深入 C++ 系列

Large-Scale C++ Software Design

大规模 C++ 程序设计

[美] John Lakos 著
李师贤 明仲 曾新红 刘显明 等 译

Addison Wesley

中国电力出版社
www.infopower.com.cn

Large-Scale C++ Software Design

大规模 C++ 程序设计

要想用 C++ 成功开发大规模的软件系统，不仅需要很好地理解大多数关于 C++ 程序设计的书中所覆盖的逻辑设计问题，而且需要掌握物理设计的概念。这些概念与开发技术联系紧密，甚至很熟练的软件开发人员对其所涉及的内容也可能经验很少或者没有经验。

这是一本为所有从事软件开发工作（例如数据库、操作系统、编译程序及框架）的 C++ 软件专业人员而写的权威著作。它是第一本实际演示如何开发大型 C++ 系统的书，并且是一本少有的面向对象设计的书，尤其侧重于 C++ 编程语言的实践方面。

在本书中，Lakos 介绍了将大型系统分解成较小且较好管理的组件层次结构（不是继承）的过程。这种具有非循环物理依赖的系统的维护、测试和重用从根本上比相互紧密依赖的系统更容易且更经济。此外，本书还说明了遵从好的物理设计和逻辑设计规则的动机。Lakos 给读者提供了一系列用来消除循环依赖、编译时依赖和连接时（物理）依赖的特殊技术。然后，他将这些概念从大型系统扩展到了超大型系统。本书最后讨论了一种容易理解的、适用于单个组件的自顶向下设计方法。附录包含以下内容：一种用来在最小化物理依赖时避免胖接口的有价值的设计模式“协议层次结构”，实现一个兼容 ANSI C 的 C++ 过程接口的细节，以及一整套用于获取和分析物理依赖的类似于 UNIX 工具的完整规范。另外，实用的设计规则、指导方针和原则也收集在附录中。

John Lakos 在 Mentor Graphics 公司工作。该公司编写的大规模 C++ 程序比大多数其他公司要多，并且是首先尝试真正的大规模 C++ 项目的公司之一。Lakos 从 1987 年起就一直使用 C++ 进行专业编程，并于 1990 年在哥伦比亚大学开设了面向对象编程方面的研究生课程。

ISBN 7-5083-1504-9

9 787508 315041 >

责任编辑/闫　宏
封面设计/王红柳

ISBN 7-5083-1504-9
定价：72.00 元

深圳大学出版基金资助

Large-Scale C++ Software Design

大规模 C++ 程序设计

[美] John Lakos 著
李师贤 明仲 曾新红 刘显明 等 译

中国电力出版社

Large-Scale C++ Software Design(ISBN 0-201-63362-0)
John Lakos.

图书在版编目（CIP）数据

大规模 C++程序设计 / （美）勒科著；李师贤等译. —北京：中国电力出版社，2003
（深入 C++系列）
ISBN 7-5083-1504-9
Ⅰ.人… Ⅱ.①勒…②李… Ⅲ.C 语言—程序设计 Ⅳ.TP312
中国版本图书馆 CIP 数据核字（2003）第 033472 号

责任编辑：闫宏
丛 书 名：深入C++系列
书　　名：大规模C++程序设计
编　　著：（美）John Lokes
翻　　译：李师贤等
出版发行：中国电力出版社
地址：北京市三里河路6号 邮政编码：100044
电话：（010）88515918 传真：（010）88423191
印　　刷：北京地矿印刷厂
开　　本：787×1092 1/16　　**印　　张**：40　　**字　　数**：750千字
版　　次：2003年9月北京第一版
印　　次：2003年9月第一次印刷
定　　价：72.00 元

前　言

作为 Mentor Graphics 公司 IC 分部的成员，我有幸与许多富有才智的软件工程师一起工作，共同开发了一些非常大型的系统。

早在 1985 年，Mentor Graphics 公司就是最早用 C++开发实际的大型项目的公司之一。那时，无人知道如何做，也没有人预料到项目会出现这样的问题——费用失控、偏离计划、可执行代码庞大、性能低劣以及令人难以置信的昂贵的开发周期，这是不成熟的开发方法不可避免的结果。

许多有价值的经验都是经过痛苦的历程获得的。没有什么书可以帮助指导这种设计过程，也从未有人在这样的规模上尝试使用面向对象设计。

十年后，由于积累了大量有价值的经验，Mentor Graphics 公司已用 C++开发了数个大型软件系统，同时也为其他人在做同样的工作时不用再付出高昂代价开辟了道路。

在我 13 年的 C 语言（后来转为 C++）计算机辅助设计（CAD）软件开发生涯中，我多次看到了这样的情况：提前计划好总能生产出高质量、易维护的软件产品。在 Mentor Graphics 公司，我一直强调要从一开始就确保质量，要把质量作为设计过程中一个必不可少的部分。

1990 年我在 Columbia 大学开设了一门名为“面向对象设计与编程”的研究生课程。从 1991 年起，作为这门课程的老师，我有机会将我们在 Mentor Graphics 公司获得的许多经验传授给学生，这些经验是我们在从事工业化软件的开发过程中取得的。来自数百个研究生和专业程序员的提问和反馈信息，帮助我明确了很多重要的概念。本书是这些经验的总结。据我所知，这是第一本指导开发大型 C++项目的书，也是第一本针对大型 C++项目中出现的质量问题的书。我希望这些信息在读者的工作中有用，就像在我的工作中所起的作用一样。

读者对象

本书是专为有经验的 C++软件开发者、系统设计师、前摄的软件质量保证人员等编写的。本书尤其适合于那些从事大型软件开发工作（如数据库、操作系统、编译程序和框架）的人员阅读。

用 C++开发一个大型的软件系统，需要精通逻辑设计问题，在大多数有关 C++程序设计

的书中都包括了这些问题。若要进行有效的设计，还要求掌握物理设计概念，这些概念虽然与开发的技术方面紧密联系，但是其中有些方面即便是软件开发专家也可能经验很少或者没有经验。

本书提出的多数建议也适用于小型项目。我们开发项目时通常是先从小型项目开始，然后才开发更大更具挑战性的项目。一个特定项目的范围经常会扩展，因而开始时是小项目，后来就变成大项目了。但是，在大型项目中忽略好的策略所造成的直接后果，比在较小型项目中要严重得多。

本书将高层设计概念与特定C++编程细节结合起来，以满足下面两个需求：

（1）一本面向对象设计的书，尤其侧重于C++编程语言的实践方面。

（2）一本C++程序设计的书，描述如何使用C++编程语言来开发非常大型的系统。

请别弄错，这是一本高级别的书。本书既不适用于从头开始学习C++语法，也不适合于用来学习未曾掌握的C++语言细节，这是一本教你如何使用C++的全部功能去开发超大型系统的书。

简言之，如果读者认为自己C++掌握得很好，但想更多地学习如何使用C++语言去有效地开发大型项目，那么这本书就是为你所写的。

正文中的例子

多数人通过例子来学习。通常，我们提供说明现实世界设计的例子，避免那些只说明某一点问题而在设计的其他方面暴露出错误的例子；也避免那些只说明语言的细节但没有其他意义的例子。

除非特别指出，本书正文的所有例子都表示“好的设计”。因此，前几章的例子与整本书所介绍的所有策略一致。这种方法的不足之处是：读者看到示例代码与自己以往所见的代码不一致，但不能准确地知道为什么会这样。作者认为，若能利用书中所有的例子作为参考，就能弥补这个不足。

对于这种策略有两个显著的例外：注释和包前缀（package prefix）。本书正文中的许多例子的注释，为节省篇幅简单地省略了。有些地方虽然有注释，但也是少而精的。但是，这是一处要求读者“按我说的做，而不是按我做的那样做”的地方——至少在本书中如此。笔者在实践中编写接口时会当即非常仔细地添加注释，而不是事后再加注释，这一点读者可以放心。

第二个例外是，书中前面几个例子中的包前缀的使用不一致。在大型项目环境中需要包前缀，但是开始时，包前缀不便使用，需要一段时间适应。笔者选择先不使用注册的包前缀，在第7章正式提出之后再使用，以便能专注于介绍其它重要的基础内容。

当举例说明设计中的功能时，为了文字的简洁，在例子中使用了内联函数。在正文中多次说明了这种情况。由于本书的大部分内容与如何组织的问题（如，什么时候内联）有关，

所以笔者的倾向是在例子中避免使用内联函数。如果一个函数被声明为“inline”，肯定有其正当的理由，而不只是为了表示方便。

用 C++开发大型系统是一系列的工程方法的折衷，几乎没有绝对。用“从不”或“总是”这样的词语来陈述是很吸引人的，但这样的词语只能用来对内容进行简单的描述。对于那些我希望将来阅读本书的 C++程序员来说，这样总括性的陈述会引起异议——事实确实如此。为了避免陷入这种局面，我会在申明什么是（几乎是）肯定正确的同时，为例外的情况提供脚释或线索。

目前，有多种流行文件扩展名，用于区别 C++头文件和 C++实现文件。例如：

头文件扩展名：　.h　.hxx　.H　.h++　.hh　.hpp

实现文件扩展名：.c　.cxx　.C　.c++　.cc　.cpp

在所有的例子中，我们都使用.h 扩展名来标识 C++头文件，使用.c 扩展名来标识 C++实现文件。在本书中，我们会频繁地称头文件为.h 文件，称实现文件为.c 文件。最后需要说明的是，本书正文中的所有示例都在 SUN SPARC 工作站的 CFRONT 3.0（SUN 版本）上编译过并校验过语法；同时也在 HP700 系列机的自备 C++编译器上进行过上述测试。当然，出现的任何错误都只能由作者负全责。

阅读导航

本书包含很多内容。不是所有的读者都有着相同的知识背景。所以我在第 1 章中提供了一些基本的（但却是必备的）知识以铺平学习道路。专业的 C++程序员可以略过这一部分，或者需要的话简单地参考一下。第 2 章包含了一些基本的软件设计规则，我希望每一位有经验的开发人员都能够很快地认可。

第 0 章：引言

概述大型 C++软件的开发人员所面临的问题。

第 1 部分：基础知识

第 1 章：预备知识

回顾了基本语言信息、一般设计模式以及本书的文体惯例。

第 2 章：基本规则

介绍了在任何 C++项目中都应该遵循的重要设计经验。

后面的内容分为两大部分。前一部分题为“物理设计概念”，介绍了一系列与大型系统物理结构相关的重要论题。这些章节中的内容（第 3~7 章）集中在编程方面，对许多读者来说将是全新的，并且只精选了适合于大型程序设计的内容。这部分的论述是“自底向上”的，每一章的叙述都承接了前一章的内容。

第 2 部分：物理设计概念

第 3 章：组件

介绍一个系统的基础物理构筑模块。

第 4 章：物理层次结构

阐述建立组件层次结构的重要性，这些组件在物理上没有循环依赖，便于测试、维护和重用。

第 5 章：层次化

减少连接时依赖的特殊技术。

第 6 章：绝缘

减少编译时依赖的特殊技术。

第 7 章：包

将上述技术扩展到更大型的系统。

后一部分题为“逻辑设计问题”，研究逻辑设计与物理设计相结合的传统课题。这些章节（第 8~10 章）叙述了如何将一个组件作为一个整体来设计，总结了大量有关合理接口设计的问题，并研究了在大型项目环境中的实现问题。

第 3 部分：逻辑设计问题

第 8 章：构建一个组件

全面设计组件需要考虑的重要问题的总括。

第 9 章：设计一个函数

详细探讨生成一个组件的功能接口的问题。

第 10 章：实现一个对象

针对在大型项目环境中实现对象的若干组织问题。

附录中的主题是整本书的参考。

正文页下注中给出的书目的有关信息可再参阅书后的参考文献。

感谢

如果没有我在 Mentor Graphics 公司的许多同事的共同努力，这本书是不可能出现的。他们对于公司的划时代建设和发展作出了贡献。

首先，我要感谢我的朋友、同事和大学同学 Franklin Klein 对本书所作出的贡献。他实际上审读了本书每一页手稿。Franklin 对许多概念提供了解释——对大多数软件开发人员来说，这些都是新的概念。Franklin 的智慧、学识、社交能力、领会有效交流的细微差别的能力都远远超过我。他对内容修订、叙述顺序和表达风格等方面进行了详细的评审。

在排版期间，数位有天分和献身精神的专业软件人员检查了本书的大部分内容。他们愿意花费宝贵的时间审阅本书，对此我感到很幸运。我要感谢 Brad Appleton、Rick Cohen、Mindy Garber、Matt Greenwood、Amy Katriel、Tom O'Rourke、Ann Sera、Charles Thayer 和 Chris Van Wyk。他们花费大量的精力帮助我尽可能提高本书的价值。我要特别感谢 Rick Eesley，因为

他提供了丰富的评论和实际的建议——尤其是他建议在每一章的结尾作一个小结。

许多专业软件开发人员和质量保证工程师审阅了个别的章节。我要感谢 Samir Agarwal、Jim Anderson、Dave Arnone、Robert Brazile、Tom Cargill、Joe Cicchiello、Brad Cox、Brian Dalio、Shawn Edwards、Gad Gruenstein、William Hopkins、Curt Horkey、Ajay Kamdar、Reid Madsen、Jason Ng、Pete Papamichael、Mahesh Ragavan、Vojislav Stojkovic、Clovis Tondo、Glenn Wikle、Steve Unger 以及 John Vlissides，他们在技术上作出了贡献。我也要感谢 Mentor Graphics 公司的 Lisa Cavaliere-Kaytes 和 Tom Matheson，他们为书中的一些图表提出了宝贵的建议。另外我还要感谢 Eugene Lakos 和 Laura Mengel 所作的贡献。

自从本书首次印刷以来，我要感谢以下读者帮助我排除了一些我要负全责的不可避免的错误：Jamal Khan、Dat Nguyen、Scott Meyers、David Schwartz、Markus Bannert、Sumit Kumar、David Thomas、Wayne Barlow、Brian Althaus、John Szakmeister 以及 Donovan Brown。

如果不是在哥伦比亚大学收到了一封推销信，提供给我一份免费的有关 Rob Murray 的著作的评论拷贝，这本书也许永远也不会出现。因为我只在春季学期才教课，所以在我寄回所附表格的时候，要求将那本书寄到 Mentor Graphics 而不是哥伦比亚。之后不久，我接到了 Pradeepa Siva（她是 Addison-Wesley Corporate & Professional Publishing Group 的）打来的电话，要确定我这个不寻常要求的原因。在使她相信了这个要求的合理性（也许有一些是没有理由的夸大）之后，她说："我想我的老板可能会和你谈谈。"几天之后，我见到了她的老板——出版商。我一直很欣赏该出版社制作的"专业计算机丛书"的卓越品质，正是这种声誉最终使我答应为那套丛书撰写本书。

我非常感谢 Addison-Wesley Corporate & Professional Publishing Group 的成员。出版商 Join Wait，耐心地教给我许多关于人和交流的见识，这些东西将使我终生受益。从阅读大量书籍、与很多专业软件技术人员进行讨论、到站在书店观察潜在读者的购买习惯等方面，John Wait 一直尽力把握行业的脉搏。

以 Marty Rabinowitz 为首的产品人员在各方面都是优秀的。尽管与其他出版商联系的学术界的作者们对我表示担心，但是我仍然高兴地对待 Marty 所提出的在表达作者思想时应做到技术上准确、容易使用以及美学上有魅力的提炼与加工。我特别要感谢 Frances Scanlon，为了排印这本书，她付出了艰苦不懈的努力。

Brian Kernighan，本系列的技术编辑，在风格和实质上都提供了有价值的贡献，并发现了一些印刷错误和没有被其他人发现的不一致性。他的知识的广度和深度，与简洁的写作风格结合起来，对本系列丛书的成功作出了很大的贡献。

最后，因为对其他书中基础逻辑概念和设计原则的引证，我还想感谢本系列中的其他作者。

译 者 序

C++是当前的主流技术之一。随着我国社会与经济信息化工作的发展和深入，计算机应用水平和层次正在逐步提高，应用软件系统日益复杂化和大型化，我国软件业将面对越来越多的大型软件开发问题。然而，目前国内这一方面的书籍极少，而直接基于 C++大型软件开发的资料尤其缺乏。针对这一情况，我们翻译了《大规模 C++软件设计》一书，以满足读者的需求。

在实施大型和超大型 C++软件工程时，会出现许多意想不到的严重问题。若方法不当，轻则费用失控、进度延迟、执行代码臃肿、低效，重则可能导致开发项目完全失败。作者在总结多年来从事 C++大型工程的经验基础上，提出了物理设计和逻辑设计的一些新概念和新理论，阐明了在从事大型和超大型 C++软件工程时应遵循的一系列物理设计和逻辑设计原则，讨论了设计具有易测试、易维护和可重用等特性的高质量大规模 C++软件产品的方法。在说明这些概念、理论、方法和原则时，既有定性描述又有定量分析，还有许多生动的实例，并几对如何设计复杂系统进行了示范。书中的概念、理论、方法和原则对于大型软件工程的开发具有极强的现实指导意义，有利于提高软件的质量，保证大型项目的成功。作者 John Lakos 曾在 Mentor Graphic 公司（最早用 C++成功开发大型软件的公司之一）长期承担用 C++开发 CAD 软件的工作，书中积聚了他十余年的实践功力。作者从 1990 年起在美国哥伦比亚大学开设名为“面向对象设计与编程”的研究生课程，使本书中提及的方法和实例经受了课堂实践检验，并得到了进一步的总结提炼和理论升华。

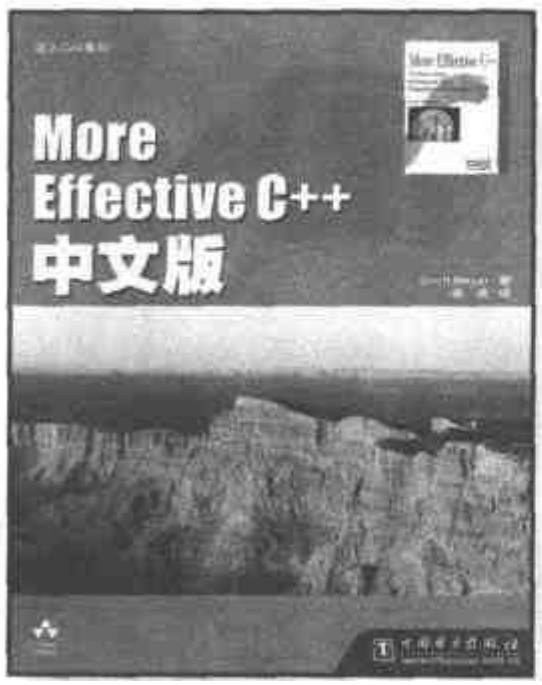
本书虽然主要面向有经验的 C++程序员、系统设计师、软件质量保证人员等中、高级程度的读者，但书中对于 C++面向对象程序设计的重要概念、设计模式与良好的编程风格的精彩再现，以及对于在任何 C++项目中都应该遵循的重要设计原则和指导方针的深刻揭示，无疑亦可以使初级程度的读者大受裨益。即使是针对大规模软件设计的一些理念和方法，亦无妨为中、小型软件开发所参考或借用。也许正是由于上述缘故，本书尽管定位于大规模软件设计这一高端问题，但并未因此而曲高和寡，其英文版问世至今，读者甚众，短短数年，重印已达九次之多。

参加本书翻译的有李师贤、明仲、曾新红、刘显明、王志刚、李皓、李智、梅晓勇、李

宏新、杜云梅、邱璟、孙恒、徐晶、舒忠梅、章远和熊春玲等。李师贤对全书进行了译校和审定。明仲初译了前言及偶数章节。曾新红初译了奇数章节。限于译校者水平所限，译文中的谬误之处在所难免，恳请读者批评指正。

译　者

深入 C++ 系列



侯捷译著

- 影响深远的经典著作
- 35 条专家经验
- 条条精彩

- 最具权威性的 STL 书籍
- 侯捷评语："另辟捷径"

经典著作　不容错过　循序渐进　可收宏效

- C++ 最佳入门书籍
- 快速掌握 C++ 的全新方法
- 单刀直入 C++ 核心部分

- C++ 最新教程
- C++ 最佳参考书
- 名著名译
- 相得益彰

- comp lang c++ 精华荟萃
- C++ 程序员进阶必备

- 斯坦福大学教授的呕心之作
- 与《Art of Computer Programming》齐名的算法巨著

- 经典权威的 C++ 进阶图书
- C++ 设计思想的精彩汇总
- 原汁原味的理论接触

- 著名网络专栏 Guru of the Week 的精华荟萃
- 47 条专家经验
- 条条精彩

目　　录

第 2 部分 物理设计概念

第 3 部分 逻辑设计问题

0 引言

开发出好的 C++程序并非易事。当项目很大时，用 C++开发高可靠、易维护的软件则更加困难，并且将引出许多新概念。就像建筑师建造摩天大楼时不能使用从建造单个家庭房屋取得的经验一样，许多从C++小型项目中学到的技术和方法也不能简单地用于大型项目的开发。

本书讨论的是如何设计非常大型的、高质量的软件系统，也是专门为有经验的软件开发者而编写的。本书可以指导他们构造出易维护、易测试的软件结构。本书不是一本有关程序设计的理论方法的书，而是引导开发者走向成功的完全的、实际的指导。它是精通 C++的程序员开发大型多节点系统的多年经验的总结。我们将示范如何设计复杂系统。这些复杂系统需要数百个程序员参加，有数千个类并且其源代码可能有数百万行。

本章将讨论在使用 C++开发大型项目时会遇到的一些问题，并为在前几章中我们必须做的基础工作提供环境知识。引言中用到的几个术语还没有定义，大多数术语应根据上下文理解。在后面的章节中，会给出这些术语的更精确的定义。真正的高潮将出现在第 5 章，我们将应用特殊的技术来减少 C++系统中的耦合（即相互依赖的程度）。

0.1 从 C 到 C++

在控制大型系统的复杂性方面，人们已充分认识了面向对象风范的潜在优点。在写作本书时，每 7 到 11 个月，C++程序员的数量就翻了一番[①]。在有经验的 C++程序员的手中，C++是人类发挥技能和工程才干的强有力的工具。然而，如果认为在大型项目中，只要使用 C++就能确保成功，那就完全错了。

① **stroustrup94**，7.1 节，163~164 页。

C++并不只是 C 的扩充：C++支持一种全新的风范。由于面向对象风范比对应的面向过程风范需要更多的努力和悟性，所以其名声并不好。C++比 C 更难掌握，而且有无数情形令程序员陷入困境。常常会出现这样的情况，即程序员会忽略一些严重的错误，等到他们发现了想要弥补这些错误，并使项目仍能满足进度要求时，却已经太晚了。即使相当小的失误，例如不加区分地使用虚函数或通过值来传递用户自定义类型，在完全正确的 C++程序中，也可能导致其运行速度比完成同样功能的 C 程序慢十倍。

许多程序员在开始接触 C++时，会经历这样的过程，在这个过程中，编程效率大大降低甚至陷入停顿，因为似乎要探索无数的设计方案。在此过程中，传统的程序设计员会深感不安，因为他们想要竭力掌握“面向对象”概念。

尽管对于最有经验的专业 C 程序员来说，C++的规模和复杂度在开始时都有点难以承受，但一个能干的 C 程序员要写出一个小的但不是微不足道的 C++程序，并不需要花太长的时间。然而，这种用 C++创建小型程序的未经训练的技术，来完成大型项目是完全不能胜任的。也就是说，C++技术的不成熟应用不适合大型项目，不谙此道所造成的后果甚多。

0.2 用 C++开发大型项目

就像 C 程序一样，一个很差的 C++程序可能会非常难以理解和维护。如果接口没有完全封装，则很难对其实现进行调整或加强。不良的封装会妨碍重用，也可能导致可测试性方面的优势完全消失。

与主流观点相反，从根本上说，最普通形式的面向对象程序要比对应的面向过程的程序更难测试和校验[1]。通过虚函数改变内部行为的能力可能使类不变式（class invariant）无效，而对于程序的正确性来说，类不变式是必要的。此外，一个面向对象系统的控制流路径的潜在数量可能十分巨大。

幸运的是，我们没有必要写这样霸道的（也是不好测试的）面向对象程序。可靠性可以通过限制只使用面向对象风范中的一个更容易测试的子集来获得。

当程序变得更大时，一种本质不同的力量会起作用。下面的小节会介绍一些我们可能会遇到的问题的具体例子。

0.2.1 循环依赖

作为一个软件专业人员，可能面临过这样一个局面：第一次看一个软件系统，似乎找不到一个合理的起点或者自身有完整意义的系统片段。不能理解或不能单独使用系统的任何

① **perry**，13~19 页。

部分就是一种循环依赖设计的征兆。C++对象有一种显著的互相混杂的倾向。图 0-1 描述了这种潜在而危险的紧密物理耦合形态。电路（circuit）是元件（elements）和电线（wires）的集合。因此，类 Circuit 知道 Element 和 Wire 的定义。Element 知道它所属的 Circuit，而且能分辨出是否与特定 Wire 相连。因此类 Element 也了解 Circuit 和 Wire。最后，一根电线可以连接到一个元件或一个电路的末端。为了完成这种连接，类 Wire 必须访问 Element 和 Circuit 的定义。

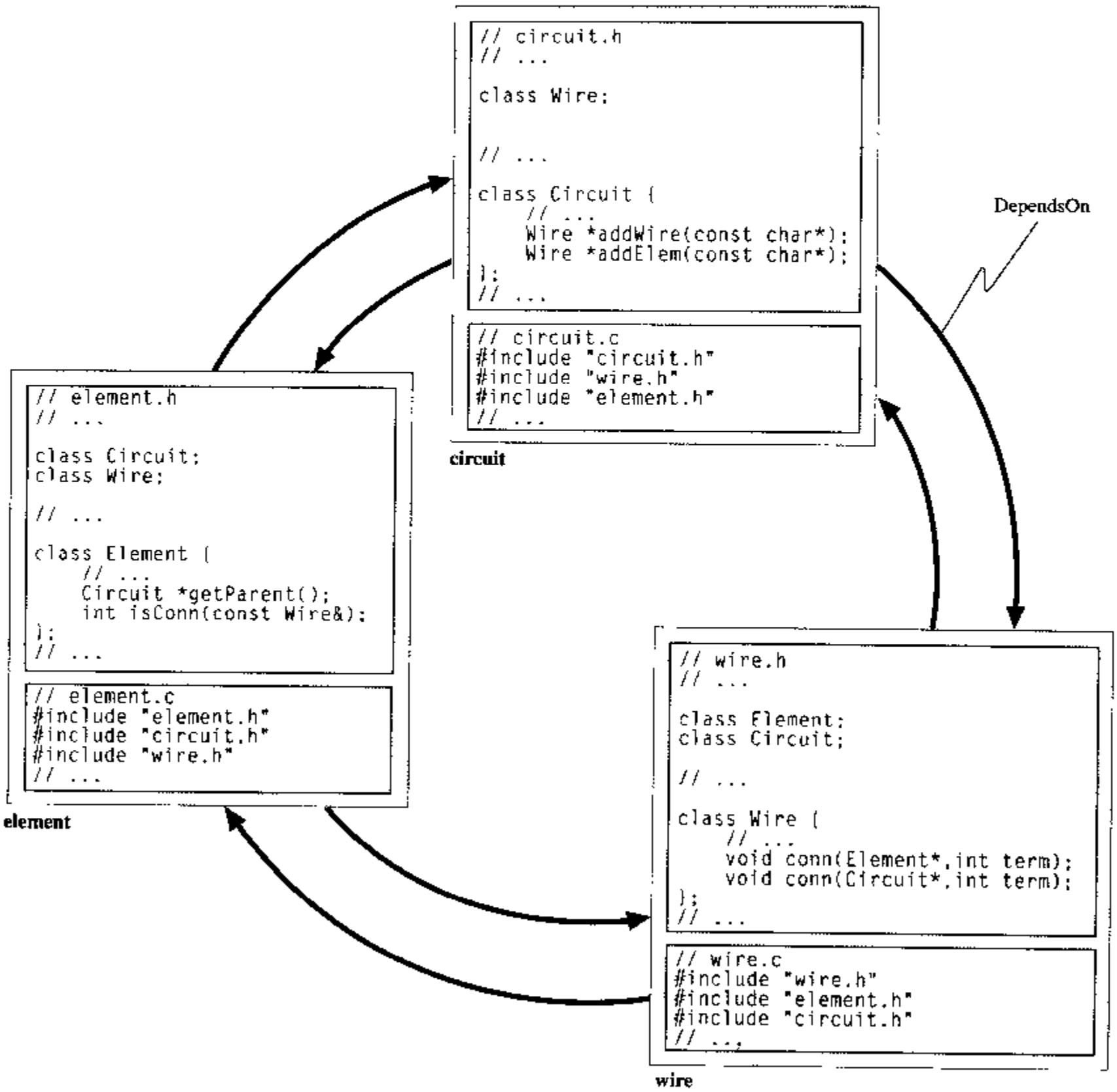


图 0-1　循环依赖组件

这三种对象类型的定义存在于单独的物理组件（编译单元）中，以提高模块化程度。虽然这些单个类型的实现被它们的接口完全封装，然而，每个组件的.c 文件都必须包含其他两个组件的头文件。这三个组件的最终依赖图是循环的。就是说，没有一个组件可以在没有其他两个组件的情况下单独使用甚至单独测试。

草率构建的大型系统会因为循环依赖而变得紧密耦合，从而强烈地抗拒分解。支持这样的系统可能是一场噩梦，通常不可能对其进行有效的模块测试。

一个适当的案例是一种电子设计数据库的早期版本。那时，它的作者没有意识到在物理设计中避免循环依赖的必要性，其结果是创建了一个相互依赖的文件集合，其中包含了数千个函数的数百个类，最终导致没有办法使用甚至测试，而只能把它当作一个单一的模块。这个系统的可靠性非常差，并且不适于扩展和维护，最终不得不抛弃它而重新编写。

比较起来，层次化物理设计（即没有循环相互依赖）相对更容易理解、测试和重用。

0.2.2 过度的连接时依赖

如果读者尝试过连接库中的少量功能程序，就会发现连接时间相对于所希望带来的利益来说已经不成比例地提高了，于是我们可能会尽量重用“重量级的组件”而非“轻量级的组件”。

使用对象的好处之一是在需要时很容易将遗漏的功能加进到对象中。面向对象风范的这种儿近诱惑的特性，使得许多谨慎的开发者将精练的、审慎考虑过的类变成巨大的“恐龙”，将数量巨大的代码合为一体，而绝大多数客户不会使用其中的大部分代码。图 0-2 描述了扩充一个简单类 String 的情形，即将简单类 String 扩充为满足所有客户需要的类。每一个客户加进一个新的特性，潜在地增加了实例体积、代码容量、运行时间以及物理依赖，从而增加了其他客户的开销。

C++程序常常比必需的大。如果不谨慎，一个 C++的可执行程序的大小可能要比同样功能的 C 程序大得多。将外部依赖忽略不计，雄心勃勃的类开发者已创建了一些复杂的类，这些类直接或间接地依赖数量巨大的代码。一个用精心制作的 String 类编写的“Hello World”程序竟产生了 1.2MB 的执行容量！

像 String 类这样的“超重”类型不仅会增加可执行程序的大小，而且会使连接过程过度缓慢和痛苦。如果连接 String（连同它所有的实现依赖）所需要的时间大于连接子系统所用的时间，那么人们就不太可能费力地去重用 String。

幸好现在已经有了避免各种形式的不必要的连接时依赖的技术。

```
// str.h
#ifndef INCLUDED_STR
#define INCLUDED_STR

class String {
    char *d_string_p;
    int d_length;
    int d_size;
    int d_count;
    // ...
    double d_creationTime;

  public:
    String();
    String(const String& s);
    String(const char *cp);
    String(const char c);
    // ...
    ~String();
    String &operator=(const String& s);
    String &operator+=(const String& s);
    // ...
    // (27 pages omitted!)
    // ...
    int isPalindrome() const;
    int isNameOfFamousActor() const;
};

// ...

#endif
```

```
// str.c
#include "str.h"
#include "sun.h"
#include "moon.h"
#include "stars.h"
// ...
// (lots of dependencies omitted)
// ...
#include "everyone.h"
#include "theirbrother.h"
String::String()
: d_string_p(0)
, d_length(0)
, d_size(0)
, d_count(0)
// ...
// ...
// ...
```

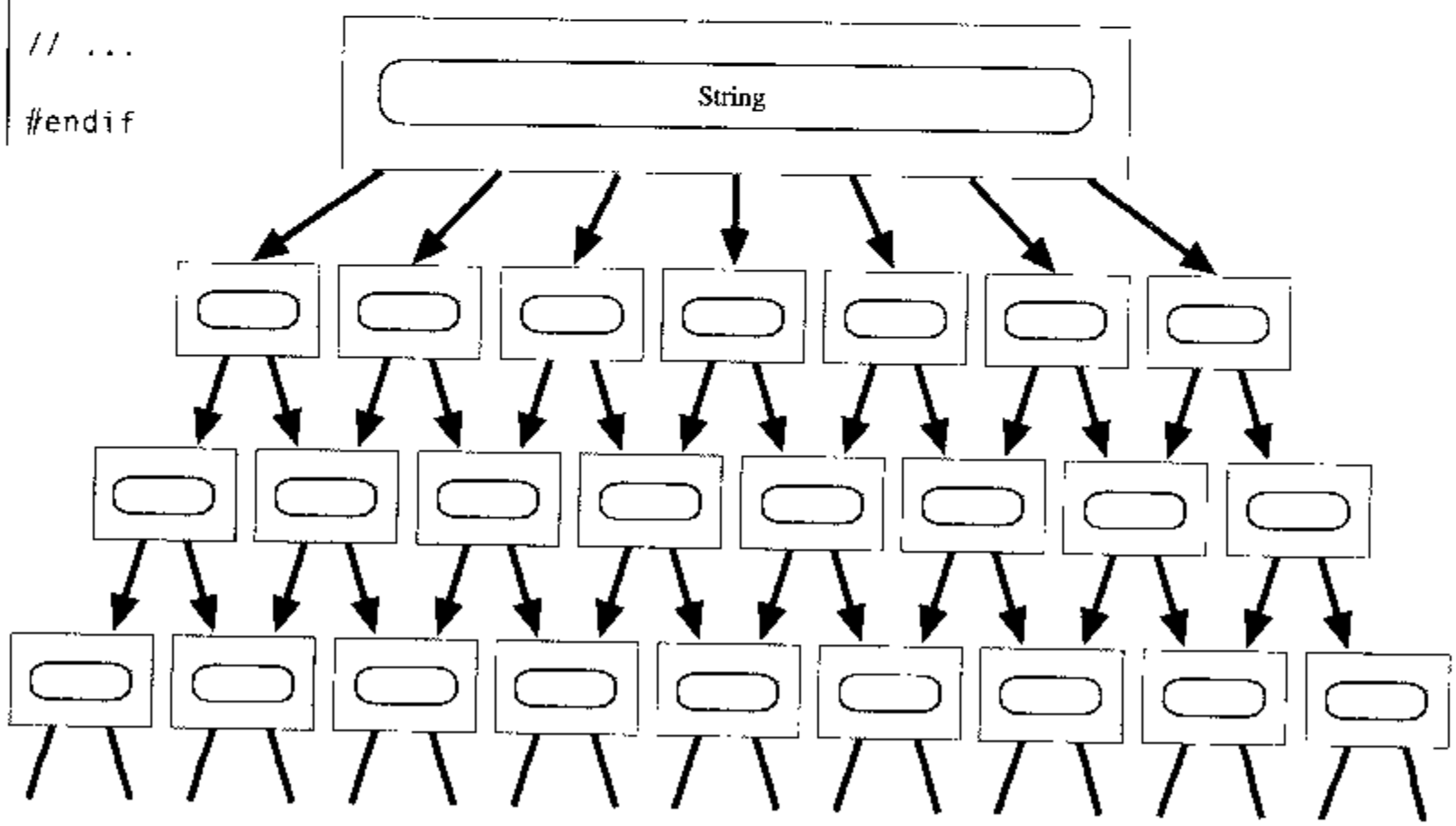


图 0-2　特大的、重量级的、不可重用的 String 类

0.2.3 过多的编译时依赖

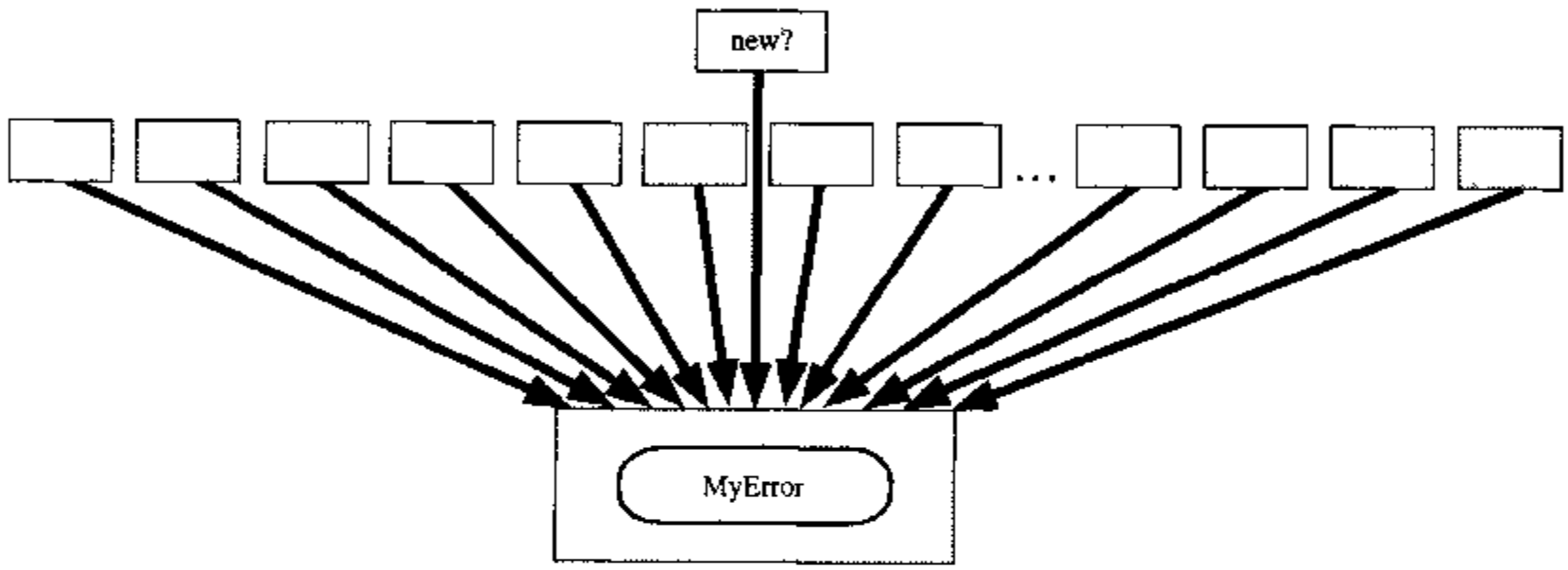
如果读者曾经尝试过用 C++开发一个多文件程序，那么就会知道改变一个头文件可能会潜在地引起多个编译单元要重新编译。在系统开发的最开始阶段，若一个修改会导致整个系统重新编译，则意味着增加毫无意义的负担。然而，当继续开发系统时，改变一个低级头文件的想法会变得越来越令人厌烦。不仅重新编译整个系统的时间会增加，甚至编译单个编译单元的时间也会增加。我们迟早会因为重编译的消耗太大而完全拒绝修改低级类。如果这些事情我们觉得熟悉，那么我们可能已经经历了过度的编译时依赖。

过度编译时耦合，对于小型项目毫无影响，对于较大型的项目却可能成为支配开发时间的重要因素。图 0-3 列举了一个过度编译时耦合的普通例子：开始时好像是一个好主意，随着系统容量的增长而变得糟糕。“myerror”组件定义了一个 MyError 结构，其中列举了所有可能的错误代码。每一个加进来的新组件都自然地包含这个头文件。但是，每一个新组件都可能有它自己的错误代码，这些代码并未标识在主列表中。

```
// myerror.h
#ifndef INCLUDED_MYERROR
#define INCLUDED_MYERROR

struct MyError {
    enum Codes {
        SUCCESS = 0,
        WARNING,
        ERROR,
        IO_ERROR,
        // ...
        READ_ERROR,
        WRITE_ERROR,
        // ...
        // ...
        BAD_STRING,
        BAD_FILENAME,
        // ...
        // ...
        CANNOT_CONNECT_TO_WORK_PHONE,
        CANNOT_CONNECT_TO_HOME_PHONE,
        // ...
        // ...
        MARTIANS_HAVE_LANDED,
        // ...
    };
};

#endif
```

图 0-3 潜伏的编译时耦合源代码

当组件的数量更多时，我们将自己的错误信息代码加入到清单中的愿望就没有那么强烈了。我们对重用已有的错误信息代码感兴趣，因为这些代码也许只要大概适合就可以，而不必修改 myerror.h 文件。最终，我们将放弃任何添加新的错误信息代码的想法，只是简单地返回 ERROR 或 WARNING 而不是修改 myerror.h 文件。到这个时候，我们就会陷入这样一个境地：所作的设计变得不可维护并且实际上已毫无用处了（参见下图）。



还有许多其他原因会引起不必要的编译时依赖。一个大型 C++程序会比同样的 C 程序有更多的头文件。一个文件包含不必要的头文件，是造成 C++中过多耦合的常见原因。例如，在图 0-4 中，在 simulator 头文件中本来没有必要包含对象的定义，只是因为 simulator 类的客户可能会发现这些定义有用处才这样做。这就迫使客户程序在编译时依赖于组件，不管这些组件实际用到与否。过多地包含（伪）指令，不仅会增加编译客户程序的开销，而且可能增加由于在较低的层次上改变了系统而必须重新编译客户程序的可能性。

如果忽略编译时依赖，有可能使得系统中的每一个编译单元包含系统中的几乎每一个头文件，从而使编译速度降低到跟爬行一样。最早真正大型的 C++项目（准确点说，数千人年的工作量）之一是 Mentor Graphics 公司开发的 CAD 框架产品。开发者最初不知道“编译时依赖”是如何阻碍他们的工作的。即使使用大型的网络工作站重新编译整个系统，也需要花一周的时间。产生这个问题的原因是组织的细节问题。图 0-4 中显示的 simulator 组件部分地说明了这个细节问题。“装饰（cosmetic）”技术可以缓解这个问题，但是真正的解决方案是消除不必要的编译时依赖。

就像连接时依赖一样，也有若干消除编译时依赖的特定技术。

0.2.4 全局名称空间

如果读者曾参加过多人合作的 C++项目，则一定知道软件集成常常会出现意想不到的事。特别是增加全局标识符很成问题。其中一个明显的危险是这些名字可能会冲突。结果造成分别开发的系统部件不经修改就无法集成。对于有数百个头文件的更大的项目来说，甚至很难

找到全局变量名的声明。

```
// simulator.h
#ifndef INCLUDED_SIMULATOR
#define INCLUDED_SIMULATOR
#include "cadtool.h"                    // required by "IsA" relationship
#include "myerror.h"                    // bad idea (see Section 6.9)
#include "circuitregistry.h"            // unnecessary compile-time dependency
#include "inputtable.h"                 // unnecessary compile-time dependency
#include "circuit.h"                    // required by "HasA" relationship
#include "rectangle.h"                  // unnecessary compile-time dependency
// ...
#include <iostream.h>                   // unnecessary compile-time dependency

class Simulator : public CadTool {      // mandatory compile-time dependency
    CircuitRegistry *d_circuitRegistry_p;
    InputTable& d_inputTable;
    Circuit d_currentCircuit;           // mandatory compile-time dependency
    // ...
  private:
    Simulator(const Simulator &);
    Simulator& operator=(const Simulator&);
  public:
    Simulator();
    ~Simulator();
    // ...
    MyError::Code readInput(istream& in, const char *name);
    MyError::Code writeOutput(ostream& out, const char *name);
    MyError::Code addCircuit(const Circuit& circuit);
    MyError::Code simulate(const char *outputName,
                           const char *inputName,
                           const char *circuitName);
    Rectangle window(const char *circuitName) const;
    // ...
};

#endif
```

图 0-4 不必要的编译时依赖

例如，我曾经使用过由数千个头文件组成的对象库。我曾想在文件作用域中把类型 TargetId 的定义寻找出来，因为 TargetId 看起来像是类定义（但实际不是）：

```
TargetId id;
```

我记得曾用“grep”[①]在数千个头文件中寻找该定义。但只收到了一个信息，大意是“有

① “grep”是 Unix 中的查找工具程序。

太多的文件”。最终，我在 shell 命令文件中使用嵌套的“grep”命令，基于首字母将头文件分拆，以便将问题简化为 26 个大小可控制的子问题。我最终找到了我要寻找的那个“类”，实际上它并不是类。它也不是 struct 或 union！正如图 0-5 说明的那样，搜索出来的结果是，类型 TargetId 实际上是在文件作用域内为一个 int 所作的 typedef 声明！

该 typedef 声明给全局名称空间引入了一个新的类型名。而没有任何迹象表明它是 int 类型，也没有暗示在哪里能找到其定义。

```
// upd_system.h
#ifndef INCLUDED_UPD_SYSTEM
#define INCLUDED_UPD_SYSTEM

typedef int TargetId;           // bad idea!
class upd_System {
    // ...
  public:
    // ...
};

#endif
```

图 0-5　不必要的全局名称空间污染

如果 typedef 的声明已嵌入到一个类的内部（如图 0-6 所建议的那样），那么可以通过该类来引用 typedef 声明（或者可以继承该声明），可直截了当地写为如下形式：

```
upd_System::TargetId id;
```

遵循上面所建议的简单规则，可以使冲突的可能降到最低，同时使得在大系统中寻找逻辑实体更加容易。

```
// upd_system.h
#ifndef INCLUDED_UPD_SYSTEM
#define INCLUDED_UPD_SYSTEM

class upd_System {
    // ...
  public:
    typedef int TargetId;       // much better!
    // ...
};

#endif
```

图 0-6　在类作用域内的 typedef 是容易找到的

0.2.5 逻辑设计和物理设计

大多数有关 C++的书仅阐述逻辑设计。逻辑设计是指那些属于类、运算符、函数等语言结构的设计。例如，一个特殊的类该不该有一个拷贝构造函数，是一个逻辑设计问题。决定一个特定操作（例如：operator==）应该是一个类的成员函数还是一个自由函数（也就是说，非成员函数），也是一个逻辑问题。甚至选择一个类的内部数据成员的类型，也属于逻辑设计。

C++支持完全的逻辑设计。例如，**继承**是面向对象必不可少的成分。另一个成分是**分层**（layering），包括用更原始的对象来构造类型，并经常直接嵌入到类的定义中。但是，在许多需要使用分层技术的时候，有很多人使用了继承。例如：一个 Telephone 不是一种 Receiver、Dial 或 Cord；而应由这些原始部件构成（或“分层于其上”）。

这种对事物的错误判断可能导致时间和空间上的低效率，并可能使结构的语义模糊，从而使整个系统更难以维护。有经验的 C++程序员知道何时使用或不使用某种特定的语言结构，这是他们成为很有用的人的原因之一。

逻辑设计并不涉及“将一个类定义放置在什么地方”的问题。从纯逻辑观点来看，所有在文件作用域中的定义都存在于单一的名称空间的同一层次上，这些定义之间没有界限。相对于类成员的定义来说，类在何处定义和是否支持自由运算符的问题与逻辑设计并不相干。重要的是这些逻辑实体如何组织在一起形成一个能工作的程序。由于将整个程序看作一个单元，所以没有单独的物理依赖的概念。程序作为一个整体只依赖于自己。

有若干关于逻辑设计的好书（参见参考文献）。但是，也有不少这些书未涉及的问题，这些问题只有当程序变得很大时才出现。这是因为与成功的大型系统设计相关的很多内容不属于逻辑设计的范畴，本书把它们作为物理设计。

物理设计涉及的问题包括与系统的物理实体有关的问题（如文件、目录和库等）以及组织问题（如物理实体之间的编译时依赖和连接时依赖等）。例如，使类 Telephone 的成员函数 ring()成为内联函数，将迫使 Telephone 的所有客户不仅要查看 ring()的声明，而且要查看它的定义，才能完成编译。无论 ring()是否声明为内联函数，其逻辑行为都是一样的。受影响的是 Telephone 与其客户之间的物理耦合的程度和特性，从而使得维护使用 Telephone 的程序的开销也要受到影响。

好的物理设计不仅仅是被动地决定系统的现有逻辑实体应如何划分。物理设计常蕴涵要支配逻辑设计的输出。在逻辑领域中类之间的关系，例如 IsA、HasA、Uses，在物理领域中压缩成组件之间的单个 DependsOn 关系。另外，一个好的物理设计的依赖关系是一种没有循环的图。所以，我们要避免在组件之间暗含循环依赖关系的逻辑设计。

同时满足逻辑设计和物理设计的约束条件，有时是一种挑战。实际上，有些逻辑设计可能必须重做甚至替换掉，以满足物理设计的质量标准。然而，在我的经验中，总是有充分地满足两个领域的解决方案，尽管开始时要费一些时间来发现它们。

对于小项目来说，一个文件目录就可满足要求，因而不必太重视物理设计。但是，对于

大型项目来说，一个好的物理设计的重要性就大大提升了。对于非常大型的项目来说，物理设计是项目成功的决定性因素。

0.3 重用

面向对象设计将易重用作为自己的优点。但是，正如其他风范一样，要获得好处，就要付出代价。重用意味着耦合，而其本身的耦合是我们不希望看到的。如果若干程序员企图使用同一组件，并且不要求功能上的修改，则重用是合理和正当的。

但是，考虑这样的情形，有若干客户分别编写不同的程序，每个人都想重用一个公共组件以达到不同的目的。如果另外一些独立客户也在积极寻求增强支持，他们会发现重用的结果彼此间不一致：某一客户程序的加强可能会毁坏其他人的程序。更糟糕的是，我们最终可能得到一个对谁都无用处的“超重”的类（如图 0-2 所示的 String 类）。

重用经常是好的方案。但是为了成功地重用一个组件或一个子系统，该组件或子系统不应该与一大块不需要的代码绑在一起。也就是说，只有那些与系统其他部分没有必然联系的部分才有可能重用。

不是所有的代码都能重用。试图实现过多的功能或者为实现对象进行完全彻底的错误检查，都会增加不必要的研制和维护开销，也会增加可执行代码的大小。

实现者的以下两方面的知识对大型项目有益：何时重用代码和何时使代码能重用。

0.4 质量

质量可以从多方面来衡量。**可靠性**是传统的质量定义（也就是说，“它有错误吗？”）。一个产品容易使用并且在大多数情况下运行正常，人们通常认为它是合适的了。但是，对于某些应用来说，如航空航天领域、医学领域以及财政金融领域，软件错误的代价会非常巨大。通常，软件不能只是通过测试这一种手段来达到可靠性要求。等到我们能测试软件的时候，软件的内在质量早已形成了。不是所有的软件都能够有效地测试。要想让软件能够有效测试，必须从一开始就本着这个目标来对它进行设计。

为了易于测试而设计，虽然很少成为小项目最关注的事情，却是成功构造大型和超大型系统的最重要的事情。易测试性与质量本身一样，不可能是事后产生的想法：它必须在编写第一行代码之前就预先考虑到。

除了易测试性外，保证质量还有许多其他方面需要考虑。例如，**功能性**是指一个产品是否能完成客户所期望的工作。有时一个产品可能被用户拒收，因为它没有具备客户所期望的足够的性能。更糟糕的是，一个产品可能得“零分”：如果客户希望购买一把螺丝刀，则世界上最好的锤子也不可能通过功能性测试。在开发开始之前就有一个符合市场需求的功能规格

说明，是保证软件具有合适功能的重要的第一步。但是，本书讨论的技术是如何设计和建立一个大型系统，而不是讨论设计什么样的大系统。

可用性是质量的另一种衡量标准。一些软件产品在正确性方面做得很好，在软件的有效使用方面却做得不够。如果一个软件产品对于具有代表性的预期客户来说，使用起来太复杂、太困难、太笨拙或者令人痛苦，那么，就不会有人去使用该软件。我们所说的用户通常是指系统的最终用户。但是，在一个大型的、设计成层次结构的系统中，组件的客户很可能是别的组件。来自客户（包括别的开发者）的早期反馈对于保证可用性是非常重要的。

可维护性是衡量支持一个系统工作相对开销的指标。支持工作包括追踪错误、移植到新平台以及扩充产品的性能以满足客户期望的（甚至没有预料到的）未来需要。用 C++（或者其他语言）编写的质量低劣的系统设计，使维护的开销非常昂贵，甚至扩充软件的开销也很大。大型的可维护设计决不是碰巧得到的，它们是遵循确保可维护性原则进行项目设计的结果。

性能是衡量产品的速度和大小的指标。尽管大家都知道面向对象设计在可扩展性和重用方面有优势，但是，面向对象风范的某些方面，如果使用不当，可能会使程序的运行速度更慢，并且需要更多的内存。如果我们的代码运行速度比竞争对手的产品慢，并且需要更多的内存，我们的产品就会销售不出去。例如，在一个文本编辑器中，将每一个字符都建模为一个对象，尽管在理论上很吸引人，但，如果我们希望达到最佳的时间/空间性能[①]，这就是一个不恰当的设计。试图用一个用户自己定义的类（例如 BigInt 类）去替代很常用的类型（例如 int），必然会降低性能。如果我们一开始没有强调性能目标，我们就可能采纳没有达到性能目标的系统结构或编程方法，除非我们重写整个系统。是否知道哪些地方可以不必太精细以及如何进行性能折衷，是区别软件工程师和单纯程序员的标准。

质量的各个方面对于产品的整体成功都很重要。质量的各方面良好性能的获得有一个共同点：必须在项目的一开始就考虑质量的各个方面。设计一旦完成，就无法再提高质量了。

0.4.1 质量检测

质量保证部（Quality Assurance，QA）一般是一个公司内负责测试产品质量已达到某种标准的部门。获得高质量软件的一大障碍是 QA 经常只在软件开发过程的后期，即已经造成了损害以后才参与。QA 通常不能影响软件产品的设计。QA 很少参与低层次的项目设计决策。QA 所执行的测试通常是在最终用户层次上的，并且依赖于开发者自己来进行任何低层次的回归测试。

在这种平淡无奇的过程模型中，由工程师开发出最初的软件，然后把它丢给 QA。这样的

① 参见 Flyweight pattern （**gamma**，第 4 章，195~206 页），那儿对这个特殊种类的性能问题有一个聪明的解决方案。

软件往往文档质量低劣，难以理解和测试，且不可靠。现在，QA 希望以某种方式一点点地提高软件质量。但是，如何提高呢？多次实践表明，在大型项目中，用上述模型来获得高质量软件的效率很低下。我们现在建议用另一个不同的模型。

0.4.2 质量保证

QA 必须成为开发过程中必不可少的部分。在这个过程模型中，开发者有责任确保质量。也就是说，软件已具备了高质量，测试工程师只是去验证而已。

在这个过程模型中，QA 与开发的界限很模糊，对 QA 人员和开发人员的技术资格要求本质上是一样的。某天，一个工程师写了一个接口，可由另一个工程师检查该接口的一致性、明确性和可用性。第二天，这两个人的角色可以互换。为真正有效，必须集体配合——在软件开发时，每个成员都帮助别的成员确保设计出高质量的软件。

提供一个完整的过程模型是一个规模宏大的任务，并且远远超过了本书的范围。但是，如果要获得高质量的软件，系统结构工程师和软件开发人员在整个开发过程中都必须把质量放在首位。

0.5 软件开发工具

大型项目可以受益于许多种工具，包括浏览器、增量式连接程序和代码生成器。即使简单的工具也可能很有用。附录 C 提供了一个简单的依赖性分析器的详细描述，我曾认为该工具在我的工作中没有用处。

一些工具可以帮助减轻低劣设计的“症状”。类浏览器可以帮助分析令人费解的设计并发现逻辑实体的定义，否则这些定义会被隐藏——淹没在一个大型项目中。带有增量式连接器和程序数据库的高级编程环境，有助于封装可以完成的设计，即使它含有低劣的物理设计。但是没有工具能解决根本性问题：即固有的设计质量问题。没有一个快捷和容易的方法可获得好的质量。工具本身不能解决由低劣设计引起的基础性问题。尽管工具可以推迟这些“症状”的发作，但是没有保证质量的专门工具，也没有工具能确保设计是完全按照规格说明进行的。从根本上说，是经验、智力和规程产生高质量产品。

0.6 小结

C++是一个整体而不是 C 的一个扩充。编译单元之间的循环“连接时依赖”会使程序难以理解、测试和重用。不需要的或过多的“编译时依赖”会增加编译开销并且不利于维护。当项目很大时，采用无组织的、无规程的或幼稚的 C++开发方法，最终一定会出现这些问题。

大多数有关 C++设计的书只阐述逻辑问题（如类、函数和继承），而忽略物理问题（如文

件、目录和依赖关系)。然而，在大型系统中，物理设计的质量将影响许多逻辑设计决策输出的正确性。

重用不是无开销的。重用蕴涵耦合，而耦合不是我们所希望的。没有保障的重用应该避免。

质量的衡量标准有多个方面：可靠性、功能性、可用性、可维护性以及性能。每一方面都会影响大型项目的成功或失败。

获得高质量是一项工程的责任：从一开始就应积极追求质量。质量不是在项目大体上完成之后可以加入的东西。若要使 QA 部门有效率，则应将它作为整个设计过程中必不可少的部分。

最后，良好的工具是开发过程的重要组成部分。但是，在大型 C++系统中，工具不能弥补固有的设计质量问题。本书主要讨论如何进行软件设计以保证质量的问题。

第1部分

基础知识

本书包含了许多与面向对象设计和C++编程有关的内容。因为不是所有的读者都有相同的背景，所以，在本书的第1部分，我们讨论基础知识，以达到进行进一步讨论的一个共同起点。

第1章回顾了C++语言、基本的面向对象设计原则和概念，以及本书中所采用的标准编码和文档编制惯例。本章的目的是帮助读者拉平基础知识。作者认为这些内容的大部分对于许多读者来说是熟悉的。这里没有介绍任何新的内容。熟练的C++程序员可以选择略过本章或只在需要时参考一下。

第2章描述了一般意义的设计惯例的精简集合。大多数富有经验的软件开发者已经发现了这些惯例。坚持这里所介绍的基本规则是成功的软件设计不可缺少的部分。这些规则也可以作为本书中所介绍的高级而微妙的原则和指导方针的框架。

1 预备知识

这一章回顾 C++程序语言和面向对象分析的一些重要的方面，这些知识对于大型系统设计来说是基本的。这里没有介绍革命性的东西，但有一些材料可能是读者不熟悉的。首先，我们仔细分析了多文件程序、声明与定义，以及在头（.h）文件和实现（.c）文件上下文中的内部连接和外部连接，然后研究了 typedef 声明和 assert 语句的使用。在介绍了一些有关命名习惯和类成员设计的风格问题之后，我们又研究了**迭代器**（iterator）——最常用的面向对象设计模式之一。在本章结尾，我们充分讨论了用于整本书的逻辑设计符号。对继承和分层也进行了简单的讨论。最后是一个关于在接口中进行最小化的建议。

1.1 多文件 C++程序

对于所有（除了最小的）程序来说，将整个程序都置于单个文件中既不明智也不实用。首先，每次修改了程序的任何一部分，都必须重编译整个程序。也不能在另一个程序中重用这个程序的任何一部分，除非把源代码拷给另一个文件。这种复制很快就会成为令维护者头疼的事。

把一个程序中紧密关联的各部分的源代码分别放在单独的文件中，可以使程序更有效地编译，同时也可以使它的局部能够在其他程序中重用。

在这一节中，我们回顾与那些从多个源文件生成的程序有关的 C++语言结构的一些基本性质。这些概念会频繁地在本书中使用。

1.1.1 声明与定义

一个声明就是一个定义，除非[①]：

① **ellis**，3.1 节，14 页。

- 它声明了一个没有详细说明函数体的函数；
- 它包含一个 extern 定义符并且没有初始化函数或函数体；
- 它是一个包含在一个类定义之内的静态类数据成员的声明；
- 它是一个类名声明；
- 它是一个 typedef 声明。

一个定义就是一个声明，除非：

- 它定义了一个静态类数据成员；
- 它定义了一个非内联成员函数。

> **定义：一个声明将一个名称引入一个程序；一个定义提供了一个实体（例如，类型、实例、函数）在一个程序中的惟一描述。**

一个声明将一个名称引入一个作用域。声明与定义的区别在于：在 C++中，在一个给定的作用域中重复一个声明是合法的。与之相比，在程序中使用的每一个实体（例如，类、对象、枚举值或函数）却只能有一个定义。例如：

```
int f(int,int);
int f(int,int);
class IntSetIter;
class IntSetIter;
typedef int Int;
typedef int Int;
friend IntSetIter;
friend IntSetIter;
extern int globalVariable;  // bad idea (global variable declaration)
extern int globalVariable;  // (see Section 2.3.1)
```

都是声明，在单个的作用域内可以重复任意次。另一方面，下面这些在文件作用域内的声明同时也是定义，所以在一个给定的作用域内出现不能超过一次，否则就会引起编译错误：

```
int x;                                  // bad idea (global variable)
char *p;                                // bad idea (global variable)
extern int globalVariable = 1;          // bad idea (global variable)
static int s_instanceCount;
static int f(int, int) { /* ... */ }
inline int h(int, int) { /* ... */ }
enum Color { RED, GREEN, BLUE };
enum DummyType {};
enum { SIZE = 100 };
enum {} silly;
const double DEFAULT_TOLERANCE = 1.0e-6;
class Stack { /* ... */ };
struct Util { /* ... */ };
union Rep { /* ... */ };
template<class T> void sort(const T** array, int size) { /* ... */ }
```

我们应该注意到，函数和静态数据成员声明是例外的，它们虽然不是定义，但是在一个类定义中也不可以重复：

```
class NoGood {
    static int i;      // declaration
    static int i;      // illegal in C++
  public:
    int f();           // declaration
    int f();           // illegal in C++
};
```

1.1.2 内部连接和外部连接

当一个.c 文件编译时，C 预处理器（cpp）首先（递归地）包含头文件，形成一个含有所有必要信息的单个源文件。然后这个中间文件（称为“编译单元”）被编译生成一个与主文件名相同的.o 文件（目标文件）。连接把在不同的编译单元中产生的符号联系起来，构成一个可执行程序。有两种截然不同的连接：内部的和外部的。连接所用的类型会直接影响到我们如何将一个给定的逻辑结构合并进我们的物理设计中。

> **定 义**：如果一个名称对于它的编译单元来说是局部的，并且在连接时不可能与其他编译单元中的同样的名称相冲突，那么这个名称有**内部连接**。

内部连接意味着对这个定义的访问被局限于当前的编译单元中。也就是说，一个有内部连接的定义对于任何其他编译单元来说都是不可见的，所以在连接过程中不能用来解析未定义的符号。例如：

```
static int x;
```

虽然定义在文件作用域内，但是关键词 static 决定了连接是内部的。内部连接的另外一个例子是一个枚举类型：

```
enum Boolean { NO, YES };
```

枚举类型是定义（不仅仅是声明），但是它们自己决不会将符号引进.o 文件。要想让有内部连接的定义影响程序的其他部分，它们必须放置在头文件中，而不是在.c 文件中。

有内部连接定义的一个重要例子是一个类的定义。类 Point（见图 1-1）的描述是一个定义，不是声明；因此，它不能在同一作用域内的一个编译单元中重复出现。如果类要在单个编译单元之外使用，那么它们必须定义在一个头文件中。内联函数定义（就像显示在图 1-1 底部的对 operator==的定义）是有内部连接定义的另一个例子。

> **定 义**：在一个多文件程序中，如果一个名称在连接时可以和其他编译单元交互，那么这个名称就有**外部连接**。

```
class Point {
    int d_x;
    int d_y;

  public:
    Point (int x, int y) : d_x(x), d_y(y) {}          // internal linkage
    int x() const { return d_x; }                     // internal linkage
    int y() const { return d_y; }                     // internal linkage
    // ...
};                                                    // internal linkage

inline int operator==(const Point& left, const Point& right)
{
    return left.x() == right.x() && left.y() == right.y();
}                                                     // internal linkage
```

图 1-1　一些有内部连接的定义

外部连接意味着该定义不仅仅局限于单个编译单元中。有外部连接的定义可以在.o 文件中产生外部符号，这些外部符号可以被所有其他的编译单元访问，用来解析它们未定义的符号。这种外部符号必须在整个程序中是惟一的，否则这个程序不能被连接。

非内联成员函数（包括静态成员）有外部连接，非内联函数、非静态自由函数（即 nonmember 函数）也一样。有外部连接的函数的例子见图 1-2。

```
// non-inline member function:
Point& Point::operator+=(const Point& right)
{
    d_x += right.d_x;
    d_y += right.d_y;
    return *this;
}                                                 // external linkage

// non-inline free function:
Point operator+(const Point& left, const Point& right)
{
    return Point(left.x() + right.x(), left.y() + right.y());
}                                                 // external linkage
```

图 1-2　一些有外部连接的函数的定义

注意，我们谈到非成员函数时都是指**自由函数**，而决不是指**友元函数**。一个自由函数不必是任何类的**友元**；无论如何它都应该是一个实现细节（见 3.6 节）。

在可能的地方，C++编译器会把一个内联函数的函数体直接替换成函数调用，不将任何符号引进.o 文件。有时编译器会选择放下一个内联函数的静态拷贝（因为各种原因，例如递归或动态绑定）。这个静态拷贝只将一个局部符号引入当前的.o 文件，这个符号不能与外部符号交互。

因为声明只对当前的编译单元有用，所以声明本身并不将任何东西引进.o 文件。考虑以下这些声明：

```
/* 1 */        int f();          // bad idea (see Section 2.3.2)
/* 2 */        extern int i;     // bad idea (see Section 2.3.1)
               struct S {
/* 3 */            int g();      // fine
               };
```

这些声明本身没有影响到最终的.o 文件的内容。每一个都只是命名一个外部符号，使当前的编译单元在需要的时候可以访问相应的全局定义。实际上是符号名称的使用（例如，调用一个函数）而不是声明本身导致了一个未定义的符号被引入到.o 文件。正是这个事实允许构建早期的原型：只要所缺的功能不是所需的，那么部分完成的对象可以用在运行程序中。

在前面这个例子中，三个声明中的每一个都激活对一个外部定义函数或对象的访问。我们也许会很随便，说这些“声明”有外部连接。但是还有其他种类的声明不能用来激活对外部定义的访问。我们常常会称这类声明有“内部”连接。例如：

```
typedef int Int;                        // internal linkage
```

是一个 typedef 声明。它既不能将任何符号引进.o文件，也不能通过它来访问一个有外部连接的全局对象；它的连接是内部的。一种重要的恰好有内部连接的声明是类的声明：

```
class Point;                            // internal linkage
struct Point;                           // internal linkage
union Point;                            // internal linkage
```

这些声明在把名称 Point 作为某种用户自定义类型引入时都有相同的作用；特殊的声明类型（如，class）不必和实际的定义类型（如，union）相匹配：

```
class Rep;
// ...
union Rep {
    // ...
};
```

这些声明潜在引用的定义也有内部连接；这个特性把类声明与前面所举例子中的外部声明区分开来。类声明和类定义都对.o 文件没有贡献，只是为当前的编译单元所用。

另一方面，静态类数据成员（在类定义内部声明）有外部连接：

```
class Point {
    static int s_numPoints;  // declaration of external object
    // ...
};
```

静态的类数据成员 s_numPoints（如上所示）只是一个声明，但在.c 文件中，它的定义有

“外部的”连接：

```
// point.c
int Point::s_numPoints;        // definition of external object
                               // (initialized to 0 by default)
```

注意，按照这种语言的规范，每一个静态类数据成员都必须在最终程序的某处准确地定义一次[1]。

最后，C++语言对待枚举类型和类的方式是不同的：

```
enum Point;                    // error
```

在 C++中不可能未经定义就声明一个枚举类型。就像我们将要看到的，类声明常常用来代替预处理器的#include 指令，以便可以未经定义就声明一个类。

1.1.3 头（.h）文件

C++中，将一个带有外部连接的定义放置在一个.h 文件中几乎都是编程错误。如果这样做了，还把这个头包含在不止一个编译单元中，那么把它们连接在一起时就会出错，且会出现下面这样的出错信息：

```
MULTIPLY DEFINED SYMBOL.
```

在 C++中，在一个头文件的文件作用域内放置带有内部连接的定义是合法的，但这种做法并不是人们所希望的。不仅因为这些文件作用域内的定义会污染全局名称空间，而且在有静态数据和函数的情况下，它们会在每一个包含有这个头的编译单元中消耗数据空间。甚至在文件作用域中将数据声明为 const 也会引起相同的问题，尤其是在这个常量的地址已经获得的情况下。将一个文件作用域常量（带有内部连接）和一个静态常量类成员（带有外部连接）进行比较：在整个程序中将只能有一个类作用域常量的拷贝。图 1-3 提供了一些应该和不应该归入头文件的（定义）例子。

双重的非成员数据定义这种冗余不仅会影响到程序的大小，还会影响运行性能，因为它破坏了主机的高速缓存机制。但是，有时也会有正当的理由在头文件的文件作用域中放置一个用户自定义对象的静态实例。特别是，这样一个对象的构造函数可以用来确保一个特殊的全局工具（例如 iostream）在使用前已被初始化[2]。虽然这种解决方案对于中小型系统来说是很好的，但对于非常大型的系统来说就可能有疑问。我们会在 7.8.1.3 小节中继续讨论这个问题。

---

① **ellis**，9.4 节，179 页；18.3 节，405 页。

② **ellis**，3.4 节，21~22 页。

```
// radio.h
#ifndef INCLUDED_RADIO
#define INCLUDED_RADIO

int z;                                  // illegal: external data definition
extern int LENGTH = 10;                 // illegal: external data definition
const int WIDTH = 5;                    // avoid: constant data definition
static int y;                           // avoid: static data definition
static void func() {/*...*/}            // avoid: static function definition

class Radio {
    static int s_count;                 // fine: static member declaration
    static const double S_PI;           // fine: static const member dec.
    int d_size;                         // fine: member data definition
    // ...
  public:
    int size() const;                   // fine: member function declaration
    // ...
};                                      // fine: class definition

inline int Radio::size() const
{
    return d_size;
}                                       // fine: inline function definition

int Radio::s_count;                     // illegal: static member definition

double Radio::S_PI = 3.14159265358;     // illegal: static const member def.

int Radio::size() const { /*...*/ }     // illegal: member function definition

#endif
```

图 1-3　应该和不应该归入头文件的定义

1.1.4　实现（.c）文件

有时候我们会选择定义一些函数和数据用于我们自己的实现，不希望这种实现暴露于我们的编译单元之外。有内部连接（而不是外部连接）的定义可以出现在一个.c 文件的文件作用域中，不会影响全局（符号）名称空间。在.c 文件的文件作用域中要避免的定义是：没有声明为静态的数据和函数。例如：

```
// file1.c

int i;                                          // external linkage
int max(int a, int b) { return a > b ? a : b }  // external linkage
```

上面的这些定义有外部连接，可能会与全局名称空间中的其他相似的名称之间存在潜在

冲突。因为内联和静态自由函数有内部连接，这些种类的函数可以在.c文件的文件作用域定义，不会污染全局名称空间。例如：

```
// file2.c

inline int min(int a, int b) { return a < b ? a : b }           // internal

static int fact(int n) { return n <= 1 ? 1 : n * fact(n - 1); } // internal
```

枚举类型的定义、声明为static的非成员对象以及（通过默认的）const数据定义等也有内部连接。在.c文件的文件作用域中定义所有这些实体都是安全的。例如：

```
// file3.c
#include <math.h>
class Link;                                                     // internal

enum { START_SIZE = 1, GROW_FACTOR = 2 };                       // internal

const double PI_SQ = M_PI * M_PI;                               // internal

static const char *names[] = { "Ntran", "Ptran", "NPN", "PNP" }; // internal

static Link *s_root_p;                                          // internal
Link *const s_first_p = s_root_p;                               // internal
```

其他结构，如typedef声明和预处理器宏，不会将输出符号引进.o文件。它们也可以出现在.c文件的文件作用域中，不会影响全局名称空间。例如：

```
typedef int (PointerToFunctionOfVoidReturningInt *)();

#define CASE(X) case X: cout << "X" << endl; // Classic C preprocessor

#define CASE(X) case X: cout << #X << endl;  // ANSI C preprocessor
```

typedef和宏在C++中的用处有限，如果滥用的话可能有害。我们将在1.2节和2.3.3节讨论typedef的危险性，在2.3.4节讨论宏的危险性。

1.2 typedef（类型别名）声明

一个typedef声明为一个已存在的类型创建了一个别名，而不是一个新的类型。因此，一个typedef只是提供了类型安全的假象。所以接口中的typedef可以轻而易举地做更多有害的事而不是好事。

思考一下图1-4中所示的类Person。我们已决定将typedef声明嵌入Person类中，以避免影响全局名称空间，并使它们更容易查找到。成员函数setWeight定义为接受一个以“Pounds”为单位的一个重量实参，同时方法getHeight返回以“Inches”为单位的高度。

```
// person.h
#ifndef INCLUDED_PERSON
#define INCLUDED_PERSON

class Person {
    // ...
  public:
    typedef double Inches;
    typedef double Pounds;
    //...
    void setWeight(Pounds weight);
    Inches getHeight() const;
    //...
};

#endif
```

图 1-4 typedef 不等于类型安全

很不幸，一个嵌套的 typedef 并不比一个在文件作用域中声明的 typedef 提供更多的类型安全：

```
void f (const Person& person)
{
    Person::Inches height = person.getHeight();
    person.setWeight(height);                    // ok ??
};
```

两个类型名称 Inches 和 Pounds 在结构上是相同的，因此是完全可以互换的。这些类型别名绝对不提供编译时类型安全，也使人不容易知道实际的类型。

但是，在定义复杂的函数参数时类型别名可以起作用。例如：

```
typedef int (Person::*PCPMFDI)(double) const;
```

把 PCPMFDI 声明为一个类型：指针，指向一个 const Person 成员函数，其参数为 double 类型，并返回一个 int。在跨越不同的编译器和计算机硬件时，有些数据成员的大小必须保持不变，在定义这样的数据成员时，类型别名也是很有用的（见 10.1.3 节）。

1.3 assert 语句

标准 C 库提供一个名为 assert 的宏（见 assert.h），用以保证一个给定的表达式求值为一个非零值；否则就会输出一个出错信息并且结束程序执行[①]。assert 语句使用方便，并且是开

① **plauger**，第 1 章，17~24 页。

发者的一个强有力的实现级文档工具。assert 语句就像活动的注释——它们不仅使假定清楚而准确，而且如果违反了这些假定，它们实际上会对此做某些工作。

要在运行时捕捉程序的逻辑错误，使用 assert 语句可能是一种有效的方法，并且它们很容易从产品代码中过滤出来。一旦开发结束，只需通过在编译过程中定义预处理器符号 NDEBUG，就可以消除这些为检测代码错误而进行的冗余测试的运行时开销。但是一定要记住，放在 assert 中的代码在产品版本中将被省略掉。请思考下面的一个 String 类的部分定义：

```
class String {
    enum { DEFAULT_SIZE = 8 };
    char *d_array_p;
    int d_size;
    int d_length;

  public:
    String();
    // ...
};
```

如果 assert 宏的表达式参数影响了软件的状态（正如以下代码那样），那么产品版本就会展示完全不同的行为。

```
String::String()
: d_size (DEFAULT_SIZE)
, d_length(0)
{
    assert(d_array_p = new char[d_size]);    // error
}
```

通过确保 assert 中的代码完全独立于正常的对象操作，就可以避免出现这个问题：

```
String::String()
: d_size (DEFAULT_SIZE)
, d_length(0)
{
    d_array_p = new char[d_size];
    assert(d_array_p);                       // fine
}
```

这种处理容错技术的通用做法是抛出像 CodingError 这样的异常信息了事[1]。甚至较高层次的软件都使用这种方式来捕捉和处理该问题。在没有错误处理程序时，默认的行为就简化为 assert 行为[2]。

① **murray**，9.2.1 节，208~210 页。

② 要想知道更多关于有意使用异常的信息，见 **ellis**，15.1 节，355 页。

1.4 有关风格的一些问题

当程序员聚集在一起开始一个项目时，常常要讨论采用什么编码标准。这些标准很少能够对产品的质量作贡献。他们经常关心诸如以下这些问题：

我们应该缩进 2、4 或 8 个空格吗？

我们应该在一个 if 语句的右圆括号与左括弧之间像下面这样加一个空格：

```
if (exp) {
```

还是应该像下面这样不加空格：

```
if (exp){
```

在一个大项目的开始阶段，往往要花费数个星期的时间来讨论标准问题。我们得出的结论是：虽然标准化有好处，标准的列表还是越小越好，并且每一条标准都应该由清晰的工程原则来推动。上述的两个例子都不符合这些规则。

我们要学习的另一件事情是什么时候要加强标准，有两个领域：接口和实现。好的接口比好的实现要重要得多。接口会对客户程序产生直接的影响，并且有全局影响力。实现只影响作者和代码维护者。

有明确的理由要求在设计接口时上实行严格的标准，尤其是在大型项目中。修补接口通常要比修补实现更困难和更昂贵。假如有一个封装得很好的接口，那么扔掉一个糟糕的实现，用一个更好的实现来取代它，通常并不会太困难。

1.4.1 标识符名称

下面这些编码惯例已经被无休止地争论过，但经受住了痛苦的考验。本书提出的大多数的建议集中在影响接口方面，在这些方面可以最强烈地感受到它们的好处。此外，这其中大部分只是个人爱好问题。如果说有一个规则的话，那就是保持一致。

1.4.1.1 类型名称

C++的语法是复杂的。有关其结构性质的精细线索总是受欢迎的。一个相当标准并被广泛接受的惯例是以特殊的考虑来处理类型名称。在本书中我们会始终把类型名称的首字母置为大写字母，而其他非类型名称则以小写字母开头。

对于我们的目的来说，类型就是那些既非数据亦非函数的实体：

- 类（Classes）
- 结构（Structures）
- 联合（Unions）
- 类型别名（Typedefs）

- 枚举（Enumerations）
- 模板（Templates）

以下是一些说明这种编程风格的声明：

```
class Point;
struct Date;
union Value;
enum Temperature { COLD, WARM, HOT, VERY_HOT } temp;①
typedef Temp Temperature;
template class Stack<int>;
int Point::getX() const;
void Point::setX(int xCoord);
```

用词典编纂的方式来区分类型名称和其他名称是一种客观的可验证的标准，对于客户和实现者来说同样可以提高可读性。如果一致地使用，这个惯例可以使接口更容易理解，代码也更容易维护。

1.4.1.2 多词标识符名称

给标识符命名的人有两种类型——一种人提倡用下划线字符（"_"）来分隔单词，另一种人则提倡第二个及以后的单词首字母大写：

```
this_is_a_very_long_identifier    与    thisIsAVeryLongIdentifier
```

两种方式都有争论。我刚开始是在下划线阵营，但是为了遵从大多数人的意见不得不作出改变。如今我意识到其实并没有什么区别：这只是一个你习惯于什么的问题。也许使用大写字母的方式更好一点，因为名称可以更短，你习惯了以后它们就会变得更容易阅读。使用大写字母也可以把下划线应用于其他的用途（见6.4.2节和7.2.1节）。重要的是要在整个产品线中始终保持一致。

在同一个产品中一批类使用一种命名习惯而另一批类使用另一种命名习惯，这种做法显得不专业，且有可能会令人困惑，尤其是在考虑到付费顾客会（或有朝一日会）直接访问基本C++类的情况下。但是，有一些程序员可能只是简单地把这些不一致性当作一种风格问题来处理。

在这本书里我采用了"大写字母标准"。然而，无论你采用了何种方式，我都要强烈地忠告：要保持一致，尤其是在接口中。

1.4.1.3 数据成员名称

如果人们记得给他们的类的数据成员始终加上前缀（如，d_），易读性和可维护性将会大

① 本文本中枚举类型和（静态）常量的名称都是大写的，并使用下划线来分隔单词。

大增强。考虑下面的类 Shoe：

```
class Shoe {
    double d_temperature;
    int d_size;
    // ...
  public:
    // ...
    void expand(double calories);
    // ...
    void setSize(int size);
    // ...
};
```

在成员函数内部，局部（自动）变量所拥有的值只是临时的；当成员函数返回之后它们就不存在了。另一方面，类成员数据定义了对象的状态，它们存在于成员函数调用之间：

```
void Shoe::expand(double calories)
{
    const double FACTOR = 42.57;        // Always initialized to same value
    // ...                              // (probably belongs at file scope).

    double factor = calories * FACTOR; // short lived automatic variables
    d_temperature += FACTOR / d_size;  // use of "state" variables
}
```

使用 d_的主要目的是要在一个独立上下文中以一种机械的方法突出类数据成员。因为这两种数据类型有非常不同的用途，使用词典编纂的方式来区分类数据成员名称和局部变量名称，可以使对象实现更易于理解。

经常可以看到这样的成员函数，把一个实例变量（如，d_size）设置成包含单个的赋值表达式：

```
inline
void Shoe::setSize(int size)
{
    d_size = size;
}
```

在数据成员前面放置 d_，也可以避免为操纵函数的参数臆想出来的怪异名称（如，sz）：

```
void Shoe::setSize(int sz)
{
    size = sz;
}
```

前缀 d_的选择是很任意的。我们之所以不单单使用一个下划线字符（_）来作前缀，是

因为以一个下划线字符开始的标识符是留给 C 编译器用的[①]。有些人更喜欢使用后下划线来达到这个目的：

```
void Shoe::setSize(int size)
{
    size_ = size;
}
```

我发现把加后缀留给其他目的是有用的（例如用_p 标识一个指针数据成员）[②]。你也可以使用不同的前缀（例如用 s_标识静态类数据）。无论是在一个类中还是在文件作用域中，非常量静态数据都潜在地包含独立于实例的状态信息。正如将在 6.3.5 节中讨论的，静态类数据成员可以移动到一个.c 文件的文件作用域中，以帮助避免编译时耦合。因为这两种数据类型具有非常相似的特性和可互换性，所以用 s_来标识.c 文件中的状态变量也是有意义的。始终遵循这种命名习惯可以方便查找一个组件中的所有独立于实例的状态变量。

静态类或文件作用域常量数据若没有状态则毫无用处。只需将其名称全部使用大写字母就可以标识这种数据的性质和生存期。对于一个类作用域的常量数据来说，一个诸如 S_DEFAULT_VALUE 或简单如 DEFAULT_VALUE 的名称可以工作得同样好。在这本书里我们选择用 S_DEFAULT_VALUE 来标识类作用域常量静态数据，用以提醒我们需要将它保持私有（见 2.2 节）。

比较而言，一个非静态常量数据成员有更有限的寿命，它的值不必在每一个对象的化身中都相同。因此，它的名称可以以小写字母出现和以前缀 d_开始。

```
class Set { /* ... */ }
class SetIter {
    Set *const d_set_p;        // d_set_p is a const pointer to a Set.
    const double D_PI;         // bad idea (should be static)
    // ...
};
```

我们整个公司都采用加 d_的惯例，没有人抱怨。

1.4.2 类成员布局

在使用一个不熟悉的对象时，要断定在哪儿能找到东西可能会比较困难。尽管在类中成员函数的顺序明显是一个风格的问题，但从客户的观点来看它可以帮助保持前后一致。对成员功能进行分类的一个基本方法就是看它是否潜在地影响了对象的状态。

在图 1-5 中描述了一个对开发者和客户都有用的组织。这个组织具有通过功能（几乎每个

① **ellis**，2.4 节，7 页。

② 标识符后缀的另一个用法见 6.4.2 节。

C++类都会提供某种功能）种类进行分组的优势。这个组织也独立于将被实现的特定的抽象。

```
class Car {
    // ...
  public:
    // CREATORS
    Car(int cost = 0);
    Car(const Car& car);
    ~Car();

    // MANIPULATORS
    Car& operator=(const Car& car);
    void addFuel(double numberOfGallons);
    void drive(double deltaGasPedal);
    void turn(double angleInDegrees);
    // ...

    // ACCESSORS
    double getFuel() const;
    double getRPMs() const;
    double getSpeed() const;
    // ...
};
```

图 1-5　创建函数/操纵函数/访问函数成员组织

创建函数（CREATOR）使对象开始存在和停止存在。注意 operator=不是一个创建函数，但却是（习惯上）第一个操纵函数。**操纵函数**（MANIPULATOR）只是非常量成员函数；**访问函数**（ACCESSOR）是常量成员函数。这种纯粹的对象分组使得其很容易在匆匆一瞥中得到校验：所有的访问函数都声明为类的常量函数，所有的操纵函数都不声明为类的常量函数。但主要的好处还在于为解剖一个不熟悉的类的基本功能提供一个公共的起点。对于更大型的类来说，在每一节中把成员按字母顺序排序可能是有用的。对于像包装器（wrapper，在 5.10 和 6.4.3 节讨论）这样的非常大型的类来说，其他组织方式可能更适用。

有些人会设法把成员函数分组为 get/set 对，如图 1-6 所示。对某些用户来说，这种风格是概念误导的结果，认为一个对象不过就是一个有数据成员的公共数据结构，每一个数据成员都必须既有一个“get”函数（访问函数）又有一个“set”函数（操纵函数）。这种风格本身可能（有时候）会阻碍真正封装接口的产生，在真正封装的接口中，数据成员没有必要透明地反映在对象的行为中。

最后，还有在哪里放置数据成员的问题。完全封装好的类没有公共数据。从逻辑角度来看，数据成员只是类的实现细节。因此，许多人都愿意把类的实现细节（包括数据成员）放在类定义的末尾，如图 1-7 所示。

```
class Car {
    double d_fuel;
    double d_speed;
    double d_rpms;

  public:
    Car(int cost = 0);
    Car(const Car& car);
    Car& operator=(const Car& car);
    ~Car();

    double getFuel() const;
    void setFuel(double numberOfGallons);

    double getRPMs() const;
    void setRPMs(double rpms);

    double getSpeed() const;
    void setSpeed(double speedInMPH);

    // ...
};
```

图 1-6　get / set 成员组织

```
class Car {
  public:
    Car(int cost = 0);
    Car(const Car& car);

    // ...

  private:
    double d_fuel;
    double d_speed;
    double d_rpms;
};
```

图 1-7　尾部数据成员组织

虽然这个组织对于不成熟的客户来说可能更具有可读性，把实现细节藏在类定义末尾的尝试掩饰了它们并未被隐藏的事实。在头文件中实现细节的存在会影响编译时耦合的程度，这种耦合不会因为在类定义中重新放置这些细节就简单地消失。

因为本书要研究物理的和组织的设计问题，我们始终把头文件中的实现细节放在公共接口的前面（部分原因是要强调它们的存在）。在第 6 章中，我们会讨论像这样的实现级的混乱如何才能被整个地从一个头文件中移走，从而确实对客户隐藏。

1.5 迭代器

也许在面向对象设计中最普通的模式就是迭代器（iterator）模式[①]。一个迭代器就是一个与某种原始对象密切耦合的对象，并且这个原始对象被提供给了这个迭代器；它的用途是允许客户程序顺序地访问原始对象的部件、属性或子对象。

对象常常会表现为其他对象的集合，这种对象一般称为**容器**（containers）。集合、列表、栈、堆、队列、哈希表（hash tables）等等就是典型的容器对象。请注意在有关的地方，我们常常会用一段前导注释来标识代码体源文件。例如：

```
// stack.h
#ifndef INCLUDED_STACK
// ...
```

```
// stack.c
#include "stack.h"
// ...
```

例如，我们考虑一下图 1-8 所示的实现一个整数集合的简单类。我们可以从它的头文件中看到，IntSet 使用 IntSetLink 对象来实现，但那是一个封装好的类的实现细节。在这个最小的实现中，我们已经作了这样的选择：通过使这些另外自动产生的函数成为私有函数，来防止用户建立一个 IntSet 拷贝或赋值给一个 IntSet 对象。（注释 NOT IMPLEMENTED 指示此功能不存在，即使是私有的也一样。）只允许 IntSet 的用户建立一个空集合，加进整数，检查成员关系，以及删除它。

```
// intset.h
#ifndef INCLUDED_INTSET
#define INCLUDED_INTSET

class IntSetLink;
class IntSetIter;
class ostream;

class IntSet {
    // DATA
    IntSetLink *d_root_p;  // root of a linked list of integers

    // FRIENDS
    friend IntSetIter;

  private:
    // NOT IMPLEMENTED
    IntSet(const IntSet&);
    IntSet& operator=(const IntSet&);
```

① **gamma**，Iterator，第 5 章，257~271 页。

② **stroustrup**，5.3.2 节，160 页；7.8 节，243 页；8.3.4 节，267 页。

```
public:
  // CREATORS
  IntSet();
      // Create an empty set of integers.
  ~IntSet();
      // Destroy this set.
   // MANIPULATORS
   void add(int i);
       // Add an integer to this set.  If the given integer is
       // already present, this operation has no effect.

   // ACCESSORS
   int isMember(int i) const;
       // returns 1 if integer i is a member of the set,
       // and 0 otherwise.
};

#endif
```

图 1-8　一个简单的整数集合类

图 1-9 中显示了一个用来测试这个有限功能的微小的测试驱动程序。注意，本书中的驱动程序用文件名后缀.t.c 来标识。

```
// intset.t.c
#include "intset.h"
#include <iostream.h>

main()
{
    IntSet a;

    a.add(1); a.add(2); a.add(3); a.add(2); a.add(4); a.add(6);

    for (int i = 0; i < 10; ++i) {
        cout << ' ' << i << '-' << (a.isMember(i) ? "yes" : "no");
    }

    cout << endl;
}

// Output:
//      john@john: a.out
//       0-no 1-yes 2-yes 3-yes 4-yes 5-no 6-yes 7-no 8-no 9-no
//      john@john:
```

图 1-9　测试 IntSet 功能的微小的驱动程序

假设我们要找出这个集合中存在什么成员，然后把它们打印出来。理论上，我们可以写我们自己的输出函数（如图 1-10 所示），但是这种实现的性能可能会稍显不足。

```
#include <limits.h>     // defines INT_MIN and INT_MAX
ostream& operator<<(ostream& o, const IntSet& intSet)
{
    o << "{ ";
    for (int i = INT_MIN; i <= INT_MAX; ++i) {
        if (intSet.isMember(i)) {
            o << i << ' ';
        }
    }
    return o << '}';
}
```

图 1-10　IntSet 输出运算符的行不通的实现

一个显而易见的解决办法就是把 operator<<函数设为 IntSet 类的一个友元，以便利用其内在表示法。我们可以这样做，但是如果客户不喜欢这个运算符的实现所提供的格式怎么办？如果后来我们发现需要访问内部成员又会发生什么？比方说，要比较两个 IntSet 对象。

我们可以持续加入新的成员和友元，但是每次我们这样做，都会把我们的客户和我们自己置于增加类的复杂度的危险之中。不断地修改和扩充对象功能是一种众所周知的把程序错误引进软件的方式。同样，除非计划支持多个版本，否则不关心这些新功能的其他客户将被迫承担它们。

我们可以提供一种普遍而有效的方法来访问集合的单个成员，从而一次性地解决大多数的不足，而不是一次一个地处理这些不足。假设我们决定把这种能力直接加入类 IntSet 本身，就像图 1-11 描述的那样。这时一个客户程序就有可能对 IntSet 类的一个实例进行迭代，并以任何需要的格式打印出那个对象的所有内容。图 1-12 说明了迭代器的一些功能。如果不考虑集合的实现可能如何变化的话，客户的代码将不受影响。

```
class IntSet {
    // ...
  public:
    // ...

void reset();
    // Reset to beginning of sequence of integers.  The Current
    // integer will be invalid only if the set is empty.

void advance();
    // Advance to the next integer in the set.  If the current
    // integer was the last in the set, the current integer
    // will be invalid after advance returns.  Note that the
    // behavior is undefined if the current integer is already
    // not valid.

int current() const;
    // Return the current integer in the sequence.  Note that the
    // behavior is undefined if the current integer is not valid.
```

```
    int isCurrentValid() const;
        // Return 1 if the current integer is valid, and 0 otherwise.
        // Note that the current integer is valid if the set is not
        // empty and we have not advanced beyond the last integer
        // in the set.
    }
```

图 1-11 尝试给容器自身增加迭代能力

```
ostream& operator<<(ostream& o, const IntSet& intSet)
{
    o << "{ ";
    for (intSet.reset(); intSet.isCurrentValid(); intSet.advance()) {
        o << intSet.current() << ' ';
    }
    return o << '}';
}
```

图 1-12 IntSet 输出运算符的另一种实现

但是，图 1-12 中的设计仍然存在问题。对于一个给定的对象，在任何时候都至多只能有一个迭代器在运行。假设我们要为 IntSet 实现一个比较函数，为了调试目的，我们决定打印出比较迭代过程中的集合内容。打印例程可能会有破坏比较函数迭代状态的有害副作用。问题是 IntSet 要分配足够的空间以便为每一次迭代保留状态信息。无论该迭代是否是活动的，被分配的空间都要保留。如果因为某种原因我们需要有一对嵌套的 for 循环，它们在同一个集合的元素上迭代，我们就必须复制整个集合。

这个问题可以通过让客户程序保持内部状态或保留一些其他形式的位置占有来解决。如果客户程序动态地分配状态，客户必须记着删除状态以避免内存泄漏。

如果位置占有是一种整数索引的形式，那么可能会有一些对集合底层实现的附加实际约束。例如，若集合实现为一个链表（而不是数组），那么在每次迭代中都会有潜在的二次指数时间复杂性（即：$O(N^2)$），因为 for 循环的每次迭代都会导致不得不遍历链表。

标准的方法是为每一个容器类都提供一个迭代器类（在同一头文件中）。迭代器声明为容器的友元，因此可以访问它的内部组织。把迭代器和容器类定义在同一个头文件中，是为了避免有关“远距离”友元关系的问题（在 3.6 节中讨论）。具体容器（如 IntSet）的迭代器一般创建在程序堆栈中；这样当迭代器离开作用域时其状态会自动删除。迭代器对象可以更具有空间效率，因为每次迭代的空间只需在迭代过程（本身）中存在。同样，在一个给定的容器中，任意数量的迭代器在任何时间都可以独立地活动，不会互相干扰。

作为一个实践问题，对迭代器来说通常都假设它们所操作的对象在迭代过程中不会被修改或删除。在迭代中对象出现的顺序通常也是依赖于实现的，并且容易遭受擅自的改变。理想的做法是，迭代器开发者应明确声明是否定义迭代的顺序。为安全起见，迭代器的客户程序不应该假定一个顺序，除非指定了一个顺序。

图 1-13 描述了本书中所使用的标准迭代器模式的设计。这个迭代器对象将使用 for 循环。该迭代器的语法相当简洁。运算符的使用很不明显，尤其是对于以前从未见过这种用法的人来说。人们很容易认为这种风格是运算符重载的滥用，因为易读性降低了。然而，这种做法还有其他理由。

```
class IntSetIter {
    // DATA
    IntSetLink *d_link_p;    // root of linked list of integers

  private:
    // NOT IMPLEMENTED
    IntSetIter(const IntSetIter&);
    IntSetIter& operator=(const IntSetIter&);

  public:
    // CREATORS
    IntSetIter(const IntSet& IntSet);
        // Create an iterator for the specified integer set.

    ~IntSetIter();
        // Destroy this iterator (an unnecessary comment).

    // MANIPULATORS
    void operator++();
        // Advance the state of the iteration to next integer in set.

    // ACCESSORS
    int operator()() const;
        // Return the value of the current integer.

    operator const void *() const;
        // Return non-zero value if iteration is valid, otherwise 0.
};
```

图 1-13 IntSet 容器的一个标准迭代器

因为在大型设计中迭代器可能会也确实会频繁出现，开发者需要考虑的最重要的事情一定是一致性问题。如果我们不使用运算符重载而使用函数，那么每一次使用相同的函数名称就很重要；否则我们将发现自己会无意识地给这些函数取错名，并总是要为寻找语法细节而查看头文件。图 1-14 中列出了一些具有代表性的可能是等价的函数名称。

it	it.more()	it.isMore()	it.valid()	it.notDone()
++it	it.next()	it.getNext()	it.advance()	it.getMore()
it()	it.item()	it.getItem()	it.element()	it.value()

图 1-14 我们应该用哪些名称

经验表明，每一个标准迭代方法都采用图 1-14 的左栏中的运算符，形成了超越具体迭代类型的一致的、易用的、很快就会被熟悉的和易于公认的习惯用语。无论你决定用什么，都

要保证在整个产品线中保持一致。IntSet 输出运算符的最后实现如图 1-15 所示：简明扼要的迭代器符号提供了一种简洁的实现。

```
ostream& operator<<(ostream& o, const IntSet& intSet)
{
    o << "{ ";
    for (IntSetIter it(intSet); it; ++it) {
        o << it() << ' ';
    }
    return o << '}';
}
```

图 1-15 使用简洁的迭代器实现的 IntSet 输出运算符

在图 1-15 中，使用前加 1（++it）而不是后加 1（it++）是经过深思熟虑的；后加 1 这种版本需要第二个形式参数并且不是总能得到[1]。此外，当应用到基本类型时，迭代器加 1 的语义更近似于前加 1 的语义（见 9.1.1 节）。

1.6 逻辑设计符号

面向对象设计需要一个丰富的符号集合[2]。这些符号的大多数用来表示设计的逻辑实体之间的关系。

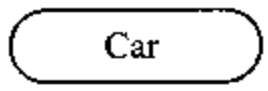
定 义：

符 号	含 义
X	X 是一个逻辑实体（如，类）
x	x 是一个物理实体（如，文件）
B —IsA→ A	B 是 A 的一种
B —Uses-In-The-Interface— A	B 在其接口中使用 A
B —Uses-In-The-Implementation— A	B 在其实现中使用 A

在本书中，我们始终用一个椭圆形的框来标识逻辑实体（例如：类、结构、联合）；

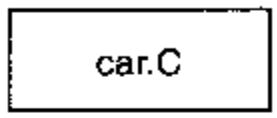
① 见 **ellis**，13.4.7 节，338~339 页。

② **booch**，第 5 章，171~228 页。



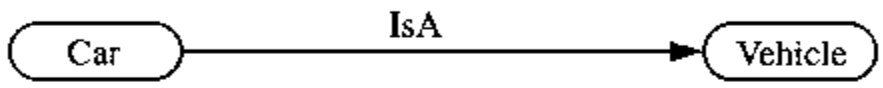

```
class Car {
    // ...
};
```

而用长方形来标识物理实体：

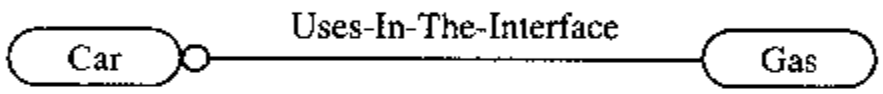



```
// car.c
#include "car.h"
// ...
```

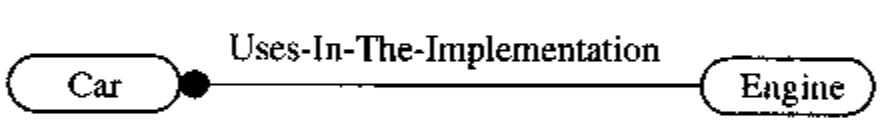
对于我们的目标，三种逻辑符号就足够了：




```
class Car : public Vehicle {
    // ...
};
```




```
class Car {
    // ...
  public:
    void addFuel(Gas *);
    // ...
};
```




```
class Car {
    Engine d_motor;
    // ...
};
```

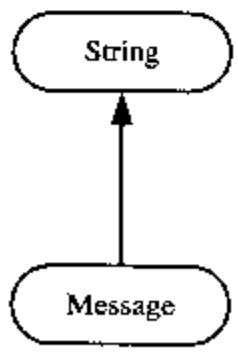
如果还需要另外的逻辑符号，一个清楚地标识关系的加了标签的箭头就足够了。

1.6.1 IsA 关系

假设 Message 是一种 String，也就是说，Message 类型的对象可以用在需要 String 对象的任何地方。

```
class String {
    // ...
  public:
    // ...
};

class Message : public String {
    // ...
  public:
    // ...
};
```

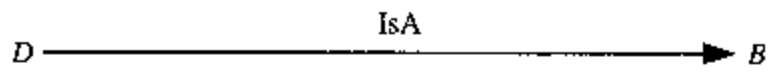
（a）省略的类定义



（b）符号表示法

图 1-16 IsA 关系

正如我们从图 1-16（a）的定义中可以看到的，Message 类继承 String 类，在图 1-16（b）中用一个箭头来表示这种关系：

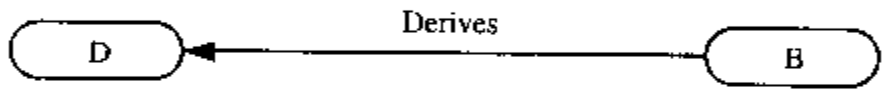


也就是说，*D*——▶*B* 的意思就是“*D* 是一种 *B*”和“*D* 继承 *B*”。

箭头的方向是很重要的，它指向隐含依赖的方向。类 *D* 依赖 *B*，因为 *D* 由 *B* 派生而来，*B* 必须先出现，因为 *D* 要把 *B* 作为一个基类命名：

```
class B { /* ... */ };
class D : public B { /* ... */ };
```

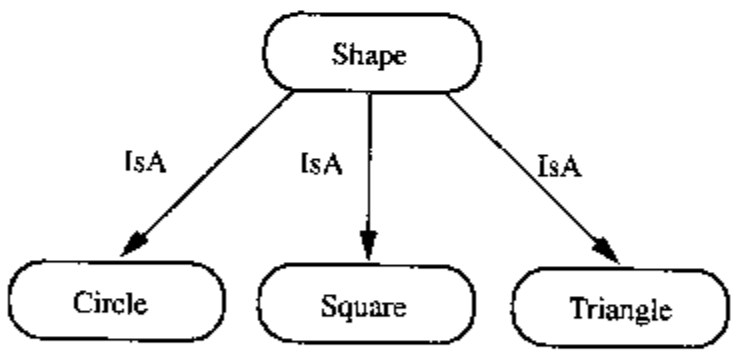
我们常常会看到箭头指向相反的方向，这可能会令人误解。箭头表示两个实体间的一种不对称关系，关系名称由标签指出（在此例中是“IsA”）。如果把箭头画成其他的方式，我们在逻辑上将不得不改变关系名称，例如“派生（Derives）”或“是...的基类（Is-A-Base-Class-Of）”：



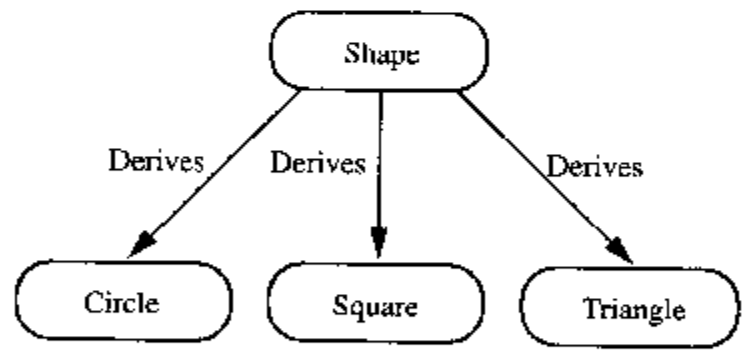
（很少用）

这种可替代的符号很不受欢迎，因为箭头的方向与隐含依赖的方向相反。

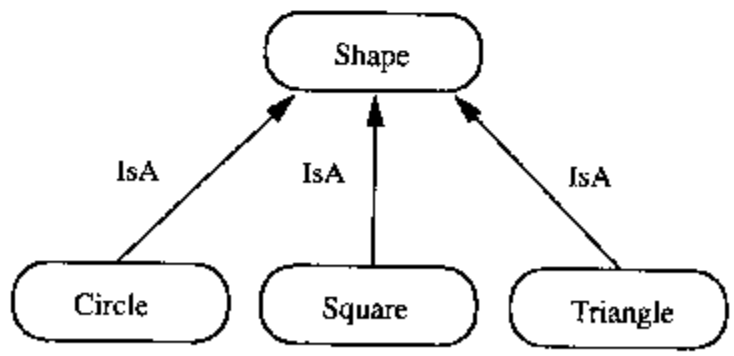
因为分析物理依赖对于好的设计来说是必需的，所以我们采用了使用 IsA 标签和指向隐含依赖方向的箭头的符号。图 1-17 用经典的图形示例展示了继承表示法。



（a）不正确的符号



（b）很少使用的符号



（c）正确的符号

图 1-17　用于标识派生的符号

1.6.2 Uses-In-The-Interface 关系

无论何时一个函数在它的参数表中命名了一个类型或把一个类型命名为一个返回值，就称这个函数在它的接口中使用了那个类型。也就是说，如果一个类型名称是函数返回类型或基调（signature）的一部分，该函数的接口就使用了该类型[①]。

> **定义**：如果在声明一个函数时提到某个类型，那么就是**在该函数的接口中使用了该类型**。

例如，自由函数

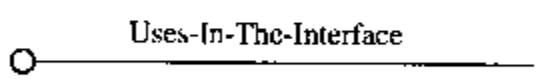
```
int operator==(const IntSet&, const IntSet&);
```

在它的接口中明显地使用了类 IntSet。该函数碰巧返回一个 int，所以 int 也将被认为是这个函数接口的一部分。然而，基本类型是普遍存在的，在实践中可以忽略基本类型。

> **定义**：如果在一个类的（公共）成员函数的接口中使用了某类型，那么就是**在这个类的（公共）接口中使用了该类型**。

在 C++中对类的逻辑访问有三个级别：公共的（public）、保护的（protected）和私有的（private）。一个类的公共接口定义为那个类的公共成员函数的接口的并集。一个类的保护接口的定义与此类似。换句话说，当类 B 的一个（公共）成员函数在它的接口中使用了类 A，我们就说类 B 在 B 的(公共)接口中使用了类 A[②]。例如，类 IntSetIter 的构造函数 IntSetIter(const IntSet&)在它的接口中使用了类 IntSet；因此在 IntSetIter 的接口中使用了 IntSet。

“Uses-In-The-Interface（在接口中使用）”关系是最常见的关系之一，表示为



也就是说，*B* ○——*A* 的意思是“*B* 在 *B* 的接口中使用了 *A*”。我们有时会随便些，说“*B* 在它的接口中使用了 *A*”，但是我们的意思始终是 *B* 在 *B* 的接口中使用了 *A*，而决不是 *B* 在 *A* 的接口中使用了 *A*。

你可以把符号○——想像成一个箭头，它的尾巴在○上，头不见了（或者想像成正指着一个管弦乐队成员的乐队指挥棒）。隐含的箭头方向很重要——它指向隐含依赖的方向。也就是说，如果 *B* 使用了 *A*，那么 *B* 依赖 *A*，但反之则不然。（我们会在 3.4 节中更多地讨论隐含依赖）。

图 1-18 显示了 intset 组件的逻辑视图，包括其中定义的逻辑实体（类和自由运算符函数）间的“Uses-In-The-Interface”关系。图中反映出 IntSetIter 和两个自由运算符都在它们各自的

① 不包括 typedef 的可能使用，它只是同义名。

② 友元关系和 Uses-In-The-Interface 关系的交互在 3.6.1 节讨论。

接口中使用了 IntSet。

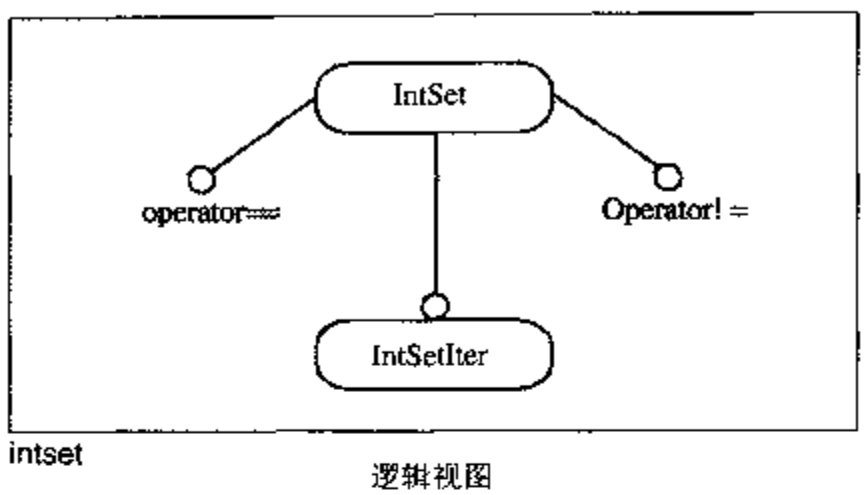


图 1-18 在 intset 组件内部的“Uses-In-The-Interface”关系

“Uses-In-The-Interface”关系是一种对逻辑设计和物理设计都颇有价值的工具。当要将逻辑实体（类和自由运算符）限制在文件作用域内时，这个符号最有用。自由运算符经常会从逻辑图表中省略，以减少符号的混乱。

一个类的实际的逻辑接口可能相当大而且复杂。通常我们最有兴趣展示的特性是一个本质的依赖关系而不是详细的使用方法。在一个类的接口中所使用的类型集合，比被任何特定的成员函数所使用的类型集合都更稳定（即在开发和维护过程中不大可能改变）。所以，与单独的成员函数的使用特性相比，作为整体的类的使用特性越抽象，对在逻辑接口上的小变化就越具有弹性。

1.6.3 Uses-In-The-Implementation 关系

“Use-In-The-Implementation（在实现中使用）”关系增加了设计者抽象表达逻辑依赖的能力。一个逻辑实体将在它的实现中使用另一个逻辑实体（即使没有在它的接口中使用）的表示法，在分析一个设计的基础结构时可能非常有用。和“Uses-In-The-Interface”关系一样，“Uses-In-The-Implementation”关系暗示了两个逻辑实体之间的一种物理依赖。当体系结构设计者们提取高层设计并把它们分布到分散的物理组件中时，可以很好地利用这个信息。

> **定义**：如果在某函数的定义中提到了某类型，那么在这个函数的实现中就使用了该类型。

考虑下面的自由函数 operator==的实现，它**假设**迭代器总是以**相同的顺序**返回等价的 IntSet 对象的成员：

```
int operator==(const IntSet& left, const IntSet& right)
{
    IntSetIter lit(left);
    IntSetIter rit(right);
```

```
    for (; lit && rit; ++lit, ++rit) {
        if (lit() != rit()) {
            return 0;
        }
    }
    // At least one of lit and rit now evaluates to 0.
    return lit == rit;
}
```

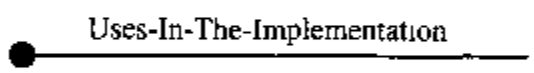
在上面的实现中，创建了两个迭代器：为每个 IntSet 参数都创建一个。只有当两个迭代器都指向有效集合元素时才会进入 for 循环体。随着每个迭代经过循环，集合里的相应位置的整数被比较。若任何一次这样的比较失败，则集合立刻被认为是不相等的。要从 for 循环退出，以下两个条件都必须为真：

（1）至少有一个迭代器已经到达了集合的末端并且当前是无效的。

（2）没有发现集合的相应条目是不相等的。

当且仅当两个迭代器当前都无效时，两个 IntSet 对象才会相等。

注意 operator==(const IntSet&, const IntSet&) 不是类 IntSet 的友元。因此这个运算符的任何有效的实现都必须利用类 IntSetIter。在 operator==与类 IntSetIter 之间的使用关系产生了一种 operator==对类 IntSetIter 的隐含依赖关系。因为在这个运算符的实现中（而不是在它的逻辑接口中）使用了 IntSetIter，所以我们使用一种稍微不同的符号来表示这种关系：

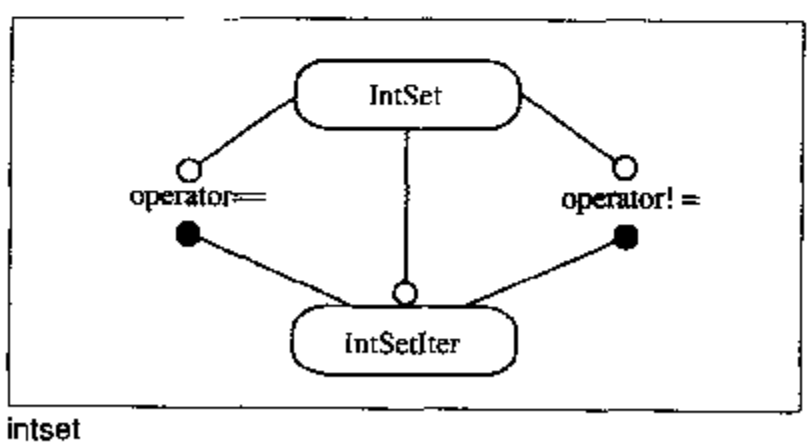


也就是说，*B* ●——*A* 的意思就是在 *B* 的实现中使用了 *A*。

图 1-19 再次为我们展示了带有两种使用关系的 intset 组件的逻辑视图。特别是我们可以看到

```
int operator==(const IntSet&, const IntSet&)
```

在它的接口中使用了类 IntSet，在它的实现中使用了类 IntSetIter。虽然 operator!=显示为与 operator==对称实现，但实际上 operator!=有可能根据 operator==来实现。



逻辑视图

图 1-19　intset 组件中的两种使用关系

如果在一个函数的接口中使用了一个对象，就自动地认为在那个函数的实现中使用了该对象。所以我们看到符号●——就可以推断它所指示的使用不是用在接口中。例如，我们可以从图 1-19 直接推断，operator!=没有在它的接口中使用 IntSetIter。

> **定义**：如果一个类型（1）被用在某个类的一个成员函数中，（2）在某个类的一个数据成员的声明中被提到，或者（3）是某个类的一个私有基类，那么这个类的实现中就使用了这个类型。

一个类可以以多种形式在它的实现中使用另一个类型。正如我们将在 3.4 节提到的那样，类使用一个类型的特定方式，不仅会影响到类依赖那个类型的方式，而且会影响到被迫依赖于那个类型的类的客户范围。我们暂时只简单地展示类在它的实现中可以使用一个类型的方式：

> **定义**：
>
> "Uses-In-The-Implementation"关系的特定种类：
>
> | 名称 | 含义 |
> |---|---|
> | Uses | 该类有一个成员函数命名了该类型。 |
> | HasA | 该类嵌入了该类型的一个实例。 |
> | HoldsA | 该类把一个指针（或引用）嵌入了该类型。 |
> | WasA | 该类私有继承于该类型。 |

1.6.3.1 使用（Uses）

如果一个类的任何成员（包括一个私有成员）函数在它的接口或它的实现中命名了一个类型，就认为在该类的逻辑实现中使用了该类型。

图 1-20 举例说明了因为在类 Crook 的一个成员函数（bribe）体中命名了类型 Judge，所以在 Crook 中的实现中使用了 Judge。换句话说，类 Crook 使用了 Judge。

```
class Crook {
  private:
    void bribe();
    // ...
};

class Judge;

void Crook::bribe() {
    Judge *bad = 0;
    // ...
};
```

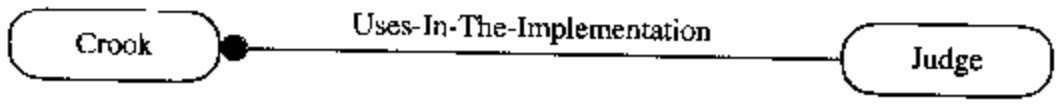


图 1-20 Crook 使用了 Judge

1.6.3.2 HasA 和 HoldsA

当一个类 X 嵌入了类型 T 的一个（私有）数据成员，就出现了另一种使用形式。这种内部使用法通常称为 **HasA**。即使类 X 包含了一个数据成员，其类型只是从 T 派生而来（在 C 语言意义上）（例如：T*或 T&），仍然认为在 X 的逻辑实现中使用了 T。我们偶尔会把这种内部使用法称为 **HoldsA**。

图 1-21 显示了类 Tower 的定义和类 Cannon 的声明。在类 **Battleship** 的实现中使用了这两个类型。具体说来，就是 Battleship **HasA** Tower 和 Battleship **HoldsA** Cannon。我们没有在其所用的符号上进行区别：HasA 和 HoldsA 都用通常的 ●——符号来表示。

```
class Tower { /* ... */ };
class Cannon;                          // declaration only

class BattleShip {
    Tower d_controlTower;
    Cannon *d_replaceableForwardBattery_p;
    Cannon& d_fixedAftBattery;
    // ...
};
```

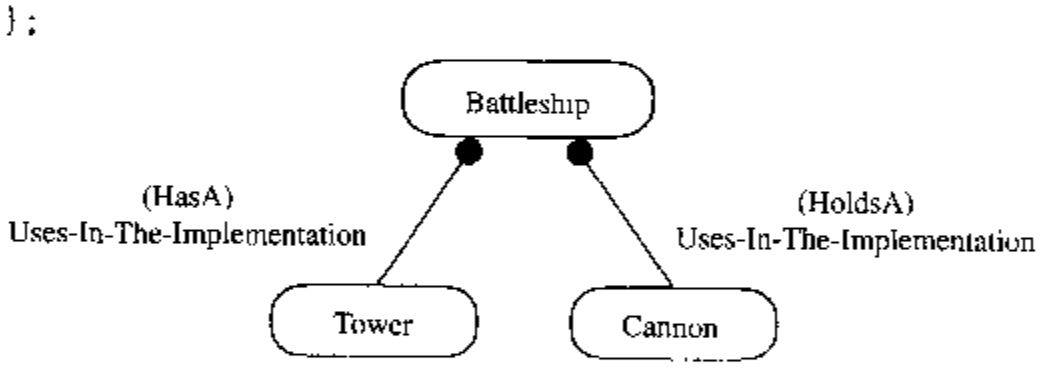


图 1-21 Battleship HasA Tower 和 HoldsA Cannon

1.6.3.3 WasA

私有继承一个类型也是在一个类的逻辑实现中使用该类型的一种方式。私有继承是一个派生类的实现细节。从逻辑角度来看，一个私有的基类（就像一个私有的数据成员）对客户程序是不可见的。私有继承是一种技术，它只可以用来传播其基类属性的一个子集。这种很少使用的关系已经被生动地赋予了一个术语 WasA，如图 1-22 所示。

图 1-22 显示了类 Battleship 的定义，它充当 ArizonaMemorial 的一个私有基类。一旦处在激活服务状态，战舰 Arizona 就是 1941 年珍珠港轰炸中的损失战舰之一。Arizona 目前是一个有礼物商店和展览品的博物馆。

虽然私有继承是一种实现细节，公共继承和保护继承却不是。继承加大了与基类型兼容的类型集合。因此非私有继承会引入信息（客户程序通过编程可以访问这些信息）。公共继承和保护继承的独特性质使它们值得拥有自己的符号，如 1.6.1 节中所示。

```
class Battleship { /* ... */ };
class Shop { /* ... */ };
class Exhibit;                    // declaration only

class ArizonaMemorial : private Battleship {
    Shop d_giftShop;
    Exhibit *d_current_p;
    Exhibit& d_default;
    // ...
};
```

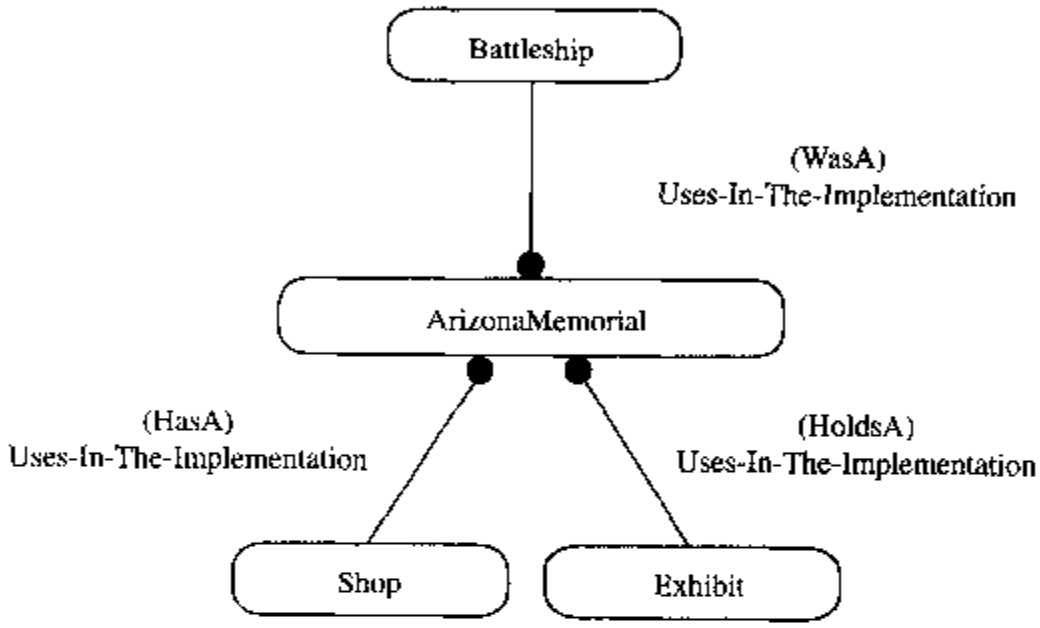


图 1-22 ArizonaMemorial WasA Battleship

现在我们已经回顾了所有的逻辑符号，我们需要开始认真考虑严肃的物理设计问题了。设计的逻辑和物理方面是紧密结合的。每一种逻辑关系——IsA、Uses-In-The-Interface 以及 Uses-In-The-Implementation——都隐含了逻辑实体间的一种物理依赖。正如我们将在第 3 章介绍的，最终正是这些逻辑关系决定了系统内的物理相互依赖关系。

1.7 继承与分层

在面向对象设计的上下文中，当有人提到“层次结构”这个词时，许多人就会想到“继承”。继承是逻辑层次结构的一种形式——分层则是另一种形式。到目前为止，在面向对象设计中更常见的逻辑层次结构形式产生于分层。

> **定义**：如果某个类在它的实现中实质地使用了某个类型，则该类分层于该类型之上。

分层是把更小、更简单或更原始的类型建成更大、更复杂或更精密的类型的过程。分层经常通过组合（例如 HasA 或 HoldsA）来进行，但任何形式的实质使用（即任何导致物理依赖的使用）都具备分层的资格。

通常，客户不能通过更高层对象的接口来编程访问分层类型的实例。隐含的意思是基本类型是在一个较低级的抽象层次上。例如，一个人有一个心脏、一个大脑、一个肝脏等等，但这些分层的器官对象不是大多数健康人的公共接口的一部分。像一个列表这样简单的对象经常作为一个连接集合来实现，但 Link 类本身不会用在写得最好的 List 类的接口中。

继承，连同动态绑定，可以用来区别面向对象语言（如 C++）和基于对象语言（如 Ada），后者支持用户自定义类型和分层，但不支持继承[1]。继承的语义与分层有很大的不同。例如，基类和派生类的公共功能都可以被客户访问[2]。对继承来说，越特殊越具体的类依赖越一般越抽象的类。对分层来说，在较高抽象层次上的类依赖较低抽象层次上的类。

分层是面向对象设计者“兵工厂”中的一种重要而常常配置不足的武器。程序员新手在需要分层的地方试图使用继承的情况并不少见。图 1-23 显示了逻辑层次结构的两个例子。在两个案例中，Person 为了尽其职责都隐含依赖 Heart、Brain 和 Liver。在这里分层显然是正确的方法，因为一个 Person 不是一个 Heart、一个 Brain 或一个 Liver。应改为，一个 Person 有（has）一个 Heart、一个 Brain 和一个 Liver。而且，这些器官不应该暴露在一个 Person 的接口上。有了分层，客户不必屈从于这些内部细节的接口。

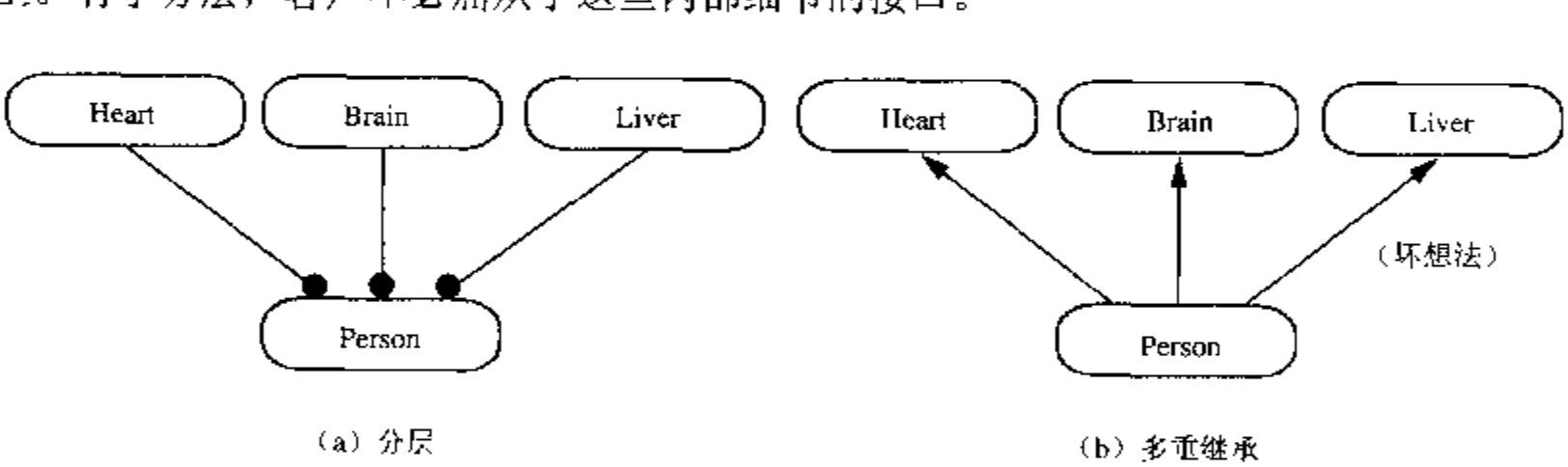


图 1-23　分层与多重继承

1.8　最小化

有些类作者想让他们的类满足所有人的所有需要。这样的类已经被亲切地称为温尼贝戈（Winnebago）类。这种很常见和似乎高贵的愿望令人忧虑。作为开发人员，我们必须记住，只因为一个客户要求增加功能并不意味着对所有类都是适当的。假设你是一个类的作者，10 个客户中的每一个都请求你进行不同的增加。如果你同意，会发生两件事情：

（1）你将不得不实现、测试和存档 10 个新特征，你开始并没有认为这些新特征是你要实现的抽象的一部分（这本身就是问题的一个症状）。

① 见 **booch**，第 2 章，39 页。

② 注意：私有继承是分层的一种形式。

（2）你的 10 个客户的每一个都会得到 9 个他们并没有请求和可能不必要或不想要的新特征。

每次你增加一个特征去取悦一个人，你就扰乱和潜在地骚扰了你的客户库的其他人。曾经发生过这样的事情：原本是轻量级的和很有用的类，经过一段时间后变得过于臃肿，不但不能做好每件事情，而且毫不夸张地说，它们已经变得每件事情都做不好了。

注意，在 1.5 节中，我们选择了通过将其成员函数声明为私有明确地禁止对 IntSet 和 IntSetIter 的实例进行初始化或赋值。为一个集合做一个拷贝可能导致并非微不足道的开发工作量，而这样的迭代器功能在实践中很少需要。我们可以推迟多余功能的实现和测试，除非或直到对那个功能的需求出现。推迟实现也是保持我们的选择权的一种方法。这样做不仅实现、测试、存档和维护软件所需做的工作更少，而且通过特意不过早提供功能，我们既不用为对它的性能负责，也不用对它的实现负责。事实上，不实现功能可以改善可用性。例如，使拷贝构造函数私有可以阻止通过值不经意地传递一个对象——一种用于输入输出流软件包（iostream package）的技术[1]。

这种只要组件足够但不必完备的最低限要求方法适用于正在开发的大型项目，在这种项目中，组件的用户是“内部的”或组件的用户处在一个一旦需要即可快速请求和接收额外功能的位置。最极端的情况是组件高度专业化并且作者是惟一有意向的用户。在这种情况下，实现任何不必要的功能都可能是没有根据的。当然，若一个功能实现对一个抽象来说是本质的，则省略该功能实现将没有意义，比方说，对于一个商业组件库，其用户是付费顾客，他们会期望强壮而且完全的功能对象。这个问题并不是黑白分明的，在这两个极端之间存在一个范围，它对应着一个组件将被广泛使用的程度。在进行这种权衡时，记住要考虑到功能总是更容易增加而不容易删除。

1.9 小结

大型 C++程序分布在不止一个源文件中。把程序分割成单独的编译单元可以使重编译更有效，并且更有可能重用。

虽然大多数 C++声明可以在一个给定的作用域中重复，但每一个用在 C++程序中的对象、函数或类都只能有一个定义。

把有内部连接的定义限制在单个的编译单元中，不能影响其他的编译单元，除非它放在一个头文件中。这样的定义可以存在于 .c 文件的文件作用域内，不会影响全局（符号）名称空间。

有外部连接的定义在连接时可以用来解析其他编译单元中的未定义符号。把这样的定义放在头文件中几乎肯定是一个编程错误。

---

① 通过值传递用户自定义类型是引起不必要的性能降级的一个常见原因（见 9.1.11 节）。

Typedef 声明只是类型的别名，并不提供额外的编译时类型安全。

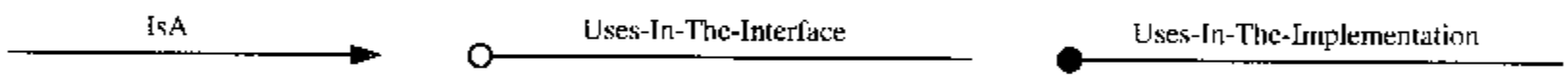
在开发时可以用 assert 语句来有效地发现代码错误，不会影响程序的大小或在一个产品版本中的运行时性能。

在本书中我们采用以下风格惯例：

- 类型标识名以一个大写字母开头。
- 函数和数据以小写字母开头。
- 多词标识名的第二个及以后的词的首字母大写。
- 常量和宏都用大写字母（用一个下划线分隔单词）。
- 类数据成员以 d_为前缀（或者静态成员以 S_为前缀）。
- 类成员函数将按照创建、操纵和访问分类来组织。
- 在类定义中私有细节将放在公共接口之前（主要是强调它们在头文件中的存在）。

迭代器设计模式用于顺序访问某个原始对象的部件、属性或子对象。迭代器被声明为该原始对象的友元，并且它的定义所驻留的头文件应该与该对象的头文件相同。本书中所使用的迭代器表示法是一种简洁的 for 循环模型。

面向对象设计需要一个丰富的逻辑符号集合。但是在本书中，我们只使用三种符号：



每个符号的方向（这里所示的是从左到右）应该和它的标签及所指出的隐含依赖的方向保持一致。一些特殊种类的 Uses-In-the-Implementation 关系有一些特定名称（Uses、HasA、HoldsA 和 WasA），但是，用来表示这些变体的符号都是一样的。

继承和分层是逻辑层次结构的两种形式。到目前为止，分层更常用。它通常包含一个只在实现中的依赖。当问题中的类不能被明显地认为是一种所建议的基类时，分层，尤其是合成比派生好。最后，为了满足若干客户程序的需求而扩展单一类的功能，常常会导致类超重而不受欢迎。对于那些不被广泛使用的类来说，实现过分完全的功能会不必要地增加开发时间、维护开销和代码容量。延缓实现还不需要的功能可减少开发时间，同时预留了选择实现版本的机会。另一方面，用户则希望商用组件库功能完善并且健壮。

2 基本规则

本章将介绍基本设计规则的精简集合。实践证明这些规则是很有用的，并且可以作为框架来讨论本书后面将要介绍的围绕更高级规则的素材。这些基本规则描述基本的规程，例如，限制成员数据访问和减少在全局名称空间中的标识符数量等。特别是，我们将讨论在一个头文件的文件作用域内可以安全放置的结构类型。讨论了为什么需要内部包含卫哨和外部包含冗余卫哨。本章以什么可构成充分文档（例如显式标识未定义的行为）的讨论结束，随后是一个关于标识符命名惯例的简短清单。

2.1 概述

任何精美艺术的美感不仅来源于创造，而且来源于规范。编程也是如此。C++是一种大型语言，有充足的空间进行创造。但是，由于设计空间太大，以致于没有约束——也就是说没有设计结构上的一些适当的约束——大型项目很容易变得难以管理和维护。本书将以设计规则、指导方针和原则的形式介绍这些约束。

设计规则：经验告诉我们，某些编程习惯虽然在 C++中完全合法，但是决不能简单地用于大型项目环境中。直截了当地禁止或毫无例外地要求某种惯例的建议在本书中称为**设计规则**。检验是否遵守了这些规则的过程不能是一种主观过程。设计规则必须足够准确、详尽和良好定义，以便可以客观地检验是否遵守了这些规则。为了效果更好，设计规则必须适合于进行非人为的、借助自动工具的机械验证。

指导方针：经验也告诉我们，有一些习惯应尽可能地避免，这种具有更抽象特性的建议规程称为指导方针，这样的规程有时允许一些合法的例外。指导方针就像必须遵守的经验法则，除非是别的更紧迫的工程原因要求遵守其他方针。

原则：有一些观察和事实在设计过程中经常被证明是有用的，但必须在特定设计的上下文中评估。这些观察和事实称为原则。

确保独立的程序员遵守一致的软件编码标准，是一件很有挑战性的事情。每一个程序员都有自己的习惯扩展集。我给自己强加了许多规则，这些规则比我可能与读者共享的要多得多，但是，大多数只涉及风格，而不是实质。如果我们同意这些规则中的 10%（能给我们带来 90%的实际利益），那么我们就确实做得很好了。

本书包含许多建议。在本章中，我们介绍了一个非常基本的设计规则的集合，称为基本规则，本章解释并证明了（我希望是）每一个规则。读者开始可能不同意所有规则，但是，这些规则经过时间检验，已被证明对非常大型的项目来说是既可行又有效的。

我们把设计规则分成两个不同的类：主要的和次要的。主要的设计规则是指那些必须一直遵守的规程。偏离主要的设计规则很可能不仅影响所涉及的组件的质量，而且会影响系统中其他组件的质量。甚至偶尔违反这些规则就可能动摇一个大型项目成功的基础。贯穿本书，我们假设决不违反主要的设计规则。通常来说，never 决不意味着 NEVER。如果特殊的环境和常识要求违反一个或者多个主要的设计规则，那么设计者有义务完全理解和评估他们的行为所隐含的意义和可能的后果。次要的设计规则是指那些我们强烈推荐但对于一个项目的整体成功不是必不可少的关键因素——例如，只在实现中使用的结构所涉及的问题，不可能影响别的开发者的结构所涉及的问题，以及相对地包含在孤立的实例中并且容易修复的结构所涉及的问题。严格遵守次要规则的要求没有达到至关紧要的地步，因为每个次要规则的违反可能只是增加项目的开销（不像遵守主要规则会影响项目的成功基础）。

因为我们不希望以工程方面的理由来违反设计规则（主要的和次要的），所以对任何禁止一种方法的设计规则，都提供了一个合适的、在所有情况下都能工作的可选方案。

2.2 成员数据访问

封装是一个术语，用于描述在过程接口后面隐藏实现细节的概念。类似的术语有**信息隐藏**和**数据隐藏**。直接访问一个类的数据成员违反封装原则。

主要设计规则

保持类数据成员的私有性。

考虑图 2-1 中类 Rectangle（长方形）的定义。Rectangle 是由标识其左上角和右下角的两个点对象来定义的。因为在 Rectangle 的这个特定实现中，是在内部存储这些 Point 的值，所以我们可以尝试将数据成员变成公共的，以避免为每个类提供操纵函数（即 set）和访问函数（即 get）。

```
// rectangle.h
#ifndef INCLUDED_RECTANGLE
#define INCLUDED_RECTANGLE

class Rectangle {
  public:
    Point d_lowerLeft;     // bad idea (public data)
    Point d_upperRight;    // bad idea (public data)

  public:
    // CREATORS
    Rectangle(const Point& lowerLeft, const Point& upperRight);
    Rectangle(const Rectangle& rect);
    ~Rectangle();
    // MANIPULATORS
    Rectangle& operator=(const Rectangle& rect);
    void moveBy(const Point& delta);
    // ...
    // ACCESSORS
    int area() const;
    // ...
};

// ...

inline
void Rectangle::moveBy(const Point& delta)
{
    d_lowerLeft += delta;
    d_upperRight += delta;
}

// ...

#endif
```

图 2-1 糟糕的（未封装的）Rectangle 类接口

现在考虑，当我们发现 Rectangle 是经常移动的对象时对客户程序的影响。为了改进性能，我们可以试着改变 Rectangle 对象的表示方式。例如，不再存储右上角的绝对位置，而是通过存储其相对于左下角的位置来隐含地表示右上角的位置：

```
class Rectangle {
  public:
    Point d_lowerLeft;                  // same purpose as in Figure 2-1
    Point d upperRightOffset            // new "relative" representation
```

用这种新的表示法，moveBy 成员函数可以在一行中实现（而不是两行），因为右上角与

左下角的相对位置不受移动的影响。

```
inline
Rectangle::moveBy(const Point& delta)
{
    d_lowerLeft += delta;
}
```

右上角的位置不再存储在 Rectangle 对象中，因此当需要它时必须通过计算来得到：

```
void client(const Rectangle& rect)
{
    Point upperRight = rect.d_lowerLeft + rect.d_upperRightOffset;
    // ...
}
```

任何以前直接访问过 d_upperRight 数据成员的客户程序现在都被迫要重新编码。组件重用使这个问题更严重。如果一个定义公共数据的类在可执行程序中是共享的，那么，对单个类的数据表示方式的改变，可能会使得有必要修改所有独立程序的源代码。

> **定义**：若不能通过某个类的逻辑接口编程访问或检测到其包含的实现细节（类型、数据或函数），则称这些实现细节被该类封装了。

封装是面向对象设计的一个重要工具[①]。封装意味着我们将低层次的信息集合在一起，使它们以一种紧密耦合的密切方式潜在地交互。而信息隐藏则用于限制外部世界与某些细节交互，这些细节与类要帮助实现的抽象无密切关系。

使所有的数据成员保持私有，并提供适当的访问函数和操纵函数，如图 2-2 所示，这样我们就可以自由地改变内部表示而不会迫使客户程序重新编码。getUpperRight()的实现可能已经被修改来计算右上角值了，但没有改变其逻辑接口。

除了可维护性外，不要含有公共数据成员还有一些其他原因。例如，在一个类中的数据成员的值很少是独立的。直接（可写的）访问数据（如图 2-2 中的 d_area）很容易使对象处于不一致的状态。只提供一个函数接口就可以授予类作者必要的控制级别，以确保其对象的完整性。提供操纵函数和访问函数也赋予了开发者插入临时代码的机会（例如，为了调试的打印语句，为了性能调整的参考计数，以及为了可靠性的 assert 语句）[②]。

注意，对自身整个被隐藏（不是私有地隐藏在另一个类中就是局部地隐藏在一个.c 文件中）的结构（或类）的数据成员的公共访问是一件不同的事情，不适用上述规则（见 6.4.2 节和 8.4 节）。当数据成员并非私有时，通过使用关键字 struct 而不是 class 来表示认为不需要封装的结构更合适。

---

① **booch**，第 2 章，49~54 页。

② 对于为什么要避免公共数据的进一步的讨论，见 **meyers**，Item 20，71~72 页。

```
// rectangle.h
#ifndef INCLUDED_RECTANGLE
#define INCLUDED_RECTANGLE

class Rectangle {
    Point d_lowerLeft;     // Yet another representation!
    int d_width;           // Fortunately, these data members are private.
    int d_height;
    int d_area;            // Store this redundantly to improve performance.

  public:
    // CREATORS
    Rectangle(const Point& lowerLeft, const Point& upperRight);
    Rectangle(const Rectangle& rect);
    ~Rectangle();

    // MANIPULATORS
    Rectangle& operator=(const Rectangle& rect);
    void moveBy(const Point& delta);
    // ...

    // ACCESSORS
    int area() const;
    Point getLowerLeft() const;
    Point getUpperRight() const;
};

// ...

inline
void Rectangle::moveBy(const Point& delta)
{
    d_lowerLeft += delta;
}

// ...

inline
Point Rectangle::getUpperRight(const Point& delta) const
{
    return d_lowerLeft + Point(d_width, d_height);
}

// ...
```

图 2-2 更好的（封装的）Rectangle 类接口

有人主张使用保护的数据以方便来自派生类的任意访问。但是从维护的角度看，保护访问与公共访问是一样的，因为任何想要得到保护数据的人，只要稍微增加一点派生一个类的

工作即可。和友元关系不同（友元关系明确地表示谁可以访问私有细节），使类数据成为保护的数据会导致封装出现一个大裂口。

适用于公共接口的观点同样也可应用于保护接口。通过将保护接口和公共接口看成是独立但同等重要的，基类作者可以保持可维护性。保持所有数据成员的私有性并提供合适的保护函数，将使得对基类实现的改变独立于任何派生类。

2.3 全局名称空间

对于中等规模的项目来说，如果参加的开发者不止一人，当将开发者各自独立开发的部件集成到一个程序时，就会有命名冲突的危险。问题的严重性会随着系统规模的扩大而呈指数级增长。若冲突是由于第三方提供的集成软件引起的，则情况会更加恶化。

有许多污染全局名称空间的方式，其中一些方式比其他方式麻烦。在大型系统环境中，所有这些方式都是起反作用的。我们现在分别研究几个这样的问题，并以一个设计规则作为本节的结论，该规则描述什么类型的声明和定义可以安全地存在于C++头文件的文件作用域中。

2.3.1 全局数据

有人将全局变量比喻为癌症：不能与之一起生存，但是一旦建立了，则很难将它消除。在一个新的C++项目中，我们总是可以避免使用外部全局变量而获得成功。这条规则的例外可能涉及两种访问：一种是通过全局变量进行通讯的结构复杂的程序访问（如Lex或YACC）；另一种可能是在嵌入式系统中的访问。

主要设计规则

避免在文件作用域内包含带外部连接的数据。

文件作用域中的带有外部连接的数据，存在与其他编译单元中的全局变量冲突的危险（这些编译单元的作者是以自我为中心的，他们认为自己拥有整个全局域）。但是“名称污染”只是全局变量破坏程序的许多方式之一。全局变量将对象和代码绑在一起，这种方式使得在别的程序中实际上不可能有选择地重用编译单元。在大型项目中，调试、测试，甚至理解大量使用全局变量的系统的开销也可能变得非常惊人。

假如我们不是被迫使用一个已要求在其接口上使用全局变量的系统，则有两种简单的变换方式能将这些变量非全局化：

（1）将所有全局变量放入一个结构中。

（2）然后将它们私有化并添加静态访问函数。

假如我们有下列全局变量：

```
int size;
double scale;
const char *system;
```

通过将这些变量放入一个结构中并使它们成为该结构的静态成员，可以把它们从全局名称空间中删除[①]：

```
struct Global {
    static int s_size;                  // bad idea (public data)
    static double s_scale;              // bad idea (public data)
    static const char *s_system;        // bad idea (public data)
};
```

当然，要记住在相应的.c 文件中定义这些静态数据成员。现在不要用

```
size、scale 或 system
```

来访问全局变量，而是改为分别使用

```
Global::s_size、Global::s_scale 或 Global::s_system
```

来访问。命名冲突的概率现在减少到只可能会与一个单一类名相冲突（即使是这种概率的冲突，应用 7.2 节讨论的技术也容易解决）。

尽管我们已经解决了全局名称空间问题，我们仍未做完我们应做的事。经验表明，就像直接访问非静态成员数据（即特定实例）一样，直接访问静态成员数据（即特定类）会使得大型系统非常难以维护。如果我们要把一个成员（如 s_size）的输出数据类型从 int 变为 double，那么这种改变将是接口的改变；所有的客户程序都会受到影响（不管我们采取了什么措施）。但是我们可以决定把 s_size 的实现改变为一个基于其他更原始的值（如 s_width 和 s_height）而计算得到的值。倘若利用静态函数成员访问（和操纵）静态数据成员，我们就可以进行上述局部修改而不会扰乱整个作用域的客户程序。

消除公共数据的下一步措施是使 Global 成为一个类，并且提供静态的操纵和访问方法，如图 2-3 所示。类 Global 现在所起的作用是作为在程序的任何地方都可以访问的逻辑模块。由于所有的接口函数都是静态的，所以没有必要实例化一个对象来使用该类。将默认构造函数声明为私有并且不具体实现它，可以加强这种使用模型。

为了获得灵活的设计，我们应该谨慎小心，以免滥用全局状态信息。我们希望只有一个对象的单个实例这一事实，并非是使它成为模块（而不成为一个可实例化的类）的充分理由。当全局可访问模块相当于固有的惟一实体（例如系统控制台）时，或者是对于一个没有被特定应用程序所控制的系统范围的常量（如在 limits.h 中可找到的）（见 6.2.9 节）来说，全局可访问模块是有意义的。如果有别的更局部化的（如基于对象的）实现足以胜任，则最好避免

① meyers，Item 28，93~95 页。

使用全局模块[1]。

```
class Global {
    static int s_size;
    static double s_scale;
    static const char *s_system;

  private:
    // NOT IMPLEMENTED
    Global(); // prevent inadvertent instantiation

  public:
    // MANIPULATORS
    static void setSize(int size) { s_size = size; }
    static void setScale(double scale) { s_scale = scale; }
    static void setSystem(const char *system) { s_system = system; }

    // ACCESSORS
    static int getSize() { return s_size; }
    static double getScale() { return s_scale; }
    static const char *getSystem() { return s_system; }
};
```

图 2-3 包含全局状态信息的逻辑模块

2.3.2 自由函数

自由函数也会对全局名称空间形成威胁，尤其是参数基调中不包含任何用户定义类型时。如果一个自由函数在一个.h文件中定义为有内部连接或者在一个.c文件中定义为有外部连接，那么在程序集成过程中它可能会与有相同名称和（基调）的另一个函数定义相冲突。运算符函数例外。

主要设计规则

在.h 文件的文件作用域内避免使用自由函数（运算符函数除外）；在.c 文件中避免使用带有外部连接的自由函数（包括运算符函数）。

幸好，自由函数总能分组到一个只包含静态函数的工具类（结构）中。这样产生的内聚性不一定是最佳的，但它减少了全局名冲突的可能性。下面是一个例子：

```
int getMonitorResolution();                                  // bad idea
void setSystemScale(double scaleFactor);                     // bad idea
int isPasswordCorrect(const char *usr, const char *psw);     // bad idea
```

① 见 singleton 设计模式，**gamma**，第 3 章，127~134 页。

该自由函数总是可由下面的静态方法代替：

```
struct SysUtil {
    static int getMonitorResolution();
    static void setSystemScale(double scaleFactor);
    static int isPasswordCorrect(const char *usr, const char *psw);
};
```

惟一有冲突危险的符号是类名 SysUtil。

但是，自由运算符函数不能嵌入类中。这不是一个严重的问题，因为自由运算符要求至少有一个参数是用户自定义类型，因此自由运算符冲突的可能性很小，而且这种冲突在实践中一般不成问题。

2.3.3 枚举类型、typedef 和常量数据

枚举类型、typedef 和（默认的）文件作用域常量数据都有内部连接。人们经常在头文件的文件作用域内声明常量、枚举类型或用户自定义类型，这是一个错误。

主要设计规则

在.h 文件的文件作用域内避免使用枚举类型、typedef 和常量数据。

因为 C++完全支持嵌套类型，因此可以在一个类的作用域中定义枚举类型（或者声明 typedef）而不会在全局名称空间中与别的名称冲突。选择一个更受限的作用域，在该作用域中定义一个枚举类型，我们可以确保所有的该枚举类型中的枚举元素的作用域相同，并且不会与在该作用域之外定义的别的名称冲突。

考虑下列枚举类型：

```
// paint.h
enum Color { RED, GREEN, BLUE, ORANGE, YELLOW };      // bad idea

// juice.h
enum Fruit { APPLE, ORANGE, GRAPE, CRANBERRY };       // bad idea
```

这两个枚举类型可能不是同一个开发者开发的，但是却很可能在某一天包含在同一个文件中，导致 ORANGE 有二义性，无法解析！

```
// picture.c
#include "picture.h"
#include "paint.h"
#include "juice.h"
```

如果将这两个枚举类型改为在不同的类中定义，则我们可以很容易地使用作用域标识符

来解决二义性问题：Paint::Orange 或 Juice::Orange。

因为同样的原因，typedef 和常量数据也应该放在头文件的类作用域内。大多数常量数据是整型的，并且嵌套枚举类型能很好地在类的作用域中提供整型常量。其他的常量类型（如 double、String）必须是类的静态成员，并且在.c 文件中初始化。

```
// array.h
#ifndef INCLUDED_ARRAY
#define INCLUDED_ARRAY

class String;

class Array {
    enum { DEFAULT_SIZE = 100 };
    static const double DEFAULT_VALUE;
    static const String DEFAULT_NAME;

    // ...
};

#endif
```

```
// array.c
#include "array.h"


#include "str.h" // class String



double Array::DEFAULT_VALUE = 0.0;
String Array::DEFAULT_NAME = "";

// ...
```

在大型项目中，除了全局名称冲突外，还有在文件作用域中寻找枚举类型、typedef 和常量等非常现实的问题。将一个 typedef 嵌套在一个类中将迫使名称被完全界定（或声明被继承），这样的名称相对容易发现。同样的推理也适用于枚举类型，但是对于在类中的枚举类型嵌套，我们已经给出了更强有力的论据。

2.3.4 预处理宏

在 C++中几乎不需要宏。它们对包含卫哨（guard）是有用的（见 2.4 节），并且只有在少数情况下，在一个.c 文件中它们的好处才超过它们的问题（最特别的是，当用于为移植或调试获得条件编译时）。但是，一般来说，预处理宏对软件产品是不合适的。

主要设计规则

除非是作为包含卫哨，否则在头文件中应避免使用预处理宏。

预处理器不是 C++语言的一部分：它的基本原则是完全不改变原文（即不对原文进行编译），这使得宏非常难以调试。尽管宏可以使代码易于编写，但是它们的自由形式经常使得代码更难阅读和理解。考虑下列代码片段：

```
#define glue(X,Y) X/**/Y
glue(pri,ntf) ("Hello World");
```

在源代码层次上，我们如何告诉调试者、浏览器或其他的自动工具去处理上述代码呢？

和在.c 文件中包含宏一样糟糕，甚至有更充足的软件工程理由要求头文件不能包含宏。这样的情况是允许的：在一个头文件中使用#define 来定义预处理器常量。因为宏不是 C++语言的一部分，它们不能置于一个类的作用域中。任何包含一个带有#define 的头文件的文件将具有该预处理器常量的定义。

假设 theircode.h 定义了一个常量值 GOOD 作为一个预处理器常量：

```
// theircode.h
#ifndef INCLUDED_THEIRCODE
#define INCLUDED_THEIRCODE

// ...

#define GOOD 0  // bad idea

// ...

#endif
```

```
// ourcode.c
#include "ourcode.h"
#include "theircode.h"

// ...

int OurClass::aFunction()
{
    enum { BAD = -1, GOOD = 0 } status = GOOD;

    // ...

    return status;
};

// ...
```

当编译 ourcode.c 文件时，编译器首先调用预处理器。即使 GOOD 在函数的保护作用域内定义，它对于预处理器也是不安全的，预处理器将毫不留情地将枚举值 GOOD 置为精确的整数 0：

```
// ...

int OurClass::aFunction
{
    enum { BAD = -1, 0 = 0 } status = 0;

    // ...

    return status;
};
```

当编译器遇到枚举类型时，会弹出 Syntax Error（语法错误），但是直到我们经历了一个持续的“grepping”，在所有的.h 文件中寻找谁定义（#define）了我们的一个枚举值后，才能知道出现错误消息的原因。注意，如果在文件作用域中预处理器符号由 const 或 enum 代替，这个问题将不会发生（根据 2.3.3 节的讨论，这也违反了设计规则）。

```
// theircode.h
#ifndef INCLUDED_THEIRCODE
#define INCLUDED_THEIRCODE

// ...

const int GOOD = 100;  // bad idea
    // file-scope constant data

// ...

#endif
```

```
// theircode.h
#ifndef INCLUDED_THEIRCODE
#define INCLUDED_THEIRCODE

// ...

enum { GOOD = 100 };  // bad idea
    // file-scope enumerated value

// ...

#endif
```

在丢失或没有充分实现 C++语言特性的情况下，预处理器宏也可以用于实现模板。如果宏用于此目的，那么宏函数将出现在头文件中。有一些解决这个问题的方法（不借助于宏），可能更适合大型项目。无论如何，与模板相关的问题应在开发过程的早期解决。

2.3.5 头文件中的名称

在头文件的文件作用域中声明的名称，可能潜在地与整个系统中任何一个文件的文件作用域名称冲突。即使在一个.c 文件的文件作用域中声明的带有内部连接的名称也不能保证一定不与.h 文件的文件作用域名称冲突。

主要设计规则

只有类、结构、联合和自由运算符函数应该在.h 文件的文件作用域内声明；只有类、结构、联合和内联函数（成员或自由运算符）应该在.h 文件的文件作用域内定义。

我们希望只能在一个头文件的文件作用域中找到类声明、类定义、自由运算符声明和内联函数定义。在类作用域中嵌入所有其他的结构，可以消除大多数与名称冲突相关的麻烦。

为了辅助说明这个规则，图 2-4 中提供了一个在其他方面没有意义的带有注释的头文件。这个头文件包含了几个结构。注意，一个用户自定义类型的静态实例是一个特殊情况，将在 7.8.1.3 中讨论。现在，把避免在.h 文件中使用这些静态用户自定义对象看作是指导方针而不是规则。

```
// driver.h                              // fine: comment
#ifndef INCLUDED_DRIVER                  // fine: internal include guard
#define INCLUDED_DRIVER                  // fine: (see Section 2.4)

#ifndef INCLUDED_NIFTY                   // fine: redundant include guard
#include "nifty.h"                       // fine: CPP include directive
#endif                                   // fine: (see Section 2.5)
```

```
#define PI 3.14159265358                    // AVOID: macro constant
#define MIN(X) ((X)<(Y)?(X):(Y))            // AVOID: macro function

class ostream;                                // fine: class declaration
struct DriverInit;                            // fine: class declaration
union Uaw;                                    // fine: class declaration


extern int globalVariable;                  // AVOID: external data declaration
static int fileScopeVariable;               // AVOID: internal data definition
const int BUFFER_SIZE = 256;                // AVOID: const data definition
enum Boolean { ZERO, ONE };                 // AVOID: enumeration at file scope
typedef long BigInt;                        // AVOID: typedef at file scope


class Driver {
    enum Color { RED, GREEN };                // fine: enumeration in class scope
    typedef int (Driver::*PMF)();             // fine: typedef in class scope
    static int s_count;                       // fine: static member declaration
    int d_size;                               // fine: member data definition
private:
  struct Pnt {
      short int d_x, d_y;
      Pnt(int x, int y)
      : d_x(x), d_y(y) {}
    };                                          // fine: private struct definition
    friend DriverInit;                          // fine: friend declaration
  public:
    int static round(double d);               // fine: static member
                                              //       function declaration
    void setSize(int size);                   // fine: member function declaration
    int cmp(const Driver&) const;             // fine: const member
                                              //       function declaration
};                                            // fine: class definition

static class DriverInit {
    // ...
} driverInit;                               // special case (see Section 7.8.1.3)

int min(int x, int y);                      // AVOID: free function declaration

inline
int max(int x, int y)
{
    return x > y ? x : y;
}                                           // AVOID: free inline
                                            //       function definition


inline
void Driver::setSize(int size)
{
    d_size = size;
```

```
}                                       // fine: inline member
                                        //       function definition

ostream& operator<<(ostream& o,
                const Driver& d);       // fine: free operator
                                        //       function declaration

inline
int operator==(const Driver& lhs,
               const Driver& rhs)
{
    return compare(lhs, rhs) == 0;
}                                       // fine: free inline operator
                                        //       function definition
inline
int Driver::round(double d)
{
    return d < 0 ? -int(0.5 - d)
                 : int(0.5 + d);
}                                       // fine: inline static member
                                        //       function definition

#endif                                  // fine: end of internal include guard
```

图 2-4　头文件作用域中各种各样的结构

2.4　包含卫哨

即使我们遵守了上面的建议：只在头文件的文件作用域中定义类、结构、联合和内联函数，如果相同的头文件在一个编译单元中被包含两次，仍然会有问题。如图 2-5 所示，这个问题可能出现在一个简单的包含图中。

当编译组件 c 的.c 文件时，预处理器首先包含相应的头文件 c.h，依次地，c.h 文件包含 a.h，c.h 包含 b.h，触发 b.h 又第二次包含 a.h。如果 a.h 有任何定义（在 C++中它几乎肯定有），则编译器会抱怨有多种定义。

主要设计规则

在每个头文件的内容周围放置一个惟一的和可预知的（内部）包含卫哨。

解决这个问题的传统方法是，在每个头文件的内容周围加上一个内部的保护包装器。不管包含图是什么样的，包装器都能确保类和内联函数在一个给定的编译单元中只出现一次。注意，我们不是在企图阻止循环包含（这可能是一个设计错误）；我们要阻止的是重复包含，这种重复包含源自一个非循环包含图中的再收敛。对前面例子中存在的编译问题的一个解决方案如图 2-6 所示。注意，我们仍然没有加入冗余（外部）包含卫哨（在 2.5 节中讨论）。

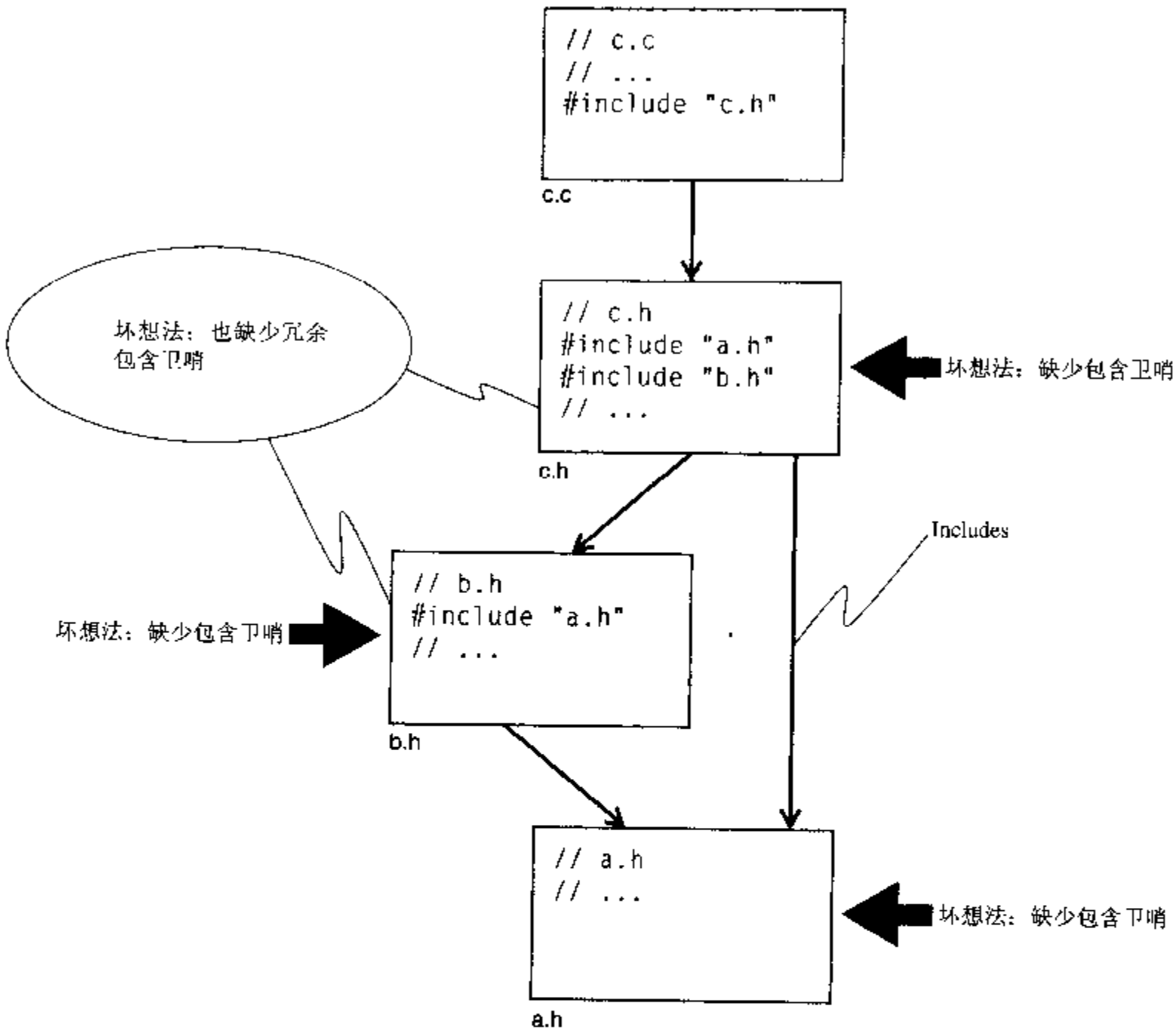


图 2-5 再收敛（reconvergent）包含图引起编译时错误

例如，当 a.h 被包含在一个编译单元中时，预处理器首先检查预处理器符号 INCLUDED_A 是否已定义。如果没有定义，将定义一次卫哨符号 INCLUDED_A，并且是为整个编译单元定义的，然后通过读包含在头文件其余部分中的定义，继续进行预处理。第二次（或更多次地）包含头文件时，预处理器将忽略#ifndef 条件从句中的内容（即文件的其余部分）。

用于包含卫哨的实际符号并不重要，只要它不与整个系统中的其他符号冲突就行。因为包含卫哨是与特定的头文件绑定在一起的，而该头文件名必须在整个系统中是惟一的，所以将头文件名称并入卫哨符号可以保证没有哪两个卫哨符号会是一样的。

预处理器不知道 C++的作用域规则，因此我们必须确保包含卫哨符号不与其他系统中的名称冲突——甚至不能与在一个.c 文件中定义的函数名称冲突。

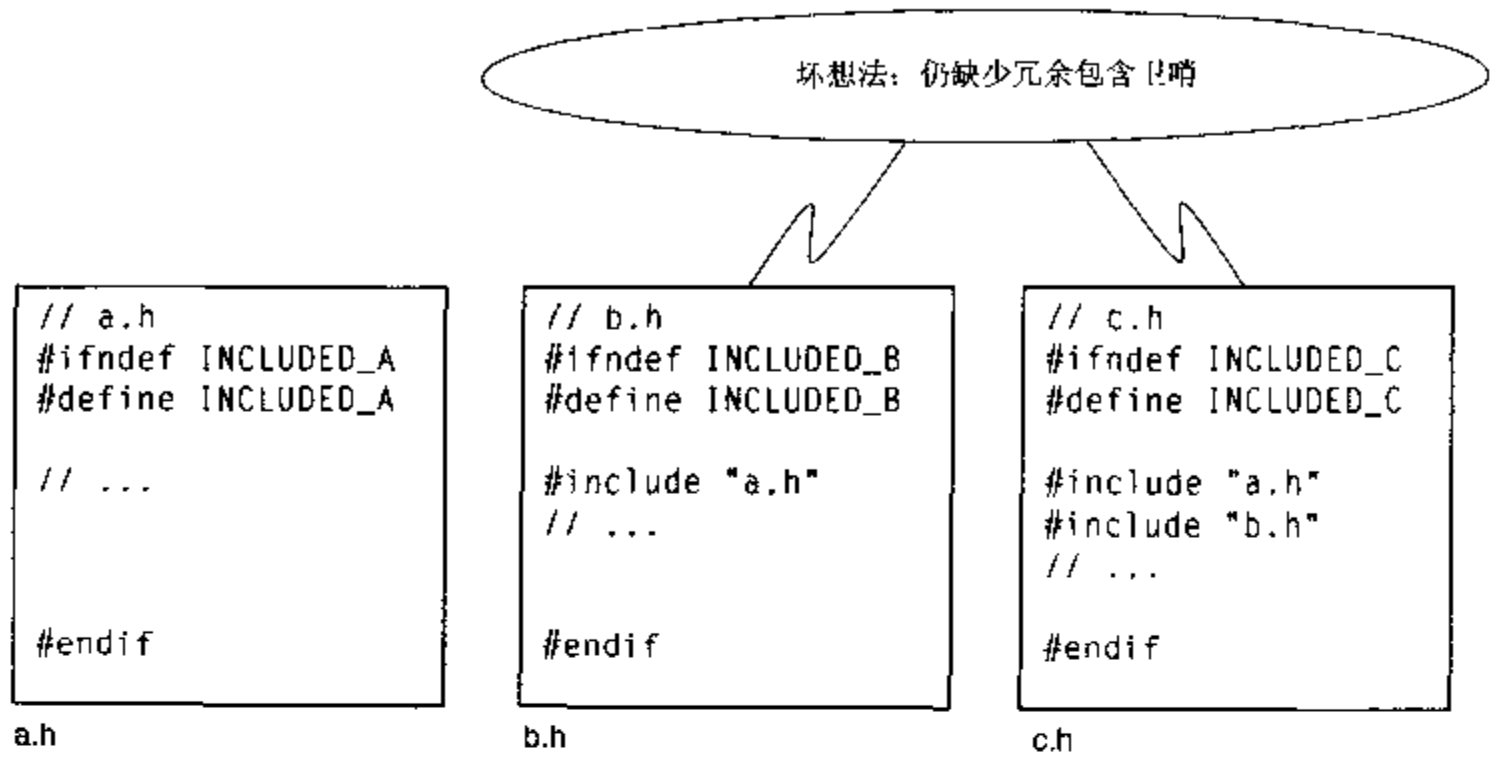


图 2-6　使用包含卫哨提供重复的包含

采用标准的命名习惯，即在头文件的大写根名称（如，STACK）前加上全局保留前缀（如，INCLUDED_），可以确保卫哨名称是惟一的并且可预知的：

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK
// ...
#endif
```

```
// iccad_transistor.h
#ifndef INCLUDED_ICCAD_TRANSISTOR
#define INCLUDED_ICCAD_TRANSISTOR
// ...
#endif
```

对可预知性的需求将在 2.5 节中讨论。

2.5　冗余包含卫哨

可行的并不总是恰当的，下面要介绍的情况就是一个例子。理论上讲，惟一的内部包含卫哨就足够了。但是，对于大型项目而言，若不做进一步的考虑，则开销可能会很大。

一个设计良好的系统由多层抽象构成。只要可能，我们总是希望先创建少量的基本对象类型，然后在更高的抽象层次上将这些对象组织成新的对象。一个科学的应用可以将各种不同种类的原子建模为对象。在元素周期表中有 100 多种原子。这些相对较少的基本类型的实例以不同的方式和比例通过分层构成宇宙中各种不同类型的分子。

分层设计的另一个例子可能是一个面向对象的窗口系统。假设我们有一个 N 个基本窗口部件的集合（如按钮、刻度盘、滑动开关、显示控件等）。为简洁起见，我们将这些基本窗口部件类命名为 W1、W2、…、W*n*。每个窗口部件 Wi 存在于自己的独立编译单元 wi.c 中，带

有相应的头文件 wi.h。可以通过组织各种类型的部件对象来创建新的屏幕类型。我们将这 M 个屏幕类称为 S1、S2、…、S*m*，每一个类 Si 都存在于自己的独立编译单元中，带有头文件 si.h。

一般来说，每个屏幕会使用相当数量的可用窗口部件。为讨论方便，假设每个屏幕类型都充分使用了所有的（或者绝大部分的）基本类型，这样就促使实现者在每个 s1.h 文件中包含所有的 w1.h、w2.h、…、w*n*.h 文件。一个典型的屏幕头文件 s13 如图 2-7 所示。

```
// s13.h
#ifndef INCLUDED_S13
#define INCLUDED_S13

#include "w1.h"
#include "w2.h"
#include "w3.h"
// ...
#include "wn.h"
#include <math.h>

class S13 {
    W1 d_w1a;
    W1 d_w1b;
    W2 d_w2;
    W3 d_w3;
    // ...
    Wn d_wn;
};

#endif
```

图 2-7　由许多部件组成的典型的屏幕（Screen）

读者是否已经看出了一个潜在的问题？让我们继续讨论。假设读者研制了许多 screen，并且在系统的某个编译单元 ck.c 中，读者需要包含所有的 screen 头文件（假定要创建它们）。一个窗口应用程序的包含图如图 2-8 所示，该图有 *N*=5 个窗口部件和 *M*=5 个 screen。

当预处理器看到 ck.c 已包含 s1.h 时，预处理器会包含 w1.h 到 w5.h。当遇到 s2.h 时，在寻找结束符#endif的整个过程中，部件的头文件仍然必须重新打开并逐行地重新处理（只是要查明没有其他事情需要做）。这种冗余的处理对于 s3.h、s4.h 以及 s5.h 都会发生。尽管该程序也能被编译并正常地工作，但是在只用 5 个部件头文件就能工作的情况下，我们却不得不等候处理 25 个部件头文件。

除非特别小心，否则 C++倾向于拥有大型的、高密度的包含图（比 C 多很多）。尽管继承和分层有助于解决该问题，但是根本的原因是部分 C++程序员经常被这样的信念所误导：通过在客户的头文件中包含每一个客户可能需要的其他头文件的方式来帮助客户。

```
// ck.c
#include "ck.h"
#include "s1.h"
#include "s2.h"
#include "s3.h"
#include "s4.h"
#include "s5.h"
```

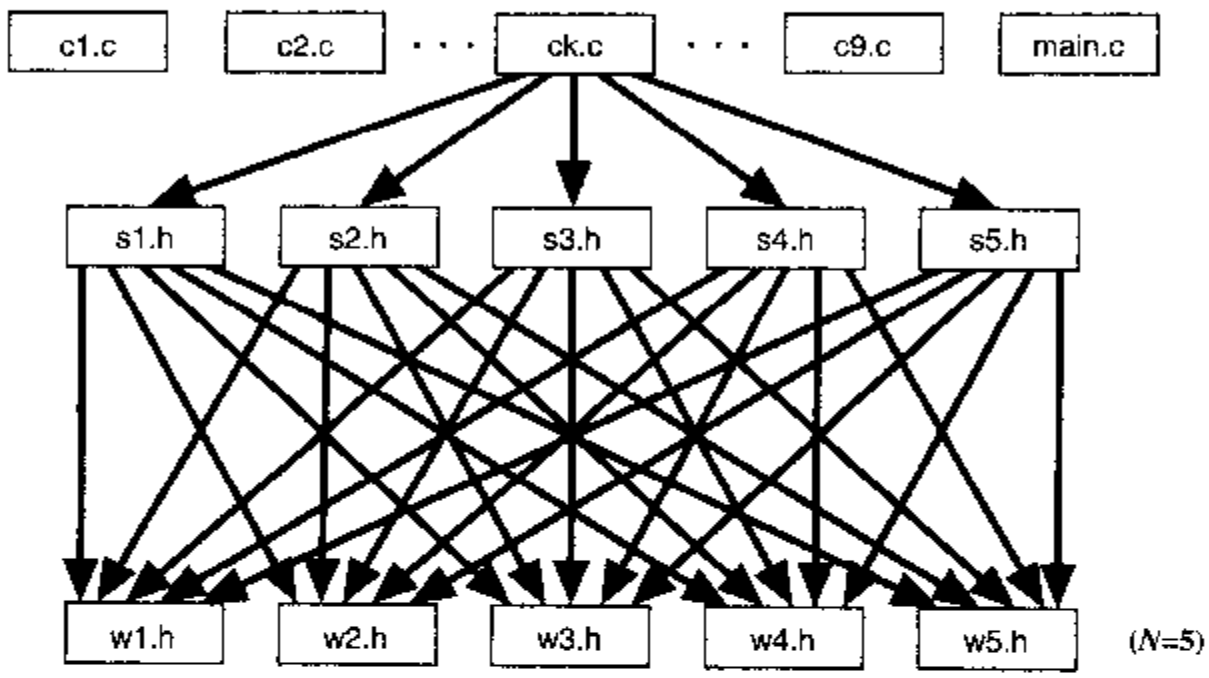


图 2-8　规模 N=5 的窗口系统中的一个组件的包含图

避免高密度的包含图是“绝缘”论题的一部分，将在第 6 章中讲述。下面介绍一个规程，即使在一个不良设计中，遵守该规程也可以减少这样的再收敛包含的影响。注意，有一些开发环境非常智能，可以跟踪记录以前包含的头文件，但是，大多数普通的环境则不能。如果可移植性是一个问题，则最好先保证安全而不要留下遗憾。

次要设计规则

在每个头文件预处理器的包含指令周围，放置一个冗余（外部）包含卫哨。

在每一个出现在头文件中的预处理器包含指令周围放置一个冗余（外部）包含卫哨，这项技术应用于一个典型的 screen 头文件的情况如图 2-9 所示。第一次处理文件 S13.h 时仍然要包含 w1.h、w2.h、…、wn.h。但是，包含另一个 screen 时，不会引起部件头的任何冗余的解析。

注意，数学标准库的头文件的冗余包含卫哨与其他的不同。尽管 math.h 确实有自己的内部包含卫哨，但是它可能不遵守我们的标准。由不同的编译器提供的运行时库，很可能对它们所使用的包含卫哨应用不同的命名规则，并且这些包含卫哨的名称可能不总是一致的。由第三方供应商提供的组件还可能使用另一种规则。对于所有不能保证遵守我们的包含卫哨命名规则的组件，有必要在相应的包含指令之后（如 math.h 中使用的那样）添加一行，用以定义适当的包含卫哨符号。

使用冗余包含卫哨确实令人不快。使用冗余包含卫哨时，在一个头文件中包含一个文件头不只需要一行代码，而是要占三行。若被包含的文件头来自影响范围之外，则要占用四行。冗余包含卫哨不仅使头文件更长，而且使头文件难以阅读。使用冗余包含卫哨还要求遵守一致和可预知命名规则，值得吗？

具有高密度包含图的真正大型项目的开发经验表明，上述问题的回答完全是肯定的。由几百万行 C++源代码构成的项目最初结构，使用大型工作站网络来进行编译要用一星期的时间。插入冗余包含卫哨后，在没有对代码进行本质修改的情况下，能显著地减少编译时间。

```
// s13.h
#ifndef INCLUDED_S13
#define INCLUDED_S13

#ifndef INCLUDED_W1
#include "w1.h"
#endif

#ifndef INCLUDED_W2
#include "w2.h"
#endif

#ifndef INCLUDED_W3
#include "w3.h"
#endif

// ...

#ifndef INCLUDED_WN
#include "wn.h"
#endif

#ifndef INCLUDED_MATH
#include <math.h>
#define INCLUDED_MATH          // extra line
#endif

class S13 {
    W1 d_w1a;
    W1 d_w1b;
    W2 d_w2;
    W3 d_w3;
    // ...
    Wn d_wn;
};

#endif
```

图 2-9 带有冗余包含卫哨的典型的 Screen 组件

一般来说，我们刚刚讨论的问题对于小型甚至中等规模的系统都不是问题。但是，如果我们正在处理相当于包含有数百个基本部件和数百个基本 screen 的问题时，会发生什么事呢？为了提供量化信息以说明使用冗余包含卫哨的好处，我们进行了如下试验。

我们将 *N* 作为部件和 screen 的数量。然后生成子系统并且对单个编译单元测量编译时间（该编译时间主要是C预处理器时间），包括所有带有和不带有冗余包含卫哨的screen头文件。我们对 10 行的头文件和 100 行的头文件进行了试验，将加速因子（speedup factor）定义为：没有带冗余包含卫哨的编译时间除以带冗余包含卫哨的编译时间。结果如图 2-10 所示。

	10 行/头文件			100 行/头文件		
	CPU 秒		加速	CPU 秒		加速
N	没有	有	因子	没有	有	因子
1	0.2	0.2	1.00	0.2	0.2	1.00
2	0.2	0.2	1.00	0.2	0.2	1.00
4	0.3	0.3	1.00	0.4	0.3	1.33
8	0.5	0.3	1.67	0.7	0.4	1.75
16	0.7	0.4	1.75	1.7	0.5	3.40
32	1.5	0.5	3.00	5.8	0.9	6.44
64	5.8	1.1	5.27	22.1	2.0	11.05
128	25.9	3.5	7.40	89.5	5.2	17.21
256	126.5	13.6	9.30	376.5	17.1	22.02
512	702.3	61.6	11.40	1697.4	68.6	24.74
1024	4378.5	306.6	14.28	8303.8	330.6	25.12

图 2-10　带与不带冗余包含卫哨的预处理器时间

对于少于 8 个部件和 8 个 screen 的系统，要么不存在加速，要么加速很小。但是，若总的编译时间小于 1CPU 秒，则无关紧要。

在 C++中，头文件很少只有 10 行；100 行的也还算是小的，但是比较典型。对于一个有 32 个部件的系统来说，在作者的机器上花费在 C 编译器上编译每个客户端组件的时间可以缩短，加速因子超过 6（从 5.8 到 0.9CPU 秒）。对于一个有 64 个部件的系统来说，加速因子超过 11!冗余包含卫哨很难看，但是没有真正的危害。而不使用冗余卫哨则要冒编译时间复杂度为二次方（即 $O(N^2)$）的危险。

注意，冗余卫哨对于.c 文件不是必要的。如果在.c 文件中未能谨慎地复制#include 命令，对于 *N* 个不同的.h 文件，（不合理的）最坏情况下的性能为 $2N$，仍然是线性的（即 $O(N)$）。

本节中的数据反映的是运行于基于 Unix 工作站上的 CFRONT。其他并发环境可能有所不同。在第 6 章中，读者将了解到在头文件中使用嵌套的#include 命令不仅不是所希望的，而且经常是不必要的。冗余包含卫哨的丑陋性提醒我们，应该避免将#include 指令放在头文件中（在这样做有意义的任何时候）。

2.6 文档

本书中的例子并不是为了给产品代码树立什么是足够注释的好榜样（否则，本书就不是一本，而是二本了）。但是，注释，尤其是接口中的注释，是开发过程中的一个重要组成部分。

指导方针

为接口建立文档以便其他人可以使用，至少请另一个开发者检查每个接口。

想要理解为什么让别的开发者检查接口是有价值的，可以假设你自己就是试图理解你的类的客户或测试工程师。你自己非常了解如何使用接口——毕竟是你自己设计的！你用来给成员函数命名的简洁名称是“明显的”和“自解释的”。但除非你已花时间让别人检查了你的接口和文档，否则一定有很多需要改进的地方——特别是在它的可用性方面。

可用性主要是指拿到一个不熟悉的头文件就可以开始使用它。在实践中，头文件注释经常是与接口相关的仅有的文档（或者至少是惟一的更新过的文档）。如果客户必须被迫查看实现以便能够领会到如何使用组件，那么该文档就是不合适的。

指导方针

明确地声明条件（在该条件下行为没有定义）。

文档的另一个重要方面是明确地确定行为没有定义的条件。考虑下面的声明：

```
struct MathUtil {
    // ...
    static int fact(int n);
        // Returns the product of consecutive integers between 1 and n.
};
```

你认为函数 fact 的注释怎么样？我们可能猜测 fact 应该是普通数学函数 factorial($n!$)，fact(0)实际上是 1 而不是 1·0=0 或者没有定义。但是，这不是该注释的意思。该注释没有说明这样的意思：当 n 是非正数时，情况会如何。

阶乘对于负整数值是没有定义的。在这种情况下，我们的特定实现将返回 0。当 n 的值是负数时，fact(n)返回什么值是一种实现技巧，而不是规范的一部分。必须明确告知客户不能依靠这种行为。另一种替代该实现的实现很容易对 n 的负值（包括引起程序崩溃的值）提供不同的行为。

除非明确地在注释中声明，否则客户和测试工程师一般来说没有办法区分什么是特意设计的或必需的行为，什么是仅由特定实现选择引起的巧合行为。一个较好较实用的接口如下：

```
struct MathUtil {
    // ...
    static int factorial(int n);
        // Returns the product of consecutive integers between 1 and n
        // for positive n.  If n is 0, 1 is returned.
        // Note that the behavior is not defined for negative values
        // of n nor for results that are too large to fit in an int.
};
```

没有明确地规定未定义行为的条件，会不经意地使得软件支持无关的行为，这种行为会影响性能或限制实现的选择。如果一个测试工程师不是很内行，读者可能会发现自己的实现选择产生的无关行为会被一套可以明确测试那些行为的回归测试无情地击中。更糟糕的是，通过不适当地（或无意识地）使用，客户可能会依赖这种巧合的行为。

原　则

使用 assert 语句有助于为程序员实现编码时的假设建立文档。

对系统的每一层次进行错误检查，以便找出逻辑错误，这样做代价很高，对于大型系统来说尤为如此。好文档是编写额外代码的可行的替代方案。例如，有些软件开发者认为有必要处理进入函数的每一个指针，即使那个指针为空。如果某个函数是一个广泛使用接口的一部分，那么支持鲁棒性将被证明是一种好的决策。另一个选择是，只要使客户清楚传递一个空指针将导致无定义就够了，可以在函数实现的开始用一个 assert 语句来实现：

```
// stdio.c
#include <stdio.h>
#include <assert.h>
/* ... */
int printf(const char *format ...)
{
    assert(format);
    /* ...
    */
}
```

文档和 assert 语句的有效使用可以使我们得到更简练但仍十分有用的代码。如果有人误用了某个函数，这是他们自己的错——而且他们很快就能发现这个错误！

如果每个开发者总是很清楚函数的指针参数不能为空，这是值得称赞的。但是，负责任的客户不应该假设指针参数可以为空，除非结果行为是明确声明的。

2.7　标识符命名规则

维护一个大型系统时，能以一种一致的、可客观证实的方式区分数据成员、类型和常量

名称与其他标识符可能是一种十分重要的优点。1.4.1 节给出了一个命名规则的集合，在这里我们将用三个设计规则和两个指导原则简洁地将其区分开。

标识类数据成员的词典编纂原则可以简洁地表达出来，它的价值不仅仅是风格问题。因此，将这种原则作为设计规则来介绍。

次要设计规则

使用一种一致的方法（例如加前缀 d_）来突出类成员函数。

读者也可选择使用 s_来区分静态数据和实例数据。上述规则是一条次要规则，因为客户程序决不是一定要处理这类问题（因为根据 2.2 节，数据成员应该总是私有的）。

次要设计规则

使用一种一致的方法（例如第一个字母大写）来区别类型名称。

上述原则被作为一条规则而不是一个指导方针，因为它是一个已被广泛接受并且可客观检验的标准，一般可提高可读性，它使得接口更容易理解，代码更容易维护。它是一个次要规则，因为一个单独的失误并不代表世界末日。

次要设计规则

使用一种一致的方法（例如所有的字母都大写以及使用下划线）来标识不变的值，如枚举值、const 数据和预处理器常量。

上述规则可以帮助我们把常数（因而是“无状态的”）变量与局部变量和成员（状态）变量区别开来。它被作为一个设计规则而不是一个指导方针，因为它有助于提高可维护性，它是客观可验证的，并且它要求没有例外。

指导方针

标识符名称必须一致；使用大写字母或下划线（但不能同时用两种）来分隔标识符中的单词。

上述原则是客观可验证的，但不是每个人都能确信它的优点，并且它很大程度上是一种风格问题。它的用处是可使标识符名称在某种程度上更易于记忆，并可对多数客户展示一种更专业化的形象。在这里它是作为一条指导方针提出来的（尤其对于接口），但在实现中也容许存在某种程度的个性。（在本书中，我们采用大写字母标准）。

指导方针

以相同方式使用的名称必须一致；特别是对于递归设计模式（recurring design patterns）（例如迭代），要采用一致的方法名称和运算符。

在一个大型系统的整个接口中获得一致性，可以增强可用性，但要达到此目的也可能会遇到出人预料的困难。在大型项目中，授权一个顶级开发小组担当“接口工程师”已被证明是有效的，可以跨越开发小组获得一致性。容器类以及它们的迭代器也有助于模板的实现（见10.4节），实现模板可有效地加强跨越其他无关对象的一致性。

2.8 小结

C++是一种大型语言，为更大的设计空间开辟了道路。本章我们描述了基本设计规则和指导方针的一小部分，这些规则和指导方针在实践中被证明是非常有用的。

主要设计规则被认为是绝对不能违反的。甚至偶尔的违反也可能危及大型系统的完整性。在本书中，我们自始至终遵守了所有的主要设计规则。

次要设计规则也被认为是要遵守的，但也许不必严格地遵守。在一个隔离的实例中违反一个次要规则不大可能产生严重的全局性影响。

指导方针是作为经验法则提出来的，因此必须遵守，除非有强制性的工程方面的原因要求遵守别的原则。

把一个类的数据成员暴露给其客户程序违反了封装原则。提供对数据成员的非私有访问意味着表示上的局部改变可能迫使客户重新编写代码。此外，由于允许对数据成员进行可写访问，无法阻止偶然误用导致数据处在不一致的状态。保护的成员数据就像公共成员数据一样，无法限制因数据改变而可能影响到的客户的数量。

全局变量会污染全局名称空间，而且会歪曲设计的物理结构，使得实际上不可能进行独立的测试和有选择的重用。在新的C++项目中没有必要使用全局变量。我们可以通过将变量放置在一个类的作用域中作为私有静态成员、并提供公共静态成员函数访问它们的方法来系统地消除全局变量。但是，对这种模块的过度依赖是一种不良设计的症状。

自由函数，特别是那些不在任何用户自定义类型上操作的函数，在系统集成时很可能与别的函数冲突。将这样的函数嵌套在类作用域中作为静态成员基本上可以消除冲突的危险。

枚举类型、typedefs以及常量数据也可能威胁全局名称空间。通过将枚举类型嵌套在类作用域中，任何二义性都可以通过作用域解析来消除。一个在文件作用域中的typedef看起来有点像类，但是在大型项目中极难发现。通过将typedef嵌套在类作用域中，它们就变得相对容易追踪。一个在头文件中定义的整数常量，其最好的表达方式通常是通过在类作用域中的一个枚举值来表达。其他常量类型可以通过使它们成为某个类的静态常量成员来限定其范围。

预处理宏对于人和机器来说都难以理解。由于宏不是 C++的一部分，所以宏不遵守作用域约束，并且，如果将宏放置在一个头文件中，宏可能与系统中的任何文件的任何标识符冲突。因此，宏不应该出现在头文件中，除非是作为包含卫哨。

总的看来，我们应该避免在一个头文件的文件作用域中引入除了类、结构、联合和自由运算符之外的任何东西。当然，我们允许在头文件中定义内联成员函数。

一个定义被包含两次会引起编译时错误。因为大多数 C++头文件包含定义，我们有必要防止再收敛包含图的可能性。在一个头文件中，用内部包含卫哨围绕定义可以确保每个头文件的内容在任何一个编译单元中最多被加入一次。

冗余（外部）包含卫哨虽然不是一定必需的，但是它可以确保我们避免编译时的二次包含行为。通过用冗余卫哨围绕头文件中的包含指令，我们可以确保每个编译单元最多两次打开一个头文件。

良好的文档是软件开发必不可少的一部分。缺少文档将降低可用性。文档的一个重要部分是声明什么是没有定义的。否则，客户可能会依赖巧合的行为，这种行为只能来自特定的实现选择。

不是所有的代码都必须是鲁棒的。在系统每个层次上的冗余的运行时程序错误检查，可能对性能产生无法接受的影响。文档和断言（assertion）的结合可以达到同样的目的，但在最终产品里可以获得更优越的运行时性能。

最后，提供一个一致的命名规则的集合来区别数据成员、类型和常量，可以提高跨越开发小组的可读性。我们建议对数据成员使用一个“d_”前缀（如果是静态的再加上“s_”）；用第一个字母大写来表示一个类型，而用小写的字母表示一个变量或函数。用所有的字母都大写（包括下划线）表示枚举值、const 数据和预处理器常量；用第一个字母大写来分隔多单词标识符。我们也建议为递归设计模式（如迭代器）使用一致的名称。

第2部分

物理设计概念

用 C++开发大规模软件系统需要的不仅仅是对逻辑设计问题的可靠理解。逻辑实体，例如类和函数，就像一个系统的皮肉。构成大型 C++系统的逻辑实体分布在许多物理实体中，例如文件和目录。物理体系结构是系统的骨架——如果它是畸形的，没有任何美容疗法可以减轻它令人讨厌的症状。

一个大型系统的物理设计质量，将决定其维护开销以及它所具有的让其子系统独立重用的潜能。要进行有效的设计需要彻底掌握物理设计概念，尽管这些概念与许多逻辑设计问题紧密联系在一起，但是对于这些物理设计概念的某一方面，甚至老练的专业软件开发人员都可能缺少经验或者没有经验。本书的第 2 部分全面介绍了优秀物理设计的基本概念。

第 3 章介绍了作为设计基本单位的组件。本章还介绍了若干物理设计规则，以确保我们所有的设计都具有可靠的、重要的和合乎需要的特性。许多逻辑设计关系（例如，IsA、HasA、Uses）都压缩为一种物理关系：**依赖（DependsOn）**。我们会介绍逻辑设计决策如何能够潜在地影响物理依赖。我们也会介绍怎样从一个已存在的组件集合中有效地提取物理依赖。

第 4 章描述了物理层次结构（即，分层）对于开发、维护和测试的重要性。在这一章中我们要探讨如何根据物理依赖来刻画单个的组件、子系统以及整个系统的特性。我们还会介绍如何开发可靠物理设计的层次结构，通过隔离的、增量式的和分层次的测试，以较低的开销来获得较高的可靠性。本章还定量分析了系统中的物理依赖在连接时间和磁盘空间方面对维护开销和回归测试的影响。

第 5 章研究了许多过度连接时依赖的常见原因。这一章罗列了减少一个系统中的连接时依赖的若干技术和变换——一个在本书中称为**层次化（levelization）**的过程。我们使用了许多选自不同的应用程序的实例来说明这些技术。

第 6 章研究了与过度编译时耦合有关的维护开销。这一章展示了若干常用语言结构，这些语言结构会迫使客户依赖于封装好的实现细节；介绍了减轻或消除对个别细节的编译时依赖的技术，以及使客户远离实现的一整套技术——一个在本书中称为**绝缘（insulation）**的过程。最后，以与绝缘有关的运行时开销作为度量标准，讨论了不适当使用绝缘的情形。

第 7 章把层次化的概念扩展到超大型系统。需要超出单独组件物理结构之外的额外物理结构来支持这种系统的复杂功能。**包（package）**代表了一种在物理上内聚的可协同运作组件的集合，并且提供的物理抽象的层次比单独使用组件时所能达到的物理抽象层次更高。在本章中，在把软件包作为一个整体的上下文中我们再次用到了层次化和绝缘的概念。我们也谈到了属于开发和发布一个超大型系统的稳定快照过程的问题。最后，我们讨论了面向对象系统中 main()的角色以及不同初始化策略的相对优点。

3 组件

本章介绍物理设计（与逻辑设计对应）的概念，以组件作为基本的设计单位来介绍。随后研究物理规则的一个子集，它能在大型设计中确保重要的合乎需求的特性。紧接着讨论组件之间的 DependsOn 关系，以及在设计时如何从抽象的逻辑关系中推断出这种关系（DependsOn 关系）。我们还将介绍怎样通过检查组件之间的#include 图表来有效地跟踪物理依赖。最后，我们研究在组件内部和外部对友元关系进行授权的微妙物理含义。

3.1 组件与类

逻辑设计强调系统内部定义的类和函数的交互作用。从纯粹的逻辑角度来看，一个设计可以看作是类和函数的海洋，那里没有物理分区存在——每一个类和自由函数都驻留在一个单一的无缝空间里。交互式的面向对象语言如 Smalltalk 和 CLOS 有丰富的适合于单个开发者的运行环境，无疑已经助长了这种类和自由函数都驻留在单一空间的观点。

但是逻辑设计只考虑了设计过程的一个方面。逻辑设计不考虑像文件和库这样的物理实体。编译时耦合、连接时依赖以及独立重用等都不是仅仅通过逻辑设计就能解决的。例如，一个函数无论是否声明为内联（inline）都不会影响它所要完成的任务，但是可能会极大地影响到它的一些可显式测量的特性，如运行时间、编译时间、连接时间和可执行程序的大小。如果不从物理角度来考虑设计，就不可能考虑到在设计超大型系统时变得很重要的组织问题。

原　则

逻辑设计只研究体系结构问题；物理设计研究组织问题。

物理设计集中研究系统中的物理实体以及它们如何相互关联的问题。在大多数传统的C++程序设计环境中，系统中每一个逻辑实体的源代码都必须驻留在一个物理实体中，一般

称之为一个文件。基本上每一个 C++程序都可以描述为一个文件的集合。这些文件中有一些是头文件（.h），而有一些则是实现文件（.c）。对于小型程序，这种描述足够了。但对于大一些的程序，我们需要利用额外的结构来生成可维护、可测试和可重用的子系统。

> **定义：一个组件（component）就是物理设计的最小单位。**

组件不是类，反之亦然[1]。从概念上来讲，一个组件包含一组逻辑设计的子集，这些逻辑设计使组件作为一个独立的、内聚的单位存在时是有意义的。类、函数、枚举等等都是构成这些组件的逻辑实体。特别地，每个类定义都严格地只驻留在一个组件中。

在结构上，组件是一个不可分割的物理单位，没有哪一部分可以独立地用在其他的组件中。一个组件的物理形态是标准的，并且独立于它的内容。一个组件严格地由一个头文件（.h）和一个实现文件（.c）构成[2]。

一个组件一般会定义一个或多个紧密相关的类和被认定适合于所支持的抽象的任何自由运算符。像 Point、String 和 BigInt 这样的基本类型，每一个都会在包含一个单一类的组件中实现，如图 3-1（a）所示。像 IntSet、Stack 和 List 这样的容器类，一般会在至少包含原理类（principle class）和它的迭代器（iterator）的组件中实现，如图 3-1（b）所示。涉及多种类型的更复杂的抽象（如 Graph）可以在单个的组件中具体化为若干个类，见图 3-1（c）所示。最后，为整个子系统提供包装器（wrapper）的类（见 5.10 节）可以构成薄薄的、由一个或多个原理类和许多迭代器组成的封装层，如图 3-1（d）所示。

图 3-1 中的每一个组件（像每一个其他的组件一样）都既有物理视图也有逻辑视图。物理视图由.h 文件和.c 文件组成，.h 文件被包含在.c 文件中，作为.c 文件的第一行有效语句。组件的物理实现在编译时总是依赖它的接口。这种内部物理耦合要求把这两个文件当作一个单一的物理实体来对待。

原　则

一个组件就是设计的适当的基本单位。

一个组件（不是一个类）是逻辑和物理设计的适当的基本单位，至少有三个理由：

（1）一个组件把便于管理的一定数量的内聚功能（常常跨越多个逻辑实体，例如类和自由运算符）塞进一个单个的物理单位中。

（2）一个组件不仅把捕捉的整个抽象作为单一的实体，而且允许不通过类级设计来考虑物理问题。

---

① 组件的概念在 **stroustrup**（12.3 节，422~425 页）中介绍。在本章中，我们在那个讨论上进行了扩展，介绍了在 C++中使一个组件的定义具体化的物理设计概念。

② 我们将忽略可能允许一个组件有不止一个.h 文件或.c 文件的特殊环境。

图 3-1　若干组件的逻辑与物理视图

（3）一个设计适当的组件（是一个不像类的物理实体）可以作为一个单一的单位从系统中提出来，不必重写任何代码就可以在另一个系统中有效地重用。贯穿本书，这种既考虑物

理设计问题又考虑逻辑设计问题的要求会变得越来越明显。

作为一个具体的实例，图 3-2 显示了组件 stack 的头文件，该组件包含了两个在文件作用域定义的类，即 Stack 和 StackIter。我们还可以看到有两个自由（即不是成员）运算符函数实现了两个 Stack 对象之间的“==”和“!=”。稍微看一下实现，我们就会发现 operator==使用了 StackIter，而 operator!=是根据 operator==来实现的。图 3-3a 显示了组件 stack 的文件作用域中的完整的逻辑实体集合。物理实体（stack.h 和 stack.c）连同它们规范的物理关系在图 3-3（b）中描述。

我们选择了一个简单的堆栈以保证应用程序功能不会遮盖我们要阐明的要点。在这个例子中，几乎每一个成员都有注释（这是为了使产品代码最小化）。堆栈是一种容器。访问一个堆栈的非头元素通常不会认为是堆栈抽象的一部分。我们已经提供了迭代器来保证在这个 stack 组件中定义的功能，在保持封装的同时能被客户更广泛地扩展（见 1.5 节）[①]。我们没有提到最大堆栈容量，因为一个堆栈抽象没有最大容量。提供诸如 isFull 或一个返回状态（该状态反映由不规范实现强加的人为限制）这样的功能，不仅会干扰抽象，而且会使其应用复杂化。这种意外的、基于实现的限制最好视之为异常。但有时候我们会允许客户“帮助”一个对象预见未来的事件，潜在地改进性能。为了避免暴露一个特定的实现选择，这样的“帮助”——像 C 中的 register 或 C++中的 inline——应该只是一个提示而且无法通过编程检测到（见 10.3.1 节）。

> **定义**：一个组件的逻辑接口就是可被客户通过编程访问或检测到的东西。

组件的逻辑接口是定义在头文件中的类型和功能的集合，它们可以被该组件的客户通过编程访问。因为组织原因而驻留在.h 文件中的私有实现细节将被封装，不被认为是逻辑接口的一部分。

> **定义**：一个组件的.h 文件的文件作用域中定义的任何类或声明的任何自由（运算符）函数，其公共（或保护）接口里如果使用了一个类型，则称在这个组件的接口中使用了这个类型（Used-In-The-Interface）。

一个类的公共接口由那个类的公共成员的接口的并集构成（1.6.2 节），同样的道理，一个组件的“公共”接口也是由该组件的.h 文件中所声明的所有公共成员函数、typedef、枚举类型和自由（运算符）函数的集合构成。

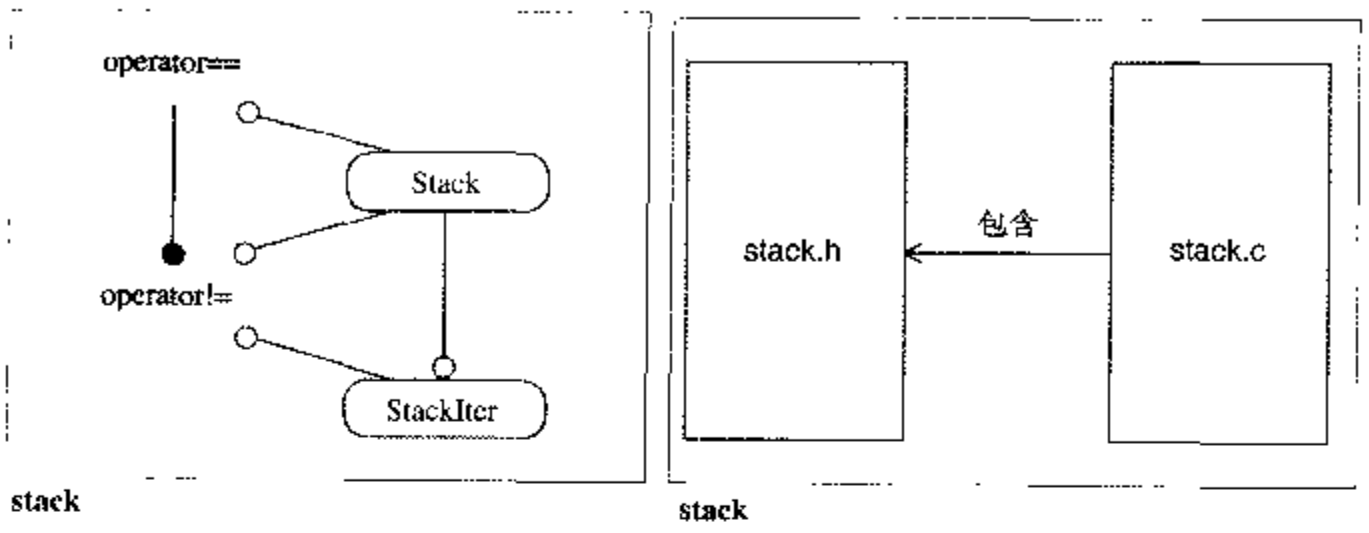
例如，Stack 和 StackIter 的公共成员函数都属于组件 stack 的逻辑接口。自由运算符函数

```
operator==(const Stack&, const Stack&)
```

① 在所有定义容器类的组件中需要一个迭代器，这是在 **meyer**（2.3 节，23~25 页）中讨论的 open-closed 原理的一个直接结果。

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK

class StackIter;

class Stack {
    int *d_stack_p;                          // pointer to array of int
    int d_sp;                                // stack pointer (index)
    int d_size;                              // size of current array of int
    friend StackIter;                        // (no comment needed)

  public:
    // CREATORS
    Stack();                                 // create an empty Stack
    Stack(const Stack& stack);               // (no comment needed)
    ~Stack();                                // (no comment needed)

    // MANIPULATORS
    Stack& operator=(const Stack& stack);    // copy Stack from Stack
    void push(int value);                    // push integer onto this Stack
    int pop();                               // pop integer off this Stack
                                             // undefined if Stack empty
    // ACCESSORS
    int isEmpty() const;                     // 1 if empty else 0
    int top() const;                         // integer on top of this Stack
};                                           // undefined if Stack empty

int operator==(const Stack& lhs, const Stack& rhs);
    // 1 if two stacks contain identical values else 0

int operator!=(const Stack& lhs, const Stack& rhs);
    // 1 if two stacks do not contain identical values else 0

class StackIter {                            // iter order: top to bottom
    int *d_stack_p;                          // points to orig. stack array
    int d_sp;                                // local stack pointer (index)
    StackIter(const StackIter&);             // not implemented
    StackIter& operator=(const StackIter&); // not implemented

  public:
    // CREATORS
    StackIter(const Stack& stack);           // initialize to top of Stack
    ~StackIter();                            // (no comment needed)

    // MANIPULATORS
    void operator++();                       // advance state of iteration
                                             // undefined if done
    // ACCESSORS
    operator const void *() const;           // non-zero if not done else 0
    int operator()() const;                  // value of current integer
};                                           // undefined if done

#endif
```

图 3-2　组件 stack 的头文件 stack.h



（a）逻辑视图　　（b）物理视图

图 3 3　组件 stack 的两种视图

不是 Stack 的成员，因此不被认为是类 Stack 的逻辑接口的一部分。但是，这个运算符会扩展组件 stack 中定义的可编程访问的函数集合，因而也会扩展组件的逻辑接口。这个有关友元关系的微妙的问题在 3.6 节讨论。

图 3-4 显示了一个小型驱动程序 stack.t.c，它创建了一个 Stack 对象，在其中压入了一些整数，然后按照从头到尾的顺序把它的内容打印出来。这个驱动程序可以访问 stack 组件的逻辑接口，但不能访问它的组织结构。然而，stack 组件的.h 文件中有比可编程访问的信息更多的信息。

```
// stack.t.c
#include "stack.h"
#include <iostream.h>

main()
{
    Stack stack;
    stack.push(111);
    stack.push(222);
    stack.push(333);

    for (StackIter it(stack); it; ++it) {
        cout << it() << endl;
    }
};

// Output:
//      333
//      222
//      111
```

图 3-4　组件 stack 的驱动程序 stack.t.c

> **定 义**：一个组件的**物理接口**就是它的头文件中的**所有东西**。

一个组件的物理接口由.h 文件中可利用的所有信息构成（不考虑访问特权）。组件的.h 文件中包含的信息越多，对组件实现的修改就越有可能影响组件的客户程序而导致它们要重编译。

任何程序员看到 stack.h 就可以断定这是一个基于数组的堆栈（它还未实现，例如，实现为一个链表）。一个编译器要完成工作，在整个过程中都必须查看 stack.h。编译器甚至必须考虑私有信息（例如，d_stack_p），从逻辑角度来看，这些信息确实是一个实现细节。这种物理暴露的结果是，一个未触及 stack 组件接口逻辑视图的实现变化仍然可能强制包含 stack.h 的所有客户重新编译。

作为程序员，我们观察到在 stack.h 中没有函数声明为内联（inline）。修改这个组件中的任何函数体都不会因此改变物理接口而迫使客户程序重新编译。不利的方面是对于像 Stack 这样的轻量级对象，删除内联函数可能导致运行时性能方面的一个数量级的损失（见 6.6.1 节）。

> **定 义**：如果在一个组件的任何地方通过名称引用了一个类型，则称在**这个组件的实现中使用了这个类型（Used-In-The-Implemention）**。

从逻辑角度看，在组件的实现中有或没有使用什么都是封装细节，并不重要。从物理角度看，这样的使用可能隐含着对其他组件的物理依赖。在大型系统中正是这些物理依赖会影响可维护性和可重用性。

优秀的设计要求开发者既懂得逻辑设计知识又懂得物理设计知识。逻辑设计是天然的出发点。我们必须考虑哪些逻辑实体应该天然地在一起或者充分相互依赖，因此不能被合理地分割。我们也必须考虑在物理接口上需要暴露多少实现细节。而且我们还必须决定我们的组件会依赖于哪些其他的组件，在这些组件中有哪些变化会对我们自己的组件及其客户程序产生影响。所有这些问题都已经研究解决之后，才正确地设计了一个组件。

3.2 物理设计规则

本节介绍物理设计的基本规则。如果要使我们其他的规程和技术生效，这些规则是必要的。若一个大型设计从一开始就没有从本质上遵循这些规程，那么我们事实上不可能改正它。

主要设计规则

在一个组件内部声明的逻辑实体不应该在该组件之外定义。

这似乎是显而易见的，但是这条规则应该明确地陈述一次。对一个可重用的组件来说，它必须合理地自我包含。一个组件可以有对其他组件的依赖，但是，一个组件在它的头文件

中声明的任何逻辑结构（类声明除外）——如果定义了——则应该全部定义在该组件内部。

图 3-5 是一个如何不把逻辑实体分割成物理单位的例子。类 Stack 已经定义在组件 stack 中，但是它的实现没有限制在组件 stack 内。Stack::push 定义在 intset.c 中，而 Stack::pop 却定义在 main.c 中！

```
// intset.h
#ifndef INCLUDED_INTSET
#define INCLUDED_INTSET
class IntSet {
    // ...
  public:
    // ...
    // ...
    // ...
};
#endif
```

Intset.h

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK
class Stack {
    // ...
  public:
    // ...
    void push(int i);
    int pop();
};
#endif
```

stack.h

```
// intset.c
#include "intset.h"
// ...
Stack::push(int i)
{
    // ...
}

// ...
```

Intset.c

```
// stack.c
#include "stack.h"
// ...
// ...
// ...
// ...
// ...

// ...
```

stack.c

```
// main.c
#include "intset.h"
#include "stack.h"
int Stack::pop()
{
    // ...
}

// ...
```

main.c

图 3-5 逻辑实体的不合理的物理分割

不能严格遵守上述设计规则除了引起维护麻烦，还可能导致一个设计的许多合乎需求的物理性质的丢失（特别是在其他程序中获得和重用编译单元的能力）。小心谨慎地遵循这条规则将会改善任何规模项目的模块性和可维护性。

次要设计规则

组成一个组件的.c 文件和.h 文件的根名称应该严格匹配。

严格匹配一个组件文件的根名称，对于可维护性来说是重要的。例如，知道 stack.c 和 stack.h 组成一个组件，不仅方便手工维护，而且为简单的面向对象设计自动工具打开了大门（见附录 C）。

但是，一些已存在的对象代码档案文件在对象文件名称上设置了相对较低的字符数限制

（例如i：13）。因此组件的.c 文件的名称不是总能与它的主要类的名称对应。更糟糕的是，一些操作系统限制文件名称只允许有八个字符（加上三个字符的后缀），在开发超大型系统时这可能会是一个相当大的负担。

主要设计规则

每个组件的.c 文件都应该将包含它自己的.h 文件的语句作为其代码的第一行有效的语句。

我们必须把组件的.h 文件包含在其.c 文件中，因为编译器必须先看到一个类成员的声明才能编译它的定义。这个惯例是C++语言以及许多常用的独立分析工具所要求的。把#include指令放在文件顶端的原因则有些微妙。

原　则

通过确保一个组件自己分析自己的.h 文件——不要外部提供的声明和定义，可以避免潜在的使用错误。

把包含.h 文件的语句作为.c 文件的第一行，可以确保组件物理接口的固有信息之关键段不会从.h 文件中遗漏（或者，如果有的话，在你试图编译.c 文件时会马上发现它）。

考虑下面组件 wildthing 的头文件：

```
// wildthing.h
#ifndef INCLUDED_WILDTHING
#define INCLUDED_WILDTHING

class WildThing {
    // ...
  public:
    WildThing();
    // ...
};

ostream& operator<<(const ostream& o, const WildThing& thing);
    // Note: uses class ostream in the interface

#endif
```

注意，我们已经重载了左移运算符（<<），所采用的方法对于数据流输出是正常的，也是惯例。下面考虑实现：

```
// wildthing.c
#include <iostream.h>
#include "wildthing.h"
// ...
```

```
ostream& operator<<(const ostream& o, const WildThing& thing)
{
    // ...
}
```

我们尝试编译这个实现，它编译得很好。下一步我们为 wildthing 创建一个测试文件：

```
// wildthing.t.c
#include <iostream.h>
#include "wildthing.h"

int main()
{
    WildThing wild;

    // ...
    // ...

    cout << wild << endl;

    return 0;
}
```

文件 wildthing.t.c 编译和连接了。程序运行得很好，然后我们去告诉所有的朋友我们完成了。但是，这里面有一个错误并且是一个物理错误！下面的程序不能通过编译，为什么？

```
// product.c
#include "wildthing.h"
#include <iostream.h>

int main()
{
    WildThing wild;

    // ...
    // ...

    cout << wild << endl;

    return 0;
}
```

问题出在我们试图在 operator<<（在 wildthing.h 中声明的）的接口中使用类 ostream 时没有事先声明它。客户代码中#include 指令的顺序颠倒了，现在头文件自身不能进行语法分析，因为 ostream 标识符还没有声明。我们怎样处理这个问题呢？

一旦找出了错误，修正就很简单：在第一次使用 ostream 之前将声明“class ostream”[1]

[1] 不是预处理器指令#include <iostream.h>（如同 6.3.7 节所解释的那样）。

加入 wildthing.h 的文件作用域：

```
// wildthing.h
#ifndef INCLUDED_WILDTHING
#define INCLUDED_WILDTHING

class ostream;          // was missing before, oops!

class WildThing {
    // ...
  public:
    WildThing();
    // ...
};

ostream& operator<<(const ostream& o, const WildThing& thing);

#endif
```

更重要的问题是我们怎样预防此类问题？答案同样简单。始终让每个组件的.c 文件在包含或声明任何别的东西之前包含该组件的.h 文件。通过这种方法，每个组件都能确保它自己的头文件对于编译来说都是能自我满足的。

指导方针

客户程序应该包含直接提供了所需类型定义的头文件；除了非私有继承，应避免依赖一个头文件去包含另一个头文件。

一个头文件是否应该包含另一个头文件是一个物理问题，不是逻辑问题。为了编译，头文件本身需要另一个头文件中的一个定义，在这样的情况下（见 6.3.7 节），把适当的#include 指令（当然，要被冗余外部包含卫哨所包围，如 2.5 节所述）放在那个头文件中是正确的。

但是，除了公共和保护的继承之外，包含一个类型的定义而不是预先在头文件中声明它，这种需求几乎总是由封装的逻辑实现细节来决定。

例如，如果类 Stack 在它的实现中使用了类 MyType，可能有必要在 mytype.h 中包含 stack.h，以确保 mytype.h 的编译。如果决定用 List 代替 Stack 来改变 MyType 的实现，则不再需要 mytype.h 中的#include "stack.h"指令。任何依赖 mytype.h 包含 stack.h 的客户程序现在将不得不修改成直接包含 stack.h。

我们如何将一个类型在另一个类型上进行分层，这将影响编译时耦合的程度。逐渐减少编译时耦合是 6.3 节的主题。例如，使用 MyType HasA（嵌入）Stack 还是 HoldsA（指向）Stack 决定了是 mytype.h 包含 stack.h，还是简单地事先声明类 Stack（见 6.3.2 节）。如果改变了 MyType 的实现，用 HoldsA（代替 HasA）Stack，那么#include 指令可能不再需要写在 mytype.h 中。如果删除了该条指令，那么客户程序也将被迫作出改变（这些客户程序依赖于在 MyType

的实现中使用 Stack 的方式）。即使 MyType 的逻辑接口中使用了 Stack，mytype.h 仍然可能不需要包含 stack.h（见 6.3.7 节）。这就要求每一个实质上使用 Stack 的用户程序都直接包含它的定义。

继承是一个例外，因为继承总是隐含一个编译时依赖，并且是派生类的逻辑接口的一部分。改变继承层次结构（采用允许组件作者从组件头文件中删除#include 指令的任何方式来改变），也将改变逻辑接口，同时将迫使客户程序被再次访问（不考虑物理问题）。因此，客户程序只包含一个派生类的定义，并依赖这个派生类的头文件来包含基类的定义是合理的。

因为类似的原因，客户程序依赖某个组件的头文件来事先声明一个只用在该组件的逻辑实现中的类是不明智的。

主要设计规则

在一个组件的.c 文件中，避免使用有外部连接并且没有在相应的.h 文件中明确声明的定义。

为了分析、维护，尤其是测试，确保某人（或某工具）只看到一个组件的物理接口就能了解那个组件的全部逻辑接口是很重要的。要求一个组件在它的头文件中声明它的完整的逻辑接口有助于提高：

（1）可用性——使客户程序只从接口就可以全面了解一个组件所支持的整个抽象。

（2）可重用性——确保组件提供的所有支持功能对所有客户都是同样可访问的。

（3）可维护性——避免不支持的“后门”接口（它会干扰组件支持的抽象）。

假设某人在组件 foo 的.c 文件中定义了一个外部自由函数（或变量），在 foo.h 中，没有把它作为外部函数（或变量）进行声明。另一个刚好和 foo 相连接的组件 bar，通过局部创建适当的外部声明可以获准访问那个函数（或变量）。图 3-6 中描述了这个不幸的场景。请注意这个例子描述的是一个糟糕的设计（我已尽量避免在本书中展示这种例子）。

如图所示，bar 的.c 文件依赖于 foo 的物理实现提供的定义，但它独立于 foo 的物理接口。这里有一个 foo 的“后门”用法和一个 bar 对 foo 的隐含物理依赖很难被发现。只考虑#include 图的 makefile 的自动依赖产生器（mkmf、gmake 等等）很难找出这种微妙依赖的线索。此外，这种代码的维护者也没有这两个组件耦合的直接证据。可是，当我们去重用 bar 时，在连接阶段会失败，因为函数 f（以及全局变量 size）的定义将会丢失。

如果在 foo.h 中指定完整的接口，客户组件 bar 就可以简单地在它自己的.c 文件中包含 foo.h 文件，使 bar 的实现对 foo 的接口的依赖显性化。这种新的稍有改进的实现在图 3-7 中描述。但是，一个外部全局变量或一个外部自由函数的使用仍然违背了在 2.3.1 节和 2.3.2 节介绍的设计规则。

完全定义在一个组件.c 文件的文件作用域中的类，可能会很容易违背这条规则，因为非内联类成员函数和静态成员数据有外部连接。如果我们对完全定义在一个.c 文件中的类施加

某种约束，我们就可以避免创建外部定义，从而避免违背这条规则。这种约束与 C++语言本身对局部类定义（即完全定义在一个单一函数内的类）[1]所施加的约束是一样的。

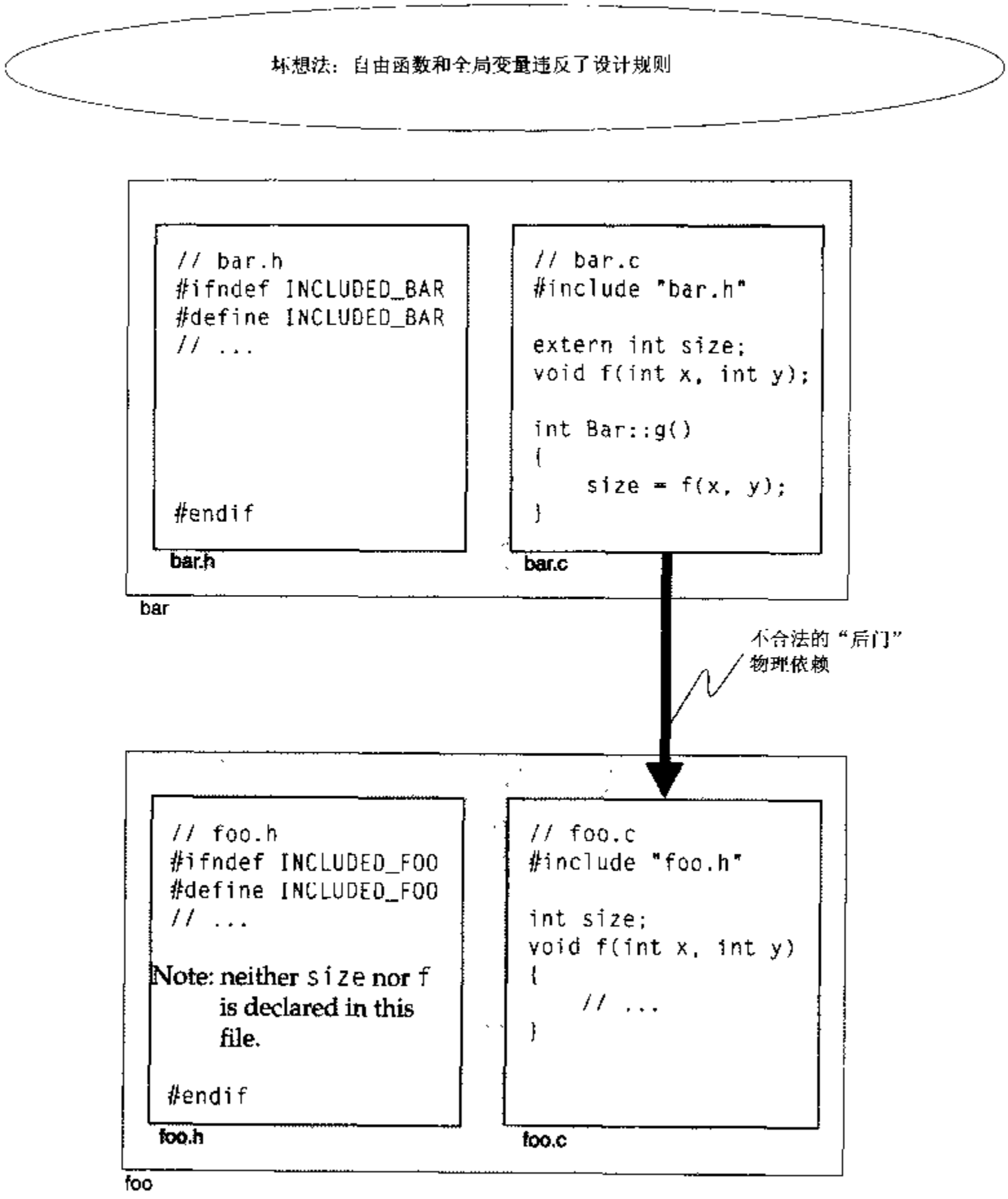


图 3-6　bar.c 直接依赖 foo.c 的糟糕的物理设计

① ellis，9.8 节，188~189 页。

坏想法：自由函数和全局变量仍然违反了设计规则

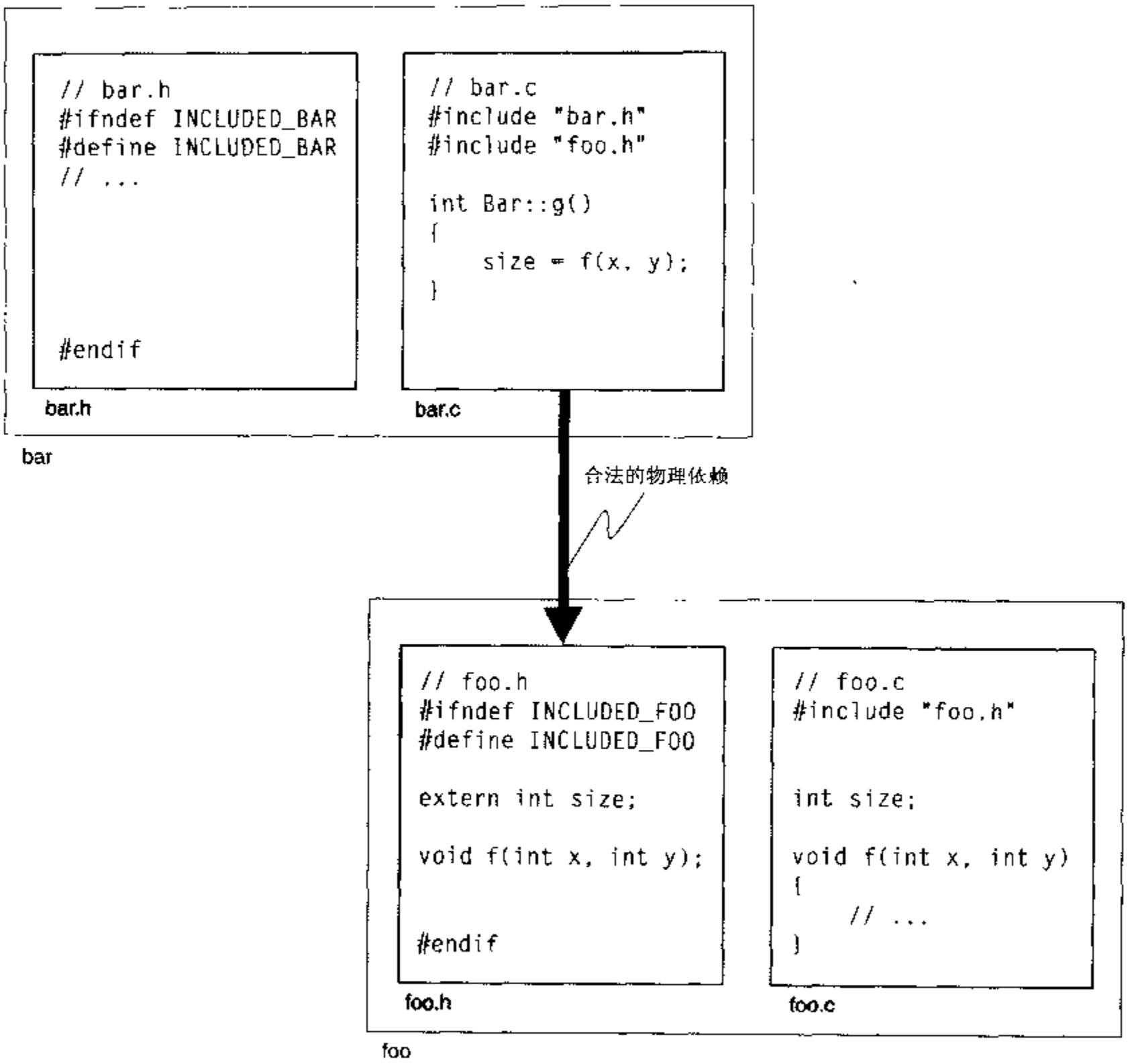


图 3-7　bar.c 依赖 foo.c 的（稍微）好些的物理设计

在一个.c 文件内完整地定义一个类，虽然在技术上违背了一条规则，但是在实践中是相对无害的，因为名字冲突趋向于阻止人们尝试直接使用外部符号。惟一真正的危险是外部定义可能会与一些其他的同样的定义相冲突（如果那个类定义在它自己的单独的组件中仍会是

同样的情形）。避免在一个.c 文件内完整定义类的一个更重要的理由也许是这种类随后不能被直接测试（见 8.4 节）。

避免后门用法对于好的物理设计和有效的重用来说是关键的。只要求组件作者这样做是不够的。要堵上所有的漏洞，我们必须制定一种对大家都有好处的要求：客户程序不能试图通过局部声明来使用任何带有外部连接的结构。相反，客户程序要包含一个组件的.h 文件，以便访问组件所提供的任何定义。

主要设计规则

避免通过一个局部声明来访问另一组件中带有外部连接的定义，而是要包含那个组件的.h 文件。

遵循这条规则的理由主要是要让对其他组件中的外部定义的依赖显性化。

用包含头文件来取代提供一个局部函数声明，对客户也有好处。有时候文件头会改变。你的局部声明将如何改变来反映这些文件头的变化？一个声明错误的有 C++连接的函数至少在连接时就可以被发现[1]，但是来自标准 C 库的不正确的函数局部声明（带有 C 连接），可能直到运行时才能发现。

例如，下面的程序可以编译和连接：

```
// foo.c
#include "foo.h"
extern "C" double pow(double, int); // bad idea: local extern declaration

double Foo::func(double x, double y)
{
    return pow(x, y) + pow(y, x);
}
```

然而，我们在运行时会得到错误的结果，因为局部 extern 声明与目前的 pow 定义不匹配[2]：

```
extern "C" double pow(double, double)
{
    /* ... */
}
```

不匹配的声明将导致 pow 的第二个参数变得混乱不清。我们可以通过包含.h 文件来避免这样的问题并使依赖显性化：

① 见 **ellis** 的 Type-Safe Linkage，7.2c 节，121~126 页。

② **plauger**，第 7 章，138 页。

```
// foo.c
#include "foo.h"
#include <math.h>      // pow()

double Foo::func(double x, double y)
{
    return pow(x, y) + pow(y, x);
}
```

通过包含头文件，与具有任何一种连接特征的函数的不一致之处都将在**编译时**捕获，这样比在连接时或运行时捕获要好得多。

有一件重要的事要说明，以下形式的类声明：

```
class QueueLink;        // forward declaration of class QueueLink
```

是完全不同的问题，因为类定义有内部连接。这样的声明不仅是普遍的而且是合乎需要的，尤其是在它们可以消除头文件中的预处理器#include 指令的地方。类声明的这种用法与连接时依赖有关，将在 5.10 节中讨论；与编译时依赖有关的用法，将在 6.4.3 节中讨论。

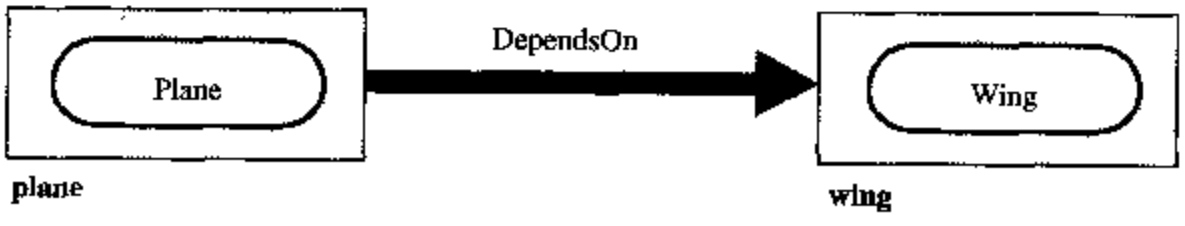
3.3 依赖（DependsOn）关系

一个系统的组件之间的物理依赖，将影响系统的开发、维护、测试和独立重用。类和自由（运算符）函数之间的逻辑关系，隐含了它们所驻留组件之间的物理依赖。如果编译和连接一个函数体时需要一个组件，那么通过说那个函数依赖于那个组件，我们可以宽松地为函数定义实现依赖。我们也可以用相似的方法为类定义实现依赖。更普遍地，我们可以精确地定义组件间的重要而纯粹的物理关系。

> **定 义**：如果编译或连接组件 y 时需要组件 x，则组件 y **DependsOn** 组件 x。

DependsOn 关系非常不同于我们已经介绍的关系。IsA 和 Uses 是逻辑关系，因为它们应用于逻辑实体，与逻辑实体驻留的物理组件无关。DependsOn 是一种物理关系，因为它应用于作为一个整体的组件，组件本身是物理实体。

用来表现一个物理单位依赖另一个物理单位的符号是一个（肥大的）箭头。例如：



意味着组件 plane 依赖组件 wing。也就是说，除非组件 wing 也是可用的，否则不能使用

组件 plane（即，它不能连接进一个程序或甚至不可能编译）。

逻辑实体用椭圆形表示，而物理实体用长方形表示，这已是我们的惯例了。注意，用来指出物理依赖的箭头画在组件之间而不是单个的类之间。永远不要将用来表示物理依赖的（肥大的）箭头符号和用来表示继承的箭头符号相混淆。继承箭头总是在两个类之间（类是逻辑实体）；DependsOn 箭头则连接物理实体（例如文件、组件和包）。

为了动态地描述 DependsOn 关系，考虑下面的一个串组件的框架头文件。顺便说一句，不要试图把组件命名为“string”；在标准 C 库头文件 string.h 存在的情况下它也许不能很好地工作。

```
// str.h
#ifndef INCLUDED_STR
#define INCLUDED_STR

#ifndef INCLUDED_CHARARRAY
#include "chararray.h"
#endif

class String {
    CharArray d_array;  // HasA
    // ...
  public:
    // ...
};

// ...

#endif
```

这里刚好有足够的可见信息使我们可以了解到，类 String 有一个 CharArray 类型的数据成员。我们从 C 可以知道，如果一个 struct 有一个用户自定义类型的实例作为数据成员，那么即使对 struct 的定义进行语法分析，也必须知道那个数据成员的大小和布局。

> **定 义**：如果编译 y.c 时需要 x.h，那么组件 y 展示了对组件 x 的**编译时**依赖。

更明确地说，如果不首先包含 chararray.h，就不可能编译任何需要 String 定义的文件。为此我们将#include "chararray.h"连同伴随物——冗余包含卫哨——一起嵌入组件 str 的头文件是正确的。

图 3-8 描述了组件 str 对组件 chararray 的物理依赖。一个组件的.c 文件在编译时必须总是依赖它的.h 文件。因为 str.c 没有 str.h 就不能编译，而 str.h 没有 chararray.h 不能编译，所以 str.c 有一个对 chararray.h 的间接的编译时依赖。请再次注意用于表示物理依赖的箭头是画在两个物理实体（在此例中是文件）之间的。一种组件级的更抽象的表示法如图 3-9 所示。

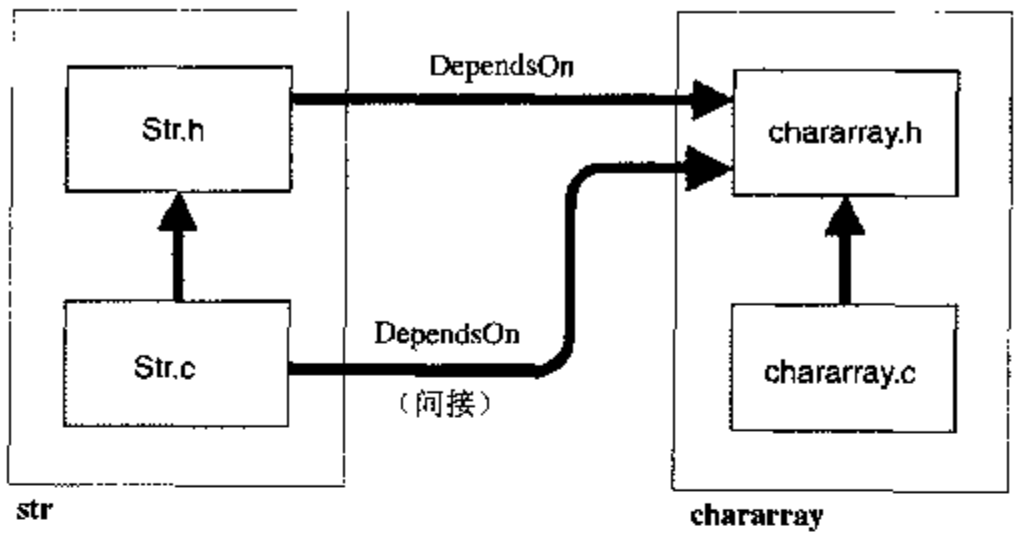


图 3-8 str.c 对 chararray.h 的间接的编译时依赖

图 3-9 组件依赖的抽象表示法

一个组件可能在编译时不必依赖另一个组件，而在连接时要依赖它。考虑下面的组件 word 的实现和组件 str 的候补实现：

```
// word.h
#ifndef INCLUDED_WORD
#define INCLUDED_WORD

#ifndef INCLUDED_STR
#include "str.h"
#endif

class Word {
    String d_string;  // HasA
    // ...
  public:
    Word();
    // ...
};

#endif
```

```
// str.h
#ifndef INCLUDED_STR
#define INCLUDED_STR


class CharArray;


class String {
    CharArray *d_array_p;  // HoldsA
    // ...
  public:
    String();
    // ...
};

#endif
```

```
// word.c
#include "word.h"


// ...
```

```
// str.c
#include "str.h"
#include "chararray.h"

// ...
```

编译 chararray.c 当然需要 chararray.h。编译 str.c 时需要 str.h 和 chararray.h。最后，编译 word.c 时需要 word.h 和 str.h。注意编译 word.c 时不需要 chararray.h。组件 word 对组件 chararray 没有编译时依赖。但是，word 仍然展示了对 chararray 的物理依赖，一旦我们将一个测试驱动程序连接到 word，这种依赖就会变得明显。

> **定 义**：如果对象文件 y.o（通过编译 y.c 产生）包含未定义的符号，因此可能在连接时直接或间接地调用 x.o 来辅助解析这些符号，那么就说组件 y 展示了对组件 x 的一种**连接时依赖**。

回顾一下，除了内联函数，在 C++中所有的类成员函数和静态数据成员都有外部连接。对于所有的实际用途，我们都可以说如果一个组件为了编译需要包含另一个组件，那么它将在连接时依赖那个组件在对象代码级解析未定义的符号。

原 则

一个编译时依赖几乎总是隐含一个连接时依赖。

正如图 3-10 所示，word.o 依赖定义在 str.o 中的外部名称。即使 word.o 不直接使用定义在 chararray.o 中的名称，它却使用了定义在 str.o 中的名称。用在 str.o 中解析这些未定义符号的名称仍将可能引入新的未定义的名称，它们的定义必须由 chararray.o 提供。综上所述，我们得出了一个有趣而重要的结论。

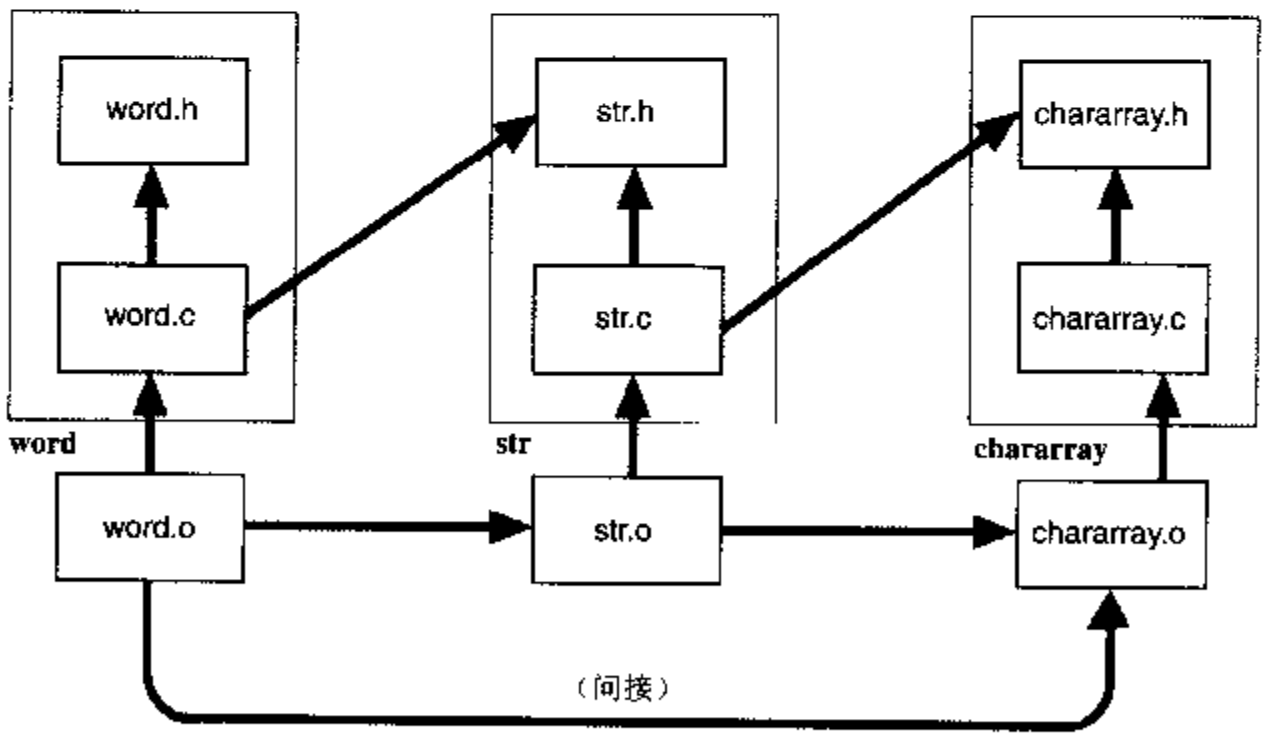


图 3-10　word 对 chararray 的连接时依赖

原 则

组件的 DependsOn 关系具有传递性。

例如，假设 x、y 和 z 是组件。如果 x 依赖 y，而 y 依赖 z，那么 x 依赖 z。组件之间的这种传递性质并没有涉及一个组件中的哪一个文件依赖另一个组件中的哪一个文件。任何这样的文件级依赖都足以使作为一个整体的组件产生实现依赖。

前面这个例子的抽象的、组件级依赖关系显示在图 3-11 中。word 对 str 的编译时依赖和 str 对 chararray 的编译时依赖已经产生了 word 对 chararray 的间接（连接时）依赖。

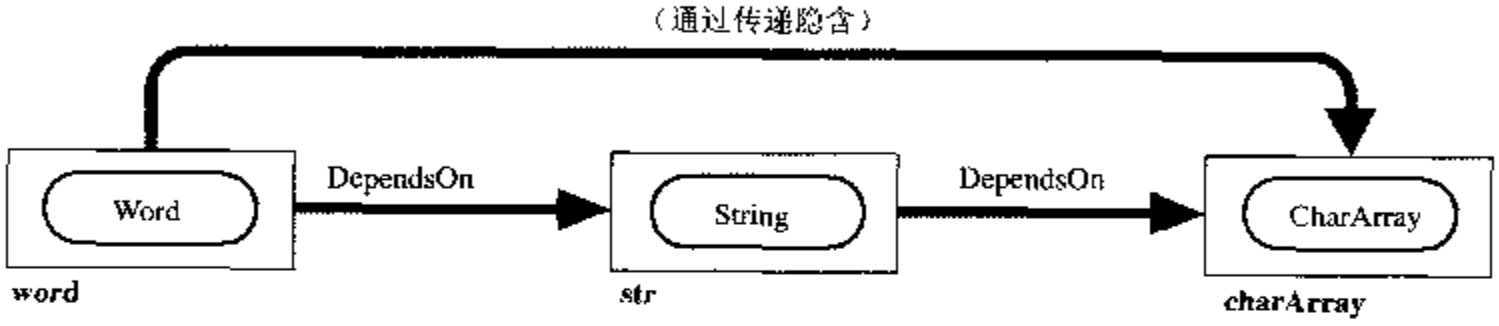


图 3-11　抽象的、组件级依赖关系

DependsOn 关系对物理设计来说是很重要的，因为它指明了在维护、测试和重用时由一个给定的组件所提供的功能所需要的所有组件。我们刚刚已经介绍了如何从源代码本身来推断物理依赖。正如我们将在 3.4 节介绍的，直接从像 IsA 和 Uses 这样的抽象逻辑关系来推断物理依赖是可能的。在设计阶段推断物理依赖有助于我们在开发过程的早期就获得一个可靠的物理体系结构。

3.4　隐含依赖

逻辑设计隐含一定的物理特性。在我们的逻辑设计实现**之前**，我们期望能够充分利用已知的逻辑关系来预测它的物理隐含。读者可能经常因为逻辑设计导致了不合需要的物理特性而被迫修改甚至完全重做逻辑设计。在这一节，我们从 Uses 关系开始集中讨论逻辑设计中有关物理依赖的隐含关系。

原　则

定义了某个函数的组件，通常会物理依赖于定义了某个类型（那个函数使用了该类型）的任何组件。

除非另外指定，否则我们将假设，如果一个函数使用了一个用户自定义类型，那么它会采用一种“实质的”方式来这样做。为解释“实质的”是什么含义，让我们暂时假设，若一个函数在它的接口中使用了一个类型，则定义了该函数的组件就必然要包含定义了该类型的组件的.h 文件。

图 3-12 描述了我们的假设：如果函数 Two::getInfo 在它的接口中使用了类 One，那么它可能会在它的实现中用 One 做某种事情，这就要求看到 One 的定义。在这个例子中，Two::getInfo

调用了类 Onc 的 const 成员函数 info，这就要求编译器在编译 two.c 时要找到 One 的定义。

```
// two.c
#include "two.h"
#include "one.h"
// ...
int Two::getInfo(const One& one)
{
    return one.info();
}
// ...
```

```
// one.h
#ifndef INCLUDED_ONE
#define INCLUDED_ONE
// ...
class One {
    // ...
    int info() const;
    // ...
};
// ...
#endif
```

图 3-12　Uses 关系常常隐含一个编译时依赖

Uses 关系隐含一个编译时依赖的假设太强了。但是，这个假设相当精确地预测了物理实现依赖。Uscs 关系没有必要为了减少一个间接的物理依赖而引起一个编译时依赖。为了了解一个间接的连接时依赖怎样才能出现，可以考虑在前一个例子中再加上另一个组件 three 和另外两个文件 two.h 和 three.c。

如图 3-13 所示，three.c 定义了一个成员函数 x2info，x2info 在它的接口中使用了 One。但是，x2info 的参数是通过引用传递的，x2info 在把它的参数传递给 Two 的静态成员函数 getInfo 之前，并没有实质地使用 One 的定义。getInfo 也通过引用接受了一个 One 对象。组件 three 中的 x2info 函数把类 One 看成是不透明的，除了知道它是一个类、结构或联合之外，不知道有关 One 的任何事情。

```
// three.c
#include "three.h"
#include "two.h"
// ...
int Three::x2info(const One& one)
{
    return 2 * Two::getInfo(one);
}
// ...
```

```
// two.h
#ifndef INCLUDED_TWO
#define INCLUDED_TWO
class One;
// ...
class Two {
    // ...
  public:
    static int getInfo(const One& one);
    // ...
};
// ...
#endif
```

图 3-13　Uses 关系几乎总是隐含一个连接时依赖

假设类 Three 中没有其他的函数（实质地）使用了 One，而且组件 three 对组件 one 没有其他的编译时依赖。也就是说，编译 three.c 只要包含 three.h 和 two.h 就足够了，即使 three

的接口中使用了 one。但是函数 x2info 是间接依赖 One 的。如果我们要测试 three，我们必须先编写和编译 two.c 才能够连接。要那样做，two.c 就必须包含 one.h。

如图 3-14 所示，由于使用一个对象而引起的实现依赖具有传递性。就是说，如果 Three 用了 Two 而 Two 用了 One，那么定义了 Three 的组件（几乎总是）DependsOn 定义了 One 的组件。在这个案例中，组件 three 中的函数 x2info 使用了 One（一般地），但也使用了定义在组件 two 中的函数 getInfo。函数 getInfo 又在它的接口中使用了 One，但这次 getInfo 在它的接口中实质性地使用了 One。

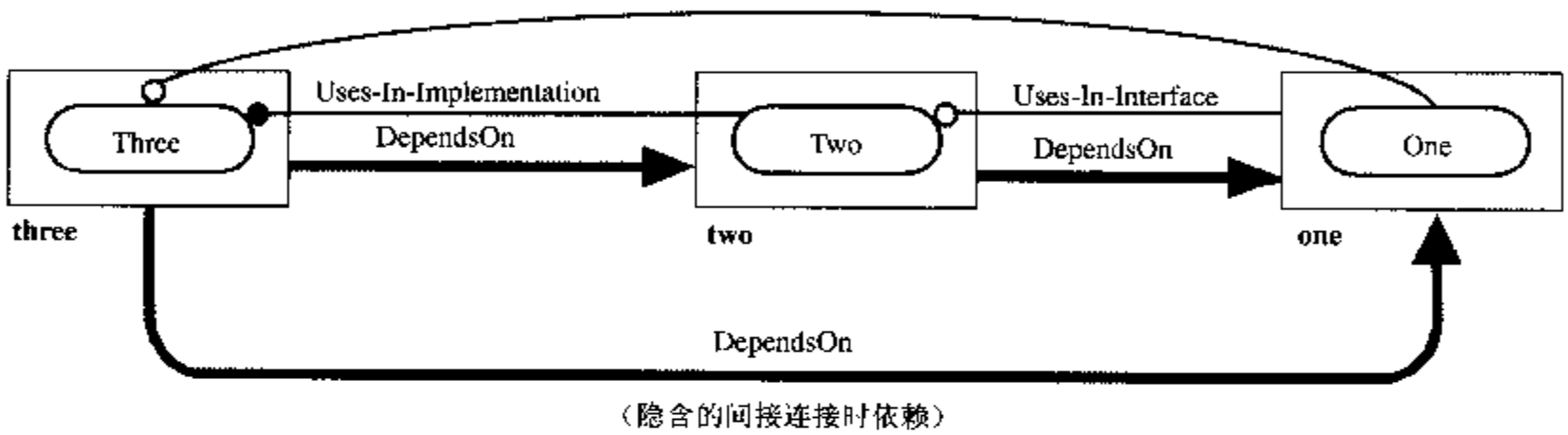


图 3-14 逻辑使用（Logical Uses）关系隐含组件依赖

有这种可能：使用了一个对象而没有引起对该对象的编译时依赖或连接时依赖。在实践中，这种受限的使用有时会在设计中出现，但几乎永远不会碰巧出现。我们将在 5.4 节中探讨这种设计技术。

原 则

如果一个组件定义了某个类，且该类 IsA 或 HasA 用户自定义类型，那么那个定义了类的组件总是在编译时依赖那个定义了类型的组件。

某些逻辑关系有很强的物理隐含。例如，从一个类型派生（IsA）或嵌入一个类型的实例（HasA）总是隐含一个类将在编译时依赖那个类型。事实上，这些逻辑关系隐含的编译时依赖不仅针对类本身，也针对这个类的任何客户程序。

图 3-15 描述了图 3-11 例子中的 IsA 和 HasA 的物理隐含。这一次 Word 被重新实现为一种 String，而 String 有一个 CharArray 数据成员[①]。为了编译 word.c，String 和 CharArray 的定义都必须是可访问的。此外，每一个 Word 的客户程序要想通过编译也都需要 String 和 CharArray 的定义。实质使用了一个类型的私有派生和内联函数拥有这些同样强烈的物理隐

① 从逻辑设计角度来看，这种结构的继承方式是潜在不可靠的。假设 Word 的语义要求它保存由 String 基类支持的任意数据的一个适当子集（例如非空格、标点或控制符），如果使用了公共继承，就没有办法阻止基类功能被客户使用，从而违反了这个要求（见 **meyers**，Item37，130~132 页）。

含。在所有这些案例中，我们都应该把需要的#include 指令嵌入组件的.h 文件中。

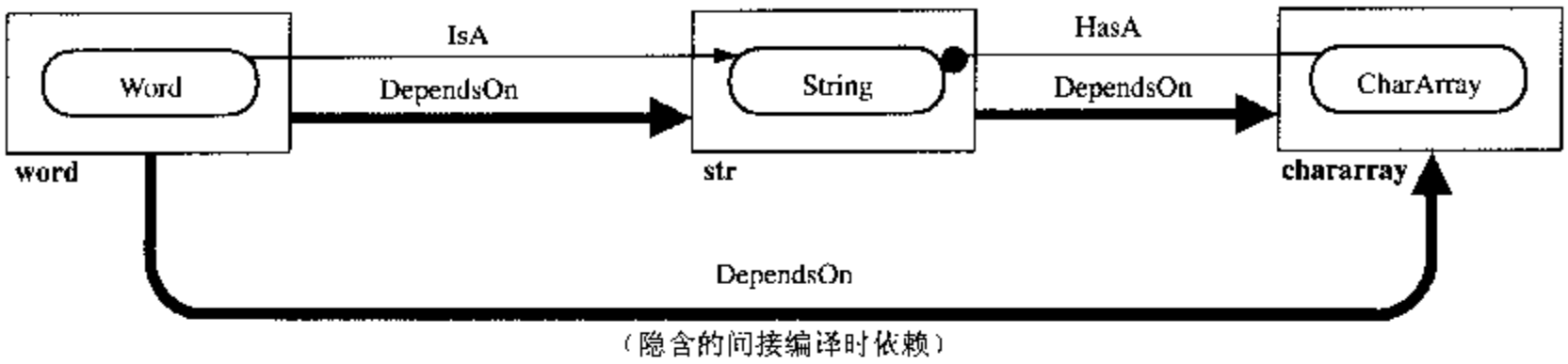


图 3-15　逻辑的 IsA 和 HasA 关系隐含组件依赖

如果一个类 HoldsA 一个类型（就是说如果它指向或引用那个类型作为一个数据成员），或者这个类型被实质地用在一个非内联函数体中，那么就不一定隐含这种强烈的物理耦合。这种用法是不正当的，因为它迫使组件的用户程序在编译时依赖它的实现类型，正如把#include 指令嵌入组件的文件头所导致的结果那样。在第 6 章中，这些微妙而重要的区别将被用来减少编译时耦合。

迄今为止我们一次只涉及了两个或三个类。现在我们要从一个给定的抽象逻辑表示，推断出一个稍微大些的组件集合之间的物理依赖。图 3-16 中的图描述了一个用于支持在线词汇表的小型子系统。

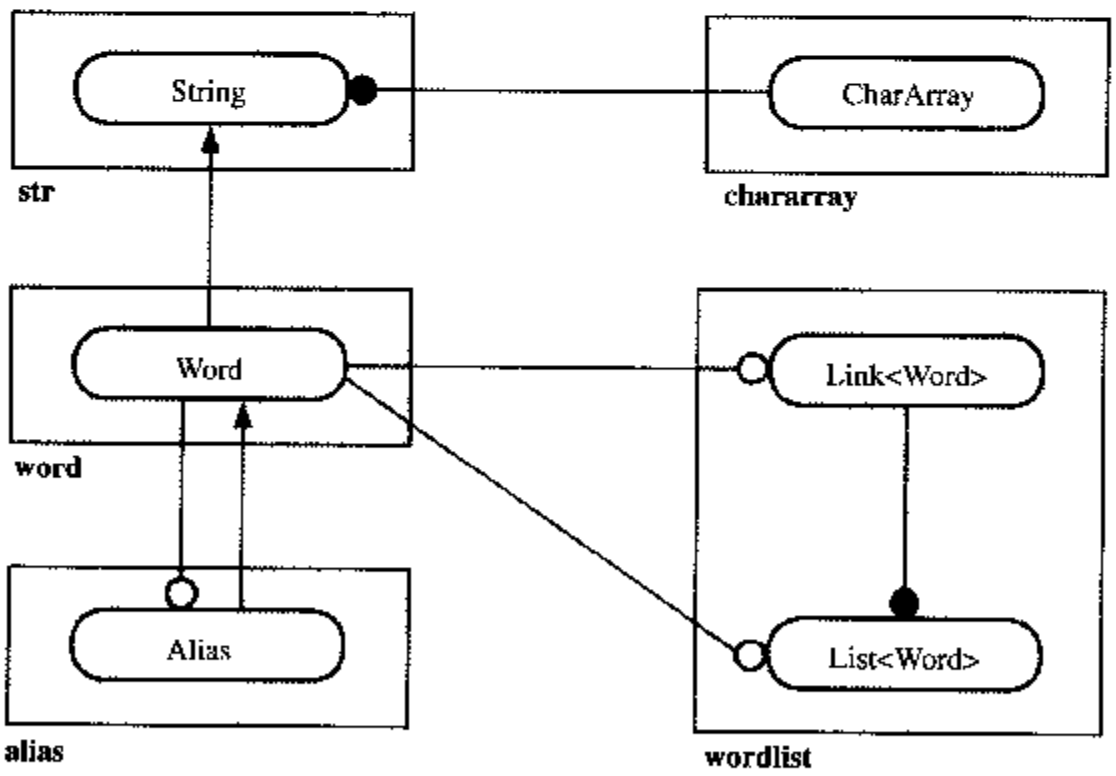
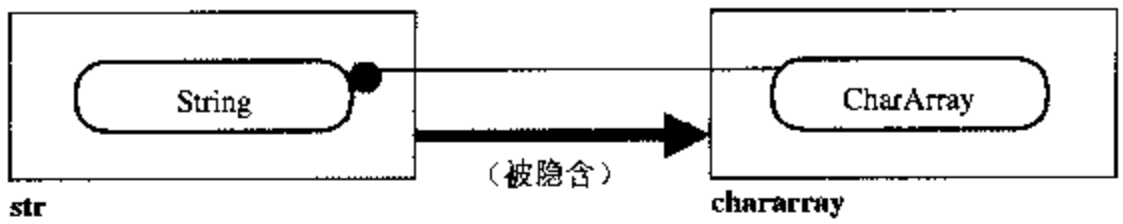
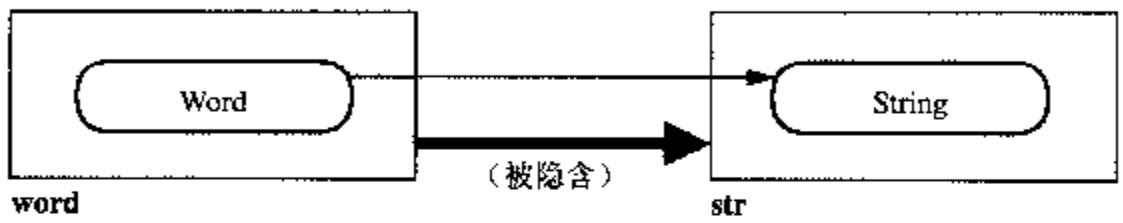


图 3-16　组件之间的逻辑关系

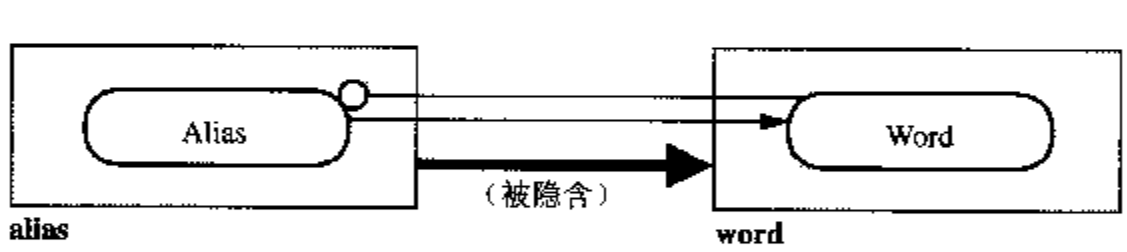
在图 3-16 的右上角，可以看到类 CharArray 在它自己的单独的组件中。类 String（在它的左边）在其实现中使用了类 CharArray，这样我们推断出组件 str 对组件 chararray 的一个可能的物理依赖：



在这个设计中，一个 Word 实现为一种 String（也许并不适当）。一个从类 Word 到类 String 的箭头被用来表示这种关系。我们也知道 IsA 关系总是隐含着定义组件之间的一个编译时物理依赖（和继承箭头相同的方向）。因此 word 明确地依赖 str。



正如我们可以从图 3-16 中看到的，Alias 不仅是一个（IsA）Word 而且在其接口中使用了（Uses）Word。但是要注意，Uses 关系的隐含依赖和表示 IsA 关系的箭头指向同一方向（从 Alias 到 Word）。因此，在 Word 和 Alias 之间没有隐含的循环依赖。所以有可能在一个没有包含或连接到 alias 的程序中使用 word。



现在考虑图 3-16 中的 wordlist 组件，它定义了两个假定的模板类 Link<Word>和 List<Word>。在组件 wordlist 内，我们可以看到 Link<Word>和 List<Word>之间有一个逻辑的“Uses-In-The-Implementation”关系。因为这些类已经定义在同一个组件内，它们之间的逻辑关系不会影响物理依赖。

Link<Word>和 List<Word>都在它们各自的接口中使用了 Word。这些逻辑关系的任何一个都足以使我们推断出一个可能的整个 wordlist 组件对 word 的物理依赖。请再次注意组件 word 可以存在于一个不包含或连接到 wordlist 的程序中，但反之则不然。

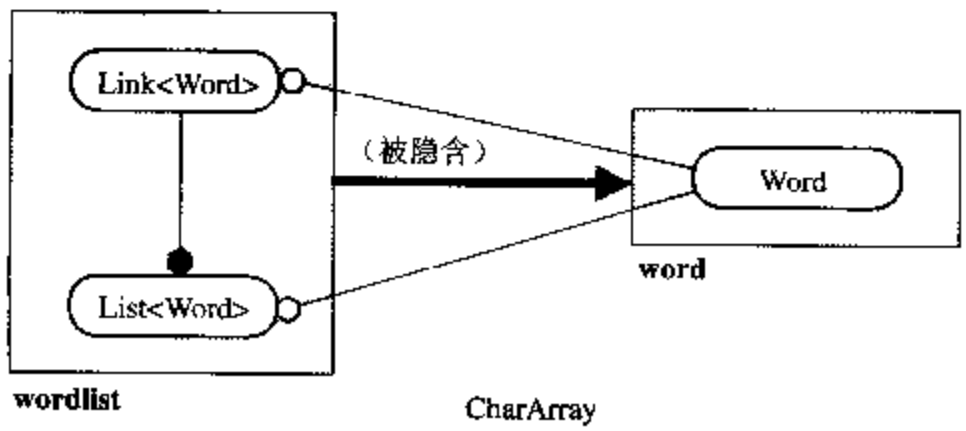


图 3-17 显示了从图 3-16 中的抽象逻辑关系推断物理依赖而产生的一个组件依赖图。实际上，这个图没有显示出完整的组件依赖关系。例如，它没有明确显示 word 对 chararray 或 wordlist 对 str 的间接依赖。如果我们把每个间接依赖都视同直接依赖那样画出来，就可以得到完整的依赖图。这样的图称为一个**传递闭包（transitive closure）**。

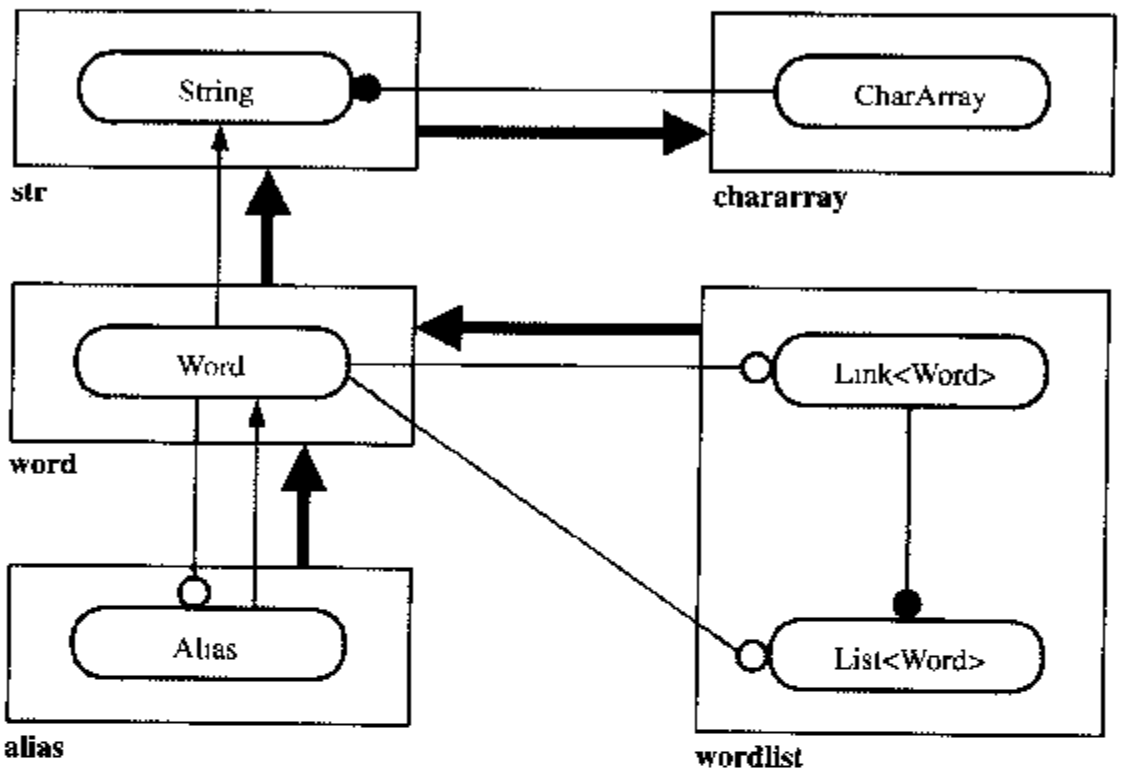
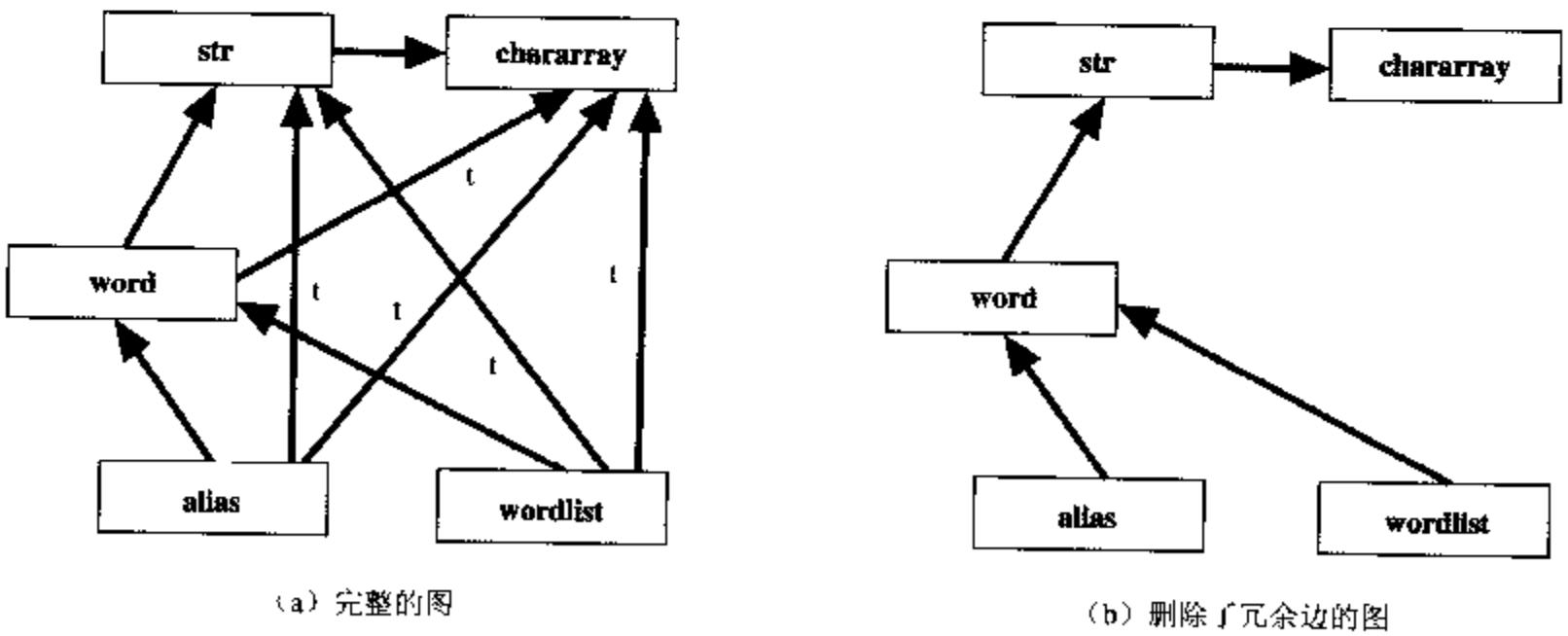


图 3-17　只显示直接依赖的组件图

图 3-17 的依赖图的传递闭包显示在图 3-18（a）中。在这张图中所有标识为 t 的边被称为**传递边（transitive edges）**，因为其他代表“直接”依赖的边隐含了它们的存在。删除这些冗余的传递边不会损失基本的信息，但是它们的存在可以减少混乱，使图更容易理解。



（a）完整的图　　（b）删除了冗余边的图

图 3-18　直接依赖图的传递闭包

从图 3-18（b）中很容易知道 word 间接依赖 chararray，而 wordlist 间接依赖 word。一般来说，当且仅当依赖图中有一条从 x 到 y 的路径时，组件 x 才依赖（DependsOn）组件 y。

扼要重述：逻辑关系隐含物理依赖。像 IsA 和 HasA 这样的逻辑实体之间的关系，在跨越组件边界实现时将总是隐含编译时依赖。像 HoldsA 和 Uses 这样的关系可能隐含跨越组件的连接时依赖。通过在设计阶段考虑隐含的依赖，我们可以远在编写任何代码之前就评价出我们的体系结构的物理质量。在第 4 章我们将讨论既可以提高易测性又可以促进重用的组件依赖的特性。在下一节中，我们将讨论如何更有效地从源代码中提取实际的物理依赖。

3.5 提取实际的依赖

现在假设我们根据隐含依赖设计一个大型的项目。在大部分设计工作完成之后，开发正在进行，我们很想有一种能够提取我们的组件之间的实际物理依赖的工具，以便我们可能跟踪实际的组件依赖，并且将它们和我们最初的设计期望进行比较。

虽然通过语法分析整个 C++程序的源代码来确定准确的组件依赖图是可能的，但这样做既困难又相对较慢。假如我们已经遵守了在 3.2 节介绍的设计规则，就可能通过只分析 C++预处理器#include 指令直接从组件的源文件中提取组件依赖图。这个过程相对较快，因为有许多标准的、公共领域的依赖分析工具（如 gmake、mkmf 和 cdep）可以做这个工作。

原 则

假如系统编译成功的话，仅凭由 C++预处理器#include 指令产生的包含图，就足以推断出系统内部的所有物理依赖。

想要了解为什么这个论断是正确的，考虑下面的推理。如果组件 x 直接实质地使用了组件 y，那么为了编译 x，编译器就必须查看 y.h 提供的定义。惟一能做到这一点的方法就是组件 x 直接或间接地包含 y.h。作为 3.2 节的设计规则的一个结果，任何这样的直接实质使用都等同于一个编译时依赖。

假如 x 通过了编译，则相反的说法（即如果 x 不包含 y.h，那么 x 就没有对 y 的编译时依赖）无疑也是正确的。

从另一个角度来说，组件 x 会合理地包含组件 y 的头文件的惟一的理由是：组件 x 事实上确实直接实质地使用了组件 y。否则包含物本身将是多余的，并且会引进不必要的编译时

耦合。

相反的说法（即如果x没有一个对y的本质的编译时依赖，那么x不包含y.h）应该总是正确的——但是有时候，由于人们的疏忽，也可能不正确。

指导方针

只有当组件x直接实质地使用了定义在y中的一个类或自由运算符函数时，x才应该包含y.h。

无论先前是否已经有一个编译时依赖存在，一个组件包含了另一个组件的文件头这个事实都会再强加一个编译时依赖。如果我们假设一个组件中的所有的#include指令都是必要的，那么将有一个连接时依赖伴随着编译时依赖（我们已经知道它具有传递性）。换句话说，“实质的使用”应该等同于“头文件包含”，实质的使用几乎总是隐含一种传递的物理依赖。

一个组件集合的#include图恰好相当精确地反映了组件间的依赖关系。如果我们把“x包含y.h（直接或间接）”解释为“x直接DependsOn y”，那么#include图展示的关系精确地反映了编译时物理组件依赖。

设计规则规定，所有的对一个组件的实质的使用都必须通过包含它的头文件（而不是通过局部的外部声明）来标志，以保证Include关系的传递闭包能指明组件之间所有实际的物理依赖。

这些提取的依赖有时候会因为太机械而犯错。以这种方式提取的依赖图可能会包括由不必要的#include指令（应该被删除）引起的额外的、假的依赖。但是如果所有的主要设计规则都遵守了，依赖图就决不会遗漏一个实际的组件依赖。

从一个潜在的大型组件集合快速而精确地提取实际物理依赖的能力，使我们在整个开发过程中都能够检验这些依赖是否与我们整体的体系结构计划相一致。附录C中描述了一种物理依赖提取/分析工具。

3.6 友元关系

我们现在偏离本章主题来讨论有关友元关系以及授权的友元关系如何影响一个类和一个组件的逻辑接口。友元关系和物理设计之间的交互程度强烈得令人惊讶。虽然表面上是一个逻辑设计问题，友元关系却会影响我们把逻辑结构收集进组件的方式。避免友元关系穿越组

件边界的愿望，甚至可能导致我们要重建我们的逻辑设计。我们将在整本书中经常提到本节中介绍的内容。

指导方针

避免把（远距离的）友元关系授权给定义在另一个组件中的一个逻辑实体。

根据《Annotated C++ Reference Manual》（注释 C++参考手册），“友元是一个类的接口中的和一个成员差不多的部分[①]”。在作这个论断时，有一个隐含的假设，即友元不可分离地与一个授予它友元关系的对象结合在一起。

从纯粹的逻辑角度来看，如果一个类作了一个友元关系的声明，那么，按照封装的定义（2.2 节），该声明就不是这个类的一个封装细节。任何人只要定义了一个函数并且其声明严格与一个类中的一个友元声明相匹配，那么假如没有与定义在同一程序内的友元声明相匹配的其他函数，他就可以获得编程访问那个类的私有成员的权利。在那种非常严格的意义上，友元声明本身是类接口的一部分——实际的函数定义却不是。

原　则

一个组件内部的友元关系是该组件的一个实现细节。

通过把组件（而不是类）视为设计的基本单位，我们得出了一种完全不同的观点。只要友元关系被局部授权（即只要它只授权给定义在相同组件内的逻辑实体），那么，这些友元事实上就与授权友元的对象不可分离地结合在一起了。

定　义：若通过一个组件的逻辑接口不能通过程序访问或探测到该组件包含的一个实现细节（类型、数据或函数），则称这些实现细节被该组件**封装**了。

根据该定义，一定可以通过程序判定什么在逻辑（公共）接口中。请思考定义在组件 stack 中的等式运算符：

```
int operator==(const Stack&, const Stack&);
```

如果这个运算符突然被声明为类 Stack 的一个 friend（友元），那么按照上面的说法，把运算符本身放在 Stack 的（公共）接口中，应该能通过程序发现这个变化——对吧？但是，假如运算符是定义在该组件内的，则授予运算符友元状态对该组件的逻辑接口绝对没有影响。事实上，从任何客户的角度来看，operator==是不是类 Stack 的友元是被这个组件所封装的一个实现细节！

为了进一步说明这一点，请简单地思考一个 String 类，它（在其他东西中间）定义了成

① **ellis**，11.4 节，248 页。

员 operator+=来实现了自加功能。

```
String {
    //...
  public:
    //...
    String(const String& string);                    // copy constructor
    //...
    String& operator+=(const String& rhs);           // concatenate to me
};
```

现在我们可以选择在同一组件内实现非破坏性的连接（+），而不必使运算符成为友元：

```
String operator+(const String& lhs, const String& rhs)
{
    return String(lhs) += rhs;
}
```

如果分析之后我们发现有必要改善operator+的性能，我们可以通过声明operator+为 String 的一个友元来扩展类 String 的封装：

```
String {
    //...
    friend String operator+(const String&, const String&);
  public:
    //...
    String(const String& string);                    // copy constructor
    //...
    String& operator+=(const String& rhs);           // concatenate to me
};
```

声明 operator+为 String 的友元，可以得到一个更有效的实现，却会潜在地增加维护开销，但不会影响组件的逻辑接口：

```
String operator+(const String& lhs, const String& rhs)
{
    // clever, more efficient implementation using private members
}
```

没有简单的编程方法可以让组件的客户知道，一个给定的在组件内定义的逻辑实体是否被同一组件内定义的一个类声明为友元[①]。

原　则

为定义在同一个组件内的类授予（局部的）友元关系不会破坏封装。

① C++语言不区分类中的友元声明所处的位置。但是，将声明放在类的一个私有区域，反映了该组件的有关局部友元关系的语义。

授予局部的友元关系不会威胁到暴露一个对象的私有细节给未经授权的用户。因为被声明为友元的类被定义（局部地）在同一组件的头文件内，任何试图使用该对象的授权友元关系的人都将拥有信任他们的所有友元类的有效的定义。重定义这些友元类的任何企图都将被编译器制止，它会迅速发出错误警告：

```
MULTIPALLY DEFINED CLASS.
```

原　则

在组件内定义一个容器类的同时定义一个迭代器类，可以在保持封装的同时，使组件具有用户可扩展性、改进可维护性和加强重用性。

在单个组件内，应该在组件（作为一个整体）必需获得适当封装的地方局部地授予友元关系。当使用容器/迭代器模式时，我们总是希望让逻辑实体一直能够访问我们的实现，该实现在物理上紧密反映高度的逻辑内聚性。在一个组件之外对逻辑实体授予友元关系，在本书中称为远距离友元关系（long-distance friendship），则是完全不同的另一回事。

原　则

对一个定义在系统的单独物理部分的逻辑实体，授权（远距离）友元关系，会破坏授予该友元关系的那个类的封装。

对系统的另一个物理片段授予私有访问权，会导致封装存在漏洞，很可能会被通过插入一个假冒的组件来获得访问而滥用。例如，假设 3.1 节的 StackIter 类定义在组件 stackiter 中，与类 Stack 分离。于是没有什么可以阻止一个 stack 组件的用户定义一个定制的 StackIter 来替代他或她自己的组件，从而获得对 Stack 类的私有访问权。一旦这种事情发生，授权远距离友元关系的类就不能保护它的私有成员不受访问——它的封装已经遭到了破坏。

除了是封装的破坏者，远距离友元关系也是糟糕结构设计的一个症状。让物理上独立的逻辑实体相互紧密依赖，极易降低可维护性。特别是远距离友元关系允许对私有实现细节进行局部修改，影响系统中物理上较远的部分，因而会降低模块化。

甚至局部友元关系的过度使用也会影响可维护性。授权友元关系会扩大一个类本身的“接口”。有权访问对象实现的封装细节的函数越多，当修改实现时需要回访（可能需要重做）的代码也就越多。直接访问私有信息的代码行越少，就越容易试验可选择的实现。

3.6.1 远距离友元关系与隐含依赖

尽管我们不鼓励使用远距离友元关系，但是在一个组件外授予友元关系的可能性却导致我们必须明确，在决定涉及某个类的 Uses 关系时，是否应该考虑与该类的一个友元声明相匹配的函数。对这个问题的回答是重要的，但仅限于在友元关系超出了单个组件并且要基于 Uses

关系来推断出物理依赖的情况下。

原　则

友元关系影响访问特权但不隐含依赖。

一个类是一个不可分割的逻辑单位。一个自由函数是一个不同的逻辑单位。自由函数是否是一个类的友元决不会影响系统中的任何隐含物理依赖。

考虑定义在其自己的组件 barop 中的自由运算符函数

```
// barop.h
class Bar;
int operator==(const Bar&, const Bar&);
```

图 3-19 显示了这个运算符，以及类 Stack 和类 StackIter（现在也显示在单独的组件中）。这个自由运算符既不是 Stack 的成员也不是它的友元，因此它显然不会扩大类 Stack 的接口。但是当我们把这个运算符声明为 Stack 的友元时会确切地改变什么呢？

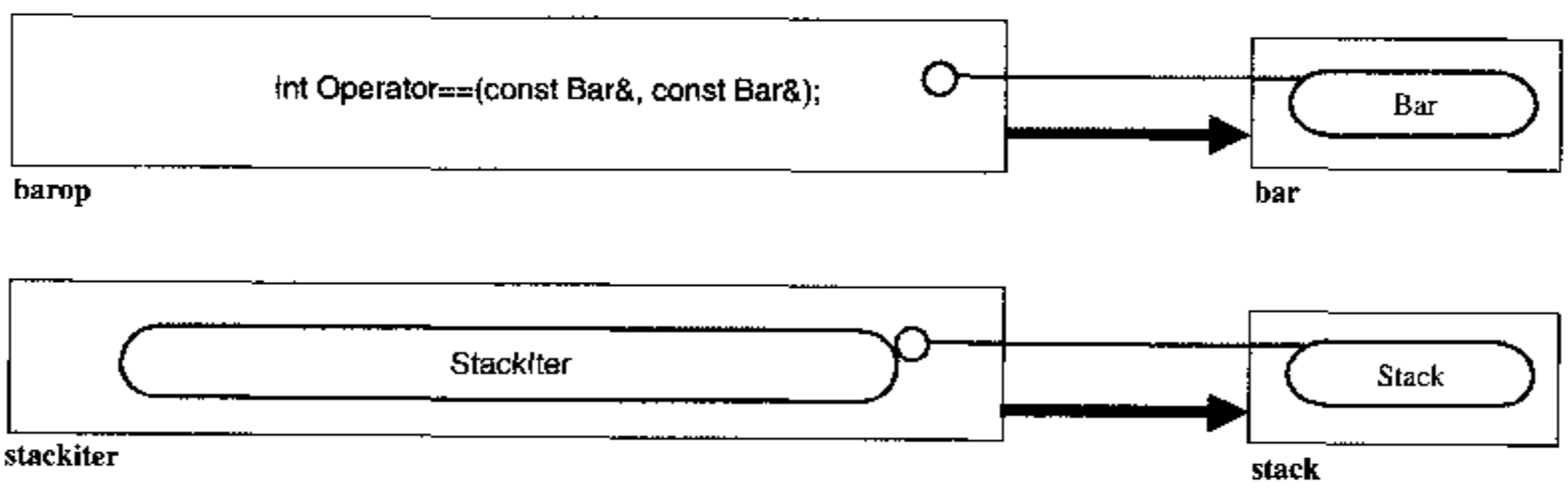


图 3-19　在不同的组件中的相关的逻辑实体

现在 operator==(const Bar&, const Bar&)会被认为是类 Stack 的接口的一部分吗？如果会，那么 Stack 在它的接口中使用了 Bar，并且有一个错误的、组件 stack 对组件 bar 的隐含依赖，如图 3-20 所示。

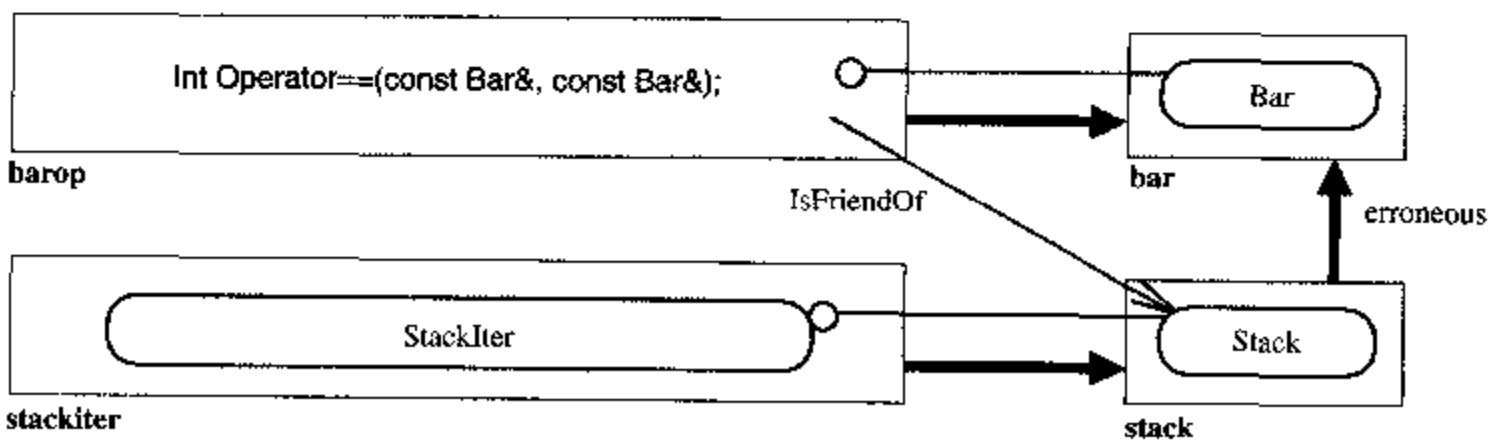


图 3-20　错误推断的 stack 对 bar 的依赖

只授权运算符友元关系不会使得 stack 突然物理依赖于 barop。使用一个类型会隐含一个对它的**所有**成员的依赖，但**不**一定会隐含对它的**任何**友元的依赖。特别地，operator==(const Bar&, const Bar&)是 Stack 的一个友元，StackIter 使用了 Stack，但这决不意味着 StackIter 直接或间接地使用了 operator==(const Bar&, const Bar&)。

注意图 3-20 中用于指示 IsFriendOf 关系的箭头的方向。这个箭头表示 operator==(const Bar&, const Bar&)现在被允许以一种比以前更紧密的方式依赖 Stack，但它并不保证任何实际的依赖。无论什么在箭头的相反方向都不会有物理依赖——把 operator==(const Bar&, const Bar&)看成是 Stack 的逻辑接口的一部分时，就会隐含上述观点。总而言之，授权友元关系改变的只是访问特权而不是物理依赖。

下面，通过一对用于比较 Stack 类型和 Foo 类型（对称地）的对象的自由运算符来举例说明这条原理的重要性：

```
int operator==(const Stack&, const Foo&)
int operator==(const Foo&, const Stack&)
```

我们不必浏览 Stack 或 Foo 的头文件就可以知道，这些运算符不是成员，因而不是这两个类的逻辑接口的一部分。因为这些运算符不是类的一部分，我们可以把它们定义在一个完全独立的组件中，从而使其可以被客户程序只在需要的时候包含。若不考虑访问特权，Uses-In-The Interface 关系指向一个方向，即从运算符到类，如图 3-21 所示。

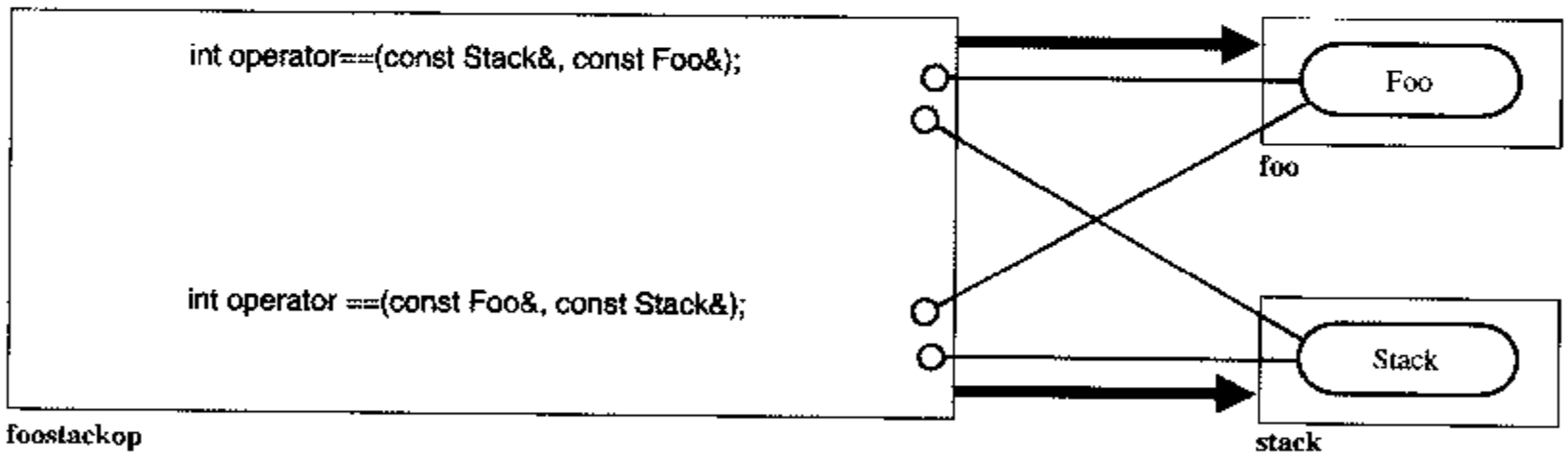


图 3-21　自由运算符对类的非循环依赖

现在考虑这个非常可疑的决定：改为加上以下两个 operator==**成员**函数：

```
int Stack::operator==(const Foo&   rhs) const;
int   Foo::operator==(const Stack& rhs) const;
```

作为成员，这些运算符函数显然是它们各自类的接口的一部分。每一个成员运算符都在它的接口中使用了另一个类。这些运算符的存在，在 Foo 和 Stack 之间引入了一个不合乎需要的、循环的 Uses-In-The-Interface 关系，如图 3-22 所示。当这些运算符是自由的并定义在一个独立的组件中时，不会导致这样的循环依赖。加入自由（运算符）函数绝对不会影响任何

类的逻辑接口（不考虑访问权），因为自由运算符（不像成员）不是任何类的固有的部分。（注意：使 operator==成为一个成员，按照纯粹的逻辑设计考虑，是一个糟糕的决定，正如将在 9.1.2 节中所讨论的那样）。

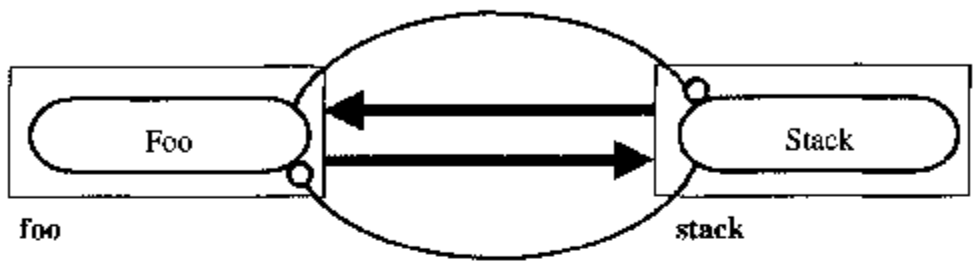


图 3-22　成员运算符对类的循环依赖

虽然授权友元关系在本质上绝不会直接影响隐含依赖，但是友元关系可能间接影响物理耦合。在试图避免这些与远距离友元关系有关的问题时，我们可能发现自己正在把若干紧密依赖的逻辑实体聚合进一个单个的组件内，从而在物理上使它们建立了耦合关系（见 5.8 节）。

3.6.2　友元关系与“骗局”

对于大型系统（可能跨越若干管理层次以及若干地理位置）来说，保护某一实现不被未经授权地使用是很重要的。在这种情形下，只是说“我留了一个漏洞，但请不要进来！”是没有用的。已经在源代码级访问了你的代码的人（尤其是顾客）将做一些他们需要做的事情来使他们的程序运转。如果使用你的一个私有数据成员能够解决他们的问题，即使只有一半的机会他们也可能使用它。如果用户能够直接访问你的实现，那么在你将来试图改进实现的时候，你可能会遇到不必要的阻力。

一个不谨慎的开发者可以简单地通过局部地（在文件作用域）定义友元类来获得访问私有细节的权利。于是这个开发者可以经由内联函数来使用这些细节，内联函数没有外部连接，因而不会与合法的函数定义发生冲突（即使它们被连接进了程序）。同样的原因，把一个单独的非内联自由（运算符）函数声明为一个友元——即使是局部地——也不能免于经由内联置换的“骗局”。人们实际上在产品代码中已经这样做了。你已经被警告了！

图 3-23 描述了一个非常值得怀疑的实践：通过使用远距离友元关系来故意利用封装上遗留的漏洞。类 Jail 定义了一个私有成员 release()，并且视一个定义在 jail 组件之外的名为 JailKey 的类为友元。经授权的 JailKey 被定义在组件 jailkey 内，被连接进程序中。一个恶意的 visitor 组件声明了类 JailKey 的一个局部版本，整个隐藏在 visitor.c 文件内。因为这个 JailKey 的违法版本没有带外部连接的成员，它可以静静地共存在一个文件中，依然利用 Jail 提供的友元关系。定义在 main()中的名为“bugsy”的 Visitor 对象的构造函数，创建了一个它自己的 JailKey 的一个实例，它在构造时调用 Jail 的私有的 release()方法。泄漏是不可避免的了。

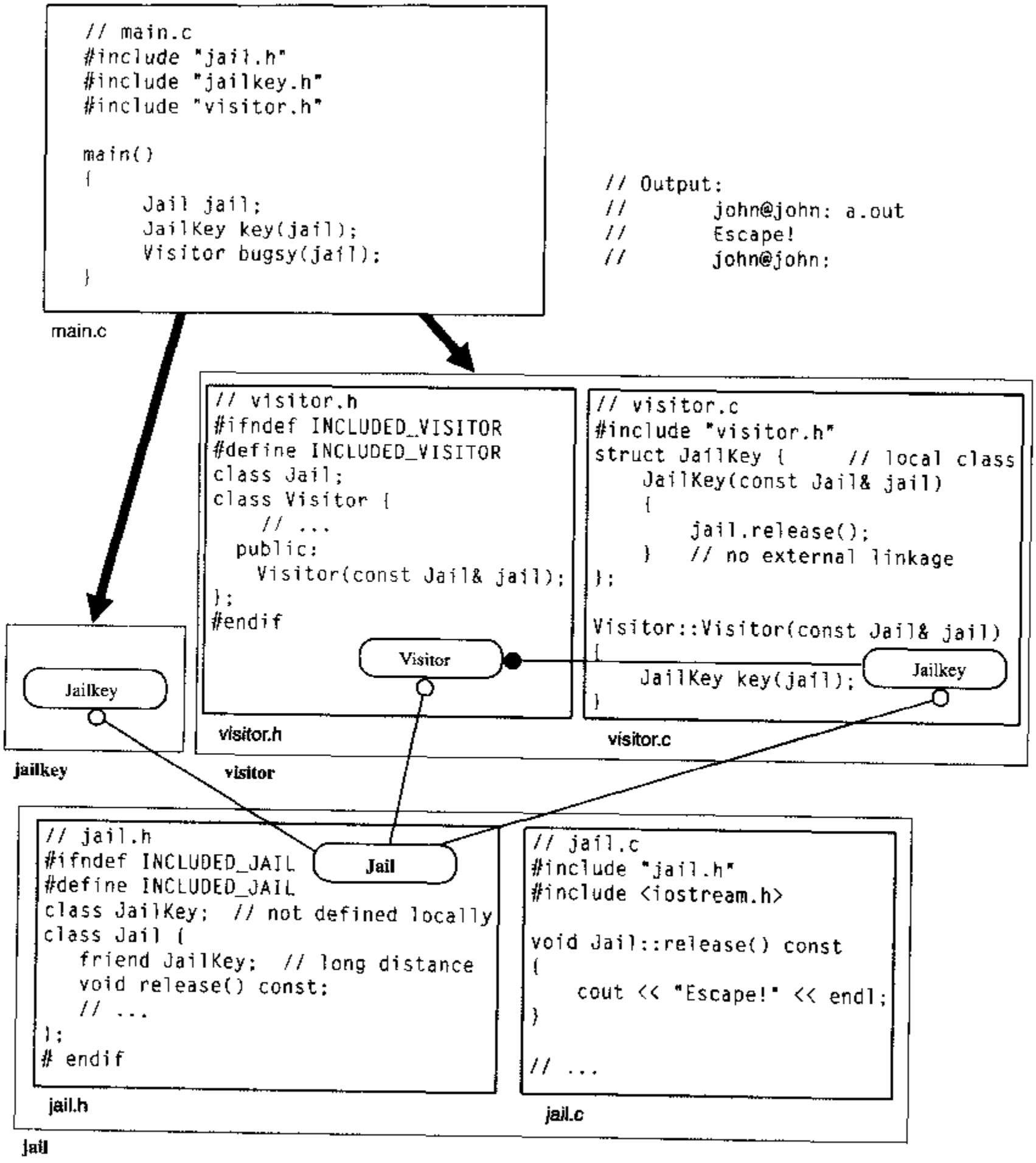


图 3-23 滥用友元关系的例子

令人悲哀的是，甚至还有更容易和更可恨的方法可以破坏封装：

```
// felon.c
#define private public        // capital offense
```

```
#include "jail.h"

void Felon::breakOut(Jail *jail)
{
    jail->release();
}

// ...
```

然而，像这样写头文件

```
// jail.h
#if !defined(INCLUDED_JAIL) && !defined(protected) && !defined(private)
#define INCLUDED_JAIL

class Jail {                    // maximum security
// ...

#endif
```

就可能走得太远了。

3.7 小结

开发可维护、易测试和可重用的软件需要全面的物理设计和逻辑设计的知识。物理设计研究组织的问题，超出了逻辑领域的范畴，物理设计很容易影响可测量的特性，例如运行时间、编译时间、连接时间以及可执行文件大小。

一个组件是由一个.c文件和一个.h文件组成的物理实体，它具体表达了一个逻辑抽象的具体实现。一个组件一般包含一个、两个甚至多个类，以及需要用来支持全部抽象的适当的自由运算符。一个组件（而不是一个类）是逻辑设计和物理设计的适当单位，因为它能够

（1）让若干逻辑实体把一个单一的抽象表现为一个内聚单位；

（2）考虑到物理问题和组织问题；

（3）在其他程序中选择性地重用编译单元。

一个组件的逻辑接口仅限于指能够被客户程序通过编程访问的部分，而物理接口则包括它的整个头文件。如果在一个组件的物理接口中使用了一个用户自定义类型T，即使T是一个封装的逻辑细节，也可能迫使那个组件的客户程序在编译时依赖T的定义。

组件是自我包含的、内聚的和潜在可重用的设计单位。在一个组件内部声明的逻辑结构不应该定义在那个组件之外。一个组件的.c文件应该直接包含它的.h文件，以确保.h文件可基于自己进行语法分析。对于每一个需要的类型定义，都始终包含其头文件，而不是依赖一个头文件去包含另一个，这样，当一个组件允许一个#include指令从其头文件中被删除时不会出现问题。想要改进可用性、可重用性和可维护性，如果某个带有外部连接的结构没有在一

个组件的.h 文件中声明，那么我们应该避免把该结构放在这个组件的.c 文件中。同样的原因，我们应该避免使用局部声明去访问有外部连接的定义。

DependsOn 关系标识组件之间的物理（编译时或连接时）依赖。一个编译时依赖几乎总是隐含一个连接时依赖，而且组件间的 DependsOn 关系具有传递性。

我们可以从一个跨越组件边界的逻辑 IsA 或 HasA 关系推断出一个确定的编译时依赖。在这样的情形下，逻辑 HoldsA 和 Uses 关系隐含了一个可能的连接时依赖。通过利用抽象逻辑关系来推断我们的设计决策的物理衍生物（依赖关系），我们可以在编写任何代码之前预知和修正物理设计缺陷。

我们希望在整个开发过程中跟踪实际的物理依赖，以确保与我们初始的设计保持一致。分析一个大型 C++系统的所有源代码太耗费时间。但是，倘若我们已经遵守了本书中的主要设计规则，从组件的包含图就可推断出它们之间的所有物理依赖。附录 C 中提供了对这样一个工具的描述。

最后，友元关系虽然表面上是一个逻辑问题，却会影响物理设计。在一个组件内部，（局部的）友元关系是那个组件的一个封装的实现细节。为了改进可用性和用户扩展性，一个容器类常常会把同一组件内的一个迭代器视作友元，不会破坏封装。

因跨越了组件边界，（远距离）友元关系变成了一个组件的接口的一部分，并且会导致该组件的封装被破坏。远距离友元关系还会通过允许对一个系统的物理上较远的部分进行密切访问而进一步影响可维护性。

友元关系直接影响访问权限但不隐含依赖。但是，我们避免远距离友元关系的愿望会间接地迫使我们把紧密相关的逻辑实体打包进一个单个的组件内，从而使它们在物理上耦合。忽略这些物理考虑会导致客户去使用封装漏洞，这种封装漏洞可由所有的远距离友元关系引起，甚至可由局部的友元关系对单个的非内联自由（运算符）函数作用所致。

4

物理层次结构

由 DependsOn 关系所定义的组件之间的物理层次结构，与分层所隐含的逻辑层次结构很相似。对于有效理解、维护、测试和重用来说，避免循环物理依赖是中心问题。设计良好的接口应该是短小的、易于理解和易于使用的，但是，这种接口会使用户级的测试代价很大。

本章我们将探讨如何利用物理层次结构来简化“好”接口的有效测试。根据组件间的物理依赖关系，我们引入层次号的概念来帮助表达组件的特点。我们将使用一个复杂的例子来说明隔离测试、分层测试以及增量式测试的价值。最后，我们导出了一种客观度量方法，该方法可以用于量化一个任意子系统中的物理耦合程度。通过使物理设计质量这个概念更客观并且更具体后，这种测量方法就可以帮助我们评估各种设计选择方案的效果。

4.1 软件测试的一个比喻

当一个顾客对一辆小汽车进行测试驾驶时，她（或他）主要注意小汽车作为一个整体的的性能如何——小汽车的操纵、急转弯、刹车等方面怎么样。顾客对主观可用性也感兴趣——小汽车的外观是否漂亮，座位是否舒适，内饰是否豪华，总的来说就是拥有该车的满意度。一般的客户不会去测试安全气囊、球窝接头（ball-joints）或引擎安装，以了解这些部件在所有的条件下的运行是否和所期望的一样。当顾客从一家著名的生产商那儿购买一辆新的小汽车时，会想当然地相信这种重要的低层可靠性。

小汽车若要功能正常，重要的是汽车赖以工作的每个对象也都能很好地工作。顾客不会单独地测试小汽车的每一个部件——但有人会这么做。对小汽车进行质量检测不是顾客的责任。顾客为质量好的产品付款，而质量好的一个方面就是顾客对小汽车的正常工作性能满意。

在现实世界中，小汽车的每一个部件都被设计成带有定义良好的接口，并且在极限条件

下进行了隔离测试，以保证部件在被集成进一辆小汽车之前能满足指定的耐用期限。为了对汽车进行维护，机械师必须随时能够访问各种部件以便诊断和修理。

复杂的软件系统就像小汽车。所有低层次的部件都是具有良好定义接口的对象。每个部件或组件都可被隔离进行重点测试。通过分层技术，这些部件可以集成到一系列日益复杂的子系统中——对每个子系统都有一套测试，以保证这种增量式的集成能够正常地进行。这种分层体系结构使得测试工程师能访问在更低抽象层次上实现的功能，而不会将低层次的接口暴露给产品的客户。最终的产品也要进行测试，以确保产品能满足顾客的期望。

总的来说，一个设计良好的汽车是用分层的部件建造的，制造商对这些部件进行了彻底的测试：

（1）在隔离的条件下；

（2）在一系列部分集成的子系统中；

（3）作为一个完全集成的产品。

一旦装配完成，机械师很容易查看部件，以便进行正确的测试和维护。在软件中，这些概念本质上是一样的。

4.2 一个复杂的子系统

作为软件子系统的一个具体例子，考虑一个计算机辅助电子设计应用程序中的点对点路径问题。该子系统以一种相对简单的描述解决了一个相当复杂的问题。

问题声明

假定：

（1）一个封闭的区域（描述为一个简单的封闭的多边形）。

（2）在封闭区域中的一组障碍物或“洞”（每一个都被描述为一个封闭的多边形），这些洞不彼此重叠（但可以彼此邻接），也不与该封闭区域的周界线重叠（但可以邻接）。

（3）一个起点（表示为一个点）。

（4）一个终点（也表示为一个点）。

（5）宽度（表示为一个整数）。

确定在指定的起点和终点之间是否有一条指定宽度的直线路径存在。如果有，则（有选择地）返回最短直线路径（作为一个开放的多边形）的中心线。

图 4-1 说明了该点到点路径问题的一个实例[①]。这个封闭区域包含 3 个洞，有一条成功的路径可以到达但不重叠。起点由 *s* 表示，终点由 *e* 表示。指定宽度的多种可能的最短直线路径之一由中心线定义，在图中显示为 *s* 和 *e* 的连接。

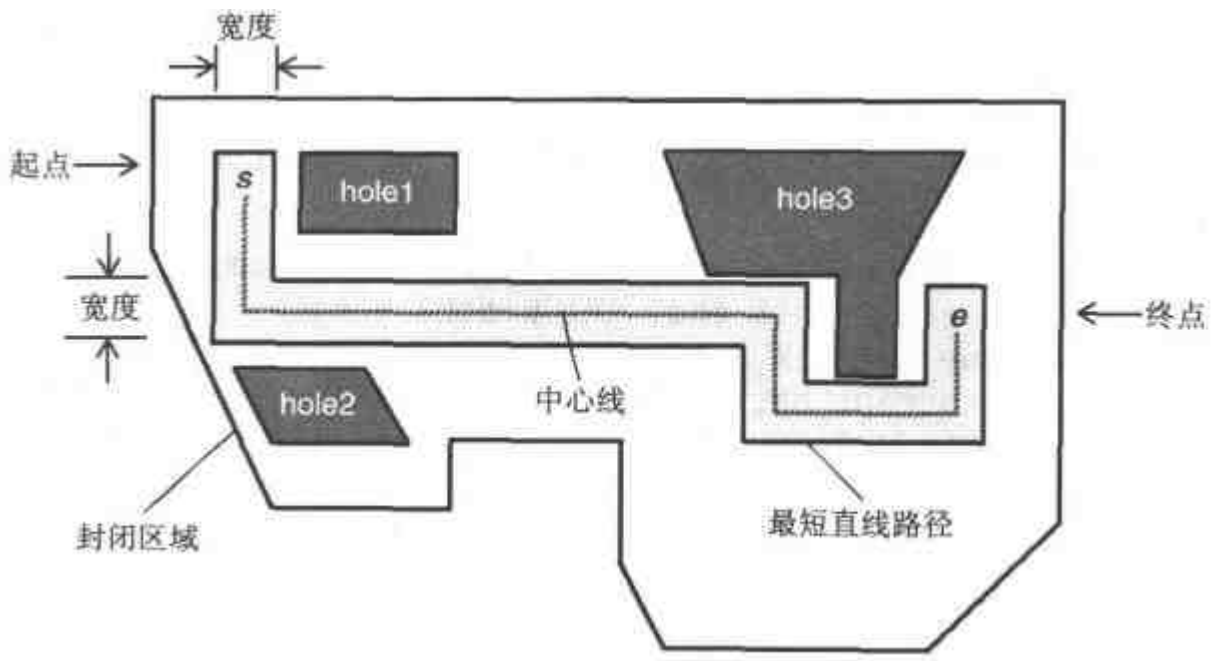


图 4-1 一个点到点路径问题的例子

解决这个复杂问题的一个组件的逻辑接口可能会令人误以为很简单。描述点到点路径子系统的客户程序接口的头文件 p2p_router.h 的全貌如图 4-2 所示。（注册的）类前缀 p2p_表示该组件属于 p2p 软件包，同时消除了属于不同包的类之间的标识符名称冲突的可能性（见 7.2 节）。

```
// p2p_router.h
#ifndef INCLUDED_P2P_ROUTER
#define INCLUDED_P2P_ROUTER

class geom_Point;
class geom_Polygon;
class p2p_RouterImp;

class p2p_Router {
    p2p_RouterImp *d_data_p;

    // NOT IMPLEMENTED
    p2p_Router(const p2p_Router&);
    p2p_Router& operator=(const p2p_Router&);

  public:
    // CREATORS
    p2p_Router(const geom_Polygon& enclosingRegion);
        // Create router for specified enclosing region.
```

① 我们提供了这个真实例子的所有细节。但想从后面的讨论中获益并没有必要理解这个例子的每一个方面，粗略的阅读就足够了。

```
        // The region must be a simple, closed polygon.
    ~p2p Router();

    // MANIPULATORS
    int addObstruction(const geom_Polygon& hole);
        // Add obstruction; obstruction must be a simple, closed polygon.
        // If obstruction overlaps another obstruction or the perimeter
        // of the enclosing shape, return non-zero with no effect and 0
        // otherwise.  Note: Regions are allowed to touch but not overlap.

    // ACCESSORS
    int findPath(geom_Polygon *returnValue, const geom_Point& start,
            const geom_Point& end, int width) const;
        // Determine whether a rectilinear path of specified width exists
        // in the current obstructed region between specified start and
        // end points.  Return 1 if such a path exists and 0 otherwise.
        // If a path exists and returnValue is not 0, store the center
        // line of any shortest path in (*returnValue).
};

#endif
```

图 4-2　p2p_Router 的完整的头文件

该点对点路径子系统的逻辑接口中使用了两个用户自定义类型。这些类型(geom_Polygon 和 geom_Point）是公共几何类型的软件包（geom）的一部分，该软件包在整个系统中广泛使用。为方便参考，图 4-3 粗略地显示了 geom_Point 和 geom_Polygon 各自的接口。

```
class geom_Point {
    // ...
  public:
    geom_Point(int x, int y);
    geom_Point(const geom_Point& point);
    ~geom_Point() {};
    geom_Point& operator=(const geom_Point& point);
    void setX(int x);
    void setY(int y);
    int x() const;
    int y() const;
};
```

```
class geom_Polygon {
    // ...
  public:
    geom_Polygon();
    geom_Polygon(const geom_Polygon& pgn);
    ~geom_Polygon() {};
    geom_Polygon& operator=(const geom_Polygon& pgn);
    void appendVertex(const geom_Point& point);
    // ...
    int numVertices() const;
    const geom_Point& vertex(int vertexIndex) const;
    // ...
};
```

图 4-3　类 geom_Point 和 geom_Polygon 的接口概貌

该子系统的一个实际实现包含 5000 行 C++源代码（不包括注释），但使用点对点路径组件却是非常容易的。图 4-4 完整地显示了一个简单明了的、运行图 4-1 中例子的驱动程序。注意，这种长的、线性风格的优点是其简单性。这是一个在开发和测试中实际使用的典型的驱动程序。

```
// p2p_router.t.c
#include "p2p_router.h"
#include "geom_polygon.h"
#include "geom_point.h"
#include <iostream.h>

main()
{
    geom_Polygon enclosingRegion;
    enclosingRegion.appendVertex(geom_Point(0, 1000));
    enclosingRegion.appendVertex(geom_Point(0, 600));
    enclosingRegion.appendVertex(geom_Point(700, -100));
    enclosingRegion.appendVertex(geom_Point(2100, -100));
    enclosingRegion.appendVertex(geom_Point(2100, 100));
    enclosingRegion.appendVertex(geom_Point(3000, 100));
    enclosingRegion.appendVertex(geom_Point(3000, -200));
    enclosingRegion.appendVertex(geom_Point(3200, -400));
    enclosingRegion.appendVertex(geom_Point(4500, -400));
    enclosingRegion.appendVertex(geom_Point(5000, 100));
    enclosingRegion.appendVertex(geom_Point(5000, 1000));
    enclosingRegion.appendVertex(geom_Point(0, 1000));

    geom_Polygon hole1;
    hole1.appendVertex(geom_Point(800, 900));
    hole1.appendVertex(geom_Point(800, 700));
    hole1.appendVertex(geom_Point(1400, 700));
    hole1.appendVertex(geom_Point(1400, 900));
    hole1.appendVertex(geom_Point(800, 900));

    geom_Polygon hole2;
    hole2.appendVertex(geom_Point(600, 300));
    hole2.appendVertex(geom_Point(800, 100));
    hole2.appendVertex(geom_Point(1600, 100));
    hole2.appendVertex(geom_Point(1400, 300));
    hole2.appendVertex(geom_Point(600, 300));

    geom_Polygon hole3;
    hole3.appendVertex(geom_Point(2600, 900));
    hole3.appendVertex(geom_Point(2900, 600));
    hole3.appendVertex(geom_Point(3800, 600));
    hole3.appendVertex(geom_Point(3800, 300));
    hole3.appendVertex(geom_Point(4200, 300));
    hole3.appendVertex(geom_Point(4200, 600));
    hole3.appendVertex(geom_Point(4500, 900));
```

```
    hole3.appendVertex(geom_Point(2600, 900));

    p2p_Router router(enclosingRegion);
    router.addObstruction(hole1);
    router.addObstruction(hole2);
    router.addObstruction(hole3);

    geom_Polygon centerLine;
    geom_Point start(400, 800), end(4600, 500);
    int width = 400;

    if (router.findPath(&centerLine, start, end, width)) {
        cout << centerLine << endl;
    }
    else {
        cout << "Could not find path." << endl;
    }
}

// Output:
// john@john a.out
// { (400, 800) (400, 500) (3400, 500) (3400, 200) (4600, 200) (4600, 500) }
// john@john
```

图 4-4 点对点路径问题的简明的驱动程序

4.3 测试“好”接口时的困难

面向对象技术的一种实际有效的应用是把极大的复杂性隐藏在一个小的、定义良好的、易于理解和易于使用的接口后面。但是，正是这种接口（如果被不成熟地实现）会导致开发出来的子系统测试起来极其困难。

例如，p2p_router 组件（图 4-2）只包含四个公共函数：

（1） 一个建立封闭区域的构造函数。

（2）一个析构函数。

（3） 一个在封闭区域内收集障碍物的函数。

（4） 一个决定区域内任意两点之间（不包含目前已收集的障碍物）指定宽度的最短直线路径的函数。

图 4-4 末尾的输出告诉我们，该组件产生了一个答案。现在让我们停一下，并把自己想像为一个委派给该项目的质量保证测试工程师。我们会如何去彻底测试这样一个接口呢？

首先考虑，一般来说对这类问题的实例会有许多等价的好的解决方案。检验一个解决方案是否是连接两点之间（在一个有障碍物的区域内）的给定宽度的最短直线路径，不是微不足道的事，但也可能并不需要付出特别的气力就能完成。要检验该问题的一个解决方案是否是最佳的，一般来说，与发现这个解决方案一样困难。

我们可以通过尝试几种测试实例并进行手工检查来检验输出结果。尽管费时，手工检查在开发过程中可能是很有效的。考虑当开发阶段结束、子系统进入维护/调试阶段时会发生什么。认为我们或者开发者愿意或能够手工检查每一个版本的每个子系统的输出是不现实的。

> **定义**：回归测试指的是这样的规程：运行一个程序（该程序被给定了一个有固定期望结果集合的特定输入），比较其结果，以便检验程序从一个版本升级到另一个版本时是否能够继续如所期望的那样运行。

一种通常用于帮助自动回归测试的方法是在系统顶层运行大量的测试用例并捕捉运行的结果，然后手工检查一遍这些结果以确定它们的准确性。在每个版本发布之前获得新的结果，并将新结果与原来的结果进行比较。可以这样推测，如果新的输出结果与旧的输出结果完全一样，则该子系统是正确的。

对于许多复杂的问题来说（包括这个问题），回归测试的一个明显缺陷是可能有多重正确的解决方案。尽管点对点路径子系统的每个组件都可以有完全可预知的特性，但规范中仍然有空间允许开发者改变 p2p_Router 的实现，这种实现对一个给定的输入会产生不同的（但同样好的）最终结果。

在一个小得多的规模上，考虑一个在某个集合上的简单迭代器的规范。一般来说对元素的出现顺序没有限制，集合里的每个元素只能出现一次。验证一个迭代器在隔离情况下工作正常并不困难。但是当迭代器嵌入到一个复杂子系统（如以 p2p_router 组件为首的子系统）的实现中时，有效测试迭代器可能很难。

尽管点对点路径系统在最优路径存在的情况下一定会产生一个最优的结果，但很多复杂问题在合理的时间内很难用最优的方案解决。在这些情况下，可以使用启发式（heuristic）方法来产生好的（但不一定是最优的）解决方案。启发式技术经常采用一种智能的 trial-and-error 策略的形式，它们本质上是不可预测的，要用实验来决定哪个启发式技术倾向于产生最优的解决方案。依赖于启发式方法的软件抵制高层的回归测试，因为在启发式技术中的任何改进都会使回归数据无效。

在高层的接口上测试复杂的、基于启发式技术的软件更加复杂，因为在较可预测的基础组件中的失败，可能不会引起整个子系统完全失败，但这些潜在的错误会悄悄地降低子系统输出结果的质量。因为不是总能验证结果的最优性，所以这种质量的降低可能很难探测到。

比基于启发式系统的伪随机行为[①]更糟的是，那些与使用异步通信的系统有关的完全不可预测的行为。这样的系统产生的结果一般来说是不可重复的。在这种情况下，高层回归测试实际上是无用的。

在设计中最小化“表面区域”（即，提供足够的但最小的接口）是优秀软件工程的基石。但是也有非常出人预料的事情：我们非常努力得到的接口却对传统的测试技术设置了可怕的

① 要想更多地了解伪随机功能，见 **plauger** 中的 rand()（第 13 章，337 页）。

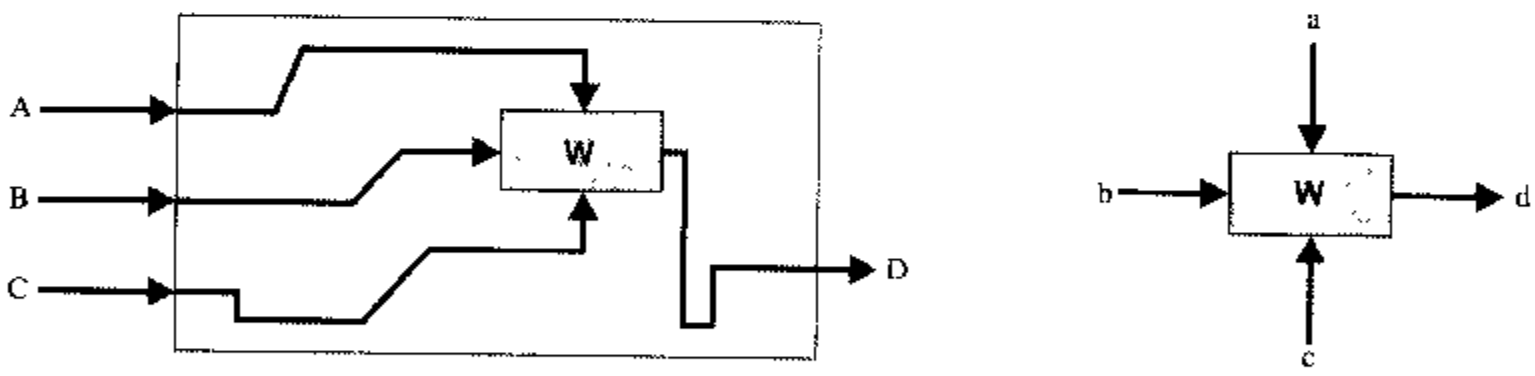
障碍。幸好我们有可以用来解决这些测试问题的技术。以下谚语用在这里很恰当：一盎司预防相当于一磅治疗。

4.4 易测试性设计

质量设计的一个主要部分是易测试性设计（Design For Testability，DFT）。DFT 的重要性在集成电路（IC）工业界是公认的。在许多情况下，只从外部管脚来测试 IC 芯片是不现实的，一些芯片有超过一百万个的晶体管。

当一个 IC 芯片制成之后，它就充当了一个“黑盒子”，可以只通过外部输入和输出（管脚）进行测试。图 4-5（a）描述了只使用提供给芯片本身的普通客户的接口来测试一个硬件子系统 w 的过程。为了测试 w，不仅有必要弄清楚 w 的一个好测试组合的组成，而且也要知道如何穿过芯片传送该测试组合给 w 的输入。如果不出错的话，w 产生的每个结果必须从 w 的输出传送到芯片本身的某个输出，以便观察和检验 w 的行为是否正确。要确保这种信息的传送，需要了解有关整个芯片的详细知识——即那些与 w 的功能毫无关系的知识。

IC 芯片的 DFT 有一种形式称为 SCAN，SCAN 使用那些只用于测试目的的额外的管脚和附加的内部电路来完成。使用这些特殊部件，测试工程师能够隔离芯片内部的各种子系统。在这样做的过程中，他们能够直接访问内部子系统的输入和输出并且直接测试它们的功能。换句话说，这种 DFT 方法试图授权测试者直接访问子系统，从而消除通过整个芯片传送信号的开销。使用这种方法，子系统的全部功能都可以有效地探测到，如图 4-5（b）所示（没有考虑在更大的系统中如何使用该子系统的细节）。



（a）从系统层次测试一个组件　　（b）直接测试一个组件

图 4-5　在集成电路中的易测试性设计

第一次应用 DFT，可以极大地提高质量。但是，IC 设计者们不喜欢这些额外的设计要求。这不仅需要额外的考虑，而且会使得它们的设计更大因而导致成本更高。许多设计者感到沮丧，认为这种严格的方法是对他们的创造力的侵犯。

目前 DFT 是 IC 工业的一种标准。没有哪个称职的硬件工程师会在考虑设计一个复杂芯片时不直接考虑易测试性问题。比较起来，大型软件系统的功能可能比在最复杂的集成电路中的功能复杂好几个数量级。令人吃惊的是，很多软件系统在设计过程中并没有适当的计划

来确保软件是可测试的。在过去的十年中，要求软件具备易测试性的努力经常遇到在 IC 工业界中遇到的同样的挫折。通常是人而不是技术本身对解决一个技术问题形成巨大的挑战。

原　则

对于测试来说，软件中的一个类类似于现实世界中的**实例**。

就像 IC 设计一样，面向对象软件涉及相对少量的类型的创建，然后，这些类型被重复地实例化以形成一个工作系统。例如，在许多软件系统中，一个 String 类是一个基本类型。在一个典型的系统调用中，可以创建该类的许多实例。

两个设计规范都要求对这些类型中的功能进行彻底的测试，以确保实例化时的行为正确。但是，软件设计不像 IC 设计那样要求对类型的每个实例都必须进行物理故障测试，软件对象对于这样的故障具有免疫力。如果一个类的实现是正确的，那么，由定义可知，该类型的所有的实例也都被正确地实现。

从测试的观点来看，每个软件类型都类似于现实世界的实例。如果直接对它进行操作，而不是试图将它作为一个更大系统的一部分来测试，那么测试一个 String 类的功能是最简单和最有效的。并且，和 IC 测试不同，我们自动地拥有对该软件子系统的接口（String 类）的直接访问权。

原　则

对整个设计的层次结构进行分布式测试，比只在最高层接口进行测试有效得多（就每美元测试费用而言）。

换种方式来说，假设我们只有 X 美元可花在测试上，那么如果我们把分布式测试遍及到整个系统，我们就可以获得比只从最终用户接口进行测试所能获得的更完全的覆盖。

让我们再次考虑图 4-2 中的 p2p_router 组件例子。即使假设整个行为都是可预测的，试图从最高层次来对该组件进行整体测试的效率也是很低的，尤其是在所给定的接口极小的情况下。这就像在 IC 测试中（见图 4-6）只通过两个管脚来测试一个有一百万个晶体管组成的微处理器一样！[①]

软件测试本质上比硬件测试要容易，因为在系统中产生的类的实例与同一个类在系统之外独立产生的类的实例**没有什么不同**。如果一个复杂的软件子系统真的类似于 IC 芯片，其实

① 其他种类的 IC 测试策略，例如集成自测试（Build-In Self Test，BIST），在芯片上放置了附加的电路，该电路激活后可校验芯片是否正常工作，而不必向接口传递特定的信息。BIST 有点类似于软件中使用的 assert 语句。添加公共的功能（例如 testMe()）后则更加相似。但是，我们的软件体系结构中的物理层次结构使得我们可以获得同样的结果，而不必在一个组件的接口中添加任何专门用于测试的功能。

现将整个地驻留在一个单一的物理组件中。如果在 p2p_router.h 中声明的功能全部在 p2p.router.c 中实现，那么我们将有可能被迫违反封装的原则，在公共接口上提供额外的功能——仅仅是为了能够进行有效的测试。

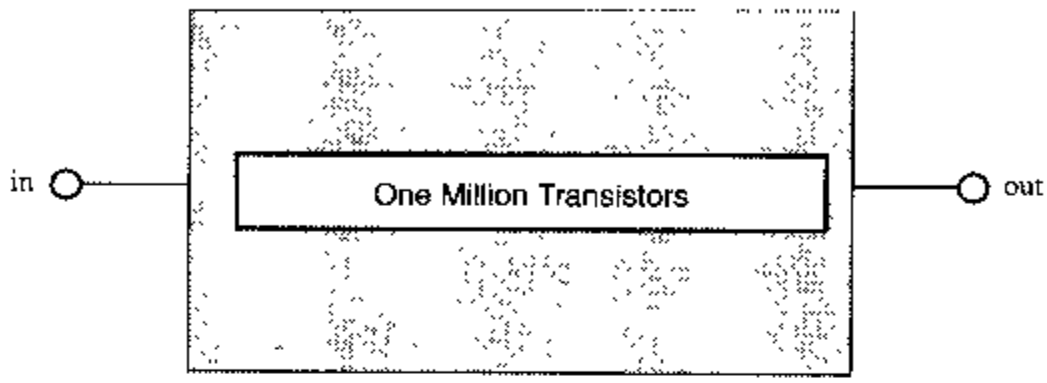


图 4-6　虚构的只有两个管脚的、高度抗拒测试的芯片

幸好，点对点路径的实现并非驻留在一个单一组件内。该实现被特意分散到整个组件物理层次结构中。即使一个 p2p_Router 对象的客户不能编程访问构成路径实现的分层对象，但测试工程师仍然可能识别有可预测行为的子组件，对这些子组件在隔离的情况下进行测试和检验会更有效。

4.5　隔离测试

在一个设计良好的模块化了系统中，可以对许多组件隔离测试。考虑一个涉及点对点路径了系统的非常真实的情况，该子系统最终将支持所有角度的几何形状。该系统目前暂时还处于原型阶段并且只能处理曼哈顿（90 度）角。该点对点路径系统是基于对象的，因此它在许多对象之上进行分层，这些对象目前大多数支持所有的角度。由于有些组件还没有升级为支持所有的角度，所以该 p2p_router 本身只能接受曼哈顿测试案例。

原　则

独立测试降低了一部分与软件集成相关的风险。

考虑如图 4-7 所示的 p2p_router 的物理体系结构。通过设计 p2p_router 使它的每个子系统都可以单独地开发和测试，我们可以确保它们升级后的每一个功能都是合适的，即使直到将来某一天它们才能通过完全的路径接口来检验。如果发生了程序方面的错误，可以并行地检测和修复这些错误。

一个可供选择的、不很严格但被广泛使用的软件集成的“方法”是，一直等到所有的软件都已经到位了再来试用它。这种方法通常称为大爆炸（big bang）方法。这个名字多少有些误导——预期的“巨响”经常只是轻微的嘶嘶声。

集成测试会检测出大多数规范中的错误。当集成系统不能按预期执行时，开发小组必须

尽快诊断出问题。他们不可避免地会发现很多代码错误，这些错误本质上与系统集成本身没有关系。独立测试至少能使这些代码错误在开发过程的更早期就被检测出来并进行修复。

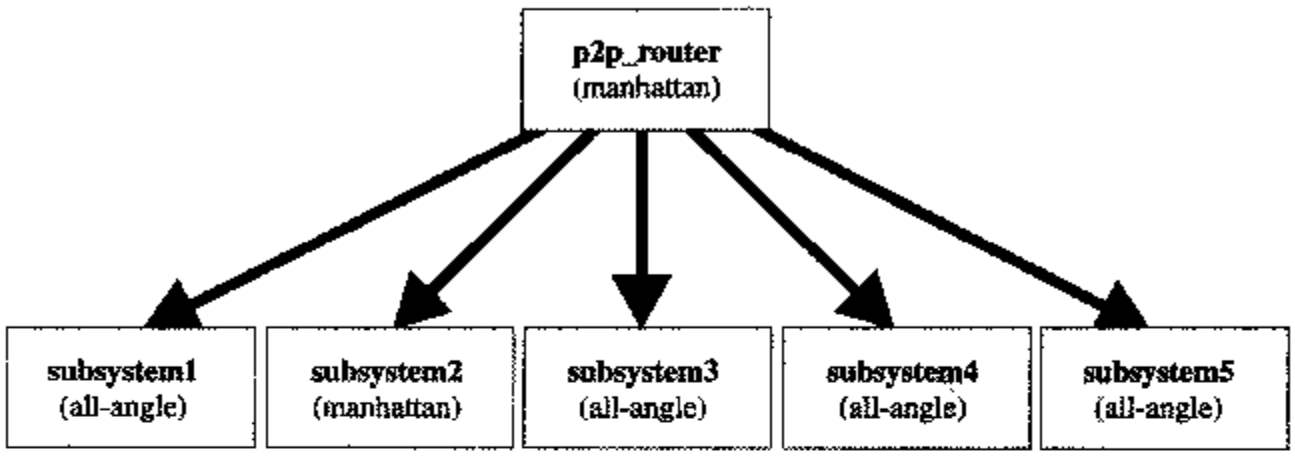


图 4-7　一个 p2p_router 实现的物理依赖关系

定义：隔离测试是指这样的规程：独立于系统的其他部分对单个组件或子系统进行的测试。

原　则

隔离测试组件是确保可靠性的有效办法。

在一个复杂系统的最底层，组件经常被深度优化，这增加了细微错误出现的可能性以及对详细的回归测试的需求。例如，仔细设计的特定对象内存管理经常可以使运行时性能提高一倍。但是，常规的内存管理程序很容易出错，并且这些错误最难检测和修复。在一个隔离的组件测试驱动程序中设置全局操作符 new 和 delete，可以确保内存管理方案在各种条件下正常工作，包括那些在实践中很少碰到的情况。

不是所有的程序都可使用可重用组件中的所有功能。例如，如果程序没有调用 Stack 类的 pop()成员函数，就没有办法只通过测试那个程序来测试 pop()。即使有一个特殊的程序调用了每一个函数，也可能有这样的状态：对象应该可以正确运转，但是环境软件不允许它完成。

考虑一个 String 类，该类被开发为一个解释器的一部分。该解释器从未见过长度为 0 的标识符，因此它决不会产生一个空 String 来表示该标识符。（通过为 string 组件特别设计的一个完全测试，这个边界条件肯定可以处理好。）随着系统的演化，我们可能在将来的某个时候在同一系统的其他部件中重用 String 类，但以新的方式（例如，拥有 String 变量）。此时一个空 String 的实例**可能**出现在本系统中。这种增强可能是在系统的相当高的层次上进行的，但是潜在的错误存在于最低的层次——在 String 类中——该类已在相当长的时期内工作得很“完美”！

在大型项目中，String 类的作者与有效地增强该类并暴露出问题的人可能不是同一个人。检测并修复这样的错误，暂且不提随后出现的失败，比从一开始就通过早期的组件级隔离测试来避免这样的错误代价要高得多。

对每个使用库程序（如，iostream）的系统都进行测试，以验证所需要的 iostream 功能是否工作正常，这是一种多余的、不必要的高昂代价。人们已经假定 iostream 如所预期的那样工作。对于大型项目，可能会有很多内部开发的应用程序库。没有一个单个的可执行程序会利用所有这些功能，但是所有这些程序都必须进行完全的隔离测试。

我们可以通过对组件本身的测试进行分组来避免冗余。在这样做的时候，我们扩展了面向对象设计的概念，作为一个单独的单元，它不仅包括组件，而且包括支持测试和支持文档。此外，对精心编写的组件级测试，可以通过给预期的用户提供一整套小但易于理解的例子来重用。现在可以在一个地方对每个组件提供的功能进行完全的测试，依赖于这些组件的客户可以合情合理地认为这些组件是可靠的。

隔离测试对于找出由增强引起的低层次问题是很理想的，并且对将一个系统移植到一个新的平台特别有用。这些低层测试可确保维持基本功能，并使测试人员很容易发现矛盾之处。偶尔也会有一些缺陷逃过了局部检测，而被更高层的测试捕捉到。应该及时更新低层次组件测试，以便在缺陷修复**之前**暴露错误行为。这样通过使组件的测试独立于任何特定的客户，既可以方便修复又可以保护模块性。

对于隔离测试有一个关键点，过了这个关键点，隔离测试的回报就会减少。例如，将一个简单的 List 对象的 Link 类定义放在一个不同的组件中，以便可以隔离测试它，这样做是荒谬的，原因有两个：

（1）一个 List 对象的正常的操作会完全使用 Link 的功能。

（2）附加的组件会不必要地增加系统的物理复杂性，使系统更难以理解和维护。

应该基于对开销/利益的分析客观地决定组件级隔离测试的这个点，而不是只凭某个开发者的好恶来测试。

4.6 非循环物理依赖

对于一个能被有效测试的设计来说，一定能够将其分解成复杂性可以控制的功能单元。组件是实现此目的的理想部件。考虑图 4-8 中描述的组件 c1、c2 和 c3 的头文件。注意我们已经在组件头文件 c2.h 和 c3.h 中声明了类 C1 但没有提供其定义，因为没有必要定义一个出值返回的类[1]来声明那个函数。更多有关内联函数的内容见 6.2.3 节。

读者可以观察到（见 3.4 节）c1 没有隐含依赖于任何其他的组件。类 C2 在其接口上使用类 C1。因此，组件 c2 很可能依赖于组件 c1，但是，我们希望它不依赖于 c3。类 C3 在其接口上使用 C1 和 C2，因此，c3 很可能依赖于 c2 和 c1。该系统中的隐含依赖形成了一个有向的非循环图（Directed Acyclic Graph，DAG），如图 4-9（a）所示。

① 对于通过值传递的用户自定义类型也一样，但是要参见 9.1.1 节。关于内联函数使用用户自定义类型的情况请见 6.2.3 节。

```
// c1.h
#ifndef INCLUDED_C1
#define INCLUDED_C1



class C1 {
    // ...
  public:
    C1 f();
};

#endif
```

C1.h

```
// c2.h
#ifndef INCLUDED_C2
#define INCLUDED_C2

class C1;


class C2 {
    // ...
  public:
    C1 g();
};

#endif
```

C2.h

```
// c3.h
#ifndef INCLUDED_C3
#define INCLUDED_C3

class C1;
class C2;

class C3 {
    // ...
  public:
    C1 h(const C2& arg);
};

#endif
```

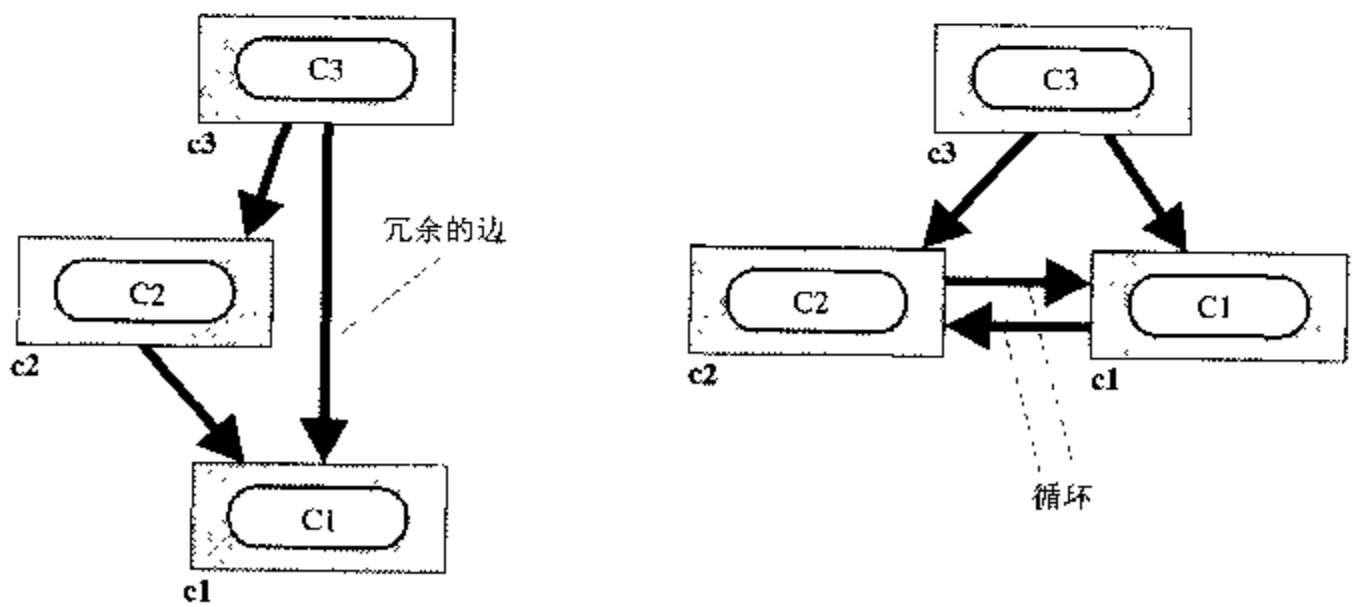
C3.h

图 4-8　带有非循环隐含依赖的组件

不含循环的组件依赖图非常有利于实现易测试性，但不是所有的组件依赖图都是非循环的。为了弄清楚原因，考虑这样的情况：如果我们将 C1::f 的返回类型从 C1 改变为 C2（如下所示），会产生什么结果？

```
class C1 {
    //...
  public:
    // C1 f();          // old
    C2 f();             // new
};
```

现在 C1 在其接口上使用 C2 并且（可能）依赖于它。这个修改后的系统的隐含组件依赖图现在有一个**物理**循环，如图 4-9（b）所示。



（a）非循环依赖　　（b）循环依赖

图 4-9　非循环的和循环的物理依赖

有非循环物理依赖的系统［如图 4-9（a）所示的这个］远远要比那些带有循环的系统更容易进行有效的测试。一个系统中的组件依赖只要是非循环的，就（至少）存在一个测试该系统的合理顺序。由于组件 c1 不依赖于任何别的组件，应该首先编写在隔离的条件下验证其功能的测试程序。接下来我们看到组件 c2 只依赖于组件 c1。因为我们能够为 c1 编写有效的测试程序，我们可以假设 c1 的功能是正常的，因此可以给对由 c2 添加的功能编写测试程序。我们不需要重新测试其中的 c1 部分，因为其功能已经被覆盖了。然后我们来看依赖于 c1 和 c2 的 c3。因为我们已经编写好了验证由 c1 和 c2 提供的功能的测试程序，所以我们只需要对付由 c3 实现的额外功能。

4.7 层次号

在本节中我们将介绍一种方法，将组件（基于它们的物理依赖）划分成等价的类（称为层次）。每个层次都与一个非负整数位标相联系，在本书中称为**层次号**。接下来的几个段落会描述层次号的起源以及它们如何被用于它们最初的上下文中。然后我们将这些精心建立的概念应用于一个新的上下文中——软件工程。

4.7.1 层次号的起源

层次号的概念是借用数字电路、门层次、零延迟电路仿真领域的概念[1]。在这里，一个 **gate**（门）实现一个低层次布尔功能块。每个门有两个或者更多的称为 **terminal**（接头）的连接点。一个 **circuit**（电路）由一个相互连接的门的集合构成。像一个门一样，一个电路也有输入接头和输出接头。初级输入（primary input）是到电路本身的输入。通过成对的称为 **wire**（电线）的接头连接器，这些输入与一些电路中的门的输入相连接。这些门的输出又通过电线与别的门的输入相连接，如此一直下去。一个有四个初级输入（a、b、c 和 d）的简单电路如图 4-10（a）所示。

仿真一个电路涉及用逻辑值设置其初级输入，然后依次求出每个（分层的）门的值。但是在一个特定的门的值能求出之前，我们必须证实其输入是有效的，方法是确保输入到这个特定门的所有门的值已经求出。

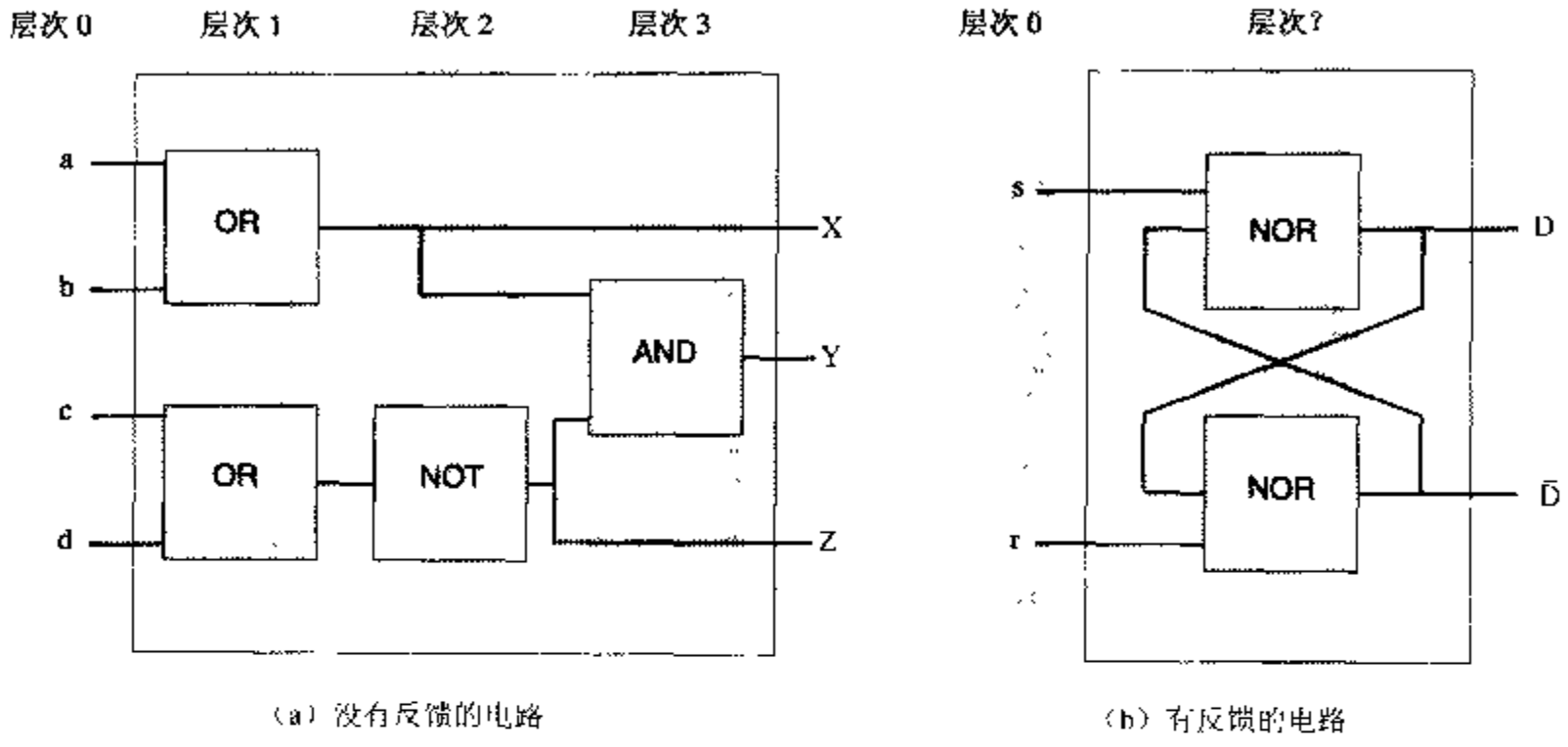
一个电路是一种图。这里，门和初级输入作为图的节点，而电线作为（有向）边[2]。在这个上下文中的层次号表示的是特定的门到初级输入的最长路径。初级输入被定义为第 0 层。

---

① 零延迟渐进法主要用在一种特殊的电路仿真器中（称为 fault simulator）。这种硬件和软件相似性的发现，部分源于作者在哥伦比亚大学与 Stephen H. Unger 教授所进行的博士学位的研究工作。

② 门本身影响边的方向，该方向反映了门对其输入源（例如，要么是一个初级输入，要么是电路中另一个门的输出）的依赖。

通过按照递增层次的顺序计算这些门的值，我们可以保证每个门的输入都是有效的。



（a）没有反馈的电路　　（b）有反馈的电路

图 4-10　有与没有反馈的逻辑电路

初级输入值是设定的，并不需要计算。在仿真期间，第 1 层的门只接受初级输入。这些门首先被计算，计算的顺序是任意的。接着计算第 2 层的门。由于第 2 层的门只接受一个或多个第 1 层的门（也可能接受初级输入）的输入，所以我们可以保证，在这个时候第 2 层的门的所有输入都已被求值。由于在第 N 层的门只依赖[0…*N*–1]层作它的输入，所以按照层次顺序计算门的值可以保证得到一个成功的仿真。

在图 4-10（a）中，一个在第 1 层上的 OR 门是 NOT 门的惟一输入，使得 NOT 门成为第 2 层的门。而 AND 门的输入包括第 1 层的 OR 门和第 2 层的 NOT 门。从 AND 门到初级输入的最长路径是 3（通过 NOT 门到初级输入 c 或 d）。AND 门属于最高层次（3），并且最后被求值。

原　则

每个有向非循环图都可被赋予惟一的层次号；一个有循环的图则不能。

注意，图 4-10（b）中一对交叉耦合的 NOR 门，从任一门到任一初级输入（r 或 s）的最长路径是无约束的。该电路不能被层次化——也就是说，它不能被赋予惟一的层次号。使一个电路可层次化的特性是它没有反馈。没有反馈使得电路在性质上更易于理解、开发、分析和测试。正是由于这些原因，反馈在大型系统中只是在非常严格的环境下才使用。出于完全相似的原因，“没有反馈”也正是我们希望软件设计所拥有的特性。

定　义：一个可被赋予惟一的层次号的物理依赖图称为是**可层次化的**（levelizable）。

4.7.2 在软件中使用层次号

让我们回到软件工程。如果软件系统中的组件依赖正好形成一个 DAG 图，那么我们可以给每个组件定义**层次**。

> **定 义**：
> 层次 0：一个在我们的软件包之外的组件。
> 层次 1：一个没有局部物理依赖的组件。
> 层次 *N*：一个组件，它在物理上依赖于层次 *N*–1 上（但不是更高层次上）的一个组件。

在这个定义中，我们假设所有在我们当前软件包之外的组件[①]（例如：iostream）都已经测试好了并且会正确地发挥作用。这些组件被看作“初级输入”并具有 0 层次号。一个没有局部物理依赖的组件被定义为层次 1。另外，一个组件被定义为比它所依赖的组件的最高层次高一个层次。

图 4-11 显示的是 3.4 节的图 3-17 的组件依赖图，该图恰好没有任何循环，因此是可层次化的。层次号显示在每个组件的右上角。组件 chararray 在本地并没有依赖任何其他组件，但是依赖于标准库组件（这种组件被设定为层次 0），因此 chararray 是层次 1。一个层次号为 1 的组件（如 chararray）如果只依赖于编译器提供的库，这样的组件就称为叶子组件。总是可以对叶子组件进行隔离测试。

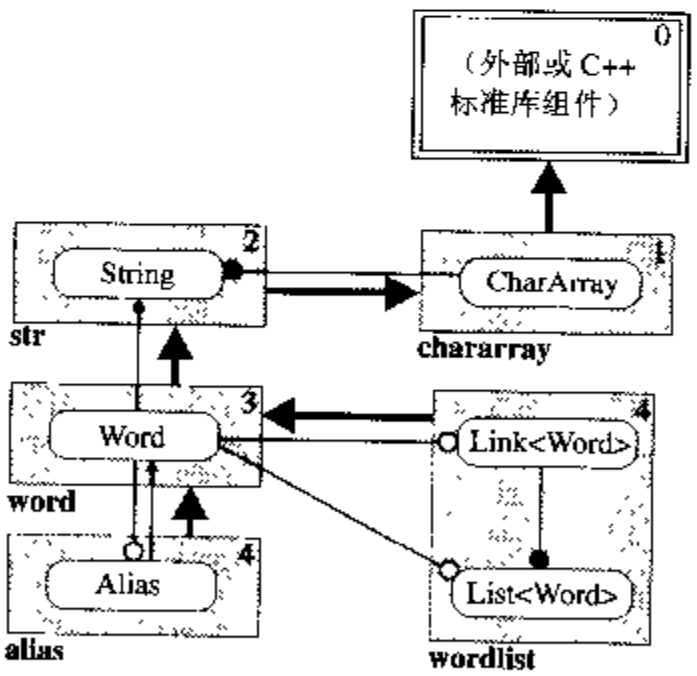


图 4-11 层次化的组件依赖图

① 假设现在 package 的意思是当前项目目录。

组件 str 只依赖于 chararray。str 的层次号为 2，比 chararray 的层次号多 1。组件 word 依赖 str（并且间接地依赖 chararray），由于 str 的层次号为 2，所以 word 的层次号为 3。因为 word 在第 3 层，而且 alias 直接依赖的惟一组件是 word，所以 alias 在第 4 层。wordlist 组件也直接依赖于 word 但不依赖于 alias，所以 wordlist 也是在第 4 层。

> **定 义**：一个组件的层次是一个最长路径的长度，该路径是指从那个组件穿过（局部）组件依赖图到外部（或由编译器提供的）库组件的集合（可能为空）的路径。

由于有了层次化的组件图，所以很容易指出系统中哪些组件是可隔离测试的。在图 4-11 中只有一个可独立测试的组件：chararray。通过从最底层开始（即层次 1）并在移到下一个更高层次之前测试当前层次的所有组件，我们可以保证当前组件依赖的所有组件都已经测试过了，在图 4-11 的例子中，我们可以最后测试 wordlist 或者 alias，但是其余的测试顺序则由层次号决定。

原 则

在大多数真实情况下，如果大型设计要被有效地测试，它们必须是可层次化的。

注意，层次化这一术语适用于物理实体而不是逻辑实体。虽然一个非循环的逻辑依赖图也可能隐含着有一个可测试的物理划分存在，但是（物理）组件的层次号以及我们的设计规则，隐含了有效测试的一个可行顺序。此外，图 4-11 标识出了哪些子系统可以被独立地重用。图 4-12（右栏）列出了每个组件重用时必须相应用到的其他一些组件。

为测试或重用	你还需要
$chararray_1$:	
$string_2$:	$chararray_1$
$word_3$:	$string_2$ $chararray_1$
$alias_4$:	$word_3$ $string_2$ $chararray_1$
$wordlist_4$:	$word_3$ $string_2$ $chararray_1$

图 4-12　可独立重用的子系统

可层次化设计的另一个显著优点是它们更容易被渐进地理解。理解一个可层次化设计的过程可以按照某种顺序（自上而下或自下而上）。不是所有的分层次设计的子系统都是可重用的。但是，如果要可维护，每个组件都必须有一个良好定义的接口，无论其适用性如何广泛，该接口都应该很容易理解。

当然，不是所有的设计都是可层次化的。有时一个设计是否可层次化不能从一个逻辑图中很明显地看出。考虑图 4-13，读者能否从该图中说出这个设计中的组件是否是可层次化的？

在这个设计中的逻辑关系并**没有**隐含任何组件中的循环物理依赖。事实上，我们的设计规则可以确保没有隐藏的物理依赖（例如，对外部全局变量的依赖）。图 4-14 显示了隐含的组件依赖以及最终得出的该设计的组件层次号。

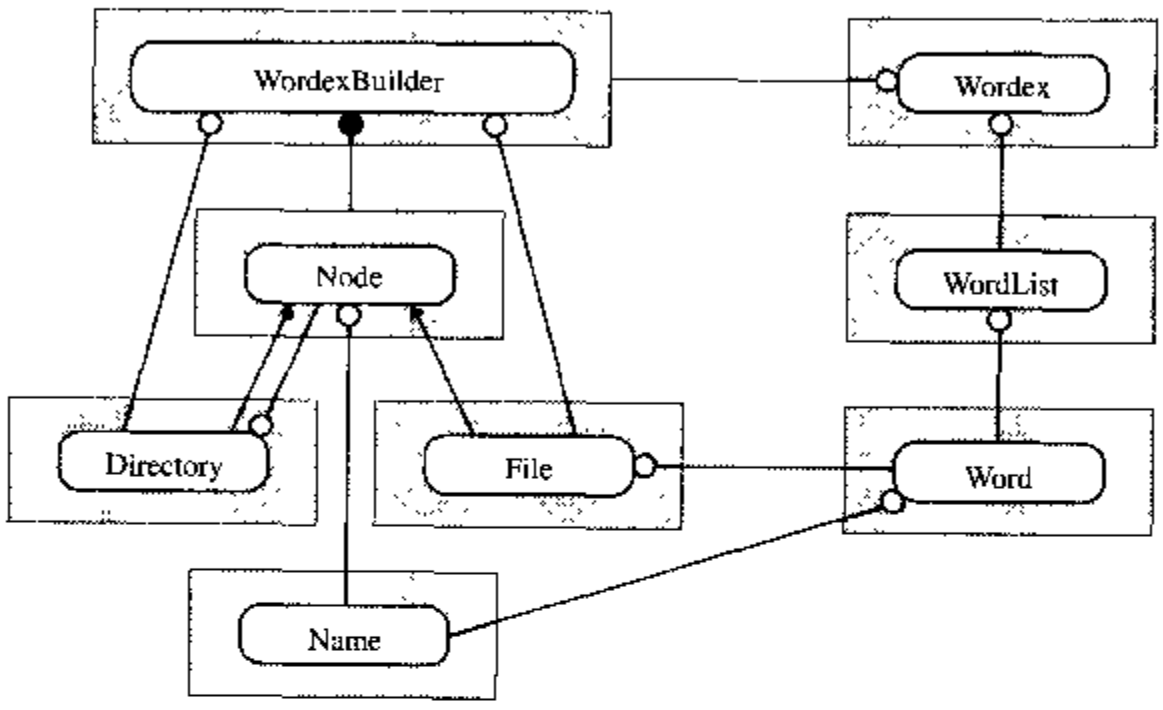


图 4-13　这个设计是可层次化的吗

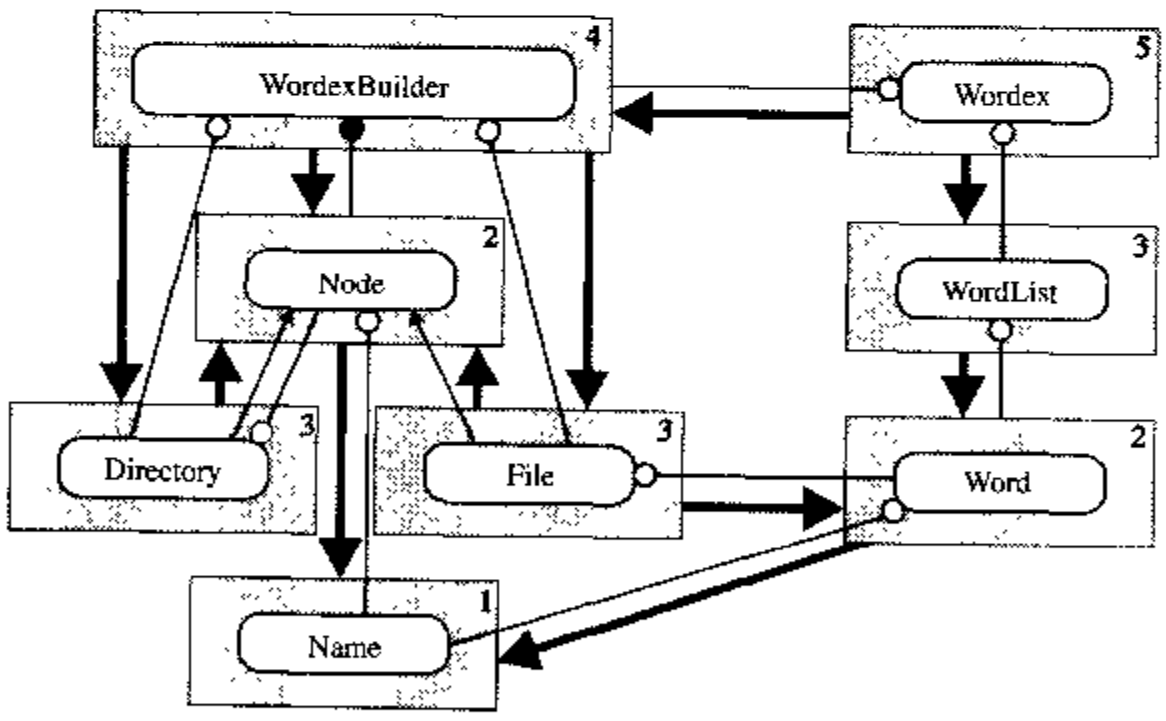


图 4-14　组件/类图

这个组件/类图是混乱的，并且包含的信息比理解系统的物理结构所需要的信息更多。如果我们重新安排组件的位置并且消除逻辑细节，我们可以得到非常清晰的组件依赖图，即图 4-15。

图 4-15 所示的依赖图中有一条冗余的边。组件 wordexbuilder 直接依赖组件 directory、file 和 node。正如我们在 3.3 节所介绍的那样，依赖关系具有传递性。因为 directory（和 file）依

赖 node，wordexbuilder 依赖于 node 的关系是隐含的，可以删除，不会影响层次号。图 4-15 所示的图很明显是非循环的，并且是典型的满足一个特定应用的子系统的图表。在这个抽象层次上，设计看来是合理的。

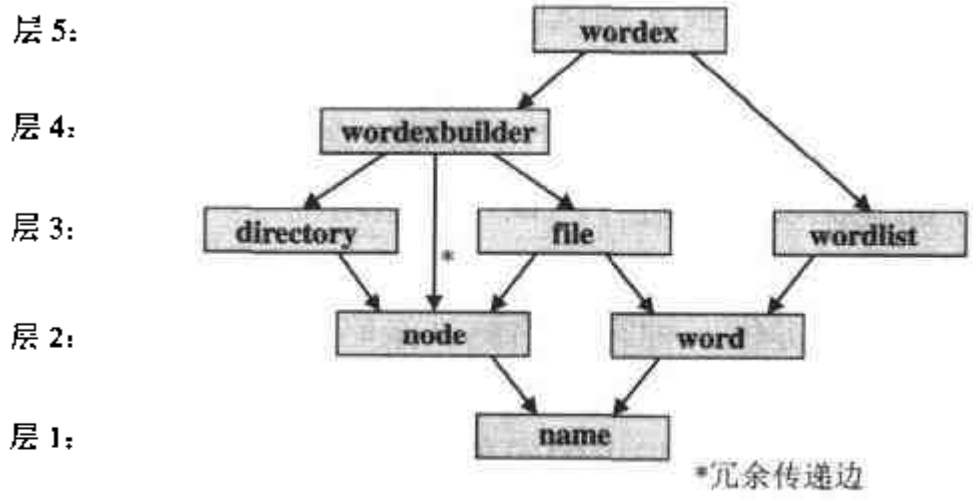


图 4-15　组件（直接）依赖图

这种分析的最大价值之一是，解开组件依赖图后，我们能够对物理设计的完整性作出实质性的定性评论，甚至不需要应用领域的最小的讨论。使这个过程自动化的简单工具很容易编写，并且实践已经证明它们对于大型项目的价值无法衡量。附录 C 描述了一个简单的组件依赖分析器。

4.8　分层次测试和增量式测试

组件是一个系统的基本构造部件。每一个组件都是不同的。每个都是*物理设计模式*的一个“实例”：“组件”。表面上，它们都有相同的基本物理结构——一个物理接口（.h 文件）和一个物理实现（.c 文件）。

在这个意义上，实现和测试一个软件系统就象建造一座房子。总体体系结构完成之后，砖（即**组件**而不是对象）一块一块地垒起来。每块砖的成功添加不仅依赖它自己的完整性，而且依赖于泥灰的完整性，泥灰被用来集成这块砖和这块砖所依赖的更低层次的砖。按照这个办法很容易检查每块砖的缺点。但是，一旦完成，这座房子经常很大也很复杂，为检查每个细节设置了太多的障碍。

> **定义**：分层次测试是指在每个物理层次结构上测试单个组件的惯例。

在建造房子的比喻中，一块砖代表一个惟一的组件（即，一个或者更多的类），不是单独的实例。在实践中，彻底的测试要求在装配每个组件之前测试该组件的完整性。在安装之前试运行每一个组件决不能排除后来在隔离的情况下进行更彻底检查的可能性。在物理层次结构的每个层次上测试组件接口，在本书中称为**分层次测试**（**hierarchical testing**）。

原　则

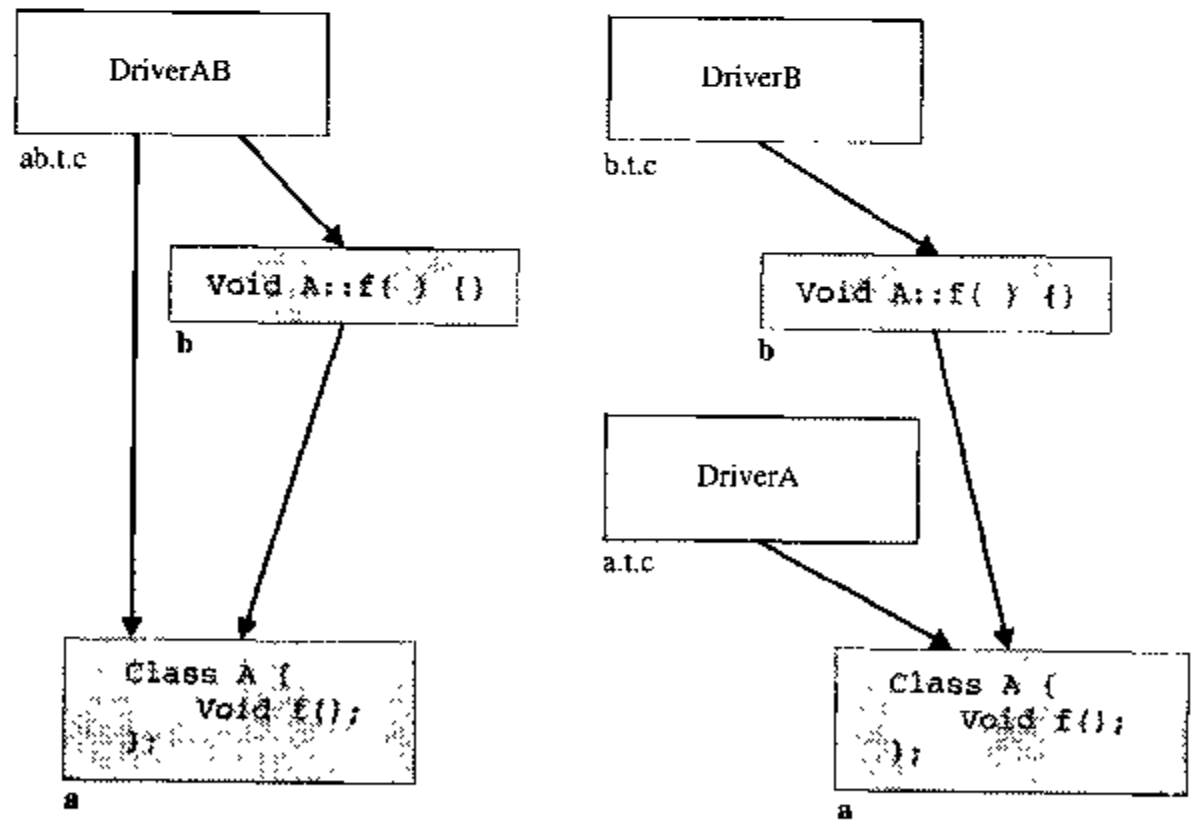
分层次测试需要为每个组件提供一个独立的测试驱动程序。

在这个方法中，为每个组件提供的独立测试驱动程序是由开发者建立的，同时用组件本身去试运行和验证在那个组件中实现的功能。这种测试驱动程序不仅在开发期间被广泛使用，而且以后还可用于质量保证（quality assurance），帮助描述它所验证的组件的预设行为。

每个组件都可使用单个测试驱动程序进行测试，该驱动程序试运行了在该特定的组件中实现的功能。物理依赖决定组件被测试、开发和运行的顺序。层次号可用于描述一个软件包中的局部组件的相对复杂性，还可提供一种客观的测试策略。

单独的驱动程序是保证遵守物理设计规则所必须的——否则我们将不能证明，在一个组件中声明的功能只能在由组件的依赖图所指出的组件子集中得到。为了说明为什么是这样，考虑一个违反设计规则的情况（显示在图 4-16 中），其中的组件 a 定义了一个有成员函数 f() 的类 A，还有一个组件 b（在 a 的上层），它非法地实现了 A::f()。

正如图 4-16（a）所示，一个单一的、同时连接到 a 和 b 的测试驱动程序不能检测出这种违反主要设计规则的情况。谁都可以从依赖图中知道，组件 a 是独立于组件 b 的，所以可以独立于组件 b 进行重用。如果某人试图独立于组件 b 重用组件 a，并且调用 f()，A::f()将在连接时作为一个没有定义的符号出现。



（a）用于许多组件的一个驱动程序　　（b）每个组件一个驱动程序

图 4-16　对单独的组件驱动程序的需求

在图 4-16（b）中，我们提供了不同的驱动程序来试运行每个组件中的功能。当连接组件

a 的驱动程序时，组件 b 被故意排除在连接过程之外。如果 a 的驱动程序是非常全面的（即，至少调用了每个函数一次），那么如果 A::f 没有定义，错误将在连接时被捕捉到——也就是说，甚至不必运行这个驱动程序。同样的技术也可以用来检测不可层次化的组件。

坚持编写单独驱动程序的另一个强制性的原因是，单个的组件一般都为一个测试驱动程序进行彻底测试提供了丰富的功能。在一个单个的驱动程序中对几个组件进行集中测试会导致过大的测试（或者，更可能的是，不足的测试）。

图 4-17 说明了分层次测试策略的抽象物理结构。在层次 1 上的每个组件可以只依赖外部组件（所有的外部组件都在第 0 层）。因此在第 1 层上的每个组件可以独立于所有其他（局部的）组件进行测试。

当我们继续深入到物理设计层次结构的更高层时，子系统的复杂性经常会呈指数级地增长。这种爆炸性的增长意味着我们不久会到达这样的境地：设计来覆盖一个高层次接口的全部行为的测试程序因太困难而无法编写，或者因花费时间太长而难以运行。

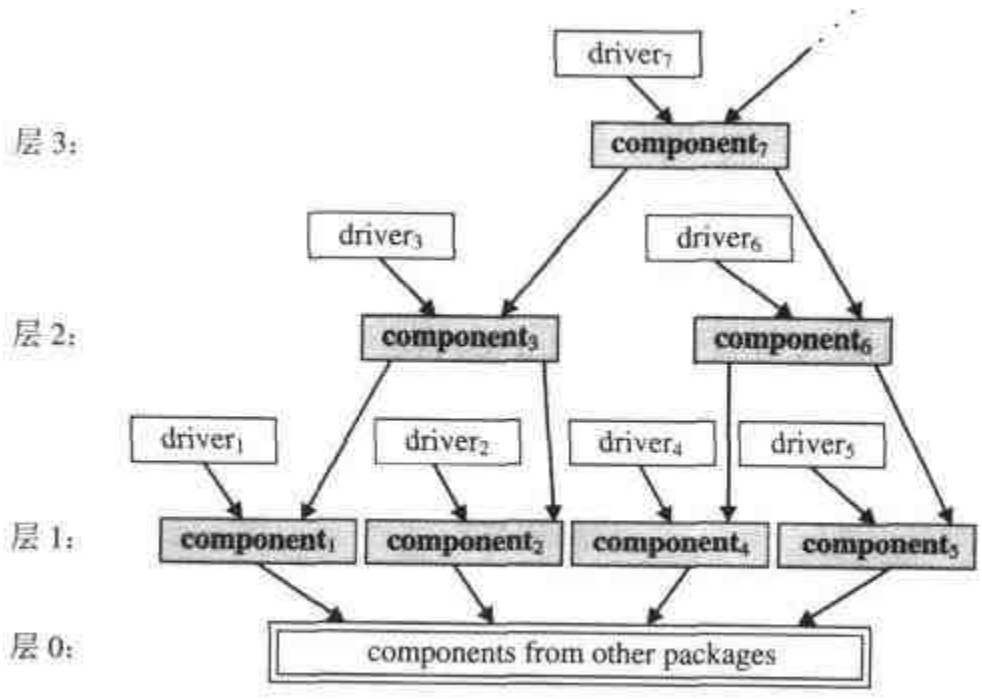


图 4-17　分层次测试策略

定义：增量式测试是指这样的测试惯例：只测试真正在被测试组件中实现的功能。

分层次方法使我们不必重新测试低层次组件的内部行为。如果我们只测试由一个特定组件添加的功能值，那么每个组件的测试复杂性就更可能保持在一个便于管理的水平上。只瞄准一个特定组件添加的新功能的测试方式，在本书中称为增量式测试（incremental testing）。

原　则

只测试在一个组件中直接实现的功能，能够使测试的复杂性与组件的复杂性相当。

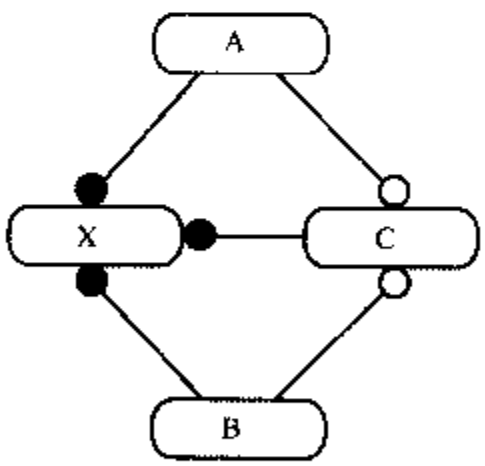
由于我们可以假设低层次的组件是工作正常的供应对象，所以增量式测试的任务经常简化为测试“低层次的对象联合起来形成高层次对象”的方式。编写增量式测试程序在实践中并不总是很容易，它需要编程者知道组件实现的内部知识。

例如，假设一个用户自定义类型 X 分层于 3 个其他类型（A、B 和 C）之上，这三个类型每个都在一个独立的组件中。图 4-18（a）显示了类 X 的部分定义。从这个局部的头文件中我们可以观察到图 4-18（b）的逻辑使用（uses）关系。现在假设每个类存在于一个独立的组件中，我们可以推断出其组件的依赖关系如图 4-18（c）所示。

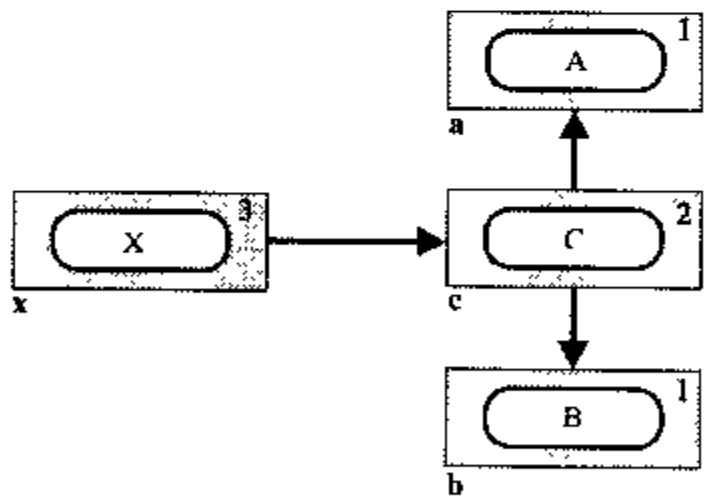
在这个高度简单化的例子中，测试类 X 的函数 f 和 g，实际上就是检验函数 X::f 和 X::g 是否各自与它们的适当的基础函数 C::u 和 C::v 正确地挂钩。由于组件 c 比组件 x 的层次更低，所以我们可以假设 c 已经被测试好了并且其内部也是正确的，在组件 x 的测试驱动程序中没有必要重新测试 C::u 和 C::v。与此相对照的是，X::h 的实现是实质性的，所以该组件的大多数测试工作应该集中于此。

```
class X {
    A d_a;
    B d_b;
    C d_c;
  public:
    //...
    int f() { return d_c.u(d_a); }
    int g() { return d_c.v(d_a, d_b); }
    int h();
};
```

（a）类 X 的定义



（b）类之间的逻辑关系



（c）组件之间的物理关系

图 4-18　为了测试目的分析一个分层对象

定义：白盒测试是指通过查看组件的底层实现来验证一个组件的期望行为。

查看组件实现是一种测试类型，称为白盒测试。通过小心选择试运行一个对象的所有内部功能的测试案例，白盒测试允许测试者用小得多的测试驱动程序测试几乎全部的内部代码。

白盒测试在帮助开发者排除低层次的程序错误方面是有效的，如简单的代码错误、甚至是经常导致内存泄漏或者强迫程序终止的基本算法错误等。因为白盒测试是依赖于实现的，一个基础对象若完全重新实现，将致使这样的测试无效。

白盒测试和百分之百代码覆盖度对于确保高质量的组件来说是必要的，但不是充分的。例如，如果作为一个开发者在分析一个问题时，遗漏了一个要求额外处理的特殊情况，那么不大可能只通过白盒测试来发现这种遗漏的情况。

> **定义：黑盒测试**是指仅基于组件的规范（即不必了解其基础实现）来检验一个组件的期望行为的惯例。

白盒测试检验代码工作情况是否与开发者的意愿相符，与此不同，黑盒测试检验组件是否满足其需求和是否符合规范。

黑盒测试由组件需求和规范直接驱动。黑盒测试对于大部分组件而言是独立于实现的。黑盒测试对于一个独立的测试员来说也是合适的，例如，QA 部门那些只能依靠文档来理解组件行为和正确用法的测试员。

正如图 4-19 所建议的那样，黑盒测试和白盒测试是有某种程度重叠的相互补充的技术。两种技术都很重要，各自侧重于质量的不同方面。白盒测试倾向于保证我们已经正确地解决了一个问题，而黑盒测试则帮助我们确认我们已经解决了正确的问题。

开发时往往趋向于利用白盒测试来确保可靠性。QA 可以使用白盒测试来确保覆盖度，但是也可以使用黑盒测试来检验伴随软件的规范和文档。另外，黑盒测试可以作为一种认**可测试**提供给客户来证实组件功能。而依赖于实现的白盒测试则可能作为内部测试。复杂组件的完全测试要有效地利用这两种策略。

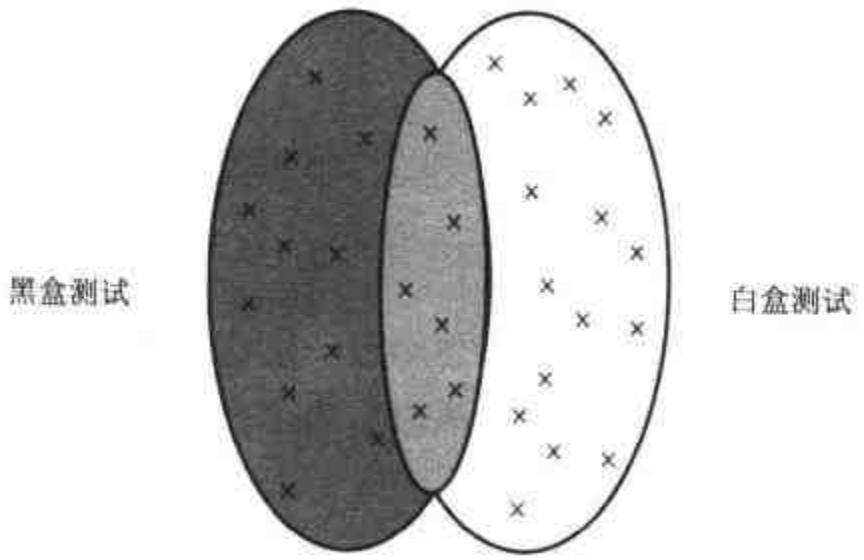


图 4-19　通过测试可发现错误

增量式测试的一个吸引人的特性是，测试任一特定组件的困难与组件本身添加的功能值大致成比例，而不是与组件所依赖的低层次组件的复合复杂性成比例。不管图 4-18 中的组件 a、b 和 c 的功能有多广泛，都有可能为组件 x 编写一个相对较短但却完全彻底的增量式测试程序，因为 X::f 和 X::g 只与一个有效的 C 子对象来回传递信息。

对本节总结如下：我们希望测试的复杂性与被测试的组件的复杂性相对应。我们希望在隔离的情况下测试所有的叶子组件。测试所有高层的组件时都假设它们所依赖的低层次的组件是内部正确的。这种增量式的、分层次的策略使得我们能将测试工作集中于它能做得最好的地方，并且可以避免冗余地重新测试已测试好的软件。

4.9 测试一个复杂子系统

让我们再次返回到图 4-2 中的点对点路径的例子。正如前面所讨论的，p2p_router 的接口很难有效地进行测试。正是这种接口最需要使用分层次测试来保证质量。

这个例子的一个实际实现分布于可层次化的等级结构中，如图 4-20 所示。这个子系统中的一些（但不是全部）组件被更高层的其他组件所重用。geom_point 和 geom_polygon 组件都属于一个独立的软件包 geom，并且被 p2p 软件包的实现者认为是内部正确的。这些可重用的库组件在 router 的实现中占有并非微不足道的部分。

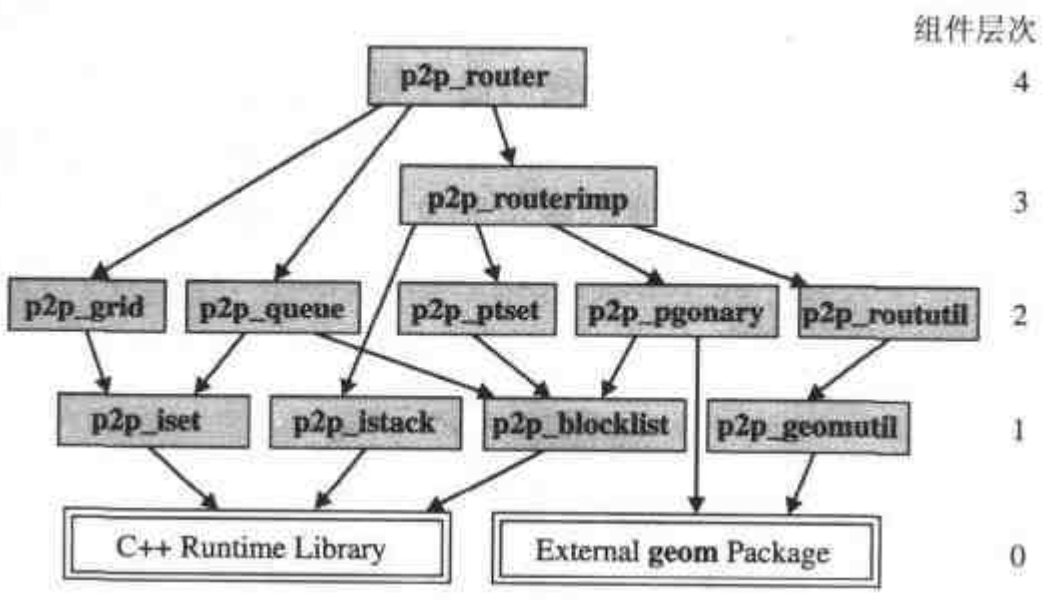


图 4-20 点对点路径的组件依赖图

让我们假设，在 p2p_router 子系统中，每个最低层次的组件都有可预测的行为并且在可测试方面很突出。层次 1 的组件仍然是可隔离测试的——独立于任何其他的 p2p 组件。在这个子系统中，层次 2 中的每个组件最多依赖于层次 1 中的两个组件。层次 2 中的每个组件实现了适当数量的附加功能，这些功能与已经测试好的低层次的功能复合在一起，不难理解和检验。

p2p_router 组件把 router 的客户与其实现的所有细节隔离开来，把许多实现推给了

p2p_routerimp 组件。反过来，p2p_routerimp 将把在 p2p_router.c 中另外定义的不可访问的子功能暴露给测试工程师。

在实际的实现中，p2p_router 只实现了少于百分之十的解决方案。其工作主要是协调在 p2p 子系统较低层次组件中实现的功能。通过小心地将这个复杂组件的功能的实现分布于有 10 个其他的局部子组件的层次结构中，我们已经提高了可维护性。每个子组件都有独立的、定义良好的接口，并且每个子组件都实现了可管理数量的可预测功能。

小结：通过将组件的实现分解成独立可测试（也可能是可重用）组件的一个可层次化的层次结构，我们已经为潜在地难以测试的组件的易测试性进行了设计。把测试工作分布到整个 router 子系统中，将指数级地减少在最高层次上要获得相同质量所需的回归测试的总量。人工检验代价高昂且容易出错，由于缺少时间和资源经常不进行这种检验。但是，层次化的结构使得子组件的可预测行为可以通过不需要人工干预的更健壮的方法来进行测试。

简言之，复杂子系统的分层次物理实现的测试与非层次化结构方案实现的测试相比，既可靠又节省开销。

4.10 易测试性和测试

易测试性（Testability）和测试（Testing）不是一码事。实际上，它们在很大程度上是质量的独立的两个方面。“可测试的”（testable），是指有一种有效的测试策略，这种策略使我们能检验由接口（以及支持文件）描述的功能是否已经实现了。“已测试的”（tested），是指产品已经被证实遵从了它的规范。**可测试性**是我们从设计一开始就努力追求的目标。**已测试**是一种我们的产品在交给客户之前必须达到的状态。**测试**（testing）是我们一直在做的事情。

原 则

完全的回归测试是昂贵的但也是必需的。建立彻底的回归测试的适当时间，与要测试的子系统的稳定性密切相关。

了解何时和花费多少进行测试是一种工程上的折衷。在实现过程中开发者对代码测试得越彻底，就越有可能避免不可预见的错误影响开发进度。

另一方面，开发彻底的测试程序非常费时并且会显著地增加前期的开发费用。这种额外的工作代价，仅通过减少花在维护、将来的功能增强甚至当前开发上的时间来补偿是不够的。

但是，在开发过程的早期阶段，许多组件的接口不可避免地会发生相当大的改变。一些组件会被分拆，另一些组件会被合并，还有一些组件会整个地消失。因此，在一个项目的早期就开发彻底的回归测试程序在有些情况下是不划算的。

随着项目的进展，各种组件将变得成熟。这些组件的接口也将变得更稳定——它们变化的频率会降低，比如少于一个月一次。正是在这个时候，QA 应该编写彻底的、系统的回归测

试程序来检验这些组件，并报告遗漏的或二义性的文档。

只要开发者设计的组件是可测试的，并提供了足够的和合适的文档，测试工程师应当可以十分直截了当地编写详细而系统的测试程序来检验每个组件提供的功能。[①]

如果开发人员在设计系统时不考虑易测试性，那么测试过程就可能不是简单明了的或有效的。为了方便进行有效的测试，必须远在对组件进行测试之前考虑一个系统的易测试性。

4.11 循环物理依赖

设计经常开始于非循环依赖，随着设计的演进，在系统功能的增强过程中，循环依赖会悄悄地混进来。例如，在 4.6 节的图 4-8 中的类 C1 中加入成员函数 g()，通过值返回一个 C2，如下所示：

```
class C1 {
    // ...
  public:
    C1 f();
    C2 g();     // new
};
```

成员函数引入了一个附加的依赖，导致了如图 4-9（b）的循环。由于加入了这个函数，组件 c1 和 c2 必须彼此“了解”（即它们各自的组件必须包含彼此的头文件），因而变得相互依赖。此时不再可能在没有另一个的情况下单独测试或使用 C1 或 C2。

原　则

组件之间的循环物理依赖限制了组件被理解、测试和重用。

组件之间存在循环物理依赖是不受欢迎的，不仅因为循环物理依赖使得组件难以测试且不可能独立重用，而且还因为循环物理依赖使得人们理解和维护这些组件更困难。一旦两个组件相互依赖，就需要同时理解两个组件，以便充分地理解其中的任何一个。

指导方针

避免组件之间的循环物理依赖。

紧密相关的类之间相互依赖很平常，但是，这些类将完全驻留在一个单个的组件中。如果我们发现两个（或更多）组件 c1 和 c2 相互依赖，我们有三种选择：

（1）对 c1 和 c2 重新打包以便它们不再相互依赖。

① 系统测试的全面论述见 **marick**。

（2）将 c1 和 c2 从物理上组合成一个组件 c12。

（3）将 c1 和 c2 当作一个单一的组件 c12 来考虑。

最好的解决方案是在循环依赖发生之前纠正它；或者，如果它们确实已经混入了，那么就应该立即进行检测和纠正。第 5 章研究了一种技术，该技术可在保持预期特性的同时，对一个循环依赖的设计进行重新构造，以消除循环。

当复合抽象中的对象自然地紧密耦合在一起并且不需要考虑别的问题时，把组件合并为单一的组件是正确的解决方案。如果一个类是另一个类的友元，则进一步暗示这些类应归入同一个组件（见 3.6.1 节）。合并紧密耦合的内聚组件也有如下受欢迎的好处：减少了组件数量，从而降低了系统的物理复杂性，又不会进一步危及易测试性或独立重用。

有时单个的、紧密耦合的抽象会被认为太大了，不适合在一个组件中实现，它们将被分拆成相互依赖的组件。但是，大多数情况，抽象的紧密耦合的部分可能与实现的其他部分隔离开来，并被放在一个单个的组件中，这个组件反过来依赖其他独立的组件。这些独立的组件现在可以在隔离的情况下进行彻底的测试（见 5.9 节）。

如果没有别的解决方案，那么我们可以主观上把相互依赖的组件看成是一个大的组件，如图 4-21 所示。这种方法就目前来说是容易的，但是，从长远观点来看则是最不受欢迎的方案。这些物理上独立但是紧密耦合的组件必须被人为地看成是一个单一的物理单元，这种单元会降低可维护设计的一致性。尽管这样的依赖不是我们所希望的，但是只要这种“污渍”的数量和大小保持最小，系统的整体易测试性就不会丧失。

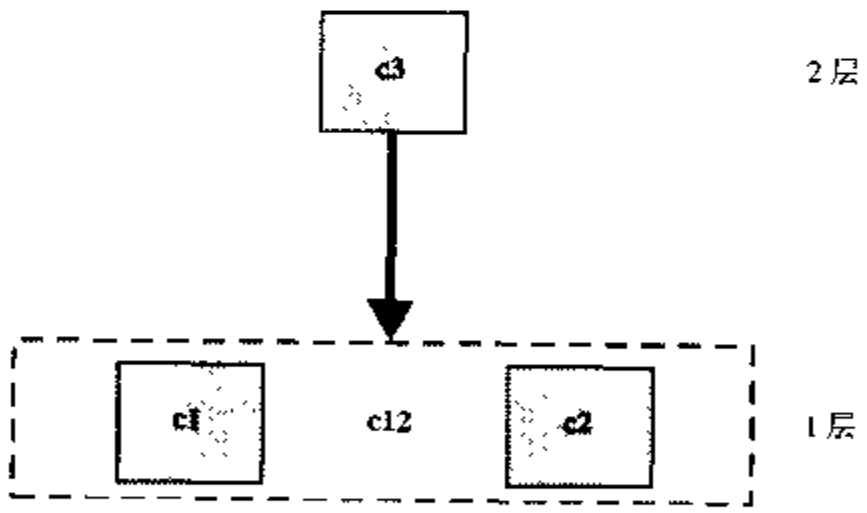


图 4-21　将两个相互独立的组件当作一个来看待

4.12　累积组件依赖

现在我们通过提供一种度量标准来形式化有关设计质量方面的讨论。这种度量标准在本书中称为一个子系统的**累积组件依赖**（cumulative component dependency，CCD）。CCD 与增量式回归测试的连接时开销紧密联系。更确切地说，CCD 提供了一种数字表示的值，表示与

开发和维护一个特定子系统有关的相对开销。

> **定 义**：**累积组件依赖**（CCD）就是在一个子系统内的所有组件 C_i 之上，对每个组件 C_i 在增量式地测试时所需要的组件数量进行求和。（即**累积组件依赖**就是为了增量式地测试子系统中所有组件所需的组件数量的总和）

连接大型程序要花费很长的时间。在创建组件及其测试驱动程序的过程中，开发人员一般需要多次连接一个组件。此后，无论何时运行回归测试程序，组件都必须与其驱动程序相连接。对于小型的项目，连接时间与单个组件的编译时间相当。当项目变大时，连接时间显著增长，甚至比最大的组件所需要的编译时间还要多得多。

我们的大多数开发时间消耗在低层次的组件上，主要是因为低层次的组件比高层次的组件多很多。系统的这些低层次的部件可能是错综复杂的，有时可以选择它们来调节性能。使开发、测试和维护低层次组件的过程简化并更有效率，这对我们是有好处的。

为了方便讨论，我们假定一个设计中的依赖形成了一个完美的二叉树。正好过半的组件在层次 1 上，并且能够在完全隔离的情况下进行测试。另外的四分之一组件，每个依赖于两个叶子组件。如果我们用 L 代表树中层次的数量，那么 2^L-1 个组件中只有一个组件会实际上依赖所有其他的组件。尽管实际的设计没有这样规整，但是测试非循环依赖的组件层次结构的优点仍然很清晰。

考虑与开发一组组件相关的开销。让我们暂时假设连接时间与被连接的组件的数量是成比例的[1]。例如，如果将一个组件连接到一个测试驱动程序要花费 1CPU 秒，那么连接 5 个组件大约会花费 5CPU 秒。

存在循环依赖时，可能有必要连接大多数或全部的组件，以便能测试其中的任意一个。一个完全相互依赖的设计，并不一定每个组件都直接依赖于每一个其他的组件[2]。假设我们的系统是非常紧密耦合的，并且每个组件都直接或间接地依赖所有其他的组件。如果我们用 N 代表系统中的组件数，将这些组件中的任何一个连接到其测试驱动程序的开销与 N 成正比。那么为这些组件建立全部 N 个测试驱动程序的单独的连接开销会与 N^2 成正比。这个事实解释了为什么大型系统中连接的开销常常是运行彻底的回归测试的开销的主要部分。

原 则

N 代表系统中组件的数量。

$\text{CCD}_{\text{循环依赖图}}(N)$ =（组件的总数）·（测试一个组件的连接时开销）$= N \cdot N = N^2$

① 这个假设当然只是一个粗略的估计，因为连接开销会明显受到组件规模变化以及函数调用层次结构的构造的影响。

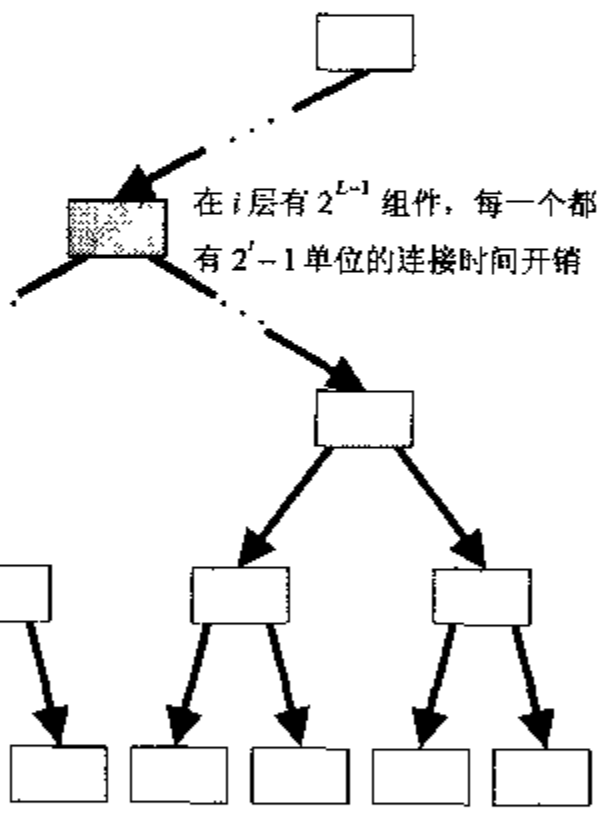
② 一个完全相互依赖的设计有一个直接依赖图，这个图是“强连接的（strongly connected）”，但不一定是“完全连接的（complete）”。这些不同术语的正式定义见 **aho**（5.5 节，189 页以及 10.3 节，375 页）。

现在考虑，如果依赖是非循环的并且形成了一个二叉树时会出现什么情况。现在，不是所有的组件的连接开销都相同。在层次 1 的组件在一个单位时间（例如，1CPU 秒）内可以被连接到各自的测试驱动程序。与组件测试相关的连接开销有至少一半实际上可以被消除。层次 2 上的每个组件依赖层次 1 上的两个组件，并且组成一个大小为 3 的子系统（它将消耗 3CPU 秒来进行连接）。也就是说，与连接相关的测试开销的 1/4 可以戏剧性地减少（乘以一个因子 $N/3$）。在这个假设的系统中只有一个组件，根（root），会需要 N CPU 秒的连接时间，而以前，N 个组件中的每一个都需要这么多时间。

在数学上我们可以表示出增量式地测试一个系统（其物理依赖形成一棵二叉树）的总的连接开销与 $N\log(N)$成正比，而不是与 N^2 成正比（见图 4-22）。例如，在有 15 个组件的情况下：

$$\text{CCD}_{\text{平衡二叉树}}(15) = (15+1) \cdot (\log_2(15+1) - 1) + 1 = 49$$

该层组件数量	连接需要的时间单位	层次
1	2^L-1	L
.	.	.
.	.	.
.	.	.
2^{L-i}	2^i-1	i
.	.	.
.	.	.
$\frac{2^L}{8}$	7	3
$\frac{2^L}{4}$	3	2
$\frac{2^L}{2}$	1	1



设 L 是系统中层次的数量（二叉树的深度）。

设 $N = 2^L - 1$ 是系统中组件的数量。

$$\text{CCD}_{\substack{\text{平衡}\\\text{二叉树}}}(N) = \sum_{i=1}^{L} \left(\begin{array}{c}\text{在 } i \text{ 层上的}\\\text{组件数量}\end{array} \right) \cdot \left(\begin{array}{c}\text{测试一个 } i \text{ 层上}\\\text{组件所需的连接}\\\text{时间开销}\end{array} \right)$$

$$= \sum_{i=1}^{L} 2^{L-i} \cdot (2^i - 1)$$

$$= \sum_{i=1}^{L} 2^L - \sum_{i=1}^{L} 2^{L-i}$$

$$= 2^L \cdot \sum_{i=1}^{L} 1 - \sum_{i=1}^{L} 2^{i-1}$$

$$= 2^L \cdot L - (2^L - 1)$$

$$= 2^L \cdot (L-1) + 1 \quad \{ \text{可用于与高度为 } L \text{（整数）的依赖关系二叉树进行比较}$$

$$= (N+1) \cdot (\log_2(N+1) - 1) + 1$$

$$= (N+1) \cdot \log_2(N+1) - N \quad \{ \text{可用于与大小为任意正数 } N \text{ 的理论上的二叉树进行比较}$$

$$= O(N \cdot \log(N)) \quad \{ \text{近似的连接时开销}$$

图 4-22　为一个依赖关系的二叉树计算连接时开销

原　则

非循环物理依赖可以明显减少与开发、维护和测试大型系统相关的连接时开销。

非循环依赖的好处很多。如果一个非循环设计的依赖关系是树型的层次关系，那么其单个测试驱动程序的平均连接时间是与组件数的 log 成正比的，而不是像循环设计的情况那样，与组件数本身成正比。

原　则

设 N 为系统中组件的数量。

CCD 循环依赖图 $(N) = (N+1) \cdot (\log_2 (N+1) - 1) + 1$

图 4-23 比较了当 N=1、3、7 和 15 个组件时，测试出的与循环的系统和层次结构的系统相关的连接时开销。显示在对应于依赖图中的每个组件的位置上的数字，表示与增量式测试该组件相关的连接开销。每个系统的 CCD 被计算并显示在依赖图的底部。每个树型系统的 CCD 按两种方式进行计算：一种是一层一层地计算，另一种是使用图 4-22 导出的方程进行计算。

假设我们正在开发一个有 63 个组件的系统，每个组件都有自己的测试驱动程序。在循环设计中，每个组件将花费 63 秒时间来重新连接以便于测试。将此系统与一个层次结构的设计相比（如图 4-24 所分析的那样），超过一半的组件可以在 1CPU 秒中完成连接，1/4 的组件在 3CPU 秒内完成连接，1/8 的组件在 7CPU 秒内完成连接，以此类推。63 个组件中只有一个组件花费了全部 63CPU 秒来进行连接。连接全部 63 个测试驱动程序的总的开销，在图 4-24 中用两种方法计算的结果都是 321CPU 秒（5.35CPU 分）。将这个开销与连接所有 63 个测试驱动程序到一个循环依赖的系统中所要花费的开销（63^2=3969CPU 秒(1.1CPU 小时)）比一比。

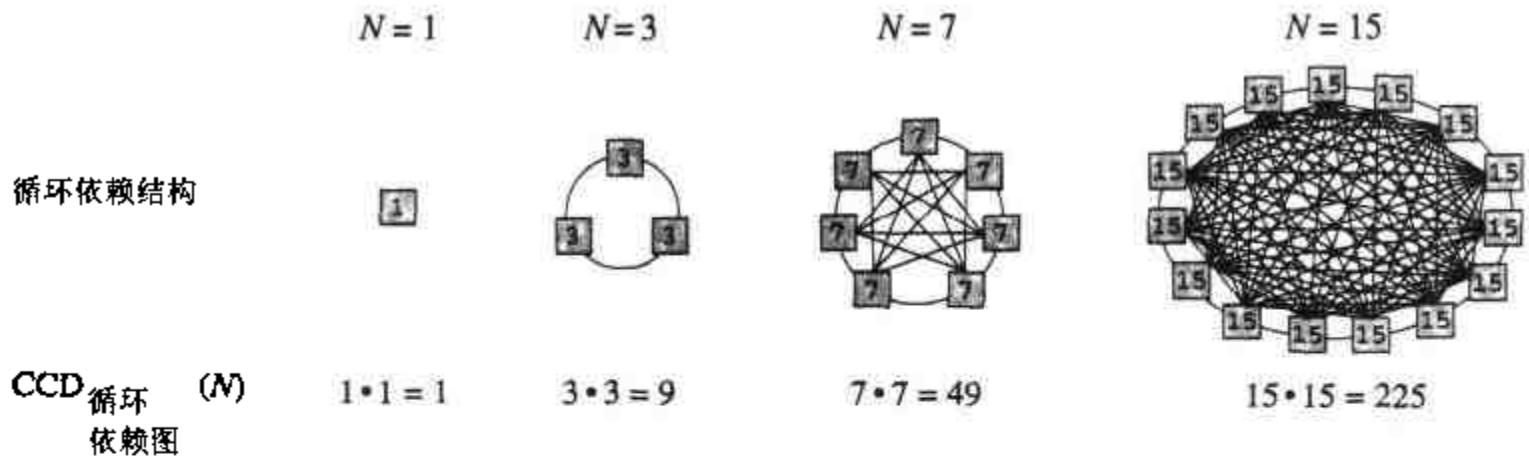


图 4-23 与增量式测试相关的相对连接开销

层次号	该层上的组件数量		连接该层上的一个组件所需的开销		该层上所有组件的连接开销
1	32	•	1	=	32
2	16	•	3	=	48
3	8	•	7	=	56
4	4	•	15	=	60
5	2	•	31	=	62
6	1	•	63	=	63
				总数 =	321

$$CCD_{\text{平衡二叉树}}(63) = (63+1) \bullet (\log_2(63+1) - 1) + 1$$
$$= 64 \bullet 5 + 1$$
$$= 321$$

图 4-24 N=63 的平衡二叉树层次结构的连接开销

如果组件的数目更大，例如为 1023 个组件，那么在一个树型设计中连接一个组件的平均开销将比循环设计的开销少 2 个数量级(连接每个组件的平均开销分别为 9CPU 秒和 1023CPU 秒)。我们可以使用在图 4-22 中导出的方程来预测这个系统的 CCD。在一个有 1023 个组件的

系统上建立组件回归测试的总的连接时间的范围，可以从层次化设计系统的 1024×9+1=9217CPU 秒（刚超过 2.5 小时）到循环依赖系统的 1023×1023=1046529CPU 秒（超过 12 天）。

一个单一的系统不太可能不被进一步分成多个软件包而由 1023 个组件直接构成。确保软件包之间的非循环依赖比确保单个组件中的非循环依赖更重要。（见 7.3 节）

CCD 也是增量式回归测试对累计磁盘空间需求的预测指标。当我们并行地增量式测试一个大型系统时，磁盘空间会变成一个需要考虑的重要因素。在磁盘上每个独立的可执行测试程序占用的磁盘大小，与测试驱动程序必须静态连接的组件的数量大致成正比。因此，循环依赖系统比层次设计需要多得多的磁盘空间。

小结：我们的目标是能够为每个组件建立一个测试驱动程序，该驱动程序与要测试的组件及其所依赖的（少量）组件相连接。CCD 是一种测量方法，它根据与增量式地测试每个组件相关的总的连接时间来量化一个系统的耦合程度。在增量式地测试组件所需的连接时间和所需要的磁盘空间方面，循环依赖的组件展示了二次方的特性。相反，形成一个非循环的（树形）层次结构的组件依赖关系则可以极大地减少增量式组件测试的连接开销。

4.13 物理设计的质量

在本节中，我们将基于物理依赖关系来刻画什么因素使得一个设计是可维护的。我们将继续讨论 CCD 并且用它来表示一个子系统的整体可维护性。我们还将讨论如何使用 CCD 来衡量物理设计质量上的增量式的改进。

设想我们加入了一家开发一个超大型系统的公司。分配给了我们一个有 150000 行 C++ 代码的子系统，并要求我们理解它是做什么的，还要就如何改进它提出建议。通过仔细检查，我们发现（大部分的）组件与第 2 章和第 3 章中提出的规则和指导方针一致。接着我们又发现系统中的大多数组件（直接或间接地）依赖其他的大多数组件。我们该怎么办呢？很不幸，这个故事没有令人愉快的结局。任何人能做的最好的事情是努力将整个设计装入自己的头脑，这也可能需要几个月。

如果同样的子系统着眼于用最小 CCD 的方法来设计，则大多数（如果不是全部的话）的循环依赖会被消除。因此有可能在隔离的情况下研究子系统的部件，以便测试、检验、调整甚至替换它们，而无论在主观上还是在物理上都不必涉及整个子系统。换句话说，有效地减少组件间的相互依赖（通过 CCD 来量化）可以提高可理解性，因而也提高了可维护性。

可理解性是几个难以量化的性质之一，但是它对于最小化组件之间的依赖非常有好处。有选择地重用是另一个这样的性质。考虑图 4-25 所示的子系统的体系结构。这个系统由 7 个组件组成，每个组件直接或间接地依赖系统中的其他每个组件。每个组件可以直接测试，但是没有一个组件可以在隔离的情况下测试或者独立于其他组件而重用。因为每个独立的测试驱

动程序被迫与整个系统连接，仅存储这些独立的驱动程序所需的磁盘空间的总量也是二次的。

现在假设图 4-25 中的设计的循环依赖被消除了，它是可层次化的了。尽管层次性（levelizability）是我们非常希望的，但是一些可层次化的体系结构比其他结构更易于维护和重用。考虑图 4-26 所示的结构设计，每个结构包含 7 个组件，并且每个都是可层次化的。图 4-26（a）显示了层次化的一个极端版本。这种设计被称为**垂直的**（**vertical**）。在这个系统中，每个组件依赖于在更低层次上的所有其他组件。垂直的子系统显示出高度的耦合程度，这抑制了独立重用。重用一个规模为 *N* 的垂直系统中的一个随机选择的组件，将导致平均要连接(*N*–1) / 2 个额外的组件。相应地，用于存放增量式测试驱动程序的平均磁盘空间也非常大。

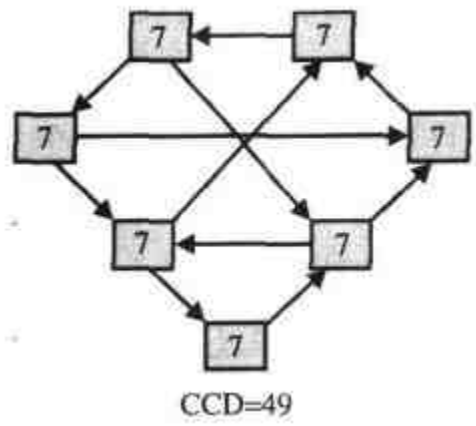


图 4-25　规模为 7 的循环依赖子系统体系结构

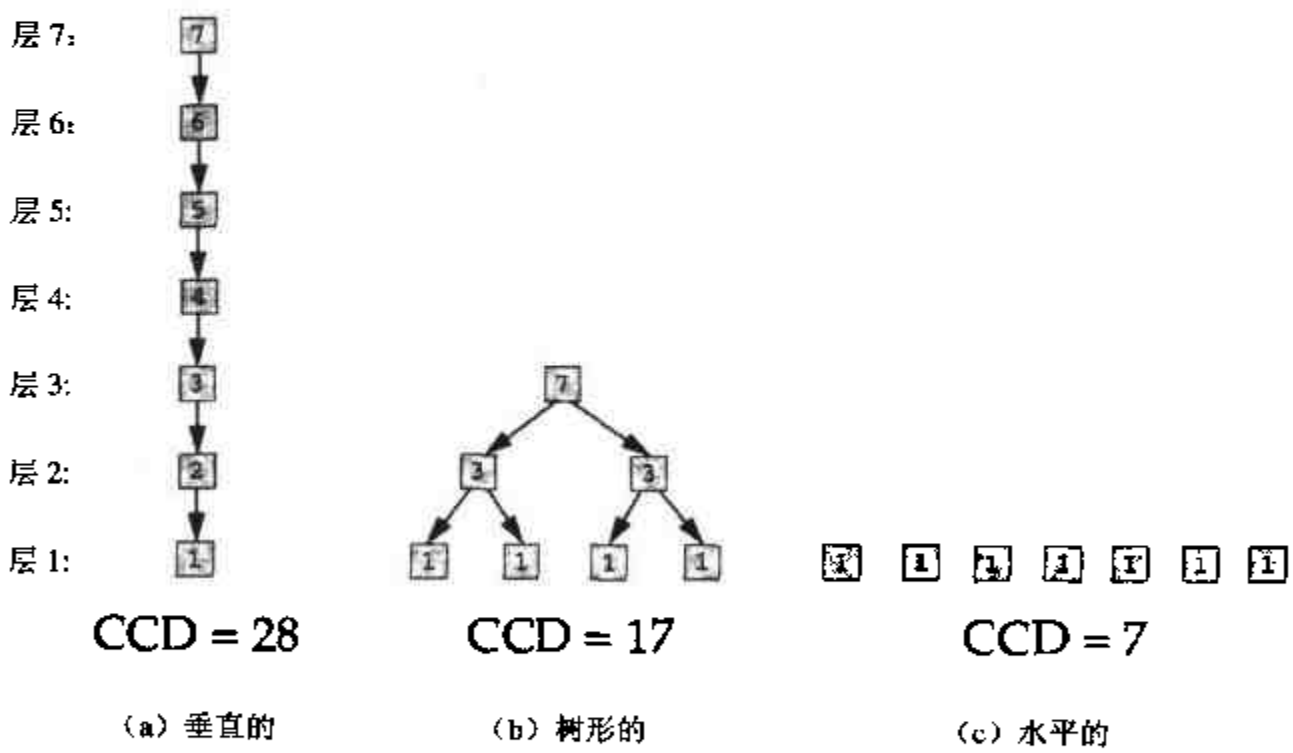


图 4-26　规模为 7 的各种组件体系结构的 CCD

垂直系统在测试和重用方面极不灵活。测试纯垂直系统的次序只有一种，这个次序完全由它的层次来决定。从连接时间来看，开发一个垂直的子系统也是相对昂贵的。这个系统的 28 个单位的总连接开销（CCD）比在图 4-25 中显示的循环依赖子系统的 49 个单位的总连接开销的一半还要多。另外，将一个垂直子系统划分为并行的开发工作并分配给多个开发者，也会相对比较困难。但是，一个垂直的子系统是非循环的，所以，从性质上看，它比循环依赖的系统更容易维护。

图 4-26（b）显示了一个二叉树形式的层次结构设计。正如我们所知道的，这个设计中有

超过一半的组件每个只在 CCD 中占一个单位。设计不会是完美的二叉树，但是一个二叉树的 CCD 为比较许多典型的应用提供了一个好的基准。树型设计的耦合程度更低，比垂直设计灵活得多并且更适合重用。在每个层次上，一般都有几个可以独立于系统的其他部分进行测试并可能重用的子系统。存储大多数增量式测试驱动程序所需要的磁盘空间也相对较小。

通过使依赖图更平而不是更高，我们增加了灵活性。设计越扁平，独立重用的潜力就越大。使依赖关系扁平还有助于减少理解和维护的时间。设计越扁平，就越有可能对一个单一的隔离组件或一个小的子系统进行错误追踪，因而就需要越少的磁盘空间来存储检查错误的测试驱动程序。

图 4-26(c)显示了层次化领域的另一极端。这种类型的设计被描述为**水平的**(**horizontal**)，因为所有的组件完全是独立的，并且彼此之间没有耦合。对属于纯水平子系统的组件可以以任何次序进行测试，并且可以以任何所希望的组合进行重用。每个增量式测试驱动程序所需的磁盘空间十分少。这种依赖特性在可重用组件库中是具有代表性的，但对于子系统来说则一般不具有代表性。

基于设计的 CCD，我们可以对一个给定大小的设计的可维护性和可重用性（但不一定是“长处”）做某种客观的和定量的描述。设计依赖形成了从循环到垂直、到树型、到水平这样一个连续区间。即使存在循环，每个设计也可以被赋予一个 CCD。在其他条件都相同的情况下，CCD 越低，开发和维护系统的代价就越低（就连接时间和磁盘空间而言）。

还有另外一个原因使我们努力追求一个具有最小 CCD 的分层次的系统。需求很少是一成不变的，在项目的开发过程中可能会有变化。通过将实现分布在一个由组件组成的层次结构中，可以使设计更能适应需求的变化。一个体系结构越扁平，软件规范的变化对整个系统产生影响的可能性就越小。这种规范的变化引起的预期开销，直接与系统中的平均组件依赖（Average Component Dependency，ACD）相关。

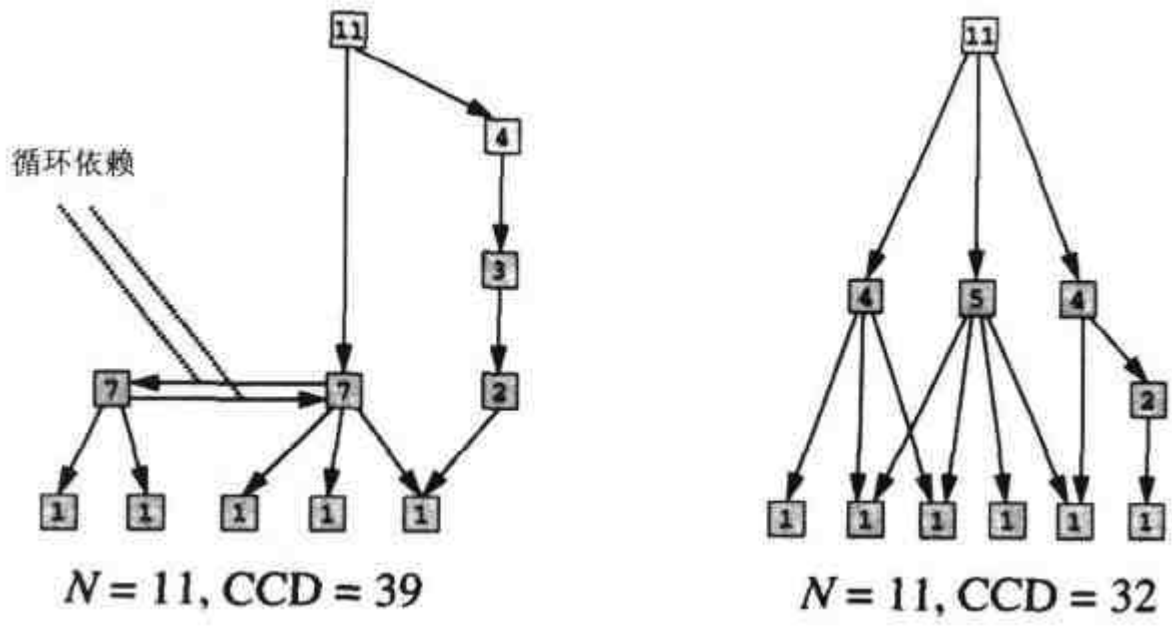
> **定义：平均组件依赖**是指一个子系统的 CCD 与系统中的组件数量 N 的比值：
>
> $$\mathrm{ACD}(\text{子系统}) = \frac{\mathrm{CCD}(\text{子系统})}{N_{\text{子系统}}}$$

例如，在一个完全水平化的系统中，一个单一组件的规范的改变只会引起一个组件变化。对于一个有 N 个组件的树形体系结构，平均需要改变大约 $\log(N)$个组件。而对于垂直结构，改变一个组件的接口，我们可能预期要重新访问$(N+1)/2$ 个组件。最后，对于一个完全循环依赖的设计，单个组件接口的变化可能会影响所有 N 个组件。

原　则

CCD 的主要用途是，对一个给定体系结构的较小改动引起的整个耦合结构的变化进行量化。

作为减少 CCD 的一个例子，考虑图 4-27 所示的具有相似依赖结构的两个系统。A 设计有两个组件循环依赖。测试这些组件中的任何一个，都需要连接这两个组件以及其中的任何一个组件所依赖的所有组件。这使得它们每一个都有一个单独的组件依赖（7）。还要注意 A 设计的右部，层次结构的一部分是纯垂直的。



（a）初始设计 A　　（b）修改后的设计 B

图 4-27　一个子系统的两种设计方案的依赖图

在第 5 章，我们将详细地介绍几种减少连接时依赖的技术。为了改进这种设计，我们首先应打破循环依赖，然后检查垂直部分，看能否使它不那么顺序化。在这种情况下，只有通过逐步将代码提升到更高的层次以及/或者提取出一个共享的资源，才有可能打破循环。至于垂直的那段，有可能可以将垂直链中的一个或多个组件移走并使它们变成叶子组件，从而独立于其他组件。做了这些修改后的结果是设计 B，如图 4-27（b）所示。我们的原始设计的 CCD 值 39 远远低于完全相互依赖的设计的 CCD 值 121。但是我们仍然能够把这个差不多分层的系统的 CCD 值从 39 降到 32——改进了大约百分之十八。

CCD 值是一种在系统中描述物理耦合的客观度量方法。CCD 可以用非常高的增量式开发和维护开销来标记子系统。例如，一个垂直的链是有最高 CCD（$N(N+1)/2$）的可层次化配置。因此，CCD 大于 $N(N+1)/2$ 就说明至少有一个循环依赖存在。但是，CCD 本身不是一个子系统的质量测量指标。

我们可以方便地使用图 4-22 导出的替换方程来确定规模与图 4-27 中所显示的设计一样的、（理论上）类似二叉树的体系结构的 CCD 值。图 4-28 证实了一个有 11 个组件的类似二叉树的体系结构有 32.02 的 CCD 值，这与设计 B 的 CCD 值差不多。

$$CCD_{\substack{平衡\\二叉树}}(N) = (N+1)\bullet\log_2(N+1) - N$$

$$CCD_{\substack{平衡\\二叉树}}(11) = (11+1)\bullet\log_2(11+1) - 11$$

$$= 12\bullet\log_2(12) - 11$$

$$= 32.02$$

图 4-28　计算规模为 11 的理论平衡树的 CCD

定　义：**标准累积组件依赖**（NCCD）是指包含 N 个组件的子系统的 CCD 值与相同大小的树型系统的 CCD 值的比值。

$$NCCD(子系统) = \frac{CCD(子系统)}{CCD_{平衡二叉树}(N_{子系统})}$$

一个系统的 NCCD 可以用来描述该系统相对于同样规模的理论二叉树系统的物理耦合程度。返回去查阅图 4-27，设计 B 的 NCCD 是 32/32.02=1.00，设计 A 的 NCCD 为 39/32.02 = 1.21（而完全相互依赖的实现的 NCCD 是 121/32.02 = 3.78）。

如果 NCCD 的值小于 1.0，则可以认为是较“水平化的”或松散耦合的；这样的系统可能很少使用重用。如果 NCCD 的值大于 1.0，则可以认为是较“垂直的”和/或紧密耦合的；这样的系统可能正在大量地重用组件。如果 NCCD 的值远远大于 1.0，则表明在系统中可能有明显的循环物理耦合。

就我们可以获得的 CCD 值来说，可维护性的程度依赖于子系统的特性。我们不可能总是获得理想的树型的可维护性。对于水平的组件库，我们会希望有一个低得多的 CCD。对于大量使用重用的高度相互连接的拓扑结构，如 2.5 节中图 2-8 的窗口系统，其 CCD 将会更高。

NCCD 不是一个系统的相对质量的衡量指标。NCCD 只是一种描述一个子系统内的耦合程度的工具。增加在一个系统中的组件的数量可以人为地减少 NCCD。这样做的一种方法是：消除完全有效的重用；但这不大可能是一种改进。

图 4-29 显示了两个有等价功能的设计。设计 B 比设计 A 大 50%，CCD 也大 25%。另一方面，设计 A 通过重用显示出比设计 B 更多的物理耦合。虽然如此，设计 A 仍然非常可能是更精心策划的和更好维护的设计。

原　则

使一给定组件集合的 CCD 最小化是一个设计目标。

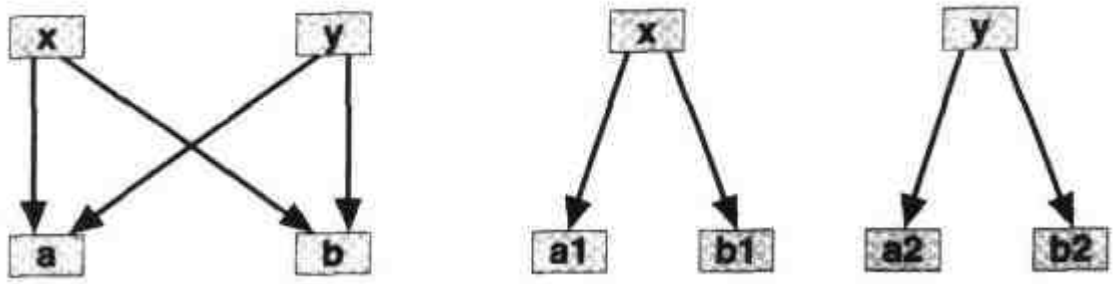


图 4-29　举例说明冗余的影响

我们几乎总是希望减小给定规模系统的 CCD 值。减小一个系统的规模（组件数）也是我们所希望的，但是不能以引入循环依赖、不适当的合并组件或建立不便管理的大型编译单元为代价。当增加一个系统的组件数量实际上减小了 CCD 的值时，很可能因此提高了设计的整体质量。

总之，CCD 度量方法已被引进来显式地标识（我们想最小化的）依赖的种类。NCCD 提供给了我们一种定量方法，这种方法能够把子系统的物理依赖区分为水平的、树形的、垂直的或循环的。一个系统的准确的 CCD（或 NCCD）数值并不重要。重要的是要积极主动地设计系统，使每个子系统的 CCD 值始终不会比必要的更大。

4.14　小结

高质量的复杂子系统由许多分层组件形成的非循环的物理层次结构组成。在系统层次上进行完全测试不仅昂贵而且根本不可行——如果不是不可能——对“好”接口尤其如此。

一个“好”接口将实现的复杂性封装在一个简单的易于使用的外表后面。同时，这使得我们通过这个接口来测试其实现异常困难。

本章中的许多测试策略是由十多年前易测试性设计（DFT）的成功激发出来的。但是，和现实世界中的对象不同，定义在一个软件系统中的类的实例与定义在该系统之外的同一个类的实例并没有什么两样。我们可以利用这个事实来隔离地验证设计层次结构的各个部分，从而减少集成的部分风险。

隔离测试是确保复杂低层次组件的可靠性的一种很合算的方法。通过尽可能将测试推到设计层次结构的最底层，我们可以确保如果组件或子系统被增强、移植或在另一个系统中重用，它将独立于客户程序而保持其指定的特性。

层次号在一个子系统内基于组件对其他组件的物理依赖来描述组件。另外，层次号给有非循环依赖组件图的系统提供了能够进行有效测试的一种顺序。若某子系统的组件依赖形成

了一个直接的非循环依赖图（DAG），则称该子系统是**可层次化的**（**levelizable**）。一个层次化的组件依赖图使得一个系统的物理结构更容易理解，因而也更容易维护。

分层次测试是指测试物理层次结构中的每一层的组件。每个较低层次的组件都应该提供独立于更高层次组件的良好定义的接口，并实现可被测试、验证和重用的功能。

增量式测试是指使单个测试驱动程序只测试被测试组件中实际实现的功能。在物理层次结构的更低层次中实现的功能，此时假设为是内部正确的。因此，增量式测试反映的是被测试组件的实现的复杂性，而不是这个组件所依赖的组件的层次结构的复杂性。增量式测试是白盒测试的一种形式，它建立在了解组件实现的基础上，以提高可靠性。黑盒测试源于需求和组件规范，并独立于实现。这两种测试形式是相互补充的，它们都有助于保证整体质量。

易测试性是一个设计目标。循环物理依赖抑制测试、理解和重用。累积组件依赖（CCD）提供了一种与增量式测试一个给定子系统相关的整体连接时开销的粗略数值度量。更一般地，CCD 是一个给定设计的相对可维护性的一个指示器。

循环依赖设计不是可层次化的。众所周知，这样的系统难以维护，并相应地有高 CCD 值。在设计中，层次结构越扁平，CCD 值就越低。使物理依赖（结构）更扁平有助于减少理解、开发和维护所需的时间，同时又提高了一个系统的灵活性、易测试性和可重用性。NCCD（标准 CCD）可以帮助我们将任意的设计的物理结构分类为循环的、垂直的、树形的和水平的。

5 层次化

在确立一个系统的全面物理质量时，系统内的连接时依赖（以 CCD 量化）扮演主要角色。质量的更传统方面，例如可理解性、可维护性、易测试性和可重用性，都紧密地依赖于物理设计的质量。如果不小心加以防范，循环物理依赖将使系统失去这种质量，使得它缺乏灵活性和难以管理。

甚至对于可修正的设计，维护和改进起来也可能要付出不必要的昂贵代价。对大型的、低层次子系统的被迫依赖，会给更高层次的子系统造成显著的开发负担。最小化这种依赖的影响有助于提高系统的物理质量。

在本章中，我们将研究消除循环依赖或其他过度连接时依赖的若干技术。升级（escalation）和降级（demotion）是相关的技术，它们把设计中的循环依赖部分移到物理层次的不同级别上。不透明指针（opaque pointer）和哑数据（dumb data）可用来消除概念上的依赖关系的物理隐含。冗余（redundancy）和回调（callback）也是我们要讨论的其他两种技术，它们可用于防止不必要的物理依赖。最后，将介绍一个管理类以及两种通用技术（分解封装和升级封装），以帮助编程人员创建可测试、可重用组件的、有效的、封装的层次结构。

贯穿本章，我们使用了许多取自若干应用领域的例子来阐释这些技术（在各种不同的上下文中）。偶尔我们会为了参考目的提供一段真实的源代码来使例子具体化。

5.1 导致循环物理依赖的一些原因

本节我们将讨论循环物理依赖在实践中可能出现的二种方式。为了说明这个问题的宽度，我们在单独的小节中介绍并讨论每一个例子（没有尝试去解决它们）。这些特定的问题以及许多其他的问题，在本章的其余部分介绍适当的技术时将得到解决。

5.1.1 增强

最初的设计通常都经过精心策划，常常是可层次化的。此时，未预见到的客户需求可能会使我们在增强系统时考虑不周，从而导致不必要的循环依赖。例如，我们有时候会发现相似的对象会因为这样那样的原因（例如：性能原因）而共存于一个系统中，但它们本质上包含的是同样的信息。

图 5-1 显示了一个简单但可说明问题的例子，它由两个类组成，每个类表示一种盒子。一个 Rectangle 由两个点定义，分别表示它的左下角和右上角。一个 Window 由一个中心点、一个宽度和一个高度所定义。这些对象有着截然不同的性能特征，但包含着相同的逻辑信息。

```
// rectangle.h
#ifndef INCLUDED_RECTANGLE
#define INCLUDED_RECTANGLE

class Rectangle {
    // ...
  public:
    Rectangle(int x1,
              int y1,
              int x2,
              int y2);

    // ...
    int lowerLeftX() const;
    // ...
};

#endif
```

```
// window.h
#ifndef INCLUDED_WINDOW
#define INCLUDED_WINDOW

class Window {
    // ...
  public:
    Window(int xCenter,
           int yCenter,
           int width,
           int height);

    // ...
    int width() const;
    // ...
};

#endif
```

图 5-1 一个盒子的两种表示法

这些对象中的每一个都将被用于帮助实施一个图形终端上的超大型交互设计；绘制速度是关键。因为性能原因，我们甚至不考虑使用虚函数，并且将大部分的函数都声明为内联。

原 则

允许两个组件通过#include 指令彼此“知道”隐含了循环物理依赖。

结果，客户程序将偶尔需要能够在这两类盒子之间转换，或许是为了获得另一类的性能特征。这是好的设计可能有时候开始变糟的一种方式。

考虑图 5-2 中前面的“解决方案”集合。我们已经给每一个类加了一个构造函数，该函数使用对另一个类的 const 引用作为它的惟一参数。我们现在可以传递一个 Window 对象给一个需要 Rectangle 的函数，反之亦然，转换将隐含地执行。你觉得听起来怎么样？

```
// rectangle.h
#ifndef INCLUDED_RECTANGLE
#define INCLUDED_RECTANGLE

#ifndef INCLUDED_WINDOW
#include "window.h"
#endif

class Rectangle {
    // ...
  public:
    // ...
    Rectangle(const Window& w);
    // ...
};

inline
Rectangle::Rectangle(const Window& w)
{
    // ...
}

// ...


#endif
```

```
// window.h
#ifndef INCLUDED_WINDOW
#define INCLUDED_WINDOW

#ifndef INCLUDED_RECTANGLE
#include "rectangle.h"
#endif

class Window
    // ...
  public:
    // ...
    Window(const Rectangle& r);
    // ...
};

// ...


inline
Window::Window(const Rectangle& r)
{
    // ...
}

#endif
```

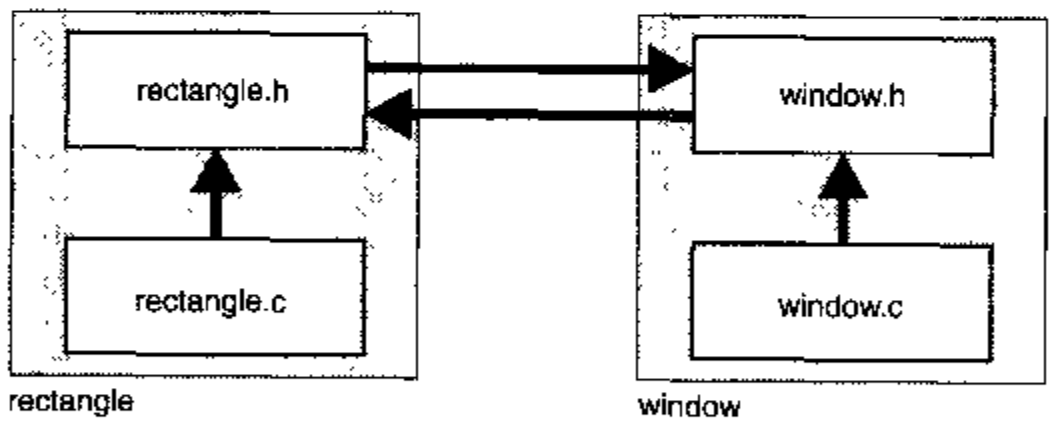


图 5-2　两个互相依赖的组件

如果你觉得这主意听起来不错，你并不是惟一这么想的人。但是它不是一个好的解决办法。首先，任何可能实现的速度优势都可能丧失，因为在进入一个函数时必须构造另一个类型的一个临时对象。由于转换是隐含和自动的，你的客户甚至可能没有意识到这个额外的临时对象的创建（并且会因为你的“缓慢的”类而责备你）。

更加重要的是，我们已经在这两个先前是独立的组件的头文件之间引入了一种循环物理依赖。这两个组件中的每一个现在都必须“知道”对方。如果没有另一个组件就不可能编译、连接、测试或使用两者中的任何一个组件。大多数客户不会关心这些类之间的性能特性方面

的细微差别，他们会选择使用其中的一个，但很少同时使用两个。这种不能层次化的增强却迫使他们两个都要使用。

我们可以把预处理器#include 指令从.h 文件移到.c 文件（如图 5-3 所示），但是这样做并不能消除物理耦合。两个组件在编译时仍然彼此依赖，并且每一个都在连接时潜在地依赖对方。我们需要做更激进一点的事情。

```
// rectangle.h
#ifndef INCLUDED_RECTANGLE
#define INCLUDED_RECTANGLE

class Window;

class Rectangle {
    // ...
  public:
    // ...
    Rectangle(const Window& w);
    // ...
};

#endif
```

```
// window.h
#ifndef INCLUDED_WINDOW
#define INCLUDED_WINDOW

class Rectangle;

class Window {
    // ...
  public:
    // ...
    Window(const Rectangle& r);
    // ...
};

#endif
```

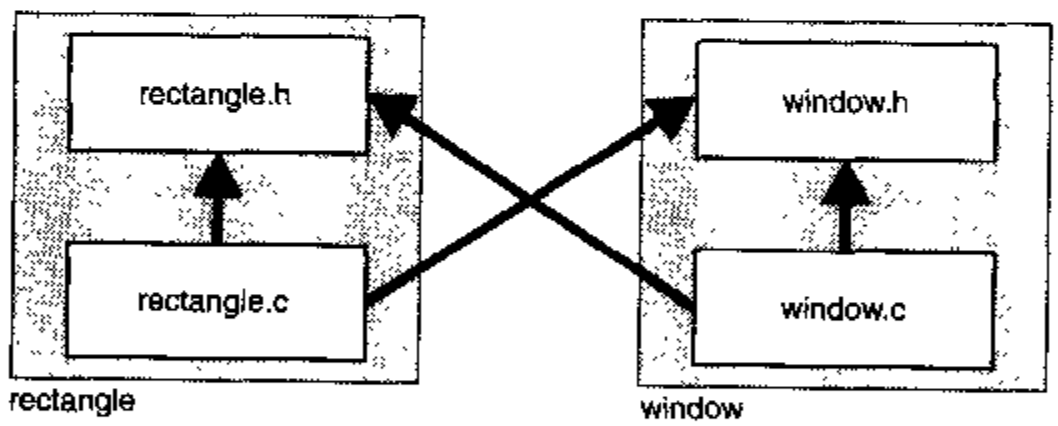


图 5-3　两个组件仍然互相依赖

定义：如果一个子系统可编译，并且单个组件（包括.c 文件）的包含指令隐含的依赖图是非循环的，则称这个子系统是可层次化的。

假设一个子系统由遵守了第 2 章和第 3 章所介绍的所有主要设计规则的组件构成。我们可以使用上述的可层次化的候选定义来帮助我们避免会导致组件变得物理耦合的增强。我们必须以某种方式找到一种方法来允许客户程序在 rectangle（长方形）和 window（窗口）之间转换，而不要求每个组件都包含对方。

5.1.2 便利方法

开发人员在使一个系统可用的过程中，常常会试图创作一些在结构上不可靠的设计。为了说明这个主题，我们引入了第二个例子——一个图形 shape 编辑器，它的设计被抽象地描述在图 5-4 中。类 Shape 是抽象的，它定义了一个协议，要求所有的具体 shape 都必须实现。每个 shape 都有一个位置坐标，我们假定现在必须尽可能快地操纵它（即，通过内联函数）。因为 Shape 类的一些功能已经实现了，所以 Shape 不仅可用来定义一个公共的接口，而且可用来分解实现的共有部分[①]。

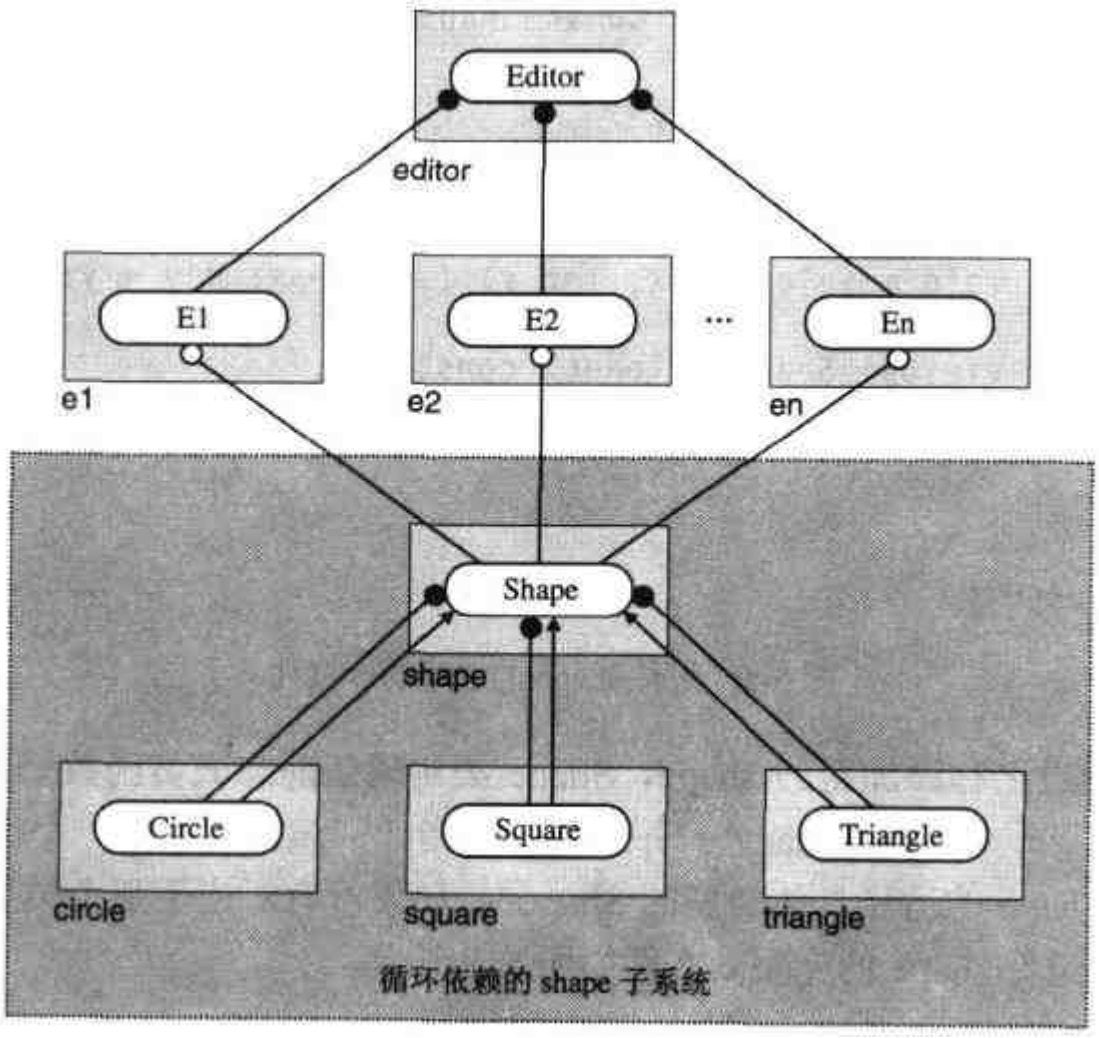


图 5-4 一个图形 shape 编辑器的没有层次化的设计

Shape 类能够潜在地定义大量的纯虚函数。shape 组件头文件的一个简略表示展示在图 5-5 中。Shape 类的客户需要能够建立真实的 shape，但是它们不需要直接与派生类接口交互。为了把 Shape 的客户与派生自 Shape 的具体类绝缘，创建特定种类 Shape 的能力被直接合并进了 Shape 的接口。

① 6.4.1 节描述了如果放宽对 moveTo 函数的速度要求，我们如何能够减少 Shape 接口的消费者与提供者之间的编译时耦合。

```
// shape.h
#ifndef INCLUDED_SHAPE
#define INCLUDED_SHAPE

class Screen;

class Shape {
    int d_xCoord;
    int d_yCoord;

  protected:
    Shape(int x, int y);
    Shape(const Shape& shape);
    Shape& operator=(const Shape& shape);

  public:
    static Shape *create(const char *typeName);
    virtual ~Shape();
    // ...
    void moveTo(int x, int y) { d_x = x; d_y = y; }
    // ...
    virtual Shape *clone() const = 0;
    virtual void draw(Screen *s) const = 0;
    // ...
};

#endif
```

图 5-5　组件 shape 的省略的.h 文件

为了更容易通过名称添加新的 shape，Shape 类实现了静态成员函数 create。该方法接受 Shape 的类型名称（一个 const char *）并返回一个指针，该指针指向一个动态分配的、重新构造的、派生自 Shape 的适当具体类型的 Shape[①]。如果没有对应于那个类型名称的 shape 存在，函数返回 0。组件 shape 的完整.c 文件如图 5-6 所示。

```
// shape.c
#include "shape.h"
#include "circle.h"
#include "square.h"
#include "triangle.h"
#include "screen.h"
#include "string.h"    // strcmp()
```

① 返回一个指向动态分配对象的指针容易产生错误，因为这样会把重新分配的责任留给客户。如果未能捕获一个异常可能很容易导致内存泄漏。句柄类（在 6.5.3 节讨论）可以用来减少内存泄漏的可能性。

```
Shape::Shape(int x, int y)
: d_xCoord(x)
, d_yCoord(y)
{}

Shape::Shape(const Shape& s)
: d_xCoord(s.d_xCoord)
, d_yCoord(s.d_yCoord)
{}

Shape& Shape::operator=(const shape& s)
{
    d_xCoord = s.d_xCoord;
    d_yCoord = s.d_yCoord;
    return *this;
}

Shape::~Shape() {}

Shape *Shape::create(const char *s)
{
    if (0 == strcmp(s, "Circle")) {
        return new Circle(x, y, 1);                    // unit radius
    }
    else if (0 == strcmp(s, "Square")) {
        return new Square(x, y, 1);                    // unit side
    }
    else if (0 == strcmp(s, "Triangle")) {
        return new Triangle(x, y, 1, 1, 1);            // unit side
    }
    else {
        return 0;                                      // unknown shape
    }
}
```

图 5-6　组件 shape 的完整的.c 文件

Editor 类本身在许多单独用在 Editor 的实现中的顾客类型（E1、…、En）之上分层。这些类型中的每一个都在其接口中使用了 Shape，以便执行不同的有关 shape 的抽象操作（例如：moveTo、scale、draw 等等）。只有一个实现组件 e1（实现加法命令）需要能够从一个类型名称创建一个 shape。其余的组件可以利用 Shape 的虚函数来访问一个特定 Shape 的功能，并且不需要直接依赖任何具体的 Shape。这听起来合理吗？

虽然这个设计从可用性的角度来看似乎是吸引人的，但它有一个设计缺陷，使得它维护起来比必要的代价会昂贵许多。Shape 的 create 成员函数使用了派生自 Shape 的每一个类的一个构造函数，这就在 Shape 和所有派生自 Shape 的类之间强加了一种相互的依赖关系。因此不可能独立于其他所有的 Shape 来测试一种特定的 Shape，从而显著地加大了在增量式测试过

程中所需的连接时间和磁盘空间。这个 shape 子系统在其他方面是水平的因而是高度可重用的，但它变成了一个要么全有要么全无的待解问题。

给这个子系统增加一种新的 shape 需要修改 Shape 基类，这可能导致其他独立派生类固有的功能出现错误。由于让一个基类“知道”它的派生类而导致的高度耦合，隐含了相当大的维护开销的增加和相当大的灵活性和重用方面的损失。

当我们考虑到只有组件 e1 需要创建每个 Editor 的具体 shape，因而只有 e1 需要依赖所有单个的具体 shape 组件时，维护方面的劣势就进一步加剧了。组件 e2、e3、…、en 只是通过抽象基类 Shape 的虚函数使用这些 shape。如果我们可以假设每个 shape 的功能都正在正确地工作，那么我们只需要测试出每一个编辑器子系统组件都正在正确地与 Shape 协议交互。可能会有几十个甚至几百个不同种类的 shape，始终用每一个类型的 shape 去测试每一个编辑器子系统组件，既无必要也不实际。但是，因为 shape 子系统中存在的耦合，无论何时想要测试任何一个编辑器实现组件，我们都被迫要连接所有的 shape。

为了提高这个系统的可维护性，我们必须找到一种方法来重新打包 shape 子系统，使它变成非循环的，从而可层次化。

5.1.3 本质的相互依赖

对象的相互连接网络给软件系统体系结构设计者们提出了一种工程挑战。这种高度的内在耦合（尤其是在接口中的）使得很难明显而直观地实现层次化。在这个最后的介绍性的例子中，我们将分析实现一个存在于最基本的对象网络中的图的困难。

原 则

相关抽象接口中的内在耦合使它们更抗拒层次分解。

考虑图 5-7 中的图。该图由节点和边的集合构成。这个图内的节点由有向边连接。通常，图中的边会形成循环[①]。每个节点由一些数据和一些信息（有关节点如何组织进图的信息）组成。在这个例子中节点的数据只是一个名称。连通性被简单地表示为一个边（从某个节点开始或到该节点为止的边）的列表。

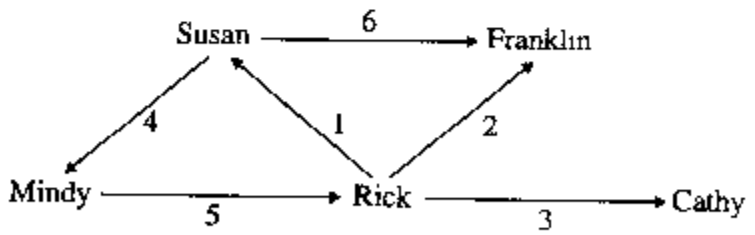


图 5-7 由节点和边组成的简单图

图 5-8 描述了与 node 组件有关的最小的功能。给定一个 Node，可能要请求它的名称，找出连接到它的边的数量，然后通过提供 0 到 *N*–1 之间的整数下标在这些边上进行迭代（这里

① 注意：这些是实例之间的循环，不是类之间的循环。

N 是由 Node::numEdges()返回的当前值）[1]。

```
// node.h
#ifndef INCLUDED_NODE
#define INCLUDED_NODE

class Edge;

class Node {
    // ...
    Node(const Node&);                          // not implemented
    Node& operator=(const Node&);               // not implemented

  public:
    Node(const char *name);
    ~Node();
    const char *name() const;
    int numEdges() const;
    Edge& edge(int index) const;
};

#endif
```

图 5-8　组件 node 的公共接口

图 5-9 描述了与 edge 组件有关的最小功能。在这个系统中，一个 Edge 用于连接节点。像节点一样，边也同时包含局部的和网络相关的功能。在这个例子中，与 Edge 有关的、独立于网络的信息只是它的权值，并且连通性信息也只是 Edge 连接的两个 Node 对象。

```
// edge.h
#ifndef INCLUDED_EDGE
#define INCLUDED_EDGE

class Node;

class Edge {
    // ...
    Edge(const Edge&);                          // not implemented
    Edge& operator=(const Edge&);               // not implemented

  public:
    Edge(Node *from, Node *to, double weight);
    ~Edge();
    Node& from() const;
    Node& to() const;
    double weight() const;
};

#endif
```

图 5-9　一个 edge 组件的公共接口

① 为迭代提供一个整数下标暗示着底层实现有可能是一个某种数组而不是边的连接链表。一个笨拙的链表实现会导致迭代过程中的二次运行时性能。

最初，我们面对的是图 5-10 中展示的毫无吸引力的设计。Node 在它的接口中使用了 Edge，反之亦然。按照现在的情况，似乎类 Node 和类 Edge 一定是相互依赖的——否则客户怎么可能遍历这个图呢？另外，还存在谁拥有这些对象的内存以及谁有权产生和取消 Node 和/或 Edge 的实例的问题。

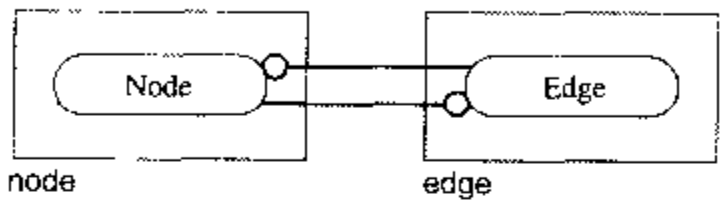


图 5-10 循环依赖的 node 和 edge 组件

回忆一下 3.6 节的介绍，友元关系本身不会引入物理依赖，但是为了保持封装，它可能间接地导致物理耦合的出现。为了避免出现封装缺口和由于远距离友元关系导致的缺乏模块性，可能有必要将若干可层次化的类归入一个单一的组件内（正如在 5.9 节的结尾所解释的那样）。这种耦合的常见例子实际上可以在每一个提供迭代器的容器组件内看到。迭代器永远是容器的一个友元，因而定义在相同的组件内。

以上介绍的只是实践中通常会出现的循环耦合种类的一些例子。本章的其余部分致力于论述各种技术和转换，以理清可能在其他方面挑战非循环物理实现的设计。

5.2 升级

现在让我们返回到包含两个循环依赖组件（rectangle 和 window）的例子（显示在图 5-1 中）。假设不再让 rectangle 和 window 彼此“知道”，而是由我们随机地决定 rectangle 比 window 更基本。我们可以把两个转换都移进 Window 类中。现在 Window “使用” Rectangle，但反之则不然，如图 5-11 所示。

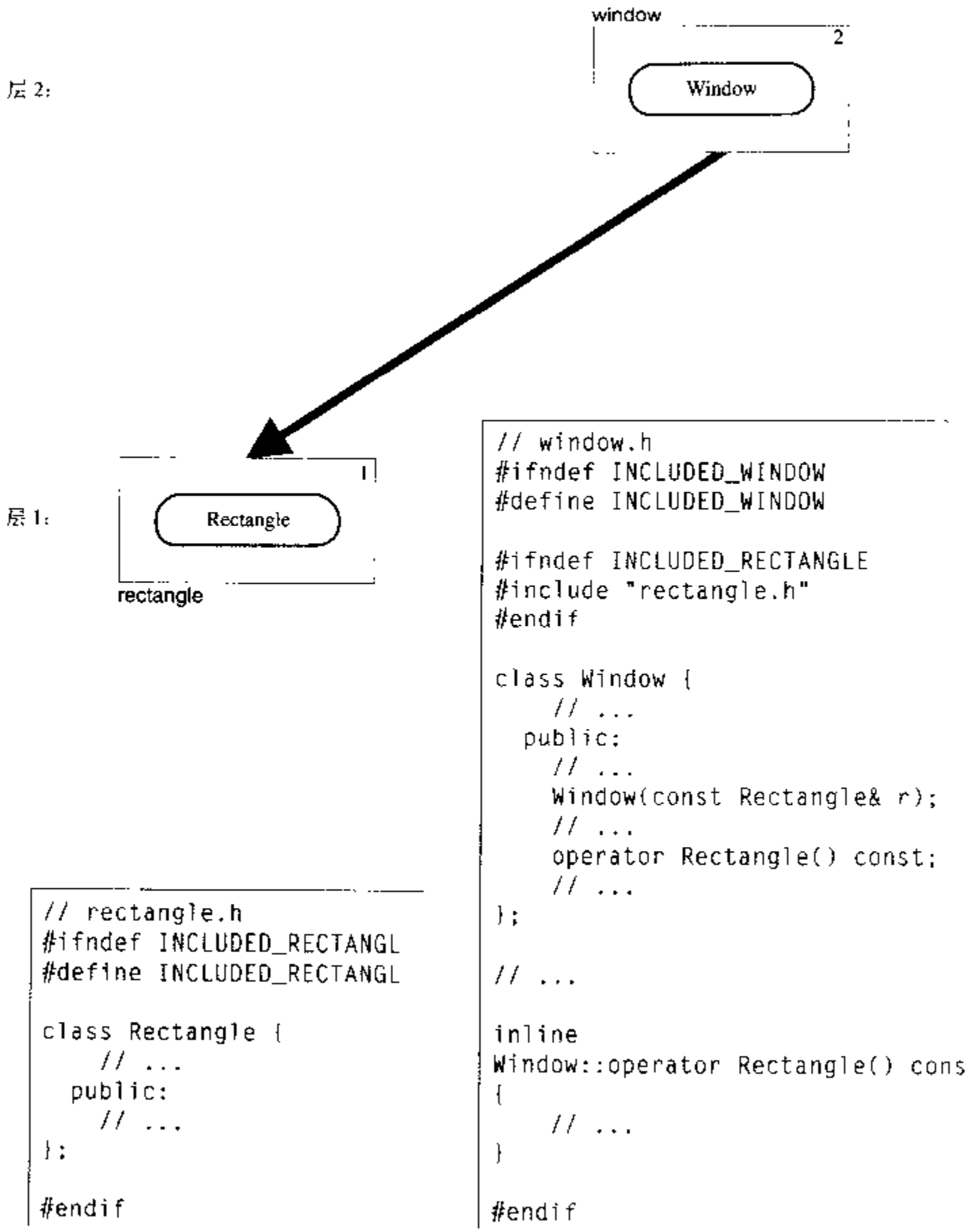
这种解决办法要求我们稍微改变一下我们的观点，因为 Rectangle 和 Window 类不再是对称的了。Rectangle 居于第一层，而 Window 现在定义在第二层。如果我们需要任何旧的盒子，我们可以重用 Rectangle 而不用担心 Window 或类之间的转换。然而，如果我们需要一个 Window，我们仍将不得不使用 Rectangle。

> **定义**：如果组件 y 处在比组件 x 更高的层次上，并且 y 在物理上依赖 x，则称组件 y 支配（**dominate**）组件 x。

支配是一种组件之间的属性，它和一个单一派生对象内的虚基类之间的同名属性大致相同[1]。我们现在介绍了组件之间的支配概念，并提到图 5-11 中展示的一个例子，其中组件

① **ellis**，10.1.1 节，204～205 页。

window 支配组件 rectangle。在后面的小节中我们会提到“支配”这个定义。




```
// window.h
#ifndef INCLUDED_WINDOW
#define INCLUDED_WINDOW

#ifndef INCLUDED_RECTANGLE
#include "rectangle.h"
#endif

class Window {
    // ...
  public:
    // ...
    Window(const Rectangle& r);
    // ...
    operator Rectangle() const;
    // ...
};

// ...

inline
Window::operator Rectangle() const
{
    // ...
}

#endif
```

```
// rectangle.h
#ifndef INCLUDED_RECTANGL
#define INCLUDED_RECTANGL

class Rectangle {
    // ...
  public:
    // ...
};

#endif
```

图 5-11　window 支配 rectangle

如图 5-12 所示，组件 u 支配组件 r 和组件 s。虽然组件 v 处在一个比组件 r 或组件 s 都更高的层次上，它却只支配组件 t。组件 w 支配所有的五个组件：r、s、t、u 和 v。

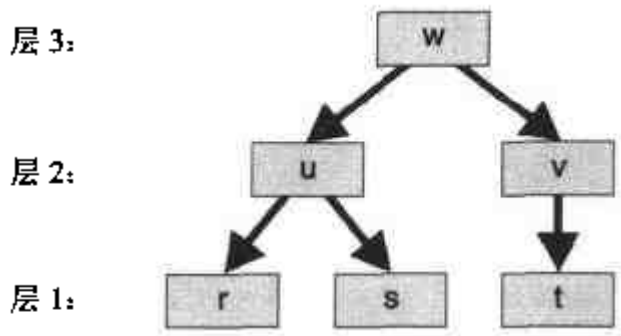


图 5-12　组件支配性质的阐释

支配的重要性在于它能够提供简单的层次号之外的额外信息。例如，增加一个高层次组件对低层次组件（如在图 5-12 中，u 对 t）的依赖决不会引入一个循环依赖或改变层次号[如图 5-13（a）所示]。

在同一层的两个组件之间（如在图 5-12 中，v 对 u）增加一个依赖也不会引入一个循环，但是会影响层次号，如图 5-13（b）所示。最后，甚至也有可能增加一个从低层次组件到高层次组件的依赖（如在图 5-12 中，t 对 u，没有引入一个循环依赖）。当且仅当组件 u 不是已经支配组件 t 时，才可能增加依赖而不引入循环。在这里，组件 u 没有支配组件 t，增加 t 对 u 的依赖的结果显示在图 5-13（c）中。

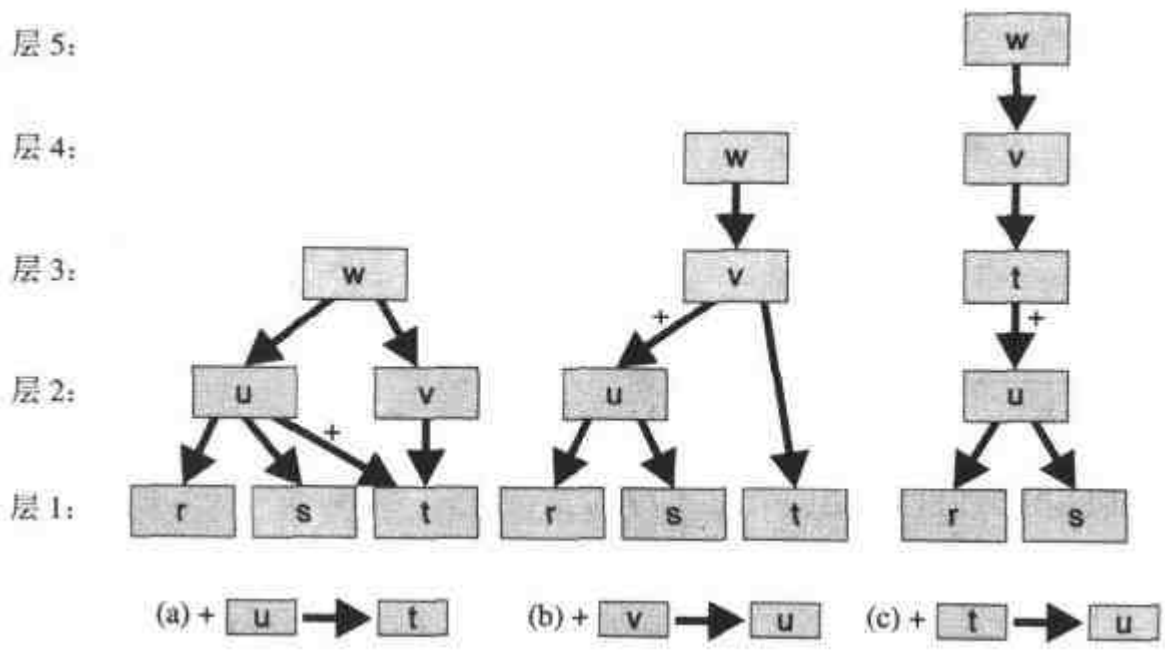
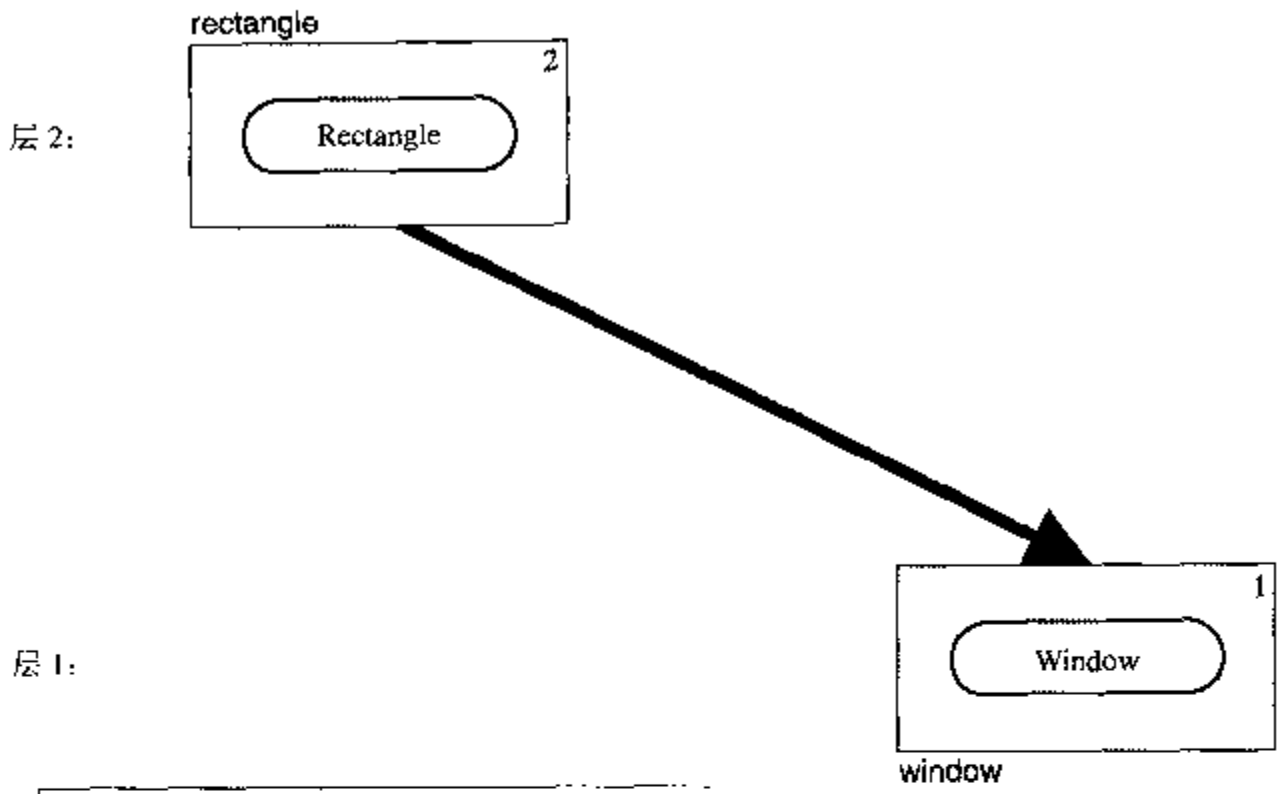


图 5-13　增加一个依赖可能引起层次号的改变

当然我们可能已经走到另一条道上去了，并使 Window 成了基本的对象。在那种情形下，rectangle 知道 window，但反之则不然。这种情形在图 5-14 中描述。请注意在这个例子中我们已经选择了将#include "window.h"指令移到 rectangle.c 文件中，这就意味着转换例程将不是内联的。




```
// rectangle.h
#ifndef INCLUDED_RECTANGLE
#define INCLUDED_RECTANGLE

class Window;

class Rectangle {
    // ...
  public:
    // ...
    Rectangle(const Window& w);
    // ...
    operator Window() const;
    // ...
};

#endif
```

```
// window.h
#ifndef INCLUDED_WINDOW
#define INCLUDED_WINDOW

class Window {
    // ...
  public:
    // ...
};

#endif
```

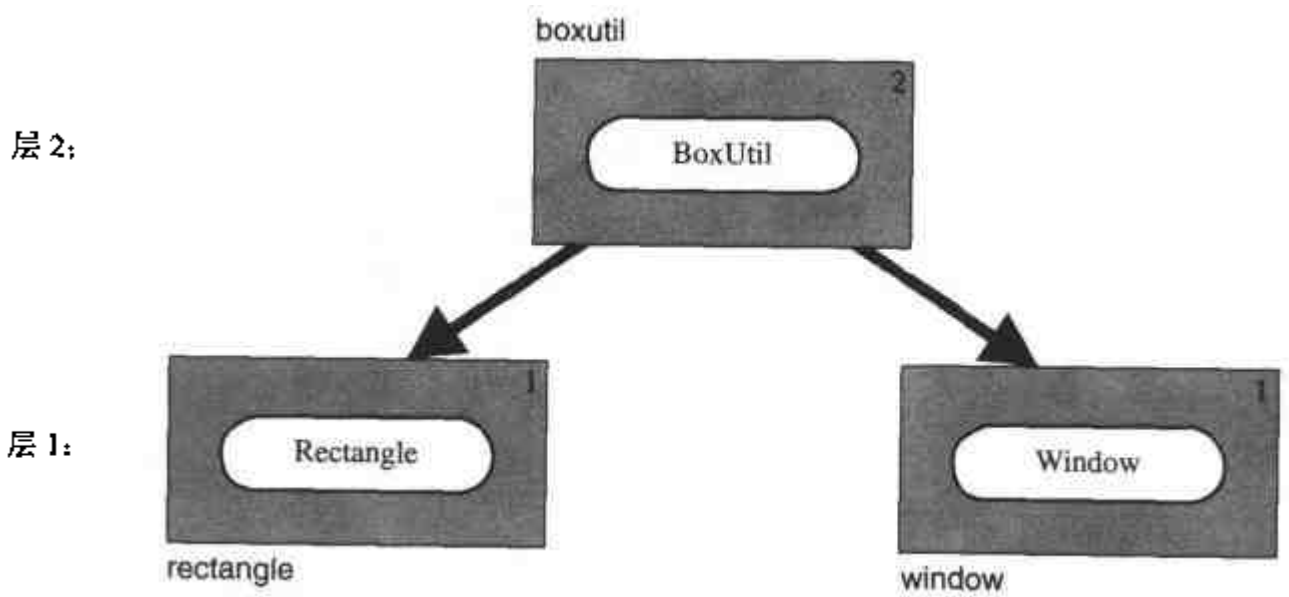
图 5-14　rectangle 支配 window

两种解决方法都意味着只有一个组件的使用能够独立于另一个组件。两种解决方案都是基于最初的循环依赖设计改进的，但我们还能做得更好一些。许多使用这些组件的客户都只需要其中的一个而不是同时需要两个。在那些确实需要使用两个组件的客户中，只有少许会需要在它们之间转换。为了使独立可重用性最大化，我们可以通过将引起循环的功能移到一个更高的层次上来避免让一个组件支配另一个组件——这种技术在本书中称为升级（escalation）。

原　则

如果同层次的组件是循环依赖的，那么就有可能把互相依赖的功能从每一个组件升级为一个潜在的新的更高层次组件（依赖于每一个初始的组件）的静态成员。

在公司里，如果两个雇员不能解决一个争论，通常的做法是将问题提交到更高层。在对象为了“支配”而竞争的情况下，同样的解决方法通常也是有效的。我们可以创建一个名为 BoxUtil 的工具类，它既知道 Rectangle 类也知道 Window 类，然后把这个类的定义放在一个完全独立的组件中，如图 5-15 所示。




```
// boxutil.h
#ifndef INCLUDED_BOXUTIL
#define INCLUDED_BOXUTIL

class Rectangle;
class Window;

struct BoxUtil {
    static Window toWindow(const Rectangle& r);
    static Rectangle toRectangle(const Window& w);
};

#endif
```

```
// rectangle.h
#ifndef INCLUDED_RECTANGLE
#define INCLUDED_RECTANGLE

class Rectangle {
    // ...
  public:
    // ...
};

#endif
```

```
// window.h
#ifndef INCLUDED_WINDOW
#define INCLUDED_WINDOW

class Window {
    // ...
  public:
    // ...
};

#endif
```

图 5-15 rectangle 和 window 都不支配对方

现在对 Rectangle 类或 Window 类有兴趣的客户可以自由地独立使用任何一个类。如果一个客户碰巧使用了两个类但不需要在它们之间转换，情况也是如此。如果还有其他客户需要转换程序，他们也可以得到。但是要注意，Rectangle 和 Window 之间的转换通常是隐式的，现在则必须显式地执行（要了解更多关于隐式转换的内容请看 9.3.1 节）。

注意，在前面的例子中，定义 BoxUtil 时我们选择了使用关键词 struct 来代替 class，暗示这个类型只是为公共嵌套类型和公共静态成员函数提供一个作用域。在这个约定中，一个 struct 的所有成员都是公共的，因此没有数据成员。虽然创建这样一个类型的一个实例是无意义的，但它不会造成真正的危害。如果我们抑制住将未实现的默认构造函数声明为 private 的冲动，我们就可以减少一些不必要的混乱。

现在让我们再来思考定义在图 5-4 的 shape 层次结构的基类中的静态 create 函数引起的物理耦合。假设我们通过引进一个新的工具类 ShapeUtil（其惟一的用途是创建 shape）将 create 在它的派生类层次之上升级。这个新的类将放在它自己的组件内，并且包含来自初始的 Shape 类的 create 函数，如图 5-16 所示。

```
// shapeutil.h
#ifndef INCLUDED_SHAPEUTIL
#define INCLUDED_SHAPEUTIL

class Shape;

struct ShapeUtil {
    static Shape *create(const char *typeName);
};

#endif
```

图 5-16　新的 shapeutil 组件的头文件

通过增加一个新的组件把 Uses 关系升级到一个更高的层次，我们已经消除了 shape 子系统中所有组件之间的循环依赖。新系统的层次化图如图 5-17 所示。

现在对每一个具体的 shape 都有可能进行隔离的测试。在 shape 组件的测试驱动程序中，通过从 Shape 派生一个具体的“桩（stub）”类，甚至能够对 Shape 类提供的部分实现进行模块化测试。现在每一个具体的 shape 在任何组合中都能独立于其他 shape 而重用。例如，一个系统现在可以重用 circle 和 square，而不必连接 triangle。

现在也有可能测试 E2、…、En 中的每一个，而不必连接每一个具体的 shape。因为这些组件只需要 shape 基类接口，所以，只需在所有可访问的具体 shape 中的一个有代表性的例子上，测试由每一个编辑器组件 e2、…、en 添加的增加值，可以认为这已经足够了。

这个基于最初设计的新设计的好处是减少了耦合，这种减少在扩大重用潜力时，将直接

转换成开发和维护开销的减少。当编辑器中的实现组件的数量，尤其是具体 shape 的数量较少时，也许很难意识到这种设计方法的重要性。真正的好处在于这种新的设计方法在更多的编辑器命令和新的 Shape 加进来时，在按比例扩展方面要比原来的方法好得多。

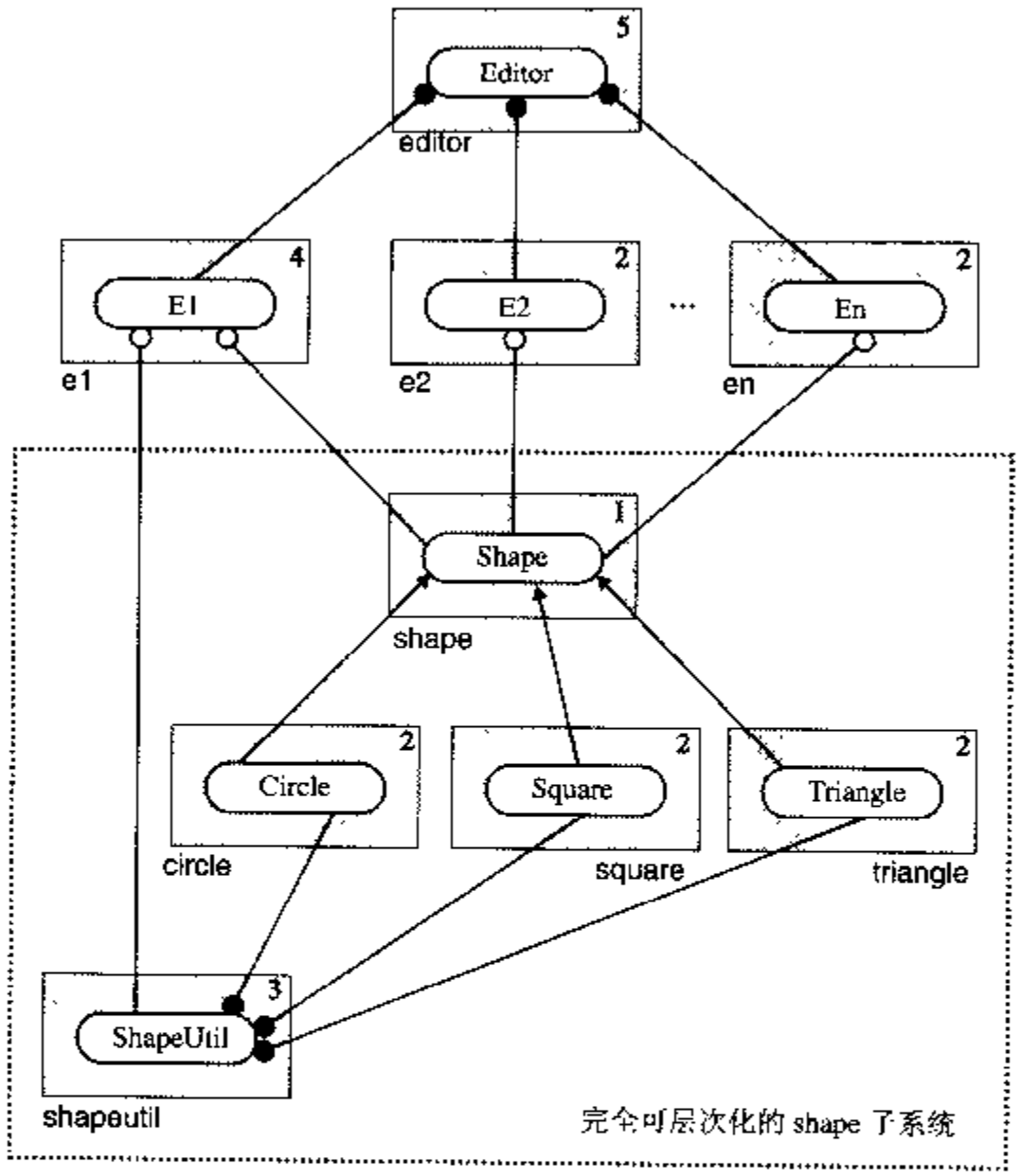


图 5-17　对 Shape Editor 的改进设计

作为对层次化给这个设计所带来改进的客观的、定量的估算，让我们来思考这个编辑器系统的四种不同的形式。在图 5-18（a）中的这个系统的“缩减”版本中，编辑器以及它所影响的 shape 的数量都是较少的（三种 shape 和三个编辑器实现组件）。rectangle 组件左上角的数字指出了为了增量式地测试该组件而必须连接的组件数量。

乍一看，这种新的设计似乎有不必要的复杂性，但事实上它简化了开发人员和客户的工作。甚至包括新设计中的额外组件，与分层次测试这个 shape 子系统有关的耦合（以 CCD 计算）也减少了整整 25%。与增量式测试编辑器子系统有关的耦合降低了 17.4%，以 CCD 计算总的减少了 20.5%。

图 5-18（b）描述了当编辑器子系统变得更大时（30 个实现组件，而不仅仅是 3 个）的

效果。现在编辑器子系统组件耦合的减少量接近 46%，将以 CCD 来计算的整体减少量提到 43.3%。

原　则

在庞大的低层子系统中的循环物理依赖将最大程度地增加维护系统的总开销。

在物理层次结构的较低层次的循环耦合可能会对维护客户程序的开销产生戏剧性的影响。正如在图 5-18（c）中可以看到的，当 shape 层次结构变得较大时（30 个具体类型，而不仅仅是 3 个），新设计的优势，以 CCD 来计算，不仅总计降低了 shape 子系统的超过 90%的耦合，而且降低了编辑器子系统超过 44%的耦合，整体减少将近 85%。当 shape 子系统和编辑器都很大时，整体降低耦合的百分比还会继续上升，如图 5-18（d）所示。

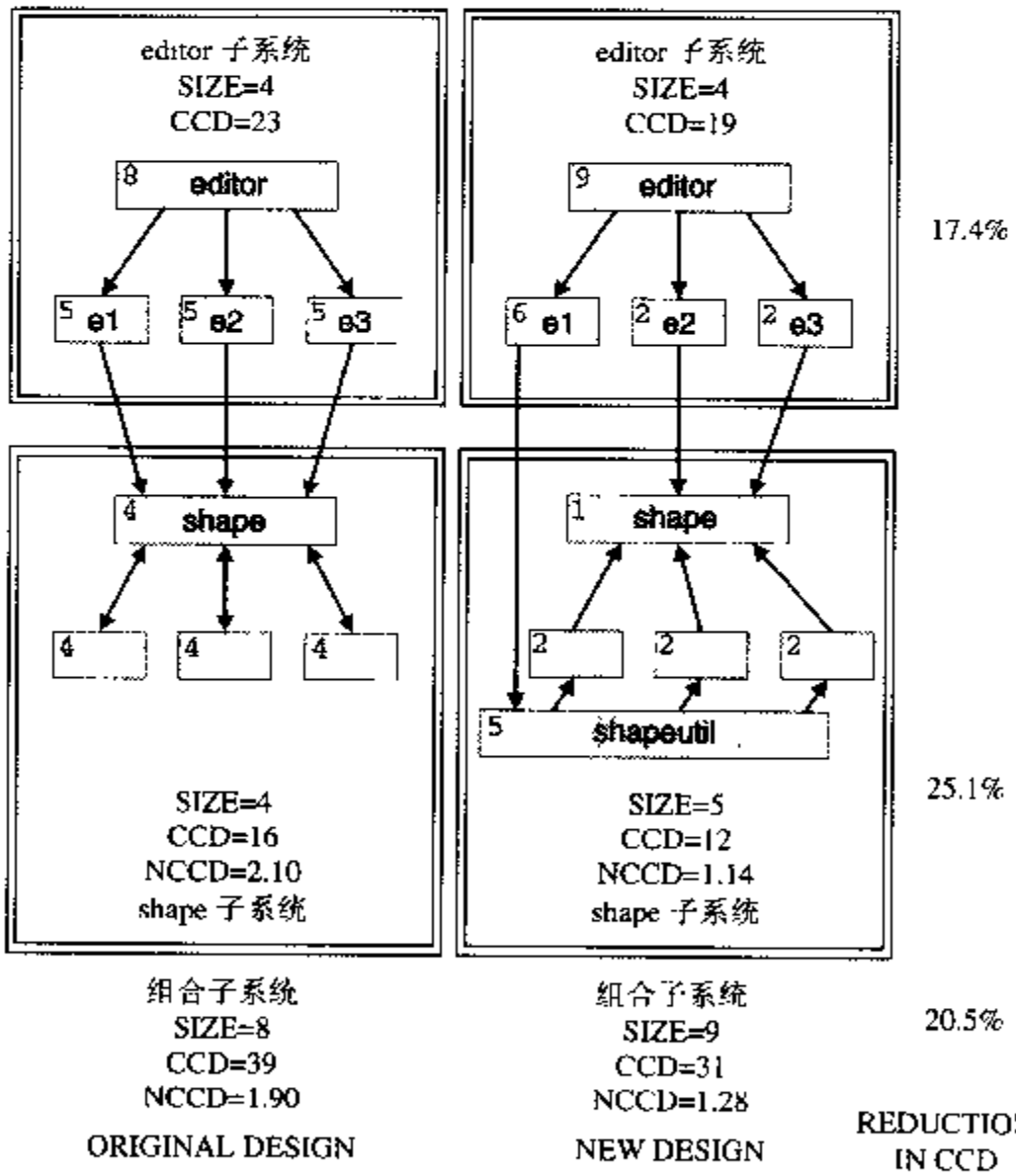


图 5-18（a）　小型 Shape 层次结构（3 个组件），小型 Editor（3 个组件）

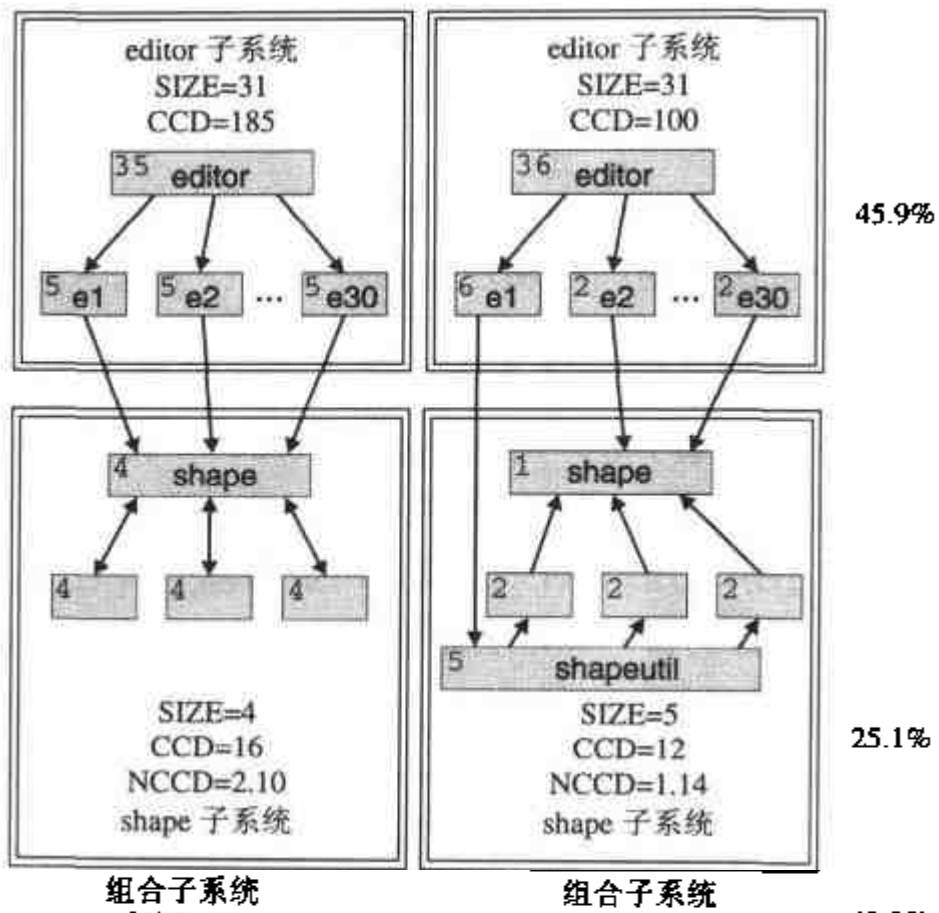


图 5-18（b） 小型 Shape 层次结构（3 个组件），大型 Editor（30 个组件）

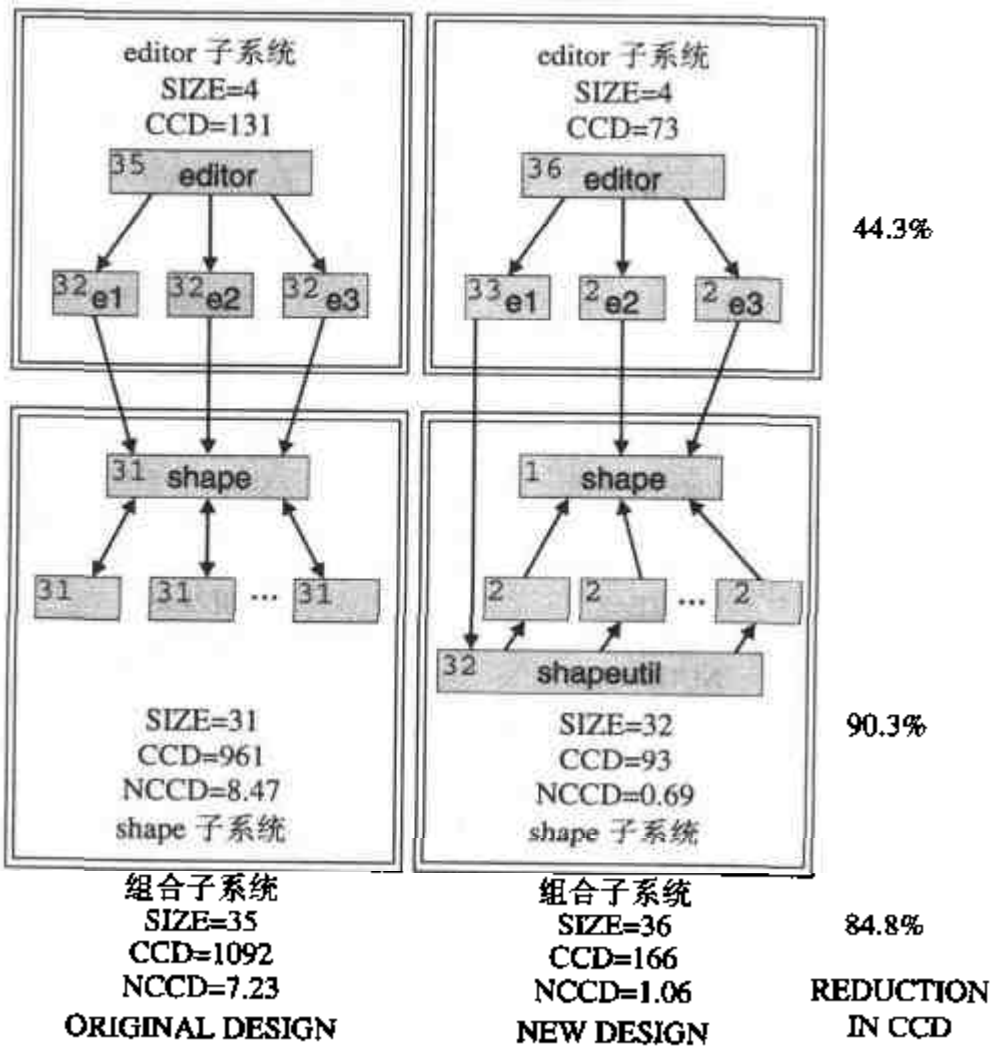


图 5-18（c） 大型 Shape 层次结构（30 个组件），小型 Editor（3 个组件）

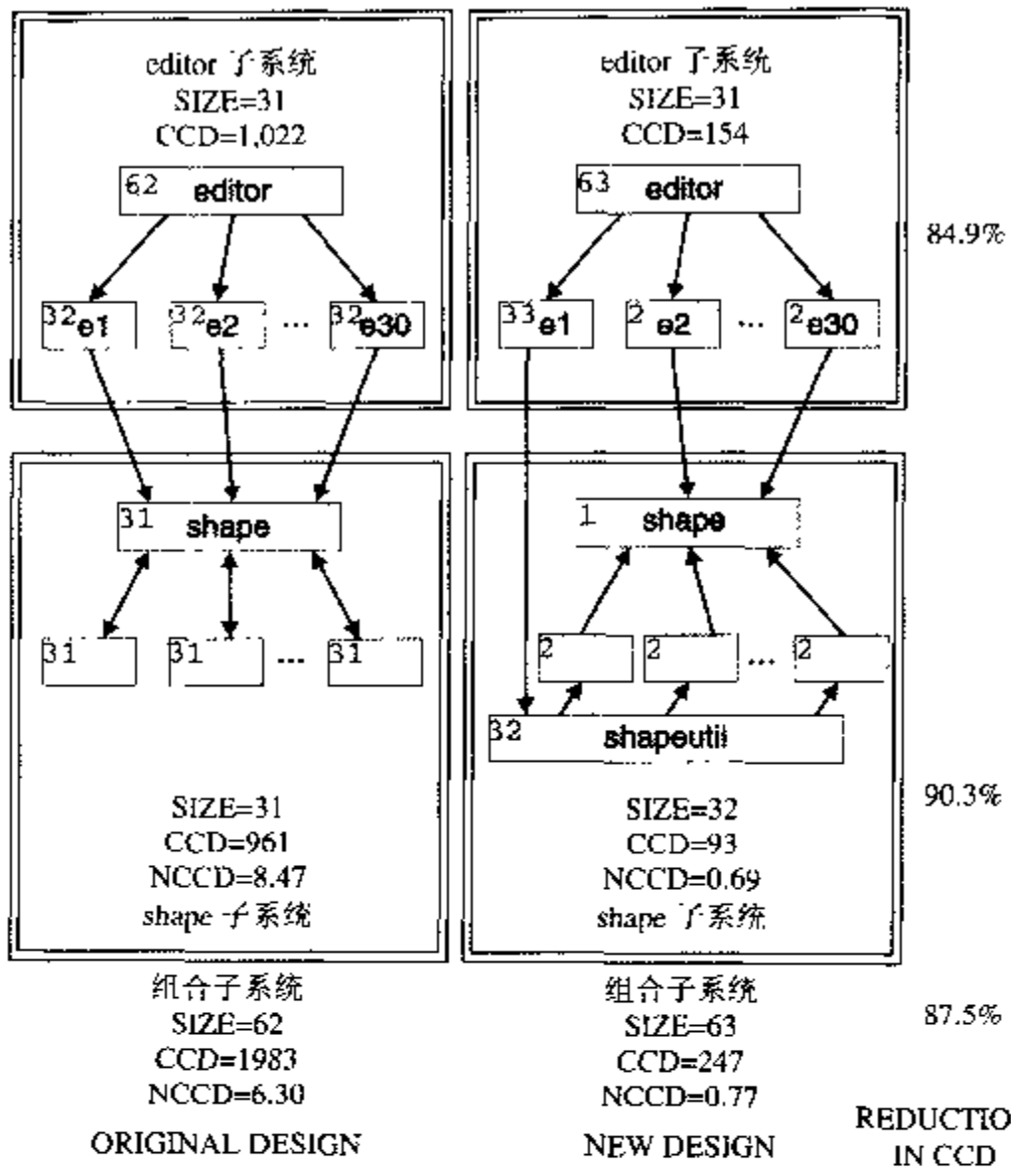


图 5-18（d）　大型 Shape 层次结构（30 个组件），大型 Editor（30 个组件）

从这个分析中得到的重要结论是，与低层次子系统有关的高度耦合可能会戏剧性地增加开发和维护较高层次客户程序和子系统的开销。

总结：可以通过把相互依赖升级到更高的层次将循环依赖转换成受欢迎的向下依赖。通过避免子系统本身内部组件之间不必要的依赖，可以显著降低子系统和它的所有客户程序的维护开销。同时，子系统也可以变得更灵活从而更可重用。这个改进设计的好处对于一个系统的较小版本也许并不显著。

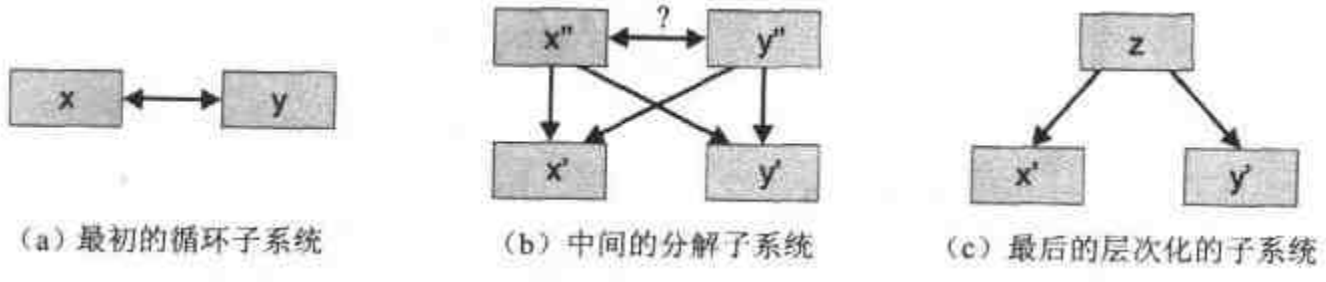
5.3　降级

到现在为止，我们已经努力通过把互相依赖的功能推到物理层次结构的更高层来消除循环依赖。在这一节中，我们要探讨把公用的功能推到物理层次结构的更低层的技术，在那里公共功能可以被共享甚至也许可以被重用。这种把公用的功能移到物理层次结构的更低层的技术在本书中称为**降级（demotion）**。

原　则

如果同一层次的组件是循环依赖的，那么就有可能把互相依赖的功能从每一个组件降到一个潜在的新的较低级（共享）组件中，每一个原来的组件都依赖于这个新组件。

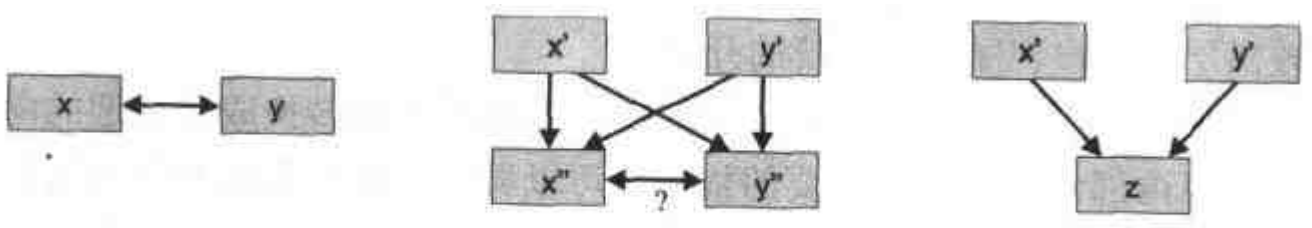
升级和降级是相似的，因为在这两种情况下，都是通过把循环依赖功能移到物理层次结构的另一层的方法来消除组件之间的循环依赖。让我们从分析在一个更通用的升级形式的过程中会发生什么开始。如图 5-19 所示，两个互相依赖的组件（a）被分解成四个组件（b），其中两个也许是相互依赖的而另外两个是独立的。如果有必要避免一个循环依赖，或者如果是内聚的，要降低物理复杂度，这两个较高层次的组件有可能被组合成（c）。



（a）最初的循环子系统　（b）中间的分解子系统　（c）最后的层次化的子系统

图 5-19　使用升级来解开循环依赖

现在，将此过程和常用的降级过程相比较。如图 5-20 所示，两个相互依赖的组件（a）再次被分解成四个组件（b），其中两个组件依赖于另外两个组件，这两个组件也可能是相互依赖的。然后如果有必要避免一个循环依赖，或者是内聚，为了降低物理复杂度，这两个较低层次的组件有可能被合并（c）。



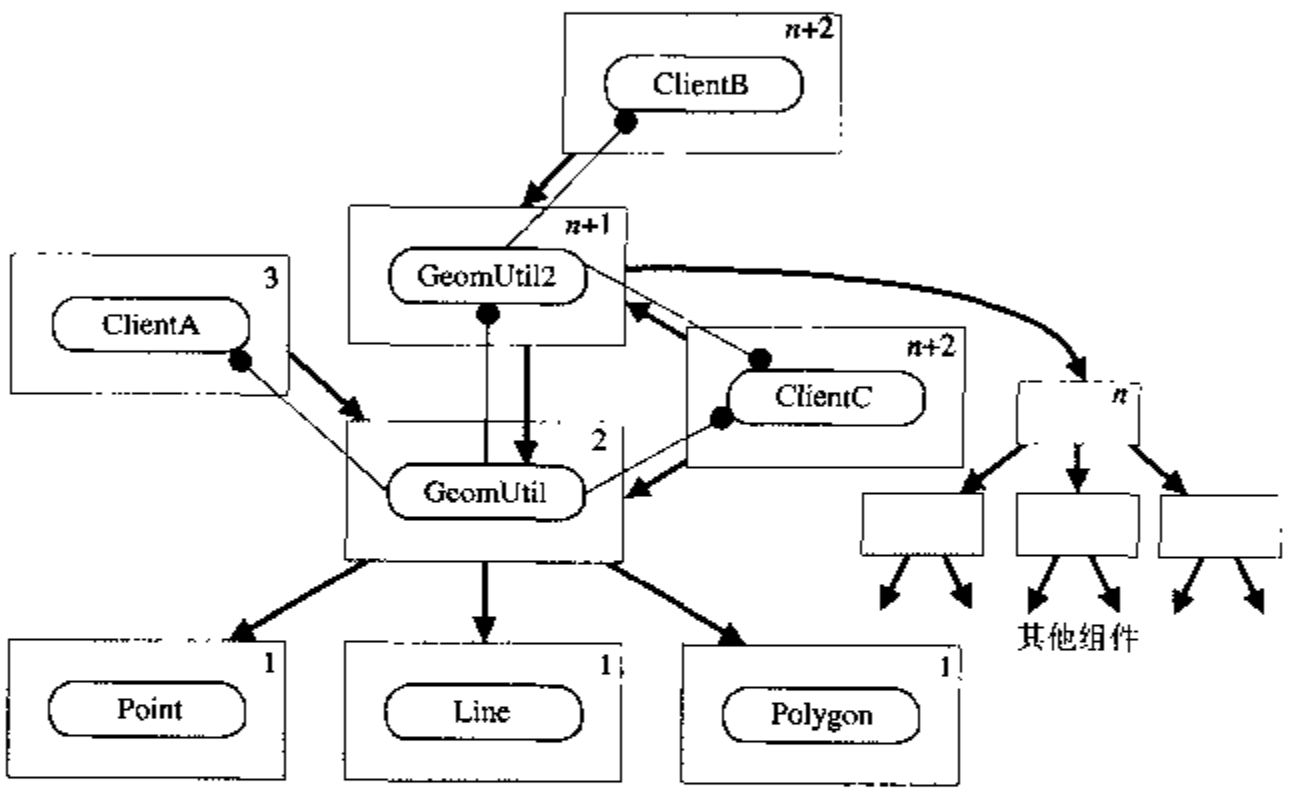
（a）最初的循环子系统　（b）中间的分解子系统　（c）最后的层次化的子系统

图 5-20　使用降级来解开循环依赖

考虑图 5-21 所示的情形，图中有两个几何工具类，GeomUtil 和 GeomUtil2。每一个工具都提供一套对点、线和多边形操作的函数。外部客户直接使用其中一个类或同时使用两个。和 geomutil 不同，geomutil2 是复杂的，它依赖许多其他的组件，甚至在它的接口中暴露了一些新的类型。那些只需要 GeomUtil 提供的基本几何功能的客户程序不需要和 geomutil2 组件连接。

最初这两个组件是可层次化的，geomutil2 依赖 geomutil。但是，不是所有的开发人员都会小心地考虑他们工作中的物理隐含。开发人员有一天会发现，由于不小心的增强，这两个

```
// geomutil2.h
#ifndef INCLUDED_GEOMUTIL2
#define INCLUDED_GEOMUTIL2

class Line;
class Polygon;

struct GeomUtil2 {
    static int crossesSelf(const Polygon& polygon);
    static int doesIntersect(const Line& line1, const Line& line2);
    // ...
};

#endif
```




```
// geomutil.h
#ifndef INCLUDED_GEOMUTIL
#define INCLUDED_GEOMUTIL

class Point;
class Line;
class Polygon;

struct GeomUtil {
    static int isInside(const Polygon& polygon, const Point& point);
    static int areColinear(const Line& line1, const Line& line2);
    static int areParallel(const Line& line1, const Line& line2);
    // ...
};

#endif
```

图 5-21　两个几何工具组件：geomutil 和 geomutil2

几何工具组件已经变得相互依赖了。GeomUtil2::crossesSelf 现在依赖于 GeomUtil::areColinear，而 GeomUtil::isInside 现在依赖于 Geomutil2::doesIntersect。我们应该怎么办呢？

我们有几个选择。首先我们可以对功能重新打包，这样就可以恢复单向依赖，这也许是正确的答案。例如，我们可以把 doesIntersect 移到 GeomUtil，把 isInside 移到 Geomutil2。现在这些组件之间不再有循环依赖了，虽然这些组件的客户程序可能受影响。（一种通用的给组件重新打包的技术在本节结尾部分阐述。）

也可能有这样的情形，由于那些依赖它们的客户程序的要求，两个组件已具有截然不同的特征。在那种情况下，分解出共有的功能，并把它降级到物理层次结构的较低层也许更合适，如图 5-22 所示。就是说，我们可以把 doesIntersect 和 areColinear 函数都移到 GeomUtilCore 中去。

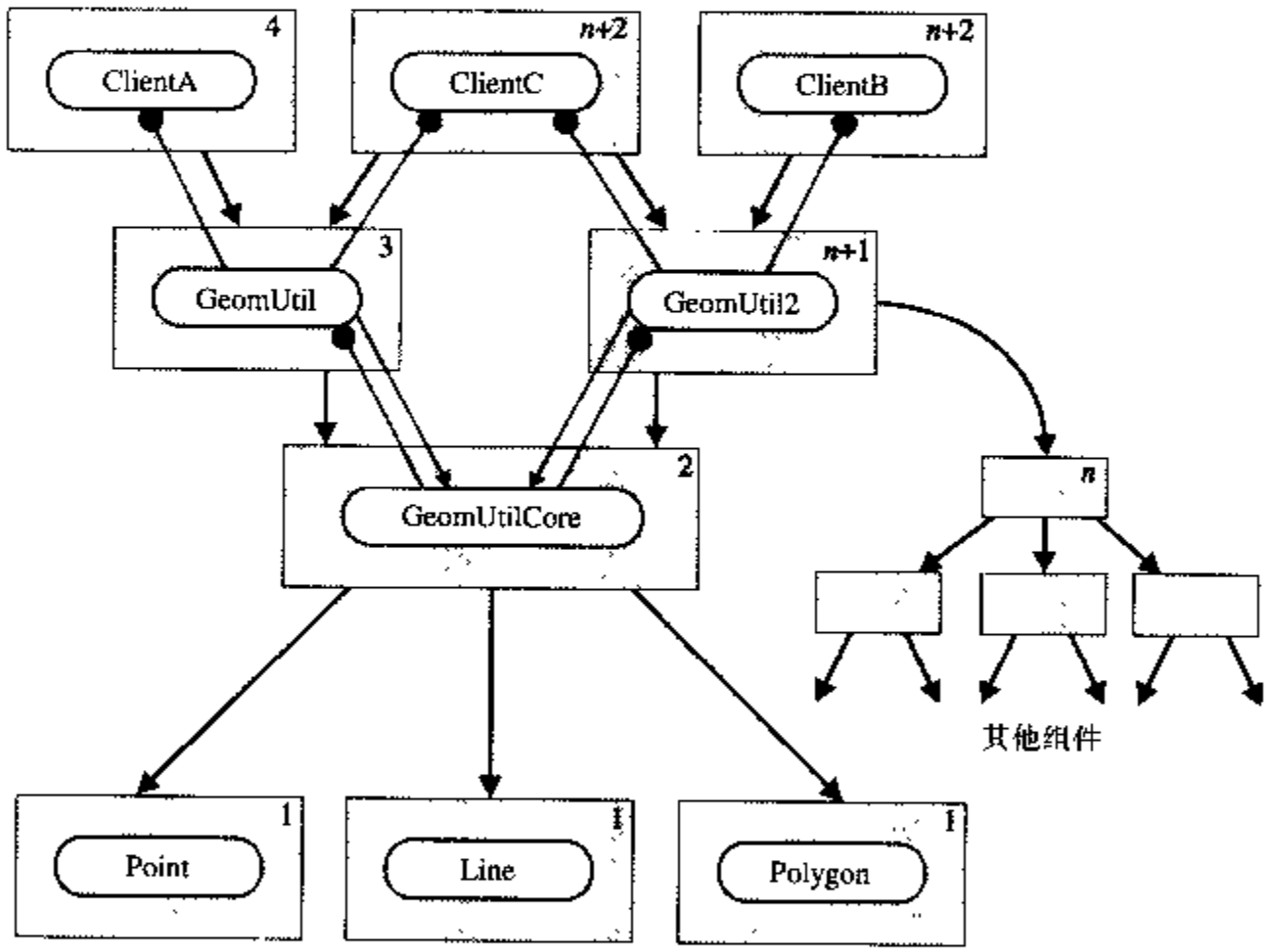
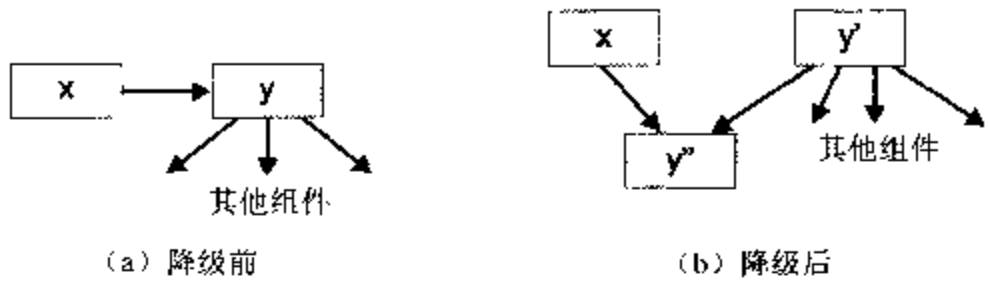


图 5-22 将（工具）类之间的共有功能降级

请再次注意这些工具类，只是在其中声明静态成员函数的作用域——绝对没有准备用它们来创建对象。如果通过变“戏法”让两个原有的工具都公共派生于一个公共核心，那么当一个或多个所使用的工具函数被降级时，原有工具的客户程序将不必修改他们的代码。

降级对于减少某些设计的 CCD（甚至在没有循环依赖时）来说是一种很有用的工具。假设一个 x 组件以高 CCD 只依赖于另一复杂组件 y 的一部分，如图 5-23（a）所示。如果我们能够把 y 的共有部分降级，我们也许能免除 x 的一些由 y 导致的物理依赖，见图 5-23（b）。



（a）降级前　　（b）降级后

图 5-23　使用降级减少 CCD

原　则

将共有代码降级可促成独立重用。

图 5-24 描述了这样一个情形：定义在子系统 A 中的枚举值被用于整个系统，但是子系统 B 是另外独立于子系统 A 的。

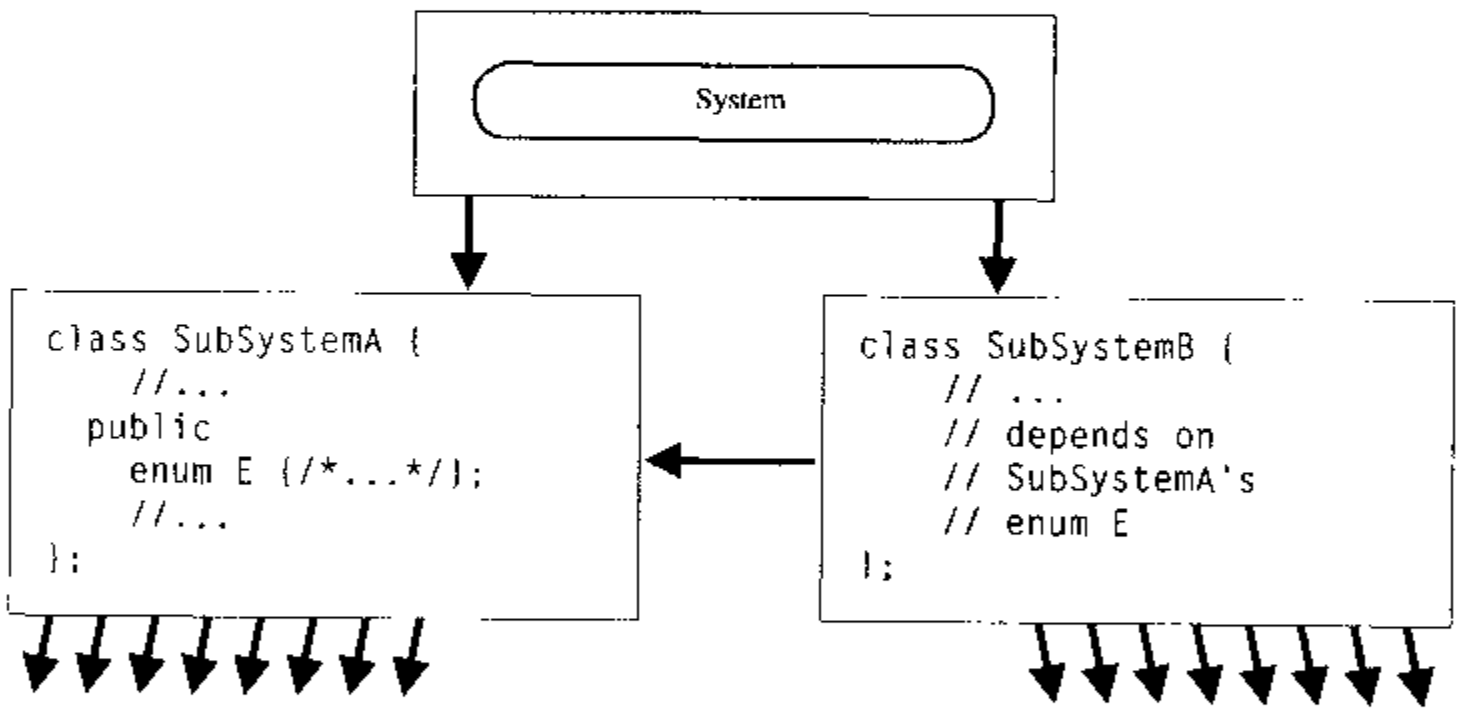


图 5-24　不良分解的系统结构

虽然目前没有子系统 B 对子系统 A 的连接时依赖，但这个事实仅通过检查提取出的包含图是搞不清楚的。包含图指出，最初的结构设计已经允许子系统 B 中的组件任意地依赖子系统 A 中的组件。日常维护将立刻不可避免地引起有更多实质性的、B 对 A 的连接时依赖，这又会影响维护子系统 B 的连接时间开销。

通过一开始就在 SubSystemA 中创建一个单独的类（或 struct）来限制枚举 E 的作用域，并将此作用域受限制的枚举移到一个单独的组件中，我们可以消除子系统 B 对子系统 A 的任何物理依赖。如图 5-25 所示，将枚举 E 降级会减少耦合和简化理解子系统 B 的任务，从而降低维护整个系统的开销。

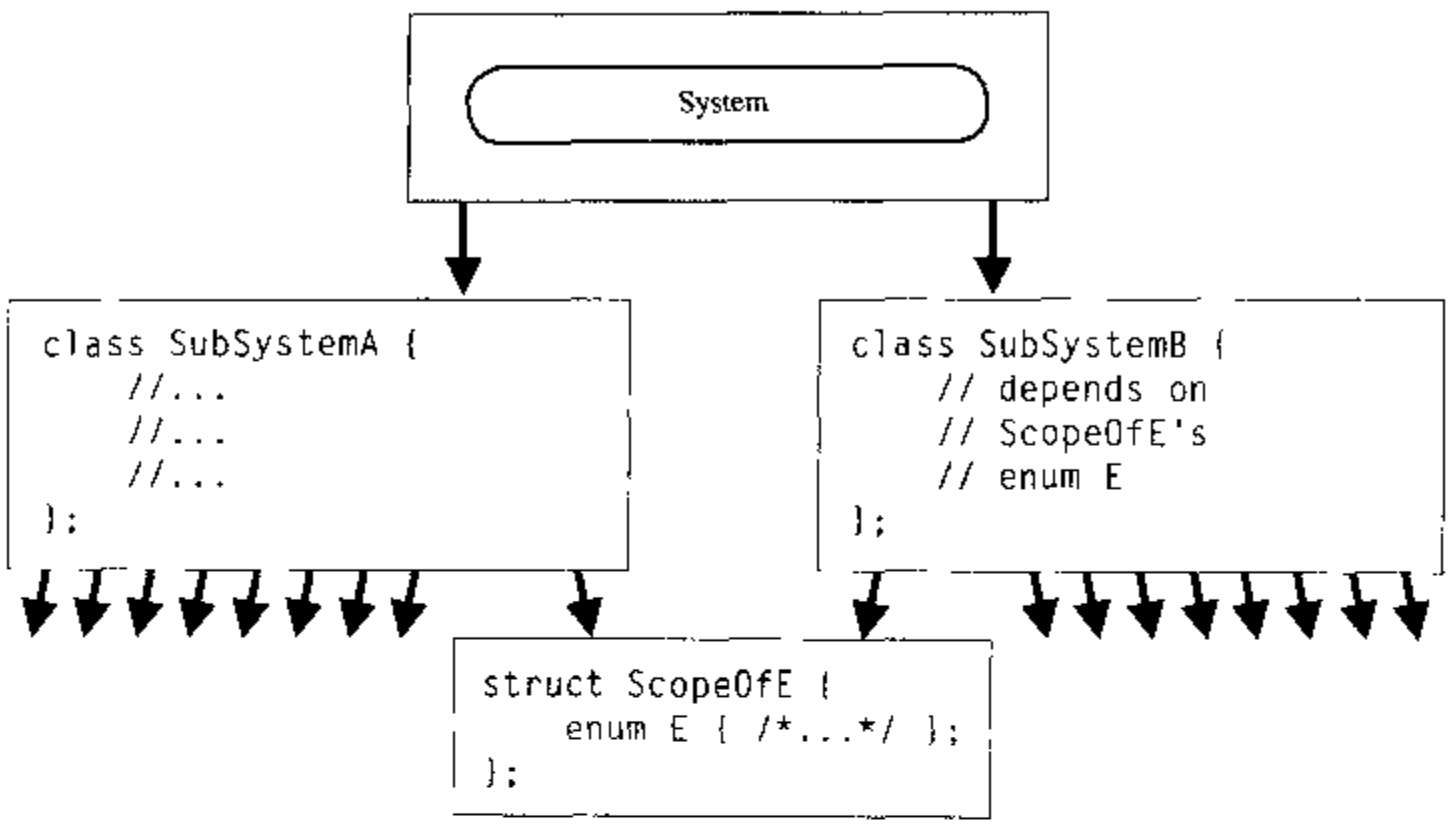


图 5-25 将枚举类型降级后的新系统结构

把单个的枚举置于它自己的类中似乎有点矫枉过正。在某些情形下是，但在这里不是。请注意，将这极少的代码放在它自己的组件中，已经把子系统 B 从不得不拖着整个子系统 A 到处跑的相当沉重的维护负担中解放出来了。

原 则

可以将升级策略与基础设施降级结合起来，以增强独立重用。

另外一个可能会有高 CCD 的常见的体系结构例子可以在这样一个系统中找到：它解析一个文本文件来建立一个运行时数据结构，然后对该数据结构进行操作，以执行一些需要的计算。

图 5-26 所示的体系结构中，在该系统层次结构底部的一个单个子系统中，解析器与运行时数据结构紧密耦合。因此，我们可能期望看到一个如下形式的成员函数：

```
RuntimeDB::Status RuntimeDB::read(const char *fileName);
```

其中 Status 是一个嵌套在类 RuntimeDB 内的枚举类型。大概这个 read 函数调用了一个解析器，用基于 fileName 指定的文件内容的信息来装载运行时数据结构。

在下一层，脱离运行时数据库进行操作的处理器被迫依赖于这个联合的解析器以及运行时数据库子系统。组件 system 相对较小，它同时管理运行时数据库的装载和处理。

虽然上面的体系结构是可层次化的，但却预示着一些有关维护和增强的潜在的严重后果。即使处理时并不需要一个解析器，一个处理器的开发也会既与解析器相耦合，又与运行时数

据库相耦合。随着系统的扩展，我们决定加入更多的处理器，在开发过程中每一个处理器都必须承担连接解析器的不必要的负担。

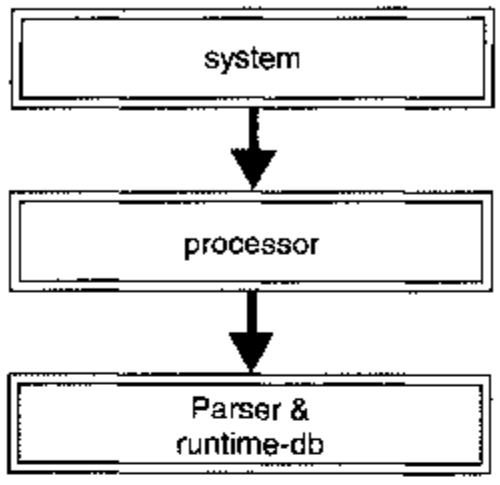


图 5-26　不良分解的运行时数据库体系结构

假设我们决定改变输入文件的格式或者（更糟糕）使用多重格式。现在，运行时数据库不再只支持单个的读命令，它必须支持若干个了：

```
RuntimeDB::Status RuntimeDB::readFormatA(const char *fileName);
RuntimeDB::Status RuntimeDB::readFormatB(const char *fileName);
RuntimeDB::Status RuntimeDB::readFormatC(const char *fileName);
```

这个结构将要求多个解析器连同运行时数据库共存于一个子系统中，如图 5-27 所示。

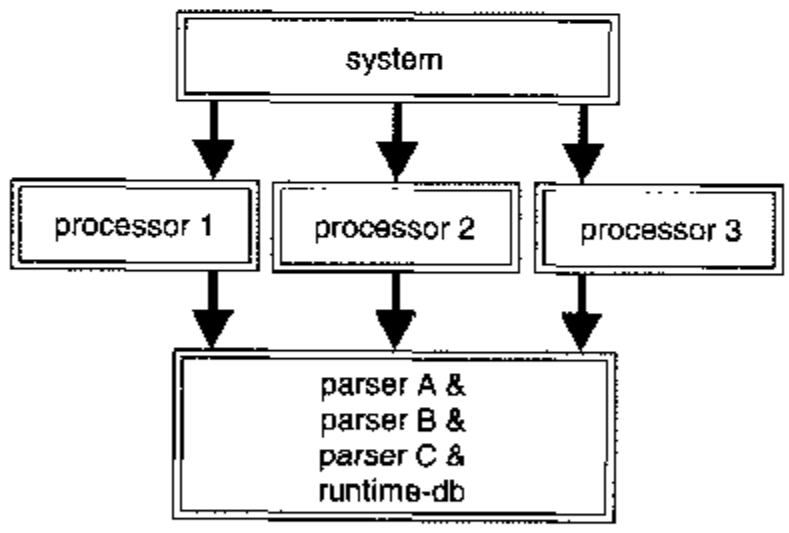


图 5-27　增强一个不良设计的结果

这个子系统体系结构的结果是，无论何时我们做以下事情都必须连接全部现有的解析器：

- 增强运行时数据库；
- 增强或开发一个新的解析器；
- 增强或开发一个新的处理器；
- 测试以上任意一项；
- 在一个处于独立状态的产品中重用运行时数据库。

在最初的体系结构中，数据库依赖解析器来装载信息。但是，进一步细查（见图 5-28）后，我们意识到在运行时数据库和解析器之间有（或应该有）一种几乎是非循环的关系。数据库是一个低层次的信息仓库，客户程序（例如解析器）存放信息进去，像处理器这样的客户又从那儿存取信息并（可能）操纵信息。每个解析器都依赖运行时数据库来存储被解析的信息。问题出在运行时数据库对一个解析器的无理由的“向上”依赖。

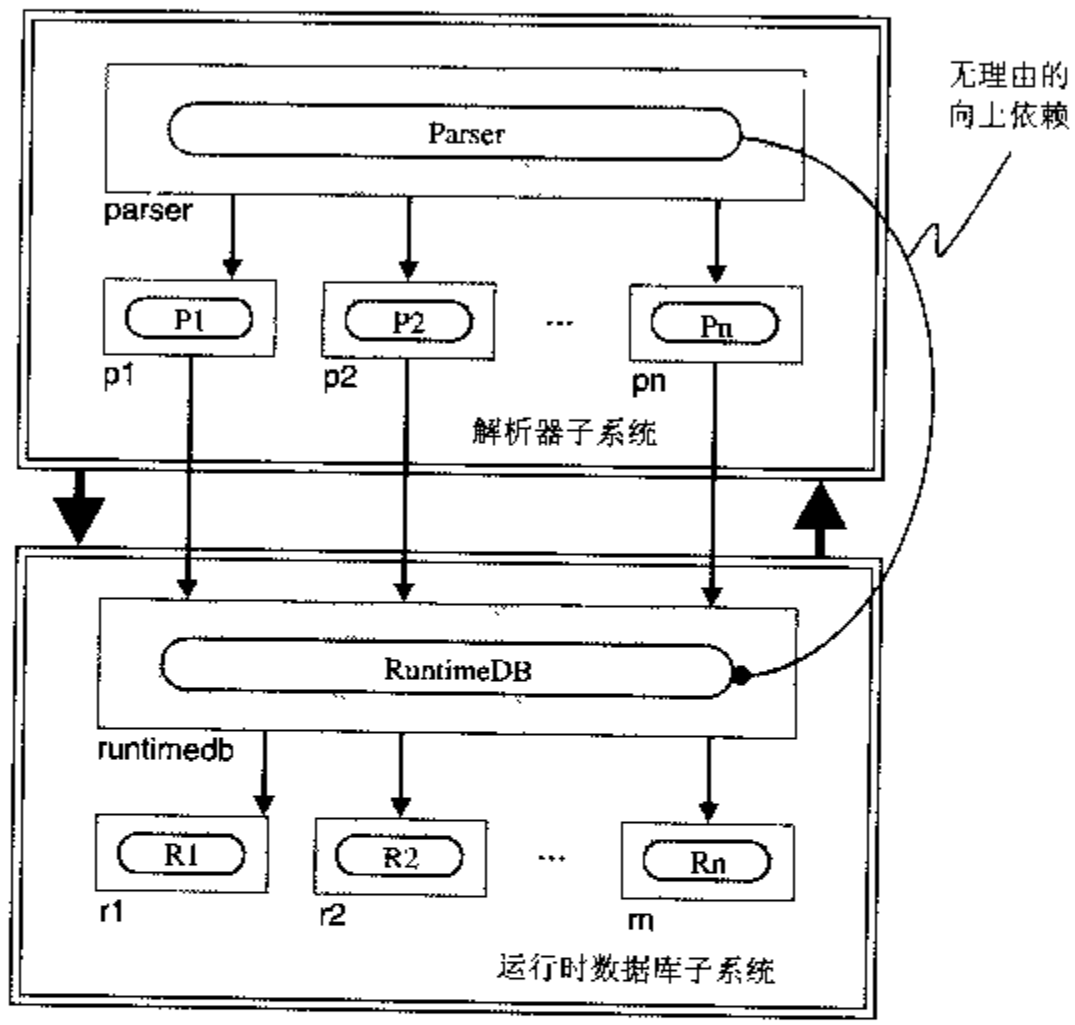


图 5-28 不良的解析器/数据库子系统体系结构的特写

升级和降级的结合为这个问题提供了一种有效的解决方案。我们可以重新构建原来的系统（见图 5-29），首先把解析器的 read 函数的调用从 RuntimeDB 升级到系统层次，然后将共有的运行时数据库子系统降级。通过把数据库变成一个“哑的”仓库（用一个过程接口来完成，该接口可以通过编程来装载和取回信息），每一个解析器就变成了数据库的“另一个客户”。现在系统只管理那些要被解析的文件，然后调用适当的解析器，传递给它一个指向要被装载的 RuntimeDB 对象的可写（非 const）指针：

```
ParserA::parse(RuntimeDB *db, const char *fileName);
```

如果后来的处理不会改变运行时数据库，而假定只是产生报告，系统可以通过给处理器传递一个只读的、到要被装载的 RuntimeDB 的（const）引用来确保数据库不被改写：

```
Processor1::quarterlyReport(ostrstream *ostr, const RuntimeDB& db);
```

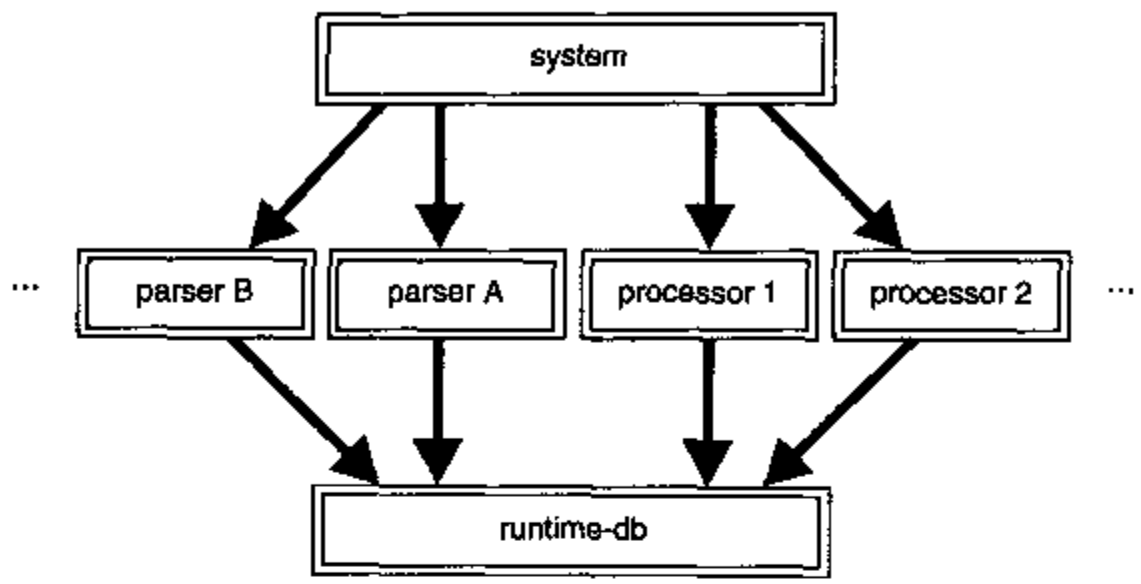


图 5-29　一个更好维护的系统体系结构

有了这个新的体系结构，任何数量的独立处理器都可以加入到系统中，它们不会依赖于任何解析器。同样地，解析器也可以被替换或增加，不会以任何方式影响运行时数据库、处理器或其他解析器。在这种体系结构下也不难想象，数据库、解析器和处理器可以在其他的处于独立状态的应用程序中，以不同的组合重用（例如：编译器、档案库存储器和浏览器）。

作为讲解降级威力的最后一个例子，考虑图 5-30 所示的子系统，其中的三个相关组件是循环依赖的。

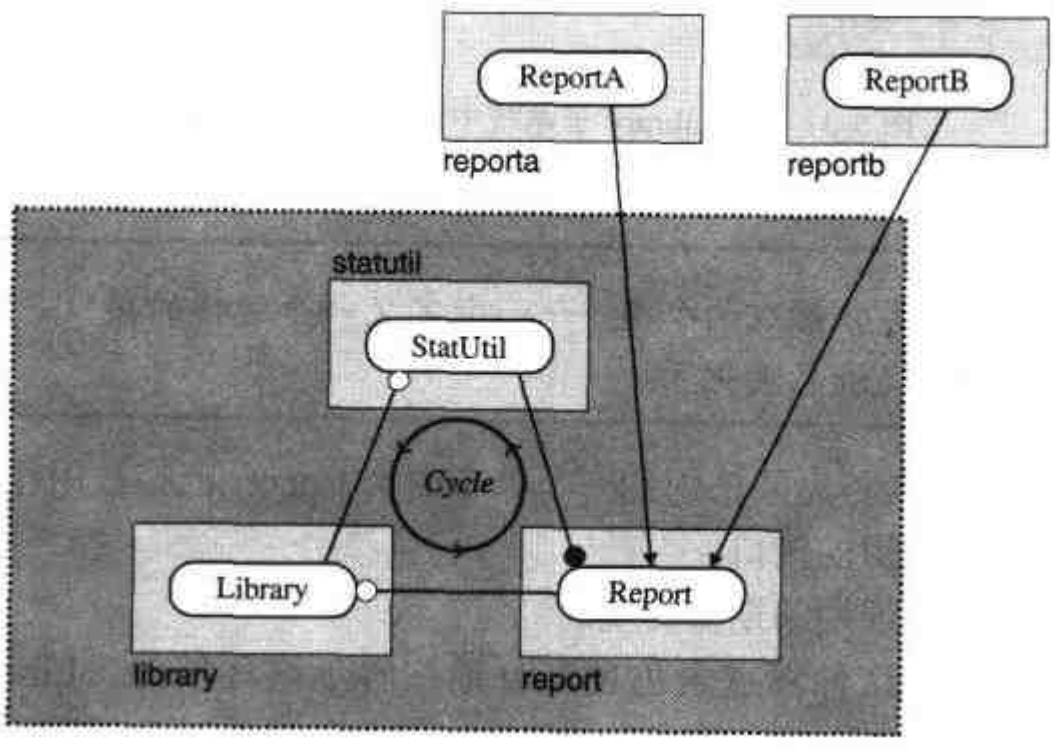


图 5-30　循环依赖的 Library 子系统

Library 包含了一个低层次信息数据库以及一个不同种类 Report 对象的集合。几乎所有种类的 Report 都提供依赖于聚集统计数字的信息，这些聚集统计数字是从存储在 Library 的低层

次数据计算出来的。一个统计工具类 Statutil 被用于帮助获得这个聚集信息[1]。在（抽象的）Report 基类中实现的共有功能使用了 StatUtil，StatUtil 因而依赖 Library，导致了 library、statutil 和 report 组件之间的循环依赖。

原　则

把一个具体的类分解为两个包含更高和更低层次功能的类可以促进层次化。

这个问题的出现部分是因为单个的 Library 类既用作低层次信息的仓库又用作（高层次）报告的集合。幸运的是有两个可供选择的解决办法。首先，通过把低层次仓库降级到子系统其余部分之下，我们可以消除循环耦合（如图 5-31 所示）。

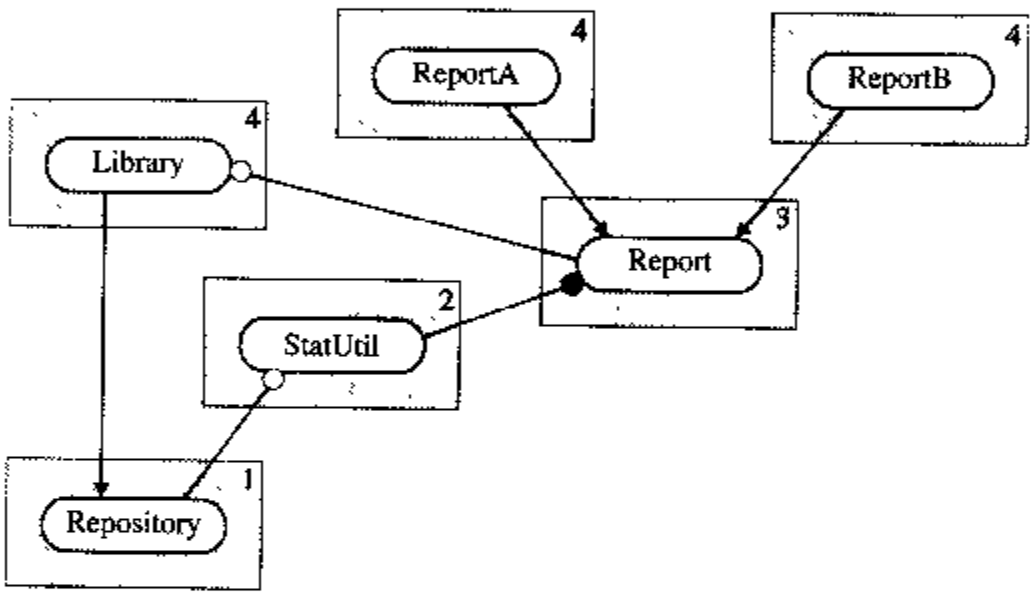


图 5-31　将 Library 子系统中的低层次信息降级

原　则

将一个抽象的基类分解成两个类——一个定义一个纯粹的接口，另一个定义它的部分的实现，可以促进层次化。

再看另一个解决办法，首先要认识到一个单个的类 Report 已经被用作两个不同的目的：

（1）提供所有报告通用的接口。

（2）提供所有报告都通用的期望实现。

让一个单个的类承担这种双重角色也部分地归咎于循环依赖。Library 直接依赖基类

[1] 通常，一个工具类要么只是一个为相关自由函数集合提供一个作用域的 struct，要么是一个模块（module）（即这个类只包含静态数据成员）。在以上任何一种情况下实例化这样一个类的实例都没有意义，因为它们不包含与一个特定实例相关的状态。见 **booch** 中的“Class Utilities”（第 5 章，186~187 页）。

Report 的接口，但只是通过使用虚函数间接依赖它的实现。

考虑一下，如果我们把 Report 分离成两个类会发生什么。第一个类将定义一个在原来的 Report 类中指定的接口，但不实现任何函数。也就是说，Report 类中的每一个函数现在都将被声明为一个纯虚[1]函数。第二个类，名为 ReportImp，将派生自 Report，通过覆盖适当的虚函数来提供类属的报告实现。

现在，通过只把定义在 Report 基类的接口降级到 Library 层次之下，就有可能解开原有系统（图 5-30）的循环依赖。实现共有功能并依赖 StatUtil 的 ReportImp 类，仍保留在物理层次结构的一个更高层次上，如图 5-32 所示。

我们考虑这些转换是升级还是降级并不重要。重要的是，我们能够用两种方法把单个的类分离成两个类，从而可以到达物理层次结构的不同层次。

哪一种解决方案更好些呢？第一种分解 Library 的方案对维护来说是理想的，因为低层次仓库和统计工具组件一样，可以独立于报告集合开发。第二种解决方案（图 5-32）对完整的未分解的 Library 施压，从而使低层次仓库和统计工具夹在接口和 Report 的部分实现之间。从这个角度来看，第一个解决方案更可取。但是还有另外的理由要求一个单个的基类应该定义在接口或者分解的实现中，但不能同时定义在这两个地方。从一个基类的（部分）实现中分离出接口，会在 6.4.1 节中详细讨论。

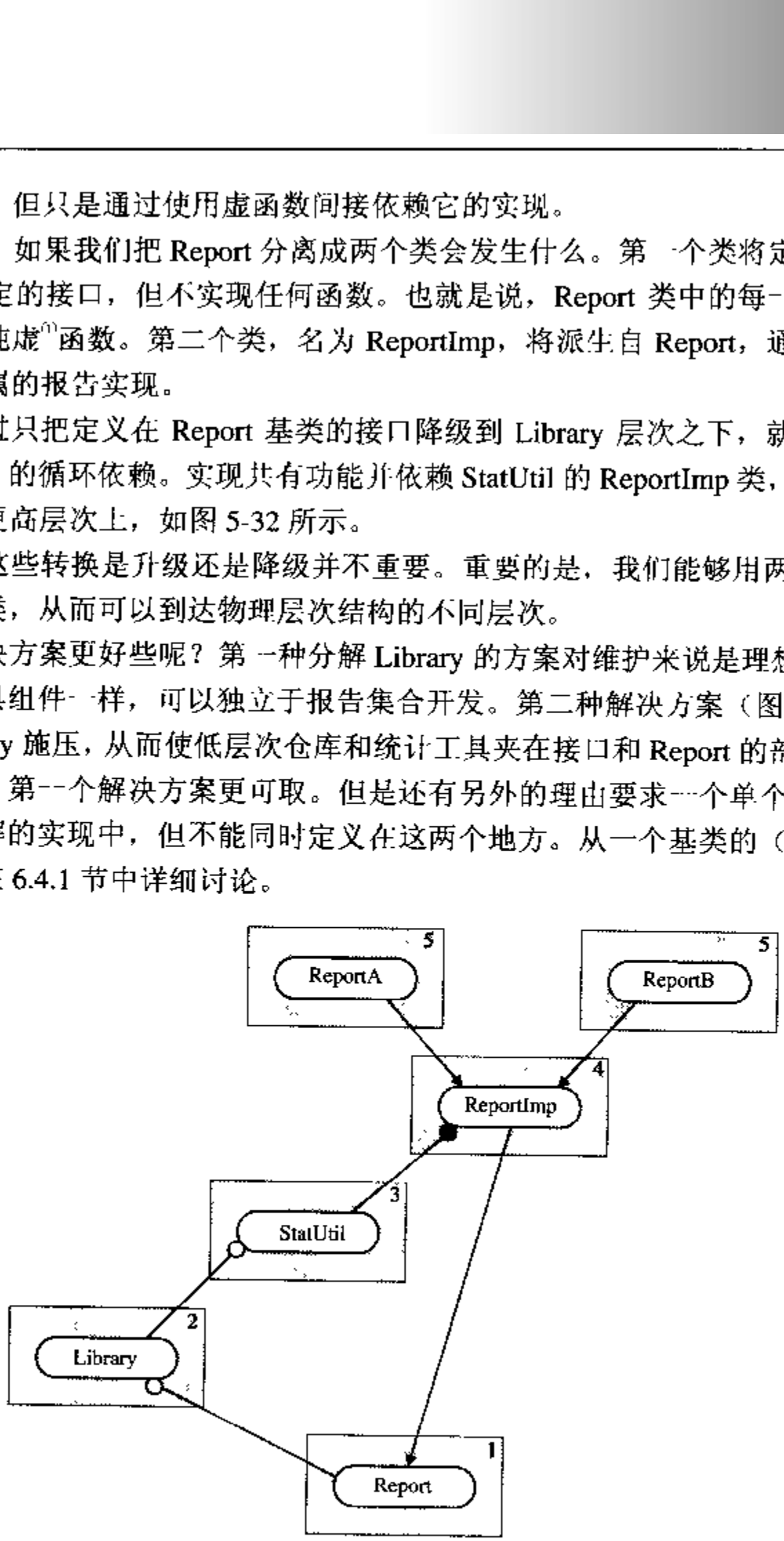


图 5-32 “只”降低 Report 基类接口的层次

① 析构函数也可以声明为虚函数，但不可以声明为纯虚函数（见 9.3.3 节）。

原　则

把一个系统分解成更小的组件，既可以使它更灵活，也可以使它更复杂，因为现在要与更多的物理部件一起工作了。

采用这两种转换后，可以产生一个更加灵活的体系结构。因为一个报告集合作为一个独立的抽象才有意义，所以可以通过允许报告集合独立于 Repository 被测试和重用来进一步改进这个体系结构。

在这个新体系结构中（图 5-33 所示），物理结构看起来比以前任何一个体系结构都更灵活。为了避免不必要的编译时耦合，我们必须把 Report 从它的部分实现中分离出来（在任何情况下）。这样做也使得我们能够创建一个很简单的、没有使用或依赖 StatUtil 的测试桩（ReportC）来测试 Report、Collection 和 Library（见图 5-34）。

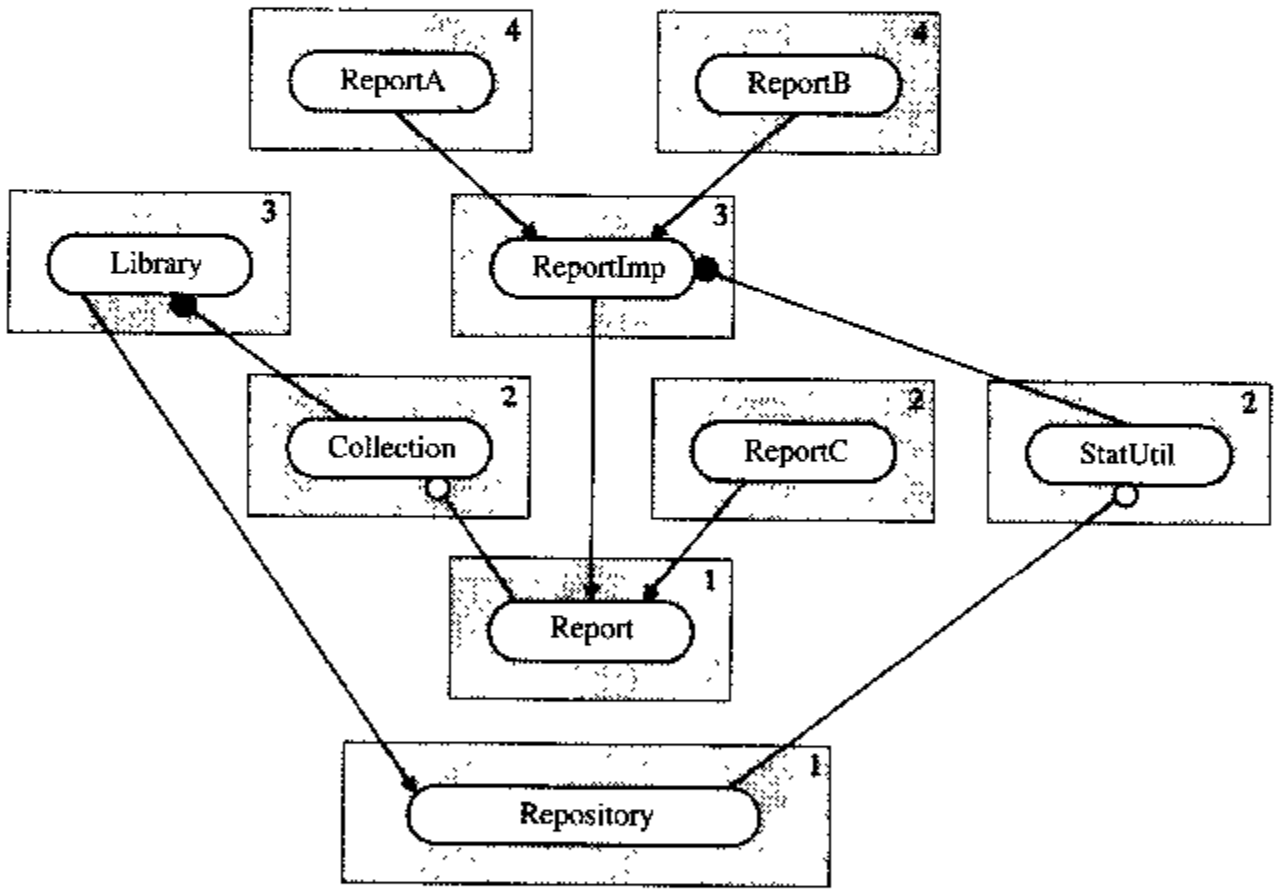


图 5-33　采用所有三个体系结构的改进

分解 library 组件是有利的，因为它进一步减少了子系统中的物理耦合。分割也特别适当，因为我们已经使 StatUtil 只依赖于 Repository，使 Collection 只依赖于 Report，所以给层次结构增加了相当大的灵活性。

降级 Repository 使其可以独立地被测试和重用[图 5-35（a）]，或者与 StatUtil 协同作用[图 5-35（b）]。单独的、复杂的报告（如，ReportA）可以被测试和重用，而不必依赖一个最后也许会也许不会保存它们的集合[见图 5-35（c）]。

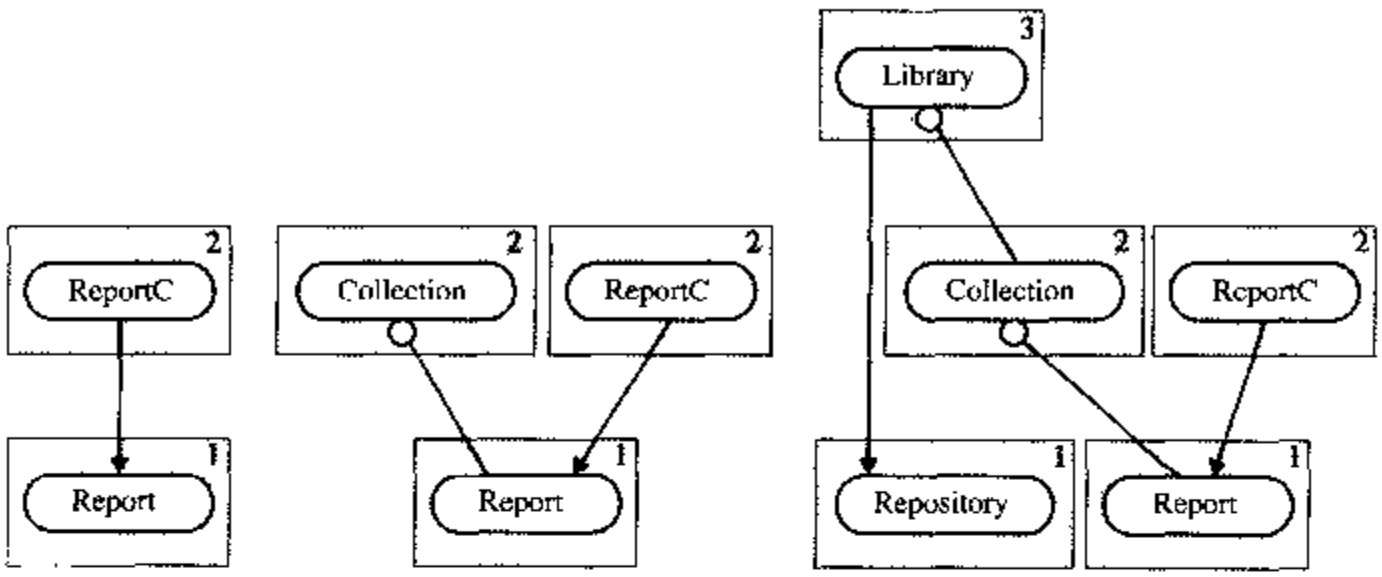


图 5-34 可独立测试和重用的子系统

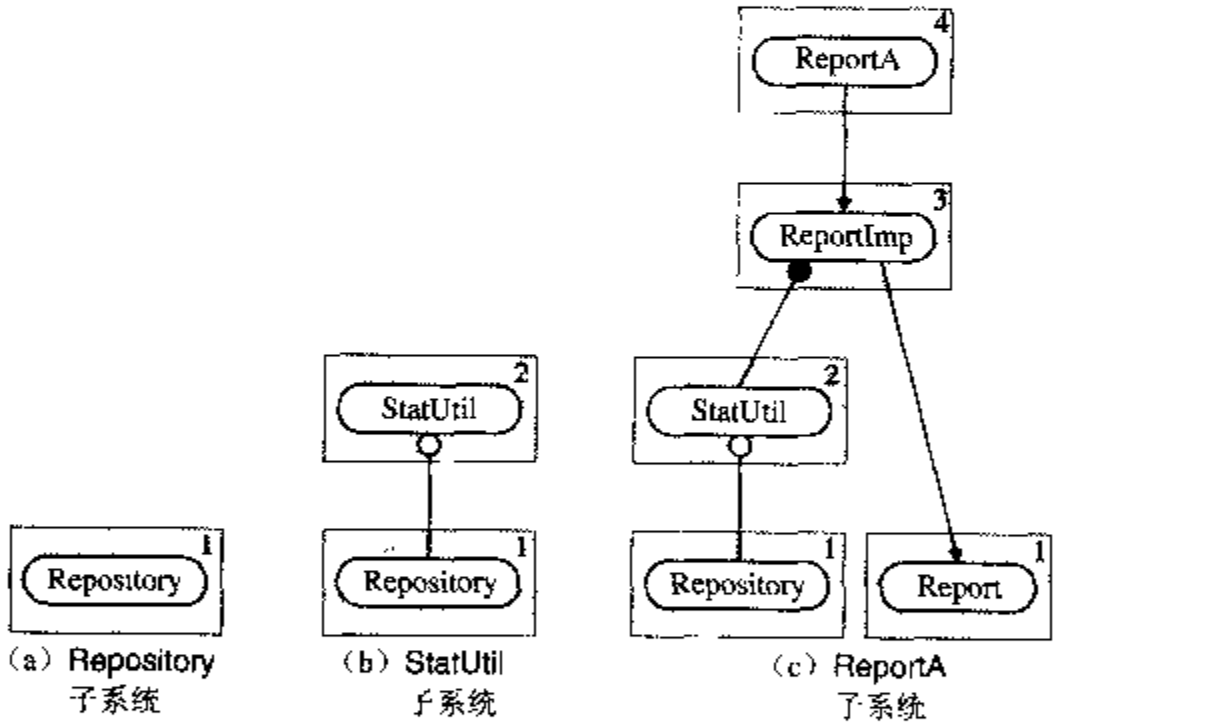


图 5-35 更可以独立测试和重用的子系统

升级和降级紧密相关。升级与降级的本质区别只是功能移动的方向不同（这些被移动的令人讨厌的功能的数量相对来说较少）。事实上，升级和降级实际都只是图 5-36 中所描述的通用重打包技术的特殊情况。

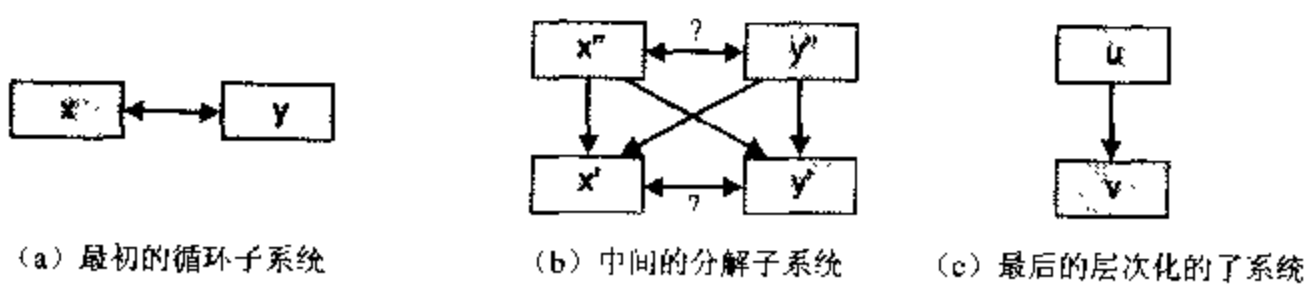


图 5-36 通用重打包技术

在这里，两个互相依赖的组件（a）再一次分解成四个组件（b）。其中的两个组件，x'和y'，可能只是彼此依赖，而另外两个组件，x"和 y"，潜在地依赖其他三个组件的每一个。这两个分别的、也许互相依赖的组件对，现在可能组合成两个新的组件（c）。组件 u 现在依赖组件 v，而组件 v 是独立的。这种通用重打包技术被非正式地应用于本节一开始讨论的 geomutil 组件和 geomutil2 组件。

总结：我们有时可以通过分解出普遍需要的功能，并将它移到物理层次结构的更低层来消除组件之间的互相依赖。降级不仅对改进循环相互依赖设计有用，而且也对减少非循环体系结构的 CCD 有用。将共有子系统降级可同时改进可维护性和可扩展性。一个分解适当的系统会更加灵活，因为它的内部物理依赖允许它的组件以更多种类的有用方式被独立测试和重用。

5.4 不透明指针

通常，我们假设如果一个函数使用了一个 T 类型对象，那么它以一种需要知道 T 的定义的方式使用 T。也就是说，为了编译函数体，编译器需要知道它所用的对象的大小和布局。在 C++中，一个编译器获悉一个对象的大小和布局的方法就是让使用这个对象的组件包含含有该对象类定义的组件的头文件。

> **定义**：如果编译函数 f 的函数体时要求提前看到类型 T 的定义，则称函数 f **实质（in size）使用了类型 T**。

如果一个函数体在只是看到类型 T 的声明（例如，classT;）的情况下就可以被编译，那么那个函数本身并不依赖于 T 的定义。实质使用一个类型的特点在于这样的用法会导致对定义 T 的组件的一种直接的编译时依赖。（避免不必要的编译时依赖是第 6 章的主题。）虽然函数 f 一般只在名称上使用而不是实质使用类型 T，但是如果 f 调用了其他组件中的一个或多个函数，这些函数依次地依赖 T 的定义，那么在这种情况下仍然有一个 f 对 T 的连接时依赖。

> **定义**：如果编译函数 f 以及 f 可能依赖的任何组件时，不要求提前看到类型 T 的定义，则称函数 f **只在名称上（in name only）使用了类型 T**。

如果函数 f 和 f 依赖的所有组件在只看到了 T 的**声明**（而不是定义）的情况下就能够编译和连接，那么 f 就被认为**只在名称**上使用了 T。例如：

```
// util.h
#ifndef INCLUDED_UTIL
#define INCLUDED_UTIL

class SomeType; // used in name only

struct Util {
    SomeType *f(SomeType *obj);
}

#endif
```

```
// util.c
#include "util.h"

SomeType *Util::f(SomeType *obj)
{
    static SomeType *lastType=0;
    return obj ? lastType = obj : lastType;
}
```

说明函数 f 只在名称上使用了类型 SomeType。只在名称上使用一个类型的特点在于这样的用法没有隐含的物理依赖——即使是在连接时。没有物理依赖，耦合也就几乎全部消除了。

也可以对类建立相似的定义，即类**实质**或**只在名称上**使用了一个类型。更有用的是这些定义能够扩展应用于作为一个整体的组件。

> **定义**：如果编译组件 c 时要求必须提前看到类型 T 的定义，则称组件 c **实质**使用了类型 T。

> **定义**：如果编译组件 c 以及 c 可能依赖的任何组件时不要求提前看到类型 T 的定义，则称组件 c **只在名称上**使用了类型 T。

我们会在第 6 章使用第一个定义。现在，我们集中讨论这两个组件级定义中的第二个的定义。注意，如图 5-37 所示，组件 u 在名称上使用了一个 T 对象并且依赖于另一个组件 v，因为传递的原因，组件 u 实质使用了 T，而不仅仅只是在名称上使用 T。组件 u 物理依赖于组件 v，并且间接依赖于组件 t。

使用符号的虚线形式“○----”表示使用“只是名称上的”，并且强加了一个概念上的但不是物理上的依赖。

在这里，我们没有使用 Booch 提出的用于

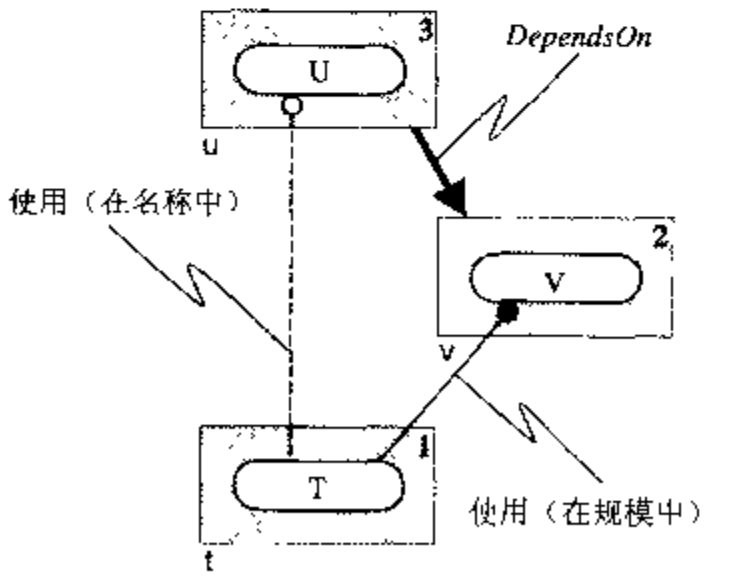
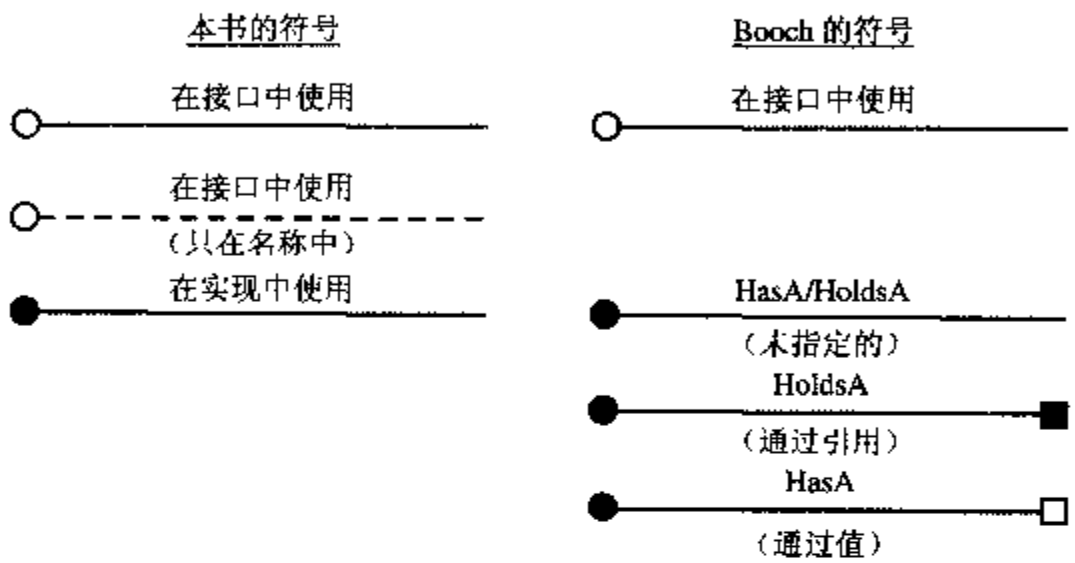


图 5-37　组件 u 不仅仅在名称上使用了类型 T

“通过引用（by-reference）”依赖的符号定义[①]，有两个原因：（1）这种新的 In-Name-Only 符号的逻辑意义和它的 In-Size 对应物是同样的——不同的只是**物理**隐含；并且（2）只在名称上使用（不像通过引用使用）清楚地否定了任何直接或间接的编译时或连接时依赖。对于我们的目标，有三种风格的使用符号就足够了：



原　则

只在名称上使用了对象的组件可以独立于被命名的对象被彻底测试。

涉及只在名称上使用一个类型的情况很少会自然出现；这样做只是为了避免不必要的物理依赖。当组件只通过指针或引用来“使用”对象，而决不以任何方式直接与对象交互（除了保存它的地址）时，只在名称上使用一个类型是可能的。

如果一个指针所指向的类型的定义不包含在当前的编译单元中，这个指针就被称为是**不透明的（opaque）**。图 5-38 显示了一个小例子，一个类拥有一个不透明指针，它指向某一名为 Foo 的类的实例。Handle 类的客户最终都将不得不包含定义了 Foo 的组件的头文件，以便产生一个 Foo 的对象。为了测试的目标，任何形式的 Foo 类都可以出现，只有一个类声明也可以，如图 5-39 所示。

```
// handle.h
#ifndef INCLUDED_HANDLE
#define INCLUDED_HANDLE

class Foo;

class Handle {
    Foo *d_opaque_p;
```

① **booch**，5.2 节，Figure 14，191 页。

```
  public:
     Handle(Foo *foo) : d_opaque_p(foo) {}
     void set(Foo *foo) { d_opaque_p = foo; }
     Foo *get() const { return d_opaque_p; }
};

#endif
```

图 5-38　只在名称上使用了 Foo 的句柄类

```
// handle.t.c
#include "handle.h"
#include <assert.h>

main()
{
    Foo *p1 = (Foo *) 0xBAD;
    Foo *p2 = (Foo *) 0xB0B;
    Handle handle(p1);
    assert(p1 == handle.get());
    h.set(p2);
    assert(p2 == handle.get());
}
```

图 5-39　handle 组件的微小测试驱动程序

这个例子的重要性在于说明了，在不包含或连接任何定义了 Foo 类的组件的情况下可以完全试运行 Handle 类的功能。这是一种决定性测试——测试是否有另外一个类型不仅被不透明地使用了，而且是只在名称上使用。

在开发一个具体的应用程序时，一个较高层次的对象常常会将信息存储于定义在物理层次结构的较低层次的对象中。如果信息以一种用户自定义类型的形式存在，就有可能导致下级对象依赖那个类型。只要这个下级不需要主动地对那个类型进行任何实质的使用，这个下级就没有必要包含该类型的定义。

原　则

如果一个被包含的对象拥有一个指向它的容器的指针，并且要实现那些实质地依赖那个容器的功能，那么我们可以通过以下方法来消除相互依赖：（1）让被包含类中的指针不透明；（2）在被包含类的公共接口上提供对容器指针的访问；（3）将被包含类的受影响的方法升级为容器类的静态成员。

假设 Screen 是 Widget 对象的容器，并且进一步假设每个 Widget 都拥有一个指针 d_parent_p，标识这个 Widget 属于的 Screen。现在考虑图 5-40 中给出的 widget 和 screen 组件

的接口，尤其是 Widget 类的访问成员函数 numberOfWidgetsInParentScreen。

这个函数允许一个只拥有一个 Widget 的客户程序找出这个 Widget 所属的 Screen 有多少其他的 Widget 对象。从纯粹易用性的角度来看，这个体系结构似乎是吸引人的；从维护角度来看，它是昂贵的。

```
// screen.h
#ifndef INCLUDED_SCREEN
#define INCLUDED_SCREEN

class Widget;

class Screen {
    Widget *d_widgets_p;
    // ...
  public:
    Screen();
    // ...
    void addWidget(const Widget& w);
    // ...
    int numWidgets() const;
    // ...
};

#endif
```

（a）容器组件 screen

```
// widget.h
#ifndef INCLUDED_WIDGET
#define INCLUDED_WIDGET

class Screen;

class Widget {
    Screen *d_parent_p; // Screen to which
    // ...              // this widget belongs
  public:
    Widget(Screen *screen);
    // ...

    // operations involving parent screen

    int numberOfWidgetsInParentScreen() const;

    // ...
};

#endif
```

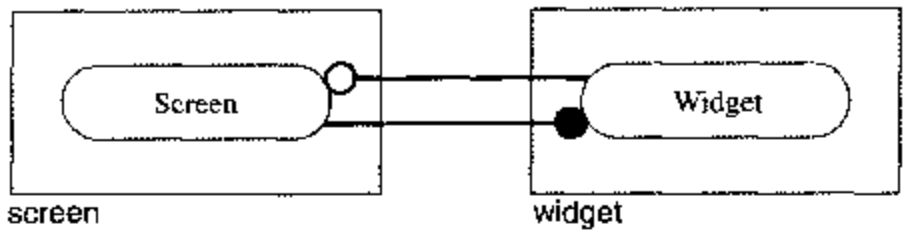
（b）被包含的组件 widget

图 5-40 Screen/Widget 设计导致循环依赖

这个设计的维护问题在于，为了实现 widget.c 文件中的 numberOfWidgetsInParentScreen 方法，我们必须“请求”父 Screen 来取得这个信息。请求 Screen 任何事情都意味着已经看到了它的定义，这要通过首先把 screen.h 包含在 widget.c 中来实现。但是这样做导致了图 5-41 中描述的不可层次化的情形。

这里的基本问题是 Widget 试图做得比它应该做的更多。一个 Widget 提供在它自己的上下文中有意义的功能，但它通常不可能不询问它的父 Screen 就知道其他的 Widget 对象。再考虑一个公司的类似情况，你可以问任何雇员“你正在干什么？”，而雇员应该能够告诉你。同样地，你也可以总是问雇员“谁是你的老板？”。但试着问问雇员为他（或她）的老板工作的雇员的数量，通常雇员不会知道答案，并且需要去老板那儿问。

实际上，了解有多少雇员为老板工作不是雇员的职责。考虑一个可替代的方法。假设你想知道有多少雇员为我的老板工作。不要问我这个问题，而是问我“谁是你的老板？”，我会告诉你，然后你可以自己去问她有多少雇员为她工作。如果她想告诉你，她就会告诉你。




```
// widget.c
#include "widget.h"
#include "screen.h"
// ...
int Widget::numberOfWidgetsInParentScreen() const
{
    return d_parent_p->numWidgets();
}
// ...
```

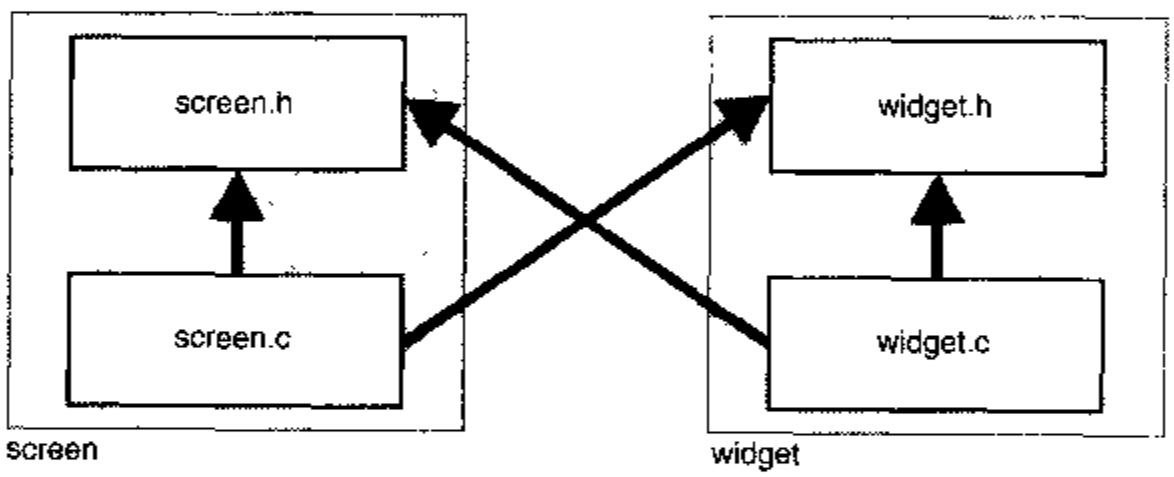


图 5-41　类 Widget“知道”容器类 Screen

使用不透明指针（只在名称上使用）可以用来打破不需要的循环组件依赖。返回到我们的设计实例，考虑图 5-42 所示的 widget 组件的可替代的定义。在这种使用模式中，可以问 Widget 来获得它的父 Screen。然后我们就能够问这个父 Screen 有关它的其他的 Widget 对象的事情（或有关那个问题的其他任何事情）。但是，这个模式的主要益处在于，组件 widget 在编译时或连接时都不再依赖组件 screen。此时 widget 对 screen 的依赖只是名称上的了。

组件 screen 和 widget 的新的组件依赖图显示在图 5-43 中。有了这个新的体系结构，就有可能独立于 screen 组件测试 widget 的所有功能。其他使用了 widget 但不关心 screen 的组件不需要包含 screen.h 或连接到 screen.o。

一个说明 widget 对 screen 的物理依赖的小型测试驱动程序如图 5-44 所示。

这个体系结构变化的一个直接结果是，客户程序必须执行两个操作（不再是一个操作）来获取父 Screen 中的 Widget 对象的数量：

```
widget.parentScreen()->numWidgets()
```

```
// widget.h
#ifndef INCLUDED_WIDGET
#define INCLUDED_WIDGET

class Screen;

class Widget {
    Screen *d_parent_p; // screen to which this widget belongs
    // ...
  public:
    Widget(Screen *screen);
    // ...

    // operations involving parent screen

    Screen *parentScreen() const;
};

#endif
```

```
// widget.c
#include "widget.h"
#include "screen.h" // no longer needed

// ...

Screen *Widget::parentScreen() const
{
    return d_parent_p;
}
```

图 5-42 组件 widget 的修改过的体系结构

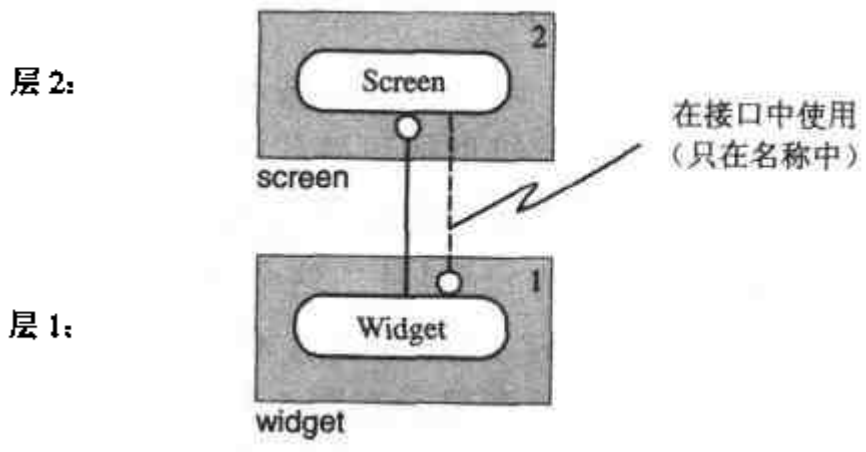


图 5-43 widget 和 screen 的可层次化的组件依赖

```
// widget.t.c
#include "widget.h"
#include <iostream.h>

class Screen;     // not necessary when including widget.h

main()
```

```
{
    Screen *const screen = (Screen *) 0xbad;

    const Widget widget(screen);

    if (screen != widget.parentScreen()) {
        cout << "Error!" << endl;
    }

    // ...
}
```

图 5-44　widget 组件的隔离测试驱动程序

为了方便，这两个操作可以合并成 Screen 的一个静态成员函数或其他的较高层次的类。不再用

```
widget.numberOfWidgetsInParentScreen()
```

而是要用

```
Screen::numberOfWidgetsInParentScreen(widget)
```

来获得这个值。在两种情况下，接口都强迫客户必须在 widget 组件的接口之外寻找才能获得他们所提问题的答案。

注意，当把功能从被包含对象移到容器时，新静态成员的第一个参数必须要么是一个 const 引用，要么是一个指向被包含对象的非 const 指针——取决于原来的成员是一个 const 还是一个非 const 函数。这种参数传递风格的基本原理将在 9.1.11 节继续讨论。

扼要重述：通过使 Widget 类的内部 Screen 指针不透明，以及把 Widget 中实质使用了 Screen 类的那部分移出 Widget，移入 Screen 类本身，我们能够获得一张非循环的组件依赖图。我们也在 Widget 的公共接口上暴露了 Screen 类型，并且使 widget 的客户必须在那个组件之外寻找的问题答案。但在这样做的时候，相互的物理依赖被概念上的协作所取代：较低层次的对象只同意保持在较高层次上使用的信息（只在名称上指定）。

5.5　哑数据

术语**哑数据（dumb data）**是对不透明指针概念的一种概括。哑数据是一个对象拥有但不知道如何解释的任何种类的信息。这样的数据必须用在另一个对象的上下文中，通常用在一个更高的层次上。

让我们考虑通过实现一个简化的子系统来建造一个赛马跑道模型时可能会涉及些什么。作为一个起点，我们期望能够问一些诸如图 5-45 所示的问题。顶层的组件应该提供在赛马比

赛（races）上进行迭代和通过名称来识别一匹马的能力。为了使这个例子更有趣，一个跑道（track）也应该可以接受赌注（bet）和赎回赌注（wager）。

```
void questions(const Track& track,
               const Race&  race,
               const Horse& horse)
{
    // 1. What races do you run here?
    for (RaceIter it1(track); it1; ++it1) {
        cout << it1().number() << endl;
    }

    // 2. What time does a given race start?
    cout << race.postTime() << endl;

    // 3. What horses are running in a given race?
    for (HorseIter it3(race); it3; ++it3) {
        cout << it3().name() << endl;
    }

    // 4. What is the number of this horse?
    cout << horse.number() << endl;
}
```

图 5-45 对一个赛马跑道会问的一些常见问题

顶层 track 组件的最初片段在图 5-46 中给出。在这个体系结构中，一个 Track 拥有 Race 对象的一个集合，并提供一个 RaceIter 来对这个 track 上今天的赛跑比赛进行迭代。Track 接受赌注（bets）并且发（指针给）给 Wager 对象，Wager 可以在赛跑比赛结束后被赎回。

```
// track.h
#ifndef INCLUDED_TRACK
#define INCLUDED_TRACK

class Horse;
class Race;
class RaceIter;
class Track;

class Wager {
    const Horse& d_horse;
    double d_amount;
    // ...
    Wager(const Horse& horse, double amount);  // For track's use only
    Wager(const Wager&);                       // -- i.e., not for use
    Wager& operator=(const Wager&);            // by the public.
    friend Track;
```

```
  public:
    const char *horseName() const;
    int raceNumber() const;
    Track& track() const;
    double amount() const;
};

class Track {
    Race *o_races_p;
    // ...
    friend RaceIter;

  public:
    // ...
    const Race *lookupRace(int raceNumber) const;
    const Horse *lookupHorse(const char *horseName) const;
    Wager *bet(const Horse& horse, double wagerAmount);
    double redeem(Wager *bet) const;
};

class RaceIter {
    // ...
  public:
    RaceIter(const Track& track);
    void operator++();
    operator const void *() const;
    const Race& operator()() const;
};

#endif
```

图 5-46　顶层组件 track 的头

每个 Race 对象保持着该场比赛的号码、起始时间以及正在比赛的马的集合。组件 race 也提供一个 HorseIter 来迭代在一场指定 Race 中赛跑的马。假如有一个 Race 对象，就有可能确定比赛将在哪一个跑道上进行。组件 race 的一个粗略版本显示在图 5-47 中。

```
// race.h
#ifndef INCLUDED_RACE
#define INCLUDED_RACE

class HorseIter;

class Race {
    // ...
    friend HorseIter;

public:
  Race(const Track& track, int raceNumber, double postTime);
  // ...
```

```
    int number() const;
    double postTime() const;
    const Track *track() const;
};

class HorseIter {
    // ...
  public:
    HorseIter(const Race& race);
    void operator++();
    operator const void *() const;
    const Horse& operator()() const;
};

#endif
```

图 5-47 中间层组件 race 的头

Horse定义在赛马跑道子系统的物理层次结构的最底层。一个Horse保持它的名字和号码，可用来确定它被安排在哪一场比赛中跑。处在我们的叶子层的 horse 组件的开始片段如图 5-48 所示。

```
#ifndef INCLUDED_HORSE
#define INCLUDED_HORSE

class Race;

class Horse {
    const Race& d_race;
    char *d_name_p;
    int *d_number;
    // ...
  public:
    Horse(const Race& race, const char *HorseName, int horseNumber);
    // ...
    const char *name() const;
    int number() const;
    const Race *race() const;
};

#endif
```

图 5-48 叶子层组件 horse 的头

在这个最初的实现中，一个 Wager 只用了两个数据成员来实现，如下所示：

```
class Wager {
    const Horse& d_horse;
    double d_amount;
```

```
    // ...
  public:
    // ...
};
```

若拥有一个指向一个 Horse 的指针，在足够高层次的客户就有可能使用被惟一识别的 Horse 来获得一个指针，指向那匹马将要参加的 Race（无论在世界的何处）。然后可以遍历该 Race 指针，并且作为结果的 race 对象通常会获得一个指针，指向那个将要举行比赛的跑道。

上面所描述的赛马跑道系统的功能意味着存在一种循环内部数据结构：每个 Track 知道它所举行的比赛，每个 Horse 知道它要参加哪一场比赛，每个 Race 既知道它将在哪个跑道举行也知道将要参加本次比赛的马匹。但是，这种数据结构可以通过使用不透明指针以非循环物理依赖的形式来实现，如图 5-49 中的组件/类图所示。

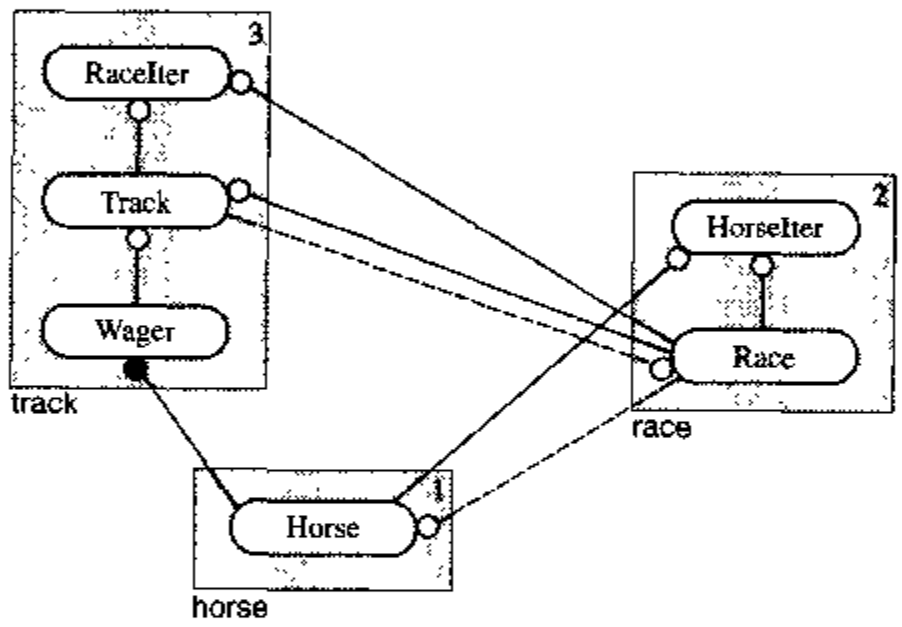


图 5-49　赛马跑道子系统的组件/类图

最初为赛马跑道子系统设计的体系结构没有循环物理依赖，但在名称上却是循环依赖的。虽然有两个组件知道一个或多个定义在其他组件中的对象的名称不一定是坏事，但需作一些权衡，我们对此略作讨论。

假设我们识别某个系统的对象不是通过它们的绝对地址，而是根据对象的索引顺序表，则这些对象只在父对象的上下文中才有意义。

于是 Track 将拥有一个 Race 对象的序列（数组），并且每个 Race 都将有一个关联整数“index”。Race 索引只在一个 Track 对象的上下文中有意义。因为 Race 索引可以相应于公共可访问的 Race 编号，倘若我们为 Track 提供一个访问函数来报告今天举行的比赛的总数，对 RaceIter 的需求就减少了。

通过同样的参数，一个 Race 中的每个 Horse 都被自然地赋值一个号码。假如一个 Race 有一个马的序列，我们可以通过提供马相对于那场比赛的索引来识别一个 Race 内的 Horse。

因此我们也可以为 Race 免除 HorseIter。

当要赎回赌注时，Track 定义了一个比整个地址空间小得多的上下文（可通过指针访问）。在最初的实现中，我们使用了不透明后退指针（back pointers），以 Horse 开始，以一种自下而上的方式移动到达 Race，最后到 Track。在这个推荐的实现中，Track 的受限制的上下文被用来识别 Race 和 Horse，借助于一对整数下标来实现，如图 5-50 所示。

```
class Wager {
    const Track& d_track;
    double d_amount;
    short int d_raceIndex;
    short int d_horseIndex;
    // ...
  public:
    Wager(const Track& track,
          int horseNumber,
          int raceNumber,
          double amount);
    const Track& track() const;
    double amount() const;
    int horseNumber() const;
    int raceNumber() const;
    // ...
};
class Track {
    Race *d_races_p;
    int d_numRaces;
    // ...
  public:
    Wager *bet(int race, int horse, double amount);
    double redeem(Wager *bet) const;
    const Race *lookupRace(int raceNumber) const;
    const Horse *lookupHorse(const char *horseName) const;
    const Horse *lookupHorse(const Race& race, int horseNumber) const;
    int numRaces() const;
    // ...
};
```

图 5-50　对组件 track 的修改

观察一下，因为这个非常受限制的上下文，我们可以安全地使用 16 位整数而不是 32 位整数。如果赌注的数量在任何一个时候变得非常巨大，这个事实将是很有意义的。例如，在我的 32 位机器上，一个双精度型是 8 字节长且是自然排列的[①]，当我们把索引变成 short 整数时，wager 对象的大小从 24 字节降到 16 字节——节约了 33%！

① 自然排列在 10.1.1 节中讨论。

图 5-51 展示了赛马跑道子系统的修改过的体系结构。新的系统明显地更简单了。这个系统没有组件间的循环依赖——即使是名称上的——而且明显有更少的类。主要的变化只是识别 Horse 的方式。

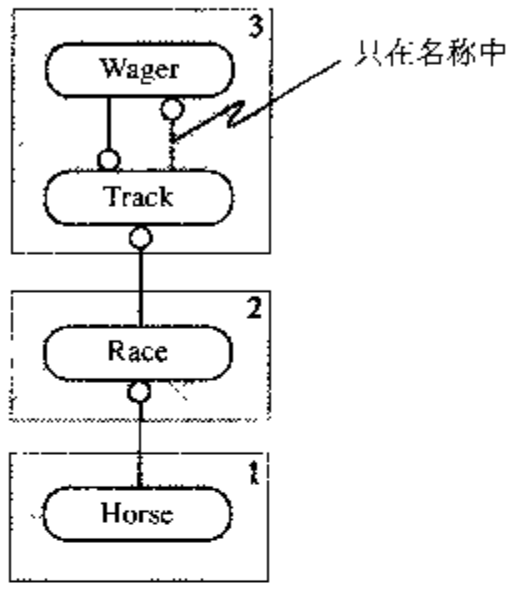


图 5-51 赛马跑道子系统的修改过的组件/类图

哑数据可能比用于识别其他对象的不透明指针更方便并且偶尔会更简洁。如果新的 Wager 对象是通过不透明指针而不是 short 整数索引来识别 Race 和 Horse，那么在我的机器上 Wager 的大小将又是 24 字节而不是 16 字节。

另一个好处是存储成哑数据的值不是机器地址，因而包含了可以被显式测试的有意义的值。在这个赛马应用程序中，索引的方法尤其吸引人，因为索引（是公共可访问的）确实在用户领域有着合理的效用。在下注窗口听到一个经常出入赛马场的跑道赞助人要求一个赌金代理“给我 2 美元赌 4 号在第 9 场（赢）!”并不少见。

这种索引方法的一个缺点是，与不透明指针相比，它确实牺牲了相当程度的类型安全，因为 Race 和 Horse 索引只是整数。另一个缺点是这种实现强迫 Race 和 Horse 集合被编入索引而不是保持任意的集合。最后对封装造成的侵蚀如果暴露给一般的公众，可能很容易对可维护性产生消极影响。

在我们的赛马实例以外的情形中，用来以这种方式识别了对象的哑数据索引也许对子系统的客户是毫无意义的。因此，哑数据的使用是一种典型的封装在一个子系统内的优化实现技术，它不会在一个系统的更高层次暴露。

原 则

哑数据可以用来打破 **in-name-only** 依赖，促进易测试性和减少实现的大小。但是，不透明指针可以同时保持类型安全和封装；而哑数据通常是不能的。

作为一个类似但更严肃的例子，我们来考虑这样一个任务：为一个电路组件内部的连通

性建立模型，该电路由电气组件的一个异类集合组成[1]。一个门级电路，例如在 4.7 节介绍层次化时引入的图 4-10 中的那一个，可以描述为一幅由节点[称为 **gate**（门）]和边[称为 **wire**（电线）]组成的图。每个 gate 都有一个电气上不同的连接点[称为 **terminal**（接头）]的集合。在概念上，表现一个电路就是要维护门的一个异类集合和双向电线的一个同类集合。每一根电线都被接在电路内的两个不同的接头上，建立连通性。

在传统的实现中，一个电路可能包含接头的一个集合，用于定义电路的主要输入输出。电路中的每一个门也将包含接头的一个集合。在这个模型中电线不是明确的对象。但是，每个接头都包含一个指向（其他）接头的指针的集合。指向另一个接头隐含建立了一个到该接头的连接。在图 5-52 所示的例子中，门 g0 的接头 z 被连接到门 g1 的接头 x 上；因此，g0 的接头 z 将拥有一个指向 g1 的接头 x 的指针。对称地，g1 的接头 x 也连接到 g0 的接头 z 上，这样 g1 的 x 也将拥有一个指向 g0 的 z 的指针。

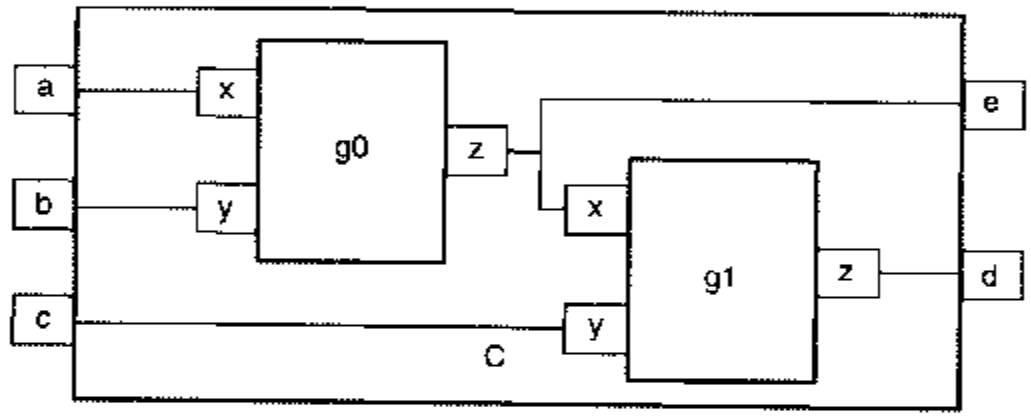


图 5-52 按两个门（g0 和 g1）来实现的电路 C

为了遍历图，一个 Terminal 必须保持一个指向它的父 Gate 或 Circuit 的不透明指针。注意，Circuit 可以看作是一个特殊种类的 Gate，它包含了其他门的实例[2]。循环物理组件依赖可以通过使用不透明指针来避免，如同在图 5-53 的部分组件/类图中显示的那样（集合迭代器被省略）。

这里又有一个机会允许我们通过“在上下文中”定义一个连接来打破即使是名称上的循环依赖。如果一个 Circuit 包含 gate 的一个索引的集合（一个数组），并且每个 Gate 都同样地包含一个接头的数组，那么我们可以在一个 Circuit 的上下文中将一个连接点标识为一对简单的整数下标。

---

① 这个例子描述了哑数据在一个很不一样的上下文中的应用。但是基本技术和用在赛马跑道例子中的技术是相同的。

② 这个例子解释了递归复合（recursive composition）——一种称为复合（*Composite*）*的设计模式*（**gamma**，第 4 章，163~173 页）——的另一个实例。这种模式在前面介绍过（在 4.7 节的图 4-13 中根据 Node、File 和 Directory 阐释过）。这种复合设计模式已被有效地用于实现分层次的电路描述。

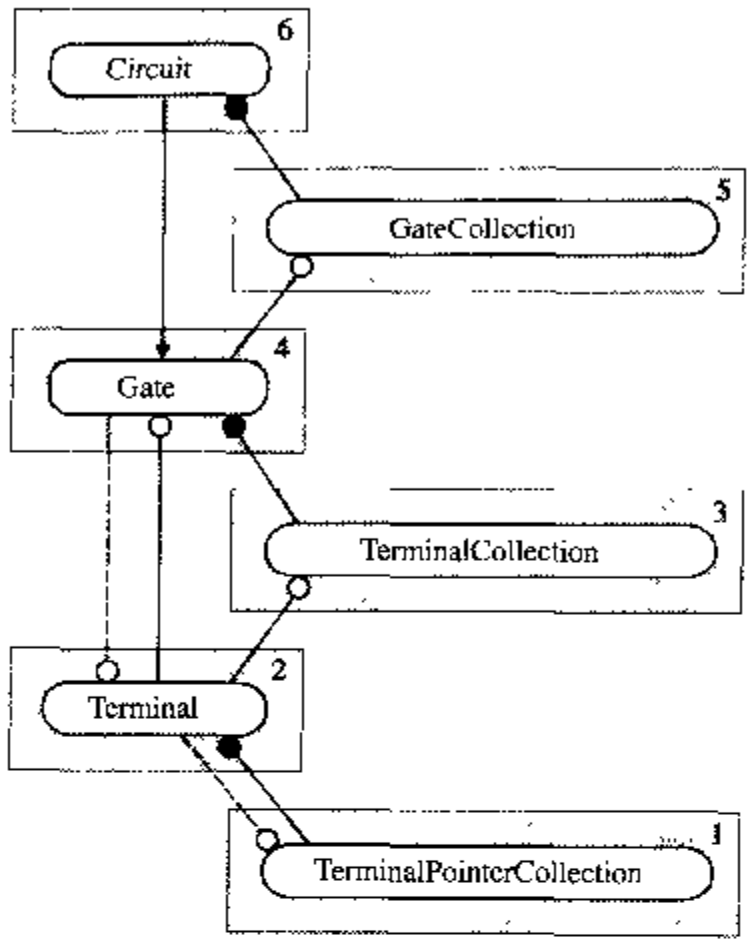


图 5-53　Circuit 实现的部分组件/类图

再考虑图 5-52 中的例子。假设 circuit 的实现由一个两个门（g0 和 g1，分别带着下标 0 和 1）的数组组成。g0 和 g1 的接头都是 x、y 和 z，并且碰巧分别有下标 0、1 和 2。我们现在可以把连接点“门 g1 的接头 x”描述为一对整数下标（1，0）。同样地，我们也可以把连接点 g0 的 z 描述为坐标对（0，2）。

按照惯例，我们可以通过为门的索引使用一个合法范围之外的索引（例如–1）来标识封闭的电路。如果电路的接头 a 有下标 0，它的连接坐标可以表示为下标对（–1，0）。这个电路的连接的完整列表以整数坐标描述，如图 5-54 所示。

```
C.a = (-1, 0) <-------> ( 0, 0) = g0.x
C.b = (-1, 1) <-------> ( 0, 1) = g0.y
C.c = (-1, 2) <-------> ( 1, 1) = g1.y
g0.z = ( 0, 2) <-------> ( 1, 0) = g1.x
g0.z = ( 0, 2) <-------> (-1, 4) =  C.e
g1.z = ( 1, 2) <-------> (-1, 3) =  C.d
```

图 5-54　用整数下标表示连通性

一对整数不存在对任何东西的物理依赖。因此我们可以像下面这样在一个叶子组件中定义一个 Connection 类：

```
class Connectior {
    int d_gateIndex;
```