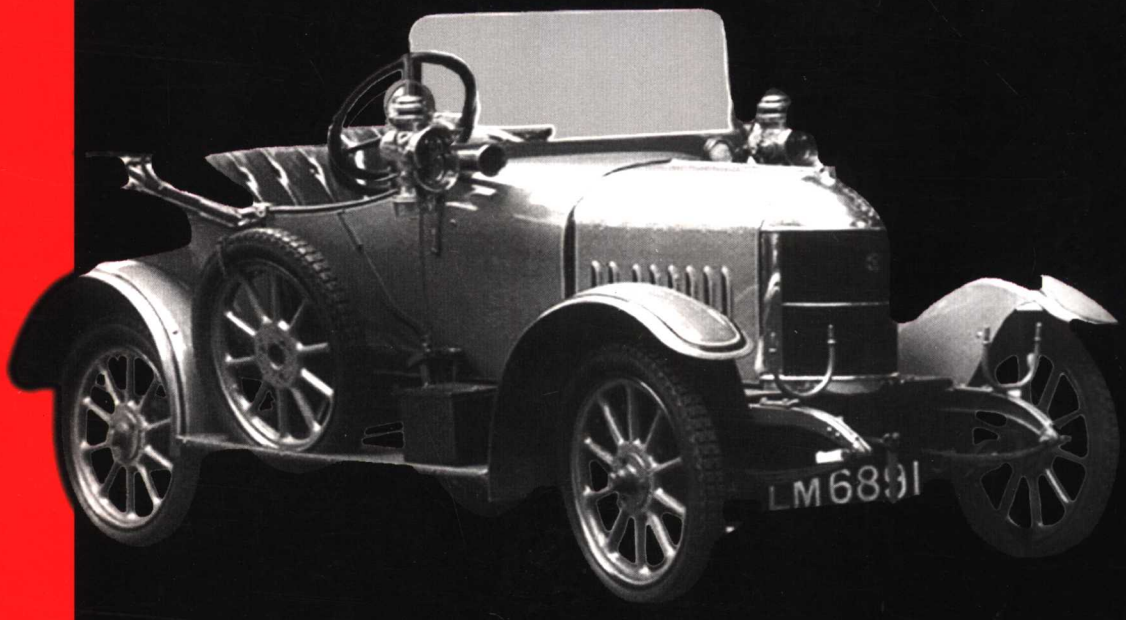
HZ BOOKS

Apress®

设计模式初学者指南

Holub on Patterns
Learning Design Patterns by Looking at Code

(美) Allen Holub 著
徐迎晓 等译



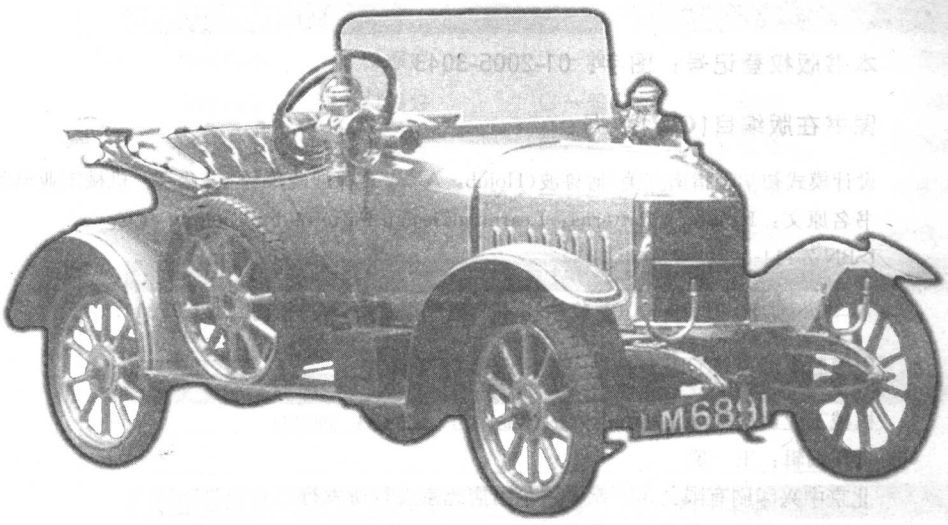
机械工业出版社
China Machine Press

设计模式初学者指南

Holub on Patterns

Learning Design Patterns by Looking at Code

（美） **Allen Holub** 著
徐迎晓 等译



机械工业出版社
China Machine Press

本书系统介绍如何使用模式来解决面向对象编程的常见问题。主要内容包括：面向对象与设计模式初步，使用接口创建型模式编程，涉及对基类问题的分析，对 extends 缺点的分析，对 get/set 方法的剖析等。本书通过两个完整的程序，反映了实际编程中模式应用的技巧，对掌握设计模式很有启发性。

本书适合软件开发技术人员阅读，也可作为高等院校计算机专业相关课程的教学参考书。

Allen Holub：Holub on Patterns：Learning Design Patterns by Looking at Code
(ISBN：1-59059-388-X).

本书版权登记号：图字：01-2005-3043

图书在版编目(CIP)数据

设计模式初学者指南/(美)何鲁波(Holub，A.)著；徐迎晓等译．-北京：机械工业出版社，2006.9
书名原文：Holub on Patterns：Learning Design Patterns by Looking at Code
ISBN 7-111-19799-2

Ⅰ.设…　Ⅱ.①何…　②徐…　Ⅲ.程序设计-模式-指南　Ⅳ.TP311.1-62

中国版本图书馆 CIP 数据核字(2006)第 097647 号

机械工业出版社(北京市西城区百万庄大街 22 号　邮政编码　100037)
责任编辑：王　璐
北京中兴印刷有限公司印刷·新华书店北京发行所发行
2006 年 9 月第 1 版第 1 次印刷
186mm×240mm·24.5 印张
定价：49.00 元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：(010)68326294

译者序

有关设计模式已经有大量的书籍面世，但本书还是令我的眼睛为之一亮。翻开前面两章，首先被作者关于模式的生动比喻所吸引，“穿堂风”模式和“角落办公室”模式使我们很轻松地熟悉了有关模式的基本概念，也纠正了很多人普遍会有的错误思维。作者在很多地方直言不讳，观点鲜明，文风诙谐，对微软进行了多次毫无顾忌的嘲讽，对 Java 本身的某些设计失误也作了较深入的分析，这些都给我留下了深刻的印象。

本书前两章的内容不仅和模式有关，也对面向对象技术作了不少讨论。书中的很多观点为我们打开了另一扇窗户，让我们关注到平时不大注意的问题。比如对脆弱的基类问题以及 extends 的缺点的大量分析、对 get/set 方法的批判、对模板方法模式所存在的问题的分析等。在作者的网站中有个幻灯片，标题是“你所知道的一切都是错误的!”(Everything You Know is Wrong!)，该标题作为本书特点的写照颇为合适。

在对模式的讲解中，本书让我印象最深的地方有两点：一是模式的代码实现与众不同；二是全书通过两个完整的例子来讲解模式。

很多设计模式书籍中的代码实现都大同小异，而本书中的代码实现则大量使用了匿名内部类，给人耳目一新的感觉，很有启发性。通过案例讲解模式在很多设计模式书籍中已非鲜见，但本书中的案例还是非常独特。第一个案例“生命游戏”非常有趣，尽管作者为了讲解模式强行塞进去很多模式，但其中的模式应用还是能给我们不少启发。第二个案例“嵌入式 SQL”是产品级代码，更具挑战性，书中对其体系结构以及模式应用作了详细的分析。尤其有特色的是，两个案例中的模式应用都不像其他的设计模式书籍中的案例那样理想化，而是将各种模式以错综复杂的关系交织在一起，经常是一个类参与多个模式，这真实反映了现实世界中模式应用的情况。为了表现这种错综复杂的关系，两个案例都使用了模式图以及 UML 类图，尤其是模式图起到了很好的效果。

只要学习过一种面向对象编程语言，各种层次的读者都可以从本书中汲取到有用的内容。随着翻译的进展，我在讲授各个层次的面向对象技术相关课程时，穿插了本书中的绝大部分内容。如果读者以前从来没有接触过面向对象分析与设计，阅读本书会有一点难度，可先涉猎一些面向对象分析与设计的初步知识；如果以前从来没有接触过设计模式，本书的很多内容还是偏难些，应该在阅读到相关模式时针对相应的模式先参考一下网上或者其他书籍中一些基础的内容。

翻译是一项费时费力的浩大工程，尽管我们在翻译过程中尽力做到完美，但每一遍的校译都会发现一些新的值得改进之处。读者可以在译者的网站 http://javabook.126.com 中找到译者的联系方式并进行意见反馈。

本书翻译过程中，周逸勋、李波、谢维分别翻译了第 3、4 章和附录，徐迎晓翻译其他内容并负责全面的校译与润色，刘昉参与了各章的校译以及疑难翻译的讨论。

徐迎晓

2006 年 8 月于复旦大学

前　言

本书介绍如何以面向对象的方式来编程，以及如何使用模式来解决面向对象系统中的常见问题。

本书最基本的观点是：学习和理解设计模式的最佳方法是在实际代码中观察它们，这时你会发现各种模式混杂在一起，现实世界中的模式就是如此。

因此，本书通过查看计算机程序的方式将设计模式呈现给你。我的意图是使得 Gamma、Helm、Johnson 和 Vlissides 的开创性著作《设计模式：可复用面向对象软件的基础》(Addison-Wesley，1995 年，中文版已由机械工业出版社出版)更加清晰和易于理解。(这四个人经常被称为“四人帮”或“四人组”(GoF)，他们的书也通常被称为“四人组”的书。)本书将“四人组”的书放在具体环境中讲解，按照设计模式在现实世界中的实际使用方式来演示和讲解设计模式。当你看完本书后，将接触到所有的“四人组”设计模式，而且是在真正的计算机程序环境中接触的。

不要误解，本书不是要取代“四人组”的书，而是作为其补充。Gamma、Helm、Johnson 和 Vlissides 的工作为面向对象设计作出了巨大贡献，没有他们的工作，本书当然也不会存在。不过“四人组”的书对于许多程序员来说太深奥且晦涩难懂，因此本书有其存在的必要。

本书和其他设计模式的书相比是不合常规的，是“由内而外”的。本书不是对设计模式进行编目并在编目的每一节给出脱离现实的、简单的例子，而是讨论了两个计算机程序，讨论时使用的术语是这两个程序所使用的模式。你将看到模式在实际的程序中是什么样子以及模式是如何以复杂的方式彼此交织在一起的。

设计模式编目方法(如设计模式原著的做法)将各个模式隔离了开来，不容易理解现实世界使用模式的方式。如果你以前已经使用过演示这些模式的代码的话，这种编目方法当然是很优秀的。但是如果你以前没有这类代码的编程经验，则这种编目方法会难以理解。同时，编目方法只能使你获得对设计模式知识性的理解，而几乎不能使你理解如何使用模式来编写实际的代码。

预备知识

本书假定你已经知道 Java 并至少使用 Java 编写过几个程序。特别是，我在书中大量地使用了匿名内部类，因此你必须熟练掌握该语法。同时你必须精通“核心”Java 包(比如 java.io)并具备用户界面体系(如 Swing 和 AWT)的基本知识。这些在你学习语言的时候可能已经掌握了。

本书还假定你已经掌握了面向对象编程的基础：继承、接口和多态等。本书后面会讨论到诸如 extends 的缺点之类的东西，为了使讨论有意义，你必须知道 extends 是做什么的。我假定你已经知道了其优点，所以不会再烦述面向对象语言特征诸如继承有什么优点。我在讨论某个编程惯用法或者语言特征的负面因素时，请不要认为该编程惯用法或者语言特征没有值得肯

定的一面。我假定你已经知道的东西就不需要在书中讲解了。

最后，我假定你对 UML 有初步的了解。UML(统一建模语言)是一种对设计的图形化表示法，可用于显示程序的静态(类)结构以及组成程序的对象在运行时是如何相互交互的。

如果在你继续阅读本书之前需要在技术上更上一层楼的话，下面的“参考资料”一节中列出的 Web 页面给出了这些主题的所有参考资料。如果你没有真正理解 UML，或许还可以应付过去；但如果对 Java 不熟悉的话，阅读本书将很困难。

假设

本书假设你希望知道如何构建纯粹的面向对象解决方案，因此没有对基于该假定的每个陈述都加以特别说明。面向对象策略通常都包含基于过程的替代方案，但本书没有讨论这些替代方案。

我之所以提起这些显而易见的东西，是为了防备一些不可避免的吹毛求疵者，他们会仅仅因为书中没有对每个问题都讨论所有的方案(包括忠实基于过程的方案)而抱怨说整本书都是错误的。

本书有些话题会引发争议，包括对实现的封装(即指应该尽力避免使用大多数常用的 get/set 函数)和过度使用实现继承(这将创建不必要的耦合关系)。

注意！注意！本书将特立独行

最后，我想强调一下我的行文风格。

到目前为止你应该能看出来，我有什么想法就会说出来，我通常不会通过辩解来证明我的观点，如果你不喜欢这样，那就买其他书吧。

每个人都有自己的观点，如果将这些观点隐藏在不偏不倚的虚饰之下，那么除了让人困惑外没有什么用处。你可以不同意我的观点，但请在你有了强有力的证据支持你的观点时再来和我争论。“没有人这样做”并不是强有力的证据，反之亦然：“每个人都这样做”也不是强有力的证据。

有时我被指责为太“独断”。如果你所说的“独断”是指我激烈地捍卫我的思想(那些在实际代码中效果很好的思想)，并且反过来对实践中很糟糕地失败了的思想大加贬低的话，那么我想我确实是独断的。但我自认为我是现实的，而不是独断的。我顽固地遵循着面向对象的准则，因为每次当我背离了这些准则时，都必须重写代码。我只是没有时间把同一件事做两次而已。

我对实践的偏好反映在本书的结构中，本书的目录安排是围绕代码而不是学术化的分类学进行的。为了使有些内容易于理解，我没有严格遵循编程语言的语法，这可能会使你觉得有些不满。当然，本书是针对程序员，而不是针对理论家的。(很奇怪，有时我会被指责为太“学院派”，好像学院派是个贬义词一样。真正的学院派人士通常不喜欢我的书，因为我的书不够形式化，太多地依赖代码而不是数学。)

有时本书也会离题到不相关的设计话题上，而不是严格关注于模式。我假定你不仅知道我

在做什么，还知道为什么这样做。在写本书时，我们就像坐在桌子边上交谈一样，而不是我站在讲台上作正式的报告。如果你想要正式的东西，建议你阅读“四人组”的设计模式书籍。“四人组”的书是使用高度结构化的方式讲解这些内容的杰出著作，它可能更符合你的胃口。

参考资料

本书没有加上“参考资源”一节来加大篇幅，因为这些内容在书籍面世之前就会过期，所以我建立了一个设计模式资源网页(http://www.holub.com/goodies/patterns)。在该网页上你可以找到本书的所有代码，也可以找到参考书目以及其他与模式有关的 Web 站点的链接。

结束语

既然前面已经提醒了你，现在到了开始工作的时间了。设计模式(以及用模式来思考)是杰出的东西，它们能使你的工作更有效率，使你的代码更加易于维护，并能为你提供交流的词汇，使得你与其他程序员和设计人员的交流更加有效。本书向你展示的是设计模式是如何真正起作用的，以及如何使用设计模式编写出优秀的代码。

致谢

当然要向“四人组”(Gang of Four)：Gamma、Helm、Johnson 和 Vlissides 致以极大的感谢。如果没有他们，本书也不会存在。

Ken Arnold 精心审阅了本书，我从来没有见过像 Ken 这样透彻而细心的技术审稿人。他提出的详细意见极大地改进了本书，对此我十分感激。

本书的一小部分出自我在 JavaWorld(http://www.javaworld.com)上的 Java Toolbox 专栏中的内容。

关于作者

Allen Holub 自 1979 年开始就在计算机行业工作。他目前从事咨询工作，帮助公司避免在软件上浪费不必要的金钱。他提供面向对象设计及 Java 方面的培训，也提供软件过程指导和设计评估服务，提供技术审议，偶尔也会写一些程序。

Allen 的编程经历覆盖了从操作系统到编译器、从应用程序到 Web 服务整个范围。他是 Java 的最初尝试者，自从 Java 在 1995 年发布以来就在使用 Java 编程了。在这之前他使用了 8 年 C++，也使用 C、Perl、Pascal、PL/M、FORTRAN、SQL 和各种汇编语言等。他备尝艰辛方才领悟到设计的真谛，总结了很多他自己都不愿承认写过的徒劳无功的程序后，现在他已经成为公认的面向对象设计、UML 和软件过程专家。他曾任 NetReliance 公司首席技术官、Ascenium 和 Ontometrics 公司的顾问。他还是软件开发大会安全专题的主席。

Allen 在 1998 年至 2004 年期间为 JavaWorld 撰稿，目前是《SD Times》的特约编辑。他编写了 9 本书(包括《Holub on Patterns》、《Taming Java Threads》和《Compiler Design in C》等)，在杂志上发表过 100 多篇文章(包括《Dr. Dobb's Journal》、《Programmers Journal》、《Byte》、《MSJ》以及其他杂志)。他是 IBM 开发者资源网站 (IBM developerWorks) 的 Component 专区深受欢迎的“面向对象设计过程”专栏的作者，也是 CMP Media 的 Java 解决方案的技术编辑。Allen 定期在加州大学伯克利分校讲授面向对象设计和 Java 编程的课程。

Allen 的联系方式为 http://www.holub.com/allen.html。

目　录

第1章

面向对象与设计模式初步

通常，这类书籍都会以引用建筑师 Christopher Alexander 的话作为开场白，正是 Alexander 提出了设计模式的思想。不过我发现，Alexander 虽然是多部优秀书籍的作者，但他的表达有时却晦涩难懂，所以本书将跳过别的书中必不可少的对 Alexander 书中内容的引用。当然，不可否认，Alexander 的思想引发了整个设计模式运动。

类似地，软件领域设计模式的开创性书籍是 Gamma、Helm、Johnson 和 Vlissides 的《设计模式：可复用面向对象软件的基础》(Design Patterns：Elements of Reusable Object-Oriented Software)(Addison-Wesley，1995 年)。这四个人被大多数设计人员开玩笑地称为“四人帮”(“四人组”)。如果没有“四人组”的书，我的书也不会存在。我和其他的面向对象程序员们向这四位作者致以极大的感激。然而，“四人组”的书是用正规的、学术化的方式来表达模式的，大多数初学者都觉得它太深奥难懂。冒着丢失部分学术严密性的风险，我的书将采用更加亲切和友好的叙述风格。

1.1 模式和惯用法

我们首先通过讨论简单的编程惯用法来探索模式的思想。很多设计模式是如此通用，以至于大多数程序员的头脑中根本不把其作为模式，而是作为语言的惯用法。换句话说，就是不把这些模式当做任何特殊的东西——这些模式只是做事情的步骤。有些人基于其表达方式来区分模式和惯用法(比如模式是使用正式的方式来表示的，而惯用法不是这样)。但是我没有看出这之间有任何区别，惯用法就是模式，只是惯用法的使用已经成为很平常的事情了。

派生关系(derivation)是从模式进化到惯用法的一个出色的例子。追溯到 20 世纪 80 年代早期，C 语言红极一时，派生关系是一种设计模式。在 C 语言中你可以发现很多使用扩展(extends)关系的例子。比如 malloc( )的标准实现中使用头文件(基类)进行扩展而创建另一个结构(派生类)，这个新的结构实际上继承了基类的 free( )方法。

抽象函数也是派生模式的一部分，C 语言中经常会传递由函数指针组成的表，该函数指针表可针对不同的“类”进行不同的初始化。这也是 C++ 实现抽象方法和接口继承的方法，但回到 C 语言的世界，我们没有为这种用法起一个名字。

派生关系没有内置在 C 语言中，而且大部分 C 程序员不会用面向对象的方式来编程，所以派生关系不是编程惯用法，而是模式。在很多必须解决类似问题的程序中，你都能看到这种

用法，而这种用法对普通的C程序员来说不是天生就会使用的。

当然，现今派生关系和接口已经内置在编程语言中；它们已经成为惯用法。

1.2 究竟什么是设计模式

首先，也是最重要的一点是：设计模式是被发现而不是被发明出来的。Christopher Alexander在研究很多成功的建筑时发现，当他关注于建筑的某一方面(比如是什么东西使房间“令人感觉舒适”)时，就能发现特定的模式。一个设计得成功的房间之所以令人感觉舒适，得归功于对几类特定问题(如采光)的解决，其解决手法都是相似的。出于同样的原因，当查看不同程序员编写的各个程序时，如果你关注于这些程序必须解决的特定问题(比如对子系统进行隔离)，同样会发现模式。你会发现不同程序员在解决类似的问题时，他们会各自独立地开发出类似的技术来解决这些类似的问题。只要你对技术比较敏感，就会开始逐渐从所观察的一切中发现模式。不过只有在多个独立开发的程序中都可以看到的才能称为模式。如果有作者说：“我们发明了一个设计模式……”，很明显他们并不知道他们在说什么。他们提出的可能只是一个设计，但并不是模式，除非有几个人独立地发明了它。(当然，有可能出现这样的情况，发明出的“模式”被足够多的人采用，从而变为真正的模式。)

设计模式是用来解决一类相关问题的通用技术，而不是解决问题的特解。对于房间的采光的设计，或许每位建筑师会采用不同方式来设计出令人感觉舒适的房间；同样，对于某类问题的解决方案，或许每位编程人员也会用各自不同的方式来实现。模式就是这种解决方案的通用结构——如果你愿意的话，可以将模式称为“元解决方案”，而不是解决方案本身。

音乐是一个很好的类比。你可以将“传统音乐”这一概念看做编曲模式。你会因为一段音乐听起来像传统音乐而识别出其符合“传统音乐”模式，而符合“传统音乐”模式的各种音乐本身则是千差万别的。

尽管模式本质上是通用的，但你还是不能将模式直接从一个程序剪切-粘贴到另外一个程序中(尽管在当前的上下文和原有的上下文相类似时，有可能能够复用特定的解决方案)，这个问题常给模式的初学者带来巨大的困扰。从我在Web上看到的评论来看，很多程序员似乎认为，如果一本书所提供的例子和“四人组”书中的例子不同，就说明该书的作者不懂模式。这种态度其实只能说明写这些评论的人不懂模式；他们将用来演示模式的代码片断和模式本身混淆了。正因为如此，我对所讨论的每个模式都试图给出几个不同的例子，以便你看到模式可以有完全不同的实现——我不会使用“四人组”的例子，除非他们的例子和实际的编程问题相关(其实大部分都没有关系)。

更让事情变得错综复杂的是，参与模式的实际的对象和类几乎总是会同时出现在其他模式中。从一个角度来看像这种模式，而换个角度来看就像另外一种模式了。更加令人困惑的是，很多模式的实现使用的静态结构都一样。当你看“四人组”书中的UML静态结构图时，会发现这些静态结构图看起来都一样：都有接口、客户类和实现类。模式之间的不同点在于系统的动态行为和程序员的意图各不相同，而不是类以及类之间的连接方式。

我将试图用建筑架构的例子展示这些问题，让我们关注于两个领域：通风和采光。

在通风领域，我不希望房间让人感觉气闷。看了几间确实让人感觉舒适的房间后，我的脑海中浮现出一个我称之为“穿堂风”(Cross Ventilation)的模式。采用这个模式的房间在与窗户相同的高度上从房间的一头到另一头有进风口和出风口相对着。空气从进风口穿过房间离开出风口。识别出这个模式并给其命名后，我给出简洁的描述，总结该模式所处理的问题和对应的解决方案——这在“四人组”的书中称为“意图”(intent)。在穿堂风模式这个例子中，我的意图是“允许空气在半人高的高度直接水平地穿过房间，从而减少闷热并使房间更加舒适”。任何符合这一意图的架构机制都是该模式合法的具体化(reification，我在后面会解释这个词)。[“四人组”的书中使用意图这个词比较奇怪，我在本书中会更多地使用目的(purpose)这类词。]

具体化是一个晦涩的词，但我发现这个词相当有用。然而这个词在文献中用得并不多。表面上看，reify 是指“使其成为现实”。一个想法的具体化是指该想法的具体实现(concrete realization)，一个想法可以有数百万种可能的具体化。我使用 reify，而不是其他普通的单词，来强调模式不是什么。比如模式不是实例化(instantiated)。每个类的实例都是相同的(至少结构上相同)，而设计模式不是这样。类似地，具体化也不是模式的实现(implementation)——模式的具体化是设计，而不是代码，一个给定的设计有很多合法的实现。

那么，穿堂风模式的具体化是什么呢？可以是窗户正对着窗户、窗户正对着门、门正对着门等，也可以是窗口正对着抽风机，或相对的两堵墙上分别放置抽风机和鼓风机，还可以是黑猩猩对着对面墙上的洞咆哮。事实上，你甚至不需要墙：房间的相对的两面如果没有墙也符合该模式。给定的模式有无数种具体化。

尽管模式的具体化有很大的弹性，但你不能根据自己的喜好来对模式中规定的特性任意取舍。比如，仅是有空气的入口和出口是不够的，还必须满足高度和直接相对这两个要求。如果将空气的入口和出口放在天花板上，就不是该模式的合法具体化。(那些住在不通风的高层办公楼的人可以作证，尽管天花板有通风设备，还是令人感觉气闷。)

总之，穿堂风模式的意图是“允许空气在半人高的高度直接水平地穿过房间，从而减少闷热并使房间更加舒适”。该模式的参与者，无论是窗户、门还是大猩猩，必须有空气入口和出口这两个要素。

下面来看采光领域。在看了很多房间之后，我注意到最令人惬意的房间有个特点：在相邻的两堵墙上都有窗户。这也是为什么拐角处的办公室令人向往的原因：多方向的自然光源使房间更加令人舒适。我将这种模式命名为“角落办公室”(Corner Office)模式，与之相配，我给出以下的模式意图：“通过在相邻的两堵墙上设置两个自然光源，使房间更加令人心旷神怡。”同样，这一模式有无数的具体化：两堵墙上各有一个窗户；一堵墙上有窗户而另一面是玻璃门(French Door)；两堵墙上都是玻璃门。你可能会指出一堵墙上有窗户，再在相邻的墙上装个镜子也可以，因为镜子反射的自然光也充当了光源。如果我是比尔·盖茨，我会在一堵墙上放窗户，另一堵墙上如果没有窗户的话就放 600 英寸等离子显示器。不过这不是合法的具体化，因为等离子显示器不是自然光。当然，你同样可用无数种方式实现窗户和玻璃门模式。

现在我们来考虑具体的设计——建筑设计。图 1-1 在一个设计中显示了穿堂风模式和角落办公室模式的具体化。在图中我既给出建筑设计图又给出了对应的 UML 图。使用 UML 1.5

的协作符号来表示模式。模式的名字放在椭圆框中，椭圆框带有虚线，虚线延伸到参与模式的各个类上。虚线上加有注释表明对应的类在该模式中所充当的角色。

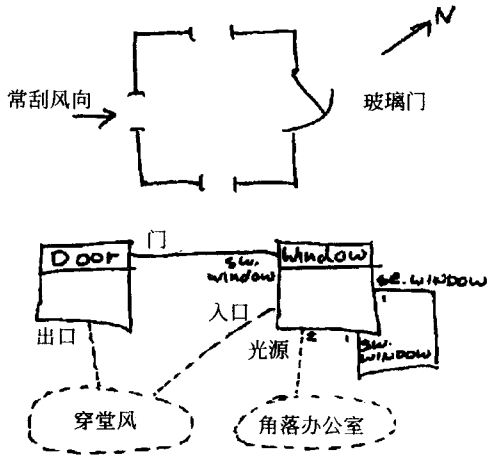


图 1-1 穿堂风模式和角落办公室模式相结合的具体化

西南端的窗户(SW. window)充当穿堂风的入口，其对面的门作为风的出口。因为常刮风向是来自西南方向的，所以其他两个窗户并不参与穿堂风模式。西南和东南窗(SE. window)作为两个光源参与角落办公室模式(太阳光从南面射入)。门和西北方向的窗户都不作为参与者，因为它们不是主要的光线来源。西南方向的窗户比较有趣，因为它同时参与了两个模式。在穿堂风模式中其角色是“空气入口”(air source)，在角落办公室模式中其角色是“光源”(light source)。不同模式中的对象和类经常会以这种方式交织在一起。

值得注意的是，不能简单地通过结构来识别模式。比如另一个同样结构中，如果没有窗户可以充当空气入口，风就会被阻挡住。出于同样的原因，某扇窗户可能与隔壁建筑的外墙只隔两英尺远或对着走廊，这样就不能作为主要的光源(尽管这时可以作为风的入口或出口)。如你所见，当你查看实际的模式时，必须有上下文信息——包括架构的意图——来识别计算机程序中的设计模式。你不能仅仅看一眼 UML 图就识别出所有模式，你必须知道使用这些对象和类的意图。在后面章节的例子中，你会看到很多有关这个现象的例子。

重新回到剪切-粘贴问题，我希望你现在能看到模式是如何具体化为大量不同的设计的，每种设计都可用无数的方法来实现。有人说能通过设计工具剪切粘贴一个模式，这种说法是毫无意义的。然而，很多面向对象 CASE 工具宣称拥有“模式库”，使用这些模式库可以将模式插入到你的设计中。实际上，这些库包含了预制的 UML 结构图，这些结构图只是“四人组”书中给出的给定模式的某个具体化。在你的设计中粘贴这些结构尽管在有些时候是有用的，但不要将这些“粘贴”操作和在设计中实际使用模式混为一谈。好的设计几乎总是需要针对其具体环境进行定制，盲目的剪切粘贴不是设计，就像填图不是绘画一样。

1.3 模式究竟有什么用途

既然这样，如果模式没有固定的形态，那么模式究竟有什么用途？

当我第一次读“四人组”的书时，我并没有留下深刻印象。模式似乎只是将优秀的程序员已经发现的一些东西用文绉绉的方式表达出来，这些程序员通常都削尖了脑袋来发现对应问题(模式所要解决的问题)的最佳解决方案。确实，如果我早几年读这本书，对这些问题就会少碰几次壁了，但这一切似乎有些小题大做。

我的这种想法一直持续到我和另外一个设计人员讨论某个项目时。他指着一段设计说：“这个接口构成两个子系统之间的桥，桥本身是用这一组对象适配器实现的。”我被如此简洁的描述打动了。只用了两句话，他省去了大约半个小时的详细的解释。或许在所有这些模式中都存在一些本质的东西。

后来我参加了第一届 Java One 的报告会，那里所有的 AWT 都是用模式的术语来描述的。这些描述都简洁而明晰——实际上，和演讲者不使用模式的方法来进行描述相比，大部分都更加简单明了。

我在开始下一个设计项目之前，回头重新阅读了设计模式的书，并有意识地用模式的术语来思考我的下一个设计。换句话说，我开始问我自己：“我在这里要达到什么目的？是否已经有针对这一问题的模式了？”(使用模式描述的“目的”一节来查看哪些模式是相关的)。当答案是“是”时，我会立即使用模式。我发现采用这种方式显著地缩短了设计时间，同时也得到了更好的设计结果。我对模式知道得越多，事情就做得越快。此外，我的初始设计只需作少许改进就能被接受了，比以前需要做的改进少多了。

我已经上瘾了。

模式提供了一个组织化的框架，大大增强了交流能力，设计归根到底其实就是交流。以前需要几个小时的讨论现在几分钟就好了，每个人都可用更少的时间来做更多的实际工作。我回去读了我能找到的所有有关模式的书籍，发现“四人组”的书只是触及了模式的表面。Web 和文献中已经有了上百个记录在案的模式，其中很多都可以应用到我正在做的工作中来。实际上，我发现完全(solid)精通与我的工作相关的模式会使我的工作做得更快更好。(这里使用“solid”是指彻底熟悉材料，不用到书中去查找。)

1.4 模式在设计中所充当的角色

模式在设计过程中何时进入我们的视线？它们在设计中充当着什么角色？这个问题的答案随着你在设计时所采用的方法学的不同而不同——我希望你确实在使用某种方法学——但设计模式主要在代码实现级别上有用，因此设计模式在你考虑代码实现时开始进入我们的视线。更深层次的问题是，什么时候开始分析阶段[该阶段主要考虑问题域(problem domain)]，什么时候又开始设计阶段(该阶段主要关心实现问题)？

我所知道的最好的类比是建筑的设计和建造。建筑的设计并没有给出所有的建造细节，建筑设计只给出什么地方有墙，而不是给出如何建造这堵墙；给出管道设备的走向，而不是如何

铺设管道。建造房屋时，确实会进行涉及墙如何建造、管道如何实际进行铺设等设计活动，但这些设计的图纸很少保留，因为这些具体做法是自成解释的。比如木匠可能使用“放置立柱”模式建造坚固的墙。设计只是给出墙在什么地方，而不是如何建造墙。

将这种类比用到软件中：大多数的项目中，当你觉得熟练的编程人员可以毫无困难地进行代码实现时就可以停止设计活动了。我从来不会考虑将使用 Swing 创建窗口之类放在设计中。创建窗口是程序员自己应该知道如何去做的。如果代码是按照专业标准来写的话(选择适当的名字、好的编排格式以及必要的注释等)，则代码实现中的各种选择是自成文档的。

因此，设计模式在设计文档中通常并不写得清清楚楚，而是代表代码实现时所作的决策。某个实现者所应用的模式很少记录得很细，当然模式参与者的名称或其他注释应该表明使用了什么模式(如 WidgetFactory 是 Factory 模式的具体化)。

当然，这种“不需要设计的模式”也有例外的情况。比如角落办公室模式所使用的窗户，其软件对等体应该出现在设计文档中(在设计文档中应该给出窗户安排在什么地方)。类似地，非常复杂的系统通常需要在设计中给出更多的细节(同样，摩天大楼的架构设计比小房间的设计需要给出更多的细节)，因而在文档中需要深入记录模式的细节。

模式与简洁性之间的平衡

一个相关的问题是模式会给系统带来更多的复杂性。如果“愚蠢地恪守教条是头脑僵化的幽灵在作怪”[⊖]，那么，不必要的复杂性就是蹩脚程序员制造的幽灵。就像爱默生所说：“头脑僵化的政治家、哲学家和神学家”崇拜教条，很多头脑僵化的程序员、建筑师出于自身的目的认为模式好，认为只要有可能就应该使用模式。这种盲目的做法几乎注定会导致程序脆弱、混乱而难以维护。每个模式都有其缺点，常引发是否应该使用该模式的争论。

简单的系统比复杂的系统易于建造、易于维护、短小而且运行更快。简单的系统让“不需要做的工作”尽量地多，从而使得“已经做的工作”完成得尽可能地好。程序必须确切地按照用户的要求工作，增加不必要的功能大大延长了开发时间并降低了稳定性。

简洁性通常不是一个容易达到的目标。程序员喜欢复杂，他们具有将工作过度复杂化的倾向。快速建造一个过度复杂的系统通常比花时间使得系统简化来得容易。程序员如果猜测随着时间的推移需求可能会增加或变更，则该程序员会倾向于在程序中支持可能在未来才会存在的需求。然而，因为觉得未来可能会发生什么变化而使代码变得复杂并不是一个好主意(每当我预测未来，我总是错了)。程序员需要这样来编写代码：使程序在未来易于添加新的特性或修改现有的特性，而不是现在就增加这些特性。

这一问题的反面是将本质上复杂的问题过度简单化。你确实想“严格按照”所需要的来做，但将必要的功能删除和增加不必要的功能一样糟糕。一个过度简化的例子是 undo(撤消)功能。Alan Cooper(Visual Basic 的发明者和著名的图形界面的权威)指出：你从来不会询问用户是否确实想做某件事。当然这样啦——不然他们一开始就不会要求做这些事情。多少次你会因为弹出了一个愚蠢的确认对话框而没有真正把文件删除掉？对不小心删除及类似问题的最好的解决

⊖ 哲学家、诗人爱默生(Ralph Waldo Emerson)的名言。——译者注

方案是：先按照用户的要求去做，然后提供 undo 功能，以防用户犯错误。你使用的编辑器软件正是这样做的(想像一下如果编辑器在你每次删除某个字符时都问你“你是否真正想删除该字符?”会是什么样子)。然而，undo 表面上看起来简单，其实很难实现，“完全的 undo 系统增加了太多的复杂性，因此，让我们只是抛弃确认对话框吧。”

.简洁、完整和易于修改，这三个需求有时会发生冲突。本书所描述的模式在发生变化或需要增加一些东西时相当有帮助，但出于同样的原因，模式也使代码复杂化。不幸的是，没有严格的规则告诉我们什么时候使用模式比较好，它是一种直觉的判断。这种敏感来自于多年的经验，许多资历尚浅的设计人员/程序员不可能拥有。(Ken Arnold 这位 Java 编程原创书籍的作者之一指出，它来自一种天生的审美感，后天无法培养。)因此，过多地使用模式将导致糟糕的程序，仅仅使用模式并不能确保成功。

另一方面，即使有时完整的模式并不合适，但模式的构造块，比如大量的接口，总是值得在代码中使用。接口并不增加复杂性，如果接口已经在适当的位置上，彻底地重构代码将容易得多。使用接口成本较低而潜在的回报较高。

1.5 模式的分类

为了方便选择合适的模式，对模式进行分类有时很有用。表 1-1 取自“四人组”的书，该表展示了“四人组”模式的一种查阅方式。但你也可以按照自己的方式创建类似的表，以此来对模式进行分类。

表 1-1 “四人组”设计模式分类

		目　的		
		创造型	结构型	行为型
范围	类	Factory Method	Class Adapter	Interpreter Template Method
	对象	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Facade Flyweight Object Adapter Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

“四人组”将模式分为两个大类：一个是类模式(class pattern)，类模式在具体化时需要使用继承(关键字 extends)；另一个是对象模式(object pattern)，对象模式在实现时除了接口继承(关键字 implements)外不需要其他关键字。事实上，对象模式比类模式要多得多。(对这类问题在下一章会进一步讨论。)

在每个大类内部，模式进一步被分为三类。第一类是创建型模式，关注的都是对象的创建，如抽象工厂模式提供了一种方式，可以在不知道对象的实际类型的情况下生成对象(这里