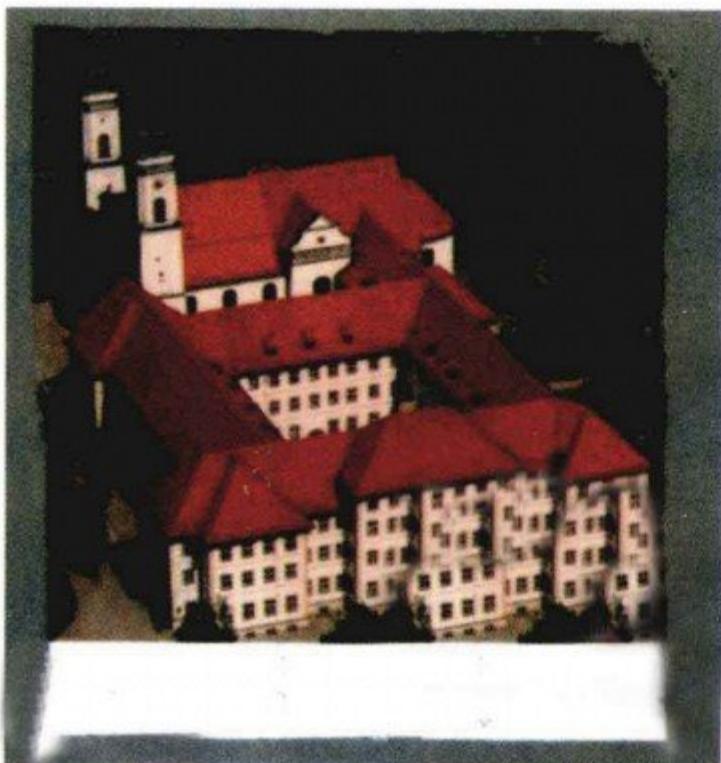


Pattern-Oriented Software Architecture Volume 5
On Patterns and Pattern Languages

面向模式的软件架构 模式与模式语言



卷5

[德] Frank Buschmann

[英] Kevlin Henney

著

[美] Douglas C. Schmidt

肖鹏 等译



人民邮电出版社
POSTS & TELECOM PRESS

Pattern-Oriented Software Architecture **Volume 5**
On Patterns and Pattern Languages

面向模式的软件架构

模式与模式语言 卷5

“本书很可能成为经典……对于任何想理解软件架构的人，这都是一本好书，强烈推荐！”

——StickyMinds.com

软件模式已大大改变了我们设计、实现和思考计算系统的方式。模式给我们提供了用于描述架构愿景的词汇，以及清晰明了、切中要点的典型设计和详细实现示例。然而，在模式概念方面很久都没有紧跟技术发展的著作出现了。对于模式概念的基础知识的新认识也仅停留在模式社区少数专家和思想先驱们的头脑中。本书的目的正是要总结模式与模式语言的知识，满足软件开发社区的需求。

本书是“面向模式的软件架构”系列的收官之作，将呈现、讨论、比较和关联众多模式概念（如单个模式、模式互补、模式复合、模式故事、模式序列和模式语言）中已知的特性和应用。本书由三部分组成：第一部分介绍了过去十多年来单个模式的用法，第二部分介绍了各个模式之间的关系，第三部分以前两部分的内容为基础介绍模式语言。

- 经典POSA系列收官之作
- 深入剖析模式和模式语言
- 名著佳译，相得益彰



图灵社区：www.ituring.com.cn
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/软件工程

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-26173-1

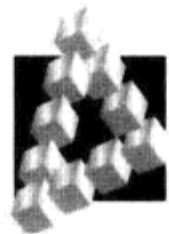


ISBN 978-7-115-26173-1

定价：59.00元

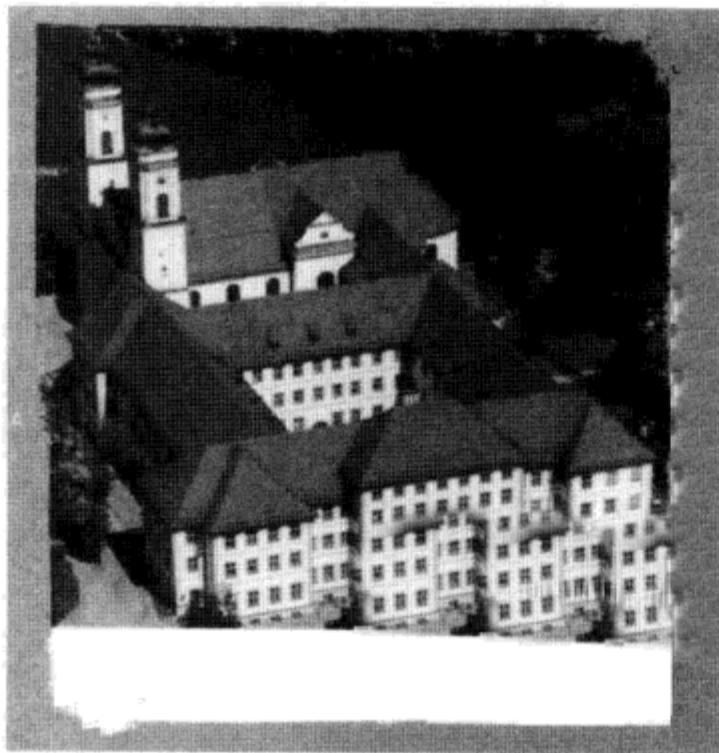
TURING

图灵程序设计丛书



Pattern-Oriented Software Architecture Volume 5
On Patterns and Pattern Languages

面向模式的软件架构 模式与模式语言



卷5

[德] Frank Buschmann

[英] Kevlin Henney

著

[美] Douglas C. Schmidt

肖鹏 等译

人民邮电出版社

北京

图书在版编目 (C I P) 数据

面向模式的软件架构. 第5卷, 模式与模式语言 /
(德) 布施曼 (Buschmann, F.) , (英) 亨尼 (Henney, K.),
(美) 施密特 (Schmidt, D. C.) 著 ; 肖鹏等译. -- 北
京 : 人民邮电出版社, 2011.9
(图灵程序设计丛书)
书名原文: Pattern-Oriented Software
Architecture Volume 5: On Patterns and Pattern
Languages
ISBN 978-7-115-26173-1

I. ①面… II. ①布… ②亨… ③施… ④肖… III.
①软件设计②面向对象语言—程序设计 IV. ①
TP311. 5②TP312

中国版本图书馆CIP数据核字(2011)第161944号

内 容 提 要

本书共分 3 部分, 首先介绍了单个模式, 详细阐述了过去累积的关于如何描述和应用模式的诸多见解, 接着探究了模式之间的关系, 从组织的角度说明了各个模式的领域, 最后介绍了如何将模式和模式语言相结合。

本书适合软件架构师和开发人员阅读。

图灵程序设计丛书 面向模式的软件架构 卷5: 模式与模式语言

◆ 著 [德] Frank Buschmann [英] Kevlin Henney
[美] Douglas C. Schmidt
译 肖 鹏 等
责任编辑 王军花
◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京艺辉印刷有限公司印刷
◆ 开本: 800×1000 1/16
印张: 17.75
字数: 430千字 2011年9月第1版
印数: 1-3 500册 2011年9月北京第1次印刷
著作权合同登记号 图字: 01-2008-0487号
ISBN 978-7-115-26173-1

定价: 59.00元

读者服务热线: (010)51095186转604 印装质量热线: (010)67129223

反盗版热线: (010)67171154

版 权 声 明

Original edition, entitled *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*, by Frank Buschmann, Kevlin Henney, Douglas C. Schmidt, ISBN 978-0-471-48648-0, published by John Wiley & Sons, Inc.

Copyright © 2007 by John Wiley & Sons, Inc., All rights reserved. This translation published under License.

Simplified Chinese translation edition published by POSTS & TELECOM PRESS Copyright © 2011.

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

本书简体中文版由John Wiley & Sons, Inc.授权人民邮电出版社独家出版。

本书封底贴有John Wiley & Sons, Inc.激光防伪标签，无标签者不得销售。

版权所有，侵权必究。



谨以此书献给Anna、Bebé和Martina。

——Frank Buschmann

谨以此书献给Carolyn、Stefan和Yannick。

——Kevlin Henney

谨以此书献给Lori和Bronson，也献给我的爸爸和妈妈。

——Douglas C. Schmidt

谨以此书献给John。

——Frank Buschmann、Kevlin Henney和Douglas C. Schmidt

译 者 序

本书是《面向模式的软件架构》系列 (*Pattern-Oriented Software Architecture*) 的收官之作，其重点不在于具体的模式或者模式语言，而在于模式之间的关系和表现。作者深厚的功力和丰富的经验确保本书值得每个关注架构和设计的人收藏和阅读。

对于我而言，除了对模式和模式语言有了更加深刻的理解之外，我从这部书中学到的更有价值的是作者分析问题的思路。作者反复在模式、模式组合、模式序列、模式语言等各个不同层次上采用类似的分析方式展现问题的本质；他抽丝剥茧，以过程的方式来分析问题；他还在不同的地方使用了框架的方法来辅助分析问题。凡此种种，对于我日常的开发和咨询工作都带来了莫大的裨益。

没有做过大部头翻译的同学大概很难体会翻译的痛苦。这本中文版能够呈现在你的面前，其背后无数个周末和假期，我的大部分业余时间或者贡献给了这本书，或者是在本书尚未译完的愧疚阴影中度过的。感谢我的家人在本书（包括卷4和卷5）翻译过程中给我的支持和理解。当你看到本书的时候，我的孩子已经出生了，在他（她）出生前陪他（她）的时间太少了，希望以后可以补回来。

我非常荣幸能够得到这本书的翻译机会。这样一部经典著作对我来说挑战大于机会，我的整个过程都是小心翼翼、如履薄冰。特别感谢本书的审校，限于译者的水平，审校想必更加痛苦。

从卷4翻译开始到现在已经将近4年了，正是在这期间我加入了ThoughtWorks。这是一个神奇的公司，这里有很多人让我受益匪浅。特别感谢我的同事荣浩、李彦辉、乔梁、张凯峰、韩楷，他们或者帮我翻译了其中的部分章节，或者在审校中帮我指正错误，或者跟我讨论其中的技术细节。

本书第2、3、9、10章由李锐翻译，第4章由乔梁翻译，第5章由李彦辉翻译，第14章由郭晓刚翻译，其余部分由肖鹏翻译，全书由肖鹏通稿。由于译者水平有限，翻译错误或者风格不合口味在所难免，在此我谨代表本书的译者对给你造成的阅读上的不便表示歉意。

你的任何意见既可以发表到我的博客 (<http://xiaopeng.me>)，也可以通过邮件直接发给我，我的邮件地址是——eagle.xiao@gmail.com。

感谢并期待你的批评和指正！

Richard P. Gabriel序

“软件模式已大大改变了我们设计的方式……” POSA5在其开头“关于本书”中如此写道——这是它的自白抑或可以看做它的判词。然而，在我们设计的过程中到底发生了什么？设计是关乎问题还是关乎美？对驱动力的化解是否在解决问题的同时也向设计中注入了美的元素？抑或原始材料只有经过（模式）语言加工之后，美才会随着理想解决方案一起浮现？有人曾经告诉我，如果一个建筑的入口显而易见，那么它就不值得造访。

在《牛津英语词典（第2版）》中，对“设计”（design）是这样解释的：

1. 一项智力计划。
- 1.a. 为了后续行动在头脑中形成的计划或者方案；有关某个即将实施的想法的基本概念；方案。

Guy Steele（盖·斯蒂尔）1998年在OOPSLA上的讲话“Growing a Language”中指出：

设计是关于如何构建某个事物的规划。设计是尚未在真实世界中构建之前在人的头脑中构建一个事物——或者，进一步说，就是计划如何构建真实的事物。

我曾经写道：

设计是构建之前所做的思考。

Carliss Baldwin和Kim B. Clark是这样定义设计的：

设计是一系列指示，它所基于的是将资源转变为有用、有价值的事物的知识。

—*Between “Knowledge” and “the Economy”: Notes on the Scientific Study of Designs*

以上这些定义和描述无不是围绕着先于构建时刻的想法、计划、知识、问题、价值和意图、线索而展开的。听上去合情合理，这也是我们希望设计所具备的特点——可靠。正因为它是值得信赖的、接受过良好教育的人想出来的，因而能够给人安全感，不用担心它会左右摇晃甚至坍塌。但是，让我们回过头来考虑一下“我们”是谁。

几千年来，人们在“设计和建造”城市的过程中一直会使用各种工具和概念，比如Christopher Alexander的模式语言。这些作品宏大而复杂。对于像雅典、罗马、柏林、伦敦、伊斯坦布尔，甚至纽约、旧金山和波士顿这样的城市，我们似乎很难说清楚计划所在，因为它实际上发生在数年、数十年、数百年，乃至上千年的时间里。因此，本序言第一句话里所说的“我们”必然是指更接近现代的我们——软件设计者，这些人倾向于上述定义中那些更为保守或者不太现实的想法。即

使在我自己给出的那句定义里面——我尽量给设计、构建和反思留下了融合的空间，这样，即使对构建城市也同样适用——我们很难对构建之前的思考视而不见。

设计的广阔内涵以及含义上的细微差别让人不禁想起Sussex大学Adrian Thompson所定义的一系列设计的特征。我把这些特征称为设计元启发式理论，即有关设计的种种思维方式，比如如何使用模式和模式语言。设计元启发式理论有3种形式。先看第一种。

逆模型易处理：如果对于系统而言，存在易处理的“逆模型”，那么总有办法提前制定一系列变量以获得期望的结果。

这里，设计就是一个计划，或者说蓝图，逆模型是这样的：一旦待设计的对象确定了，总可以找到一个方法来确定如何构建出它来。这是预先设计的基本条件，适用于之前至少构建过一次的情况，通常都构建过很多次了。Alexander了解这种风格的工作方式：模式和模式语言的思想来源于工作在行业一线的人们（比如瑞士建谷仓的人自然知道瑞士谷仓相关的模式和模式语言），这是他们从小就耳濡目染、潜移默化的结果。从这个意义上讲，逆模型很容易确定：我们头脑中对未来谷仓的形象跟隔壁家的谷仓没有什么两样，构建步骤也大同小异。模式语言对于设计师和建造者具有（暗含的或者显式的）指导作用，而那些必不可少的适当调整正是常见的构建过程。这就引出了Thompson的第二个设计元启发式理论。

逆模型不易处理，但是正向模型易处理：这意味着我们可以根据目标值预测各种变化的影响，但是系统本身并不可逆，所以我们也就不可能提前确定一系列变量以获得期望的结果。这暗示我们采用迭代的方式工作，根据正向模型精心挑选的变量按顺序使用。这种迭代式的“设计—测试”的方式是最常见的、传统的工作方式。

设计一点、构建一点、反馈一点——这就是启发式的含义。我认为，这也是敏捷方法论、演进式设计/编程以及Alexander的“基本过程”（Fundamental Process）背后的本质。下面的内容摘自敏捷宣言。

欢迎需求的变化，即使在开发末期。敏捷流程通过驾驭变化保持客户的竞争优势。

频繁地交付可运行的软件，时间间隔从一两周到一两个月不等，越短越好。

.....

最好的架构、需求和设计都源自自组织的团队。

——<http://agilemanifesto.org/principles.html>

Alexander在《秩序的本性》一书中用“基本过程”来描述设计和构建。在这部4卷的鸿篇巨著中，他讨论了如何将关注点组织成整体。下面是第(1)步、第(2)步、第(4)步、第(6)步和第(8)步。

- (1) 在整个流程中——不论是构思、设计、制作、维护还是修复阶段——我们必须始终对于要做什么在头脑中有一个整体的认识。我们关注整体，理解它，努力探寻其深层结构。
- (2) 我们不断地问自己接下来应该做什么对于整体最有好处。
- (4) 在我们对新的居住中心做强化的时候，我们做事的方式跟创建或者强化那些更大的居住中心时的做事方式是一样的。

(6) 我们不断地检查所做的是否真正提高了整体的生活和感觉。如果整体的感觉未得到任何深化，我们刚做的事情就没有意义，应该退回去。否则，我们继续。

(8) 如果没有进一步可做的事情来深化整体的感觉，我们就停下来。

不论是Alexander还是软件开发人员，对于模式语言都是这样用的。这种用法只是说明第一种设计元启发式理论只适用于设计（大部分）是重复的情况。这时候，我们总是能够发现一些模型用来模拟“基本过程”中所描述的“设计一点/构建一点”的模式。或者真有一个模型——物理、数学、科学——可以主宰整个设计，并假装我们真能把设计给“试出来”，或者我们自欺欺人地认为“基本过程”中的大步骤（版本1、2、3）每个都可以看做是独立的设计。之所以说自欺欺人，是因为我们不愿意将视角拉远一点。Alexander的“基本过程”对于各种敏捷方法也是适用的，虽然有人可能认为我更应该提一下重构。

你手中拿的这本书（不知道你是否已经开始阅读）总体上是讲模式语言的。构建之前的思考可以用第(1)步、第(2)步和第(4)步来描述。在这种情况下，设计其实是为了解决问题。正如Jim Coplien所说：

设计是一系列有意识的活动，从发现问题到解决问题。所谓问题，是指目标状态和当前状态的差别。

这种说法似乎遗漏了（对某些人来说）很重要的两个方面：美和新。一个非常好的葡萄酒杯既实用又美观，它是由精致的、具有异国情调的材料制成的，它的形状能够恰到好处地衬托起杯中的美酒，其优雅的线条也令人赏心悦目——每一种美酒都有一种与之相配的酒杯，凝聚其芳香，提高其口感，握起来手感舒适。说到“新”(newness)，我恨不得用“新颖”(novelty)这样的词语，但是“新颖”往往意味着知识或者学术上的创新，似乎不能切中要害。设计并不是发明一种全新的东西，而是当你看到一种以前没有见过的东西的时候，它带给你的是快乐和惊奇。我想到了1936年推出的Cord 810汽车。

Cord 810的车身是由设计师Gordon M. Buehrig领导的包括年轻的Vince Gardner在内的团队完成的。新车在1936年11月的纽约车展上引起了轰动。人们挤在Cord 810的周围，甚至有人站到别的车的保险杠上，只为一睹Cord 810的芳容。订单纷至沓来……

——维基百科

人们以前从没见过这样的汽车——前面棺形的车鼻，隐藏的头灯，（每个汽缸一个）铬合金的排气筒弯曲着伸出了两边，前面流畅的曲线——这简直不是设计师的作品，而是时尚的造型师的杰作。在外观、工程设计和用户体验方面最值得注意的有两点，一个是内陷的头灯，另一个是置于发动机前方的变速器，它使得乘客厢相对于当时的其他汽车来说更加平坦而且也更低。有人说，Cord 810的设计算得上是20世纪最独特的设计了。1996年，*American Heritage*杂志宣布Cord 810轿车是“最漂亮的美国车”。

举个例子罢了。

我觉得不能说Cord是为了解决某个问题而得到的产物：肯定有别的什么在里面。Rebecca

Rikner具体指明了这个“别的什么”。

设计不是关于如何解决问题的。设计关乎如何发现美。

大家可能认为有关Thompson的启发式理论，我已经说完了，其实不然，还有第三种情况。而第三种情况才是他研究的主题。

正向模型和逆向模型均不易处理：既不能找到一个规律说明哪些变化有利于改善目标值，也无法预期变化会对目标值产生怎样的影响。离开了进化，一切都将消失。

——*Notes on Design Through Artificial Evolution: Opportunities and Algorithms*

进化！

Thompson曾经设计过一个模拟电路，使用遗传算法通过现场可编程门阵列（FPGA）来区分1kHz和10kHz的方波^①。人们将其结果表述为“可能是迄今报道过的最离奇、最神秘、最突破常规的无约束演进电路”，这主要是因为最终研究人员也没搞清楚这个电路是怎么工作的。下面的评价，大家不必关心其细节，只需看看人们表达怀疑的方式就行了。

然而不知何故，在脉冲结束之后200ns之内，该电路就能“知道”它花了多长时间，尽管这期间它是完全没有响应的。

这确实难以置信，所以我们不得不通过各种独立的观察来佐证这一发现，所有的观察都表明在脉冲期间电路确实是没有响应的。

对于遗传算法（genetic algorithm）、遗传编程（genetic programming）、神经演化（neuro-evolution）、神经网络（neural net）以及统计/随机方法（statistical/aleatoric method）的研究均向人们展示了令人震惊的结果——有些结果甚至难以理解，比如由美国航天局（NASA）用人工进化方法设计并测试的天线，其大小如婴儿手掌，各个部件弯曲成一种奇怪的隐对称（cryptosymmetry）结构，就像黑色的灌木从刚刚经历了一场冻霜。它看起来不像是人设计出来的东西，但是它在某些方面确实表现出更优秀的特性，所以从解决问题的角度来说，它算是一个好的设计。

但是我们有没有感觉到那是一个好的设计呢？我们应该怎样利用这类演化呢？人工演化是怎么工作的？简而言之，人工复制就是将来自于双亲的特征结合在一起。由此产生的特征还会发生少量的随机突变，所以包含了原始双亲大部分特征的结果，还要经过适应性测试。假设这些特征跟设计的某些方面相关，我们可以简单地说，它采用了某个看起来不错的解决方案中的一些设计元素，将其与另一个也还不错的解决方案中的设计元素结合在一起，然后再加入一些变化，看它是否可行。除了有一些随机的成分外，整体上听起来并不随意。

从我来说，这样“设计”出来的东西都有有意思的特征，问题是这种方式算不算是设计的一种形式。有人会说，不是由人类设计者作出的设计就不算设计，Coplien描述中的“有意识”和牛

^① 见Adrian Thompson等人著的*Explorations in Design Space: Unconventional Electronics Design Through Artificial Evolution*。

津词典定义中的“头脑”是设计的基本特征。那么，“基于知识的指令”体现在哪里呢？如果把前述的FPGA鉴频器和诡异天线交给Pierre Menard^①照搬一遍，质疑的人说不定就承认它们是设计了。或者可以进一步说，只有给出了人类可理解的解释，我们才承认它是设计。

这让我想到了定向发育（canalization^②）的概念，它的描述是这样的。

如果绑定发育过程以产生一个特定的终止状态，而不论初始状态和发育过程中环境发生什么变化，那么其结果就是定向的。

——（改写自）Andre Ariew, *Innateness is Canalization: In Defense of a Developmental Account of Innateness*

某物一旦落入某条河流的支流，遵循一定的过程，它终将进入那条河流，我们将这样的产出物称为定向的产出。“定向”（canalized）来自于“运河”（canal）这个词——一旦进入了运河，便不可逃脱。复杂科学中的奇异吸引子（strange attractor）即属于定向的。人工演化并不关心设计或者至少是它的产出物是美丽、优雅、容易理解、成本低廉、可维护、可扩展、有序、光亮、简单，还是多余、丑陋等，它所关注的只有一点——是否符合功能要求。其他有关人的因素不会对定向设计产生任何影响，尽管人类的设计中总会多多少少考虑这些东西。

从另一个方向说，这个天线的设计有些诡异，因为它不像是人设计出来的东西。我们还没有能力穷尽整个设计空间，所以只能关注对我们来说看起来更靠谱的那部分。

本书从模式和模式语言的角度来谈设计。在阅读的过程中，先要把思绪集中在设计理念上，或许之后会发散到更远的地方。放任自己去感受设计的华美和真实。这3种元启发式设计理论或许构成了一个连续的统一体，如果确实如此，也将存在某种设计使人变成非人类（non-human），甚至后人类（post-human）。

模式语言包含了大量的设计空间。它们构成了软件设计和构建的工具箱的一部分。它们帮助人们作出更好的设计。对模式语言应用得如何反映了设计做得怎么样。设计，设计，设计，设计。

Richard P. Gabriel

① 出自博尔赫斯的短篇小说Pierre Menard, Author of *The Quixote*。在故事中，一位20世纪的作家 [Pierre Menard (皮埃尔·梅纳尔)] 立意重写塞万提斯的《堂吉诃德》中的部分篇章，而且要用跟塞万提斯的原文一模一样的文字，但这绝不是照抄，而是不谋而合。请看摘录：

将塞万提斯和梅纳尔对比一下就明白了。塞万提斯这样写道（第一部分第9章）：

……真理，是历史所孕育，是时间的敌人，是事迹的堆积，是往昔的见证，是当今的圭臬，是未来的辅佐。

“外行天才”塞万提斯17世纪写下的这一番排比，无非是对历史的虚辞赞美。

而梅纳尔却是这样写道：

……真理，是历史所孕育，是时间的敌人，是事迹的堆积，是往昔的见证，是当今的圭臬，是未来的辅佐。

历史孕育了真理：这个想法实在是令人惊奇。

作为与威廉·詹姆斯同时代的作家，梅纳尔不把历史看做事实的反映，而是将其视作事实的来源。历史的真相对他来说，不是发生过什么，而是我们认为发生过什么。最后一句话——当今的圭臬，未来的辅佐——更是具有赤裸裸的实用主义色彩。

其风格上的对比也非常鲜明。梅纳尔对于古体风格毕竟有点陌生，他这样写总有做作之嫌。塞万提斯就没有这种毛病，他对当时通行的西班牙语驾轻就熟。

② 生物/心理学名词，也译为限向发育、限向发展等。——译者注

Wayne Cool序

遥想当年，我遇到一群人，他们肩扛来自一个建筑狂人的异端思想的沉重大旗，奔波在编程革命的路上。那个狂人自己一心要通过重现过去的辉煌——不过这是指设计感（sense^①）的辉煌，而非设计本身——来振兴未来。这伙牛人离群索居，与那些正统的大师不同，他们从不现身于主流的聚会，他们有自己的研讨会。他们退隐于世外桃源，那些地方虽然鲜有人知，但却值得一访^②。他们并不像那个疯子一样整天把美挂在嘴边，他们撸起胳膊、挽起袖子，拿起螺钉、螺母，将实践与程序、实用性与严谨性紧紧拧在一起。他们的工作面向一线工作者，又未弃理论于不顾。

我随着PLoP会议转战南北，流连于各地的咖啡厅、酒吧和星巴克，见证了Beck的宣讲，见证了研讨会的结束，见证了阿勒顿的暴风雨，见证了伊尔塞旭日初升，见证了维肯贝里骑马之旅的尘埃落定，同时也见证了相关书的编写、审校、付印和赞美。在这中间，我看到了人们对于设计和人造物结构的兴趣慢慢增长，字里行间，人们似乎希望创建一个由抽象和想法构成的世界，而又希望这个世界像由木头、油漆、石头和金属构成的世界那样触手可及。

这就是我要谈的：软件模式将表象的世界（其内部机制隐藏在纷繁与复杂之中）转换成一个简单而透明的世界，隐藏的机制成为了精工细作的目标，那其实也是一种期望，期望把事情做得更好，但是只是为了做好而做好，不图回报，图的只是运用技能时的自豪感，而靠的是敏锐的洞察力和深入的思考。对于这些人来说，手艺隐藏于某种用户界面之下算不得什么——因为总有一天会有人拨开乌云，让其重见天日。这些表象尽管有意义，但是他们并不会把它隐藏起来。

艺术。手艺。工程。科学。这正是设计模式的灵感源泉。艺术和科学是故事，手艺和工程是行动。

手艺是介于艺术和科学中间的形式，艺术和手艺同工程和科学之间又是对立的。艺术是独特的、首要的，是对天赋与欲望浓缩的产物。手艺则是可靠的质量的产物。工匠^③可能会失意，但是很少失败。手工艺品是由人采用某些材料制成的。工程是指人通过某些设施的帮助可靠而高效地生产产品。科学的产物通常在工程中采用。如果一本书叫做“什么什么的艺术”，它通常记录了这种东西的制作人的秘密。“什么什么的手艺”讲述的则是一个一个作出这些东西来的艰辛和汗水。“什么什么的工程”讨论的则是规则和长远计划，其产品通常是面向企业客户的。“什么什么的科学”是指像数学这样的书。

然而，科学和手艺的角色被人们误读了。很多人坚定地认为科学必然走在手艺和工程的前面，任何构建过程首先需要抽象的认知。比如，要设计一个蒸汽机，你必须懂得（正确的）热力学原

① 或者用敏捷阵营常用的那个词——“味道”（scent）。

② 向Bill Knott致歉。

③ 这回我用了个漂亮的词语craftman。

理。然而，在发明蒸汽机的时代，科学家们坚持“热质论”，即热的本质是一种称为“热质”（卡路里，caloric）的“微妙的流体”。整个宇宙中这种物质的数量是恒定的，它从热的物体传递给冷的物体。讲得不错。热质论能够解释很多现象，但是建造第一个蒸汽发动机的人根本就不知道这个理论，甚至都不关心什么理论。在使用锅炉的过程中，他们注意到了体积、压强和温度之间的关系，于是他们建造了蒸汽发动机。说不定科学家们正是通过对蒸汽发动机的观察才发展出了“现代”热力学理论。

工匠了解自己用的东西。他们会观察。领悟力和系统的思考使他们能够将自己的理解表达出来。成熟的手艺催使技术的发展。当然也包括科学。在科学和工程之间存在着反馈环，但是很难讲谁占有主导地位。

这都是有关构建的事情，这就要求要有计划和执行。计划是先于构建的思考。执行则是理解计划并生产出所要的东西。从20世纪早期开始，管理科学就强调要将计划和执行分开。Frederick Winslow Taylor是这样说的：

尽可能将脑力劳动从工作现场剔除，集中到计划和规划部门。

——科学管理原则（*Principles of Scientific Management*）

真贴心！

计划和执行在艺术领域几乎融为一体，而在工程领域则分得很远，在手艺方面则介于二者之间。科学则属于思想和思路领域。

将计划与执行相分离，并不是为了效率。也不是为了得到最大的价值。也许能，也许不能。计划需要思考，思考也不是免费的。如果你能做很少的思考，而构建出大量的产品，你多半能挣到钱。流程代替了执行过程中的思考，培训代替了教育——Dancing Links和X^①算法就是这样解决数独问题的。有了这两个算法，解决数独问题只需要一台计算机，不需要人。将计划与执行相分离，考察的是成本。

这也不是为了质量。质量最好的汽车绝对不会是机器人生产线造出来的。成本。

如果成本是采用工程的动力，那么模式则是将我们拉回到手艺和艺术的力量。

模式。匠艺：我们认为它来自人们建成这个世界的纹理之中。在这个世界里，质量和对细节的注意体现在事物的表面之上，体现在恰当的缝隙之间，体现在巧妙而自然的连接之中。在这个世界里，天赋总是伴随着知识和智力。我们来看一下George Sturt在*The Wheelwright's Shop*^②中对如何制造轮辋（木质车轮边缘部分）是怎样描述的。

① Dancing Links是由Donald Knuth提出的高效实现X算法的技术。X算法是一个递归的、非确定的、深度优先的暴力算法，用来找到完全覆盖问题的所有解决方案。

② 在1923年首次出版。

然而，此时谈论过多的细节是徒劳的。通过一个看似简单的流程就把这些简单的装置组装到了一起，但材料带来的变化却是无穷无尽的。当两个轮辋完成的时候，它们有多大的相似性呢？车匠必须让它们相似。他把两块完全不同的轮辋墩子做成了相似的轮辋，因为从来不存在两个一样的墩子。这儿有几个结，那儿有几道裂纹^①、几块伤疤^②，高低不平^③的边缘，过厚或者过薄，种种情况总是带来新的可能或否定原有的解决方案，频频挑战工人的创造力。他没有带锯可用（目前，1923年），只能用最原始的方式来处理眼前的一切。木材不是机器面前无辜的牺牲品，而是让懂得迁就它的人体会到它特有的品性。

——*The Wheelwright's Shop*

你可以像木工那样感受木头。

模式是那些对代码有系统认识的人总结出来的，他们深入理解代码，并长期以作出正确的设计和编码为己任，他们不相信计划可以与执行分开。正是由于这类人的存在，科学才得以发展。这不仅仅是贩卖抽象的理念，这是思考。艺术和工艺的进步并非来自于模式，而是来自于以模式为工具的人。需要抵制的是模式的自动化运行，这会将计划和执行隔裂开来。

计算机科学家站在黑板前写写擦擦、左顾右盼、高谈阔论。模式爱好者关注于手法，躬身于代码之中，埋头于鼠标键盘之间。他们工作的原材料也是：这儿有几个结，那儿有几道裂纹、几块伤疤，高低不平的边缘，过厚或者过薄，种种情况总是带来新的可能或者否定原有的解决方案。他们几乎将编程看做一项体力劳动，袖子卷得高高的，露出结实的臂膀，炯炯有神的眼睛背后是他们闪光的思想指导着双手。



模式社区发起伊始，我便在其中了。我站在山头；我躲在林间；我奔跑于海滩；我漫步于胡德山；我驾着小艇，穿过绳网，倾听着日落的歌声；我蒸着桑拿，大汗淋漓；我骑着大马，像野人一样飞驰；我和牛仔们一起分享牛排和玉米。但是，我却未曾贡献只言片语，人们的想法和著作中连个标点符号都没有我的份。然而，我相信你们知道我是怎么想的，对不对？

Wayne Cool于威尼斯海滩

① 由风霜造成的木材中的裂纹。

② 树木早年间遭受的损伤，导致树干中生出新的树皮。

③ 从不太方正的原木上切下来的板子边缘过于锐利或者不均匀，或者在试图切成方形的过程中也可能出现这种情况。

关于本书

软件模式已大大改变了我们设计、实现和思考计算系统的方式。模式给我们提供了用于描述架构愿景的词汇，以及清晰明了、切中要点的典型设计和详细实现示例。使用组成模式描述软件，可以使我们使用更少的词汇进行更为有效和清晰的沟通。

自20世纪90年代中期以来，很多软件系统（包括Java及C#编程语言和库的主要部分）都是在使用模式进行开发。以前模式有时会被选择性地用于解决特定的挑战和问题，或者从基线架构的定义到细粒度细节的实现来支持软件系统的构建。今天，模式的使用则成为软件专业人员的至爱。

过去的15年里，有大批的文献从软件开发的方方面面阐述了已知的模式，内容涉及组织和流程、应用和技术领域以及最佳编程实践。这些文献为软件工程师提供了具体的实践指南，并日益影响学生的教育。每年都有关于更多模式的书和会议纪要发表，以模式的形式从深度和广度上扩展软件开发知识。

同样，我们关于如何应用模式方面的知识和经验在稳步增长，模式概念本身相关的知识亦是如此：内在的属性，不同的风格以及与其他技术的关系。自20世纪90年代中期以来，尽管在软件模式领域中概念性的知识飞速增加，记录和重构具体模式的出版物的数量也在不断增长，而关于模式及模式概念的出版物却只在少数领域稍有更新。《面向模式的软件架构（卷1）》[POSA1]、《设计模式》[GoF95]以及《软件模式》白皮书[Cope96]中关于软件模式的介绍仍是模式概念最相关的来源。另外，最近才有出版物开始明确讨论和阐述模式序列[CoHa04][PCW05][Hen05b]。

总而言之，在模式概念方面没有紧跟技术发展的著作出现。此外，关于模式的概念性基础知识的最新发展也仅停留在模式社区少数专家和思想先驱们的头脑中。本书的目的在于挖掘并阐述模式相关的知识，以满足广大软件开发社区的需求。同时本书是“面向模式的软件架构”系列丛书的第5卷，也是丛书的最后一卷。

本书将呈现、讨论、比较和关联众多模式概念中已知的特性和应用：单个模式、模式互补、模式复合、模式故事、模式序列和模式语言。对于每个概念特性，我们将研究其基础属性和高级属性，探究模式社区广为接受的见解，以及目前仍在探讨和争论的观点。我们还将讨论模式与软件开发中广泛使用的其他技术如何相互作用和支撑。简而言之，我们纵览了软件模式知识和实践当前发展的最新动态。

尽管如此，读者需要注意到我们从总体上对模式概念本身进行了宽泛的阐述和探讨，我们用来举例说明或激发模式不同方面的各具体实例，主要将重点放在软件设计模式上，而不是诸如组织模式、配置管理模式和特定的应用领域模式之类的其他模式上。上述（自我）限制的原因有两个：首先，大多数已经文档化的软件模式都是软件设计模式，因此我们有丰富的材料用作举例；其次，软件模式的最大用户群是架构师和开发者，因此将重点放在软件设计模式上使我们能够使

用这个群体中最熟知的实用的例子，来解释模式背后的“理论”。

目标读者

本书的主要读者是对模式的概念性基础原理感兴趣的软件专业人员。我们的首要目标是帮助上述软件专业人员，从深度和广度上完善他们对模式概念的认识和理解，从而使他们知道模式能够给他们的项目带来什么，以及如何对他们的项目有所贡献。其他的目标是帮助软件专业人员避免常见的模式方面的错误概念，并且将具体的模式有效地应用到日常的软件开发工作中。

本书同样适合在软件工程、编程语言、运行时环境和工具方面有扎实基本功的本科生或研究生。对于这类读者，本书能够帮助他们了解什么是模式，以及模式如何对设计和实现高质量的软件有所帮助。

结构和内容

本书分为3部分，每部分又由几章内容来展开阐述其观点和内容。

第0章揭示了模式概念定义的来龙去脉，探讨了软件社区中是如何衍生出这些模式的定义的，以及如何理解这些定义。我们的分析表明调整和改进模式的相关概念，对于深入理解模式和防止在软件项目中错误使用模式大有裨益。作为介绍性章节，本章详尽阐述和讨论了模式相关概念的调整和改进，给读者提供了一个更为完整、一致的模式概念视图，为本书后续的3个主要部分奠定基础。

第一部分讲解了单个模式的用法，从整体上剖析了过去10多年中我们收集的关于模式的观点。这些观点作为现有模式定义的补充，有助于我们从更深层次理解模式。

第二部分将讲解的重点从单个模式转移到模式之间的关系：有时几个模式组合在一起成为另外一个模式的替代或补充，有时这些模式紧密组合成为一个模式集合。除了常见的关于模式集合的观点之外，该部分还探讨了如何将模式组织为模式序列，使用一个接一个的模式形成模式描述流，从而在设计流程中动态有效地使用模式。

第三部分在前两部分的概念和结论的基础上介绍了模式语言。在为特定技术或应用领域设计和实现软件时，与单个模式和模式序列相比，模式语言更加从整体上使用模式。为达到该目的，模式语言针对相关领域中的每个问题选择多个模式，将它们组织在一起，形成一个高效的领域相关软件开发流程。

第14章撷取了前面3部分讨论的模式相关概念，得出如下结论：尽管模式相关技术为其他软件技术提供支撑，但模式的主要受众还是人。

第15章回顾了在2004年“面向模式的软件架构”系列丛书的第3卷中关于模式未来发展趋势的预测，讨论了过去3年里模式真正的发展方向，分析了模式及模式社区的现状。基于对模式以往发展历史的回顾，我们重新修订了关于模式及模式语言未来发展趋势的预测。

本书是我们计划出版的POSA系列丛书的最后一卷，至少目前是如此。第16章概括了过去超过15年的模式相关工作和经验，检查了在这期间我们所著的5卷POSA系列丛书。

本书最后总结了我们所讨论过的模式概念。最后简短概括了本书使用的所有模式，列出了所有对完成本书有所贡献的参考文献。

毫无疑问，我们难免遗漏个别的模式概念的属性和定义，新的模式概念也会随着对模式及其在实际软件开发中应用的理解不断深入而出现。如果你对本书的内容和写作风格的提高有任何评论、建设性意见或建议，请发电子邮件到 siemens-patterns@cs.uiuc.edu。在模式主页 <http://hillside.net/patterns/> 可以找到模式相关的注册指南。该链接同样提供了关于模式方方面面的重要信息来源，如已出版和即将出版的书、模式相关的会议及论文等。

致谢

非常感谢在完成本书的过程中支持我们的诸位，他们有的分享他们的知识，有的帮忙审校本书的各部分草稿并提供有用的反馈。

首先，谨以此书献给 John Vlissides，以表达我们的谢意。John 是软件模式领域中最为才华横溢的大师之一，他是软件模式领域开辟新天地的思想领导人和探路者，同时也是《设计模式：可复用面向对象软件的基础》[GoF95]一书的合著者，以及当今众多世界级知名软件模式专家的顾问。他的著作给予我们灵感以及模式概念的基础，深深地影响我们，并极大地帮助我们在本书中详细讲解和讨论模式的相关概念。

审校奖的桂冠颁发给 Wayne Cool、Richard P. Gabriel、Michael Kircher、James Noble 和 Linda Rising，他们仔细地审校了本书的所有内容，从正确性、完整性和一致性上保证了本书的质量。他们的反馈极大地提高了本书内容的质量。Wayne Cool 还贡献了本书深入探讨的诸多观点和思路。

另外，本书的部分内容在四届 EuroPLoP 模式会议上进行了演讲，并呈送给几位模式专家。Alan O'Callaghan、Lise Hvatum、Allan Kelly、Doug Lea、Klaus Marquardt、Tim O'Reilly、Michael Stal、Simon St. Laurent、Steve Vinoski、Markus Völter、Uwe Zdun 和 Liping Zhao 给予我们大量的反馈，使我们更正了大大小小的关于模式概念诸多定义和表述的错误。

非常感谢 Mai Skou Nielsen，允许我们在本书中使用她作品中的照片。

特别感谢 Lothar Borrman 和 Reinhold Achatz 在管理上给予的支持，以及德国慕尼黑 Corporate Technology of Siemens AG 软件工程实验室给予的帮助。

尤其感谢我们的编辑 Sally Tickner、前任编辑 Gaynor Redvers-Mutton 以及 John Wiley & Sons 的其他人员。没有他们，就没有本书的出版。在参加 2002 年 EuroPLoP 会议时，Gaynor 于某个晴朗的夜晚说服我们著本书，并陪伴我们度过了写作过程的前两年。在接下来的两年里，Sally 以极大的耐心陪伴我们完成了本书的手稿。同样要感谢来自 WordMongers 公司的文字编辑 Steve Rickaby，他为本书的内容增色不少。Steve 以他的建议和支持，陪伴我们完成了这 5 卷 POSA 书。

最后，感谢我们的家人，感谢他们在本书写作过程中的耐心和支持！

读 者 指 南

White Rabbit戴上眼镜。

“陛下，我从哪儿开始？”他问到。

“从起点开始，”国王非常严肃地说道，“继续直到终点，然后结束。”

——Lewis Carroll, 《爱丽丝梦游仙境》

基于本书的组织结构和写作特点，最为便捷的阅读方式就是从头读到尾。如果你知道自己需要读哪部分，也可以按照自己的方式来阅读本书。为了更有效地阅读本书，下面的提示可以帮你选择阅读主题和顺序。

鸟瞰模式

本书深入探讨了模式的概念。以一个广为流传、简短但不完整的模式定义开始，我们首先引入、审视和说明单个模式的内在属性。深入理解什么是单个模式，什么不是单个模式，有助于在软件开发中有效应用各模式。

随后，探究了模式之间的区别与联系。模式多组合在一起使用并通过多种关系联系在一起。组合在一起的模式或者成为其他模式的备用模式，或者成为其他模式的天然互补，或者定义为特定的模式排列以便作为一个整体来使用。模式亦可组成特定的模式序列，当应用模式序列时，可以生成并通知具体软件系统的架构。知道并理解各模式关系的方方面面，有助于在软件开发中有效使用一系列模式。

最后，我们以模式之间各种形式的关系来充实现单个模式的概念，并详尽说明模式语言的概念。模式语言将一系列模式组织在一起，为特定应用或技术领域进行软件设计而生成一套软件开发流程。当我们10多年前开始写作“面向模式的软件架构”系列丛书时，模式语言实现了我们心中基于模式的软件开发的目标和愿景。

本书中所探究的所有模式概念都是相互依赖的。模式之间的各种关系使用了单个模式的属性，模式语言则进一步以模式之间的关系为基础，并加以利用。本书首先对模式的特点进行了非正式、直觉上的定义，然后逐步挖掘，深入讲解了模式概念的不同属性和方面，最后我们描述了一幅完整、一致的模式图像：模式的定义、模式的误区，模式如何辅助我们进行软件开发，以及模式如何与其他软件技术和技巧相关联。

从特定视角看模式

如果你更喜欢采用其他的顺序来阅读本书，比如想将重点放在某个模式概念的特定方面，我们建议你按照最感兴趣或最相关的顺序来阅读相应部分和章节。不过需要注意的是，本书每一部分和章节都是在前面各部分及章节的基础上来探究和阐释概念的。如果你对正在阅读的章节中的某些内容不熟悉，可以参考本书最后的“模式概念总结”，其中概要介绍了本书涉及的模式概念的方方面面。如果你不熟悉我们提到的任何模式，可以参考本书最后的“参考模式”，其中列举了每个模式的来源以及对该模式的简短描述。

模式之同源同宗

因为我们采用基础的入门级的方式来阐述模式概念，所以即使你初次接触模式，也能够读懂本书。同样，我们继承并集成了20世纪90年代中晚期发表的关于模式概念的著作，包括《面向模式的软件架构（卷1）》[POSA1] 和《设计模式》[GoF95]（其中概述了软件架构及设计模式相关的基本概念和术语），以及Jim Coplien的《软件模式》白皮书 [Cope96]（其中详细讲解了模式语言的概念）。如果熟悉了这些书，你将发现本书是如何增强、扩展模式知识的，我们希望能够彻底阐述清楚这些知识。

目 录

第 0 章 尘埃落定	1
0.1 源源	1
0.2 成功与失败	2
0.2.1 观察	2
0.2.2 状态	3
0.2.3 处方	4
0.3 模式定义及解释	5
0.4 深入理解模式	11

第一部分 模式剖析

第 1 章 问题之解决方案及其他	15
1.1 问题之解决方案	15
1.1.1 一个例子 (1)	15
1.1.2 重现与良好	16
1.2 流程和物件	16
1.3 “好”的解决方案	17
1.4 驱动力：模式之心脏	19
1.4.1 一个例子 (4)	19
1.4.2 dysfunctional、bad 还是 anti	21
1.5 上下文	22
1.5.1 一个例子 (5)	23
1.5.2 上下文的一般性	23
1.5.3 独立上下文	24
1.5.4 一个例子 (6)	25
1.6 一般性	25
1.7 一图胜 (逊) 千言	27
1.8 模式命名	29
1.8.1 模式命名的语法分类	30
1.8.2 字面命名还是隐喻	30
1.9 模式是循序渐进的	31
1.10 模式既是讲故事，又能发起对话	33

1.11 模式不能代替思考	33
1.12 从“问题-解决方案”到模式	34

第 2 章 多种多样的模式实现

2.1 是否存在一个通用的模型呢	35
2.1.1 Observer 模式：快速回顾	36
2.1.2 结构的变化与角色	36
2.1.3 行为的变化	37
2.1.4 内部差异	38
2.1.5 语言及平台的差别	39
2.1.6 领域、环境相关的变化	40
2.1.7 再论假设	41
2.2 模式与框架	42
2.2.1 工具和上下文环境	44
2.2.2 两个框架的故事	44
2.3 模式与形式主义	47
2.4 通用性与特殊性	48

第 3 章 模式格式

3.1 风格与实质	50
3.2 格式的功能	52
3.3 格式的元素	53
3.4 细节	55
3.4.1 案例	56
3.4.2 图示	57
3.4.3 <code>...</code>	57
3.5 鸟瞰图	58
3.5.1 从金字塔到托盘	58
3.5.2 模式骨架	58
3.5.3 总结意图	60
3.5.4 模式抽象	60
3.6 不同的格式	60

3.6.1 演变	61	6.4 是元素还是组合	100
3.6.2 选择	62	6.4.1 组合的含义	100
3.7 风格与实质 (Redux)	63	6.4.2 深入 MVC	101
第二部分 模式之间的关系			
第 4 章 模式孤岛	67	6.5 复合分析与综合	102
4.1 模式的联系	67	6.5.1 非设计模式的复合	102
4.2 设计实验：将模式作为孤岛	68	6.5.2 设计模式复合	103
4.3 第二个设计实验：交织在一起的模式	72		
4.4 模式密度	73		
第 5 章 模式的互补性	75	第 7 章 模式序列	104
5.1 一个问题，多种解决方案	75	7.1 模式讲述软件工程成功的故事	104
5.2 互相竞争的模式	76	7.2 模式故事	105
5.2.1 以状态为例	76	7.2.1 一个小故事	105
5.2.2 模式族	79	7.2.2 已经发表的故事	107
5.2.3 迭代开发	80	7.3 从故事到序列	108
5.2.4 适配开发	83	7.4 模式的序列	109
5.2.5 遵从康威定律	84	7.4.1 一个早期的例子	110
5.2.6 与风格的设计对话	86	7.4.2 模式序列既是流程也是物件	110
5.3 互相合作的模式	87	7.4.3 再次回到以前提到的小故事	111
5.3.1 一个关于值的例子	88	7.5 回顾模式复合和模式互补	112
5.3.2 设计上的完善	88	7.5.1 重组	112
5.4 模式结合	89	7.5.2 再论 Batch Iterator	113
5.4.1 再论迭代	89	7.5.3 再论 Interpreter	114
5.4.2 再论适配编程	91	7.5.4 再论 Align Architecture and	
5.5 互补性：竞争、完善、结合	92	Organization	115
第 6 章 模式复合	93	7.6 回到上下文的问题	116
6.1 常见模式排列	93	7.6.1 定义上下文	116
6.2 从元素到复合	93	7.6.2 专用化与差异化	118
6.2.1 Pluggable Factory	94	7.7 模式间的联系	118
6.2.2 Composite Command 的两种			
视角	95	第 8 章 模式集合	120
6.2.3 模式复合的格式	96	8.1 模式手册	120
6.3 从补充到复合	97	8.2 组织模式集合	121
6.3.1 重申	97	8.3 即时组织	121
6.3.2 适配	99	8.4 根据层次组织	122
		8.4.1 设计和架构	122
		8.4.2 惯用法	123
		8.4.3 混合层次	124
		8.4.4 层次	124
		8.5 根据领域组织	125
		8.6 根据分区组织	125

8.6.1 阶层架构	126	10.8 完整的语言胜过千幅图	163
8.6.2 分区	126	10.9 面向领域的命名帮助忆起模式语言	164
8.7 根据意图组织	126	10.10 模式语言展开对话并讲述很多故事	165
8.7.1 根据意图划分 POSA 的模式	127	10.11 路还很长	165
8.7.2 根据意图划分 GoF 的模式	128	10.12 模式语言对创造性智慧的回报	167
8.7.3 根据意图划分 DDD 模式	128	10.13 从模式网络到模式语言	168
8.7.4 反思模式意图	128		
8.8 组织模式集合 (重奏)	128		
8.9 问题框架	129	第 11 章 亿万种不同的实现	169
8.9.1 问题框架	130	11.1 众口难调	169
8.9.2 问题框架和模式的对比	131	11.2 演进式成长	169
8.9.3 问题框架与模式的组合	132	11.2.1 面向系统的、进化的 设计方法	170
8.10 模式符号学	132	11.2.2 演进式成长和敏捷开发	171
8.11 模式集合与风格	134	11.3 并没有排斥重构	172
8.11.1 Unix 接口设计模式	135	11.4 一次一个模式	174
8.11.2 Web 2.0 设计模式	136	11.4.1 明白手头上的问题的关键	174
8.11.3 风格与概念一致性	137	11.4.2 优先级驱动的设计决策	175
8.12 走向模式语言	138	11.4.3 模式集成先于模式实现	175
		11.5 基于角色的模式集成	176
		11.5.1 选择 1: 识别并且保持 已经实现的角色	176
		11.5.2 选择 2: 识别并分离已经 实现的角色	177
		11.5.3 选择 3: 将缺失角色分配给 既有设计元素	178
		11.5.4 选择 4: 将缺失角色作为 新设计元素来实现	179
		11.5.5 基于角色的模式集成和 演进式成长过程	180
		11.6 模式语言和参考架构	180
		11.7 模式语言与产品线架构	181
		11.8 从十亿个到一个	184
		第 12 章 模式语言的格式	185
		12.1 风格与本质	185
		12.2 格式的作用	185
		12.3 格式的元素	186
		12.3.1 展示全貌	187
		12.3.2 简洁与细节	187

12.3.3 模式连接	189	14.3.2 用户需求	206
12.3.4 再说元素	189	14.3.3 交到用户手中	207
12.4 细节, 细节, 细节	190	14.4 对模式作者的支持	207
12.4.1 模式语言的格式	190	14.4.1 协同写作	207
12.4.2 鸟瞰图	190	14.4.2 作者研讨会	208
12.4.3 展示顺序	194	14.4.3 牧放	208
12.4.4 示例	194	14.5 技术为人	209
12.4.5 细节程度	195		
12.5 再论风格与本质	197		
第 13 章 模式与模式语言	198	第 15 章 模式的过去、现在和未来	210
13.1 模式和模式语言: 共性	198	15.1 过去的 3 年	210
13.1.1 共同的核心属性	198	15.1.1 模式与模式语言	210
13.1.2 共同的根源	199	15.1.2 理论和概念	212
13.1.3 一个模式的模式语言	199	15.1.3 重构与集成	212
13.2 模式与模式语言: 区别	200	15.1.4 GoF	213
13.2.1 模式和细节	200	15.2 模式的现状	213
13.2.2 模式语言和交互	200	15.3 模式的明天在哪里	214
13.2.3 两个独立的世界	201	15.3.1 模式与模式语言	214
13.3 模式“对”模式语言	201	15.3.2 理论和概念	216
第 14 章 从模式到人	202	15.3.3 重构与集成	216
14.1 模式以人为本	202	15.3.4 支持其他的软件开发方法	216
14.1.1 模式价值体系	203	15.3.5 对其他学科的影响	217
14.1.2 人类读者	204	15.3.6 其他学科对我们的影响	217
14.2 对软件开发者的支持	204	15.4 简述模式的未来	218
14.3 对软件使用者的支持	206	第 16 章 万事如意	219
14.3.1 用户界面	206	模式概念总结	221
		参考模式	225
		参考文献	238

第0章

尘埃落定



把失误当做取得更深理解和更高成就的阶梯。

—— Susan L. Taylor, 记者

本章回顾了模式的定义，该定义出自1996年《面向模式的软件架构》系列丛书的第一卷，并针对该定义进行了细致彻底的分析。我们的分析结果表明：为了避免模式概念理解上的偏差，有必要甚至必须做一些调整和改进，以避免在软件项目中错误使用模式。

0.1 源渊

说起软件开发中模式的出现和采用，关键事件是Bruce Anderson于1990年在ECOOP会议上主持的名为“birds of a feather”（同类人）的聚会。其议题名为*Towards an Architecture Handbook*（关于架构手册），这之后的三次OOPSLA会议上均成立了针对该议题的专题研讨会。“四人帮”（GoF）也正是结识于此[GoF95]：

关于软件架构研究方面的第一次合作经历，来自于OOPSLA'91会议上由Bruce Anderson所主持的专题讨论。这次专题研讨会的目的是为了创作一本软件架构手册（从内容上看，我们认为与“架构手册”相比，“架构百科全书”更合适作为该书的书名）。

OOPSLA会议的这些专题研讨会也促成了早期面向模式软件架构方面的图书的出现。整理出一本软件架构手册或者百科全书是一项很耗时的工程，它鼓舞了模式社区内的很多人，深入挖掘和总结其软件编程经验。其中最值得一提的是Grady Booch，他调查了现行的软件系统，寻找其中好的模式实例[Booch]。

GoF的模式奠基著作[GoF95]的出现更加激发了人们对于模式的热情。之后出现的Pattern Languages of Programming（程序模式语言，PLoP）会议的精选模式集[PLoPD1]、[PLoPD2]、[PLoPD3]、[PLoPD4]、[PLoPD5]，《面向模式的软件架构》系列的前4卷[POSA1]、[POSA2]、[POSA3]、[POSA4]，将人们对模式的热情推向了高潮，更早期的火种包括软件惯用法图书[Cope92]和用户接口模式[BeCu87]等。

自从1994年GoF图书发布，全世界的软件开发者们就开始拥抱这种“新思想”了，他们期望

这些高雅、简洁、全能的模式能够帮助他们轻松解决复杂的问题。模式的概念和很多特定的模式已出现在众多软件项目的词汇和代码中，以及相关图书和大学课程中。模式革命的星星之火已成燎原之势。

0.2 成功与失败

模式之所以成功的一个原因是，它们只是构成一个基本的方案框架，然后由熟练的软件设计者的集体智慧来逐步完善。虽然项目开发各有不同，但这些项目出现的很多全新问题，基本不需要新颖的解决方案。项目开发者们有时会开发类似的软件，或者常常会记起他们曾经在别的系统成功解决过类似的问题，对于这类问题，开发者可重用解决方案的核心，并根据新问题的细节作相应调整。高级的架构师和开发人员可能会勾画出这些“解决方案模式”的大体情况，以解决在开发新系统或维护已有系统时出现的设计问题。

从具体应用的设计问题及其解决方案中提炼出其中的共同点，这正是模式概念的核心主题：模式刻画了设计问题和相应的解决方案之间的关系，并以易用的形式归纳出来。模式传授给初学者很多复杂工程中需要用到的概念，并帮助他们在选择合适的设计和实践方式时提高信心。模式可帮助专家开发复杂的软件系统，提供常见问题的不同视角，并提供其他专家的经验以供参考。软件开发领域中很少有其他概念具备这样的属性。

现在有很多成功的模式案例：在这些案例中，系统架构及开发大获成功，因为它们有意识地、精心地应用模式进行设计和改进。这些案例来自电信系统[Ris98]、商业信息系统[KC97]、工业自动化[BGHS98]、仓库管理[Bus03a]、网络管理[KTB98] [Sch98]、网络协议[HJE95]、通信中间件[SC99]以及其他工业级系统[BCC+96]。

模式的角色及其成功应用还扩展到软件设计领域之外。模式已经不断地推广和应用到很多不同的领域，包括组织引入变革模式[MaRi04]、组织结构模式[CoHa04]、开发实践模式[Cun96]、教学模式[PPP]、配置管理模式[BeAp02]、安全模式[SFHBS06]、文档模式[Rüp03]，甚至还有撰写模式的模式[MD97]！很多模式实例及其应用在其他领域中也不失为高效的策略。

0.2.1 观察

不过成功并不是绝对的，而是需要条件的。现实中失败的案例也有很多：有时我们在设计中对模式的运用很用心，也很明确，但到头来，设计出的架构体系却表现出不必要的复杂。下面来看一个案例。

在软件开发过程中，当某一对象需要依赖其他对象提供的信息时，通常会带来一些问题。信息提供者应能随时将更改的信息通知给信息接收者。团队知道Observer这个模式，而且看起来也像是解决此类问题“正确的”方法——信息提供者就是Subject，而信息消费者则是Observer。因此，团队决定采用Observer模式解决该问题。

信息消费者在作为Observer的同时又是另一套组件的信息提供者。这时可以再次使用Observer模式。这就带来了新问题。在GoF的书[GoF95]中，没有提及一个类同时作为Subject

和Observer的情况。那么，怎样构建这个组件才能不违背Observer模式呢？

解决方案是为该组件引入3个类，如图0-1所示。第一个类实现组件的Observer功能，而第二个类实现组件的Subject功能，第三个类通过多继承的方式从这两个类派生而来，它所实现的功能是对其两个超类进行状态协调和计算。

换句话说，这两处都可以使用Observer模式，团队都忠实地遵循了GoF书中所介绍的Observer类图，然后将二者通过一个黏合组件连接起来。这样一项独立而内聚的职责被分散到3个不同的类里面，降低了设计的内聚性，这样做的代价是设计变得更加复杂而难懂。

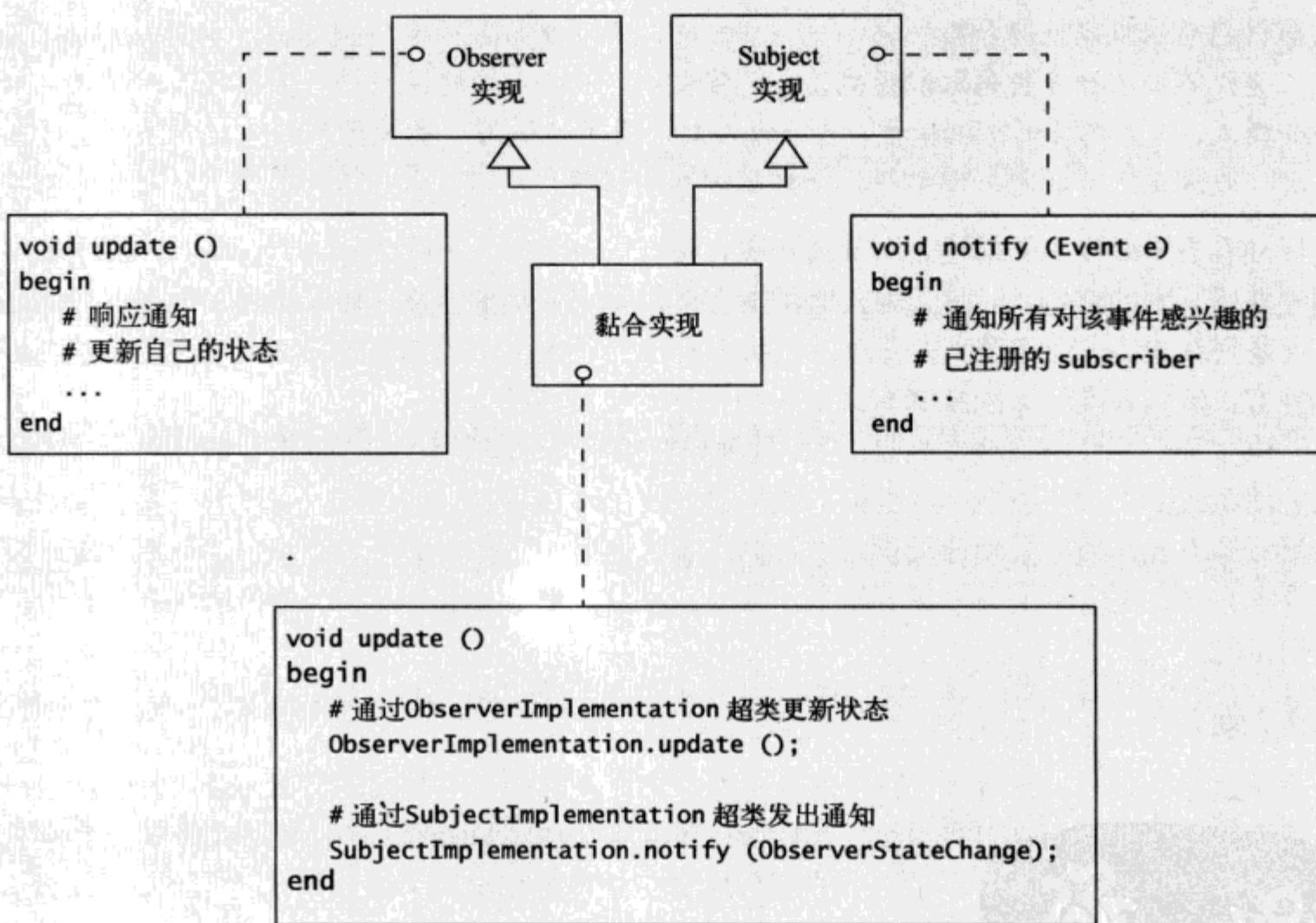


图 0-1

根据技术专家的建议，团队找到了一个更好的解决方案——将3种角色实现在一个类里面。这个“集成”的设计比那个“3个类”的设计更容易解释，也更容易维护。而且尽管它没有完全遵循GoF书中的类图结构，但仍然是Observer模式的准确实现[GoF95]。

0.2.2 状态

当然，Observer模式不是唯一的例子。

另一个开发小组，最近引入了UML和GoF模式，发现他们系统中的很多类都是以模型

及事件驱动方式描述对象的。最初他们曾为以序列图描述关键方面的方式而争论过，后来他们想到了状态图，并通过使用这些状态图极大地简化了对各个对象生命周期及对象间交互的描述。

接下来就是怎么把这些状态图转换为代码了。他们在GoF的宝典中发现了State模式。State模式是在GoF编目中唯一的直接处理这种状态行为实现的模式，而且名字也正相符。这是一个复杂的模式，它向系统中引入了多个新类，将对象的行为分散到这些类里面。依据这种预先设计的方法，团队选择将他们所建模的每个对象的生命周期实现为一个状态图，于是他们得到了很多状态图——这就是他们心目中的State模式。结果导致了类爆炸，设计越来越复杂，越来越难以理解和维护。

这样不加选择地使用State模式，一方面来自于对其名称的误解——认为它是一种狭义的State模式，而非广义的State模式；另一方面则是简单地认为一类设计问题必须用同样的方式解决，或者说所有的状态图都得用一样的方式来实现。

仔细看看State模式的描述，你就会发现它还有一个名字，叫做Objects for States，这个名字更精确地描述了它的结构，同时又为其他解决方案保留了足够的空间。往深里说，很多设计问题的相关性是很肤浅的，比如很多人喜欢使用状态机作为对象生命周期的抽象，但是这并不意味着使用这种方式建模的类，必须或者有必要采用一样的设计方法。

无论这种一刀切的方法多么吸引人，这种凭空想象出来的简化方式都是不靠谱的。在本例中，它只是使系统变得过于复杂，这里根本就不适合使用这种重量级的模式。有时候，通过查找表或对象集合的方式管理对象生命周期是合理的方案。在另外一些情况下，引入一个特定的标志变量或者查询方法，可以使这个方案更简单。当然，有些案例使用Objects for States模式是合情合理又简单明了的。

0.2.3 处方

之所以称为设计，是因为人们在这个过程中需要找出多个可替代的方案。模式可以使这些方案连同选择的标准——它们的驱动力（force）和结果（consequence）——更加清晰。这或许揭示了在之前提到的State故事中，小组中的某个成员在发现有超过23个设计模式可用于软件开发后，表示失望的原因。人们原本希望GoF为他们煲好了万能药送到嘴边，可是这个愿望如今落空了。

问题在于为什么这样或那样类似的事情会反复上演。为什么很多项目不能成功应用模式呢？更严重的是，为什么会有这么多项目不顾以往积攒的宝贵经验教训，一次又一次地失败呢？毕竟，所有这些项目都期望能从使用模式中获益。一种愤世嫉俗的观点是质疑整个模式概念是否真是获取和交流各种设计知识的有效方法。通过细致地探索本书中的模式概念和例子，我们希望能重申模式是一种更有效的命名、组织和论证设计知识的方式。

一个相对缓和但却更加悲观的观点认为没有什么是完美的，在任何人类活动中错误都不可避免。一个相对有建设性的观点是在接受人类不完美的观点的同时，认为还是有通过不断磨练提升的空间。很多项目未能有效应用模式，他们往往宣称是严格遵循流行的模式定义，完全按照模式的描述来使用的。或许软件模式概念当前的定义，甚至各个模式自身及其内容中，还有不足的或

者被忽略的部分?

0

0.3 模式定义及解释

为了寻找前述问题的答案,帮助开发者得到更充分、完整、现实和准确的模式描述,我们需要对一些流行的模式定义以及软件社区中对模式概念的主流解释进行研究。

很多新近的关于软件开发的图书中都提及了软件模式的定义,如果作者本身未提出定义,他们一般都会明确地讨论或者参考别人提出的定义。在这些书中,作者的一般意图是介绍模式的概念,并根据图书的需要和范围尽可能准确和贴切地对其加以阐述。以下定义来源于《面向模式的软件架构(卷1)》[POSA1]。

软件架构模式描述的是在一定设计上下文中反复出现的某个特定的设计问题,并提出一个已被证明的解决方案的一般性结构。该解决方案包括对主要组件的描述,组件的职责和关系,以及组件之间协作的方式。

依据这一定义,[POSA1]和[Sch95]讨论了软件架构模式的很多属性,下面我们来重温一下。

□ 模式文档的最佳实践基于久经考验的设计经验。模式并不是仅针对模式本身而人为发明和创造出来的,而是“提炼并提供一种方法来重用经验人士总结的设计知识”,这样熟悉一系列模式的开发人员“可以立即把它们应用于设计问题而不需要重新总结”[GoF95]。如果模式应用适当,其效率通常是可信的。

经验和最佳实践一般存在于有经验的开发人员的脑中,或者深埋于复杂的系统源代码里。通过模式清晰地说明关键的设计策略和技巧,可以帮助初学者和新参与到项目中的人员快速提升他们的水平。这样可以让有经验的开发人员更有效地分享他们的知识,让所有参与者能够基于一个良好的角色模型展开工作。

□ 模式所提供的抽象概念要高于单独的对象、类和组件。仅凭一个特定的应用组件,很少能解决复杂问题——至少不会是非常有效的解决方案,往往也不够内聚。大部分模式的解决方案都引入了多种角色,而各个应用组件(包括可部署的组件、类以及对象等)扮演着各个角色。另外,这些方案指明了它们引入的角色之间的关联和协作。例如,Observer模式引入了两个主要角色:subject和observer。它们通过推送(push-based)和变更传播(change-propagation)机制协作,保持组件间状态的一致性。

□ 模式为设计概念提供了公用的词汇表和一致的理解。这些词汇方便了架构知识及相关物件的重用,即便是特定算法、实现和接口以及一些细节设计不可能得到重用的情况。如果开发者了解Observer模式,他们就不需要对如何使系统中两个组件协调工作进行太冗长的讨论。他们的对话可以简化成:“好的,我引入Observer模式,一个对象扮演被观察者角色,其他对象是它的观察者。事件通过回调方式通知观察者,观察者依据对事情的兴趣注册到被观察者中,同时被观察者将其状态信息发送到观察者。”

熟悉Observer模式的开发人员可以很容易理解讨论中的设计并画出类似图0-2的框架图。

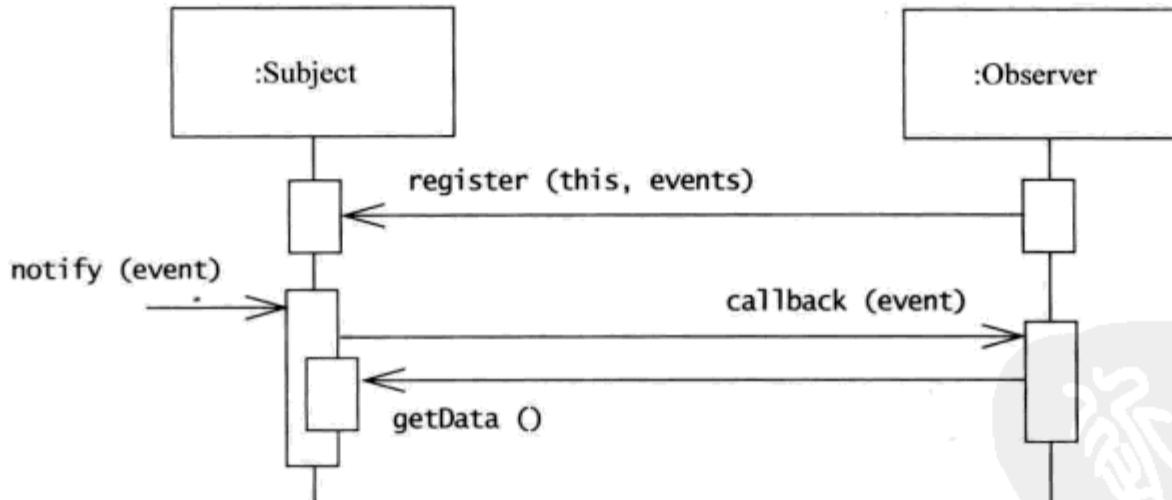
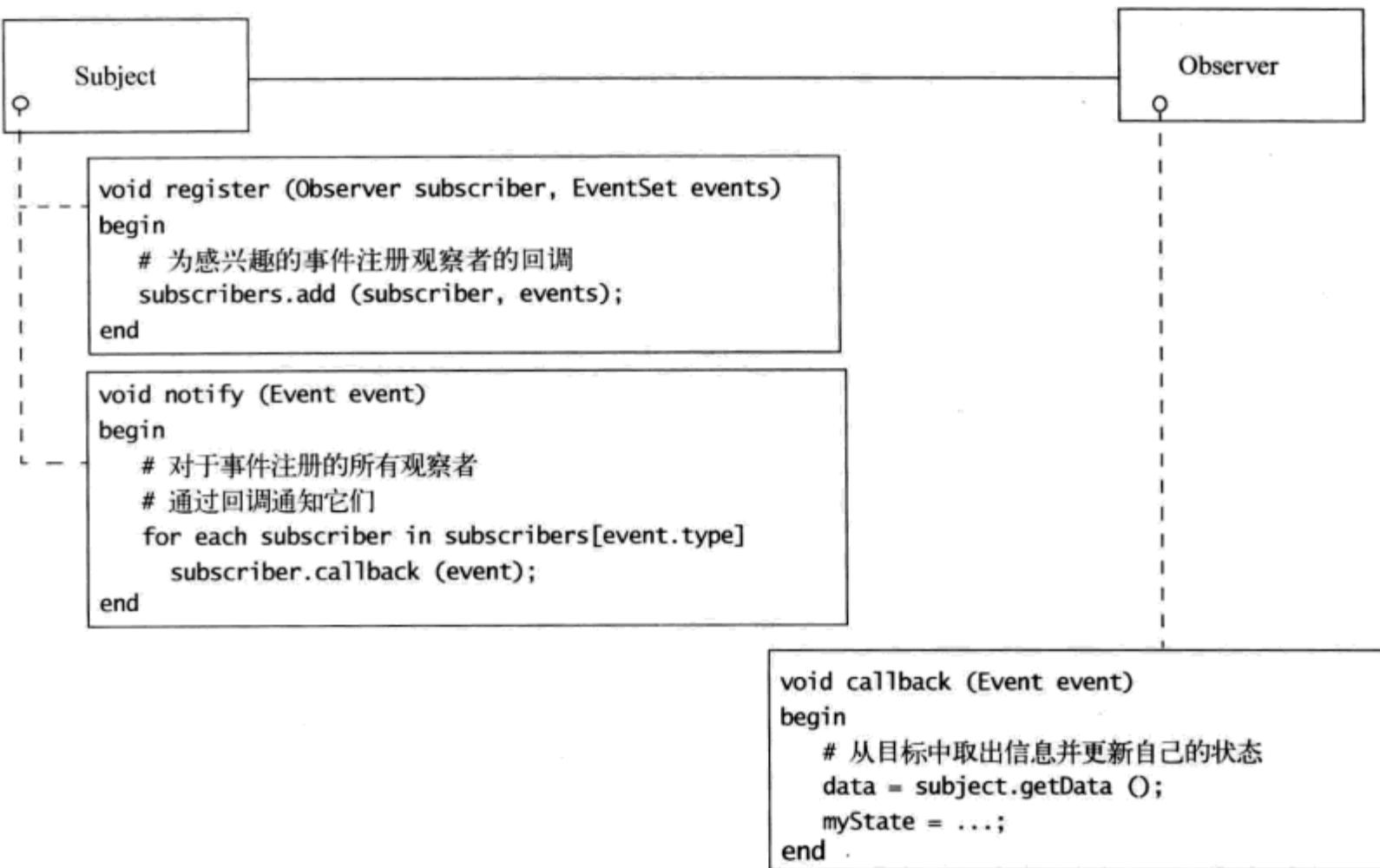


图 0-2

模式有助于化解掉我们对于编程语言和API无谓的执著，后者常常给软件项目带来麻烦。模式促使有着不同编程语言文化的开发者可以共享相互有兴趣的设计知识，而不会被“语言大战”所束缚。所以模式不仅有利于不同社区成员间的交流，而且由于识别出了哪些属于惯用法，特定的语言社区也可以重新回到自己的关注点上来。C++语言中的惯用法对于Java来说并不总是有用的，有时候甚至完全不适用，反之亦然。但是这些惯用法对于自己社区来说却是非常重要的。

模式也有助于不同时代的开发人员传递知识和他们熟悉的技术。代码看上去不同，“缩

写”各式各样，但支持这些新旧代码的模式却是相同的。例如，在《面向模式的软件架构（卷2）》[POSA2] 和 *Server Component Patterns* [VSW02] 中记录了很多起源于分布式对象计算领域的模式，其中“面向服务架构（SOA）”[Kaye03] 和“组件模型”[VSW02] 等方法直到现在还是很流行、很适用。知识传递也发生在应用概念，并不局限于技术本身，领域驱动设计[Evans03]就是一个例子。

- 模式是对软件架构进行文档化的方法。由于模式可以以显式方式评估后果和实现中的权衡，我们常常使用模式来记录选择某个设计和否定另一个设计的原因。在软件开发过程中，以模式的方式记录设计的意图、结构和行为，有助于搞清楚软件开发和维护[Prec97] [PUS97]。对产品线架构而言，模式的应用是尤其有益的[Bosch00] [Coc97]。开发者可以努力正确有效地使用产品线架构，而不需要了解其基础结构、控制流以及针对特定应用所作的合理修改，而且这些架构核心也不会被实现细节所掩盖。在基于一些如 GUI 框架或通信中间件等之类的软件基础设施之上构建应用时，也存在类似的情况。基于模式的架构文档有助于解决这种冲突，它让开发人员关注于关键的设计抉择及其背后的原因，避免过于拘泥细节，虽然这些细节对它的实现者而言很有意义、很重要，但对其他使用它的开发人员来说可能就不需要深入了解了。
- 模式所支持的软件结构应该有着明确定义的属性。一些模式定义了行为主干，可以满足特定领域应用的功能需求。例如，一些模式专门针对公司财务和会计系统[Fow97]、通信系统[Ris98]以及提升响应系统的能力[Mes96]。有的模式则关注于软件系统的运营和开发需求，例如资源管理[POSA3]、安全[SFHBS06]、性能与可伸缩性[POSA2]和扩展性[GoF95]。
- 模式阐述经验的形式不依赖于特定项目的细节和约束、实现范例甚至编程语言。通过抽象理解领域中潜在的陷阱和缺陷，模式可以帮助开发人员在选择合适的软件架构、设计和编码原则解决新项目中新的设计挑战时，避免浪费时间和精力去实现那些低效的、易出错的、难以维护的或者脆弱的设计方案。正如 POSA 系列[POSA1] [POSA2] [POSA3] [POSA4] 所描述的，它们几乎可用于所有类型的系统中：面向对象的或者面向过程的，分布式的、并发的或者可配置的、顺序的，基于组件的或者整体架构的，基于现成的中间件的或者完全自行开发的。

其他著名的软件模式的定义和我们在 *A System of Patterns* 一书中的定义大同小异。例如，四人帮[GoF95]提出的定义以及 Brad Appleton[App96] 从不同来源中搜集到的定义讨论的都是类似的模式属性。我们以及模式社区其他人所参考的定义都是来自 20 世纪末期。

这些“早期的”定义很好地将模式介绍给了广大听众，现在它们仍然很有用。参与了成功应用模式的项目的开发人员信任模式，并确信模式包含我们以上讨论的那些特征。但是这些定义未能足够深入地让人们真正理解模式。由于这些定义不愿意表现得过于规范或者过于正式，很容易产生误解并因此被误用。比如，我们发现了以下常见的陷阱和缺陷。

- 开发人员可能会给他们所有的软件开发活动和产出物都带上模式的帽子。例如，他们会把现有的算法和数据结构（例如二分法查找和链表）以模式的形式记录到文档中。另一个例子是来自于 *CORBA Design Patterns* [MM97] 的所谓的“Java 模式”，它试图将 Java 平台的一

些属性和益处以模式的方式重新讨论，并暗示Java是唯一有着这些属性和益处的方案。虽然这样的练习满足了作者的智力需要，但如果不能显著提高软件开发的关键质量，比如沟通或可懂性，或者软件的运营质量，例如效率和可扩展性，那么它只会带来不良的影响。

□ 模式并不只是一些优雅的设计。开发人员可能会迷恋于把每个新颖的复杂的设计，甚至每个跟系统相关的设计决定，作为他们发明的新模式。使用模式形式记录这些设计是有益的。*CORBA Design Patterns* [MM97]中提出了针对分布式计算技术的模式，如CORBA-CGI GATEWAY，但是这些模式定义是与CORBA的上下文和规范紧密联系的。另一些*CORBA Design Patterns*中提出的方案描述的设计虽然一再出现，但却是不切实际的，例如Distributed Callback模式中的解决方案。

把这些设计文档称为“模式”，给人的印象是这个应用程序的架构都基于合理的已验证的设计知识，但有时候并不是这样。这种印象使其他开发人员不去怀疑这些设计以及实现，模式常常被认为意味着高质量设计，所以如果一个设计是基于模式的，那它一定是完善的。盲目地信任任何设计，不管它是否基于模式，是无法保证软件达到期望的质量的。如果设计想获取模式“商标”，需要重现性和通用性，好的设计是可重现的、已验证的解决方案。

□ 开发人员倾向于把模式当做不变的东西——孤岛、蓝图或者类的特定的配置。虽然规范的图表和示例代码常常都是软件模式描述的一部分，但它们并不等于模式。然而，正如本章开始那个Observer模式的故事所描述的，图表即是模式的观点极为盛行。这种狭隘的观点使很多开发人员无法了解模式的实际含义：适应应用的特定需求是模式不可或缺的组成部分。模式应该被视为“一组互相作用的部件的协作，同时有着各种各样的限制”[Bat79]。遗憾的是，现有的模式定义中并没有阐述这种变化的含义。

□ 模式不是编程指南。编程指南可以从着眼于编程风格的模式中摘取内容，但反之却是不可靠的。虽然很多的编程指南属于惯用法，但不能直接把它们当做模式。例如，Sun公司的JavaBeans规范[Sun97]就主张使用get和set前缀的属性访问器，这点和“设计模式”相同。当这些前缀从“正则表达式模式匹配”意义上说成为“模式”时，无论是作为通用的设计还是特定于Java代码的设计，它们并不能解决真正的设计问题。这里确实用到了模式，但它不是使用get和set的问题：它通过使用一致的命名规则建立一个支持Dynamic Invocation Interface的协议。这个协议基于get和set前缀，但也可以使用foo和bar后缀的方式，虽然这相对不直观也不流行。

□ 错误地理解或者未能充分理解模式词汇会导致开发人员在面对问题时采用错误的模式。针对特定问题应该采用哪个模式需要准确的判断：误用模式会导致不合理的设计和实现。同样，对模式一知半解会让开发人员误认为他们已经充分了解了问题的解决方案。例如，对于Proactor和Reflection等复杂的模式，简单地了解其架构和参与者只是有效地应用模式的一个起点。陷阱存在于细节之中，模式的本质在于它的上下文和效果，而不是它的结构。因此，模式成功的实现依赖于可靠的设计和实现技术，而不是用以替代这些设计和技术的。

□ 开发者相信有了模式，只要通过遵循固定的规则或者应用方式，就能实现复杂的架构。把模式集成到软件开发过程中是一个劳动密集型活动。除了有效的软件技术和工具之外，

要想成功地应用模式，需要开发人员良好的团队合作，这些开发者们积累领域中的具体经验，并从无数的实现细节中获取软件架构和组件的有效属性。尽管信息技术的“平坦的世界”已经征服了广阔的地域[Fri06]，但奇怪的是，现在仍然只有很少的开发者能同时具备这两种能力。

- 有些软件系统，要想使其开发成功，必须要求开发人员不断探索新思想。对于这些“进入”新领域而缺乏相应软件开发经验的系统而言，创新和探索尤其重要。这方面的例子有早期开发的电子商务系统和早期的工业自动化软件系统。仅考虑验证过的模式（即在过去已成功解决问题的模式）并不能解决前沿的、创新的系统中的新的设计挑战。狭隘地关注模式，实际上只会阻碍这类系统的成功。当然，模式的作用也是必不可少的，但它们只关注于支撑平台（例如，配置管理和通信中间件）以及开发过程（例如，采用敏捷开发过程，裁减研发周期），而并不关注领域相关的设计细节和整体的系统架构。
- 要小心地控制期望，防止模式变得臭名昭著。开发人员和管理者常常误解如何有效地把模式应用到软件项目中以及把什么模式应用到软件项目中。例如，使用模式并不意味着模式自身能保证软件的正确性、可扩展性和有效性。那些有着紧张的发布计划的组织，以及那些把软件开发视为机械的、无创造性过程的组织，常常都不把足够的时间和开发资源投入到他们的系统设计及设计文化中。而他们从应用模式中也只能得到有限的回报，因为模式需要在适当的场合下，通过深思熟虑的实现才能发挥出其精彩之处。
- 应用模式不是完全的工具化的或者自动化的。虽然，模式对于生成式软件技术的成功使用有很大的促进作用，特别是面向方面的软件开发[KLM+97]和模型驱动软件开发[Sch06a] [SVC06]，但它们的使用没有完全机械化[Vö05b]，这也是管理者和开发人员在谈及模式及其在软件开发中的应用时常说起的一个期望。管理者和开发者们常常忘记人才是模式的主要使用者。模式的意图是成为骨架而不是蓝图。使用模式的关键在于了解要解决的问题，如何解决问题，以及解决方案存在什么样的取舍。只有在理解了问题并仔细选择了合适的模式作为其解决方案之后，才能将此方案的特定的实现自动化或具体实现到工具和语言中。换句话说：理解才是软件开发的瓶颈，而不是将源代码输入到文本编辑器中。自动化是软件开发的重要环节，但过度的模式自动化并不值得推荐。
- 实现一个可以自动识别代码和模块中的模式的工具的想法过于天真了[PK98] [ACGN01] [Pet05]。一些模式（例如Composite模式）具有典型的外观、不同的签名。然而，很多模式如果脱离其意图和动力则很难分辨：Strategy和Objects for States的类图是一样的；在一些设计中，Strategy又可以被看做Bridge的一个特例。
- 模式不是组件：它们并不是开箱即用的日用品，我们无法简单地按照某个预先的计划将其放入系统中。因为很多软件模式提供各种角色的组件的图表，人们期望能把它们视为架构或者模板，只要填写好空白处即可运行。最初的UML规范明确说明了它对模式即插即用方面的观点[RJB99]。

参数化的协作代表一系列参数化类元（classifier）、关联和行为，通过将模型（一般是类）中的元素绑定到模式的角色，可用于多种场合。这是协作的模板。

尽管这个定义将模式在UML领域内提高到了某个层次，但它也使得模式失去了很多自由。只有排除模式的很多关键特性（问题的本质、解决方案的基本原理、应用该解决方案的结果），才能把模式的概念强硬地结合到模型领域中。在软件中，参数化是一个有用的工具，它有时可作为描述模式的可变性和泛型的便捷的隐喻。然而，隐喻不是唯一的，“有时”不等于“总是”，泛型并不意味着参数化，还有很多其他的模式可以实现泛型。

UML 2规范通过了解UML领域中的模式和一般的软件模式在目标和范围上的区别，以相对广阔的视角优化了UML 1中提出的精确的（但不准确的）模式定义：

模板协作描述了设计模式的结构。UML模式相对于设计模式社区提出的模式是有限的。

一般来说，设计模式包含很多非结构化的方面，例如模式的启发式使用和使用开销。

这个定义是迈向正确方向的一步，但很明显，流行的方法学需要很长的时间来反映出模式概念的深层次含义。

□ 基于模式的设计与重构并不冲突，也不会被代替。人们容易认为代码中存在的模式总是有意识地使用的，实现它的人清楚自己采用了什么样的模式，以及为什么采用这样的模式。事实却未必如此。好的架构师根据自己的经验作出合理的新的设计，同时又是成熟的设计。他们使用模式，不管他们的设计思想是基于下意识的行为或者明确的文档化的。

重构相对于软件相当于园艺相对于花园：一个持续的添加、删除、重组和反思的过程。一系列好的重构[FBBOR99]不需要模式词汇？毕竟，重构不知不觉中提出了一些关于提升设计和过程的词汇。如果不直接采用，那么模式的作用是什么呢？它们只是大规模预先设计的一种方式？

开发人员一定要在模式和重构间做一个“非此即彼”选择的想法，是对模式和重构的本质的误解。很多模式描述了这种从问题到解决方案的转换，但这些相应的实现转变的机制只有在开发新软件的时候才被采用，重构则提供了针对现有代码的转换方法。有时候这种关系很清晰。例如，Null Object模式描述了这个模式的“缘由”和“如何”将其应用到新的设计中，而Introduce Null Object重构[FBBOR99]描述了如何将此应用于已有的代码中。重构的转换过程常常是专门的、机械的、着眼于代码的，在现代的开发环境中有很多实现了简单的自动化重构。

重构描述了一种转换，但常常并不描述最终的结构或者它的意图。重构的名称常常是动词短语，描述一个指令，然而模式的名称常常是名词短语，描述了结构化解决方案。例如，虽然Introduce Null Object在它的名称中明确包含了最终的设计结构，但大多数重构还是倾向于以编程语言特性命名的：Extract Method、Extract Class和Extract Interface各自作用于方法、类和接口，但其意向结构却没有指明。方法、类或接口存在的理由很多，但这些结构都不是自描述的。相反，Null Object模式的名称则反应出它的角色和意图。

从实际中可以看出，一个结构良好的设计往往是模式使用得很好的设计。如果抛开关于系统历程的故事——“于是我们有一个大的需要切割的类，于是……”——其他的开发者

除了讨论系统中的类和方法之外，还能讨论什么呢？模式和重构是相互补充的而不是相互冲突的[Ker04]。

其他的一些软件研究者和从业者们都报告了相类似的关于理解和使用模式的谬误[Vlis98b]，或许你可以把这个列表引用到你自己的资料中。然后，有那么多误解、误译和错误发生的事说明在流行的模式概念的理解和定义中存有错误。这些误解不可避免地导致了对模式的错误使用和实现。例如，正如0.1节提及的那个故事中所说的，在很多实际的项目中，一些开发者们把模式理解为一些特定的类图，或者患上了“如果你仅有的一个工具是把锤子，你往往会把一切问题都看成钉子”的毛病。

0.4 深入理解模式

为了避免或者说减少对模式的误解，我们需要对模式的概念进行回顾、讨论、澄清和修订。举例如下所示。

- 现有的定义很少能覆盖到模式的内在属性。这些定义一般都关注上下文-问题-解决方案这三要素，而且常常只关注于问题-解决方案的标题。一些模式属性则被忽略，比如上下文的角色、驱动力和效果，事实上，设计模式的方案描述类的角色而不是完整的类和模式泛型等。任何对模式的公正评价都应该会包含以上这些方面，而不是其中一部分。另外，正如0.3节中提出的我们自己的定义，其中只简要介绍了模式的属性，并且只着重说明了如果模式应用正确和成功，会带来哪些益处。如何正确地使用模式，如果不正确使用会发生什么，是关键之所在。遗憾的是，这种简化并不利于模式使用者和模式作者理解模式概念，不利于他们了解怎么写才能让模式的描述更加清晰、易理解和使用，也不利于他们了解如何在每天的工作中有效地应用模式。广告并不是用户指南。
- 模式定义中常有互相矛盾之处。例如，0.3节开头的模式定义中说软件模式的解决方案中引入了特定的软件组件。这个定义给人的感觉是模式实际上是由这些组件组成的。然而，在接下来的内容中，我们强调了模式引入了角色，而不是组件规范，这些角色由一定的应用组件担任。这些矛盾之处可能会被误解，特别是如果开发者们之后回想起的概要不够精确的情况。
- 模式的概念是各有各的风格的。例如，《面向模式的软件架构（卷1）》[POSA1]和“四人帮”的书[GoF95]中的模式概念就是面向工程的并以开发为核心的，而Christopher Alexander[Ale79]和Jim Coplien[Cope97]则提倡一种更全面的、跨学科的、着眼于交流方法的概念。遗憾的是，没有哪种方法能满足基于模式的软件开发的全部需求。例如，一些方法着眼于理解和实现特定模式的细节[POSA1]，而另一些则着眼于强调模式语言所提供的全貌[Cope97]。还有一些则另辟蹊径，例如反模式[BMMM98]。研究不同的模式风格，有利于从广度上而不仅仅是深度上理解模式的概念，可以避免遵循某个特定视角所造成的对模式的误解。
- 大多数的模式定义并不强调或未能明确强调模式之间存在的关系。理解和使用某个模式时只把它当成孤岛，而不考虑它和其他模式之间的关联，或许可以解决一些局部性的问题。

题，但这样的情况毕竟不多。对于关注于设计的模式而言，这意味着很难对全局产生影响：大规模的设计问题依然存在，小规模的设计问题层出不穷，开发人员对于架构没有一个清晰的认识，没有全方位的视角，要想全面、高效地使用单个模式也不是那么容易。研究模式之间的各种关系，不仅可以帮助人们更好地理解模式，而且可以帮助人们有效地、整体地将模式应用到软件项目中。模式复合、模式集合、模式序列以及本书之后介绍的模式语言提供了各种不同的模式间关联的使用方法，这些方法可用于衡量方案以及定位大型的、复杂的问题空间和应用领域。

□ 现在的模式概念的定义常常难以解决关于使用模式设计整个软件系统的问题。开发整个系统的视角和开发系统中的一部分或单一组件的视角是不同的。从系统化的视角看，很难独立地应对某一个需求或者某一个方面，因为其解决方案可能给其他必须实现的属性带来副作用。例如，设计既需要满足适应性和可变性的需求，又要满足与之相抵触的性能和可扩展性的需求。

单个模式很难解决这种冲突：根据定义，模式都定位在某个专门的问题之上，忽略了它和其他问题的解决方案之间的潜在关系或者影响。仔细学习这些问题，理解它们，了解如何处理这些问题，是有效使用模式的一部分。对“使用模式设计”提供概念上的支持是很有必要的，但现在很多的模式定义都只是在案例学习中含蓄地处理该问题，而没有明确地把它作为模式概念的重要组成部分。

□ 很多软件模式的定义都没有将模式的适用范围描述出来。模式不是万能药，也不是解决常说的软件危机的银弹[Bro86]。当考虑设计或者开发过程时，模式是对现有的软件开发技术和最佳实践的补充。准确理解模式可以为软件开发带来什么以及模式有哪些不足，对正确使用模式是必要的。确定模式面向的对象是谁是非常重要的（例如管理人员、开发者和测试人员），理解模式如何支持或阻止使用其他软件开发技术和技巧也是非常重要的。遗憾的是，很多模式概念的定义中都没有包括这些部分。

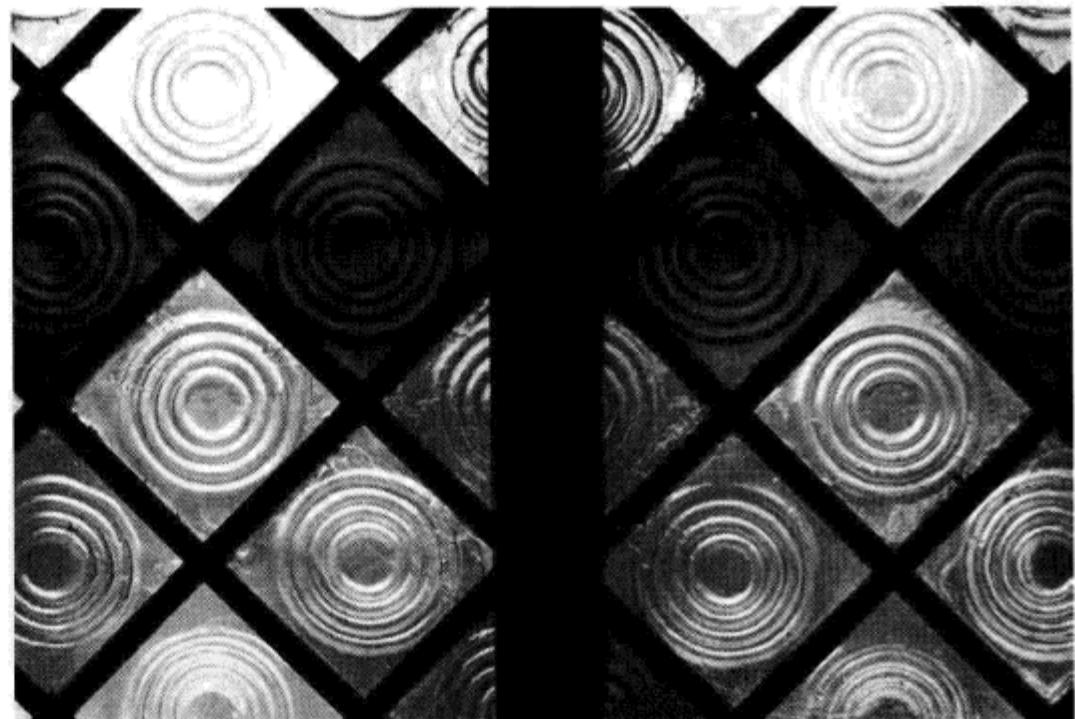
解决现有模式定义的不足的一个方法就是重写和修订，总结出一个新的更好的模式定义，一个可以替代其他所有定义的好定义。但这不是本书的目标，因为我们重新总结出一个新定义并不会很有用。我们更倾向于讨论正在进行的模式特性。

本书第一部分阐述了我们从使用、讲述、阅读、发现和编写模式中学习到了什么。首先尝试详细一致地描述每个模式是什么，特别是与软件架构相关的部分，并叙述每个模式概念的特色。基于这些描述，第二部分探究了模式之间的关系，从组织的角度说明了各个模式的领域。各个模式之间的关联知识是我们应用模式进行设计的概念基础。第三部分将模式与模式语言的概念相结合。在整本书中，我们将模式面向的对象、模式的适用性以及模式和其他软件技术的关联性这3个方面相结合，阐述了模式的概念。

为了深入模式概念，我们既说明了在模式使用上的经验，又以故事的方式讲述模式及模式语言，讲述它们出色的细节和广泛的应用。通过这个故事，希望你对模式的理解能比之前任何单纯的正式定义来得更充分。

第一部分

模式剖析



拉脱维亚某古老修道院的玻璃马赛克图案
版权归Mai Skou Nielssen所有

本书第一部分中，我们将重点放在诸多单个模式上，阐述并讨论自《面向模式的软件架构（卷1）》及模式相关书出版十年来我们积累的关于如何描述和应用模式的诸多见解。这些见解使得当前关于模式的讨论和定义更为完善，帮助我们从更深层次来理解模式，从而反映出模式领域的成熟。

在软件社区中，单个模式或许是应用最广泛、最为成功的模式概念。另外，大多数软件模式在相关文献中也是被“单独”描述的。为了更好地理解单个模式，我们从概念、实现和文档3个方面来进行剖析。

第1章从概念的角度探究了单个模式。我们首先解释了模式的基本结构，讨论了评判所提解

决方案优劣时需要考虑的关键因素，然后我们从驱动力、上下文和通用性上重点描述模式解决方案的作用及重要性，最后讲解了图表的使用、模式的命名以及模式的发展与成熟。

第2章探究了大多数单个设计模式概念与更为具体的特定模式实现类型之间的矛盾。为了达到这一目的，我们也将模式实现与通用实现、参考实现及范例实现进行了对比，考察了设计模式在框架分析方法、产品族及生产线分析方法、面向方面的软件开发、基于模型驱动的软件开发以及其他软件技术中的作用。

第3章讲解和讨论了描述模式的格式。具体的模式格式提供了连通模式理论与实践的桥梁，因此，关注模式的格式对于模式的开发者和使用者同等重要。

本书对于单个模式的讲解不是局限于我们自身对软件模式的理解。为确保本书讲解的内容更为完整、可信、与时俱进，我们也撷取了诸多广为认可的观点与认识，加以归纳概括，融会贯通，这些观点是从软件社区过去十多年里诸多应用模式的项目经验教训中提炼而成的。我们的目的是尽我们所能绘制出完整而条例清晰的模式图像，使读者能分清什么是单个模式，从而能够在软件项目中成功应用这些模式。单个模式的概念也是理解该模式其他方面的基础，包括模式互补、模式复合、模式故事、模式序列以及模式语言。

尽管我们的研究重点在于设计模式，但是本书对单个模式的关注点、见解、看法及表述是针对通用模式的，并不局限于特定类型的模式。

问题之解决方案及其他

哲学，如果它所回答的问题未能如我们所期望的那么多，至少它能够通过提问增加这个世界的趣味性，揭示出日常生活中最常见的事物表面所掩盖的神秘与奇妙。

——波特兰·罗素（1872—1970）

本章我们采用简单的问题-解决方案的形式，通过一步步的推理，逐渐扩展和优化，使其真正算得上是合格的模式描述。其间我们会研究模式所体现的各个方面，这也是对模式进行描述时应该包含的内容。这种渐进式的方法恰好揭示了模式编写过程迭代的特点。对反馈、自省、改进和提炼的使用是模式编写过程中正常的组成部分，而不是我们要展示模式描述的各个方面所臆想出来的。

1.1 问题之解决方案

如果被问到如何用一句话来描述模式，对模式有所了解的程序员可能会这样回答：

“模式是针对某个具体上下文中出现的问题之解决方案。”

乍一看似乎没什么问题。这固然算不上错误，但是却不够全面。看看我们现在常用的模式，确实可以这样说：它们提供了具体的、可行的、可调整的方案，用于解决在某种组织或者编程上下文中反复出现的软件开发的问题。这个说法也符合最流行的模式定义。

1.1.1 一个例子（1）

然而，模式绝不是简单的摘录！我们可以这样来描述一个模式。

上下文：客户端需要在聚合数据结构上执行某些操作。

问题：客户端可能需要获取或者更新集合中的多个元素。然而，如果对聚合的访问成本很高，那么用一个循环来挨个访问每个元素可能会带来严重的性能问题，比如往返时间（round-trip time）过长、阻塞性延迟过大或者上下文切换开销过高。如何才能对聚合对象进行高效的、无中断的批量访问呢？

解决方案：定义一个可在聚合对象上重复执行多次访问的方法。客户端在一次方法调用

中访问多个元素，而不是在一个循环中直接访问聚合中的每个元素。

这个描述显然是对某个特定上下文所产生的问题的解决方案。但是，术语的使用是不是让这个模式看上去很深奥？

1.1.2 重现与良好

对于某个问题，虽然你想出了解决方案，并据此作出了合理的设计，这样的设计并不能直接称为模式。它必须是反复出现的，这正是“模式”这个词本身的含义，所有的词典都会提到这一点。在软件领域，我们说到“模式”，通常指的都是“好的”模式，也就是说它应该是完整的、平衡的、描述清晰的，并且有助于整个设计的完整性和质量。因为大家都认同这种观点，所以我们通常会省掉“好的”这个修饰词，而直接说“模式”。

然而，如果说模式代表的是反复出现的设计故事——只是每次复述的时候都会有点变化——就不光是好故事了。有些实践也是反复出现的，但是它们或者不完整，或者不平衡，或者不足以让人们认可它的质量。它们只是很流行，但却算不上很好——虽说经常有人混淆，但“常用实践”（Common Practice）和“最佳实践”（Best Practice）可不是一回事[Gab96]。

有些模式是反复出现的，但是质量不高。在最初关于模式的描述[Ale79]中，Christopher Alexander将其称为“坏模式”（Bad Pattern），以区别于“好模式”。“坏模式”会带来很多问题，经常会导致设计越来越糟糕，使开发人员陷入无休止的问题循环中，不停地修复上次修复所引起的问题。

为了找到好的模式，并将其与坏模式或者算不上模式的模式区分开，我们首先得分清楚哪些是反复出现的，哪些只会出现一次，后者作为一种特定的设计，无论其多么精妙，都不能算得上是通用的模式。我们可以将重复出现的解决方案称为“候选模式”（Candidate Pattern）。

要让模式在我们的开发过程中发挥作用，只具有重现性显然是不够的。我们需要知道，使用它能带来好处，而不是用不用没什么区别甚至会带来坏处。当然，我们也不能因为某个模式不能为某些设计带来好处而草率地将一个“候选模式”贴上“无用”的标签，我们必须搞清楚真的是模式有问题，还是我们的用法有问题——总之，不能因为“误用”而说某个模式“无用”。为了合理评估这些模式的好坏，需要用一种使它们可见的形式呈现候选模式。必须清晰描述模式，使人们能够看清楚模式的各项素质——描述的不好可能会抹杀一个好的模式，而有倾向性的褒奖则会让人们高估了一个模式[Cool98]。

回过头来看看前面的例子，我们说它称得上是一个“候选模式”，但是以目前的表述形式，我们还看不到其他我们所关心的素质。在我们对模式的可行性和相应的文档表达能力有足够的信心之前，我们只能把它叫做“原型模式”（proto-pattern）。那么，原型模式的描述中到底还需要哪些必备因素才能成为好的模式呢？

1.2 流程和物件

对上例中的原型模式稍作分析即会发现，其解决方案部分显然是不够清晰的：尽管它描述了

创建一个解决方案的过程，但却没有告诉我们具体的解决方案是什么。诚然，一个模式可以有“成千上万种不同的”实现[Ale79]，其实现的多样性来自实际问题及其上下文的细节，而不是所建议的解决方案结构的模糊性。作为模式，它必须既能够比较通用地告诉读者如何创建一个解决方案，同时又必须能够避免模糊和偶尔的歧义。

上例中的原型模式有很多种可能的实现：我们可以说它们都是符合其解决方案流程的，但是符合该模式所要表达的精神的实现却很少。比如，我们可以在客户端定义一个方法，在这个方法中完成循环。这种设计只需对客户端代码进行简单的重构——抽取方法——即可，但是它却解决不了我们所强调的问题。

另一种实现是在聚合对象上定义一个方法，使其通过回调的方式将每个结果“推送”给客户端。然而，这种实现却并不可行，由于其昂贵的往返时间的积累，它引入的问题比解决的问题还多。字面上，它是符合前面所说的解决方案的，但是却不符合其精神。对于最初的问题，我们有很多已知的（更好的）解决方案“路径”：模式描述有必要对其所选择的路径作出更准确的描述。

一个好的模式，其解决方案部分要兼顾到流程和物件（thing）：“物件”通过“流程”创建[Ale79]。在大多数的软件模式中，“物件”代表一种特定的高层设计大纲或者有关代码细节的描述。这里所说的“流程”和软件开发生命周期中所指的“流程”不是一回事。通常，传统的软件开发流程的目标是开发整个系统，从启动阶段到部署阶段。它规模上更大，性质上更为笼统，使用也更加独立。

相反，一个单独的模式关注的则是如何解决某个特定的问题。它的规模可能没有多大，但它是确定的，具有良好定义的边界，并且其一般性来自于深度而非广度。在项目中，模式通常都要跟其他模式结合起来使用，而不是单独使用。一个模式可以认为是一个微流程——甚至是纳流程——这与其背后的软件开发生命周期流程是截然不同的。

一个例子（2）

原型模式最初的解决方案部分没有指明具体构建的“物件”。我们修改一下那段内容，于是得到了第二个版本。

定义一个在聚合上重复执行访问的方法。这个方法作为聚合对象接口的一部分。在声明这个接口的时候，它应该能够接收这个动作执行过程中所需的所有参数，比如通过数组或者集合，结果也通过类似的方式返回。客户端不是在循环中挨个直接访问聚合中的元素，而是在一次方法调用中访问多个元素。

这个版本比原先那个要清晰得多：我们知道解决这个问题的具体方案是什么，而不只是如何构建这个方案。

1.3 “好”的解决方案

既然新的描述更好地阐述了要构建的东西，我们也就更能够看清什么是问题，什么是解决方案，以及二者之间的区别。前面建议将所有对聚合对象的重复访问封装在一个方法里面，问题的

描述如下。

客户端可能需要检索或者更新某个集合中的多个元素。如果对聚合的访问成本很高，通过循环单独访问每个元素可能会带来严重的性能问题，比如往返时间、阻塞延迟以及上下文切换的开销。如何才能在一个聚合对象上高效地执行批量操作，而不至于发生中断呢？

这并不是说对聚合对象进行访问或者操作只有一种形式。它指的只是批量访问(bulk access)，其中检索和更新是最具代表性的例子。如果我们使用第二种方案来实现，所有的批量访问均是由一个方法来完成的。这样，我们就需要包含一些额外的参数来适应针对聚合对象的各种操作。这个方法是低聚合的，在其接口中需要支持所有针对聚合对象的输入、输出数据接口的超集，并且要包含某种选择器以指明要执行的操作。这样的方法冗长而且使用和维护都容易出错。

Singleton是一个众所周知的烂模式。其本意是：确保一个类只有一个实例，并且提供一个全局的访问点[GoF95]。它提出的解决方案是禁止通过类的构造函数创建实例。这个模式引入一个全局可访问的方法，它可以根据需要创建并返回该类的一个实例：如果对该Singleton实例的类级静态引用为null，则创建并返回一个新的实例，否则，直接返回已经存在的Singleton实例的引用。

这个方案的问题在于对对象使用了单一的全局访问点，这在现代编程中被认为是一种恶劣的习惯。不论是在企业级系统还是嵌入式系统中，人们都认为这种做法将使用单例类的代码与某个特定的上下文紧密地耦合在一起，将架构绑定到一系列不稳定的假设上面。所以，使用Singleton的时候必须考虑以下（但不限于）问题。

- 如何实现线程安全的Singleton？
- 如何定制Singleton的行为？
- 如何处理或者替换Singleton的实例？
- 如何对依赖Singleton的代码实施单元测试？

在[Vlis98b] [BuHe03] [POSA2]等文献中对这些问题的讨论远远超过了在GoF中原始模式的描述！在使用Singleton的系统中，各种妥协方案层出不穷。解构Singleton模式，发现它带来的问题比它解决的问题还要多——正如Kent Beck所说的：“怎样脱离全局变量来提供全局访问的能力呢？放弃吧。有这个时间还不如去看看自己的设计。”[Beck03]

在实践中，有用的模式不能只是提出问题的解决方案，而且这个方案必须是健壮的、优良的。解决方案必须通过验证才可能成为一个“好”模式。不是说有个不错的想法就能算作是一个合格的模式了，模式必须是反复验证过的——正是这些成功应用的记录才能证明它是一个模式。

GoF是这样说的：“模式提炼并提供了一种手段，使得有关设计的经验得以重用。”[GoF95] Brian Foote说得更绝：“模式完全无视独创性。”[PLoPD3]所以，新的想法必须首先在实践中证明其自身的价值——往往要反复证明多次——才能被称为模式。

一个例子（3）

下面这个版本对原型模式解决方案的描述具备了我们希望“好的”模式所应当具备的特性。

对于给定的操作，定义一个可以在聚合上重复执行操作的方法。将这个方法定义为聚合对象接口的一部分。它可以接收每次执行操作时所需要的参数——比如通过数组或者集合，并通过类似的方式返回结果。这样客户端就可以在一次方法调用中访问多个元素，而不是以循环的方式直接访问聚合中的每个元素。

每个方法将重复的动作转化为一个数据结构，这样就不需要在客户端循环执行这个动作，而只需在方法调用前后通过循环进行一些准备和后续操作。于是，对聚合对象的访问成本得到了有效降低，只需要一次或者几次较大型的访问即可完成。在分布式系统中，这样的“压缩”可以大大提高性能，减少网络错误，并节省宝贵的带宽。

当然，从复杂性的角度来看，我们也要付出一定的代价，访问之前的设置和之后对结果的处理都比原来更为复杂。这个方案还意味着需要中介数据结构来传递参数和接收返回结果。调用的网络、并发和其他准备工作的开销越大，这种方案就越合适。

这个版本更好地解决了原来的问题，避免了第二个版本中的问题。

1.4 驱动力：模式之心脏

从前面的讨论中可以看出，原型模式所揭示的问题解决起来并不是那么简单。这个问题不能孤立地去看待。有几个需求是必须考虑的，带宽就是其中之一。光从这个简单的问题描述里面是不可能驱动出这样的需求的。要获得良好的实现，这些问题必须得到合理的解决。

另一方面，将这样的需求与原型模式的问题描述合在一起才算比较明确，具体的解决方案才可以正确处理该问题。对于解决方案应该提供的其他属性也是一样的。它们会对解决方案空间产生影响，或者在解决问题的过程中需要考虑更多。给定问题的具体解决方案必然会受到需求、特性、约束和其他各方面的影响。模式社区将这些影响因素统称为驱动力（force），该术语取自 Christopher Alexander 早期的模式作品[Ale09]。

驱动力为我们揭示了为何模式所要解决的确实是个问题，为什么这个问题微妙而困难，为什么需要一个聪明的甚至是违反直觉的解决方案。驱动力还告诉我们为何我们得到的解决方案是这样的而不是那样的。最后，驱动力还帮助我们避免在不合适的情况下误用某个模式。

以 Singleton 为例，要消除对其理解的混淆，最初对驱动力的描述[GoF95]可能应该是这样的：“对于一个类来说，‘只能有一个实例’这样的属性应该是其所代表的类型所具备的属性，比如某个类实现了系统中某个硬件的有状态的 API，而不是某个应用中碰巧只用到类的一个实例这样的情形。”这样的话， Singleton 还会在模式空间占有一席之地，但不会像今天这样被广泛地误用——这正是 Singleton 造成大量问题的根本原因所在[Ada79]。

1.4.1 一个例子（4）

为了激励原型模式所建议的具体解决方案，我们为它的描述增加以下 4 项驱动力[Hearsay01]。

在并发环境中，由多个客户端共享（比如本地多线程共享或分布式共享）的聚合对象，最好将同步机制封装成为适合单个独立方法调用的形式，而非多个方法调用的形式——比如

在循环中反复调用。

在计算每次跨线程或进程访问的开销时，必须将相关的阻塞、同步和线程管理计算进来。与之类似，其他的针对每次调用的内务（housekeeping）操作（比如授权）会进一步降低系统的性能。

如果聚合对象相对于客户端是远程的，那么每次访问都会引入延迟和抖动（jitter），并消耗一定的网络带宽。

分布式系统可能会出现部分失败的情况，有时候服务器崩溃了，客户端却仍然活着。在迭代过程中进行远程访问，可以为每一次循环引入一个潜在的失败点，这可能会导致客户端面临只完成部分遍历的情形，比如不完整的状态快照，或者不完整的更新。

很容易看出第三个版本对解决方案的描述已经能够满足前三项要求，或者是直接的或者是间接的。尤其是，同步被隔离到一个单独的方法调用中，用一次调用取代多次调用降低了并发和远程访问的开销。第四项驱动力在第三个版本中并未得到满足，这是因为如果方法调用中出现问题，仍然可能造成部分更新或者产生不完整的查询结果。我们能注意到这一缺点，恰恰是因为我们认真地考虑了各种驱动力因素，并将其明确出来。我们也注意到，我们可以将其他后果更为明确地展示出来。

这就形成了第四个版本的解决方案。

对于给定的操作，定义一个可以在聚合对象上反复执行该操作的方法。

将这个方法定义为聚合对象接口的一部分。在声明的时候，让这个接口可以接收该操作所需要的所有参数，比如通过数组或者集合的方式。结果也通过类似的方式返回。这样客户端只需要一次方法调用就可以访问到多个元素，而不是在一个循环中直接单独访问聚合对象的多个元素。

每个方法将重复操作封装在数据结构里面而不是在客户端使用循环来完成，这样客户端只需要在方法调用前后通过循环操作来执行一些准备和后续活动。访问聚合对象的开销降低为一次访问或者几次“大型”访问。在分布式系统中，这样的“压缩”可以极大地提高性能，减少网络错误，节省宝贵的网络带宽。

对聚合对象的单次访问开销更大了，但是批量访问的总体开销却降低了。对于这些访问，可以在方法调用中做适当的同步。每次调用可以实现为事务性的，这样就只有成功和失败两种结果，而不会出现部分成功的情况。

这种做法在复杂性方面所作的平衡是要在调用之前和之后执行很多操作。同时，这种做法还需要更多的中介数据结构来传递参数和接收结果。网络、并发以及针对每次调用的开销越高，越值得采用这种方式。

现在所有的4个驱动力均得到了满足，我们的解决方案也更加健壮。尽管第二项和第三项驱动力是同时满足的，但是将二者作为独立的驱动力列出来还是非常重要的：它们之间是独立的、有区别的。网络带宽和上下文切换的开销不属于同一种资源：通信方面开销的变化可以独立于平

台的处理能力。这种变化强调单独列出所有驱动力的重要性，尽管可能乍一看这样做有些多余。

然而，我们必须知道不见得一定要通过一个解决方案解决所有的驱动力：有些驱动力是相互矛盾甚至是相互冲突的。要解决某项驱动力，有可能会影响到解决方案应对另一个驱动力的质量。比如，有效性往往要以灵活性为代价，反之亦然。这种情况下，解决方案必须平衡这些驱动力，或许不能完全解决某项驱动力，但是总体上必须使得所有的需求得到有效的满足。

1.4.2 dysfunctional、bad 还是 anti^①

我们讨论了半天驱动力，现在应该回过头来看看术语上的区别了。Christopher Alexander喜欢将经常出现的有问题的方法称为bad pattern [Ale79]，而我们则将其归为dysfunctional pattern。或许你还听说过anti-pattern这样的说法，这个听起来也挺合适的。

anti-pattern这个词听起来煞有其事，因为anti表示相反或者冲突的含义。但是它作为术语却不够清晰。我们不妨看看其他带anti的词语，它们表述的含义大有不同。比如抗生素（antibiotics）和解毒药（antidote），按照这种解读方法，anti-pattern应该是挽救“模式狂热者”（patternitis）的良药，使其不再为模式迷恋得忘乎所以。

是不是anti-pattern为模式提供了某种宇宙平衡（cosmic balance），可以相互抵消，就像物质和反物质（antimatter）那样？也许它们只是跟模式相反，就像在很多政治标语中“反这反那”一样；或者它可以像对防空装备（anti-aircraft device）那样将pattern搞下来？也许它可以帮助我们更加深入地理解模式，就像正餐之前的开胃餐（antipasto）？或者它们只是提供了一个令人沮丧的结论，即虎头蛇尾（anticlimax）？Jim Coplien[Cope95]做了如下澄清：

anti-pattern是一种以模式的格式记录无法工作或者具有破坏性的实践的文献。anti-pattern由Sam Adams、Andrew Koenig[Koe95]及笔者独立开创。很多anti-pattern在“驱动力”一节中记录了一些伪专家是如何文饰自己的糟糕设计的。[……]

anti-pattern并不像模式那样提供了应对驱动力的解决方案。作为教学工具，这是很危险的：好的教学方法应当构建学生能够记住的正确的例子而不是错误的例子。anti-pattern可能是一个不错的诊断工具，可以帮助我们理解系统存在的问题。

这个描述出现在第一本anti-pattern书[BMM98]将这个术语普及之前。在那本书里面，anti-pattern的理念有时是模糊的：有时候anti-pattern表示一种反复出现的错误情况——一种错误的解决方案——往往是由对设计实践的误用造成的。有些时候，anti-pattern则包含了应对那些问题的解决方案。

前一种情况接近医学诊断上的隐喻[Par94][EGKM+01][Mar02a] [Mar02b][Mar03]，根据其症状和根本原因确定问题的状况。而后一种方法也基本符合这个隐喻，因为它提供了一种缓解方法和预断^②，但是它同时也符合基本的模式概念：失败的解决方案是问题，疗法则是真正的解决方案。在后一种形式中，看上去没有什么特别之处——尽管叫做“anti-pattern”，其实也是问题加解

^① 三种命名分别可以译为“伪模式”、“坏模式”和“反模式”，此处为了保持与正文呼应保留原文不译。——译者注

^② Prognosis，医学术语，预测疾病可能的原因。——译者注

解决方案。这种挥之不去的模糊性使得我们不得不寻求一个新的更为清晰的名称来指代反复出现的错误方法。

Jim Coplien所提供的定义看起来是最有用并且最严谨的。因为有些驱动力未得到恰当的处理而打上“anti”的标签有点说不清道不明，并且不能正确反映这种反复出现的问题的本质。相反，“bad pattern”则显得过于直白。作为一种描述手段，将设计划分为“好的”和“坏的”未免过于简单，甚至有好为人师之嫌。这种价值判断可能对个人来说是有意义的，但是不利于与别人进行平等开放的交流。

如果我们希望通过模式来进行交流和理解，有必要在选词的时候注意其中立性。因此，我们希望将成功地解决问题的各种驱动力的解决方案定义为“完整的”(whole)，否则定义为“不完整的”(dysfunction) [Hen03a]。“完整”意味着平衡和全面而不是半成品，而“不完整”意味着设计在某些方面存在功能缺失，或者影响到整体的稳定性。

1.5 上下文

既然驱动力是问题描述的补充，并且问题的解决方案也会相应调整，那么，接下来要考虑的就是原型模式中的上下文(context)。

客户端需要在某个聚合数据结构上执行操作。

这个描述相当笼统。这种情况下发生的问题可能有很多，绝不只是前面原型模式所描述的那一种。实际上，就算是把它去掉也没啥损失，因为这个描述实在是太宽泛了。

然而，在一个模式中上下文所扮演的角色是非常重要的，因为它定义了模式所适用的情形。对适用情形描述得越精确，开发人员用错模式的几率就越低。理想情况下，上下文所描述的情形应该是恰好能够导致模式所要解决的问题。

相反，如果上下文描述得越宽泛，其价值就越低。Bridge模式^①就是这么一个例子，在[GoF95]里面的原始描述是：“将抽象与其实现相分离，以便二者可以独立地变化”。通过这种间接性带来的好处的设计比比皆是，比如使“实现绑定”(implementation binding)更灵活，编写强异常安全保证(strong exception-safety guarantee)的C++代码[BuHe03]。

Bridge所解决的各种可能的问题却都有各自的上下文，比如“我们要创建一个组件，其实现必须可以在运行时替换”或者“我们要实现一个C++应用，并且必须是异常安全的(exception-safe)”。正是因为存在这些区别，相应地，问题的驱动力以及具体的解决方案也会有所不同。要知道，模式往往被简单地认为是某个特定上下文下对某个问题的解决方案——本章开头的(不完整)定义就是这样的。所以，如果上下文描述得不精确，模式就成了各说各话，而彼此又不兼容[BuHe03]。

^① GoF对于Bridge模式上下文的描述主要是通过一例子来表现出来的，所以导致读者很难把握其真正的上下文是什么样的。其上下文可以笼统地表述如下：在某些系统中，一方面抽象概念的实现会发生变化(对应于原书中的IconWindow和TransientWindow)，这是显然的，这正是抽象的本义所在；另一方面，抽象概念如何实现也会发生变化(对应于原书中的XWindowImpl和PMWindowImpl)。更详细的解释可以参见译者的博客。——译者注

1.5.1 一个例子 (5)

为了使原型模式的上下文更清晰，这里我们把问题描述的前两句话和最初的上下文合并一下，因为在问题描述中也包含了上下文的信息，然后向上下文中添加有关设计活动的信息以及应用的使用情况。

在一个分布式系统中，聚合数据结构的客户端可能需要在聚合上执行大量操作。比如，客户端可能需要取回集合中所有具有某个特定属性的元素。如果对聚合的访问成本很高，比如它相对于客户端来说是远程的，在一个循环中独立地访问每个元素可能会引入严重的性能问题。

现在这个上下文就清晰得多了，通过它我们能够明白问题到底出现在什么情形下。上下文还告诉我们，原型模式可以在什么地方使用，当然我们也就知道了它不能在什么地方使用。理解了模式的上下文，开发人员就可以决定何时不引入该设计。比如，如果聚合对象相对于客户端不是远程的，我们可以考虑选择别的实现方式，比如在我们第一次修改原始解决方案之前的某个方案（参见1.2节）。这种情况下，为了最小化操作开销而采用原型模式目前的解决方案可以说是“高射炮打蚊子”。

将问题中的信息挪到上下文中需要重新叙述问题描述其余的部分。

如果访问的成本很高并且有可能出现故障时，怎样才能在聚合对象上高效地执行批量访问而不出现中断呢？

将上下文信息从问题描述中移走，这样处理还带来一个额外的好处，“真正的”问题显得更加明显。问题越发简明扼要，我们就越容易理解它确实是个问题。

1.5.2 上下文的一般性

降低上下文的一般性可以让我们的描述更为精确，但是其缺点也很明显。因为我们的原型模式中所阐述的问题在其他情形中也可能发生，同样的解决方案对于那样的情形也许同样有效。例如，考虑下面的上下文。

在应用中使用自定义的内存数据库，其中涉及复杂的非关系型对象，针对单个对象，我们可以进行独立的基于键值的搜索，但是如果这个操作要反复执行，则可能会形成瓶颈。然而，对于这种数据库来说，势必会存在很多情况需要在符合某个搜索条件的全部对象上执行已定义的标准操作。

既然可能有多种情形，问题就来了：如何才能保证上下文的完整性呢？上下文描述得太笼统了，可能包含了多种可能的问题情况，这可能会造成对模式的误用。一方面，在上下文中包含太多细节可能会给阅读带来困难，因为前面的描述过于冗长。另一方面，如果上下文有过多限制，开发人员可能就不知道其他可以应用模式的情形。他们只能按照模式中所写的去使用模式，而意

识不到其他的可能性。

解决这个问题的一个方法是在上下文中先给出已知的可以使用模式的应用场景，如果发现了新的场景，及时更新。这就像GoF所采用的格式[GoF95]里面的Applicability（适用性）部分，其中GoF列出了所有可以使用该模式和已经使用该模式的特定场景。

解决这个问题的另一个方法是采用《面向模式的软件架构（卷2）》[POSA2]所使用的方式。上下文描述聚焦于一个主题：并发和网络。每个模式的上下文仅涉及与这个主题相关的内容，模式在其他情形中的应用在一个单独的章节里面介绍。

如果模式集合围绕着一个共同的主题，那么限制上下文范围的方式是可以的：每个模式的上下文内容精简了，读者可以更容易地识别出模式在某个领域内的适用性。其他有关的情形尽管有所提及但并不做重点描述。这样的描述更加专注于特定的上下文，而不是追求面面俱到。

然而，这两种方法都没有真正解决模式上下文完整性的问题：极有可能忽略了某种适用的情形。结果就是，在《面向模式的软件架构（卷1）》和本节的厚型模式例子中这种笼统的上下文、在《面向模式的软件架构（卷2）》中的聚焦于某一方面的上下文，以及GoF的模式中关于适用性的章节，哪一个看起来都不够完美。另一方面，后两种方式看起来似乎比第一种方式要更可行一些。它至少能够指导模式在某几个特定情形中的应用，总比列出所有可能的应用，但却样样皆有一无所用要好。

1.5.3 独立上下文

一种完全不同的处理方式是不把上下文看做单个模式的一部分。我们已经看到对于例子中的原型模式，可能有多个上下文，但却只有一个问题描述和一个解决方案描述。也许上下文只有在描述模式语言的时候才用得到，我们在模式语言中使用上下文将相关的模式集成在一起。换句话说，上下文定义了一个模式网络（即模式语言），而对于网络上的节点——单独的模式，它们是独立于这个网络的，所以它们跟上下文是没有关系的。这样上下文完整性问题就不存在了。如果在多个模式语言中用到了同一个模式，那么只要每个语言为该模式提供自己的上下文就可以了。

尽管这个方法看起来很整洁，但是它却会带来另一个问题。如果模式中不包含问题出现的情形，又怎么能说模式如实地反映了问题及其范围的本质呢？问题与上下文不是相互独立的，问题不能简单地插入到任意的上下文中。上下文不是简单地起到胶水的作用，在上下文和问题之间存在内在的关联而不是简单的依附关系。因此，问题的特征隐含上下文的特征。

从某种程度上说，上下文是个视角的问题。从单个模式的角度来看，要知道它可以在什么样的条件下成功应用是非常重要的。到底如何与其他模式集成也有价值，但是重要性却低得多。因此，上下文很可能是模式的一部分，也就是说上下文完整性问题仍然是存在的。

从模式语言的角度来看，我们有必要知道语言中的模式如何联系在一起。没有必要了解是不是也有其他的模式语言也包含某个特定的模式。因此，上下文只需要定义某个特定的模式语言，而不需要恰当描述某个特定的模式以及这个模式所适用的各种条件。这时上下文完整性的问题似乎并不那么严重了。然而，这种简化所带来的问题是相关模式的适用范围受到了限制。这种一般性和具体性之间的权衡，在我们对模式概念的探索中是个永恒的主题。

本章关注的是单个模式，而不是模式语言，所以我们从模式的视角来看待这个问题：某些或者所有的上下文属于模式的一部分。大多数软件模式都是这样的，这些模式各自独立，还没有形成模式语言。我们在后续内容中还会讨论上下文的问题，但是我们的重点将不再限制在单个模式上。

1.5.4 一个例子（6）

现在，我们可以修正一个原型模式的上下文，使其可以覆盖模式的适用情形和它关心的状况，本例中即包含多线程和（或）分布式环境下的系统。

在分布式或者并发系统中，客户端可能需要在聚合数据结构上执行大量操作。比如，客户端可能需要取得所有具有某个属性的元素的集合。如果对聚合的访问成本很高——比如，聚合相对于客户端是远程的或者是被多线程共享的，在一个循环中单独访问每个元素可能会引入严重的性能问题，比如往返时间、阻塞延迟和上下文切换开销。

不考虑“上下文完整性问题”，我们的原型模式再次得到改进。开发人员可以更好地判断模式适合于哪些情形，尽管新的上下文不能穷举所有可能的情形。

1.6 一般性

在我们对上下文、问题描述和解决方案部分做了修正之后，上例中的原型模式比最开始看起来更有价值了。但是，还没完。尽管在解决方案部分我们非常明确地提到了对象和方法，但这并非我们的本意。尤其是，这样的解决方案是否会过于限制模式的使用呢？解决方案和问题是否在一个层次上呢？

在对象结构中，特定的问题需要使用面向对象的工具和概念去解决。然而，如果问题的描述更为笼统，我们就不应该直接将其束缚在面向对象框架下。反过来，如果我们的模式要处理问题中某个编程语言的细节，其解决方案也必须处于那个层次上——如果太笼统了，就会显得模糊而不精确。般来说，模式对具体实现技术的依赖既不能太强也不能太弱，而是要恰到好处地表达出模式自身的精髓。比如，在原型模式中，我们面临的问题可能出现在面向对象代码中，也可能出现在面向过程的代码中。我们的解决方案可能以对象方法的形式表达，也可能以普通函数的方式表达，或者是有线协议^①（wire-level protocol），甚至是海外包裹快递中也会采用类似的方式。这些都没有违反解决方案的核心原则^②。如果模式所解决的问题要求某个特定的技术或者范型，我们应该明确指出来。否则，我们应该有意识地考虑这样做是否对目标读者有好处。如果没有好处，那么我们的模式就不应该依赖于这样的技术或者范型，因为这样会限制模式的适用性和可选的实现。

^① 也有人翻译成“线级协议”或者“线路层协议”。它表示一个底层的接口，通常指直接位于物理层之上进行传输或者交互的编程接口。常见的协议包括CORBA、DCOM、RMI、SOAP等。——译者注

^② 其实，有些模式看上去依赖于特定的实现范型（implementation paradigm），比如对象技术，但是实际上却不尽然：以Proxy为例，即使我们不使用继承，模式的精髓损失也不多，而Strategy模式在C语言里面也可以用函数指针的方式实现，而不见得一定要使用对象层次上的多态。

一个例子 (7)

假设我们的目标是提供面向对象的解决方案。如果使用对象技术是一项需求，我们可以为模式增加一个相关的驱动力 (force)；如果在我们的场景中使用对象技术是一个先决条件，也可以把它放到上下文里面。这里我们采用后一种方案，对上下文的第一句和最后一句做一些修正。

在一个分布式或者并发的对象系统中，聚合对象的客户端可能需要在其上执行大量操作。比如，客户端可能需要取回符合某个特定属性的元素集合。如果对聚合的访问成本很高，因为它相对于客户端是远程的或者是在多个线程中共享的，通过循环的方式单独地访问每个元素（不论是通过索引、键值还是Iterator）都可能导致严重的性能问题，比如往返时间、阻塞延迟或者上下文切换。

对于指明在解决问题时要使用对象技术，这样的修改可能并不是最好的方式，但是至少措辞上更严谨更明确了。这也提示了另一个问题：聚合对象的接口。我们来看看当前解决方案部分的第二句话。

每个方法被定义为聚合对象接口的一部分。

对于聚合对象，这意味着什么呢？是不是说不论它使用了什么接口，都必须包含新方法的声明呢？还是说可能采用其他的方式？比如，定义独立的Explicit Interface，它仅提供迭代功能，并且它可以用在其他对象类型上通过统一的方式来处理不同的聚合类型。

那么聚合的实现呢？到目前为止，原型模式还没有讨论到这一点。聚合类可能直接实现了这个新方法。我们也可以使用Object Adapter机制来实现，这样就不需要对聚合机制做任何修改了。Object Adapter可以帮助我们将分布式友好的模型与进程内实现适配起来。

笼统的问题解决方案不必引入特定的类、组件或者子系统，而是引入角色 (role) [RWL96][Rie98][RG98][RBGM00]，系统中具体的组件扮演相应的角色以解决原来的问题。角色定义了组件在系统中的职责以及与其他角色之间的交互。

虽然推荐将不同的角色实现区分开，但是并不是一定要将其单独封装到不同的组件中。一个组件也可以同时扮演多个角色。我们可以为某个角色创建新的组件，也可以将角色指定给某个已有的组件。如果有必要，开发人员可以为组件引入一些特定于某个角色的接口，这样客户端就只能看到它们所需要的角色了。

对于将模式无缝地集成到已有的软件架构中，角色是一个非常关键的概念，对于在大规模设计中组合使用多个模式也是如此。根据上述讨论，我们对原型模式再进行一次修正，下面是新的版本。

对于给定的操作，定义一个可在聚合上反复执行这个操作的方法。

每个方法定义为聚合对象的接口的一部分，或者直接是为聚合类型输出的Explicit Interface的一部分，也可以作为一种混入 (mix-in) Explicit Interface，仅提供调用重复动作的能力。在这个方法声明中，接受执行这个动作所需要的所有参数，比如通过数组或者集合，

结果的返回也采用类似的方式。这样访问聚合元素的时候就不是直接在一个循环中单独访问，而是由客户端调用这个方法在一次调用中访问多个元素。这个方法可以由聚合对象的基础类来直接实现，也可以通过Object Adapter方式间接实现，这样就不需要对聚合对象的类作出修改了。

[……]

现在开发人员可以将访问聚合的方法定义为已有接口的一部分，也可以单独定义一个接口。类似地，既可以在已有的类里面实现这个方法，也可以单独定义一个类来实现。在原型模式中，角色是稳定的，实现却可以变化。

对于一个模式而言，解决方案部分应该明确指出哪些角色必须单独实现为一个组件。而对于其他的角色，我们不应该在模式描述中做这样的限制，否则就会影响到解决方案的适用性。

解决方案的最终版本描述了如何构建一个解决最初问题的结构。通过引入角色，我们有了多种实现可供选择。角色对于模式的广泛适用性是非常关键的——这比适用严格类的方式要好得多。角色也使得我们可以更容易地将模式的核心思想适配到具体的应用中。同时，我们也避免了不必要的复杂性和间接层次，这使得我们的模式实现更加简单、灵活而有效。

1.7 一图胜（逊）千言

既然解决方案的最终版本提供了更好的品质，我们可以花点时间来更加笼统地讨论一下模式的解决方案。解决方案是对具体实现的抽象，它描述了设计的结构，其中包括各个角色及其关系。此外，还包括该结构中“发生的”行为。

对于以代码为中心的模式，其解决方案通常都提供一个代码片段，其中编程语言的指定元素都以特定的方式组织，并与该代码的行为合在一起展示。而对于组织模式来说，其解决方案部分引入一个特定的组织结构、该结构中的角色、这些角色的职责以及角色之间的沟通。最笼统地说，模式定义了一系列元素的空间配置（spatial configuration），以揭示或者促成某种特定的活动状态。

因此，很多模式的解决方案除了有文字描述之外，还经常配一些图表来揭示或者概括元素的配置、在该配置下的交互以及（如果有）随着时间的演化。这些图表有助于表现模式的本质和某些细节。它们从宏观上对模式提供了一种图形化的描述方式。一图胜过一千（零二十四）^①字。

图示能力与模式

有人建议将“能够通过图示方式表达出来”作为模式的一个基本属性。然而，有些软件概念，比如针对某个特定系统作出的特定设计或实现，尽管可以通过图示的方式表达，但却不能归为模式。普通的软件概念和模式之间还有相当大的差距。如果说“一个概念不能以图示的方式表达，便不能称为模式”[Ale79]，那么反过来也可以说“即使一个概念可以用图示的方式表达，也未必

^① Trygve Reenskaug, ROOTS 2001会议，挪威。

就能称为模式”。图示能力对模式来说是必要条件，却不是充分条件。

图表能够帮助人们理解模式所言不假，但是要说图示能力是定义模式的关键属性却要出言谨慎。人类的想象力是极其丰富的，对于任何抽象的概念，人们总能够通过某种形式的图表表达出来。诚然，对于一个特别抽象的概念来说，其图表可能做不到对所有的人都能够有效地传递其含义，但是，图示能力不仅在模式领域内做不到这种有效性，同时也很难将模式与其他通过构想、体验、发明等方式得到的东西区分开。

可能更有效的区分是强调问题中的概念必须是设计出来的作品，而非天然出现的东西，图表必须是基于设计的空间配置（*spatial configuration*）。当然，这种方式并不能将模式与其他设计概念完全区分开来，但是考虑其图示能力仍然是一种比较好的区分手段。

例如，我们固然可以用图表的方式来展现“区分关注点”和“封装”等基本设计原则，但是这样的图表往往是笼统的、抽象的，因此对于沟通和讨论没有太大的价值。只有跟具体的设计联系起来，这些原则才变得生动活泼，而且对于开发人员来说也更加具有实质的指导作用。比如，我们的原型模式将聚合对象中元素的迭代访问封装到一个数据结构中，在访问这些元素的时候直接返回这个数据结构。

[……]我们不是直接在循环中单独访问聚合元素，而是客户端通过一个方法调用访问多个元素。[……]

用图表来展示这种具体情形下的封装是非常容易理解的，也有助于沟通和讨论——因此，通过图表来展示笼统的封装概念不如在具体设计的上下文中展示更有价值。

实际上，即使是展示具体的设计，也是充满陷阱的工作。“空间”在非物理的不可触摸的软件领域更多的是一种隐喻，而非实指[Hen03b]。这是一种微妙但非常重要的区别，当人们将来自于实物工程（*physical engineering*）的想法和架构原则应用于非实物领域（比如软件开发）时必须牢记在心。除了软“件”（*software artifact*）中的用户界面，软件设计中从构造可见物中而来的空间的概念是一种自由选择而非指定的。所以，不同的选择对应于不同的空间概念。因此，通过简单的装点或者大量使用抽象使一个低劣的设计看上去还不错，或者由于未能对可视化的细节和选择给予足够的关注而使得一个本来还不错的设计看上去差得多，这都是有可能的。

所以，在支持为模式配备图表的同时，我们也应该明白图示只是一种口味、格式和表现，而不是什么深刻的东西。就此，我们的原型模式配备如图1-1所示。

这里我们有意没有采用流行的建模符号，比如UML[BRJ98]。目的是为了简化：正式的（UML）结构图经常被误以为是给定问题的唯一正确的解决方案。然而，对于模式来说，当然不是这么一回事。模式是笼统的，它定义的是角色而不是类，因此存在很多种不同的实现。另一方面，正式的结构图往往只能描述一种特定的情况或者路径，而实际可能的配置和角色间的交互以及演进路径却远不止一种。

另外，图表越是正式，人们越倾向于按照图表中的样子实现相关的模式，因为看起来它就是一个标准的参考解决方案，可以大规模复用而无需针对适用的情形做任何修改。第0章中的Observer故事很好地解释了这个误区。相反，模式图表越是不那么正式，开发人员就越需要思考

如何在自己的系统中实现这个模式，或者在他们所使用的标记法体系中如何表达。思考是设计的本质，设计不是死记硬背可以做到的。

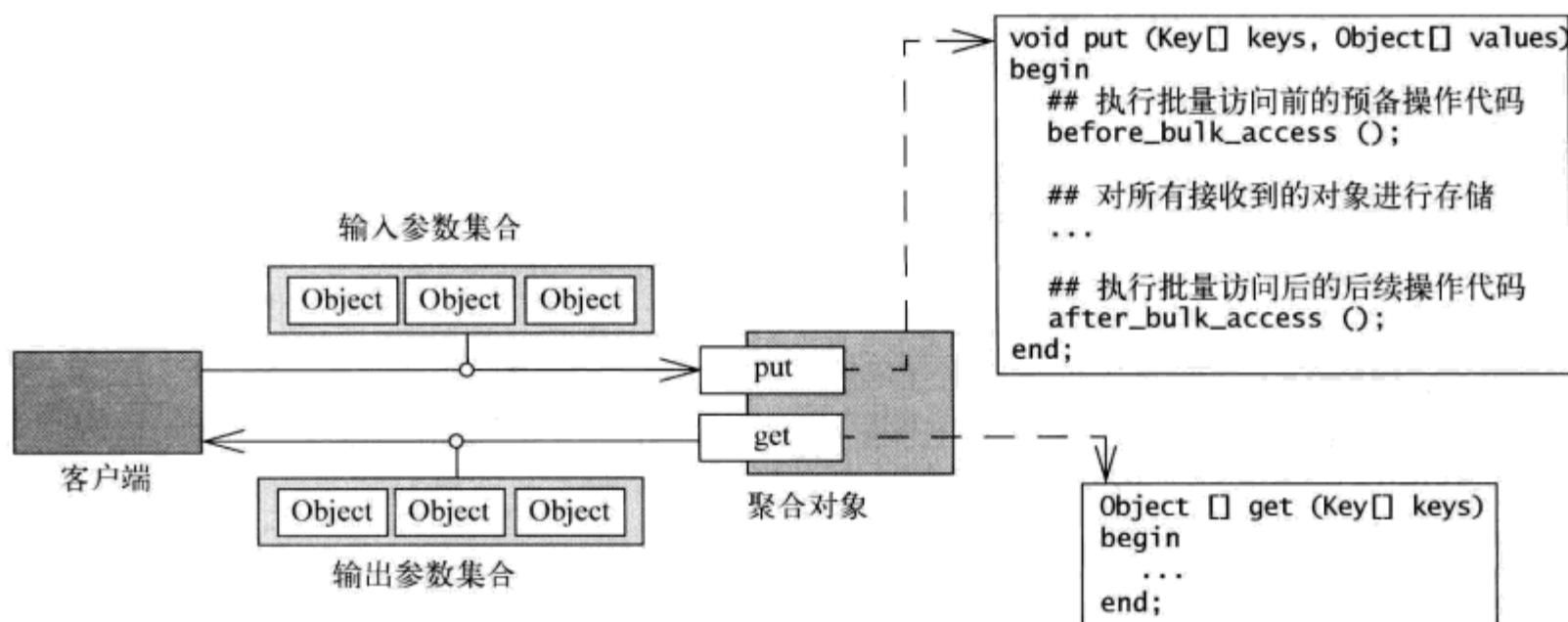


图 1-1

如果开发人员从角色的角度进行思考，并且认为图表只是展示了解决某个特定问题的众多可能的具体方案之一，选择什么样的标记法可能就不那么重要了。如果从角色的角度进行思考，可能遵从大家一致的标记法反而会好一些，因为人们对此比较熟悉。在《面向模式的软件架构》系列中，我们假定读者对模式的基本概念比较熟悉，知道模式的参与者是角色，我们在《面向模式的软件架构（卷1）》[POSA1]中也对此进行了介绍。

因此，我们在前三卷[POSA1][POSA2][POSA3]中使用了OMT和UML图表，而没有使用非正式的图表。但是并不是所有的地方都是这样的。比如，在《面向模式的软件架构（卷4）》[POSA4]中我们就使用了一个不那么正式的、现成的标记法来演示模式结构。我们在演示“模式进行时”中也使用了这种“Bizarro”^①标记法。

1.8 模式命名

模式的描述需要能够提供一个清晰而有效的解决方案来应对反复出现的问题，就像我们前面所说的，这种图文并茂的方式更能有助于模式的成功使用。但是如果我们要在设计和实现中有效使用这些模式，必须能够快速、无歧义地识别和引用每个独立的模式。而且，即时我们手头没有模式的描述或者结构图，我们也需要能够讨论这些模式以及它们会出现在哪些设计中。换句话说，我们必须能够记住这些模式，否则它就不能成为设计词汇的一部分。如果不能够记住模式，不论它的质量有多好，它也不可能在实践中反复使用。

^① 这里的小方块不禁让人想起Bizarro立方世界Htrae（Bizarro的故乡，Bizarro是DC漫画出版社超人系列中的一个角色）。Bizarro的逻辑也有些颠倒和不同，所以Bizarro建模跟UML有很大区别：在UML中使用不同的图来对系统不同的方面进行建模，而Bizarro标识法通过一个图来展示多个方面；UML更加形式化，Bizarro更加随意；UML是一种流行的技术，而Bizarro标识法对于职业发展没什么用处（除了在Htrae上以外）。

所以说，每个模式必须有一个响亮的名字。这个名字必须能够让人产生共鸣[MD97]。理想情况下，如果谁提起了某个模式的名字，其他熟悉这个模式的人也应该可以通过这个名字回忆起相关的内容。这种境界并不是那么容易达到的。过于取巧的名字可能只对一小撮人有意义，比如模式编写者，而其他人则难解其意。模式的内容是沟通的基础，词不达意的名字对沟通会产生很大的影响。最难忘的名字应该能够让人在头脑中显现出一幅画面，恰当地阐述其解决方案的本质[MD97]。

1.8.1 模式命名的语法分类

从语法上讲，常用的模式命名方式有两种，如下所示。

- 名词短语用于描述模式的结果，比如Active Object和Command Processor。名词短语的命名通常描述了解决方案的结构，有时也直接列举出解决方案中关键的角色，比如Model-View-Controller和Forwarder-Receiver。
- 动词短语通常是命令式结构，指示如何达到模式所期望的解决方案状态。比如，Engage Customers组织模式和Involve Everyone组织变革模式都是动词短语的例子。同样，Don't Flip the Bozo Bit作为软件团队领导力和人际互动的精彩模式，也是这样的例子。

名词短语比动词短语更通用一些。名词短语重点描述解决方案的结构，并且更加适合用在句子里面来表示一个模式。我们说模式“既是一个过程，也是一个事物”，名词短语强调的是“事物”，动词短语强调的是“过程”。

1.8.2 字面命名还是隐喻

命名的风格也是非常重要的。要让人能够在脑海中呈现出模式的画面，实际上有两种截然不同的方式——按字面命名和隐喻。

- 字面命名就是直接描述模式做了什么。比如Explicit Interface和Iterator属于字面命名。
- 隐喻命名则是将模式和其他概念联系起来，比如和我们的日常生活联系起来。Visitor、Observer、Broker都是隐喻的例子。

由于软件开发这件事情本身就具有抽象性，所以很多术语都是基于隐喻的，如套接字(socket)、文件(file)、窗口(window)等，所以从不同的角度来看，有些名字既可以说是按字面命名，也可以说是使用了隐喻。很多名字甚至是隐喻和字面命名的组合，比如Resource Lifecycle Manager。

哪种命名方式最合适通常跟个人喜好有很大关系。比如，按照字面命名，我们的原型模式可以命名为Repeating Method，按照隐喻的名词短语方式，可以命名为Boxcar Method。这两个名字都可以说是抓住了模式解决方案的精髓。我们比较喜欢的是另一个名字——Batch Method，这个名字兼有两种命名风格。要是能找到按照字面命名的动词短语，其实也不错，但是往往很难找到一个短小精悍的动词短语来清楚表达模式的精髓。比如，Express Each Form of Repeated Access as a Loop-Encapsulating Method能够抓住原型模式的基本含义，但是太过冗长根本就不适合做模式的名字。对于一个交流手段来说，方便与否是非常重要的。

1.9 模式是循序渐进的

经过讨论，我们终于得到了原型模式的最终描述，如下所示。

Batch Method

在分布式或者并发的对象系统中，聚合对象的客户端可能需要在聚合上执行大量操作。比如，客户端可能需要取回所有具有某些特定属性的元素。如果对聚合的访问成本非常高，因为它相对于客户端是远程的或者是由多个线程共享的，在一个循环中单独访问每个元素（比如通过索引、键值或者Iterator）都可能引入严重的性能问题，比如往返时间、阻塞延迟或者上下文切换开销。

如果访问成本很高并且可能失败，怎样才能高效地、不中断地在聚合对象上执行批量访问呢？

在解决这个问题的时候，必须考虑4项驱动力。

- 聚合对象是由并发环境中的多个客户端共享的，可能是在本地的多线程环境，也可能是跨网络的分布式环境，我们可以在一次方法调用中对其进行同步封装，但是对于在循环中重复的调用却不合适。
- 阻塞、同步和线程管理的开销必须在每次跨线程或者跨进程的访问中考虑进去。其他的针对每次调用的操作（比如授权）会进一步降低系统的性能。
- 如果聚合对象相对于客户端是远程的，那么每次访问还会带来延迟、抖动，并占用网络带宽。
- 分布式系统可能会出现部分失败的情况，这时候客户端可能还活着，但是服务器却已经崩溃了。在迭代过程中进行远程访问，可以为每次循环引入一个潜在的失败点，这可能会导致客户端面临只完成部分遍历的情形，比如不完整的状态快照或者不完整的更新。

对于给定的操作，定义一个可在聚合上反复执行该操作的方法。

每个Batch Method都是聚合对象的接口的一部分，或者是作为聚合类型的Explicit Interface暴露出来，或者是作为一个更窄的混入（mix-in）Explicit Interface——仅仅定义调用重复操作的方法。Batch Method被声明为可以接收每次执行操作所需要的所有参数，比如通过数组或者集合，结果的返回也采用类似的方式。这样我们就不需要在循环中直接挨个访问聚合中的元素，客户端直接调用Batch Method就可以在一次调用中访问多个元素。Batch Method可能由聚合对象的基础类直接实现，也可以通过Object Adapter方式实现，这样就不需要对聚合对象的类做任何修改了。

通过Batch Method，我们将重复的操作转化为一个数据结构，这样我们就不需要在客户端进行循环，而只需要在开始调用方法之前和结束之后通过循环做一些准备和后续工作。于是，对聚合的访问成本降低为一次访问或者几次“较大型”访问。在分布式系统中，这种“压缩”机制可以大幅度提高性能，并减少网络出错的概率，节省宝贵的带宽。

使用Batch Method会提高单次访问的成本，但是批量访问的总体成本还是降低了。这样

的访问也可以在方法调用中配合使用适当的同步技术。每次调用可以是事务性的，即或者完全成功或者完全失败，不存在部分失败的情况。

使用Batch Method的代价是在准备和后续阶段要做更多的工作，并且需要引入中间数据结构来传递参数和接收返回结果。网络、并发和其他针对每次调用的开销越大，使用Batch Method就越合适。

现在的版本与开始的版本迥然不同：我们可以说它已经具有了一个好的模式描述所需的所有特性。名字有渲染力，上下文具体而简洁，问题描述干净利落，各种驱动力描述全面而有效，解决方案可以恰当地解决提出的问题及其驱动力。

最新的解决方案描述中包含了一个基于角色的结构，其中既考虑了静态的角色，也考虑了动态的角色，同时也包括了创建这个结构的过程。我们也可以画几个示意图。解决方案既可以说是具体的，也可以说是抽象的：它的实现方式可能多种多样，但是其精髓却是不变的，这使得我们可以从设计中识别出该模式。最初的描述和最终的描述都可以满足“模式是一个针对某个特定上下文中出现的问题之解决方案”这样的定义，但是二者的差距却不言而喻。后者可以算是一个好的模式描述，而前者算不上。

就像大多数有用的软件会随着时间的推移逐步升级一样，模式也在成长的过程中逐步演进。这种成长的过程主要源自我们逐步深入的经验，特别是当我们采用新的有趣的方式使用模式的时候。比如，Abstract Factory模式最初在GoF的书中[GoF95]只是用于对象创建，到了后来的版本中[Hen02b][Bus03a]则同时提供对象创建和销毁。与原来的版本相比，后来的版本平衡了对象生命周期管理的因素和驱动力。

与之类似，Broker模式自从《面向模式的软件架构（卷1）》[POSA1]中的最初描述到目前已经修订过3次。第一次修订是在*Remoting Patterns* [VKZ04]中，将Broker角色从POSA1中的一个角色拆分为多个小角色，每一个小角色更加具体。后来在[KVSJ04]中进一步演进，使用了POSA模式的格式，并在修订后的结构中增加了实现的细节。第三次修订出现在《面向模式的软件架构（卷4）》[POSA4]中，它扩展了第一和第二个版本，将Broker与其他的模式集成起来以实现一个基于Broker的通信中间件。这3次修订的演进过程反映了我们对于该模式的理解逐步深入，并且通过与更多的模式组合成模式序列或者模式语言来帮助实现该模式。但是3次并不是结束，正如John Vlissides所说，“如果你需要一个理由，我们这些讨论应该能够让你相信，模式永无止境。”

永不停息的追求

我们可以试着进一步改进Batch Method模式。比如，我们可以在上下文描述中增加更广泛的适用范围，比如处理对本地内存复杂数据结构的访问（我们在1.5.2节将其作为一个可选的上下文进行了讨论，但是并未将其作为上下文描述放到模式中来）。事实上，使用面向对象技术也可以作为一个驱动力，而不是作为上下文中的一个先决条件。我们还可以在描述中添加已知的应用，比如在哪些产品软件系统中成功使用了这个模式，或者将其与有关的模式做个对比，比如Iterator。

换句话说，对模式的提高是没有止境的。因此，模式社区认为模式是一个正在发展的事物，

需要不断地修订、提高、精练、完善，有时甚至需要完全重写。只有这样，才能明白模式的演进和成长，这一点从我们演示的例子中也能看得出来。其实，只要将《面向模式的软件架构》系列中的模式与之前出版的PloPD系列中的模式做个对比，就能看出它们是如何随着时间推移而演进的。

然而，承认模式是一个正在发展的事物要求我们付出大量的时间。这也是为什么模式的使用者比作者多得多，为什么众多的模式仍然停留在最初出现的地方。另一方面，这些时间和精力上的投入让你可以从软件社区中获得更多的赞誉，并且提高你自己对模式的理解，这就是对你的回报。

比如，如果你花些时间来发掘和记录有用的模式，开发人员可能就能够在自己的新系统中使用或者从已有系统中识别出这样的模式。传播这些好的实践能够提升设计的水平。深入研究并记录模式能够大大提高自己的知识水平，我们对好的设计方案的认知不再停留在偶然性的层次，而是具有更深刻的理解。

1.10 模式既是讲故事，又能发起对话

虽然模式的“最终”版本还未确定，但提高还是很明显的——而整个长度并没有太长——我们不需要单独拿出一节来指导读者如何阅读有关这个模式的描述。模式读起来很通顺，能够自然地从一个逻辑部分过渡到下一个部分。这也是好模式的另一个性质：它必须能够像讲故事一样——尽管也许会比较短。借用Erich Gamma的话说，就我们所关心的上下文而言，它应该是一个“成功的软件工程故事”。

Bob Hanmer [CoHa97]更进一步，他说模式的名字就是故事的题目，上下文就是故事的前奏，问题描述是主题，驱动力创造了一个难以解决的冲突，解决方案是最终的宣泄：最终我们的问题得到了圆满解决，就像“公主和王子最终过上了幸福的生活”。然而，正如我们在本书后面将看到的，故事总有续集。

当然，模式并不只是在讲故事。它也为读者发起了一个对话，关于如何恰当地解决某个特定的问题：首先提出可能会影响问题解决方案的驱动力，然后给出各种可能的解决方案，最后讨论这些解决方案的利弊。模式引导读者去思考自己面临的问题：问题的本质、解决问题时要考虑的各种因素、不同的解决方案以及在读者自己的上下文中哪种解决方案是最可行的。

模式鼓励读者首先去思考，然后再作出决定并采取行动，这个过程是有意识的行为，而不是盲目地遵循某个事先确定的指导方案。模式起一个指导作用，但是所有的活动都在读者自己的控制之下。这种区别使得模式能够成为读者设计知识的一部分：随着时间的推移，这些经验变成了可以凭直觉应用的能力，而不是教条。

1.11 模式不能代替思考

尽管我们可以从详细的问题描述中找到解决方案的种子，但是二者之间的过渡和转换却并不是那么直接。模式并不意味着可以从问题自动得到现成的解决方案。模式应对问题的方式往往是间接的、独特的，有时甚至是反直觉的。

那些严格的开发方法和模型驱动工具强调的是拿来就用，模式与之相反，它非常依赖人的创造力和经验。虽然模式的驱动力限制了可能与可行的解决方案，但是这种限制并非那么严格，不是说根据某种自动的转换就可以得到唯一的输出。相对来说，绑定重构的约束是严格的而且容易形式化的：重构是指在保留功能行为的同时对代码结构进行一定的变化。

在这个例子中，没有使用常见的Iterator模式，甚至连下标索引都没有使用，这可能让一些开发人员觉得奇怪，因为对他们来说，要对聚合对象中的内容进行迭代，使用Iterator或者下标索引才是常规的做法。使用Iterator解耦迭代与集合的表现，已经成了主流语言中的标准做法，以至于它算不上是模式，简直就是默认方案。

打破这种标准做法的是“分布式”这个上下文，减少远程调用次数的基本原则破坏了我们的安宁。对Iterator做任何调整都无法满足这种需求，我们必须基于不同的推理路线找到一种完全不同的方式来解决所面临的问题。

1.12 从“问题-解决方案”到模式

现在我们了解了，模式并不只是在某个特定上下文中出现的问题之解决方案。但是也不是说“上下文-问题-解决方案”这样的格式不能用来描述模式。这是一个非常精练而重要的格式，同时也是每个模式最基本的结构特点。只是，它不能将真正的模式与“普通的”问题之解决方案清晰地区分开。“上下文-问题-解决方案”这样的三段法对于模式而言是必要的，而非充分的。

为了避免误解，这里需要澄清一点：仅仅通过改进描述方式是不可能让一个普通的解决方案自动成为模式的。模式不是靠语言大师创造出来的。只有具有了上面我们讨论的各种属性，问题之解决方案才可能成为一个好的或者完整的模式。一开始的那个例子只是描述得不够好，但是它具备了模式的精髓，所以我们可以一步一步将其完善成一个好的模式。如果某个问题之解决方案缺少一个真正的模式所必需的任何一个属性，它只能说是一个问题之解决方案，或者最多称得上是特定系统的设计和实现决策，但绝对称不上一个模式。

不要与武力对抗，而要学会利用它。

—— R. Buckminster Fuller

本章探讨了模式的普遍特征与特定模式实现具体特征之间的紧密联系。为此，我们讨论了模式在软件框架中所扮演的角色，以及模式与其通用的实现、参考实现及具体案例之间的差异。

2.1 是否存在一个通用的模型呢

我们从第1章得出的一个重要结论是，模式为反复出现的问题提供了一个通用的解决方案：该解决方案可以由不同的方式实现而不必是形式上的复制[Cool98]。我们从模式概念的角度深入讨论了这一结论的意义。但是，定义什么是模式是一回事，怎样利用模式来开发软件是另一回事。下面从代码实现的角度来看一下这个结论。

任何一个软件模式的实现必须考虑其应用环境具体上下文独特的需求。因此，模式的任一具体实现可能与其在其他情况下的实现有所不同。除非需求及其解决方案的变化可以被限定、枚举和匹配，否则，不可能通过对一个通用的实现进行配置来实现所有这些不同的需求。而对需求空间进行限制的任何假设都很可能是不现实的。

需求来自于具体应用的上下文。如果它们的变化是有限的、可枚举的，这些应用的变化和它们的实现技术也将是有限的。虽然这种情况可能存在（在某些应用中甚至是合理的），但这并不是普遍情况。所以说，通过对一个通用实现进行配置来应用模式，在大多数情况下都是不可行的。

这话听起来很绝对！但是，能不能证明呢？为了寻求这个问题的答案，我们试图进行反证。假设可能提供一种可配置的、通用（generic）^①模式实现，而该实现覆盖其整个设计空间。我们会试着针对一个例子来开发这样一种实现。让我们以Observer模式为例，来看看这个假设会将我

^① 这里的generic是指“非特指的”和“普遍适用的”，不应该与泛型编程（generic programming）中的generic混淆起来。泛型编程中的generic是指抽取以算法为中心的数据结构的共性，并基于它对组合进行编程，通常都会放入函数库组件中。

们引向何方。

2.1.1 Observer 模式：快速回顾

很多开发者都非常熟悉Observer模式。该模式在各种框架中被广泛应用，同样也是第0章案例研究中提到被误用的模式中的一种。下面简单地总结一下它解决什么问题、有什么样的解决方案、引入了哪些角色。

对象有时依赖于其他提供者对象的状态或其维护的数据。如果提供者的状态不经通知发生了改变，依赖者的状态可能会变得不一致。

因此，定义了一种状态改变的散播机制。在该机制中，每当消息提供者（provider）（也称目标）的状态发生了变化，都会通知注册的依赖者（dependent）（也称观察者），被通知的观察者可以执行任何对它们来说必要的动作。

图2-1展现了一个常见的Observer模式实现。

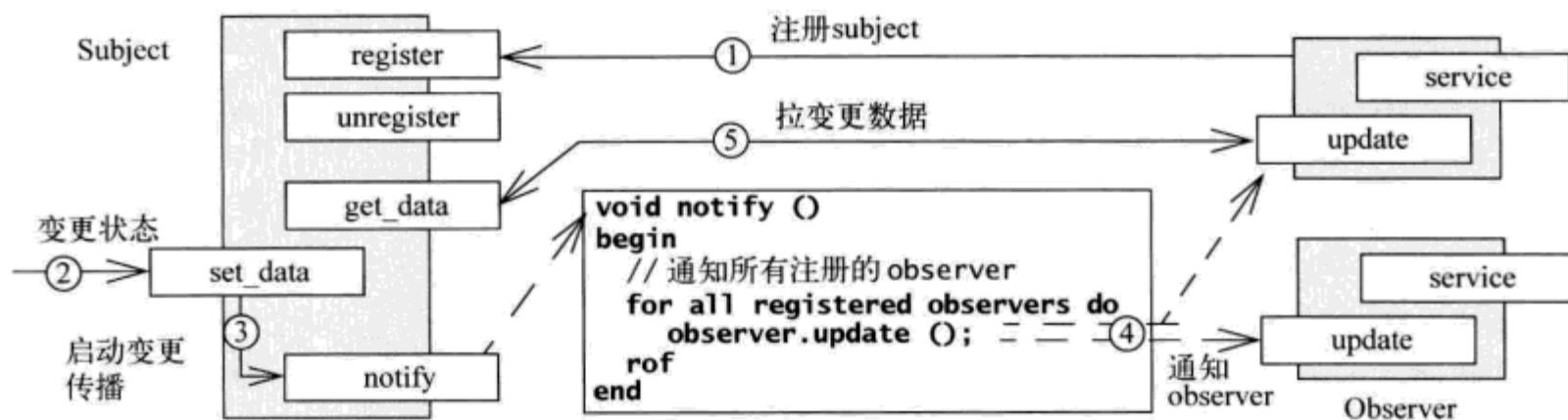


图 2-1

2.1.2 结构的变化与角色

如果我们想开发一个包容一切的通用软件模式实现，面临的第一个问题是：就像第1章所讨论的，不仅引入了组件，模式还为组件引入了角色。根据我们对Observer模式的两个主要角色——目标与观察者——的不同安排，可以得出不同的分布方法。例如，我们可以将一个已经存在的类作为目标，或者将一个新类作为目标。同样的假设对于观察者角色也是一样的。在一些情况下，两种角色甚至可以集中于同一个类身上：该类可能在一个关系中作为观察者出现，而在另一个关系中被当做目标。

观察者可能从单一目标接收消息，也可能从多个目标接收消息。同样，一个目标可能被单个观察者关注，也可能被多个观察者所观察。这种关系可以在目标的注册观察者接口中显式表达，并确定是否将目标的相关知识包含在观察者内。或者通过使用Publisher-Subscriber中间件将目标与其观察者完全解耦。Publisher-Subscriber中间件可以位于与目标（或多个目标）及其观察者（或多个观察者）不同的计算节点上，从而实现对分布式事件通知的支持。

类图2-2展示了一个“经典”的Observer模式结构和一个使用了Publisher-Subscriber中间件的Observer结构。

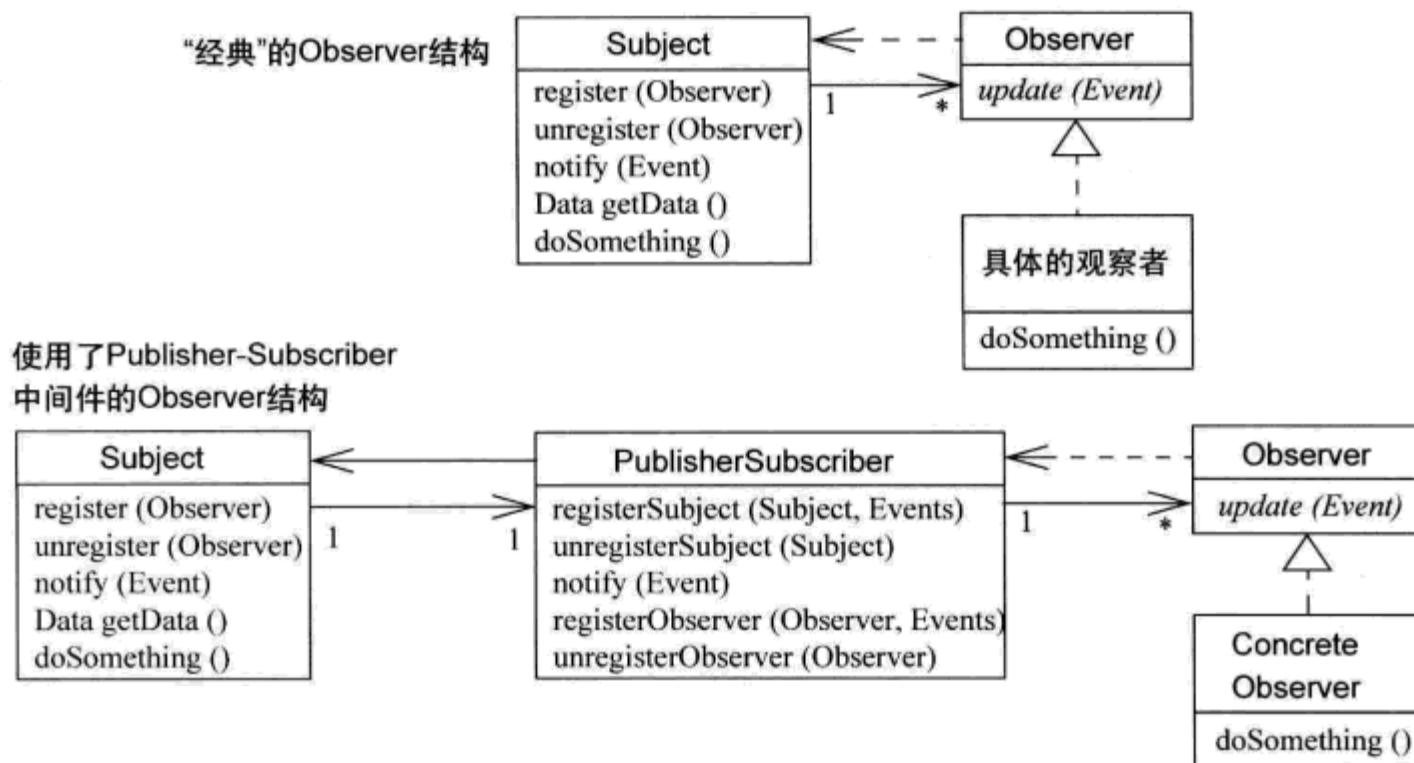


图 2-2

上面描述的各种方案产生了非常不同的Observer模式结构，对类角色进行了不同的安排，使用了不同类型的类。可以看到，即使是在相对较小的范围内发生的结构改变，我们也离实现一个通用实现的目标更远而不是更近了。从这个角度来说，寻求模式的通用实现尽管未尝不可能，但这一可能性显然更小了。

2.1.3 行为的变化

模式的变化并不仅仅局限于结构上的区别，因为从行为的角度来讲，Observer模式也可能有不同的变化。例如，实现状态散播机制时，在目标及其观察者之间的状态信息的交换既可以使用推（push）模型，也可以使用拉（pull）模型。状态信息可以随着通知回调（notification callback）传递（推模型），或者，观察者在接到消息通知后必须显式调用目标（拉模型）。还可能实现基于特定状态的散播机制，即某些状态的处理（或“推”或“拉”）与其他状态不同，而不是所有的状态都一样处理。

时序图2-3展示了一个“经典”的Observer设计的数据推和数据拉交互模型。

另外一些行为变化集中于服务配置属性的差异。例如，一些Publisher-Subscriber基础设施以其接受到的顺序将事件转发给观察者，而另一些架构根据观察者相对重要性的不同来决定事件转发的优先级[HLS97]。同样，在某些Observer模式实现中，所有已注册的观察者都能获得通知，而另一些则支持过滤和关联操作，只有观察者的部分子集会得到特定事件的通知[CRW01]。

Observer模式的行为变化可以独立于其具体结构，也可以在Observer模式的不同结构变体中以不同的方式实现。例如，以“传统”方式实现的Observer模式[GoF95]结构与使用Publisher-

Subscriber中间件实现的Observer模式，其基于特定事件的消息拉模型的实现是不同的。在前一种实现中，目标直接通知观察者。而在后一实现中，这是通过Publisher-Subscriber基础设施间接通知的。模式的通用实现这一问题被各种本质上存在区别或者相互竞争的设计激化了。当这个实现必须覆盖Observer模式的所有可能结构和行为的变体，并要求为特定应用域和运行平台的特定时间和空间限制进行优化时，这一问题会越发尖锐。

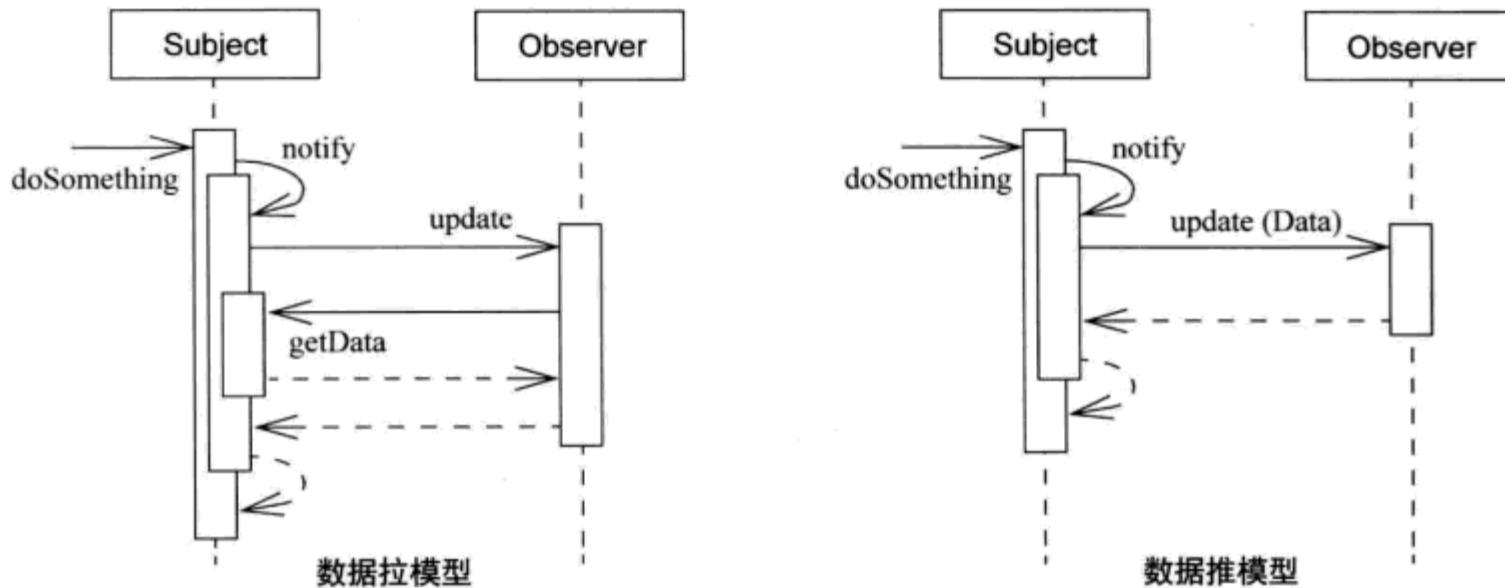


图 2-3

2.1.4 内部差异

因为内部实现也可能发生变化，我们对模式实现之可能变化的讨论还要继续。例如，在Observer模式中，目标常常会采用一个集合（collection）来维护所有对已注册观察者对象的引用。而我们有很多类型的集合可供选择，如映射（map）、集合（set）、包（bag）及序列（sequence）。我们也有许多不同的实现类型，如散列表、树、数组和列表等。这些集合类型的区别往往在于它们的接口以及针对特定操作的性能上，例如有些算法需要对数时间来完成，而有些则需要消耗线性操作时间。区别同样也可能存在于资源占用方面（有的需要采用连续的内存区间，有的使用链接节点）以及模型的组织方式（有的根据位置索引来读取，有的通过键值来操作）。脱离应用Observer模式的上下文，而试图决定哪一种集合类型最为合适是没有意义的。

从内部实现角度来说，另一个可能发生变化的地方是错误处理。在什么情况下，一个观察者在注册一个目标时，却不能被加入目标的观察者集合中？可能是已经达到一个最大观察者数量限额；亦可能是实现采用了集合（set）而非多重集合（multiset），而集合中已存有该观察者；还有可能是系统在一个资源受限的环境中低内存运行。处理这些异常情况是集合的责任，还是目标的责任呢？或是谁负责注册观察者谁来处理？又或者是别的什么组件的责任[Ada82]？

相似的情况也可能出现在Iterator模式上，该模式常常用于遍历存储在集合中的观察者对象的引用。它们是否应该支持引用完整性（referential integrity），以确保迭代器不会指向不再存在于集合中的元素？当前迭代器引用的一个观察者从目标注销，而被认为是有待回收的垃圾对象，这会出现什么结果？同前面一样，对于这样的问题，我们无法提前给出单一的答案。Observer模

式所在的、与应用和运行时环境相关的具体上下文，最终决定了哪种设计是最佳的。

讨论到现在，我们已经可以看到一个模式的任意两种实现可以有根本的区别，即使从引入的3个非常保守的角度（结构、行为、内部实现）来讲，也是这样。急剧增长的复杂度，预示着通用模式实现的设计者及使用者要面临巨大的挑战。

2.1.5 语言及平台的差别

到目前为止，我们并没有谈到编程语言及其他技术相关的话题。例如，针对.NET平台的Observer模式的通用实现无法直接应用于其他平台或其他非.NET语言。举例来说，把用C#完成的通用实现移植到其他语言不可能是最好的，并且从语法上看，不太可能取得好的语法一致性。当移植C#到C++或Ruby时，有些惯用法无法很好地进行移植，有时甚至毫无意义，反之亦然。也就是说，这些实现不是通用的：它们至多在某一给定语言、某一特定运行平台的上下文追求“通用”实现。使用面向方面的编程方法（AOP，Aspect-Oriented Programming）来实现Observer模式时，同样有这种情况[KLM+97]。有足够的理由说明在具体的Observer模式中，对于这两种角色（目标与观察者）的考虑是横切的(cross-cutting)[HaKi02]。以下代码片段展现了使用AspectJ实现observer协议的大致框架[HaKi02]：

```
public abstract aspect ObserverProtocol {
    protected interface Subject {}
    protected interface Observer {}
    private WeakHashMap perSubject0bservers;

    protected List get0bservers(Subject s) {
        if(perSubject0bservers == null) {
            perSubject0bservers = new WeakHashMap();
        }
        List observers = (List) perSubject0bservers.get(s);
        if(observers == null) {
            observers = new LinkedList();
            perSubject0bservers.put(s, observers);
        }
        return observers;
    }

    public void add0observer(Subject s, Observer o) {...}
    public void remove0observer(Subject s, Observer o) {...}

    abstract protected pointcut
        subjectChange(Subject s);

    after(Subject s): subjectChange(s) {
        Iterator iter = get0bservers(s).iterator();
        while (iter.hasNext()) {
            update0observer(s, (Observer) iter.next());
        }
    }

    abstract protected void
        update0observer(Subject s, Observer o);
}
```

将目标和观察者角色实现为方面 (aspect)，将实现Observer模式的代码从角色应用的代码中分离出来了。然而这种解决方案并不比直接实现Observer模式更为通用，因为相关的应用类使用了某种特定语言 (AspectJ) 实现。

采用面向方面方式实现Observer模式的关键优势在于，模式实现更加清楚可见，因此可以脱离应用代码来单独维护和演化[HaKi02]。然而这样做的缺点是，Observer设计中涉及的领域类的某些行为现在变得更间接，其可见性降低了。使用面向方面方法的正面和反面影响都是设计需要考虑的问题。但是，任意特定Observer模式的面向方面实现都服从前面讨论到的因素——模式的核心结构、行为以及内部实现，并且仅可能实现采用模式的所有可能变体之一。例如，上面展示的AspectJ代码片段显式声明了用于交换状态信息的模型，以及用于在目标对象内部维护观察者对象信息的数据结构。而且，AspectJ被绑定在Java语言平台上，这对使用其他语言编写程序的开发者来说可能不那么友好。

那我们能否尝试从元数据层面 (metalevel) 寻求通用性呢？举例来说，使用基于模型的产生式方法，例如在模型驱动架构 (MDA, Model-Driven Architecture) [Fra03]或模型驱动软件开发 (MDSD, Model-Driven Software Development) [SVC06] [Sch06a] [KLSB03] [GSCK04]中，将元描述类型编译为可配置的、可采用多种语言、运行于多种平台的实现？某些生成技术（如模型驱动架构）通过寻求功能最基本的共性来达到通用的目的，而不是通过对更基于上下文的解决方案进行改良来展现设计质量，而这种质量恰恰是我们试图通过模式来探寻的。另一些生成技术（如模型驱动软件开发）试图对具体上下文驱动的解决方案进行改良，但需要以其领域特定语言 (DSL, Domain-Specific Languages) 的通用性为代价。

2.1.6 领域、环境相关的变化

模式实现的变化也不会仅仅局限于模式的可见范围。每个模式的具体应用上下文都会带来许多隐含的需求和必须得到满足的隐性假设。例如，在Observer模式中，开发人员必须决定目标方何时开始传播状态的改变，以避免观察者不必要的更新或级联更新 (update cascade) [GoF95] [SGM02]。目标也可以指定在每次更新后，必须经过一段特定时间才能再次触发状态更新。在观察者一边，开发人员必须决定是否每次得到变更通知都去更新数据，还是只有收到特定通知时才发起更新，如每 n 次通知时发起。适合一种组件组合或者组件组合风格的实现未必适合其他情况。

举例来说，要在具体程序中使用上面介绍的通用AspectJ观察者协议，需要定义好哪些类扮演什么角色，设置好抽象切入点 (abstract point-cut) `subjectChange`和抽象方法`updateObserver`以满足使用该通用协议的具体上下文环境，否则该Observer模式的面向方面实现是不完整的。以下代码片段描述了这样一个应用实现，其中一个图形应用扮演观察者角色，而显示图形的屏幕扮演目标角色[HaKi02]：

```
public aspect CoordinateObserver extends ObserverProtocol {
    declare parents: Point implements Subject;
    declare parents: Screen implements Observer;

    protected pointcut subjectChange(Subject subject):
```

```

        (call(void Point.setX(int)) ||
        call(void Point.setY(int))) && target(subject);

protected void updateObserver
    (Subject subject, Observer observer) {
    ((Screen) observer).display("Screen updated " +
        "(point subject changed coordinates).");
}

}

```

2

使问题更复杂的是，模式实现可能还需要满足系统范围的要求，如实时限制、安全性方面的考虑以及运行时可配置性的实现。另外，实现受系统开发早期其他设计决策的影响。例如，目标与观察者事件的通信还依赖于并发（concurrency）、搭配（collocation）、组件的架构及进行开发的系统环境。目标和观察者是在同一线程的控制下，还是属于不同的线程控制，甚至处于不同的地址空间，随着具体情况的不同，该通信可以以不同的方式来表示。它还可能需要使用中间件平台（如Java[CI01]、.NET[PB01]或CORBA[OMG04a]）来代理通信，或者使用不同类型的网络，如局域网或广域网。

协调这些领域和环境的依赖因素，取决于具体模式实现所面对的功能上、业务运作和开发上的需求。某一应用中非常适合的实现可能完全不适合另一个应用，最终，我们无法提供一个通用的实现适合该模式可能适用的所有应用环境。

2.1.7 再论假设

看起来，前面几个小节的讨论驳回了存在模式的可配置通用实现这一假设。但是，我们可以采取一个比较简单的观察角度，把变化局限在几个精心挑选的、相对狭窄的维度中。例如。我们可以提供一个Observer模式实现，支持单一的结构和行为变化，基于特定的实现语言和平台，着力于解决特定的业务和环境所考虑问题的集合。这个实现可能允许几种错误处理模式，可在特定的语言中配置特定类型的集合和迭代器，可以使用一个满足特定标准的通信中间件平台。

即使在这样的受限情况下，我们要么受限于集合、异常处理模型、支持我们希望在实现中使用的接口的通信中间件，否则就必须插入另一适配层。此外，这一Observer模式实现并不覆盖领域方面的问题，例如一个应用的类扮演何种角色、在哪种情况下角色间发生交互。因此，即使将变化因子局限到一个固定的集合，问题依然很难协调，更不用说一个“更加”通用的模式实现了。

大多数实际的软件项目甚至更加复杂，因为项目中需要考虑的各种变化因子是没有限制的。因此，限制维度数量的变化也帮不上忙：每一个维度都是一个连续体，一杯综合了不同实现技术、应用需求、个人喜好的“鸡尾酒”。一个真正的模式通用实现必须考虑所有维度上的变化，并在该多维变化空间中，考虑每个维度上所有可行的组合变化。一个没有考虑这些变化因素及其组合的灵活的参数化实现也许听起来像那么回事，好像实现了模式的一些特定用途，但从覆盖模式所有可能变化来讲，还是不能算真正的通用。

作为对Observer模式案例研究的结果，我们可以得出结论，为任意给定模式提供一个完全通用的实现是不可行的。但是，这是否可以看做是模式概念的一个缺陷呢？乍一看，我们或许很容易得出这样的结论，但经过仔细分析就会发现，模式的绝大部分作用恰恰来源于模式并不用于解

决特定问题这一特征。

模式的强项在于刻画某一问题的许多特定解决方案的共性，同时支持将这一共性应用到其他应用中新出现的情况下。模式描述了一个连贯的、无限的设计空间，而不是该空间中一个有限的实现集合。是应用的具体要求引导我们达到某个具体的实现。模式的责任在于确保问题的任何具体解决方案服从模式提出的建议。模式的解决方案不是被动、通用的，而是一种主动、生成性的方法。

尽管试图缩小模式的解决方案空间与追求真正通用的模式实现相抵触，但这仍然是非常有意义的。仔细界定和约束“问题-解决方案”的变化是应用框架（application framework）[John97]、产品线架构（product-line architecture）[Coc97]及其他寻求产品共性的方法[CHW98]之基础。这些方法已经在软件开发的很多领域起到了提高软件产品的质量和开发效率的效果。使用这些技术来实现“半通用”的模式实现，具有以下作用。

- 支持有效、正确地使用模式，帮助避免模式的误用。
- 提高开发人员的工作效率，尤其是当这些实现恰好与你的项目基于同样的语言、平台和领域时。

与完全通用的实现相比，定制的、特定应用和领域的模式实现，甚至是常常在框架和产品线架构中可见的实现，往往是更简单、更优雅甚至更高效的选择。这些做法将设计的关键部分交还于开发者控制，而不是任其他人根据其对该模式的理解来处理。

成功提供预定义的、部分通用的模式实现之关键是在给定的应用领域内，在引导与处理可变因素的自由之间找到合适的平衡。在软件开发的某些领域，如嵌入式系统设计，可能提供相当固定的结构，只提供少数可配置的方面。在另一些领域，几乎任何一个预先确定的模式实现都会妨碍人们高效地、成功地在具体项目中应用模式，例如在软件开发组织和软件开发流程[CoHa04]领域。因此，以下的考虑在很大程度上仅仅适用于设计模式，而这些设计模式恰好是现今软件项目中被文档化和应用的一些主要模式。

2.2 模式与框架

尽管一个通用的模式实现现在看来始终只是海市蜃楼，但是还是有一些通用的模式实现得到了广泛、成功的应用，其中包括Java AWT和Swing库[RBV99]、Boost [Boost]以及ACE [SH02][SH03]。但是，这些库并不试图提供模式的完全通用实现。而是作为框架^①，给特定的应用领域提供了预定架构和部分实现。一些框架提供了特定的属性，使其可以在新的上下文中提供新的功能。在该范围外，任意框架是否适用于特定目的变得明显。“是否适用”的判断对经典的面向对象应用框架以及作为产品线架构基础的生成性框架同样适用。

例如，Swing支持使用Java实现图形界面的系统。ACE支持使用C++实现网络和并行应用。这些框架的设计主动应用了许多模式，比如“四人帮”著作中所记录的[GoF95]和POSA系列[POSA1] [POSA2] [POSA3] [POSA4]中提到的那些模式，但它们是在一个较为特定的上下文中，

^① 如非特别说明，我们这里的“框架”采用Carolyn Morris的通用定义——“工作模型基于的架构”，而非另一个常见但较为具体的说法——一组支持互操作的类在某个特定应用中的子类化[John97]。

为了一个较为特定的目的，在特定的限制条件下的实现。当满足这些限制条件时，框架在实践中可以是非常有用的。

对于一些系统来说，框架中模式的使用可能是正确的，但它们的实现却未必合适。例如，ACE框架中用C++实现的模式，在Smalltalk或Ruby应用中往往只是一些简单的直接用法。有意地、超出其能力范围地使用框架将会降低而非提升软件的质量[SB03]。不是试图使架构被某一模式驱动，开发人员必须对框架的特定实现进行权衡。例如，ACE[SH03]中的Reactor框架是Reactor模式的实现，该模式将通用I/O、计时器和网络信号事件进行多路分解，并发送到处理相应事件的应用组件。图2-4和表2-1展示了ACE Reactor框架。

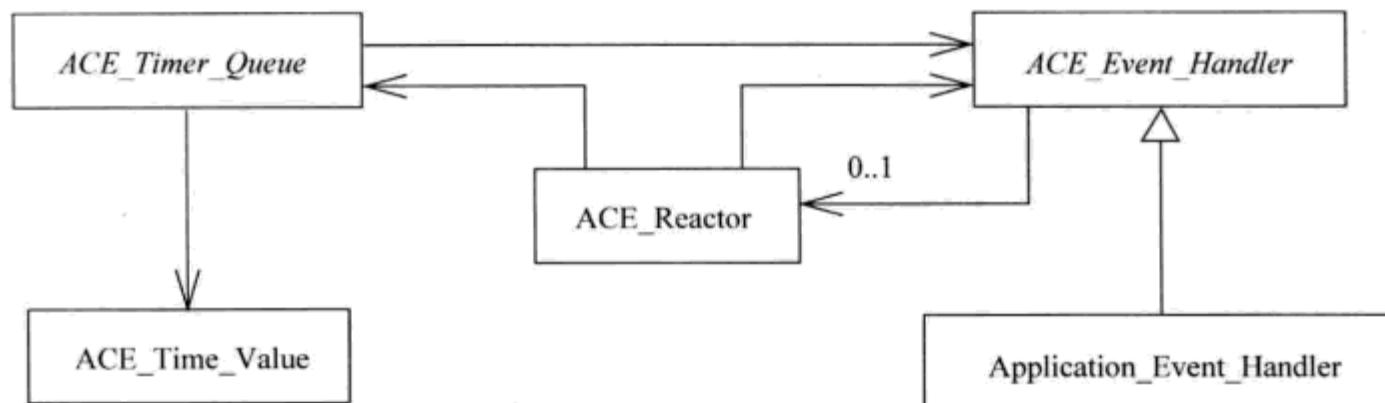


图 2-4

表 2-1

模 式	描 述
ACE_Time_Value	使用C++操作符重载的方式，提供一个可移植、标准化的时间及持续时间的表示方法，以简化时间相关的算法和相关操作
ACE_Event_Handler	一个抽象类，其接口定义了供ACE_Reactor回调的钩子方法。大多数使用ACE开发的应用事件处理程序都继承自ACE_Event_Handler
ACE_Timer_Queue	一个定义了计时器队列功能及接口的抽象类。ACE包含了很多不同的均继承自ACE_Timer_Queue的类，为不同的计时需求提供弹性支持：ACE_Timer_Heap、ACE_Timer_Hash、ACE_Timer_Wheel和ACE_Timer_List
ACE_Reactor	提供管理事件处理程序注册及执行事件循环的接口。该事件循环是ACE_Reactor框架中事件侦测、多路分解、分发的基础。ACE包含了多种具体的ACE_Reactor实现：ACE_Select_Reactor、ACE_TP_Reactor和ACE_WFMO_Reactor

但是，ACE Reactor框架实现并不能简单高效地处理GUI及其用户界面元素产生的特殊事件，而这也是Reactor模式[POSA2]适用的另一种情况。

当然，这里我们想要表达的观点并不是ACE Reactor框架不够通用，而是该框架在一个定义良好的应用上下文内做到了尽量通用。它非常适合作为C++实现的、平台独立的、面向对象的网络应用的基础。相反，我们要强调Reactor模式本身是比任何给定实现更为广义、更为深刻的概念，相关例子可以参考该模式在其他框架中的使用情况，如Interviews [LC87]、Xt工具集 [NOF92]以及Java的Selector [Lea00]。

2.2.1 工具和上下文环境

在定义良好的上下文中，框架代表了一种共性，开发人员可以使用框架避免重新发明轮子——以及所有相关的问题，比如大小、形状、颜色和纹理。这些上下文很少是独立的，就如我们在1.5节中讨论到的，解决方案及其应用上下文是否能够很好地配合是非常重要的。框架有时候会在其作者设计的上下文之外找到非常合适的应用场景。换句话说，框架是可重用的，这里的重用即通常意义上的“重用”，指在不同的上下文中为了不同的目的而使用[JGJ97]。

一个综合框架的设计往往不可避免地（有时甚至需要）包含了很多模式[Gam95]。事实上，随着模式的流行，20世纪90年代中期以来开发的框架无不是大量使用了模式。但这并不是说框架使用了模式就达到目的了，框架的目标由模式来支撑，否则，框架不过是一个模式打包商，从根本上来讲，毫无意义：纯粹是拿着方案找问题。

2.2.2 两个框架的故事

举一个反面的例子，回想一下JDK中`java.util`包的`Observable`超类和`Observer`接口。`Observable`超类实现了一个简单的、基于标志和数组的消息通知机制，需要被“目标”类型继承和使用。相应的`Observer`接口的定义就比较模糊了：`Observer`对象唯一的更新(`update`)方法是弱类型的，它接收`Observable`对象的引用（通知来源）及一个表示通知内容的`Object`对象作为参数。这种简单的一——几乎可以说是过于简单的一——通知协作架构甚至在Java自己的核心库中都没有得到应用，这多少说明了一些问题。

尽管Java有静态类型检查系统，但如果试图使用`Observer`类和`Observable`接口来建立观察者和目标之间的协议，该类型系统就会被绕开。通知消息的类型及其相关的事件信息被埋藏在其具体实现中，被隐藏在级联的运行时类型检查中。当然，开发人员可以选择使用Java提供的`Observer`跟`Observable`接口来在自己的代码中实现`Observer`模式。但是，这里的问题不在于是否能够采取某种做法：如果我们是在讨论开发人员的独创性和计算机资源，原则上一切都是可能的。我们的问题是某种做法是否有效，是否高效，这是一个有关质量而非可能性的问题。

下面的例子展示了`Observable`和`Observer`存在的问题。

假定我们有一些值可以被观察到，它们不会下降到0以下，并且还有上限。当到达0或下降到0以下时，值被置为0，观察者被通知；当超过上限时，观察者同样要被通知。使用`Observer`和`Observable`，值看起来如下所示（省略`import`语句及其方法和字段细节）：

```
public class ValueSource extends Observable {
    public ValueSource(int limit) ...
    public int getValue() ...
    public int getLimit() ...
    public void increaseValue(int amount) ...
    public void decreaseValue(int amount) ...
}
```

通过继承，`ValueSource`类获得了`Observable`类的完整接口和实现。`Observable`类提

提供了一个基于标志的方法来管理通知，需要ValueSource的实现者在发布更新前通过一个方法来设置这个标志。

默认情况下，Observable接口并不需要反映ValueSource对象的行为，因为它同时允许注册和注销任意观察者，而无需检查已注册的观察者是否知道如何处理ValueSource的变化。这一问题的检测可以推迟到通知发出的时刻，或者通过覆写原注册方法来处理。

尽管可以通过覆写的方式来改动实现，但我们还是没有合适的方法来反馈给调用者“这个观察者已注册，第二次注册将会忽略”这一信息。我们没有办法来改变返回值以返回一个成功或失败的标志，或在调用违法方法协定或响应过快而超出目标对象处理能力时抛出异常。

通知协议可以简单地通过下面两个空类来表示：

```
public class ValueExceededLimit {}
public class ValueAtZero {}
```

假设下面的观察者示例可以同时观察ValueSource和SomeOtherSource目标，代码如下：

```
public class ExampleObserver implements Observer {
    ...
    public void update(Observable subject, Object event) {
        if(subject instanceof ValueSource) {
            ValueSource valueSource = (ValueSource) subject;

            if(event instanceof ValueExceededLimit) {
                // 处理值超出限制的通知
            }
            else if(event instanceof ValueAtZero) {
                // 处理值为零的通知
            }
            else {
                // 处理未知通知的错误
            }
        }
        else if(subject instanceof SomeOtherSource) {
            // 处理SomeOtherSource的通知协议
        }
        else {
            // 处理未知目标类型的错误
        }
    }
    ...
}
```

这个例子很好地体现了，基于Observer和Observable的Observer模式实现会在一个不断增长的if...else...if结构中，包含大量的转换和 instanceof 运行时检查。这样一个实现在可维护性上多差啊！优雅更是无从谈起了！一个对于程序员和编译器来说更容易理解、效率更高的设计，会在声明这些协作类型接口时显式要求说明需要进行协作的对象类型。

相较之下，同样是JDK中的EventListener模型就更加开放，更适合具体情况，在Java库需

要通知机制的地方也得到了更广泛的应用。使用该模型，可以定义消息通知中的通知和消息的具体类型。但Event Listener并不是通知机制的通用实现，它不过是一个遵从Observer模式实现的某种风格，用于表示通知关系的标记接口。

这种通知关系在Event Listener模型中是明确表述的，并进行了良好的分割。举例来说，Window Listener与Menu Listener的区别可以从它们各自接口支持的通知类型判断出来，就跟它们名称上的区别一样明显：windowActivated、windowClosed区别于menuSelected、menuCanceled。下面的代码通过修改上面给出的Value Source的例子来展示Event Listener模型是多么清晰。

Value Source类的第二个版本遵从Event Listener模型协议，而不再像前面那样使用类库：

```
public class ValueSource {
    public ValueSource(int limit) ...
    public boolean addValueListener(
        ValueListener observer) ...
    public boolean removeValueListener(
        ValueListener observer) ...
    public int getValue() ...
    public int getLimit() ...
    public void increaseValue(int amount) ...
    public void decreaseValue(int amount) ...
}
```

除了Value Source的开发者现在必须管理注册步骤，上面这段代码与该例第一次出现的代码乍看起来非常相似。而这个注册管理步骤也不过是两个单行方法加上一个集合。这种方式的主要优势在于开发人员现在可以完全掌控注册界面及其行为，而注册动作现在是强类型的了。通知协议变得更加明确而内聚，也更容易识别。

```
public interface ValueListener {
    void valueExceededLimit(ValueSource subject);
    void valueAtZero(ValueSource subject);
}
```

与本例的第一次实现的主要区别在于通知的处理：

```
public class ExampleValueListener
    implements ValueListener, SomeOtherListener {
    public void valueExceededLimit(ValueSource subject) {
        // 处理值超出限制的通知
    }
    public void valueAtZero(ValueSource subject) {
        // 处理值为零的通知
    }
    ...
    // 处理SomeOtherSource通知协议的方法
}
```

与该例第一次讨论中的Example Observer代码相比，Example Value Listener的代码更

为清晰地揭示了观察者和目标的关系，没有类型转换，没有奇怪的运行时错误。第一种方法试图减少实现一个目标类的代码量，而第二种方法成功地处理了代码复杂度的根源，减少并简化了实现观察者这一角色所需要的代码。

不同的上下文之间目的的区别导致实现上不可避免地存在差异。一个框架怎样才能又好又通用呢？一个好的框架在定义上并不是一个普遍通用的实现，试图成就这样一个通用的实现从现有经验来看，其结果可能不会太好。相反，它可能更像是模仿优秀设计思想的赝品，而不是优秀的设计本身。

2.3 模式与形式主义

同样的问题也存在于模式的规范化定义中，相应的讨论也一直持续不休。有人一直认为，模式纯文本的描述加上几幅图表不能足够精确地确保正确的实现^①。“蓝图”一词经常与问题的精度与准确性联系在一起使用，但这也可能产生误导[Bry02]。

作为一个设计或计划的隐喻，蓝图已经被过度滥用了。如果非要使用，至少要记得蓝图是一个完整的计划，而不是一个草案。

模式并不是一个确定的目标，所有细节均已敲定，等待着直接使用。模式定义了一个空间而不是一个点，一连串的解决办法而不是一个单独的答案。它应该被看做一副草图或一种灵感，而不是蓝图或是需求。模式定义了需求同解决方案之间的关系。如果没有很好地理解动机、上下文以及问题驱动力，探讨模式的解决方案不会有任何意义。

受限的与不受限的通用

鉴于解决方案的多样性以及对特定应用的需求和约束的紧密依赖，在不损失本意和通用性的条件下，几乎无法定义模式的任何一个方面。这样的结果往往只能描述有限的几个例子，甚至有时候只能描述一个例子，而不能代表所有可能的实现。虽然这种规范可能是合理的，从产生式编程（generative programming）[CzEi02]的角度来说可能是一种有效的产生式设计，但是它并不是一种模式。

好的产生式编程案例来自于仔细界定问题，有限枚举可能的设计变化。产生式编程能够在不重复代码和工作量的情况下生成一系列软件，其关键在于它对问题和解决方案空间的把握，并且有意识地排除与应用不相关的特性。虽然最终产生的架构可能非常好用并且易于修改，但并不是说就可以满足可以使用该模式的应用的所有要求。

另一方面，如果解决方案的许多重要方面都由于无法得到通用的实现而未定义，开发者希望根据该方案来开发，就必须自己定义那些缺失的部分。如同代码生成器仅仅产生了类名、操作签名和配对的大括号一样，这种方式完全无法满足模式及其描述的通用性需要。输入速度不是软件

^① 模式的规范化是一个流行的运动，特别是在学术界。目前已经发展出了许多不同的方式，其中包括但不仅限于下列参考文献：[Hed97][LBG97][MK97][LK98][Mik98][BGJ99][EGHY99][CH00][XBHJ00a][XBHJ00b]。

开发的瓶颈，而对软件的理解才是。一个仅仅提供了充满了“待办”标记、简单地忽略了所有微妙而错综复杂的部分的代码框架并不符合设计和解决方案的定义。

开发人员既要处理好形态^① (formalism)，又要处理常常比最终代码还要多的文档规范。如果缺乏对其他工具和开发方式——如构建、验证和产品线架构演进——的支持，一个“半吊子”的形态实际上更像是一种负担而没有什么好处。在软件开发中，形态的角色依赖于多种因素，取决于特定环境，比如在强调安全性的系统中，需要可认证的 (certifiable) 属性，或在一些发展成熟的领域中，解决方案是有限的和重复的，因此可以采用自动化代码生成的方法。特定情况下证明了某种方式有效，并不意味着这种方式使用的范式可以作为一个适合所有设计的模型，特别是当该模式的关注点是通用性的时候。

任何一个声称自己是对某个模式的标准描述的描述都是荒谬的：它必定是所有可能性的一个子集。跟归纳证明方法一样，说自己是“完整”描述总是没有意义的，尽管我们可以给它添加更多的案例支持。追求全面所引入的复杂性，带给开发者的往往是问题而不是解决方案。

因此，寻求模式的通用解决方案与形式化方法对必要细节的要求相冲突。注意，这里并不是说将特定模式实现文档化毫无用处。例如，形式化技术（如领域建模语言[GTK+07]）在与基于契约的设计 (design by contract) [Mey97]及声明约束语言 (declarative constraint language) [WaKl98] 中的前置条件 (precondition)、后置条件 (postcondition)、不变量 (invariant) 结合使用时，有助于在特定应用或产品线中实现面向对象的模式。同样，UML[OMG03]衍型 (stereotype) 及参数化协作 (parameterized collaboration) 可以用于标记特定的类以标示它们的角色或给出常见的结构设计建议[BGJ99]。

图2-5使用UML衍型描述Observer模式的例子。在该图中，ClassA的实例被ClassB的实例观察，而ClassB的实例被ClassC的实例观察。

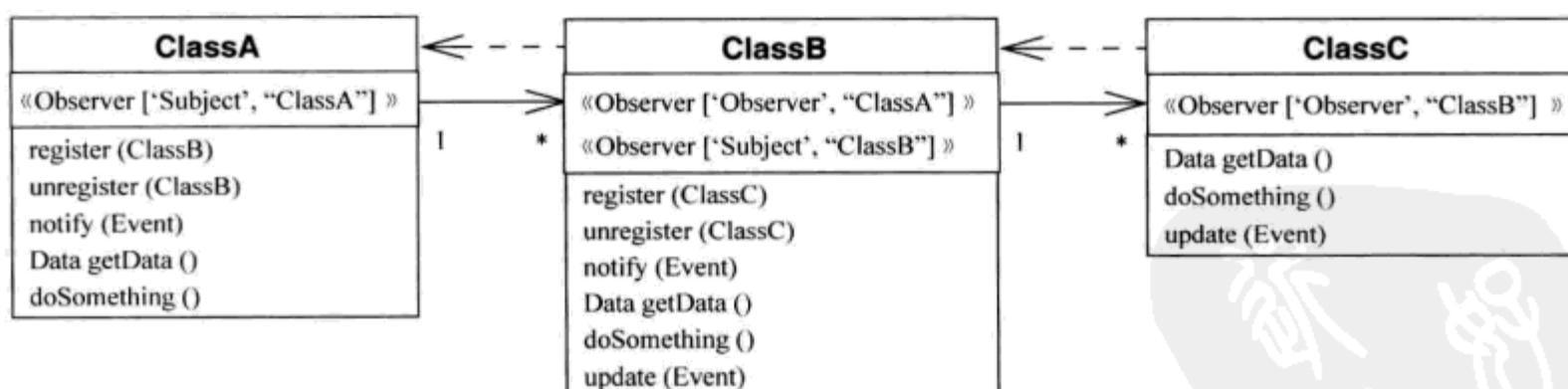


图 2-5

虽然特定模式实现的形式化表述可以帮助用户理解它们，可是这种表述并不能称为参考规范。同样，任何特定模式实现的形式化描述都不试图覆盖所有可能的模式实现。

2.4 通用性与特殊性

现在回头看看我们的假设，通过试验和在这一章作出的其他探索，我们可以看到：模式的通

^① 这里指将风格与形式置于内容与实质之上。此处根据上下文译为形态。——译者注

用实现根本只是一个神话——甚至理论上都不具可行性——在实践中只是浪费时间之举。模式既不是模板也不是蓝图，虽然它们对于指导模板和蓝图的创建确实很有用处。Christopher Alexander在他原先的模式表述中说[AIS77]。

“每一个模式都介绍了一个在我们的环境中反复出现的问题，然后描述了该问题的解决方案的核心。通过这样一种方式，你可以反复使用这个解决方案，但每次的具体做法都会有所不同。”

换句话说，针对某个应用或某类应用的好的模式实现，可能在该上下文或相似的上下文之外就不再可用了。在一种场景下所做的调整在另一种场景下，由于其内在的驱动力发生了变化，可能就完全不适用了[Cool98]。

这种限制并不是说我们无法找到具体的可重用技术（如框架、产品线架构和生成工具）中的通用性，而是说这些技术的可行性和价值恰恰是来自于对其变化类型和应用环境的限制。一个好的框架或模型驱动工具显然比一个供专门目的使用的、手工打造的实现更为通用。然而框架及工具的价值不是来自完全覆盖所有可能性的尝试，而是试图实现特定领域下的通用结构、行为及控制流。

参考实现及范例实现

我们已经看到，框架实现可以提供绑定了特定模式的机制。通过使用框架，避免开发人员再次从头实现自己的版本。我们试图澄清的误解来自于错误地认为框架实现中的模式就是模式本身。从框架实现模式这个意义上讲，它确实是模式的一个具体实现，但这种实现既不是一个参考实现，也不是模式的一个通用实现。

在软件开发中，参考实现常常是某个特定标准的实现，不光是概念的证明，同时也是作为一个“黄金标准”来衡量其他实现。如果一个实现在一些重要的方面——那些已经被标准化的方面——与众不同，会被认为是错误的。这种方法对标准来说非常有价值，对一些定义“设计模式策略”的企业架构团队来说是非常有“利”的资源，但对模式概念来说显得非常奇怪。模式并不是POSIX、Java、CORBA及XML这种意义上的标准，所以参考实现的概念在这里是没有意义的。经常会产生误导，有些情形下将某一特定模式实现认作整个模式本身甚至是有害的。

例如，ACE中包含多个受到POSA1、POSA2中模式影响的框架，比如Pipes and Filters、Reactor、Proactor、Acceptor-Connector、Component Configurator以及Active Object。但是，这些框架只是模式的实例，而不是模式的参考实现。正是在这些模式的实际例子中，我们找到了我们寻求的模式应用的多样性和正确性。模式基于反复出现的案例，所以案例有助于在模式覆盖的设计空间中进行定位。

实例实现还可以告诉我们很多模式及软件开发最佳实践的东西。是的，很多实例实现，比如[GoF95]书中的Lexi绘图编辑器（Lexi drawing editor），就是只用于教学而非生产目的的。在特定设计模式的教学中，简化表述和练习细节，避免学习者从理解模式核心概念中分心是有必要的。出于这种原因，GoF给出的Observer模式的实例实现非常简单，确是有意为之，而不是作者的疏忽。如果将它设计得更全面一些——也许有些讽刺——反而会更加难以理解。

文字的不同排列产生不同的语意，语意的不同组合产生不同的效果。

—— Blaise Pascal

本章介绍了模式的写作格式，并回顾了第1章中提到的一些概念。模式格式是模式赖以传达其精髓及其实用性的工具。模式格式对模式的读者和编写者同样重要。

3.1 风格与实质

既然我们对模式的角色及其内容已经有了一个更为准确的认识，就可以把注意力转向怎样有效地表示出它们来。尽管可以认为模式独立于其表示——表示不过是交流模式的一种工具——但是表示绝不仅仅是次要的细节。事实上，模式的表示对于整个模式的贡献与模式的内容同样重要。这并不是一个风格还是实质的问题：沟通与内容密不可分，以至于风格本身就是实质的一部分——它们之间并没有什么对立或冲突。

模式社区最初的一个目标是传播那些软件开发中的优秀实践。从实用的视角来看，如果好的实践无法得以传播，其结果等同于完全没有任何实践：就像完全不曾存在过，或跟其他没有得到很好传播的实践一样不会被人注意到。更糟糕的情况是，好的实践没有得到传播，却输给了那些得到了很好传播的较差的实践。Dick Gabriel[Gab96]在其论文“Writing Broadside”中表达了这一观点，其中他引用了Thucydides[Thu81]在两千年前的话：

“有知识但无法清楚表达的人，并不比他从未有过任何想法强多少。”

因此，合适的表达对于成功传播模式至关重要。某一模式能否取得成功，其合适而准确的描述方式也同样举足轻重。模式的最终读者，归根结底还是人。虽然可以通过使用函数库或生成工具来使用某种软件模式，但始终还是要由程序员而不是技术来理解模式。一种模式不仅仅是具有特定结构的解决方案，其使用者必须对该方案的上下文、驱动力以及使用该方案的结果[Cool97]有所了解。

模式可以通过面对面的交流、白板、信封上的草图或任何随手可得的媒介来完成沟通。但是，完成这样的交流对地域有很高的要求，而更常见的交流形式是通过文字来完成的。相较于与模式

专家进行对话，通过文字来进行模式交流更像是叙述性而不是交互性的。

在与模式作者或模式推广者进行交流时，潜在模式使用者有机会对有疑问和不理解的地方进行提问和澄清。他们也可以探究细节，谈论一些特定案例并互相交换意见。通过写作这一媒介，作者可以确保更高的准确性和对表述的掌控，但同时失去了模式文本背后的声音，从而无法在实现细节、问题的适用性以及对问题澄清方面给读者提供帮助。对话消失了：读者无法参与对话，而模式作者无法回应读者的问题。这是模式写作时需要克服的一个挑战：这里还是需要某种形式的对话。

模式文档化的叙述性要求其不仅要忠实地、明确地表述模式，同时还应该在某种范围内还原读者与作者的对话，而不是呆板的独白。如果要在以精准但是呆板的形式展现给读者和不那么全面但能更好地吸引读者这两者之间作出选择，模式作者应取后者。

再看 Batch Method 的例子

在第1章中，我们通过把一种模式文档化来探讨模式的概念，从上下文到其产生的影响，再到其表现格式。本章的重点有些不同：这里我们将会更加关注模式的表现格式而不是该模式的概念解析本身。

下面是我们之前得到的模式描述。

上下文：客户端需要在聚合数据结构上执行某些操作。

问题：客户端可能需要获取或者更新集合中的多个元素。然而，如果对聚合的访问成本很高，那么用一个循环来挨个访问每个元素可能会带来严重的性能问题，比如往返时间过长、阻塞延迟过大或者上下文切换开销过高。如何才对聚合对象进行高效的、无中断的批量访问呢？

解决方案：定义一个可以在聚合对象上重复执行多次访问的方法。客户端在一次方法调用中访问多个元素，而不是在一个循环中单独地、直接地访问聚合中的元素。

经过反复斟酌和优化，下面是我们最终得到的描述。

Batch Method

在分布式或者并发的对象系统中，聚合对象的客户端可能需要在聚合上执行批量操作。比如，客户端可能需要取回所有符合具有某个特定属性的元素。如果对聚合的访问成本非常高，因为它相对于客户端是远程的或者是由多个线程共享的，在一个循环中单独访问每个元素——比如通过索引、键或者Iterator——都可能引入严重的性能问题，比如往返时间、阻塞延迟或者上下文切换开销。

如果访问成本很高并且可能失败，怎样才能高效地、不中断地在聚合对象上执行批量操作呢？

在解决这个问题的时候，必须考虑4项驱动力：

- 聚合对象是由并发环境中的多个客户端共享的，可能是在本地的多线程环境，也可能是跨网络的分布式环境，我们可以在一次方法调用中对其进行同步封装，但是对于在循环中重复的调用却不合适。

- 阻塞、同步和线程管理的开销必须在每次跨线程或者跨进程的访问中考虑进去。其他的针对每次调用的操作（比如授权）会进一步降低系统的性能。
- 如果聚合对象相对于客户端是远程的，那么每次访问还会带来延迟、抖动，并占用网络带宽。
- 分布式系统可能会出现部分失败的情况，这时候客户端可能还活着，但是服务器却已经崩溃了。在迭代过程中进行远程访问，可以为每次循环引入一个潜在的失败点，这可能会导致客户端面临只完成部分遍历的情况，比如不完整的状态快照或者不完整的更新。

对于给定的操作，定义一个可以在聚合上反复执行该操作的方法。

每个Batch Method都是聚合对象的接口的一部分，或者是作为整个聚合类型的Explicit Interface暴露出来，或者是作为一个更窄的混入Explicit Interface——仅仅定义调用重复操作的方法。在Batch Method的声明中，可以接受每次执行操作所需要的所有参数，比如通过数组或者集合，结果的返回也采用类似的方式。这样我们就不需要在循环中直接挨个访问聚合元素，客户端直接调用Batch Method，就可以在一次调用中访问多个元素。Batch Method可能由聚合对象的基础类直接实现，也可以通过Object Adapter方式间接实现，这样就不需要对聚合对象的类做任何修改了。

通过Batch Method，我们将重复的操作转化为一个数据结构，这样我们就不需要在客户端进行循环，而只需要在方法调用之前和结束之后通过循环做一些准备和后续工作。于是，对聚合的访问成本降低为一次访问或者几次“较大型”访问。在分布式系统中，这种“压缩”机制可以大幅度提高性能，并减少网络出错的概率，节省宝贵的带宽。

使用Batch Method会提高单次访问的成本，但是批量访问的总体成本还是降低了。这样的访问也可以在方法调用中配合使用适当的同步技术。每次调用可以是事务性的，即或者完全成功或者完全失败，不存在部分失败的情况。

使用Batch Method的代价是在准备和后续阶段要做更多的工作，并且需要引入中间数据结构来传递参数和接收返回结果。网络、并发和其他针对每次调用的开销越大，使用Batch Method就越合适。

注意这段描述的结构以及它是怎样把我们认为定义该模式重要的各个方面引出来的：模式的名字、问题发生的上下文、问题的描述总结、解决该问题需要平衡的各个驱动力、推荐的解决方案概要、解决方案更详细的描述以及引入该方案会带来的影响。

此外，还有其他风格的描述方法来说明模式，例如模式名称的突出显示、驱动力列表、使用一个问句来总结问题所在。这些风格相关的选择是模式的描述格式的问题：改变描述和格式将在某种程度上影响其可读性——它怎样才能更好解读，是基于怎样的目的，其受众是谁。

3.2 格式的功能

模式的写作是其作者的一种创造。每个模式描述都是独立的，但也不是完全没有规则可循。

承载模式的是其自身的描述格式，正是格式指导着它的整体外观、结构和文风。然而，就如同软件开发者之间流行的关于终极编程语言、最“王道”的对齐方式，以及诸如此类的神圣问题的论战，对完美的模式描述格式的追求也注定没有答案。

由于个人审美的不同、模式类型的区别，一种模式描述格式不可能适用所有的情况，满足所有的需求。这些问题更加凸显了模式描述格式的重要性，而不是在暗示我们应该抛弃它。除了个人喜好的影响，模式描述格式必须提供这样一些功能：作者的审美必须与内容中所包含的技术信息以及与目标受众进行顺畅沟通等要求相结合。同模式中的驱动力一样，这些功能之间也需要取得某种形式的平衡。

格式的疑问

模式作者需要确保其技术内容有适当的深度和广度，其表达方式容易被潜在读者所接受。该模式的上下文是什么？需要平衡的驱动力有哪些？会有什么样的（好的和坏的）后果？读者期待的是一个高层次的概括描述，还是一个详细到API甚至大括号的说明？读者究竟是一个正在着手寻找具体解决方案的、经验丰富的架构师，还是一个寻求指导的新手？描述该模式的理想案例是什么样的？该模式是否是一个更大模式集合的一部分，并且应该与其他部分取得某种一致性？

如果存在这样一种唯一的格式，得到所有模式作者的遵从，就能避免很多这样的问题，将作者从这种伴随自由意志而来的负担中解放出来。跟我们在第2章中讨论到的通用实现的问题类似，参考格式可能会变得笨重而难以理解的。它可能会变成包含各种选项的元模型。模式的写作会变得非常死板，就是对一套表格的填写——阅读这些作品时也会有相同的感觉。相反，模式格式健康的多样性使得作者在遵守一定的基本原则并与其他模式保持一定相似性的基础上，可以自己判断表述格式。

作者必须对使用哪种格式来描述模式作出决定。相对于单个模式对格式的要求，对于集中在同一个编目或同一个语言下的模式来说，格式的一致性对作者和读者影响更大。单独来看，某些模式可能适合很多种不同的描述方式：简短、叙事性的描述或者较长、结构化的描述。如果把它们放在一起，对同一类模式或同种语言实现的模式使用不同格式的描述看起来会比较混乱，为模式间的比较、观察模式之间的联系和承接关系带来不必要的麻烦。格式的不一致往往意味着作者优柔寡断，甚至是稀里糊涂。

3.3 格式的元素

如果我们希望模式能指导我们有关设计、开发角色和业务实践活动，并希望它能生动地展示其优势和使用范围，模式的格式就必须反应出我们感兴趣的问题和关注点。那么，一个给定模式描述的重点是什么？是通过一种软性的、叙事性的、没有固定明确章节名称的风格来交流模式好，还是通过一种更具结构性的格式来说明会比较好？使读者意识到他们自己实践中使用了这种模式的解决方案，或是更快地将其应用到新的工作中有多容易？

在审视模式的自然流程时，我们首先遇到的是模式的名称：就我们的模式示例来说，就是Batch Method。如果模式的名称过长、过为笨重、过于隐晦或者过于吸引人，其作为设计词汇元

素的作用会很难被实现。一个长一点的名字会更精确，但又会比较不方便。如果采用比较隐晦或比较吸引人注意力的名字，可能会更好记，但相应地其信息性又会有所不足。除非读者之间能够共享比喻与模式之间的联系，否则读者更容易记住它的不明确性而非模式的洞察力。

模式并不是通用规则或普遍原则，因此，模式可以应用的某种上下文应该是清楚定义的。应用上下文将模式置于设计、管理、商业或模式的擅长处理领域的世界中，以判断模式是否适合应用于此。认为模式是上下文无关的，常常导致模式的误用。尽管计算的本质具有通用性，但在某一领域中学到的架构知识并不能自动盲目地转换到另一领域。例如，在解决本地地址空间的设计问题时，顺序性控制问题时使用的模式很少能有效或者正确地应用到具有固有并发性的分布式系统中。在Batch Method的描述中，我们将其置于分布式、并发环境中，并预览了它要解决的问题。

如果驱动力是模式的核心，这种重要性应该也要反映在模式的描述格式中。读者应该不需要解构和剖析模式，就能了解其解决的问题本质。对处理问题简洁、直观的总结，可以确保细心的读者对他们将要读到的内容有一个不错的概览，而那些随便一观的读者也有一个足够完整的印象，以不致产生误解。在表述完上下文和主要动机之后，我们的Batch Method描述将其设计难点作为一个问题提出。

用来列举模式驱动力的方式有很多种，从分类清单到自由散文。在Batch Method一例中，其4种驱动力都分别放在一个独立的段落中，并使用了项目符号以使其足够清楚。其他散文格式的描述强调了不同驱动力之间的冲突，首先提出一个假设或期望的结果，然后常使用“但是”和“然而”这样的转折词将其与其他存在冲突的假设相对比。

一个简单的解决方案描述可以确保读者对解决方案的基本结构有所了解，虽然可能还是需要继续阅读相关的细节。不应该让读者在读过之后留有这样的疑问：“整个解决方案细节的哪些部分算是模式呢？”读者应该很容易识别出哪些是用来解决问题的本质部分，哪些属于支撑、连接和扩展的细节。在对Batch Method的描述中，解决方案的陈述紧接在驱动力列表之后。这个陈述是直截了当的：不是讨论解决方案能够提供什么价值，而是简单扼要地描述了该方案的精髓。解决方案的陈述可以被解读为对问题总结时提出问题的直接回答。在其他格式中，如果问题以陈述的格式展现，解决方案的陈述常常会习惯性地以“因此”开始。

在解决方案陈述之后，模式描述可以为模式实现的各种因素提供更为详细的表述。这段表述可以混合散文、图表和代码等。解决方案的描述可以进一步结构化，为相同问题领域的一个编目下的模式提供相似的格式，如面向对象设计。举例来说，一个面向对象设计模式的解决方案可能从一个简单的解决方案陈述开始，紧接着一两段细节描述，然后可能紧接着一幅类图或是一幅时序图，随后，可能跟着一段描述性的代码片段，最后以描述常见变化的列表作为总结。有关组织的模式开头可能是差不多的，但是图示部分往往是非正式的，只要能展示出不同部门的职责、人与人以及不同部门角色之间的交互即可。

模式解决方案的格式与深度很大程度上依赖于其类型及其面向的受众。至少，读者应该能很好地掌握模式的主要原则与元素。理想情况下，他们应该对模式背后的流程有一定的理解，而不是仅知道该解决方案产生的作用：模式怎样应用于实践环境。

再次强调第1章的观点，无论好坏，任何决定必定伴随着相应的结果。没有澄清其解决方案的结果的模式非但不能对设计者考虑的问题进行回答，反而会带来更多的问题。若一种模式声称只有好处而不会带来任何不利因素，无异于通过广告而不是通过工程方法来设计。就像一个“快速致富”的骗局，相信这种仅有好消息的广告，等待你的只能是误用和失望。因此，模式的描述应该将其使用的利与弊都展示出来，让读者能够根据自己的情况来衡量模式的不利因素及其提供的好处。

模式的本质表明，它们应该是基于真实案例的。很多模式格式都会展示案例，有的是采用单独的章节，有的则是将案例与叙述性的从问题提出到解决方案的影响结果的描述交织在一起。

模式的经验性特质同时也对描述的格式提出了另一个问题，即可识别的质量。识别和文档化模式的普遍实践常常关注于证明实践，但有时候我们希望能够将好的模式与较差的模式进行比较。能让读者清楚哪种好哪种不好是非常重要的，例如对“不完整”(dysfunctional) [Hen03a]这种较少听说的模式进行显式标注。其实当我们面对那些耳熟能详的模式时，也不是所有的模式都是平等的，我们也会希望明确地表示我们对一种模式的信赖程度。Christopher Alexander采用的描述格式是对每种模式进行简单的打分[AIS77]。这种风格被很多模式作者所使用，我们在《面向模式的软件架构(卷4)》[POSA4]中也采取了这种格式。

我们所使用的格式以模式名开头，模式名可以标上一颗星、两颗星或者没有星。星的数目表明了我们对这个模式成熟性的确认度。两颗星表示我们非常确认该模式针对其所在问题域中的一个名副其实的问题，而且该模式建议的实现方案或者相应的变体方案对于有效解决问题至关重要。一颗星表示该模式描述的是一个名副其实的问题，其方案是一个不错的解决方案，但是它还不够成熟。没有星表示我们时常会发现该模式所描述的问题，而且其解决方案也是有用的，但是该模式需要很大的改进才能达到一颗星或两颗星的标准。没有星可能还表示在这种场合下有更好的备选模式。

星级的评定也不一定就是模式的绝对等级：应该将模式放到其将要面对的具体上下文环境去综合考虑。例如，在POSA4中，Blackboard模式就是模式没有标星的：从分布式计算的整体环境来考虑，这种评定可能是有道理的，但如果是在处理非确定性问题的模式集合中，该模式应该得到一个更高一些的等级。类似地，Reflection模式被标记为一星，但如果上下文是构建软件工具的模式集合，它应该被标为两颗星。

图3-1总结了模式格式的基本要素及其目的。

3.4 细节

虽然任何模式描述都应该涉及上面提到的核心元素，一些附加的细节往往能对模式在何处应用以及

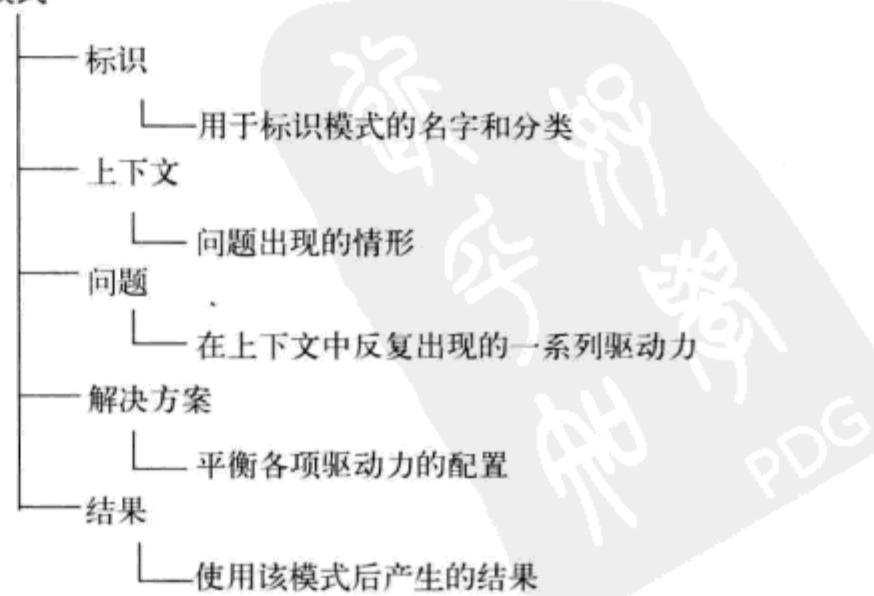


图 3-1

怎样应用模式提供有益的指导。模式描述写作的细节程度可以根据情况进行相应的变化。模式的描述究竟应该是简短的素描还是详细的探索过程？是否应该配上图示？如果需要，该配何种图示？该有多少这样的图示？相关的代码、案例研究和实例呢？答案是否合适取决于该模式的类型及其目标读者。

在某些情况下，简短格式的描述可能有卖关子之嫌，告诉你一个好的设计，却不提供关键的细节。但在另外一些情况下，它却是理想的格式。其作者理解简洁的价值，简洁性和准确性可能正是读者在思考他们的系统时所需要的。类似地，在某些情况下，过度细节化的模式描述可能容易让人分心，在追求精确的同时可能会带来误解，或者读者可能会按字面诠释。在另一些情况下，读者可能正需要这些细节，让他们增强信心，并能够正确地应用模式。

模式只是展示了设计空间的一个层面，这意味着它不能——或者不应该——覆盖所有可能出现的偶发问题和因素。模式会涉及其他模式，而不是重复其他模式的内容和概念，模式的解决方案或上下文可以引用其他模式来帮助读者了解背景，或对进一步阅读和实施细节提供建议。

3.4.1 案例

模式描述中的案例会影响该模式的受众。无论模式描述的其他部分采取何种深度和风格，其使用的案例确定了入门的级别。因此，面向新手的描述应该避免采用行业级的案例。详细、晦涩难懂的案例可以用来帮助专家理解模式，但在面向初学者的模式描述中，受众却不是专家。如果一个例子被与核心模式没有直接关系的偶然复杂性和附带的细节所主导，即使是专家级的读者也可能被误导。

虽然有些简短的模式描述可能不包含例子，但很多模式都包含某种形式的案例。如果模式描述的一项限制是它的长度——比如一页纸描述一个模式，放弃案例讨论可以把更多的空间留给探讨驱动力和解决方案细节。对于不同的模式类型，情况也不一样：以代码为中心的模式将一个具体的问题和解决方案编纂在一起是非常合适的，但是对一个组织型模式来说，非要编造出包括Alice、Bob和Charlie的故事有时会显得有点矫情。在前一种情形中，例子可以为读者提供澄清问题的资源，没有例子，读者就只能通过猜测来领会了。在后一种情形中，例子可能会使读者偏离模式所要传达的内容，没有这个例子可能反而不会出现这样的问题。

有些模式甚至包含了不止一个例子[Hen01a]，并且这是有充分的原因的。在阅读和使用模式时常犯的一个错误是假设例子就是模式本身，或者以为模式只能在与例子直接相关的软件中使用。相对于与人和流程相关的模式来说，有关设计的模式更容易出现这样的问题。

一个模式所涉及的设计空间往往不可能由一个案例完全覆盖。两个或更多个案例可以提供更多的信息，让我们从不同的视角来看待问题-解决方案空间。类似地，提供一个反例——模式误用的常见案例——也能帮助理清模式的界限所在。

在2.4节中，我们强调了在学习模式时案例所起到的作用。有两种类型的例子可供模式描述使用：展示模式已知应用的领域案例，以及以展示为目的创建的教学案例。当然，模式描述并不一定要二选一：它可以包含两种类型的案例，引用并对领域案例作出总结，详细阐述教学案例。这两种类型的案例也不一定非要有所区别。举例来说，有些领域案例往往足够简练直接，可以直

接用于教学。例如，在*Fearless Change: Patterns for Introducing New Ideas* [MaRi04]中，向组织引入新思想的经验报告和已知应用足够简洁完整，可以将它们全部包含在模式描述中。

在代码相关的模式中，盲目地追求完整性，可能会引入过多细节，因此，简短的例子可能更为合适。但是，这种“简洁”并不是说案例仅为着演示的目的来构建：POSA2中的许多代码例子都是从ACE[SH02] [SH03]的代码中抽取而来，简化并聚焦于教学目的，但还是保留了它们的领域特征。

3.4.2 图示

图表也许是模式描述元素中用的最多也错的最多一个了。图表及是否可以采用图表的问题我们已经在1.7节中进行了详细探讨。这里我们简单回顾一下这个问题，不过现在我们将把注意力集中于模式的描述格式而不是模式的概念上。

类图对于关注设计中类之间的静态关系是很有意义的。但是，当设计着重于贯穿一个简单模型的动态编排和对象之间互动的时候，类图就不那么有用了。当然，当一个解决方案不涉及对象时，面向对象的标记方法可能就无关紧要了，应该使用其他有助于展现概念的标注方法。

得到公认的标识方法（如UML）能以读者较为熟悉的术语来描述一个设计。有时候，一套由作者控制的专门符合特定解决方案特征的标识方法可能更为可取（参见1.7节中关于“Bizarro”标识方法的简短讨论）。当需要解决方案的结构图时，UML可能会由于不必要的精确导致误解。

当我们需要某种精确度时，正式的图表对很多读者来说就很有价值。正如我们在第2章中讨论到的，记住这些图表是模式的展示而不是模式本身这一点非常重要。在讨论Composite模式的特定应用时，John Vlissides提出了类似的观点[Vlis98b]。

“模式的结构”这一节展示了一幅标准Composite类结构，它采用了经过修改的OMT方法绘制。这里的“标准”仅仅是指它代表我们（GoF）所观察到的最常见的结构。但是它并不是绝对的，考虑到具体的设计和实现上的权衡，它的接口是可以变化的。

回顾第0章中讨论的模式应用的一些常见问题：模式不是图示，图示只是图示。

3.4.3 <code>...</code>

代码是否适合用在模式描述中，取决于该模式是否与编程相关，或者说与组织结构无关。与展示C++代码的模式描述相比，没有展示任何C++代码的C++实践相关的模式描述可能更没有说服力。因为代码是模式实质的一个重要组成部分，所以模式在其表现形式上也应该体现这一点。

另一方面，又有“细节太多”这样一种情况：一些模式描述迷失在代码之中。在复杂的中间件模式中，简化或忽略产品代码中的次要细节可能对读者更有好处。如果简洁性是某个模式描述的灵魂所在，删除某些代码不失为更好的选择，即使该模式是以代码为中心的。但是，就算是在这些案例中，包含一段简短的代码或者伪代码仍然可能会为模式增加一些分量和有效性。例如，Batch Method的描述采用具体案例中接口的代码片段，比如说远程目录。

```
interface RemoteDictionary ...
```

```

{
    Object[] get(Object[] keys);
    void put(Object[] keys, Object[] values);
    ...
}

```

创作以代码为核心的模式的作者们也应该记住：他们所采用的代码例子的质量也是读者用来判断模式的一部分手段。否则，即使是相对正确、合理的模式，也可能因为其代码例子的质量不合读者的期望而被忽视。一些早已被文档化的模式实际上已经被抛弃了，因为细心的读者注意到，这些模式描述中展示的代码恰恰暴露了模式自身的缺陷。

3.5 鸟瞰图

为了在大段细节之前提供一个容易理解的简介，许多模式都以概述（thumbnail）作为开头，这与戏剧中的开场白和歌剧中的序曲非常相似。概述是模式整体的一个精简视图，通常包含了问题的介绍——也许是提出一个问题，紧接着是一个解决方案的陈述。另外，也可能包含一个提示性的结构图。这些概述有时候也被叫做托盘（patlet）。

3.5.1 从金字塔到托盘

模式描述的各个部分有时候是以倒金字塔的风格来呈现的，最重要的信息最先进行简单的介绍，然后再介绍具体细节。这种方法——由于常常用在报纸中，也被称为“报纸风格”——为读者在文章的开头提供一个快速介绍。概述的问题部分和解决方案部分往往是完整模式描述的问题及解决方案的直接引用。例如，下面是Batch Method的概述。

如果访问成本很高并且可能失败，怎样才能高效、不中断地在聚合对象上执行批量操作呢？对于给定的操作，定义一个可在聚合上反复执行该操作的方法。

有时候我们会把问题和解决方案部分显式标注出来，使其更像是完整模式描述的缩略版。

问题：如果访问成本很高并且可能失败，怎样才能高效、不中断地在聚合对象上执行批量操作呢？

解决方案：对于给定的操作，定义一个可在聚合上反复执行该操作的方法。

从Alexander到POSA4，人们常常以“因此”一词来引入模式的解决方案。通过将概述中的问题部分以陈述句而不是提问的形式表达，“因此”可以在概述扮演类似的分割角色。

读取聚合对象的成本很高并且可能失败，但对聚合对象执行批量操作需要保持高效并且不间断。因此，对于给定的操作，定义一个可在聚合上反复执行该操作的方法。

3.5.2 模式骨架

概述形式也可以用于一种模式或一组模式的初始介绍。虽然忽略上下文、驱动力、结果以及其他细节意味着我们并没有从我们认为重要的所有角度来展示模式，但它毕竟包含了问题及其解

解决方案，从某种意义上来说模式也是完整的，覆盖了模式的关键，提供了进一步阐述的空间。这种初期草拟技术的一个例子是Web 2.0设计模式[ORei05]。下面是取自该集合的Software Above the Level of a Single Device模式的描述。

个人电脑不再是互联网应用的唯一接入设备，局限于单一设备的应用往往不如联网的应用有价值。因此：从一开始就将你的应用设计成横跨手持设备、个人电脑和互联网服务器的集成服务。

另一个精简描述的例子描述了Null Object模式：

3

如果

- 一个对象引用可能为null；
- 在每次使用前，该引用必须经过检测；
- 如果检测结果为null，就什么也不做，或者将其赋上合适的默认值；

那么

- 提供一个继承自所引用对象的类；
- 实现它所有的方法，这些方法可以什么也不做或者提供默认结果；
- 在对象引用为null的地方使用该类的一个实例。

这种基于规则的描述在更完整的文档[Hen02a]中作为概述放到前面，但它最初的时候是用在某个会议的幻灯片中[Hen97]。模式描述用在什么场合对格式和出现多少细节有很大影响。在该例中，在幻灯片上展示模式这一限制鼓励我们采取一种类似托盘的方式来实现一段简短但精确的文本描述和简单的图示。

除了单个模式的展示[Hen97]和概述[Wake95] [Hen02a]外，基于规则的格式还用于较大集合中的模式总结。例如，下面是Build Prototypes模式[CoHa04]的一个总结：

如果早期需求不经过测试很难验证，那么：建立一个用于帮助澄清需求和评估风险的原型。

也可以将基于规则的格式变化为“因此”句式，就如下面Architect Also Implements模式总结中使用的这样。

如果一名架构师待在象牙塔里，脱离了现实，而我们需要在实践中调整高层视图。因此，确保架构师确实参加了每日的实现工作。

基于规则的格式将问题与建议的解决方案清晰地分离开，可以在一个简单的结构性描述中包含更多细节。例如，下面是采用这种风格的Batch Method模式的另一个总结。

如果在一个聚合对象上进行单独的方法访问成本很高，并且可能失败，而我们又需要反复地访问该聚合对象，并且要能够保持高效而无中断，那么，对于给定的重复访问，定义一个可在该聚合上反复执行操作的方法，并且通过集合的方式将参数集中传入，结果也通过类似的方式传出。

3.5.3 总结意图

总结模式的另一种方法是简要记录模式的意图，即我们提出该模式的目的所在。在对模式的意图进行文档化的时候，我们关注的核心是问题，而不是解决方案。以下是Iterator模式的意图[GoF95]。

提供一种顺序访问聚合对象中元素的方法，而又不公开该对象的内部表示。

这种方法的一个好处就是它可以很容易识别出具有相同目的的多个模式，这个话题我们将在第5章中详细探讨。举例来说，Batch Method模式可以说与Iterator模式具有相同的目的。同时，这种方法的一个潜在问题是很难明确区分在某些方面相似但在其他方面上却有不同权衡的模式。我们认为模式的总结描述格式应该能够帮助我们区分不同的模式。就这个例子来说，经过优化和调整以包含更多问题上下文的表达方法更能清楚地表明Batch Method模式的意图。

提供一种高效的方法以在分布式环境中访问聚合对象中的元素。

3.5.4 模式抽象

我们也可以提供一种更传统的模式抽象以概括其动机和解决方案。这种方法在《面向模式的软件架构》各卷中均有使用：在[POSA1]、[POSA2]及[POSA3]中，抽象描述被用于每种模式的开始，一眼就能看得出来；而在[POSA4]中，抽象描述被用在每组模式之前的介绍材料中。在本卷中，抽象描述会出现在附录B中。下面是[POSA4]中出现的Batch Method模式的一个抽象描述。

Batch Method模式将对聚合对象中元素的重复访问合并在一起，以减少多次访问单个元素的开销。

有意思的是，《设计模式》中许多模式意图的说明都不仅仅是陈述这么简单，而更接近于POSA中的抽象。比如，下面是Template Method模式的“意图”[GoF95]。

在一个操作（operation）中定义一个算法的骨架，将一些步骤推迟到其子类中。Template Method允许子类重新定义算法的一些步骤，而无需改变算法的结构。

这样的一个描述既有解决方案的结构，也有应用该模式的动机，而不仅仅是模式的意图。我们这里质疑的是“意图”这个术语，而不是这种描述。这种描述的价值是很明显的。

3.6 不同的格式

记录模式的格式是多种多样的：GoF和POSA采用的高度结构化、详细的风格[GoF95] [POSA1]，Jim Coplien引入的简短的关注精髓的格式[Cope95]，Ward Cunningham所使用的叙述性的“波特兰”风格[Cun95]，Christopher Alexander使用的描述格式[AIS77]，甚至一些比较少见的风格，如Alistair Cockburn开发的一种描述风格，这种方法引入了模式“过度使用”的视角[Coc97]，让人回忆起第1章中医疗诊断的隐喻。

3.6.1 演变

例如，第1章最终所得到的Batch Method模式的描述，通过为每个部分添加相应的题头，并经过必要的重新组织，很容易转换为Coplien的格式。

名称：Batch Method

问题：如果访问成本很高并且可能失败，怎样才能高效地、不中断地在聚合对象上执行批量操作呢？

上下文：在分布式或者并发对象系统中，聚合对象的客户端可能需要在聚合上执行批量的操作。比如，客户端可能需要取回所有符合具有某个特定属性的元素。如果对聚合的访问成本非常高，因为它相对于客户端是远程的或者是由多个线程共享的，在一个循环中单独访问每个元素——比如通过索引、键或者Iterator——都可能引入严重的性能问题，比如往返时间、阻塞延迟或者上下文切换开销。

3

驱动力：

- 聚合对象是由并发环境中的多个客户端共享的，可能是在本地的多线程环境，也可能是跨网络的分布式环境，我们可以在一次方法调用中对其进行同步封装，但是对于在循环中重复的调用却不合适。
- 阻塞、同步和线程管理的开销必须在每次跨线程或者跨进程的访问中考虑进去。类似地，其他的针对每次调用的操作（比如授权）会进一步降低系统的性能。
- 如果聚合对象相对于客户端是远程的，那么每次访问还会带来延迟、抖动，并占用网络带宽。
- 分布式系统可能会出现部分失败的情况，这时候客户端可能还活着，但是服务器却已经崩溃了。在迭代过程中进行远程访问，可以为每次循环引入潜在的失败点，这可能导致客户端面临只完成部分遍历的情况，比如不完整的状态快照或者不完整的更新。

解决方案：对于给定的操作，定义一个可以在聚合上反复执行该操作的方法。

每个Batch Method都是聚合对象的接口的一部分；或者是作为聚合类型的Explicit Interface暴露出来，或者是作为一个更窄的混入Explicit Interface——仅仅定义调用重复操作的方法。Batch Method被声明为可以接受每次执行操作所需要的所有参数，比如通过数组或者集合，结果的返回也采用类似的方式。这样我们就不需要在循环中直接挨个访问聚合元素，客户端直接调用Batch Method就可以在一次调用中访问多个元素。Batch Method可能由聚合对象的基础类直接实现，也可以通过Object Adapter方式间接实现，这样就不需要对聚合对象的类做任何修改了。

基本原理：通过Batch Method，我们将重复的操作转化为一个数据结构，这样我们就不需要在客户端进行循环，而只需要在方法调用之前和结束之后通过循环做一些准备和后续工作。于是，对聚合的访问成本降低为一次访问或者几次“较大型”访问。在分布式系统中，这种“压缩”机制可以大幅度提高性能，并减少网络出错的概率，节省宝贵的带宽。

结果：使用Batch Method会提高单次访问的成本，但是批量访问的总体成本还是降低了。

这样的访问也可以在方法调用中配合使用适当的同步技术。每次调用可以是事务性的，即或者完全成功或者完全失败，不存在部分失败的情况。

使用Batch Method的代价是在准备和后续阶段要做更多的工作，并且需要引入中间数据结构来传递参数和接收返回结果。网络、并发和其他针对每次调用的开销越大，使用Batch Method就越合适。

通过添加必要的章节、图表、图示以及CRC卡，上面的描述也可以转化为POSA中所采用的格式。也可以就让它保持第1章中简单有序的格式。这些描述格式中的任意一种都可以用于文档化模式。

在某些情况下，正如刚刚展示的例子一样，从一种模式描述格式转换到另一种可以仅仅是简单的文本转换，但更常见的情况是，格式的元素和风格不一致，需要进行更多转换。用项目列表来展示驱动力的模式，当要把它转换为叙述格式时，需要的绝不仅仅是将列表条目粘贴到一起那么简单。像GoF所使用的格式那样没有显式标注驱动力，比明确指出这些驱动力需要做更多的工作。用精简、高层次视角编写的模式需要进一步的细节和更多工作来充实成较长的、较精确的描述格式。以较长格式编写的模式需要很好的编辑和总结才能以较短的格式来表达。

对于最后一条观点，我们有一个例子：《面向模式的软件架构（卷2）》[POSA2]中的所有模式都包含在《面向模式的软件架构（卷4）》[POSA4]中。虽然这两本书的厚度相当，但POSA2包含了17种模式的描述，而POSA4却包含了114种模式的描述。POSA2中的模式描述通常占到20~30页——包含了代码、图示、表格和实现选择等细节考虑，而POSA4中大多数模式的描述只占两页左右，少数模式的描述有3页篇幅。

两本书的不同着重点跟森林与树木的对比关系有些相似：POSA4提供了模式之间广泛联系的视图，而POSA2提供了一个近距离的细节视图——这些差异造成了长度和深度上的不同。在把POSA2中的模式集合到POSA4中时，根据其重要性对这些模式的长度进行缩减，而不只是去掉简单明显的细节（如代码和图示）的问题。转换后的结果应该跟原来的描述一样完整可信。

在一些例子中，我们可以很容易地从一种格式转换到另一种，正如前面演示的将Batch Method的描述转换到Coplien格式的案例。但是，这些例子不过是一些例外而非普遍情况。大多数情况下，从一种格式转移到另一种格式常常是对模式的重写和再挖掘，是真正的创造性活动。

模式格式之间的转换几乎没有能自动实现的情况。虽然读者会意识到同一模式的两种格式之间的共性，但这种规律并不能支持有效的自动化。所谓的模式标记语言的想法（即使使用一个简单的框架转换模式的表达格式）基本上是不切实际的。这种方式只能用于简单的提取，比如抽取问题陈述的第一句话（或最后一句话）和解决方案陈述的第一句话来组成模式的缩略图。

3.6.2 选择

最适合描述特定模式的格式并不能简单地由客观规则来决定。我们最多能够评估一些建议和考虑因素，但是主观的基于经验的权衡还是会起到主导作用。模式的格式很大程度上取决于模式作者想要通过模式描述来达到的意图，以及他们希望针对的目标受众。尽管有可能会显得比较笼

统,但我们还是总结了一些启发式的原则,以帮助模式作者决定在他们的模式描述中应用哪种具体格式。

如果模式描述的目标是展现模式的精髓,或者引起读者对一个问题的注意并总结其解决方案——其读者是项目经理或项目领导,往往最好采用诸如Coplien格式这样的精简格式,而不是深入讨论模式实现细节的格式。我们在第1章中对Batch Method模式进行了描述,并在本章中对其进行了回顾,它是这种格式的一个很好的例子。

与长格式相比,精简格式更好地强调了模式根本的、稳定的特征。通用模式的实现细节随着新的编程范式、语言和特性的出现会变得不合时宜。相反,优秀模式的核心精髓往往是永恒的,虽然不一定是一成不变的。更务实地讲,精简的、关注精髓的描述格式在浏览现有模式,寻找某种模式以解决特定问题时非常有用。当然,较长模式描述的缩略图(我们可以将其看做是非常精简的格式)也能担当这种角色。

但是,当模式作者希望引导开发人员使用某种语言来实现所介绍的模式时,精简格式往往是不够的。在这种情况下,我们需要模式结构及其动态特征的描述、其具体实现的方针准则以及具体的案例。换句话说,我们既要捕捉到模式的精髓,也要抓住模式的细节。

我们在《面向模式的软件架构》系列的前三卷[POSA1] [POSA2] [POSA3]中定义和使用的模式描述格式就是要满足这种需求的:这种方式让我们可以描绘出模式的“概览图”,详细提供模式的具体结构和动态特性,并指导其实现。希望了解某一模式核心观点的读者只需要阅读模式的“上下文”、模式所解决的问题及解决方案部分。这三个部分提供了模式的核心,其他部分提供的都是附加信息。“结构”、“动态特性”和“结果”这几个部分概览了模式外观。“例子”、“实现”、“已解决的例子”、“变体”、“已知应用”和“参见”部分提供了在生产中应用和实现模式的具体细节。

但总的来说,模式描述格式并不是影响其是否能够成功沟通的唯一因素。难于阅读、过于沉闷和正式的模式描述很可能打消读者的阅读兴趣。如果模式过早地引入复杂因素,或者所使用的例子中的某些细节使读者分心直至喧宾夺主,又或者如果不是相关方面的专家就无法很好地理解,这样格式的模式将无法吸引广大读者。模式描述应该是准确而翔实的,读起来也应该感觉有趣——就像讲故事一样。

3.7 风格与实质 (Redux)

模式描述使用的格式无论对于模式作者还是读者来说都非常重要。它定义了一种展现模式内容及其观点和倾向的工具。因此,虽然从某种意义上说,模式格式的选择是主观的,但从另一种意义上说,一个好的模式的精髓是独立于其描述的,但描述却决定了人们对该模式的接受程度。

为了避免我们过于专注模式格式这个问题,我们回忆一下1.12节中提到的。

为了避免误解,这里需要澄清一点:仅仅通过改进描述方式是不可能让一个普通的解决方案自动变成模式的。模式不是靠语言大师创造出来的。只有具有了上面我们讨论的各种属性,问题之解决方案才可能成为一个好的或者完整的模式。

同样的观点也适用于本章。格式是重要的——可以说非常重要，但它只是将模式本身的特质表现出来，而不是越俎代庖。如果不能满足以下条件，我们最后得到的将不过是一些虚假模式（hollow pattern）的完美描述而已[Ale79]。

所以，一种模式必须满足以下两个条件，才能判定它是好模式。

(1) **其解决的问题是真实存在的。**这意味着我们将问题表述为在实际上下文中真实出现的驱动力之间冲突的结果，而这种冲突通常无法在该上下文内得到解决。这是一个经验问题。

(2) **它解决了这个问题。**这意味着当在该上下文中采用模式所提供的方案时，这种冲突可以得到解决，而没有任何副作用。这也是一个经验问题。

总之，当模式的内容真地描述了某种实质的内容时，模式的格式才真正具有意义！因此，我们非常希望能将合适的格式、良好的描述和完整的模式结合起来。

第二部分

模式之间的关系



摄于挪威奥斯陆峡湾的“天涯海角”
版权归Mai Skou Nielssen所有

本书第二部分将视角转向模式之外探究模式之间的关系。模式之间的关系丰富多样，有时某个模式会成为其他模式的备用模式，有时和别的模式一起使用成为其他模式的补充，有时又和一系列模式绑定在一起，成为一个紧密相连的模式组。模式亦可组成模式序列，在叙述流中一个接一个地使用，这也扩展了模式在设计中的应用。最后要说明的一点是，把模式组合为模式集合会导致一个共同的问题。

没有哪个模式是一座孤岛，它们总是被组合在一起使用，模式之间充满了“生机”。模式之间的多种关系有助于实现、增强、巩固和扩展单个模式的功能，因此，相对与单个模式特定的关注点和定位，模式之间存在的多种多样的关系使得单个模式在具体的模式排列或软件设计中更具

“杀伤力”。在这一部分中，我们将探究模式之间各种各样的关系，讨论如何利用这些关系在具体的软件项目中富有成效地应用模式。

第4章描述了一个简单但令人印象深刻的设计实验，探究了相比于孤立地使用模式，如何利用模式之间的关系来创建质量更高、整体性更强的基于模式的设计。

第5章讲解了一个模式如何能使得基于另外一个模式的设计更为完善。这一章还探究了两个或更多的模式如何通过对比和竞争来使得设计流程更为完善。有时，当相互竞争的模式组合在一起使用时，互补性的两个方面（结构与流程，协作与竞争）能够融为一体。

第6章讨论了如何将一组模式紧密组合在一起形成一个新的模式。与单个模式相比，组合在一起的模式可以从更粗粒度和范围上帮助解决问题。

第7章探讨了模式如何为特定软件的设计流程提供支撑。模式故事提供了一种表述方式，用来描述如何具体应用（或可能应用）模式来构建特定的系统或特性。模式序列将这种描述方式进行归纳概括，提炼出实际的模式应用序列，形成更为基础的描述方式，这样可以在其他地方考虑使用同一模式序列。

第8章描述了模式集合与编目，模式集合可以充当模式仓库。我们讨论了管理模式集合的不同体系架构。同时，我们讨论了问题框架技巧，探讨了在何种环境和问题中应用模式。我们还讲解了模式结构与关系相关的符号学。

需要着重强调的一点是，这一部分讲解的诸多概念成熟度各异，在软件模式社区中的认可程度也不尽相同。一些概念，如模式互补与模式复合，已经人尽皆知、广为认可。另外一些概念，如模式故事与模式序列，尽管在实践中已有应用，但相关研究还不够深入，亦未得到广泛认可。类似地，模式编目和集合已成为流行的、广为认可的模式管理方式，但问题框架和模式符号学是模式领域最近才出现的产物，因此在模式社区中还没有广泛应用。

我们的目标是尽我们所能为读者全面地讲解最新的模式关系。因此，在这一部分中，无论这些概念是新近出现，还是日臻成熟，无论是鲜有应用，还是正大力推广，我们都对其做了讲解和讨论。我们也坚信这一部分涉及的每个概念都具有独特的价值，并且将这些概念集成在一起能够增强并巩固单个概念的特性，而不是产生冲突。在我们讨论的概念中，如果缺失了任何一个，模式之间的关系图都不会完整。

同第一部分一样，尽管我们的示例只是把重点放在设计模式上，但我们所有的关注点、见解看法及表述都不是针对特定类型模式的。

没有哪个人是一座孤岛，自成一体；每个人都是大陆的一部分，属于那个整体。

——John Donne

设计中用到的各种模式是一个个孤立的小岛，还是环环相扣的交织整体？哪种观点更自然呢？在著作和实践中，常见前一种观点。本章中我们将用一个简单的设计实验，合乎逻辑地推导出此观点的最终结论，探究它会导致怎样的后果。实验结果将进一步证明：设计模式不仅定义了结构部件，还定义了各结构部件的角色。

4.1 模式的联系

很多模式作者和有经验的模式使用者都承认：模式很少孤立存在。每种模式一般都与其他模式有关联。当人们有意识地应用模式，或从现有软件系统中发掘模式的时候，这种关联性体现得更为明显。

例如，GoF的名著《设计模式》[GoF95]用一幅图画出了书中所述23个模式之间的依赖关系，并且在具体介绍每个模式的时候，用“相关模式”一节内容详细说明其依赖关系。在《面向模式的软件架构（卷1）》[POSA1]中，我们也讨论过，一个模式可能从其他模式改良而来（refinement），也可能是某个模式的变形（variant），还可以与其他模式组合使用。《面向模式的软件架构》系列的每一卷在介绍任何一个模式的时候，都会参照这几个方向讲解有哪些模式与其相关联。

本系列书对模式关系的最后讨论见《面向模式的软件架构（卷4）》[POSA4]。POSA4中介绍的114个模式全都引用了其他模式，这些模式在构建分布式软件系统的上下文中使用或者实现时非常有用。例如，MVC模式的描述中引用了12个模式：Domain Object、Template View、Transform View、Page Controller、Front Controller、Command、Command Processor、Application Controller、Chain Of Responsibility、Wrapper Facade、Data Transfer Object和Observer。在POSA4中，还讨论了POSA1没有涉及的另一种关联形式：一种模式可以替代另一种模式。

在各种有关模式的著作中，模式之间的关联及互相依赖的类型已有广泛讨论。我们也在本章及后续两章中进行阐述。因此，本节将重点讨论：为什么模式之间的关联及相互依赖性对理解和使用这些模式来说至关重要。

4.2 设计实验：将模式作为孤岛

为了阐明模式之间的关系为什么是重要的，我们将进行一个简单的设计实验。实验内容是开发一个可扩展的用于处理请求的框架，将客户端发出的服务请求转译成应用中具体的方法调用。在这个实验中，我们假设模式之间根本没有关联，不允许它们具体实现之间的任何整合。换句话说，我们将每个模式都作为一个孤岛，是一个完全自我包含的整体，其角色与其他模式的角色没有交叉。

当开发这个用于处理请求的框架时，我们首先要解决的设计问题就是：将来自客户端（也许是一个真正的用户，也可能是某个计算系统）的请求“对象化”。Command模式可以解决这个设计问题，所以我们用它开始我们的设计之旅。抽象类Command声明了一些抽象方法来执行这些客户请求。从Command类派生出来的一些ConcreteCommand类分别实现该应用所处理的具体命令，如图4-1所示。

当客户端发出一个具体请求时，我们实例化一个相应的ConcreteCommand对象，并调用它从抽象类Command中继承的某个方法。然后，这个ConcreteCommand对象执行应用的这个请求操作，如果有返回值的话，将其返回给客户端。

由于多个客户端会各自独立地调用不同的Command对象，所以在一个居中的组件中协调一般的命令处理过程会较为有利。Command Processor模式提供了这样的结构。客户端将一个由其创建的具体命令传递给一个专职的命令处理器（command processor）组件，接下来的事情就由命令处理器组件来处理。

将Command Processor模式整合进现有的结构非常简单，不会引入明显的偶然复杂性（accidental complexity）：将CommandProcessor类放在客户与Command类之间即可，如图4-2所示。

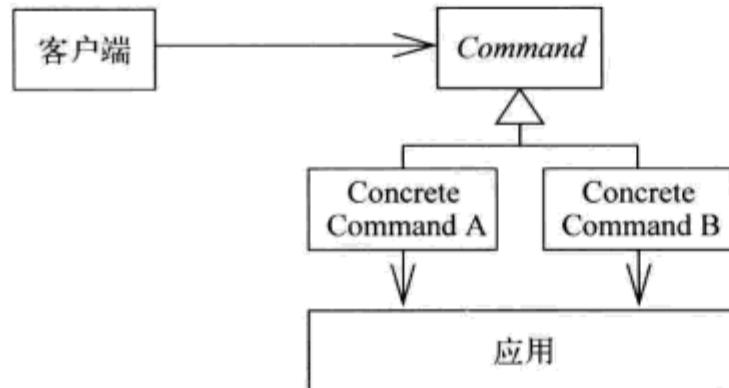


图 4-1

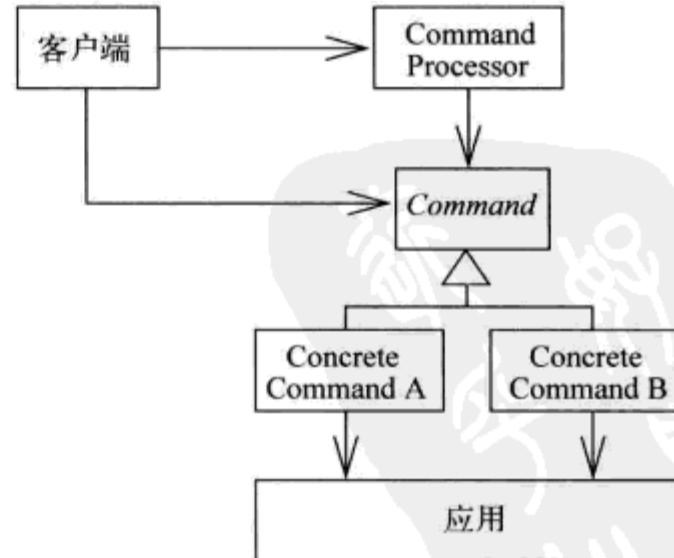


图 4-2

很多系统也支持撤销响应请求时执行的操作。实现这样一个回滚机制需要在执行请求之前，先将整个应用或某部分的状态快照保存起来，当用户以后取消请求时，再恢复到这个状态。Memento模式的实现机制是这样的，memento组件保存了另一个组件（称为originator）的当前状

态副本，caretaker组件负责创建memento组件，并且长期持有它，一旦需要，再将其交回给originator。undo机制是促成Memento模式的一种常见情形[GoF95]，我们现在正好就要设计一个undo机制。

从概念上讲，具体命令正好承担了caretaker的角色，它在执行请求之前创建memento，并在回滚具体命令时维护这些memento。当某个具体命令的undo操作被调用时，它可以将其memento传回给系统。这个memento在我们的设计中是一个新组件，它的实例捕获系统的当前状态，而系统本身相当于originator。

然而，由于实验的目的（模式和它们所承担的角色应该是严格独立的），我们不会将caretaker的职责补充到具体命令类中。事实上，我们会引入一个独立的Caretaker类，并将其与命令类Command联系起来，以便ConcreteCommand对象可以使用该类的实例来创建、维护并保存Memento的实例。这种严格分割模式职责的直接后果是设计看上去有点奇怪，而且在“撤销具体命令”这一功能领域引入了不必要的结构及逻辑复杂性，如图4-3所示。

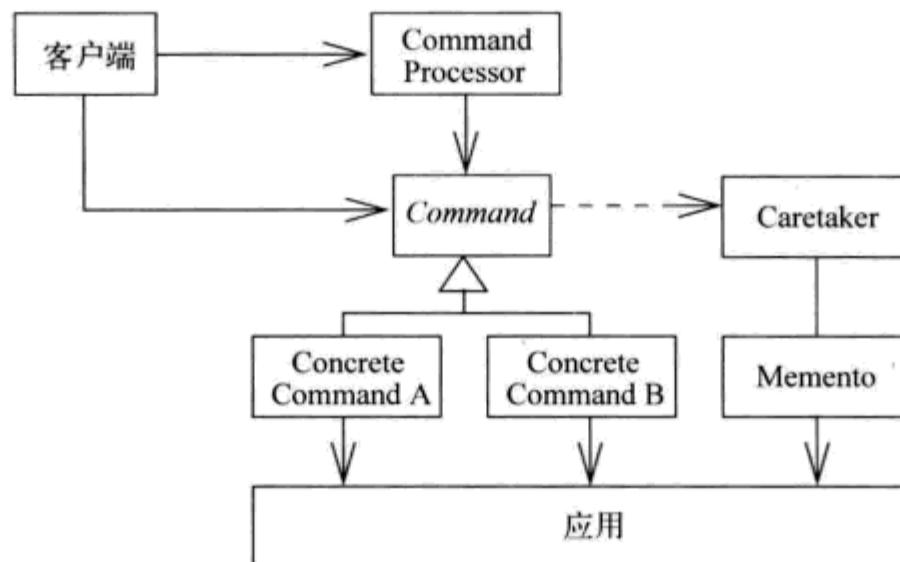


图 4-3

接下来，我们需要实现的需求是：定义某种机制，使我们的框架可以记录请求。使用该框架的开发人员对于这个日志有不同的需求，有些希望记录每一个接收到的请求，有些只希望记录某类特殊的请求，还有一些人根本不需要日志。因此，我们现在要解决的另一个设计问题是：以一种灵活而且高效的方式满足不同用户的日志需求。

解决这个问题的一种候选模式是Strategy模式，它支持在系统中对不同算法的封装与替换，从而达到可拔插的特性。然而，将Strategy模式集成进来会导致另一个冲突。理想情况下，CommandProcessor应该提供或调用日志服务（logging service）。然而由于我们不允许同一组件承担来自不同模式的不同角色，日志服务的不变部分只能在CommandProcessor之外实现，即在Strategy模式的context组件中实现。

因此，Strategy模式的实现如下：CommandProcessor类将接收到的ConcreteCommand对象传递给LoggingContext对象，而LoggingContext对象承担Strategy模式中的上下文角色。该对象实现了日志服务中的不可变部分，而将特定于客户日志操作委托给ConcreteLogging-

Strategy对象去计算，而ConcreteLoggingStrategy对象承担Strategy模式中具体策略的角色。Logging抽象类为所有的ConcreteLoggingStrategy类提供了一个共同的协议，因此它们可以互换而不必修改LoggingContext。这种设计并不是Strategy模式最优雅的应用，而且和我们设计回滚功能一样，在结构和逻辑上都引入了不必要的复杂性，但是它满足了实验对角色分离的要求，如图4-4所示。

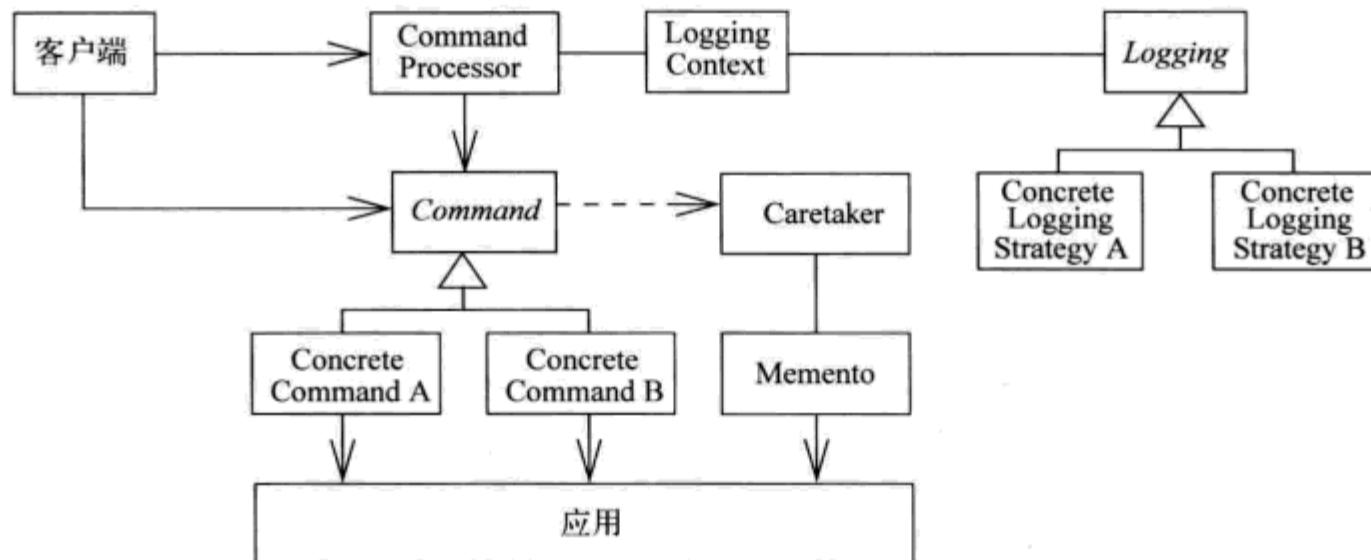


图 4-4

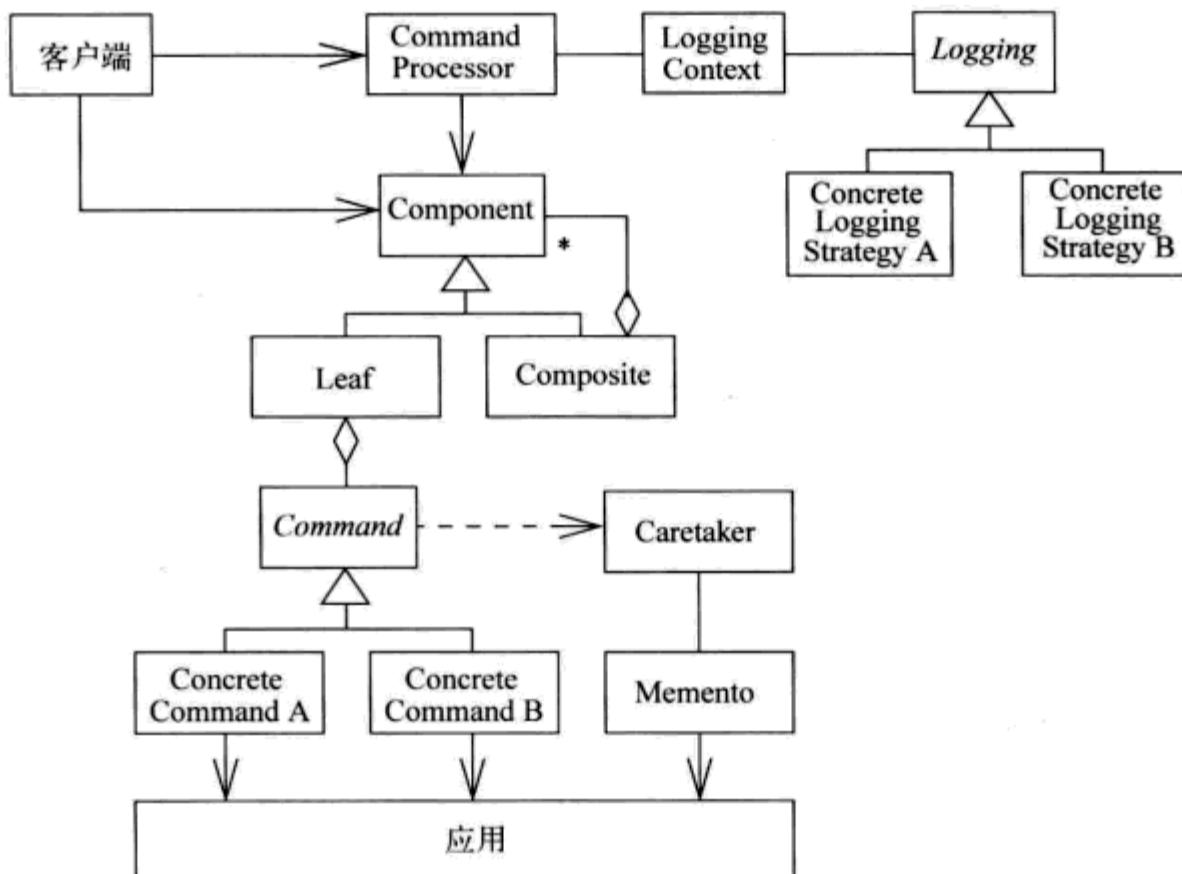
现在，到了实现实验的最后一步，即支持命令组合（compound command）。一个ConcreteCommand对象可能由其他ConcreteCommand对象按特定的执行顺序组合而成。符合这一应用场景的设计模式是Composite模式，其中命令组合可以用模式中的组合对象来表示，而“原子”命令是其叶子对象。由于不允许模式之间的集成，所以我们不能直接将Composite模式应用到设计中，需要一些小手段。

Composite模式可以这样集成到设计中：客户端实例化若干Component对象，这些Component对象既可能是Composite模式中的Composite实例，也可能是Leaf实例。Leaf对象中包含一个Command类的引用，因此可表示任何一个具体的ConcreteCommand。而Composite对象将多个Leaf对象聚合在一起，也就间接地将多个ConcreteCommand对象聚合成了命令组合，如图4-5所示。

现在，我们得到如图4-5所示的请求处理框架，它的架构设计中包括5种模式。尽管每个模式都将某个具体问题解决得很好，但是作为一个整体，这个设计完全没有反映出它应该所具有的质量。它具有完全不必要的复杂性，根本不像一个整体，很难理解、维护及扩展。另外，每执行一个请求都要涉及较多的部件，因此性能也会有问题。

以指标来衡量，这一设计在“代码行数”和“类的数量”两项的分数很高，但它们都不是质量指标。“质量”（quality）和“数量”（quantity）不应该混淆，尽管他们经常混淆。

这个设计还有一个方面可能会取悦于部分人，即如果以模式为基准对系统进行模块化分割，那么这个模块化结构很容易在文档中体现。因为每个模式的使用都是独立的，与其他模式有明确的分界线，所以很容易用一张简单的图来表示设计中用到的模式，如图4-6所示。



4

图 4-5

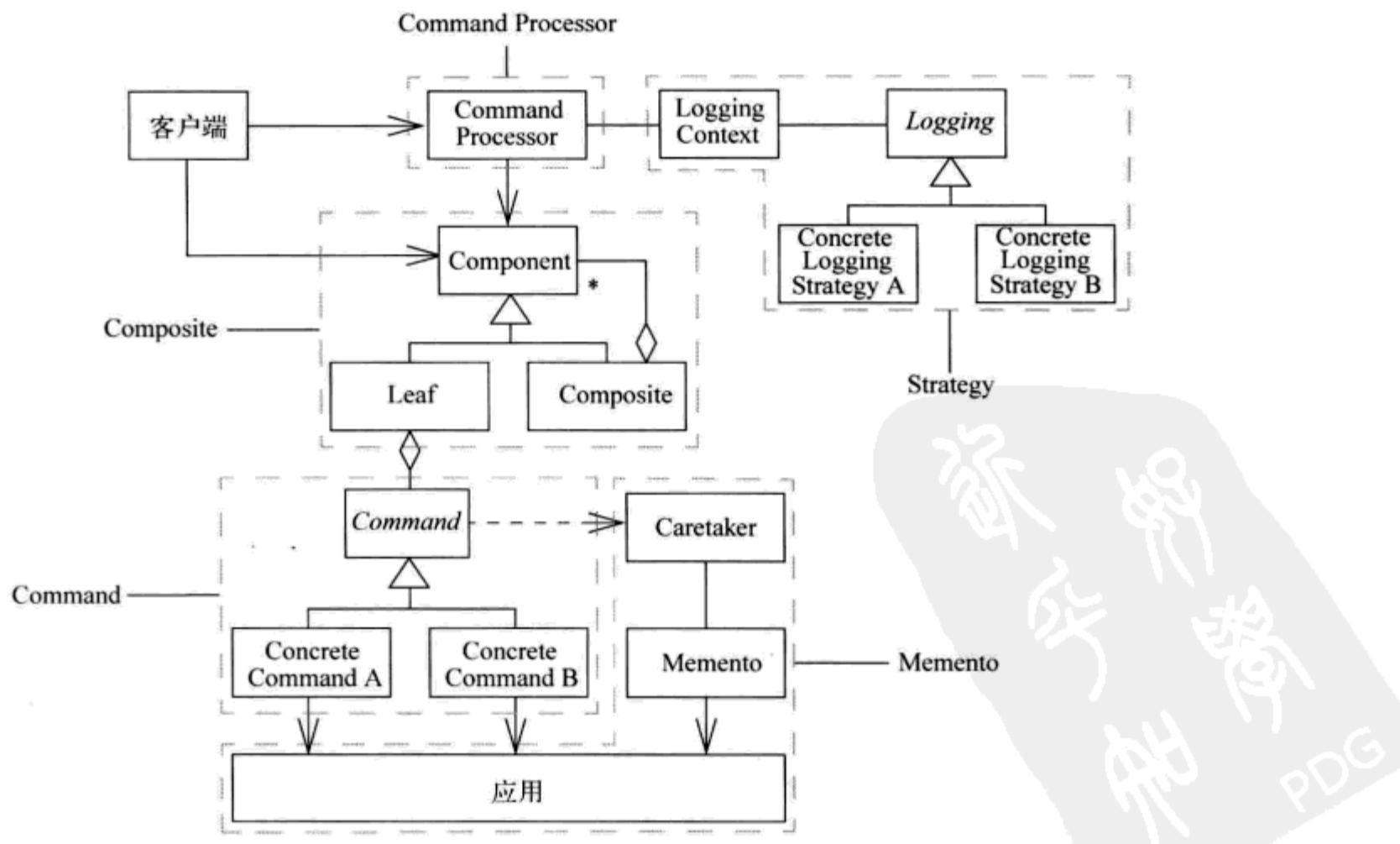


图 4-6

然而，这种看似简单的分割其实是过度简化。正如我们前面提到过的（比如第2章），模式并

不是像孩子玩的积木块那样，搭在一起就行了。

这个实验告诉我们：假如开发人员没有意识到模式是如何关联的，而只是简单地堆砌这些模式的话，就会产生非常复杂的设计。

4.3 第二个设计实验：交织在一起的模式

如果取消之前的约束条件，允许同一个组件承担不同模式中的角色，请求处理框架的架构就会大大改变，它的演变路线也将大不一样，并且较为平顺。例如，我们可以让Command类承担Memento模式引入的caretaker角色。因此，原来只执行请求的ConcreteCommand类现在也负责处理Memento。这样就消除了独立的Caretaker类，减少了不必要的结构复杂性和性能开销。Command抽象类还可以用Explicit Interface模式来实现，从而将提出请求的客户端和将请求对象化的ConcreteCommand类的具体实现解耦。

另外，还可以让CommandProcessor类来扮演Strategy模式中的上下文组件的角色，不需要单独的LoggingContext对象，因为CommandProcessor类就能单独完成对指定日志策略的调用。如果不需要日志，可按Null Object模式，在CommandProcessor中指定Null Logging策略。整合Command Processor模式和Strategy模式之后，又从原始设计中消除了一个类。

如果结合使用Command模式和Composite模式，我们还可以重构请求处理框架的架构。如果让Command模式中的Command类同时充当Composite模式中的component角色，将ConcreteCommand类当做叶子，在架构中新增一个CompositeCommand类，那么组合命令设计变得稍微简单一点。

重构以后，我们会得到如图4-7所示结果。

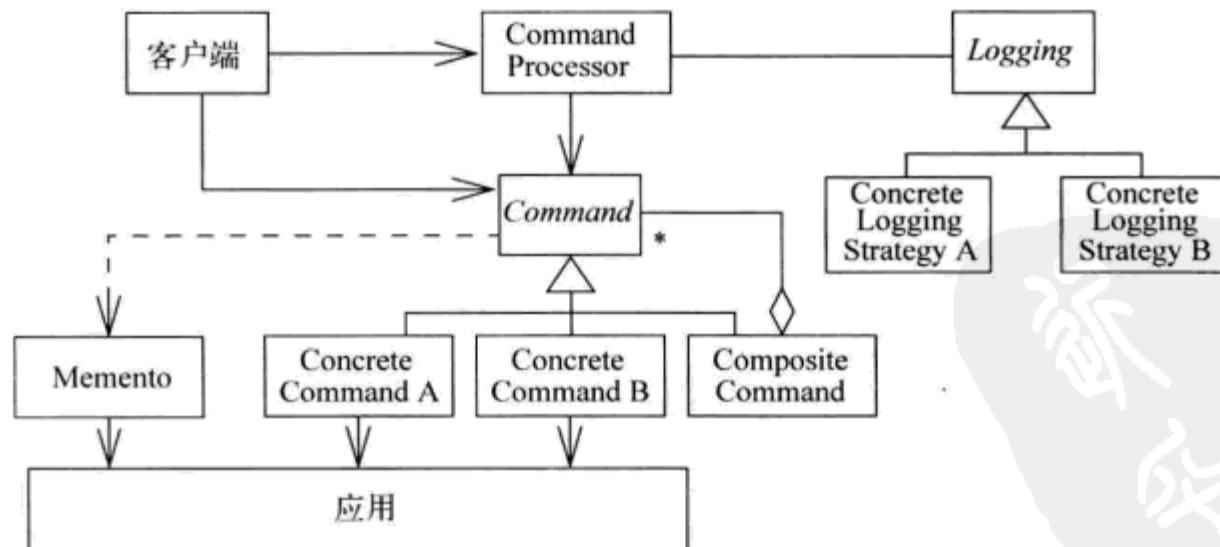


图 4-7

现在，这个设计架构比第一个实验得到的结果更容易理解和维护。从代码行数或类的数量来算，代码更加紧凑，很可能运行时CPU和内存的用量也更少。假如对性能有更高要求，当前的设计因为精简，比前一个更容易优化——“记住，什么代码都比不上‘没有代码’快” [Tal94]。

在这个架构中，充分利用这5种模式中各角色之间的关联关系，使设计非常简洁。例如，

Composite模式和Memento模式使Command模式更完整，而Command模式和Strategy模式使Command Processor更完整。某些经过重构的组件现在承担了多个模式中的角色，因此，现在的设计比之前的设计要紧凑得多。比如，Command类扮演了Command模式和Command Processor模式中的command角色，同时还扮演了Composite模式中的component角色和Memento模式中的caretaker角色。同样，CommandProcessor类同时参与了Command Processor模式和Strategy模式的实现。

我们拒绝了用模式的边界来分割模块的观点，消除了因此带来的偶然复杂性，那么自然会引出另一个问题，这样做会不会给用文档和图示来说明设计中的模式用法带来麻烦呢？由于类不再遵循某个模式来组织其层级结构，也不是某个模式所专有，模式间出现了很多重合的地方，再继续照前面的图示画法恐怕会乱成一团。因此，我们必须换一种画法，与其框出模式的范围轮廓，还不如用注解说明效果更好。图4-8用了“pattern:role”注解标记法，这是Erich Gamma在20世纪90年代中期定义的一种图解模式[Vlis98b]。

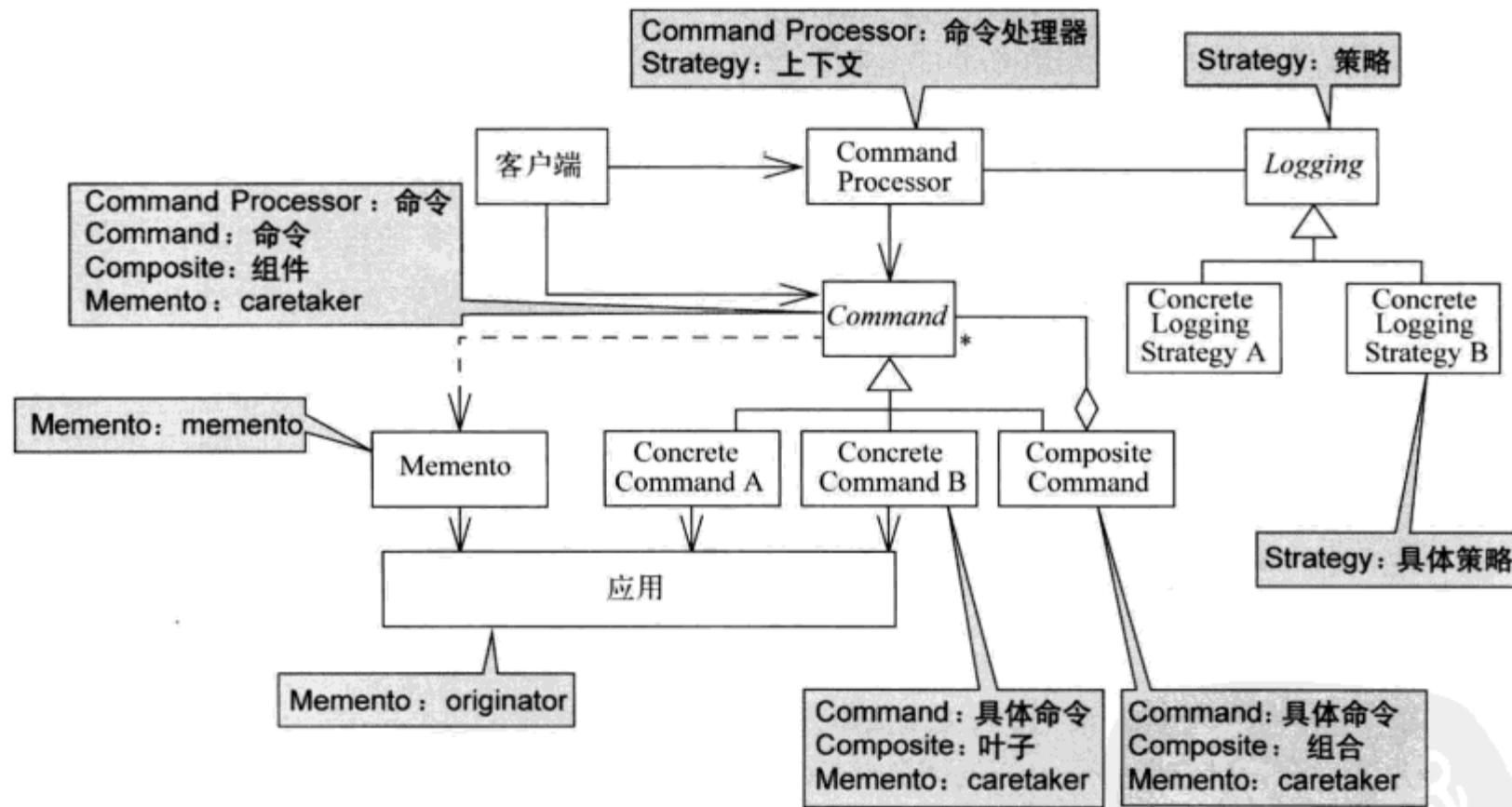


图 4-8

4.4 模式密度

通过对“好”的软件架构进行分析，我们发现它们在使用模式方面常倾向于高密度、紧密集成的方式。进一步研究之后还发现，正是这种模式密度为架构的发展留下了余地——好的架构封装未来的变化[Cope98]，好的架构论证充分、结果合理。从模式的角度来看，只有展现了模式之间多种多样的关联，才有可能创建出这样的架构。模式之间互相补充和完善，并以多种多样的方式组合成一个更大的结构。换句话说，成功的架构是不可能忽视模式之间的关联的。然而，达到较高的模式密度并不等于可以随意地将尽可能多的模式堆砌在尽可能少的类中[Vlis98b]。

密度有好处也有坏处。相对较少的类却容纳了较多的模式，这使设计多少有些深奥；好比好诗言简意赅，而言简意赅的却不一定好诗。反过来，高密度还给人缺乏才气的印象。

尽可能清楚准确地捕获这种模式间的关系是目前模式研究的一个强有力的动力。例如，《面向模式的软件架构（卷4）》[POSA4]就为构建分布式系统描述并串联了114种模式，而且它还将这114种模式与另外来自于其他模式资源的180种左右的模式联系起来。同样，Martin Fowler著的《企业应用架构模式》[Fow02]为构建基于Web的多层企业信息系统串联起51种模式，而Gregor Hohpe和Bobby Woolf著的《企业集成模式》[HoWo03]整合了66种模式来帮助理解和构建高质量的消息中间件。Grady Booch著的《软件架构手册》[Booch]无疑是已知模式研究领域中规模最大的一次模式集成活动。在本书写作之时，该项目已涵盖软件开发领域中的上千种模式。

所有这些著作都主要关注模式的综合和关联，而不是创造新的模式。它们的目标是把模式作为一张巨大的关系网中的连接点来理解，而不是将它们作为软件开发这片海域中等待发现的孤岛。孤立地应用模式会导致事倍功半的结果，本章的两个设计实验已经做了清楚的示范。

角色和关系

我们充分发挥模式关系的潜力，有赖于模式的一种重要特征：模式给组件引入新的角色与职责，却不一定需要引入新的组件来表达这些概念。在模式的集成结构中，在不违反所涉模式本意的前提下，把角色组合、集成起来的方式有很多，差别也可以很大[Cool97] [Cool98]。

前面我们提过一种论断，并且已经被我们的实验所证明：像类、包那样的组件，各组件同属一个维度，有着严格的模块边界，可以像砖块一样组合起来，而模式却不一样。开发人员只有把不同模式中的角色整合在一起时，才能在设计中自觉高效地使用模式。故意保持模式之间泾渭分明的用法，很可能导致过于复杂的架构。尽管每个应用程序似乎都能药到病除，实则应用模式带来的阻碍多于帮助。

虽然第一个设计实验的有些细节是为了解释本章观点而杜撰出来的，但它很好地反映了某些系统（错误地）使用模式的情形。比如在第0章中关于Observer模式的故事里，项目所遇到的问题的原因和第一个设计实验一样，开发团队将模式当成孤岛来使用，没有像第二个实验一样考虑成一组可以整合、编排的角色。如果将模式形象化地表示成设计空间中一块块区域，并且区域之间是有所重叠的，这不同于类图那种不可捉摸、模块化的形象。

然而，只了解角色还不足以成功地应用模式。我们还需要一些指导原则去指出哪些角色组合是真正有用的。如果没有这些指导原则，无经验的模式使用者可能会在解决问题时加入了太多或太少角色，甚至使用错误的角色。模式关系提供了部分遗漏的指导原则。

就模式的实现和使用而言，正是模式角色与关系的这种互补关系为模式提供了强大的力量。同样，由于这种在原有模式基础上构建新模式的可能性（增加新元素承担新角色，或将新角色加入现有的元素），使这种有意识地用模式去驱动的软件架构方式更像是一次有建设性的对话。

没有什么比别无选择更危险的了。

——*Émile-Auguste Chartier*

互补 (complement) 是指一种事物在某个方面使另一种事物变得更完善。本章中我们来看如何将这个概念应用到模式上。首先，我们观察多种模式如何对相似的问题提出差异化及竞争性的不同方案来丰富设计过程。其次，我们探讨一种模式如何补足另一种模式的设计。最后，我们通过实例演示竞争性模式之间的协作乃至合并，发现关于互补的两种视角——过程上的与结构上的以及竞争性的与协作性的，它们的界线其实没那么明显。

5.1 一个问题，多种解决方案

许多开发人员常有一种错觉，以为一种特定类型的问题只能被一个模式解决[BuHe03] [Bus03a]。这种错觉有时会因为特定模式的名字而被放大。比如，集合迭代问题就用Iterator模式解决，对象适配问题就用Adapter模式解决，有状态对象的生命周期问题就用Objects for States模式解决，等等，就像我们在0.2节中所讨论的那样。

把一类问题对应到一种解决方案，这种想法对某些模式用户来说，就像在陌生环境中寻找熟悉的物品以求得安全感，因为这种思维看起来简化了问题，让生活更好过。特别是，别无选择被看做是一种优势而非劣势，因为这消除了为了选择所需要的设计对话。然而，如果设计是这么简单的话，这方面的技能和判断力就不会这么稀缺而昂贵了。这种狭窄的视野把问题过分简单化了，是使用模式的一种通病。

设计词汇缺乏多样性是需要改正的问题，而不是值得追求的状态。本书的一位作者曾在授课时遇过一位学员期盼模式给他带来“免于思考的自由”，这绝非模式社区的目标，而任何期望模式能提供这么一种万能膏药的人则注定会失望！模式旨在提供一种载体去帮助设计推导，并没有免除你论证设计合理性的责任。虽然模式限制了一些自由，却不妨碍你作出选择，并且通过明确设计的效果和代价来使这种选择过程更加清晰。

软件开发和设计过程涉及许多不同种类和不同层次的思维过程。开发人员经常会为与主要问题无关的琐碎细节分心。另一方面，他们过分习惯性地依赖于特定的解决方案，没有针对性地考

查该方案是否恰当及有效。模式为我们提供了关注的焦点。

拓宽视野

我们经常听到一些故事，其中讲述解决方案如何因为应用了某种模式而变得不必要地复杂[Bus03a][Bus03b]。在这些故事里，责备通常落到模式的头上。然而，当我们稍微观察一下出问题的设计，经常发现根源其实没那么复杂。

- 用了错误的模式。例如，实现元素访问好像用一个普通的Iterator模式就可以了，可是在分布式环境下，Batch Method模式才是更好的选择。
 - 丢失了某种要素。例如，虽然设计中正确地采用了Command模式的类层次，但若不搭配使用Command Processor，则必然使代码拘泥于细节与聪明的花招。
- 记着这两点提醒，我们来看一下模式互补的概念，它有两个方面。
- 通过竞争提供的互补性。如果对于相同或相似的问题，一种模式可以作为另一种模式的替代方案，那么从设计决策的角度来说，它是对另一种模式的补充。
 - 通过结构性的完善提供的互补性。在特定的设计中，一种模式是另一种模式的天然配对，它们共同构成一个完整的设计。

上述两种互补性乍看之下界线很明显，好像它们总可以被区别对待。不过互补的概念并不局限于模式领域，倘若借鉴其他领域的经验，就会发现竞争与合作二者其实并非水火不容。

5.2 互相竞争的模式

不同的设计可能在角色的具体安排上或特定模式的实现细节上有所差异，这就使得一个特定问题有很多种不同的解决方案，就像我们在第2章中讨论的那样。设计的差异性也可能体现在模式的选取上，因为有些问题不止对应一种解决模式。例如，如果设计要求固定一个算法的步骤，同时又允许步骤的实现细节有所变化，那么Strategy和Template Method都是可选的模式。虽然这两个模式在解决问题的抽象原则上是相同的，但它们之间具体结构的差异足以让它们成为不同的模式。Strategy和Template Method各自着眼于不同的设计思路，给出了各自的特征。

5.2.1 以状态为例

另一对有代表性的例子是Objects for States（也被称为State）模式和Collections for States模式。这两种模式都解决这样的问题：一个对象的具体行为依赖于其当前模态。它们解决问题的抽象原则也相同，都是把对象的状态与行为区分开。然而不同的是，Objects for States模式封装了这个原则的结果，而Collections for States模式则把它公开给了状态相关对象的客户端。

下面是对Objects for States模式所针对的问题与解决方案的总结及示意图[POSA4]（如图5-1所示）。

一个对象的行为是模态的，其行为模式依赖于对象当前的状态。然而在对象的实现中以硬编码的多分支条件代码来达到这种效果，可能损害代码的可读性和扩展性。

因此：

把对象的状态相关的行为封装成一个状态类的层次结构，其中每个类对应到模态的一种状态。对象的状态相关行为都通过方法调用转发，交给相应状态类的实例去处理。

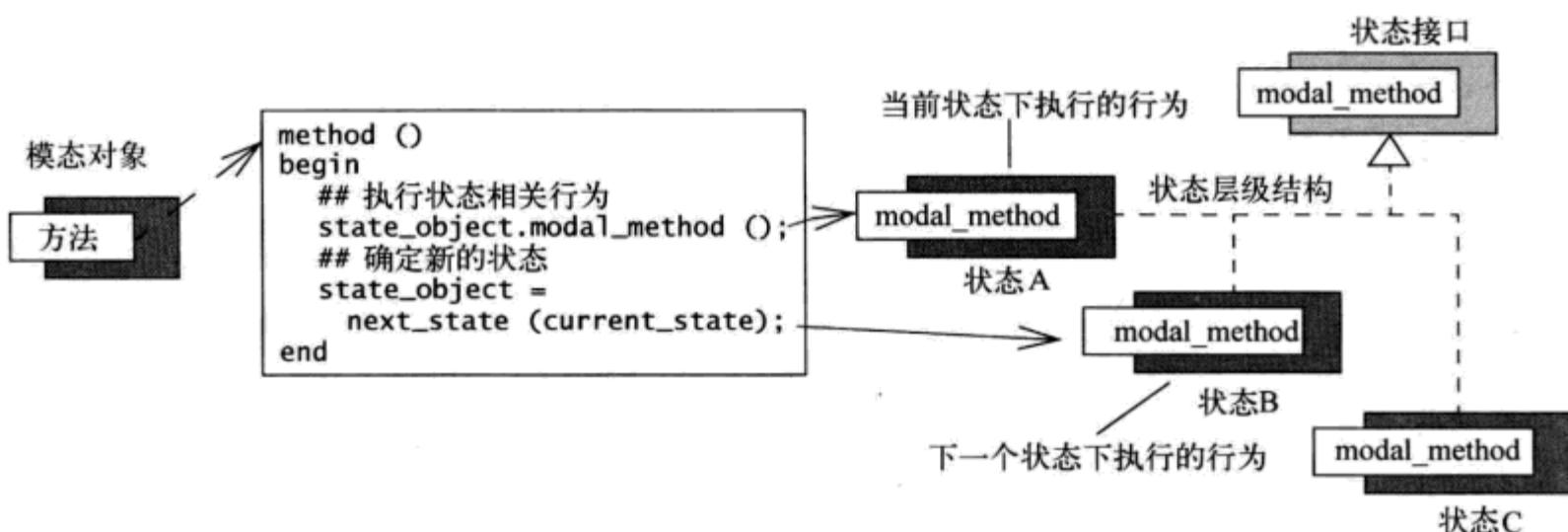


图 5-1

下面是对Collections for States模式所针对的问题与解决方案的总结及示意图[POSA4]（如图5-2所示）。

对于行为依赖于自身当前状态的对象，可以将之建模为一个个独立的状态机。然而有时，虽然对象的客户将这些对象的行为看做是模态的，可是站在对象一方来看，却完全独立于特定客户的状态模型。

因此：

在客户内部，对于感兴趣的每种状态，分别用一个集合来表示，将处在某种状态的所有对象都保存在该状态对应的集合中。

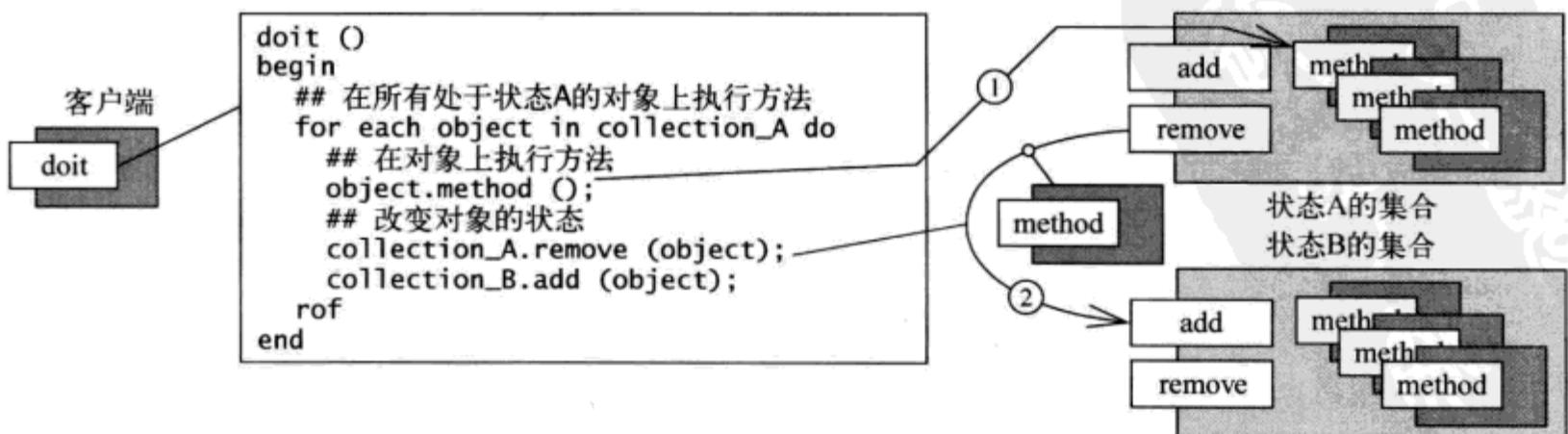


图 5-2

从而，Objects for States模式为每个状态引入相应的状态对象，同时在一个上下文对象里将状

态与状态对象关联起来。一个特定的状态对象实现了上下文对象处于那个状态时的行为。上下文对象维护所有的状态对象，并以对客户透明的方式切换状态。对比之下，Collections for States模式采取了截然相反的思路。与其让状态相关的对象去管理各自状态，不如让客户接管这个管理角色，从而状态对象本身可以不必关心自己所处的状态。

读者可能会问，难道后面的那种做法——把自身的概念放到外部表示——不是破坏了面向对象的封装原则吗？为了回答这个问题，我们来设计一个基于Command模式的请求处理框架，这个框架需要支持undo/redo机制。在这个设计中，Command对象是状态相关的。在do/redo状态时，它们执行相应的请求处理过程；在undo状态时则取消这个过程^①。很明显，在这种情况下我们可以采用Objects for States模式。这种设计将为每个Command对象引入两个状态对象，一个针对do相关行为，另一个针对undo相关行为。每个Command对象维护自己的两个状态对象：命令执行机制只要适时调用Command对象，Command对象自己会去“做正确的事”[Hearsay02]。

虽然这个方案听起来还不错，但是它毕竟刻意用了一种知名的模式。细究之下，我们却发现这个设计过于复杂了。它需要引入一大批对象，维护状态的额外负担也相当可观，这让这个设计怎么都算不上一个“好设计”。大型系统通常有很多命令，而每个命令都要增加两个状态对象，还要加上维护状态对象的代码以及透明地切换状态的代码。

假如每个命令的状态集都不一样，则方案中引入的附加成本几乎不可避免。不过因为每个命令都有着一样的do/redo和undo状态，如果用布尔变量表示这些状态，虽然不符合面向对象设计模式的一贯评价标准，却有可能起到简化方案的效果。但即便基于布尔变量的方法确实对Objects for States方案有所改进，仍然不值得仿效。

有效的命令处理机制，例如Command Processor模式所描述的机制，采用另一种策略去处理命令状态。在该策略下，Command对象用do和undo两个方法而不是状态对象，分别实现执行命令及撤销命令时的相应行为。这种设计省去了状态对象以及用于维护状态对象的代码。区分不同状态的职责被转移到了Command Processor对象。

Command Processor对象可以增加两个栈，一个用来维护处于undo状态的Command对象，另一个用来维护处于redo状态的对象。每个Command对象刚创建时处于do状态，一旦执行过后，它的状态就由do转为undo，Command Processor随即把它推入undo栈。

当用户选择undo功能时，undo栈中最上面的Command对象被弹出，然后它的undo方法被调用。Undo方法执行结束后，该Command对象的状态由undo转为redo，并被推入redo栈。当用户选择redo方法时，同样的机制作用在redo栈最上面的Command对象上。为了让Command Processor对象透明地处理这个过程，我们可以为每个栈实现一个do方法来封装上述行为，就像Command对象的do方法一样。

图5-3展示了上面所描述的结构和行为。

虽然现在命令处理器要负责直接维护各个Command对象的状态，执行命令的过程却比之前Command对象自己维护自身状态的时候（不管是基于标志变量的还是原始的Objects for States）更

^① 为了简单，本例中假定所有命令对象都能被撤销，实际上这种情况很少在产品级应用中出现。[POSA1]

简单而直接了。此外，现在的设计还消除了先前设计中引入的许多附加成本和复杂性。新设计的效果看起来是否很奇怪？希望不会，因为它可是经过广泛验证，从Collections for States模式衍生出来的一种面向对象的请求处理策略。

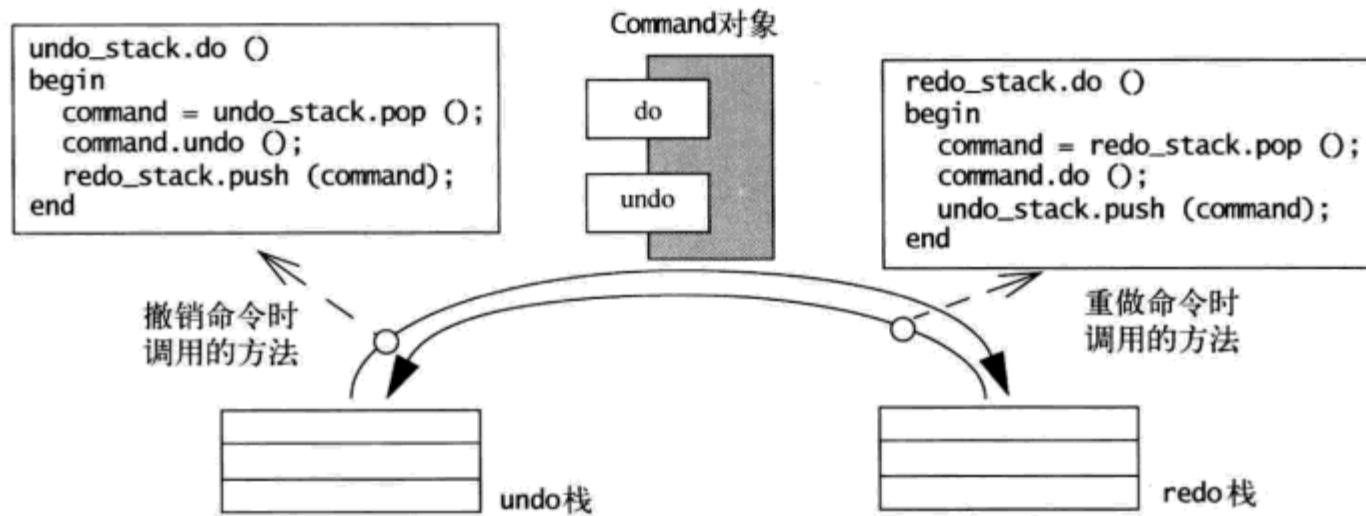


图 5-3

Collections for States的这种外在状态模型提供的解决方案在某些情况下比其他方案更简单、效果更好，我们可以找到许多这样的例子。比如对于自动垃圾回收来说，收集无用对象的过程有时成本很高，采用分代的垃圾收集器则可以降低这种成本。在这种设计中对象被组织为不同的“代”，将年轻的对象与年老的对象区分开。根据统计，大部分新创建的对象生命周期较短，所以用不同的集合来把不同“代”的对象区分开有助于优化收集过程。

操作系统上的任务调度是Collections for States的外在状态模型提供更简或更优方案的另一个例子。每个任务，不管是线程或进程，都会处在一系列不同的状态中，比如“等待运行”状态、“被I/O阻塞”状态等。任务调度器需要决定下一个要运行的任务的时候，可以扫描整个任务列表，这需要花费线性时间。如果把任务按它们的状态组织起来，那么这时所有的就绪任务都保存在“等待运行”队列里。这时搜索下一个待运行任务的成本是常量，而不是线性时间。

或许你会想起第0章那个与Objects for States有关的误用模式的故事。正因为死板地认为所有涉及状态生命周期管理的问题都必须用States模式去解决，导致开发人员放弃了对症下药的设计，即使在引入大量偶然复杂性的情况下，仍然坚持采用该模式。模式词汇的贫乏导致设计者看不到其他的可选方案。

5.2.2 模式族

模式著作里有很多成对的模式，这些成对的模式实质上解决相同的问题。例如，Active Object和Monitor Object模式都用于在并发环境下协调程序对共享对象的访问。

Active Object模式通过分离服务的请求与执行来提高并发性，简化对处于其自身控制线程中对象的同步访问。

Monitor Object模式通过将方法的并发执行同步化，确保同一时刻在一个对象上只有一个方法在执行。它还允许同属一个对象的多个方法合作地协调彼此的执行顺序。

两个模式都是对请求进行同步控制，以便我们的对象能接收来自多个线程的客户请求，但两个模式对相同的原则有着颇为不同的表达，并且也都有各自的优势及额外成本。每个Active Object都有自己独立的线程——所以是“活动的”，而方法请求的执行相对于调用者是异步的。作为对比，Monitor Object的方法调用都是同步的——相当于借用了调用者的线程。两个模式都是用来解决对象同步问题，各有自己的优点和缺点，主要表现在方法执行的灵活性以及运行时的附加成本上。

Half-Sync/Half-Async模式与Leader/Followers模式是另一对成对模式的例子，它们都用于在网络服务程序中并发地处理用户请求。

在并发系统中，Half-Sync/Half-Async模式通过分离同步和异步的服务处理过程，来达到在不过度降低性能的前提下简化编程的目标。该模式引入了两个互相通信的层，一个用于同步处理，另一个用于异步处理。

Leader/Followers模式提供了一种有效的并发模型，其中多个线程轮流共享同一组事件源，以检测、多路分解、分派并处理发生在这些事件源上的服务请求。

两种模式都能用来实现线程池，但同样它们也对同一原则作了不同的表达，并且也都有各自的优势和额外成本。Half-Sync/Half-Async设计利用一个同步队列去协调同步层上的多个线程与异步层上的线程或中断处理程序之间的交互。同步层上的线程可被阻塞，而异步层上的线程或中断处理程序则不可被阻塞。相比之下，Leader/Followers模式没有同步队列，也没有把线程分到两个层去。两种模式都支持线程池，但有不同的优点和缺点，主要表现在事件处理的顺序、可预测性以及实现的难度上。

面对类似的多样化选择，问题出现了：如果多个模式都解决同样的问题，该选哪个呢？随意选一个显然既不理性又不有效，因为作出的选择可能影响深远。我们需要更多的信息来获得合理的选择。作出“正确”选择的关键是要了解不同候选模式需要的上下文、驱动力和带来的影响，同样也要理清所面对的设计问题本身的上下文和驱动力。

竞争性的模式不仅成对出现。在GoF的模式编目中，Builder、Abstract Factory以及Prototype以创建型模式的面目出现，在相近的层面上互相竞争。类似地，Objects for States与Collections for States之争还有另一位参与者Methods for States。另外，“Executing Around Sequences”论文[Hen01a]中收集的4种C++模式都用于安全地处理异常，它们也互相竞争，但它们的结构及带来的结果则完全不同。

5.2.3 迭代开发

顺着前面章节对Batch Method模式演化的讨论，我们可以看出，第1章所描述的Batch Method模式与经典的Iterator模式形成对照与竞争，同为解决集合遍历问题的候选方案。选择哪种模式的一个决定性因素是访问单个元素的成本。对一个需要通过网络远程访问的集合来说，反复地通过远程操作一次访问一个元素的成本可能太高，不如把遍历封装到一个方法中。不过当然，这种方法的另一面是，比起Iterator简单的遍历，使用Batch Method需要编写更多的循环：比如，在调用

远程操作前，客户端需要循环准备好调用时发送的数据项，服务器端执行时也要循环处理批量数据，执行结束后客户端还要循环处理所有的结果。

除了上面提到的两种模式，还有另外一种迭代模式：Enumeration Method。这种模式的特点是“控制流程反转”，习惯于程式开发或使用“花括号”家族面向对象编程语言的程序员可能不熟悉这种特点。与Batch Method类似，Enumeration Method把循环机制封装成聚合对象的一个方法。

下面的类C伪码片段说明了Iterator模式的集合遍历方法：

```
total = 0
iterator = collection.iteratorFromStart()
while(!iterator.atEnd())
{
    total += iterator.currentValue()
    iterator.moveToNext()
}
```

控制流程由集合用户决定，但这也意味着用户必须管理这些逻辑。对于Enumeration Method来说，上述结构被反转了：

```
total = 0
collection.forEachElement(
    function(currentValue) { total += currentValue })
```

Batch Method以一个只关心输入和结果的简单任务为中心，而Enumeration Method接受一个Command对象或类似物充当循环体。Enumeration Method在聚合对象的每个元素上调用Command对象，Command对象对此作出响应。因此，用户既不能改变应用服务的控制流程，也不了解它的内部实现。这种反转模式经常被称为“好莱坞法则”，取其“别联系我们，我们会联系你”之意[Vlis98b]。好莱坞法则有别于大多数传统的程式编程或深受程式影响的面向对象编程，带有函数式编程的影子。

下面是对3种模式的总结。

Iterator模式提供了一种方法，顺序访问聚合组件中的元素，同时又无需公开元素在聚合组件内部的表现形式。用户通过一个独立的、可控的Iterator对象来遍历聚合组件。

Enumeration Method模式把在聚合组件上执行迭代以对其所有元素分别执行动作的操作，封装成聚合组件本身的一个方法。具体的执行动作由调用者作为一个对象传入。

Batch Method模式把对聚合对象内元素的重复访问进行打包，以降低多次访问单一元素的成本。

对于复杂的聚合数据结构来说，Enumeration Method通常更易于使用。迭代操作可与Visitor模式结合，把遍历行为转化为一系列以元素类型决定去向的简单回调。在此基础上，借助Mock Object模式可极大地简化测试工作，只要检测不同的Mock对象是否触发了期待的回调。最后，循环机制和策略以及同步处理及错误处理的细节全部得到了封装。

当然，事件驱动的编程风格不一定完全适合所有的问题和语言。但是，Enumeration Method

帮助我们加强了这样的观点：最流行的解决方案——有时也被（错误地）认为是最“符合直觉”的方案——并不一定是最有效的。直觉往往依赖于熟悉程度，但通常一个方案是否直接与显然，开发人员的背景和经验比方案本身的内在属性影响更大。

举例来说，早期的Eiffel函数库[Mey88]完全没有Iterator的概念，导致出现过分依赖状态的解决方案，不得不在使用中作变通处理。最初的Eiffel方案要求每个集合持有一个游标来表示当前位置。如果需要嵌套的迭代操作，还得再设置一个标志变量去记录游标位置。

通过Iterator遍历集合对象的方式在C++、C#和Java语言中被认为是“符合直觉”的。在这样的编程文化中，Enumeration Method是一种“违反直觉”的方式。Iterator方式能很好地解决大部分问题，也符合这些语言及其程序库的风格。然而，有些问题如果用Iterator模式去解决会比较别扭，实现起来较复杂，掌握Enumeration Method的知识能帮助开发人员更好地理解这些问题。例如，树的遍历问题用递归很容易解决，正好适合应用Enumeration Method模式，而用Iterator模式的话就要增加额外的状态来支持回溯处理。

另一方面，Enumeration Method在Smalltalk和Ruby中是很常见的解决方案，反倒Iterator不那么“符合直觉”。在这些语言中，Enumeration Method的用法也被极大地简化了，因为代码块可以被当做对象——闭包（closure）。因此，给Enumeration Method传递Command对象就是传递一个代码块而已。

C++不支持传递代码块。Java的内部类是一种相近的概念，但句法上太沉重，难以支撑Enumeration Method获得与Iterator对等的地位。C# 2.0提供的轻量级匿名方法倒是更可行的替代品。只不过C# 2.0同样引入了新的语法形式“迭代器方法”（iterator method）来表达Iterator逻辑，它对很多应用场景更具吸引力。

UML类图5-4展示了我们所说的互补方案，以及Enumeration Method和Iterator这两种模式在结构、角色以及职责上的差异。

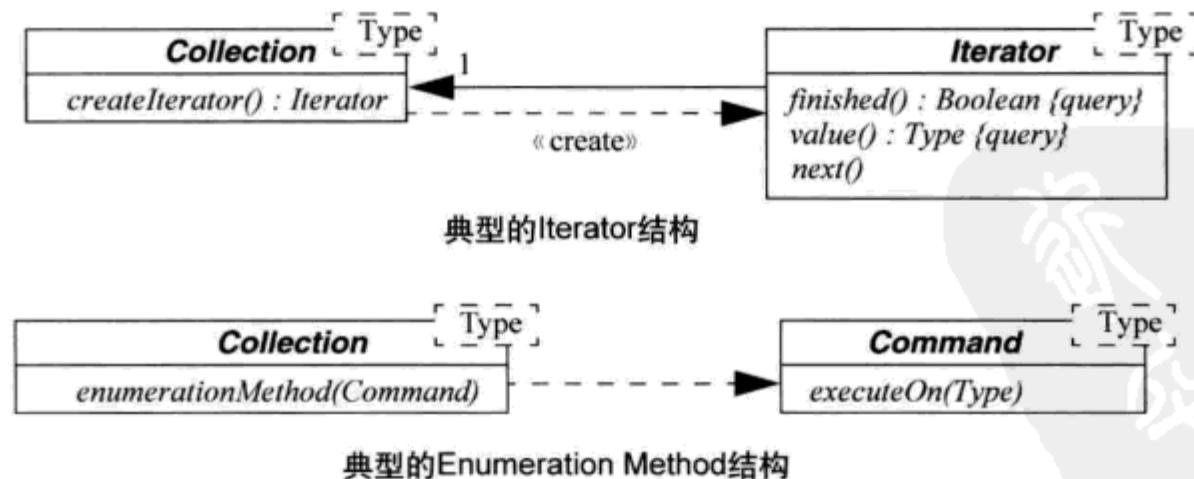


图 5-4

命名上或许有一点值得澄清：对GoF著作的细致研读显示Enumeration Method其实是已被该书记录在册的一个方案。书中关于Iterator模式的叙述大部分集中在所谓的经典Iterator模式上，即以一个独立的Iterator对象遍历一个集合。这种经典形式的另一种称呼为External Iterator。与之相

对的Internal Iterator，就是我们所讨论的Enumeration Method。

然而，如前所述，两种形式其实毫不相似：它们并不是同一种叫做Iterator的模式的两个简单变体，而是两个独立的模式，各自的方案有其独特的结构与设计权衡。我们认识到External Iterator这个名字几乎没有开发者使用，因此选择让Iterator这个名字代表它的通常含义。同时为了避免继续出现Internal Iterator和External Iterator是彼此简单变体的误解，我们赞成改用Kent Beck取的Enumeration Method代替旧的名字。Kent Beck是第一个将Enumeration Method作为单个模式记录下来的[Beck96]。

5.2.4 适配开发

一则描述之下隐藏了两种模式的例子，还有Object Adapter和Class Adapter，它们都被包含在Adapter模式[GoF95]的描述中。Adapter的意图表述如下。

将一个类的接口转换为客户端程序所期望的另一种接口。Adapter使得原本接口不兼容的类能在一起工作。

然而，经过对Adapter模式描述的分析，可以发现在相同的意图之下其实隐藏了两种截然不同的模式：Object Adapter和Class Adapter。它们并不是同一方案结构之下两种变体，它们依据完全不同的方案机制，遵从不同的实现约束，并旨在解决不同的问题。即使两种模式意图一致，但Adapter模式描述中处处可见的“双重叙述”（dual narrative）却突显出它们的互补本质。

下面的总结帮助我们看清两种模式的共性和差异。

Object Adapter模式把一个类的接口转换为客户端程序所期望的另一种接口。适配行为使得原本接口不兼容的类能在一起工作。通过用一种对象间关系来表达封装，保证了适配行为被封装在间接层的调用转发中。

Class Adapter模式把一个类的接口转换为客户端程序所期望的另一种接口。适配行为使得原本接口不兼容的类能在一起工作。Class Adapter同时继承了Adapter的接口和Adaptee的实现，集二者于一个对象——避免了额外的间接层。

因为这两个模式针对同样的问题采用了不同的解决方案结构，所以它们可以被认为是既互相补充，又互相竞争。图5-5展示了二者结构上的差异。

至于是经由对象间转发来实现完全封装的方式合适呢，还是耦合更紧密的基于继承的方式合适呢，那完全是一个设计决策。继承方式的优点有：Adapter类可以覆盖Adaptee类的方法，Adaptee类的受保护的方法和属性对Adapter类可见，实例化的时候只创建一个对象。然而，继承方式是一种侵入性较强的方法，可能会侵扰别的继承关系。从分层和封装的角度来看，Object Adapter被认为是封装更好且更稳定的设计。

在这两种模式之间进行选择，还有一些权衡条件取决于所用的语言。

□ C++允许非公有继承，而Java或C#中类的继承关系总是公开的。因此Class Adapter在Java或C#中必定呈现为紧耦合，而在C++里则呈现为离散的。

- C++和C#中可以在一个对象内部嵌套另一个对象，所以即使创建两个对象，也不等于分配了两处堆内存，而转发也不过是内存偏移而已，算不上真正的间接层。所以从内存消耗上说，Object Adapter在这两种语言中可以更廉价（C#中仅当Adaptee是struct类型时才适用上述情况）。
- C++支持多继承，而Java和C#只允许单继承另加多个实现接口。在Java或C#中，如果选择继承的适配方式，就失去了继承其他类的机会，所以一定要确保“别堵了基类这条后路”[Gun04]。

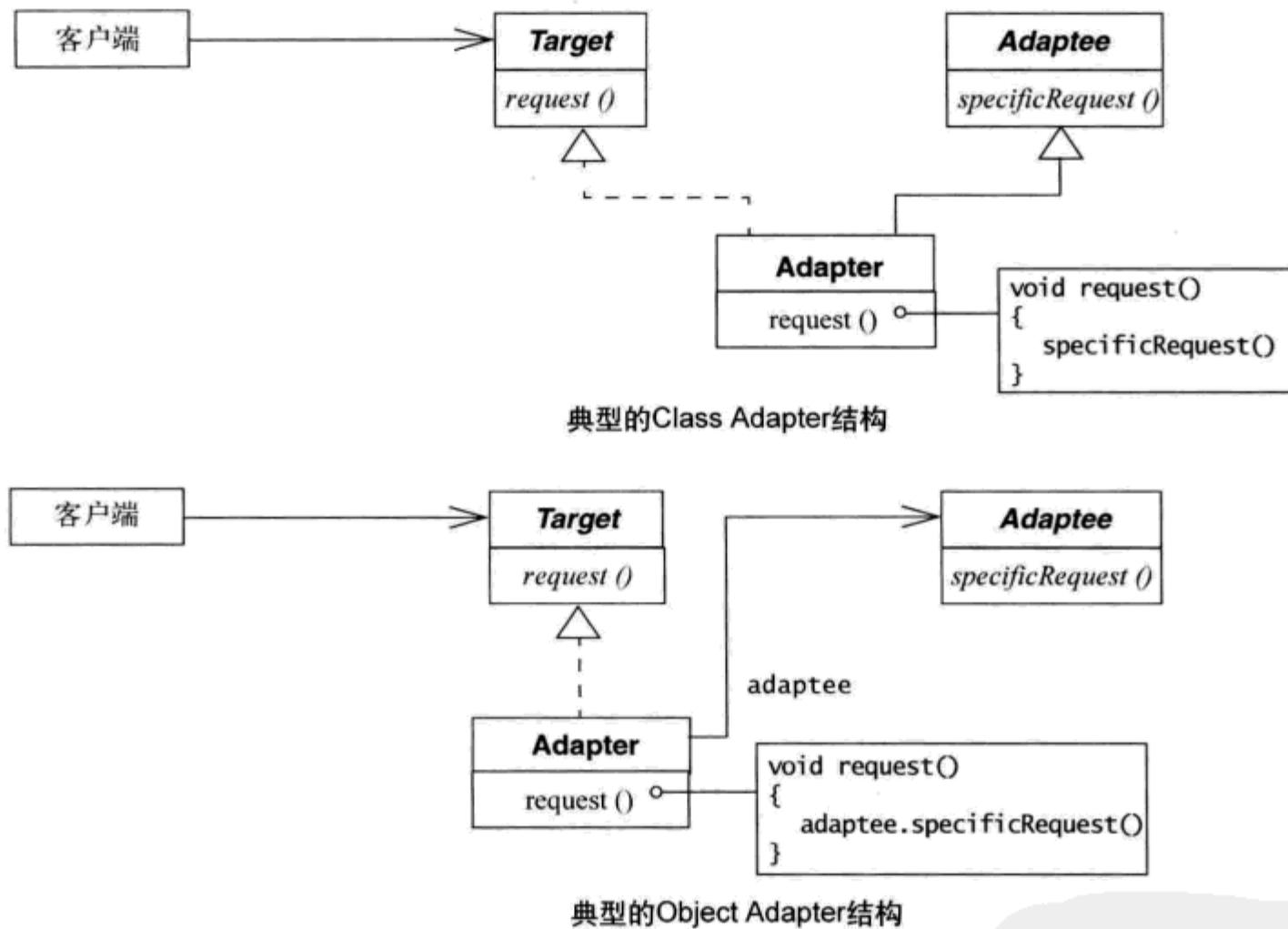


图 5-5

这些关系到具体语言的利害分析，在特定情况下可以成为最终的决定因素。在所有的优点和缺点都被掂量过之后，Class Adapter倾向于作为特例来使用——因为它有较强的侵入性，而且实现和理解起来都不够直观。最好把它保留在工具箱中，当设计者被某些事情捆住的时候才拿出来用，例如：Adapter必须访问Adaptee的受保护特性，Adapter必须覆盖Adaptee的某个方法，或者Adapter的对象标识必须与Adaptee相同。虽然我们对Object Adapter有这样的偏爱，然而这两个模式的互补本质意味着在任何需要适配行为的情况下，它们都是候选对象。

5.2.5 遵从康威定律

1968年，Melvin Conway发表了一项观察结果，这个结果将系统架构与创建系统的机构组织

联系起来[Con68]。

任何从事系统（这里“系统”的含义更广，不限于信息系统）设计的组织，将不可避免地使设计结构仿效其组织通信结构。

这项观察结果后来被称为康威定律（Conway's Law）[Ray91]。

软件的组织规则将与软件开发团队的组织规则保持一致。最初的表述是：让4个团队分别开发一个编译器，你就会得到4个（4-pass）编译器。

之所以称为“定律”（law），主要是为了表达一种塑造系统的驱动力，就像物理定律一样，而不是像一个州或地区的法律那样的规定（rule）。从这个意义上说，它不是一种法律（法律可以被违背），而是一种无处不在的驱动力：不能被违背，只能被另一种驱动力抵消。作为一种驱动力，康威定律可以被看做同态性（homomorphism）[Con68]的一种具体应用。

5

两个事物之间的结构同质关系称为同态性（homomorphism）。从数学意义上，我们可以说，一个系统的线形图（linear graph）和它的设计组织的线形图之间存在着同态性。

但我们真的希望软件架构成为组织结构的翻版吗？这听起来更像是一个应该修正的问题而非必须遵守的规定。哪里有问题，就必然存在造成这种问题的驱动力。上述的康威定律所代表的同态性就是这么一种驱动力。我们在关于康威定律的研讨课上学到，在这个问题上还可以找到与之对抗的其他力量[HvKe05]。

非正式交流对开发软件很重要。如果存在一种阻碍非正式交流的因素（比如外包或离岸外包），那就有必要引入一种补偿因素。不这么做的话就会影响到软件架构。[……]

我们并非无能为力。我们可以介入并影响系统的概念。扫除障碍需要积极的介入。[……]

如果我们不去打破同态性的影响，它将控制系统的架构，所以我们必须积极地去设计系统来打破同态性的影响。

问题在于：同态性是一种非常强大的驱动力，如果它压倒了其他的驱动力，会导致不好的结果。

识别了问题的特征及其驱动力，我们不可避免地要问怎么解决。下面的模式描述了一种解决方案[CoHa04]。

如果组织结构的各部分（如项目组、部门或子产品线）没有密切反映产品的主要部分，或者组织结构各部分之间的关系没有反映产品各部件之间的关系，那么项目就会陷入困境。[……]

因此：

确保组织结构与产品架构兼容。在当前的（模式）语言里，这个意思更可能被表述为产品架构驱动组织结构，而非相反。

组织应定期评审产品架构的项目策略[……]。在架构会议和策略会议上（如果确实是分开的话）都要花时间去关注这个问题，通过一点一滴地改变产品架构或组织结构中的任何一个，使得两者在结构上对齐。

该模式本身也被命名为康威定律，这可能有点让人迷惑。如果当初按照该方案而非驱动力来命名，Align Architecture and Organization会是一个更恰当的名称。

然而，仔细研究之后发现：要想使系统架构和组织结构一致，存在两种不同的方案。

- 使组织结构向产品架构对齐。
- 使产品架构向组织结构对齐。

虽然最终结果是一样的——二者在结构上对齐，然而这两个方案的特征大不相同，并且各自需要不同的技能和实现方式。前者基于组织结构调整，而后者基于软件架构调整。模式中所传达的意图倾向于前者，但是后者也可迁就，因为很多时候后者不管在政治上还是经济上都更容易做到，有时在技术上也更合适。

此外，把该模式解构成两种模式还有另外的材料支持。一份康威定律的文献给出了两种叫法[Cope95]：Organization Follows Architecture和Architecture Follows Organization。这两种叫法非常明确地标识了两种互补的模式，它们针对问题背后的同一种支配力量——康威定律提供了两种不同的方案。

5.2.6 与风格的设计对话

当面对一个特定的设计问题，不管这个问题有多大，只要可以成为解决方案的候选模式不止一个，我们就会被卷入一场设计的对话中。在对话中，设计的优缺点被明确地权衡，利害得失被深入讨论，最终得到一个更充分理解了设计所处上下文及后果的结论。一开始从单一的、独立的模式出发的讨论，到结束的时候把周遭的模式也考虑进去了。

我们在甄选模式的时候会有各种各样的思路，除了那些，还有一种思路是这样的：选择本身决定了一个设计的风格。风格可能被所采用的语言及其之上的编程文化强烈支配，它可能带有某个框架的鲜明特征，也可能是某个系统的局部所特有的。有时选择某个模式而非另一个模式与其说是基于解决问题的考量，还不如说是为了“入乡随俗”。因此，有效的设计包含了某些主观因素和文化因素，这些因素既不能从原则中推导，也不能自动化。

在某种程度上，风格就是主旨。例如，Christopher Alexander多次观察到，一组某种类型的模式促成了某种架构风格，这种风格不同于由另外一组模式所促成的风格[Ale79]。类似地，我们已经看到，在处理迭代问题时，默认采用Iterator还是Enumeration Method的设计，主要是由不同编程语言的特征决定的。

设计选择体现出惯用风格的例子还有：C++中惯用指针或Smart Pointer来表达迭代器。C++历经十年演化，迭代器作为一种对位置和遍历的抽象，其形象一直不甚鲜明，迭代器的概念一直若隐若现，直到标准模板库（STL）降临[StLe95]。在C++中，风格在设计上的体现不仅促成了一套有用的程序库，还带来了一种全新的编程手法：泛型编程[Aus99]。

当然，风格不仅跟语言有关，风格的问题也不总是一个纯粹的主观选择。在20世纪80年代和20世纪90年代早期，非常强调使用继承作为一种软件重用机制。结果，Class Adapter成为在此期间开发的绝大部分面向对象系统的共同特征，当时认为这样是好的。

之后另一种设计学派开始普及，认为转发（有时也被称为“委托”）优于继承。GoF促进了这种风格的发扬，他们的著作中记录了很多围绕对象关系和接口的设计模式，而非偏重于子类化的设计方案。在这种更松耦合的风格中，Object Adapter被认为优于Class Adapter。当然，选择哪种设计风格并不仅仅是个人爱好：正式及非正式的报告都表明基于继承的解决方案复杂度更高 [Hat98]。

5.3 互相合作的模式

把模式紧密联系在一起的，并非只有竞争关系。我们在某些设计中发现，一个模式可能在完整性或（更经常地）在对称性方面补充另一个模式。这并不是说另一个模式总是必要的，只不过找到那个模式经常会带来帮助，而且找不到的情况反而比较少见。

例如，Factory Method引入了一种负责创建的角色，而Disposal Method则扮演相反的角色。在那些关心资源管理的地方（不管是C++中的内存还是Java中更高层的资源），引入Disposal Method回答了被封装的创建行为如何闭环的问题。一个Disposal Method清楚地表明了资源只能被借出，而非真正被获取，并扮演着Factory Method的天然搭档。所以，在使用Abstract Factory或Builder的设计方案中，具体工厂同时提供了Factory Method和Disposal Method。图5-6即是Builder的设计。

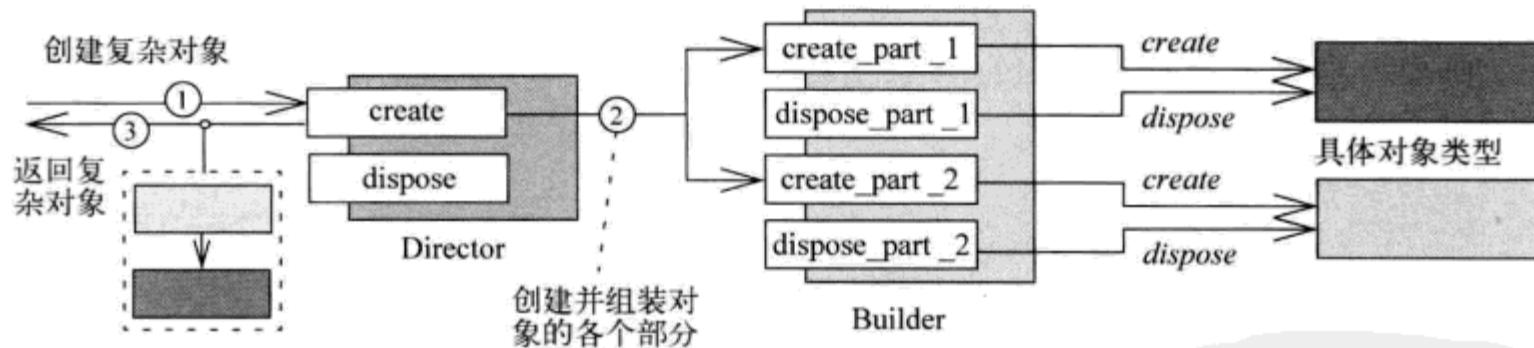


图 5-6

Command Processor模式与Command模式的例子基于一种包容关系而不是类似镜像的对称性。对于Command模式来说，Command Processor的加入经常能使整个设计更完整，带来值得称道的设计品质。如果只有Command，这种品质是缺失的。Command Processor对Command来说不是必需的，但是如果在某些场景下用得好的话，它能给代码带来显而易见的完整性和平衡性。

图5-7展示了一个这样的应用场景。

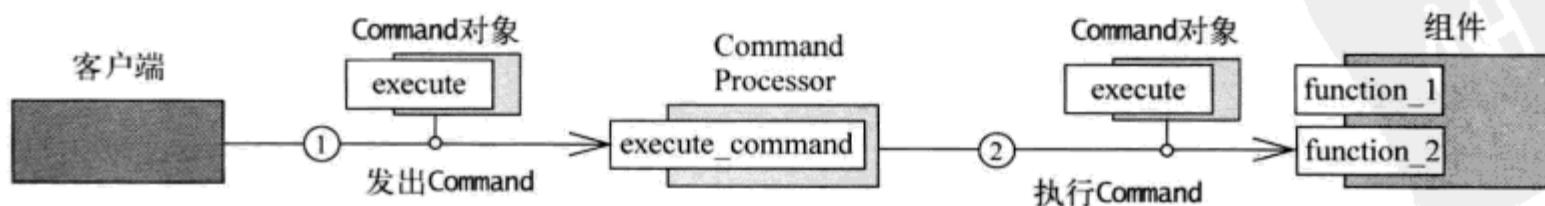


图 5-7

互补性的概念可以被看做是模式调用或模式包含概念的加强形式，其中一个模式被另一个调用或者包含。之所以说互补性是更强的概念，原因是它植根于相关模式之间一种更基本的匹配形式：这些模式或者互为镜像，或者密切配合。

5.3.1 一个关于值的例子

值对象通常按引用保存，并在多个上下文间共享，比如不同线程间。Immutable Value模式解决了基于引用的语言或框架中因为使用别名导致的问题，包括线程安全和线程透明。Immutable Value的设计思路在于：在那些对象被共享的场合（一个用户改变了对象，随后另一个用户检测到这种改变），有问题的地方不是共享，而是改变。两个持有相同数据的对象应该看上去就知道它们有相同的数据。要做到这点，一种保证是去掉所有可能改变值对象状态的方法。因此，通过去掉那些被认为是一个对象接口典型特征的功能，冲突性的因素得以解决，并使最终设计更简单易用。

然而，上面的总结并不意味着设计对话就此结束。如果一个值对象初始化后就始终保持不变，那没什么可说的。但如果存在需要改变值对象的情况，这时不变性自身就成为冲突性因素了。

任何时候，如果需要改变一个Immutable Value对象，要么使用另外一个已经存在的Immutable Value实例，要么创建一个新的实例。在计算比较复杂的情况下，获得最终结果的计算过程将产生大量临时对象，这些临时对象的创建和销毁过程被认为是浪费的，招致不必要的性能开销。

为了弥补这样的缺点，我们引入Mutable Companion，一个与不变类型相伴的可变类型。Mutable Companion不能直接替换其相应的不变类型，它主要用于要求变化累积的或频繁改变的操作。Mutable Companion模式可以看做是Builder模式的特殊形式，它也包含了用于创建Immutable Value的Factory Method，该Immutable Value实质上是Mutable Companion当前状态的一个快照。图5-8展示了这种设计。

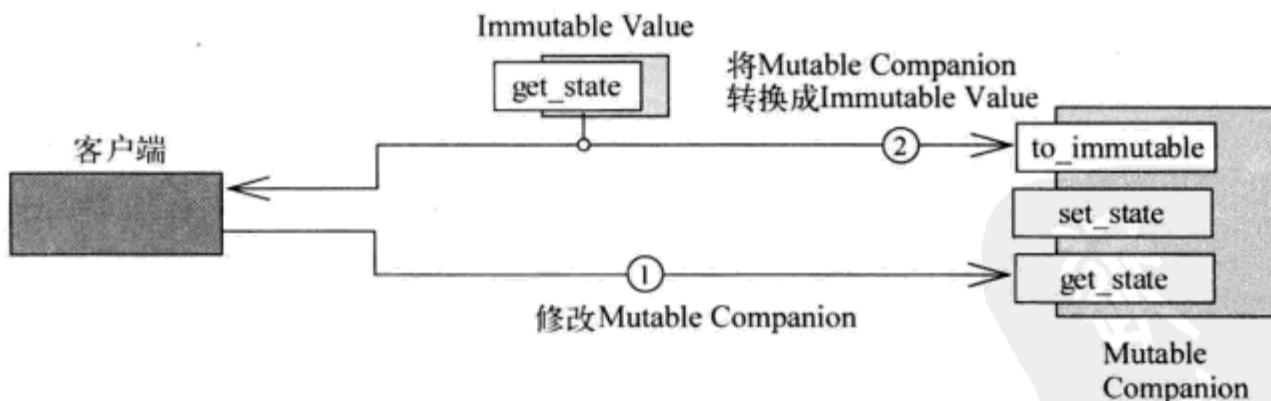


图 5-8

虽然Factory Method也使Mutable Companion得到完善，但是它们是一种在实现细节层次上的使用关系，明显区别于Mutable Companion和Immutable Value之间的完善关系。这是零部件和整体关系与伙伴搭档关系之间的区别。

5.3.2 设计上的完善

有些模式的应用推动了设计，提供了新特性和新的可能性。另一些模式的应用可认为是完善

了设计，这种完善不是那种把设计推向全新方向并获得全新能力的情况，而是使现有的设计变得圆满，协作原设计达到它在逻辑上的终点。

相互竞争的模式以提供可替代的选择来激发一场设计对话，而相互合作的模式靠的是另一种设计中的互补性，即完善关系。这种完善不是说一个模式一定要使用另一个模式，而是一个模式可以选择性地使用另一个模式，并在某些情况下使设计更完整。一种内在的联系使得两个或更多模式合起来组成一种更大的、更平衡的设计，像：Factory Method和Disposal Method、Immutable Value和Mutable Companion、Command和Command Processor。

5.4 模式结合

一开始讨论模式的互补性时说过，不能简单地把模式组合归类为竞争与合作、对立与关联。的确，很多情况下采用这种还是那种模式是非此即彼的选择，也的确有一些模式常常一起出现。但并不等于两种情况之间存在不可逾越的界限。有时它们会有交集，形成一种更像是阴阳相生的互补关系。

回到Objects for States与Collections for States的问题，很明显，在给定情况下这些设计中的某个会成为最佳选择。例如，Objects for States适合这样的场景：有少量对象分别响应外界的请求，伴随着大量状态，并且需要维护由这些状态构成的生命周期。相反地，在Collections for States适用的场景中，大量对象共同响应外界的请求，伴随着由少量状态构成的一个简单的生命周期。以上特征只是在这两个模式之间进行选择的标准的简要总结，但也足够我们处理下面这个问题了：假如有这么一种情况，其中大量对象具有模态行为，一些状态关注于管理者对象，另一些状态只对每个对象自身有意义，这时我们该怎么选择呢？我们可以用Objects for States实现整体的方案。另一方面，对于那些需要给群体进行分类并设计相应行为的情况，Collections for States提供了一种优化方案。在这样的设计中，对象同时具有内部表示的状态和外部表示的状态。我们发现，相反的模式可以被组合使用，而非总是互相排斥。

在讨论康威定律时，我们知道Conway's Law（或Align Architecture and Organization）模式包含了两个互相竞争的模式：Organization Follows Architecture和Architecture Follows Organization。虽然这两个模式提供了不同的解决方法，没有理由把它们看做是对立的。康威定律的表述本身即表明这两个互补的解决方案可以被组合使用：通过对任何一个（或两个同时）的渐进式改变来使二者最终在结构上一致。

5.4.1 再论迭代

当我们考虑Iterator和Enumeration Method这两个明显竞争的模式时，我们可以再次看到类似的对立模式间的组合。这两个模式分别提供了两种截然不同的遍历模型，其中主要的区别在于控制流的方向以及哪边控制主循环。虽然一个集合对象同时提供两种接口是可行的，可以看做是一种面面俱到的设计思路，但实际上非但没有照顾周详的效果，反而显示出相矛盾和犹豫不决。

因此，根据问题所处的上下文，在Iterator或Enumeration Method之中选定一种对外接口，而不要同时提供二者。然而，即使采用Enumeration Method作为对外接口，其内部实现却很可能是

在一个内部集合对象上应用Iterator。在这样的设计中你几乎可以认定，Enumeration Method就是一个给底层“拉”型实体（iterator）加上“推”型实体（callback）包装的适配器。

例如，下面的伪码展示了一个结构化序列对象上Enumeration Method的实现，其中对一个底层链表应用了Iterator模式。

```
class Queue
{
    public push(element) {...}
    public pop() {...}
    public forEachElement(callback)
    {
        iterator = list.iterateFromStart()
        while(!iterator.atEnd())
        {
            callback(iterator.currentValue())
            iterator.moveToNext()
        }
    }
    ...
    private elements = new List()
}
```

另一个更明显地展示了竞争者互相合作的迭代例子是分布式环境下集合的遍历，其中双向通信的时间代价相对高昂。在这种情况下，简单地使用Iterator的话就太细粒度了，因为每次调用都会带来远程通信的开销。不过如果采用Batch Method，又太粗粒度了，因为一次简单的Batch Method调用在处理大量元素访问时会带来巨大的开销。

Batch Method的开销在于准备结果所花费的时间与所要处理的数据量成正比。而且，虽然双向通信的开销只有一次，但这一次的数据传输量很大，因此接收批量处理的结果时就会有可观的时间延迟。调用端在收到结果前将被长时间阻塞。当网络或服务端出错或延迟时，调用端将除了异常外什么都得不到，而且考虑到超时，异常事件本身可能也被延迟。

这样的问题场景不是说凸显了Iterator或Batch Method的设计缺陷或不足，然而它确实说明了在这样的限制条件下这两个模式中的任何一个都不能单独成为候选者。在这样的情况下，有一种解决方案，那就是两种模式都用。

我们从Iterator的核心思路出发，它代表了元素序列中的一个位置，我们可以通过移动它来遍历元素。然后我们沿着Batch Method的思路，它表示了某种获取多个元素信息的单一服务。现在我们综合这两种思路，保留Iterator作为序列中元素位置的概念，但是我们改善一下它的接口，通过Batch Method来定义遍历。图5-9展示了这样的配置。

由此，改善了后的方案既不像Iterator那样一步一个元素，也不像Batch Method那样一步就覆盖了所有元素，而是以一种较大的步骤遍历集合。这样一来，Iterator的粒度变粗了，但是通过Iterator控制遍历的概念得以保留。Batch Method仍然存在，但是定义在Iterator对象上而非目标集合对象上。如今，Batch Method被用来从子序列（而非整个序列）中拉出元素，同时Iterator可以被看做是当前进度的标志。这种设计通过组合对某些问题来说截然相反的方案，更周全地解决了分布式计算环境下的特定问题。

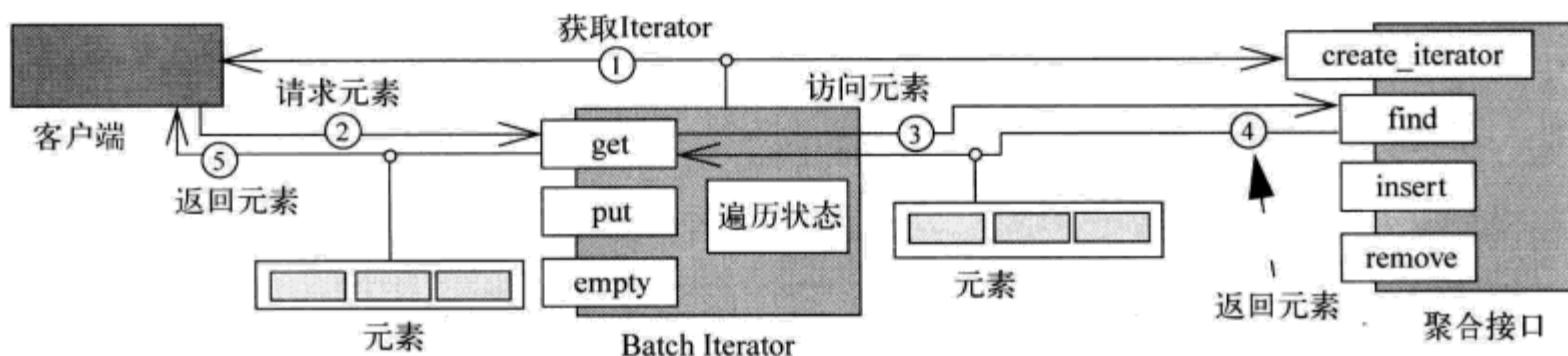


图 5-9

5.4.2 再论适配编程

考虑另一个Java中适配的例子。假如有一个类，我们想要复用它的代码，但它的接口与我们的框架不匹配。我们的直觉可能是使用Object Adapter，其中Adapter类持有一个Adaptee对象的引用，它接收Adapter转发的调用。这个设计一般是没问题的，除非结果证明我们需要覆盖Adaptee类的某些方法，或者需要访问它的受保护特性。

一种可能的解决方案是使用Class Adapter，但有两个缺点。

- 它使所有的Adaptee对象拥有Adapter类的公有接口。
 - 如果Adapter为了加入到框架协作，已经继承了某个类，就没办法继承Adaptee类了。

死路？紧急掉头？为了找出办法，我们这么想：Class Adapter至少让我们把思路放在了我们需要的功能上，所以确实起到了适配的作用。到现在为止一切还好：我们有了一个合适的类。可是，这个类不支持正确的接口，而且已经继承了原始的Adaptee类。怎样才能适应一个有着错误接口的类呢？而且它唯一的超类已经被占用了。答案是使用Object Adapter，使之包括我们已有的Adapter类的对象，这样相应的类现在扮演着Adaptee对象的角色了。

最终的完善方案是把Class Adapter作为Object Adapter的内部类。下面的这个Java类结构展示了整个方案：

```
class ObjectAdapter extends FrameworkSuperClass
{
    public void method()
    {
        ... // 转向合适的Adaptee方法
    }
    ...
    private class ClassAdapter extends SuperClassToReuse
    {
        public void methodToOverride()
        {
            ... // 访问超类保护的功能
        }
    }
    private ClassAdapter adaptee = new ClassAdapter();
}
```

结果是在代码中同时整合了Object Adapter和Class Adapter，通过互补性的两步来解决一个适配器模式的实现。

配问题，并且无论单独用哪个模式都无法解决这个问题。

5.5 互补性：竞争、完善、结合

本章阐明了一件事：我们已经超越了每个模式独自作为设计工具的层次，到达了一个新的高度。某些模式之间存在着一种自然的张力，这种张力源于模式间显著的相似与宝贵的差异。面对这样的模式，我们认为设计中需要有一个步骤去考虑那些竞争者的候选模式，它们将带领我们更深入地理解设计场景。然而，为了作出一个考虑充分的选择，我们需要理解每个选项及其带来的影响[Cool02]。

在另一些情况下，我们看到两个解决方案并驾齐驱，各自致力于解决一个大问题的不同方面。而且，我们看到，互补性的竞争和协作这两侧面并非水火不容。有时，如果一个设计故事有两个发展方向，可以把两个方向都说出来，设计对话可由此深化，设计的发展空间也得到拓宽。

众所周知，水滴汇成大海，但很少人知道大海也汇成了水滴。

——Kabir, 纺织工、神秘家及诗人

本章介绍了模式复合，即可重用的设计模式的组合，其中的设计元素都是可重用的模式！因为设计常常涉及系统很多不同的层面，所以我们在模式的组合及分解中看到这一点也就不足为奇了。无论覆盖了何种粒度的概念，各个设计模式旨在提供进行设计交流的描述和方式。

6.1 常见模式排列

我们常常很明显地看到很多模式结合到了一起，而且当我们进一步深入观察时，会发现这些模式中还包含很多其他模式。例如，Command模式可以结合到Enumeration Method模式中。此外，我们还可以经常看到一些结合了两个及多个模式的应用，例如Command模式结合了Composite模式，毫无疑问，我们可以很自然地将其视为一个整体而命名，例如Composite Command模式。在自然语言中，复合词的含义是两个或更多词的组合，与之类似，我们可以将复合模式和模式复合也理解成这个含义。以化合物做对比，化合物是由多种化学元素组合而成的，这也让你从另一个视角理解组合的含义。

模式复合最初被称为组合模式（composite pattern）[Rie98]，但在模式领域中，这很容易和一个已熟知的Composite模式产生混淆——大声说出“*This design is based on the Composite pattern^①*”和“*This design is based on a composite pattern^②*”，体会一下这两者之间的细微差别。现在，Compound pattern这个词[Vlis98c]比Composite pattern使用更广泛。这里我们使用模式复合（pattern compound）这个词，目的是与后面章节中的概念形成一致的风格：模式故事、模式序列、模式集合以及模式语言等。

6.2 从元素到复合

我们可以将模式复合定义成常见的其他模式的具名的（named）组合方式。它将相互支撑的

① 该设计基于Composite模式。——译者注

② 该设计基于某个组合模式。——译者注

模式组合到一起，定义一种比我们常说的模式元素更大的设计片段。各种模式常常集中到一起解决某个场景下的特定问题，因此模式复合自身就是一个模式，而不是模式的特定应用，同时其中包含的模式也是显而易见的。

这种将单个模式视为由多个模式组成的概念来源于Bureaucracy模式[Rie98]的开篇描述中。

Bureaucracy模式是一种常见的设计主题，常用于实现多层次的对象或者组件架构，允许各层次间相互作用，同时其内部也保持各自的一致性。Bureaucracy模式是一种基于Composite、Mediator、Chain of Responsibility及Observer模式的组合模式^①。

由于模式复合对应于一个特定的结构，因此与单一模式相比，它不需要额外的通用性。尽管能够从多种模式的角度观察模式复合的各个方面，但既然内部各个部分的角色都显而易见，所以在其组成部分的结构和使用上也没有过多的选择。同时，这样的稳定性有利于我们将这种复合视为整体来掌握及讨论，将其等同于单个模式。这样模式复合更容易命名和使用，在开发中能够发挥更大的作用。

6.2.1 Pluggable Factory

Abstract Factory模式常被用于创建一系列相关的对象，即Product。一个Factory提供了一个用于创建Product对象的Factory Method的接口，同时也可能提供用于销毁对象的Disposal Method。一个具体的Factory类提供了创建特定的具体Product的接口，因此我们能够根据不同的Product类型来区分不同的Factory。然而，尽管Factory的使用者与某个具体Factory间的绑定是基于运行时多态性的，并且这样是很宽松的，但是每个具体的Factory还是会绑定到特定的一系列Product上。如何应对由于应用的灵活性而需要更多Factory类型的问题呢？

一种方式是提供多种具体的Factory类型，显式地枚举更多的可能的创建组合。然而，这种方式会带来大量的额外代码，即“理论上的一般性”的副作用[FBBOR99]。此外，这些类中的代码很可能都是重复的，很多类可能都不常用，而且也会有很多可能性没考虑到。提供大量具体的Factory类型并不是解决这个问题的理想方案，所以我们不妨退一步，重新考虑这些问题及可行的方案。

退一步就是Pluggable Factory[Vlis98c][Vlis99]模式复合推荐的方式：加一个间接层。提供一个Factory类型并以Product类型作为参数。这些Product类型作为原型插件，在工厂的创建方法被调用时原型被克隆，如图6-1所示。这种设计避免了公开Abstract Factory的具体实现产生的Product类型，所以抽象和变化都通过可插拔的产品表达。Pluggable Factory可以看做是Abstract Factory模式和Prototype模式的复合，尽管实践中可以使用Strategy模式来替代Prototype模式来实现可插拔。Strategy可以提供一个创建单一Product类型的Factory Method，在C++中，它可以以函数指针的方式传入，在C#中则通过委托的方式传入。

^① 本章引用的[Rie98]中的段落均使用组合模式（composite pattern）这样的术语，注意区分。——译者注

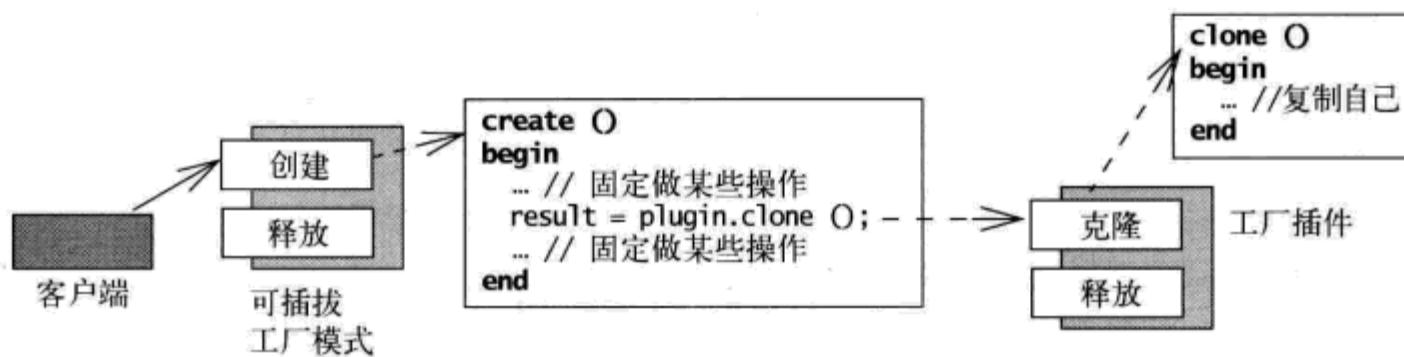


图 6-1

6.2.2 Composite Command 的两种视角

一个模式复合至少包含两个模式。我们可从其内部模式的角度对其进行观察和描述。同时，它必须代表一个合理的、整体的、可重用的设计片段，它是一个通用的、稳定的配置，可以将其自身作为一个模式的角度加以描述，而不用参考其支持的模式。这两种视角都是正确的，但这两者中谁更能有效地描述模式复合呢？

考虑之前提到的较常用的将 Composite 应用于 Command 模式中的例子，这里将一个 Command 以 Composite 的方式实现，这与将一个 Composite 以 Command 的方式实现完全不同。Command 模式和 Composite 模式的组合有很多叫法，比如 Composite Command、Macro Command、Command Sequence 等。这种重现性及它的两套可区分的角色表明了它是一种模式复合 [ViHe99]。下面的模式描述说明了我们如何从一个模式自身的角度展现模式，而不参考组合内部的模式元素及其角色。

Composite Command

假设需要执行某个请求，通常是响应对象上的方法调用，而且请求的执行与何时选中这些请求无关。不仅请求的执行与请求的选择无关，而且有些请求需要组合到一起、按一个组合执行。例如，在多线程中将一组请求视为一个不可打断的序列执行。对于请求的初始化和执行过程中涉及的所有角色来说，单个请求和多请求执行间的差别应该尽可能透明和简化。

那么，将请求封装成对象，通过一个共同的接口来隐藏单个请求和多个请求之间的差别。

创建者定义请求，然而将请求交予执行器来处理和执行。请求对象的创建者决定执行器是将请求分别处理还是将请求视为一组处理。将请求分别处理或者视为一组处理对执行器而言是透明的。创建者可自由按需扩展请求序列，执行器不受创建者的选择的影响。

接口的一致性不仅意味着多请求可以被当做单个请求统一处理，而且单个请求和多请求可以被嵌入到一个请求中。

另一方面，我们可以从模式内部展示模式。

Composite Command

假设需要执行某个请求，通常是响应对象上的方法调用，而且请求的执行与何时选中这些请求无关。不仅请求的执行与选择哪些请求无关，而且有些请求需要组合到一起、按

一个组合执行。例如，在多线程中将一组请求视为一个不可打断的序列执行。对于请求的初始化和执行过程中涉及的所有角色来说，单个请求和多请求执行间的差别应该尽可能透明和简化。

那么，每个请求都是一个Command对象，将多个Command按Composite组合到一起，这样可以将单个请求和多个请求做统一的对象化和处理。

第一个描述是完备的、准确的，但它过于深入细节，而第二个描述与第一个描述有相同的目标和问题描述，但同时从总体上用已有的模式词汇简要、直接地描述了方案架构。对于熟悉组成模式的读者而言，第二个描述优于前者，而对于其他读者而言，第一个描述的切入深度更合理。

6.2.3 模式复合的格式

回答前面小节中的问题，选择何种方式文档化模式复合在很大程度上取决于它所面对的读者群以及对其内部模式的详细描述的需要。关于如何文档化组成模式，基本上有3种方式。

- 用自身的术语文档化模式复合，不引用其内部的模式。这种方式对相对简单的模式复合很有效，因为其内部的模式元素很少，不需要过多文档化。然而，对于有很多角色存在的模式复合而言，如果不引用其内部的模式，则需要对实现细节进行大量描述，这样就不合适了。
- 引用内部的模式来文档化模式复合。这种方式适用于相对复杂的模式复合，简要进行描述，但这种方式需要读者熟悉模式或者手边有很多相关资料。显然，这种方法适用于那些由常用模式（那些在GoF或者POSA书中提到的模式）组成的模式复合，对于由不常用模式组成的模式复合就不合适了。
- 对模式复合及其内部模式都进行文档化。这种方式首先将模式复合的文档视为模式集合，然后对集合中的角色进行说明。这种方式需要作者花更多精力，在某种意义上说这种方式是篇幅最长的形式。这种方式给读者更大的灵活性：他们可以选择哪些内容值得细读，哪些内容仅需要一带而过。同时这种方式也会基于其内部的模式元素对模式复合整体做简要的描述。

第三种方式基本上是第二种方式的一种变型。可以说第二种方式是一种优化，特别是对那些仅包含常用模式的模式复合而言[Vlis98c]。

如果说模式复合有啥优势，那一定是它是基于多个模式的组合。我们没理由把多个模式的描述都囊括到一起，因为读者一般可以分别研究各个模式。如果一个模式复合包括很多模式，它很快就会变得很庞大。所以我们必须引用内部的模式元素。模式复合必须要有附加值。模式复合的基本原则是将读者和作者的注意力都集中到模式的组合效果上。

然而不是所有的模式元素都必须是常用的。因此，我们会考虑怎么描述模式复合才是完备的。对于某种模式复合，如果第三种方式看起来最合适，我们也不必要一一描述所有模式元素。相反，我们应该关注于模式复合中最关键的那些模式元素。所以，我们使用模式复合的上下文来限制描

述每个模式元素的篇幅。例如，如果我们在Composite Command的上下文中描述Command和Composite模式，就不需要包括那些和Composite Command的上下文无关的例子和讨论。

因为对于单个模式的格式的讨论也适用于模式复合，所以我们在第3章中提及的大多数讨论都很适用。然而，还有两个问题需要考虑：命名和图示。

在命名方面，我们需要从整体上考虑，或者使用一种隐喻方式，比方说Bureaucracy模式。我们经常将各个模式元素的名字用连字符连接起来作为模式复合的名字，这样可读性差但却能准确地表达含义。我们可能想找一个相对简洁而仍能表达出模式本质的名字。例如，Command-Composite这个叫法很直接明了，而Composite Command这种形容词前缀的叫法不仅自然而且更易读。有时候还是需要花点精力为模式复合找一个合适的名称[Vlis98c]。

协作要从复合模式^①的名称开始。我们总是可以直接把每个模式元素的名字串起来作为复合模式的名称，但是这样做的附加值很少——如何表达出协作呢？对于含有两个以上模式的复合模式而言，这种命名方式就过于笨拙了。Prototype-Abstract Factory模式^②就是这样一个例子。

一个好名称对复合模式而言是很重要的：名称必须简洁、便于记忆和想起。它应该是朗朗上口的。

“插件式”用在这里很适用，因为它涵盖了动态配置的含义，我们可以看到它是多么合适。我们依然从Abstract Factory的角度讨论工厂。工厂的“抽象”和插件方式是如何实现的（本例中使用Prototype模式）都和本复合模式不再相关。Abstract和Prototype就不需要在名称中显出，只用Pluggable Factory就可以了。

当用图表的形式描述模式复合时，图表能简单地展现出整体的架构以及模式复合的主旨，而且可以进一步对每个模式元素进行描述。例如，在4.3节中使用的方式展现出了各种模式及其在Bureaucracy中扮演的角色，如图6-2所示。

6.3 从补充到复合

第5章阐述了模式间的竞争关系以及合作关系。5.4节阐述了作为相同问题的不同选择，模式也常常可以组合到一起使用。尚未提到的则是这些例子都不是偶然的：它们都是常见的。

6.3.1 重申

回顾5.4节中提到的迭代例子，Iterator和Batch Method的组合是很常用的，我们可以将它视为一种设计词汇——Batch Iterator。下面概述了这种模式复合。

^① 本章引用[Vlis98c]中的段落均使用复合模式这样的术语，注意区分。——译者注

^② 因为里面可能还包含了Strategy、Factory Method等模式，所以不能将所有的模式都列出来。——译者注

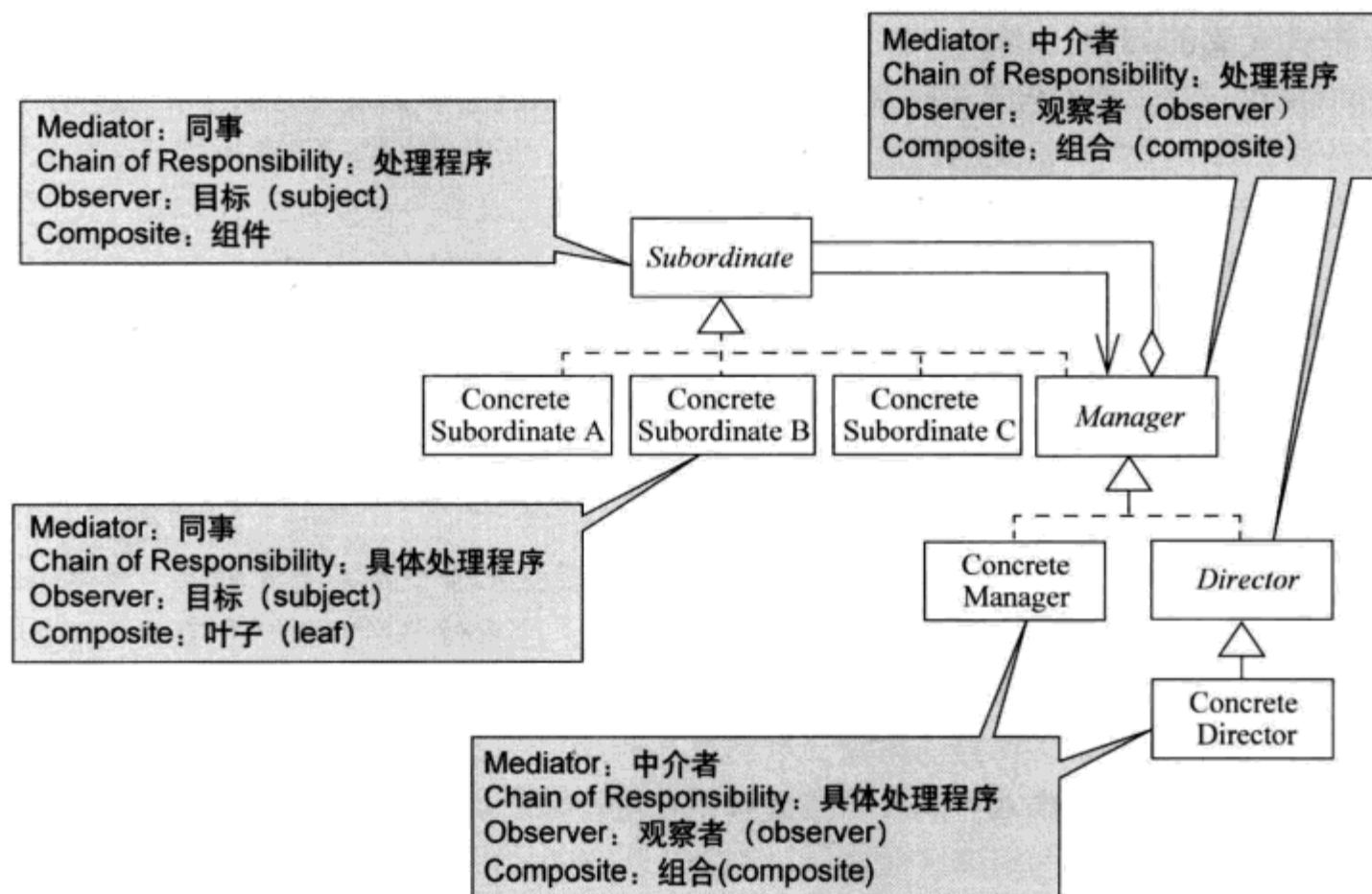


图 6-2

Batch Iterator

在分布式环境中, 使用单步Iterator依次访问远程聚合对象的属性的代价是很高的。然而, 对于大量的属性, 使用Batch Method的代价也是很高的。在前一种情况下, 累计往返时间远远超过客户端和服务端计算所花费的时间。在后一种情况下, 单个数据的传输开销都会导致客户端和服务端阻塞。

因此, 定义一个Iterator, 它可以使用Batch Method大跨步地遍历元素的序列。客户端和服务端将保持响应能力, 同时数据往返的开销在可接受的范围以内。

作为一种独立的设计理念, 这种设计常见于很多中间件框架中, 包括各种在技术上(或政策上)有所不同的CORBA和COM。

下面描述COM中Batch Iterator的动机和风格[Box97]。

为了应对将大数组作为方法参数的相关问题, COM有一种标准的接口设计惯用法, 它允许数据接收者显式执行数组元素的流控制。这种惯用法基于将特别的COM接口指针而不是实际数组作为参数传递。这种特别的接口可称为枚举器, 它帮助接收者将发送者那边过来的数据维持到适合接收者的速度。

下面的微软IDL片段展现了遍历某个连接端点所有连接的标准接口:

```
interface IEnumConnections : IUnknown
{

```

```

...
HRESULT Next(
    [in] ULONG cConnections,
    [out, size_is(cConnections), length_is(*lpcFetched)]
        CONNECTDATA * rgcd,
    [out] ULONG * lpcFetched);
HRESULT Skip([in] ULONG cConnections);
...
}

```

下面的OMG IDL片段展现了遍历OMG命名服务（Naming Service）中命名绑定的标准接口：

```

typedef sequence<Binding> BindingList;

interface BindingIterator
{
    boolean next_n(
        in unsigned long how_many,
        out BindingList b1);
    ...
};

```

这两个Batch Iterator的例子中都包含了获取多份数据的Batch Method。COM Iterator还包括另一种不检索数据的Batch Method——忽略数据，不进行封送（marshal）、发送和解封送。

6.3.2 适配

考虑到这两种解决方案的主题，我们在5.2节中提到，将GoF的Adapter视为两种不同的模式的集合——Object Adapter和Class Adapter——比将其视为一个模式更有效。虽然它们的主题和意图相同，但这两种方案在适用性、结构以及后果等方面却并不相同，所以我们无法简单地将二者放在一起统一进行描述。

在一种设计情形中，Class Adapter的凝聚力和Object Adapter的封装都是必不可少的，5.4节为这种情形提供一种解决方案。Class Adapter适配原有的Adaptee，在需要时重写方法或者访问受保护的特性。由此产生的类被Object Adapter封装，所以在前一个模式中扮演Adapter角色的类现在扮演Adaptee的角色。

这种组合方式已被大家熟知，特别是在Java代码中很常见，因此可被视为一种模式复合。我们可以把这两层解决方案称为Wrapped Class Adapter。Java中的内部类机制有助于实现这两个互相竞争的模式中Adaptee和Adapter的混合角色。

虽然并不互补，但我们可以将Object Adapter和Iterator组合起来作为一种常用的模式。我们可以将其称为Adapted Iterator。在C++STL中，就应用了Adapted Iterator，其中通过Iterator完成迭代，通过Object Adapter将一个接口适配为另一个接口。例如，`reverse_iterator`封装了双向Iterator，因而前向迭代成为后向迭代，反之亦然。

回到Batch Iterator，因为它需要额外的配置，如果要给开发人员提供一个相对简单的Iterator，也会用到Adapted Iterator而不是Batch Iterator。Adapted Iterator处理调用Batch Method而产生的额外的状态和循环，但是其内部仍然采用Batch Iterator这样的大跨步操作结果集。Adapted Iterator

可以处理缓存，并完成对Batch Iterator中Batch Method的每次调用的缓存的迭代。

6.4 是元素还是组合

我们可以发现很多常见的模式在其他“大”的模式中扮演某种角色。所以自然会产生这样的问题：模式一定是要严格划分的吗？它们是元素还是组合？

一方面，从一个模式总是被另一个模式使用或者包含这个角度上讲，原则上，我们可以认为大多数模式都是模式复合，而每个模式借助于其他模式来充分表达。因此，模式的划分可能无意义。

但另一方面，将大多数模式视为模式复合是不必要的。模式复合的叫法一般需要达到某个粒度——比我们考虑的模式集合更粗的粒度。例如，如果我们关注迭代，那么就只用把Iterator、Enumeration Method和Batch Method视为基本模式元素，即把Batch Iterator视为模式组合，忽略Enumeration Method中对Command的使用等复合属性。因此，对于模式的划分，可定义一条兴趣点的基线：低于该基线，不需要考虑细粒度的模式结构，而高于该基线，则将常用的模式组视为模式复合。

6.4.1 组合的含义

如果将组合视为一种“模式元素”，那么Bureaucracy模式、Composite Command及模式Interpreter模式都可被视为模式复合。Bureaucracy模式作为模式复合的典型代表，已经被明确记录并公开发表[Rie98]。

第二个模式复合——Composite Command——被视为Command上下文中一种方便的Composite应用[GoF95]。然而，它的可重用性及可命名性都证明了它不仅仅只是一个例子[VIHe99]，我们可以将其文档化。

第三个模式复合——Interpreter——在GoF的书中[GoF95]也被视为一种通用的模式元素，但是进一步仔细观察会发现，它包含的一些更细粒度的结构可以用模式术语加以描述，在其上下文中的使用更加具体而已。

Interpreter不是一种常用的面向对象的框架模式，而是一种领域特定的模式，它将一些基础的、简单的模式元素集合起来作为一个整体实现了一种简单的编程语言语法。具体点就是说Interpreter的目标是实现一种简单的脚本语言。因此，一个脚本可被视为一个对象，对象既可以反映脚本的语法结构，又可以执行。

将动作具体化为对象的格式是Command模式的本意。要执行脚本，需要一个保存执行状态的环境以及一个符合Context Object模式的需求。一个典型的语法包括递归的整体-部分结构，这必将导致Composite结构。事实上，GoF主要通过将Interpreter的实现和Composite的实现结合起来对Interpreter进行描述。最简单的整体-部分关系即是脚本及其顶层语句的关系，如图6-3所示。

因此，Interpreter可以看做由Command、Context Object及Composite构成的模式复合。我们也经常将Visitor和Enumeration Method与Interpreter的实现联系到一起来完成遍历。然而，这些是可选的补充，不是必需的部分。

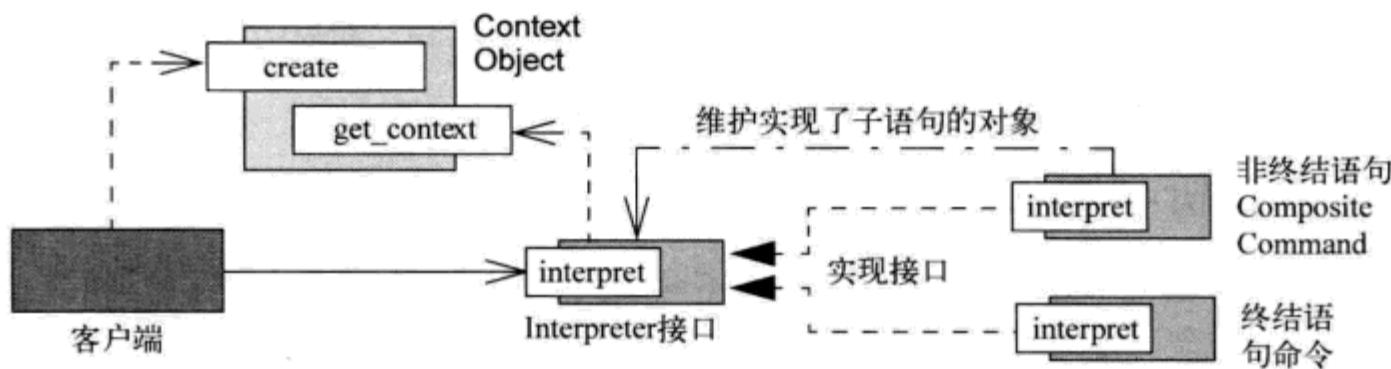


图 6-3

6.4.2 深入 MVC

POSA1及很多之前的图书都将MVC (Model-View-Controller, 模型-视图-控制器) 视为一个单一的模式。然而, 我们也可以从它的各个组成模式入手来分析它。这两种方式并不冲突。下面的分析^①也得出了同样的结论[Rie97]。

6

一个精通软件设计模式的开发人员可能从设计交互软件系统的角度将MVC模式解释为: “(a)MVC模式帮助我们设计有图形用户界面的软件。它的核心是3个交互的对象: Model对象代表着一个领域概念的实例, View对象提供了一个对应于该Model的用户界面展现, 而Controller对象则根据用户的输入来操作Model对象。(b)这三个对象之间的交互又表现为Observer、Strategy及Composite: View观察着Model——因此View是Model的Observer, 而Model是Subject。View不处理用户输入, 而由Controller处理用户输入——因此Controller是处理用户输入的Strategy。此外, View又由展现用户界面更多细节的子视图组成, 子视图又会包含更细一层次的子视图——因此View是一个Composite模式中的组件, 不同的视图要么是Leaf, 要么是Composite。

从MVC的总体上 (以上的(a)部分描述) 看来, 每个设计模式的应用 (以上的(b)部分描述) 都是有目的: 每个模式都用来解决特定的问题。我们将这些模式组合起来以设计一个可重用的、灵活的用户界面架构, 这些模式间的协作所实现的整体远大于单纯的集合。MVC是一种组合设计模式。

GoF未将MVC作为一种模式, 而倾向于Smalltalk特有的视角。然而, 他们在其模式分解方面达成了相似的结论[GoF95]: Observer用于通知, Composite用于嵌套View, Strategy模式用于参数化Controller的行为——可选的Factory Method用于给系统指定一个默认的Controller类, 此外还可能用Decorator来给View增加特性。

POSA1将MVC定义为一种模式, 同时也包含介绍其内部模式的内容, 但它提供另一种模式分解, 这种分解使用更多的模式词汇: Observer、Command Processor、Factory Method、View Handler、Composite、Chain of Responsibility和Bridge。

^① 引文中使用了组合而不是复合。——译者注

这些不同的视角都证明无论模式复合的概念是否为大家所知，将MVC视为一种设计模式并同时从其细粒度的结构方面考虑是有益的。

6.5 复合分析与综合

对于任何一种系统来说，我们都经常看到不同层次上重现相同的模式。例如，系统由子系统组成，每个子系统既可视为一个整体，又可分解成各个更细小的模块（这些小模块还可以进一步分解）。这种递归的整体-分解特性是设计领域中一个常见的主题。我们如此递归地审视系统，是因为一个整体并不仅仅意味着各个部分的和，否则我们就不必支持多种不同的视角了。

将整体独立考虑可以帮助我们看到本来面目，思考并理解整体的结构。当然，通过模式复合，我们得以观察和理解的超越了各个组成部分的和，因此我们希望对其不同的层次进行命名并给予关注。这种“模式的模式”代表对模式及其重现性更深入的思考。

6.5.1 非设计模式的复合

我们可以发现所有常见的例子，包括那些已被文档化的模式复合以及那些显而易见的模式复合，都是关于设计的例子。非设计的例子这么少的原因可能有很多，具体如下所示。

- 我们还没有发现或注意到它们。
- 设计模式和非设计模式间传统上的差异，例如组织式模式及教学模式往往偏向于不同的模式风格和表现方式。举例来说，软件设计模式领域的经典著作《设计模式》[GoF95]倾向于基于单个模式的角度思考模式，而开发过程模式方面的著作（例如*A Generative Development-Process Pattern Language*[Cope95]）则倾向于采用模式的集合，并组织成模式语言的结构。模式复合本身被看做是一种完整的模式，所以它只有在关注单个模式的领域才能找到共同语言。
- 模式复合的结构化特性和固定的角色分配或许不适合关注于人和实践的模式。虽然人员角色间的关系和责任可以用一种空间的隐喻来表现，然而这种展现与设计中存在的结构有很大的差别，硬要这样处理结果是不够人性化的简化，容易忽视人以及人与人之间关系的流动性本质。这种缺失使得以稳定性为主的模式复合更加不适合用作对其进行分组的机制。

当然，这并不是说我们无法从一组非设计模式中找到它们之间的交互关系或者发现反复出现的一组非设计模式。恰恰相反，我们总是能发现这种关系，这正是为关于人员和实践的模式定义的一种主题。例如，Align Architecture and Organization就在Organization Follows Architecture和Architecture Follows Organization之间找到一种平衡。然而，此例中的相互关系并不遵循固定的角色安排：是否倾向于某一种方案，或者将两者并用（以及如何并用两种方案），主要是由反馈驱动的，而很难在模式复合中找到答案。

在文档化非设计模式时，我们会发现很多其他的方式（例如模式集合、模式序列以及模式语言）都比模式复合更能把握适当的变化、重用及反馈。

6.5.2 设计模式复合

在设计中，模式复合的组成部分（相对于基础的模式元素）是可变的，它们更多的是从效用的角度定义的，而不是从模式的内在属性角度定义的。因此，模式复合更像是一个观察一组模式的视角，而不是一个特殊类型的模式。例如，John Vlissides在确定模式复合时，做了以下的论述 [Vlis98a]。

Composite-Strategy-Observer，即著名的Model-View-Controller，源自于Smalltalk世界。Buschmann & Co.将其视为一种架构模式[POSA1]，但是将其视为一种组合模式似乎可以简化讨论，也不会带来什么不利影响。

如何及何时将一个模式视为模式复合主要取决于在哪个层面上对模式进行描述。根据定义，任何可命名的常见的重复设计片段，只要其粒度比较粗，都可能成为模式复合。对于一个模式集合来说，如果文档化做得够细，就相当于把原子进行分裂；如果粒度够粗，模式复合本身就可以看做是原子。

在实践中，模式复合的概念并不那么容易接受。我们一般倾向于使用那些已被人们所知并广泛使用的名称和视角。作为一种分析方法，模式复合的概念是一种很好的工具，可以用于识别一致性、分组或在给定的模式集合中给模式排序。作为一种综合方法，它还可以从设计者熟知的部分着手，提供展现设计思路的新方法。

尽管有点相对主义的嫌疑，我们还是有必要重申模式复合并不是常用模式的随意结合 [Rie98]。

组合模式（composite pattern）首先是一个模式：它提供了一个常用于特定领域的设计方案。我将其称为组合模式，是因为它最应该被理解为若干模式的组合。然而，组合模式并不仅仅意味着组合：它还意味着协作，模式在整体组合结构中扮演着不同的角色。正因如此，组合模式不仅仅是模式元素的简单堆砌。

回顾我们提供的实例和描述，复合一词在语言上的隐喻可能更胜于在化学方面的含义。复合词语及短语——实际上正是自然语言中词语和短语演化的方式——相对于化学上的隐喻显得不那么神秘或者更接近量子力学的视角，化学变化往往意味着原子分裂或元素变化^①。既然我们把模式看做设计词汇的基础以及设计对话中的要素，对于模式的思考自然引入更多与语言学相关的对比。

^① 原文如此。通常认为化学变化中原子是最小的不会变化的单位。——译者注



老实说，自从我长大并且变得足够聪明，能看出隐藏在寓言中的寓意时，我就一直不喜欢寓言。我更喜欢历史，不管是真实的还是伪造的，对于读者的思想和经验来说，历史具有宽泛的适用范围。我认为很多人将“适用”混淆为了“寓言”，其实前者的自由权在于读者，而后的自由权则是作者有意强加的。

——J.R.R. Tolkien

本章建立在我们之前在这一部分中所讨论过的模式关系基础之上。模式故事提供了一种表述格式，能够描述模式是怎样已经被（或将要被）具体应用以创建一个特定的系统，实现一个特定的功能，或去转换一个给定的情况。模式序列把这种描述归纳到了一个更基本的层面，提炼出使用这些模式的实际顺序，因此同样的序列可以被应用到其他地方。要讨论模式序列，我们就要再一次回到上下文的问题上来。

7.1 模式讲述软件工程成功的故事

我们不能从某一个单一的视图来看一个软件系统是什么或者它是怎么被开发出来的，也不存在这样的“特权”视图。每一个视图都强调了一个侧面，这个侧面对于别的视图可能是不可见的，并且传达了一个有些细微区别的信息。举例如下。

- 以代码为中心的视图。它强调细节和准确性。虽然现代编程语言已经支持模块化结构表述，但是代码中所表现的大多数关系在这个层次上来说是不明显的。举例来说，类和继承强调了该语言支持面向对象。不过，类之间对象到对象的关联却没有相应的语法结构来表示，这种关联需要仔细阅读代码或借助一种合适的工具才能发现。
- 图表化类模型视图。将关系网有区别地映射到系统中，强调类以及所有关系——不论类之间是关联还是继承——而忽略类中的具体细节。这样做的目标是为了了解依赖关系和宏观结构，但这就不可避免地以损失精确性和精细结构为代价。试图将这些细节引入类图当中是CASE工具和建模人员的一个通病。无论一个类模型是画在一个信封的背面还是基于某种底层存储模型而显示在屏幕上，如果它充斥着应该被抽象掉的细节，那么它的读者就会对它失去兴趣（甚至失去活下去的意志）。

□ 以模式为基础的视图。以描述软件系统的架构和基本原理的方式，强调了用来共同组成一个软件系统而解决的问题和引入的角色。该视图可以和软件的其他模型和视图相关联，或者作为它们的注释。它也可以作为一个单独的描述而存在，无论是存在于一个文档之中，还是在需要对系统进行描述时将该视图中的某一部分重新单独拿出来作为注释。

一般情况下，以模式为基础的视图处于以代码为中心的单元和类模型视图之间，关注组合、质量和相互关系。它不需要关注那些能够构建系统的各个程序块，而是关注组成系统的成分，这些成分也就是那些程序块所要完成的任务。从某些方面来看，拿烹饪做比喻比用建筑来比喻更贴切：各种成分是如调料一样相互混合并相互影响的，这与那个由钢筋、玻璃、砖块和木头构成的世界不存在直接的相似之处。

上面对3种视图的简短描述只是对它们的一个简单说明，而不是对它们的详尽总结——要把所有可能的视图都详细描述一遍，甚至能写成一本书[Kru95][Zach]！从某个角度来说，每一个视图都只是重复描述、扩展或者忽略了其他视图的某些特征。它们还有一个更普遍的基础：它们都是静态的。它们都是某一事物出现在某一时刻时的一个快照，这些快照能够捕获不同的东西，例如：

- 存在于版本控制系统中的代码；
- 反映已写好的代码或者期望将要写的代码的类关系；
- 已经被使用过的模式。

巧合的是，类模型也被认为是静态的，因为它并不能描述一个和时间相关的、正在运行的系统的行为。然而，我们感兴趣的静态-动态差异是和系统开发相关的，而不是和系统运行相关的。

7.2 模式故事

听说使用了67种知名模式和24个元模式的系统实现是非常令人印象深刻的……但从本质上说这没有什么用。从一个系统所使用的模式清单去理解这个系统或许比从一个以字母表排序的类列表去理解要更好些，但这是一个令人望而却步的想法。我们从哪里开始呢？

有很多种方法可以着手处理这个理解的任务，例如第8章中所讲的多种组合方案。另一种方式是采用讲故事的方式。由于它的教育性并且对用户友好，这种方式变得流行起来。讲故事的方式将系统开发看做一段叙述，其中描述了逐步运用模式创建一个系统的过程。

回到那个烹饪的隐喻，准备各种材料的先后顺序以及对材料的组合和材料本身是一样重要的。随着故事的展开，对系统的设计和对原理的解释也随之展开。比起用模式清单来展示一个系统架构，这种方式肯定更易于理解、也更合理。

7.2.1 一个小故事

我们可以重新回到第4章提到的请求处理实验，并对它的第二次设计进行扩展，这样我们就能引出一个简单短小的故事，并将设计中的决策过程提出来。

我们正在开发一个可扩展的请求处理框架。怎样发出和处理请求，才能使该框架能够显

式地处理请求？

将请求标准化为Command对象，该对象基于执行客户请求的通用接口方法。可以在类层次结构上对Command类型进行描述。系统用户靠实例化一个具体的Command类并调用其执行接口来发出特定的请求。然后该对象在应用层上执行被请求的操作，如果有结果的话，将其返回给用户。

实现该框架的语言应该是一种静态类型语言。对于大部分或者是全部的Command类来说，可能存在一些通用的实现。什么样的Command类层次结构是最好的呢？

应该将所有类的根类表示为Explicit Interface。框架和用户都可以将它作为一个稳定、公开的接口，将它与影响其他层次的实现决策解耦。具体的Command类实现这个Explicit Interface。通用代码可以放到Explicit Interface的下层抽象类中，而不应该放在类层次的根部，具体的实现类在层次中可以表示为叶子。

系统中可能有多个用户，他们可以独立发出命令。通常应该怎样协调命令的处理呢？

Command Processor提供一个中央管理组件，客户将他们的Command对象传递给该组件，由其对它们进行进一步处理和执行。该Command Processor仅仅依赖于Command类层次结构中的Explicit Interface。

同时，Command Processor还可以很容易地引入回滚机制，以便针对请求作出的响应可以被取消。在Command的Explicit Interface中增加undo方法的声明，这样在实现类具体化的时候都会产生影响，并可以由Command Processor进行管理。

在引入取消机制的同时，公认的做法是提供一个重做机制，这样前一次被取消的Command对象可以被再次执行。Command Processor要怎样做才能更好地维护Command对象的取消记录和重做队列呢？

将Collections for States加入到Command Processor中，这样就可以用一个集合保存已经被执行过的Command对象——因此这些Command就能够被取消，另一个集合保存已经被取消的Command对象——因此这些Command能够被再次执行。这两个集合都是序列，访问时遵循“后进先出”的原则。

某些操作可能很容易被取消（或重做），但是有些操作可能包含重要的状态改变，这样就会使回滚（或前滚）操作变得很复杂。我们需要一个简单、统一的回滚机制，同时也需要处理一些复杂和多样的操作，那么如何兼顾这两种需求呢？

可以选择将每个Command和一个Memento联系起来，该Memento保存了在这个Command执行之前相关应用状态的全部或部分副本。需要使用Memento的Command类型在设置和使用Memento的状态时将会共享一套通用的结构和行为。这种通用性可以在依次实现Command的Explicit Interface的抽象类中进行描述。然后以Memento为基础的Command类型继承这个抽象类。那些不以Memento为基础的Command类型将不会继承这个抽象类，它们转而直接实现Explicit Interface或者继承其他适合他们的抽象类。

框架需要为请求提供一个日志机制。如果允许用户选择他们想要的日志处理方法而不是固定的处理方式，应该怎样设计日志功能？

以Command Processor的Strategy模式提供日志功能，这样框架的用户就可以通过为Strategy接口提供合适的实现来选择以何种方式记录请求。有些用户仅需要使用标准的日志选项，因此框架需要提供一些预定义的日志类型，反之，其他用户可能希望设定自定义日志。

鉴于可配置的日志系统不会对运行中的框架做什么功能上的改动，在框架里我们如何将其尽可能透明地实现？

为日志Strategy提供一个Null Object实现。当被调用时，该日志对象不做任何操作，但是与有具体操作的日志实现使用同样的接口。这种利用多态性进行选择的方式保证了框架不需要修改控制流结构就可以适应这种可配置性。

框架需要支持复合请求。复合请求是指顺序执行的多条请求，它们作为一个整体要么全部执行，要么全部取消。在现有的基础设施中，怎样才能在不打乱原有简单、统一的Command处理方式的基础上实现这个需求？

将一个复合请求实现为一个聚合了其他Command对象的Composite Command对象。要正确地初始化一个Composite Command对象，必须按照顺序将其他的基本Command对象或Composite Command对象添加进来。

我们用几段简单的描述讲述了这个模式故事，其中使用和问题-解决方案对相对应的请求-应答模式。还可以通过代码段和图表对它进行补充，不管是以图标的方式显示应用模式的顺序，还是以图表的方式显示由每个模式形成的类图。这个故事抓住和传达了设计思想，这个设计思想可以说是隐藏在框架结构之下的，这比用一个简单的包含所使用模式或所涉及类的列表来表达要更深入、更相符、更合理。

7.2.2 已经发表的故事

很多已经发表的例子说明了故事怎样被用于展开一个设计或者揭示一个情况是怎样发生变化的，他们展示了作出的决策和在做决策时使用的模式。如同一些简单的激励型例子可以为单独模式所做的那样，模式故事也可以为模式集合做到：把它们变得更生动，并用实例说明它们实际上是怎样工作的。这种对例子的交织使用可以追溯到Christopher Alexander关于建立体系结构的著作中。举例来说，在*A Pattern Language*[AIS77]中，他用一个简短的、说明性的例子给出了一些设计中的基本步骤。在*The Oregon Experiment*[ASAIA75]中，我们能找到一个包含更多基本原理的更长的版本。

我们在软件模式中也看到了一些讲故事的情况。软件开发的一个基本属性不可避免地要求使用例子使一些抽象的东西具体化。在大多数情况下，故事被认为是具有教育性并且包含丰富的附加信息的，而不是用来组成模式概念的一个部分。然而，就像我们考虑将实例的使用作为一种形式而又不是简单的流于形式一样，我们对模式故事所扮演的角色也持相同的看法。

例如，在《设计模式》[GoF95]一书中，就通过文字叙述的方式说明了Lexi文本编辑器的设计，突出了使用设计模式进行设计的过程。书中逐一提到了下面这些领域的问题：文档结构，格式编排，美化用户界面，支持多种外观感觉的标准，支持多窗口系统，用户操作，拼写检查和断字。在设计过程中应用以下模式逐一解决了上面提到的问题——Composite、Strategy、Decorator、

Abstract Factory、Bridge、Command、Iterator和Visitor，这些模式的顺序和前面列出的问题的顺序是一一对应的。

Lexi故事的目的就是将模式的概念和在实际中怎样应用模式介绍给读者，并展示出真正的设计模式。然而，为了使故事和范围更易于处理，这里只介绍了少数几个特征和模式。如果将文本编辑器中涉及的所有设计模式都包含进来，那么整个章节的篇幅就会变得过长，而且文章中的叙述也就不那么吸引人了。

在*The Design Patterns Smalltalk Companion* [APW98]一书中，就从字面上表达了说明性故事的想法：一个简短的小说化的戏剧。

它包含3个场景：两个为MegaCorp Insurance公司工作的Smalltalk程序员3天的生活。我们听到了Don（一个面向对象的新手，不过是一个资深的商业分析师）和Jane（一个面向对象和设计模式的专家）两个人之间的对话。Don因为设计上的问题来找Jane，然后他俩一起解决了问题。虽然人物是虚构的，但是设计是真实的，这些设计是使用Smalltalk语言写的实际系统的一部分。

使用第一人称的叙述，并且加入了代码，这就为个人提供了在实际情况中使用设计模式的指南。举例来说，*Pattern Hatching* [Vlis98b]一书通过一个层次式文件系统的例子让读者明白了设计需求、问题和C++解决方案。*Smalltalk Best Practice Patterns* [Beck96]一书则通过一段处理多种货币汇率的Smalltalk代码向读者展示了基于模式的代码改进。

经验丰富的记者从现实世界中寻找故事。*Organizational Patterns of Agile Software Development* [CoHa04]一书简要介绍了4个这样的故事，每一个都展示了所涉及的一系列模式。在*Fearless Change* [MaRi04]一书中，使用了4个不同作者的故事来说明在实践中怎样组合和使用书中提到的引入组织变革的模式。

POSA系列丛书也使用了模式故事来说明怎样进行模式组合才能产生应用。

- 在《面向模式的软件架构（卷2）》[POSA2]中提到的JAWS并发Web服务器框架的系统结构是通过一系列模式来进行说明的，其中每一个都对应于一个使用问题-解决方案描述的上下文。
- 在《面向模式的软件架构（卷3）》[POSA3]中提到了两个故事：一个支持移动设备和分布式服务的ad-hoc网络解决方案，以及一个支持移动电信网络的系统架构。
- 《面向模式的软件架构（卷4）》[POSA4]则用一个很长并且很详细的故事描述了一个多重抽象层次数据仓库中过程控制系统的开发过程，包括它的基本应用的系统架构及其通信中间件的基础设施。
- 最后，在本书中，我们不止一次地讲述了一个请求处理框架。

7.3 从故事到序列

通过理解那些按照时间顺序被应用的模式的方法，我们可以捕捉到一个系统的演化，这种观点将我们从摄影带到了电影拍摄当中。这也提出了一个重要的问题。很多故事都是虚构的，在现

实中不需要具有任何坚实的基础[PaCle86]。但从工程师的角度来说，我们最终想了解的正是关于系统的事。这里有冲突吗？

从某方面我们可以说，很多模式故事在现实世界中是完整存在的。对于很多故事来说，它们抓住了所发生事件的细节的精髓（虽然不一定是事实）。我们很难轻易地使我们的设计思想符合命名分类和时序排列，所以必须有一定的回溯修正。回忆一段在POSA2和POSA3中都使用过的引自哲学家James Burke的话。

历史很少按照正确的顺序或在正确的时间发生，但历史学家的工作就是使它看起来像那样。

在这里，故事序列的准确性并不是问题。如果故事在正确的地方结束，那么它的目的就是作为沟通工具，故事在构成现存系统的各种重要的（和不重要的）决定和细节上穿梭而行——不论系统是好的、坏的还是丑陋的。就像Galadriel's Mirror [Tolk04]中所说的那样，这些故事可能描述系统的过去、现在以及将来。作为一种教育和建模的工具，它能比传统的案例研究揭示更多的东西——而如果能够将我们建议的这种描述应用到案例研究中去，案例研究将是非常有益的。

在渐进式成长的过程中着眼于每个部分

模式故事的实用性也暗示一些比设计的渐进式成长更深刻的东西。如果我们能够通过逐步地应用模式的方式来描述系统开发的故事，那么是否能用这种方式，而不是总体规划设计和大模块开发的方式来开发系统呢？就像上面所提到的，系统开发可以被看做是一个描述性的例子，在这个例子当中关于设计的问题被提出并被解答，各种结构也由于某些特定的原因被组装起来。因此，这种以经验为基础并且具有互动性的描述也可以被合理地作为开发的一种有效途径。

如果我们能把某些设计过程看做是对特定模式的应用过程的话，那么转过头来将这种观点再次集中到模式之间的关系上，又会怎样呢？到目前为止，在本书这一章中所讨论的模式关系主要集中在它的范围和使用上，偶尔会提及它的先后顺序，但是没有具体的建议。虽然前面对模式的描述中我们使用了故事作为比喻，但是对于模式之间的描述关系，我们却没有更深入地进行指导。我们还把模式比作一种词汇表，将注意力集中在它的适用性上，但是，这次我们也没有讨论怎样才能在最佳的顺序下使用最好的词语。

7.4 模式的序列

怎样一个接一个地应用特定的模式，这在很多系统中都属于设计的一部分，或者在很多环境下是因为情况改变而发生变化。我们很希望序列具有一种可以重复并且能够被描述的能力。没有必要通过描述整个系统的故事来找到其中有用的子序列。无论是整个序列还是部分序列，都能像单个模式那样从现有系统中提取出来，并且作为该架构的部分被展现出来。

就展示方面来说，模式故事其实就是描述：它可能包含一些图表，但它实际上应该是文本化的，而不是符号化的。简单的序列可能包含我们前面讨论过的协作模式，将其展现在一个更大范围的上下文中，说明怎样由一个模式引出另一个模式以迎接相应的设计挑战。如果出现多个模式

可供选择的情况，当两个选择都被认真权衡时可以视作戏剧张力的体现，亦可以考虑为故事的分叉点，从此发展出两个迥然不同的故事。

模式故事有各种展示形式，模式序列亦是如此。可用简单线性的应用模式的列表来表示[AIS77] [BeCu87] [Hen05b]，也许可以同时对模式加入一些注释。另外，模式序列也可以用一个包含了对所使用模式的描述的表格（例如缩略图）来表示，因此特定的问题和解决方案就是可见的了[ASAIA75] [Ale04a]。

7.4.1 一个早期的例子

在提到序列时，在特定时间顺序下应用模式一般来说是被轻描淡写的或者隐晦的，而不是特别能引人注意。虽然它不能像其他模式概念那样吸引人，但是按照一定顺序应用模式的重要性很早就已经成为了模式传统中的一部分[AIS77] [Ale79]，甚至在软件模式的领域也是这样。

例如，下面的一段描述是从早期的软件模式论文中摘抄的，它仔细描述了一个模式序列。

考虑一个设计Smalltalk窗口的很简单的模式语言。我们建议使用以下模式：

- (1) 每个任务一个窗口；
- (2) 每个窗口仅有几个面板；
- (3) 标准面板；
- (4) 短菜单；
- (5) 名词和动词。

我们将这组模式交给一个小组，其成员都是为一个特殊用途的编程环境写说明书的专家。他们不需要详细了解任何Smalltalk的接口机制（例如MVC），而只要通过一天的实践就能够确定合理的接口。请注意，我们为这些模式编了号并排了序。模式1必须首先被处理。它决定了什么窗口是可用的以及将在这些窗口中做什么。其次，模式2和模式3将每个窗口分割成了面板。最后，模式4和模式5决定了在每个面板中应该做什么选择和动作。这个顺序是由各模式之间影响力拓扑结构决定的。

模式故事提供了特定的描述，与此对应的模式序列也就提供了描述的模型。它构建了我们能够讲述的故事，同时也将它和相似的其他故事区分开。

7.4.2 模式序列既是流程也是物件

然而，比起一遍又一遍地讲述模式故事来说，不管这些故事多么有价值，模式序列都有更多的作用。如同每个模式一样，模式序列既是流程也是物件。序列代表了由更小的流程组成的流程——其组成模式按照一定顺序来组合，以实现特定架构和架构特征或者实现有预期属性的环境变化。

就像一个单独的例子可以说明一个特定的模式那样，我们可以将模式故事看做是一个扩展的例子，它不仅可以说明一系列模式，而且还可以说明一个模式序列。因此，模式序列就是设计决策和转化的相继流程。在最极端的情况下，我们甚至可以将单个模式看做是只包含一个模式的序列。

我们说过单个模式可能是完整的或是不完整的：这个特征是否也适合模式序列呢？当然，基于不完整的模式的模式序列是不可能完整的。但是由完整模式构成的序列呢？在我们对模式概念的研究中，不断反复出现的推断是模式不能被任意应用，不论在其他情况下这个模式是多么有效。结果就是，就算模式序列是由完整模式组成的，它也有可能是不完整的，就像由简单和相互关联的类形成的一个框架，它本身也可能是复杂的、低效的。

当谈到序列时，怎么才能区分这个序列是好的还是坏的呢？当然，如果去解决错误的问题，这就是问题。例如，后续模式可能会解决一些没有表现出来的驱动力，也可能错过了已经表现出来的驱动力。我们也可以去关注这样一个属性：是什么决定了位于整个过程中每一阶段的设计在某些方面是否是完整并且稳定的。

健康的模式序列应该是这样的：其中每一步都代表了一个合理的、非常稳定的结构。虽然它不必是最终的设计，但是它不能没有明显的后续步骤或者处于一种不稳定的中间状态——这种状态既不能被实现，也不能被轻易描述出来。序列代表了通过设计空间的一条路径，我们当然更愿意这个序列中的每一步都是尽量完整和稳定的。

再次重申，我们现在所达到的程度是通过渐进式成长的方式来进行增量开发。无论它是一个工程浩大的全系统开发还是一个关注部分类的适度开发，它都可能受目标支配。然而，实现这个目的的每一个步都必须脚踏实地，而不是在一个巨大的前期设计中信心满满。模式序列中的每一个点都是对反馈的响应，是由前面的设计产生的上下文。遵循模式序列更像是遵循一个菜谱，而不是遵循一个计划——如果你烧得一手好菜或是对某个菜谱很熟悉，我们鼓励即兴创作。

当然，就像我们已经知道的那样，一个模式是不可能适用于所有情况的，模式序列也是这样。怎样在相关的模式序列之间进行选择，这将把我们引到模式语言上来，这是本书第三部分的主题。

7.4.3 再次回到以前提到的小故事

回到那个请求处理系统的故事，我们能发现存在于它内部的序列。序列和特定上下文的细节已经被详细叙述过了，但是我们可以提炼上下文细节，抓住过程的主线。

Command是通过Explicit Interface来表示的。然后引入了Command Processor，并为它添加了Collections for States。接着又将Memento扩充到了Command中，然后使用Strategy细化了Command Processor，这又引出了Null Object。最后引入了Composite Command。

这个摘要有点像一个剧本的故事概要，略去细节，只在抽象层面上讨论它。例如，一出三幕剧的情节可以被这样概括：首先是开场，介绍主角、配角和这出戏剧的主题；第二幕随着剧情的发展，主角和现实情况有了冲突；在最后一幕中，有高潮、宣泄和结局。这虽然不是一个能够使潜在的戏剧迷去抢购门票的概要，但却忠实地描述了过程。

我们甚至能够更进一步、更正式地将我们的请求处理故事的模式序列总结归纳为：

<Command, Explicit Interface, Command Processor, Collections for States, Memento, Strategy, Null Object, Composite Command>

然而，模式序列并不是一种简单的练习活动，它的存在是有目的的。上述序列只突出描述了模式序列的一个方面，而为了能够应用到实践当中，我们需要在开始时对上下文、目标和我们想要构造的目标属性作出更多的描述。我们还会想要归纳由每个模式引入的不同角色之间的相互关系——只是简单地说一个模式跟着另外一个模式，这并不能告诉我们如何归纳。我们甚至想要给这个序列命名，因此我们能够在我们的同事之间提起它并简明地表达它。

7.5 回顾模式复合和模式互补

之前我们已经讨论过它们了。我们可以将模式复合看成是子序列：在这些小故事里，已经能够清楚地知道情节和情节中的元素，角色和相互作用也基本上被确定好了，因此我们不必知道所有的细节。当在6.3节中对这两种描述Composite Command的方式进行比较时，我们当然也就证明了这种相似性。

7.5.1 重组

在承认序列很重要的基础上，我们就能发现不管这个模式复合的名字是什么，最能表达Composite Command的二元组是<Command, Composite>而不是<Composite, Command>。为什么呢？这是由完整性和稳定性决定的。Composite细化了Command，这就意味着Command是最先被引入的。如果需要的是对Command的抽象，那么Command本身就是稳定的设计，但是如果失去了这样一个有意义的上下文，那么Composite指的是对什么进行Composite呢？这就是这个模式应该被称作Composite Command而不是Commanded Composite的原因。

我们也能在生成代码的过程中看到这个问题。Command开始是一个简单并且完整的步骤，紧接着会有Composite的实现。反过来，如果只以一个含糊不清的Composite的概念开始，那么是什么也写不出来的。对于一个写好的Command接口来说，一个Composite，甚至是任何其他类型的具体命令，都可以被实现。而对于一个含糊不清的Composite概念来说，什么也无法被写出来——什么东西的复合？

这个短小的序列同时也强调了角色兼容性和集成的重要。我们不是在一个模式之后简单地应用另外一个模式，而是应该看角色之间的兼容性以及它们是否能被集成起来。举例如下。

- Composite模式为某些组件类型定义了一个递归的整体-局部方案。在这个方案中，以通用组件类型的类层次为基础创建了具体的组件。
- 当一些对象的主要用处是执行任务时，Command模式为这些对象定义了一个类层次结构，其中各具体的命令能够实现在根层次上引入的命令接口。

首先引入了Command，这样不仅解释了类层次结构的概念，而且也确定了明确的、有针对性的组件类型。然后引入了Composite，我们简单地实现了一个具体的Command，并可能会根据反馈对根接口做些修改。这些角色吻合并指明了后续步骤。

如果能重新分析一下4.3节，可能会注意到虽然里边有一个叫做CompositeCommand的类，但Composite Command模式的名称却从来没有被提起过：随着故事的展开，应用的模式是Composite。这是一个完全不同的故事吗？还是只是用不同的方式讲述了一个同样的故事？

我们应该问的问题是，设计结果是否有什么不同。答案是没有：在早先的情形中，Composite是在最后被应用并且被集成到最先引入的Command上的。而在最近的叙述中，Composite Command是最后被应用的，然后被集成到最先被引入的Command中。这样形成的结构是一样的。在应用Composite Command时，所有的Command角色都已经存在并且被集成，所以就只需要引入Composite方面的组件。引入模式复合的价值是很明显的：它进一步明确了哪些模式角色被加入和集成。回忆那个被归纳为如下形式的序列：

<Command, Explicit Interface, Command Processor, Collections for States, Memento, Strategy, Null Object, Composite Command>

把它归纳为下面这种形式也不失精确：

<Command, Explicit Interface, Command Processor, Collections for States, Memento, Strategy, Null Object, Composite>

在这种情况下，使用模式复合不会使最终的设计结果有任何改变。它只是简单地解释了Composite的组件角色是对于Command而不是Strategy来说的，其实Strategy也被应用到了模式序列中，而且它的角色也能够和Composite兼容。当然，我们也能够提出一种更花哨的符号来表明角色是怎样组合的，但是请记住我们的目标是为了明确事实而不是为了显示我们有多聪明。如果我们想展示的是角色引入的过程和集成的过程的话，散文可能是比尖括号更加有效的方式。

7.5.2 再论 Batch Iterator

再次回到5.4节，在此节中我们首次有意地将有竞争关系的模式进行组合。即使没有强调，我们也能看到应用模式的顺序也是很重要的。

从Iterator的核心概念出发：Iterator代表了在一系列有序元素中的某个位置，并且可以为了遍历元素而移动。接着提出了Batch Method的核心概念：Batch Method代表了可以跨越多个元素的单个服务。现在将它们组合起来：保留Iterator在一系列元素中代表某个位置的概念，同时使用根据Batch Method定义的遍历方式来细化它的接口。

换句话说，Batch Iterator能够被归纳为< Iterator, Batch Method>而不是<Batch Method, Iterator>。第一种设计代表了一个更加稳定的过程：引入Iterator，然后使用Batch Method细化它的接口。

相反，对角色和代码来说，第二个序列却有很明显的干扰：首先在集合上引入了Batch Method，然后又由于Iterator而将其放弃，并且在Iterator上重新实现了Batch Method。将Batch Method作为第一步会对接下来的步骤产生错误的上下文。但这也并不是说这个序列永远不会在实践中使用：如果有人首先引入Batch Method，然后发现这不是一个正确的方法，那么这个序列对他来说就是适合的。然而，总地来说，模式序列<Batch Method, Iterator>并不是我们认可和提倡的序列。

7.5.3 再论 Interpreter

在6.4节中，我们分析了Interpreter并将它分解为了模式三元组：Command、Context Object和Composite。这个顺序正好反映了它表面上的顺序，但是它内在的顺序呢？在这个例子中，它们是一样的，我们可以通过详细描述该复合背后的故事来证明。

我们需要为一种简单的脚本语言开发一个Interpreter。给定的脚本是在客户端程序内部被访问执行的。如何以编程方式操作并执行一个脚本？

将整个脚本表示为一个Command对象。引入方法接口，这些方法用于执行脚本，并且能够执行客户端应用在使用脚本时需要的任何辅助查询和修改。

在客户端应用的整个生命周期内，给定的脚本很可能需要不止一次的、以独立于其他任何脚本的方式执行。脚本也得能够接收或回传一些环境设置和操作，这样的脚本才是有用的并且能够正确执行。运行中的脚本怎样获取对这些外部状态和行为的访问？

用封装了必要上下文的对象来表示相关状态和行为，将它作为参数显式传递给代表脚本的Command对象。如果需要的话，执行状态能够被传递给另一个脚本，或者被保存下来以备后用。

该脚本语言遵循一个简单的类似代码语句和表达式结构的语法。最简单的脚本可以是一行代码，或者可能是最简单的表达式。该语言的语法也可能包含上下文无关的语法。在运行时应该怎样表示该脚本呢？

根据语言的语法，分解整个脚本的Command对象，这样的话每个可执行的结构就都是一个Command。语法中的包含和递归通过Composite来实现。在运行时，当前的执行状态在Context Object中被保存和传递。

我们也可以用图表的方式来说明这个序列，由3个模式步骤中的每一步来表示设计过程，首先是Command，然后引入Context Object，最后将Composite应用到Command当中，如图7-1所示。

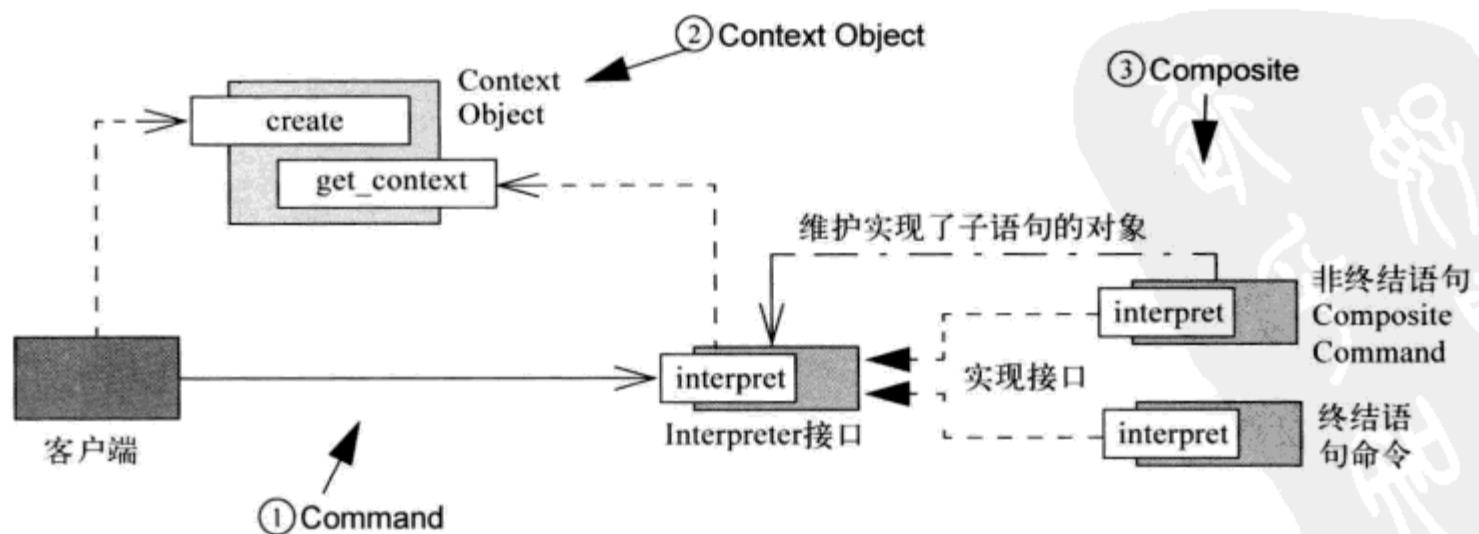


图 7-1

简而言之，图7-1代表了下面的元组：`<Command, Context Object, Composite>`。这个路径代表

了这3个模式最优、最稳定的实现方式，而使用其他的路径也可以达到同样的结果，但是那样会需要更多的重构，走更多的弯路。

现在，考虑一下如果我们希望在请求处理框架中加入脚本化的请求处理，这时将会发生什么。模式复合保持不变时，将得到如下的序列：

<Command, Explicit Interface, Command Processor, Collections for States, Memento, Strategy, Null Object, Composite Command, Interpreter>

如果对复合进行扩充，就能得到如下的序列：

<Command, Explicit Interface, Command Processor, Collections For States, Memento, Strategy, Null Object, Composite, Context Object, Composite>

Interpreter和现有的Command模式角色结合在一起，因此Context Object与首先被应用的Command模式的角色有关联。然而，有趣的是，Composite是重复的。在两种情况下，它都可以被认为是Composite Command的一部分。不过，由于涉及脚本的语法，在第二次应用中递归的组合并不是对任意Command对象的组合。它精确描述了负责根据语法来表示脚本的Command对象类型，并和该Command对象类型结合在一起，而且它不需要通过引入Composite来实现对任意Command对象的分组。

7.5.4 再论 Align Architecture and Organization

模式复合代表了一个受限的并且相对固定的模式序列结构。其他序列在它们自己的结构中可能会更加灵活。例如，在5.2节中，我们探讨了康威定律的含义，即它是作为驱动力存在于开发组织和与它对应的软件结构体系之间的。Conway's Law模式提出了一个通用的解决方案，对该方案比较好的描述是Align Architecture and Organization。然而，通过更进一步的分析，我们发现在这个高层解决方案中存在明显互补的具体模式：Organization Follows Architecture和Architecture Follows Organization。

在实践中，对于这两种互补的模式来说，我们能够使用哪些模式序列呢？在极端的情况下，我们选择哪种都不使用，结果就是一个空序列：<>。然而，如果这个架构被证明是稳定的，那么围绕架构来组织开发就会更合理，这样产生了有些孤单的只含有一个模式的序列：<Organization Follows Architecture>。还有另外一种选择，当架构还不是很稳定但组织却是不可变更的时候，可能更适合使用另一种方式：<Architecture Follows Organization>。

对这种情况的一个更开放、更符合反馈驱动的响应有可能是“通过对两者做一些小的改动来将架构的结构和组织的结构统一起来”[CoHa04]。这种方案产生了一个系列的序列，而不是一个单独的模式序列。这个系列的形式就像这样：<Organization Follows Architecture, Architecture Follows Organization, Organization Follows Architecture...>或者<Architecture Follows Organization, Organization Follows Architecture, Architecture Follows Organization...>。

7.6 回到上下文的问题

就像我们在第1章中重点提到的，对于是否应该将模式的上下文算作模式本身的一部分，有时是有争论的。既然我们已经更深入地讨论了模式间的关系，现在就能够更好地来重新审视这个问题了。

如果上下文决定了模式的适用性并且是模式本身的一部分，那么这就存在失去模式的通用性的危险，而且使其固定在了某个问题的解决方案上。但是如果上下文不是模式的一部分的话，那又是什么决定了模式的适用性？模式并不是一个能应用在任意地方的任意结构，所以如果没有上下文引导的话，它的存在就显得漫无目的。

在Christopher Alexander的著作中，我们看到他是非常重视上下文的，其中我们或许还能找到一些说明[Ale79]。

每个模式包含一个三步规则：一个存在于特定上下文中的关系，重复地在那个上下文中出现的一些特定的驱动力，以及使这些驱动力得以解决的特定软件配置。

这个定义看起来的确表明了上下文是模式的一部分。但引用这段话并不仅仅是为了引起争议。它并没有解决我们上面提到的那个矛盾。

7.6.1 定义上下文

更进一步的研究显示，很多我们认为是关于上下文的问题其实可以简化为术语问题。真正的问题并不是上下文是否是模式的一部分：我们需要提出的问题应该是“什么是上下文？”

简单说来，模式的上下文确定了模式能够在什么地方被应用。但这意味着什么呢？假设模式用来处理由某些特定驱动力产生的问题，那么这些驱动力和模式的上下文之间肯定有联系。如果上下文不是模式的一部分，那么这些驱动力也不是。然而，如果这些驱动力是模式的一部分——它们肯定是，否则模式就变成解决方案了——那么它们肯定是基于上下文的，并且是从上下文中引出来的。因此，上下文不能从模式中分离出来。但是，从某些方面来看，与在一个模式序列中考虑模式的上下文相比，将模式的上下文作为单个模式来考虑是一种更通用的做法。

那么，如何定义模式的上下文？正如我们在1.5节里探讨的那样，一种方法就是通过详尽列举，也就是列举出所有可能产生需要由该模式处理的设计问题的程序状况。遗憾的是，那将是一个长得没完没了的列表。另外，永远没人能看懂它。

举例来说，第1章的Batch Method的例子可以应用在迭代访问代价很高的很多情形中，其中包括分布式和并发系统、内存数据库存取，甚至那些只能靠例子而不是详尽列举来表达的范围。既然我们的兴趣在分布式和并发系统上，我们目标当然要集中在这两种系统的上下文上——做到但求一样（或者两样）精，不求样样通。

在模式序列中，如果模式应用在其他模式的上下文中，可以将该上下文的描述缩小来把它定义为这个模式的先行模式。在既定的故事里，所有细节都是固定的，从而序列的概述可以进一步被限定。讨论中的上下文现在就很具体了。不过，这并非所有模式的描述，而是具体序列里模式

的描述。通过删除对无关的驱动力和结果的描述，更具体地描述解决方案，我们可以选择使它更细化。

例如，Batch Method可以被应用到Iterator的上下文中，这样我们得到Batch Iterator模式复合。当然，如我们所知，Batch Method不光是能用在Iterator后面。然而，在Batch Iterator的序列中，Batch Method的其他属性或许并不相关，甚至是多余的。

对于某些模式作者来说，上下文代表一种具体的情形描述，在描述该情形时，列出相关的所有模式，这些相关模式都可能用到作者所关注的那个模式。我们知道在模式序列中这个列表比较短，因为模式关注的范围被大幅度缩减了。在不同的序列中，缩减版的上下文可能会不同。为了让它更直接并增强它的适应性，我们可能会去重新描述、重新命名或者重新构建这个模式。

虽然这种看法反映了在某种特定约束条件下模式的使用情况，但是它却没有表达出模式所特有的模式本质部分：它是在不同的情况下反复出现的。当我们提到模式时，什么才是其中稳定不变的因素呢？这里列出了一些选项。

- 是解决方案结构的本质吗？当然。
- 是互相冲突的驱动力的本质吗？当然是，因为如果不是的话，我们就会有一个可以循环使用的解决方案模式，而不用去了解驱动这个模式的问题类型。
- 是上下文的本质吗？这个也许就是关键：它是对上下文必要的解析，而不是一个重复发生的特定列举。它是对上下文的分类或描述，而不是在故事中或甚至在语言中相对具体的实例。

如果不借助一些看起来一般的内在基本特征，不提及驱动力的来源，我们就很难说模式定义了重复出现的一组驱动力。

一个模式序列或者甚至多个包含同一模式的模式序列，都能够将上下文的范围缩小到能够被命名和列举的程度，但是它自己没法完全定义模式的范围。当然，尝试列出模式所有可能的用法使模式概念的实用性大打折扣，这并不是我们想要的。为了给出通用的上下文陈述，我们打算使用上下文的一般属性来代表其种类——使用内涵的方式而不是外延的方式——通过列出上下文的成员的方式来定义。

例如，一般而言，我们可能按如下语句来定义某个模式的上下文：“……你使用了Active Object将并发任务引入Broker，并使用Monitor Object来序列化访问通用资源……”这在特定的序列中更精确地定位出了模式，同时也确定了它在设计中的角色。然而，关于上下文的一个普遍看法是它不应包含太多细节，但又要足够准确，例如“……系统或者系统的一部分存在着某个活动的并发线程，它们同步访问某些对象……”对于关注包含Active Object、Broker和Monitor Object的模式序列的用户来说，这个看法没有太大用处，但对于那些想在模式序列之外考虑模式的人来说却很有用，无论是自由地使用它还是将它包含到自己的新故事之中。

随着对模式的表述变得越来越具体，它们的上下文也变得越来越具体了。对模式来说，最简单也是最直接的联系形式是：在有限的范围内列出一组模式，然后再如同上下文的定义那样将先行模式列出。然而，要想在封闭系统之外表示上下文，最合理的形式不是直接使用设计步骤本身，而常常是使用归纳前一个设计步骤产生的上下文属性这样的常规方法。

7.6.2 专用化与差异化

当从范围的广度和专一性上来考虑上下文时，我们能从不同的视角来看待模式如何专用化。比如Proxy，它是在分布式系统的很多部分都会用到的通用模式，包括：

- 在防火墙的处理层，也就是Firewall Proxy；
- 在类的层级结构中，经典的“四人帮”（Gang-of-Four）Proxy例子；
- 在类似于Client Proxy的分布式对象模型中，用来调用远程对象操作的stub客户。
- C++中特有的Smart Pointer形式，管理运行时对象的生命周期或者它的同步。

每种情况通常都会根据对Proxy的使用有一个关联的名称，但在一般术语和用语中，它们还是都被称为Proxy，而不是这些特定的名称。在所有这些情况中，Proxy怎么都是同样的模式呢？实际上，在具体的情形中（比如设计Firewall Proxy和定义Smart Pointer），差别还是非常明显的。另外，基本的上下文也相当不同：连接在因特网上的进程不同于可以独立运行的程序中的C++代码。关于Proxy的通用隐喻适用于上面提到的所有情况，而且我们也能够明确给出一个适合Proxy的上下文描述。然而，这样一种描述或许过于抽象，以致对于大多数开发者和解决方案的设计来说并不实用。

与最一般的情况相比，缩小范围（也就是上下文的范围）允许我们讨论更多关于特定驱动力、解决方案结构和最终结果的问题。这同时也允许我们根据需要对模式进行重命名，例如Firewall Proxy和Smart Pointer。虽然这种想法通常被认为是就专门化和普遍化方面而言的，但是从缩小或者扩大上下文的实用性的角度看，有时这或许是更有益的。

有一点很明确，那就是除了人口模式之外，模式序列中每个模式的上下文都是很具体的：先行模式。作为写在模式序列上下文中的模式描述，这对于读者来说就足够了。它们可以自由抽象出不受上下文约束的模式，但是这样产生的模式可能无法被模式描述所支持。对于单独文档化的模式，读者最感兴趣的是上下文的特征，因为它们限定了模式的应用范围。

7.7 模式间的联系

本章主要关注使用或思考模式时序列的重要性，这里的“序列”表示模式的“句子”。更广义地说，本书的第二部分强调了模式是外向的、协作的和社会性的。当然，我们不要低估单个模式的价值，就像我们不能低估单个词语的力量那样。然而，就像词语的力量源于它们所处的上下文——包含它们的句子、段落和章节，模式最主要的价值也是当它们组成一个团体时显现出来的。

正是模式的这个网络反映了单个及多个设计的本质，也反映了那些设计元素不只在一个方面相互作用的系统的复杂性的本质。合成、覆盖、增强和平衡的概念根据它们自己所扮演的角色，影响不同的设计元素。这些概念强化了一个让某些人初看起来违反常规的设计思想：在一个设计中找到的组件也许不是最能代表这个设计的历史、基本思路或者未来的。

就像一幢房子是由不同材料建成的那样，是那些材料的相互作用而不是那些单独的材料本身使房子成为家。因此，那些在一个情景中看起来不错也行得通的在其他情况下可能就是错误的。比如，一套新安装的隔板可能会使之前安装的画作看起来有点歪。同理，新添加的一种事件可能

会将现行的事件处理模型变得不易于编程和优化。在这些情况中，部分是不能被孤立考虑的，否则就会丢失某些有价值的观点。

紧密的联系

这还不足以说明模式之间相处融洽，或者它们关系紧密。必须给这种关系概念引入一些精确的描述。回顾一下本章和第二部分之前的几章，就会发现模式之间有两种基本类型的关系对描述架构和刻画情形最有用。

- 空间关系。包含的联系，用来描述格式。
- 时间关系。时间上的顺序，用来描述进程。

当这些基本的成分和可选性、对立性以及抽象结合起来的时候，我们可以从简单被动的设计词汇表进入更深远的境界：主动语言，它真正服务于沟通，有意义、有对话、有反思。本书第三部分探讨了这类语言的形式、内容和属性。

科学即事实；正如房子由石头构成，科学是由事实构成的；但是正如一堆石头并不是一座房子，一堆事实也未必构成科学。

——Henri Poincaré^①

在前几章中，我们学习了如何将模式关联起来构成设计，并影响设计活动。本章中我们来看一下将多个模式放到一起考虑和组织的相关事宜，并探讨一本软件架构手册应该如何组织。本章关注的是模式集合（collection）与编目（catalog），它们承担了模式仓库（repository）的角色。为了有助于组织这些模式仓库，我们还提出了问题框架（problem frame）技术来描述模式应用的上下文和模式要应对的问题，同时从符号学的角度研究模式的结构和关系。

8.1 模式手册

模式本来就是要组合起来使用的。在任何重要的设计中，不论是有意的还是无意的，都不可避免地会用到多个模式，实际上独立的模式几乎是不存在的。将多个模式组织在一起的最简单的形式就是模式集合。

迄今为止，我们所知道的最牛的模式集合是Grady Booch的*Handbook of Software Architecture* [Booch]。该项目的目标是完成最初由Bruce Anderson在一本架构手册中提出的设想，这在0.1节中提到过。当时，*Handbook of Software Architecture*列出了大约2000个模式，其中绝大部分只有很简略的描述。

大部分集合在详细程度和模式数量方面做了更多的平衡，通常会关注某个具体问题类型、某种上下文或者某种系统，模式的数量也通常不超过几十个。每个集合的目标和媒介也有所不同。有些集合比较通用并编纂成书（比如*Pattern Languages of Program Design*系列[PloPD1] [PloPD2] [PloPD3] [PloPD4] [PloPD5]），有些则是围绕社区形成的并且通常是在线的，比如*Pedagogical Patterns Project* [PPP]。

集合、编目和仓库

用来描述模式集合的术语有很多。比如，集合本身可能就是用得最多的一个。它经常用在某

① 儒勒·昂利·庞加莱（1854—1912），法国数学家、理论科学家和科学哲学家。

篇论文中表示大量模式。相反，编目则经常用在更大规模的情形中，比如书或者网站，它强调的是索引的重要性。对于在线编目，特别是如果由多个作者共同维护，也经常使用仓库这个术语，比如最初的Portland Pattern Repository[PPR]。这些用词并不严格，它们的精确含义可以很容易从上下文中推断出来。

前几章中，我们已经把重点从单个模式转移到小规模的模式族或者模式群上来。我们所展示的每个概念都有相应的关注点，主要解决某个问题或者某个问题空间。要作出一个百科全书式的架构手册，只关注某些方面是不够的——我们需要一种组织方法。

8.2 组织模式集合

组织模式集合有可能只是为了自己，也可能是业务需要或者出于出版的目的：作者可能为了更好地理解某个特定领域而将一组模式组合在一起；某些模式可能是跟某个组织的业务相关的，将这样的模式结集出版可以帮助我们更好地分享领域知识。有时候我们将模式组合在一起，只是为了帮助编写和整理其中的某几个模式。并非只有模式的作者（或者称为编纂者）会这样做。模式的使用者也经常这样做，比如某个团队可能将自己在开发某个系统的过程中使用到的设计实践和风格组织在一起形成一个模式集合。这正反映了软件模式作为一种架构手册的本意。

对于编纂者来说，集合是否有一个有效的组织模型是非常重要的。组织编目有多种方式。模式可以组成临时的集合，没有特别的架构或者统一的主题：比如会议纪要。模式也可以根据某个特定的目的组织在一起，比如在展示特定架构中的模式时，按照该架构所应对的问题和分区来组织模式。不同的方式未必互相排斥，一个集合可以根据意图做纵向切割，也可以跨级别做横向切割。与之类似，从规模和抽象的角度看待模式与从问题域的角度看待模式是互补的。

8.3 即时组织

即时组织（ad hoc organization）并不等于随意组织。但是它确实意味着如何组织以及包含哪些模式更多的是应激式的（reactive）而不是预先设计好的。给定一组模式，什么样的组织架构是最恰当的分组方式呢？

Pattern Languages of Program Design 系列图书 [PLoPD1][PLoPD2][PLoPD3] [PLoPD4] [PLoPD5] 的每一卷给出的答案都略有不同。每一卷的内容都是从某一次 PLoP 会议上提交的模式里面选择的，所以几乎不可能恰好符合某种分类学规则。有些模式是根据领域分组的，有些是根据层次，有些根据意图，等等。每卷都不一样。除此之外，这些编目在结构上是松耦合的：在模式之间很少有交叉索引，甚至连表现形式也各不相同。

索引对于任何集合来说都是用的。对于即时编目来说，由于其缺少整体性的组织模型来引导读者，索引就显得更加重要了——编目越大，索引的重要性也越明显。*Pattern Alamanac* [Ris01] 可能算得上是同时代最大规模的索引和编目了。其目的并不是展示模式，而是根据对 2000 年之前发表的大部分模式做一个总结。很显然它的目的就是要创建一个年鉴，其内容就是模式，除此之外没有别的什么能把这些模式联系到一起了。年鉴的编目只能选择即时组织的形式，将所有现存

的模式组织在一起。

模式集合的组织形式是即时的，但是仍然要能够方便读者阅读。这些模式整体上很难说是组成了一个互相联系的群体，更像是零碎的、偶然拼凑在一起的单个模式或者模式组。

即时集合的一个潜在问题是它看起来好像是为了模式而鼓励模式。它没有明确的目标读者群——只是对模式感兴趣的人，它并不针对特定的领域，比如分布式系统开发、项目管理或者Java中的多线程编程。组织很难从即时的模式集合中真正受益。所以，我们并不将其作为通用的组织架构鼓励大家去使用，但还是在一些情况下人们希望使用这种即时的组织方式，比如会议纪要、通用的参考资源等。

8.4 根据层次组织

最早的模式分类架构中有一个是根据抽象、粒度或者规模的层次划分的。这类架构最常见的例子便是《面向模式的软件架构（卷1）》[POSA1]。这个方案将模式按照粒度划分为3个层次：最粗粒度的称为架构模式（architectural pattern）——有时称为框架模式（framework pattern），中等粒度的称为设计模式（design pattern），最细粒度的称为惯用法（idiom）。同样的划分方式在其他地方也有应用，比如在*Pattern Languages of Program Design*系列图书[PLoPD1][PLoPD2][PLoPD3]中对架构模式和通用意图的设计模式做了区分。

8.4.1 设计和架构

有些人仅仅根据术语学的观点对这种3层结构提出了质疑。构建架构和成功使用编程语言惯用法从某种程度上讲都是设计活动，因此它们是不是都应该归为设计模式？确实如此，既然我们已经对模式的概念做了深入的研究，我们可以将设计模式看做是为了跟组织或者教育学等模式区分开的一个概念，当然这里也包括其他无关的“模式”，比如针织样式（knitting pattern）、墙纸图案（pattern on wallpaper）、正则表达式的模式匹配（pattern matching）、图像处理中的模式识别（pattern recognition）等。然而，设计模式这个词却无法将多种不同的与设计相关的模式区分开。

现在看来，我们在POSA1中可以将设计模式定义为“与GoF的《设计模式》书中模式的粒度相似的模式”，这固然精确却又不够简洁。同样，鉴于所有的模式从本质上都具有某种架构的性质，那么凭什么这种粒度称为架构模式，而另一种粒度就不是呢？显然，它们从某种程度上都具有架构的性质。除非说是大家对架构的理解跟“PowerPoint架构^①”中的用法类似——“架构是幻灯片上的大框”，否则单独将一种粒度称为架构就没有什么意义。架构最终包含并横切所有层次上的细节：它不应当单独考虑。

下面关于设计和架构的定义或许能给我一些启示[Booch06]。

作为名词的设计，指的是系统中具名（named，虽然有时很难命名）的结构或者行为，它用于解决或者有助于解决系统中某个或者某些驱动力。设计代表的是潜在决策空间中的一个点。设计可能是单独的（叶子决策，leaf decision），也可能是集合（一组决策）。

^① PowerPoint架构是指PowerPoint模板中的主题、布局、样式等。——译者注

作为动词的设计，指的是作出这种决策的活动。给定一系列驱动力、一组资源、一个可以施展拳脚的地方，最后的决策空间可能是很大、很复杂的。设计既是一门科学（基于经验的分析可以告诉我们理想的方案大概在什么地方，甚至直接给出完整的答案）又是一门艺术（除了基于经验的设计决策，我们还有可能关注优雅、美妙、简单、新奇和聪明）。

[……]

所有的架构都是设计，但是并非所有的设计都是架构。架构代表的是重要的设计决策，对于系统具有决定性的影响，这里所说的重要性主要指的是修改的成本。

因此，如果人们认为架构只是关于系统设计中最重要的决策，而不是所有的设计，那么很明显对于给定模式来说，它有时候可以称为架构，有时候则不可以。因此，是不是架构就不是模式的内在本质，而是应用的属性。从这个意义上讲，我可能更应该区分模式所扮演的角色——战略性的（strategic）、战术性的（tactical）还是辅助性的（logistical）。

举例来说，如果在整个程序的状态机解释器实现中采用Objects for States模式，它就扮演战略性角色，影响整个程序的架构。如果只是用来实现某个连接对象的状态生命周期，Objects for States就是作为一种战术决策来使用：如果将其换成Methods for States，对其他架构决策没有太大的影响。当然，很多模式明显可以归为某一类：Null Object很少扮演战略性角色，而Proactor则与之相反。

8.4.2 惯用法

8

对于惯用法，也有同样的问题。它主要关注语言用法和约定俗成的风格[Booch94]。

惯用法指的是某个编程语言或者应用文化所特有的表达方式，代表一种约定俗成的语言使用方式。

[……]

惯用法显著的特点是，一旦忽视或者违反了惯用法，人家就会认为你是个粗人，甚至是外行，不值得尊敬。

惯用法的概念在自然语言中非常常见：作为一种跟编程语言相关的设计技能，这个概念是由Jim Coplien [Cope92]推广开的。后来，惯用法也指特定于某个编程语言的模式。然而，有时候，惯用法保留了其最初的语言学含义，只是一个约定俗成的符号，与之相对的是问题的解决方案。很多时候，惯用法兼具这两种特征。所以，这个术语很模棱两可，因为有时候它表示解决问题的实践，有时候它仅仅表示约定俗成。

我们在POSA1中使用惯用法表示某个编程语言中的模式，但是进一步考查后我们发现还有其他的情况。有时它也指在某个特定上下文（领域、粗粒度的架构划分或者技术）下的模式。编程语言只是解决方案领域的一个例子，它里面的实践只是整个实践空间中的一部分，所以如果必要的话，我们最好明确地采用编程语言惯用法（programming language idiom）这样的术语。

有些实践在某个特定上下文中也可以称为惯用法，但是并不能算是编程语言模式。比如，在

分布式对象计算系统设计中，Batch Method模式用来完成迭代，但是在其他上下文却未必合适。而且，有些模式被认为是通用的而不算是惯用法，但是只要对编程上下文做少许变化，它可能就属于惯用法了。举个例子来说，使用Iterator来完成迭代往往被认为是通用的，然而，在Smalltalk中它就是通过Enumeration Method来作为惯用法出现的。根据原来的分类方式，在C++和Java中Iterator应该定义为惯用法而不是设计模式。当然，有些情况下惯用法所指的东西在不同的语言中有很大不同。比如，Iterator在C++的标准模板库与Java中的惯用法表达就是不一样的。

惯用法这个术语也存在滥用的现象。就我们的理解，在自然语言中，惯用法代表的是某个文化群体中常用的语言模式。有些打上“惯用法”标签的模式，实际上在其所在的领域内用的并不多，甚至知者寥寥。如果这些问题反复出现，把它称为模式也就罢了，将其标榜为惯用法，说好听点是可能造成混淆，说难听了就是不厚道——因为惯用法这个词暗含着“好”的意思。我们经常听到某些人提出“新惯用法”，这种自相矛盾真可谓令人称奇。他们提出的可能是一种新技术，但是在得到广泛接受之前打上“惯用法”的标签可能太早了。

8.4.3 混合层次

Jim Coplien也指出了这种3层结构的问题[Cope96]。

由于抽象的层次本身是连续的，没有明确的边界，[……]所以将模式按照3层结构来划分从这个层面上看是有些随意的。这可能使得分类过于主观。

除此之外，还有更深层次的原因。许多模式是贯穿3个层次的。就拿Model-View-Controller来说，起初它是Smalltalk文化中的一种惯用法。随着它越来越得到人们的广泛理解，它变成了面向对象用户界面设计的主要方式。Buschmann等人[POSA1]将其作为框架层面的模式，因为它是系统最高层次的基本结构原则。这样的模式很难根据3层抽象模式进行分类。

我们发现很多模式可以归为3层中的任何一层。比如，在分布式系统中，Proxy模式就是这样的。如果看一下它的各个变体，就很容易发现在系统层面我们有Firewall Proxy，在基础设施层面有Client Proxy，在编程语言API中有Smart Pointer。所以这不仅是粒度的问题，同时也是我们打算以何种方式描述架构的问题。

8.4.4 层次

虽然3层抽象方式很吸引人，并且也有不少例子支持，但是将其作为一种通用的或者灵活的方式来容纳所有的情况是不合适的。我们可以断定，任何一种层次架构都无法做到将模式以一种一致的方式进行分类，要不就要在模式这边妥协，要不就得在层次架构上妥协。独立地看这些术语都没有问题，但是把它们用作分类方式就不合适了。比如，架构、设计和惯用法作为专业术语都有各自的价值，只是不适合用来描述不同的模式层次。

如果我们将重点放在某个特定的应用上下文，这种层级和包含结构可能更有价值。对于给定的系统来说，对它们所包含的模式进行分层是相对容易的。针对一个特定的上下文，给定的模式很容易知道它是战略性的、战术性的还是某个范围内的辅助方案。在这种情况下，按照架构或者

意图进行划分更有意义。

8.5 根据领域组织

领域 (Domain) 这个概念有时很宽泛, 但最常见的定义是符合某个规则的一个范围。规则这个词有很多种解释, 所以领域这个词在软件开发世界里也有不同的含义。

从最宽泛的角度来说, 我们可以区分开问题领域和解决方案领域。问题领域 (也称为应用领域), 包括真实世界中的各种行业, 比如电信、金融、医疗、航空电子、物流、教育等, 我们可以按此划分模式。有些模式可能特别适合分解问题或者建立架构, 并且适用于上述各种问题领域。从其观点和应用来看, 我们可能认为这些模式是以用户或者客户为中心的。当看解决方案领域的时候, 我们就进入了 (虚拟的) 机器王国, 我们看到的是软件的内部结构, 考虑的主要是跟软件相关的东西。解决方案领域的模式包括与特定编程语言相关的惯用法、平台技术、架构风格以及面向对象框架等。

这两个领域是相辅相成的。如果开发团队要创建医疗系统, 他们可能既要求助于与医疗系统设计相关的模式, 也要求助于与多层Web系统设计相关的模式。团队可能也会对软件开发流程这个领域感兴趣, 目前人们总结出了很多有关敏捷开发实践的模式[Cun96][BDS+99][CoHa04]。前面所说的这些情形中, 领域可以看做是对系统或者系统的开发过程中相关属性 (比如技术、流程、业务) 的反映。

回到我们正在讨论的模式中上下文所扮演的角色, 领域定义了模式所应用的部分或者全部上下文。所以根据领域来组织模式的概念和根据上下文的某些方面来组织模式如出一辙。通过领域或上下文组织模式还可以配合问题框架来进行, 这在本章后面还会详细探讨。

从任何一本手册的角度看, 按领域组织模式可以让开发人员、架构师、项目经理具有相同的理解。另一方面, 这种组织架构可能会边缘化或者模糊化跨领域的模式。很多编程、架构和管理的观点是从不同的领域抽象出来的共同经验。按照领域组织的方式的一个潜在缺点就是视角过窄或者约束过严。

8.6 根据分区组织

比按照领域划分更窄的一种划分方式是按照模式归属于架构的哪个部分进行划分。如果架构中引入了分区^① (partition) 的概念, 每个分区所使用的技术和考虑因素都可能有所不同, 因此有望在不同的分区中使用不同的模式来描述设计的细节。层 (layer)、阶层 (tier)、组件和包都是分区的例子。

在给定的架构中, 不同的分区扮演不同的角色, 因此会使用不同的模式来定义各自内部的微架构。比如一个组件可能代表框架, 而另一个代表应用逻辑。用来构造可扩展库的模式和使用扩展库来构建应用的模式通常会有很大的不同。

^① 注意这里与磁盘分区是完全不同的概念, 这里是指对架构的划分方式。——译者注

8.6.1 阶层架构

粗粒度的架构划分最常见的就是Layers。在使用Layers模式的时候，我们给不同的层赋予不同的职责，所以每一层都为本层所使用的模式定义了不同的上下文。在*Core J2EE Patterns*的编目中就有这样一个例子[ACM03]。我们将其称为分层的方式（tiered approach），是因为它里面有5个独立的逻辑层——客户端层（client tier）、表现层（presentation tier）、业务层（business tier）、集成层（integration tier）和资源层（resource tier）——它们是对模式进行分组的依据。该书记录了中间3个层中的模式。

表现层。表现层封装了服务客户端所请求的表现逻辑。它负责生成UI元素——UI元素实际上在客户端层执行。这一层通常使用JSP和Servlet，常见的模式有Front Controller和Context Object。

业务层。业务层提供了应用客户端所要求的底层业务服务。业务对象位于业务层。业务对象可以由EJB实现，并且依赖很多模式，比如Business Delegate和Transfer Object。

集成层。集成层处理与非本地的资源和外部系统的通信，比如遗留系统和数据仓库。这一层经常用到JDBC、JMS和各种中间件技术。这层的模式包括Database Access Object和Domain Store。

以这样的方式组织集合中的模式所带来的好处是，它清晰阐述了如何将模式放到多层企业级Java系统架构中。

8.6.2 分区

有时候，架构分区与领域的概念存在交叠。比如，对于用户界面设计领域来说，从架构分区的角度看也是可以明确界定的：用户界面是表现层的一部分。在数据库设计中也存在类似的情况。在这两种情况中，关注的领域和对架构的分区都是为提供解决方案的组织模式构建了合适的上下文。然而，虽然二者存在较多的交叠，但却很少出现二者完全重合的情况，因此，这两种划分方式并不等价。

与按照领域组织类似，按照分区的方式组织，读者可以根据自己所使用的架构很容易找到合适的模式，这类似于一个工程手册的作用。尽管如此，这种做法可能会漏掉一些适用的情况。比如，有些模式所记录的设计和编码风格对所有的分区都是适用的。这样的模式应该放到哪个分区里面呢？所以说，将其作为唯一的方式来组织所有的模式是不恰当的。类似地，根据罗素悖论^①，描述分区的模式放到哪个分区里面都不合适。

8.7 根据意图组织

模式集合可以根据一组模式共同的意图来组织，这个意图可以确定架构特征、共同的目标或

^① 集合理论中的罗素悖论是这样描述的：“城里面有一个理发师，他给所有不给自己刮脸的男人刮脸。那么他给自己刮脸吗？”

者职责等。

8.7.1 根据意图划分 POSA 的模式

我们在《面向模式的软件架构 (卷2)》[POSA2]、《面向模式的软件架构 (卷3)》[POSA3]和《面向模式的软件架构 (卷4)》[POSA4]中大量使用了根据意图组织的方式。在POSA4中所使用的大多数基于意图的编目都是从前几卷里面抽取出来的，部分做了改名、重组或者进一步划分。

在POSA2中，我们将模式分为4组：

- 服务访问与配置
- 事件处理
- 同步
- 并发

在POSA3中，模式被分为3组：

- 资源获取
- 资源生命周期
- 资源释放

在POSA4中，模式被分为13组：

- 从混沌到结构
- 分布式基础设施
- 事件分离与分发
- 接口划分
- 组件划分
- 应用控制
- 并发
- 同步
- 对象交互
- 适配与扩展
- 对象行为
- 资源管理
- 数据库访问

我们在《面向模式的软件架构 (卷1)》POSA1里面也用到了根据意图划分的方式，比如Layers、Pipes and Filters和Blackboard被划到从混沌到结构 (From Mud to Structure) 里面。与之类似，交互式系统包括 Model-View-Controller 和 Presentation-Abstraction-Control，通信包含 Forwarder-Receiver和Observer，管理包括Command Process和View Handler。然而，POSA1中所使用的意图架构可以说是根据抽象层次划分的子方法，而且其中的命名在一致性上也有待改进。比如，交互式系统更像是与领域相关的命名，而不是基于意图的命名，对于后者来说，交互或许是个更好的名字。这一点或许无关紧要，但是当我们详细讨论和反思归类问题的时候（如在本章所做的），

它就变得很显眼了。

8.7.2 根据意图划分 GoF 的模式

《设计模式》[GoF95]中使用的基于意图的分类方式大概是最为不朽的例子了。里面的模式根据其意图分为创建型、结构型和行为型3种。

尽管这个分类方式非常流行，但是这种分类方式到底意味着什么或者说它为什么有用都不甚明朗——实际上，我们都不确定这种方式是不是有用。当然，创建型这个类别是很清楚的，有用也有效。但是行为型和结构型到底是怎么一回事？所有的面向对象设计都是通过结构来实现行为的，所以在这个层面上二者看不出本质的区别。Composite（结构型）和Interpreter（行为型）就是很好的例子。与之类似，Chain of Responsibility这个名字明显带着结构的意味，却归为行为型，Decorator明显带着行为的意味，却归为结构型。所有这些例子都给出了支持某些特定行为的结构，反之亦然。

在结构型模式里面，一个反复出现的主题就是通过某种形式的直接转发来支持特定的行为，比如Proxy是转发给一个对象，Composite是转发给多个对象。但是这只能说是一类常见的解决方案结构，而不是意图，并且从全部模式来看，这一点也不是一致的。行为型就更不靠谱了，好像是算不上创建型和结构型的就归为行为型了。

8.7.3 根据意图划分 DDD 模式

当然，根据意图组织模式并不是说就不能涉猎直接关注设计产物（比如对象和组件）的模式。这种按照意图组织模式的方式同样适用于跟过程相关的模式。*Domain-Driven Design* [Evans03]为系统提供了一种围绕着领域模型来完成概念建立和开发的方式。*Domain-Driven Design*所基于的模式也是根据意图分组的。比如，相关的编目包括：*Binding Model and Implementation*、*Isolating the Domain*、*Supple Design*和*Maintaining Model Integrity*。这些分组的命名很明显带有意图的色彩，读者可以根据他们要解决的问题很容易判断应该看哪一部分。

8.7.4 反思模式意图

看上去，意图提供的组织模式集合的方式是很稳定的，而且也确实部分反映了模式结构。通过将重点放在要解决的问题身上，基于意图的分类方式为其包含的模式提供了很好的提纲。它所遵循的面向目标的原则很合大部分模式读者的口味。然而，作为模式集合的编纂者，应该明白这里面有一项主观的元素：模式的意图并不是唯一的，而且也不是完全客观的。

8.8 组织模式集合（重奏）

为了回答在何处和为何使用某些模式的问题，我们需要对模式进行编目。模式编纂者在表现形式上有很多选择，根据共同的上下文——比如领域和分区或者共同的目的（即意图）进行分组的方式所提供的框架看上去最为一致。模式的适用性来自某种特定情况下的意图：希望在某个特定上下文中达成某个目标。这种情况和意图的结合也构成了模式集合的组织特征。

如果模式集合基于通用的目的，其情形的广泛和意图的模糊会使得它意义降低。然而，如果适用情形和意图非常清晰，就可以帮助编纂者深入了解如何组合模式，读者在浏览模式的过程中也可以获得更清晰的认识。比如Java中线程安全的设计，其目的是线程安全、性能良好、对代码侵入小，这在清晰定义的范围内提供了一个具体的目标。

适用情形既可以用问题领域来划分，也可以用解决方案领域来划分。这种根据问题领域——尤其是软件系统所在的问题领域——组织模式的思路构成了与分析和设计相关的不同观点之间的纽带。它进一步阐明了模式上下文的角色和意义以及驱动力是如何产生的。由于它关心的是待解决问题的性质，并以此来组织模式，这也就意味着另一种相关的方法（比如模式框架）可能跟如何提炼与展现模式是相关的。

8.9 问题框架

经常有人批评软件开发不论是个体还是整体文化上都过于以解决方案为中心[BTN06]。

软件开发者通过代码解决问题。我们天生就是要分解问题、解决问题或者快速有效地找到解决方案。我们自然地倾向于“解决方案空间”，其中架构、设计、模式和惯用法组合在一起解决各种难题——这正是客户仰仗我们的地方。

软件开发人员为解决方案的世界所吸引，乃至偏离了要解决的问题，更有甚者，解决方案反客为主，比其要解决的问题提前来到这个世界，以至于我们经常看到拿着“解决方案”寻找问题的情形。

以设计为中心的模式在问题领域和解决方案领域架起了一道桥梁，凸显了解决方案结构、角色、问题上下文和驱动力之间的设计关系。虽然我们在第1章中所给出的模式的定义不够精确，但是它至少指出了模式所涵盖的两个方面：“模式是针对某个上下文中反复出现的问题之解决方案。”然而尽管如此，还是有很多开发人员和评论家只看到了模式面向解决方案的一面。他们或许太健忘，或许没有意识到模式的问题驱动的本质，错误地将模式所建议的解决方案结构认为是模式本身。所以，当他们看到具有相同解决方案结构的模式时，便会因为无法分辨其不同而无所适从。

所以，尽管模式世界已经有意识地拉近问题和解决方案的关系了，人们对解决方案的偏爱仍然屡见不鲜，就像人们对问题领域已经理解得差不多了一样[BTN06]。

模式就像“蛇和梯子”（Snakes and Ladders）桌面游戏中的梯子——对于给定的上下文和问题（桌面上的方块），它们给我们提供帮助以便达到更高的层次。设计模式恰好落在解决方案空间的正中央，提供了应对解决方案空间驱动力[Jac95]的设计结构的面向对象片段。但是他们确实假定我们已经充分理解问题和上下文了，所以可以选择正确的模式。那么，如果我们还不具有这个能力，会怎么样？如果发现我们自己只是在一个虚无缥缈的问题空间瞎逛，而无法为模式找到任何立足之地或者在哪儿使用模式所提供的架构片段，又该怎么办？

针对这些问题，常见的答案总是采纳某种指定的方法，这种方法可能会在其生命周期内规定广泛的分析活动，并符合某种特定的思想流派。比如，“世界是由对象组成的”与“世界是由过程组成的”，这是两种截然相反的分析角度。然而，很多类似的方法最终都是追求合成（组成问题之解决方案）而不是分析（理解问题本身），这多少都有对问题削足适履之嫌。

8.9.1 问题框架

Michael Jackson的问题框架的概念[Jac95][Jac01]绕过“分析与设计合一”的方法进行更深入的分析：理解问题域本身，而无需接受编程世界的隐喻和结构。虽然我们可以说真实的世界里面既有对象也有过程，然而与软件开发中的对象和过程相比，前者不过是肤浅的类比。实际上，将两种事物等同起来往往都是分类错误，并且会带来可笑的、严重的后果。地图不是领土。

问题框架为一类反复出现的问题命名，界定了其边界，并提供相应的描述[Jac95]。

问题框架是对一类问题的普通化 (generalization)。如果对于你正在处理的问题，你发现又有别的问题适合相同的框架，那么那些问题的解决方案很有可能对于你目前要解决的问题也是适用的。

5个基本的问题框架（绝非完整或权威的架构）列举如下[Jac01]。

Required Behaviour (请求行为)：在物理世界中的某些部分，其行为被控制以满足某些条件。这样的问题要求创建一个机器来实时此控制。

Commanded Behaviour (命令行为)：在物理世界中的某些部分，其行为根据操作者发出的命令而被控制。这样的问题要求创建一个机器接收操作者的命令，并实施相应的控制。

Information Display (信息展示)：在物理世界中的某些部分，有必要持续获得有关其状态和行为的某些信息。这样的问题要求创建一个机器以便从物理世界获得此种信息并在必要的地方以必要的形式表现出来。

Simple Workpieces (简单工件)：用户使用某种工具创建和编辑某类计算机可处理的文本、图形对象或其他相似的结构，以便用于后续的复制、打印、分析等。这样的问题要求创建一个机器来充当这样的工具。

Transformation (转换)：对于某些输入文件，它对于计算机是可读的，其数据必须转换为某种要求的输出文件。该输出数据必须是某种特定格式的，而且必须从输入数据中按照某些规则得到。这样的问题要求创建一个机器以便从输入文件中产生要求的输出。

每个框架给出的是问题的清晰描述，而不是关于解决方案的建议。不同框架之间的区别在于它描述的问题、方式以及可能采取的解决办法。不同的问题框架适应不同的描述方式。比如，Transformation框架下的问题最适合采用数据流技术或者是Jackson Structured Programming (JSP) 进行描述，而Simple Workpieces则比较适合用类、实体模型，再加上一些用例和动态模型。现实的问题并非完全契合某个单一的框架，于是就出现了多框架 (multi-frame) 或者组合框架 (composite frame) 的概念[Jac95]。

问题框架总是简单的。这是它们存在的价值。它们略去现实问题中各种纠缠在一起的复杂性，留给我们一个简单易懂的问题和一个简单高效的解决办法。

如果你所面对的问题具有现实的复杂性，那么可以尝试将其拆分为几个简单的问题。当然，这几个简单的问题应该是你可以解决的问题。如果把一个问题拆分为几个你仍然无法解决的问题，那是没有什么意义的。对于这几个简单的问题，你应该充分理解相关的问题框架和方法。

正如我们可以识别并命名模式复合一样，我们也可以识别出常见的组合框架，比如 Workpieces 和 Command Display[Jac01]。

问题框架和模式具有很好的兼容性——它们都力求具体，反对常见于其他方法的过于泛化的倾向。设计方式不止一种，看待问题的方式也是多种多样。

8.9.2 问题框架和模式的对比

在成功的软件开发中，确定问题的边界是重要的组成部分。基于模式的方法试图通过理解模式要解决的问题所在的上下文中的驱动力来达到这一点。但是对于如何确定问题的上下文、驱动力和动机，并没有给出正式的指导。

相反，问题框架提出了界定问题世界的课题，而不涉及解决方案方面的内容。在给定的问题框架中，某些模式在策略层次上是适合该框架的，有些则没有。比如，在由 Simple Workpieces 和 Information Display 组成的组合框架中，Model-View-Controller 可能就是一个合适的战略性模式，它决定了该架构的核心风格。

Michael Jackson 说“问题框架就是一种模式”，那么问题就来了：问题框架到底算哪种模式呢？即使根据我们在第1章开头所给出的模式的定义，问题框架也算不上本书所关注的模式——它缺少解决方案。问题框架的目的乃是基于问题类别的重现更精确、更清晰地理解所出现的问题。正是这种重现性，让我们觉得它从某种程度上像是一种模式，但是我们也应当注意虽然二者有相近之处，但问题框架既不是一种特殊的模式，也不是我们所理解的模式概念的子集。

有人可能要把问题框架称为问题模式。既然它关注的是问题，而且是反复出现的问题，打上问题模式的标签是自然而然的。然而，认为问题模式给问题领域建立框架这样的想法多少有些误导性。人们就该把模式改名为解决方案模式。但是考虑到我们花了那么多力气去说明模式不是一个严格以解决方案为中心的概念，这个改名似乎有些不恰当了。换句话说，解决方案模式恰好能够表述某些开发人员对模式的误解，他们认为模式是代码或者类图中通用的结构，完全不考虑其意图和问题动机。虽然问题-解决方案模式似乎更确切一些，但是却显得冗长，不过是对我们刚才对模式所做解释的重述。

问题框架和模式的共同点是它们都为我们建立一套词汇，鼓励采用一致的表现形式帮助理解。它们最恰当的结合是，问题框架可以为某些关键的模式提供适用的上下文。如果从分析模式的角度看待问题框架[Fow97]，只不过在它的上下文中对于解决方案（即对问题所处情形的理解）涉及较少，我们也可以以模式的形式来理解问题框架——甚至有人已经将其转换成了模式的形式

[BTN06]。

8.9.3 问题框架与模式的组合

过去，很多方法学追求通过一个宏大的、排他的架构来解决或应对开发中的问题，问题框架则试图为不同的架构方式定义其适用性和边界。

比如，在有关转换的问题上，Pipes and Filters是一种天然的架构，支持从输入流到输出流的连续的、直接的转换。一种更本地化的战术性的看法是使用Iterator来表现输入和输出流，并通过转换代码完成从一个到另一个的驱动。另一个基于控制反转的方法是使用Enumeration Method（有时结合Visitor使用）来完成从输入到输出的遍历和转换。这种区别可以从XML解析器中窥得一斑，有些API采用拉模型，有些则采用推模型。当然，在模式词汇和模式序列中还有其他的变化和考虑可以选择，但是很明显框架名副其实：它定义了一个合适的边界，我们可以以此为界专注于设计。

就算不提Jackson的问题框架，至少问题框架的这种方式为我们提供了一个组织模式集合的好方法。我们可以认为问题框架描述了增强的、详细的意图，抓住了问题领域及其对解决方案的约束的本质。

将这个概念应用到架构手册上，问题框架方法激发了一个非常关注于问题的方法，进一步支持了这个基本的理念——通过情形和意图组织编目比前面讨论到的其他架构更为内聚，也更为有用。

8.10 模式符号学

如果说模式定义了个词汇表，模式描述采用某种文学作品的形式，那么是不是某些语言学或者文学分析工具可以帮助我们理解模式之间的关系，如何使用模式，以及如何有效地组织模式呢？符号学是研究符号和标记的学问，其研究对象主要是语言中的符号，特别是社会上使用的符号所代表的角色和含义。

最早的时候，符号学将符号定义为两部分：Signifier（能指）和Signified（所指）。能指是符号的表达方式，是物的方面。所指是由“能指”所引起的相应的“意”。比如，在英语中能指“red”与它所指的红色的概念，共同构成一个符号。能指和所指之间的关系被认为是任意的，而非给定的。在德语中，能指“rot”代表的是同样的能指角色，虽然从表面上有很明显的不同，但是其所指的概念确实是一致的。“rot”在英语中是一个完全不同的符号，其所指是“腐烂”。

模式与模式描述构成符号

根据符号学的观点，我们可以说符号=能指+所指（sign = signifier + signified）。这种划分方法支撑了James Noble和Robert Biddle对模式的解构（deconstruction）[NoBr02]。

将模式文档化最大的挑战就是避免误解：因为有些人看到模式描述的时候可能无法正确（或者完全正确）地理解其解决方案以及模式的意图。在读程序的时候（或者观察一栋建筑

的时候)，我们也一样可能会误解所看到的模式——楼梯下面的碗柜实际上是通往地下室的楼梯，我们认为应该推开的门可能是拉的，我们认为应该使用Observer模式的可能实际上使用的是Mediator模式，甚至只是一个很烂的设计，等等。

符号学方法通过将所有可能性明示出来应对这类误解。能指，就其本质而言就是具体的、有形的、公开的，而所指是抽象的、无形的、属意的概念——在看到程序或者观察建筑的时候，每个人对所有的符号都会形成自己的所指。因此，对于任何一个能指来说，都极有可能会产生“错误的”所指形象。

[……]

符号学明确地告诉我们，只要是用到符号的地方，就必然会出现这种误解，所以对于模式来说，出现这种误解也没有什么可大惊小怪的。

使用这种方法，模式被认为是由名字（能指）和模式描述（所指）所构成的符号。而模式描述也可以认为是一个符号，它是由可见的设计（能指）和意图（所指）——包括模式的上下文、问题、基本原理和已知应用（而不只是GoF的模式格式里面意图那一节）——共同构成的。这种观点意味着模式实际上是一个二阶符号系统，其中包括模式和模式描述两个层次：

模式=模式名字+模式描述

模式描述=解决方案+意图

这种两级分解使得模式我们可以在一个概念框架内理解模式、模式的关系、模式的含义以及它们的误用，这与我们通常用来讨论和分析模式的方法有所不同。这种符号学方法看上去正是对我们在前面章节中所做的讨论的提升，在某些情况下使得我们讨论更为明晰。

例如，当我们从符号学的角度来看待模式时，更容易意识到剥离掉意图，将整个解决方案等同于整个模式是错误的。模式的读者可能并不了解以代码形式给出的具体表达和代码背后的概念——上下文、问题、基本原理以及范例——之间的联系。模式被看做一个简单的一层的符号，模式描述被解决方案所取代，而不是解决方案与意图的结合。

换句话说，读者看到的是：

模式=名字+解决方案

而不是：

模式=模式名字+模式描述 其中模式描述=解决方案+意图

这个问题最常见的例子是当解决方案只有一个方面被考虑的时候，比如用类图的方式表达。Strategy和Objects for States模式的结构化类图是完全一样的。这个类图跟Bridge模式也非常相似，如果在Handle一端没有继承结构，那么它们就是完全一样的。所以，这些模式经常被弄混。其类图如图8-1所示。

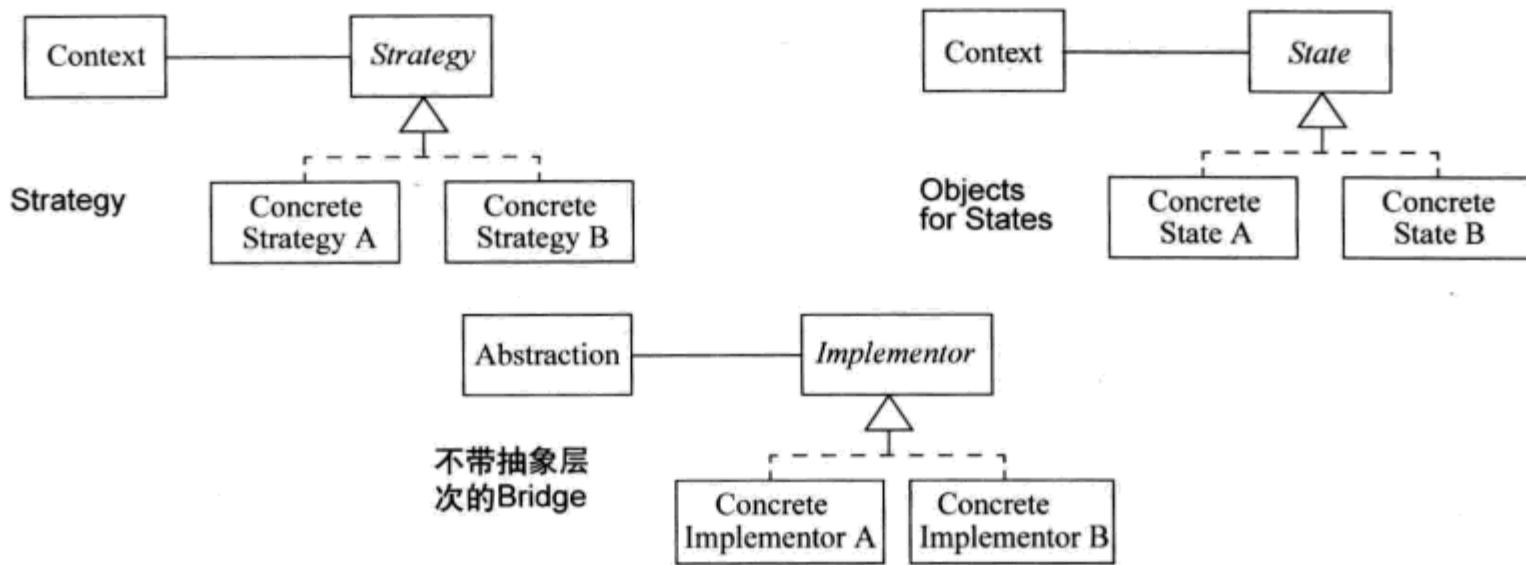


图 8-1

当然，有时候，模式的这种互补性体现为某个具体的设计确实实现了多个模式：每个设计元素扮演不同模式中的不同角色，如4.3节所示。然而，这不应当成为模式实现出现混淆时的通用借口。

另一个更加深入的例子处理GoF的编目中Adapter混乱的含义 (problematic meaning)。把Adapter作为一个单独的模式提出来除了GoF的书之外再无他人。它本质上就不是模式描述的一个属性，它至少包括了4个截然不同的模式：Object Adapter、Class Adapter、Two-Way Adapter和Pluggable Adapter，我们在第5章中只涉及了前两个模式。

尽管这4个模式都以适配作为共同的主题，但适配还是比较适合作为一组或者一族模式的集合而不是一个单独的模式。后面，在对Decorator进行符号学分析以说明它与Wrapper和Adapter都是同义词时，我们还会进一步讨论这个观点。再加上基于Wrapper层的Facade，我们可以说以特定的adaptation或者wrapping作为分类要有用而且清晰得多，而《设计模式》[GoF95]中使用的Structural和Behavioral——正如我们在本章前面所说——对所有模式都适用，既包罗万象，又模糊不清。这种分类方式既简洁又通用，即使在GoF的《设计模式》之外也可以应用。比如，Wrapper Facade也属于这个类型。

这种符号学方法使得缺失的分类和联系更为清晰。就本例而言，符号学视角为我们揭示了一个清晰的意图分类 (适配)，这样对于开发人员来说，要比将各种不同的模式和变化统统归为结构型分类的子集要有意义得多。由于任何分类都难以避免主观性和多变性，而清晰的组织结构对于模式集合非常重要，所以对于模式集合的编纂者来说，理解可能的解释和误解是非常必要的。这种反馈和对细节的关注也是符号学方法所带来的好处。

8.11 模式集合与风格

我们在5.2节中指出，如果要在两个或者几个具有相似意图的模式中进行选择，其他的方面如果没有差别的话，剩下的就是风格的问题了。在C++、C#和Java编程中，倾向于使用简单的Iterator来完成迭代，而在Ruby和Smalltalk编程中，Enumeration Method更为自然，Batch Method

和Batch Iterator在分布式对象计算系统中是一种约定俗成的做法。

作为模式的特征，风格并非局限于编程惯用法或者多个模式之间的细微差别。风格也同样适用于模式集合和相应的架构[Booch06]。

在任何给定类型的系统中，我们都会发现一些反复出现的设计。如果我们能命名这些设计，并以相对通用的风格将它们编纂在一起，就可以将其称为该领域的模式。

有时候，我们会提到某种风格（或者流派）的设计。一种风格是一套可以区别于其他风格的设计的集合。

风格可以看做是规则，因为它规定了在某个架构或者组织结构内需要遵循的游戏规则，该游戏规则决定了这个架构或者组织结构的内在，以及与其他架构或者组织结构的区别。

很多模式集合都可以看做是一种风格。比如，Erich Gamma的早期论文[Gam92]以模式的形式记录了ET++框架的设计风格。《面向模式的软件架构（卷2）》[POSA2]中的模式记录了常见的面向对象和基于Broker的中间件的设计风格。很多相关（或者不太相关）的系统展现了共同的架构特征和开发流程特点，其风格可以从Big Ball of Mud模式[FoYo97]中找到。

8.11.1 Unix 接口设计模式

The Art of UNIX Programming [Ray04]记录了Unix操作系统的文化、工具和技术。在这本书中还列出了使用命令行shell的接口设计模式：Filter、Cantrip、Source、Sink、Compiler、ed、Roguelike、Separated Engine and Interface、CLI Server和Polyvalent-Program。

8

把这些模式的描述都放在这里，内容有点多，但是我们可以总结如下。

Filter模式定义了一个非交互式程序或者程序的执行（execution of a program），它可以在标准输入接受数据，将其转换，并将结果发送到标准输出上。比如，`sort`和`grep`程序都可以作为Filter执行。

Cantrip模式定义了一个非交互式程序，既不接受输入，也不产生输出，只有一个状态结果。Cantrip作为一个简单的独立命令被调用。比如，用来删除文件的`rm`程序可以作为Cantrip。

Source模式定义了不接受输入、只向标准输出流输出数据的非交互式程序。比如，`ls`程序用于列出目录，它属于Source模式。

Sink模式定义了一个只从标准输入接受数据，而不产生输出的非交互式程序。比如，`lpr`程序将输入Spool到打印队列，它属于Sink模式。

Compiler模式代表将文件从一种格式转换为另一种格式的非交互式程序。它们除了向标准错误流输出消息之外，既不接受输入数据，也不产生输出。比如，C编译器`cc`和`gzip`、`gunzip`压缩工具都属于编译器。

ed模式定义了具有简单交互模型的程序。它们可以解释简单的标准输入命令行语言，因此是可以脚本化的。不可视的`ed`行编辑器是最经典的例子，GNU调试器`gdb`也属于ed模式的例子。

Roguelike模式所定义的程序具有更加完整的图形化界面，并通过按键驱动。它们是交互式的，但不容易脚本化。*rogue* Dungeon Crawling游戏是一个最早的例子，更为人们熟知的例子包括*vi*和*emacs*编辑器。

Separated Engine and Interface模式定义了一个程序架构，将程序的engine部分（持有应用领域的核心逻辑）同interface部分（负责表现逻辑和用户交互）区分开。engine和interface角色通常在独立的进程里实现。

CLI Server模式定义的程序在前台运行时，在标准输入输出流上提供一个简单的命令行接口，在后台运行时，其输入输出则发生在TCP/IP端口。很多基于协议的服务器遵循这种模式，比如POP、IMAP、SMTP和HTTP。

符合Polyvalent-Program模式的程序，其架构使得它们可以具有许多接口，包括从可编程的API到命令行和GUI，以及Cantrip到Roguelike。

在所有上述情况中，对于遵循这些模式的程序来说，它们的参数既包括启动环境，也包括命令行参数。

这些模式清晰地刻画了一种独特的设计和工作方式，它跟图形用户界面或者其他命令行shell环境（比如DOS或者VMS）的风格都截然不同。这方面最具代表性的具有“支柱”（backbone）[Vlis98b]地位的模式是Pipes and Filters。实际上很多模式是Pipes and Filters在某个上下文中的具体细节使用，当然不失一般性和有效性。这些模式所提供的解决方案主要是应对Transformation问题框架的。其他的模式有的是针对Simple Workpieces框架的（比如ed模式），有的是针对Information Display框架的（比如Roguelike）。

8.11.2 Web 2.0 设计模式

Web 2.0跟技术的关系不是太大，它主要描述如何针对网络进行设计和业务活动。人们已经将这个方法以模式描述的形式总结出来了[ORei05] [MuOr+07]。

下面的模式和总结来源于原始的模式大纲[ORei05]。

The Long Tail（长尾）。小站点组成了互联网大部分的内容，小众市场构成了互联网上可能存在的应用的主体。因此：利用客户自助服务和基于算法的数据管理，将触角深入到整个互联网，延伸至各个边缘而不是只关注中心，延伸至长尾而不仅仅是头部。

Data is the Next Intel Inside。越来越多的应用是数据驱动的。因此为了取得竞争优势，需要寻找独立的、难以再造的数据源。

Users Add Value。在互联网应用中要取得竞争优势，能从多大程度上由用户添加自己的数据是非常关键的。因此不要把“参与架构”（architecture of participation）限制在软件开发上。将用户直接或间接地包含进来，让它们为你的应用增值。

Network Effects by Default。只有很少的用户会主动为你的应用增值。因此在应用中设置默认值，用户数据的聚合只是其访问行为的副产物。

Some Rights Reserved。知识产权保护会对重用和尝试起到限制作用。因此当我们从互相

兼容（而不是私有约束）中获益时，就应该降低兼容的门槛。遵循现有规范，在许可中尽量少做限制。应用的设计要易于接入和易于与其他应用混搭。

The Perpetual Beta。 当设备和程序连上了互联网，应用便不再是软件产品，而是一种正在进行的服务。因此不要把新特性打包成一个完整的发布，而是定期地添加到系统中，成为正常的用户体验的一部分。吸引用户对系统进行实时测试，并对服务进行检测，以便知道怎么使用这些新特性。

Cooperate, Don't Control。 Web 2.0应用建立在协作数据服务的网络之上。因此对外提供网络服务接口和数据同步，重用别人的数据服务。为构建松耦合的系统支持轻量级编程模型。

Software Above the Level of a Single Device。 PC不再是访问互联网应用的唯一设备，如果应用被限制到单一的设备上，其价值就会比那些互联的应用低。因此从设计的开始阶段就考虑让服务支持手持设备、PC和互联网服务器。

Web 2.0以及上述模式越来越流行，其风格与Web 1.0时代的设计和业务模型具有鲜明的区别。

8.11.3 风格与概念一致性

风格不仅有助于奠定一种文化，同时也赋予系统完整性和自己的特色。概念完整性比独立的设计决策持续的时间要久，并且采用公共的设计理念[Bro95]。

我认为概念完整性是系统设计中最重要的因素。系统为了反映一套设计理念而舍弃某些反常的特性和改进，要比包含众多好的独立而互不协调的理念好得多。

我们可以将这套设计理念与一组模式集合关联起来。当然，正如我们看到的，任意的模式集合不能提供概念完整性：它们的角色和结果必须协调，必须能够满足最终的目的[PP03]。

概念完整性意味着系统的核心概念必须能够构成一个平滑而内聚的整体。组件之间能够相互配合工作，架构则在灵活性、可维护性、高效性和响应能力之间达到平衡。

品质和一致性对于将不同风格的作品和架构区分开非常重要，所以风格对于引导开发人员如何基于给定系统工作或者在给定系统内工作也非常重要。

例如，如果要以某种特定的方式遍历对象的集合，可以选择经典的面向对象编程方式，也可以选择泛型编程。在经典的面向对象方式中，共性的部分将被抽取到类层次结构里面。因此，我们会抽取一个超类来实现算法的共性部分，比如通过Template Method的方式，访问细节被推迟到子类中实现。需要支持遍历操作的集合类型必须从这个超类继承而来。而在泛型编程方式中，我们将集合与算法解耦合。该操作通过一个单独的函数表现出来，它可以对一个元素序列进行操作，该元素序列可以通过 Iterator表现。集合类型与其上的操作是正交的，但是它必须支持提供必要迭代器的接口。这两种方式在不同的方面做了妥协，在指导开发上具有不同的风格[Hen05a]。

8.12 走向模式语言

经过组织的模式集合提供了有用的知识信息库。从设计的角度来说，该信息库超出了任何单个模式或者前几章所讨论的模式组合所能触及的深度和范围。在考虑如何为这些集合和模式关系设计结构的时候，我们在本书第二部分中展示的根据上下文和目的（purpose）组织模式和模式结构观点可以作为一个起点。

诸如模式故事和模式序列之类的概念赋予模式集合以生命。模式可以根据共同的上下文或者共同的目的进行划分。对于特定的系统、架构或者框架，可以用模式集合的方式记录它们的风格、架构或框架。有了模式故事和常用的模式序列（如POSA4中介绍仓库管理流程控制的那些章节），我们可以对风格形成一个生动而清晰的认识。从本质上讲，通过将风格记录下来，模式代表特定系统中地道的工作方式，给未来的扩展和其他工作指明方向。

在前面几章和本章讨论模式概念时，有关语言的隐喻占了很大部分，包括故事结构以及符号学等。我们反复强调在设计中不能只盯着一个模式，要将更多的模式综合起来考虑，本章中我们专门讨论了模式集合的问题，现在应该可以讨论模式语言了。

第三部分

模式语言



奥尔胡斯莫斯格史前博物馆的石碑
版权归 Mai Skou Nielssen 所有

在本书第三部分，我们基于前两部分提出的概念和结论引入了模式语言的概念。模式语言的目标是，为使用模式开发特定技术或者应用领域的软件提供全方位的支持。为了达到这一目标，它们为各个领域内可能出现的问题列出多个相关的模式，并将其组织在一起，构成一个高效的、领域相关的软件开发流程。

采用模式成功地设计软件仍然是《面向模式的软件架构》系列的出发点。我们在第一部分和第二部分中所讨论的那些概念——单个模式、模式互补、模式复合、模式故事和模式序列——无不是为了这个出发点服务的，但是还不能说是已经实现了。在第三部分中，我们将详细阐述模式语言的概念，以期为采用模式开发软件提供一个完整的、系统性的、建设性的支持，既

包括具体要创建的设计，也包括如何作出设计的流程。为此，模式语言集成了前面所列的各种概念，并且对其进行扩展，这些扩展主要集中在流程方面。

本书第三部分包括以下章节。

- 第9章详细介绍了模式语言。我们试图从建立一个设计语言的角度回顾了目前为止涉及的模式的概念。
- 第10章更加深入地介绍了模式语言。首先解释和讨论了模式语言的基本概念和结构，然后介绍了模式语言的各个质量方面、驱动力所扮演的角色及其重要性和上下文，还介绍了语言的解决方案空间的通用性。接下来，讨论了图表的使用、对模式语言的命名，以及它们随时间流逝逐渐成熟。
- 第11章分析了模式语言流程方面的内容。首先展示并讨论了流程中的组成元素，展示了如何在实践中应用模式语言。然后探讨了如何对比模式语言，或者将其与其他方法（比如重构、产品线架构以及单个模式）集成。
- 第12章讨论了对模式语言进行文档化的问题。对模式语言进行文档化绝不是简单地描述其中的每个模式，然后把它们放到一起。我们也讨论了如何就这些模式语言的目标和细节进行沟通以提高读者的效率。
- 第13章讨论了单个模式和模式语言之间的共同点和不同点，得出二者相辅相成的结论。

在软件社区中，模式语言并不是什么新概念，在软件模式运动开始的时候，人们就在描绘和讨论它了[Cope96]。然而，由于缺乏更新，它们无法真正反映在撰写和使用模式语言方面的新概念和经验[Hen05b]。所以，这一部分所讨论的模式语言的各个方面在模式社区中并非都有成熟的认识或者并非都是完善的。例如，模式语言所引入的流程的基本要素和属性（如渐进式成长）是被广泛接受和应用的，而另一些要素和属性（如模式序列在定义模式语言的语法方面所扮演的角色）则被认为是新的概念，甚至是争议的。

另外，也不是模式社区中的所有人都认为模式语言是一个有用或现实的概念。一部分人认为语言是模式成就之顶峰。而另一部分人则认为模式语言严重约束了他们对模式的应用，只是让他们去模仿那些毫无意义或者陌生的架构风格。我们的观点是遵从实效原则：我们在很多产品软件系统中成功地使用了模式语言，所以认为它是一个有用的概念。当然，我们也不会盲目地使用，有些情况下它们会给我们带来无谓的限制，或者其他模式概念（比如模式序列，甚至单个模式）可能更有效。在实践中，我们也发现当模式语言严重约束他们对模式的应用时，就需要对所使用的语言做一个修订，因此这两种观点在实践中并不像理论上存在过多的冲突。

在这一部分中，我们的目标仍然是按照使用模式方面最新的知识和经验，尽量保持完整并反映最新的情况。因此，我们在编写过程中力求完整，而较少考虑其各方面的成熟程度和模式社区的接受程度。

由于我们擅长设计模式领域，所以这里展示模式语言概念的例子大多来自设计模式领域。然而，这里所探讨的模式语言的属性和要素对于软件（模式）中的其他领域，甚至对于笼统的模式语言都是有意义的。

我们的母语规定了如何去剖析自然……

语言不仅是经验的报告工具，而且也可以是它的定义框架。

—— Benjamin Whorf，语言学家和人类学家

本章开始介绍模式语言。从试图建立一套设计语言而不只是一些设计元素词汇的角度，我们重新探讨了本书一开始介绍和应用的模式的概念。模式集合、模式故事和模式序列都是其中的一部分，但不是全部。

9.1 使用模式进行设计

在第二部分中，我们浏览了设计模式空间，讨论了模式互补、模式复合、模式故事、模式序列以及模式集合。这些不同概念的一个共同主题就是它们对使用模式进行设计的支持。尤其是它们肯定了这样一点：软件系统产品必须解决很多不同的设计和实现问题。对于其中的每一个问题，都有必要仔细比较现有解决方案，了解选中解决方案可以影响的（或者被影响的）其他相关问题的潜在解决方案空间。

如我们在第一部分谈到的，单个模式并没有试图解决这样的问题。根据定义，它们仅关注特定驱动力下的单一问题，而不考虑（或只是简单考虑）替代的解决方案和它们对其他问题及其解决方案的影响。而软件开发则恰恰相反，它们必须同时对项目面临的主要问题和设计细节进行全面透彻的理解。如果没有这样一种洞察力，我们几乎不可能构建出一个满足和平衡所有需求的、稳定的软件架构。

仔细研究第二部分中介绍的概念，我们可以发现，它们仅努力解决使用模式进行设计的一些特定或局部的方面。每一个概念只是“蜻蜓点水”，基于模式的软件系统应该是什么样的呢？下面给出一点提示。

- 模式互补协助我们探索解决特定种类的问题的模式空间，但不是设计整套软件系统的模式空间。
- 模式复合通过使用其他“更小”的模式并以特定配置方式安排它们的角色来解决问题，从而引入单个模式，并为它们定义明确的上下文、目的、流程以及结构。然而模式复合

的概念仅仅比模式“原子”的概念宽泛一点点，因此它们对整个软件架构的影响也会受到相似的限制。模式复合仅仅描述其组成模式的一种特定配置，而没有跳出自身的范畴，探讨这些模式其他可能的有用组合。

□ 模式故事就像日记一样，讲述了模式怎样引导整个软件系统、子系统以及大型组件的开发过程，这在第7章中很好地进行了演示。模式故事还讨论了在构建软件系统的过程中需要以怎样的顺序处理哪些问题。另外，模式故事还就选择哪些模式来解决这些问题进行了探讨，以及这些被选择的模式怎样在目标软件系统架构中被实例化。

然而模式故事与上下文、需求以及特定系统的开发限制是紧耦合的。面向同一领域但针对不同的上下文、需求及开发限制的两套软件系统可能会遵从不同的模式故事，只是因为它们使用不同的模式或者以不同的顺序应用模式。因此，当其对应的上下文、需求及限制与需要进行开发或重构的系统的相关因素不吻合时，模式故事的实用性可能会有所降低。虽然开发者可以对详细情况进行总结以发掘共性，但他们可能无法由特定案例得到想要的结论。

□ 模式序列从根本上说，就是去掉了模式故事中的故事部分（特别是，模式故事中特定的上下文和案例所描述的系统的具体限制）以更好地强调模式应用的原始顺序。因此，我们可以说模式序列比模式故事更具普遍性。如果软件系统与特定软件序列所处理的问题具有一致的需求，那么我们就可以利用该序列进行设计，或者至少可以应用其中的某一部分。当软件系统的需求与给定模式序列不重合时，模式序列提供的设计建议再好其价值也是非常有限的。因为每个模式序列只能提供给定需求集合下一种可能的系统设计和实现，不同的设计和实现会需要不同的模式序列来应对。

□ 模式集合可以以不同的方式来组织，其中某一种特定的方法可能是最无关紧要的。在不同程度上，根据领域、分区、层次和意图组织的编目，提供了很多种方法来描述一组可能的设计，甚至设计风格。相较于模式序列而言，它们提供了更大的宽度，但它们的深度也会有所降低。传统编目方法欠缺的是关联性——这比仅仅知道集合中的其他模式是有某种关联的要重要得多。当集合以某种联合的方式展现时，比如说模式序列，我们会得到一种更强的关联。

从上面的简单分析，我们可以得出结论：当对模式概念的理解和应用超出了有用方法这一广义概念时，它们仍然不能满足面向模式的软件架构这一期望。虽然这些概念可以为第8章中描述的软件架构手册提供有用的内容，但对基于模式的软件开发而言，它们并不比一本教导正确语法和熟练对话的词典更具指导性。

9.2 从模式故事和模式序列到模式语言

我们寻找的这种“缺失关联”是一个概念，它支持将模式系统化地用于以下两个方面。

- 创建应用于特定领域的软件，例如仓库管理和医学图像处理。
- 处理软件开发技术方面的问题，例如分布式计算、容错或C++中的对象生命周期管理。

这个概念应该能够对特定领域可能会出现哪些问题，应该以怎样的次序处理这些问题，可以应用哪些模式来处理这些问题提供明确的指导。同时，这个概念还应该支持为解决给定问题，组合模式时彼此之间互补与冲突的权衡，以及将模式合成为具体的设计。而且该概念还应该能够在模式互补、模式复合、模式故事、模式序列以及模式集合的工作结果的基础上进行延伸和完善。此外，这个概念应该与单个模式建立在相同的属性上，并从整个系统、技术领域或开发角度的更广范围内对它们进行加强。

使用这一概念，我们可以在软件架构手册中为软件开发中的每个探讨主题，添加一个基于模式的开发或解决方案的条目。根据应用领域的差异（如企业系统、通用系统和嵌入式系统的种类不同）会有相应的条目，同样，根据技术方面的不同，如分布式、并行、安全、容错处理和使用语言的不同，都会有不同的对应条目。该手册中的任意根条目、“框架”或章节可以看做是特殊目的的模式语言，可以帮助开发人员构建特定类型的软件系统，利用模式帮助处理特定需求集合下技术方面的问题。

9.2.1 一个未完成的故事

如果是这样的话，模式故事和模式序列无疑是最为接近模式语言这一想法的概念，至少从直觉上可以这么判断。它们描述了特定类型的系统或技术问题是怎样的（或者说可以怎样）在模式的帮助下，被系统化地设计、实现或处理的。举例来说，第7章中的模式故事描述了怎样为灵活请求处理框架创建一个基于模式的设计方案：Command、Explicit Interface和Command Processor定义了该机制的基础结构，而Collections for States、Memento、Strategy、Null Object和Composite明确了其细节。

该模式故事的设计如类图9-1所示。

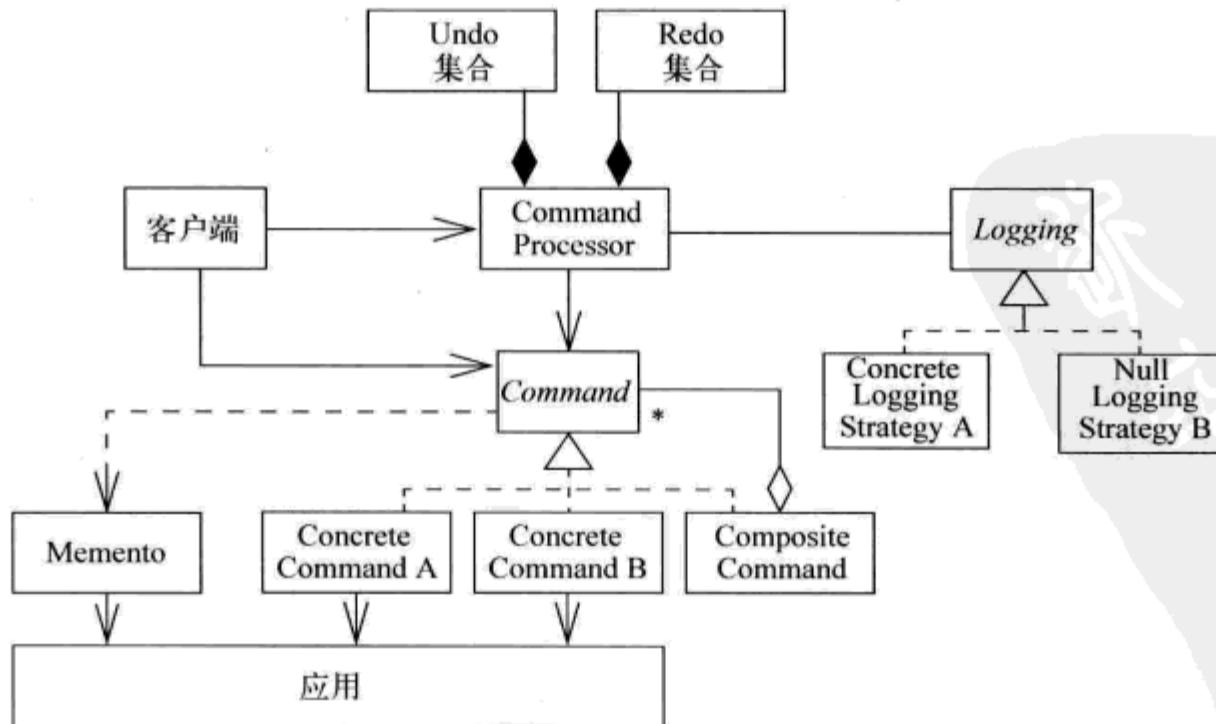


图 9-1

正如9.1节所展示的，模式故事的主要缺陷在于它们仅仅描述了特定类型的应用，或技术问题的一种可能的基于模式的解决方案。当请求处理框架的上下文条件、限制和需求与模式故事所描述的情况有所区别时，模式故事可能就会发生变化。

如果我们倾向于使用继承而不是委托来支持“辅助性”功能的不同变体，比如日志，则相应的模式故事应该采用Command、Explicit Interface、Command Processor、Collections for States、Memento、Template Method以及Composite模式，其中Template Method取代了Strategy和Null Object。

该替代方案的具体设计可用类图9-2表示。

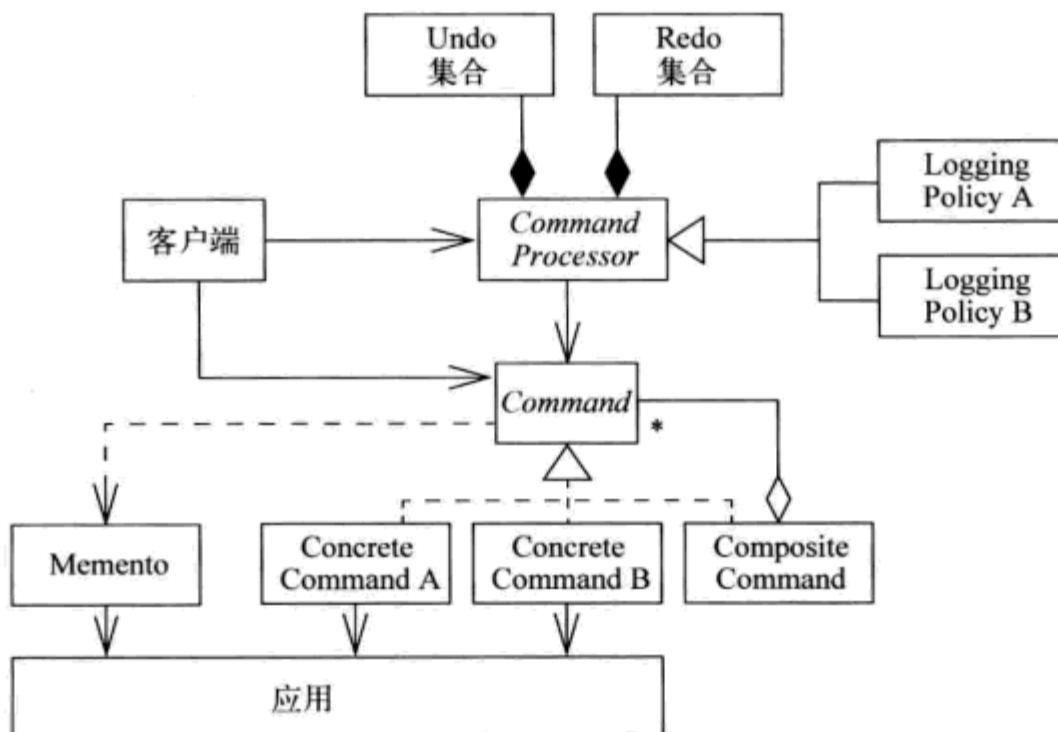


图 9-2

要描述请求处理框架怎样接收请求消息，而不是必须转换为可执行命令的过程式请求时，我们只能使用另一个模式故事。在该故事中，可以应用的模式是Message、Command Processor、Interpreter、Command、Explicit Interface、Collections for States、Memento、Template Method和Composite。

相应设计的类图如图9-3所示。

到目前为止，我们已经有3种请求处理框架的模式故事了，但还可能有更多的故事，例如一种线程安全的Command Processor。

而每一个模式故事说明了为什么选择特定的软件序列来实现相应的软件。对比不同的模式故事，可以帮助我们理解不同的上下文与需求，为什么和怎样产生不同的基于模式的软件解决方案。因此，模式故事对模式语言的概念提供辅助，但是它们的范围与目的太小，以至于无法构成模式语言的概念基础，以满足本节一开始列出的所有需求。

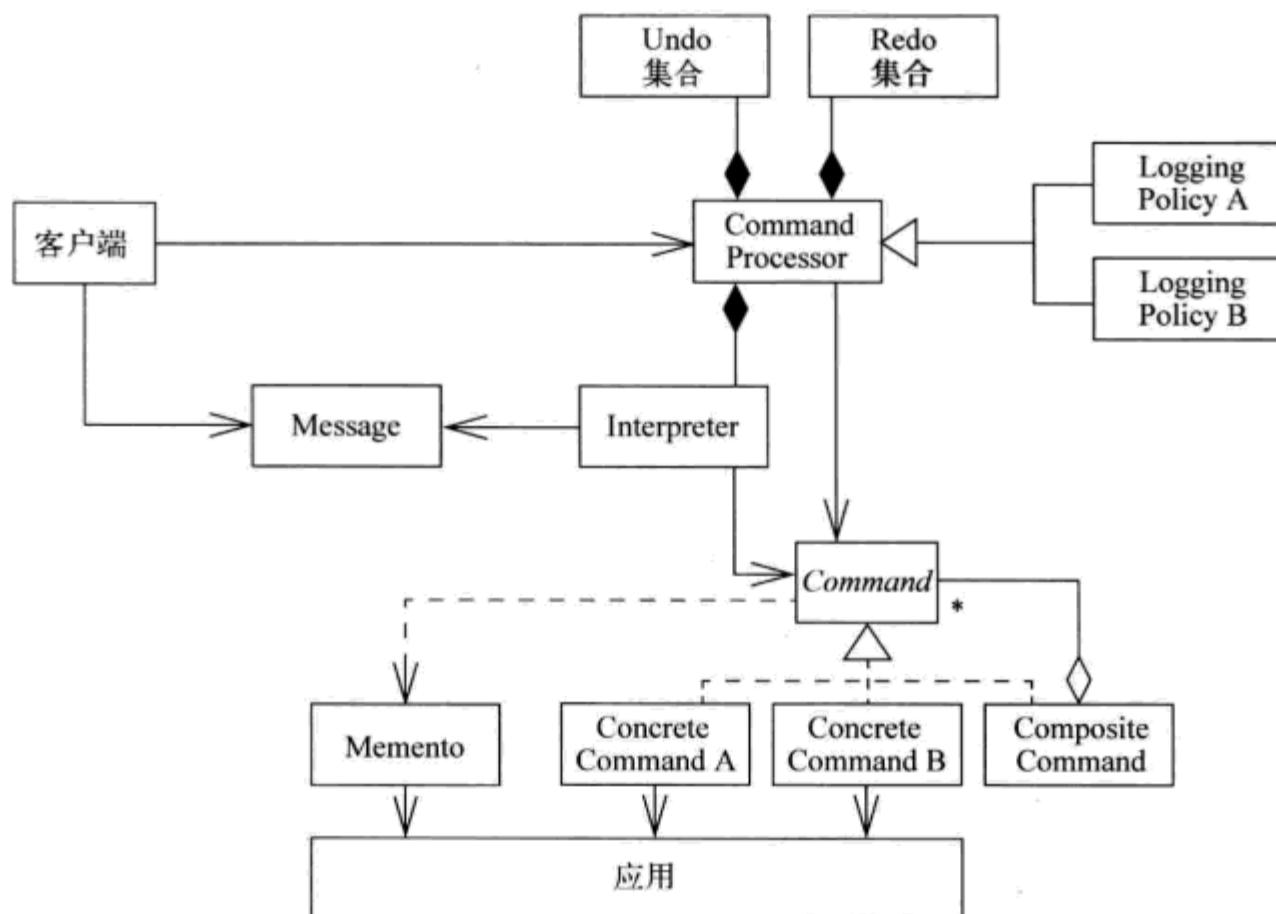


图 9-3

下面这段内容摘自一本最早使用“模式故事”的作品，这段内容支持我们的观点[Hen99b]。

当模式语言是叙述性的框架时，系统开发可以被看做一个叙述故事，在该故事中，设计问题被提出并被解答，结构为特定原因进行组合等。我们可以将一些设计过程看做是特定模式的渐进式应用。

9

“模式故事”的结构是叙述性的，它们在现实世界中是完整存在的。如同很多故事一样，它们抓住事件的精髓而无需抓住真实的细节。我们的设计思考很少能够一蹴而就，达到足以进行分类和时序安排的阶段，总是有一定的回溯和修正。作为教学工具，这种方法可以比传统的案例研究揭示更多的问题，因为每一个例子都讲述了一个设计故事。这样的说明能够在设计的学习过程中提供有价值的帮助。

9.2.2 序列的组合

模式序列对相关模式故事进行了压缩和抽象，只要与序列中模式处理的需求一致，开发人员就可以使用它在自己的系统中创建相应的解决方案。所以，模式序列可以说是将关注的重点从例子转移到了问题的本质。但是，从模式语言的预想属性的角度来看，它们与模式故事有同样的限制，正如我们在本章一开始所列出的那样。

但是，如果我们将描述同一问题领域的解决方案的多个模式序列合并起来的话，会有什么效果？这个集成的序列集合会不会形成我们找寻的模式语言的概念？这个假设至少听起来非常有前途。

□ 集成的模式序列很可能定义了构建特定领域软件的解决方案空间，以应对同样需求的不同设计以及不同需求的不同设计。如果仔细组合单个序列，这一解决方案空间也可能会比原来的模式序列例子覆盖更多的设计途径。这一整体会比其包含的各个模式序列的简单累加具有更大的价值，能够更好地支持同样设计空间中同样模式的其他应用。

例如，将上面模式故事中描述的模式序列集成起来，可以产生典型的、具有两种变体的迷你“模式语言”。首先，如果请求处理机制接收请求消息，我们可以从如下模式序列片段开始：Message、Command Processor、Interpreter、Command和Explicit Interface。其次，如果我们采用的机制接收过程式请求，可以从这样的模式序列片段开始：Command、Explicit Interface和Command Processor。两种模式序列片段中的Command模式可以通过Memento和Composite模式进行优化；Command Processor则可以通过Collections for States进行优化；如果我们希望应用继承来支持行为的变化，则使用Template Method模式；如果希望通过委托的方式来支持这种行为，可以使用Strategy和Null Object模式。注意这种迷你“模式语言”是怎样描述4种设计选项的，较组成它的模式序列还多了一个。

□ 如果使用集成的模式序列构建新的软件时能够产生新的模式序列，而这样的序列被添加到模式序列集合中，那么这个集合定义的解决方案空间就可以不断演化，总是能够代表特定应用领域中的软件构建最新水平。

作为这种持续集成的结果，我们可以产生一种模式语言，这种语言由定义了构建特定软件系统（或其某一部分）的全部——至少足够全面的——词汇和语法的模式与序列组成[Cool02]。该语言也可以发起软件架构师与开发者之间的对话，就什么样的模式序列最适合特定类型的系统，或给定上下文、需求集合以及限制的部分系统进行探讨。有了足够的经验，我们甚至可以更快地、更自信地估计哪一种模式序列可以产生完整而平衡的软件架构，哪一种会造成不完整的、不平衡的设计。而且，我们可能为判断替代软件设计方案的价值提供更好的基础[BS00] [SCJS99] [SGCH01]。

对我们的假设总结如下。

- 模式故事帮助我们从特定案例中进行理解和学习。
- 模式序列帮助我们总结从模式故事中学习到的经验，以及如何再次应用这些知识。
- 集成的模式序列可以形成模式语言，而模式语言使我们从将同样的模式序列应用到不完全吻合上下文的困境中解放出来。

该假设听起来极具前景。因此，后面我们将详细探讨模式语言这一概念，也会继续讨论模式语言是否能够达到我们对使用模式进行设计的期望，或者说，达到这一目标，还有什么需要克服的限制。

现在我们要仔细看看一个丰富多彩又错综复杂的小镇布局是如何通过成千上万个创作步骤发展起来的。一度，在小镇上我们拥有一种通用的模式语言，而通过普通的方式，我们要将生命力赋予这个小镇的街道和建筑。就像一粒种子一样，这种语言是让无数个小动作形成一个整体的力量的通用系统。

—— Christopher Alexander, 《建筑的永恒之道》

本章通过重新审查之前的请求处理框架例子，总结可能的相关语言而不仅仅是产生该框架的序列，深入探讨了模式语言。我们还要回顾模式语言中上下文和相关性的问题。语法为我们建立这种连接提供了一个有用且相关的隐喻。

10.1 模式网络

结合第9章的讨论，我们可能会试图将软件模式语言概括为如下定义。

它是紧密交织的模式网络，定义了系统化解决相关且相互依赖的一系列软件开发问题的流程。

虽然该描述与模式语言的相关讨论和出版物内容相吻合（例如，[Cope96]、[Gab96]、[Cope97]、[POSA2]、[HaKo04]），但它并不完整。正如单个模式并不只是“特定上下文产生的问题的一个解决方案”，模式语言也绝不仅仅是“紧密交织的模式网络，它定义了系统化地解决相关且相互依赖的一系列软件开发问题的流程”。因此，本章会更深入地讨论软件模式语言是什么以及软件模式语言可以是什么。

10.2 流程与物件

第9章针对请求处理的模式语言草案的分析显示，它强调了单个模式的一个重要属性：它既提供了流程又提供了物件，而该“物件”是由该“流程”创建的[Ale79]。在我们的例子中，物件是请求处理框架。一般来说，它可以是特定类型的软件系统、系统的某一部分、系统的某一个技

术视角或者感兴趣的开发流程，比如分布式系统[POSA4]、中间层资源管理[POSA3]、安全[SFHBS06]、C++中的对象生命周期管理[Hen01b]，或是某个企业的开发流程[Cun96]。该流程是模式语言的有效模式序列及其组成模式的创建流程共同定义的。

对于模式语言来说，同时作为流程和物件是很重要的属性，否则，它们将不能像上面的模式语言定义所宣称的那样，支持系统地解决一系列相关并相互依赖的软件开发问题。提供这种支持既需要如何解决这些问题的程序化指导，也需要从设计或实现的角度对哪种具体解决方案进行建议的信息。

10.2.1 流程的迭代本质

在一种模式语言中，流程和物件紧紧交织在一起。一个或者多个模式定义了它的“切入点”，来处理构建语言的“物件”时必须解决的最基本的问题。这些切入点模式同样也定义了语言流程的初始点：从这里开始使用模式语言。

例如，在第9章请求处理模式语言的例子中，该模式语言合并了3个不同的模式序列，它有两个切入点：如果服务请求是过程式的，就用Command模式；如果请求消息来自远程客户端，就使用Message模式。两种模式都着力处理怎样表示客户端发往应用的服务请求。举例来说，在《面向模式的软件架构（卷4）》[POSA4]中，Command模式的描述包含下面的问题陈述。

通常，访问对象就是调用它的方法。但有时我们希望将请求发送者与请求的接收者解耦合，或者是将对请求和接收对象的选择与何时执行请求解耦合。

既然模式既是流程又是物件，那么每一个切入点模式都描述了解决该问题需要创建哪种具体结构，以及实施哪些具体活动。举例来说，Command模式定义了以下解决方案及其相关的创建过程。

将发送给接收对象的请求封装在Command对象中，并为这些对象定义一个通用接口来执行它们所代表的请求。

在定义核心解决方案及其基本实现步骤的同时，切入点模式的创建过程还建议模式语言中的哪些模式可以用于处理原问题的子问题。如果解决特定子问题时涉及多个替代模式，那么应该列出每种替代模式的优劣权衡，同时也为特定应用提供如何选定“正确”模式的线索。

下面是探讨Command模式时的一段节选。

要实现Command模式，首先要定义Explicit Interface来统一Command的执行。该接口将会定义一个或多个执行方法，并根据需要接收参数。具体的Command实现该接口来具体化特定的请求。在具体的Command初始化过程中传入其执行过程所需要的状态以备后用，比如接收者对象或者方法参数。Interpreter是一种特殊形式的Command，在这种模式中，简单语言的脚本被转换为可执行的运行时结构。

Command对象可以通过保留回滚它们执行的行为所必需的状态，来提供一种取消机制。但是，如果状态的数据量很大，或者难以恢复，Memento模式在执行Command前对接收者的

状态进行快照可能是更简单、更轻量级的方案。

Composite结构支持创建和执行宏Command，将多个Command对象聚合起来，统一到一个单独的接口背后，并可以以特定顺序执行或回滚这些Command。独立的Command Processor根据发送者行为执行Command对象，可以提供额外的请求处理支持，如多次取消/重做、调度和日志功能。

请注意，关于Command模式的讨论是怎样说明其实现的某些方面是必需的，而其他方面是可选的。Explicit Interface是必需的，因为如果不是这样，就无法让所有的命令对象共享同样的接口，也无法解耦客户端与命令对象的实现。而实现Command的其他方面都是可选的。在不使用宏Command时，不应该使用Composite^①。由于客观世界中事物的改变而导致请求不能撤销（如格式化磁盘或打印文档）的系统中，Command实现不需要使用Memento对象。只有需要集中化的命令执行基础设施时，才用得上Command Processor。最后，只有在处理脚本时，Command才会实现为Interpreter——这个角度在我们的示例模式语言中没有考虑到。

Message模式的描述[POSA4]主要是关于如何表示请求、该问题的核心解决方案以及该解决方案基于模式的实现的。

分布式组件像并列组件一样协作，调用彼此的服务并相互交换数据。但是，HTTP这样的网络协议只支持比特流这种最简单的数据传输形式，并不能识别服务调用和数据类型。

[……]

因此：

将需要通过网络传输的方法请求与数据结构封装到Message中：比特流包含一个信息头的比特流（它定义了传输的信息类型、其来源、目标、大小以及其他结构信息）和载体(payload)，后者包含实际的信息内容。

[……]

独立的Command Processor可以帮助我们将特定Message（或是Message队列）转换为对接收者具体方法的调用，并能提供更多的请求处理支持，例如多重取消/重做、调度和日志功能。

在请求处理的模式语言示例中，实现Command及Message模式时，某些要素是必需的，而其他要素可以根据情况权衡，这就限制了可用的有效模式序列集合和我们对未经定义的新模式序列的使用。

举例来说，根据上面的模式定义，不包含Explicit Interface模式而实现Command的模式序列是无效的，而模式序列Command、Explicit Interface既有效又完整。这种限制并不是说Command只能与Explicit Interface一起使用——不使用Explicit Interface，这样一个框架可以通过动态类型的语言或是通过使用部分实现的抽象类来完成。但在示例模式语言的上下文中，Explicit Interface是一

^① 尽管实现Command时Composite是可选的，但是在实际应用中，这是一个常见的配置。实际上，这种配置已经被作为一种模式复合记录在文献中，称为Composite Command，参见第6章。

个可取的必要的设计，同时也构成语言风格的重要组成部分。这种严格的松耦合简化了跨越组件和进程界限的扩展行为。对比而言，没有包含Command Processor的用于实现Command模式的模式序列依然是有效的：在Command模式描述中，Command Processor是可选项。

因此，在模式语言中，模式的创建过程明确指出了哪些引用模式处理模式实现中的必需部分，哪些引用模式处理可选部分。为了确保模式实现稳定可靠，强制使用的引用模式以一种建议的应用顺序被呈现出来。因为可选引用模式只是对模式核心设计的某些方面进行优化，或是仅仅针对特定应用对模式进行扩展，它们可以按任何不影响强制模式使用顺序的安排来进行应用。如果没有这种对过程的显式引导，作为一种基本的模式网络，模式语言可能就无法提供合适的过程，从而无法帮助确立一个足够完整合理的设计。

在模式语言的上下文中，一个模式的引用模式后面往往跟着另一个模式及其相关的创建过程。这个模式也被用于解决其所针对的问题。这里的第二个模式还可以引用其他模式，来解决最初问题的子问题的子问题，正如下面的请求处理模式语言示例所采用的方法所示。

Command和Message模式都建议使用Command Processor来提供高级请求处理功能的基础设施。Command Processor引用了其他6种模式：Collections for States、Template Method、Interceptor、Strategy、Strategized Locking以及Monitor Object。

其中一些模式彼此之间可以相互替换：Template Method、Interceptor和Strategy都着力于命令处理协作功能的多样性。Strategized Locking和Monitor Object都处理线程安全的问题。

Command Processor引用的几个模式可以进一步分解为其他模式。例如，前面说的着力于命令处理协作功能的多样性的模式中的Strategy，又建议使用另外两种模式——Explicit Interface和Null Object来对其进行实现。

图10-1展示了请求处理模式语言示例中这方面的内容。

这种“关系谱”的迭代过程——一个模式后面跟着它的引用模式及其创建过程——反复呈现，直到模式语言中没有其他的引用模式为止。该完整路径通过模式语言示例定义了一个模式序列，产生了请求处理框架的一种可能设计。

其他模式语言定义的过程也遵从同样的原则。举例来说，在POSA4中描述的分布式计算模式语言以Domain Model模式开始，该模式引用了其他13种模式以帮助其实现[POSA4]。对于Domain Model实现的5个方面，都提供了两种或更多的替代模式，如图10-2所示。

每个引用模式都使用语言中的其他模式，而被用到的模式进一步由其他模式组成。总地来说，该模式语言连接了模式著作中大约300种与构建分布式软件系统相关的模式。

类似地，*Organizational Patterns of Agile Software Development*收录了有关项目管理、开发过程、项目组织以及“人和代码”的管理等模式语言[CoHa04]。其中每种模式语言由23到26种紧密相关和集成的模式组成，各种模式语言之间彼此共享一些模式。

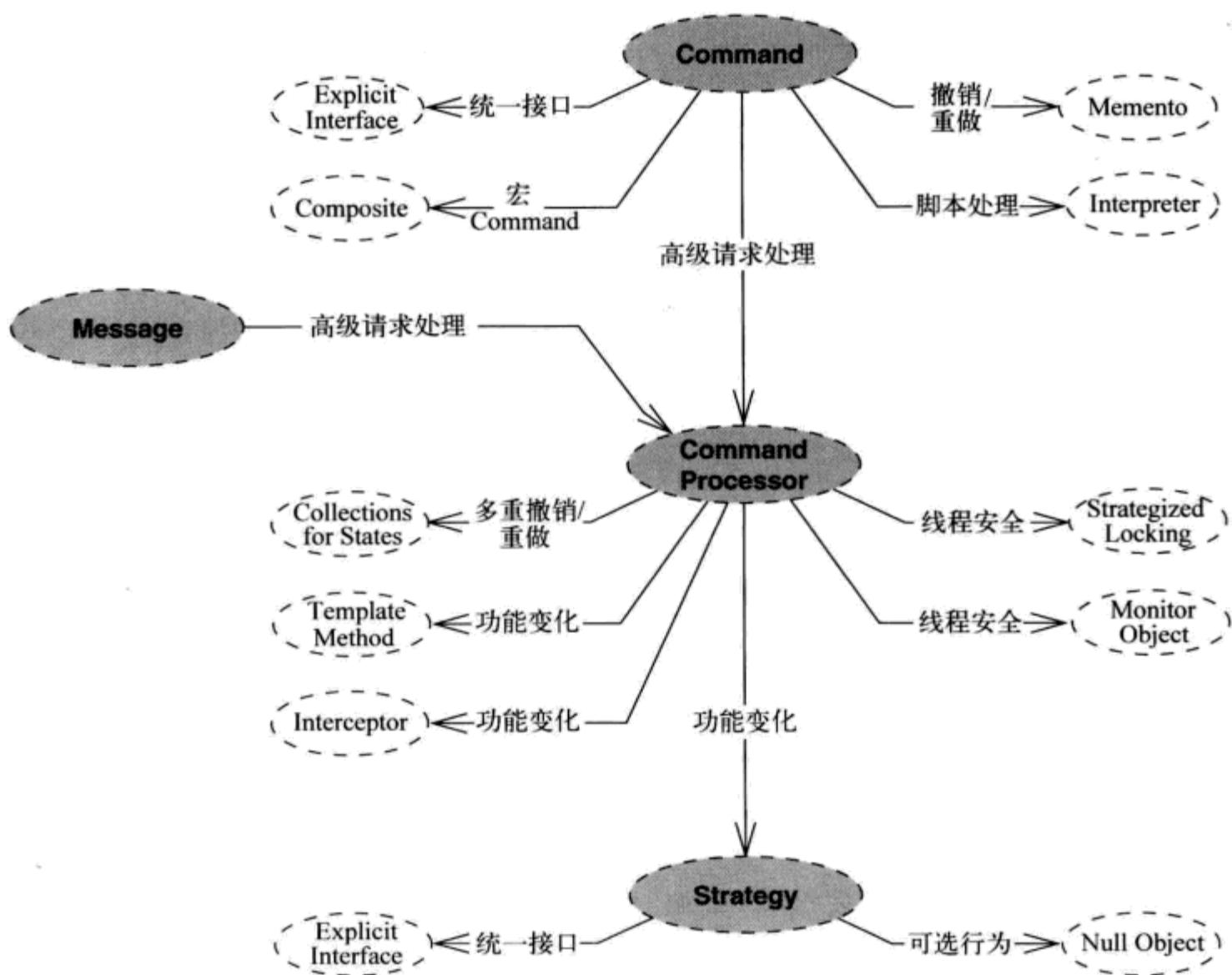


图 10-1

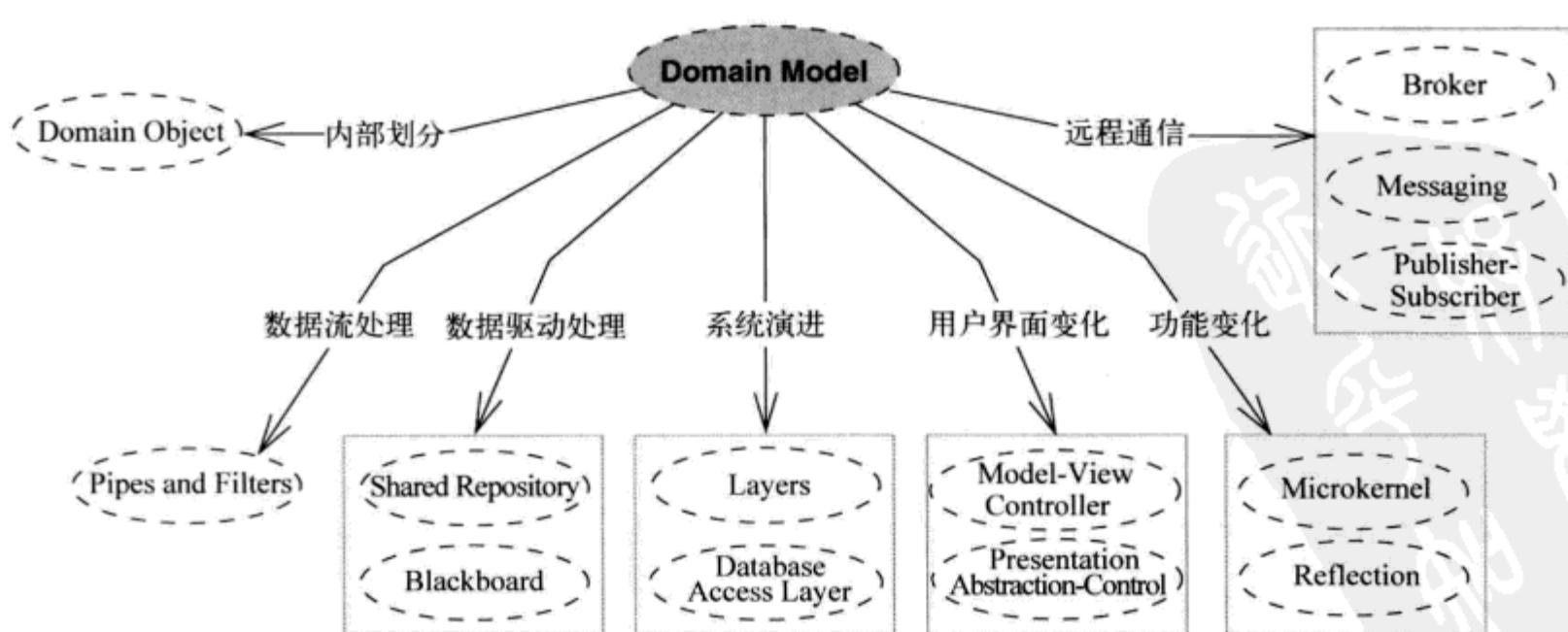


图 10-2

包含许多模式语言的图解常常能概览其定义的拓扑结构。这样的例子包括（当然不只是局

限于) C++中的对象生命周期管理[Hen01b]、上下文封装[Hen05b]、Java中的异常处理[Haa03]、通信交换系统[HaKo04]、搜索的用户界面设计[Wel05]、群件系统中的会话管理[LS05]、应用层的通信网关[Sch98]、对象请求代理[SC00]、事件分发框架[POS05]、分布式系统[POSA4]以及敏捷软件开发[CoHa04]这些领域的模式语言。

10.2.2 具体的面向领域的引导

模式语言定义的过程与通用目的的软件开发流程有着非常大的不同,如瀑布模型[Boe82]、V模型[Drö00]、Rational统一过程(RUP)[Kru00]、XP[Beck04]或者Scrum[SB01]。其中最明显的区别在于软件设计模式语言是与具体领域高度相关的。它对开发或重构某种特定类型的系统或系统方面提供了具体而周全的引导,具体包括以下几项。

- 要解决的主要问题有哪些。
- 这些问题应该以什么样的先后次序解决。
- 解决一个给定问题,有什么可用的替代解决方案。
- 怎样处理问题之间的依赖性。
- 在有“周边”问题存在的情况下,怎样最有效地解决单个问题。

Gerard Meszaros甚至更强烈地表示了需要具体指导[Mes01]:“模式语言应该用来指导生手创建系统。”在此语境中一个有效的问题是,多生疏的门外汉可以使用模式语言来构建系统?如我们在第0章看到的,模式并不能代替坚实的设计和实现技能。模式语言也是这样:没有或有少许软件开发经验的人很难创建出高效运作的软件,即使使用了模式语言也是这样。这里的生手并不是指没有任何技能,或是没有适当基础知识的人。

上面所有关于模式语言所包含的过程以及该过程所包含的指导作用的讨论,同样适用于软件设计以外其他领域的模式语言。例如, *Organizational Patterns for Agile Software Development* [CoHa04]中记录的4种模式语言提供了面向领域的具体过程以及具体全面的指导,包括对于怎样进行项目管理、组织(软件)开发流程、调整项目组织结构以及管理代码和人力。尽管这些模式语言提供了指导,但在具体的软件项目中应用它们需要相应经验和技能。一位熟悉不同组织结构和过程的模式语言的项目经理,也依然需要合适的人际和管理技巧。

然而,虽然一种模式语言就它所覆盖过程的不同使用来说具有一定的通用性,但请注意这并不是说模式语言作为一个整体可以用于开发中的任何问题。例如,关注设计的模式语言(如请求处理的模式语言示例),虽然提供了设计的细节,但一点也没有提供关于开发组织、人员管理、工具使用等方面的信息。过程和组织方面的模式语言关注这些问题[Ker95][Cun96][CoHa04],但并不涉及设计的具体方面。

尽管如此,在具体的软件项目中,多种模式语言可以一起应用,每种模式语言用于处理软件开发中特定的方面。例如,开发分布式系统的项目可以使用分布式计算的模式语言来定义系统的软件架构[POSA4],使用渐进式成长的模式语言来定义系统的开发流程[CoHa04]以及使用项目管理的模式语言来指导开发[CoHa04]。使用这样一种混合方式,所有重要的问题都有相应的模式语言处理。对于其中的每个问题,都有已定义的、面向领域的具体指导,这些指导以“物件”以及

创建它的过程的形式提供。

对照之下，通用目的的开发流程反而着力于任意软件系统的构建。因此，通用目的的开发流程必须是宽泛而笼统的，也就只能提供非常基本的建议。举例来说，RUP只是解释了整个软件开发活动应该以什么样的顺序和混合方式进行，以及在设计软件架构时应该考虑哪些主要的技术[Kru00]。

但是，根据这些彼此之间没有联系的指导信息，没有经验的软件开发人员依然没有办法了解怎样建立或重构他们的软件架构，也无法在面对特定架构挑战时得到具体的提示或支持。相反，模式语言定义的过程对怎样设计特定类型的软件系统和系统方面，或是怎样处理软件开发中其他相关问题，如项目组织和管理、开发流程、配置、代码和人员管理等，提供了完整、明确的指导。当然，这种权衡是因为每种语言在其特殊的上下文中才能被有效应用，在其上下文之外就不那么有效或是根本就无效了。

10.3 单项最佳

为确定和完善适当的软件架构或软件的特征，模式语言不仅提供了具体周密的过程，还支持基于成功生产系统的可行开发经验的、同样具体、周密的软件设计和实现。在这里，我们要再次强调和调整在第1章中提出的关于单个模式的观点，好的模式语言并不是对可能能够工作的简单想法的展示，而是提供了在以往经验中被反复成功应用的设计和实现，这使得它们成为模式语言，而不是设计文档。

请求处理的模式语言示例同样证明了自己的价值：根据4.3节中介绍性例子的第二部分所示的设计，多个产品级应用[Bus03a] [BGHS98] [Dorn02]实现了请求处理和（用户）命令执行。

要确保模式语言能够创建“单项最佳”的解决方案并实现高效的开发，需要遵循以下3个标准。

(1) 充分覆盖。模式语言必须包括正确的模式来对应其主题之下各个方面和不同的子问题，否则它将无法支持有用软件的创建。

(2) 进展可持续。模式语言必须将它们所包含的模式恰当联系起来，以确保能够以正确的顺序来克服遇到的问题，这对于通过稳定的中间步骤，渐进地创建可持续的设计是非常关键的。

(3) 紧密集成。模式语言必须根据每个模式引入的角色和模式之间的相互关系，紧密地将组成它们的模式集成起来。

这里的第三个标准对于确保不同设计和实现挑战的解决方案之间相互支持和补充、从架构层提高它们的质量，是很有必要的。举例来说，回顾我们在第4章中通过应用同样的模式序列创建的两种设计。第一种设计（有意地）认为模式是孤立、自包含的构建块，以一种严格模块化的形式被塞到一起——这种观点是与上面的第三个质量标准完全对立的。这种“忽略”导致了一个不甚理想的解决方案，虽然它非常满足第一条标准，同时也基本满足第二条标准。

相反，由于严格遵照第三条标准，第二个设计反映了所有三条标准，即它使用的模式以交互角色集合的形式运作，并根据它们之间的内部关系进行组合和集成。与前一种设计相比，该设计也更好、更充分地满足了第二条标准。

图10-3回顾了这两种设计。两种设计的质量高下立见。虽然“模块”化的设计过度强调使其中涉及的每个模式都可以清楚分辨出来，但它过于复杂、不灵活、效率低而且难以理解。而“集成”式的设计尽管由于其紧密的集成有时候难以发现和辨识其所组成的模式，但它精简、灵活、高效、易于理解并且非常平衡。

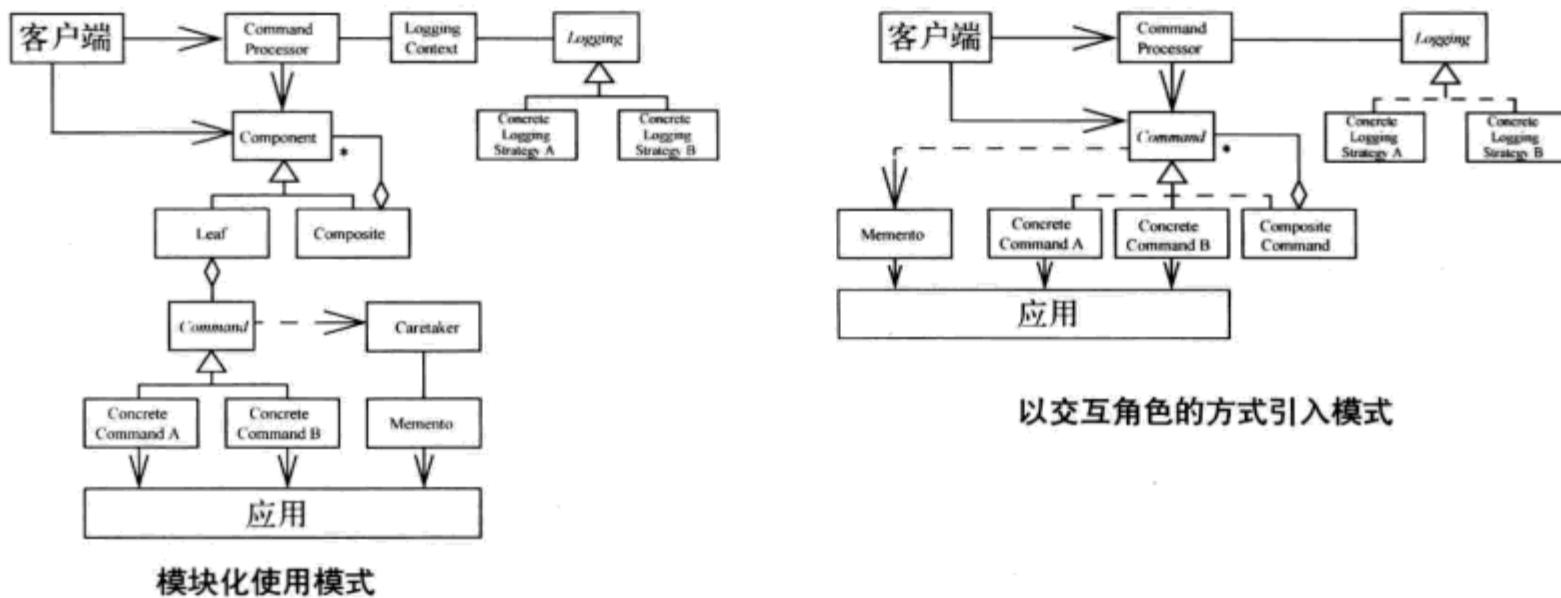


图 10-3

如果在这些模式描述中没有具体的引导，在应用模式示例语言时，很可能创建出第一种设计。因此，模式语言中每种模式的描述必须准确指明其他哪些模式帮助实现它的哪种角色。仅仅描述其他模式帮助解决哪些子问题是不够的。

下面是对Command Processor模式描述的节选[POSA4]。注意这段描述怎样介绍其他模式来帮助实现其命令处理器的角色，而不仅仅是Command Processor模式实现的大概说明。

在Command Processor的内部，它可以通过多个Collections for States来管理它接收的请求。例如，可以用do集合来保存所有待执行的请求和被撤销后可以重做的请求，用undo集合来保存组件上所有执行过的请求，以便将来进行回滚。如果Command Processor收到的请求本身就是Command对象，它可以简单地调用这些对象来执行组件上封装好的请求。如果收到的请求是Message对象，则可以使用Interpreter将其内容转换成Command。

调度和其他辅助性功能（比如日志和授权）可以通过Template Method、Strategy或者Interceptor来实现或者进行配置。通过这些模式，我们可以使用不同的请求执行策略来对Command Processor进行配置，对绑定时间和耦合程度作出权衡和取舍。[……]

尽管在上面讨论了3条质量标准，但就像我们无法保证单个模式能够被正确应用一样，也无法确保模式语言能够被正确应用。例如，虽然可以使用得到验证的模式语言进行设计，但无法保证开发者将会选择合适的特定解决方案，或者说会完全按照建议解决方案的步骤和细节来完成。导致的结果可能是虽然系统最终还是能够运作，但跟要求不一致或者说效果不够理想。同样，对于主题的某个特定重要方面，模式语言还有可能无法处理或者无法完全解决。使用该模式语言创

建的软件将不可避免地具有相应的缺陷。但是在模式语言文档化时，仔细地、有意识地处理这些问题对于减少错误并创建高质量的软件还是很有帮助的。

不完整的模式语言产生蹩脚的解决方案

模式语言的误用和不完整并不是导致不适当解决方案的唯一原因。我们可以分辨出“差”的模式和“好”的模式，或者分辨出不完整的模式和完整的模式，同理，我们也可以将不完整的模式语言同完整的模式语言区别开来。这些不完整的模式语言在很大程度上来源于不满足前面提到的一条或多条质量标准，如下所示。

(1) 覆盖不充分。在不完整的模式语言中，往往存在用不合适的模式来应对主题中的各种问题或子问题的情况，或者就是缺少处理这种问题和子问题的模式。

(2) 进展不可持续。在不完整的模式语言中，很多模式往往是独立存在的，或是仅仅公开了与其他模式的一部分关系，而不是完全地与模式网络相联。另一个常见的问题是选择了不适当的切入点：模式语言的根定义了它的出发点和假设，但并不难见到模式语言在处理领域问题之前，从一个实现细节开始的做法。这种做法最终导致我们很难提供一个关于语言所面向主题的、提供具体引导的过程，用以确保以正确的顺序处理各种问题，并按照渐进的、稳定的步骤创建可持续的设计。实际上，提供可能的模式组合和序列作用有限，它们并不能帮助创建稳定完善的设计。

(3) 松散集成。由于不支持可持续进展，不完整的模式语言无法支持其组成模式的基于角色的紧密集成，这很可能导致将模式作为实现之间几乎没有重叠的构建块进行处理的设计。这会引起这些语言中单个模式允诺的很多质量属性（比如性能和适应性）在使用这些语言创建的设计中无法得到满足。

应用不完整模式语言的后果很明显：所产生的软件无法满足它在功能、运行和开发上的需求。换句话说，使用不完整的模式语言可能会设计出不完整的解决方案。

我们没有发现什么正式的方法，可以帮助我们区分完整的模式语言与不完整的模式语言。但有一个不太正式的技巧：组成模式语言的成功模式序列越多，以及来自生产系统的（成功）模式故事越多，该语言完整的可能性就越大。这个简单的观点重新回到了本节开始时的讨论上：好的模式语言不是构建于传奇故事之上，而是构建在验证过的经验和真实的成功案例之上：经验主义对比理想主义。

这些关于质量的观点和陈述适用于各种类型的模式语言，而不仅仅针对软件设计。举例来说，在项目管理中应用不完整的模式语言可以导致失败的项目管理。

10.4 驱动力：模式语言之心脏

成功的面向设计的模式语言，其基本前提在于它是否能够创建可持续的软件架构。但是，模式语言能够很好地处理特定主题的问题，并不能保证它能够被成功应用。开发团队的技术水平与产品应用的具体需求一起，决定了使用模式语言创建的设计或实现的质量。如果某个解决方案没能处理和满足其目标软件系统的需求，那么这个方案就没有达到系统的质量要求。而且，在某领域一个应用中运作得非常完美的解决方案，也许对其他领域中的其他应用就不那么合适了。

因此，模式语言通过列举它们所针对的特定需求、其所面临的种种限制条件以及它们能够产生的设计所支持的特性，描述了它们的应用范围。这种精确的描述可以帮助我们减少模式语言的误用。通过比较模式语言处理的需求和根据特定应用定义的需求，开发者可以决定给定模式语言是否适用于他们的需求。

有用的模式语言能够为开发者提供常见设计实现问题的现有解决方案，还能建议其他可替代模式语言，甚至提出不使用模式的解决方法。例如，请求处理的模式语言示例关注的主要问题是灵活性：可配置的命令集合、命令组合、命令调度以及辅助性功能。与只需要原始功能的应用相比，需要灵活的命令处理功能的应用可以从该模式语言中得到更多的益处。

正如1.4节中描述的，在问题的上下文中，单个模式可以使用这些显式驱动力来明确它们所处理、支持和考虑的具体需求、属性和限制。驱动力告诉我们为什么模式处理的问题需要“聪明”的解决方案。它们也是了解特定解决方案基本原理的关键。那么很自然地，我们也可以为模式语言定义它们的驱动力，以显式表示在使用它们进行设计和实现时所处理的需求、所支持的属性及所考虑的限制。

驱动力的三个等级

但是，模式语言的驱动力有哪些呢？对于这个问题，最直接的答案或许是：模式语言的驱动力是其所有组成模式的驱动力的并集。这个答案当然是正确的：如果模式是某主题模式语言的组成部分，那么该模式所处理的问题与该主题的具体上下文相关，因此，是驱动力影响了该问题的解决方案。但这只是部分回答了我们的问题，从整个模式语言层面上来看，甚至没有什么意义。

模式语言必须平衡与其整个主题全局相关的驱动力，而不是仅仅平衡在为该主题创建特定设计和实现时遇到的单个问题的驱动力。举例如下。

- 对于请求处理的模式语言示例，可配置性的需求是一个全局的驱动力，对服务质量进行控制是另一个。
- *Server Component Patterns*[VSW02]中定义的组件设计的模式语言针对几个全局驱动力，如对独立组件开发、演进、重用、扩充性和可用性的支持。
- POSA4分布式计算的模式语言关注分布式软件系统中的易变性，而不影响它们的操作质量（如性能和可扩展性）[POSA4]。
- 模式语言Caterpillar's Fate（帮助我们将详细的分析文档转化为最初的软件设计）的全局驱动力是尽可能无缝转换[Ker95]。

因此，应该是模式语言中的所有模式作为整体一起来解决全局驱动力，而不只是任意给定单个模式中的驱动力——这将对解决方法的某些部分有所帮助。

有些模式语言还面向与它们的主题相关的几个问题空间。例如，POSA4分布式计算的模式语言由13个问题空间组成，包括分布式基础设施、并发、事件处理和数据存取等。常常存在与问题空间相关的驱动力，这些驱动力由组成语言的所有模式来处理。举例来说，在POSA4模式语言中，事件处理的问题空间列出了几个与事件吞吐、异步事件到达和并行时间处理相关的驱动力。同样，Caterpillar's Fate模式语言由4个问题空间组成：并发、交易、程序形态和数据设计[Ker95]。每个

问题空间针对相应的驱动力，比如在程序形态问题空间中怎样捕获和应对刺激 (stimuli)，在数据设计问题空间中怎样处理下行数据的兼容性。请求处理的模式语言示例同样能根据驱动力集合识别出两个问题空间。

- 请求表现。与问题空间相关的驱动力是对可配置请求组合和可扩展请求表现的要求。
- 请求处理。与问题空间相关的驱动力是对可变请求处理行为的要求。

总地来说，模式语言的驱动力可分成3种级别：系统（部分或方面）级别、问题空间级别和单个问题级别。较高层的驱动力总结和压缩了下一层的驱动力。同样，较低层的驱动力展开和描述了其上层的某些驱动力。所有的驱动力都是相互关联的，引导模式语言的使用者逐渐从系统级别的视角细化到实现级别的视角。因此，在创建软件的过程中，用户获得了使用模式语言时需要平衡的驱动力的准确细节。

图10-4展示了在请求处理的模式语言示例中，可配置请求处理的全局驱动力是怎样通过它的两个问题空间中相应的驱动力以及两个问题空间中所选择的模式表现出来的。

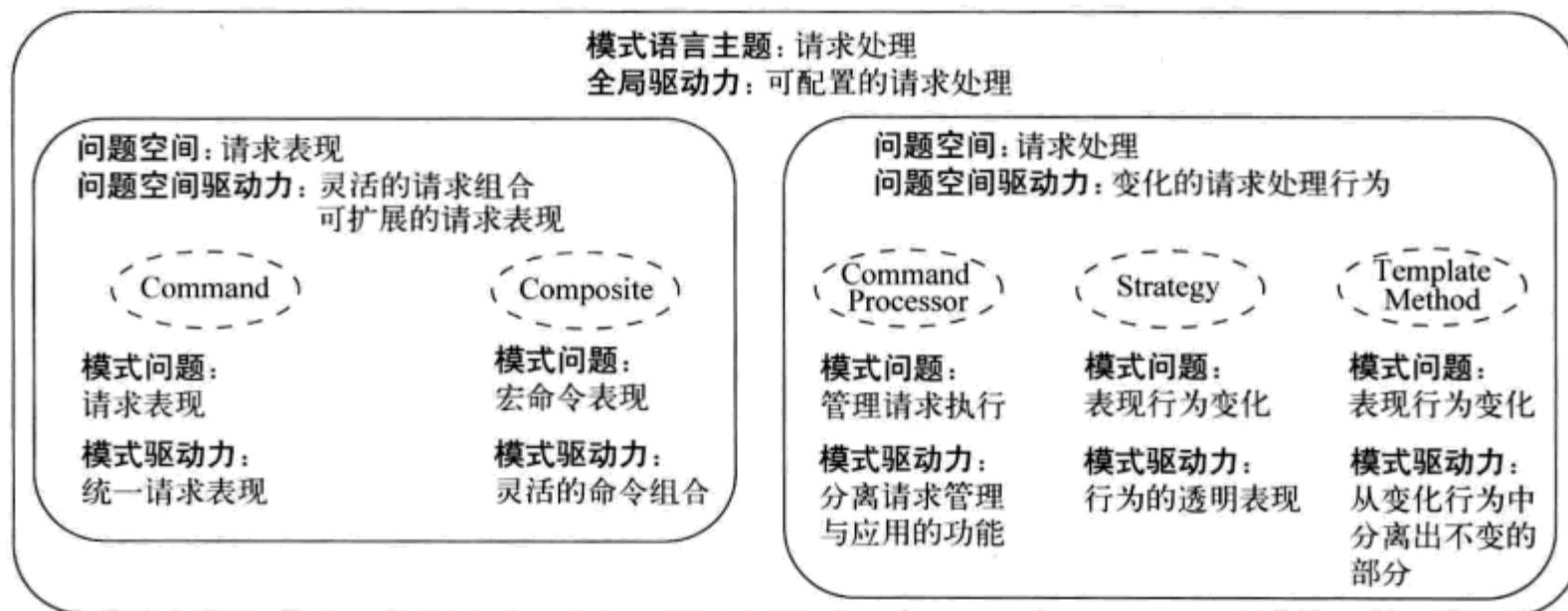


图 10-4

除了帮助定义模式语言的需求、限制和属性外，驱动力还帮助用户理解它们之间的拓扑结构。当模式语言中一种模式的描述引用了其他模式时，它应该向使用者说明引用模式可应用的环境。这种帮助往往表示为处理特定问题的上下文中引出的相应驱动力。当涉及多种替代模式时，这种指导特别有用，因为在给定的条件下，正是与替代模式相关的驱动力决定了哪种模式最为适合。

例如，在请求处理的模式语言示例中，Command Processor模式为处理请求的辅助性功能建议了3种替换选项[POSA4]。

调度和其他辅助性功能（比如日志和授权）可以通过Template Method、Strategy或者Interceptor来实现或者配置。通过这些模式，我们可以根据绑定时间和耦合程度，使用不同的请求执行策略来对Command Processor进行配置。

应用哪种替代模式，或者说是否使用任何替代模式，是由相关驱动力决定的，如Command

Processor示例所示。如果模式的驱动力满足当前的具体需求和限制，就可以应用该模式，否则就不用。这样，在通过模式语言选择特定实现方式的过程中，驱动力起到了非常关键的作用。它们像路标一样，在模式语言范围内指导用户从一种模式到另一种模式，为开发中的软件系统选择最合适的模式序列。

10.5 模式上下文定义拓扑结构与架构风格

在第一部分中，我们探讨了单个模式的上下文，描述了可能需要使用该模式的各种情况的关键特征，展示了上下文怎样为模式定义了一个准确清晰的应用范围。单个模式的上下文、问题及驱动力形成了一个三部曲：上下文必须要能引发问题，驱动力确定了需求，而这些需求要在给定上下文中得到满足才能解决问题。

在第二部分中，我们从模式序列的视角继续讨论了问题的上下文。除了第一种模式，模式序列中的其他模式都是实现了序列中一个或多个特定模式之后才得以应用的。例如，在产生请求处理机制设计的模式序列中，Command Processor是在Command之后被应用的，而Strategy又在Command Processor之后使用。因此，我们可以将模式序列中某成员的上下文限定在所有在它之前的模式的列表中，或仅仅是它前面的一种模式。这种精简列表定义了精确的应用范围，并且让使用者无需关心与创建该序列主题无关的情况。

因此，列出之前的模式也是定义模式语言中模式上下文的一个常见方法。举例来说，在请求处理的模式语言示例中，Strategy模式的上下文可以非常简单、精练、准确地描述为：

Command Processor被用于实现高级的请求调度和辅助性功能。

模式语言中的模式可能有不止一个切入点，相应地，它们的上下文可以枚举不止一种模式。比如，在我们的例子中，Explicit Interface的上下文可以同时列出Command与Strategy模式。

模式语言中所有模式的上下文联合在一起，定义了该语言的拓扑结构。该拓扑结构描述了一个模式怎样与其他模式相联系，还包含了语言中哪些模式可能按照哪种顺序排列这样的信息。因此，这些联系形成了模式的一个部分排序集合[PCW05] [Hen05b]。

模式（比如说上面的Strategy模式）的上下文也与其所在的模式语言相关。举例来说，在对象关系数据映射的模式语言中，Strategy会有不同的上下文。与专门模式编目中模式之间的关系不一样，模式语言并没有为其模式提供通用的连接构架（connection schema），但是却描述了它们的模式在特定领域内怎样相互联系。

与“唯一真正的解决方案”不同，模式语言定义了一种或多种架构风格。在使用其他模式的同时，同一领域的不同模式语言可以以不同的方式连接同样的模式。例如，比较两种将对象映射到关系数据库的模式语言[BW95] [KC97] [Kel99] [Fow02]：尽管有很多共同之处，但两者分别采用了不同的方式来设计数据库访问层。正如互补模式之间的不同往往简化为设计风格的不同，互补模式语言之间也是这样，只不过换成了它们的架构风格的不同。这两类不同的区别在于，如第8章中讨论的那样，模式的设计风格主要由它的驱动力决定，而模式语言的架构风格由组成它的模式之上下文所定义。

语言特定上下文与通用上下文

当要理解模式是如何联系起来定义语言的架构风格时, 特定于语言的上下文是必需的。但当你试图理解特定模式时, 不管模式是否集成到语言中, 这些上下文常常没什么用处。前边提到的Strategy模式的上下文是一个很好的例子。虽然在请求处理的模式语言示例中, 其上下文精确陈述了Strategy模式怎样与其他模式相关, 但该上下文并没有提供何时应用Strategy模式的信息。因为上下文、问题和驱动力这个三部曲不甚完整, 使用者很难理解Strategy在该语言中扮演的特定角色。

不管Strategy模式是怎样集成到模式语言中的, 现在缺少的是它何时应用的通用信息。这些信息应该能够提供理解该模式的一个起点。因此, 除了列出前面应用的模式外, 模式语言中模式的上下文可以加上它作为单个模式被文档化时的一些描述。

当使用Command Processor来实现请求调度和辅助性功能时……

……我们必须考虑到在不同的应用中, 为实现这些功能可能需要用到不同的算法和策略。

该上下文向读者说明了自己在模式语言中所处的位置, 以及激励使用特定模式的关键特征。读者可以借此理解语言中的每种模式, 而不必去阅读和消化前面所有可能使用过的模式。而且, 语言中每种模式的可用范围可以更加准确清楚: 该模式的上下文、问题和驱动力三部曲显得更加平衡, 联系更加紧密——更像是三幅相连的画而不是一团神秘的密码了。

很多模式语言在自己的模式描述中采用了上面讨论的上下文的形式。例如, 在POSA4中, 分布式计算的模式语言就非常严格地遵照了这种格式。语言中其实现可以得益于所描述模式的包含模式首先被列出, 紧跟着是一段关于需要使用该模式的情况的描述。*Remoting Patterns*[VKZ04]、*Server Component Patterns*[VSW02]以及*Software Configuration Management Patterns*[BeAp02]中使用的上下文形式也很相似。这些书中的每种模式都列出了其包含模式和产生问题的主要情形, 但并没有像POSA4那样将两个方面显式分开。

10.6 模式构成词汇, 序列展示语法

当将模式组合成模式语言时, 模式可以扮演语言中可用词汇的角色, 即语言中的基本单词, 其中每种模式被当做一个“单词”。针对特定模式语言上下文中的每一个相关问题, 至少有一个“单词”说明了该问题的解决方法。

解决给定问题的替代模式跟自然语言中的同义词类似: 它们具有几乎一致的意思但又有不同之处, 而这种不同是通过替代模式所处理的单个驱动力来表现的。

但是, 词汇自身只是恰当的、有用的模式语言的一部分。其余的部分则是模式语言的语法。什么是模式语言的语法呢? 让我们回到将模式语言看做是网络的观点上, 一种可能是模式语言的语法是由它的拓扑结构定义的。如前面所述, 这种拓扑结构由构成语言的模式的上下文所定义。但是, 这种观点或多或少有点幼稚, 因为拓扑结构不可能是语法。

在10.2.1节探究模式语言定义的过程时，我们注意到如果模式语言中的一个模式引用其他多个模式，这些被引用的模式不能总是以任意顺序被使用。换句话说，并不是模式语言的拓扑结构图中任意一条可能路径都可以创建出可持续、可用的设计。因此，仅仅将模式语言当做一个网络而衍生出来的可能解决方案空间需要缩小，过滤掉一些没有意义的或者不实用的路径。

反过来说，当仅仅把模式语言视做网络时，从语法上看，并不是所有的模式序列都是合适的。要开始理解模式语言的语法，也许应该从语言的序列出发，而不是从其抽象的拓扑结构开始。要继续自然语言的比喻，每种模式序列可以视做模式语言中一个结构良好的句子。

举例来说，请求处理的模式语言示例包含了下面的模式序列。

空序列 $\langle \rangle$ 是一种有效的序列，这意味着不为请求处理使用模式语言。有用的非空模式序列包括 $\langle \text{command, Explicit Interface} \rangle$ 、 $\langle \text{command, Explicit Interface, Memento} \rangle$ 、 $\langle \text{command, Explicit Interface, Memento, Composite} \rangle$ 、 $\langle \text{command, Explicit Interface, Composite} \rangle$ 、 $\langle \text{command, Explicit Interface, Composite, Memento} \rangle$ 和 $\langle \text{Command, Explicit Interface, Memento, Composite, Command Processor, Collections For States, Strategy, Null Object} \rangle$ 。

为了简洁，我们省略了一些模式序列，这些序列主要是上面最后一个模式序列的不同排序版本，或是在序列中包含了Message和Template Method模式。

因此，模式语言支持的所有模式序列的并集可以理解为语言中所有语法正确的语句的全集。但是，模式语言的语法只在这些模式序列内部可见，因为使用模式语言最后看到的是它的应用结果，而不是语言的规则。

有两种方案来使模式语言的语法显式可见。

- **直接集成。**语法规则可以直接被集成到组成语言的模式的描述之中，如10.2.1节Command模式的例子所示。但是，要在模式描述中明确表达语法规则，可能是一项非常复杂和费力的活动。而且，将语法规则直接嵌入到模式描述中并不能给我们提供一个“概览图”——一个对整个语言语法的概览。
- **使用单独的标识。**文档化模式语言的语法的另一种方法是使用正式的标识。举例来说，CSP模型的关键元素[Hoare85]：进程代数（process algebra）被应用在并行系统中通信顺序过程（Communicating Sequential Processes）的描述中[HoJqq]。在模式语言中有一些与之相似的东西：模式语言中的模式构成其开发流程的一个基本符号系统（alphabet），对模式的应用构成事件，语言中可能的模式序列构成流程中一系统轨迹（trace）。另外，也可以使用图形或一段散文来描述模式语言的语法。

作为简单标识的例子，我们可以用 \emptyset 来指示一个开始点，用 \rightarrow 来表示一个强制序列部分，用 \rightarrow^* 来表示可选序列部分，用 $|$ 来表示可选项，并用 $\langle \rangle$ 表示分组。使用这种标识方法，可以得到请求处理模式语言语法的一个片段，它展示了上面所讲的模式序列。

```

 $\emptyset \rightarrow^* (\text{COMMAND} \rightarrow \text{EXPLICIT INTERFACE} \rightarrow^*$ 
 $(\text{MEMENTO} \rightarrow^* \text{COMPOSITE} \rightarrow^* \text{COMMAND PROCESSOR} \rightarrow$ 
 $\text{COLLECTIONS FOR STATES} \rightarrow \text{STRATEGY} \rightarrow \text{NULL OBJECT})$ 
 $| (\text{COMPOSITE} \rightarrow^* \text{MEMENTO}))$ 

```

用于定义编程语言语法的巴科斯范式[EBNF96], 可以作为另一种方法来表示模式语言的语法。散文描述也是一种可选方案。

可以选择是否使用Command模式, 但如果应用了Command模式, 随后必须应用Explicit Interface模式。Explicit Interface模式之后可选地应用Memento模式, 而Memento模式之后必须使用Composite模式, Composite模式之后可选地应用Command Processor模式, 如果后面应用了Command Processor模式, Command Processor之后必须使用Collections for States模式, Collections for States之后则必须使用Strategy模式, Strategy之后又必须使用Null Object模式。或者, Explicit Interface之后可选地应用Composite模式, 且Composite之后可选地应用Memento模式。

图形标识法也可以用来描述模式语言的语法。一种可能的选择是使用特征建模标识, 这种表示方法能够从视觉上捕捉软件设计中的共性和变化[CzEi02], 而且它有一种变体, 专门用于表现上下文封装的模式语言[Hen05b]。在特征建模中, 一个核心的概念就是将事物层级化地分解为特征, 并将特征之间的合法组合关系展示出来, 如两种特征是否是互斥的, 是否是强制一起使用的, 是否是可选的。然而, 在借鉴这种标识法时, 不应该将模式与特征混淆, 尽管这种标记法可以通过可视化的方式将有效的组合展示出来, 以表达概念上的相关性。

模式语言语法的另一种图形表示是使用“路轨”标识法——一种自20世纪70年代开始流行的、用于描述编程语言语法的图解技术 (特别是用于Pascal语言的描述中) [JeWi74]。

请求处理的模式语言示例的这部分可以用“路轨标识”表示为图10-5。

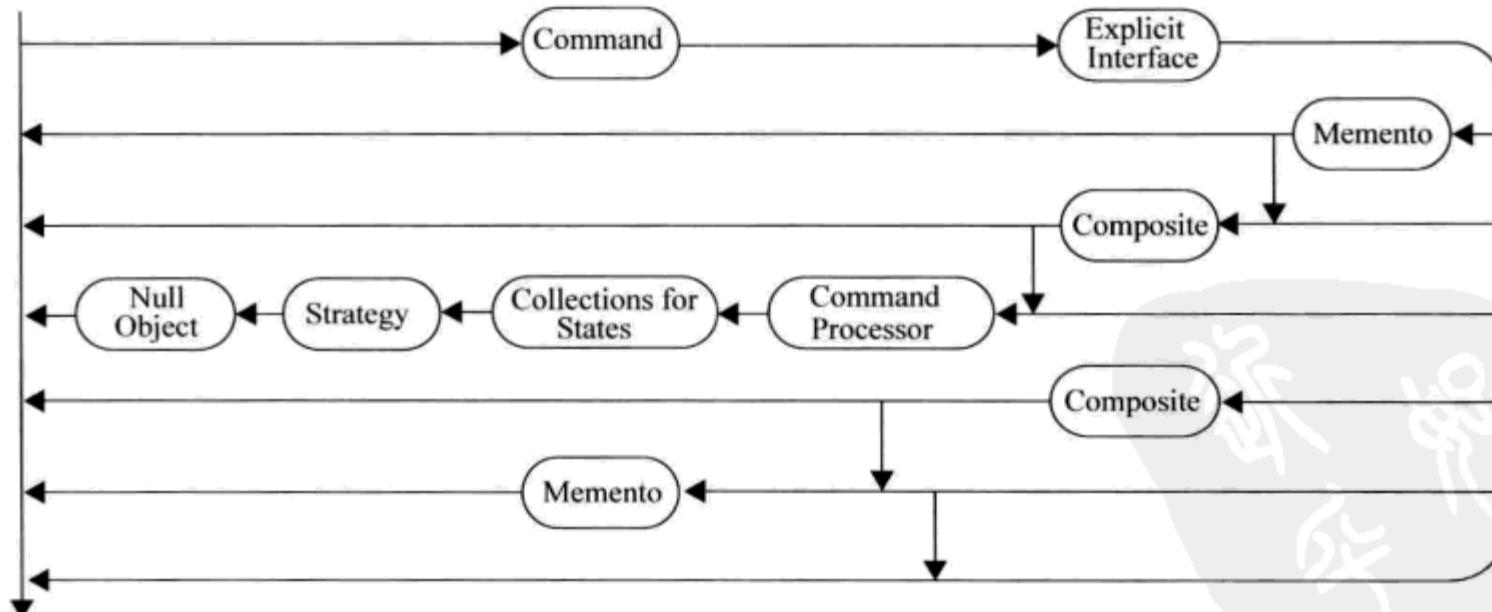


图 10-5

模式语言作者的喜好或目标读者的需求决定了最为合适的语法表示方法——模式序列的列表、正式或半正式的散文、描述语法规则的图形形式、是否穿插着模式描述或是与模式描述分开。例如, POSA4中分布式计算的模式语言将其语法规则用散文的形式表现, 其中穿插了模式描述[POSA4]。这种形式为软件领域中的大多数模式语言所采用, 包括从以设计为中心的模式语言

[VSW02] [Fow02] [HoWo03] [VKZ04] 到针对开发流程和组织的模式语言, 以及项目管理和人员管理的模式语言[Ker95] [Cun96] [CoHa04]。

但是, 无论采用哪种语法形式, 重要的是文档化的模式语言确实能够就某一问题提供有意义的方法指导。否则, 很难避免用到有问题的模式序列, 最后创建出完全失败的软件。语言中合理的序列集合是语言自身的一部分, 而不是什么偶然发现或者独立的事物。因此, 将模式语言的语法表现得更加清楚明白能够帮助其更恰当地应用在生产中。但是, 我们必须意识到这种尝试也有一些限制: 大型的、模式之间联系比较复杂的模式语言的语法很难以容易理解的形式来表现。在这种情况下, 模式的使用步骤说明也许更适合放在每种模式的描述中, 而不是以一种独立的形式存在。

10.7 通用性

单个模式的一个关键属性是为给定问题提供了一个解决方案空间, 而不是一个单独的解决方案。在实现这些模式时, 根据具体情况, 使用者可以高效又正确地应对特定的需求, 同时遵循模式的指导原则以及模式所建议的解决方案的核心思想。

类似地, 模式语言也会为特定类型的系统或系统的某个方面或者某个部分, 产生相应的设计空间, 而不是一个单独的、固定的、适用于所有情况的解决方案。该属性对于模式语言在产品软件开发中的成功应用是非常关键的, 甚至比单个模式更为重要。否则, 模式语言就无法应对某种应用的不同案例中各种不同的需求和限制。但是模式语言怎样才能实现这种通用性呢? 接下来, 我们会探讨这个问题的两种可能答案, 一种是模式语言中可以有很多不同的模式序列来应对不同的需求, 另一种是模式语言中较小型的模式可以组合起来处理很大范围内的问题。

10.7.1 不同的模式序列

在10.2节中, 我们探讨了模式语言中的一种应用步骤可以产生一种模式序列, 而应用该序列可以创建特定的设计或导致特定的情境。几乎所有的模式语言对于它们要解决的问题都提供了多种替代方案。既然有些问题并不会在所有的系统中出现, 那么模式语言中可以有很多不同的模式序列, 并且每一种生成了一种不同的设计和实现。举例来说, 请求处理的模式语言示例就支持创建很多可替换的设计方案。

模式序列〈Command, Explicit Interface〉实现了简单的请求处理机制: 将过程式的服务请求的发送者与提供服务的组件解耦。在此之上, 并不支持其他的请求处理功能。

模式序列〈Command, Explicit Interface, Memento, Composite, Command Processor, Collections for States, Strategy, Null Object〉创建支持高级请求处理功能的设计: 反复撤销/重做、日志功能和调度功能。大部分功能可以使用不同的算法来配置。

模式序列〈Message, Command Processor, Interpreter, Command, Explicit Interface, Memento, Composite, Collections for States, Strategy, Null Object〉创建可替代的设计方案, 该方案中客户端发送请求消息而不是函数调用。

我们可以适当修改Christopher Alexander的一段话作为总结[AIS77]：“每一种模式语言描述了实现特定应用范围或技术领域的软件的整个解决方案空间。通过这种方式，你可以上百万次（甚至上亿次）以不同的方式实现该解决方案。”

10.7.2 模式的组合

软件设计的模式语言具有通用性的另一个原因来源于一个事实：它们所包含的模式并不是以固定的、附有具体代码的设计来描述的，而是通过几个“更小型的”模式组合来定义的。这些更小型的模式，通过它们被包含时的具体安排、它们自身的通用性，以及它们的实现变体的可能组合方式，隐式而不是显式定义了它们组合形成的模式的解决方案空间。

当这些更小型的模式以特定的有效组合和实例的形式实现时，它们会自动生成一个“较大”模式的特定实例。

当使用示例模式语言实现请求处理框架时，如果我们使用Strategy模式，那么有两种方法来实现Command Processor的可变行为。第一种方法是为每种可变算法引入独立的策略层次结构，第二种方法是为每组可变的相关算法创建独立的策略层次结构，如图10-6所示。

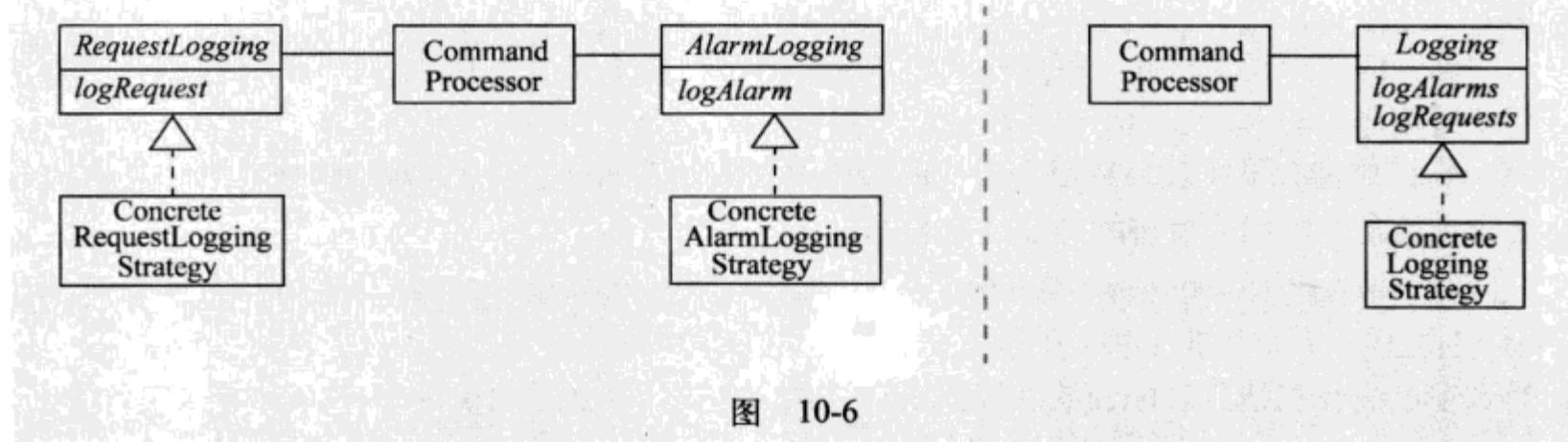


图 10-6

这种“更小型”模式自身可以被定义为更小型模式的组合，而且大多数情况下就是这样定义的。因此，整个模式语言的通用性随着其拓扑结构深度的增加而不断加强。而且，这种通用性也远比语言中单个模式提供的通用性之和更强。每一种集成到模式语言中的新模式都会加强这种通用性。

10.8 完整的语言胜过千幅图

作为补充，为单个模式的文字描述配上图示来描述其组成元素的可能配置是很有用的，但对模式语言来说就难得多。这种额外的复杂性来源于语言的通用性及其包含的相互交织的过程。特别是，模式语言所支持的不同具体设计与实现的数量可能是很惊人的，而两种设计或实现之间又可以存在很大的区别，因此往往很难灵活地将它们总结到一幅图甚至少数几幅图中。

即使模式语言所包含的设计空间可以放在一幅图中，这样一幅图也很可能是另人费解的，从而失去其应有的价值或容易造成误解。这种图形化只有在针对一些非常小的模式语言时才可能有

用。因此，大多数模式语言不会试图提供这样一张试图描述其主题的整个解决方案空间的图。定义完整的语言比1000（或1024）幅图拥有更强的表达力。

虽然图并不适合用来描述模式语言的解决方案空间，但是这并不意味着模式语言的描述中不应该带有任何图示。事实上，模式语言中的很多地方可以从图示的解说和描述中获益。特别是下面几点。

- 如第1章中所谈到的，语言中的每种模式可以提供一幅图来描述其提出的解决方案。
- 如10.2节中所用到的图示那样，一幅或多幅图可以用于展示模式语言的拓扑结构。
- 如10.6节中所讨论的，一幅或多幅图可以图形化语言的语法。

但是所有这些图示都不是尝试将语言主题的整个解决空间图形化，而是分别关注了模式语言描述中特定的问题。

10.9 面向领域的命名帮助忆起模式语言

成功的模式语言不仅要支持创建可靠的设计和实现。模式语言作为日常工作中使用的语言，我们必须能够准确无异议地辨识它、引用它。如果我们无法简单直接地忆起一种模式语言，无论其自身的质量如何，该语言都无法成为设计工具集中的一件工具，并且很难在实践中得到准确的应用。

因此，同单个模式一样，每种模式语言都有名字。如果使用者通过名字引用某种模式语言，任何知道该语言的人应该能够很容易地回忆起它。单个模式中那些好记的名字，如Proxy、Broker和Reactor，应该能够唤起目标受众对它的印象并能表达其解决方案的精髓[MD97]。相比之下，模式语言通常是根据其覆盖的领域来命名的。

例如，*Server Component Patterns*[VSW02]或*A Pattern Language for Distributed Computing*[POSA4]这样的名字指明了相应的模式语言所应用的领域及其处理问题所在的问题空间。具有生动的名字但没有总结其目标领域的模式语言（如CHECKS模式语言[Cun95]）常常需要一个副标题来做澄清，比CHECKS语言的副标题表明CHECKS语言是“一种确保信息完整性的模式语言”。从这个角度来说，请求处理的模式语言（*A Pattern Language for Request Handling*）这个名字看起来是比较合适的。

乍一看，面向领域的命名是一种与单个模式的面向解决方案命名相反的命名方式。后者是根据模式提出解决方案的核心架构特征进行命名的。举例来说，企业应用架构模式语言的面向结构的名字是三层架构。但面向解决方案的命名方式并不能以一种帮助开发者为其系统选择适当的模式语言的方式来说明该语言的可应用性。例如，三层架构可以用于创建很多种类型的系统，但并不是所有的可以创建三层架构的模式语言在企业应用领域内都有用。

不论其实际质量如何，不提供面向领域命名的模式语言很难记住，从而很少被引用。这种“被遗忘”的模式语言的一个典型案例是Caterpillar's Fate，它描述了将分析转换到设计的过程[Ker95]。这个名字与模式语言的应用领域毫无关系，而是对其引入的过程的一种比喻。遗憾的是，只有很少的人真的记得Caterpillar's Fate，能够真正理解它的恐怕就更少了。无趣的模式名称可能不那么令人兴奋，但经验说明这样的命名更让人记忆深刻。

因此，面向领域的命名方式是让人们记住模式语言并使用它进行交流的最有效的方法，而这恰恰是它得到更普遍、更广泛应用的第一步。

10.10 模式语言展开对话并讲述很多故事

到目前为止，我们所探讨的模式语言的内在属性有一个有趣的结论。根据其本质，模式语言并不是简单地展示了一个特定主题的可持续解决方案的无限集合：模式语言还与其使用者展开对话，指导他们探索其解决方案空间。该对话概览了模式语言在其解决方案中处理的挑战；提示使用者在处理特定挑战时，可以替换使用潜在的设计和实现；还有，在存在这种替换方案时，对选择条件进行显式说明。

换句话说，模式语言不能作为预定义的参考解决方案被盲目复制、粘贴和改造，以便用于开发和重构的软件系统、重组的项目组织以及配置管理等，而是邀请其使用者参与到一个创造性设计过程中来。因此，模式语言帮助创建的解决方案来自于很多自发的、全面的决定和考虑，而不是一种专门的、完全机械化的软件开发方法。

而且，与单个模式或其他模式组织形式展开的对话相比，模式语言展开的对话从范围上看更广，从效果上更深。举例来说，单个模式展示并探讨了关于解决特定本地问题的知识。虽然这种简短的对话比不进行探讨好一些，但一旦超出模式讨论的应用和可用性范围时，其作用就很有限了。

模式互补通过回顾解决给定问题的替代方案对前面的对话进行了扩展。这种回顾比单个模式提供了更好的问题覆盖率，因为它考虑到了问题出现的具体上下文。模式语言对整个软件系统、系统部分或系统方面的设计，更进一步扩展了该讨论。因此，模式语言的阅读者可以更深刻地理解开发语言主题及语言主题本身的相关问题。

每一个由模式语言发起的具体对话都产生了特定的模式序列，该模式序列在应用时创建了特定的模式故事。这个模式故事继而说明了具体系统、系统部分或系统方面的设计是如何展开的。这个观察让我们直接得到这么一个结论：正是包含在模式语言里的所有模式序列构成的集合使得对话最终得以实现。

现在我们完善第9章末对于“模式语言可以是什么”的假设。我们提出模式故事可以帮助我们理解特定的案例，而模式序列可以帮助我们总结从模式故事中学习到的经验以及将这些知识再利用。我们进一步假设，把多个模式序列集成起来可以形成模式语言，可以在试图将模式序列应用到不完全吻合的上下文中时，把我们从这个一物多用的麻烦中解放出来。现在看起来这种假设是正确的一步，在本章中得到了进一步的证明。

10.11 路还很长

软件产品开发中非常有用的模式语言必须足够完整和成熟[HaKo04]。特别是，就问题和解决方案空间的覆盖率来说，模式语言必须完整；就其组成模式的质量和相互联系来说，语言又必须成熟。任何品尝过红酒、干邑或纯麦威士忌的人都知道要得到高的质量和成熟度是急不得的，需

要很多耐心和时间来酝酿。

模式语言同样服从下面的永恒规律：每种模式语言只可能记录过去和现在的应用经验，只可能完全覆盖其作者注意到的或有意关注的问题和解决方案空间。类似地，模式语言的质量和成熟度只能是由其组成模式（词汇）及其构成网络（语法）决定。正是长期积累的实际经验（加上实践之后的总结）才最终决定了一种模式语言是否适用，它有什么不足，以及通过哪些方式可以解决这些缺陷和漏洞。

模式语言随着不同的事件和见解而不断演化，比如通过某种系统开发中获得的新经验。类似地，相应模式下也可能会诞生新的技术，已解决了与特定类型系统相关的具体问题。例如，面向服务架构（SOA）带来了一个新的针对软件集成和松耦合的模式集合[MaLu95]，但也是基于很多著名的分布式计算模式的[POSA1][POSA2][VSW02]。将这些新的经验和模式集成到现有相关的模式语言中，使其与时俱进当然再好不过了。从最粗糙的到最为成熟的所有的模式语言，通过不断修订、增强、优化和完善，有时候甚至可以完全重写，都可以总是被认作不断演进的作品。请求处理的模式语言示例也是一个不断演进的作品。

一些应用必须确保每个子系统中、每个部分中甚至整个软件中只有一个Command Processor。Object Manager可以为特定的应用实例或部署提供一个可控的对象创建机制。Command Processor可以通过引用Object Manager进行扩充，从而获得生命周期控制的机制。类似地，Command Processor的并发访问可以通过将其实现为Monitor Object的方式，或是为其提供Strategized Locking来实现。将这两种模式都加入到请求处理的模式语言中，会使其更加完整和成熟。

与单个模式相比，将模式语言作为演进中的作品，甚至需要更多的“修剪”工作。创造一种高效的模式语言是一个渐进、增量的过程，因为它需要一段较长的时间来充分理解给定领域——它的范围、问题和解决方案。类似地，模式语言常常包含了很多可以也可用作单个模式的模式，这些模式自身也需要足够成熟，才能构成模式语言的基础。

拓扑结构是决定模式语言逐渐增加成熟度的另一个因素：理解单个模式怎样连接到特定模式语言中往往需要我们在构建软件方面有足够的经验。类似地，通过加入模式来扩充模式语言并不是简单地填空。模式及它们之间的关系并不是模块化的，因此，往往需要对模式语言及其模式进行紧密集成和改造。

文档化模式语言和文档化单个模式或模式编目所需花费工作的比率，与开发面向对象的应用框架和开发独立类或类库所需工作量的比率非常相似。将一份单为某个客户编写的软件总结抽象出一个满足多个客户（甚至整个市场需求）的可配置框架，这需要花费大量的时间。同样耗时的是：要将一种特定模式序列总结为一种包含并集成了很多有意义的模式序列的模式语言。举例来说，即使从很多现有的成熟模式及模式语言中提取资源，文档化POSA4中分布式计算的模式语言还是花去了我们将近6年的时间。类似地，*Organizational Patterns of Agile Software Development* [CoHa04]（出版于2005年）的第一版草稿在1995年就已经初次出现在Jim Coplien的论文“Generative Development-Process Pattern Language” [Cope95]中了。

高质量的模式语言的创作和成熟需要付出巨大的努力,这可能是被文档化的优秀模式语言数量不成比例地小于相应的优秀单个模式的数量的主要原因。但是,文档化和维护模式语言所需的努力可以从创作模式语言,与其他开发者共享模式语言,并不断对它们进行改良的满足感中得到回报。

10.12 模式语言对创造性智慧的回报

本章前面的讨论可能会给读者一种(错误的)印象,即一旦可用的模式语言是成熟的、完整的,它们就是一种开发高质量软件的绝对可靠的方式也就是说,无论其使用者的经验是否丰富,应用模式语言都可以得到“正确的结果”[Hearsay02]。如果这是真的,我们就解决了“软件危机”[Dij72]了!但在现实中,模式语言并不是“银弹”[Bro86][Bell06]。它们也不是“香肠制造机”,不能在应用者毫无半点技术知识或经验时也生成绝对可用的完美解决方案。

实际情况是:成熟的模式语言包含了软件开发专家的集体经验,如他们深刻理解了过去成功解决某种问题的解决方案以及这些解决方案是怎样高效实现的。还有一点,模式语言来源于人类的创造力,而不是那些死板的软件开发方法。虽然它常常间接地解决问题,甚至有时是反直觉的,但是模式语言的解决方案是比大多数使用传统的、广泛应用的官僚式方法(如瀑布模型[Boe82]和V模型[Drö00]过程)更优秀的方法。

例如,请求处理的模式语言示例提供了一些方法来处理创建请求处理组件的核心问题,乍一看可能有些奇怪。

传统上,客户端通过调用相应的方法来调用服务。但这种过程式调用增加了应用的结构复杂性:客户端直接绑定到它们使用的服务上,依赖其具体服务方法的签名。Command模式将请求封装为与特定服务操作的签名无关的对象,来打破这种紧耦合。但是,Command对象并不遵循传统的面向对象原则[Sch86],因为Command对象代表的是一种逻辑对象而非物理对象,它们主要实现行为而不是状态。

类似地,根据《设计模式》[GoF95]一书的建议,使用(Objects for) State(s)模式将模块化行为转移到对象内部被认为是一种优秀的设计和编程实践。但是,这种设计可能会污染请求处理组件,因为它会引入很多小型的类,仅为每种支持的Command实现了do和undo行为。一种更加优雅、高效的解决方案是在Command Processor中使用两个Collections for States外置Command的状态。一个集合保存了所有undo状态的Command对象,另一个集合则保存了所有redo状态的Command对象。

虽然成熟的模式语言中包含了专家经验,但是完全没有经验的模式语言的使用者是不可能创建出好的软件的——尽管有时候听起来是非常诱人的。如我们在10.2节中所讨论的,缺乏经验的软件开发者即使使用了模式语言,依然不太可能创建出有意义的产品。引用Louis Pasteur的话:“幸运眷顾有准备的人”——模式语言颂扬人类的智慧,但它们只奖励那些有足够能力、准备和创造力的人。

当然,无论其背景是什么,模式语言的学习可以帮助开发者更快地从新手进阶到大师。特别

是，学习和应用模式与模式语言可以帮助开发者避开那些传统情况下只能通过长时间的、高成本的实验和出错来发现的软件陷阱与缺陷。但我们希望强调的是，尽管模式语言有很多优点，但它并不能替代人的创造性智慧。

10.13 从模式网络到模式语言

支撑《面向模式的软件架构》愿景的模式语言绝不只是紧密交织的模式网络，其中的模式定义了系统化解决相互联系的一系列软件开发问题的过程。虽然这种精简定义本身并没有错，但是实际上定义了模式语言本质的是本章讨论的模式语言的内在质量及其所有组成模式的个体质量。如果模式网络没有展示出这些质量，它可能是一个模式网络，但不能被认为是一种有效的、完整的模式语言。

本章定义的模式语言与Christopher Alexander在*A Timeless Way of Building* [Ale79]和*A Pattern Language* [AIS77]中的定义有很多共同点，具体如下所示。

- 模式语言既是一种流程又是一种物件。
- 语言协助创建的物件必须是高质量的。
- 驱动力和上下文是重要的。
- 语言总是在进化之中。
- 模式采集实际经验并回报人类的创造性智慧。

同我们一样，Christopher Alexander也提到模式语言并不是万能药，也不是创建特定事物过程中的绝对真理。这些警告正是他把针对城市、建筑物及建筑工作的模式语言命名为一种模式语言的原因，并且强调了“一种” [AIS77]。

我们对模式语言的探讨也超越了Alexander的作品所包含的范围，包括模式语言所针对问题的可替代模式的集成、模式应用的强制和可选方面、拓扑结构和语法的标记法和角色、模式语言和模式序列之间的关系、命名和图表的角色。这些扩展是对Christopher Alexander的模式语言理解的补充而不是对立。

亿万种不同的实现

11

雪花从来不会有两片是重复的，人类的模式也是如此。我们的思想和行为不可思议地错综复杂，我们所面临的问题最令人费解，并且也经常在不知不觉中产生。

——Alice Childress (1887—1948)，人类学家

本章主要探讨模式语言的执行过程。到现在为止，我们已经接连叙述了一些单个模式和成组模式按照叙述顺序被应用的例子，本章我们将要开始了解模式语言是怎样被应用的。我们还会介绍与其他一些方法（例如重构、敏捷开发或产品线架构等）相比，模式语言如何帮助改善软件的可理解性、可持续性和重用性。

11.1 众口难调

在第2章中，我们向大家演示了每个单个模式都可以有很多种不同的实现。这更加坚定了我们的信念：对于一个模式来说，并不存在一个放之四海皆准的实现。这个信念同样可以用在模式语言上，原因很简单，它们集成了很多不同的模式，而每种模式都有百万级的实现方式。实际上，模式语言的解决方案空间比它所拥有的那些模式的所有解决方案空间的和还要大。如果一个单个模式有百万级的实现方式，那么一个模式语言的实现方法应当有十亿级的数量。这一节我们不去争论模式语言是否真的覆盖了整个解决方案空间，只去探讨为什么模式语言会有这么巨大数量的实现方式。

11.2 渐进式成长

模式语言能够支持这么广阔的解决方案空间，是因为它们经历了一个渐进式成长[Ale79]的过程。特别是通过使用模式序列，一个解决方案通过创造性的活动一步步得以形成，直到最后完成并且所有的部分都具有一致性。在渐进式成长的过程中，每个单独的动作都会分离已有的解决方案空间。从而，有着特定问题的设计或情形就这样逐渐变为另一个设计或情形，其中的问题已经通过使用正确的模式语言得以解决。

11.2.1 面向系统的、进化的设计方法

对渐进式成长过程的分析显示，这是一个面向系统、进化的过程。说它面向系统，是因为它始终将重点放在正在创建的对象的“整体”性上，然后这个“整体”一步步地被展开去开发它的组成部分。特别是，模式一旦被应用，就把它集成到已有的部分设计中，这样它就不会破坏整个设计的愿景和关键属性。相反，它解决了由这些愿景和关键属性引起的问题。这样一来，对模式的应用虽然造成结构上的变形，但是仍然保持^①了原有的愿景和属性。此外，这种步步展开的过程避免了架构上的偏移，因为所有的设计行为都在它们所处的更大的框架的制约下进行 [Cool02][Cool06]。

例如，用于分布式计算的模式语言[POSA4]从要开发的分布式软件的Domain Model的愿景开始。这个愿景制约并指导着此软件的后续开发步骤，这些步骤大到指定其基础架构，小到处理并发实现中的线程同步的细节等。与此类似，在 *Organizational Patterns of Agile Software Development*[CoHa04]中用来定义开发组织的模式语言在定义组织的细节（例如项目中特定角色的特征）之前，就首先描述了一个总的项目组织——Community of Trust。另外，还有一个例子是用于创建文档的模式语言[Rüp03]，在它处理格式和排版的细节之前，也要先确定整个文档的形状和结构。换句话说，在模式语言中，“整体总是先于其部分之前” [Ale79]。

渐进式成长的过程同时也是一个进化的过程，因为它在发展的时候支持基于自身需求的逐步自适应和优化。进化通过逐步加固来达成，一开始开发者“编织一个虽然大体上完整但摇摇欲坠的架构，然后开始逐步对其加固，直到最后整个架构完全坚固并且强壮” [Ale79]。

例如，模式语言用于软件设计的时候，建议先应用模式来为将被建构的软件勾勒出一个具体的、完整的架构愿景，即结构基准。这些“基准”模式并不指导架构愿景中的任何具体细节，它们仅仅建议其他的模式去展开它。具体要应用哪一个模式不会有预先提示，只有到需要处理特定细节的时候才会作决定。因此，逐步加固使得开发者能对关于细节和主题的最新需求作出反应。这个过程递归又迭代地持续着，直到软件所有的部分和方面都被相应的模式完整地定义和优化。

请求处理的模式语言示例同样支持渐进式成长的过程。

Command和Explicit Interface的结构可以用于定义请求处理框架的核心结构。如果需要高级的请求处理功能，可以使用Command Process or对其进行扩展。如果客户通过Message而不是方法调用来发起请求的话，可以用具有Interpreter的Command Process or进行扩展。其他一些模式帮助展开由这4个模式所创造的基准设计。用Strategy和Null Object，或者是用Template Method，可以为Command Processor的请求执行和辅助性功能提供变化。Memento可以支持Command的单次undo/redo操作，Collections for States允许Command Processor进行多次的undo/redo操作，Composite扩展了请求处理结构，使其支持宏Command。

^①乍一看，这个论述使用的“结构保持”一词[Ale79][Ale04a]好像和Christopher Alexander的理解相悖。关于这个词的理解只可取其要义，不可直接按字面意思理解，因为每个模式都会使结构发生改变，所以这里不能认为是完全保持了原有的结构。这个词的意思主要是，虽然每个模式都改变了给定架构中的一些元素，但是原有结构的“核心要素”没有改变。

所有由请求处理模式语言示例创建的设计都共享同一个结构核心：拥有Explicit Interface的Command结构。其他模式针对另外的需求渐进地扩展和优化这个核心。其他模式或者所选模式的其他实现有助于创建有多种目的和功能的请求处理框架。换句话说，普通的基线设计通过渐进式成长的过程逐渐被加固，直到它从众多可能的选择中定义出一个完整的、可以工作的请求处理框架。

与此类似，分布式计算的模式语言[POSA4]支持分布式软件系统的软件架构的逐渐加固。对于每个分布式系统背后的核心愿景的具体问题和细节，在设计过程中尽量推迟做决策的时间，也就是加固其架构。结果，有关应用的设计对于像使用哪种通信中间件平台或者是某个子系统或组件使用什么样的并发模型这样的问题的依赖性大大降低。

另外一个逐步加固的例子是在*Organizational Patterns of Agile Software Development* [CoHa04]里面描述的一个项目管理模式语言的模式故事。这个故事主要讲了在项目开始以后，由于一些没有预料到的但很紧急的问题的发生，如何将新的角色加入到整个项目中去。也就是说，在项目进行中出现了新的情况，需要对已经开始的项目管理实践进行调整，而项目管理模式语言提供了恰当的“响应”去应对这个新情况。

11.2.2 渐进式成长和敏捷开发

渐进式成长的过程明显有别于其他通用目的的软件开发过程。首先，它不是一个组合过程，比如和在图形界面工程里面将很多已经定义好的部分装配在一起成为一个整体这样的过程不同。同时，它也不是像瀑布模型[Boe82]和V模型[Drö00]那样的顺序过程，它们打算在系统开始实现之前就开发一个“总体规划”，这个总体规划包括了所有细节的详细设计。

这些通用目的的过程适用于它们特定的目标、特定的上下文以及特定的约束——但前提是彻底理解和掌握了这些目标或上下文的需求和约定。对于那些由许多未知而不确定的需求和约束来决定具体结构和设计的软件开发，这些过程的处理能力尤其不足。同时，它们很不适合开发那些设计决策互相依赖的软件。

而渐进式成长的过程拥有很多敏捷（软件）开发过程的特点[CoHa04]，比如极限编程[Beck04]、Scrum[SB01]或者是Crystal[Coc01]。举个例子说，在开发过程中将软件设计的最终决定推迟到最后时刻才作出，这就是*Lean Software Development*[PP03]的基本思想。而敏捷软件开发过程通过Stable Intermediate Forms[Hen04a]演进，同样也为了保证系统能够自始至终的进化，而不是通过功能不全又零散的步骤来完成。每个系统不同的需要和需求产生了不同的设计和实现。

有一点特别要强调的是，渐进式成长并不是专门只限于软件开发或者是应用模式语言的过程。它的实践方法可以用在任何创建和设计活动中，事实上它们自己就构成了一个模式语言[CoHa04]。

这也许令人惊讶，但请注意没有一个模式[属于渐进式成长模式语言的那些模式]只限于软件开发。只要是团队想要通过创建什么东西来解决问题，这些模式都是很有用的。它们对软件服务就像对构建产品一样有用，而对于硬件开发和软件开发同样如此。它们是关于人类天性和人类组织的模式，也是关于人类如何聚到一起解决问题的模式。

从上面的讨论可以得出，敏捷开发过程和模式语言具有渐进式成长过程的相同基础：运作方式都是迭代的、渐增的，是基于经验的总结和反馈来指导下一步行动的。它们和其他过程的主要区别在于所涉及的范围、方式以及细节。现在渐进式成长过程已经被嵌入到很多高质量的模式语言中去，这一点让我们确信它真的能帮助我们开发能够持久发展的东西，比如软件、开发组织[CoHa04]、开发过程的各个部分[Ker95] [Cun96]或者文档[Rüp03]。

11.3 并没有排斥重构

得知所有的模式语言在理论上都包括“正确的事”或许是令人欣慰的，但是这并不能保证开发人员在实践中应用它们时永远“做正确的事”[Hearsay02]。他们在写文档的时候或者是使用模式语言的时候，可能会忽略一些重要的因素。例如，模式语言中某些重要的变体或细节可能被忽视。或者在一个特定的应用中，某个设计决策可能看起来模棱两可，甚至自相矛盾。另外，情况在不断发生改变，例如变化可能关乎客户对各个模块要求的优先级、一些平台相关的内容、对所用到技术的理解和组织结构等，都有可能按着事先没有预期的或模式语言不能正确处理的方向变化。

不合适的问题解决方法可能来源于模式选择不恰当，或者是情况的变化导致模式已经不适合，还可能是因为不正确地实现模式。最好的情况是，整个系统实现中只有一小部分被影响。而最坏的情况则是，整个系统的基线架构都必须作改动。

在软件开发的过程中，如果当前的设计不是理想的设计，我们可以选择重构[Opd92] [FBBOR99]。重构通过改写程序的一部分来改善程序的结构或可读性，同时保证程序本身的逻辑含义不变。用在模式语言上的时候，重构主要是发现不合适的模式并且回溯，回到用错模式之前的那个模式，另外选择一个模式。这样的重构过程在模式语言内创建了一个有部分差异的路径，也就产生了一个不同的（通常是更恰当的）模式序列，根据这个模式序列可以对软件设计进行重构以满足软件需求。

下面描述了一个需要重构的例子。

在第9章中，我们描述了用模式序列<Command, Explicit Interface, Command Processor, Collections for States, Memento, Template Method, Composite>创建一个灵活的请求处理框架。在这个框架设计中，一个灵活点是由Template Method实现的。这个模式支持根据不同算法的不同策略来配置Command Processor，例如调度和日志记录。

类图11-1展示了这个解决方案。

如果所有从Command Processor类继承过来的策略类都不与其他策略类共享任何钩子方法的实现的话，此解决方案还是能够良好运转的。否则的话，就要为维护和改进作很多工作。例如，两个策略类可以在日志策略上不同，但是都支持同样的调度策略。当用Template Method应对这种情况时，要求两个策略类的调度策略的实现一致。如果调度策略发生了改变，那么这两个类都必须被更改，这样又麻烦又容易出错。因为首先所有需要作改动的类都要被标识出来，然后所有的

类都要用同样的方式进行改动。越多的类共享这个钩子方法的实现，情况就越复杂。

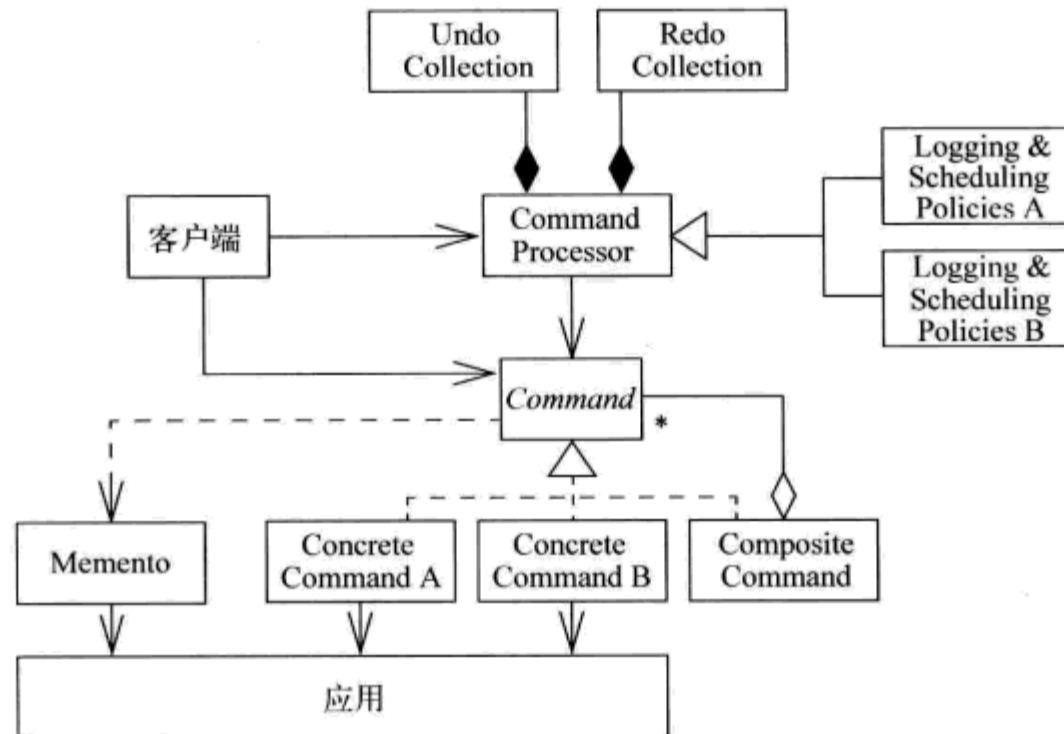


图 11-1

为了避免这种情况，需要对请求处理框架的设计进行重构，选择另一个能够实现可更改 Command Processor 行为的解决方案，而不是 Template Method。事实上，Command Processor 模式的描述已经给了我们两个不同的解决办法：Interceptor 和 Strategy[POSA4]。使用 Strategy 模式而不是 Template Method 模式得到的模式序列与刚开始的那个有些不同：<Command, Explicit Interface, Memento, Composite, Command Processor, Collections for States, Strategy, Null Object>。应用这个被重构过的模式序列为请求处理框架创建如图 11-2 所示的设计。

在这个设计中，日志和调度策略通过具体的 Strategy 类以可插入的方式实现，所以每个请求处理框架的实例都可以按照特定的策略进行配置。每个具体日志和调度策略都只被实现一次，从而避免了由 Template Method 引起的维护和进化上的问题。这样做的结果就是，重构过的设计符合请求处理框架的灵活性需求，并且优于最初的设计。

总结一下，模式语言产生的任何一条路径都是严格有序的模式序列。在这个路径上，多个模式被依次应用。然而这种有序并不像一个严格的瀑布式开发过程，因为瀑布式开发过程一旦确定了具体的设计或实现，就再也不会重新审视这个设计或实现了。在这种过程中，如果模式序列中的任何一个部分不适合情况要求了，都应该被更加适合的部分所代替：重构就是从一条路径（因而产生一种设计）切换到另外一种路径的机制[KerO4]。

重构不仅能解决像从模式语言中选择不恰当的模式或模式序列之类的问题。再来看看第 4 章里第一个试验中的那个问题。模式虽然选对了，但是在设计中它们没有被正确地集成成为一个整体。把那些带有“坏味道”的部分重构至“气味芬芳”，就是我们在第 4 章的第二个试验中所做的事。

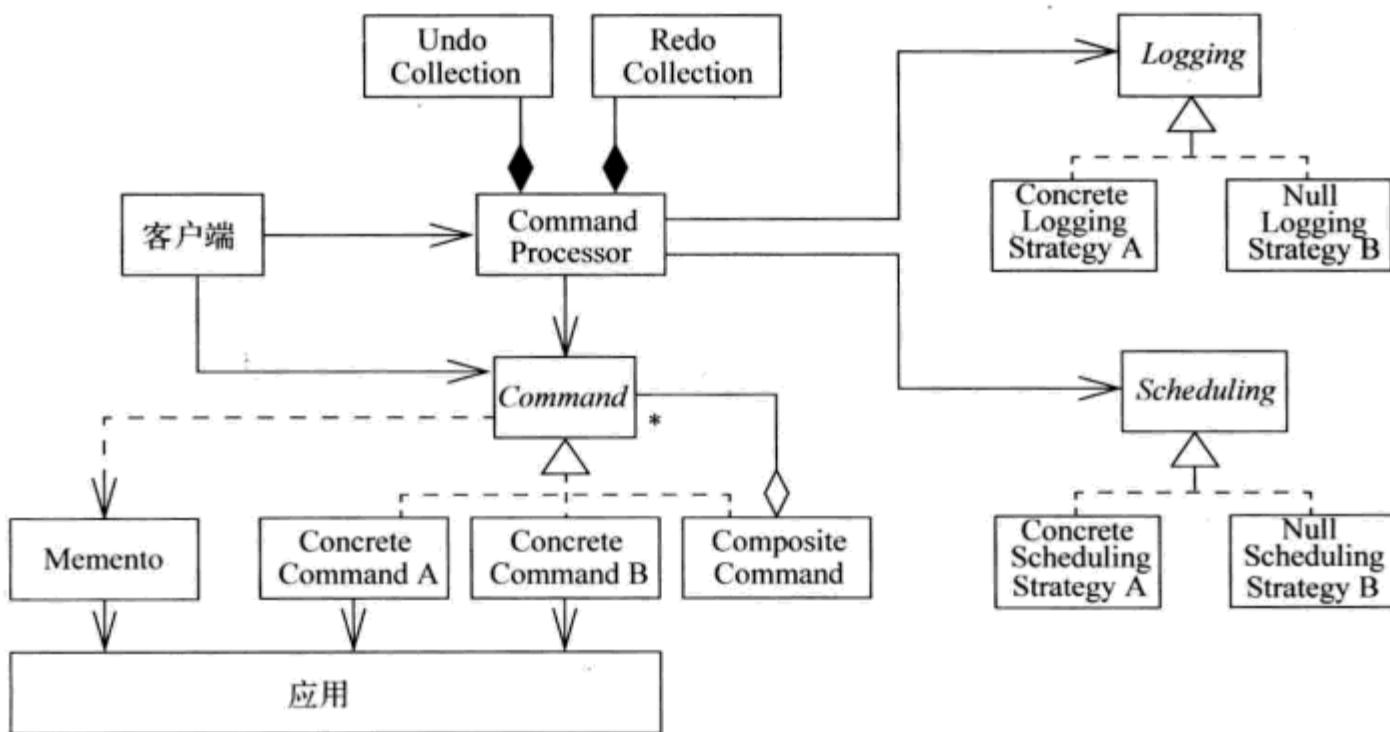


图 11-2

因此，重构是渐进式成长过程中不可缺少的一部分，同时也是在软件设计中创建和应用模式语言的重要一环。这个过程说明了人类从已有的经验和实时反馈中学习，人类会犯错，而且这些错往往是在细节处的。基于这三点，要想在一开始就保证模式序列选择和实现的正确性是不可能的。重构则既是对这种情况的安全保障，又是渐进式成长过程中变化的润滑剂。

11.4 一次一个模式

模式语言能支持一个很宽广的设计空间的另一个原因在于模式应用的技巧。最重要的是，模式语言被结构化以支持并且鼓励一次一个模式的应用方式，这种模式的应用同样是按照已经在模式语言中选择好的一个模式序列来顺序执行的。这个原则保证了渐进式成长过程的核心属性被保存：“整体”总是先于其“部分”。每个被应用的模式都在一个接一个地展开和构成独特的体系架构、设计、结构或者是模式序列中此模式之前的那些模式的已有实现。

11.4.1 明白手头上的问题的关键

一次只应用一个模式有一个合理的理论依据：在任意时刻，所有的焦点都应该集中在解决特定的问题上，这个问题由被使用的模式所处理，并且显然存在于实现语言对象实例的具体上下文中。这个问题应当在一系列既定驱动力下被恰到好处地解决。但是在同时应用多个模式的时候，要求我们把更多的注意力放在那些模式之间的组合和集成上，而不是放在解决当前的问题上。当待解决问题是新出现的、未知的，或者该问题的适当解决方案是待开发软件成功与否的关键的时候，这种注意力上的转移尤其不值得提倡。

一次只应用一个模式使得我们能够将重点放在要解决的问题上，即使我们已经知道其他模式也要被应用，也不会分散注意力。除非在前几次应用中已经解决了更大的问题，或者是发现一个

模式复合能够解决这个问题，我们才会放心地大踏步前进，一次应用多个模式。当我们发现步子迈得太大时，可以适当调小步子，确信一切尽在掌握中。

11.4.2 优先级驱动的设计决策

一次只应用一个模式的另一个理由是：模式语言鼓励用户明确哪个问题比其他问题更重要。每个由模式所创建的设计、结构或是实现都会限制和约束其之后被应用模式的解决方案空间。一次应用多个模式会使用户感到困惑和犹豫，不知道接下来要处理的问题哪个最重要。若几个问题之间相互关联，并且它们的解决方案有重叠之处，也应该先解决最重要的问题，而且这个问题的解决方案也应该成为其他几个问题解决方案的指导。否则，会越来越倾向于同时解决太多问题，从而失去了对全局的把握。

即使在需要多个其他模式来帮助解决当前模式的问题的时候，也应当遵循一次应用一个模式的规则。为了一举击破许多问题而将这些“小”模式和“大”模式一起应用，这可能看起来很诱人。然而，在我们真正理解“大”模式是怎样与正使用的设计结合之前，很难确定应用全部、一些或者不应用“小”模式这件事是否恰当。与此类似，如果原本的模式意味着其他模式要去解决一个子问题，在弄清是什么具体的驱动力引起这种选择前，很难说要选择哪个模式。

当我们没有完全理解设计时，可能会选择一个不恰当的方法来解决面前的问题，从而处理得不尽人意。只有当我们了解模式在整个设计中所扮演的角色时，才能决定哪些“小”模式可以被用来优化主模式或者是哪些模式可以用来替代原来的模式。再次强调，一次应用一个模式能够避免到头来看上去什么都会却一样也没有真正掌握。

11.4.3 模式集成先于模式实现

然而，一次应用一个模式的规则并不强制要求完全实现一个模式后，才能应用下一个模式。很多情况下，如此理解这个规则甚至会毫无益处。

例如，首先将一组模式和一个（部分上）完整、有用并自成一体的设计或结构集成起来，然后在其稳定后实现这个设计或机构，这样做经常更合理、更高效。换言之，一个作品的设计（例如软件组件的设计或部分软件开发组织的设计）是通过一次应用一个模式创建的。相比之下，这个作品的实现往往通过一步即可完成。无论何时，当我们把新模式集成到（部分）解决方案里去的时候，这样的方式使得修改或重构模式实现的需要降到最小。它同样允许我们在开始实现设计之前从整体方面对设计进行仔细斟酌。

另外一个不必在一个模式结束后才使用下一个模式的例子是参考多个其他模式的模式实现，其中一些是模式必需的，而一些是可选的。为了更好地明确该应用哪些可选模式，可以先集成并实现那些必须要应用的模式，这样就让那个原来的“大”模式处于未完成状态，直到它的核心部分变得相当坚实。然而，这个过程仍然蕴含着一次应用一个模式的规则。在实现原本的“大”模式的时候，最初我们使用深度优先的方法去实现那些定义了“大”模式核心部分的模式，然后再用广度优先的方法依次实现那些可选的模式，从而完成了整个“大”模式的实现。

11.5 基于角色的模式集成

我们在11.2节中讲过，应用模式语言的模式把发生问题的结构转换为解决了问题的结构。这种解决方案应当保持原结构的愿景和核心属性，同时又能处理当前问题的各种因素。为了顺利完成这种转换，模式语言里的模式不能够被看做是被整个插入到设计中的独立模块。因为这样就会产生一个没有清晰愿景的、内部不连贯也不一致的蹩脚的拼凑物——不管这些东西是软件、组织、项目管理实践或者是其他什么东西。相反，模式语言相关的开发过程应该保证，基于各自的角色与系统的整体结构和设计保持协调统一。

11.5.1 选择1：识别并且保持已经实现的角色

当应用模式语言中的一个模式时，我们建议首先确定现有结构或设计中的元素是否已经提供了那些由该模式引入的角色。倘若如此并且模式并没有规定要单独实现这些角色，那么就让这些已有的元素保持它们的角色并且用模式所定义的那些角色缺失部分来扩展它们。

下面描述了在请求处理的模式语言示例中遇到的这种情况。

Command Processor模式引入了Command Processor角色，该角色负责为客户执行Command对象，例如通过使用调度算法的方式。

如果这个算法会随着具体的Command Processor的实例不同而改变的话，可以应用Strategy模式来实现这种变化。Strategy引入了3种角色：上下文、策略接口和具体策略。

然而对Strategy来说，Command Processor已经扮演了上下文的角色。所以应用Strategy模式就包括重构现有的Command Processor的设计，以便它能够使用不同调度算法的各种具体策略进行配置。

图11-3展示了这种情况。

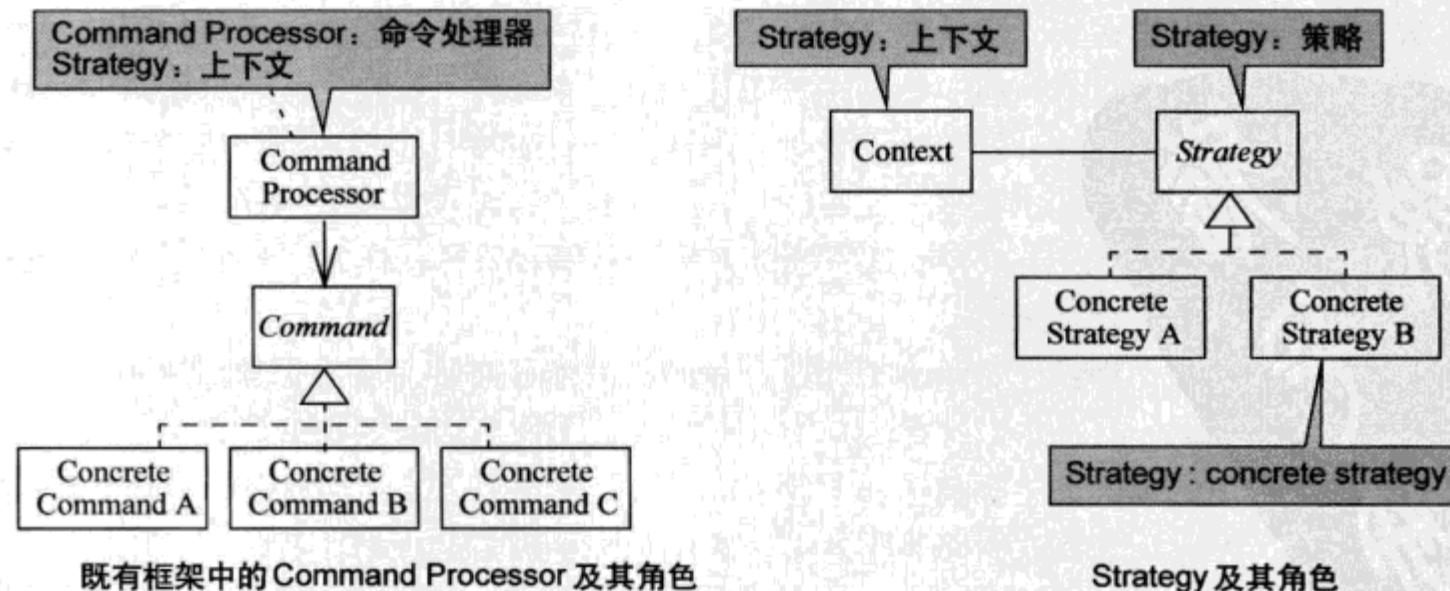
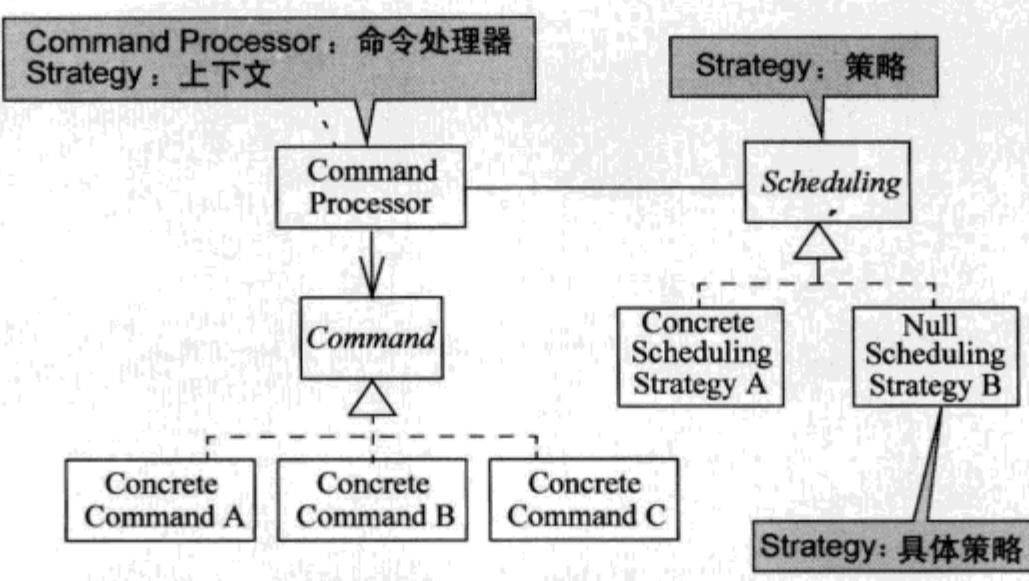


图 11-3



改进框架中Strategy与Command Processor的集成

图 11-3 (续)

11.5.2 选择 2: 识别并分离已经实现的角色

另一方面,如果已有解决方案中的一个元素已经实现一个或多个由模式引入的角色,但是当模式指定要单独实现这些角色的话,请依照下面的这种方式进行重构。

Command对象能够提供简单的undo/redo机制,这要求它们在执行请求之前先捕获有关应用状态的快照,在undo方法被调用的时候恢复当时的状态。但是如果直接在Command对象里实现undo/redo数据结构的话,会造成它们与正执行的应用之间的耦合过紧。特别是当应用的数据结构发生变化的时候,Command对象的数据结构也要做相应的改变。于是,Command模式建议将其功能分成两部分:一部分实现请求执行功能,同时分出一个Memento来捕捉和维护应用特定的数据。

图11-4描述了这种情形。

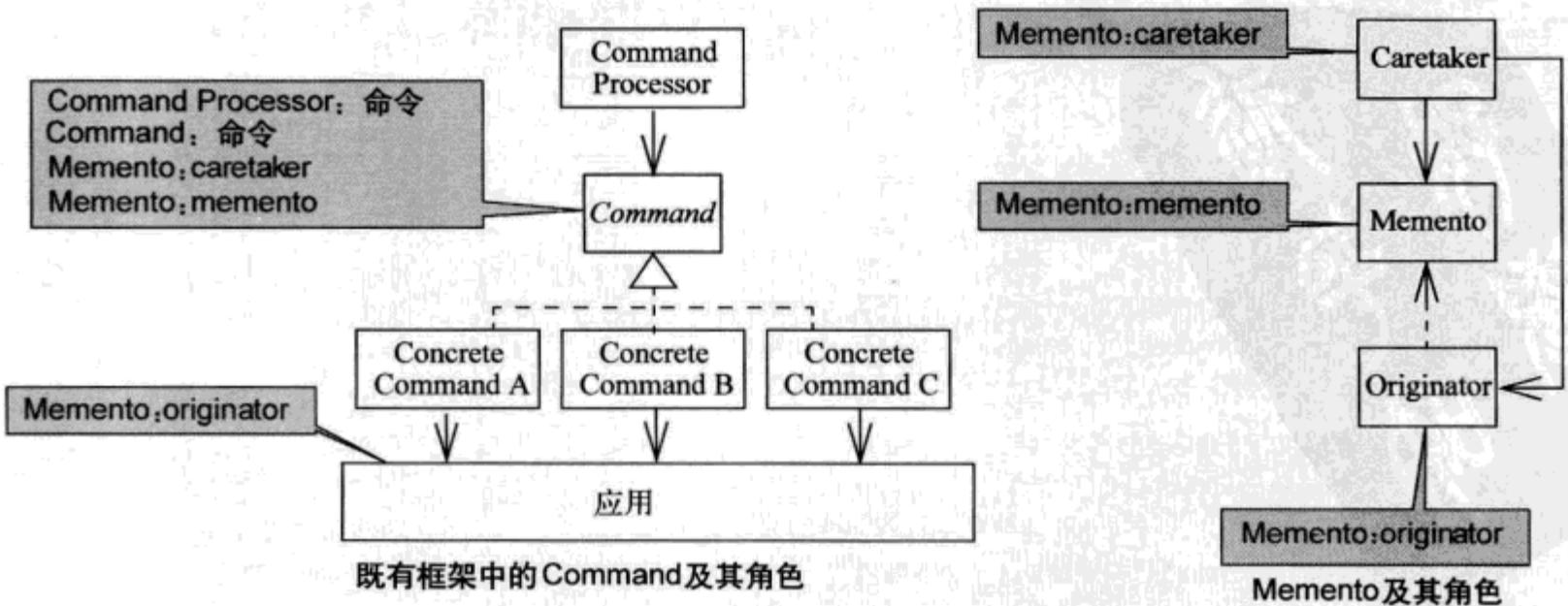


图 11-4

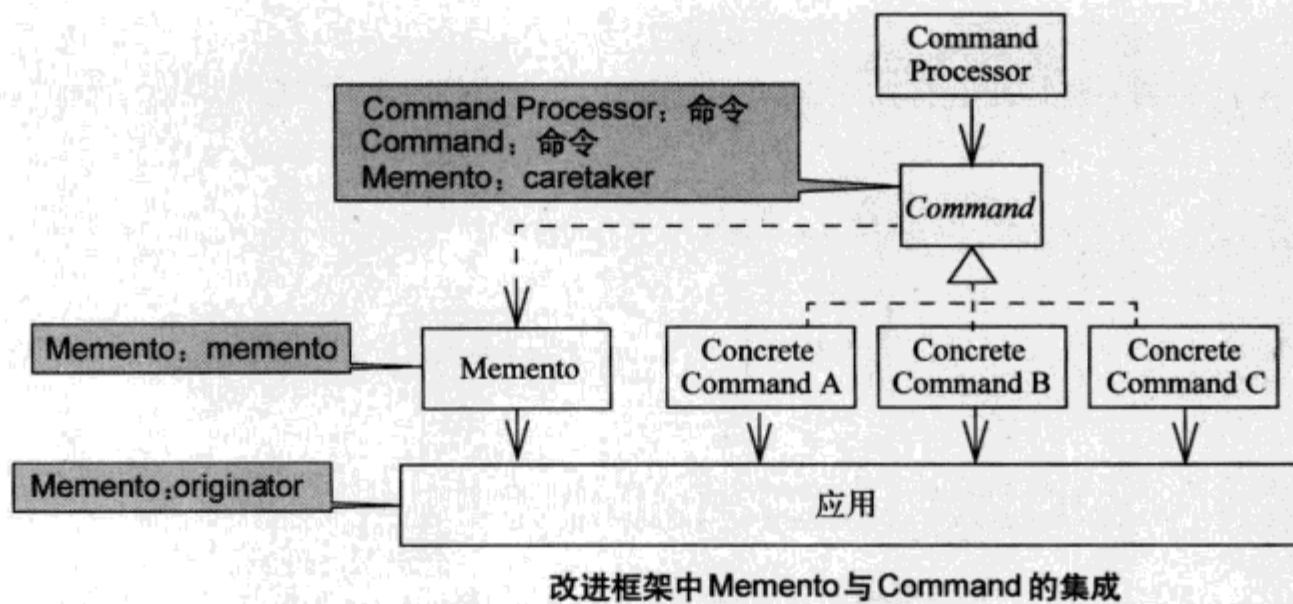


图 11-4 (续)

11.5.3 选择 3：将缺失角色分配给既有设计元素

如果这个模式引入了在已有解决方案中还没有实现的角色，哪个既有成员的角色对它构成补充，就把该角色分配给它。

为了支持宏Command，Command序列可以被当做Composite结构来实现。Composite引入了3种角色：组件、组合和叶子。从Composite的角度来看，Command对象属于叶子，而它们共享的Explicit Interface则是一个组件。而Composite这个角色没有在Command的设计中体现出来。因此，为了给出一个有意义的宏Command的结构，一个具体的Command对象需要被扩展来承担由Composite模式指定的组合角色。

图11-5描述了这种情况。

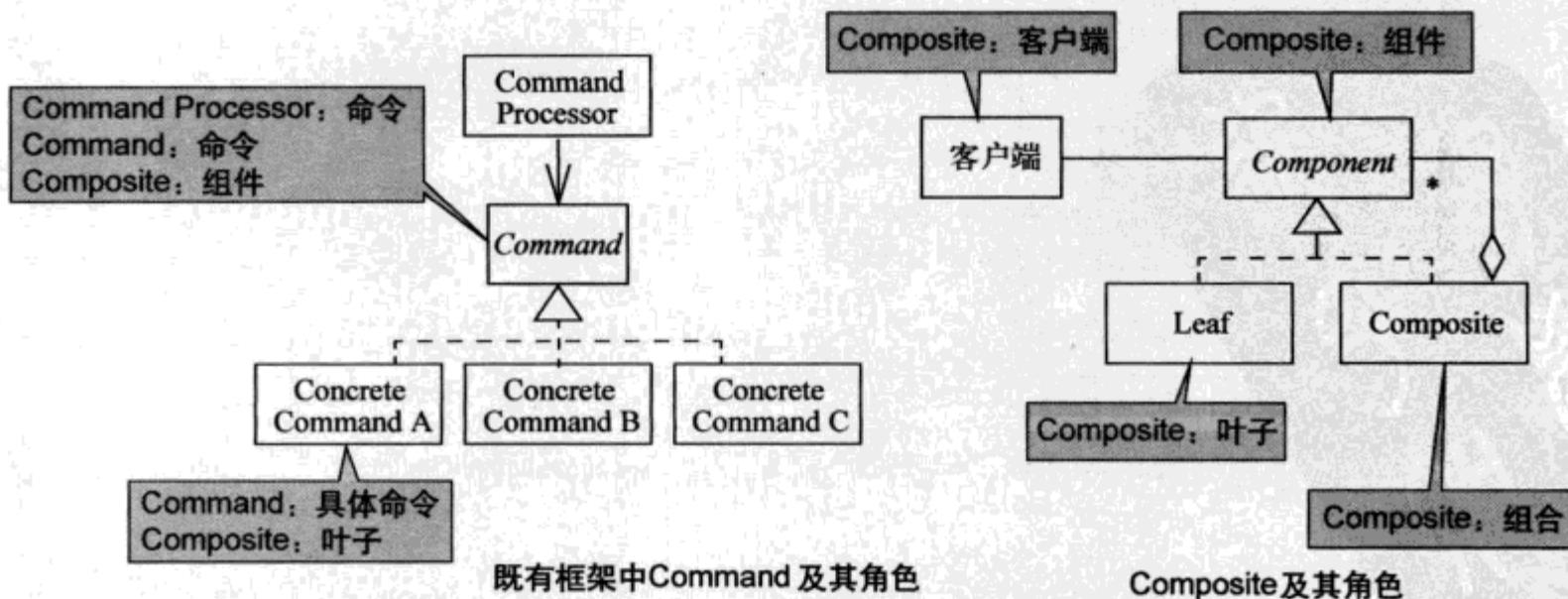


图 11-5

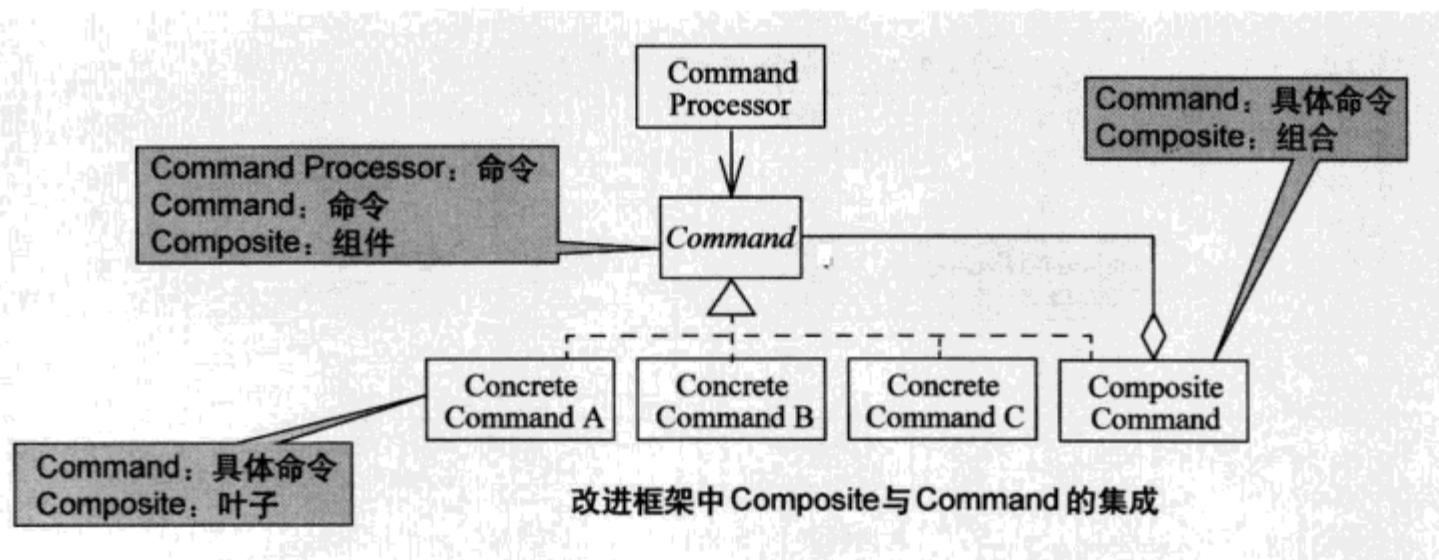


图 11-5 (续)

11.5.4 选择 4: 将缺失角色作为新设计元素来实现

最后, 根据模式自身的描述, 那些必须单独实现的模式角色, 则由相应的新元素来承担, 并集成到既有解决方案中。

Command Processor这个角色由Command Processor模式所引入, 它是一个为客户执行Command命令的中心实体, 提供了高级的请求处理功能, 如undo/redo、调度和命令历史记录。当把Command Processor模式集成到已有的Command模式序列上时, Command Processor角色必须单独作为一个实体来实现, 不然它就不能有效地实现自己的功能。这个设计如图11-6所示。

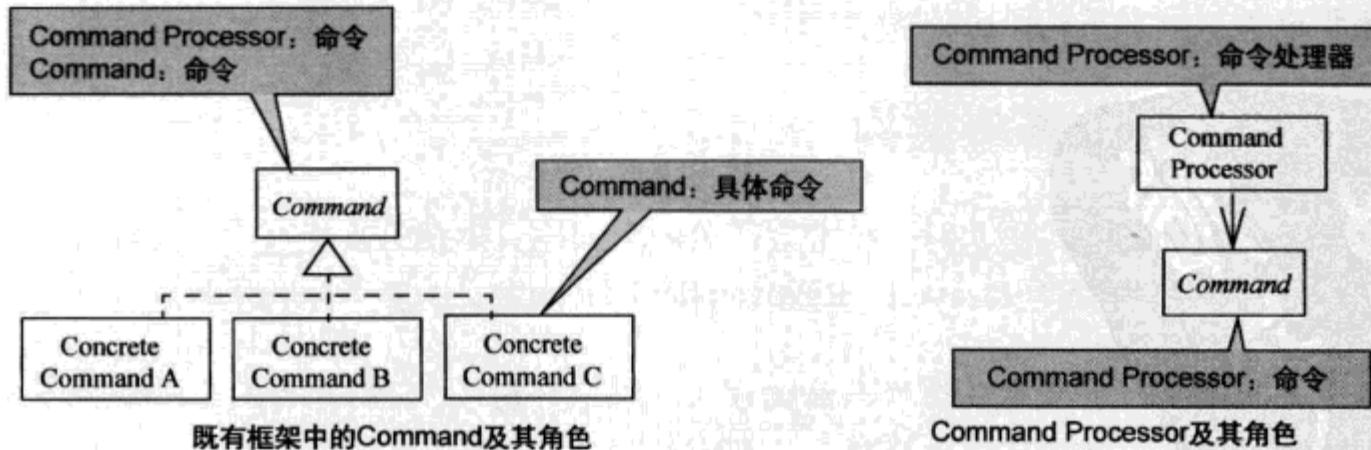


图 11-6

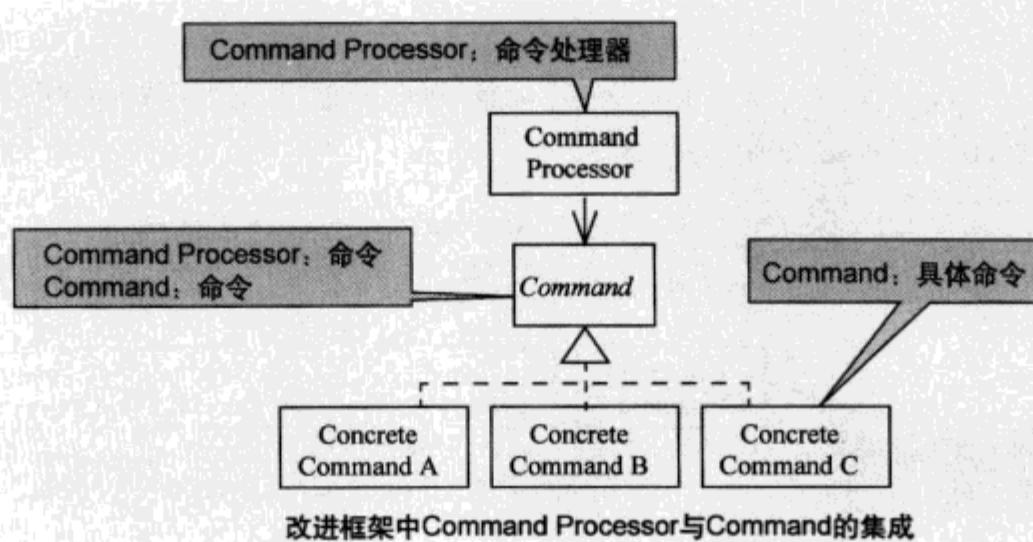


图 11-6 (续)

11.5.5 基于角色的模式集成和渐进式成长过程

遵循基于角色的模式集成的原则，在做一个具体的设计或解决方案时有助于成功地利用渐进式成长过程来实现基于模式的（软件）开发。同时也能够根据更大的设计、实现、结构或是它们内部元素的配置来量身定做模式的实现方式。这样的实现能够处理那些尚未得到处理的驱动力，也能完成或扩展软件以满足所缺失的功能，同时还能够加强和平衡软件架构的愿景和属性 [HaKo04]。

回顾我们在第4章的实验中创建的两个设计例子。只有当请求处理的模式语言示例支持早些讨论的基于角色的集成时，我们才可能将第一个模块化但不平衡的请求处理框架转换成第二个集成且平衡的框架。同样，因为*Organizational Patterns of Agile Software Development*中用于描述项目管理的模式语言将其组成模式基于角色进行紧密集成，所以描述该模式语言的模式故事中的项目管理组织能被扩展和重构[CoHa04]。

11.6 模式语言和参考架构

在学习软件设计的模式语言时，经常会产生一个问题：模式语言能否为其主题“产生”或者“描述”参考架构呢？下面是Jazayeri、Ra和van der Linden定义的参考架构的概念[JRv00]。

参考架构是一组有关系统结构和组成方式的概念和模式，它们允许具有一致参考架构的系统可以交互通作，并且可以被同样的工具和过程管理。

举个例子，分布式计算的模式语言[POSA4]可以被用来描述OMG参考架构中的关键设计，这个关键设计是针对CORBA的ORB（Object Request Broker，对象请求代理）的。与此类似，它也可以描述其他中间件的架构，如.NET和J2EE[Stal03]。它还能够创建给定参考架构的具体实例，例如设计TAO CORBA-compliant ORB[SC99][POSA4]。

是否使用模式语言来创建参考架构或具体项目的架构取决于具体的需求和开发者的意图。例如，在一个特定组织中和特定情况下，架构师可能倾向于创建一个参考架构来作为所有属于这种类型或这种领域的具体系统的基础。使用合适的模式语言对创建参考架构会有帮助，同时也可能受其他开发约束和考量的影响。其他组织可以应用同样的模式语言来为系统实现不同的参考架构。反过来，如果架构师想要得到参考架构的具体实例的话，使用模式语言同样可以有帮助。最后，如果架构师想要专门为特定的项目定做架构的话，模式语言的应用依然有效。

模式语言并不总是为它们中的每个领域都提供参考架构的原因是模式语言总是覆盖了一个解决方案空间。相比之下，特定的参考架构可能仅仅覆盖解决方案空间一个特定的角落。当然，覆盖一个解决方案空间常常意味着覆盖多个角落，从而就有可能创建多个参考架构。与标准类似，所以通过模式语言创建的架构的好处就是总有很多选择！

总而言之，模式语言并不是详细描绘具体的（参考）架构的设计蓝图。它们提供了通用的过程，可以创造出十亿种软件架构，其中每种软件架构可以被调整，以便符合其所服务系统的需求。

11.7 模式语言与产品线架构

软件工程领域的许多研究者和实践者都注意过模式语言对创建产品线架构很有帮助[Bosch00][Coc97][Bus03a][Bus03b][POSA4]。Bosch定义的产品线架构如下[Bosch00]。

产品线架构是软件架构，这个软件架构是整个相关软件产品族的通用基准。

总地来说，产品线架构由刻画了产品族中所有成员的共性的稳定基线和一组集成到该基线中的可调节、可配置的设计组成[Bus03b]。产品族中特定成员的产品线架构的实例可以通过为每个可变的设计选择具体的配置和适配选项来实现。

分布式计算的模式语言[POSA4]就是一个用模式语言创建产品线架构的例子。例如，它为用于仓库管理的流程控制系统的产品线架构提供了概念性基础，这个产品线架构支持不同类型、大小和组织形式的仓库[Bus03b]。模式语言同样也指导着TAO CORBA-compliant ORB的产品线架构，通过配置，它可以支持很多分布式实时和嵌入式系统的开发和服务质量（Quality-of-Service）的场景，这其中就至少包括仓库管理流程控制系统。

图11-7以及POSA4中的相关讨论概览了TAO服务器端的ORB Core的基准架构。

TAO的架构的一个关键设计目标就是处理通信中间件的所有必要问题，从而以一种一致的、健壮的方式支持分布式实时、嵌入式系统，包括并发、同步、传输、容错、请求和事件分离和（解）封送等，同时要保证ORB是可配置的、可扩展的、可修改的和可移植的。从而，在支持严格的操作质量和支持大范围客户和应用需求之间，TAO的架构做到了平衡。平衡这些需求在定义产品线架构时是老生常谈的话题[Bus03a]。

设计TAO服务器端的ORB Core所用的模式序列如表11-1所示。

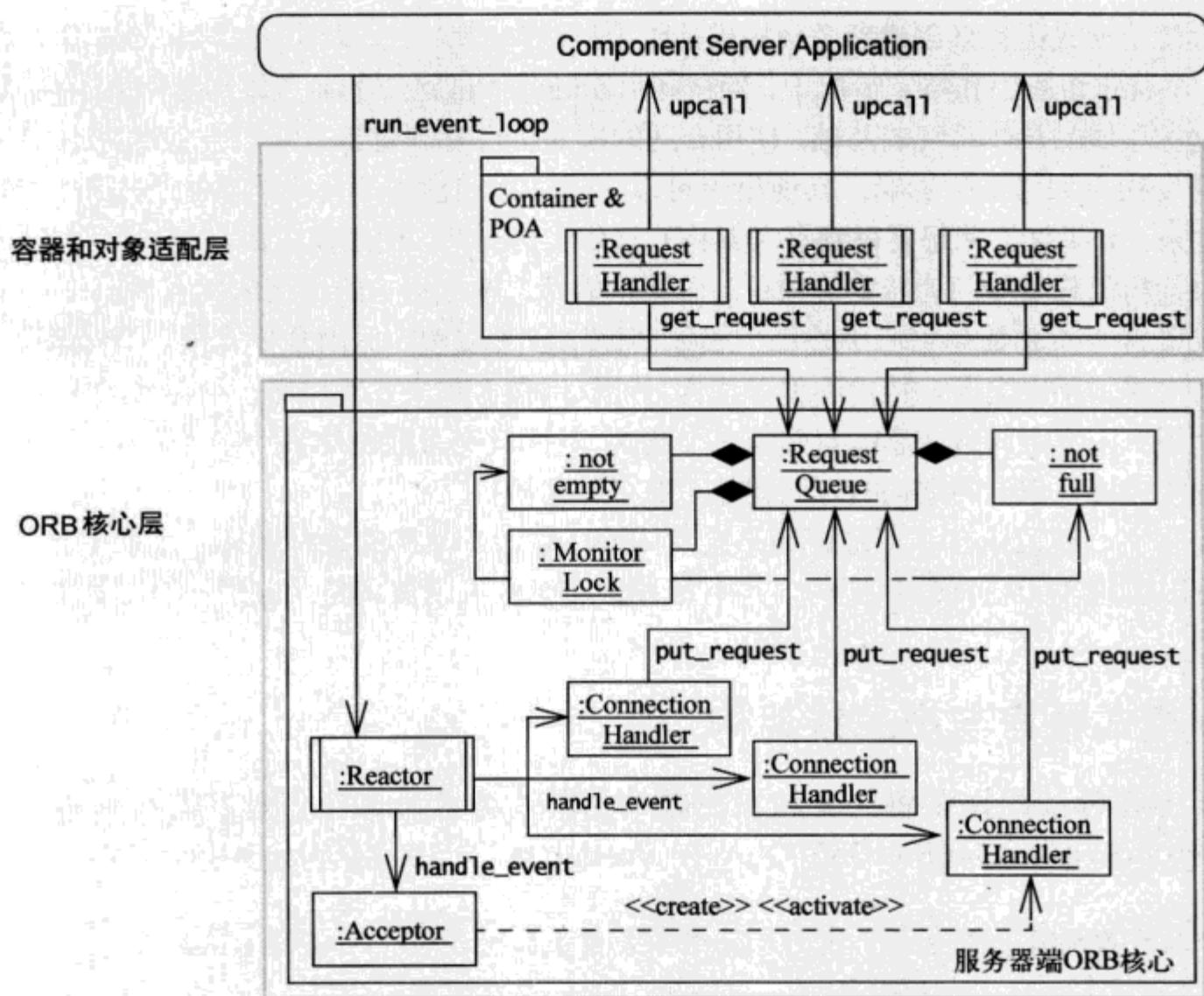


图 11-7

表 11-1

模 式	挑 战
Broker	定义ORB的基线架构
Layers	结构化ORB内部设计以支持重用，并对关注点进行分离
Wrapper Facade	封装低层系统功能，提高可移植性
Reactor	高效地分离ORB核心事件
Acceptor Connector	高效地管理ORB连接
Half Sync/Half Async	通过并发地处理请求，提高ORB的可扩展性
Monitor Object	高效地同步Half Sync/Half Async请求队列
Strategy	透明地替换内部ORB机制
Abstract Factory	将ORB机制策略合并成语义兼容的策略组
Component Configurator	动态配置ORB策略

上面这个模式序列中的前两个模式Broker和Layers，定义了CORBA-based ORB的核心结构。Broker模式将应用功能从通信中间件功能中分离出来，而Layers根据抽象层次的不同，将通信中间件服务分为不同层次。第三个模式Wrapper Facade将整个ORB设计的最底层（操

作系统抽象层)模块化，并且成为可复用的独立构建块。

第三个模式到第七个模式主要负责服务器端的ORB的核心层。Reactor模式提供了一个分离、分派的基础设施，它与底层的分离机制(如WaitForMultipleObjects和select)无关，并且可以方便地扩展以处理不同的事件处理策略。Acceptor Connector模式建立在Reactor模式之上，通过引入专门的事件处理程序来初始化和接受网络连接事件，从而将网络连接的建立从TAO的ORB核心里的通信模块中分离出来。Half Sync/Half Async以及Monitor Object扩充了Reactor模式，这样客户请求可以被同时处理，从而改善了TAO的服务器端ORB的扩展性。

最后3个模式主要负责在通用的设计框架中进行配置，这是产品线架构中的核心要素。Strategy被用在任何一个可能发生变化的地方，例如可变的连接管理、传输、并发、同步和事件/请求的分离机制。为了用一组特定的语义兼容的策略去配置ORB，客户端和服务器端的ORB需要用Abstract Factory模式来实现。这两个模式一起应用可以创建不同的ORB来满足不同用户和应用场景的需要。Component Configurator可以调整TAO ORB中的Strategy和Factory的有关配置，而不需要修改已有的代码，也不需要重编译或重链接已有的代码或者结束和重启已有的ORB及其应用组件的实例。

尽管TAO ORB明确设计成面向模式的产品线架构，但实际应用模式语言的结果并不总是一个产品线架构。实际上，绝大多数有意无意地、明确地或追溯地基于模式语言的架构或架构性功能都不是产品线架构。诚然，模式语言帮助开发者创建架构，但是不存在会生成产品线架构的固有需求：因为一些架构是经典的面向对象框架，而另外一些仅仅是一次性的系统设计而已。

如果对TAO进行重新设计，限制支持灵活性的那些模式的应用(如Strategy、Abstract Factory和Component Configurator模式)，虽然其结果仍然是通信中间件的设计，但是不是产品线架构了。一些领域(例如嵌入式系统)通常只需要很狭窄范围内的可变性和可配置性。对这些领域来说，限定TAO的灵活性是正确的并且与模式语言相符合。当然，TAO的创建不同ORB配置的能力也会被它所支持的领域、这些领域里特定的行为，以及它自身的支持多样化操作需求的能力所限制^①。

模式语言和产品线架构不同的原因与模式语言不能等于或指定参考架构的原因类似。对于特定领域的系统来说，模式语言覆盖了一个解决方案空间，并且提供了生成该解决方案空间中特定解决方案的过程。但是，模式语言既没有给出一个专门的解决方案(例如自定义的架构)，也没有给出这个解决方案空间的特定子集(比如产品线架构)。

根据不同的软件系统或是软件开发项目的需求和驱动力，应用模式语言可以创建出一个产品线架构或者是基于具体应用的架构。在不同的需求和驱动力下，模式语言也可以生成不止一个产

^① 特定上下文中间件自动专门化(automated context-specific middleware specialization)工具的发展使得支持能够离线定制其可变性的产品线架构成为可能[KGSHR06]。这种方法可以帮助解决基于标准的中间件平台和基于特定服务的不同产品之间的不协调，前者主要为了满足广泛的应用需求而产生的一般性，而后者是高度优化的、专门定制的中间件的实现。为了解决这两者之间的矛盾，可以把模式语言和自动化工具结合起来，这样就可以保持标准中间件的可移植和互操作能力。

品线架构，并且每个产品线架构都不一样。类似地，模式语言也可以创建出不同的基于具体应用的架构，而且每个架构非常适合要扮演的角色。

不过，模式语言仍然是创建稳定的产品线架构的有力工具，因为它们的模式网可以捕获到给定领域里的共性、可变性和设计的权衡，如同我们在第10章所讨论的一样。通过前面举的例子可以看出，用模式语言还是可以帮助创建出很多成功的产品线架构的。因此，模式语言是产品线架构的关键，这两个概念并不一致，但是相辅相成。

11.8 从十亿个到一个……再到一些

本章描述了每个模式语言里的渐进式成长过程是如何在某个特定领域一步一步地、面向系统并不断演化地开发出这个领域的具体设计和解决方案的。我们同时也看到了模式语言通过不同途径为这些设计和解决方案打开一个实现的空间，而这个实现空间由于其巨大的模式的排列方式而丰富了解决方案空间。

因此，模式语言并不是一个开箱即用的或者配置一些参数就可以使用的蓝图或者参考架构。相反，它们鼓励使用者去思考将要创建的设计和解决方案，并且采取明确的、合理的、经过深思熟虑的步骤一步步地去完成设计。模式语言的目标是为当前情形支持一个正确的、可持续的解决方案的开发，而并不是简简单单的一个解决方案。让我们引用Dick Gabriel在这本书的序里的话来强调这种观点。

设计是构建之前所做的思考。

尽管我们赞同这种观点，但是对于软件系统来说，不是所有的设计工作都从零开始，也不是所有的设计工作都是新颖的、与众不同的，或者表现得多么有创意和生产力。如同有很多方法去组织一个项目或系统代码一样，也有很多可用并且合理的方法去设计一个软件系统或组织结构。但是在实践中，一些解决方案比其他的解决方案出现的频率更多，因为它们能满足很多不同情形和用户的需要。

既然观察到了这一点，把这些常见的解决方案描述为可重用的参考架构和产品线架构是很自然的。因此，模式语言支持创建参考架构和产品线架构，这对一个系统来说，就使无数可能的解决方案缩减到了具有代表性的几个。正所谓“三思而后行”，模式语言鼓励有经验的设计者和开发者去思考特定领域中众多解决方案的共性和不同点，并且在设计参考架构和产品线架构时明确体现他们的洞察力和从分析中得出的经验和教训。

如果这些压缩封装的解决方案仍然不能满足你或者特定的情况和上下文的要求，模式语言还能帮助你构建自定义的解决方案。在这种情形下，保持对系统和具体需求的持续关注，实际上是渐进式成长过程在系统地指导你在广袤的解决方案空间里找到最适合你的那一个。

模式语言的格式

12

在处理音符上，我和别人没有什么两样。我擅长的是处理音符之间的停顿——嗯，这正是艺术所在。

—— Artur Schnabel

记录模式语言并不是简单地将每个独立的模式记录下来放在一起。为了能够有助于向读者传递模式语言的愿景和细节，我们还需要考虑其他方面。本章将介绍整个模式语言的格式来拓展我们对格式的讨论。

12.1 风格与本质

我们在第3章中讨论了表现形式对于模式来说绝对不是可有可无的附属物，它对于实现模式沟通的功能至关重要。模式的具体编写格式就是我们沟通的工具。既然模式语言是由紧密集成在一起的模式构成的，那么不言而喻，格式对于模式语言来说也一样重要。因此可以说，恰当的格式对于沟通模式语言更为重要。总地来说，我们对于单个模式的格式和模式复合的格式的讨论对模式语言同样成立。但是，对于模式语言中格式的元素和细节来说，还有一些额外要考虑的东西，具体如下所示。

12.2 格式的作用

对于模式和模式语言来说，格式的作用差不多：作者的审美和风格喜好必须与语言要表达的技术信息以及目标受众的期望达到某种平衡。由于这三个参数中的任何一个都可能发生变化，所以显然不存在一个格式是适合所有作者、所有模式语言和所有受众的。变化归变化，模式语言的格式还是有一些相同的东西，那就是反映了模式语言的通用意图[BeCu87]。

模式语言通过为设计过程中可能出现的已知问题提供可行的解决方案，来为设计人员提供指导。这可以看做是一系列知识，它们以某种风格和顺序记录下来，目的是指导设计人员在正确的时刻问出（和回答）正确的问题。Alexander以模式的形式将这些知识记录下来，并且共享统一的结构。每个知识点都包括对问题的描述、对出现该问题的环境的介绍，最重要

的是还包括在这些环境中下可行的解决方案。模式语言以一种完整的结构记录下这些模式，可能是关于建筑的，也可能是关于交互式计算机程序的。在模式语言中，模式与模式之间相互关联、彼此影响。我们在记录模式的时候，通常会在开头或者结尾介绍这些关联关系。Alexander已经为我们展示了，对于具有相当规模的模式语言来说，不需要引入循环影响也能将其组织得很好，这就是说，在设计推进过程中不需要保留之前的设计决定[AIS77]。

有关格式的问题

根据前面所说的模式语言的目的和结构，模式语言的作者必须要回答几个问题。最终，模式语言的作者必须保证其技术内容在广度和深度方面是合适的，并且其格式也应该是能够为其目标受众所接受的。

我们在3.2节中总结了与构成模式语言的单个模式相关的问题。当我们把它们放到模式语言这个背景下时，必然有新的问题出现。这些问题的目标是指导用户从模式语言中选择合适的模式序列来完成自己的设计（或者对设计进行重构），同时引导设计人员“在正确的时间提出（并回答）正确的问题”。

例如，在整个模式语言层次上，我们可以问下面的问题以帮助了解它的目的。

- 这门语言所针对的是哪个领域？
- 这个领域中关键的驱动力有哪些？
- 这个领域中有哪些不同的问题域是相互关联的？

在如何连接语言中的模式这个层次上，我们可以提出更加详细的问题，以指导具体的设计，举例如下。

- 该语言从何处开始？
- 该语言中有哪些路径？
- 这些路径在何处分支，何时选择何种分支？

最后，在组成语言的模式这个层次上，我们可以提问每个模式在语言中的位置，举例如下。

- 对于语言中的某个模式来说，有哪些模式可以辅助其实现？
- 在实现其他模式的时候，有没有或者能不能使用该模式？
- 在语言中某个具体模式的实现过程中，应该在何时决定是否使用其他模式？

有效的模式语言的作者必须能够精准而明确地回答这些问题。在任何时候，模式语言的使用者应该能够知道：我在哪儿，我当前可以做什么，我从哪里来，要到哪里去？

没有哪个模式语言格式可以为所有的受众同时解决好上面所有的问题。根据我们在3.2节中提到的：模式语言格式的多样性使得作者可以自行选择表现方式，同时与文献记载的其他模式语言保持一定的相似性。

12.3 格式的元素

所有模式语言格式主要关心的是保证语言具体而易懂。模式语言中所包含的模式越多，这一

点就越重要。如果模式语言的受众迷失在底层细节的海洋中，当他读到最后的时候早就忘了开始讲了什么，那么这个模式语言也很难被人们使用。反过来，语言中的单个模式的基本信息也不能丢失。要平衡好这两个方面，并不是件容易的事情。

12.3.1 展示全貌

要做到易懂而且不丢失全貌，模式语言的格式必须能够明确强调其目的和范围以及构成语言的模式之间的关系。正确地传递这些信息，对于完整地理解模式语言和语言所要表达的信息至关重要。没有一个清晰的结构，就无法将故事讲好，无法引导对话和传递知识。语言的受众就像进了模式的迷宫，不知从何入手，也不知从何退出：语言所要表达的信息就完全丢失了。

要表现模式语言的全貌，有3个操作特别需要注意，首先是命名。模式语言的名字应该能够清晰表达它是关于哪方面的，它要处理什么内容，它对什么有好处，它构建的对象是什么。比如，在与API设计有关的模式语言或者分布式计算的模式语言中，我们都可以使用Batch Method。

模式语言的名字也揭示了该语言的范围。比如，接口设计模式语言的范围要比分布式计算模式语言的范围小得多。如果潜在的读者或用户没有意识到模式语言的目的领域，他们很可能不会去阅读或者使用它。很多有价值的模式语言被人们遗忘，仅仅是因为其名字不能提供有效的信息，比如我们在10.9节中提到的Caterpillar's Fate模式语言的命运。

在展示模式语言全貌时，第二个操作是更为详细地描述了其所在的领域，包括对那个领域中出现的主要驱动力的概述。这个概述对模式语言的范围作了进一步澄清。比如，该领域覆盖了哪些方面和类型，有哪些关键需求、约束和期望的属性来驱动该模式语言所支持的具体设计。

在展示模式语言全貌时，第三个操作是关于其内容的概述。这部分总结了它所包含的模式以及这些模式之间的关系。通常，这些模式只是包含一些简略的描述，如3.5节所讨论的那样，通过一个关系图描绘它们之间的关键关系。内容概述部分还为潜在的用户提供了有关解决方案的建议，以及它可能创建的具体设计和解决方案。

与单个模式类似，模式语言本质上也是基于经验的。使用一个或者几个例子来演示模式语言所支持的解决方案可以对笼统的、概念性的描述起到补充作用，而且更有助于其使用和理解。

12.3.2 简洁与细节

对于简洁性的追求，自然要求作者只能提供语言中模式非常基本的信息。正如我们在第3章所讲，这些信息包括模式的名字、上下文、在该上下文中出现的核心问题、在解决该问题时需要注意平衡的驱动力、问题的核心解决方案以及解决方案的后果——特别是对于驱动力的处理。冗长的描述和细节实现通常都被省略了，或者建议参考附录和其他出版物，因为这些信息会分散读者对于语言基本信息的注意力。

比如，为了将其集成到模式语言中，我们可以提供如下精简版本的Batch Method，请对比我们在3.1节中较长的细节描述。

Batch Method

我们可能需要在聚合组件上执行批量访问。

有时候客户端希望在聚合组件上执行批量访问，例如从集合中获取满足某种特性的所有元素。如果访问聚合组件的开销很大，比如它是远程的或并发的，单独访问每个元素会引入相当大的性能损失和并发开销。

如果聚合是远程的，每次访问会引入延迟和抖动，占用网络带宽，并引入额外的故障点。如果聚合是并发组件，每次访问的开销中还得把同步和线程管理计算在内。类似地，每次调用还要完成一些辅助性代码，如授权等，这进一步降低了性能。尽管如此，我们必须能有效且完整地对聚合执行批量访问。

因此：

设计单独的Batch Method以在聚合上重复执行该操作。该方法的声明中接收每次执行操作所需的全部参数，例如通过数组或集合，并且结果的返回也采用类似的方式。其类图如图12-1所示。

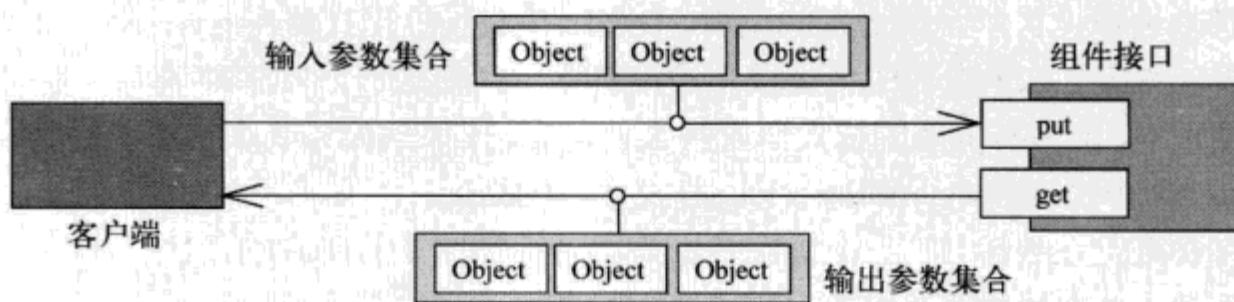


图 12-1

Batch Method将重复操作合并到数据结构中，而不是在客户端进行循环，因此循环是在调用前后执行的，分别为调用做准备及完成后续操作。

Batch Method将访问聚合的开销减少为一次访问或几次“大块”的访问。在分布式系统中，这样能显著提高性能，减少网络错误并节省宝贵的带宽。尽管使用Batch Method后，对聚合的每次访问成本提高了，但批量访问的总体成本降低了。批量访问也可以在方法调用内部进行适当同步。

本质上，这个版本的Batch Method提供的信息与3.1节里面更长的版本是相同的。我们知道它的使用时机、它应对的问题、相关的驱动力、解决方案和构建的关键方面，以及模式的后果。另一方面，这个缩写的版本省略了一些具体的实现信息，对上下文、问题、驱动力、解决方案和后果的描述也比较简洁，没有提供详细的讨论和探究。

如果模式语言的规模比较小，只包含有限的几个模式，对其中的模式描述得稍微详细一些是可以的。然而，模式语言的作者通常都不会提供太多信息，而倾向于使用较短的描述。如果一次阅读多个模式，读者可能会对模式语言的意图产生混淆：它是将某个领域系统地贯穿起来还是只是将一些毫不相干的设计和编程技巧裹在一起？模式语言无疑应该具有第一个意图，通过适当的上下文将设计与编程技巧串联起来服务于一个明晰而可行的目标。

12.3.3 模式连接

要想在产品软件项目中成功应用模式语言，必须明确地列举和讨论其组成模式之间的连入 (inbound) 和连出 (outbound) 关系。模式语言中一个模式的上下文不仅是由其所要解决的问题环境决定的，而且是由模式语言中其他所有模式所共同决定的。其中连入模式限制了模式的“面向情形” (situation-oriented) 的上下文，因为在一个语言中，任何一个模式都内嵌于同一个语言的其他模式之中或者与其他模式联合使用的。因此，如果某些驱动力可能出现于更为一般的环境中，但是在该语言的上下文中不会出现，就不需要讨论或者处理。

按照上面所说的，将模式添加到Batch Method的笼统的、独立于语言的上下文中，我们可以得到如下的特定于语言的上下文，正如我们在分布式计算的模式语言[POSA4]中所使用的那样。

Batch Method

在Explicit Interface、Iterator或者Object Manager中……

……我们可能希望在聚合组件上执行批量访问。

有时候客户端需要在聚合组件上执行批量访问[……]

同样，模式语言中某个模式的实现也经常从语言的其他模式中受益。因此，语言的使用者关心的是，存在哪些连出模式，在实现该模式的过程中何时应该使用那些模式，它们（如何）应对哪些方面或问题，对于这些方面或问题有没有可以替代的模式。请记住，模式语言应该引导其用户提问和回答正确的问题，并作出具体的决策。

通常，对语言中其他模式的应用会在模式描述的讨论部分提出，这与单个模式是类似的。比如，我们可以将下面一段添加到前面的Batch Method的描述中，以将其与分布式计算的模式语言[POSA4]中的其他模式联系起来。

Batch Method通常是特定的而不是通用的。它们以特定的操作命名，而且其参数直接反映了输入和结果。例如，查找匹配某个键值的结果可以表示为单一方法。但是如果需要更加通用，则需要更多的参数以控制所封装的循环，例如查找所有早于特定日期或者大于特定值的条目。更加一般化的方式可能是传入谓词 (predicate) 或者采用Command对象形式的控制代码，这就使得Batch Method更像是Enumeration Method，它们在分布式环境中有着相似的不足。

对连出模式的讨论跟一般的模式描述一样，都遵循简洁的规则。我们的关注点在于将语言的各个模式连接起来，而不是连出模式的具体实施和实现细节。

12.3.4 再说元素

模式语言格式中的元素至少包含3个方面：全貌、基本的模式描述和如何将语言中的模式编织在一起。这3个方面处理得越清晰明确，该模式语言就越可能在产品软件项目中成功地被应用。

12.4 细节，细节，细节

单个模式的作者可选的格式很多，既可以使用简单的概述，也可以使用细节详述，而记录模式语言则通常都是使用短模式格式。虽然短模式格式忽略了很多模式细节，但对于记录模式语言来说，却是一个理想的选择。特别是，它们平衡描述语言的全貌和组成模式的核心内容与模式间的关系。

12.4.1 模式语言的格式

在描述模式语言中模式的众多格式中，Alexandrian的格式是最杰出的[AISTT]。它具有本章前面所说的所有特征。

- 核心问题和核心解决方案构成其骨架。
- 对问题驱动力、解决方案实现及其后果的讨论扩充了该核心。
- 上下文和相关模式引用确定了模式在语言中的位置。

所以，有不少软件模式语言是使用Alexandrian的格式或者相似的格式编写的，比如本章中对Batch Method的描述。这方面的例子包括分布式计算的模式语言[POSA4]、Server Components模式语言[VSW02]、Object-Oriented Remoting 模式语言[VKZ04]以及 Human Computer Interaction模式语言[Bor01]。

另一种可用的格式是Portland格式，CHECKS [Cun95]和Caterpillar's Fate [Ker95]使用的就是这种格式。Coplien格式用得更多，*Pattern Languages of Program Design*系列的第一卷和第二卷[PLoPD1][PLoPD2]中的多个模式语言都用了这种格式。当然，还有其他格式，但是通常跟Alexandrian的格式比较接近或者跟Coplien的格式比较接近。

格式“只是”一个交流的载体，模式语言也可以转换成其他格式。但是，跟单个模式类似，将模式语言从一个格式转换到另一个格式并不是简单的复制粘贴然后再做点修改就可以了。这是一个翻译过程，不同的格式代表了不同的风格和结构。比如，Alexandrian格式是一种自由体，而Coplien格式则包含严格的结构。Alexandrian格式不要求讨论“结果上下文”，而Coplien格式则要求。Portland格式大体上是符合Alexandrian格式的，但是它既不要求有上下文，也不要求在问题和驱动力之间以及解决方案和解决方案的讨论之间做明确的区分。

恰当的格式跟作者的审美和风格喜好有关，同时也跟语言的技术信息以及受众的期望有关。3.6节讨论了哪些方面会影响到如何选择使用什么样的格式记录组成模式语言的模式。最终，由作者根据自己的观点和经验来权衡这3个方面以作出自己的选择。

12.4.2 鸟瞰图

即使使用最短的模式格式——或者包含再多的超链接，理解模式语言的完整性也是不容易的，不论其包含多少个模式。正如我们在12.3节列出的那样，绝大多数模式语言都包括一个绪论，其目的是概述该语言。这种鸟瞰式的概述有利于引导语言的读者浏览语言的主题、目的、信息、全貌和内容。这种概述采用哪种观感比较合适依赖于（跟选择模式的格式一样）作者的喜好、技

术内容、读者期望以及该语言的大小。

对于小型模式语言来说, 简单介绍一下其所在领域, 然后对于语言中的每个模式分别进行阐述, 然后用一个图表展示模式之间的核心关系就足够了。类似的例子包括C++ Reference Accounting模式语言[Hen01b]和Java Exception Handling模式语言[Haa03]以及Organizational Patterns For Agile Software Development模式语言[CoHa04]。对于后者, 模式语言中的每个概述还包括一个简短的模式故事, 描述如何在具体的组织和项目中使用该模式语言。

这种描述方式对于大型的模式语言来说是不可行的。几十页模式简介加上大量用来展示各种模式关系的图表, 实际上对于读者来说意义不大——他们更可能干脆扔掉而不会采用。

分布式计算的模式语言[POSA4]采用了另外一种方式。我们将其划分为多个模式组, 每个组代表分布式计算的一个问题域。语言概述部分简要介绍了每个问题域, 列出了相应的模式, 展示了一个用于说明问题域之间核心关系的图表, 如下所述。

分布式计算的模式语言包括114个模式, 它们被划分为13个问题域。每个问题域代表与构建分布式系统相关的一个技术主题, 包含了语言中所有与该主题相关的模式。问题域的主要目的是保证语言和模式更加具体易懂: 应对相关问题的模式在统一的、清晰的上下文中展示。问题域(总体上)是按照相关性和构建分布式系统时使用的顺序出现的, 具体如下所示。

- 从混沌到结构[简要列出问题域]。
- 分布式基础设施[……]
- 事件处理[……]
- 接口划分[……]
- 组件划分[……]
- 应用控制[……]
- 并发[……]
- 同步[……]
- 对象交互[……]
- 适配与扩展[……]
- 模态行为[……]
- 资源管理[……]
- 数据库访问[……]

上述13个问题域相辅相成, 涵盖了分布式系统构建中的各个技术方面。问题域之间的主要关系如图12-2所示。

这个高度压缩的模式语言概述后面是较为详细的问题域介绍, 介绍的格式符合前面提到的短模式语言结构。换句话说, 分布式计算的模式语言使用了两层结构进行介绍: 宏观概述主要是让读者关注作为一个整体的语言, 第二层更为详细, 为每个问题域设定明确的上下文。这样读者就可以在阅读每个模式之前决定自己要阅读和消化多少概述知识。读者也可以决定要详细阅读语言的哪一部分, 或者选择什么样的顺序阅读。

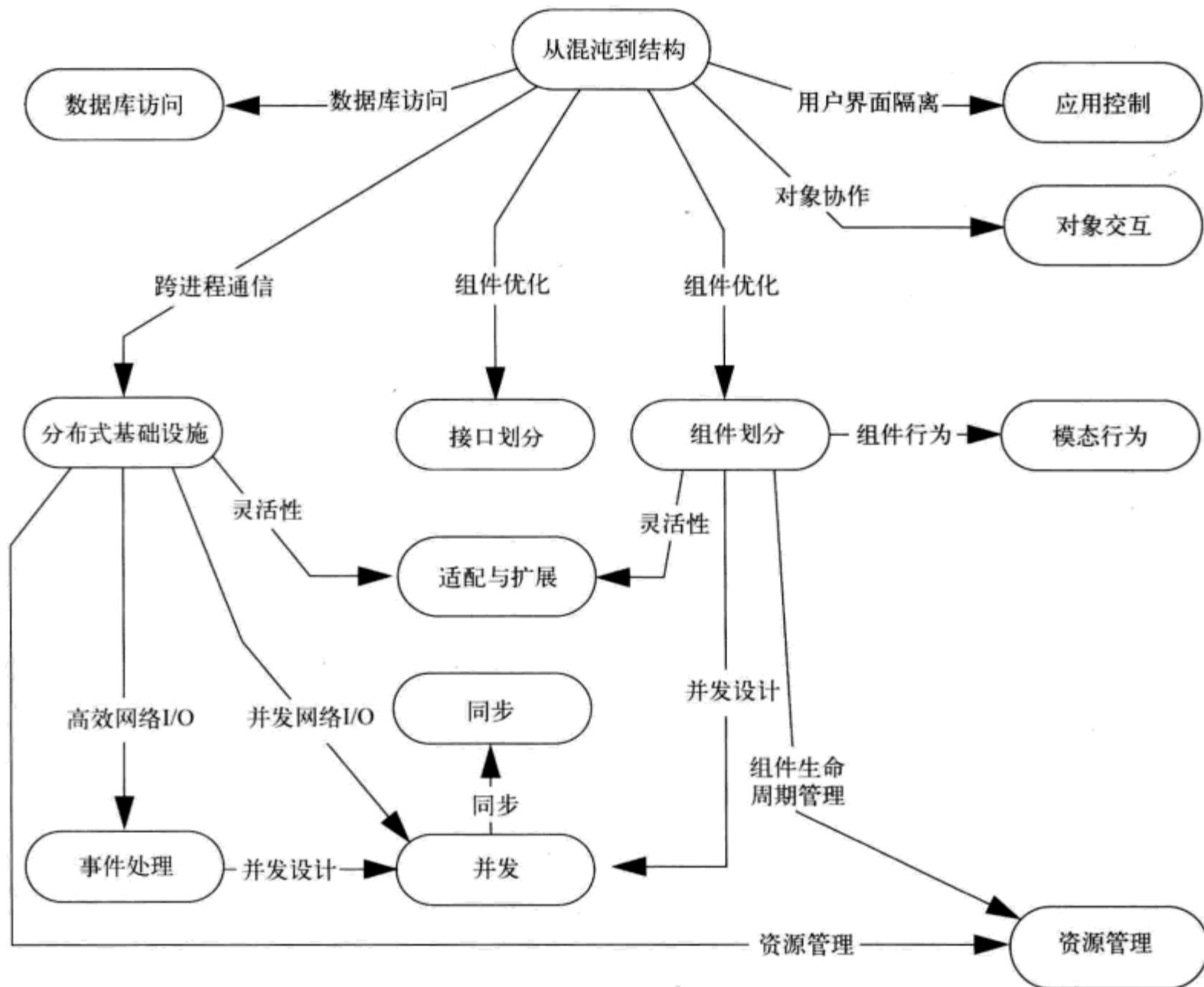


图 12-2

下面的内容摘自[POSA4]对事件处理模式的介绍。

即使中间件平台为应用提供了更复杂的通信模型，如请求/响应操作或异步消息传输，但分布式计算从根本上说仍然是事件驱动的。事件驱动软件所要面对的挑战与一般的“自主”(self-directed) 控制流软件有所不同，具体如下所示。

- 事件的异步到达[关于该挑战的简要介绍]
- 多个事件的同时到达[……]
- 事件到达的不确定性[……]
- 多种事件类型[……]
- 隐藏事件分离和分发的复杂性[……]

为了优雅而有效地应对上述挑战，事件驱动软件通常采用控制流倒置的Layers架构。
[……]

尽管Layers方案通过单独处理每个关注点的方式将事件驱动软件的不同关注点分离开, 但它并没有解释怎样在一系列驱动因素下来优化解决某一特定关注点的问题。例如, 事件分离层本身并不能确保将事件高效、简单地分离并分发到事件处理程序中。

分布式计算模式语言的4个事件处理模式有助于解决这一问题。它们为事件驱动软件中关键的事件分离和分发问题提供了高效、可扩展、可重用的解决方案。

- Reactor模式[模式简介]
- Proactor模式[……]
- Acceptor-Connector模式[……]
- Asynchronous Completion Token模式[……]

图12-3描绘了Reactor和Proactor是怎样集成到模式语言中的。

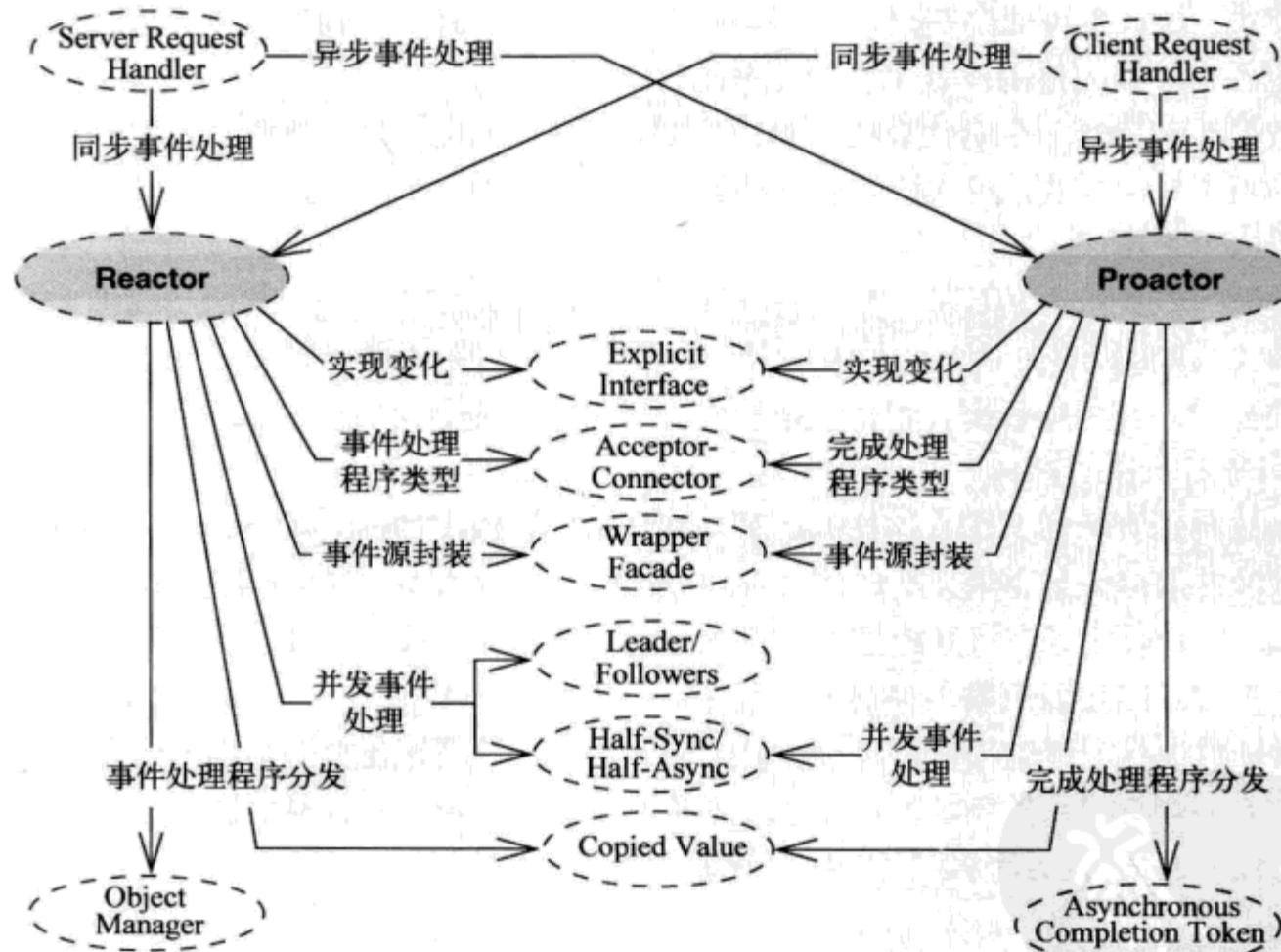


图 12-3

[另一个图展示了Acceptor-Connector和Asynchronous Completion Token模式是如何集成到分布式计算模式语言中去的。]

无论对模式语言的介绍多么简洁或者多么全面, 无论它包含了多少细节, 它必须能让读者理解和应用该语言。

12.4.3 展示顺序

模式语言中模式的展示顺序是另一个重要问题。没有一个清晰的顺序，就不能提供一个完整的故事脉络：语言就成了随机组合的模式集合，失去了它的重要本意。一条经验是将模式按照“广度优先”的方式展示，首先展示入口模式，然后递归地往下展示随后的模式，最后是叶子模式（它们不包括其他的模式了）。

大多数模式语言（比如*Organizational Patterns For Agile Software Development* [CoHa04]中的模式语言和分布式计算模式语言[POSA4]）根据其范围对其包含的模式进行排序。但是这种简单的方式并不总能奏效。比如，有些语言有多个入口模式。如果入口模式的范围有所区别，我们可以先展示范围较宽的模式，然后依次展示较窄的模式。

如果不适用范围标准，或者这种标准不可行，也可以先展示从最易使用的模式，然后展示较为复杂的模式。模式也可能会在语言中不同的地方使用时，具有不同的范围和影响。这时可以根据最广的范围、最窄的范围或者主要的范围进行排序。

另一条经验是选择一个有代表性的模式序列，这个序列要能够用到所有的模式，通过该序列按照相应的顺序展示模式。模式语言所支持的模式序列越多，单个序列的代表性就越差，也就越适合根据范围组织模式。

另一种组织模式的方式是按照其关键性展示——战略型模式最先，其后是战术型模式，最后是辅助型模式。这种组织方式使得语言作者可以清晰地表明哪些模式对于形成架构或者核心结构来说是最重要的，因为这些模式将排在最前面。介绍架构或解决方案局部细节的模式排在后面，如我们在8.4节所讨论的那样。

在诸如分布式计算模式语言这样的大型模式语言中，模式按照问题域进行了分组，表示也可以根据问题域的顺序分组整理。通过将语言划分为较小的语言块，可以保证应对相关问题的模式集中在一起，这样简化了在可互换解决方案之间的交叉引用，减少了预读和章节跳跃。在每个问题域内部，模式可以按照范围、关键性或者使用的顺序进行组织。

像往常一样，对于给定的语言来说，模式的组织顺序除了跟受众、范围、规模有关之外，还跟作者的个人喜好有很大关系。有效的排序方式应该能够引导读者选择合适的模式序列，帮助他们为手头的任务形成可持续的设计和解决方案。

12.4.4 示例

为了展示模式语言的正确用法，例子是必不可少的。例子为我们演示了如何使用语言中的模式构建生产质量体系或者解决方案，跨越从理论概念到日常实践的鸿沟。在选择例子时，有很多情形，具体如下所述。

能够展示模式用法的例子可以很好地演示模式之间的互动：它们如何连接、集成和相互支持的。如果要选择可运行的例子，有两种方案。

- 将其与语言中的每个模式集成起来。这种做法的好处是能够即时展示模式。
- 将其与语言分开，并在单独的章节里面展示。这种方法虽然可以整体上展示语言的全貌，

但是要找到能够用到所有模式的例子却不是件容易的事情。

大多数模式语言都是通过单独的章节来展示一个或几个模式故事, 详细分析语言中的某个领域。

例如, 分布式计算模式语言向读者介绍了仓库管理领域的一个大型模式故事。这个故事展示了构建分布式系统的3个方面: 创建基线架构、设计通信中间件和详述系统的特定组件。虽然这个例子只覆盖了大约三分之一的模式, 但是几乎所有的问题域都覆盖到了。因此, 这个例子足以完整地展示了语言的使用[POSA4]。另一个例子是, 在*Organizational Patterns For Agile Software Development* [CoHa04]中记录的4个模式语言中, 每个语言各包含一个短小的故事, 其长度不超过一页, 并且列出了它们在具体项目和组织中的用法。

有些模式语言也采用多个模式故事的方式, 共同将语言中的模式联系起来。例如, Context Encapsulation模式语言[Hen05b]包含3个模式故事, 分别使用C#、C++和Java描述。

如果找不到有代表性的例子, 模式语言中的模式也可以提供单独的例子。尽管这种方式能够很好地解释每个模式, 但是却不能告诉人们在整个模式语言的背景下各个模式是如何相辅相成的。*User Interface Design for Searching* [Wel05]就是为每个模式提供一个例子的模式语言。

如果模式语言被划分为多个问题域, 还可以采用别的方式。对于每个模式组, 我们可以给出一个典型的、示范性的模式故事来将所有的模式包含进来。

理想情况下, 3种方式都可以用到。

- 模式语言中每个模式的基本问题通过小例子在其描述中演示。
- 较大的、独立的“专门的模式故事”总结了影响问题域或者模式组中所有模式的方面。
- 一个单独可运行的例子讲述整个模式语言中某个特定的模式故事。

如此一来, 所有的模式都做了详细描述并附有实例, 但又不至于忽视全貌。这种方式还为模式语言带来了多样性, 这跟为一个单个模式准备多个例子演示其不同方面和应用差不多。我们在覆盖模式语言的主题及其不同路径之间取得平衡。

编写模式语言的目的是支持构建可工作的系统[Mes01]。所采用的例子必须是实际可行的。可行的例子或者是基于某些产品系统的, 或者是从中提炼出来的。产品的例子可以增加模式语言的可信度, 因为这是在实际的产品中所采用的解决方案。与之类似, 从产品系统中提炼出来的例子忽略了一些不重要的或者会分散读者注意力的细节, 保留和强调模式语言及其组成模式中的关键部分。相反, 纯杜撰的例子绝对无法模拟或者代替现实, 在用于指导产品软件开发方面缺乏说服力。

除了求实之外, 例子的规模和复杂性应该跟它所要说明的模式语言或者模式的规模和复杂性相匹配。过于简单的例子难以说服人们采用复杂的、相互交织的模式语言或者高级的模式。同样, 过于复杂的例子则要求读者将注意力放到理解例子上面, 分散了读者对于语言或者模式本身的注意力, 不论它是大是小, 简单还是复杂。一个好的经验法则是, 例子只要能够说明作者要解释的问题就好了, 没有更好, 因为这会转移大家对模式语言及其模式的注意力。

12.4.5 细节程度

与讨论例子类似的一个问题就是有关设计的模式语言要不要用到代码。显然, 有关编程的模

式语言（比如Smalltalk Best Practice Patterns [Beck96]、C++ Reference Accounting [Hen01b]、Java Exception Handling [Haa03]和Context Encapsulation [Hen05b]）如果没有代码，就不会有什么说服力，也没什么用处。到底使用多少代码取决于语言的细节程度。

总地来说，代码只要能够解释清楚自己要描述的意思就可以了。为了保证把重点放到语言和模式上面，我们最好只在最关键的编程问题上使用代码，在不太重要的部分使用文字解释。如果模式语言跟编码实践关系不大，而主要是关于一般的设计和架构的，比如分布式计算模式语言[POSA4]，那么使用代码或者使用太多代码就可能分散其主题。在这种情况下，作者应该少使用或者不使用代码。

在模式语言中既限制代码数量又能传递关键的编程信息的一种方法是，采用一种标记法将代码片段附着到以设计为中心的元素（比如类）上。我们在分布式计算模式语言中大量使用了这种方法，如图12-4所示的Proactor模式图。

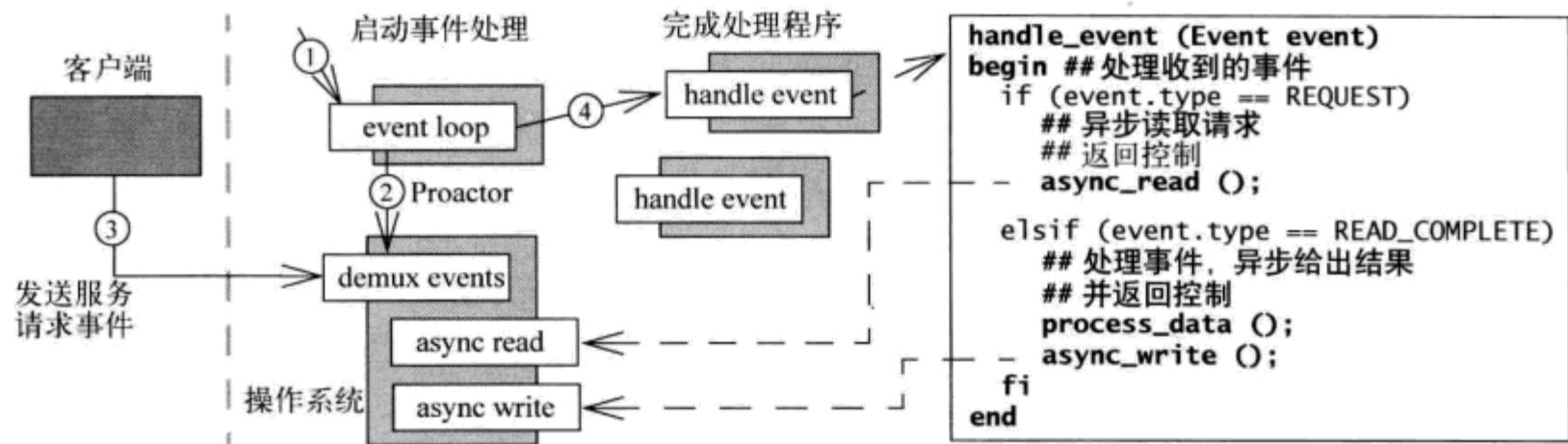


图 12-4

这个方式使得我们可以在展示模式的设计结构时，把一些有助于理解模式结构的代码（伪代码）展示给读者。

是否要展示代码牵扯到一个更基本的问题：模式语言中的模式到底要涉及多少细节？此前我们说了，模式语言的作者应该只展示最基本的信息，否则既要囊括所有重要方面，又要保证语言的易懂性，并展示模式语言的全貌，这个平衡很难实现。

即使作者找到了一个能够满足大多数读者的平衡点，总会有人觉得不够详细、不合自己的口味。要增加更多的细节，作者必须三思而后行，否则可能引入其他方面的不平衡。一个好办法是引用可以在网络上找得到的论文或者代码，来补足缺失的信息。这种方法保持了语言的简洁性和专注性，同时读者在必要的时候也可以找到自己需要的内容。

理想情况下，我们总能找到模式的更深入的描述。我们在POSA4的分布式计算模式语言中使用了该方法。几乎所有的组成模式已经被文档化独立模式了，而且绝大多数都具有良好的结构和丰富的细节。如果模式语言的读者想要得到更多的细节，可以很容易、很方便地找到。模式语言通过一个全景图将这些独立的描述结合在一起。如果在别处找不到详细的描述，提供进一步阅读的指导或者附录是有帮助的。

即使模式语言是使用短格式描述的，仍然会有一些读者觉得包含了太多细节。比如，POSA4

分布式计算模式语言包含了100多个模式，分成几个子模式语言，将近400页。我们使用了短格式进行描述，但是仍然无法一口气读完。遇到这种情况，作者从模式中直接删除内容以缩小语言的规模不见得合适，因为这样可能会使得其内容过于肤浅。

Christopher Alexander [AIS77]发现了一个跳出这种困境的简单而优雅的方式，我们在分布式计算模式语言中也使用了这个技术：模式描述中包含它们自己的摘要。这个摘要包含了核心问题和核心解决方案的陈述，参见12.3.2节Batch Method模式的描述。我们以粗体格式来记录这两项内容，读者可以从视觉效果和概念上对每个模式的精髓有所把握。读者在浏览语言的时候，可以先阅读粗体的部分，然后阅读自己关心的部分细节——比如，与问题相关的驱动力和对解决方案的解释。通过这种方式，读者可以提高阅读大型模式语言的速度。

12.5 再论风格与本质

与模式的描述一样，我们可以断定描述模式语言的格式对于语言的作者和读者来说同样重要。模式语言格式定义了一种展示和交流信息的工具，当然这里面也包括特定的观点和成见。虽然模式语言的本质是独立于其表述的，但是表述的格式还是会影响到人们对语言的认知、理解和应用。

尽管如此，所有的模式语言的格式只是向目标受众传递模式语言某方面特质的一种手段。这个格式也跟受众建立一个对话——关于某个主题的具体设计，比如软件系统、开发组织或者住宅建设。没有实质内容，考虑格式就毫无意义。我们在本章开头所引用的Kent Beck和Ward Cunningham有关模式的论文中的段落也印证了这个观点[BeCu87]。

模式语言通过为设计过程中可能出现的已知问题提供可行的解决方案，来为设计人员提供指导。这可以看做是一系列知识，它们以某种风格和顺序记录下来，目的是指导设计人员在正确的时刻问出（和回答）正确的问题。

简而言之，语言的内容真正描述有实质的东西时，其格式就越发显得重要。所以，最好的模式语言必然是既有合适的格式，又有恰当的描述，同时还有完善的内部模式网络。

领导者应该是实干的、务实的，但同时也能够与空想者和理想主义者沟通。

—— Eric Hoffer，美国社会作家

本章比较了独立模式和模式语言。基于对二者概念上的探索，我们总结出它们的异同点，进而理解我们从概念以及表述中得到的实践和理论方面的经验教训。

13.1 模式和模式语言：共性

当谈及模式及模式语言时，常常有这样的一些问题：在何种角度下这两种概念是一致的？在何种角度下它们是不同的？这些问题的答案有助于我们理解和正确评价它们各自在模式概念中的作用。

13.1.1 共同的核心属性

将我们在模式上的探索与在模式语言上的探索相比较，可以发现“好的”、完整的模式及模式语言的核心属性以及它们的文档格式是相通的。具体来说有以下几点。

- 二者都定义了过程和物件。例如，在设计模式中，“物件”是针对特定问题的一个特别的设计轮廓或者代码细节。对于以设计为核心的模式语言，“物件”是整个系统或系统中针对特定应用和技术领域的某个方面的设计和实现。对于组织模式而言，“物件”是特定的工作流或组织的一部分。对于组织模式语言而言，“物件”是组织的全部或其中一致的组成部分。过程则是基于模式或模式语言来指导用户如何构建这样的“物件”。
- 二者都保证它所支持的“物件”在结构、行为以及核心属性方面都是经过验证的、高质量的。
- 二者都明确着眼于各种驱动力，这些驱动力都会影响具体设计的观感及它所建立的环境的特性。
- 二者都通过描述完整的解决方案空间，而不仅仅是单点解决方案，来支持通用性。
- 二者都支持有助于理解现有方案的性质、驱动力以及结果的交流、讨论以及对话。

- 二者都有一个具体的名字，这样读者可以记住它们，并立即知道它们各自的主题以及所针对的领域。
- 二者都是在不断发展着的，它们需要针对它们所在主题领域的新经验和知识不断更新。
- 二者都有很多不同的实现方式，甚至不存在两个完全相同的实现。每个实现都依赖于它所应用的系统或环境的具体需求、限制及范围。

当然，模式和模式语言有着这么多相同的属性，这并不奇怪，因为模式语言是由模式组成的！为了提供它们自身的质量和特性，整个语言以其组成模式的属性为基础并加以增强。

13.1.2 共同的根源

很多模式语言可以视为其根模式的详细阐述和分解，这也正是它们有很多共同基础的另一个原因。这些语言建立在根模式基础之上，并以递归的方式逐步展开和增强。一个著名的例子就是Remoting Patterns[VKZ04]，这个模式语言将Broker模式分解到一个针对现代通信中间件架构的全面描述中。作为一个独立模式，Broker模式很难有那么强的感染力，但Remoting模式语言揭示了它的通用性。这样的例子还有很多：GoF的Objects for States模式被分解到State模式语言[DyAn97]，Allan Kelly的Encapsulated Context模式[Kel04]被分解到一个针对上下文封装的模式语言中[Hen05b]。

模式与模式语言之间的关系类似于类和框架之间的关系。类是一组数据以及操作这些数据的方法的联合体。而框架则将类引入一个更具体的、相互联系的上下文中，提供了一个整体，而类只是其中一部分。与之类似，独立模式只是某个上下文中一个特定问题的解决方案，但在模式语言中，它则是构成完整语言必不可少的一部分。

13.1.3 一个模式的模式语言

模式与模式语言的比较涉及了基数。只有一个模式的模式语言算不算是模式语言？或者说模式语言是不是至少要有两个模式？

从理论的角度看来，没有任何论据否定模式语言可以只包含一个模式。例如，前面我们给了空模式序列和单个模式序列的例子。如果一个问题域很窄，只包括一个问题，而且如果这个问题只有一个可行的、可重用的解决方案，那么针对该领域的模式语言就可以只包括一个模式。

然而在实践中，不管我们分析哪个领域，都会发现现实世界不会这么简单。即便是再小的领域，往往也不会只有一个问题存在。此外，针对任何问题，相应的解决方案也往往不止一个。结论就是，实际中常用的模式语言都包含多个模式，原子模式语言更侧重理论性而非实用性。

不过很多实用的模式语言都是小型的。例如，针对信息完整性的CHECKS模式语言就只包括11个模式[Cun95]。Executing Around Sequences[Hen01a]和Context Encapsulation[Hen05b]模式语言甚至更小：这两个语言都只有4个模式，而且这些模式很好地覆盖了相应的领域。Localized Ownership: Managing Dynamic Objects in C++[Car96]则更小，它只有3个模式。然而，只含有一个模式的模式语言到目前为止还没有谁提出过。此外，除非将模式复合从模式序列升格为语言，否则我们不曾见过只有两个模式的模式语言。

简单地说，只有一个模式的模式语言只存在于理论中，但实际上却不太会发生。

13.2 模式与模式语言：区别

模式与模式语言的最主要区别在于它们的范围、重点和意图。

13.2.1 模式和细节

不考虑其他因素，所有模式都是独立的。每个模式关注一个特定的问题和与之相对应的解决方案，或多或少的和其他模式及问题都是相对独立的。有些模式的范围相对大一些，或者在具体的设计或解决方案上的战略性影响会大一些，例如，用于构建系统基线架构的模式与关注于编程实践的模式相比较，但这些模式仍然只针对各自单一的问题。因此，独立模式的效果对于整个系统而言（比如关于架构的观感和质量属性）是有限的。

正因为独立模式的范围相对狭窄，所以它们可以将重点放在深度上探索实现解决方案的细节，而不会变得太宽泛、太复杂。使用像GoF和POSA之类的格式，很多模式描述都关注于将解决方案的架构和动态的附加信息应用于完善和补充它们各自的问题—解决方案核心上。此外，它们还提供了一些实现解决方案的帮助信息，借助实例说明问题和解决方案，详细讨论已知的应用。

独立模式的描述常常采用各种图表来表述模式的细节。有些时候，这些图表会使用正式的符号，比如UML。有时候也通过代码段或者其他详细的构造细节来说明模式是如何实现的。通过提供这样的细节描述，独立模式有助于各种领域中具体的设计和开发活动。因此，读者不仅可以概要了解模式是什么，而且可以有效地将模式应用到他们的系统、方案和环境中去。

13.2.2 模式语言和交互

与独立模式关注深度不同，模式语言则描述了如何将模式族应用到构建系统或者某个特定领域的解决方案中。每个模式语言都提供了针对各自领域的“旅游指南”。领域决定了可能发生的问题及相应的（可能不止一个）解决方案，各个问题和各个方案间的依赖，以及解决问题的顺序。

在模式语言中，每个模式都是相对狭隘的，只定位于该领域发生的某个特定问题，应对先行模式的上下文。模式语言作为一个整体，必须保证其中的问题不是孤立解决的，而是充分考虑到了其他相关的问题和解决方案。因此，模式语言在整个系统或方案的架构、设计、实现等方面起着相当重要的作用。而且，它们的范围还会扩展到除软件领域之外的其他领域中，比如项目管理和开发组织。

模式语言把各个模式系统地、一致地联系到一起，每个模式语言定义一种架构风格或者相关的一系列风格。架构风格常常在各个细粒度层次上影响系统或方案的设计和实现，贯穿于它的各种抽象和各个部分。为了达到这样的效果，模式语言的文档格式往往关注于模式网络，而较少关注独立模式中常见的模式细节。这些格式强调模式间的包含关系以及它们之间的次序。通过提供这些信息，模式语言将有助于（软件）设计。因此，读者可以从模式语言中全面理解特定领域及其挑战，并利用这些信息在他们的解决方案和实现中合理地映射这些领域。

13.2.3 两个独立的世界

软件模式社区的一些讨论和观点认为模式只有在模式语言中，才能发挥其最大功能。从这个观点中我们可以得出结论：记录独立模式是没有太大意义的，作者只需要关注于模式语言的编写。真实情况是这样吗？

我们并不赞同这个观点。我们相信既需要以问题-解决方案形式记录单个模式，又需要丰富的与领域紧密相关的模式语言。获知和理解某个领域和可行的架构设计只是问题的一个方面。任何架构都需要具体的实现，但是模式语言的描述常常并未提供必要的、深入到内部的模式实现信息。相应的独立模式的描述则可以关注于细节，而且大多数模式的描述就是这样的。这种观点适用于有关软件设计和实现的模式和模式语言，以及与软件开发、项目组织、设计过程以及文档等相关的模式和模式语言。

我们观察到以下相关联的两点。

- 关于模式语言的文档目前还很少。虽然这个数量在不断增加，但与软件开发领域的工作范围方面的文档相比，此类文档还是比较少的。
- 开发人员一旦得到授权，他们的工作就会变得高效而充满智慧。他们可以把很多不同的模式组合起来应用到他们的工作中。独立模式的文档针对其适用性提供了宽泛的上下文以支持这种组合。

因此，虽然这些文档没有提供模式交互方面的高深建议，但独立模式让开发者自由发挥、组合和发现未被记录的模式序列。这种情况正如早期的面向对象，那时候早期的从业者们还在致力于学习如何写出高效的、可重用的类和独立的类库。直到多年之后，面向对象社区掌握了足够的技术，把很多类集成到框架中，为一系列相关应用提供高效的、可重用的架构[John97]。

13.3 模式“对”模式语言

首先回到第9章开头提出的问题：如何才能最好地从整体上使用模式来设计和构建软件系统。模式语言有助于针对某个特定领域的系统和解决方案定义合理的架构和设计。相应地，对于每个组成模式的描述，可以保证架构和设计的构造、实现是高效的，并且都采用了合理的方式和方法。

换句话说，缺少了其中任何一个模式的实际概念都是不完整的：独立模式和模式语言的描述格式是相互补充的，而不是互斥的。在很多方面，将这两者分开是不合理的——模式和模式语言不存在“对立”。

模式要怎么用，以我的理解……它是一种工整的编排，这没问题。而我们开启的模式语言确实还有别的功用，我不知道那些方面有没有被转译到贵学科。我是说，整件事后面有其根源——也就是对“在何种情势下有益于环境”这一问题的持续关注。在我的领域中，这个问题是有意义的。

—— Christopher Alexander, OOPSLA 1996 主题演讲

人是模式的首要读者。从独立的模式到模式语言，模式内包含了强烈的人的观点，最明显的是以模式读者的角度体现出来的，其他角色也会有所体现。本章将探索对于与软件有利害关系或受其影响的所有各方，模式为何有那么大的吸引力。我们还将列举常常与模式社区相联系的各种价值，以及能助模式作者一臂之力的若干写作技巧。

14.1 模式以人为本

前几章主要从内容、性质和呈现方面探讨了模式。但它们的目标受众是谁呢？为了谁而花费如此巨大的精力去识别并记录种种好模式呢？模式社区中 Christopher Alexander [Ale79] 和 Jim Coplien [Cope00] 等开拓者相当清楚这个问题的答案：模式以人为本！

鉴于我们总是将各种模式放在技术上下文中去呈现，乍看起来这个答案有些出乎意料。但进一步观察迄今所揭示和强调的模式的各种性质，它们其实支持这种说法：模式的各种性质总是与作为受众的人有关。当然不是所有的人，而是与具体模式或模式语言针对的学科、领域打交道，负责该领域的设计与开发或受其影响的人。例如，分布式计算模式语言[POSA4]主要是为分布式系统的软件架构师和开发者而创作的。相对地，针对项目管理的模式语言就把读者群设定在项目经理、项目行政人员、开发主管、人力资源等相关人员。

具备良好文档的模式帮助它设定的目标受众理解特定的问题、问题出现的情形、左右解决方案的驱动力、潜在的解决方案以及不同解决方案的具体实现及其代价。如何以好于“仅可读”的形式去呈现模式，我们也投入了很大的精力。我们向往着一种“讲故事”并启发对话的呈现形式。

模式经过阅读消化，它（设定）的读者群应能够判断模式针对的问题是否正好是读者的（软

件) 系统或(组织)环境中存在的问题。如果问题是相符的, 模式描述应能帮助读者从解决方案空间中选出最切实可行的方案。模式复合、模式序列及模式语言与独立模式具有类似的目标, 但它们还要针对一个更大的(特定)领域为读者架起问题空间到解决方案空间的桥梁。

14.1.1 模式价值体系

模式对人的关注由“模式价值体系”作为支撑。该体系的根源是1994年首届PLoP (Pattern Languages of Programming, 编程中的模式语言)会议上Norm Kerth主持的一次研讨会。它还可以联系到软件模式的早期倡导者奉为圭臬的伦理观念。Jim Coplien对模式价值体系做了如下总结 [PPR]。

我们不因为新想法本身而看重其价值。别的社区有理由重视创新, 但我们希望模式承载的是那些在实际应用中反复证明了其效用的想法。

我们相信荣誉要让该得的人得到。如果你用了别人的模式, 请注明来源; 如果你借了别人的想法来写一个模式, 请记下他们的功劳。

我们相信模式是软件工具箱中的一件工具, 它们不是灵丹妙药, 它们不会让生产力暴涨。它们只是对基本技能的培养。超出以上朴素的定位而对模式做浮夸的宣传, 是对软件开发的不负责任。

我们看重实打实的东西。模式以经验为根基, 传达能真正解决问题的方向和结构。它们不是教你如何找到问题的解决方案, 它们就是问题的解决方案。与其谈论如何编写模式, 我们宁愿编写模式。编写及批评各种模式能让我们学到很多东西, 这才是我们关心的重点。

我们为“无名特质”而奋斗。我们重视审美。

我们关心软件中人性的那一面。所有的软件都服务于人的某些需求或愿望, 应用某种模式能给人带来怎样的便利, 模式应该毫不含糊地说明白。我们相信模式永远不能取代提供专业知识的专家, 也不能取代熟练地解读知识并建造、制作、雕琢出编程杰作的程序员。

我们相信模式格式上的约束其实是对我们的解放, 因为它们让我们超越形式, 专注于设计中更重要的方面。

我们相信编码者的工作蕴含着极高的价值, 因为书写代码的人才真正触摸到交付给最终用户的软件制品。我们相信对于设计一个坚实可靠、功能完备、符合审美的软件系统, 编码者的贡献丝毫不弱于架构师和方法学理论家, 虽然人们常常认为对系统设计的理解力和影响力是后一类人的专属。

我们相信系统的使用者也应该参与系统的设计, 无论是正在使用的系统还是将要使用的系统。总体而言, 模式社区营造一种共同参与的氛围, 努力——有时候艰难地——缔造一个开放的社区。我们相信我们可以从其他学科中学到很多, 软件社区的“近亲繁殖”现象还十分严重。

14.1.2 人类读者

明确地将人类读者作为中心将模式与软件开发领域的其他工具及技术区分开，比如统一建模语言（UML）、模型驱动软件开发、面向方面的软件开发（AOSD）等。这些技术的创造产物虽然也打算给人阅读，但同时还要提供给工具作为输入，用工具对它们实施规范化的一致性及正确性检查，模拟它们，甚至用它们生成代码。从某个角度看，这些技术似乎超越了模式，因为它们显然拥有更大范围的读者，涵盖了人和（虚拟）机器。

这些技术都着眼于以某种方式为开发者和他们的开发工作提供支持。恰恰是“支持”方式的内在特点，令模式与众不同。比如说，琳琅满目的各种软件技能和技术主要把处理对象设定在技术产物的范畴：构成规范或代码的机制、为特定问题产生具体解决方案的机制。因此，在实践中，它们最适合用在系统的实现、文档编制以及调优。

与之相对，模式着眼于确立读者作为建筑师的角色，帮助他们用这样的身份去设计和理解系统，并针对它进行交流，模式语言将这一点体现得更为明显。哪种方案最适合解决特定环境下的特定问题？方案对系统质量微观及宏观上的影响是什么？方案是否平衡有度，易于理解和沟通？模式及模式语言帮助建筑师回答这些问题，作出明确而深思熟虑的设计决策。回顾一下Richard P. Gabriel在本书序中的总结。

模式语言能够涵盖很多种设计空间。它们是各种软件设计和构造工具箱的组成部分，其功用体现在帮助人更好地设计。模式语言的书写反衬出设计的优劣。设计，设计，设计，设计。

然而，把其他软件技术和模式一分为二，认为两方不可并存，是一种错误的想法。每种技术都有明确而实用的目标定位，各自都在软件开发过程中占有一席之地。其实模式对面向方面软件开发[KLM+97]、模型驱动软件开发[SVC06]等生成技术[旨在（半）自动化选中的软件开发活动]也有支持和帮助，这在本书中已不止一处讨论过。

上面讨论的关系也适用于技术与模式，乃至软件设计及开发领域之外的模式语言。例如，有一系列的体系和评估制度定位于实现、控制和改进开发组织及过程，包括CMM（能力成熟度模型）及其后继者CMMI（能力成熟度模型集成）[CKS06]，以及RUP[Kru00]中关于组织和过程的建议。与这些技术方法对应的模式语言支持针对特定业务及项目需要，量身设计具体的组织及过程。

本章余下的内容将深入探讨模式的人性本质，而会将讨论重点放在软件设计和开发领域，毕竟这个领域才是我们的专业所在。然而类似的讨论话题也可以延伸到其他领域的模式及模式语言，包括开发的组织和过程、业务的组织和过程、文档、住宅建筑等。

14.2 对软件开发者的支持

大多数软件模式支持的人类对象是开发者，他们实践“一种科学与艺术，设计和构建应用程序、框架等类似结构，使其安全地抵抗可能承受到的内外压力，同时兼顾经济与美感[Hen03b]”。

可以形象地说，模式支持的人是那些工作和生活在他们创造、重构、维护的系统的设计和代码之中的开发者。因此让他们有家的感觉，体会到安全与舒适是非常重要的，不能让开发者对系统有陌生感，觉得迷茫和别扭。模式是创造这样一种宜居环境的卓越手段[Gab96]。

从技术角度来说，大部分以开发者为中心的模式引导设计师和程序员在系统构建期间的关键决策，协助他们完成高质量的软件系统。这类模式还使开发者高效地探索与软件相关的问题，并研究评估解决方案的细节和不同选择。另一部分以开发者为中心的模式有助于开发者理解和应用第三方软件，例如通信中间件和面向服务的架构平台。研究在背后支撑起这类软件的模式，有助于开发者恰如其分地定制、扩展及使用第三方软件。

还有一系列模式及模式语言针对开发过程本身，涵盖每日的开发活动和围绕代码的周边环境[Cun96] [CoHa04]。这类模式比较不注重制作的产品，而较为注重制作过程和制作者。

由人驱动的开发

模式不总是硬邦邦地讲求技术性。它们也涉及软件开发中一些较为柔软、更明显地以人为导向的方面。

- **沟通。**模式塑造一组专门而通用的语汇，让软件架构师和开发者用来讨论项目中出现的具体问题。通用的语汇有利于人们互相交换解决问题的构思，在同事之间交流整个系统的架构和实现。拓宽语汇，使开发者更充分地表达自己正在做或计划要做的事情，使开发者群体对问题及其解决方案有更充分的共同理解，以及对具体架构及其实现有共同的理解。
- **信心与体谅。**凭借模式所蕴含的经过验证的既有知识，软件开发者有把握确定自己为系统选择了坚实可靠的技术根基，做了正确的设计及实现决策。在项目会议上，开发者从所用模式的约束条件和实施后果出发，既能解释设计背后的原理和折中，又能欣赏和理解别人的设计思路。
- **理解、责任与控制。**模式允许软件开发者在尝试解答重要的设计和实现问题之前先做推导。此外，开发者在解决问题的过程中运用模式中的概念，也会产生出更高效的实现。作为将模式纳入求解过程的结果，开发者变得更加投入，更负责任，对进行中的工作也更有把握。假若情况正好相反，开发者对产生问题的大背景不了解，就会无所适从，不清楚自己在做的是怎样的工作，到底为了什么理由和什么人。
- **兴致和乐趣。**模式有利于建设性地解决问题，从而有利于保持软件开发者对工作的兴致，因为他们不必花费大量时间去东拼西凑临时的解决方案，那样的工作不但乏味又容易出错。此外，大多数模式不仅“在技术上”解决相应的问题，它们还展现出一定的美感与优雅。开发者对自己的设计与实现感到自豪并且非常喜欢它们，这进一步增加了他们对工作的兴趣。

简而言之，以开发者为中心的模式除了给予开发者技术上的建议，它们还鼓励、肯定、支持软件开发者在工作中的人的身份。反过来又促进了开发者在个人与团队两方面的工作效率——这是成功的软件开发团队的一个关键因素。

14.3 对软件使用者的支持

有些软件模式设定的读者不是那些构建并生活在软件之中的人，而是最终用户、产品客户、产品经理这些被软件影响、忍受着软件的人。这群人往往对软件开发参与得太浅，虽然正是他们使用、购买和销售软件。以用户为中心的模式的目的是将这群人拉回到开发中去，帮助他们与软件开发者针对他们在软件系统中所关心的问题展开有意义的讨论。

以用户为中心的模式有意避开属于软件系统内部的技术问题，比如事件处理模型、并发设计。它也不是将以开发者为中心的模式改头换面，重新包装给另外的目标群体。以开发者为中心的议题往往与最终用户、产品客户和产品经理没什么关系，他们必须依赖开发者“做正确的事”[Hearsay02]。以用户为中心的模式专注于软件系统的那些可从外部感知及观察的性质，比如它们的观感、功能组合还有可操作性。这些易用性方面的属性决定了与软件共事共处的生活体验，因此提升这些属性可以提高受软件影响的人的兴趣。

14.3.1 用户界面

软件系统的观感密切决定着使用系统的容易和舒适程度。它还在软件是否被客户和最终用户接受的问题上扮演着关键角色。因此，让最终用户、产品客户和产品经理加入开发者的行列，参与用户界面的设计与实现，是软件项目成功不可或缺的因素。

在以用户为中心的模式中，有一组突出的例子恰能佐证上面的讨论，也就是Jan Borchers的人机交互(HCI)模式[Bor01]。例如，Invisible Hardware和Domain Appropriate Devices模式强调对用户隐藏陌生的复杂硬件技术，让用户通过他们已经在本身领域中熟悉的设备及“抽象”来与系统沟通。另一些HCI模式，如Flat and Narrow Tree和Information Just in Time模式维持着信息充分与信息泛滥之间的微妙平衡。

[Bor01]中介绍的模式可以被软件项目中的不同群体有效利用。例如，系统工程师和产品经理可以利用HCI模式说明在系统的用户界面中如何呈现信息，系统该提供哪些协同工作的便利设施。类似地，软件开发者可以用相同的模式，向最终用户、产品客户、产品经理介绍并使他们熟悉软件系统的行为和观感。

易用性和以人为本的视角看似是模式社群的一个新发展方向，但其实它根本就是将模式引入软件开发领域的原始推动力！文献记载的第一批软件模式，Kent Beck和Ward Cunningham记录下来的Window Per Task、Few Panes Per Window、Standard Panes、Nouns and Verbs和Short Menus[BeCu87]等模式，指导开发者如何创建最终用户能够理解并高效使用的软件，这明显抱有以人为本的愿望。

14.3.2 用户需求

在系统观感的外表之下，应该是充分相信系统能够满足应用领域的用户需求。需求从用户到开发者的传达与反馈，同样是让最终用户、产品客户、产品经理参与开发进程的一部分，这与让他们参与设计用户界面一样重要。或许产品客户与产品经理在这个过程中会邀请有经验的分析师

作为协助，但我们不会就此认为这个分析场景中的模式是以开发者为中心，而是依据它们面向用户的立场，将其看做是以用户为中心的。

举例来说，编写用例的模式[Coc00]以一种场景驱动的视角，帮助用户和开发者清晰表达出功能性需求。其他一些分析模式，比如医疗保障和企业金融领域中的一些例子[Fow97]，容许领域专家与软件开发者讨论如何在软件系统中建立应用领域的模型，商议需要怎样的特性才能恰当地满足功能性需求。对接开发领域的语汇和应用领域的语汇，创造一种开发者和领域专家的共同语言，正是*Domain-Driven Design*[Evans03]的目标之一。领域驱动设计模式[Niel06]协助分析师和开发者建立能准确定义和验证业务规则的领域模型。

14.3.3 交到用户手中

Alexander说，在理想状况下，模式应当保证“关于构建什么、如何构建的一切决策都交到用户手中”[ASAIA75]。“建筑师-施工者有责任确保实际的设计掌握在住户的手中”[ADMC85]。因此，与以开发者为中心的模式相比，以用户为中心的软件模式与Christopher Alexander的理想状况联系得更紧密也更明显。不过取决于环境、预算、技术追求等因素，在软件项目中可以或多或少地逼近此种理想境界，但难以完全达到。有心又有权决定观感和易用性等议题的人与拥有建筑专业知识和经验的人，两者之间的距离无法完全消除。

以用户为中心的模式及模式语言有利于缩小这种距离。让最终用户、产品客户和产品经理更全面、更明确地参与软件开发全过程，可使他们对开发中的事物更有责任感和控制感。具体来说，他们不必等系统交付才看到结果，而是可以确保系统为满足他们的需求而演进。敏捷开发过程及其一些相关模式[CoHa04]体现了这种迭代的、由反馈驱动的方式。

将以用户为中心的模式施加于软件开发过程，有助于提升最终用户、产品客户、产品经理对软件的信心，也能提升他们对开发者的信任度。以用户为中心的模式把受到软件开发影响以及有利害关系的所有各方纳入对话，凸显了它强烈的人性本质。因此，对于让最终用户、产品客户、软件开发者协同定义软件系统观感及架构的“参与式设计”[DFAM02]领域，以用户为中心的模式广受欢迎实在是顺理成章的事情。

14.4 对模式作者的支持

模式作者是另一群重视模式及其与人关系的群体。好的模式写作并非一蹴而就，不可能在单一作者的笔下就一次塑造得完美、全面。本书已经探讨过模式的格式，论证了模式文档的结构与风格可对读者有重要的影响。我们也用了很多例子说明模式（尤其是模式语言）的撰写过程是迭代的、增量的，甚至“完成”状态也只是一种中间状态。我们还没有提到写作背后活跃着的不仅只有作者的身影。

模式社区中已涌现了众多实现高效写作的新方法，它们强调协作与反馈，请看下面的讨论。

14.4.1 协同写作

许多形成文档的模式都是不止一位作者的写作成果。当前的科技论文大有要把同一屋檐下或

者邮件列表上所有正式研究人员全部列为作者的趋势，可是就模式文档作品而言，“集体创作”的说法相当名副其实。

有时候共同创作是面对面进行的，有时候是通过电子邮件进行的。例如，《面向模式的软件架构（卷1）》之后的大部分著作是由居住在不同国家身处不同大洲的众多作者共同完成的。创作团队常常利用时差来实现“夜以继日”的写作接力！

除了传统意义上的共同创作，模式写作还利用更大范围的网络效应去吸收更多贡献者。例如 *Fearless Change* [MaRi04] 和 *Design Patterns for Distributed Real-Time Embedded Systems* 两本书中的大部分模式都是从历年的会议研讨中收集、提炼而来的。Ward Cunningham 创造的 Wiki 更是一个实实在在的例子。Wiki 的初始目标之一就是记录、协同创作和浏览各种模式，*Portland Pattern Repository*（波特兰模式仓库）[PPR] 即体现了这种目标。各种 Wiki 系统的发展已经远远超越了它的初始目标，产生了像维基百科[Wiki]那样前所未见的大规模协作成果，而将 Wiki 用于模式的共同创作的最初设想依然有着旺盛的生命力[Shir]。

14.4.2 作者研讨会

评审书面作品有很多种方法。学术出版物要经过匿名的同行评审。商业出版有面向市场的评审。如果同事值得信赖，人们也经常采用非结构化的评审来征求评语。模式作者的研讨建立在信任与开放的基础之上[Cope96]。

好的评审可以保证模式的可读性，确保它能打动读者，确保它完成了使命。模式也是文学作品，我们认为值得为它投入更多努力，超过我们通常对软件文档所做的干巴巴的设计评审。Richard Gabriel 身为一名诗人（以及 CLOS——Common Lisp 对象系统的发明人之一），教给了我们诗会上诗歌的评审方式。我们的评审环节借用了那种方式，将称为“作者研讨会”。

研讨会的参加者都是模式作者。作者选几个模式作品，然后就自己看自己的去了，直到研讨会的最后阶段。其中一位评审者总结该模式。然后开放讨论该模式为何可行，以及它有何优点。接下来，评审者们可以提议如何改进该模式：内容、形式、用途，任何方面都可以。最后，作者可以要求澄清。

除了上述基本形式，还有许多细节与变通[CoWo99] [Gab02]，但基本过程就这样简单。这种研讨会形式构成了 PLoP 系列会议的主干，它们同样适用于用户组、公司内部乃至各种感兴趣的团体的非正式会议。

14.4.3 牧放

牧放（shepherding）是模式社区采用的另一种结构化的评审形式。它提供了一种有导向的结构化评审形式，评审的目的是改进基于模式的作品。它不同于传统上的一次性评审，有意设计成迭代式进行[Har99]。它还怀有尊重之心：牧人不是作者，所以建议仅仅是建议而已，不能篡改或者提硬性的要求。

牧放通常用在审议投稿给 PLoP 会议的论文，准备用于作者的研讨会。它还被用于若干模式

书的出版（如[POSA4]），以及非正式地用于同事之间（如[Kel04]）。最近几年，学术会议也开始采用牧放的方法，以在出版之前改善投稿论文的质量。

14.5 技术为人

本章前面的内容一直在强调模式的首要目标——使设计和技术能被人类读者所认知和理解。这个目标是要确保人们（包括用户与开发者）有意识地、深思熟虑地为他们的系统或环境选择恰当的表达方式或开发方式。

与其他技术相比，模式不仅诉诸具体项目中出现的各种角色，比如软件开发中的项目经理、软件架构师、程序员，它还明确诉诸角色背后的人。尽管有自动化软件搭建等技术上的先进性以及因此产生的收益，模式承认，软件还是由人构建的，并且最重要的是构建它们是为了人。

绘画永远不会结束，它只是停在了一个有趣的地方。

——保罗·加德纳，画家

人们很少有机会反省自己对于未来的预测。有3次机会回顾（和修订）自己的预测，更是千载难逢！这就是我们写系列书的好处。

本章回顾一下2004年我们在《面向模式的软件架构（卷3）》一书中所做的展望。我们讨论一下过去3年里模式前进的方向，分析其现状，以后见之明对我们的预测进行修正。

15.1 过去的3年

随着时间的推移，我们对于模式的未来越来越清晰。与我们在《面向模式的软件架构（卷1）》和《面向模式的软件架构（卷2）》中所做的展望形成鲜明对比的是，我们在《面向模式的软件架构（卷3）》中所做的展望绝大部分都得到了验证，或者正在成为事实。让我们先来回顾一下，过去3年都发生了什么，有哪些相关的作品出现——尽管我们难免有所遗漏。更多的信息请参考 <http://hillside.net/patterns/>。

15.1.1 模式与模式语言

3年前，我们最主要的预测是，模式语言将变得越来越重要，越来越流行：“人们将更加关注于记录模式语言而不是独立的模式，大多数新的模式和模式语言是属于特定领域的而不是通用的” [POSA3]。通过简单的调查，不难发现我们的预测非常准确，而今最有影响力的模式作品绝大多数都是针对某个具体的模式语言而非独立的模式，涉及的也是具体的领域，这跟我们的预测极为相符。

□ 安全性。安全性从本世纪初期开始成为一个常见的模式主题，这和人们对软件系统中安全性的广泛关注密不可分[GSS03]。该领域的领军人物出版了 *Security Patterns* [SFHBS06]，其中记载了大量安全相关领域的模式和模式语言，比如验证、授权、完整性和机密性。其中很多模式和模式语言都在早期的模式著作中有所记载，绝大部分来自过去3年的模式

会议。*Security Patterns*最突出的贡献是基于POSA的形式重新组织了这些模式的描述，并将其集成为一个连贯而一致的模式语言。*Core Security Patterns for J2EE Web Services* [SNL05]是另一本网络软件安全领域的模式著作。

- 容错和故障管理。在过去10年里，容错和故障管理一直广受关注[ACGH+96] [IsDe96] [FeRu98] [BRR+00] [SPM99] [Sari02] [Sari03]。随着软件日益融入功能和安全性都非常关键的系统——特别是大型分布式系统，故障管理对于这类系统来说越来越必要。最近的几本书和论文包含了与容错和故障管理相关的模式和模式语言[Han04] [LoWo04] [Sari05] [Utas05] [POSA4]，它们有助于系统获得更高的运营质量。还有一本介绍软件容错的书正在筹备当中[Han07]。
- 分布式系统。尽管分布式系统已经被模式作者讨论了很多年了，但这仍然是一个热门话题。例如，《面向模式的软件架构（卷4）》为构建分布式软件系统提供了一个完整的模式语言[POSA4]。该语言包含114个核心模式和150多个相关的模式，其内容涵盖了定义和选择合适的基线架构和通信基础设施以及指定组件接口、组件实现、组件交互，以及分布式计算的关键技术，比如事件处理、并发、同步、资源管理、适配和扩展、数据库访问等。其他最近出版的有关分布式的书记录了模式的可靠性、可用性和可扩展性，其针对的上下文包括所谓的电信级系统[Utas05]、分布式对象计算的关键原则和实践[VKZ04]、点对点计算[GrMü05]和时间触发的分布式嵌入系统[HKG04]。
- 语言惯用法和特定领域的惯用法。过去10年间涌现出好几种新的编程风格，比如面向方面的软件开发[KLM+97][Ladd03]、领域驱动的软件开发[Evans03] [Niel06]和模型驱动的软件开发[Fra03] [GSCK404] [SVC06]以及产生式编程[CzEi02]。每种风格都有自己的惯用法，彼此之间有很大的差异。因此，涌现出很多模式和模式编目用以记录这些编程风格中最常见的惯用法，特别是面向方面的软件开发[Sch04a] [GCKL05a] [GCKL05b] [ScHa05] [Sch06b]和模型驱动的软件开发[VöBe04] [Vö05a] [Vö05b] [Vö06]。仍然有大量著作关注于主流语言的编程语言惯用法，比如Java [Hen04b]、C# [Nash06]、C++ [AbGu05]和Python [Chr01]。
- 流程。虽然很多软件开发流程在软件模式社区的早期就已经以模式的形式做了描述，在过去的3年里仍然有几个新的流程模式著作出版。有些著作关注的是软件开发流程的特定类型，比如分布式、敏捷、测试驱动、领域驱动开发等[BHCY+04] [Evans03] [Ber05] [BrJo05] [HCSH05] [NoWe05] [Mes07]。有些则是将已有的软件开发流程和组织模式集成到互连的模式语言中[CoHa04]。
- 教育。近年来，模式和模式语言已经成为一个被广泛接受的工具，教师和顾问通过它们交流如何讲授编程和软件工程的知识。因此，在过去的3年里也涌现出不少关于教育模式的书。PLoPD系列第五卷[PLoPD5]中就有一些这方面的论文。最近很多论文[StFl04] [CiTy04] [LRP04] [Chr04] [PoCa04] [BSDM05] [TZP05] [VeSa06]覆盖了如何在大学本科教育中使用模式的内容。这个领域的主要研究人员和参与者创立了“教育模式项目”(pedagogical patterns project) [PPP]，它通过网站的形式发布教育模式，并提供讨论教学

模式的论坛，交流教授好的软件实践方面的经验。

除了我们预测到的主题之外，在软件开发的其他领域也有一些有关模式和模式语言的书出现，比如（通用的）软件架构[Mar04a] [Mar04b] [Zdun04] [AvZd06] [Mar05]和群体互动（Group Interaction）[LS04] [Sch04b] [LS05] [SFM05] [SL06]。

2004年，当我们在POSA3中作出将有关于移动计算和普适计算的模式和模式语言发表的预测时，我们是很有信心的。虽然这两个领域有几部作品出现[Hon04][Bor06]，相对于移动和普适软件系统的飞速增长来说还是相当落后的。我们仍然相信这两个领域对于软件专家和模式作者来说仍然是极具吸引力的。要掌握这两种类型的系统是很困难的，因为这里面有很多技术和社会挑战，比如管理低带宽和不稳定带宽、低功率和不稳定功率、经常性的连接和服务中断。所以我们需要经过时间考验的模式来随时随地地帮助开发人员创建这样的系统，但同时也应当尊重个体，不宜过于指手画脚。

15.1.2 理论和概念

3年前，我们在POSA3的预测中提到“如今我们应该回过头看看过去所取得的成果，以后见之明调整和完善我们对模式的理解——模式（到底）是什么，它们具有什么样的属性、目的和受众，使用模式的时候有哪些注意事项。”当时我们希望有关模式和模式语言概念的书或者论文会出版或者发表，但是也知道这个圈子实际上很小。

不过这个预测大体上还是得到了印证：一批模式社区内的思想领袖开始撰写有关模式概念、模式序列和模式语言的东西。不出所料，这方面的论文确定很少[PCW05][Hen05b]，但是它们对于软件模式概念的知识编纂和理解都具有里程碑的意义。这个领域规模最大的著作就是本书，它深入探讨了模式概念，将其各个方面集成在一起。

15.1.3 重构与集成

随着经验的增长，模式社区慢慢意识到很多已有的模式描述需要重新修订，以覆盖当前的进展，既包括软件设计，也包括模式概念本身。早在2003年，就有著作[BuHe03]对某些最常见的模式进行了重构，形成了今天我们所看到的样子。基于这本书在模式社区所获得的反馈，我们在POSA3中预测：“后面还会出现更多的此类模式重构，特别是经典的模式书中发表过的模式。”事实证明这个预测是非常准确的：最近发表的论文和出版的书更多的是修订已有的模式，而不是描述新模式。

比如，在分布式计算模式语言[POSA4]中记录的114个模式中，有100多个模式在其他地方有记载，但是我们对其进行修订，把它们放到分布式计算这个上下文中，并且统一了它们的格式。Security Patterns [SFHBS06]也差不多，其中大部分模式都在各种模式会议上发表过，但是Security Patterns统一了它们的格式、写作风格和技术深度。在其他著作中，大部分是对GoF [VK04] [POSA4]和POSA [VKZ04] [KVSJ04] [Ort05] [MoFe06]中的模式做的重构。

有些模式其实已经重构了很多遍，比如Broker模式在《面向模式的软件架构（卷1）》中[POSA1]就有记录，正如我们在1.9节中所讲的那样。甚至对于最近发表的模式，也有人进行了优化。比

如，尽管Context Object最早是在Scheme的eval步骤中用来包含环境参数，同时也在Interpreter模式中扮演重要角色，但是作为单独的模式提出来却是最近的事情。2003年，它被作为J2EE特定的设计实践[ACM03]或者技术名词[Kel04]被记录，2005年专门为分布式计算做了修订[KSS05]并分解成一个模式语言[Hen05b]。

如我们所预测的，近年来模式作者们将更多的精力放在如何将模式更紧密地集成在一起上面 [SFHBS06] [VKZ04] [POSA4]。这个集成过程是一个自然的趋势，因为人们更加倾向于发表模式语言而不是独立的模式。

15.1.4 GoF

虽然*Design Patterns* [GoF95]已经出版了十多年，但它仍然是最具影响力模式著作。这本书具有开创性，对于我们思考软件设计具有深远的影响，但是现在毕竟已经不是1994年了。如今我们对模式和软件开发的理解已经越来越深刻，特别是对GoF的模式而言更是如此。此外，C#和Java已经成为主流的编程语言，Python和Ruby等脚本语言也得到了很大的普及，也算得上是主流语言了。而*Design Patterns*中的大部分例子主要是以C++语言描述的，个别使用了Smalltalk。我们在POSA3中预计，人们将继续讨论GoF的模式。

这一预测从多个角度得到了证实。从2004年以来，好几本书和论文从各种角度讨论GoF的模式。有些是讲述如何使用其他编程语言实现这些模式的[Met04]，有些则是讲述如何将GoF的模式与特定的应用上下文联系起来的[VK04] [VKZ04] [SFHBS06] [POSA4]。

在2006年举行的OOPSLA上，我们为John Vlissides举行了一个感人的追悼会。在追悼会上，GoF的其他几位成员回顾了该书的许多方面，包括该书在面向对象软件领域内的影响和John在本书编写过程中的重要贡献。虽然他们约定创作第二卷，但发布时间未定。

15.2 模式的现状

经过十多年的成长和演变，模式作为主流的软件开发方法已经深入人心。模式被人们有意识地应用于各种产品软件项目[YaAm03]和大学课程[Hor06]中，通常是为了解决某个具体的问题，但是偶尔也有整个项目应用模式和模式语言完成分析、设计和实现。模式社区最初的目标——记录并弘扬好的软件工程实践——似乎已经达到了。

在过去的3年里，从覆盖领域来说，我们看到了显著的增长。十几年前，人们发表的模式大部分是通用的，主要是关注于设计的，而对特定的领域（如分布式或者需求分析）提到的就非常少。今天则是另一番景象。各种领域的模式和模式语言层出不穷，比如并发、资源管理、分布式、事务处理、消息传输、安全、故障管理、事件处理、用户界面设计、企业应用集成、因特网应用、音乐、需求分析、电子商务等。我们很高兴看到这么多模式和模式语言的出现，因为这说明在实践中模式具有无可比拟的作用和表现力。

更重要的是，总体来说，现在的软件开发人员能更好地掌握模式，他们在使用模式完成软件项目方面具有更多的经验，同时对于模式概念的理解也更加深刻。发表的模式和模式语言的质量也普遍提高了。我们在15.1节所提到的模式和模式语言，大部分比过去发表的时候表现力更强、

更加全面、更加简洁、更加易读。从很多方面来看，可以说软件模式终于不再是“噱头”，达到了它的“黄金时代”[ReRi85]。

15.3 模式的明天在哪里

如果模式真的到达了自己的黄金时期，那么有一个问题就不能不问：“关于模式，还有哪些需要继续研究？”很多软件开发人员对模式的概念已经有了一定的认识，并且熟悉其所在领域的某些模式。我们希望熟悉模式基本知识的程序员会越来越多，熟悉了基本知识并想进一步深造的程序员也越来越多。除了学习的问题之外，我们对于模式的未来作如下预测。

15.3.1 模式与模式语言

首先，我们期待更多特定技术和特定领域的模式和模式语言的出现。到目前为止，还有很多的软件开发技术和领域是模式所未曾覆盖到的，也许还要几十年的时间才能覆盖到所有的领域。我们预计最先关注的技术和领域包括以下几个。

□ 面向服务的架构 (SOA)。SOA是一种将不同组织或集团控制和拥有的分布式功能整合利用的一种方式。这个术语最初是在20世纪90年代提出的[SN96]，它是对当时互操作中间件标准的一个概括，比如基于RPC、ORB和消息的平台。面向服务的架构目前已经发展到能够将中间件同更大的企业级IT基础设施集成在一起，同时也为其他领域的应用和产品线架构提供平台，比如第三代无线系统[Par06]和防护系统[Lau04]。SOA方法以来自分布式计算和企业系统集成的原则和技术为基础，因此有很多现成的模式和模式语言可以利用[CI01] [Fow02] [HoWo03] [MS03] [ACM03] [POSA4]。

某些SOA技术，比如业务流程建模、服务编排 (service orchestration) 和超大规模系统仍然未被触及，还没有覆盖这些方面的模式。最近我们看到有些作者对这方面有兴趣，希望在不久的将来看到这方面的模式著作。

□ 分布式实时嵌入系统。开发分布式实时嵌入系统过程中出现的问题是开发人员面临的最大挑战之一，因为在这些系统中，由计算机处理器来控制物理、化学和生物流程或者是相关设备。汽车、飞机、空中交通管制系统、核反应堆、炼油、化工厂、病人监护系统等就是这样的例子。在这些系统中，如果响应速度过慢，正确的响应也同样会造成故障，所以保证可靠的端到端的实时性能是非常关键的。此外，嵌入式设备的平台和应用使用的内存通常非常受限 (64KB~512KB)。

开发高质量的分布式实时嵌入系统非常困难，甚至从某种程度上讲是一种“黑色艺术”。有一本基于OOPSLA [DRE02]和PLoP [DRE03]会议的书[DiGi07]即将出版，它介绍了开发分布式实时嵌入式软件用到的模式。最近有些论文[Kon05]则关注于如何验证这样的系统。我们认为有关分布式实时嵌入系统的模式会越来越多，因为这对于开发关键任务和要求极高安全性的应用来说是必不可少的。

□ 群体互动。世界越来越网络化，因特网和Web使得我们可以随时随地获取信息。电子邮件

和因特网论坛、聊天室和社区使得人们可以随时和家庭、朋友以及商务伙伴保持联系和互动。即时通信、计算机电话和各种协作工具（如Wiki、NetMeeting、Webex）使得来自不同地区、不同文化、不同国家和大洲的人们可以一起完成项目，比如全球软件开发[SBM+06]，还有我们的POSA系列的撰写。

在虚拟的电子环境中进行的群体互动与现实世界中面对面的群体互动有很大的不同。比如，人们缺少表达和展现自我的途径，而且沟通方式也受到计算机和其他电子设备的限制。此外，在网络上交互时可能出现延迟、错误、误解，即使同一个文化、同一个时区的人们交互时也难免出现问题。要提高这种环境下的群体沟通和互动效果，需要专门的解决方案以弥补电子协作基础设施的不足，尽可能地模拟“人”的交互，比如（可视的）个性、感觉和手势。

虽然近几年发表了不少人机人交互的模式[Sch03] [LS04] [Sch04b] [LS05] [SFM05] [SL06]，但是还没有人将它们整合在一起。而且，在电子环境中的群体互动方面，还有些领域是模式尚未触及的——比如，支持大量玩家的虚拟世界和游戏。基于对电子协作需求的快速增长，我们预计在不久的将来，在人机人交互领域将有大量的模式和模式语言发表。

基于Web的群体互动也包括使用更为动态、开放的业务模型，而“利用群体智慧”[ORei05]成为重要的商业手段，Web 2.0成为媒介。如8.11节所述，最初的Web 2.0模式集包括了这个领域出现的一些成熟的实践。我们预计随着我们在这个领域经验的增长，会有更多的模式出现。

- 软件架构。尽管我们记录的模式语言越来越多，但软件行业还是没有一本类似于其他设计行业的完整手册。虽然*Design Patterns*、*Pattern-Oriented Software Architecture*和*Pattern Languages of Program Design*在这方面取得了稳定的进展，但是还没有达到我们的目标。最近，Grady Booch加入了这个队伍，他收集了上千个模式，来组成一本*Handbook of Software Architecture* [Booch]。他正在编纂一系列有意思的软件密集型系统的架构，展示其基本的模式，并提供跨领域和跨架构风格的对比。
- 移动和普适系统。正如15.1节所述，我们仍然认为移动和普适系统的模式和模式语言对于模式作者来说是一个非常重要的主题。虽然我们在POSA前几卷中就此作出的预言已经落空两次，我们仍然愿意重申一次。如果你在这些领域工作，并且熟悉有关的最佳实践和模式，请帮助我们使其应验！
- 流程。这回我们预测对了，在15.1节中我们讲到软件开发流程将被更多模式作者所关注[BoGa05] [MaRi04] [Ker04]，我们预计这个趋势还将持续下去。随着越来越多的组织采纳敏捷开发流程，人们在这种流程上的经验越来越多，我们将看到相应的模式文献也会越来越多。某些文献关注于大的流程方面，比如整个生命周期以及与商业的交互，有的则关注于小的流程方面，比如测试驱动开发、重构和工具的使用。

除了上面列出的这些之外，其他的模式、模式集合、模式序列和模式语言也可能会发表。正如本节开始所说，根据我们当前对于模式和模式语言的认识，上述主题是最有可能的。

15.3.2 理论和概念

虽然我们认为对于模式概念的各个方面我们已经做了充分的研究，具有了较为深刻的理解，但是未来的几年对模式概念方面的底层理论研究将继续下去。就目前来看，我们认为有两个领域值得关注。

□ 深化模式概念已知的各方面知识，特别是模式序列。

□ 探索新的观点，比如8.9节中列出的问题框架的观点。

总之，我们希望模式概念新的领域或者存在争议的领域得到进一步澄清和发展。

虽然Christopher Alexander通过他的4本关于*The Nature of Order*的书 [Ale03a] [Ale04a] [Ale04b] [Ale03b] 拓展了他的贡献，但我们不希望*The Nature of Order*定义我们关注的重点领域。它的贡献主要在于深化我们对于模式概念的理解，比如模式序列以及设计中“中心”的概念。毋庸置疑的是，有些研究者和实践者会选择深入地研究如何使用*The Nature of Order*里面的概念，但是我们不希望这成为主要的关注对象。

不过，我们认为对于模式概念及其各个方面知识和理解已经相当成熟。因此，虽然我们希望这些想法进一步得到传播和吸收，但是不认为将来在这个方面会有很大的突破。

15.3.3 重构与集成

我们也希望继续有人从事模式重构和集成的工作，并反映他们具体的使用经验、最新的（概念性）理解，以及软件设计和实现的最新实践。模式和模式语言总是在发展中，所以作出这样的预测也是自然而然的。

15.3.4 支持其他的软件开发方法

从20世纪90年代早期以来，我们记录的设计模式绝大部分来自于面向对象软件，其编写语言通常是第三代编程语言。然而，模式和模式语言已经开始影响和支持其他的软件开发方法，特别是面向方面的软件开发（AOSD）和模型驱动的软件开发（MDSD）。实际上，我们根据在文献中发表的模式和最佳实践达到了什么样的程度，衡量这些方法的成熟度和接受度。

例如，在过去的几年里，MDSD技术[SVC06]迅速成为了主流，特别是在嵌入系统领域[KLSB03]。MDSD技术基于领域特定语言（DSL）[GSCK04]，其类型系统确定了应用的结构、行为和特定领域的需求，比如软件驱动的无线电、航空电子任务计算、网上金融服务、仓库管理、甚至是中间件平台领域。DSL使用元模型描述，其中元模型定义了领域内概念的关系，精确地指定这些领域概念相关的关键语义和约束。开发人员使用DSL来构建应用，他们使用由原模型所刻画的类型系统中的元素，用声明的方式表达设计意图，而不是使用第三代编程语言所采用的命令式的方式。然后转换引擎和产生器分析模型的特定方面并合成不同类型的产物，比如源代码、模拟输入、XML部署描述及其他模型表现[Sch06a]。

有些人[VöBe04] [Vö05a] [Vö05b]已经开始记录与流程和组织、领域建模、工具架构和应用平台开发相关的MDSD模式了。发现和记录更基本、更详尽的有关MDSD技术的模式和模式语言，

为将来的模式作者打下了更为坚实的基础。我们目前能看出的有关该技术的模式工作领域包括以下几个。

- 支持使用各自方式设计和开发软件系统的模式和模式语言。比如，我们可以想象，更多的模式[Vlis96]帮助我们将使用第三代编程语言编写的软件与模型驱动工具或者第N代软件架构集成起来。
- 在这些软件开发方式中，如何最大程度地利用已有的模式和模式语言。比如，在MDSD方法中，将面向领域的、独立于技术的应用模型转换为更特定于技术的或者代表具体应用架构的模型的产生器，可以使用适当的模式和模式语言设计。同样，模式和模式语言也可以指导产生器执行的特定类型的转换，比如生成基于具体模式（特定于某个技术的）实例的目标架构[SVL06]。

我们希望模式和模式语言在面向方面的软件开发上产生的影响与在模型驱动的软件开发上产生的影响类似。一些基本的作品已经出版了，如上面提到的[HaKi02] [GSF+05] [CSM06] [CSF+06]，但是还有大量工作等待我们去完成。

15.3.5 对其他学科的影响

对于软件社区来说可能有点受宠若惊的是，其他学科也在从我们的实践中吸取知识并试图应用在自己的工作中。更有意思的是，建筑领域赫然在列，尽管软件工程（特别是软件架构）过去总是从他们那里借鉴和吸取各种方式和方法，包括模式方法[Ale79]。另一个从软件模式社区借鉴想法的学科是音乐创作[Bor01]。

另一方面，在模式感兴趣的所有学科中，可以说最成功地运用了模式方法的还是软件开发。因此，很少有机会让软件社区对其他学科在采用模式方法上产生影响，甚至将从软件开发中获得的关于模式和模式语言的经验知识回馈给建筑领域都很困难。

这种情况能否发生，不仅是预测的准确性的问题，它很大程度上超出了我们软件开发人员的控制——不过机会应该还是有的。

15.3.6 其他学科对我们的影响

正如前面所说，软件模式社区长期以来都受到Christopher Alexander在建筑架构方面作品的影响。除此之外，还受其他学科（比如音乐[AGA06]、文化人类学[Mead60]和人机交互[Bor01]）的影响。一个引起更大兴趣的影响来自于自然科学领域，生物学家逐渐明白DNA中哪些基本元素和模式在有机体之间共享，又是哪些使得它们各自不同。

POSA5的贡献之一就是实现了我们在POSA2中的预言。当时，我们预言模式和模式语言的研究和进展必将导致有关软件核心属性的突破性的发现，就像生物学家破译生物遗传密码一样。10.6节中所使用的针对模式序列语法结构的标记法正是我们探寻“软件DNA”的尝试。随着对格式良好的模式序列语法和语义定义（特别是系统控制实证研究）的进展，我们希望这些成果能够让架构师和开发人员在掌控大规模软件复杂性方面能比目前更加有效。

15.4 简述模式的未来

我们的预测（或者说展望）来自于十几年来在模式社区所积累的经验和深刻分析。虽然我们希望预测能够得到验证，但是人们应当对其抱正常的、怀疑的态度。

未来，模式也许会朝我们所想象不到的方向发展。因此，本章作出的预测只是众多可能中的一个，而不是绝对可靠、万无一失的神谕。



力量与你同在。

—— Grady Booch

最后一章回顾一下15年来我们的使命——面向模式的软件架构。回到15年前我们开始的地方，回顾一下走过的路。

天下没有不散的筵席，终于到了说再见的时候了。POSA系列该到结束的时候了。15年来，我们的工作几乎没有离开过模式——挖掘、整理、应用、教授，研究其概念基础。最开始是在1992年，那时我们的目的是记录下软件项目中所积累的经验，从而避免在今后再做重新发明轮子的事情。随着时间的推移，POSA影响的范围越来越广，越来越多的来自模式社区的朋友加入到这个项目中来作出自己的贡献，或者给我们提供建议或者帮助。我们出版了一系列图书，本书是最后一本了。这也是我们工作的顶峰，它将我们在前几卷中介绍的模式和模式语言联系起来。

从一开始，我们的工作就围绕着一个目的——面向模式的软件架构，这既是书名也是这一系列书的主题。这也正是我们的信念，模式不应该独立地应用于解决某个本地的或者孤立的设计或者开发问题，而应该系统性地用来指导整个软件系统的开发。阐述这一构想是朝正确方向迈出的一步：我们花了将近20年的美好时光探索并将这种构想记录下来，目的就是为了在产品系统中成功地应用模式。

在写《面向模式的软件架构（卷1）》的时候，我们天真地以为寥寥几章就能把模式和用模式构建软件的概念基础描述清楚。我们很快意识到，仅靠一本关注模式系统的书是无法将模式和模式的有效应用讲清楚的。随着我们对模式掌握得越来越深入，我们越发意识到还有太多的东西要讲，要想把模式的概念说清楚，我们还有太多东西要学。现在回过头来看，我们可以说《面向模式的软件架构（卷1）》只是对模式进行了初步的但却完备的介绍。

但是我们始终没有放弃最初的使命——面向模式的软件架构。在开发产品软件项目的时候，每当用到模式，或者发掘了“新的”模式，或者在会议上展示和讨论模式，或者在POSA前几卷中记录模式的时候，总会提醒自己我们对于模式概念本身（通过实践）学到了什么。

我们一点一点地收集针对模式概念的各种观点和意见，看它们是如何集成和关联在一起构成一个整体的概念。从POSA系列的前四卷中也可以看出，我们对于模式的概念理解以及对不同方

面的挖掘、演讲和整合的演进过程。

- 在《面向模式的软件架构 (卷1)》中, 我们对如何使用模式进行设计的理解相对有限, 因此, 我们虚心地将我们的方法称为模式系统, 而不是模式语言。虽然我们理解并强调模式之间关系的重要性, 以支持其在软件构建中的系统性使用, 但只考虑到了模式之间互相使用的关系, 其中一个模式的实现中使用到另一个模式, 而没有考虑到模式之间的互补、互换和复合。我们也没有讨论模式和模式语言流程方面的内容。虽然在模式中也提到了上下文和驱动力, 但是它们在模式语言中所扮演的重要角色却未被深入详细地探讨。我们只是简单地提到什么是模式, 如何使用模式进行设计, 但是并没有将其作为重点进行描述。
- 在《面向模式的软件架构 (卷2)》中, 我们第一次尝试用模式语言的形式记录知识。这两卷将其所包含的模式连接起来形成小型的模式语言。虽然这两个模式语言并未将其领域内的所有模式都包含进来, 但它们揭示了模式语言概念的重要属性。比如, 两个模式语言都提供了设计其所在领域内的系统和系统组件的流程。这两个语言在考虑模式之间的关系时, 不仅考虑了相关模式之间使用的关系, 更包含了替换和其他互补关系。不过, 对于上下文和驱动力的讨论不够明确和完整。
- 在《面向模式的软件架构 (卷4)》中, 我们弥补了前三卷中对于模式语言和使用模式进行设计相关方面讨论不够完整的不足。POSA4记录了构建分布式软件系统所使用的模式语言, 我们相信模式语言所需的各个方面都已经讨论到了。这个语言——特别是其关键部分——被应用于各种领域的软件项目中, 包括分布式流程控制和工厂自动化系统、电信管理网络系统、实时的航空电子应用以及通信中间件等。通过在POSA4中记录模式语言, 进一步完善和提高了我们关于模式和使用模式进行软件设计方面的知识和理解, 同时也强化了我们在撰写POSA前几卷时所学到的模式的概念。

在完成POSA4之后——距离POSA1的出版已经过去了10年, 我们觉得应该迈出在模式研究方面的最后一步了: 详细研究模式概念, 什么是模式, 模式是如何支持系统性和建设性软件设计的。本书便是我们交出的答卷, 它记录了我们对于模式的概念性基础方面最新的认识, 以及对于实现面向模式的软件架构的需要的方法、技术和工具方面的理解。我们相信我们在1992年所立下的使命已经完成了。

《面向模式的软件架构》系列写到了最后一章, 但这并不意味着有关模式的东西都说尽了。也不是说, 模式概念已经完全成熟, 没有其他的模式等待着我们去发掘和记录了。相反, 我们将本书所做的工作视为进一步前进的垫脚石——Stable Intermediate Forms, 在对模式的研究中, 本书可以看做一个坚实的基础, 未来是什么样子的, 未来在何处都需要我们进一步探索。正如模式和模式语言是一项进展中的工作, 模式概念本身也是这样的。

我们将继续参与这项工作并对这项工作作出贡献, 但是至少目前我们还没有再写第六卷的计划。天下没有不散的筵席, 是时候让新一代的研究者接过担子, 继续探索新的领域, 寻找新的模式和模式语言, 大胆记录那些别人未曾记录的问题的解决方案。

愿力量与你同在。

模式概念总结

本附录总结了我们在本书中使用到的并详细定义的关键模式术语。如果一个定义中包含了其他的术语，我们将使用楷体标注。这里关注的是软件和软件开发，否则像“架构”、“设计”这样的术语将有更加宽泛的定义。

分析模式	描述在某个业务领域中如何对特定问题建模的模式。
反模式 (anti-pattern)	参见“不完整模式” (dysfunctional pattern)。
架构风格	关键的软件架构设计决策，该决策横跨多个不同的架构，这些架构在某个层次上可以认为是相同的。架构风格使得我们可以界定和比较相似及相异的架构。
架构	参见软件架构。
受益	使用某个模式的后果（如果该模式被认为是有益的）。
代码坏味道	一种代码层次上的问题，它为重构创造了动机和上下文。代码坏味道可能出现在不健全模式中，或者与之等价。
组合模式	参见模式复合。
复合模式	参见模式复合。
后果	应用某个模式的结果。后果分为受益和不利。
上下文	应用某个模式的情形。上下文可能是笼统的，用于描述总体的环境属性；也可能是具体的，用于在模式语言或者模式序列中描述前置模式。
设计	一种创造性的有意识的活动：为了某个目的有意构思和创建某种结构。设计对于软件和软件开发来说，就是开发人员构思和创建一个系统的软件架构。通常，“设计”用于表示这类活动的结果。
设计模式	设计模式提供了一个架构，用于完善软件系统中的元素或者它们之间的关系。它描述了一种经常反复出现的角色交互的结构，用于解决某个特定上下文中通用的设计问题。
不完整模式	一种模式，其问题和解决方案不吻合，因此应用它的后果是问题没有得到解决甚至出现恶化。模式不健全的原因往往是它未能正确地或者完全地刻画一个问题，或者对于相应的问题来说，其解

	决方案太差。
例子	参见动机例子。
驱动力	在某些情形中，一些特性或者特征出现在一起的时候相互之间会产生冲突或者成为问题。在为该问题考虑有效解决方案的时候，驱动力必须得到平衡。
有生产力	一种通用流程所具备的特性或者使其能创建某个特定解决方案的规则。
惯用法	惯用法是一种模式，它特定于某个编程语言或者编程环境。惯用法描述了如何在给定的语言或者环境中，使用代码实现特定的行为或者结构。该术语也用于更为宽泛的情况，用来指编程语言或者环境中不必称为模式的通用实践。
不利	使用某个模式的后果（如果该模式被认为是有问题的）。
字面模式名	一种直白的模式称呼，使用其基本含义而不带任何比喻的味道。
隐喻模式名	隐喻模式名在模式和另一个概念之间建立某种联系，例如我们日常生活中的概念，读者往往对这些概念比较熟悉。
动机例子	一个具体的例子，用于解释某个特定模式所解决的问题的类型或者其解决方案是如何实现的。
名词短语模式名	名词短语模式名描述模式所创建的结果。它们通常描述模式的解决方案的结构，有时候直接枚举其中关键的角色。名词短语模式名强调应用某个模式的产物而非过程。
组织模式	组织模式关注开发组织的结构。它描述了常见的反复出现的人际交互或者部门角色交互的结构，用于解决开发流程上的问题。
托盘 (Patlet)	一种模式的简化描述，通常（但并不是非如此不可）带有一个简单的图表。
模式	模式描述某个特定的设计上下文中反复出现的设计问题，以及经过证明的该问题的解决方案。该解决方案描述了其各个组成部分的角色、它们的职责和关系，以及它们是如何协作的。
模式分类	参见模式集合。
模式集合	任何一组模式的组织形式。其内容可能包括临时的或者也可能层级组合在一起属于某个领域，解决某个问题，或者符合某个层次上的抽象。集合的组织可能是结构化的，也可能是非结构化的。
模式复合	一种由一组模式构成的模式。一个经常反复出现的模式群体，它们自己也被看做是一种模式。模式复合也称为复合模式或者组合

	模式。
模式描述	以某种模式格式对模式进行的描述。模式描述描述了模式解决的问题和模式建议的解决方案。理想情况下，模式描述应该清晰地给出模式所应用的上下文、问题中存在的驱动力，以及应用该解决方案的后果。模式描述还可能包含其他的细节，比如源代码和图表。
模式格式	描述一个模式所用到的文档结构和风格。
模式语言	一个相互关联的模式网络，它定义了系统化地解决软件开发中问题的过程。
模式名字	模式为人所知的名字。最常用的命名模式的语法风格是名词短语模式名，但动词短语模式名也很常见。从描述上讲，模式命名分为字面模式名和隐喻模式名。模式可能会有多个名字，这要看它所在的技术上下文和不同模式描述的历史。
模式仓库	参见模式集合。
模式序列	在某个特定的情形中，应用模式来创建某个特定架构或者设计的序列。从模式语言的角度来说，模式序列代表语言中特定的路径。
模式故事	一个关于模式序列和特定设计问题的描述，包括创建具体系统或者创建具体设计的例子。
教育模式	一种关于教学（特别是软件开发教学）中反复出现的实践的模式。
渐进式成长	以一种稳定增长的方式构建系统的过程，在这个过程中对反馈作出响应，与之相对的是大块头、专家计划方式，使用后者所述的方式，所有的设计往往在创建之前完成。
问题	在某个上下文中不希望出现的情形，模式采用某种解决方案来应对这些情形。问题可以理解为需要平衡的相互冲突的驱动力。
问题框架	问题框架对反复出现的一类问题进行命名、界定和描述。它是对某一类问题以及用于理解和推理这类问题的技术的概括。
原型模式	被认为是反复出现的问题之解决方案，但是对于模式描述或者问题的本质没有充分的理解。
无名特质	参见整体。
重构	一种用于提高软件内部结构的渐进式的活动，它通常是一个连续的、行为受保护的转换。
关系	设计元素之间的关联。关系可能是静态的，也可能是动态的。静态的关系直接展示在源代码中。它们处理架构中的组件的位置。

动态关系处理组件之间的交互。它们可能不容易从源代码和图表中看出来。

角色 相关元素所在的上下文中设计元素的职责。例如，面向对象类定义了一个角色，所有的实例均支持该角色。另一个例子是，接口定义了一个角色，所有的实现都支持该角色。如果一个元素支持某个给定的角色，它必须提供一个定义该角色的接口的实现。元素通过实现不同的接口公开不同的角色。不同的元素可能通过实现一样的接口公开同样的角色，这使得客户端可以根据特定的角色以多态的方式处理它们。一个实现元素可能具有不同的角色，即使是在同一个模式中。

运行例子 用于解释某个特定模式集合、模式序列或模式语言所处理的某类问题，以及模式在实践中如何应用的具体的例子。运行例子是在每个独立的模式描述中增量式建立起来的。

SEP 别人的问题 (Somebody Else's Problem)、软件工程流程 (Software Engineering Process) 或者模式软件工程 (Software Engineering with Pattern)，随便哪个。

软件架构 软架构代表系统中一系列重要的设计决策。这类决策包括（但不限于）对软件系统中子系统和组件及其关系的描述。通常会从系统的功能、操作性、开发等各个不同的角度描述子系统和组件。系统的软件架构是软件设计活动的产物。

解决方案 模式用于解决问题和驱动力的结构和活动。

缩略图 模式的简述，通常包括简短的问题陈述和模式解决方案的精髓总结。

动词短语模式名 动词短语模式名是命令式的，给出如何实现模式所期望的解决方案状态的描述。动词短语模式名强调模式的过程而非产物。

整体 设计在完整性、可持续性、有效性、适用性和合意性等方面的质量。

参考模式

本章对书中所引用到的模式做了一个总结。所有的模式总结都引用了原始的模式来源，这样你可以找到更多的细节。模式按照字母顺序列出。有多个名字的模式——或者如果在很多情况下可以互换的模式——只在其中一条记录中给出完整定义，别的地方都采用交叉引用的方式。对于那些我们或者模式社区认为其质量或者分类仍然存疑的模式，我们简要地解释了它们特定的问题或者为什么它们被认为质量不够高。

Abstract Factory	Abstract Factory模式[GoF95][POSA4]提供了一个接口，用于创建和删除一族相关的或者相互依赖的对象，而无需将客户端与具体类相耦合。具体的Factory实现给定的一族对象类型的Factory接口。
Acceptor-Connector	Acceptor-Connector模式[POSA2][POSA4]将网络系统中连接和初始化相互协作的对点服务（peer service）与该对点服务连接和初始化之后所执行的处理过程解耦合。
Active Object	Active Object模式[POSA2][POSA4]将服务请求和服务执行解耦合，以增强并发能力，简化对处于自身线程控制中的对象的同步访问。Proxy代表Active Object的接口，以便服务调用在客户端线程中执行。Servant对象在自己的线程中执行请求，它代表所谓的Active部分。
Adapted Iterator	Adapted Iterator模式（参见第6章）通过Object Adapter来表达Iterator。
Adapter	Adapter模式[GoF95]将一个类的接口转换为客户端期望的接口。由于其变体有根本上的不同，Adaptation可以认为是一组互补的模式，而不是一个模式[NoBr02]。它们包括Class Adapter、Decorator、Façade、Object Adapter、Pluggable Adapter、Two-Way Adapter和Wrapper Façade。
Align Architecture and Organization	参见Conway's Law。
Application Controller	Application Controller模式[Fow02][POSA4]将用户界面导航与应用的工作流控制和编排相分离。Application Controller从应用的用户界面接收服务请求，根据工作流的当前状态决定调用哪个具体服务的功能和向用户界面展示哪个视图作为响应。

Architect Also Implements	Architect Also Implements模式[CoHa04]确保架构师不会脱离他们的设计决策所带来的后果和现实中的实践。不要把架构师雪藏在象牙塔里面，让他们参与日常的开发实现。
Architecture Follows Organization	Architecture Follows Organization模式（参见第5章）推荐将架构的结构向开发组织的结构靠齐，以减轻由于二者不匹配所带来的问题。
Asynchronous Completion Token (ACT)	Asynchronous Completion Token模式[POSA2] [POSA4]允许事件驱动的软件高效地分离和处理它有效调用服务上的异步操作时所产生的响应。
Automated Garbage Collection	Automated Garbage Collection模式[POSA4]为回收由不再使用的对象所占用的内存提供了安全而简单的机制。如果内存中的对象不在被应用中任何活跃的对象引用，就可以自动回收。
Batch Iterator	Batch Iterator模式（参见第6章）通过Batch Method实现了一个Iterator，完成一次遍历多个元素。
Batch Method	Batch Method模式[Hen01c][POSA4]将对聚合对象上的多次重复访问合并在一起，以减少多次单独访问的开销。
Blackboard	Blackboard模式[POSA1][POSA4]适用于那些没有确定解决方案策略的问题。几个特化（specialized）的子系统可以通过Blackboard集合它们的知识以构建一个不完全或者近似的解决方案。
Bridge	Bridge模式[GoF95][POSA4]将抽象与实现解耦合，以便二者可以独立地变化。它将一个对象划分为代表抽象的句柄和包含实现的主体。
Broker	Broker模式[POSA1][POSA4][VKZ04]支撑分布式软件系统，其组件通过远程方法调用进行交互。多个Broker联合起来管理组件间跨进程通信的关键方面，从请求转发到结果和异常传输。
Build Prototype	Build Prototype模式[CoHa04]用于降低创建需求未经验证的产品的风险。通过创建产品的原型，我们可以收集和澄清需求，更好地估计风险和范围。
Builder	Builder模式[GoF95][POSA4]将复杂对象的创建和销毁与其表现隔离开，以便同样的创建和销毁过程可以创建和删除不同的表现。
Bureaucracy	Bureaucracy模式[Rie98]帮助实现对象或者组件的层级结构，它允许与该层级结构中的任何一层进行交互，同时保持其内部的一致性。

Business Delegate	Business Delegate模式[ACM03][POSA4]将与远程组件访问相关的基础设施方面的考虑进行封装，比如查找（lookup）、负载均衡和网络错误处理，使得使用这些组件的客户端不必关心这些细节。Business Delegate使得在分布式应用中调用这些组件时位置透明。
Cantrip	Cantrip模式[Ray04]定义了一个非交互式的程序，它没有输入也不产生任何输出，只是一个状态结果。Cantrip只是作为一个简单的独立命令被调用。
Chain of Responsibility	Chain of Responsibility模式[GoF95][POSA4]避免将请求的发送者和它的接收者耦合在一起，而是将处理请求的机会交给多个对象。接收对象串联在一起，请求沿着这个对象链传递直到某个对象处理该请求。
Class Adapter	Class Adapter模式[GoF95]将一个对象的接口转换为客户端所期望的另一个对象的接口。适配使得那些原来由于接口不兼容而不能协作的类可以在一起工作。Class Adapter通过对Adaptee的实现进行子类化，并实现适配的接口——有可能也属于子类化——从而产生一个对象，避免额外的间接层。
CLI Server	CLI Server模式[Ray04]刻画了一种程序，当它在前台执行的时候，它提供一个基于标准输入输出流的简单的命令行界面；当它在后台运行的时候，其输入和输出在TCP/IP端口上执行。
Client Proxy	Client Proxy模式[POSA1][POSA4][GoF95][VKZ04]在客户端与远程组件交互的时候，提供一致的编程接口。客户端以一种独立于位置的方式访问远程组件，就像它们与客户端部署在一起一样。
Collections for States	Collections for States模式[Hen99a][POSA4]通过将每个关注的状态与一个单独的引用所有处于那个状态的对象的集合关联起来，从而将对象的状态外在化（externalize）。状态转换变成了集合之间的传递。
Combined Method	Combined Method模式[Hen01c][POSA4]将多个经常一起使用的方法组合成一个方法以保证正确性，并提高多线程和分布式环境中的效率。
Command	Command模式[GoF95][POSA4]将请求封装为对象，从而使得客户端用不同的请求作为参数，并支持可撤销的对象。
Command Processor	Command Processor模式[POSA1][POSA4]将服务的请求和执行分离开。Command Processor使用单独的Command对象管理请求，调度其执行，并提供额外的服务，比如日志和请求对象存储以支持

	日后请求的撤销和重做。
Command Sequence	参见Composite Command。
Community of Trust	Community of Trust模式[CoHa04]提供了一种组织框架，在该框架中一个团队中的人可以彼此信任以达成共同的目标。特别地，所有的事物和活动都以显式的方式展示给团队中的每个人，这说明它们都是为着同一个目标工作，而不是为自己工作。
Compiler	Compiler模式[Ray04]代表非交互式的程序，它将文件从一种格式转化为另一种格式。它们可能会在标准错误流上产生消息，但除此之外不接收其他的输入，也不产生其他的输出。
Completion Headroom	Completion Headroom模式[CoHa04]使用工作队列报告中对剩余工作量的估计来帮助估计完成的日期。统计每个贡献者的最早可能完成日期，找出其中最迟的一个，将其与项目的固定交付日期相比较。其中的差距即为Completion Headroom。
Component Configurator	Component Configurator模式[POSA2] [POSA4]允许应用在运行时加载和卸载其组件的实现，而无需修改、重编译或者静态链接该应用。它也支持对组件进行重新配置来构成不同的应用进程，而无需关闭和重启运行中的进程。
Composite	Composite模式[GoF95][POSA4]为表现有相似类型对象组成的“整体-部分”层次结构定义了一种划分。代表单独对象的类和代表组合对象的类实现共同的接口，以便客户端可以统一地对待它们。
Composite Command	Composite Command模式（参见第6章）将请求封装成一个对象，将单个请求和多个请求的区别隐藏在统一的接口后面。
Composite-Strategy-Observer	参见Model-View-Controller。
Context Object	Context Object模式[ACM03][Kel04][KSS05][Hen05b][POSA4]以组件对象格式描述环境服务和信息，该格式被传给需要访问相关执行上下文的插件组件对象或服务。
Conway's Law	Conway's Law模式[Cope95][CoHa04]推荐将架构和开发组织结构对齐，以消除由于它们的不匹配带来的问题。具体实现包括Organization Follows Architecture、Architecture Follows Organization或者将二者结合使用。
Cooperate, Don't Control	Cooperate, Don't Control模式[ORei05]建议Web应用通过Web Service或者内容的企业联合组织化实现开放，同时也应当构建在

	别人提供的对等服务上面。
CORBA-CGI Gateway	CORBA-CGI Gateway 模式 [MM97] 在 CORBA 对象模型和非 CORBA 的 Web 前端之间通过使用引入一个网关。然而，该模式的范围不必局限于 CORBA，同样的原理可以应用在很多其他跨越不同技术交互上面，这时简单的 Web 接口可能无法处理。
Data Access Object (DAO)	Data Access Object 模式 [ACM03] 支持将持久化数据的访问和操作封装在一个独立于应用逻辑的层中。
Data is the Next Intel Inside	Data is the Next Intel Inside 模式 [ORei05][MuOr+07] 关注于独特的数据而不是功能，并将其作为 Web 应用的商业优势。
Data Transfer Object (DTO)	Data Transfer Object 模式 [ACM03][Fow02][POSA4] 通过将一组属性打包成一个简单对象，在一次调用中传入或者返回，来减少对远程组件对象更新或者查询调用的次数。
Decorator	Decorator 模式 [GoF95][POSA4] 支持动态地将额外的行为添加到某个对象上面。为了在实例这个层次上适配和扩展已有的类，Decorator 比子类化提供了一个更为灵活的选择。
Disposal Method	Disposal Method 模式 [Hen02b][POSA4] 将对象清理的细节封装起来，提供一个用于销毁或者结束一个对象的方法，而不是交给客户端去销毁对象本身或者交给垃圾回收器处理。
Distributed Callback	Distributed Callback 模式 [MM97] 通过在 CORBA 环境中为回调引入 Explicit Interface 来获得异步所带来的好处。在 CORBA 环境中，每个针对接口的回调方法在 IDL 中定义为 <i>oneway</i> 操作。CORBA 中 <i>oneway</i> 操作没有很好的定义，因此该模式对于通过更恰当的 Message 机制来表现异步的情形并不适用，比如异步方法调用或者所谓的“可靠的” <i>oneway</i> 操作。
Domain Appropriate Devices	Domain Appropriate Devices 模式 [Bor01] 通过使用与交互系统应用领域中真实对象相似的输入设备来简化人机交互。
Domain Model	Domain Model 模式 [Fow02][POSA4] 为某个应用领域（包括其变体）的结构和工作流定义了简洁的模型。模型中的元素是在应用领域中有意义的抽象，它们的角色和交互反映领域工作流，并且能够映射到系统的需求。
Domain Object	Domain Object 模式 [POSA4] 封装了一个自我完备的内聚的功能性或者基础的职责，封装得到的良好定义的实体通过一个或者多个 Explicit Interface 来提供其功能，同时又将其内部结构和实现隐藏起来。

Domain Store	Domain Store模式[ACM03]从核心领域对象模型中分离出持久化的部分，以支持对象模型的透明持久化。持久化机制对于对象模型是无侵入的。
Don't Flip The Bozo Bit	Don't Flip The Bozo Bit模式[McC96]通过使团队中每一个成员都来为与团队成功完成项目相关的任何主题和事情贡献自己的想法，以帮助提高团队的沟通能力。
Dynamic Invocation Interface (DII)	Dynamic Invocation Interface模式[POSA4]为静态类型接口提供了一个补充接口。DII允许客户端以更为动态的方式调用对象上的方法，在运行时组成调用，而不是在静态声明中选择它们。
ed	ed模式[Ray04]定义了一个具备简单交互模型的程序。它们可以解释标准输入中的简单命令语言，因而是可脚本化的。
Encapsulated Context	参见Context Object。
Engage Customers	Engage Customers模式[CoHa04]帮助开发组织确保和维护客户满意度，鼓励客户与关键开发角色（比如开发人员和架构师）进行沟通。
Enumeration Method	Enumeration Method模式[Beck96][Hen01c][POSA4]将针对某个聚合组件的迭代封装起来，针对组件的每个元素所执行的动作封装成该聚合的一个方法。这个动作由调用者以一个对象的方式传入。
Explicit Interface	Explicit Interface模式[POSA4]将组件的使用与实现细节相分离。客户端只依赖于组件接口所定义的契约，而不依赖于组件的内部设计、具体实现、位置、同步机制和其他的实现细节。
External Iterator	参见Iterator。
Facade	Facade模式[GoF95][POSA4]为子系统中一系列接口提供了一个统一的高层接口，以方便子系统的使用。
Factory Method	Factory Method模式[GoF95][POSA4]通过为对象创建提供一个方法来封装具体的对象创建细节，而不是将具体类的实例化留给客户端去完成。
Few Panes Per Window	Few Panes Per Window模式[BeCu87]确保窗口应用的用户不会被复杂的窗口所淹没。
Filter	Filter模式[Ray04]定义了一个非交互式的程序或者程序的执行，该程序从标准输入获得数据，将其转换成某种式样，并将结果发送给标准输出。
Firewall Proxy	Firewall Proxy模式[SFHBS06][POSA1][POSA4]通过为服务引入一

	个代理，检查服务请求中的攻击负载以检测和阻止可疑内容，保护应用免受外部攻击。
Flat And Narrow Tree	Flat And Narrow Tree模式[Bor01]通过以不超过5层深度和每个节点不超过7条分支的树状层级结构来组织系统内容，从而避免在交互式系统的用户界面上出现庞大的信息层级结构。
Forwarder-Receiver	Forwarder-Receiver模式[POSA1]为P2P交互模型软件系统提供了一种透明的进程间交互。它将每个对点与底层的通信机制解耦合。
Front Controller	Front Controller模式[Fow02][POSA4]建立了一个应用的单一入口点——Front Controller——它将通过其用户界面发出的服务请求进行统一的处理和执行。
Half-Sync/Half-Async	Half-Sync/Half-Async模式[POSA2][POSA4]将并发系统中的同步和异步服务处理解耦合，以简化编程而又不过度地影响性能。该模式引入了两个交互的层，一个用于处理同步服务，另一个用于处理异步服务。
Harnessing Collective Intelligence	Harnessing Collective Intelligence模式[ORei05][MuOr+07]确保在Web应用的设计中“参与架构”是开放的，这样用户可以通过添加数据为应用增加价值。
Immutable Value	Immutable Value模式[Hen00][POSA4]通过对外只提供查询私有状态的方法，在构造时设置值对象的内部状态而不允许之后更改。Immutable Value可以在并发程序中的线程间共享，而无需额外的同步。
Information Just In Time	Information Just In Time模式[Bor01]通过在交互式系统中提供用法指导，以避免用户迷失。这些指导推迟到用户需要了解的时刻提供，而且应该尽量简短。
Interceptor	Interceptor模式[POSA2][POSA4]允许以透明的方式将与事件相关的处理配置到框架中，当特定事件出现的时候，自动触发该处理。
Internal Iterator	参见Enumeration Method。
Interpreter	Interpreter模式[GoF95][POSA4]为简单的语言定义了一个解释器，语法被建模成对象，并使得语法的表现可以直接执行。Context Object用来提供调用的状态。
Invisible Hardware	Invisible Hardware模式[Bor01]将交互式系统中计算机相关的硬件尽量隐藏。用户只能看到用于生成交互式系统中与应用领域相关图像的设备。

Involve Everyone	Involve Everyone模式[MaRi04]通过确保每个人都有机会作出贡献和支持，使得新的想法在组织内得以传播。
Iterator	Iterator模式[GoF95][POSA4]提供了一种方式来顺序访问一个聚合组件的元素，而无需公开其底层的表现。该模式通过一个独立的、可控的Iterator对象来遍历聚合组件。
Layers	Layers模式[POSA1][POSA4]帮助建立一种应用结构，它把应用分为几组子任务，每一组子任务代表一个特定层次的抽象、粒度、硬件距离 (hardware-distance)、开发变化率等。
Leader/Followers	Leader/Followers模式[POSA2][POSA4]提供了一种高效的并发模型，其中多个线程轮流共享一套事件源，以检测、分离、分派和处理出现在事件源上的服务请求。
Leveraging the Long Tail	Leveraging the Long Tail模式[ORei05][MuOr+07]指出Web（当然包括了众多网站和应用）的价值大部分在于缝隙市场（niches），而不仅是位于“中心”的明显的应用。
Macro Command	参见Composite Command。
Manager	参见Object Manager。
Mediator	Mediator模式[GoF95][POSA4]使Mediator自己作为唯一具有其他对象中方法的细节的对象，从而将一组对象解耦合。对象之间通过Mediator通信，而不是互相直接引用。
Memento	Memento模式[GoF95][POSA4]捕捉和外化（externalize）对象的内部状态，而不违反其内部封装。这种外化的状态可以在后面用于重置对象的状态。
Message	Message模式[HoWo03][POSA4]将两个应用对象交换的信息封装到一个数据结构里面，使其可以在网络上传递。
Methods for States	Methods for States模式[Hen02c][POSA4]将对象的所有行为实现在一个类里面，而不是多个类里面。使用几组方法的引用来定义对象在某个模式下的行为。
Mock Object	Mock Object模式[MFP00][Beck03]用于与外部依赖交互的代码的单元测试。Mock对象可以为单元测试用例模拟某种行为，并允许通过对Mock对象的检查了解被测代码是如何使用它的。
Model-View-Controller	Model-View-Controller模式[POSA1][POSA4][Fow02]将交互式应用分为三种角色。Model包含核心功能和数据。View向用户展示信息。Controller处理用户输入。View和Controller合在一起构成用户

	界面。通过某种变更传播机制保证用户界面和Model的一致性。
Monitor Object	Monitor Object模式[POSA2][POSA4]同步并发方法的执行，确保在一个对象内同一时间只有一个方法在运行。它也允许一个对象的多个方法通过协作的方式调度它们的执行序列。
Mutable Companion	Mutable Companion模式[Hen00]支持通过逐步修改状态的方式创建状态不变的值对象。
Network Effects by Default	Network Effects by Default模式[ORei05]建议不要依赖于用户向应用添加值，而是设定默认值并将用户数据作为用户使用应用的额外结果。
Nouns and Verbs	Nouns and Verbs模式[BeCu87]定义了一个用户界面模型，用于处理需要持久化的元素和暂态元素之间的平衡。事物的列表——所谓的Nouns——使用列表窗格提供，在交互期间持久化。而操作——所谓的Verbs——通过菜单的形式提供，弹出然后在开始执行操作时消失。
Null Object	Null Object模式[And96][Woolf97][Hen02a][POSA4]将对象不存在的情况进行封装，提供一种替代品来实现合适的默认的“什么也不做”的行为。
Object Adapter	Object Adapter模式[GoF95][POSA4]将一个类的接口转换为客户端期望的另一个接口。适配使得那些原本接口不兼容的类可以在一起工作。用对象关系来表现包装，确保这种适配是被封装的。
Object Manager	Object Manager模式[POSA3][POSA4]将对象的使用和对象的管理隔离开，从而支持显式的、集中式的、高效的组件、对象和资源处理。
Objects for States	Objects for States模式[GoF95][DyAn97][POSA4]将对象划分为两个部分：依赖于对象状态的行为和普通实例数据的表现。持有实例数据的对象将方法调用转交给状态对象，这些状态对象是某个类的层级结构的实例，该层级结构代表某个特定状态下的对象行为。
Observer	Observer模式[GoF95][POSA4]通过从Subject对象到其Observer的单向传播来同步相互协作的组件对象的状态。当Subject对象的状态发生变化时，其Observer可以被通知。
Organization Follows Architecture	Organization Follows Architecture模式（参见第5章）推荐将开发组织的结构向软件架构的结构靠齐，以减轻由于二者不匹配带来的问题。

Page Controller	Page Controller模式[Fow02][POSA4]为基于表单的用户界面中的每个表单引入了一个入口点，以统一通过每个表单发出的服务请求的处理和执行。
Perpetual Beta	Perpetual Beta模式[ORei05][MuOr+07]将应用看做演进中的服务，而不是静态的软件产品。应用的特性是一步一步递增添加上去的，而不是一下子把很多功能打包发布。
Pipes and Filters	Pipes and Filters模式[POSA1][POSA4][HoWo03]为处理数据流的系统提供了一种结构。每个处理步骤封装在一个Filter组件中。Pipe用于在相邻的Filter之间传递数据。
Pluggable Adapter	Pluggable Adapter模式[GoF95]实现了一种灵活的适配形式，通过Strategy来对Adapter进行参数化。
Pluggable Factory	Pluggable Factory 模式 [Vlis98c][Vlis99] 提供了一种可配置的Factory。使用Prototype实例或者某种形式的Strategy参数化单个产品类型的创建。
Polyvalent-Program	Polyvalent-Program模式[Ray04]描述了一种程序，其架构允许它们有各种不同的接口，从编程API到命令行和GUI，从Cantrip到Roguelike。
Presentation-Abstraction-Control	Presentation-Abstraction-Control 模式 [POSA1][POSA4] 以协作的Agent构成层级结构的形式为交互式软件系统定义了一个结构。每个Agent负责应用功能的一个特定方面，并由三部分组成——Presentation、Abstraction和Control。这种划分方式将Agent的人机交互方面与其功能核心以及与其他Agent的交互隔离开来。
Proactor	Proactor模式[POSA2][POSA4]允许事件驱动的软件高效地分离和分派由异步操作完成所触发的服务请求，因而能够获得并发所带来的好处而又不会引入过多的问题。事件是异步处理的，而“完成”(completion)是在应用的控制线程中处理的。
Prototype	Prototype模式[GoF95]使用原型实例来指定要创建的对象的类型，并且允许通过复制其原型来创建新的对象。
Prototype-Abstract Factory	参见Pluggable Factory。
Proxy	Proxy模式[POSA1][POSA4][GoF95]允许客户端以透明的方式通过代理与组件通信，而不是直接与组件通信。这里代理所起到的作用包括简化客户端编程，提高效率，避免非授权访问。

Publisher-Subscriber	Publisher-Subscriber模式[POSA4]为那些服务或者组件通过事件异步交换的方式进行交互的分布式软件系统建立一种一对多的结构。事件的Publisher和Subscriber通常互相并不知晓。Subscriber关心消费事件，而不需要知道其Publisher。同样，Publisher只提供事件，而并不关心也不知道谁订阅了这些事件。
Reactor	Reactor模式[POSA2][POSA4]允许事件驱动的软件分离和分派由一个或者多个客户端发送给应用的服务请求。一个事件处理基础设施同步地等待多个事件源，但是一次只分离和分派一个事件。
Reflection	Reflection模式[POSA1][POSA4]提供了一种机制，使得我们可以动态改变软件系统的结构和行为。它支持改变某些非常本质的方面，比如类型结构和函数调用机制。这个模式将应用分为两个部分。一个是基础层次，包括核心的应用逻辑。另一个是元层次，它观察基础层次的运行时行为，并维护有关选中系统属性的信息，以使得软件可以意识到其运行时上下文。由元层次保持的信息的修改可以影响并引导随后的基础层次的行为。
Remote Proxy	参见Client Proxy。
Resource Lifecycle Manager	参见Object Manager。
Roguelike	Roguelike模式[Ray04]定义了一种程序，它更加充分地使用视觉显示，并由按键驱动。它们属于交互式程序，但不容易脚本化。
Separated Engine and Interface	Separated Engine and Interface模式[Ray04]为程序定义了一个架构，将持有应用领域核心逻辑的Engine与Interface部分隔离开，后者负责表现逻辑和用户界面。Engine和Interface通常实现在不同的进程中。
Short Menus	Short Menus模式[BeCu87]确保弹出菜单长度较短、条目固定，并且保持一层，这样保证菜单项容易搜索。
Singleton	Singleton模式[GoF95]确保一个类只有一个实例，并且为其提供一个全局的访问点。对这个模式的滥用和模糊历时已久。它往往扮演着全局变量的角色，而没有真正起到约束创建的作用。而且基于Singleton的设计很难演进，因为它们往往与某个特定的上下文紧密耦合，这一点与全局变量非常相似。
Sink	Sink模式[Ray04]描述了一种非交互式的程序，它只从标准输入接收数据而不产生任何输出。

Smart Pointer	C++中的Smart Pointer模式[Mey96][Hen05a]通过一个间接层帮助控制对对象的访问。一个类定义一个常规的指针操作，比如operator*和operator->，这样对目标对象的访问就是受管理的。
Software Above the Level of a Single Device	Software Above the Level of a Single Device模式[ORei05][MuOr+07]通过确保Web应用在设计的时候将集成和多目标机(而不是单一的PC)纳入考虑范围，来增加其价值。
Some Rights Reserved	Some Rights Reserved模式[ORei05]鼓励在网站上使用已有的标准和非限制性的许可，以方便试验、大量采用 (collective adoption)、聚合 (aggregation) 和混入 (mixing)。
Source	Source模式[Ray04]描述了一种非交互式程序，它只产生数据并输出到标准输出流上而不接收任何输入。
Stable Intermediate Forms	Stable Intermediate Forms模式[Hen04a]用于减少流程变化时所引入的风险。在事物从一个状态转换为另一个状态的过程中，如果这个过程是不可以原子化的，那么它不可避免地会引入多个步骤，而每个步骤都可能因为这样或那样的原因失败，从而使得转换过程不完整或者出现不确定的情形。我们可以通过确保转换过程中的每个中间步骤表现为一个内聚的状态来降低这方面的风险，这个中间状态在某种意义上应该具备一定的完整性，而不是一种部分状态。
Standard Panes	Standard Panes模式[BeCu87]使得用户不必费力去学习如何操作各种类型的窗格。每个窗格都应该是标准窗格中的一种。
State	参见Object for States。
Strategized Locking	Strategized Locking模式[POSA2][POSA4]对一个组件进行参数化，确保用户可以选择最恰当的同步机制来对组件中的关键段进行序列化。
Strategy	Strategy模式[GoF95][POSA4]描述了一组一起变化的操作。每个变体封装在一个对象中，这些对象共享统一的接口。使用哪个可插入的对象是独立于这些变体的实现的。
Template Method	Template Method模式[GoF95][POSA4]定义了某个操作的算法结构，将某些步骤推迟到子类中。这个模式允许子类重新定义算法中的特定步骤，而不需要修改算法的结构。
Template View	Template View模式[Fow02][POSA4]引入了一个模板，为如何将动态产生的数据展示给用户预先定义了一个总体的结构。Template

原
书
缺
页



原
书
缺
页



原
书
缺
页



原
书
缺
页



原
书
缺
页



原
书
缺
页



原
书
缺
页



原
书
缺
页



- Illinois, USA, September 8–12, 2003
- [Drö00] W. Dröschel: *Das V-Modell 97*, Oldenbourg, 2000
- [DyAn97] P. Dyson, B. Anderson: *State Patterns*, in [PLoPD3], 1997
- [EBNF96] EBNF: *Information technology — Syntactic metalanguage — Extended BNF*, ISO/IEC 14977, 1996
- [ECOOP97] M. Aksit, S. Matsuoka (eds.): *ECOOP '97 – Object-Oriented Programming*, Proceedings of Eleventh European Conference on Object-Oriented Programming, Jyväskylä, Finland, June 1997, Lecture Notes in Computer Science 1241, Springer, 1997
- [ECOOP98] E. Jul (ed.): *ECOOP '98 – Object-Oriented Programming*, Proceedings of the Twelfth European Conference on Object-Oriented Programming, Brussels, Belgium, July 1998, Lecture Notes in Computer Science 1445, Springer, 1998
- [ECOOP99] R. Guerraoui (ed.): *ECOOP '99 – Object-Oriented Programming*, Proceedings of the Thirteenth European Conference on Object-Oriented Programming, Lisbon, Portugal, June 1998, Lecture Notes in Computer Science 1628, Springer, 1999
- [ECOOP02] B. Magnusson (ed.): *ECOOP '02 – Object-Oriented Programming*, Proceedings of the Sixteenth European Conference on Object-Oriented Programming, Málaga, Spain, July 2002, Lecture Notes in Computer Science 2374, Springer, 2002
- [EGHY99] A. H. Eden, J. Gil, Y. Hirshfeld, A. Yehudai: *Motifs in Object Oriented Architecture*, IEEE Transactions on Software Engineering, 1999
- [EGKM+01] S.G. Eick, T.L. Graves, E.F. Karr, J.S. Marron, A. Mockus: *Does Code Decay? Assessing the Evidence from Change Management Data*, IEEE Transactions on Software Engineering, Volume 27, No. 1, pp 1–12, January, 2001
- [Evans03] E. Evans: *Domain-Driven Design*, Addison-Wesley, 2003
- [FBBOR99] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999
- [FeRu98] L. Ferreira, C. Rubira: *Reflective Design Patterns to Implement Fault Tolerance*, Proceedings of OOPSLA '98 Workshop on Reflective Programming in C++ and Java, Vancouver, Canada, October 18, 1998
- [FJS99a] M. Fayad, R. Johnson, D.C. Schmidt (eds.): *Implementing Application Frameworks: Object-Oriented Frameworks at Work*, John Wiley & Sons, New York, 1999

- [FJS99b] M. Fayad, R. Johnson, D.C. Schmidt (eds.): *Implementing Application Frameworks: Object-Oriented Frameworks at Work*, John Wiley & Sons, New York, 1999
- [Fow97] M. Fowler: *Analysis Patterns*, Addison-Wesley, 1997
- [Fow02] M. Fowler: *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002
- [FoYo97] B. Foote, J. Yoder: *Big Ball of Mud*, in [PLoPD4]
- [Fra03] D. Frankel: *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons, 2003
- [Fri06] T.L. Friedman: *The World is Flat: A Brief History of the Twenty-First Century*, expanded and updated version, Farrar, Straus and Giroux, 2006
- [Gab96] R.P. Gabriel: *Patterns of Software: Tales from the Software Community*, Oxford University Press, 1996
- [Gab02] R.P. Gabriel: *Writers' Workshops and the Work of Making Things*, Addison-Wesley, 2002,
<http://www.dreamsongs.com/Files/WritersWorkshopTypeset.pdf>
- [Gam92] E. Gamma: *Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliotheken, Werkzeuge*, Springer, 1992
- [Gam95] E. Gamma: *personal communication*, 1995
- [GCKL05a] A. Garcia, C. Chavez, U. Kulesza, C. Lucena: *The Interaction Aspect Pattern*, Proceedings of the Tenth European Conference on Pattern Languages of Programs (EuroPLoP 2005), Irsee, Universitätsverlag Konstanz, July 2006
- [GCKL05b] A. Garcia, C. Chavez, U. Kulesza, C. Lucena: *The Role Aspect Pattern*, Proceedings of the Tenth European Conference on Pattern Languages of Programs (EuroPLoP 2005), Irsee, Universitätsverlag Konstanz, July 2006
- [GoF95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
- [GrMü05] D. Grolimund, P. Müller: *A Pattern Language for Overlay Systems*, Proceedings of the Eleventh European Conference on Pattern Languages of Programs (EuroPLoP 2006), Irsee, Universitätsverlag Konstanz, July 2007
- [GSCK04] J. Greenfield, K. Short, S. Cook, S. Kent: *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, John Wiley & Sons, 2004

- [GSF+05] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. von Staa: *Modularizing Design Patterns with Aspects: a Quantitative Study*, Proceedings of the Fourth International Conference on Aspect-Oriented Software Development, Chicago, pp. 3–14, March 14–18 2005
- [GSS03] S. Garfinkel, G. Spafford, A. Schwartz: *Practical UNIX and Internet Security*, Third Edition, O'Reilly and Associates, February 2003
- [GTK+07] J. Gray, J.P. Tolvanen, S. Kelly, A. Gokhale, S. Neema, J. Sprinkle: *Domain-Specific Modeling*, in *Handbook of Dynamic System Modeling*, P. Fishwick (ed.), CRC Press, December 2007
- [Gun04] B. Venners and B. Eckel: *A Conversation with E. Gunnerson: Insights into the .NET Architecture*, February 2004,
<http://www.artima.com/intv/dotnet.html>
- [Haa03] A. Haase: *Java Idioms: Exception Handling*, Proceedings of the Seventh European Conference on Pattern Languages of Programming (EuroPLoP 2002), Irsee, Universitätsverlag Konstanz, July 2003
- [HaKi02] J. Hannemann, G. Kiczales: *Design Pattern Implementation in Java and AspectJ*, Proceedings of the ACM OOPSLA 2002 Conference, Seattle, pp. 161–173, November 2002
- [HaKo04] R.S. Hanmer, K.F. Kocan: *Documenting Architectures with Patterns*, Bell Labs Technical Journal, Volume 9, No. 1, pp. 143–163, 2004
- [Han04] R.S. Hanmer: *Detection Techniques for Fault Tolerance*, Proceedings of the Eleventh Conference on Pattern Languages of Program Design, Monticello, Illinois, September 2004
- [Han07] R.S. Hanmer: *Patterns For Fault Tolerant Software*, John Wiley & Sons, 2007
- [Har99] N.B. Harrison: *The Language of Shepherding*, in [PLoPD5]
- [Hat98] L. Hatton: *Does OO Sync with the Way We Think?*, IEEE Software, 15(3), 1998
- [HCSH05] L.B. Hvatum, A. Crotia, T. Simien, D. Heliot: *Patterns and Advice for Managing Distributed Product Development Teams*, Proceedings of the Tenth European Conference on Pattern Languages of Programs (EuroPLoP 2005), Irsee, Universitätsverlag Konstanz, July 2006
- [Hearsay01] Kloster Hearsay (the daily EuroPLoP newspaper), Issue 02/2001: *Close to Heresy*, Irsee, Germany, 2001
- [Hearsay02] Kloster Hearsay (the daily EuroPLoP newspaper), Issue 02/2002, Joe Bergin: *Do the Right Thing*, Irsee, Germany, 2002

- [Hed97] G. Hedin: *Language Support for Design Patterns Using Attribute Extensions*, Proceedings of the Workshop on Language Support for Design Patterns and Frameworks, June, 1997
- [Hen97] K. Henney: *Java Patterns and Implementations*, BCS OOPS Pattern Day, October 1997, London
- [Hen99a] K. Henney: *Collections for States*, Proceedings of the Fourth European Conference on Pattern Languages of Programming (EuroPloP 1999), Irsee, Universitätsverlag Konstanz, July 2001
- [Hen99b] K. Henney: *Substitutability: Principles, Idioms and Techniques for C++*, JaCC Conference, 1999
- [Hen00] K. Henney: *Value Added*, Java Report 5(4), April 2000
- [Hen01a] K. Henney: *C++ Patterns – Executing Around Sequences*, Proceedings of the Fifth European Conference on Pattern Languages of Programming (EuroPloP 2000), Irsee, Universitätsverlag Konstanz, July 2001
- [Hen01b] K. Henney: *C++ Patterns – Reference Accounting*, Proceedings of the Sixth European Conference on Pattern Languages of Programming (EuroPloP 2001), Irsee, Universitätsverlag Konstanz, July 2002
- [Hen01c] K. Henney: *A Tale of Three Patterns*, Java Report, SIGS Publications, October 2001
- [Hen02a] K. Henney: *Null Object*, Proceedings of the Seventh European Conference on Pattern Languages of Programming (EuroPloP 2002), Irsee, Universitätsverlag Konstanz, July 2003
- [Hen02b] K. Henney: *Patterns in Java: The Importance of Symmetry*, JavaSpektrum, Issue 6, 2002, SIGS-DATACOM GmbH, Germany
- [Hen02c] K. Henney: *Methods for States*, Proceedings of the First Nordic Conference on Pattern Languages of Programming, VikingPLoP 2002, Helsingør, Denmark Universitätsverlag Konstanz, July 2003
- [Hen03a] K. Henney: *The Good, the Bad, and the Koyaanisqatsi*, Proceedings of the Second Nordic Pattern Languages of Programs Conference, VikingPLoP 2003, 2003
- [Hen03b] K. Henney: *Beyond Metaphor*, endnote of the Eighth Conference on Java and Object-Oriented Technology, JAOO 2003, Aarhus, Denmark, 2003
- [Hen04a] K. Henney: *Stable Intermediate Forms*, Proceedings of the Ninth European Conference on Pattern Languages of Programming (EuroPloP 2004), Irsee, Universitätsverlag Konstanz, July 2005

- [Hen04b] K. Henney: *Relative Values — Defining Comparison Methods for Value Objects in Java*, Proceedings of the Third Nordic Pattern Languages of Programs Conference, VikingPLoP 2004, 2004
- [Hen05a] K. Henney: *STL Patterns: A Design Language of Generic Programming*, SD West 2005
- [Hen05b] K. Henney: *Context Encapsulation – Three Stories, A Language, and Some Sequences*, Proceedings of the Tenth European Conference on Pattern Languages of Programming (EuroPLoP 2005), Irsee, Universitätsverlag Konstanz, July 2006
- [HJE95] H. Hueni, R. Johnson, R. Engel: *A Framework for Network Protocol Software*, Proceedings of the ACM OOPSLA 1995 Conference, Austin, Texas, October 1995
- [HKG04] W. Herzner, W. Kubinger, M. Gruber: *Triple-T (Time-Triggered-Transmission)*, Proceedings of the Ninth European Conference on Pattern Languages of Programming (EuroPLoP 2004), Irsee, Universitätsverlag Konstanz, July 2005
- [HLS97] T. Harrison, D. Levine, D.C. Schmidt: *The Design and Performance of a Real-Time CORBA Event Service*, Proceedings of OOPSLA '97, ACM, Atlanta, GA, October 6–7 1997
- [Hoare85] C.A.R. Hoare: *Communicating Sequential Processes*, Prentice Hall, 1985
- [HoJi99] C.A.R. Hoare, H. Jifeng: *A Trace Model for Pointers and Objects*, in [ECOOP99], 1999
- [Hon04] S. Honiden: *A Pattern Oriented Mobile Agent Framework for Mobile Computing*, First International Workshop on Mobility Aware Technologies and Applications (MATA2004), pp. 369–380, Florianópolis, Brazil, October 20–22 2004
- [Hor06] C.S. Horstmann: *Object Oriented Design and Patterns*, John Wiley & Sons, 2006
- [HoWo03] G. Hohpe, B. Woolf: *Enterprise Integration Patterns – Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley, 2003
- [HvKe05] L. Hvatum, A. Kelly: *What Do I Think about Conway's Law Now?*, Proceedings of the Tenth European Conference on Pattern Languages of Programming (EuroPLoP 2005), Irsee, Universitätsverlag Konstanz, July 2006
- [IsDe96] N. Islam, M. Devarakonda: *An Essential Design Pattern for Fault-Tolerant Distributed State Sharing*, Communications of the ACM, Volume 39, No. 10, pp. 65–74, October 1996

- [Jac95] M. Jackson: *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*, Addison-Wesley, 1995
- [Jac01] M. Jackson: *Problem Frames: Analyzing and Structuring Software Development Problems*, Addison-Wesley, 2001
- [JeWi74] K. Jensen, N. Wirth: *Pascal: User Manual and Report*, Springer, 1974
- [JGJ97] I. Jacobsen, M. Griss, P. Johnsson: *Software Reuse: Architecture, Process, And Organization for Business Success*, ACM Press, 1997
- [John97] R. Johnson: *Frameworks = Patterns + Components*, Communications of the ACM, Volume 40, No. 10, October 1997
- [JRV00] M. Jazayeri, A.C.M. Ran, F. van der Linden: *Software Architecture for Product Families*, Addison-Wesley, 2000
- [Kaye03] D. Kaye: *Loosely Coupled – The Missing Pieces of Web Services*, Rds Associates, 2003
- [KC97] W. Keller, J. Coldewey: *Accessing Relational Databases*, in [PLoPD3], 1997
- [Kel99] W. Keller: *Object/Relational Access Layer*, in Proceedings of the Third European Conference on Pattern Languages of Programming (EuroPloP 1998), Irsee, Universitätsverlag Konstanz, July 1999
- [Kel04] A. Kelly: *Encapsulated Context*, Proceedings of the Eighth European Conference on Pattern Languages of Programming (EuroPloP 2003), Irsee, Universitätsverlag Konstanz, July 2004
- [Ker95] N. Kerth: *Caterpillar's Fate: A Pattern Language for the Transformation from Analysis to Design*, in [PLoPD1], 1995
- [Ker04] J. Kerievsky: *Refactoring to Patterns*, Addison-Wesley, 2004
- [KGSHR06] A.S. Krishna, A. Gokhale, D.C. Schmidt, J. Hatcliff, V.P. Ranganat: *Context-Specific Middleware Specialization Techniques for Optimizing Software Product-Line Architectures*, Proceedings of EuroSys 2006, Leuven, Belgium, April 18–21, 2006
- [KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin: *Aspect-Oriented Programming*, in [ECOOP97], 1997
- [KLSB03] G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty: *Model-Integrated Development of Embedded Software*, Proceedings of the IEEE, Volume 91, No. 1, pp.145–164, January 2003
- [Koe95] A. Koenig: *Patterns and Antipatterns*, Journal of Object-Oriented

- Programming, 8(1), 1995
- [Kon05] S. Konrad: *Assurance Patterns for Distributed Real-Time Embedded Systems*, Proceedings of the Twenty-Seventh international Conference on Software Engineering, St. Louis, pp. 657–657, ICSE '05, May 15–21, 2005
- [Kru95] P. Krutchen: *Architectural Blueprints: The '4+1' View Model of Software Architecture*, IEEE Software 12(6), pp. 42–52, 1995
- [Kru00] P. Kruchten: *The Rational Unified Process: An Introduction*, Second Edition, Addison-Wesley, 2000
- [KSS05] A. Krishna, D.C. Schmidt, M. Stal: *Context Object: A Design Pattern for Efficient Middleware Request Processing*, Proceedings of the Twelfth Pattern Language of Programming Conference, Allerton Park, Illinois, September 2005
- [KTB98] R.K. Keller, J. Tessier, G. von Bochmann: *A Pattern System for Network Management Interfaces*, Communications of the ACM, Volume 41, No. 9, pp. 86–93, September 1998
- [KVSJ04] M. Kircher, M. Voelter, C. Schwanninger, K. Jank: *Broker Revisited*, Proceedings of the Ninth European Conference on Pattern Languages of Programming (EuroPloP 2004), Irsee, Universitätsverlag Konstanz, July 2005
- [Ladd03] R. Laddad: *AspectJ in Action*, Practical Aspect-Oriented Programming, Manning Publications, 2003
- [Lau04] Y. Lau: *Service-Oriented Architecture and the C4ISR Framework*, CrossTalk: The Journal of Defense Software, September 2004
- [LBG97] K. Lano, J. Bicarregui, and S Goldsack: *Formalising Design Patterns*, in First BCS-FACS Northern Formal Methods Workshop, Electronic Workshops in Computer Science, Springer, 1997
- [LC87] M.A. Linton, P.R. Calder: *The Design and Implementation of InterViews*, Proceedings of the First USENIX C++ Workshop, November 1987
- [Lea00] D. Lea: *Concurrent Programming in Java, Design Principles and Patterns*, 2nd Edition, Addison-Wesley, 2000
- [LK98] A. Lauder, S. Kent: *Precise Visual Specification of Design Patterns*, in [ECOOP98], 1998
- [LoWo04] A. Longshaw, E. Woods: *Patterns for Generation, Handling and Management of Errors*, Proceedings of the Ninth European Conference on Pattern Languages of Programming (EuroPloP 2004), Irsee, Universitätsverlag Konstanz, July 2005

- [LRP04] T. Lewis, M. Rosson, M. Pérez-Quiñones: *What Do the Experts Say?: Teaching Introductory Design from an Expert's Perspective*, Proceedings of the Thirty-Fifth SIGCSE Technical Symposium on Computer Science Education, Norfolk, Virginia, USA, pp. 296-300, SIGCSE '04, ACM Press, New York, March 03-07, 2004
- [LS04] S. Lukosch, T. Schümmer: *Patterns for Managing Shared Objects in Groupware Systems*, Proceedings of the Ninth European Conference on Pattern Languages of Programming (EuroPloP 2004), Irsee, Universitätsverlag Konstanz, July 2005
- [LS05] S. Lukosch, T. Schümmer: *Patterns for Session Management in Groupware Systems*, Proceedings of the Tenth European Conference on Pattern Languages of Programming (EuroPloP 2005), Irsee, Universitätsverlag Konstanz, July 2006
- [MaLu95] D. Manolescu, B. Lublinsky: *Orchestration Patterns in SOA*, <http://orchestrationpatterns.com>
- [Mar02a] K. Marquardt: *Patterns for the Treatment of System Dependencies*, Proceedings of the Seventh European Conference on Pattern Languages of Programming (EuroPloP 2002), Irsee, Universitätsverlag Konstanz, July 2003
- [Mar02b] K. Marquardt: *Diagnoses from Software Organizations*, Proceedings of the Seventh European Conference on Pattern Languages of Programming (EuroPloP 2002), Irsee, Universitätsverlag Konstanz, July 2003
- [Mar03] K. Marquardt: *Permititis*, Proceedings of the Eighth European Conference on Pattern Languages of Programming (EuroPloP 2003), Irsee, Universitätsverlag Konstanz, July 2004
- [Mar04a] K. Marquardt: *Ignored Architecture, Ignored Architect*, Proceedings of the Ninth European Conference on Pattern Languages of Programming (EuroPloP 2004), Irsee, Universitätsverlag Konstanz, July 2005
- [Mar04b] K. Marquardt: *Platonic Schizophrenia*, Proceedings of the Ninth European Conference on Pattern Languages of Programming (EuroPloP 2004), Irsee, Universitätsverlag Konstanz, July 2005
- [Mar05] K. Marquardt: *Indecisive Generality*, Proceedings of the Tenth European Conference on Pattern Languages of Programming (EuroPloP 2005), Irsee, Universitätsverlag Konstanz, July 2006
- [MaRi04] M.L. Manns, L. Rising: *Fearless Change: Patterns for Introducing New Ideas*, Addison-Wesley, 2004
- [McC96] J. McCarthy: *Dynamics of Software Development*, Redmond, Washington, USA, Microsoft Press, 1995

- [MD97] G. Meszaros, J. Doble: *A Pattern Language for Pattern Writing*, in [PLoPD3], 1997
- [Mead60] M. Mead: *Cultural Patterns and Technical Change*, New American Library, 1960
- [Mes96] G. Meszaros: *A Pattern Language for Improving the Capacity of Reactive Systems*, in [PLoPD2], 1996
- [Mes01] G. Meszaros: *Introduction to Patterns and Pattern Languages*, presented at the Workshop on Patterns for Distributed, Real-Time, Embedded Systems at OOPSLA 2001, Tampa, Florida, 2001
- [Mes07] G. Meszaros: *XUnit Test Patterns: Refactoring Test Code*, Addison-Wesley, 2007
- [Met04] S.J. Metsker: *Design Patterns in C#*, Addison-Wesley, 2004
- [Mey88] B. Meyer: *Object-Oriented Software Construction*, First Edition, Prentice Hall, 1988
- [Mey96] S. Meyers: *More Effective C++*, Addison-Wesley, 1996
- [Mey97] B. Meyer: *Object-Oriented Software Construction*, Second Edition, Prentice Hall, 1997
- [MFP00] T. Mackinnon, S. Freeman, P. Craig: *Endo-Testing: Unit Testing with Mock Objects*, Proceedings of the XP 2000 Conference, 2000
- [Mik98] T. Mikkonen: *Formalizing Design Patterns*, Proceedings of the Twentieth International Conference on Software Engineering, pp. 115–124, IEEE Computer Society Press, 1998
- [MK97] A. Mester, H. Krumm: *Formal Behavioural Patterns for the Tool-Assisted Design of Distributed Applications*, IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS 97), Cottbus, Germany, September–October 1997, Chapman & Hall, 1997
- [MM97] R.C. Malveau, T.J. Mowbray: *CORBA Design Patterns*, John Wiley & Sons, 1997
- [MoFe06] P. Morrison, E.B. Fernandez: *Securing the Broker Pattern*, Writing Workshop paper at the Eleventh European Conference on Pattern Languages of Programs (EuroPloP 2006), Irsee, July 2006
- [MS03] Microsoft Corporation: *Enterprise Solution Patterns using Microsoft .NET*, Microsoft Press, 2003

- [MuOr+07] J. Musser, T. O'Reilly, O'Reilly Radar Team: *Web 2.0 Principles and Best Practices*, O'Reilly, 2007
- [Nash06] T. Nash: *Accelerated C#*, Apress, 2006
- [NFG+06] L. Northrop, P. Feiler, R. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, K. Wallnau: *Ultra-Large Scale Systems: The Software Challenge of the Future*, The Software Engineering Institute, 2006, <http://www.sei.cmu.edu/uls/>
- [Niel06] J. Nilsson: *Applying Domain-Driven Design and Patterns: with Examples in C# and .NET*, Addison-Wesley, 2006
- [NoBr02] J. Noble, R. Biddle: *Patterns as Signs*, in [ECOOP02], 2002
- [NOF92] D. Flanagan (ed.): *X Toolkit Intrinsics Reference Manual: Volume 5*, O'Reilly, 1992
- [NoWe05] J. Noble, C. Weir: *Amethodology*, Proceedings of the Tenth European Conference on Pattern Languages of Programs (EuroPloP 2005), Irsee, Universitätsverlag Konstanz, July 2006
- [OMG03] Object Management Group: *Unified Modeling Language: OCL Version 2.0*, Final Adopted Specification, OMG Document ptc/03-10-14, October 2003
- [OMG04a] Object Management Group: *Common Object Request Broker Architecture*, Version 3.0.3, March 2004
- [Opd92] W. F. Opdyke: *Refactoring Object-Oriented Frameworks*, Ph.D. Thesis, University of Illinois at Urbana Champaign, 1992
- [ORei05] T. O'Reilly: *What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software*, September 2005, <http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html>
- [Ort05] J.L. Ortego-Aronja: *The Pipes and Filters Pattern. a Functional Parallelism Architectural Pattern for Parallel Programming*, Proceedings of the Tenth European Conference on Pattern Languages of Programming (EuroPloP 2005), Irsee, Universitätsverlag Konstanz, July 2006
- [PaCle86] D.L. Parnas, P.C. Clements: *A Rational Design Process: How and Why to Fake It*, IEEE Transactions on Software Engineering 12(2), pp. 251–257, 1986
- [Par94] D.L. Parnas: *Software Aging*, Proceedings of the Sixteenth International Conference on Software Engineering (ICSE-16), Sorrento, Italy, May 1994
- [Par06] The Parlay Group: *Parlay X Web Services Specification*, Version 2.0, 2006

- [PB01] D.S. Platt, K. Ballinger: *Introducing Microsoft .NET*, Microsoft Press, 2001
- [PCW05] R. Porter, J.O. Coplien, T. Winn: *Sequences as a Basis for Pattern Language Composition*, Science of Computer Programming, Elsevier, 2005
- [Pet05] N. Pettersson: *Measuring Precision for Static and Dynamic Design Pattern Recognition as a Function of Coverage*, Proceedings of the Third International Workshop on Dynamic Analysis (WODA), ACM Press, pp. 1–7, St. Louis, May 2005
- [PK98] L. Prechelt, C. Krämer: *Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns*, Journal of Universal Computer Science (J.UCS), 4(11), pp. 866–882, December 1998
- [PLoPD1] J.O. Coplien, D.C. Schmidt (eds.): *Pattern Languages of Program Design*, Addison-Wesley, 1995 (a book publishing the reviewed Proceedings of the First International Conference on Pattern Languages of Programming, Monticello, Illinois, 1994)
- [PLoPD2] J.O. Coplien, N. Kerth, J. Vlissides (eds.): *Pattern Languages of Program Design 2*, Addison-Wesley, 1996 (a book publishing the reviewed Proceedings of the Second International Conference on Pattern Languages of Programming, Monticello, Illinois, 1995)
- [PLoPD3] R.C. Martin, D. Riehle, F. Buschmann (eds.): *Pattern Languages of Program Design 3*, Addison-Wesley, 1997 (a book publishing selected papers from the Third International Conference on Pattern Languages of Programming, Monticello, Illinois, USA, 1996, the First European Conference on Pattern Languages of Programming, Irsee, Bavaria, Germany, 1996, and the Telecommunication Pattern Workshop at OOPSLA '96, San Jose, California, USA, 1996)
- [PLoPD4] N.B. Harrison, B. Foote, H. Rohnert (eds.): *Pattern Languages of Program Design 4*, Addison-Wesley, 1999 (a book publishing selected papers from the Fourth and Fifth International Conference on Pattern Languages of Programming, Monticello, Illinois, USA, 1997 and 1998, and the Second and Third European Conference on Pattern Languages of Programming, Irsee, Bavaria, Germany, 1997 and 1998)
- [PLoPD5] D. Manolescu, M. Völter, J. Noble (eds.): *Pattern Languages of Program Design 5*, Addison-Wesley, 2006 (a book publishing selected papers from the Pattern Languages of Programming conference series from 1999–2004)
- [PoCa04] R. Porter and P. Calder: *Patterns in Learning to Program: an Experiment*, Proceedings of the Sixth Conference on Australasian Computing Education, Volume 30, Dunedin, New Zealand, R. Lister, A. Young (eds.), ACM International Conference Proceeding Series, Volume 57, pp. 241–246,

- Australian Computer Society, Darlinghurst, Australia, 2004
- [POS05] I. Pyarali, C. O’Ryan, D.C. Schmidt: *A Pattern Language for Efficient, Predictable, Scalable, and Flexible Dispatching Components*, in [PLoPD5]
- [POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, John Wiley & Sons, 1996
- [POSA2] D.C. Schmidt, M. Stal, H. Rohnert, F. Buschmann: *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*, John Wiley & Sons, 2000
- [POSA3] P. Jain, M. Kircher: *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*, John Wiley & Sons, 2004
- [POSA4] F. Buschmann, K. Henney, D.C. Schmidt: *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*, John Wiley and Sons, 2007
- [PP03] M. Poppendieck, T. Poppendieck: *Lean Software Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley, 2003
- [PPP] *Pedagogical Patterns Project*, <http://www.pedagogicalpatterns.org>
- [PPR] *The Portland Pattern Repository*, <http://www.c2.com>
- [Prec97] L. Prechelt: *An Experiment on the Usefulness of Design Patterns: Detailed Description and Evaluation*, Technical Report 9/1997, Universität Karlsruhe, Fakultät für Informatik, Germany, June 1997
- [PUS97] L. Prechelt, B. Unger, D.C. Schmidt: *Replication of the First Controlled Experiment on the Usefulness of Design Patterns: Detailed Description and Evaluation*, Technical Report wucs-97-34, Department of Computer Science, Washington University, St. Louis, December 1997
- [Ray91] E.S. Raymond (ed.): *The New Hacker’s Dictionary*, MIT Press, 1991
- [Ray04] E.S. Raymond: *The Art of Unix Programming*, Addison-Wesley, 2004
- [RBGM00] D. Riehle, R. Brudermann, T. Gross, K.U. Mätsel: *Pattern Density and Role Modeling of an Object Transport Service*, ACM Computing Surveys, Volume 32, Issue 1es, Article No. 10, 2000
- [RBV99] M. Robinson, P.A. Vorobiev: *Swing*, Manning Publications Company, 1999
- [ReRi85] S. Redwine, W. Riddle: *Software Technology Maturation*, Proceedings of the Eighth International Conference on Software Engineering, pp. 189–200, May 1985

- [RG98] D. Riehle, T. Gross: *Role Model Based Framework Design and Integration*, Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98), pp. 117–133, ACM Press, 1998.
- [Rie97] D. Riehle: *Composite Design Patterns*, Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97), pp. 218–228, ACM Press, October 1997
- [Rie98] D. Riehle: *Bureaucracy*, in [PLoPD3], 1997
- [Ris98] L. Rising (ed.): *The Patterns Handbook*, SIGS Reference Library No. 13, Cambridge University Press, 1998
- [Ris01] L. Rising: *Design Patterns in Communications Software*, SIGS Reference Library No. 19, Cambridge University Press, 2001
- [RJB99] J. Rumbaugh, I. Jacobsen, G. Booch: *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999
- [Rüp03] A. Rüping: *Agile Documentation: A Pattern Guide to Producing Lightweight Documents for Software Projects*, John Wiley & Sons, 2003
- [RWL96] T. Reenskaug, P. Wold, O.A. Lehne: *Working with Objects: The OOram Software Engineering Method*, Manning Publications Company, 1996
- [Sari02] T. Saridakis: *A System of Patterns for Fault Tolerance*, Proceedings of the Seventh European Conference on Pattern Languages of Programs (EuroPloP 2002), Irsee, Universitätsverlag Konstanz, July 2003
- [Sari03] T. Saridakis: *Design Patterns for Fault Containment*, Proceedings of the Eighth European Conference on Pattern Languages of Programs (EuroPloP 2003), Irsee, Universitätsverlag Konstanz, July 2004
- [Sari05] T. Saridakis: *Design Patterns for Graceful Degradation*, Proceedings of the Tenth European Conference on Pattern Languages of Programs (EuroPloP 2005), Irsee, Universitätsverlag Konstanz, July 2006
- [SB01] K. Schwaber, M. Beedle: *Agile Software Development with SCRUM*, Prentice Hall, 2001
- [SB03] D.C. Schmidt, F. Buschmann: *Patterns, Frameworks, and Middleware: Their Synergistic Relationships*, Proceedings of the Twenty-Fifth IEEE/ACM International Conference on Software Engineering (ICSE), Portland, Oregon, pp. 694–704, May, 2003
- [SBM+06] R. Sangwan, M. Bass, N. Mullick, D.J. Paulish, J. Kazmeier: *Global Software Development Handbook*, Auerbach, 2006

- [SC99] D.C. Schmidt, C. Cleland: *Applying Patterns to Develop Extensible ORB Middleware*, IEEE Communications Magazine, special issue on Design Patterns, April 1999
- [SC00] D.C. Schmidt, C. Cleland: *Applying a Pattern Language to Develop Extensible ORB Middleware*, in [Ris01]
- [Sch86] K.J. Schmucker: *Object-Oriented Programming for the Macintosh*, Hayden Books, 1986
- [Sch95] D.C. Schmidt: *Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software*, Communications of the ACM (Special Issue on Object-Oriented Experiences), ACM, Volume 38, No. 10, pp. 65–74, October 1995
- [Sch98] D.C. Schmidt: *Applying a Pattern Language to Develop Application-Level Gateways*, in [Ris01], 1998
- [Sch03] T. Schümmel: *GAMA: A Pattern Language for Computer-Supported Dynamic Collaboration*, Proceedings of the Eighth European Conference on Pattern Languages of Programming (EuroPloP 2003), Irsee, Universitätsverlag Konstanz, July 2004
- [Sch04a] A. Schmidmeier: *Patterns and an Antiidiom for Aspect-Oriented Programming*, Proceedings of the Ninth European Conference on Pattern Languages of Programs (EuroPloP 2004), Irsee, Universitätsverlag Konstanz, July 2005
- [Sch04b] T. Schümmel: *Patterns for Building Communities in Collaborative Systems*, Proceedings of the Ninth European Conference on Pattern Languages of Programming (EuroPloP 2004), Irsee, Universitätsverlag Konstanz, July 2005
- [Sch06a] D.C. Schmidt: *Model-Driven Engineering*, IEEE Computer, Volume 39, No. 2, pp. 41–47, February 2006
- [Sch06b] A. Schmidmeier: *Configurable Aspects*, Proceedings of the Eleventh European Conference on Pattern Languages of Programs (EuroPloP 2006), Irsee, Universitätsverlag Konstanz, July 2007
- [ScHa05] A. Schmidmeier, S. Hanenberg: *Cooperating Aspects*, Proceedings of the Tenth European Conference on Pattern Languages of Programs (EuroPloP 2005), Irsee, Universitätsverlag Konstanz, July 2006
- [SCJS99] K.J. Sullivan, P. Chalasani, S. Jha, V. Sazawal: *Software Design as an Investment Activity: A Real Options Perspective*, in *Real Options and Business Strategy: Applications to Decision Making*, L. Trigeorgis (ed.), Risk Books, 1999

- [SFHBS06] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, P. Sommerlad: *Security Patterns: Integrating Security and Systems Engineering*, John Wiley & Sons, 2006
- [SFM05] T. Schümmmer, A. Fernandez, M. Myller: *Patterns for Virtual Places*, Proceedings of the Tenth European Conference on Pattern Languages of Programming (EuroPloP 2005), Irsee, Universitätsverlag Konstanz, July 2006
- [SGCH01] K.J. Sullivan, W.G. Griswold, Y. Cai, B. Hallen: *The Structure and Value of Modularity in Software Design*, Proceedings of the Eighth European Software Engineering Conference, pp. 99–108, 2001
- [SGM02] C. Szyperski, D. Gruntz, S. Murer: *Component Software: Beyond Object-Oriented Programming*, Second Edition, Addison-Wesley, 2002
- [SH02] D.C. Schmidt, S.D. Huston: *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*, Addison-Wesley, 2002
- [SH03] D.C. Schmidt, S.D. Huston: *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*, Addison-Wesley, 2003
- [Shir] C. Shirky: *Moderation Strategies*,
<http://social.itp.nyu.edu/shirky/wiki/?n>Main.PatternLanguage>
- [SL06] T. Schümmmer, S. Lukosch: *READ.ME - Talking About Computer-Mediated Communication*, Proceedings of the Tenth European Conference on Pattern Languages of Programming (EuroPloP 2005), Irsee, Universitätsverlag Konstanz, July 2006
- [SN96] R.W. Schulte, Y.V. Natis: *Service Oriented Architectures*, Part 1, SSA Research Note SPA-401-068, Gartner, 12 April 1996
- [SNL05] C. Steel, R. Nagappan, R. Lai: *Core Security Patterns: Best Practices and Strategies for J2EE Web Services, and Identity Management*, Prentice Hall, 2005
- [SPM99] A. Silva, J. Pereira, J. Marques: *Object Recovery Pattern*, in [PLoPD3]
- [Stal03] M. Stal: *Reverse Engineering Architectures using Patterns*, tutorial notes, Seventeenth European Conference on Object-Oriented Programming (ECOOP 2003), Darmstadt, Germany, July 2003
- [StFl04] S. Stuurman, G. Florijn: *Experiences with Teaching Design Patterns*, Proceedings of the Ninth Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '04), Leeds, UK, pp. 151–155, June 28–30 2004, ACM Press, 2004

- [StLe95] A. Stepanov, M. Lee: *The Standard Template Library*, Technical Report HPL-95-11 (R.1), Hewlett-Packard Laboratories, 1995
- [Sun97] Sun Microsystems: *Java Beans*, version 1.01, 1997
<http://www.java.sun.com>
- [SVC06] T. Stahl, M. Völter, K. Czarnecki: *Model-Driven Software Development: Technology, Engineering, Management*, John Wiley & Sons, 2006
- [Tal94] Telligent Inc.: *Telligent's Guide To Designing Programs – Well-Mannered Object-Oriented Design in C++*, Addison-Wesley, 1994
- [Thu81] Thucydides: *The Peloponnesian War*, Translation by R. Crawley, Random House, 1981
- [Tolk04] J.R.R. Tolkien: *The Lord of the Rings*, Fiftieth Anniversary Edition, Houghton Mifflin, 2004
- [TZP05] W. Tsai, L. Yu, F. Zhu, R. Paul: *Rapid Embedded System Testing Using Verification Patterns*, IEEE Software, Volume 22, No. 4, pp. 68–75, July–August 2005
- [Utas05] G. Utas: *Robust Communications Software: Extreme Availability, Reliability and Scalability for Carrier-Grade Systems*, John Wiley & Sons, 2005
- [VeSa06] Z. Velart, P. Šaloun: *User Behavior Patterns in the Course of Programming in C++*, Proceedings of the Joint International Workshop on Adaptivity, Personalization, and the Semantic Web (APS '06), Odense, Denmark, August 23–23, 2006, pp. 41–44, ACM Press, 2006
- [VK04] M. Völter, M. Kircher: *Command Revisited*, Proceedings of the Ninth European Conference on Pattern Languages of Programming (EuroPLoP 2004), Irsee, Universitätsverlag Konstanz, July 2005
- [VKZ04] M. Völter, M. Kircher, U. Zdun: *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*, John Wiley & Sons, 2004
- [ViHe99] J. Vlissides, R. Helm: *Compounding Command*, C++ Report, April 1999
- [Vlis96] J. Vlissides: *Generation Gap*, C++ Report, November–December 1996
- [Vlis98a] J. Vlissides: *Composite Design Patterns (They Aren't What You Think)*, C++ Report, June 1998
- [Vlis98b] J. Vlissides: *Pattern Hatching: Design Patterns Applied*, Addison-Wesley, 1998
- [Vlis98c] J. Vlissides: *Pluggable Factory*, Part I, C++ Report, November/December 1998

- [Vlis99] J. Vlissides: *Pluggable Factory*, Part II, C++ Report, February 1999
- [Vö05a] M. Völter: *Patterns for Handling Cross-Cutting Concerns in Model-Driven Software Development*, Proceedings of the Tenth European Conference on Pattern Languages of Programs (EuroPloP 2005), Irsee, Universitätsverlag Konstanz, July 2006
- [Vö05b] M. Völter: *The Role of Patterns in Modern Software Engineering*, Presentation at the Tenth Conference on Java and Object-Oriented Technology, JAoO 2005, Aarhus, Denmark, 2005
- [Vö06] M. Völter: *Software Architecture — A Pattern Language for Building Sustainable Software Architectures*, Proceedings of the Eleventh European Conference on Pattern Languages of Programs (EuroPloP 2006), Irsee, Universitätsverlag Konstanz, July 2007
- [VöBe04] M. Völter, J. Bettin: *Patterns for Model-Driven Software Development*, Proceedings of the Ninth European Conference on Pattern Languages of Programming (EuroPloP 2004), Irsee, Universitätsverlag Konstanz, July 2005
- [VSW02] M. Völter, A. Schmid, E. Wolff: *Server Component Patterns: Component Infrastructures Illustrated with EJB*, John Wiley & Sons, 2002
- [Wake95] W.C. Wake: *Account Number: A Pattern*, in [PLoPD1]
- [WaKl98] J.B. Warmer, A.G. Kleppe: *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1998
- [Wel05] T. Wellhausen: *User Interface Design for Searching – A Pattern Language*, Proceedings of the Tenth European Conference on Pattern Languages of Programming (EuroPloP 2005), Irsee, Universitätsverlag Konstanz, July 2006
- [Wiki] Wikipedia: <http://wikipedia.org/>
- [Woolf97] B. Woolf: *Null Object*, in [PLoPD3], 1997
- [XBHJ00a] S. Xia, A. Black, J. Hook, M. Jones: *JPat, A Pattern Definition Language*, Technical Report of Department of Computer Science and Engineering, Oregon Graduate Institute, 2000
- [XBHJ00b] S. Xia, A. Black, J. Hook, M. Jones: *Describing Patterns Using Index Types*, ECOOP '01 submission, 2000
- [YaAm03] S.M. Yacoub, H.H. Ammar: *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems*, Addison-Wesley, 2003
- [Zach] J.A. Zachman: *Zachman Framework*, The Zachman Institute for Framework Advancement, <http://www.zifa.com>

- [Zdun04] U. Zdun: *Some Patterns of Component and Language Integration*, Proceedings of the Ninth European Conference on Pattern Languages of Programming (EuroPloP 2004), Irsee, Universitätsverlag Konstanz, July 2005



[General Information]

书名 = 面向模式的软件架构 第 5 卷 模式与模式语言

作者 = (德) 布施曼等著

丛书名 = 图灵程序设计丛书

页数 = 262

SS号 = 12865986

出版日期 = 2011.09

出版社 = 人民邮电出版社

尺寸 = 26 cm

原书定价 = 59

参考文献格式 = (德) 布施曼, (英) 亨尼, (美) 施密特著. 面向模式的软件架构 卷 5

模式与模式语言. 北京市: 人民邮电出版社, 2011.09.

内容提要 = 本书共分三部分, 首先介绍了独立模式, 详细阐述了过去累积的关于如何描述和应用模式的诸多见解, 接着探究了模式之间的关系, 从组织的角度说明了各个模式的领域, 最后介绍了如何将模式和模式语言相结合。