行为中如何协作。UML定义了若干形式的交互图，最常见的是序列图。

通常，序列图捕获单个场景的行为。序列图展示用例内部的许多例子对象，以及这些对象之间传递的消息。

作为讨论的开始，考虑一个简单的场景。我们有一个订单，打算调用其上的一个命令来计算它的价钱。为此，需要查看订单上的所有订单行条目，决定它们的价钱，价钱基于订单行的产品的定价规则。针对所有订单行条目做这个计算后，订单需要计算出一个整体的折扣，折扣规则和顾客有关。

图4.1是一张展示该场景实现的序列图。序列图这样来展示交互：每一个参与者有一条生命线，垂直地沿着页面朝下运行，消息也沿着页面向下排序。

序列图的一个好处是，我几乎不必解释表示法。你可以看到，订单实例发送getQuantity和getProduct消息给订单行。你也可以看到我们如何展示订单调用自身的方法，以及该方法如何发送getDiscountInfo给一个顾客实例。

然而，这张图没有很好地展示每件事情。消息序列getQuantity、getProduct、getPricingDetails和calculateBasePrice需要由订单上的每一个订单行完成，而calculateDiscounts只被调用一次。你不能从这张图辨别出来，不过后面我将引进更多一些表示法来处理。

图4.1 一张中央控制的序列图

大多数时候，你可以把交互图中的参与者想象成对象，在UML 1中确实如此。但在UML 2中，它们的角色要复杂得多，完全解释这些角色已经超越了本书的范围。因此我使用术语**参与者（participant）**，一个没有在UML规格中正式使用的词。在UML 1中，参与者是对象，因此它们的名字带下画线，但在UML 2中，它们应该没有下画线，正如我在上图中所做的。

在这些图中，我使用anOrder形式来命名参与者。这在大多数时候都行得通。更完整的语法是name：Class，名字和类都是可选的，但如果你使用了类，必须留着冒号。（73页的图4.4使用了这种形式。）

每条生命线都有一根激活条，它展示在交互中参与者何时是活动的。激活条对应于栈中的参与者的方法之一。激活条在UML中是可选的，但我发现它对澄清行为极其有价值。我不用它的一个例外是在设计会议期间探索设计时，因为在白板上画激活条比较困难。

命名经常是有用的，可以把图上的参与者联系在一起。调用getProduct返回aProduct，名字和getPricingDetails调用发送到的aProduct一样，因此是同一参与者。注意：我只为这个调用使用一个返回箭头，这样做是为了展示其对应关系。有些人为所有调用使用返回，但我更喜欢只在能提供更多信息时使用；否则，它们只会把事情搞乱。即使在这个例子中不画出返回符号也不会使读者混淆。

第一个消息没有发送它的参与者，因为它来自一个不确定的源。它被称为**寻获消息（found message）**。

这个场景的另一种实现如图4.2所示。基本的问题依然一样，但参与者协作实现的方式有很大不同。Order请求每一个Order Line计算它自己的Price。Order Line本身又进一步把计算交给Product；注意我们如何展示参数的传递。类似地，计算折扣时，Order调用Customer上的方法。因为它需要来自Order的信息来做这件事情，Customer对Order做了一个重入调用（getBaseValue）来获得数据。

关于这两张图，第一件要注意的事情是，序列图如何清晰指出参与者如何交互的差异。这是交互图很棒的长处。序列图不擅长于展示算法的细节，例如循环和条件行为，但它们使得参与者之间的调用十分清晰，并且很棒地勾勒出了哪个参与者做了哪些处理的画面。

要注意的第二件事情是，两种交互之间风格的差异是清晰的。图4.1是**中央控制（centralized control）**，其中一个参与者负责做所有处理，其他参与者只提供数据。图4.2使用**分布控制（distributed control）**，处理分散在许多参与者之间，每一个参与者做一点点计算。

图4.2 分布控制的序列图

两种风格各自有各自的长处和弱点。大多数人，特别是刚使用对象的人，更习惯于中央控制。在许多方面，它更简单，因为所有处理放在一个地方；分布控制正好相反，在尝试发现程序的脉络时，你会有追着对象绕圈的感觉。

尽管如此，像我这样的对象偏执狂强烈地偏好分布控制。好的设计的一个主要目标是把改变的影响局部化。数据和访问数据的行为经常一起改变。因此，把数据和使用它的行为放在一个地方，是面向对象设计的第一原则。

更进一步，使用分布控制，你创造了更多使用多态而不是使用条件逻辑的机会。如果产品定价算法对于不同类型的产品是不同的，分布控制机制允许我们使用产品的子类来处理这些变化。

总体上，OO风格使用许多带有小方法的小对象，这样我们就有了许多覆盖和变化的插入点。这种风格对于习惯于长例程的人来说是非常困惑的；事实上，这个改变正是面向对象**范型转移（paradigm shift）**的核心。这些东西传授起来也很难。看起来真正理解它的唯一办法是在一个强制采用分布控制的OO环境里工作一段时间。许多人说他们“啊

哈”一声，突然理解了面向对象的风格。此时，他们的大脑重新连通，开始认为非中央控制实际上更容易。

4.1 创建和删除参与者

序列图（图4.3）展示了一些额外的表示法来创建和删除参与者。为了创建一个参与者，你把消息箭头直接画进参与者方框。如果在这里使用构造器，消息名是可选的，但我通常不管是什么情况都用“new”来标记它。如果参与者一旦被创建就立即做某些事情，例如查询命令，你可以在参与者方框的下面马上开始一个激活。

参与者的删除用一个大的×表示。一个消息箭头指向×表示一个参与者显式删除另一个；生命线末端的×展示一个参与者删除自身。

在一个有垃圾回收的环境中，你不直接删除对象，但依然值得使用×来表示一个对象不再被需要并准备回收。对关闭操作来说，加上×也是合适的，表明对象不再可用。

图4.3 创建和删除参与者

4.2 循环、条件等

序列图的一个常见问题是如何展示循环和条件行为。第一件要指出的事情是，这不是序列图所擅长的。如果要展示类似的控制结构，你最好使用活动图或者代码本身。把序列图当做对象如何交互的可视化，而不是建模控制逻辑的方式。

话是这么说，还是有表示法可以使用的。循环和条件都使用**交互框（interaction frame）**。交互框是标记一小块序列图的方法。图4.4展示了一个基于以下伪代码的简单算法：

图4.4 交互框

一般来说，框包含序列图的某些区域，序列图被划分成一个或多个片断。每个框有一个操作符，每一个片断可以有一个警戒条件。（表4.1列出了交互框的常见操作符。）为了展示循环，使用单个片断的loop操作符，把循环条件放在警戒条件的位置。对于条件逻辑，你可以使用alt操作符，把条件放在每一个片断之上。只有警戒条件为真的片断会执行。如果你只有一个区域，就使用opt操作符。

表4.1 交互框常见操作符

交互框是UML 2里新增的。因此，你可以看到UML 2之前的图使用了不同的做法；也有一些人不喜欢用框，更喜欢一些老的习惯。图4.5展示了一些非官方的变通方式。

UML 1使用迭代标记和警戒条件。**迭代标记（iteration marker）**是一个添加到消息名称的*号。你可以在方括号中添加一些文本，来表示迭代条件。**警戒条件（guard）**是一个放在方括号中的条件表达式，表示消息只有在警戒条件为真时发送。这些表示法已经从UML 2序列图中删除，但它们在通信图中依然是合法的。

图4.5 旧习惯表达控制逻辑

虽然迭代标记和警戒条件有所帮助，但它们确实有缺点。警戒条件不能表示互斥的警戒条件集合，例如图4.5中的那两个。两种表示法都只在单个消息发送时管用，如果在同一个循环或条件块内，有多个消息从单个激活发送出去，就不那么管用了。

为了处理这个问题，有一种流行的非官方习惯是使用**伪消息（pseudomessage）**，把循环条件或警戒条件放在自调用表示法的变体上。在图4.5中，我没有用消息箭头来展示；有些人会展示消息箭头，但不展示它更能增强它不是真实调用的印象。有些人也喜欢把伪消息的激活条加上阴影。如果你有多选一的行为，你可以在激活之间加上多选一标记来展示。

虽然我发现激活很有帮助，但在使用dispatch方法这种情况下，它们也不能起到太多作用。你只是靠它发送消息，在接收者的激活里面没有其他事情发生。通常，如图4.5所示，对于那些简单的调用应该把激活条去掉。

UML标准没有图形手段来展示传送的数据，而是通过消息名中的参数和返回箭头展示。许多方法会用**数据蝌蚪（data tadpole）**来表示数据的移动，许多人依然喜欢和UML一起使用它们。

总的来说，虽然序列图中添加了各种条件逻辑的表示法，但是我没有发现它们比代码或伪代码更管用。特别是，我发现交互框非常笨拙，会使图的要点变得模糊，因此我更喜欢伪消息。

4.3 同步和异步调用

如果你异常警觉，你应该已经注意到前两张图的箭头和之前的不同。这微小的差异在UML 2中十分重要。在UML 2中，实心箭头展示同步消息，条形箭头展示异步消息。

如果调用者要发送**同步消息（synchronous message）**，它必须等待，直到消息完成，就像调用一个子程序。如果调用者要发送**异步消息（asynchronous message）**，它可以继续进行其他处理，不必等待响应。你会在多线程应用和面向消息的中间件中看到异步调用。异步能带来更好的响应性，减少瞬时的耦合，但更难以调试。

箭头差异非常细微；事实上，太过细微了。它也和UML 1.4引进的改变后向不兼容，此前，异步消息用半箭箭头表示，如图4.5所示。

我认为箭头的区分太过细微。如果你要强调异步消息，我推荐使用已废弃的半箭箭头，这样更方便我们的眼睛分辨重要的区别。如果你正在阅读一张序列图，小心假设箭头是同步的，除非你确认作者有意做出区分。

4.4 何时使用序列图

当你要查看单个用例内若干对象的行为时，你应该使用序列图。序列图擅长于展示对象之间的协作；它们不太擅长于精确定义行为。

如果你要查看跨越许多用例的单个对象的行为，使用状态图（参见第10章）。如果你要查看跨越许多用例或许多线程的行为，考虑活动图（参见第11章）。

如果你要快速地探索多个多选一的交互，你最好使用CRC卡，这样可以避免许多绘制和擦除的工作。举行一个CRC卡会议来探索设计的可选方案通常很方便，然后再使用序列图捕获你在后面要引用的交互即可。

其他有用的交互图形式还有：通信图，用于展示连接；时间图，用于展示时间约束。

CRC卡

在思考如何得到好的OO设计时，最有价值的技能之一是探索对象的交互，因为这样做聚焦于行为而不是数据。由Ward Cunningham在20世纪80年代后期发明的CRC（Class-Responsibility-Collaboration）图，经过时间的考验，被认为是探索对象交互的极其高效的方法（图4.6）。

虽然它们不是UML的一部分，却在熟练的对象设计人员中间非常流行。

图4.6 CRC卡样例

要使用CRC卡，你和同事要聚在一张桌子旁。思考各种场景，用卡片演绎出来，把活跃的卡片举在空中，移动它们，表演它们如何发送消息给其他对象及四处传送对象。这项技能在书中描述出来几乎不可能，但很容易展示；学习它的最佳途径是让有经验的人展示给你看。

CRC式思考的一个重要部分是识别责任。一项**责任（responsibility）**是一个短句子，概括了对象应该做的某些东西：对象执行的动作、对象维护的一些知识或者对象做的一些重要决定。思路是，你应该能够拿起任何类，用一些责任来概括它。这样做能够帮助你更清楚地思考类的设计。

第二个C指**协作者（collaborator）**：需要和这个类一起工作的其他类。这样能帮助你思考类之间的链接——依然在高级别。

CRC卡的一个主要好处是，鼓励在开发人员中间做模拟讨论。当你围着一个用例看看类如何实现它时，本章中的交互图画起来就显得慢了。通常，你需要考虑备选方案；如果使用图，可能需要花很多时间来绘制和擦除。使用CRC卡，你可以拿起卡片并且移动它们来建模交互。这能让你快速地考虑备选方案。

在你这样做时，你形成了关于责任的想法，把它们写在卡片上。考虑责任是重要的，因为它能让你免于把类当成哑数据的持有者，让团队成员容易理解每一个类更高级别的行为。责任可能对应于一个操作、一个属性，或者更有可能的是，对应于不确定的一团属性和操作。

我所看到的常见错误是产生长长的低级别责任列表。这样做就迷失了目标。责任应该很容易放进卡片中。问问你自己这个类是否应该被分裂，或者责任是否应该上滚到更高级别的陈述。

许多人强调角色扮演的重要性，团队中的每个人扮演一个或多个类的角色。我从没有看到过Ward Cunningham这样做，我发现角色扮演比较碍事。

有专门写CRC的书，但我发现它们都没有真正描述这项技能的核心。CRC最原始的论文是由Kent Beck写的[Beck and Cunningham]。要了解更多关于CRC卡和设计中的责任的知识，可以看[Wirfs-Brock]。

第5章 类图：进阶概念

第3章描述的概念对应于类图中的关键表示法。这些概念要首先理解并逐渐熟悉，因为它们占了建造类图时90%的工作量。

然而，类图技能还繁殖出了许多用于附加概念的表示法。我不经常使用这些表示法，但发现它们在适用时用起来非常方便。本章我将逐个讨论并指出使用时的一些要点。

你可能会发现本章读起来有些困难。好消息是，如果你是第一次通读本书，你可以放心地略过本章，以后再回来阅读。

5.1 关键词

图形语言的挑战之一是，你不得不记住符号的含义。符号太多，用户会发现记住所有符号的含义非常困难。因此，UML经常尝试减少符号的数量，使用关键词来代替。如果你发现你需要一个UML中没有的建模构造，但UML有类似的东西，就使用现有的UML构造符号，但用关键词标注它，以展示某些不同。

例子之一就是接口。UML接口（**interface**）（86页）就是一个类，它只有公开的操作，但没有方法体。这对应于Java、COM（组件对象模块）和CORBA中的接口。因为它是一种特殊的类，它用带关键词«interface» 的类图标展示。关键词通常展示为双尖括号之间的文本。作为关键词的一个替代，你可以使用特别的图标，不过你又碰到新问题了：每个人必须记得图标的含义。

一些关键词，例如{abstract}，展示为花括号。技术上，到底在花括号中还是双尖括号中，没有清晰的定义。幸运的是，如果你搞错了，只有严重的“UML洁癖者”才会注意——或者在意。

一些关键词如此常用，所以它们经常被缩写：«interface» 经常被缩写为«I»，{abstract}经常被缩写为{A}。这样的缩写非常有用，特别是在白板上画图时。但它们不是标准，因此如果你使用缩写，确认你在某个地方写出它们的意思。

在UML 1中，双尖括号主要用于**构造型（stereotype）**。在UML 2中，构造型的定义非常严密，而描述什么是和什么不是构造型超越了本书的范围。然而，出于UML 1的原因，许多人使用时认为构造型的含义等同于关键词，虽然这样做不再是正确的。

构造型作为扩展机制的一部分使用。**扩展机制（profile）**为了特定目的使用一组相关的构造型（例如业务建模）来扩展UML的某个部分。扩展机制的完全语义超越了本书的范围。除非你很关心元模型设计，否则你不会需要创建你自己的扩展机制，而是使用已经为特定的建模目的创建的扩展机制，但幸运的是，使用扩展机制不要求你知道它们如何结合到元模型的很小的细节。

5.2 责任

通常，在类图中展示类的责任（79页）很方便。最佳途径是作为注释字符串在自己的栏中展示（图5.1）。如果你愿意的话，你可以给栏命名，但我通常不这样做，因为混淆的潜在可能性很小。

图5.1 在类图中展示责任

5.3 静态操作和属性

UML把应用于类而不是实例的操作或属性称为**静态的（static）**。这等同于基于C的语言中的静态成员。静态特性在类图中带下画线（见图5.2）。

图5.2 静态表示法

5.4 聚合和组合

UML中最频繁的混淆源头之一是聚合和组合。随口解释一下挺容易：聚合（**aggregation**）是“……的一部分”的关系。例如，汽车有引擎和轮子作为它的部件。这听起来挺好，但困难的是理解聚合和关联之间有什么差异。

在没有UML的时代，人们通常对什么是聚合和什么是关联比较模糊。不管是不是模糊，它们在每个人心里的定义也总是不一致的。许多建模人员认为聚合是重要的，但原因各不相同。因此，UML把聚合（图5.3）包含进来，但几乎没有任何语义。正如Jim Rumbaugh所说：“把它看做建模安慰剂。”[Rumbaugh，UML Reference]

图5.3 聚合

和聚合相比，UML对于组合（**composition**）的性质有更多的定义。在图5.4中，Point（点）的实例可以是Polygon（多边形）的一部分，或者可以是Circle（圆）的圆心，但不能二者皆是。通用规则是，虽然一个类可以是许多其他类的组件，但任何实例都必须是唯一拥有者的组件。类图可以展示多个类作为潜在的拥有者，但任何实例只能有单个对象作为它的拥有者。

图5.4 组合

你会注意到：我没有在图5.4中展示反方向的多重性。在大多数情况下，例如在这里，是0..1。它的唯一其他可能值是1，这样设计组件类，使得它只能有一个其他类作为它的拥有者。

“无分享”规则是组合的关键。另一个假设是，如果你删除Polygon，它应该自动地确保任何它所拥有的Point也被删除。

组合是展示按值拥有性质、值对象（91页）性质或者对于其他特定组件有强的排他的所有关系的性质的好办法。聚合没有严格的含义；因此，我推荐你在图中忽略它。如果你在别人的图里看到聚合，你需要挖得更深，以发现它们的含义。不同的作者和团队会出于非常不同的目的而使用它。

5.5 派生性质

派生性质（derived property）可以基于其他值计算出来。当我们考虑一个日期范围时（图5.5），我们可以思考3个性质：开始日期、结束日期和这段时期内的天数。这些值是相互关联的，因此我们可以认为日期长度由另外两个值派生。

图5.5 time period内的派生属性

软件视角中的派生可以用几种不同的方式解释。你可以使用派生来指明计算值和存储值之间的区别。在这个例子中，我们将图5.5解释为开始日期和结束日期是存储值，但日期长度是计算值。虽然经常这样用，但我不是很喜欢，因为它揭露了DateRange内部太多的东西。

我偏好的思路是，它指明了值之间的一种约束。在这个例子中，指的就是三个所持有的值之间的约束，但这三个值中的哪一个是计算的并不重要。在这个例子中，选择哪一个属性标记为派生是随意的，没有

必要很严格，不过派生是有用的，它能帮助提醒人们其中的约束。这个用法也适用于概念图。

使用关联表示法，派生也可以应用于性质。在这个例子中，你可以简单地在名称上标记一个/。

5.6 接口和抽象类

抽象类（abstract class）是指不能被直接实例化的类。然而，你可以实例化其子类的实例。通常，抽象类有一个或多个抽象操作。**抽象操作（abstract operation）**没有实现，它是纯声明，所以客户不能绑定到抽象类。

在UML中指明一个抽象类或操作，最常见的方式是把名称变成斜体。你也可以使性质变成抽象的，说明这是抽象的性质或存取器方法。在白板上画斜体很麻烦，因此你可以使用标签{abstract}。

接口是没有实现的类，即所有特性都是抽象的。接口和C＃及Java中的接口有直接的对应关系，在其他类型化语言中也是常见的概念。接口用关键词«interface» 来标记。

类跟接口之间有两种关系：供给和需求。如果类可以替换接口，那么就说类**供给一个接口（provides an interface）**。在Java和.NET中，类可以通过实现接口或实现接口的子类型来做到。在C++中，通过继承接口类来做到。

如果类需要一个接口的实例才能工作，那么，类**需要一个接口（requires an interface）**。本质上，就是依赖于该接口。

图5.6基于Java的一些集合类展示了这些关系。我可以写一个Order（订单）类，它有一个订单行条目列表。因为我在使用列表，Order类依赖于List接口。让我们假设它使用方法equals、add和get。当对象连接

时，Order实际上使用一个ArrayList实例，但不需要为了使用那三个方法而知道这一点，因为它们都是List接口的一部分。

ArrayList本身是AbstractList类的子类。AbstractList提供一些List行为的实现，但不是所有的实现。特别地，get方法是抽象的。因此，ArrayList实现get而且覆盖AbstractList上的其他一些操作。在这个例子中，它覆盖了add，但很乐意继承equals的实现。

为什么我不简单地避免这样做，让Order直接使用ArrayList？通过使用接口，后面需要时更容易改变实现。另一个实现或许可能提供性能改善、数据库交互特性或者其他好处。通过对接口而不是对实现编程，如果需要一个List的不同实现，我不必改变所有代码。你应该总是尝试像这样对接口编程，总是使用你能用到的最通用的类型。

图5.6 接口和抽象类的Java例子

在这里，我也应该指出一个实用的诀窍。当程序员使用集合时，他们通常用一个声明初始化该集合，像这样：

注意，这样做绝对会引进从Order到具体ArrayList的依赖。理论上，这是一个问题，但人们在实践中不必担心它。因为lineItems的类型声明为List，没有Order类的其他部分依赖于ArrayList。如果我们要改变实现，只有一行初始化代码需要改变。这是十分常见的：在创建期间一次性地引用一个具体类，但随后只使用接口。

图5.6的完全表示法是表示接口的一种方式。图5.7展示了更简洁的表示法。ArrayList实现List和Collection的事实通过小球图标（这经常被称为棒棒糖）展示。Order需要List接口的事实通过球窝图标展示。它们之间的关系通过箭头表示。

图5.7 小球-球窝表示法

UML使用棒棒糖表示法已经有一段时间，但球窝表示法是UML 2新引进的。（我想这是我喜欢的新增表示法。）窝球是可选的，所以你经常会看到如图5.8所示的风格的图。当你使用第13章中讨论的组合结构的部件时，可以搭配使用小球和窝球——但是你只能在部件中使用。

图5.8 老的带依赖的棒棒糖表示法

任何类都是接口和实现的混合。因此，我们会经常看到一个对象通过它的一个超类的接口被使用。严格地说，为超类使用棒棒糖表示法是不合法的，因为超类是一个类，不是一个纯接口。但为清晰起见，我忽略了这些规则。

除了类图之外，人们发现棒棒糖表示法在其他地方也有用。交互图长久以来的问题之一是它们没有为多态行为提供非常好的可视化表示法。虽然不是规范的用法，你可以通过图5.9中的线看到这一点。在这里，我们可以看到，虽然我们有一个Salesman（销售员）实例被Bonus Calculator（红利计算器）使用，Pay Period（付款周期）对象只通过它的Employee（雇员）接口使用Salesman。（你可以在通信图里做同样的处理。）

图5.9 使用棒棒糖表示法在序列图中展示多态

5.7 只读和冻结

在45页，我描述了{readOnly}关键词。你使用这个关键词来标记客户只能读取不能更新的性质。和它相似但不同的是来自UML 1的{frozen}关键词。如果性质在对象的生命期内不能改变，它就是**冻结的（frozen）**；这样的性质经常被称为不可变的。虽然已经从UML 2中删除，{frozen}还是一个非常有用的概念，因此我愿意继续使用它。你既可以标记个别性质为冻结的，也可以把该关键词应用到一个类，来指明所有实例的所有性质都是冻结的。（我听说冻结可能很快又会在标准中恢复。）

5.8 引用对象和值对象

关于对象，一件常说的事情是它们有标识。这是真的，但并不是那么简单。在实践中，你会发现标识对引用对象是重要的，但对值对象就没那么重要。

引用对象（reference object）的例子是Customer（顾客）。在这里，标识非常重要，因为你通常只需要一个软件对象来指代现实世界中的一名顾客。任何引用Customer对象的对象会通过引用或指针来做到；所有引用这个Customer的对象引用的是同一个软件对象。这样，改变Customer对该Customer的所有用户都会有影响。

如果你有两个到Customer的引用，希望看看它们是否是同一个，你通常会比较它们的标识。副本应该是不允许的；如果允许也要尽可能少，也许为了存档或者为了跨网络复制会制造副本。如果制造了副本，你需要想办法同步改变所有副本。

值对象（value object）就是像Date这样的东西。你经常用多个值对象代表现实世界中的同一个对象。例如，存在数百个指代1-Jan-04的对象是正常的。这些都是可以互换的副本。新的日期（Date）会被频繁创建和销毁。

如果你有两个日期，希望看看它们是否相同，你不会查看它们的标识，而是查看它们代表的值。这通常意味着你不得不编写相等测试操作符，对于日期来说，要测试年、月、日——或任何的内部表示。每个引用1-Jan-04的对象通常都有自己专用的对象，但你也可以共享日期。

值对象应该是不可变的；换句话说，你应该不能获得一个日期对象1-Jan-04，然后改变为2-Jan-04。相反，你应该创建一个新的2-Jan-04对象并使用它。这样做的原因是，如果日期被共享，你会以不可预测的方式更新另一个对象的日期，这个问题称为**走样（aliasing）**。

随着时代的发展，引用对象和值对象之间的区别更加清晰了。值对象是类型系统的内建值。现在你可以用你自己的类扩展类型系统，因此这个问题需要进行更多的思考。

UML使用**数据类型（data type）**的概念，作为类符号上的关键词。严格地说，数据类型和值对象不是一个东西，数据类型不能有标识。值对象可以有标识，但不要使用标识来判断相等。Java中的原生（primitive）类型就是数据类型，但日期不是，虽然它们是值对象。

如果强调它们很重要，在关联到值对象时，我使用组合。你也可以在值类型上使用一个关键词，我所看到的通常惯例是«value» 或«struct»。

5.9 限定关联

限定关联（qualified association）是UML里关于数组、图、哈希和词典这些编程概念的等同物。图5.10展示了使用限定符来表达Order（订单）和Order Line（订单行）类之间的关联的一种方式。限定符说明在和一个Order的连接中，对于每一个Product（产品）实例，可能会有一个Order Line。

图5.10 限定关联

从软件的视角看，限定关联暗示以下接口：

这样，到给定Order Line的所有访问都需要一个Product作为参数，说明这是一个使用键/值数据结构的实现。

人们经常对限定关联的多重性比较困惑。在图5.10中，一个Order可能有许多Line Item，但限定关联的多重性是限定符上下文中的多重性。因此该图说明一个Order针对每个Product有0..1个Line Item。多重性为1表示Order必须为每个Product实例准备一个Line Item。*表示每个Product会有多个Line Item，但到Line Item的访问由Product索引。

在概念建模中，我只在为了展示约束“在Order上每个Product有单个Order Line”时，才使用限定符构造。

5.10 分类和泛化

我经常听到人们谈论，说子类型化为是一个（is a）关系，我极力主张在这么想的时候要小心，因为是一个可以有不同的含义。

考虑以下语句。

1．Shep是一只博德牧羊犬。

2．一只博德牧羊犬是一只犬。

3．犬是动物。

4．一只博德牧羊犬是一个品种。

5．犬是一个物种。

现在尝试结合语句。如果我结合语句1和2，得到“Shep是一只犬”；结合语句2和3，得到“博德牧羊犬是动物”；结合1、2和3，得到“Shep是一只动物”。目前为止，一切都好。现在尝试结合语句1和4，得到“Shep是一个品种”。结合语句2和5，得到“博德牧羊犬是一个物种”这些就不那么合理了。

为什么我可以结合一些语句，而不能结合另一些？原因是一些是**分类（classification）**——对象Shep是类型博德牧羊犬的一个实例——一些是**泛化（generalization）**——类型博德牧羊犬是类型犬的子类型。泛化是传递性的，分类则不是。可以在分类后跟着泛化，但反之不然。

我说这些的意思是让你小心是一个。使用它会导致不合适地使用子类型化及混淆责任。在这个例子中，更好的子类型化测试是语句“犬是一种动物”和“每个博德牧羊犬实例是犬的实例”。

UML使用泛化符号来展示泛化。如果你要展示分类，需要使用带«instantiate» 关键词的依赖。

5.11 多重和动态分类

分类（classification）指对象和它的类型之间的关系。主流编程语言假设一个对象属于单个类。此外分类还有更多选项。

在**单个分类（single classification）**中，对象属于单个类型，可能从超类型继承。在**多重分类（multiple classification）**中，对象可以由若干类型描述，这些类型未必通过继承连接。

多重分类不同于多重继承。多重继承说一个类型可以有许多超类型，但必须为每一个对象定义单个类型。多重分类允许一个对象有多个类型，不需要为此而定义一个特定的类型。

例如，考虑人的子类型，可能是男人或女人、医生或护士、病人或非病人（参见图5.11）。多重分类允许任何这些类型以任何允许的结合方式分配到一个对象，不需要为所有合法的结合定义类型。

图5.11 多重分类

如果你使用多重分类，你需要确定你清楚哪些结合是合法的。UML 2通过把每一个泛化关系放进一个**泛化集（generalization set）**来做到这一点。在类图上，在泛化箭头上加上泛化集名称的标签，在UML 1中称为鉴别器。单个分类对应于单个不命名的泛化集。

泛化集默认是不相交的：任何超类型的实例可以是泛化集中唯一一个子类型的实例。如果你把泛化合并成单个箭头，它们必须都是同一泛化集的一部分，如图5.11所示。或者，你可以使用具有一个相同文本标签的若干箭头。

为了阐述，注意图中以下合法的子类型结合：（Female，Patient，Nurse）；（Male，Physiotherapist）；（Female，Patient）和（Female，Doctor，Surgeon）。结合（Patient，Doctor，Nurse）是非法的，因为它包含了两个来自角色泛化集的类型。

另一个问题是，对象是否可以改变它的类。例如，当一个银行账户透支，它的行为将大大改变。特别是，若干操作，包括“取款”和“关闭”被覆盖。

动态分类（dynamic classification）允许对象在子类型化结构内改变类；**静态分类（static classification）**不允许。使用静态分类，类型和状态被分离，动态分类结合了这些概念。

你是否应该使用多重、动态分类？我相信对于概念建模是有用的。然而，从软件视角看，这些概念和实现之间有太大的鸿沟要跨越。在大

多数UML图中，你只会看到单个、静态的分类，所以这应该是你默认的做法。

5.12 关联类

关联类（**association class**）允许你给关联添加属性、操作和其他特性，如图5.12所示。从图中我们可以看到，一个人可以参加许多会议。如果我们需要记录人们开会的时候有多清醒，我们可以通过给关联添加属性attentiveness（专注程度）来做到。

图5.13展示了表示这个信息的另一个方法：使Attendance（出席）变成一个有自己的权利的完整的类。注意，多重性是如何转移的。

对关联类这样做有好处，但是代价是你不得不记住额外的表示法。关联类添加了一个额外的约束，在任何两个参与对象之间只能有一个关联类实例。我感觉需要另一个例子。

图5.12 关联类

图5.13 提升关联类为完整的类

看一看图5.14中的两张图。这两张图的很多地方是一样的。然而，我们可以想象一个Company（公司）在同一个Contract（合同）中扮演不同的角色，但很难想象一个Person（人）在同一种技能上有多个Competency（能力）；事实上，你可能认为那是一个错误。

在UML中，只有后一种情况是合法的。对于每一个Person和Skill的结合，你只可以有一个Competency。图5.14上部的图形不允许一个

Company在单个Contract上有多于一个的Role。如果你想允许这样做，你需要把Role变成一个完整的类，就像图5.13那样。

实现关联类的方式并不是很明显。我的建议是，就像它是一个完整的类一样实现关联类，但提供获取信息的方法给关联类链接的类。因此对于图5.12，我会为Person类创建以下方法：

图5.14 关联类的细微差别（角色不应该是关联类）

这样，Person的客户可以知道参加会议的人；如果需要细节，可以获取Attendance。如果你这样做，记得强制执行这个约束：对于任何一对Person和Meeting来说，只有一个Attendance对象。你应该在每一个创建Attendance的方法里放置一个检查。

在涉及保存历史信息时，你经常会发现这种构造，例如在图5.15中。然而，我发现创建额外的类或关联类会使模型变得更难理解，还有，将实现向特定方向倾斜通常也是不合适的。

图5.15 用一个类表达瞬时关系

如果我有这种瞬时信息，我会在关联上使用一个 «temporal» 关键词（参见图5.16）。该模型指明，一个Person在一个时间只可以为一个Company工作。然而，随着时间的推移，一个Person可以为若干Company工作。建议使用以下接口：

图5.16 为关联使用 «temporal» 关键词

«temporal» 关键词不是UML的一部分，但我在这里提及它有两个原因。首先，在我的建模生涯中的若干场合里，我发现它是一个有用的概念。其次，它展示了可以如何使用关键词来扩展UML。你可以在这里看到更多关于这方面的资料：
http://martinfowler.com/ap2/timeNarrative.html。

5.13 模板（参数化）类

一些语言有参数化类（**parameterized class**）或模板（**template**）的概念，最著名的是C++。（模板在不久的将来会出现在Java和C＃中。）

这个概念对使用强类型语言的集合用处最明显。这样，通过定义模板类Set，你可以为集合定义通用的行为。

这样做以后，你可以使用通用定义来为更特定的元素做Set类：

在UML中，你通过使用如图5.17所示的表示法声明一个模板类。图中的T是一个类型参数的占位符（可以多于一个）。

图5.17 模板类

参数化类的使用，例如Set<Employee>，称为派生（**derivation**）。你能够用两种方式展示派生。第一种方式借鉴C++的语法（参见图5.18）。在尖括号中描述派生表达式，形式为<parameter-name：：parameter-value>。如果只有一个参数，习惯上经常省略参数名称。另一种表示法（参见图5.19）加强了到模板的链接，允许重命名绑定元素。

图5.18 绑定元素（版本1）

图5.19 绑定元素（版本2）

«bind» 关键词是精化关系上的一个构造型。这个关系指明EmployeeSet会遵从Set接口。你可以认为EmployeeSet是Set的子类型。这适合实现类型特定集合的其他方式，其他方式声明所有合适的子类型。

但是，使用派生和子类型化有不同之处，它不允许给绑定元素添加特性，绑定元素完全由它的模板规定；添加的只是限定类型的信息。如果要添加特性，必须创建一个子类型。

5.14 枚举

枚举（图5.20）用于展示一个固定集合的值，这些值除了它们的符号值没有其他任何性质。它们表示为带 «enumeration» 关键词的类。

图5.20 枚举

5.15 主动类

主动类（active class）有实例，每个实例执行和控制它自己的控制线程。方法调用可以在客户的线程或主动对象的线程中执行。一个好的例子是命令处理器，它从外部接受命令对象，然后在它自己的控制线程内执行命令。

从UML 1到UML 2，主动类表示法有变化，如图5.21所示。在UML 2中，主动类在两侧有额外的竖线；在UML 1中，它用粗边框表示，称为主动对象。

图5.21 主动类

5.16 可见性

可见性（visibility）这个主题在原则上简单，但在细节上很复杂。简单的认识是，任何类都有公开和私有的元素。公开元素可以被任何其他类使用；私有元素只能被拥有它的类使用。然而，每一种语言都有它自己的规则。虽然许多语言使用的术语类似，如public、private和protected，但它们在不同的语言里代表着不同的东西。这些差异很小，但会导致混淆，特别是对我们这样使用多种语言的人而言。

UML尝试在不会造成可怕的混淆的情况下解决这个问题。本质上，在UML中，你可以用可见性标记任何属性或操作。你可以使用任何你喜欢的标记，它的含义依赖于语言。UML提供4个可见性缩写：**+**（公开）、-（私有）、~（包）和#（保护），这4个级别在UML元模型内使用，也在UML元模型内定义，但它们的定义随着语言变化会有微小的不同。

当你使用可见性时，要使用你正在使用的语言的规则。当你从别处查看UML模型时，小心可见性标记的含义，那些含义会随着语言的不同而改变。

大多数时候，我在图中不画可见性标记；我只在需要强调某个特性的可见性差异时使用。即使如此，我也尽可能使用+和-，至少容易记住。

5.17 消息

标准UML不在类图上展示关于消息调用的任何信息。然而，我有时还是会看见像图5.22那样的传统的图。

图5.22 带消息的类

图中在关联旁边添加了箭头。箭头上的标签写着一个对象发送到另一个对象的消息。因为你不需要到类的关联来发送消息给类，你也可能需要添加依赖箭头，以展示没有关联的类之间的消息。

这个消息的信息跨越多个用例，因此它们没有编号来展示序列，不像通信图。

第6章 对象图

对象图（**object diagram**）是某时间点上的对象在系统中的快照。因为它展示实例而不是类，对象图经常被称为实例图。

你可以使用对象图来展示对象的配置案例（参见图6.1，它展示一组类，图6.2展示一组相互关联的对象集合）。当对象之间的可能连接比较复杂时，后者非常有用。

图6.1 Party组合结构的类图

你可以辨别出图6.2中的元素是实例，因为名称带下画线。每个名称的形式都是“实例名 ： 类名”。两部分都是可选的，因此，John、：Person和aPerson都是合法的名称。如果你只使用类名，则必须包含冒号。还可以显示属性和链接的值，如图6.2所示。

图6.2 展示Party实例的对象图

严格地说，对象图的元素是实例规格而不是真正的实例，因为强制属性为空或者展示抽象类的实例规格都是合法的。你可以认为实例规格（**instance specification**）是部分定义的实例。

还可以将对象图看做没有消息的通信图（159页）。

6.1 何时使用对象图

对象图对于展示连接在一起的对象很有用。在许多情形下，虽然可以通过类图精确地定义结构，但该结构依然难以理解。此时，使用对象图会大有帮助。

第7章 包图

类代表面向对象系统结构的基本形式。虽然类十分有用，但你还需要更多东西来构造大型系统的结构，因为大型系统可能有上百个类。

包（package）是一种分组构造，它允许你选择UML里的任何构造，把它的元素组织在一起，成为更高级别的单元。包最常见的用法是组织类，所以我紧接着类图讲解包图，但要记住，你也可以为UML的其他元素使用包。

在一个UML模型中，每一个类都是单个包的成员。包也可以是其他包的成员，因此你可以得到一个具有层级的结构，顶层包分解为子包，而子包又有它们自己的子包等，直到层级的底部只有类。一个包中可以包含子包和类。

用编程术语来说，包对应于包（Java）和命名空间（C++和.NET）这样的分组构造。

每个包代表一个**命名空间（namespace）**，命名空间意味着每一个类在拥有它的包里必须有唯一的名称。如果我要创建一个称为Date的类，而在System包中已经存在一个Date类，我仍然可以有我的Date类，只要我把它放在另一个包中即可。为了说清楚我指的是哪一个Date类，我可以使用**完全限定名称（fully qualified name）**，即指出拥有它的包结构的名称。在UML中，使用双冒号来展示包名称，因此Date可能是System：：Date和MartinFowler：：Util：：Date。

在图中，包展示为带标签的文件夹，如图7.1所示。你可以简单地展示包名或展示包名和内容。在任何时候，你都可以使用完全限定的名称或者只用常规名称。在内容中展示类图标允许你展示类的所有细节，

甚至可以在包里展示类图。如果你要做的只是指出哪个类在哪个包里，简单地列出名称就足够了。

图7.1 在图上展示包的方式

经常见到带有某些标签的类，而不是完全限定的形式，像Date（from java.util）。这种风格是被Rational Rose大量使用的习惯；它不是标准的一部分。

UML允许包中的类是公有的或私有的。公有的类是包接口的一部分，可以被其他包中的类使用；私有类是隐藏的。关于包构造之间的可见性，不同的编程环境有不同的规则；你应该遵从编程环境的习惯，即使这意味着不符合UML的规则。

在这里，一项有用的技能是，通过只输出一个小的子集来减少包的接口，子集里面的操作都和包公开的类有关联。你可以把所有类的可见性定为私有的，这样它们只能被同一个包的其他类看到，然后为公有行为添加额外的公有类，这些额外的类称为门面（Facade）[Gang of Four]，然后把公开的操作委托给包中不对外暴露的类。

你如何选择哪个类放在哪个包里？这确实是一个十分复杂的问题，需要很多设计技巧。这有两个有用的原则：共同封闭原则和共同复用原则[Martin]。共同封闭原则说明包中的类应该由于相似的原因而改变。共同复用原则说明包中的类应该一起被复用。把类按包分组的很多原因和包之间的依赖有关，接下来我将讨论这一点。

7.1 包和依赖

包图（package diagram）展示包和包之间的依赖。在58页，我介绍了依赖的概念。如果存在表示包和领域包，且表示包中的类依赖于领域

包中的类，那么从表示包到领域包有依赖。这样，包之间的依赖就概括了它们包含的内容之间的依赖。

UML有许多种类型的依赖，每个都有特定的语义和构造型。我发现从没有加构造型的依赖开始更加容易，并且只在需要时使用更加特定的依赖，不过我几乎没有这样做过。

在中型到大型系统中，绘制一张包图是你可以控制系统的大规模结构的最有价值的事情之一。在理想情况下，这张图应该从代码基生成，这样通过这张图你可以看到系统里真正有什么。

好的包结构有清晰的依赖流，依赖流这个概念很难定义，但一般很容易识别。图7.2展示了一张包图的例子，这是一个企业应用，这张包图结构良好，并且有清晰的依赖流。

图7.2 企业应用包图

通常，你可以识别出一个清晰流，因为所有依赖运行在单个方向上。这是结构良好的系统的良好指示器，但图7.2中的数据映射器包展示了这个经验法则的一个异常。数据映射器包作为领域和数据库包之间的隔离层，是映射器模式[Fowler，P of EAA]的一个例子。

许多作者说应该没有环状依赖（无环依赖原则[Martin]）。我不把它当做一个绝对的规则，但我确实认为环应该局限在一个范围内，特别是，你不应该有跨层的环。

有越多依赖进入一个包，该包的接口就需要越稳定，因为接口的任何改变都会波动到所有依赖于它的包（稳定依赖原则[Martin]）。因此在图7.2中，比起leasing data mapper（租赁数据映射器）包，asset domain（资产领域）包需要更稳定的接口。通常，你会发现更稳定的包倾向于有更高比例的接口和抽象类（稳定抽象原则[Martin]）。

依赖关系不是传递性的（60页）。如果想要知道为什么这对依赖来说很重要，请再看一下图7.2。如果一个asset domain包中的类改变了，我们可能要改变leasing domain（租赁领域）包内部的类，但这个改变不一定波动到leasing presentation（租赁表现）包（它只有在leasing domain包改变其接口时会被波动）。

一些包会在许多地方被使用，如果画出所有依赖线，会显得很乱。在这个例子中，习惯是在包上使用关键词，例如 «global»。

UML包也定义构造，以允许从一个包到另一个包导入及合并类，这时使用带依赖的关键词来表示。然而，这种东西的规则随着编程语言的不同变化很大。总体上，我发现通称“依赖”在实践中更有用。

7.2 包的分解

如果你考虑图7.2，你会认识到这张图有两种结构。一种是应用的分层结构：表现、领域、数据映射器和数据库。另一种是主题区域结构：租赁和资产。

你可以通过分离这两种结构使得这一点更清晰，如图7.3所示。在这张图中，你可以清晰地看到每一种结构。然而，这两个部分都不是真正的包，因为你不能分配类给单个包（你将不得不从每个部分挑选一个包）。这个问题对应于编程语言的层级命名空间问题。虽然类似图7.3这样的图形不是标准UML，但它们经常在解释复杂应用的结构上很有帮助。

图7.3 把图7.2分离成两个部分

7.3 实现包

通常，你会看到这样的情况，一个包定义了一个可以被许多其他包实现的接口，例如图7.4所示的包。这个例子中的实现关系指出，Database Gateway（数据库入口）定义了一个接口，其他入口类提供实现。在实践中，这意味着Database Gateway包所包含的接口和抽象类完全由其他包实现。

图7.4 被其他包实现的包

接口和它的实现被分离到不同的包非常普遍。事实上，客户包经常包含接口，另一个包包含实现：和我在88页讨论的需求接口概念是一样的。

想象一下我们要提供一些用户界面（UI）控件来打开和关闭某些东西。我们需要控件可以和许多不同的东西一起工作，例如加热器和灯。UI控件需要调用Heater（加热器）上的方法，但我们不想让该控件依赖于Heater。我们可以在控件包中定义一个接口来避免这个依赖，然后这个接口由任何想要和这些控件一起工作的类实现，如图7.5所示。这是一个分离接口模式[Fowler，P of EAA]的例子。

图7.5 在客户包中定义一个需求接口

7.4 何时使用包图

我发现包图对大规模系统极其有用。通过包图可以获得系统主要元素之间的依赖关系。这些图形和常见的编程结构对应得很好。绘制包和依赖的图可以帮助你保持对应用依赖的控制。

包图代表编译时的分组机制。要展示对象在运行时如何组合，应使用组合结构图（163页）。

7.5 更多资料

关于包和如何使用包，我所知道的最好的资料是[Martin]。Robert Martin长久以来对依赖有接近于病态的痴迷，写了许多如何留心依赖以便可以控制和最小化依赖的文章。

第8章 部署图

部署图展示系统的物理布局，揭示哪个软件运行在哪个硬件上。部署图真的非常简单，因此这一章很短。

图8.1是一个简单的部署图。图上的主要条目是由通信路径连接的节点。**节点（node）**是上面能驻留一些软件的环境。节点有两种形式。**设备（device）**是硬件，它可以是计算机或更简单的、连接到一个系统的硬件部件。**执行环境（execution environment）**是软件，它本身作为软件或包含其他软件，例如操作系统或容器进程。

节点包含**工件（artifact）**，工件是软件的物理显现：通常是文件。这些文件可能是可执行的（例如.exe文件，二进制、DLL和JAR文件，程序集或脚本），或数据文件、配置文件、HTML文档等。列出节点内的一个工件，表示在运行时的系统中，工件部署到该节点。

可以用类方框，或者通过列出节点内的名称来展示工件。如果展示为类方框，可以添加一个文档图标或者 «artifact» 关键词。你可以用标记值标记节点或工件，来表示各种关于节点的有趣信息，例如厂商、操作系统、位置或任何其他你感兴趣的事情。

图8.1 部署图实例

通常，会有多个物理节点执行同一个逻辑任务。你既可以用多个节点框，也可以用标记值数字展示。在图8.1中，我使用标记number deployed来表示三个物理的Web服务器，但对于标记并没有标准。

工件经常是组件的实现。为了展示这一点，你可以在工件框中使用标记值。

节点之间的通信路径表示如何沟通。你可以给这些路径加上标签，标签上有所用通信协议的信息。

8.1 何时使用部署图

不要因为本章简短让你认为不应该使用部署图。它们非常方便展示什么地方部署了什么，因此，复杂一点的部署都可以好好利用部署图。

第9章 用例

用例是捕获系统功能需求的技能。用例描述系统用户和系统本身的典型交互，这样就提供了系统如何被使用的说明。

比起正面描述用例，我发现通过描述场景来从侧面慢慢接近它更加容易。**场景（scenario）**是描述用户和系统之间交互的步骤序列。因此，如果我们有一个基于Web的在线商店，可以有一个这样的“购买产品”场景：

顾客浏览目录并添加想要的条目到购物车。当顾客想要付款时，顾客描述配送和信用卡信息并确认购买。系统检查信用卡授权并立即确认此次交易，随后发送一封确认E-mail。

这个场景描述的是一件能够发生的事情。然而，信用卡授权会失败，这会产生另一个场景。在另一种情况下，你可能有一个固定顾客，你不需要捕获配送和信用卡信息，这是第三个场景。

所有这些场景不同但又相似。它们相似的本质是，在所有这三个场景中，用户都有同一个目标：购买产品。用户不会总是成功，但目标是一样的。这个用户目标是用例的关键：**用例（use case）**是通过共同用户目标绑在一起的场景集合。

在用例的说法中，用户叫做执行者。**执行者（actor）**是用户扮演的跟系统相关的角色。执行者可能包括顾客、客服代表、销售经理和产品分析师。执行者执行用例。单个执行者可以执行许多用例；反过来，一个用例可能有若干执行者执行。多人可以扮演一个执行者。一个人也可以扮演执行者，例如一个销售经理做了客服代表的工作。执行者不一定是人。如果系统为另一个计算机系统提供服务，其他系统就是执行者。

执行者其实不是合适的术语；角色要好得多。显然，这是一个来自瑞典语的错译，执行者是用例社区使用的术语。

用例被普遍认为是UML重要的一部分。然而，令人惊讶的是，UML中用例的定义相当简略。UML里没有描述你应该如何捕获用例的内容。UML所描述的是用例图，展示用例之间的相互关系。但几乎用例所有的价值都在于它的内容，图的价值相当有限。

9.1 用例的内容

书写用例的内容没有标准的方式，不同的格式适用于不同的情况。图9.1展示了常用的风格。一开始，你挑选场景之一作为**主成功场景（main success scenario）**。接着，你开始书写用例体，把主成功场景书写为编号的步骤序列。然后，你再书写其他场景作为**扩展（extension）**，描述主成功场景上的变化。扩展可以继续扩展——直到用户达到目标，如3a——或失败，如6a。

每个用例有一个主执行者，它调用系统来交付一个服务。主执行者是带有目标的执行者，用例会尝试满足它的目标。主执行者通常是（但不总是）用例的引发者。执行用例时，还有其他执行者和系统进行通信，这些称为辅助执行者。

图9.1 用例文本实例

用例中的每个步骤是执行者和系统之间的交互元素。每个步骤应该是一个简单的陈述，清晰地展示谁执行该步骤。步骤应该展示执行者的意图，而不是执行者如何做的具体细节。因此，你不用在用例中描述用户界面。事实上，通常在设计用户界面之前书写用例。

用例中的扩展命名了一个条件，这个条件导致和主成功场景（MSS）所描述的不同交互，并陈述了差异是什么。扩展首先指出条件被检测的步骤，并提供短的条件描述。条件之后就是步骤，编号风格和主成功场景是一样的。如果你愿意，可以通过描述在何处返回到主成功场景完成这些步骤。

用例结构是用头脑风暴法产生主成功场景的备选方案的很棒的办法。针对每一个步骤询问：这可以有什么不同的情况？特别是，什么地方会出错？通常的最佳做法是，首先用头脑风暴法产生所有扩展条件，先不要受结果困扰。这样，你可能会想到更多条件，后面要收拾的烂摊子就会少一些。

用例中的复杂步骤可以变成另一个用例。在UML术语中，我们说，第一个用例**包含（includes）**第二个用例。如何用文本来展示被包含的用例没有标准的方式，但我发现用下画线表示成超链接的做法非常好，在许多工具里确实就是超链接。这样，在图9.1中，第一个步骤包含用例“browses catalog and select items to buy”（浏览目录并选择要购买的条目）。

对于会把主要场景弄乱的复杂步骤和在若干用例中重复的步骤来说，写成被包含用例会有用。然而，不要尝试使用功能分解方法来分解用例为子用例和子子用例。这样分解只会浪费许多时间。

和场景中的步骤一样，你可以给用例添加一些其他的共同信息。

- **前置条件（pre-condition）**描述在系统允许用例开始之前，系统应该确保为真的东西。前置条件的用处是，告诉程序员在他们的代码中不必检查这些条件。

- **保证（guarantee）**描述用例结束时系统会确保的东西。成功场景结束后得到的是成功保证；任何场景结束后都会得到最小保证。

- 触发器（**trigger**）规定了触发用例开始的事件。

当你考虑添加元素时要谨慎。少做一点比多做一点要好。另外，要努力保持用例简短和易读。我发现长的、详细的用例不容易阅读，这也与使用用例的目的相背。

你需要在用例中表现出细节数量依赖于用例中风险的数量。通常，在早期，你只需要少数关键用例的细节；其他用例可以直到要实现时再充实。你不必写下所有细节，口头沟通通常非常高效，特别是在一个需要快速见到运行代码的迭代周期内。

9.2 用例图

正如我之前所说，UML很少涉及用例的内容，但确实提供了图形格式来展示，如图9.2所示。虽然图有时是有用的，但它不是强制的。你在做用例时，不要花太多精力来画图。相反，应该将精力集中于用例的文本内容。

图9.2 用例图

思考用例图的最佳途径是，把它看做用例集内容的图形目录。它和结构化方法中使用的上下文图相似，因为它展示了系统边界和外部世界的交互。用例图展示执行者、用例和它们之间的关系：

- 哪个执行者执行哪个用例。
- 哪个用例包含其他用例。

除了简单的包含关系之外，UML还包含用例之间的其他关系，例如«extend»。我强烈地建议你忽略它们。我看见过太多团队纠结于何

时使用不同的用例关系，这会大大地浪费精力。相反，集中于用例的文本描述，这是这项技能的真正价值所在。

9.3 用例的级别

一个常见的问题是，用例聚焦于用户和系统之间的交互，你可能会忽略有时改变业务流程可能是处理问题的最佳途径。通常，你会听到人们谈论系统用例和业务用例。这个术语不精确，但一般来说，**系统用例（system use case）**是和软件的交互，而**业务用例（business use case）**讨论业务如何响应顾客或事件。

[Cockburn，use cases]建议了一种用例的级别结构。核心用例在“海平面级别”。**海平面级别（sea-level）**用例通常代表主执行者和系统之间独立的交互。这样的用例会交付某些对主执行者有价值的东西，通常需要主执行者花几分钟到半小时来完成。被海平面级别用例包含的用例是**鱼级别（fish level）用例**。比海平面级别更高级别的用例是**风筝级别（kite-level）**用例，展示海平面级别用例如何在更广的业务交互中发挥作用。风筝级别用例通常是业务用例，而海平面级别和鱼级别用例是系统用例。你应该将你的大多数用例放在海平面级别上。我更喜欢在用例的顶端指出级别，如图9.1所示。

9.4 用例和特性（或故事）

许多方法使用系统特性（极限编程把它们叫做用户故事）来帮助描述需求。一个常见的问题是特性如何与用例关联。

在计划一个迭代项目时，特性是堆砌系统的好办法，每一个迭代交付许多特性。用例提供执行者如何使用系统的叙述。因此，虽然两种技术都描述需求，但它们的目的是不同的。

虽然你可以直接描述特性，但许多人发现这样做很有帮助：先开发用例，然后生成特性列表。特性可能是整个用例、用例中的场景、用例中的步骤或一些行为变体，例如为你的资产评估添加另一个折旧方法，这在用例叙述中展示不出来。通常，特性比用例有更细的粒度。

9.5 何时使用用例

用例是帮助理解系统功能需求的有价值的工具。在项目早期，应该粗略地描述每个用例，然后在开发该用例之前再去做更详细的版本。

重要的是，要记住用例表达系统的外部视图。所以，不要期望用例和系统内部的类之间有任何关联。

可以看到的用例的内容越多，用例图的价值就越少。集中你的精力在用例的文本上而不是图上。尽管事实上UML没有提到用例文本，但正是文本包含了该技能的所有价值。

关于用例，一个很大的危险是人们把它们搞得太复杂而卡住了。通常，做太少比做太多受的伤害更少。在大多数情况下每个用例一两页纸正好。如果你写少了，至少你会有、可读的短文档作为问题的开始点。如果你写得太多，很少有人会去阅读和理解它。

9.6 更多资料

用例最开始因Ivar Jacobson的[Jacobson，OOSE]而流行。

虽然用例已经出现一段时间，但是它们的使用标准化程度很小。UML在用例的重要内容上涉及很少，只是把重要性小得多的图标准化了。因此，你可以发现各种关于用例的有分歧的观点。

然而，最近几年，[Cockburn，use cases]变成了这个主题的标准书籍。过去，当我和Alistair Cockburn观点不同时，通常最后都是我同意他的

观点。由于这个特别的原因，本章中我遵从了该书的术语和建议。他也维护了一个网站：**http://usecases.org**。[Constantine and Lockwood]提供了通过用例获得用户界面有说服力的过程，可以参照**http://foruse.com**。

第10章 状态机图

状态机图是描述系统行为的常用技能。从20世纪60年代开始，各种状态图形式相继出现，最早的面向对象技术采纳了状态图来展示行为。在面向对象方法中，为单个类画状态机图来展示单个对象的生命期行为。

无论何时，人们写到关于状态机的东西时，例子必然是巡航控制或售货机。我对它们有点厌烦了，所以我决定使用哥特城堡中一块秘密面板的控制器来做例子。我想把我的贵重物品放进这个城堡的一个保险箱中藏起来。因此，为了把锁露出来，我必须把蜡烛从烛台移开，但只有门关闭时才能开锁。一旦我可以看到锁，就可以插入我的钥匙打开保险箱。为了确保安全，我确保只有首先替换蜡烛才能打开保险箱。如果小偷忽略这个预警，我就放出一个丑恶的妖怪吞掉他。

图10.1展示了指挥我那独特的安全系统的控制器类的状态机图。状态图从控制器对象被创建时的状态开始，就是图10.1中的Wait状态。这张图用**初始伪状态（initial pseudostate）**指出这一点，初始伪状态不是状态，但它有一个箭头指向初始状态。

图10.1展示了控制器可以有3个状态：Wait（等待）、Lock（上锁）和Open（打开）。图上也给出了控制器从一个状态到另一个状态的改变所遵从的规则。这些规则以转换的形式存在：连接状态的线。

图10.1 一个简单的状态机图

转换（transition）表示从一个状态到另一个状态的转移。每个转换都有一个标签，标签有3个部分：trigger-signature [guard]/activity（触发器-签名[警戒条件]/活动）。所有部分都是可选的。trigger-signature通

常是触发状态潜在改变的单个事件。guard，如果有的话，是对要执行的转换来说必须为真的布尔条件。activity是一些在转换期间要执行的行为。它可以是任何行为表达式。trigger-signature的完整形式可能包括多个事件和参数。因此在图10.1中，你可以把从Wait状态向外的转换阅读为“在Wait状态，如果蜡烛被移走时门是关闭的，把锁露出来并转换到Lock状态”。

转换的所有3个部分都是可选的。如果没有活动部分，说明在转换期间你不做任何事情。如果没有警戒条件部分，说明事件发生时你总是执行该转换。没有触发器-签名部分的情况很少，但确实也有，这种情况说明你要立即执行该转换，这就是最常见到的活动状态，稍后我将讨论这一点。

当事件在某个状态发生时，你可以只执行一个向外的转换。因此如果你在同一事件上使用多个转换，就像图10.1中的Lock状态，警戒条件必须互斥。如果一个事件发生，没有生效的转换——例如，Wait状态上的safe-closed（保险箱关闭）事件或者带门打开条件的candle-removed（蜡烛移除）事件——该事件被忽略。

终止状态说明状态机是完整的，暗含着控制器对象的删除。这样，如果某人粗心大意，掉进了我的陷阱，控制器对象会终止，因此我需要“把兔子放进笼子，打扫战场”，并重启系统。

记住，状态机只能够展示对象直接观察或激活的东西。因此，虽然你可能期望我在打开保险箱时添加或拿走东西，但我没有把它放在状态图上，因为控制器不能辨别。

当开发人员谈论对象时，他们经常引用对象的状态，状态意味着对象的字段中所有数据的结合。然而，状态机图中的状态是更加抽象的状态概念；本质上，不同的状态意味着对事件的不同反应方式。

10.1 内部活动

状态可以在没有转换时响应事件，使用**内部活动（internal activity）**：把事件、警戒条件和活动放在状态框里面。

图10.2展示了一个带character（字符）和help（帮助）事件内部活动的状态，正如你可能在UI的文本域中看到的。内部活动类似于**自转换（self-transition）**：转回到同一状态的转换。内部活动的语法与事件、警戒条件和例程遵从同样的逻辑。

图10.2也展示了两个特别的活动：入口活动和出口活动。**入口活动（entry activity）**无论何时进入一个状态都要执行；**出口活动（exit activity）**无论何时离开一个状态都要执行。然而，内部活动不触发入口活动和出口活动，这就是内部活动和自转换之间的区别。

图10.2 一个文本域的typing（输入中）状态的内部事件展示

10.2 活动状态

到目前为止，我所描述的状态，对象是安静的，在它做某些事情之前会等待下一个事件。然而，你可以有这样的状态：对象正在做一些持续进行的工作。

图10.3的Searching（搜索中）状态是这样一个**活动状态（activity state）**：持续进行的活动标记为do/，因此我们采用术语**do-活动（do-activity）**。一旦搜索完成，就会执行任何没有事件的转换，例如显示新硬件。如果在活动期间，cancel（取消）事件发生，do-活动嘎然而止，我们会回到Update Hardware Window（更新硬件窗口）状态。

图10.3 带活动的状态

do-活动和常规活动都代表执行一些行为。两者之间的关键差异是常规活动“突如其来地”发生，不能被常规事件中断，而do-活动的时间有限制，并且可以中断，如图10.3所示。对于不同的系统，“突如其来”意味着不同的东西：对于硬实时系统，可能是执行一些机器指令的时间，但对于桌面软件，可能是若干秒。

UML 1为常规活动使用术语**动作（action）**，只为do-活动使用活动。

10.3 超状态

通常，你会发现若干状态共享共同的转换和内部活动。在这些情况下，你可以把它们当做子状态，把共享行为移进超状态，如图10.4所示。要是没有超状态，你将不得不为Enter Connection Details（输入连接细节）状态里面的所有3个状态都画一个cancel转换。

图10.4 带有嵌套子状态的超状态

10.4 并发状态

状态可以被分解成若干正在并发运行的正交状态图。图10.5展示了一个简单小巧的闹钟，它可以播放CD或收音机，以及展示当前时间或闹钟时间。

选择CD/收音机和当前/闹钟时间是正交的选择。如果你想要用非正交的状态图表达，这张图可能会非常乱，而且当需要更多状态时，就会失控。把两个区域的行为分离到各自的状态图中，就清晰得多了。

图10.5也包含了一个**历史伪状态（history pseudostate）**。这个状态表明当闹钟开机时，收音机/CD选择回到关机时的状态。历史伪状态箭

头所指向的状态代表了在没有历史时第一次进入该区域，应该进入哪个状态。

图10.5 并发正交状态

10.5 实现状态图

状态图的实现主要有3种方式：嵌套switch、状态模式和状态表。处理状态图最直接的方法是嵌套switch语句，如图10.6所示。虽然直接，但即使在简单的情况下，也会得到长长的烦琐的代码。这个方法也非常容易失去控制，因此即使是简单的情况我也不喜欢使用它。

状态模式（state pattern）[Gang of Four]创建状态类层次来处理状态的行为。图中的每个状态都有一个状态子类。控制器有针对每一个事件的方法，它简单地转发给状态类。实现状态图10.1的类如图10.7所示。

图10.6 处理图10.1状态转换的C＃嵌套switch

图10.7 图10.1的状态模式实现

层次的顶层是一个抽象类，它将所有事件处理方法实现为什么都不做。对每一个具体状态，简单地覆盖状态有转换的特定事件方法。

状态表（state table）方法捕获状态图信息为数据。因此图10.1可以表达成像表10.1那样的一个表。然后，我们建造运行时使用该状态表的一个解释器或者一个基于该状态表生成类的代码生成器。

表10.1 图10.1的状态表

显然，状态表一次要做比较多的工作，但是之后每一次有状态问题时都可以使用。运行时状态表也可以在无须重新编译的情况下修改，这在某些情况下是很方便的。当你需要时，状态模式更容易组织，虽然对于每一个状态它需要一个新的类，但每一种情况只要写少量代码即可。

上面这些实现相当小型化，但它们应该为你如何实现状态图提供了一些思路。在每一种情况下实现状态模型都会得到非常公式化的代码，因此，通常最好使用某些形式的代码来完成。

10.6 何时使用状态图

状态图擅长于描述跨越若干用例的对象的行为。状态图不是很擅长于描述涉及许多对象协作的行为。如果是这种情况，把状态图和其他技能结合很有用。例如，交互图（参见第4章）擅长于描述单个用例中的若干对象的行为，活动图（参见第11章）擅长于展示若干对象和用例的通用活动序列。

不是每个人都能接受状态图的思考方式。留心人们如何使用状态图。可能你的团队会发现状态图这种形式没有用。这不是一个大问题；就像之前所说的，应该混合使用对你来说有用的技术。

如果你使用了状态图，不要尝试为系统中的每一个类画状态图。虽然盛典完美主义者经常这样做，但几乎是浪费精力。只为那些呈现有趣行为的类使用状态图，这时状态图可以帮助你理解发生了什么。许多人发现UI和控制对象用状态图描绘比较有用。

10.7 更多资料

《UML用户指南》[Booch，UML user]和《UML参考手册》[Rumbaugh，UML Reference]中都有更多关于状态图的信息。实时设计人员使用状态模型的机会更多，因此不必惊奇[Douglass]写了许多关于状态图的东西，包括如何实现状态图的东西。[Martin]里包含非常好的实现状态图的各种方式的章节。

第11章 活动图

活动图是描述过程逻辑、业务流程和工作流的技术。在许多地方，它们扮演的角色类似于流程图，但和流程图表示法之间的首要区别是它们支持并行行为。

随着UML版本的变化，活动图的变化是最大的改变之一，因此不必惊奇，它们在UML 2中被再次大大扩展和改变了。在UML 1中，活动图被看做状态图的特例。这导致人们在建模活动图擅长的工作流时出现了许多问题。在UML 2中，状态图和活动图之间的这种纽带被移除了。

图11.1展示了一个简单的活动图实例。我们在**初始节点（initial node）**开始动作，然后做动作Receive Order（接收订单）。一旦完成，我们面临一个分叉。**分叉（fork）**有一个进入流和若干离开的并发流。

图11.1说Fill Order（填写订单）、Send Invoice（发送发票）和后续的动作并行发生。本质上，这意味着它们之间的序列是不相关的。我可以填写订单，发送发票，交付，然后接收付款；或者，我可以发送发票，接收付款，填写订单，然后交付：你懂的。

图11.1 一个简单的活动图

我也可以交错做这些动作。我抓起第一个订单行条目，敲出发票，抓起第二个订单行条目，把发票放进信封，等等。或者，我可以同时做一些事情：一只手敲出发票，同时抓起另一个订单行条目。根据上图，任何这些序列都是正确的。

活动图允许做该流程的任何人选择做事情的次序。换句话说，活动图只是说明本质的不得不遵从的先后顺序规则。这对于业务建模来说是重要的，因为流程经常并行发生。对于并发算法，活动图也很有用，在并发算法中，独立的线程可以并行做事情。

当你碰到并行的情况时，你需要同步。直到它被交付和付款之前，不能关闭订单。我们在Close Order（关闭订单）动作前加上结合（**join**）展示这一点。使用结合时，只有当所有进入流到达结合时，离开流才会执行。因此你只可以在接收付款和交付后关闭订单。

因为在UML 1中活动图是状态图的特例，所以UML 1有平衡分叉和结合的特定规则。在UML 2中，不再需要这样的平衡。

你会注意到活动图上的节点称为动作，而不是活动。严格地说，活动指一个动作序列，因此活动图展示由动作组成的活动。

条件行为由判断和合并来勾画。判断（**decision**），在UML 1中称为分支，有单个进入流和若干带警戒的外向流。每个外向流有一个警戒条件：一个放在方括号里面的布尔条件。每一次到达一个判断，你只可以执行其中一个外向流，因此警戒条件应该互斥。如果使用[else]作为警戒条件，意思是如果判断上的所有其他警戒条件均为假，应该使用[else]流。

在图11.1中，填写订单之后，就有一个判断。如果你有一张紧急订单，就做Overnight Delivery（隔夜交付）；否则，做Regular Delivery（常规交付）。

合并（**merge**）有多个输入流和单个输出流。合并意味着由判断开始的条件行为结束了。

在我的图中，每一个动作都有单个进入流和单个输出流。在UML 1中，多个进入流有一个隐式的合并。即任何流触发，你的动作都会执

行。在UML 2中有了变化，用一个隐式的结合代替；这样，只有在所有流触发时动作才执行。由于这个变化，我推荐你只对动作使用单个进入和输出流，并显式地展示所有结合和合并，这样可以避免混淆。

11.1 分解一个动作

动作可以分解为子活动。可以把图11.1的交付逻辑拿出来，定义它自己的活动（图11.2）。然后可以把它称为一个动作（147页的图11.3）。

图11.2 一张次级的活动图

动作可以实现为子活动或者类的方法。可以用耙子符号展示子活动，用语法class-name：：method-name展示方法上的调用。如果被调用的行为不是单个方法调用，也可以在动作符号里写一个代码片段。

图11.3 修改图11.1的活动为调用图11.2的活动

11.2 分区

活动图告诉你发生了什么，但它们没有告诉你谁做什么。在编程中，这意味着活动图没有传达每一个动作由哪个类负责。在业务流程建模中，就是没有传达组织的哪一部分执行哪个动作。这不一定是问题，集中于做了什么而不是谁做了哪部分行为通常更有意义。

如果你要展示谁做什么，可以把活动图划分为分区，分区展示哪一个动作由哪一个类或组织单元执行。图11.4（149页）展示了一个简单的例子，它表明订单处理涉及的动作如何分离到各个部门。

图11.4 活动图上的分区

图11.4的分区是简单的一维分区。这种风格经常被称为**泳道**，原因显而易见。在UML 1.x中，这是唯一的使用形式。在UML 2中，你可以使用二维网格，因此游泳隐喻不给力了。你也可以针对每一个维度层级划分行或列。

11.3 信号

在如图11.1所示的简单例子中，活动图有一个清晰定义的开始点，对应于调用程序或子程序。动作也可以响应信号。

时间信号（time signal）因为时间的流逝而发生。这样的信号可以指示财务周期中的月末，或者实时控制器的每一微秒。

图11.5展示了一个监听两个信号的活动。一个**信号（signal）**指示该活动从一个外部进程接收一个事件。这说明活动不断监听那些信号，该图定义了活动如何反应。

图11.5 活动图上的信号

在如图11.5所示的情况下，在航班离开前2个小时，我需要开始打包。即使我收拾得足够快，我还是不能离开，直到出租车到达。如果在我打好包之前出租车到达了，它不得不等待我完成打包才能出发。

除了接收信号，我们还可以发送信号。如果我们不得不发送一个消息，然后等待回复才能继续，这时信号就很有用。图11.6展示了一个超时的好例子。注意：两个流在竞赛，第一个到达终止状态的将获胜

并终止其他流。在这里我用了单个活动结束，意思等同于分开画两个图标（即对于活动结束没有隐式的结合）。

图11.6 发送和接收信号

虽然接受通常只是等待一个外部事件，我们也可以展示一个进入它们的流。这说明，直到流触发了接受，我们才开始监听。

11.4 令牌

如果你足够勇敢去冒险探寻UML规则极深处的秘密，你会发现规则的活动部分大量谈及令牌及它们的生产和消费。初始节点创建一个令牌，然后传给下一个动作，该动作执行，再把令牌传给下一个。在分叉处，一个令牌进入，分叉在每一个向外的流上产生一个令牌。反过来，在结合处，每一个向里的令牌到达，没有任何事情发生，直到所有令牌出现在结合处；然后一个令牌在外向流上产生。

你可以通过使用硬币或筹码在图上移动的方式来使令牌可视化。一旦你要处理更复杂的活动图，令牌经常会使得可视化更容易。

11.5 流和边

UML 2使用术语流（**flow**）和边（**edge**）来描述两个动作之间的连接。最简单的一种边就是两个动作之间的简单箭头。如果你喜欢，可以给边命名，但大多数时候，一个简单的箭头足够了。

如果你在连线上有困难，可以使用连接器，连接器让你不必画一条穿过整张图的线。当你使用连接器时，必须成对使用：进入流一个，离开流一个，两者标签相同。我倾向于尽可能避免使用连接器，因为它们破坏了控制流的可视化。

最简单的边传送一个令牌，这个令牌除了控制流之外没有别的含义。然而，你也可以沿着边传送对象；这时对象扮演令牌的角色，同时携带数据。如果你正在沿着边传送一个对象，可以通过在边上放一个类框来展示，或者你可以在动作上使用针脚，不过，针脚暗示着一些更细微的东西，接下来我将描述它。

图11.7展示的所有风格是等价的；哪一种最能传达你尝试沟通的东西，就应该用哪一种。大多数时候，简单的箭头已经足够。

图11.7 展示边的4种方式

11.6 针脚和变换

动作可以有参数，就像方法一样。在活动图上你不需要展示参数信息，但如果你愿意，可以用**针脚（pin）**展示。如果你正在分解一个动作，针脚对应于已分解的图上的参数框。

当严格地画活动图时，你不得不确保向外动作的输出参数匹配另一个动作的输入参数。如果它们不匹配，你可以用一个**变换（transformation）**（图11.8）来从一个参数得到另一个参数。变换必须是一个无副作用的表达式：本质上，输出针脚参数上的查询为输入针脚提供一个正确类型的对象。

你不必在活动图上展示针脚。当你要查看各种动作所需要和生产的数据时，针脚是最佳选择。在业务流程建模中，你可以使用针脚来展示由动作生产和消费的资源。

图11.8 流上的变换

针脚可以让你安全地展示进入同一动作的多个流。针脚表示法加强了隐式的结合，UML 1中没有针脚，因此不会和早期版本混淆。

11.7 扩展区域

在活动图中，你经常碰到这样的情形：一个动作的输出触发另一个动作的多个调用。有若干种方式来展示这一点，但最佳途径是使用扩展区域。**扩展区域（expansion region）**标记活动图的一个区域，在该区域里动作为一个集合中的每个条目执行一次。

在图11.9中，Choose Topics（选择主题）动作生成一个主题列表作为它的输出。然后，这个列表中的每一个元素变成令牌，输入到Write Article（写文章）动作。类似地，每一个Review Article（评审文章）动作生成单篇文章，文章被添加到扩展区域的输出列表。当扩展区域中的所有令牌汇集到输出集合后，该区域为列表生成单个令牌，传送给Publish Newsletter（出版简报）。

图11.9 扩展区域

在这个例子中，输出集合的条目数和输入集合的条目数是相同的。但是，也可能条目会变少，在这种情况下，扩展区域扮演过滤器的角色。

在图11.9中，所有文章并行写作或评审，用 «concurrent» 关键词标记。你也可以有一个迭代扩展区域。迭代区域必须一次性地完全处理每一个输入元素。

如果你只有单个需要多次调用的动作，可以使用如图11.10所示的速记方式。速记方式假设并发扩展，因为这是最常见的。这个表示法对应于UML 1中的动态并发的概念。

图11.10 扩展区域中的单个动作的速记

11.8 流结束

一旦像在扩展区域中一样得到多个令牌，即使活动作为一个整体还没有结束，流通常也结束了。**流结束（flow final）**指一个特定流的结束，不终止整个活动。

通过修改如图11.9所示的例子，允许文章被拒绝，图11.11展示了流结束。如果一篇文章被拒绝，令牌被流结束销毁。和活动结束不同，流结束后活动的剩下部分可以继续。这个方法允许扩展区域扮演过滤器的角色，使得输出集合比输入集合小。

图11.11 活动中的流结束

11.9 结合规格

默认地，当所有输入流到达一个结合时，结合让执行通过它的向外流（或者用更正式的说法，当每个输入流上的令牌都到达时，它释放出一个它的输出流上的令牌）。在一些情况下，特别是当你有带多个令牌的流时，更加复杂一些的规则是有用的。

结合规格（join specification）是一个附加到结合上的布尔表达式。每一次一个令牌到达该结合时，结合规格被计算，如果为真，就释放出一个输出令牌。因此在图11.12中，无论我何时选择饮料或者投入硬币，机器都会计算结合规格。仅当我放进足够的钱时，机器才会解我的渴。正如这个例子，如果你要指明已经接收了每一个输入流上的令牌，就把该流加上标签，并将它们包含进结合规格。

图11.12 结合规格

11.10 其他更多内容

我应该强调，本章只谈到了活动图的一些皮毛。UML里还有更多内容，你可以为这项技术单独写一整本书。事实上，我认为活动图非常适合作为一本书的主题，深入研究这个表示法及如何使用它。

重要的问题是：它们用得有多广泛？当前，活动图不是被最广泛使用的UML技术，它们的流建模前辈们也不是非常流行。以这种方式描述行为的图形技术还没有流行起来。另一方面，这也反映了许多社区确实有这样的潜在需求，亟待一种标准技术来满足。

11.11 何时使用活动图

活动图的长处在于这样一个事实：支持和鼓励并行行为。这使得活动图成为工作流和流程建模的一个很棒的工具，确实，UML 2中的许多改进都来自人们对工作流的关注。

你也可以使用活动图作为符合UML规则的流程图。虽然这允许你以UML的方式来做流程图，但并不令人兴奋。原则上，你可以利用分叉和结合的优势来为并发程序描述并行算法。虽然我不常在并发圈子里混，但我没有看到有很多人这样使用活动图。我想其原因是并发编程的大多数复杂性是避免数据争用，活动图在这一点上帮助不大。

可能在人们使用UML作为编程语言时，用活动图来做是一个主要优势。在这种情况下，活动图是一种表达行为逻辑的重要技术。

我经常看到活动图被用来描述用例。这个方法的危险是，领域专家经常不容易看懂。如果是这样，你最好使用常用的文本形式。

11.12 更多资料

虽然活动图总是相当复杂，在UML 2中尤其如此，但并没有一本好书深入地描述它。我希望这个缺憾某一天会被填补。

各种面向流的技术的风格都和活动图相似。其中一个有较多人知道的是Petri网——也算不上广为人知。在这方面，**http://www.daimi.au.dk/PetriNets/**是一个好网站。

第12章 通信图

通信图（communication diagrams），交互图的一种，强调交互的各种参与者之间的数据链接。通信图不像序列图那样，把每一个参与者画成生命线并按垂直方向展示消息序列，通信图允许自由放置参与者，通过画链接来展示参与者如何连接，并使用编号来展示消息序列。

在UML 1.x中，这种图被称为协作图（**collaboration diagrams**）。这个名字生命力很强，我怀疑人们在习惯新名字之前这个名字还会被使用一段时间。（和协作[171页]有所不同；所以名称变了。）

图12.1展示了和图4.1中相同的一个中央控制交互的通信图。使用通信图，我们能够展示参与者如何链接在一起。

除了展示关联实例的链接，我们也可以展示瞬时链接，瞬时链接只在交互上下文中出现。在这个例子中，从Order到Product的 «local» 链接就是一个局部变量；其他瞬时链接还有 «parameter» 和 «global»。这些关键词在UML 1中存在，但从UML 2中消失了。不过，因为它们有用，我期望它们能在习惯用法中继续保留下来。

图12.1 中央控制的通信图

图12.1的编号风格简明而常用，但实际上在UML中是不合法的。为了符合UML的规定，你不得不使用嵌套小数编号体系，如图12.2所示。

图12.2 嵌套小数编号的通信图

采用嵌套小数编号是为了解决自调用的模棱两可问题。在图4.1中，你可以清晰地看到，getDiscountInfo在calculateDiscount方法内部被调用。如果用图12.1中的扁平编号，就不能辨别getDiscountInfo是在calculateDiscount内调用，还是在整体的calculatePrice方法内调用。嵌套编号体系解决了这个问题。

尽管它是非法的，许多人更喜欢扁平的编号体系。嵌套编号可能会变得混乱，特别是当调用被嵌套多次时，会导致这样的序列编号：1.1.1.2.1.1。在这种情况下，模棱两可问题没有得到改进，反而更糟糕了。

除了编号，你也可以在消息上放置字母，这些字母代表不同的控制线程。因此消息A5和B2在不同的线程中，消息1a1和1b1是并发嵌套在消息1内的不同线程。也可以在序列图上看到线程字母，虽然这样做没有可视化地传达并发。

通信图对于控制逻辑没有精确的表示法。它们确实允许你使用迭代标记和警戒条件（74页），但它们不允许你完全详述控制逻辑。创建或删除对象没有特别的表示法，但使用 «create» 和 «delete» 关键词是通常的惯例。

12.1 何时使用通信图

关于通信图的主要问题是，何时使用它们而不是使用更常见的序列图。这个问题的答案很多时候依赖于个人偏好：一些人喜欢通信图胜过序列图，或者喜欢序列图胜过通信图。通常，偏好比任何其他事情更能驱动选择。总体上，大多数人看起来更喜欢序列图，这一次，我随大流。

更理性的方法是：当你要强调调用序列时，序列图更好；当你要强调链接时，通信图更好。许多人发现通信图在白板上更容易修改，因此

通信图是探索备选方案的好方法，虽然在需要通信图的情况下，我通常更喜欢CRC卡。

第13章 组合结构

UML 2的一个最重要的新特性是深入内部结构层级分解一个类的能力，这允许你把一个复杂的对象分解为部件（part）。

图13.1展示了一个TV Viewer（电视机）类，以及它的供给和需求接口（86页）。我以两种方式展示：使用小球-球窝表示法和在内部列出。

图13.1 展示TV Viewer及其接口的两种方式

图13.2展示了这个类如何在内部分解为两个部件，以及部件支持和需要的接口。每一个部件以name：class的形式命名，两个元素可以只保留一个。部件不是实例规格，因此它们用粗体表示，而不是用带下画线表示。

图13.2 组件的内部视图（来自Jim Rumbaugh的例子）

你可以展示一个部件存在多少个实例。图13.2说明，每一个TV Viewer包含一个发生器（generator）部件和一个控制（control）部件。

为了展示实现接口的部件，可以从接口画一个委托连接器。类似地，为了展示部件需要接口，可以增加一个到该接口的委托连接器。你也可以用一根简单的线展示部件之间的连接器，正如我在这里所做的，或者用小球-球窝表示法（89页）。

你可以向外部结构添加端口（图13.3）。端口允许你将需求和供给接口分组为组件与外部世界的逻辑交互。

图13.3 带多个端口的组件

13.1 何时使用组合结构

虽然一些更老的方法有相似的思路，但是组合结构是UML 2新增的。考虑包和组合结构之间的区别的一个好方法是：包是编译时分组，而组合结构是展示运行时分组。组合结构非常自然地适合展示组件及组件如何分解成部件，因此这个表示法大都在组件图中使用。

因为组合结构在UML中比较新，要辨别它有多高效言之过早，需要经过实践才能显露出来。UML委员会的许多成员认为添加这些图非常有价值。

第14章 组件图

OO社区中总会有大范围的争论，争论组件和常规类之间的差异是什么。这个争论不是我在这里要解决的，但我能够向你展示UML使用的表示法，以辨别它们之间的差异。

UML 1为组件提供了特有的符号（图14.1）。UML 2移除了这个表示法，但允许你用一个形状相似的图标来标注类框。或者，你可以使用«component»关键词。

图14.1 组件的表示法

除了这个图标之外，组件不引进我们没见过的任何表示法。组件通过实现接口和需求接口来连接，经常使用第13章介绍的组合结构的表示法。

图14.2展示了一个组件图的例子。在这个例子中，till（收银台）使用sales message（销售消息）接口，通过消息队列连接到Sales Server（销售服务器）组件。该消息队列既供应sales message接口来和收银台交流，还需要该接口和服务器交流。服务器分解成两个主要组件，transaction processor（事务处理器）组件实现sales message接口，accounting driver组件和accounting system交流。

图14.2 组件图例子

整体而言，这些组件展示为更大的零售系统的部件，因此有了组件名字前面的冒号。多重性标记意味着有许多收银台和服务器，但只有一

个队列和一个会计系统。如果不显示多重性，就默认是1，就像我在Sale Server的内部表示的。一般来说，我更喜欢在重要的地方显式地展示多重性。你可以使用小球-球窝表示法来表示连接器，也可以简单地使用一根线。小球和球窝对展示接口有用，而简单的线更容易画。

图14.2是用组合结构图的风格绘制的。事实上，我认为就是和组合结构图一模一样，只是部件组件的装饰少一些。你也可以用类图的方式绘制组件图，通常聚焦于组件之间的依赖。一种好的想法是，这种组件类图展示了组件之间可能的连接，而组件的组合结构展示了特定上下文中组件之间的实际连接。

正如我已经说过的，什么是组件的问题是永恒争论的主题。我发现以下陈述很有帮助：

组件不是一种技术，技术人员可能认为这难以理解。组件关注顾客要如何与软件发生关系。顾客要能够每次购买一小片软件，能够像升级音响一样升级软件。他们需要新的小片能无缝地和旧的小片一起工作，能够按照他们自己的进度升级，而不是按照制造商的进度升级。顾客要能够从各种制造商那里混合和匹配小片。这是非常容易理解的需求，只不过难以满足。

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

重要的一点是，组件代表可独立购买和升级的小片。因此，把系统分解成组件，既是市场决定的，也是技术决定的，在这方面，[Hohmann]是一本优秀的指南。同时也提醒你小心过细粒度的组件，因为太多组件会难以管理，特别是版本控制会昂起它那丑陋的头，导致“DLL地狱”。

在早期版本的UML中，组件用于表达物理结构，例如DLL。现在不是这样了，现在使用工件（119页）表达物理结构。

14.1 何时使用组件图

当你把系统分解成组件并要展示它们通过接口的相互关系时，或者把组件分解为更低级别的结构时，使用组件图。

第15章 协作

不像本书的其他章那样，本章不对应于UML 2中官方的图。UML标准把协作当做组合结构的部件来讨论，但在UML 1中它和组合结构没有任何关系。因此我感觉最好单列一章来讨论协作。

让我们考虑一下拍卖的概念。在任何拍卖中，可以有一个卖家（seller）、一些买家（buyer）、许多要拍卖（lot）的货物和一些出价（offer）。我们可以用类图的术语描述这些元素（图15.1），也可以加上一些交互图（图15.2）。

图15.1不是十分常见的类图。首先，它被一个虚线的椭圆包围，椭圆代表拍卖协作。其次，协作中的所谓的类不是类，而是应用协作时要实现的**角色（role）**——所以它们的名称不是以大写字母开头的。看到对应于协作角色的真实接口或类也不稀奇，但不一定要有。

在交互图中，参与者的标签和一般情况稍微不同。在协作中，命名体系是participant-name/role-name：class-name。和一般情况一样，所有这些元素都是可选的。

图15.1 带角色类图的协作

图15.2 拍卖协作的序列图

当你使用协作时，你能够通过在类图上放置一个**协作发生（collaboration occurrence）**来展示你正在使用一个协作，图15.3展示

了拍卖应用中一些类的类图。从协作到那些类的链接表明类如何扮演协作定义的各种角色。

图15.3 协作发生

UML建议使用协作发生表示法来展示模式的使用，但几乎没有模式作者这样做过。Erich Gamma开发了一种很好的备选表示法（图15.4）。图中的元素加上了模式名称或者pattern：role结合的标签。

15.1 何时使用协作

虽然协作在UML 1中已经存在，但我承认我几乎没有用过，即使在我的模式作品中。当角色由不同的类扮演时，协作确实提供了一种方式来组织交互行为。

然而，在实践中，我没有发现协作是一种有竞争力的图形类型。

图15.4 展示JUnit（junit.org）中模式使用的非标准方式

第16章 交互概述图

交互概述图是活动图和序列图嫁接起来得到的。你可以认为交互概述图就是活动图，不过其中的活动替换为了小的序列图，或者认为交互概述图就是序列图被活动图表示法打碎了，用于展示控制流。不管哪一种方式，都做了一些奇怪的混合。

图16.1展示了一个简单的例子；交互概述图的表示法和你已经见过的活动图和序列图很类似。在这张图中，我们要产生和格式化一个订单汇总报表。如果顾客是外部的，我们从XML得到信息；如果顾客是内部的，我们从数据库得到信息。小的序列图展示了两个备选方案。一旦我们得到数据，就格式化报表；在这个例子中，我们不展示序列图，而是通过引用交互框简单地引用序列图。

16.1 何时使用交互概述图

交互概述图是UML 2新增的，同样，到底它在实践中有多少用处，现在还言之过早。我对交互概述图不是很感兴趣，因为我认为它混合了两种风格，而这两种风格不可能真正混合得很好。而是使用活动图还是使用序列图，取决于哪一种图能更好地为你的目的服务。

图16.1 交互概述图

第17章 时间图

离开中学以后，我开始学习电子工程，后来才转到学计算机。因此当我看到UML定义时间图为它的标准图之一时，感到了某种似曾相识的郁闷。长久以来，时间图一直在电子工程中使用，看起来绝不会需要UML的帮助来定义它们的含义。但既然它们在UML中，还是值得简要提及一下。

时间图是另一种形式的交互图，它的焦点是时间约束：针对单个对象，或者针对一束对象更有用。让我们看一个简单的场景，这个场景基于咖啡壶的泵和电炉。让我们想象一个规则：泵开始工作和电炉开始工作之间的间隔至少要10秒。当盛水容器变成空的时，泵关闭，电炉不能继续打开超过15分钟。

图17.1和图17.2是展示这些时间约束的可选方式。两张图展示同一基本信息。主要差异是图17.1展示从一根水平线移到另一根水平线导致的状态改变，而图17.2维持同一水平位置，但用一个交叉展示状态改变。当只是一些状态时，图17.1的风格更好，正如在这个例子中；当有许多状态要处理时，图17.2更好。

{ > 10s}约束上的虚线是可选的。如果你认为它们有助于确切澄清时间约束是哪一个事件的，就使用它。

图17.1 展示状态为线的时间图

图17.2 展示状态为区域的时间图

17.1 何时使用时间图

时间图有助于展示不同对象上的状态改变之间的时间约束。这种图对硬件工程师来说特别亲切。

附录A UML版本之间的变化

当本书第一版上架时，UML的版本是1.0。其中大多数内容看起来已经稳定，并进入OMG的认可流程。从那时起有了数次修订。在这个附录中，我描述1.0版本以后该书发生的重要变化以及那些影响本书中材料的变化。

这个附录概括了这些变化，因此如果你有本书的早期印本，你可以保持更新。我已经修改了本书，让它跟上UML的脚步，因此如果你有后续的印本，它描述的就是该印本印刷时的情形。

A.1 对UML的修订

最早公开的、最后成为了UML的发布版本是统一方法（Unified Method）版本0.8，于1995年10月在OOPSLA上发布。统一方法0.8的工作由Booch和Rumbaugh完成，这时，Jacobson还没有加入Rational。1996年，Rational发布版本0.9和0.91，包含了Jacobson的工作。后面的版本，名称改成了UML。

1997年1月，Rational和一群伙伴提交了UML版本1.0给OMG的分析和设计任务组。随后，Rational及其伙伴，还有其他提交者的工作被结合起来，于1997年9月提交了一份OMG标准建议书，即UML版本1.1。该建议书在1997年年底被OMG采纳。然而让人困惑的是，OMG把这个标准版本称为1.0版。因此，现在UML既有OMG版本1.0，也有Rational版本1.1，所以要小心和Rational 1.0混淆。在实践中，每个人都把标准版本叫做1.1。

此后，UML有许多进一步的开发。1998年UML 1.2出现，1999年UML 1.3出现，2001年UML 1.4出现，2003年UML1.5出现。1.x版本之间的

大多数变化发生在UML的内部，仅UML 1.3导致了一些看得见的变化，特别是用例和活动图。

在UML 1系列出现以后，UML的开发人员把他们的目光放到UML 2的大幅度修订上。第一个RFP（征求建议书）于2000年发行，但UML 2直到2003年才开始适当地稳定下来。

UML的进一步开发几乎肯定是要发生的。UML Forum（**http://uml-forum.com**）通常是一个寻找更多信息的好地方。我也把一些UML信息放在了我的网站上（**http://martinfowler.com**）。

A.2 《UML精粹》中的变化

在这些UML修订进行时，我也尽可能保持修订《UML精粹》，推出新的印本，同时也就这些机会修正错误和做一些澄清。

保持更新最频繁的时期是《UML精粹》第1版期间，那时我们经常不得不在每次印刷时做更新，以跟上突然添加的UML标准。第一次到第五次印刷都基于UML 1.0。这些印本之间UML变化很小。第六次印刷使用了UML 1.1。

第七次到第十次印刷基于UML 1.2；第十一个印本第一次使用了UML 1.3。自UML 1.0后，印本所基于的UML版本号会出现在封面上。

第2版的第一个到第六个印本基于UML 1.3。第七个印本开始考虑UML 1.4的小变化。

本书第3版的更新根据是UML 2（参见表A.1）。在这个附录的剩下部分，我概括了UML从1.0到1.1，从1.2到1.3，以及从1.x到2.0的主要变化。我不讨论发生的所有变化，只讨论那些改变我在《UML精粹》里讲的某些东西的变化，或者阐述我要在《UML精粹》中讨论的重要特性的变化。

我延续《UML精粹》的精神：讨论影响UML在现实世界项目中应用的UML关键元素。和之前一样，这些选择和建议是我自己的东西。如果我所说的和官方UML文档之间有冲突，应该遵循UML文档。（但一定要让我知道，这样我可以做更正。）

表A.1 《UML精粹》和相应的UML版本

我也趁此机会指出之前印本的重要错误和遗漏。感谢向我指出这些错误和遗漏的读者们。

A.3 从UML1.0到1.1的变化

A.3.1 类型和实现类

在第1版的《UML精粹》中，我谈到了视角及它们如何改变人们画模型和解释模型的方式——特别是类图。现在，UML考虑了这一点，认为类图上所有的类都可以特化为类型或实现类。

实现类（**implementation class**）对应于你开发所用的软件环境中的类。类型（**type**）的概念要更加模糊一些，它代表更少绑定实现的抽象。可以是一个CORBA类型，一个规格视角或概念视角下的类。如果需要，你可以添加构造型来进一步区分。

对特定的图，你可以声明所有类遵循一个特定的构造型。当从特定视角画图时，这是你要做的。实现视角会使用实现类，而规格和概念视角会使用类型。

你使用实现关系来说明实现类实现一个或多个类型。

类型和接口有差别。接口直接对应于Java或COM风格的接口。接口只有操作，没有属性。

对于实现类，你可以只使用单一、静态分类，也可以使用带类型的多重和动态分类。（我假设这是因为主要OO语言遵从单一、静态分类。如果以后你使用有幸支持多重或动态分类的语言，那么这个限制确实应该不适用。）

A.3.2 完整和不完整的鉴别器约束

在之前的《UML精粹》印本中，我说到泛化上的{complete}约束指超类型的所有实例也必须是该分区的子类型的实例。对此，UML 1.1定义{complete}表示分区内的所有子类型都被指定，意思有所差别。我发现关于这个约束有一些不一致的解释，因此你应该小心。如果你确实要指出超类型的所有实例应该是其中一个子类型的实例，我建议你使用另一个约束来避免混淆。目前，我使用{mandatory}。

A.3.3 组合

在UML 1.0中，使用组合暗示着链接是不可变的或者冻结的，至少对于单值组件是这样的。这个约束不再是定义的一部分。

A.3.4 不可变和冻结

UML定义约束{frozen}来在关联角色上定义不可变。这个定义当前看起来对属性或类不适用。我现在的实践是使用术语**冻结（frozen）**来代替不可变，我乐意把这个约束应用到关联角色、类和属性上。

A.3.5 序列图上的返回

在UML 1.0中，序列图上的返回通过使用一个条形箭头而不是实心箭头来区别（参见之前的印本）。这让人有些不舒服，因为其中的差别太细微，容易忽略。UML 1.1使用虚线箭头来表示返回，这让我很高兴，因为这使得返回看起来明显多了。（因为我在*Analysis Patterns*

[Fowler，AP]中使用了虚线返回，这也让我看起来像是有影响力的人物。）你可以使用enoughStock：=check()来命名返回了什么。

A.3.6 术语“角色”的使用

在UML 1.0中，术语**角色（role）**主要是指明关联的方向（参见之前的印本）。UML 1.1把这个用法称为**关联角色（association role）**。还有一个**协作角色（collaboration role）**，指类实例在协作中扮演的角色。UML 1.1更多强调协作，看起来好像“角色”的这种用法变成了首要用法。

A.4 从UML 1.2（和1.1）到1.3（和1.5）的变化

A.4.1 用例

用例的变化包括新的用例之间的关系。UML 1.1有两种用例关系：«uses»和 «extends»，两者都是泛化的构造型。UML 1.3提供了3种用例关系。

- «include» 构造是依赖（dependency）的一个构造型。这说明一个用例的路径被包含进另一个用例中。通常，这发生在一些用例共享共同步骤的时候，可以将共同行为分解到被包含用例中。以ATM为例，Withdraw Money（取钱）和Make Transfer（转账）可能都使用Validate Customer（验证顾客），这替换了平常使用的 «uses»。

- 用例**泛化（generalization）**说明一个用例是另一个用例的变体。这样，我们可以有一个用例Withdraw Money——基用例——还有单独一个用例来处理由于资金不足取款被拒绝的情况。拒绝可以处理为特化取款用例的用例。（你也可以简单地处理为Withdraw Money用例内的另一个场景。）像这样的特化用例可能改变基用例的任何方面。

- «extend» 构造是依赖的一个构造型。比起泛化关系，扩展提供了更可控制的形式。在这里，基用例声明许多扩展点。扩展用例只可以改变扩展点处的行为。因此，如果你正在在线购买产品，你可能有一个购买产品用例，带有捕获配送信息和捕获付款信息的扩展点。然后，这个用例可以因常规顾客扩展，因为信息可以以不同方式获取。

旧关系和新关系之间存在一些混淆。大多数人使用«uses»的方式和使用ULM 1.3的«includes»是一样的，因此对于大多数人来说，我们可以说«includes» 替换了«uses»。大多数人使用UML 1.1的«extends» 时，既用做UML 1.3的«extends»的控制风格，也用做UML 1.3泛化的通用覆盖风格。因此，你可以认为UML 1.1的«extends» 已经被分裂为UML 1.3的«extend»和泛化。

虽然这个解释覆盖了我见过的大多数用法，但也不是使用旧关系严格正确的方式。然而，大多数人不遵从严格的用法，在这里我不真正深入探讨所有问题。

A.4.2 活动图

当UML到达版本1.2时，关于活动图的语义，有很多地方非常松散。因此，1.3版本的工作包括使这些语义变得更加严密。

对于条件行为，现在你可以使用菱形判断活动来合并或分支行为。虽然分支和合并都不是描述条件行为所必需的，但这种展示风格越来越常见，这样你可以把条件行为加上括号。

同步条现在称为分叉（**fork**）——把控制分裂开——或结合（**join**）——把控制同步在一起。然而，你可以不给结合再添加条件。还有，你必须遵从匹配规则，以确保分叉和结合匹配。本质上，这意味着每一个分叉必须有一个相应的结合，将从分叉开始的线程结合。你可以

嵌套分叉和结合，不过，当线程直接从一个分叉到另一个分叉时，或从一个结合到另一个结合时，你可以在图上消除分叉和结合。

仅当所有进入的线程到齐时，结合才会被引发。然而，你可以在从分叉出来的线程上加一个条件。如果此条件为假，线程被认为需要结合才能完整。

多触发器特性不再存在。取而代之，活动中可以存在动态并发，用活动框里面的*展示。这样一个活动可以被并行调用若干次；在任何离开的转换执行之前，它的所有调用必须完成。这大致等同于多个触发器和匹配同步条件，虽然弹性差一些。

这些规则减少了活动图的一些灵活性，但确实确保了活动图真的是状态机的特殊情况。活动图和状态机之间的关系在RTF上有一些争论。

A.5 从UML 1.3到1.4的变化

UML 1.4中最看得见的变化是添加了**扩展机制（profile）**，扩展机制允许将一组扩展收集在一起，放进一个一致的集合。UML文档包含一些扩展机制的例子。不过，定义一个构造型更多是一种形式主义。模型元素现在可以有多个构造型，而在UML 1.3中，它们被限制为一个构造型。

工件（artifact）被添加到UML中。工件是组件的物理显现，例如，Xerces是一个组件，我硬盘上的所有Xerces jar副本是实现Xerces组件的工件。

在UML 1.3之前，UML元模型中没有东西来处理Java的**包可见性（package visibility）**。现在有了，符号是“~”。

在UML 1.4中也用条形箭头在交互图中标记异步，这是一个相当尴尬的向后不兼容的变化。这难住了一些人，包括我。

A.6 从UML 1.4到1.5的变化

首要的变化是给UML添加了动作语义，这是使UML变成编程语言的必需步骤。这个版本让大家可以先用着，不用等待完整的UML 2。

A.7 从UML 1.x到UML 2.0

UML 2代表了到目前为止UML发生的最大变化。这一次修订，各种东西都变了，许多变化影响了《UML精粹》。

在UML中，UML元模型有了深层的变化。虽然这些变化不影响《UML精粹》中的讨论，但对一些人来说，它们非常重要。

最明显的变化之一是引进了新的图形类型。对象图和包图之前已经被广泛绘制，但一直不是官方的图形类型，现在它们是了。UML 2把协作图的名称变为了通信图。UML 2也引进了新的图形类型：交互概述图、时间图和组合结构图。

许多变化不影响《UML精粹》。我忽略了状态机扩展、交互图中的门和类图中的强类型等构造。

因此对于这一部分，我只讨论对《UML精粹》有影响的变化，包括之前的版本中我讨论过的变化，以及我在这个版本中才开始讨论的新变化。因为变化如此之广，所以我将它们按照本书的章节组织如下。

A.7.1 类图：基础（第3章）

属性和单向关联现在简单地被看做性质概念的不同表示法。不连续的多重性，例如[2，4]已经废弃。冻结性质也被废弃。我添加了一张列表，上面有常见的依赖关键词，它们中的一些是UML 2新增的。«parameter» 和 «local» 关键词已经废弃。

A.7.2 序列图（第4章）

大的变化是用序列图的交互框表示法来处理迭代、条件和各种其他控制行为。这允许你在序列图中相当完全地表达算法，不过我不太相信序列图会比代码更清晰。消息上旧的迭代标记和警戒条件已经从序列图中废弃。生命线的头不再是实例；我使用术语参与者（**participant**）来指代它们。UML 1的协作图在UML 2中更名为通信图。

A.7.3 类图：概念（第5章）

构造型现在定义更加严密。因此，我现在用双尖括号中的词语来表示关键词，其中只有一些是构造型。对象图上的实例现在是实例规格。类现在可以需要接口，也可以提供接口。多重分类使用泛化集来将泛化分组。组件不再用特别的符号绘制。主动对象使用双竖线取代了粗线。

A.7.4 状态机图（第10章）

UML 1分离了短生命期的动作和长生命期的活动。UML 2把它们都叫做活动，并使用术语do-活动来表示长生命期的活动。

A.7.5 活动图（第11章）

UML 1将活动图处理为状态图的特例。UML 2打碎了这个链接，废弃了UML 1中活动图不得不保持的匹配分叉和结合的规则。因此，它们最好理解成令牌流而不是状态转换。这样，就出现了一整套新的表示法，包括时间和接受信号、参数、结合规格、针脚、流变换、子图耙、扩展区域和流结束。

一个简单但尴尬的变化是UML 1将进入一个活动的多个流处理为隐式的合并，而UML 2则处理为隐式的结合。因此，在做活动图时，我建

议使用显式的合并或结合。

泳道现在可以是多维度的，一般称为分区。

参考文献

索引