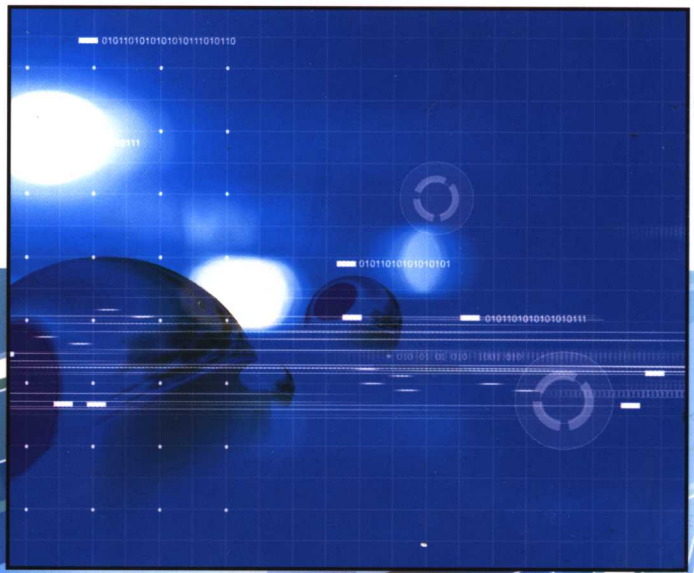
PEARSON
Addison
Wesley

程序设计的模式语言·卷3

Pattern Languages of Program Design 3



Robert Martin
Dirk Riehle
Frank Buschmann
著

朱志博 等 译

清华大学出版社

程序设计的模式语言·卷3

Robert Martin
Dirk Riehle 著
Frank Buschmann
朱志博 等译

清华大学出版社
北京

图书在版编目(CIP)数据

程序设计的模式语言·卷3/(美)马丁(Martin, R.), (美)里尔(Riehle, D.), (美)布希曼(Buschmann, F.)著；朱志博等译. —北京：清华大学出版社，2005.1

书名原文：Pattern Languages of Program Design 3

ISBN 7-302-09935-9

I. 程… II. ①马… ②里… ③布… ④朱… III. 程序设计语言学 IV. TP311.1

中国版本图书馆 CIP 数据核字(2004)第 120388 号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦

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

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

责任编辑：常晓波

封面设计：立日新

印 刷 者：北京四季青印刷厂

装 订 者：北京鑫海金澳胶印有限公司

发 行 者：新华书店总店北京发行所

开　　本：185×230　**印张**：32.25　**字数**：712 千字

版　　次：2005 年 1 月第 1 版　2005 年 1 月第1 次印刷

书　　号：ISBN 7-302-09935-9/TP·6825

印　　数：1～3000

定　　价：64.80 元

简介　杂交优势和雪地中的脚印

我们希望，许多人阅读、使用这种语言，并尝试着改善这些模式——愿意费精力来寻找更真实的、更深刻的不变量——我们希望随着时间的推移，这些被发现的更真实的模式会逐渐地进入一个大家都可以共享的共同语言中来。

Christopher Alexander 等编著，*A Pattern Language*，*xv*.

[Alexander +77]

新东西是这里没有任何新的东西　模式是关于有用之道的，模式给了我们一种讨论有用之道的方法。Stewart Brand 在他的著作 *How Buildings Learn* [*Brand94*] 中描述了一个广泛流传(有可能是假的)的故事，一个聪明但懒惰的学院设计师，她设计的校园没有人行道。在第一个冬天，她守候着拍下大楼之间人们在雪地里踏出的路。第二年春天，她在那里铺上人行道。模式与故事中的过程类似，与仓促、不成熟、很可能是错误地猜测有用之道相反，模式编写者在雪地里寻找脚印，然后描述出已经证明有效的东西。

谈论有用之道似乎显而易见。为什么以前它没有发生？为什么它现在发生了？学术界永远关注新鲜的事物。学者们像真正的猎奇者一样，贪婪地吸取新思想。这个现象在计算机科学领域尤为突出，诞生 4 个月的技术可以被认为是“传统的”技术。模式在另一方面，是在有用之道上传递的急件。模式描述的是重复的解决方案，如果很多人在很长时间里一直做着某样事情，那么它还能被称为新鲜事物吗？

由于模式来自于经验，所以本质上它们与发明和创造无关。模式编写者不是上帝，他们不会通过想象创建宇宙，但他们扮演亚当和夏娃的角色，他们为自己花园中的居民起名字，这不是一个能够轻松承担的责任。

Ralph Johnson 指出，在很多学科中，研究人员研究的对象都不是研究人员创造的物体。昆虫学家收集蝴蝶并对它们分类，然后讨论这些分类，思考它们为什么会存在。而生物学家并没有创造生命(至少到目前为止)。他们收集标本，进行观察，发现种类，不用担心会被人指责剽窃了大自然。类似地，模式编写者不仅可以扮演亚当和夏娃的角色，也可扮演生物学家 Carolus Linnaeus。

计算机科学是一个非常年轻的领域，在不久前还是数学和电子工程的一个继子。当第一代的计算机科学家们尝试着像数学家和物理学家一样工作时，他们经常忽略掉自己学科的丰富内容就在脚下：经验能够从实际的程序员和实际的系统中收集。现在，他们能够借用建筑、生物、文学、通信甚至他们自己的知识和经验，这是成熟和自信的标志。

模式运动与20世纪90年代的文化时代精神相吻合。它是同Java一样的一种第二代现象，它是抽样年代中的创新。在万维网中，抽样已成为一种生活方式。在90年代，一些著名的音乐家大胆地借用他们前辈的东西，不仅仅引用短句，而是把利用数字抽样器获得的其他音乐家的真正表演片段嵌入自己的音乐中。但他们的艺术前辈们为了追求原创性，破坏了和谐和作曲的每一条规则。现在，这种对新领域的越来越不好的探索已经过去了。90年代的音乐家们通过忽略这种探索来解决了他们的原创性问题。事实上，他们的作品建立在过去的作品之上并结合以新方式继承的艺术遗产的现有元素。

从一开始，模式社团就强烈漠视创新，这一点与众不同。当人们说新音乐、新诗歌或艺术根本不是真正的音乐、诗歌和艺术时，这表明有一些新的东西发生了。人们争论模式是否是真正的计算机科学或研究。如果我们现在所作的不被认为是计算机科学，那它应该就是计算机科学。

当学术界执着于改变时，工业界当然乐意关注已被测试和被证明过的事物。然而有一个问题，为什么工业界人士要把他们对系统结构的见解交给他们的竞争对手？为什么要把老底都交出去？请人来编写模式而不是程序的好处在哪儿？这些都是正当的疑问，我们不希望PLoP成为一个秘密交易的演示会。

PLoP USA在Allerton House召开，这是个由雕塑和花园环绕着的百年庄园，坐落在Illinois乡下Sangamon河岸边的树林内。模式的朝圣者们穿过40英里的玉米和大豆地，到达Robert Allerton奇异建筑的纪念碑。在穿过这片世界上最肥沃的土地时，与会者脑海中不时蹦出“杂交优势”这个词。

大自然发明了异花授粉来迅速地在种群中传播成功的变异。农民和园艺家们后来发现他们可以操纵这个过程并创造杂交后代，抵消同系繁殖和增强农民和园艺家需要的特征。从远亲中引入的新的基因物质经常可以产生极为高产和健壮的新品种，这就被称为“杂交优势”。

PLoP将人们和模式从似乎非常不同的地方汇聚起来。它吸引了学术人员和实干家，管理人员和程序员，咨询专家和学生，甚至还有建筑承包商。为什么模式社团具有这么大的中和力量？可能是因为在看起来毫不相同的领域下面确实存在着相似的底层架构原则。也许真的存在着很好的思想，这种思想能够扩展、归纳和传播。

也许在未来六个月中，当我写程序的时候，需要使用一个缓存方案，而这个方案是从一个毫不相干的应用程序模式中得到的。这种杂交的潜力是工业界对模式感兴趣的一个原因。发生在PLoP的独一无二的异花授粉式的交流，为工业界不同行业以及学术界提供了快速、广泛地传播最佳实践经验的机会，可以让所有人接触到健康的杂交优势。因此研究使用模式的人不仅可以扮演亚当、夏娃或生物学家Linnaeus，还可以扮演大农艺家Luther Burbank。工业界可以从一本真正的关于软件架构的工程手册中获益。

模式一个光彩夺目的文化将会繁荣。不幸的是，软件的架构经常是隐藏的。模式社团为软件架构设计师们提供了他们以前从未有过的东西：观众。软件架构经常属于事后才产

生的想法。我们珍惜结果，哪怕它们来自于加班加点、仓促上阵的工程实践和毫无计划、乱七八糟的设计架构。很长时间里，软件建筑师们像中世纪的工匠们一样辛辛苦苦地精心雕琢着装饰怪兽的后背，虽然知道在远离地面的高处，他们的艺术只能被上帝所看到。现在，我们有了一个论坛，实干家、理论家、程序员和教授们可以齐聚一堂，褒奖优良，无名的软件设计者们的建筑和美学成就开始获得迟到但应有的承认。

语言、库和框架是软件架构最终表述的媒体。模式告诉我们其他的架构设计师如何成功地将这些部分组合在一起。通过将注意力集中到重复发生的问题上，模式可以驱动工具、过程、语言和框架的进化，使直接解决问题的可重用的产品成为可能。我们将能够使用现成的产品，并将精力集中于高层次的架构方面的问题。

一个系统的构架可能有好有坏，但无论如何必须有架构。软件架构正在作为一门学科出现。不管是鼓励底层的基础设施在更大范围内重用，还是帮助最新的快速原型开发环境中的组件获得成功，架构都是成熟的软件工程中劳动力划分的关键。

重新应用是一个关于信任的行为。探路的先锋在做标记的时候知道自己向左转是为了避免多走 200 英里的弯路还是为了简单地躲开落下来的树枝。跟在后面的人不会了解这些情况，他们只知道路标转向了左面。架构需要能够自我支持的思想。设计者需要能够传递意思的词汇表，不需要查阅百科全书或者每一个使用者的抽象的白盒知识。

Thomas Malthus 是一位 19 世纪的牧师和经济学家，他认真地预测：成功团体的发展只能持续到需求超过可获得的资源之前。1996 年的 PLoP 收到 70 多篇投稿和近 110 个注册登记——Malthus 博士的理论开始向我们靠近。我们开始感到承受能力有限，投稿的论文比以前更为中性。似乎这些力量会把我们的钻石按照自然的缝隙割裂开。事实上，这些割裂趋势已经开始出现。EuroPLoP、TelePLoP、ChiliPLoP、UP 已经加入了经典的 PLoP USA。有关模式的新书就像炎热夏日阳光下高产的杂交作物一样生长出来，模式的成功使它们变成了新学术研究的避雷针。

强大的力量已经在行动中，同时也把我们推向了行动。将今天有关模式设计的对话与几年前相比非常有趣。例如，Gang-of-Four 的书中[Gamma +95]使用的架构概念曾经被认为非常深奥，而现在已经成了常识，利用一个简单名字就可以提议、比较和抛弃它们。几年前的模式理论正在成为设计词典中的一部分，可能将伴随我们很多年。新的词汇表给说话人带来的力量和简洁程度是惊人的。与 Alexnnder 对建筑使用的语言相比，我们离拥有相同程度的软件架构的模式语言还有一长段距离。三年前，我们中还没有什么人知道所有前进的方向，现在我们都知道自己正在目睹一件重要和持久的事情。

只不过不到 100 年前，两兄弟还可以在一个自行车铺里造飞机。仅仅一代人之后，就没人能掌握所有二次世界大战前战斗机里使用的工程技术了。先驱们被成队拥有高度技巧和专门技术的协作者们所替代，他们一起定义了架构的词典和劳动力的分配，将工作分解后，再重新整合在一起。他们能够将工作分解的原因是可以辨别工程领域中暴露出的不同的架构元素以及元素之间的分界线。小组中有经验的设计师仍然可以改动部件，但这些部

件已经被抽象成较高的层次。

在很多方面，软件架构正在告别自行车作坊式的时代。Brad Cox[Cox95]认识到“硬件产业是一个完整而精细的雨林，各个生产树互相依赖，软件产业与之不同，还处在充满单细胞、微生物的原始泥沼阶段。”

今天，州际公路穿过早年西部马车队留下的车轮印。先驱们无论如何也不可能想到建造现代桥梁和隧道所需要的工程技术，他们也不会知道可以用来搬走比金字塔还大的整座石山、混凝土和沥青的机器。在几个钟头之内，我们就可以旅行他们需要几个月走过的距离。可是，这不是我们完全自己开拓的新路，而是我们通过寻找雪地中的脚印，然后在那里铺上道路的结果。

Brian Foote, Urbana, Illinois

参考文献

[Alexander+77] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language.* Oxford: Oxford University Press, 1977.

[Brand94] S. Brand. How Buildings Learn: *What Happens After They' re Built.* New York: Viking Press, 1994.

[Cox95] B. Cox. “No Silver Bullet Revisited.” *American Programmer Journal.* November, 1995.

[Gamma+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reus-able ObjectOriented Software.* Reading, MA: Addison-Wesley, 1995.

前　言

本书是 PLoPD 系列图书的第 3 本，内容和前两本有所不同。本书是 PLoPD 系列中第一本发表的论文少于半数的书。另外，也是第一本包含不同会议论文的 PLoPD 书籍。投稿到 PLoP'96 和 EuroPLoP'96 的论文超过 80 篇，我们不能全部发表。而决定哪些论文不能发表不是一项让人高兴的任务。因为所有递交的论文质量都很高——这也是我们对 PLoP 会议所期望的——这项任务一点也不轻松。幸运的是，我们的负担由帮助进行评审和选择程序的人们减轻了许多。

创建本书的过程　我们聘请了一组评审人员和 3 位评审人，他们阅读了 80 多篇论文中的每一篇。评审人的推荐交给 3 位编辑（Robert Martin、Dirk Riehle 和 Frank Buschmann）。然后我们开始了长时间的热烈讨论，我们中没有人害怕过分挑剔，事实上，3 个人在很大一部分用于发表的论文上达成了一致。但仍然有一些论文我们没有达成共同意见。我们进行了长时间和热烈的关于确定本书题材内容的讨论。我们不认为本书非常完美，但我们都相信它是一本拔尖的优秀论文集。

什么是我们的选择标准？论文的选择受我们的目标读者（软件工程师）的限制。首先，这部分读者必须对论文感兴趣，尽管有关音乐的模式对音乐家来说很有意思，但是我们不想在这里包括这些论文。其次，论文必须对读者具有实际价值，尽管有关抽象理论的文章非常有趣，我们仍然优先选择那些给我们的读者带来能够直接使用的技术和工具的论文。最后，论文应该包括模式。关于软件工程有很多优秀的文章，但我们更侧重于描述相关软件工程模式的论文。

当然这些标准并没有预先阐述清楚。像所有高质量的项目一样，需求总是随着开发而演化。就是在这个过程中，我们才互相了解到对这本书的期望和想象，而自己的期望和想象随着讨论和争辩不断地变化。总而言之，这是一次非常有益的，让人有些筋疲力尽的体验。

按照 Ralph Johnson 建议的精神，对模式编制目录像生物学中对动物和非动物物种分类设计一样，我们按照主题组织本书。全书包括通用设计模式和针对特殊技术和业务领域的模式，还包括了设计用户界面和帮助软件开发过程的模式，甚至还包括了一章关于编写模式的模式。我们没有区别模式和模式语言，而是将模式按照主题组织在一起，这样读者可以看看这些模式是否适合自己的需要和应用领域。

设计模式，1997 年的观点　GoF 一书出版到现在已经有两年了，在这段时间里，对设

计模式的兴趣以极大的速度增长。今天任何严肃的软件工程师几乎不可能不熟悉设计模式的概念。一些主要的杂志定期地刊登关于设计模式的专栏。*C++ Report* 杂志每月刊登关于设计模式的专门文章，已经出版了由权威作者撰写的其他几本书。所有这些都表明设计模式的概念在软件工程的发展中已经非常重要，而且这种重要性在未来的日子里会继续增加。

伴随着重要性和意识的增强，也同时带来了一个危险。我们很容易将对模式本身的热情转换到一般模式概念上。按照我们的观点，这将是一个错误。在一个给定项目中使用久经考验的设计模式会带来极大的好处。但是生拉硬拽地将很多不同的模式塞进一个项目中会带来恶劣的结果，比模式仅仅不发挥作用还要糟。因为一个存在的模式并不意味着它就应该被使用。模式适合项目是因为项目有问题需要解决，而模式提供了有用的解决方案策略。我们不能因为一个东西是模式就认为使用它是正确的。例如，工程师不能因为 GoF 书中包含了 Visitor 模式就随便地使用它判断。设计模式是工具，由真正理解应该在何时何地应用模式的工程师们使用。

你的武器在哪里 Dr. Who 是一部老的英国电视科幻系列片。在有一集中，博士护送一个硅人到他的飞船里。那个硅人环顾了周围，然后大叫起来："太好了！可是它的武器在哪里？"博士直视着那个硅人的眼睛，用手指着他的太阳穴说："这儿！"

最近，本书作者之一(Robert)在芝加哥一个关于设计模式的大型会议上发表讲话。他提问说谁买了 GoF 的书，大约有 80% 的人举起了手。然后他请没有真正读过这本书的人把手放下，大约有一半的手放下了。然后他又问谁可以解释一下 Visitor 模式，几乎所有的手都放下了。

这本书中的模式，或者任何其他 1995 年以来出版的优秀模式书籍中的模式，如果读者根本不去读并理解它们的话都不会对读者有帮助。有些读者可能想把各种模式的书作为一个解决问题的方案目录，等到问题出现时就去查找，但这并不是一个有效的方法。实际上，读者必须仔细研究模式，并将模式融入到进行软件设计的思维中去。这样，在设计软件的时候，甚至在还没有意识到设计问题之前，模式就可以自己跳出来。

请阅读模式吧，仔细地阅读它们，确保将解决软件设计问题的武器牢固地安置在自己的头脑中。

致谢

首先，非常感谢 Hillside 集团承办了 PLoP 会议并提供了出版这些书籍的动力。没有他们的努力和投入就不会有任何 PLoPD 的书籍。

然后，我们要感谢 Addison-Wesley 的朋友们对我们工作的帮助。特别感谢 Anne Starr 和她在 New England 编辑服务部门的同事们，他们以复杂的图书生产过程中的胜利者姿态出现。

感谢 Doug Schmidt 带给我们的智慧。感谢 Jim Coplien(Cope)提醒我们这项工作在精神和技术上的必要性。感谢 John Vlissides 及系列丛书编辑，感谢他们在我们困难的时候掌舵。感谢 Walter Tichy 使我们保持谦虚。最后，Bob 个人感谢 Dirk 和 Frank。

我们还要感谢所有向 PLoP '96 和 EuroPLoP'96 投稿的作者，感谢论文的指导人和帮助编辑作出困难选择的评审人员。他们的名字如下：

Alan O'Callaghan, Alejandra Garrido, Alistair Cockburn, Amiram Yehudai, Amnon H. Eden, Amund Aarsten, Andreas Rüping, António Rito Silva, Becky Fletcher, Benoit Garbinato, Bernd-Uwe Pagel, Bindu Rama Rao, Björn Eiderbäck, Bobby Woolf, Bran Selic, Brian Foote, Bruce Anderson, Bruce Lom-bardi, Charles D. Knutson, Charles Weir, Chris Cleeland, Chrystalla C. Alexandrou, Clazien Wezeman, Curtis R. Cook, D. Janaki Ram, D. Schwabe, Daniel A. Rawsthorne, Daniel Megert, David E. DeLano, Davide Brugali, Dirk Bäumer, Don Roberts, Douglas C. Schmidt, Edward J. Posnak, Elizabeth A. Kendall, Erich Gamma, Eugene Wallingford, Eyun Eli Jacobsen, Fernando das Neves, G. Rossi, George A. Papadopoulos, Gerard Meszaros, Giuseppe Menga, Hans Rohnert, Harrick M. Vin, Heinz Züllighoven, Ilir Kondo, Irfan Pyarali, James Noble, Jan Newmarch, Jean Tessier, Jean-Lin Pacherie, Jean-Marc Jézéquel, Jens Coldewey, Jim Doble, Jim Lee, João Pereira, John Brant, John Vlissides, John W. Gilbert, José Alves Marques, Joseph Gil, Joseph Yoder, K. N. Anantha Raman, K. N. Guruprasad, Ken Auer, Kent Beck, Kyle Brown, Lennart Ohlsson, Leonor Barroca, Linda Rising, Liping Zhao, Lizette Velázquez, Lorraine L. Boyd, Margaret T. Malkoun, Mario Winter, Mark Bradac, Martin E. Nordberg III, Martin Fowler, Michel de Champlain, Neil B. Harrison, Omer Karacan, Palle Nowack, Pascal Felber, Paul Dyson, Pedro Henriques, Peter H. Feiler, Peter Molin, Peter Sommerlad, Prashant Jain, R. Greg Lavender, R. J. A. Buhr, Rachid Guerraoui, Ralph Johnson.

Reinhard Müller, Robert Engel, Robert Hirschfeld, Robert S. Hanmer, Rudolph K. Keller, Serge Demeyer, Steve Berczuk, Suchitra Raman, Suzanne Robertson, Ted Foster, Theo Dirk Meijler, Thomas Kühne, Tim Harrison, Timothy A. Budd, Todd Coram, Walter F. Tichy, Wolf Siberski 和 Wolfgang Keller.

Robert C. Martin
Dirk Riehle
Frank Buschmann

目　录

第1部分

通用目的设计模式

本部分中的6种模式遵循主流软件开发人员最熟悉的Design Patterns[Gamma+95]风格。这些模式非常适合收集在一起。如果用户正在进行面向对象的开发，很可能会应用或者重新改造其中的一个或多个模式。

第1章：空对象(Null Object)，Bobby Woolf编写 如果读者已经长时间使用面向对象的语言编程，就会发现需要一个对象来填补实现漏洞，或者需要强制初始化实例变量(即使它在应用程序中永不被使用)，或者可能想避免用检验空值的条件避免重叠代码。解决这些难题的方案就是空对象模式。空对象完全用空操作实现了一个接口，所以，它什么也不做(do-nothing)——准确地说，就是做null条件测试通常所做的事情。但是，功能完整的对象不做任何事情时，空对象也不做任何事情，因而可以避免null测试。空对象也是另一个多态性如何简化和规范代码的例子——这是好的面向对象设计的一个特点。

第2章：管理者(Manager)，Peter Sommerlad编写 大多数程序员都赞成这个认识，类方法和变量应该尽量少用，因为它们带有许多全局函数和变量在过程性语言中出现的同样的错误，它们很难或者不可能无限度地扩展，而且容易产生使维护变复杂和接受敏感错误的负面影响。但是，需要一起控制类的实例时，类方法和变量又非常有吸引力。Manager模式用于满足这种不产生缺陷的需求。

Manager封装了讨论的对象，并将以前的类方法实现为普通的方法。该方法并不是没有缺陷，在特殊情况下，用户很可能会禁不住诱惑，而按照危及管理对象的封装应用这种模式。但是，应用时要牢记Peter的警告，Manager能显著地改善普通的类方法和变量。

第3章：产品交易商(Product Trader)，Dirk Bäumer和Dirk Riehle编写 这种模式最初被命名为"后期创建(Late Creation)"[Riehle96]，它告知用户如何创建不命名——特别是具体类名的对象。为什么想要这么做呢？目的是将类名绑定延迟到运行期完成。后期绑定实例化的一个经典的例子是根据流创建对象，它在双重继承层次结构[Martin97]中也是非常有用的，其中一个层次中的对象用另一个层次中的对象实例化。

然而，另一个应用程序为基于想要的运行结果的接口，使用模式选择接口的最佳实现。实际上，Product Trader 是按照结构、组织和开发的分界线划分面向对象系统的关键，因为它确保软件系统确实依赖于接口而不是依赖于实现［Gamma +95］。该模式不仅讨论了产品交易的概念，而且对如何按照表的查找顺序有效实现产品交易进行了讨论。除内层循环的最内层外，它对所有循环而言速度都相当快。

第4章：类型对象(Type Object)，Ralph Johnson 和 Bobby Woolf 编写 这种模式对如何用仅有的两个类 Type 和 Instance 取代一个完整的类层次结构进行描述。Type 类的一个实例取代了原层次结构中的每个类，同时一个 Instance 的实例取代了原类的每个实例。这样，类型对象模式产生了一个对象元模型，补记了类对象、实例和属性的完整模型。它的目标是在运行期引入新的对象类型。也就是说用户不需要随着类型需求(如新产品分类或旧产品分类的扩展)的改变而开发一个新的应用程序。而且，用户管理的类空间很大时，很可能需要使用这种模式，大多数类经常发生变化，但并不发生本质性的变化。

第5章：主办者—选择器(Sponsor-Selector)，Eugene Wallingford 编写 Strategy 模式［Gamma +95］允许用户简单地转换算法，即使是在运行期也可以。但是，它并没有指出转换的分枝或者是进行转换的标准，这也就是主办者—选择器出现的地方，这种模式确保客户在请求时接收到正确的服务。客户不需要具体知道服务的范围和选择的标准，简单地说就是，客户只需要让自己的希望为别人所知，然后由 Sponsor-Selector 完成剩下的任务。如果对这种预知行为的需求存有疑问，本章“已知的用法”部分可能会使用户感到惊奇。

第6章：扩展对象(Extension Object)，Erich Gamma 编写 面向对象设计中一个烦人的问题就是很难向已有的接口中添加操作。传统的解决方案涉及到修改根类并添加新操作，很可能用默认的实现，然后重新实现每个不能处理默认操作的子类中的操作。这种方案的问题在于需要反复重新编译所做的改变，出现普遍的改变时，最精密的编程环境也不能避免大量处理。即使用户肯承担这个开销，重复的接口扩展也会导致接口变得臃肿、没有逻辑，以至于无法理解。

Design Patterns 包括了解决这一问题的两个模式：Visitor 和 Decorator。扩展对象是解决这一问题的另一种模式，由其中一位作者编写。在该模式中，扩展的操作放在独立的类中，它们通过继承或组合混合进原来的类中。在类似 COM 和 OpenDoc 的高度动态环境中，从选取模式的意义来看，这也许是迄今为止最具扩展能力的方案，但它同时也是一个复杂的方案。所以，理解如何使用扩展对象是最基本的，知道何时使用它们或是何时不能使用同样是最基本的。本章的适用性、结果和实现对它的理解是至关重要的。

参考文献

［Gamma +95］ E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*:

Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.

[Martin97] R. C. Martin. "Design Patterns for Dealing with Dual Inheritance Hierarchies in C++." C++ Report 9(4), April 1997, pp. 42-48.

[Riehle96] D. Riehle. "Patterns for Encapsulating Class Hierarchies." In J. M. Vlissides, J. O. Coplien, and N. L. Kerth (eds.). *Pattern Languages of Program Design 2*. Reading, MA: Addison-Wesley, 1996, pp. 87-104.

第1章 空 对 象

Bobby Woolf

意图

空对象(Null Object)给另一个共享相同接口但不做任何工作的对象提供一个代理。这样，Null Object 封装了如何不做任何事并隐藏其合作者细节的执行决定。

空对象(Null Object)给另一个共享相同接口但不做任何工作的对象提供一个代理。这样，Null Object 封装了如何不做任何事并隐藏其合作者细节的执行决定。

别名

Active Nothing [Anderson95]

动机

需要合作者的对象有时并不需要合作者去做任何事。但是，这个对象希望将不做任何事的合作者与实际提供行为的合作者同样对待。例如，Smalltalk-80 和 VisualWorks Small-Talk 中的 Model-View-Controller(模式—视图—控制器)框架。视图利用其控制器收集用户输入。这就是一个 Strategy，因为控制器是视图在如何收集输入方面的谋划者[Gamma +95，P. 315]。

视图可以是只读的。因为视图不收集用户输入，所以它并不需要控制器。但是，View 及其子类被实现成要求有控制器，而且它们比较普遍地使用控制器。

如果 View 类的实例都不需要控制器，则类就不需要 View 的子类。它是作为可视类实现的，与不需要控制器的 View 相似。但是，对带有一些需要控制器的实例和一些不需要控制器的实例的类来说，这并不起作用。此时，类需要 View 的子类，且所有的实例都需要控制器。这样，View 类就需要控制器，而特定的实例并不需要。

解决这个问题的常见方法是将实例的控制器设置为 nil。但是这样做并不能工作得很好，因为视图通常会发送消息给只有 Controller 能理解的控制器(例如 isControl-Wanted、

isControlActive和 startUp)。因为 UndefinedObject (nil 的类)不能理解这些 Controller 消息，所以视图必须在发送这种消息以前检查其控制器，如果控制器为 nil，视图就必须决定要做什么事情。所有的这些条件代码将视图的实现弄得乱七八糟，如果不止一个 View 类将 nil 作为控制器，则处理 nil 的条件代码将很难重用。所以用 nil 作为控制器并不能工作得很好。

例如，下面是 VisualPart(View 层次结构中最高层的类)如何实现 objectWantingControl (其关键消息中的一个)的过程

```
VisualPart >> objectWantingControl
    ...
    ^ctrl  isNil  ifFalse:
        [ctrl  isControlWanted
               ifTrue:    [self]
               ifFalse:   [nil]]
```

注意，在发送类似 isControlWanted 的控制器消息以前，要检测 ctrl 是否为 false。这个简单的检测过程可能导致大量问题出现。第一，检测过程不仅会混乱代码，也会使行为难以理解。当控制器为 nil 时，实现返回什么值呢？(方法返回正常工作的 nil，但是它是创建者所希望的值吗?)第二，目前存在几种发送 isControlWanted 给未知控制器的视图的控制器，它们都必须检测 nil 并正确、一致地处理这种情况。第三，如果开发者实现了发送 isControlWanted给控制器的新代码，他就需要增加这个检测过程，如果忘记增加检测过程，则他的代码可能会内嵌有非常敏感的故障。最后，大多数 isControlWanted 的发送者不仅必须受到保护，而且大多数其他关键控制器消息的发送者(如 isControlActive 和 startUp)也受到保护。

解决这个问题的另一个方法是使用只读控制器。一些控制器可以被设置为只读模式，这样它们就可以忽略掉输入。这种控制器仍然收集输入，但是当处于只读模式时，它对输入不做任何处理。如果处于编辑模式，它就可以在改变模式的状态后对同样的输入进行处理，对总是处于只读状态的控制器来说，这是比较有杀伤力的。这种控制器不需要依赖当前模式进行任何的处理，它的模式总是只读的，所以无需进行处理。这样，总处于只读的控制器应该编码成不执行任何处理。

相反，我们需要的是特定编码成只读的控制器。Controller 的这种特殊的子类被称为 NoController (见图 1-1)。它实现了所有的 Controller 的接口，但是不做任何事情。询问 isControlWanted 时,它自动应答 false。被告知 startUp 时，它不做任何工作，并自动返回 self。它可以做控制器完成的每件工作，但实际上不做任何工作。

图 1-1 演示了视图如何得到控制器以及控制器如何成为 NoController。NoController 实现了控制器实现的所有行为，但是它不做任何工作就做到了这点。

例如，图 1-1 演示了真正实现 VisualPart >> objectWantingControl 的过程。